```
 1:  /*
 2:  * The JVM spec definition of the various Attributes,
 3:  * and their base class.
 4:  */
 5:
 6:  #ifndef ATTRIBUTE_INFO_H[1]
 7:  #define ATTRIBUTE_INFO_H
 8:
 9:  #include <stdlib.h>
10:
11:  #include "defs.h"
12:
13:  class ClassFile[2];
14:
15:  struct attribute_info
16:  {
17:          friend struct field_info[3];
18:          friend struct method_info[4];
19:          friend struct field_method_info[5];
20:  protected:
21:          u2[6] attribute_name_index;
22:          u4[7] attribute_length;
23:
24:  public:
25:          attribute_info()
26:          {
27:          }
28:          attribute_info(u2[6] _attribute_name_index, u4[7] _attribute_length) :
29:            attribute_name_index[8](_attribute_name_index[9]), attribute_length[10](_attribute_length[11])
30:            {
31:            }
32:          void set_class_file(ClassFile[2] * cf)
33:          {
34:                  class_file[12] = cf[13];
35:          }
36:          void set_parent(field_method_info[5] * fmi)
37:          {
38:                  field_method_parent[14] = fmi[15];
39:          }
40:          virtual Result[16] read(u1[17] *& stream_pointer) = 0;
41:          virtual void debug_print(ostream & out = cout) = 0;
42:
43:          ClassFile[2] * class_file;
44:          field_method_info[5] * field_method_parent;
45:  };
46:
47:  struct ConstantValue_attribute : public attribute_info[18]
48:  {
49:          u2[6] constantvalue_index;
50:
51:                  virtual Result[16] read[19](u1[17] *& stream_pointer);
52:                  virtual void debug_print[20](ostream & out = cout);
53:  };
```

Footnotes:

1: *AttributeInfo.h:7*
2: *ClassFile.h:136*
3: *ClassFile.h:68*
4: *ClassFile.h:92*
5: *ClassFile.h:30*
6: *defs.h:172*
7: *defs.h:174*
8: *AttributeInfo.h:21*
9: *AttributeInfo.h:28*
10: *AttributeInfo.h:22*
11: *AttributeInfo.h:28*
12: *AttributeInfo.h:43*
13: *AttributeInfo.h:32*
14: *AttributeInfo.h:44*
15: *AttributeInfo.h:36*
16: *defs.h:29*
17: *defs.h:168*
18: *AttributeInfo.h:15*
19: *AttributeInfo.cpp:11*
20: *AttributeInfo.cpp:24*

```
54:
55:         struct Code_attribute : public attribute_info[1]
56:         {
57:                 struct exception_table_entry
58:                         {
59:                                 u2[2] start_pc;
60:                         u2[2] end_pc;
61:                         u2[2] handler_pc;
62:                         u2[2] catch_type;
63:                         };
64:
65:                 u2[2] max_stack; // measured in words
66:                 u2[2] max_locals; // measured in words
67:                 u4[3] code_length; // in bytes
68:                 u1[4] * code; // array of length code_length
69:                 u2[2] exception_table_length;
70:                         exception_table_entry[5] * exception_table; // array of length exception_table_length
71:                 u2[2] attributes_count;
72:                 attribute_info[1] * attributes; // array of length attributes_count
73:
74:                         virtual Result[6] read[7](u1[4] *& stream_pointer);
75:                         virtual void debug_print[8](ostream & out = cout);
76:         };
77:
78:         struct Exceptions_attribute : public attribute_info[1]
79:         {
80:                 u2[2] number_of_exceptions;
81:                         u2[2] * exception_index_table; // array of length number_of_exceptions
82:
83:                         virtual Result[6] read[9](u1[4] *& stream_pointer);
84:                         virtual void debug_print[10](ostream & out = cout);
85:         };
86:
87:         struct InnerClasses_attribute : public attribute_info[1]
88:         {
89:                 struct classes_entry
90:                 {
91:                 u2[2] inner_class_info_index;
92:                 u2[2] outer_class_info_index;
93:                 u2[2] inner_name_index;
94:                 u2[2] inner_class_access_flags;
95:                 };
96:
97:                 u2[2] number_of_classes;
98:             classes_entry[11] * classes; // array of length number_of_classes
99:
100:                 virtual Result[6] read[12](u1[4] *& stream_pointer);
101:                 virtual void debug_print[13](ostream & out = cout);
102:         };
103:
104:         struct Synthetic_attribute : public attribute_info[1]
105:         {
106:                 virtual Result[6] read[14](u1[4] *& stream_pointer);
```

Footnotes:

1: *AttributeInfo.h:15*
2: *defs.h:172*
3: *defs.h:174*
4: *defs.h:168*
5: *AttributeInfo.h:57*
6: *defs.h:29*
7: *AttributeInfo.cpp:30*
8: *AttributeInfo.cpp:103*
9: *AttributeInfo.cpp:117*
10: *AttributeInfo.cpp:142*
11: *AttributeInfo.h:89*
12: *AttributeInfo.cpp:148*
13: *AttributeInfo.cpp:179*
14: *AttributeInfo.cpp:184*

```
107:                virtual void debug_print[1](ostream & out = cout);
108:        };
109:
110:        struct SourceFile_attribute : public attribute_info[2]
111:        {
112:                u2[3] sourcefile_index;
113:
114:                        virtual Result[4] read[5](u1[6] *& stream_pointer);
115:                        virtual void debug_print[7](ostream & out = cout);
116:        };
117:
118:        struct LineNumberTable_attribute : public attribute_info[2]
119:        {
120:                struct line_number_table_entry
121:                {
122:                        u2[3] start_pc;
123:                u2[3] line_number;
124:                };
125:
126:            u2[3] line_number_table_length;
127:            line_number_table_entry[8] * line_number_table; // array of length line_number_table_length
128:
129:                virtual Result[4] read[9](u1[6] *& stream_pointer);
130:                virtual void debug_print[10](ostream & out = cout);
131:        };
132:
133:        struct LocalVariableTable_attribute : public attribute_info[2]
134:        {
135:                struct local_variable_table_entry
136:                {
137:                        u2[3] start_pc;
138:                u2[3] length;
139:                u2[3] name_index;
140:                u2[3] descriptor_index;
141:                u2[3] index;
142:            };
143:
144:                u2[3] local_variable_table_length;
145:                local_variable_table_entry[11] * local_variable_table; // array of length local_variable_table_lengt
        h
146:
147:                virtual Result[4] read[12](u1[6] *& stream_pointer);
148:                virtual void debug_print[13](ostream & out = cout);
149:        };
150:
151:        struct Deprecated_attribute : public attribute_info[2]
152:        {
153:                virtual Result[4] read[14](u1[6] *& stream_pointer);
154:                virtual void debug_print[15](ostream & out = cout);
155:        };
156:
157:        // The purpose of this structure is to read the attribute_length bytes
158:        // from the input stream (stream_pointer)
```

Footnotes:

1: *AttributeInfo.cpp:194*
2: *AttributeInfo.h:15*
3: *defs.h:172*
4: *defs.h:29*
5: *AttributeInfo.cpp:199*
6: *defs.h:168*
7: *AttributeInfo.cpp:212*
8: *AttributeInfo.h:120*
9: *AttributeInfo.cpp:218*
10: *AttributeInfo.cpp:245*
11: *AttributeInfo.h:135*
12: *AttributeInfo.cpp:251*
13: *AttributeInfo.cpp:284*
14: *AttributeInfo.cpp:290*
15: *AttributeInfo.cpp:300*

```
159:        struct Unrecognized_attribute : public attribute_info[1]
160:        {
161:                virtual Result[2] read[3](u1[4] *& stream_pointer);
162:                virtual void debug_print[5](ostream & out = cout);
163:        };
164:
165:        #endif // ATTRIBUTE_INFO_H
166:
```

Footnotes:

1: *AttributeInfo.h:15*
2: *defs.h:29*
3: *AttributeInfo.cpp:306*
4: *defs.h:168*
5: *AttributeInfo.cpp:320*

```
 1:          /*
 2:          * The JVM spec definition of the Constant Pool entries,
 3:          * Class File Format and supporting structures.
 4:          */
 5:
 6:          #ifndef CLASS_FILE_H[1]
 7:          #define CLASS_FILE_H
 8:
 9:          #include <stdlib.h>
10:
11:          #include "defs.h"
12:          #include "ConstantPool.h"
13:          #include "AttributeInfo.h"
14:
15:          class ClassFile[2];
16:
17:          /*****************************************************************************************
18:                                              Fields and Methods
19:                                                                                  ------------------
20:
21:          Each field is described by a field_info structure.
22:          No two fields in one class file may have the same name and descriptor.
23:
24:          Each method, including each instance initialization method (<init>) and the class or
25:          interface initialization method (<clinit>), is described by a method_info structure.
26:          No two methods in one class file may have the same name and descriptor.
27:
28:          *****************************************************************************************/
29:
30:          struct field_method_info
31:          {
32:                  friend class ClassFile[2];
33:          public:
34:                  // The value of the access_flags item is a mask of flags used to denote access
35:                  // permission to and properties of this field/method
36:                  // (see definitions of access flags in "defs.h")
37:              u2[3] access_flags;
38:                  // The value of the name_index item must be a valid index into the constant_pool table.
39:                  // The constant_pool entry at that index must be a CONSTANT_Utf8_info structure
40:                  // (see "ConstantPool.h" for definition) which must represent a valid field/method name
41:                  // stored as a simple name, that is, as a Java programming language identifier
42:              u2[3] name_index;
43:                  // The value of the descriptor_index item must be a valid index into the constant_pool
44:                  // table. The constant_pool entry at that index must be a CONSTANT_Utf8_info structure
45:                  // (see "ConstantPool.h" for definition) that must represent a valid field/method
46:                  // descriptor (see "Decipher.cpp" for descriptor explanations)
47:              u2[3] descriptor_index;
48:                  // The value of the attributes_count item indicates the number of additional attributes
49:                  // of this field/method (see "AttributesInfo.h" for various attribute definitions)
50:              u2[3] attributes_count;
51:                  // Each value of the attributes table must be an attribute structure (see "AttributeInfo.h").
52:                  // A field/method can have any number of attributes associated with it.
53:              attribute_info[4] ** attributes; // array of length attributes_count
```

Footnotes:
1: *ClassFile.h:7*
2: *ClassFile.h:136*
3: *defs.h:172*
4: *AttributeInfo.h:15*

```
54:
55:                 Result[1] read[2](u1[3] *& stream_pointer);
56:                 ClassFile[4] * class_file;
57:         public:
58:                 virtual void debug_print[5](ostream & out = cout) = 0;
59:                 field_method_info(ClassFile[4] * cf) : class_file[6](cf)
60:                 {
61:                 }
62:         };
63:
64:         /*
65:         * Fields
66:         */
67:
68:         struct field_info : public field_method_info[7]
69:         {
70:         public:
71:                 // type of this field (any of primitive type or reference)
72:                 BasicType[8] type;
73:
74:                 // offset of this field in the memory chunk allocated by this field;
75:                 // note that the memory itself is not stored here - the value of the field
76:                 // is calculated given the memory chunk start pointer plus this offset;
77:                 // note also, that regardless of the memory allocated for the object
78:                 // this offset is always an invariant
79:                 unsigned int offset;
80:
81:                 virtual void debug_print[9](ostream & out = cout);
82:
83:                 field_info(ClassFile[4] * cf) : field_method_info[10](cf)
84:                 {
85:                 }
86:         };
87:
88:         /*
89:         * Methods
90:         */
91:
92:         struct method_info : public field_method_info[7]
93:         {
94:         public:
95:
96:                 virtual void debug_print[11](ostream & out = cout);
97:                 Code_attribute[12] * get_code_attribute[13]();
98:
99:                 method_info(ClassFile[4] * cf) : field_method_info[10](cf)
100:                {
101:                }
102:                // Returns method's name as a wide-character string
103:                void get_method_name[14](wchar_t *& method_name, u2[15] & method_name_length);
104:                // Returns method's descriptor as a wide-character string encoded as
105:                // specified by the JVM spec
106:                void get_method_descriptor[16](wchar_t *& descriptor, u2[15] & descriptor_length);
```

Footnotes:

1: *defs.h:29*
2: *ClassFile.cpp:42*
3: *defs.h:168*
4: *ClassFile.h:136*
5: *ClassFile.cpp:100*
6: *ClassFile.h:56*
7: *ClassFile.h:30*
8: *defs.h:208*
9: *ClassFile.cpp:132*
10: *ClassFile.h:59*
11: *ClassFile.cpp:139*
12: *AttributeInfo.h:55*
13: *ClassFile.cpp:230*
14: *ClassFile.cpp:168*
15: *defs.h:172*
16: *ClassFile.cpp:177*

```
107:
108:                 // Returns true if the names and the descriptors of both methods are equal
109:                 int is_equal_signature[1](method_info[2] * mi);
110:                 // Returns true if the method in the <init> one
111:                 int is_initializer[3]();
112:         };
113:
114:
115:         class ClassLoader[4];
116:         class ClassData[5];
117:         class HeapManager[6];
118:         class InstanceData[7];
119:
120:         /*****************************************************************************************
121:                                         The Class File Format
122:                                                                    ---------------------
123:
124:         Compiled code to be executed by the Java virtual machine is represented using a hardware-
125:         and operating system-independent binary format, typically (but not necessarily) stored
126:         in a file, known as the class file format. The class file format precisely defines
127:         the representation of a class or interface, including details such as byte ordering
128:         that might be taken for granted in a platform-specific object file format.
129:
130:         A class file consists of a stream of 8-bit bytes. All 16-bit, 32-bit, and 64-bit quantities
131:         are constructed by reading in two, four, and eight consecutive 8-bit bytes, respectively.
132:         Multibyte data items are always stored in big-endian order, where the high bytes come first.
133:         (see memcpy_bigendian.cpp for the conversion functions)
134:         *****************************************************************************************/
135:
136:         class ClassFile
137:         {
138:         public:
139:                 // The following fields are in the order as required by JVM spec
140:
141:                 // The magic item supplies the magic number identifying the class file format;
142:                 // it has the value 0xCAFEBABE
143:                 u4[8] magic;
144:                 // The values of the minor_version and major_version items are the minor and major
145:                 // version numbers of this class file. Together, a major and a minor version number
146:                 // determine the version of the class file format
147:             u2[9] minor_version;
148:             u2[9] major_version;
149:             // The value of the constant_pool_count item is equal to the number of entries in the
150:                 // constant_pool table plus one. A constant_pool index is considered valid if it is
151:                 // greater than zero and less than constant_pool_count, with the exception for constants
152:                 // of type long and double
153:             u2[9] constant_pool_count;
154:             // The constant_pool is a table of structures representing various string constants,
155:                 // class and interface names, field names, and other constants that are referred to
156:                 // within the ClassFile structure and its substructures. The format of each constant_pool
157:                 // table entry is indicated by its first "tag" byte.
158:                 // (see ConstantPool.h for the definition of all constant pool structures)
159:             ConstantPool[10] * constant_pool; // number of items in CP = <constant_pool_count-1>
```

Footnotes:

1: *ClassFile.cpp:185*
2: *ClassFile.h:92*
3: *ClassFile.cpp:209*
4: *ClassLoader.h:14*
5: *ObjectData.h:92*
6: *HeapManager.h:81*
7: *ObjectData.h:113*
8: *defs.h:174*
9: *defs.h:172*
10: *ConstantPool.h:274*

```
160:            // The value of the access_flags item is a mask of flags used to denote access
161:                // permissions to and properties of this class or interface.
162:                // (See defs.h for the definitions of all the access flags)
163:            u2[1] access_flags;
164:                // The value of the this_class item must be a valid index into the constant_pool table.
165:                // The constant_pool entry at that index must be a CONSTANT_Class_info structure
166:                // (see "ConstantPool.h") representing the class or interface defined by this class file.
167:            u2[1] this_class;
168:                // For a class, the value of the super_class item either must be zero or must be a valid
169:                // index into the constant_pool table. If the value of the super_class item is nonzero,
170:                // the constant_pool entry at that index must be a CONSTANT_Class_info structure
171:                // (see "ConstantPool.h") representing the direct superclass of the class defined by this
172:                // class file. Neither the direct superclass nor any of its superclasses may be a final class.
173:            // If the value of the super_class item is zero, then this class file must represent
174:                // the class java.lang.Object, the only class or interface without a direct superclass.
175:            // For an interface, the value of the super_class item must always be a valid index
176:                // into the constant_pool table. The constant_pool entry at that index must be a
177:                // CONSTANT_Class_info structure representing the class Object.
178:            u2[1] super_class;
179:            // The value of the interfaces_count item gives the number of direct superinterfaces
180:                // of this class or interface type.
181:            u2[1] interfaces_count;
182:                // Each value in the interfaces array must be a valid index into the constant_pool table.
183:                // The constant_pool entry at each value of interfaces[i], where 0  i < interfaces_count,
184:                // must be a CONSTANT_Class_info structure (see "ConstantPool.h") representing an interface
185:                // that is a direct superinterface of this class or interface type, in the left-to-right
186:                // order given in the source for the type.
187:            u2[1] * interfaces; // array of length interfaces_count
188:                // The value of the fields_count item gives the number of field_info structures
189:                // in the fields table. The field_info structures (see definition above) represent all fields,
190:                // both class variables and instance variables, declared by this class or interface type.
191:            u2[1] fields_count;
192:                // Each value in the fields table must be a field_info structure (see definition above)
193:                // giving a complete description of a field in this class or interface.
194:                // The fields table includes only those fields that are declared by this class or
195:                // interface. It does not include items representing fields that are inherited from
196:                // superclasses or superinterfaces.
197:            field_info[2] ** fields; // array of length fields_count
198:                // The value of the methods_count item gives the number of method_info structures
199:                // in the methods table.
200:            u2[1] methods_count;
201:                // Each value in the methods table must be a method_info structure (see definition above)
202:                // giving a complete description of a method in this class or interface. If the method
203:                // is not native or abstract, the Java virtual machine instructions implementing
204:                // the method are also supplied.
205:            // The method_info structures represent all methods declared by this class or interface
206:                // type, including instance methods, class (static) methods, instance initialization
207:                // methods (<init>), and any class or interface initialization method (<clinit>).
208:                // The methods table does not include items representing methods that are inherited
209:                // from superclasses or superinterfaces.
210:            method_info[3] ** methods; // array of length methods_count
211:                // The value of the attributes_count item gives the number of attributes
212:                // in the attributes table of this class.
```

Footnotes:

1: *defs.h:172*
2: *ClassFile.h:68*
3: *ClassFile.h:92*

```
213:             u2[1] attributes_count;
214:                 // Each value of the attributes table must be an attribute structure (see "AttributeInfo.h").
215:             // A Java virtual machine implementation is required to silently ignore any or all
216:                 // attributes in the attributes table of a ClassFile structure that it does not recognize.
217:             attribute_info[2] ** attributes; // array of length attributes_count
218:
219:                 // Class file implementation-dependant fields
220:
221:                 // remember which class loader loaded myself
222:                 ClassLoader[3] * class_loader;
223:
224:                 // contains all static variables of this type;
225:                 // this field is allocated on the Heap during the preparation step of the class
226:                 // resolution process
227:                 ClassData[4] * class_data;
228:
229:                 // This table stores the methods of this class and its superclasses.
230:                 // The index of this array is an offset of the corresponding method in the method table
231:                 method_info[5] ** method_table;
232:                 u4[6] method_table_size;
233:
234:                 // Stores all the type's related data
235:                 HeapManager[7] * class_heap_manager;
236:                 // Stores all the instance's related data
237:                 HeapManager[7] * instance_heap_manager;
238:
239:                 // Points to the instance of the java.lang.Class class
240:                 // associated with this class;
241:                 // The instance will serve as an interface between the Java program and
242:                 // the internal representation of the class
243:                 InstanceData[8] * class_instance;
244:
245:                 // Helper function to build the method table of this class;
246:                 // called recursively on the hierarchy tree accumulating the virtual methods
247:                 // in the vector
248:                 Result[9] get_class_methods[10](ClassFile[11] * cf, Vector[12]<method_info*> & methods);
249:
250:         public:
251:                 ClassFile[13](ClassLoader[3] * _class_loader);
252:                 ~ClassFile[14]();
253:                 Result[9] read[15](u1[16] *& stream_pointer);
254:                 void debug_print[17](ostream & out = cout);
255:                 virtual void get_full_class_name[18](wchar_t *& class_name, u2[1] & class_name_length);
256:
257:                 // Fills in the method table array with the pointers to method's code.
258:                 // Note that only virtual methods will reside in this table
259:                 Result[9] build_method_table[19]();
260:
261:                 // Returns the private, final, static, or initializer (<init>) method
262:                 // having the specified name and descriptor;
263:                 // The method will be represented by a direct link to the method_info
264:                 // of this method.
265:                 // This function will be used by the invokespecial, invokestatic
```

Footnotes:

1: *defs.h:172*
2: *AttributeInfo.h:15*
3: *ClassLoader.h:14*
4: *ObjectData.h:92*
5: *ClassFile.h:92*
6: *defs.h:174*
7: *HeapManager.h:81*
8: *ObjectData.h:113*
9: *defs.h:29*
10: *ClassFile.cpp:899*
11: *ClassFile.h:136*
12: *vector.h:7*
13: *ClassFile.cpp:784*
14: *ClassFile.cpp:791*
15: *ClassFile.cpp:296*
16: *defs.h:168*
17: *ClassFile.cpp:256*
18: *ClassFile.cpp:442*
19: *ClassFile.cpp:965*

```
266:                // and invokeinterface instructions
267:                Result[1] get_method[2](CONSTANT_Utf8_info[3] * method_name,
268:                                        CONSTANT_Utf8_info[3] * descriptor,
269:                                        method_info[4] *& minfo);
270:
271:                // Returns the non-private, non-final, non-static, non-initializer method
272:                // having the specified name and descriptor;
273:                // The method will be represented by an offset in the method table of this class.
274:                // This function will be used by the invokevirtual instruction
275:                Result[1] get_virtual_method[5](CONSTANT_Utf8_info[3] * method_name,
276:                                                CONSTANT_Utf8_info[3] * descriptor,
277:                                                int & moffset);
278:
279:                // Oppositely to the get_method function this function looks for the
280:                // specified method only among this class' own methods.
281:                // It is used to get the "main" and "<clinit>" methods
282:                Result[1] get_method_by_name[6](wchar_t * method_name, wchar_t * descriptor,
283:                                                                        Code_attribute[7] *& code_at
        tribute);
284:
285:                // Returns the field having the specified name and descriptor as UTF8 strings;
286:                // If the field is not found in this class the function will try to find it
287:                // recursively in all the superclasses of this class
288:                Result[1] get_field[8](CONSTANT_Utf8_info[3] * utf8_name,
289:                                       CONSTANT_Utf8_info[3] * utf8_descriptor,
290:                                       field_info[9] *& finfo);
291:
292:                // Returns the field having the specified name and descriptor as wide-character strings;
293:                // If the field is not found in this class the function will try to find it
294:                // recursively in all the superclasses of this class
295:                Result[1] get_field_by_name[10](wchar_t * field_name, wchar_t * descriptor,
296:                                                field_info[9] *& finfo);
297:
298:                // This function initializes the class, i.e. calls the class' <clinit> method;
299:                // Class initialization methods <clinit> are always invoked directly by JVM itself -
300:                // never by any bytecodes
301:                Result[1] initialize[11]();
302:
303:                // The entry point of the application execution. This call will cause the main class
304:                // (and all its supertypes) to be loaded, linked and initialized prior to the execution
305:                // of the "main" method's code
306:                Result[1] run_main_method();
307:
308:                // This function is responsible for creating new instances of the class referred to
309:                // by this ClassFile;
310:                // The function returns the new object's reference or null (0) in case of failure
311:                word[12] create_new_instance[13]();
312:
313:                // This function determines whether an instance of "that_class" is an instance
314:                // of "this" class according to the JVM rules.
315:                // It will return 1 in case of true and 0 otherwise
316:                int instance_of[14](ClassFile[15] * that_class);
317:                // Returns true if this class is of type java.lang.Object
```

Footnotes:

1: *defs.h:29*
2: *ClassFile.cpp:457*
3: *ConstantPool.h:186*
4: *ClassFile.h:92*
5: *ClassFile.cpp:523*
6: *ClassFile.cpp:572*
7: *AttributeInfo.h:55*
8: *ClassFile.cpp:627*
9: *ClassFile.h:68*
10: *ClassFile.cpp:698*
11: *ClassFile.cpp:805*
12: *defs.h:195*
13: *ClassFile.cpp:870*
14: *ClassFile.cpp:1081*
15: *ClassFile.h:136*

```
318:                int is_Object[1]();
319:                // Returns true if "this" class implements "that" class
320:                int implements[2](ClassFile[3] * that);
321:                // Returns true if "this" class inherits from "that" class
322:                int inherits[4](ClassFile[3] * that);
323:
324:
325:                // This is the special case of a ClassFile;
326:                // The class file will make itself "bootable", i.e. will make a "fake" class file
327:                // consisting of an artificially constructed Constant Pool and a single method
328:                // which will be called by the primary thread of the JVM at the very beginning of its run.
329:                // The method will first call the "initializeSystemClass" static method to initialize the
330:                // java.lang.System class (basic initializations such as I/O streams, system properties etc.)
331:                // The method then will push the reference to the "String[] args" object onto its stack frame and
332:                // call the "main" method of the main class;
333:                // The function will return the code attribute of the constructed "boot" method
334:                Result[5] make_bootable[6](char * main_class_ascii_name, Code_attribute[7] *& code_attribute, InstanceDat
        a[8] *& boot_instance);
335:
336:                virtual int is_array() { return 0; }
337:                virtual int is_primitive_array() { return 0; }
338:        };
339:
340:        // This class represents class file for arrays of references
341:        class ArrayClassFile : public ClassFile[3]
342:        {
343:        private:
344:                ClassFile[3] * array_type_class_file;
345:        protected:
346:                wchar_t * type_name;
347:                u2[9] type_name_length;
348:        public:
349:                ArrayClassFile(ClassLoader[10] * _class_loader) : ClassFile[11](_class_loader)
350:                {
351:                }
352:                ArrayClassFile(ClassLoader[10] * _class_loader, ClassFile[3] * _array_type_class_file,
353:                        wchar_t * _type_name, u2[9] _type_name_length) : ClassFile[11](_class_loader[12]),
354:                        array_type_class_file[13](_array_type_class_file[14])
355:                {
356:                        type_name[15] = new wchar_t[_type_name_length[16]];
357:                        wcscpy(type_name[15], _type_name[17]);
358:                }
359:                ~ArrayClassFile()
360:                {
361:                        delete [] type_name[15];
362:                }
363:                // Allocates an array of reference values on the instance heap of the given length
364:                virtual word[18] create_new_instance[19](u4[20] count);
365:                virtual int is_array() { return 1; }
366:                virtual int is_primitive_array() { return 0; }
367:        };
368:
369:        // This class represents arrays of primitive values
```

Footnotes:

1: *ClassFile.cpp:1030*
2: *ClassFile.cpp:1041*
3: *ClassFile.h:136*
4: *ClassFile.cpp:1061*
5: *defs.h:29*
6: *ClassFile.cpp:1124*
7: *AttributeInfo.h:55*
8: *ObjectData.h:113*
9: *defs.h:172*
10: *ClassLoader.h:14*
11: *ClassFile.cpp:784*
12: *ClassFile.h:352*
13: *ClassFile.h:344*
14: *ClassFile.h:352*
15: *ClassFile.h:346*
16: *ClassFile.h:353*
17: *ClassFile.h:353*
18: *defs.h:195*
19: *ClassFile.cpp:992*
20: *defs.h:174*

```
370:        class PrimitiveArrayClassFile : public ArrayClassFile[1]
371:        {
372:        private:
373:                u1[2] array_type;
374:        public:
375:                PrimitiveArrayClassFile(ClassLoader[3] * _class_loader, u1[2] _array_type,
376:                        wchar_t * _type_name, u2[4] _type_name_length) :
377:                ArrayClassFile[5](_class_loader[6]), array_type[7](_array_type[8])
378:                {
379:                        type_name_length[9] = _type_name_length[10];
380:                        type_name[11] = new wchar_t[_type_name_length[10]];
381:                        wcscpy(type_name[11], _type_name[12]);
382:                }
383:                ~PrimitiveArrayClassFile()
384:                {
385:                        delete [] type_name[11];
386:                }
387:                virtual void get_full_class_name(wchar_t *& _type_name, u2[4] & _type_name_length)
388:                {
389:                        wcscpy(_type_name[13], type_name[11]);
390:                        _type_name_length[14] = type_name_length[9];
391:                }
392:                // Allocates an array of primitive values on the instance heap of the given length
393:                virtual word[15] create_new_instance[16](u4[17] count);
394:
395:                u1[2] get_array_type() const { return array_type[7]; }
396:                virtual int is_array() { return 1; }
397:                virtual int is_primitive_array() { return 1; }
398:        };
399:
400:        #endif // CLASS_FILE_H
401:
```

Footnotes:

1: *ClassFile.h:341*
2: *defs.h:168*
3: *ClassLoader.h:14*
4: *defs.h:172*
5: *ClassFile.h:349*
6: *ClassFile.h:375*
7: *ClassFile.h:373*
8: *ClassFile.h:375*
9: *ClassFile.h:347*
10: *ClassFile.h:376*
11: *ClassFile.h:346*
12: *ClassFile.h:376*
13: *ClassFile.h:387*
14: *ClassFile.h:387*
15: *defs.h:195*
16: *ClassFile.cpp:1012*
17: *defs.h:174*

```
1:          /*
2:           * This is a native implementation of a general class loader.
3:           * Serves as a repository of all the class files that were loaded
4:           * by this class loader.
5:           */
6:
7:          #ifndef CLASS_LOADER_H[1]
8:          #define CLASS_LOADER_H
9:
10:         #include "ClassFile.h"
11:         #include "VirtualMachine.h"
12:         #include "hashtable.h"
13:
14:         class ClassLoader
15:         {
16:         private:
17:                 // Reads the binary file into a buffer (buffer is created and must be destroyed later)
18:                 Result[2] open_file[3](char * filepath, u1[4] *& buffer, u2[5] & length);
19:
20:                 // Recursively resolve the supreclass entry of the given class
21:                 Result[2] load_superclass[6](ClassFile[7] * cf);
22:
23:                 // Recuresively resolve all the superinterfaces of the given class
24:                 Result[2] load_superinterfaces[8](ClassFile[7] * cf);
25:
26:         private:
27:                 VirtualMachine[9] * vm;
28:
29:                 // This table contains the list of the names of all classes that were
30:                 // loaded by this class loader, forming the namespace of this class loader.
31:                 // This table is in fact the main repository of all the class files loaded
32:                 // by this class loader, since the values of the hashtable are the pointers
33:                 // to the class files themselves
34:                 HashTable[10] this_namespace;
35:
36:                 // As specified by the parent-delegation model in Java1.2 each class loader
37:                 // has an associated parent class loader.
38:                 // For the bootstrap class loader this pointer will be equal to NULL
39:                 ClassLoader[11] * parent_class_loader;
40:
41:         public:
42:                 ClassLoader(ClassLoader[11] * _parent_class_loader, VirtualMachine[9] * _vm) :
43:                   parent_class_loader[12](_parent_class_loader[13]), vm[14](_vm[15]) {}
44:                 ~ClassLoader[16]();
45:
46:                 // This is the initial request to the class loader to load the class
47:                 // specified by its fully qualified class name;
48:                 // The "length" parameter specifies the length of the wide-character class_name string
49:                 ClassFile[7] * load_class[17](wchar_t * class_name, unsigned int class_name_length);
50:
51:                 // Loads the class from the binary stream and adds the loaded class name
52:                 // to the namespace of this class loader.
53:                 // The function will parse the binary data into internal data structures
```

Footnotes:

1: *ClassLoader.h:8*
2: *defs.h:29*
3: *ClassLoader.cpp:247*
4: *defs.h:168*
5: *defs.h:172*
6: *ClassLoader.cpp:294*
7: *ClassFile.h:136*
8: *ClassLoader.cpp:316*
9: *VirtualMachine.h:255*
10: *hashtable.h:28*
11: *ClassLoader.h:14*
12: *ClassLoader.h:39*
13: *ClassLoader.h:42*
14: *ClassLoader.h:27*
15: *ClassLoader.h:42*
16: *ClassLoader.cpp:12*
17: *ClassLoader.cpp:18*

```
54:                 // in the method area
55:                 Result[1] define_class[2](u1[3] * stream, u2[4] size, ClassFile[5] *& cf);
56:
57:                 // Will create a class for an array of primitive values;
58:                 // The name of the class will be the descriptor of the array;
59:                 // For example, for int[] the name will be "[I".
60:                 // This function returns the existing class file for arrays whose class is
61:                 // already created;
62:                 // Note, that the primitive array types will always be created by the bootstrap
63:                 // class loader
64:                 Result[1] define_primitive_array_class[6](u1[3] array_type, ClassFile[5] *& cf);
65:
66:                 // Will create a class for an array of reference values;
67:                 // The name of the class will be the descriptor of the array;
68:                 // For example, for String[] the name will be "[java/lang/String;".
69:                 // This function returns the existing class file for arrays whose class is
70:                 // already created;
71:                 // Note, that the reference array types will always be created by the
72:                 // class loader that has loaded the array type class
73:                 Result[1] define_array_class[7](ClassFile[5] * array_type, ClassFile[5] *& cf);
74:
75:                 // Returns NULL if this class loader has not yet loaded the given class
76:                 ClassFile[5] * get_class[8](wchar_t * class_name, u2[4] class_name_length);
77:
78:                 // If the class is not loaded yet the function will load it and return
79:                 // the instance data of the java.lang.Class instance corresponding to
80:                 // this class.
81:                 // If class name is the name of the primitive type (like "int") the special
82:                 // class for this type will be created
83:                 InstanceData[9] * find_class[10](char * class_name);
84:
85:                 // Creates class files for primitive types such as "int"
86:                 ClassFile[5] * create_primitive_type[11](char * class_name);
87:
88:                 // Prints all the classes defined in this class loader's namespace
89:                 void dump_namespace[12]();
90:
91:                 Result[1] define_boot_class[13](ClassFile[5] * cf);
92:
93:                 HashTable[14] * get_namespace()
94:                 {
95:                         return &this_namespace[15];
96:                 }
97:
98:                 inline VirtualMachine[16] * get_jvm() const
99:                 {
100:                        return vm[17];
101:                }
102:        };
103:
104:        #endif // CLASS_LOADER_H
```

Footnotes:

1: *defs.h:29*
2: *ClassLoader.cpp:70*
3: *defs.h:168*
4: *defs.h:172*
5: *ClassFile.h:136*
6: *ClassLoader.cpp:341*
7: *ClassLoader.cpp:387*
8: *ClassLoader.cpp:187*
9: *ObjectData.h:113*
10: *ClassLoader.cpp:200*
11: *ClassLoader.cpp:434*
12: *ClassLoader.cpp:454*
13: *ClassLoader.cpp:468*
14: *hashtable.h:28*
15: *ClassLoader.h:34*
16: *VirtualMachine.h:255*
17: *ClassLoader.h:27*

```
 1:          /*
 2:           * Class to deal with method code.
 3:           *
 4:           */
 5:
 6:          #ifndef CODE_MANAGER_H[1]
 7:          #define CODE_MANAGER_H
 8:
 9:          #include "defs.h"
10:
11:          class VirtualMachine[2];
12:
13:          class CodeManager
14:          {
15:          private:
16:                  VirtualMachine[2] * p_VirtualMachine;
17:          public:
18:                  CodeManager(VirtualMachine[2] * vm) : p_VirtualMachine[3](vm) {}
19:                  ~CodeManager() {}
20:
21:                  static void debug_print[4](u1[5] * code, unsigned long code_length);
22:          };
23:
24:
25:
26:
27:          #endif // HEAP_MANAGER_H
28:
```

Footnotes:

1: *CodeManager.h:7*
2: *VirtualMachine.h:255*
3: *CodeManager.h:16*
4: *CodeManager.cpp:7*
5: *defs.h:168*

```
 1:          /*
 2:           * Classes to set and retrieve information from the constant pool.
 3:           *
 4:           */
 5:
 6:          #ifndef CONSTANT_POOL_H[1]
 7:          #define CONSTANT_POOL_H
 8:
 9:          #include <stdio.h>
10:          #include <malloc.h>
11:          #include <assert.h>
12:          #include <wchar.h>
13:
14:          #include "defs.h"
15:          #include "AttributeInfo.h"
16:          #include "vector.h"
17:
18:          class ClassFile[2];
19:
20:          class ConstantPoolEntry
21:          {
22:                  friend class ConstantPool[3];
23:          protected:
24:                  int resolved;
25:
26:          public:
27:                  // Specifies the type of the CP entry
28:                  u1[4] tag;
29:
30:                  // What class file this constant pool belongs to
31:                  ClassFile[2] * class_file;
32:
33:                  virtual Result[5] read(u1[4] *& stream_pointer) = 0;
34:                  virtual void debug_print(ostream & out = cout) = 0;
35:                  ConstantPoolEntry(ClassFile[2] * _class_file, u1[4] _tag) : class_file[6](_class_file), tag[7](_tag),
36:                          resolved[8](0)
37:                  {
38:                  }
39:
40:                  u1[4] getType()
41:                  {
42:                          return tag[7];
43:                  }
44:
45:                  virtual Result[5] resolve() = 0;
46:
47:                  // Returns true if this CP entry was already resolved, false otherwise
48:                  inline int is_resolved()
49:                  {
50:                          return resolved[8];
51:                  }
52:          };
53:
```

Footnotes:

1: *ConstantPool.h:7*
2: *ClassFile.h:136*
3: *ConstantPool.h:274*
4: *defs.h:168*
5: *defs.h:29*
6: *ConstantPool.h:31*
7: *ConstantPool.h:28*
8: *ConstantPool.h:24*

```
54:         /*
55:         * All the Constant Pool entry types as in JVM spec
56:         */
57:
58:         struct CONSTANT_Class_info : public ConstantPoolEntry[1]
59:         {
60:             u2[2] name_index;
61:                 virtual Result[3] read[4](u1[5] *& stream_pointer);
62:                 void debug_print[6](ostream & out = cout);
63:                 CONSTANT_Class_info(ClassFile[7] * _class_file) : ConstantPoolEntry[8](_class_file, CONSTANT_Class[9]),
64:                         resolved_class_file[10](NULL)
65:                 {
66:                 }
67:                 virtual Result[3] resolve[11]();
68:
69:                 // Direct link to the resolved class file;
70:                 // This pointer will be assigned only after the resolution of the class
71:                 ClassFile[7] * resolved_class_file;
72:         };
73:
74:         struct CONSTANT_Fieldref_info : public ConstantPoolEntry[1]
75:         {
76:                 // index of a CONSTANT_Class_info structure representing the class or interface type
77:                 // that contains the declaration of the field
78:             u2[2] class_index;
79:                 // descriptor of the field
80:             u2[2] name_and_type_index;
81:                 virtual Result[3] read[12](u1[5] *& stream_pointer);
82:                 void debug_print[13](ostream & out = cout);
83:                 CONSTANT_Fieldref_info(ClassFile[7] * _class_file) : ConstantPoolEntry[8](_class_file, CONSTANT_Fieldr
        ef[14]),
84:                         finfo[15](NULL)
85:                 {
86:                 }
87:
88:                 // Points to the field_info of the resolved class;
89:                 // gets its value after the resolution
90:                 field_info[16] * finfo;
91:
92:                 virtual Result[3] resolve[17]();
93:         };
94:
95:
96:         struct CONSTANT_Utf8_info[18];
97:
98:         struct CONSTANT_Methodref_info : public ConstantPoolEntry[1]
99:         {
100:                // index of a CONSTANT_Class_info structure representing the class or interface type
101:                // that contains the declaration of the method
102:            u2[2] class_index;
103:                // descriptor of the method
104:            u2[2] name_and_type_index;
105:                virtual Result[3] read[19](u1[5] *& stream_pointer);
```

Footnotes:

1: *ConstantPool.h:20*
2: *defs.h:172*
3: *defs.h:29*
4: *ConstantPool.cpp:13*
5: *defs.h:168*
6: *ConstantPool.cpp:86*
7: *ClassFile.h:136*
8: *ConstantPool.h:35*
9: *defs.h:326*
10: *ConstantPool.h:71*
11: *ConstantPool.cpp:22*
12: *ConstantPool.cpp:228*
13: *ConstantPool.cpp:291*
14: *defs.h:327*
15: *ConstantPool.h:90*
16: *ClassFile.h:68*
17: *ConstantPool.cpp:238*
18: *ConstantPool.h:186*
19: *ConstantPool.cpp:321*

```
106:                 void debug_print[1](ostream & out = cout);
107:                 CONSTANT_Methodref_info(ClassFile[2] * _class_file) : ConstantPoolEntry[3](_class_file, CONSTANT_Metho
        dref[4]),
108:                         minfo[5](NULL), moffset[6](-1), total_arguments_size[7](-1)
109:                 {
110:                 }
111:
112:                 // is not in use
113:                 virtual Result[8] resolve() { return Failure[9]; } // should not be called
114:
115:                 // Will be used by invokevirtual instruction
116:                 Result[8] resolve_virtual[10]();
117:                 // Will be used by invokestatic and invokespecial instructions
118:                 Result[8] resolve_nonvirtual[11]();
119:
120:                 // The following two fields represent two different models of the resolved method
121:
122:                 // 1) For static, <init>, <clinit>, private, and final methods
123:                 //    there is the direct link to the method_info of this method;
124:                 //    This field will be used by invokespecial and invokestatic instructions
125:                 method_info[12] * minfo;
126:                 // 2) For virtual (dynamically bound methods) this offset
127:                 //    represent the index of the method_info in the class' method table;
128:                 //    This field will be used by invokevirtual instructions
129:                 //    (moffset is signed int to represent the absence of an offset by -1)
130:                 int moffset;
131:
132:                 // This variable holds the total size of all the method's arguments (in words)
133:                 // to be used in the invokation methods. It will define the depth of the
134:                 // overlapped block area between two stack frames.
135:                 // Once initialized (upon first usage) it will indicate that no more calculations
136:                 // are needed
137:                 int total_arguments_size;
138:                 // This function will return the value of the total_arguments_size (if initialized)
139:                 // or will calculate the value upon first usage.
140:                 // Note, that the returning value is in words (bytes, chars, booleans etc. occupy one word,
141:                 // longs and doubles occupy two words)
142:                 unsigned int get_arguments_size[13]();
143:
144:                 Result[8] get_arguments[14](Vector[15]<general_type*>& args, general_type[16] * return_type);
145:
146:         private:
147:                 // Helper function for the resolution methods
148:                 Result[8] start_resolution[17](ClassFile[2] *& referenced_class,
149:                                                CONSTANT_Utf8_info[18] *& utf8_name,
150:                                                CONSTANT_Utf8_info[18] *& utf8_descriptor);
151:         };
152:
153:
154:         class InstanceData[19];
155:
156:         struct CONSTANT_InterfaceMethodref_info : public ConstantPoolEntry[20]
157:         {
```

Footnotes:

1: *ConstantPool.cpp:491*
2: *ClassFile.h:136*
3: *ConstantPool.h:35*
4: *defs.h:328*
5: *ConstantPool.h:125*
6: *ConstantPool.h:130*
7: *ConstantPool.h:137*
8: *defs.h:29*
9: *defs.h:34*
10: *ConstantPool.cpp:334*
11: *ConstantPool.cpp:361*
12: *ClassFile.h:92*
13: *ConstantPool.cpp:437*
14: *ConstantPool.cpp:459*
15: *vector.h:7*
16: *defs.h:231*
17: *ConstantPool.cpp:386*
18: *ConstantPool.h:186*
19: *ObjectData.h:113*
20: *ConstantPool.h:20*

```
158:             u2[1] class_index;
159:             u2[1] name_and_type_index;
160:                 virtual Result[2] read[3](u1[4] *& stream_pointer);
161:                 void debug_print[5](ostream & out = cout);
162:                 CONSTANT_InterfaceMethodref_info(ClassFile[6] * _class_file) : ConstantPoolEntry[7](_class_file, CONST
        ANT_InterfaceMethodref[8])
163:                 {
164:                 }
165:                 virtual Result[2] resolve() { return Failure[9]; } // should not be called
166:
167:                 Result[2] resolve_on[10](InstanceData[11] * id, method_info[12] *& minfo);
168:         };
169:
170:         struct CONSTANT_String_info : public ConstantPoolEntry[13]
171:         {
172:             u2[1] string_index;
173:                 virtual Result[2] read[14](u1[4] *& stream_pointer);
174:                 void debug_print[15](ostream & out = cout);
175:                 CONSTANT_String_info(ClassFile[6] * _class_file) : ConstantPoolEntry[7](_class_file, CONSTANT_String[16])
        ,
176:                         interned_string_reference[17](null[18])
177:                 {
178:                 }
179:                 virtual Result[2] resolve[19]();
180:
181:                 // After this entry is resolved contains reference to the one
182:                 // and the only interned string representation
183:                 word[20] interned_string_reference;
184:         };
185:
186:         struct CONSTANT_Utf8_info : public ConstantPoolEntry[13]
187:         {
188:             u2[1] length;
189:             u1[4] * bytes; // array of length 'length'
190:                 virtual Result[2] read[21](u1[4] *& stream_pointer);
191:                 void debug_print[22](ostream & out = cout);
192:                 // This function will create wchar string, so it must be deleted after usage
193:                 Result[2] get_string[23](wchar_t *&); // decodes UTF8 into a wchar string
194:                 CONSTANT_Utf8_info(ClassFile[6] * _class_file) : ConstantPoolEntry[7](_class_file, CONSTANT_Utf8[24])
195:                 {
196:                 }
197:                 CONSTANT_Utf8_info[25](ClassFile[6] * _class_file, wchar_t * wstring);
198:                 virtual Result[2] resolve[26]();
199:
200:                 int operator==(const CONSTANT_Utf8_info[27] & utf8_info);
201:                 int operator!=(const CONSTANT_Utf8_info[27] & utf8_info);
202:         };
203:
204:         struct CONSTANT_Integer_info : public ConstantPoolEntry[13]
205:         {
206:             u4[28] bytes;
207:                 virtual Result[2] read[29](u1[4] *& stream_pointer);
208:                 void debug_print[30](ostream & out = cout);
```

Footnotes:

1: *defs.h:172*
2: *defs.h:29*
3: *ConstantPool.cpp:521*
4: *defs.h:168*
5: *ConstantPool.cpp:584*
6: *ClassFile.h:136*
7: *ConstantPool.h:35*
8: *defs.h:329*
9: *defs.h:34*
10: *ConstantPool.cpp:531*
11: *ObjectData.h:113*
12: *ClassFile.h:92*
13: *ConstantPool.h:20*
14: *ConstantPool.cpp:592*
15: *ConstantPool.cpp:621*
16: *defs.h:330*
17: *ConstantPool.h:183*
18: *defs.h:201*
19: *ConstantPool.cpp:600*
20: *defs.h:195*
21: *ConstantPool.cpp:628*
22: *ConstantPool.cpp:737*
23: *ConstantPool.cpp:639*
24: *defs.h:336*
25: *ConstantPool.cpp:708*
26: *ConstantPool.cpp:719*
27: *ConstantPool.h:186*
28: *defs.h:174*
29: *ConstantPool.cpp:756*
30: *ConstantPool.cpp:769*

```
209:                int get_int_value()
210:                {
211:                        return int( bytes[1] );
212:                }
213:                CONSTANT_Integer_info(ClassFile[2] * _class_file) : ConstantPoolEntry[3](_class_file, CONSTANT_Integer[4]
        )
214:                {
215:                }
216:                virtual Result[5] resolve[6]();
217:        };
218:
219:        struct CONSTANT_Float_info : public ConstantPoolEntry[7]
220:        {
221:            u4[8] bytes;
222:                virtual Result[5] read[9](u1[10] *& stream_pointer);
223:                void debug_print[11](ostream & out = cout);
224:            float get_float_value[12]();
225:                CONSTANT_Float_info(ClassFile[2] * _class_file) : ConstantPoolEntry[3](_class_file, CONSTANT_Float[13])
226:                {
227:                }
228:                virtual Result[5] resolve[14]();
229:        };
230:
231:        struct CONSTANT_Long_info : public ConstantPoolEntry[7]
232:        {
233:            u4[8] high_bytes;
234:            u4[8] low_bytes;
235:                virtual Result[5] read[15](u1[10] *& stream_pointer);
236:                void debug_print[16](ostream & out = cout);
237:                su8[17] get_long_value[18]();
238:                CONSTANT_Long_info(ClassFile[2] * _class_file) : ConstantPoolEntry[3](_class_file, CONSTANT_Long[19])
239:                {
240:                }
241:                virtual Result[5] resolve[20]();
242:        };
243:
244:        struct CONSTANT_Double_info : public ConstantPoolEntry[7]
245:        {
246:            u4[8] high_bytes;
247:            u4[8] low_bytes;
248:                virtual Result[5] read[21](u1[10] *& stream_pointer);
249:                void debug_print[22](ostream & out = cout);
250:                double get_double_value[23]();
251:                static double get_double_value[24](su8[17] bits);
252:                CONSTANT_Double_info(ClassFile[2] * _class_file) : ConstantPoolEntry[3](_class_file, CONSTANT_Double[25])
253:                {
254:                }
255:                virtual Result[5] resolve[26]();
256:        };
257:
258:        struct CONSTANT_NameAndType_info : public ConstantPoolEntry[7]
259:        {
260:            u2[27] name_index;
```

Footnotes:

1: *ConstantPool.h:206*
2: *ClassFile.h:136*
3: *ConstantPool.h:35*
4: *defs.h:331*
5: *defs.h:29*
6: *ConstantPool.cpp:764*
7: *ConstantPool.h:20*
8: *defs.h:174*
9: *ConstantPool.cpp:144*
10: *defs.h:168*
11: *ConstantPool.cpp:178*
12: *ConstantPool.cpp:152*
13: *defs.h:332*
14: *ConstantPool.cpp:173*
15: *ConstantPool.cpp:100*
16: *ConstantPool.cpp:139*
17: *defs.h:177*
18: *ConstantPool.cpp:111*
19: *defs.h:333*
20: *ConstantPool.cpp:134*
21: *ConstantPool.cpp:183*
22: *ConstantPool.cpp:223*
23: *ConstantPool.cpp:212*
24: *ConstantPool.cpp:193*
25: *defs.h:334*
26: *ConstantPool.cpp:218*
27: *defs.h:172*

```
261:             u2[1] descriptor_index;
262:                 virtual Result[2] read[3](u1[4] *& stream_pointer);
263:                 void debug_print[5](ostream & out = cout);
264:                 CONSTANT_NameAndType_info(ClassFile[6] * _class_file) : ConstantPoolEntry[7](_class_file, CONSTANT_Nam
        eAndType[8])
265:                 {
266:                 }
267:                 virtual Result[2] resolve[9]();
268:         };
269:
270:         /*
271:         * Constant Pool itself
272:         */
273:
274:         class ConstantPool
275:         {
276:
277:                 friend class ClassFile[6];
278:                 friend struct field_info[10];
279:                 friend struct method_info[11];
280:
281:         private:
282:                 ClassFile[6] * class_file; // the owner of this constant pool
283:                 long num_of_items;
284:                 ConstantPoolEntry[12] ** table;
285:
286:                 Result[2] read_constant_pool_entry[13](u1[4] tag, u1[4] *& stream_pointer, ConstantPoolEntry[12] *& cpe);
287:                 Result[2] read[14](u1[4] *& stream_pointer);
288:
289:         public:
290:                 ConstantPool(ClassFile[6] * _class_file, long _num_of_items) : class_file[15](_class_file),
291:                         num_of_items[16](_num_of_items[17])
292:                 {
293:                         table[18] = new ConstantPoolEntry[7]*[num_of_items[16]];
294:                         assert(table[18] != 0);
295:                 }
296:                 ~ConstantPool() {}
297:
298:                 // Items are indexed from 1 to numOfItems-1
299:                 ConstantPoolEntry[12] * get_item_at_index(long index)
300:                 {
301:                         assert (index[19] >= 1 && index[19] <= num_of_items[16]);
302:                         return table[18][index[19]];
303:                 }
304:
305:                 void debug_print[20](ostream & out = cout);
306:         };
307:
308:
309:
310:
311:         #endif // CONSTANT_POOL_H
312:
```

Footnotes:

1: *defs.h:172*
2: *defs.h:29*
3: *ConstantPool.cpp:775*
4: *defs.h:168*
5: *ConstantPool.cpp:790*
6: *ClassFile.h:136*
7: *ConstantPool.h:35*
8: *defs.h:335*
9: *ConstantPool.cpp:785*
10: *ClassFile.h:68*
11: *ClassFile.h:92*
12: *ConstantPool.h:20*
13: *ConstantPool.cpp:798*
14: *ConstantPool.cpp:846*
15: *ConstantPool.h:282*
16: *ConstantPool.h:283*
17: *ConstantPool.h:290*
18: *ConstantPool.h:284*
19: *ConstantPool.h:299*
20: *ConstantPool.cpp:876*

```
 1:          /*
 2:           * Class to manage the Garbage Collector.
 3:           *
 4:           */
 5:
 6:          #ifndef GARBAGE_COLLECTOR_H[1]
 7:          #define GARBAGE_COLLECTOR_H
 8:
 9:          #include "defs.h"
10:          #include "VirtualMachine.h"
11:          #include "Thread.h"
12:
13:          // Garbage collector is the VERY special thread.
14:          // In fact, it has nothing to do with the regular one except that it has the same
15:          // "step" function. Thus, we make it the proper candidate for execution that can be
16:          // picked up by the execution engine.
17:          // Usually, it will run at the lowest priority, but when needed its priority will boost
18:          // to the SYS_PRIORITY
19:          class GarbageCollector : public Thread[2]
20:          {
21:                  friend class VirtualMachine[3];
22:          private:
23:                  // This counter is used be the mark-and-sweep process
24:                  unsigned long run_counter;
25:
26:                  // Recursively mark all the instance fields that contain references to other instances;
27:                  // These two functions will interleave their recursive calls to one another
28:                  // traversing along all hierarchies of the interconnected instances.
29:              // Cycle references (when objects point to one another) will be avoided checking the
30:                  // memory chunk's mark counter. If the counter will be equal to the current one that means
31:                  // we have already investigated this object and should stop the recursion
32:                  void mark_reference_fields[4](InstanceData[5] * id);
33:                  void mark_reference_fields[6](ClassFile[7] * cf, InstanceData[5] * id);
34:
35:          public:
36:                  GarbageCollector(VirtualMachine[3] * _vm, unsigned long id) :
37:                    Thread[8](_vm[9], id[10]), run_counter[11](0)
38:                    {
39:                            state[12] = Running[13]; // garbage collector is always running
40:                    }
41:
42:                    ~GarbageCollector()
43:                    {
44:                    }
45:
46:                  // The "mark" phase of the garbage collector will visit all the instance and class roots
47:                  // and mark them with the current GC run counter:
48:                  // Instance roots are all the references that are stored on all thread's stack frames -
49:                      // both in local variable storage and operand stack;
50:                  // For each root all the references pointed by its fields are marked recursively;
51:                  // Class roots are the static class data stored in the class files that, in turn, are
52:                  // reachable from the namespaces of all the JVM class loaders.
53:                      // The "sweep" phase is working with instance and class heaps deallocating all the memory
```

Footnotes:

1: *GarbageCollector.h:7*
2: *Thread.h:17*
3: *VirtualMachine.h:255*
4: *GarbageCollector.cpp:141*
5: *ObjectData.h:113*
6: *GarbageCollector.cpp:98*
7: *ClassFile.h:136*
8: *Thread.h:73*
9: *GarbageCollector.h:36*
10: *GarbageCollector.h:36*
11: *GarbageCollector.h:24*
12: *Thread.h:45*
13: *Thread.h:25*

```
54:                     // chunks whose mark counter is less than the current GC run counter
55:                     virtual Result[1] step[2]();
56:             };
57:
58:             #endif // GARBAGE_COLLECTOR_H
59:
```

Footnotes:

1: *defs.h:29*
2: *GarbageCollector.cpp:11*

```
 1:  #ifndef HANDLE_POOL_H[1]
 2:  #define HANDLE_POOL_H
 3:
 4:  #include "hashtable.h"
 5:  #include "VirtualMachine.h"
 6:  #include "ObjectData.h"
 7:
 8:  // This is a storage of all the instances created so far;
 9:  // It handles the native pointers to the instances in the instance heap
10:  // and maps them to the abstract handles that represent the references
11:  // for all the JVM operations;
12:  // Thus, even if the native pointer is longer than word it still will work fine.
13:  class HandlePool
14:  {
15:          friend class GarbageCollector[2];
16:
17:  private:
18:          VirtualMachine[3] * vm;
19:
20:          // This is the storage of all the instances created so far
21:          HashTable[4] pool;
22:
23:          word[5] counter; // this will store the current available reference number;
24:                           // thus, we can have 4294967296 references
25:
26:          // Interned string table will contain all the string literals and interned strings
27:          // no more than once.
28:          // The key for this table is the ascii (TODO!) string representation;
29:          // The value is the InstanceData pointer to the String object representing this string
30:          HashTable[4] interned_strings;
31:
32:          // This is the collection of all the reachable objects at the current moment.
33:          // The table is "multi" because the same reference can be stored several times
34:          // by different stack frames. However, the Vector is not used because the references
35:          // must be thrown away efficiently once a stack frame is destroyed (together with
36:          // all its root set)
37:          MultiHashTable[6] roots;
38:
39:  public:
40:          // Note, that the reference counter is started with 1,
41:          // this is becasue the "null" reference is represented as 0
42:          HandlePool(VirtualMachine[3] * _vm) : vm[7](_vm), counter[8](1)
43:          {
44:          }
45:
46:          // Stores the new instance in the pool and returns the abstract handle (reference) for it
47:          word[5] put_instance[9](InstanceData[10] * id);
48:
49:          // Given a reference returns the native pointer to the instance data
50:          InstanceData[10] * get_instance[11](word[5] reference);
51:
52:          // If the string is not already in the table the function will "intern" it
53:          // (put in the table and create the String object for it) and return this
```

Footnotes:

1: *HandlePool.h:2*
2: *GarbageCollector.h:19*
3: *VirtualMachine.h:255*
4: *hashtable.h:28*
5: *defs.h:195*
6: *hashtable.h:96*
7: *HandlePool.h:18*
8: *HandlePool.h:23*
9: *HandlePool.cpp:12*
10: *ObjectData.h:113*
11: *HandlePool.cpp:25*

```
54:                 // object reference;
55:                 // Otherwise, it will return the reference to the existing object
56:                 word[1] intern_string[2](CONSTANT_Utf8_info[3] * utf8_info_string);
57:
58:                 // The given instance data must represent an instance of type java.lang.String;
59:                 // then, interned strings table lookup is performed and the character representation
60:                 // of the given String is returned
61:                 char * get_interned_string[4](InstanceData[5] * id);
62:
63:                 // Called when a stack frame pushes a reference to its operand stack or stores
64:                 // a reference into its local variable
65:                 void put_root[6](InstanceData[5] * root);
66:
67:                 // Called when a stack frame is destroyed together with all its "roots"
68:                 void remove_roots[7](Vector[8]<InstanceData*> * frame_roots);
69:         };
70:
71:         #endif // HANDLE_POOL_H
```

Footnotes:

1: *defs.h:195*
2: *HandlePool.cpp:49*
3: *ConstantPool.h:186*
4: *HandlePool.cpp:34*
5: *ObjectData.h:113*
6: *HandlePool.cpp:145*
7: *HandlePool.cpp:155*
8: *vector.h:7*

```
1:  /*
2:   * Class to deal with the Global Heap.
3:   *
4:   */
5:
6:  #ifndef HEAP_MANAGER_H[1]
7:  #define HEAP_MANAGER_H
8:
9:  #include "defs.h"
10: #include "dlist.h"
11:
12:
13: /*****************************************************************************************
14:
15:                                         Heap
16:                                         ----
17:
18: The Java virtual machine has a heap that is shared among all Java virtual machine threads.
19: The heap is the runtime data area from which memory for all class instances and arrays
20: is allocated.
21:
22: The heap is created on virtual machine start-up. Heap storage for objects is reclaimed
23: by an automatic storage management system known as a garbage collector (see "GarbageCollector.h");
24: objects are never explicitly deallocated.
25: The memory for the heap does not need to be contiguous (although it is continuous in this
26: implementation).
27:
28: A Java virtual machine implementation may provide the programmer or the user control over
29: the initial size of the heap, as well as, if the heap can be dynamically expanded or
30: contracted, control over the maximum and minimum heap size
31:
32: If a computation requires more heap than can be made available by the automatic storage
33: management system, the Java virtual machine throws an OutOfMemoryError.
34:
35: *****************************************************************************************/
36:
37:
38: // memory_chunk is the basic memory unit for all the objects residing on the heap.
39: // No direct addressing to the heap is possible, because in the process of heap defragmentation
40: // real memory blocks can be relocated.
41: // The memory_chunk structure is used to make the real addresses on the heap abstract.
42: // Object can be taken using the get_object() function of the memory_chunk and casting to
43: // the appropriate type. This is not type-safe, of course
44: struct memory_chunk
45: {
46:         u1[2] * address; // points to the memory address of this chunk
47:         u1[2] free; // free=1 if chunk is free, free=0 if chunk is used
48:         unsigned long size; // size of the chunk in bytes
49:
50:         // This counter is used by the garbage collector:
51:         // on the "mark" phase it will mark all reachable memory chunks with the current
52:         // mark counter;
53:         // on the "sweep" phase the memory chunks whose mark_counter is less than the current
```

Footnotes:
1: *HeapManager.h:7*
2: *defs.h:168*

```
54:                 // one will be deallocated (since they are not reachable)
55:                 unsigned long mark_counter;
56:                 // Will increase with each garbage collection pass. Its value means the number of
57:                 // garbage collections that this memory chunk has survived
58:                 unsigned long generation;
59:
60:                 memory_chunk(u1[1] * _address, unsigned long _size, u1[1] _free) :
61:                         address[2](_address[3]), size[4](_size[5]), free[6](_free[7]), mark_counter[8](0), generation[9](0)
62:                 {
63:                 }
64:
65:                 void * get_object()
66:                 {
67:                         return address[2];
68:                 }
69:
70:                 void mark(unsigned long counter)
71:                 {
72:                         mark_counter[8] = counter[10];
73:                         generation[9]++;
74:                 }
75:
76:                 void debug_print[11]();
77:         };
78:
79:         class VirtualMachine[12];
80:
81:         class HeapManager
82:         {
83:         private:
84:                 VirtualMachine[12] * p_VirtualMachine;
85:
86:                 unsigned long heap_size;
87:                 void * heap_start; // points to the first byte of the whole heap
88:                 DList[13] chunk_list; // contains the chunks of used and free memory
89:
90:                 // This function will defragment the heap up until the point there is
91:                 // amount of memory needed. Then the defragmentation stops;
92:                 // Success will be returned in case such amount is found
93:                 Result[14] just_in_case_defragment[15](unsigned long bytes_needed);
94:
95:                 // This heavy function will defragment all the heap until there is
96:                 // only one free memory chunk left
97:                 void total_defragment[16]();
98:
99:         public:
100:                HeapManager(VirtualMachine[12] * vm) : p_VirtualMachine[17](vm) {}
101:                ~HeapManager[18]();
102:
103:                Result[14] init[19](unsigned long bytes);
104:                Result[14] clean_up[20]();
105:
106:                // This function will return memory chunk pointing to the
```

Footnotes:

1: *defs.h:168*
2: *HeapManager.h:46*
3: *HeapManager.h:60*
4: *HeapManager.h:48*
5: *HeapManager.h:60*
6: *HeapManager.h:47*
7: *HeapManager.h:60*
8: *HeapManager.h:55*
9: *HeapManager.h:58*
10: *HeapManager.h:70*
11: *HeapManager.cpp:9*
12: *VirtualMachine.h:255*
13: *dlist.h:19*
14: *defs.h:29*
15: *HeapManager.cpp:199*
16: *HeapManager.cpp:261*
17: *HeapManager.h:84*
18: *HeapManager.cpp:52*
19: *HeapManager.cpp:19*
20: *HeapManager.cpp:35*

```
107:                // newely allocated object.
108:                // To get the object one has to call the get_object() function on it.
109:                // This design allows to relocate the chunk inside the heap without
110:                // influencing the allocated object itself.
111:                // NOTE: memory_chunk is created in this function, it will be deleted
112:                // on deallocation
113:                Result[1] alloc[2](unsigned long bytes, memory_chunk[3] *& chunk);
114:
115:                Result[1] realloc(unsigned long bytes, memory_chunk[3] *& chunk){}
116:
117:                // This function will reclaim the memory allocated by the given 'chunk'.
118:                // Note that in the process of deallocation all adjacent free memory blocks
119:                // will be merged together. That's why no two adjacent free chunks are ever possible
120:                Result[1] dealloc[4](memory_chunk[3] * chunk);
121:
122:                // This function is called by the garbage collector;
123:                // All memory chunks with mark_counter less than the given one will be deallocated
124:                // (since they are not reachable in the current cycle).
125:                // The function returns the number of chunks being deallocated
126:                unsigned long sweep[5](unsigned long mark_counter);
127:
128:                void dump[6]();
129:        };
130:
131:
132:
133:
134:        #endif // HEAP_MANAGER_H
135:
```

Footnotes:

1: *defs.h:29*
2: *HeapManager.cpp:60*
3: *HeapManager.h:44*
4: *HeapManager.cpp:117*
5: *HeapManager.cpp:265*
6: *HeapManager.cpp:290*

```
 1:          #ifndef NATIVE_HANDLER_H[1]
 2:          #define NATIVE_HANDLER_H
 3:
 4:          #include "defs.h"
 5:          #include "hashtable.h"
 6:          #include "jni.h"
 7:
 8:          class VirtualMachine[2];
 9:          class Thread[3];
10:          class InstanceData[4];
11:          struct method_info[5];
12:
13:          class NativeHandler
14:          {
15:          private:
16:                  // Points to the Java Native interface structure
17:                  JNINativeInterface_ * jni_native_interface_ptr;
18:                  // Points to the Java Native Environment structure
19:                  JNIEnv * jni_env_ptr;
20:
21:                  // Points to the Java Virtual Machine
22:                  VirtualMachine[2] * vm;
23:
24:                  // Contains all native methods defined so far
25:                  HashTable[6] table;
26:
27:                  void get_jni_compliant_method_name[7](method_info[5] * mi, char * full_name);
28:
29:          public:
30:                  NativeHandler[8](VirtualMachine[2] * _vm);
31:                  ~NativeHandler(){}
32:                  void init[9]();
33:
34:                  // This function is called by the registerNatives function of the native
35:                  // method implementations
36:                  void register_method[10](char * name, char * descriptor, void * func_ptr);
37:
38:                  // This function actually calls the native method;
39:                  // In case the method is static, InstanceData is NULL
40:                  Result[11] run_method[12](Thread[3] * thread, method_info[5] * mi, InstanceData[4] * id);
41:          };
42:
43:          #endif // NATIVE_HANDLER_H
```

Footnotes:

1: *NativeHandler.h:2*
2: *VirtualMachine.h:255*
3: *Thread.h:17*
4: *ObjectData.h:113*
5: *ClassFile.h:92*
6: *hashtable.h:28*
7: *NativeHandler.cpp:147*
8: *NativeHandler.cpp:96*
9: *NativeHandler.cpp:120*
10: *NativeHandler.cpp:136*
11: *defs.h:29*
12: *NativeHandler.cpp:189*

```
 1:         #ifndef OBJECT_DATA_H[1]
 2:         #define OBJECT_DATA_H
 3:
 4:         #include "jni.h" // Java Native Interface standard header
 5:
 6:         #include "HeapManager.h"
 7:         #include "ClassFile.h"
 8:         #include "vector.h"
 9:         #include "hashtable.h"
10:
11:         class Thread[2];
12:
13:         // Represents the synchronization monitor inherent to all objects
14:         struct monitor
15:         {
16:                 // The thread that owns this monitor
17:                 Thread[2] * owner;
18:                 // The number of times when this monitor was acquired by the same thread
19:                 unsigned long counter;
20:                 // The list of all threads that are waiting to acquire this monitor
21:                 Vector[3]<Thread*> waiting_set;
22:
23:                 monitor() : owner[4](NULL), counter[5](0)
24:                 {
25:                 }
26:
27:                 // The thread attempting to acquire the monitor will call this function;
28:                 // If the monitor is already acquired by another thread, this thread will
29:                 // be inserted in the waiting set of the monitor and its status will be set to Waiting;
30:                 // if the monitor is free, this thread will become the owner of the monitor and
31:                 // will continue to run;
32:                 // if the monitor is acquired by the same thread, the counter will be incremented
33:                 // and the thread will continue to run;
34:                 // Note, that the thread calling this function "is not aware" of the following events -
35:                 // for the thread the next instruction will be executed as usual but only after
36:                 // it will be released from the waiting set of this monitor
37:                 void acquire[6](Thread[2] * thread);
38:
39:                 // The function will return false if the thread attempting to release this monitor
40:                 // does not own it
41:                 int release[7](Thread[2] * thread);
42:         };
43:
44:
45:         // Represents the field of a class or an instance in the memory
46:         struct variable_offset
47:         {
48:                 u2[8] offset;
49:                 u2[8] size;
50:                 BasicType[9] type;
51:         };
52:
53:         class ClassFile[10];
```

Footnotes:

1: *ObjectData.h:2*
2: *Thread.h:17*
3: *vector.h:7*
4: *ObjectData.h:17*
5: *ObjectData.h:19*
6: *ObjectData.cpp:447*
7: *ObjectData.cpp:499*
8: *defs.h:172*
9: *defs.h:208*
10: *ClassFile.h:136*

```
54:
55:         // This class servers for both type and instance internal representation
56:         class ObjectData
57:         {
58:         protected:
59:
60:                 BasicType[1] get_variable_type[2](CONSTANT_Utf8_info[3] * utf8_info);
61:
62:         public:
63:                 // For ClassData:
64:                 // This memory chunk is pointing to the place in heap where this class' static
65:                 // fields are allocated.
66:                 // The field_info structures of the class file store the offsets of the
67:                 // corresponding fields inside this memory chunk
68:                 // For InstanceData:
69:                 // This memory chunk is pointing to the place in heap where this instance's
70:                 // fields are allocated.
71:                 memory_chunk[4] * data_start;
72:
73:                 // ClassFile associated with static or instance data
74:                 ClassFile[5] * class_file;
75:
76:                 // The monitor associated with this object
77:                 monitor[6] lock;
78:
79:                 // The set of threads that are waiting on this object by calling the "wait" function
80:                 Vector[7]<Thread*> waiting_set;
81:
82:                 ObjectData(ClassFile[5] * cf) : class_file[8](cf), data_start[9](NULL)
83:                 {
84:                 }
85:
86:         };
87:
88:         // This class contains all the class (static) variables of the given type.
89:         // The memory is allocated on the Heap (using the memory_chunk structure).
90:         // The access to the fields is performed using the offsets of the fields
91:         // inside the class data memory block
92:         class ClassData : public ObjectData[10]
93:         {
94:         private:
95:
96:
97:         public:
98:                 ClassData(ClassFile[5] * cf) : ObjectData[11](cf)
99:                 {
100:                }
101:
102:                // This function will be initially called on the ClassData.
103:                // It will look for all the statis variables of the type
104:                // and initialize the necessary memory block to hold them;
105:                // It will also set the variabes to their default initial values
106:                Result[12] prepare[13]();
```

Footnotes:

1: *defs.h:208*
2: *ObjectData.cpp:124*
3: *ConstantPool.h:186*
4: *HeapManager.h:44*
5: *ClassFile.h:136*
6: *ObjectData.h:14*
7: *vector.h:7*
8: *ObjectData.h:74*
9: *ObjectData.h:71*
10: *ObjectData.h:56*
11: *ObjectData.h:82*
12: *defs.h:29*
13: *ObjectData.cpp:41*

```
107:        };
108:
109:
110:        // When the JVM creates a new instance of a class, the memory on the heap
111:        // is allocated for all instance variables of this class and all its
112:        // supersclasses.
113:        class InstanceData : public ObjectData[1]
114:        {
115:                friend class HandlePool[2];
116:
117:        private:
118:                void free_vector[3](Vector[4]<variable_offset*> * v);
119:
120:                // This is the unique reference id associated with this instance
121:                word[5] reference;
122:
123:                // Recursive call to gather the information about all instance fields
124:                Result[6] get_class_fields[7](ClassFile[8] * cf, Vector[4]<variable_offset*> * variables, int me);
125:
126:        public:
127:                InstanceData(ClassFile[8] * cf) : ObjectData[9](cf)
128:                {
129:                }
130:
131:                // This function will create the new instance of the given class.
132:                // It will look for all the instance variables of the class and all the
133:                // instance variables of its superclasses and initialize the necessary
134:                // memory block to hold them;
135:                Result[6] create[10]();
136:
137:                // After the instance is created its <init> method should be called
138:                // NOTE: this method is called within the same thread of execution
139:                // which caused creation of this instance
140:                Result[6] initialize();
141:
142:                inline word[5] get_reference() const { return reference[11]; }
143:
144:                virtual int is_array() { return 0; }
145:                virtual int is_primitive_array() { return 0; }
146:                virtual Result[6] copy_from(InstanceData[12] * src, u4[13] src_offset, u4[13] dst_offset, u4[13] length)
147:                {
148:                        return Failure[14]; // does not have any meaning on non-array objects
149:                }
150:
151:                void debug_print[15](ostream & out = cout);
152:        };
153:
154:        // Array object instance internal memory representation
155:        class ArrayInstanceData : public InstanceData[12]
156:        {
157:        protected:
158:                u4[13] array_length;
159:        public:
```

Footnotes:

1: *ObjectData.h:56*
2: *HandlePool.h:13*
3: *ObjectData.cpp:290*
4: *vector.h:7*
5: *defs.h:195*
6: *defs.h:29*
7: *ObjectData.cpp:168*
8: *ClassFile.h:136*
9: *ObjectData.h:82*
10: *ObjectData.cpp:236*
11: *ObjectData.h:121*
12: *ObjectData.h:113*
13: *defs.h:174*
14: *defs.h:34*
15: *ObjectData.cpp:328*

```
160:                ArrayInstanceData(ClassFile[1] * cf) : InstanceData[2](cf), array_length[3](0)
161:                {
162:                }
163:                u4[4] get_array_length() const { return array_length[3]; }
164:
165:                virtual Result[5] create[6](u4[4] count);
166:                int is_convertable(InstanceData[7] * id){}
167:                virtual Result[5] copy_from[8](InstanceData[7] * src, u4[4] src_offset, u4[4] dst_offset, u4[4] length);
168:
169:                virtual int is_array() { return 1; }
170:                virtual int is_primitive_array() { return 0; }
171:        };
172:
173:        // Array of primitive data types internal memory representation
174:        class PrimitiveArrayInstanceData : public ArrayInstanceData[9]
175:        {
176:        public:
177:                PrimitiveArrayInstanceData(ClassFile[1] * cf) : ArrayInstanceData[10](cf)
178:                {
179:                }
180:                // This will create and initialize to the default values an array of the
181:                // primitive type values of the given length
182:                virtual Result[5] create[11](u4[4] count);
183:                // Returns true if the given primitive type is convertable to the type of this array
184:                int is_convertable[12](BasicType[13] from_type);
185:
186:                virtual Result[5] copy_from[14](InstanceData[7] * src, u4[4] src_offset, u4[4] dst_offset, u4[4] length);
187:
188:                virtual int is_array() { return 1; }
189:                virtual int is_primitive_array() { return 1; }
190:
191:                BasicType[13] get_internal_type[15](u1[16] array_type);
192:        };
193:
194:
195:        #endif // OBJECT_DATA_H
```

Footnotes:

1: *ClassFile.h:136*
2: *ObjectData.h:127*
3: *ObjectData.h:158*
4: *defs.h:174*
5: *defs.h:29*
6: *ObjectData.cpp:303*
7: *ObjectData.h:113*
8: *ObjectData.cpp:388*
9: *ObjectData.h:155*
10: *ObjectData.h:160*
11: *ObjectData.cpp:358*
12: *ObjectData.cpp:394*
13: *defs.h:208*
14: *ObjectData.cpp:400*
15: *ObjectData.cpp:339*
16: *defs.h:168*

```
#ifndef STACK_H[1]
#define STACK_H

#include <iostream.h>

#include "defs.h"
#include "vector.h"
#include "HeapManager.h"
#include "Thread.h"
#include "VirtualMachine.h"
#include "ObjectData.h"

class ClassFile[2];
class ConstantPool[3];
class Thread[4];
class InstanceData[5];
struct Code_attribute[6];

/*****************************************************************************************

                                        Frames
                                        ------

A frame is used to store data and partial results, as well as to perform dynamic linking,
return values for methods, and dispatch exceptions.
A new frame is created each time a method is invoked.
A frame is destroyed when its method invocation completes, whether that completion is normal
or abrupt (it throws an uncaught exception).
Frames are allocated from the Java virtual machine stack of the thread creating the frame.
Each frame has its own array of local variables, its own operand stack, and a reference
to the runtime constant pool of the class of the current method.

The sizes of the local variable array and the operand stack are determined at compile time
and are supplied along with the code for the method associated with the frame.
Thus the size of the frame data structure depends only on the implementation of the Java
virtual machine, and the memory for these structures can be allocated simultaneously on
method invocation.

Only one frame, the frame for the executing method, is active at any point in a given thread
of control. This frame is referred to as the current frame, and its method is known as
the current method. The class in which the current method is defined is the current class.
Operations on local variables and the operand stack are typically with reference
to the current frame.

A frame ceases to be current if its method invokes another method or if its method completes.
When a method is invoked, a new frame is created and becomes current when control transfers
to the new method. On method return, the current frame passes back the result of its method
invocation, if any, to the previous frame. The current frame is then discarded as the
previous frame becomes the current one.

Note that a frame created by a thread is local to that thread and cannot be referenced
by any other thread.
```

Footnotes:

1: *Stack.h:2*
2: *ClassFile.h:136*
3: *ConstantPool.h:274*
4: *Thread.h:17*
5: *ObjectData.h:113*
6: *AttributeInfo.h:55*

```
54:
55:                                     Local Variables
56:                                     ---------------
57:
58:         Each frame contains an array of variables known as its local variables.
59:         The length of the local variable array of a frame is determined at compile time and supplied
60:         in the binary representation of a class or interface along with the code for the method
61:         associated with the frame.
62:         A single local variable can hold a value of type boolean, byte, char, short, int, float,
63:         reference, or returnAddress.
64:         A pair of local variables can hold a value of type long or double.
65:
66:         Local variables are addressed by indexing. The index of the first local variable is zero.
67:         An integer is be considered to be an index into the local variable array if and only
68:         if that integer is between zero and one less than the size of the local variable array.
69:
70:         A value of type long or type double occupies two consecutive local variables.
71:         Such a value may only be addressed using the lesser index. For example, a value of type
72:         double stored in the local variable array at index n actually occupies the local variables
73:         with indices n and n+1; however, the local variable at index n+1 cannot be loaded from.
74:         It can be stored into. However, doing so invalidates the contents of local variable n.
75:         The Java virtual machine does not require n to be even. In intuitive terms, values of types
76:         double and long need not be 64-bit aligned in the local variables array.
77:         Implementors are free to decide the appropriate way to represent such values using the two
78:         local variables reserved for the value.
79:
80:         The Java virtual machine uses local variables to pass parameters on method invocation.
81:         On class method invocation any parameters are passed in consecutive local variables starting
82:         from local variable 0. On instance method invocation, local variable 0 is always used
83:         to pass a reference to the object on which the instance method is being invoked
84:         (this in the Java programming language). Any parameters are subsequently passed
85:         in consecutive local variables starting from local variable 1.
86:
87:
88:                                      Operand Stacks
89:                                      --------------
90:
91:         Each frame contains a last-in-first-out (LIFO) stack known as its operand stack.
92:         The maximum depth of the operand stack of a frame is determined at compile time and
93:         is supplied along with the code for the method associated with the frame.
94:         Where it is clear by context, we will sometimes refer to the operand stack of the current
95:         frame as simply the operand stack.
96:
97:         The operand stack is empty when the frame that contains it is created.
98:         The Java virtual machine supplies instructions to load constants or values from
99:         local variables or fields onto the operand stack. Other Java virtual machine instructions
100:        take operands from the operand stack, operate on them, and push the result back onto
101:        the operand stack. The operand stack is also used to prepare parameters to be passed
102:        to methods and to receive method results.
103:
104:        For example, the iadd instruction adds two int values together. It requires that the
105:        int values to be added be the top two values of the operand stack, pushed there by previous
106:        instructions. Both of the int values are popped from the operand stack.
```

```
107:         They are added, and their sum is pushed back onto the operand stack.
108:         Subcomputations may be nested on the operand stack, resulting in values that can be used by
109:         the encompassing computation.
110:
111:         Each entry on the operand stack can hold a value of any Java virtual machine type,
112:         including a value of type long or type double.
113:
114:         Values from the operand stack must be operated upon in ways appropriate to their types.
115:         It is not possible, for example, to push two int values and subsequently treat them
116:         as a long or to push two float values and subsequently add them with an iadd instruction.
117:         A small number of Java virtual machine instructions (the dup instructions and swap)
118:         operate on runtime data areas as raw values without regard to their specific types;
119:         these instructions are defined in such a way that they cannot be used to modify or break up
120:         individual values. These restrictions on operand stack manipulation are enforced
121:         through class file verification.
122:
123:         At any point in time an operand stack has an associated depth, where a value of type
124:         long or double contributes two units to the depth and a value of any other type
125:         contributes one unit.
126:
127:         ***********************************************************************************************/
128:
129:         struct stack_frame
130:         {
131:                 Thread[1] * owner_thread; // thread that owns this stack
132:                 HandlePool[2] * handle_pool; // points to the global handle pool
133:
134:                 // This vector will store all object references that were either stored into the
135:                 // local variable or pushed on the operand stack;
136:                 // When the frame is destroyed (popped of the thread's stack) this set will be used
137:                 // to delete object references belonging to this stack from the global set of roots
138:                 Vector[3]<InstanceData*> roots;
139:                 // Total size of this frame stack
140:                 unsigned int size;
141:                 // Address of the whole stack frame memory block
142:                 word[4] * frame_start;
143:                 // Address of the beginning of the local variables array
144:                 word[4] * locals_start;
145:                 // Address of the beginning of the operand stack
146:                 word[4] * operand_stack_start;
147:                 // Index of the top of the operand stack;
148:                 // Operand stack grows downwards
149:                 u2[5] operand_stack_top;
150:
151:                 // The additional stack frame data (except of operand stack and local variables);
152:                 // it holds the current method information
153:                 Code_attribute[6] * current_code_attribute; // holds current class file, current constant pool,
154:                                                             // current code, code length
155:                 // The PC register will be assigned when pushing a new frame on the stack
156:                 word[4] current_pc_register;
157:
158:                 // In case the current method is synchronized this monitor will be kept
159:                 // during the execution to be quickly accessible for release upon return
```

Footnotes:

1: *Thread.h:17*
2: *HandlePool.h:13*
3: *vector.h:7*
4: *defs.h:195*
5: *defs.h:172*
6: *AttributeInfo.h:55*

```
160:                monitor[1] * monitor_to_release;
161:
162:                // Stores the number of words that must be popped from the calling method's stack
163:                // when the stack of the method being called returns;
164:                // These were the parameters pushed by the calling method's stack for the method
165:                // being called
166:                unsigned int words_to_pop;
167:
168:                // Stores the program counter of the currently executing instruction of the
169:                // method this stack frame belongs to;
170:                // This program counter may be used when an exception is thrown in the method
171:                // while searching for the appropriate exception table entry;
172:                // The instruction_pc_register changes along with the stack frames
173:                // while the search for the exception going down the stack;
174:            // Note the difference between the current_pc_register and the instruction_pc_register:
175:                // while former always points to the next opcode to be fetched, the latter points
176:                // to the opcode being executed (it is an important difference since we cannot always
177:                // know the exact number of instruction arguments)
178:                word[2] instruction_pc_register;
179:
180:                // Note, that max_locals and max_stack are measured in words
181:                stack_frame[3](Thread[4] * owner, u1[5] * _frame_start, u2[6] max_locals, u2[6] max_stack);
182:
183:                // This constructor will be used for the frames created by a method invokation;
184:                // The operand stack of the calling frame will become the local variables
185:                // of the frame being created
186:                stack_frame[7](Thread[4] * owner, u1[5] * _frame_start);
187:
188:                // Determine size and operand stack of the uppermost frame stack
189:                // (used in conjunction with the constructor for overlapped frames)
190:                void define_size[8](u2[6] max_locals, u2[6] max_stack);
191:
192:
193:                // Basic operations over the operand stack
194:
195:                void push_int[9](su4[10] value);
196:                void push_long[11](su8[12] value);
197:                void push_float[13](float value);
198:                void push_double[14](double value);
199:                void push_reference[15](word[2] value);
200:                void push_returnType[16](word[2] value);
201:
202:                su4[10] pop_int[17]();
203:                su8[12] pop_long[18]();
204:                float pop_float[19]();
205:                double pop_double[20]();
206:                word[2] pop_reference[21]();
207:                word[2] pop_returnType[22]();
208:
209:                void pop_word[23]();
210:                void pop2_word[24]();
211:                void dup_word[25]();
212:                void dup2_word[26]();
```

Footnotes:

1: *ObjectData.h:14*
2: *defs.h:195*
3: *Stack.cpp:94*
4: *Thread.h:17*
5: *defs.h:168*
6: *defs.h:172*
7: *Stack.cpp:116*
8: *Stack.cpp:135*
9: *Stack.cpp:147*
10: *defs.h:175*
11: *Stack.cpp:153*
12: *defs.h:177*
13: *Stack.cpp:165*
14: *Stack.cpp:171*
15: *Stack.cpp:187*
16: *Stack.cpp:201*
17: *Stack.cpp:206*
18: *Stack.cpp:214*
19: *Stack.cpp:227*
20: *Stack.cpp:235*
21: *Stack.cpp:249*
22: *Stack.cpp:254*
23: *Stack.cpp:259*
24: *Stack.cpp:264*
25: *Stack.cpp:269*
26: *Stack.cpp:275*

```
213:                void swap_word[1]();
214:                void dup_x1[2]();
215:                void pop_words[3](unsigned int n);
216:
217:                // Basic operations with the local variables
218:
219:                void store_int[4](su4[5] value, u1[6] index);
220:                void store_long[7](su8[8] value, u1[6] index);
221:                void store_float[9](float value, u1[6] index);
222:                void store_double[10](double value, u1[6] index);
223:                void store_reference[11](word[12] value, u1[6] index);
224:                void store_returnType[13](word[12] value, u1[6] index);
225:
226:                su4[5] get_int[14](u1[6] index);
227:                su8[8] get_long[15](u1[6] index);
228:                float get_float[16](u1[6] index);
229:                double get_double[17](u1[6] index);
230:                word[12] get_reference[18](u1[6] index);
231:                word[12] get_returnType[19](u1[6] index);
232:
233:                void debug_print[20](ostream & out = cout);
234:        };
235:
236:        class Thread[21];
237:
238:
239:
240:        /*********************************************************************************************
241:
242:                                            JVM Stacks
243:                                            ----------
244:
245:        Each Java virtual machine thread has a private Java virtual machine stack,
246:        created at the same time as the thread.
247:        A Java virtual machine stack is analogous to the stack of a conventional language such as C:
248:        it holds local variables and partial results, and plays a part in method invocation and
249:        return. Because the Java virtual machine stack is never manipulated directly except
250:        to push and pop frames, frames may be heap allocated.
251:        The memory for a Java virtual machine stack does not need to be contiguous.
252:
253:        The Java virtual machine specification permits Java virtual machine stacks either to be
254:        of a fixed size or to dynamically expand and contract as required by the computation.
255:        If the Java virtual machine stacks are of a fixed size, the size of each Java virtual machine
256:        stack may be chosen independently when that stack is created.
257:        A Java virtual machine implementation may provide the programmer or the user control over
258:        the initial size of Java virtual machine stacks, as well as, in the case of dynamically
259:        expanding or contracting Java virtual machine stacks, control over the maximum and minimum
260:        sizes.
261:
262:        *********************************************************************************************/
263:
264:        class Stack
265:        {
```

Footnotes:

1: *Stack.cpp:292*
2: *Stack.cpp:283*
3: *Stack.cpp:300*
4: *Stack.cpp:305*
5: *defs.h:175*
6: *defs.h:168*
7: *Stack.cpp:310*
8: *defs.h:177*
9: *Stack.cpp:320*
10: *Stack.cpp:325*
11: *Stack.cpp:338*
12: *defs.h:195*
13: *Stack.cpp:352*
14: *Stack.cpp:357*
15: *Stack.cpp:364*
16: *Stack.cpp:376*
17: *Stack.cpp:383*
18: *Stack.cpp:396*
19: *Stack.cpp:403*
20: *Stack.cpp:410*
21: *Thread.h:17*

```
266:                friend class GarbageCollector[1];
267:
268:        private:
269:                Thread[2] * owner_thread; // thread that owns this stack
270:
271:                // points to the beginning of the whole memory block allocated for this stack
272:                memory_chunk[3] * stack_start;
273:
274:                // contains all frames
275:                Vector[4]<stack_frame*> frames;
276:
277:        public:
278:                Stack[5](Thread[2] * _owner_thread);
279:                ~Stack[6]();
280:
281:                // Creates and pushes a new stack frame on top of the stack;
282:                // returns pointer to the newely created stack frame;
283:                stack_frame[7] * push_frame[8](u2[9] max_locals, u2[9] max_stack);
284:
285:                // This function will be used for the frames created by a method invokation;
286:                // The operand stack of the calling frame will become the local variables
287:                // of the frame being created
288:                stack_frame[7] * push_overlapped_frame[10](word[11] * next_frame_start);
289:
290:                // Pops the top stack frame off the stack and destroys it
291:                void pop_frame[12]();
292:
293:                // Returns the topmost frame on the stack;
294:                // If the stack is empty returns NULL
295:                stack_frame[7] * get_top_frame[13]();
296:
297:                unsigned int count_stack_frames() const { return frames[14].size[15](); }
298:
299:                void debug_print[16](ostream & out = cout);
300:        };
301:
302:        #endif // STACK_H
```

Footnotes:

1: *GarbageCollector.h:19*
2: *Thread.h:17*
3: *HeapManager.h:44*
4: *vector.h:7*
5: *Stack.cpp:13*
6: *Stack.cpp:18*
7: *Stack.h:129*
8: *Stack.cpp:25*
9: *defs.h:172*
10: *Stack.cpp:45*
11: *defs.h:195*
12: *Stack.cpp:54*
13: *Stack.cpp:68*
14: *Stack.h:275*
15: *vector.h:23*
16: *Stack.cpp:75*

```
 1:          #ifndef THREAD_H[1]
 2:          #define THREAD_H
 3:
 4:          #include "defs.h"
 5:          #include "AttributeInfo.h"
 6:          #include "Stack.h"
 7:          #include "ConstantPool.h"
 8:
 9:          class Virtual_Machine;
10:          class ClassFile[2];
11:          class Stack[3];
12:          class InstanceData[4];
13:          struct stack_frame[5];
14:          struct monitor[6];
15:
16:          // The abstraction of an execution thread
17:          class Thread
18:          {
19:                  friend class ExecutionPool[7];
20:
21:          protected:
22:                  enum State
23:                  {
24:                          Dead,    // thread finished running
25:                  Running, // thread is running (is able to execute next instruction)
26:                          Waiting, // thread is blocked on the waiting set of a monitor
27:                          Native,  // thread is executing the native method
28:                          Yield,   // thread decided to behave nicely and yielded execution to another candidate
29:                          Superceded
30:                  };
31:
32:                  VirtualMachine[8] * vm;
33:
34:                  // Unique thread id
35:                  unsigned long id;
36:                  // Current thread's priority
37:                  Priority[9] priority;
38:                  // This function can be called by the ExecutionPool only !
39:                  inline void set_priority(Priority[9] p) { priority[10] = p; }
40:
41:                  // This is the Java stack that belongs to this thread
42:                  Stack[3] * stack;
43:
44:                  // Current thread state
45:                  State[11] state;
46:
47:                  // Deallocate resources allocated by this thread
48:                  void destroy[12]();
49:
50:                  // This function switches the contexts between the stack frame of the calling method
51:                  // and the stack frame of the new method being called (going up the stack);
52:                  // After this function is called the next "step" instruction will get the first
53:                  // opcode of the invoked method
```

Footnotes:

1: *Thread.h:2*
2: *ClassFile.h:136*
3: *Stack.h:264*
4: *ObjectData.h:113*
5: *Stack.h:129*
6: *ObjectData.h:14*
7: *VirtualMachine.h:355*
8: *VirtualMachine.h:255*
9: *defs.h:396*
10: *Thread.h:37*
11: *Thread.h:22*
12: *Thread.cpp:56*

```
54:                 void context_switch_up[1](ClassFile[2] * new_method_class_file,
55:                                           Code_attribute[3] * new_code_attribute,
56:                                           stack_frame[4] * new_stack_frame);
57:
58:                 // This function switches contexts between the method finishing the execution
59:                 // and the method that called it (going down the stack)
60:                 // After this function is called the next "step" instruction will get the next
61:                 // opcode (after the invokeXXX instruction) of the thread that becomes current
62:                 void context_switch_down[5](stack_frame[4] * prev_stack_frame);
63:
64:                 // In case the current method is synchronized the thread will release the
65:                 // monitor acquired upon entering the method
66:                 Result[6] release_monitor[7]();
67:
68:                 // Used by all the invokation instructions in case the method is native;
69:                 // in case the method is static InstanceData is NULL
70:                 Result[6] invoke_native_method[8](method_info[9] * mi, InstanceData[10] * id = NULL);
71:
72:         public:
73:                 Thread(VirtualMachine[11] * _vm, unsigned long _id) : vm[12](_vm), id[13](_id), stack[14](NULL),
74:                                                                       lock_counter[15](0), debug[16](0),
75:
            raised_exception_instance[17](NULL)
76:                 {
77:                 }
78:                 virtual ~Thread()
79:                 {
80:                 }
81:
82:                 inline unsigned long get_id() { return id[13]; }
83:
84:                 // Entry point to this thread's execution
85:                 Result[6] start[18](Code_attribute[3] * code_attribute, InstanceData[10] * _thread_instance);
86:
87:                 // Executes the current instruction of the thread
88:                 virtual Result[6] step[19]();
89:
90:                 // Method invocation procedures
91:
92:                 // is called by invokestatic instruction
93:                 Result[6] invoke_static_method[20](CONSTANT_Methodref_info[21] * entry);
94:
95:                 // is called by invokevirtual instruction;
96:                 // this instruction is used to call virtual methods
97:                 Result[6] invoke_instance_method[22](CONSTANT_Methodref_info[21] * entry);
98:
99:                 // is called by invokespecial instruction;
100:                // this instruction is used to call superclass, private and
101:                // instance initialization (<init>) methods, but not the <clinit> method
102:                Result[6] invoke_special_method[23](CONSTANT_Methodref_info[21] * entry);
103:
104:                // is called by invokeinterface instruction
105:                Result[6] invoke_interface_method[24](CONSTANT_InterfaceMethodref_info[25] * entry, u1[26] count);
```

Footnotes:

1: *Thread.cpp:66*
2: *ClassFile.h:136*
3: *AttributeInfo.h:55*
4: *Stack.h:129*
5: *Thread.cpp:107*
6: *defs.h:29*
7: *Thread.cpp:126*
8: *Thread.cpp:737*
9: *ClassFile.h:92*
10: *ObjectData.h:113*
11: *VirtualMachine.h:255*
12: *Thread.h:32*
13: *Thread.h:35*
14: *Thread.h:42*
15: *Thread.h:145*
16: *Thread.h:184*
17: *Thread.h:134*
18: *Thread.cpp:17*
19: *Thread.cpp:907*
20: *Thread.cpp:457*
21: *ConstantPool.h:98*
22: *Thread.cpp:342*
23: *Thread.cpp:542*
24: *Thread.cpp:644*
25: *ConstantPool.h:156*
26: *defs.h:168*

```
106:
107:                    // Used by return instruction
108:                    Result[1] return_from_method[2]();
109:                    // Used by ireturn instruction
110:                    Result[1] return_int_from_method[3]();
111:                    // Used by lreturn instruction
112:                    Result[1] return_long_from_method[4]();
113:                    // Used by freturn instruction
114:                    Result[1] return_float_from_method[5]();
115:                    // Used by dreturn instruction
116:                    Result[1] return_double_from_method[6]();
117:                    // Used by areturn instruction
118:                    Result[1] return_reference_from_method[7]();
119:
120:                    // This function is used for the predefined exceptions;
121:                    // It is called from all the instructions where an exceptional situation occur;
122:                    // That is, it is a JVM-initiated exception
123:                    void raise_exception[8](Result[1] exception_code);
124:                    // This function is used when the class of the exception is user defined
125:                    // or not known from the exceptions array;
126:                    // Called from "athrow" instruction - that is, it is user initiated exception
127:                    void raise_exception[9](ClassFile[10] * exception_cf);
128:                    // This function is called when the exception handler is defined in the body
129:                    // of the same method where an exception is thrown
130:                    void handle_exception_locally[11](word[12] handler_pc);
131:                    // Implements the exception treatment mechanism
132:                    Result[1] treat_exception[13]();
133:                    // The exception itself; it is equal to NULL if no exception is raised
134:                    InstanceData[14] * raised_exception_instance;
135:
136:                    // The following three methods implement the java.lang.Object's corresponding
137:                    // synchronization methods;
138:                    // The methods are called upon an object passed as a parameter
139:                    Result[1] wait[15](InstanceData[14] * id);
140:                    Result[1] notify[16](InstanceData[14] * id);
141:                    Result[1] notifyAll[17](InstanceData[14] * id);
142:
143:                    // This lock_counter is used when the thread is awakened from the "wait" function
144:                    // call and it should restore the lock's state to be as before the call
145:                    unsigned long lock_counter;
146:
147:                    // These variables will be availabe to the instruction execution functions
148:                    stack_frame[18] * current_stack_frame;
149:                    ClassFile[10] * current_class_file;
150:                    ConstantPool[19] * current_cp;
151:                    u1[20] * current_code;
152:                    u2[21] current_code_length;
153:
154:                    // The code that is currently executed
155:                    Code_attribute[22] * current_code_attribute;
156:
157:                    // Program Counter register;
158:                    // Contains offset of the next instruction to execute from the beginning of the bytecodes
```

Footnotes:

1: *defs.h:29*
2: *Thread.cpp:142*
3: *Thread.cpp:170*
4: *Thread.cpp:204*
5: *Thread.cpp:238*
6: *Thread.cpp:272*
7: *Thread.cpp:306*
8: *Thread.cpp:776*
9: *Thread.cpp:765*
10: *ClassFile.h:136*
11: *Thread.cpp:808*
12: *defs.h:195*
13: *Thread.cpp:824*
14: *ObjectData.h:113*
15: *Thread.cpp:976*
16: *Thread.cpp:1027*
17: *Thread.cpp:1057*
18: *Stack.h:129*
19: *ConstantPool.h:274*
20: *defs.h:168*
21: *defs.h:172*
22: *AttributeInfo.h:55*

```
159:                word[1] pc_register;
160:
161:                // Points to the instance of type java.lang.Thread which this thread is associated with
162:                InstanceData[2] * thread_instance;
163:
164:                // Following functions deal with the thread states
165:                inline void set_dead() { state[3] = Dead[4]; }
166:                inline void set_running() { state[3] = Running[5]; }
167:                inline void set_waiting() { state[3] = Waiting[6]; }
168:                inline void set_native() { state[3] = Native[7]; }
169:                inline void set_yield() { state[3] = Yield[8]; }
170:                inline void set_superceded() { state[3] = Superceded[9]; }
171:
172:                inline int is_dead() { return (state[3] == Dead[4]); }
173:                inline int is_running() { return (state[3] == Running[5]); }
174:                inline int is_waiting() { return (state[3] == Waiting[6]); }
175:                inline int is_native() { return (state[3] == Native[7]); }
176:                inline int is_yield() { return (state[3] == Yield[8]); }
177:                inline int is_superceded() { return (state[3] == Superceded[9]); }
178:
179:                inline VirtualMachine[10] * get_jvm() const { return vm[11]; }
180:
181:                inline Priority[12] get_priority() const { return priority[13]; }
182:                inline Stack[14] * get_stack() const { return stack[15]; }
183:
184:                unsigned short debug;
185:                void debug_print_instruction[16](u1[17] opcode, ostream & out = cout);
186:
187:        };
188:
189:        #endif // THREAD_H
```

Footnotes:

1: *defs.h:195*
2: *ObjectData.h:113*
3: *Thread.h:45*
4: *Thread.h:24*
5: *Thread.h:25*
6: *Thread.h:26*
7: *Thread.h:27*
8: *Thread.h:28*
9: *Thread.h:29*
10: *VirtualMachine.h:255*
11: *Thread.h:32*
12: *defs.h:396*
13: *Thread.h:37*
14: *Stack.h:264*
15: *Thread.h:42*
16: *Thread.cpp:1085*
17: *defs.h:168*

```
1:          /*
2:
3:          Changed classes:
4:
5:            java.lang.Class
6:            java.lang.Thread
7:            java.io.OutputStreamWriter (CharToByteConverter)
8:            java.io.BufferedWriter (lineSeparator)
9:
10:
11:         */
12:
13:         /*
14:                 Design goals
15:                 ------------
16:
17:           - First and foremost design goal of this JVM is the clarity and readability of the code,
18:             so that it can be easily understood.
19:
20:           - Another design goal is to take full control upon the execution and the memory management
21:             of the running application.
22:                 As long as possible, native threads, native pointers and system memory allocations
23:                 were avoided enabilng easy control over all of these entities.
24:
25:                 Design remarks
26:                 --------------
27:
28:           - To gain full control upon the execution the native threads are not used.
29:             Instead, the round-robin mechanism is implemented internally.
30:                 Only native methods are executed within native threads.
31:
32:           - Because of the internal multithreading implementation there is very clear
33:             and straightforward way to implement monitors and object locks.
34:                 The monitor is implemented in such a way that it will put the demanding thread asleep
35:                 (in case it is already acquired) by itself without even notifying the thread about this.
36:                 Symetrically, when the thread will gain the monitor it will be put as running by the
37:                 monitor itself. Thus, for a thread it is absolutely transparent whether it has succeeded
38:                 to acquire a monitor or not. For it, the monitor is always acquired and the next instruction
39:                 is ready to be executed.
40:
41:           - Lazy class resolution is used, so the class is not resolved during the linkage
42:             stage, but rather its entries are resolved upon demand. It makes the loading faster.
43:
44:           - References to instances are not represented by native pointers but rather
45:             mapped from the abstract word-wide handlers to the native pointers;
46:                 thus, reference does not depend on the underlying architecture.
47:
48:           - Each constant pool entry has a "resolved" flag indicating whether this entry is resolved
49:                 or not. If in the process of resolution of one entry some other entry turns to be not
50:                 resolved the resolution process will recursively resolve it first.
51:                 Each constant pool entry contains the direct reference to its resolved runtime counterpart;
52:                 for example CONSTANT_Class_info entry has a link to a ClassFile corresponding to this entry;
53:                 CONSTANT_Methodref_info has a link to either the method_info of the corresponding method
```

```
54:                 or the offset of the virtual method in the virtual method's table;
55:                 CONSTANT_Fieldref_info has a reference to the field_info of the corresponding class;
56:
57:           - String literals are "interned" by the JVM, i.e. the unique string is stored only once
58:             within the JVM. To create such an "interned" string special technique is used:
59:                 we have to artificially initialize this String's character array without
60:                 calling any constructors. It is done by proper initialization of the java.lang.String
61:                 fields (such as "value" and "count").
62:
63:           - There is a special heap for stack allocations used by all threads.
64:             Each time a thread is created it allocates its own chunk for the contiguous stack.
65:
66:           - Stack frames are created as overlapped "operand stack"-"local variables storage"
67:             memory block. It means that the operand stack of the stack frame of the calling method
68:                 automatically becomes the local variable storage of the stack frame of the method
69:                 being called.
70:                 This approach saves memory space and also saves time, because the execution
71:                 engine does not have to spend time copying the parameter values from one frame to another.
72:
73:           - Virtual methods are stored in the class' method tables consisting of the offsets of
74:             of these methods inside the class' layout. The important point here is that such an
75:                 offset is always an invariant along the class hierarchy.
76:             Class' method table is built during the type preparation step.
77:
78:           - Native method execution mechanism tries to imitate the regular Java method execution
79:             as close as possible. Thus, the new overlapped stack frame is put on top of the Java stack
80:                 for the native method and the parameters are passed in the stack frame local variables as
81:                 for the regular Java method. Then, these parameters are pushed onto the native C-stack
82:                 and the exeution goes to the native code. After it is finished, the return value is taken
83:                 from the eax register and pushed onto the Java stack frame as if it was the regular Java
84:                 method. The native's method Java-stack frame is discarded from the Java stack as usual.
85:
86:           - Native methods are executed within the native OS threads (otherwise their bodies
87:             would be executed as an atomic instruction which violate fairness).
88:                 While executing the native code the thread is marked as Native and not Running so that
89:                 it cannot be chosen be the execution pool as a next candidate to execute an instruction.
90:                 Upon return the thread is marked Running again.
91:
92:           - Static initializers (<clinit> methods) are treated specially. Their bodies must be
93:             executed within the body of the instruction that caused the resolution process of a new
94:                 type which, in turn, caused the static initializer of this type to be called.
95:                 Thus, the instruction cannot be completed unless <clinit> is fully executed.
96:                 That's why the <clinit> method runs inside the "fake" thread with its own "tiny
97:                 execution engine" which will run uninterruptable while the <clinit> is fully done.
98:                 Then, the original instruction can continue.
99:
100:          - Special technique is used to "boot" the virtual machine. We want the call of
101:            the static "main" method to be transparent for the JVM, i.e. the "main" method
102:                should be called just as any other static method. During the method invokation the
103:                main class of the execution will be resolved just as any other class inside the JVM.
104:                For this purpose special "bootable" class file is artificially constructed at the beginning
105:                of the process. This class contains only one method which will push the reference to the
106:                command line parameters "String[] args" object onto the stack and call the "main" method
```

```
107:                of the main class using the regular "invokestatic" instruction.
108:                The last instruction of the JVM will be the "return" instruction of this "boot" method.
109:                The "boot" method will also call the static "initializeSystemClass" method that will
110:                initialize java.lang.System class (system properites, version and I/O streams).
111:
112:            - Garbage collector uses the mark-and-sweep algorithm to collect unreachable objects.
113:              There is the performance trade-off:
114:                to efficiently reach all the roots (i.e. the object references that are stored on all
115:                thread's stack frames operand stacks and local variables) we maintain the table of all
116:                such references. Thus, there is no need to traverse all the threads and all their stack
117:                frames' operand stacks and local variables (that would be very time consuming).
118:                However, to maintain such a table every push_reference and store_reference commands
119:                over a stack frame must also store this reference in the global roots table and its
120:                own local roots set.
121:                Besides, when the frame is popped of the stack its own roots set is subtracted from
122:                the global roots set.
123:              The "mark" phase of the garbage collector will visit all the instance and class roots
124:              and mark them with the current GC run counter:
125:              1) Instance roots are all the references that are stored on all thread's stack frames -
126:                  both in local variable storage and operand stack;
127:              For each root all the references pointed by its fields are marked recursively;
128:              2) Class roots are the static class data stored in the class files that, in turn, are
129:              reachable from the namespaces of all the JVM class loaders.
130:                  The "sweep" phase is working with instance and class heaps deallocating all the memory
131:                  chunks whose mark counter is less than the current GC run counter.
132:
133:            - Garbage collector is designed to be undistiguishable from a regular thread of execution
134:              in terms of the execution engine. That's why it can be picked up as an execution candidate
135:                  by the engine as well as any other JVM thread.
136:                  However, most of the time GC will run with the lowest priority, and only when an
137:                  memory emergency event occur its priority boost to the SYS_PRIORITY.
138:
139:            - Yielding mechanism is introduced to make the multithreading more fair.
140:                  Oppositely to the Java Thread's yield() function whose functionality is mainly undefined
141:                  in the JVM implementations, our internal yield call is compulsory for the JVM and is
142:                  fairly executed by all threads performing any lengthy process.
143:                  The most lenghty process (of unpredictable duration) is the resolution process that can
144:                  cause a chain reaction in resolution of classes, methods and fields.
145:                  That's why each time the resolution is to be performed by a thread, it calls the
146:                  "yield" function wich will choose other threads to execute.
147:                  The problem here is that the current thread is in the middle of its current instruction,
148:                  so when the execution engine picks up this "yielding" thread it simply returns to the
149:                  interrupted instruction without performing any "step".
150:                  Note, that since the resolution is a recursive process, the yielding can be performed
151:                  many times during one instruction execution (at the different stages of resolution).
152:
153:              Main entities of the JVM
154:                  ------------------------
155:
156:            - VirtualMachine:
157:              this is the JVM itself. It holds the pointers to all the other entities and serves
158:                  as the mediator between all of them.
159:                  VirtualMachine performs the start-up operation and the main execution loop.
```

```
160:                VM holds a set of all class loaders - bootstrap and user-defined.
161:
162:            - ExecutionPool:
163:              holds pointers to all the current JVM threads, assigns thread identificators,
164:                  manage priority queue of thread priorities and is responsible to choose the next
165:                  thread eligible to run.
166:
167:            - Thread:
168:              the abstraction of the execution thread. Contains its own Java stack and defines
169:                  the function "step" which performes one step of execution, usually one instruction.
170:
171:            - GarbageCollector:
172:              this is the very special case of thread responsible for cleaning up the Java heap.
173:                  It also defines the "step" function in such a way that it can be picked up by the
174:                  execution pool as well as any other thread.
175:                  Garbage collector's step consists of two phases: "mark" and "sweep".
176:                  Most of the time GC is running with minimal priority. The priority boosts only
177:                  in cases of "emergency".
178:
179:            - HandlePool:
180:              Maps abstract object references to native pointers (namely to the InstanceData pointers)
181:                  that point to the real memory allocated by an object.
182:                  HandlePool also maintain the table of "interned" String literals and the table of "roots"
183:                  which is constantly updated by all the stack frame operations related to object references.
184:
185:            - ClassFile:
186:              This is the main repository of the internal Java class information.
187:                  It contains class' constant pool, method area, virtual method table, pointer to the
188:                  static class data and to the java.lang.Class object associated with each type.
189:
190:            - ConstantPool:
191:              Represents the Java class file constant pool together with the runtime constant pool.
192:                  Contains the exact image of the constant pool being read from the Java classfile.
193:                  Upon resolution all its entries eventually will point to the ClassFiles, methods
194:                  and fields of the corresponding classes.
195:
196:            - ClassLoader:
197:
198:            - HeapManager:
199:
200:            - NativeHandler:
201:
202:            - InstanceData and ClassData:
203:
204:            - Execution Engine and Interpreter:
205:                  Execution engine itself is nothing more than the endless loop that consults the
206:                  ExecutionPool on each step about the next thread that is suitable to run.
207:                  Then, the thread's "step" function is called, which consists of three simple steps:
208:                  - fetch the next opcode;
209:                  - jump to the instruction body that will interpret and execute this command;
210:                  - check the results and continue;
211:                  The Interpreter is the set of functions that implement all instructions specified
212:                  in the JVM specification. Each such function receives a pointer to the thread
```

```
213:                executing it as the only parameter. From this thread pointer, the function has
214:                access to the thread's stack, current stack frame, current method code, current class,
215:                current constant pool and the VM itself.
216:
217:              Source code notes
218:              -----------------
219:
220:              - Striving to keep the code as simple to understand as possible we make no use
221:                of parameterized macros.
222:              - No C++ operators are used (except of one obvious case of UTF8 comparison operator).
223:              - Usage of templates is minimized.
224:
225:            */
226:
227:            /*
228:             * Main class for the Java Virtual Machine.
229:             * Singleton, of course.
230:             * It has pointers to all other entities serving as a mediator between them.
231:             * All other entities has a pointer to VM to be able to access any one of them.
232:             */
233:
234:            #ifndef VIRTUAL_MACHINE_H[1]
235:            #define VIRTUAL_MACHINE_H
236:
237:            #include "defs.h"
238:            #include "vector.h"
239:
240:            #include "jni.h" // Java Native Interface standard header
241:
242:            class GarbageCollector[2];
243:            class HeapManager[3];
244:            class ConstantManager[4];
245:            class ClassLoader[5];
246:            class Thread[6];
247:            class ExecutionPool[7];
248:            class HandlePool[8];
249:            class NativeHandler[9];
250:            class InstanceData[10];
251:            struct Code_attribute[11];
252:
253:            // Virtual Machine is a subclass of the JavaVM_ class defined in the standard JNI header file.
254:            // This is necessary to use this VirtualMachine in the implementations of the native methods
255:            class VirtualMachine : public JavaVM_
256:            {
257:            public:
258:                    ConstantManager[4] * p_ConstantManager;
259:
260:                    // Meanwhile the same heap serves for both class and instance data storage;
261:                    // however, it can be easily changes, since ClassFile use different heaps
262:                    // without knowing it is the same heap
263:                    HeapManager[3] * heap_manager;
264:
265:                    // This heap is used to allocate memory blocks for all the running threads' stacks;
```

Footnotes:

1: *VirtualMachine.h:235*
2: *GarbageCollector.h:19*
3: *HeapManager.h:81*
4: *VirtualMachine.h:398*
5: *ClassLoader.h:14*
6: *Thread.h:17*
7: *VirtualMachine.h:355*
8: *HandlePool.h:13*
9: *NativeHandler.h:13*
10: *ObjectData.h:113*
11: *AttributeInfo.h:55*

```
266:                // Once the thread is run, the new stack is allocated
267:                HeapManager[1] * stack_heap_manager;
268:
269:                // Deals with the native method registration and invokation
270:                NativeHandler[2] * native_handler;
271:
272:                // Garbage collector is the very special thread;
273:                // logically, it shares only the "step" function with the "proper" thread.
274:                // But this makes him the candidate to run that can be picked up by the
275:                // execution engine.
276:                // Most of the time it will run with the lowest priority, but in case of
277:                // emergency its priority will boost to the SYS_PRIORITY
278:                GarbageCollector[3] * garbage_collector_thread;
279:
280:                // The only true JVM bootstrap class loader
281:                ClassLoader[4] * bootstrap_class_loader;
282:
283:                // Points to the primary thread of execution
284:                Thread[5] * primary_thread;
285:
286:                // At any moment points to the thread currently performing instruction
287:                Thread[5] * current_thread;
288:
289:                // Responsible for the round-robin mechanism of multithreaded execution;
290:                // will always choose the next thread suitable to run
291:                ExecutionPool[6] * execution_pool;
292:
293:                // Handles all object instances created so far, all "interned" strings
294:                // and the global roots (reachable objects)
295:                HandlePool[7] * handle_pool;
296:
297:                // The repository of all the class loaders - bootstrap and user-defined
298:                Vector[8]<ClassLoader*> * class_loaders;
299:
300:                // true when the JVM is in the process of creating the bootable class;
301:                // false all the rest of the time
302:                int boot_process;
303:
304:        public:
305:                VirtualMachine[9]();
306:                ~VirtualMachine[10]();
307:
308:                // Initialize instance data, class data and thread stack heaps
309:                Result[11] init[12]();
310:                Result[11] shut_down[13]();
311:
312:                Result[11] start[14](char * filepath);
313:
314:                // This is the main function of the JVM - the execution loop
315:                Result[11] run[15]();
316:
317:                // Current thread has decided to behave nicely and passes the execution to other threads.
318:                // The function will imitate the main execution loop but will return immediately as far
```

Footnotes:

1: *HeapManager.h:81*
2: *NativeHandler.h:13*
3: *GarbageCollector.h:19*
4: *ClassLoader.h:14*
5: *Thread.h:17*
6: *VirtualMachine.h:355*
7: *HandlePool.h:13*
8: *vector.h:7*
9: *VirtualMachine.cpp:21*
10: *VirtualMachine.cpp:56*
11: *defs.h:29*
12: *VirtualMachine.cpp:70*
13: *VirtualMachine.cpp:98*
14: *VirtualMachine.cpp:112*
15: *VirtualMachine.cpp:258*

```
319:                // as the yielding thread gains execution. Then the postponed "step" will be continued
320:                void yield[1](Thread[2] * thread);
321:
322:                void supercede[3](Thread[2] * thread, Thread[2] * by_thread);
323:
324:                void abnormal_termination[4](InstanceData[5] * exception_instance = NULL);
325:
326:                // Getters
327:
328:                ConstantManager[6] * getConstantManager() const { return p_ConstantManager[7]; }
329:                inline HeapManager[8] * getHeapManager() const { return heap_manager[9]; }
330:                inline GarbageCollector[10] * get_garbage_collector() const { return garbage_collector_thread[11]; }
331:                inline HeapManager[8] * get_class_heap_manager() const { return heap_manager[9]; } // same heap
332:                inline HeapManager[8] * get_instance_heap_manager() const { return heap_manager[9]; }// same heap
333:                inline HeapManager[8] * get_stack_heap_manager() const { return stack_heap_manager[12]; }
334:                inline ClassLoader[13] * get_bootstrap_class_loader() const { return bootstrap_class_loader[14]; }
335:                inline Thread[2] * get_primary_thread() const { return primary_thread[15]; }
336:                inline HandlePool[16] * get_handle_pool() const { return handle_pool[17]; }
337:                inline NativeHandler[18] * get_native_handler() const { return native_handler[19]; }
338:                inline Vector[20]<ClassLoader*> * get_class_loaders() const { return class_loaders[21]; }
339:
340:                // This function is called by the registerNatives function of all the
341:                // native class implementations
342:                void register_method[22](char * name, char * descriptor, void * func_ptr);
343:
344:                // Adds a new, ready to run thread to the VM with the given priority;
345:                // returns the newely created thread
346:                Thread[2] * run_thread[23](Code_attribute[24] * code_attribute,
347:                                           InstanceData[5] * thread_instance,
348:                                    Priority[25] priority = NORM_PRIORITY);
349:        };
350:
351:
352:        // This pool stores all threads existing in the JVM at any moment
353:        // It is responsible for retrieving the next thread suitable for execution
354:        // using the round-robin algorithms
355:        class ExecutionPool
356:        {
357:                friend class GarbageCollector[10];
358:
359:        private:
360:                VirtualMachine[26] * vm;
361:                unsigned long id_counter;
362:
363:                // Threads storage used as a priority queue:
364:                // Vector at array element 0 contains threads with MIN_PRIORITY
365:                // Vector at array element 1 contains threads with NORM_PRIORITY
366:                // Vector at array element 2 contains threads with MAX_PRIORITY
367:                // Vector at array element 3 contains threads with SYS_PRIORITY
368:                Vector[20]<Thread*> * threads[PRIORITY_ARRAY_LENGTH[27]];
369:
370:                // This is the current ring of priority level being executed
371:                int current_priority;
```

Footnotes:

1: *VirtualMachine.cpp:213*
2: *Thread.h:17*
3: *VirtualMachine.cpp:159*
4: *VirtualMachine.cpp:143*
5: *ObjectData.h:113*
6: *VirtualMachine.h:398*
7: *VirtualMachine.h:258*
8: *HeapManager.h:81*
9: *VirtualMachine.h:263*
10: *GarbageCollector.h:19*
11: *VirtualMachine.h:278*
12: *VirtualMachine.h:267*
13: *ClassLoader.h:14*
14: *VirtualMachine.h:281*
15: *VirtualMachine.h:284*
16: *HandlePool.h:13*
17: *VirtualMachine.h:295*
18: *NativeHandler.h:13*
19: *VirtualMachine.h:270*
20: *vector.h:7*
21: *VirtualMachine.h:298*
22: *VirtualMachine.cpp:309*
23: *VirtualMachine.cpp:287*
24: *AttributeInfo.h:55*
25: *defs.h:396*
26: *VirtualMachine.h:255*
27: *defs.h:402*

```
372:                    // This is the array of current thread's index in all priority levels
373:                    int current_index[4];
374:
375:                    int threads_number[1]();
376:
377:            public:
378:                    ExecutionPool[2](VirtualMachine[3] * _vm);
379:                    ~ExecutionPool[4]();
380:
381:                    void add_thread[5](Thread[6] * thread, Priority[7] priority = NORM_PRIORITY);
382:
383:                    // This is the main part of the multithreading;
384:                    // the function will reschedule the threads using the round-robin mechanism.
385:                    // It will return the next thread suitable for execution or NULL in case there are
386:                    // no running threads
387:                    Thread[6] * resched[8]();
388:
389:                    inline unsigned long get_next_thread_id() { return id_counter[9]++; }
390:
391:                    void change_priority[10](Thread[6] * thread, Priority[7] new_priority);
392:            };
393:
394:            /*
395:             * Class to deal with all the boring stuff like stack and heap initial size, etc.
396:             *
397:             */
398:            class ConstantManager
399:            {
400:            private:
401:                    VirtualMachine[3] * p_VirtualMachine;
402:            public:
403:                    ConstantManager(VirtualMachine[3] * vm) : p_VirtualMachine[11](vm) {}
404:                    ~ConstantManager() {}
405:
406:                    Result[12] init(/*parameters*/) { return Success[13]; }
407:                    Result[12] destroy() { return Success[13]; }
408:            };
409:
410:
411:
412:            #endif // VIRTUAL_MACHINE_H
```

Footnotes:

1: *VirtualMachine.cpp:352*
2: *VirtualMachine.cpp:319*
3: *VirtualMachine.h:255*
4: *VirtualMachine.cpp:334*
5: *VirtualMachine.cpp:344*
6: *Thread.h:17*
7: *defs.h:396*
8: *VirtualMachine.cpp:382*
9: *VirtualMachine.h:361*
10: *VirtualMachine.cpp:360*
11: *VirtualMachine.h:401*
12: *defs.h:29*
13: *defs.h:33*

```
 1:          /*
 2:           * Contains all global definitions.
 3:           *
 4:           */
 5:
 6:          #ifndef DEFS_H[1]
 7:          #define DEFS_H
 8:
 9:          #include <wchar.h>
10:          #include <fstream.h>
11:
12:          #include "vector.h"
13:
14:          #define JVM_FOR_WIN32   // to compile under Windows
15:          #define JVM_FOR_INTEL   // to convert values for little-endian architecture
16:
17:          //#define JVM_FOR_LINUX // to compile under Linux/Unix
18:
19:
20:          /*
21:          * Results of all the possible operations.
22:          * Error messages are printed in the following manner:
23:          * first, the most specific error is printed by the function issuing the error,
24:          * then the function returns the more general error result to the calling function.
25:          * That function in turn prints this result and return yet more general error result,
26:          * and so on.
27:          */
28:
29:          enum Result {
30:
31:                  // Internal messages
32:
33:                  Success,
34:                  Failure,
35:                  HeapInitializationError,
36:                  InitializationCannotAllocateHeap,
37:                  InitializationCannotAllocateStack,
38:                  BootableClassCannotCreate,
39:                  GarbageCollectorInitializationError,
40:                  Utf8Error,
41:                  MainClassFileReadError,
42:                  MainClassCannotExecute,
43:                  ClassLoaderCannotLoadClass,
44:                  ClassLoaderErrorWrongMagicNumber,
45:                  ClassLoaderErrorWrongConstantPoolTag,
46:                  ClassLoaderCannotReadFieldMethodInfo,
47:                  ClassLoaderErrorUnknownAttribute,
48:                  ClassLoaderCannotLoadSuperclass,
49:                  ClassLoaderCannotLoadSuperinterfaces,
50:                  ClassLoaderCannotPrepareClass,
51:                  ClassLoaderCannotDefinePrimitiveArrayType,
52:                  ClassFileNotFoundError,
53:                  ClassFileReadError,
```

Footnotes:

1: *defs.h:7*

```
54:                 HeapErrorCannotAllocateInitialSize,
55:                 HeapErrorCannotAllocateMemory,
56:                 HeapErrorCannotFindMemoryChunkToDeallocate,
57:                 ResolutionCannotResolveClassName,
58:                 ResolutionCannotLoadClassFromClassName,
59:                 ResolutionCannotResolveMethodref,
60:                 ResolutionCannotResolveInterfaceMethodref,
61:                 ResolutionCannotResolveFieldref,
62:                 ResolutionCannotFindCodeAttributeOfMethod,
63:                 ResolutionMethodLookupFailed,
64:                 ResolutionCannotInternStringObject,
65:                 PreparationNoFieldDescriptorFound,
66:                 PreparationUnrecognizedBasicType,
67:                 PreparationCannotAllocateClassData,
68:                 PreparationCannotBuildMethodTable,
69:                 InstanceCreationNoFieldDescriptorFound,
70:                 InstanceCreationCannotAllocateInstanceData,
71:                 InstanceCreationUnrecognizedBasicType,
72:                 InstanceCreationFailure,
73:                 ExecutionCannotFindMainMethod,
74:                 ExecutionCannotCreateNewInstance,
75:                 ExecutionCannotExecuteInstruction,
76:                 ExecutionNativeMethodNotFound,
77:                 ExecutionCannotExecuteNativeMethod,
78:                 ExecutionCannotExecuteStaticInitializer,
79:                 ExecutionCannotLoadExceptionClass,
80:                 ExecutionWrongConstantPoolEntryIndex,
81:                 ExecutionCannotResolveExceptionClass,
82:                 VirtualMachineCannotRunMainMethod,
83:                 ThreadCannotInvokeInstanceMethod,
84:                 ThreadCannotInvokeStaticMethod,
85:                 ThreadCannotInvokeSpecialMethod,
86:                 ThreadCannotInvokeInterfaceMethod,
87:                 MethodDescriptorWrongFormat,
88:
89:                 // Virtual Machine Errors
90:
91:                 VM_ERROR_START, // to distinguish JVM errors from internal errors (used only by if-statements)
92:
93:                 VM_ERROR_NoClassDefFoundError,
94:                 VM_ERROR_ClassNotFoundError,
95:                 VM_ERROR_ClassFormatError,
96:                 VM_ERROR_UnsupportedClassVersionError,
97:                 VM_ERROR_LinkageError,
98:                 VM_ERROR_ClassCircularityError,
99:                 VM_ERROR_IllegalAccessError,
100:                VM_ERROR_VerifyError,
101:                VM_ERROR_IncompatibleClassChangeError,
102:                VM_ERROR_NoSuchMethodError,
103:                VM_ERROR_NoSuchFieldError,
104:                VM_ERROR_AbstractMethodError,
105:                VM_ERROR_NullPointerException,
106:                VM_ERROR_IllegalMonitorStateException,
```

```
107:                VM_ERROR_NegativeArraySizeException,
108:                VM_ERROR_ArrayIndexOutOfBoundsException,
109:                VM_ERROR_ArrayStoreExceptionClass,
110:                VM_ERROR_ArrayStoreException,
111:                VM_ERROR_ClassCastException,
112:                VM_ERROR_ArithmeticException,
113:                VM_ERROR_Throwable,
114:                VM_ERROR_Exception,
115:
116:                VM_ERROR_END
117:        };
118:
119:        // All types of possible instruction results
120:        enum InstructionResult
121:        {
122:                InstructionSuccess,
123:                InstructionFailure,
124:                InstructionErrorDivisionByZero,
125:                InstructionErrorWrongConstantPoolEntryIndex,
126:                InstructionErrorWrongConstantPoolEntryType,
127:                InstructionErrorBranchOutsideMethodCode,
128:                InstructionErrorCannotCreateInstance,
129:                InstructionErrorCannotCreateArray,
130:                InstructionErrorCannotInvokeInterface,
131:                InstructionErrorCannotInvokeSpecial,
132:                InstructionErrorCannotInvokeStatic,
133:                InstructionErrorCannotInvokeVirtual,
134:                InstructionErrorCannotGetStaticField,
135:                InstructionErrorCannotPutStaticField,
136:                InstructionErrorCannotGetField,
137:                InstructionErrorCannotPutField,
138:                InstructionErrorNullPointerException,
139:                InstructionErrorMonitorNotOwned,
140:                InstructionErrorWrongArrayIndex,
141:                InstructionErrorWrongArrayType,
142:                InstructionErrorCannotGetInternedString,
143:                InstructionErrorAssignmentIncompatible,
144:                InstructionErrorCannotExecuteInstanceof,
145:                InstructionErrorClassCastException,
146:                InstructionErrorExceptionIsNotThrowableSubclass
147:        };
148:
149:        struct exception_class
150:        {
151:                Result[1] exception_code;
152:                wchar_t * exception_class_name;
153:        };
154:
155:        extern exception_class[2] ExceptionClasses[3][];
156:
157:        void print_error[4](Result[1]);
158:        void print_instruction_error[5](InstructionResult[6]);
159:
```

Footnotes:

1: *defs.h:29*
2: *defs.h:149*
3: *errors.cpp:7*
4: *errors.cpp:35*
5: *errors.cpp:84*
6: *defs.h:120*

```
160:        /*
161:         JVM spec data types.
162:         The bytes in the Java class file are stored in big-endian order, i.e. high-order byte first
163:         (which is different from the Intel architecture).
164:         We will compulsory store the bytes in the little-endian order using the functions
165:         memcpy_u2 and memcpy_u4 if JVM_FOR_INTEL is defined
166:        */
167:
168:        typedef unsigned char  u1; // must be 1 byte long
169:        typedef signed char    su1;
170:
171:        #ifdef JVM_FOR_WIN32[1]
172:        typedef unsigned __int16 u2; // must be 2 bytes long
173:        typedef signed   __int16 su2;
174:        typedef unsigned __int32 u4; // must be 4 bytes long
175:        typedef signed   __int32 su4;
176:        typedef unsigned __int64 u8; // must be 8 bytes long
177:        typedef signed   __int64 su8;
178:        #else
179:        typedef unsigned short u2[2]; // must be 2 bytes long
180:        typedef signed   short su2[3];
181:        typedef unsigned int   u4[4]; // must be 4 bytes long
182:        typedef signed   int   su4[5];
183:        typedef unsigned long  u8[6]; // must be 8 bytes long
184:        typedef signed   long  su8[7];
185:        #endif
186:
187:
188:        void memcpy_u2[8](u2[2] * src, const u1[9] * dst);
189:        void memcpy_u4[10](u4[4] * src, const u1[9] * dst);
190:
191:        // "word" is the basic unit of size for data values in the JVM;
192:        // it must be large enough to hold a value of type byte, short, int, char,
193:        // float, returnAddress and reference;
194:        // two words must hold long and double type variables
195:        typedef u4[4] word;
196:
197:        // represents the null reference in the JVM
198:        // relies on the fact the the HandlePool's counter of reference representations
199:        // is started with 1
200:        //#define null 0
201:        const word[11] null = 0;
202:
203:        #define MAX_FILE_PATH_NAME 256
204:        #define MAX_JNI_METHOD_NAME 2048
205:
206:        // Represents all possible primitive types of JVM.
207:        // These constants will also serve as indexes in the BasicTypes array
208:        enum BasicType
209:        {
210:                Byte      = 0,
211:                Char      = 1,
212:                Double    = 2,
```

Footnotes:

1: *defs.h:14*
2: *defs.h:172*
3: *defs.h:173*
4: *defs.h:174*
5: *defs.h:175*
6: *defs.h:176*
7: *defs.h:177*
8: *memcpy_bigendian.cpp:10*
9: *defs.h:168*
10: *memcpy_bigendian.cpp:27*
11: *defs.h:195*

```
213:                    Float     = 3,
214:                    Int       = 4,
215:                    Long      = 5,
216:                    Reference = 6,
217:                    Short     = 7,
218:                    Boolean   = 8,
219:                    Array     = 9,
220:                    Void      = 10,
221:                    UnrecognizedType
222:            };
223:
224:            struct basic_type
225:            {
226:                    BasicType[1] type;
227:                    unsigned short size; // in bytes
228:                    void * default_value;
229:            };
230:
231:            struct general_type
232:            {
233:                    enum Kind { Array, Reference, Basic, None } kind;
234:                    Kind[2] array_kind;
235:                    char qualified_name[MAX_FILE_PATH_NAME[3]];
236:                    unsigned short dimension;
237:                    BasicType[1] type;
238:
239:                    general_type() : dimension[4](0), kind[5](None[6]) {} // must set dimension to 0 and kind to None
240:                                                                          // for deciphering functions
241:                    void debug_print[7]();
242:            };
243:
244:            // This array contains primitive types (indexed by BasicType enum),
245:            // sizes of these types in bytes and their default values
246:            basic_type[8] BasicTypes[];
247:
248:            // Some "hardcoded" stuff
249:
250:            #define MAGIC_NUMBER 0xCAFEBABE // identifying the true and the only class file format
251:
252:            #define OBJECT_CLASS_NAME                       L"java/lang/Object"
253:            #define CLASS_CLASS_NAME            L"java/lang/Class"
254:            #define STRING_CLASS_NAME           L"java/lang/String"
255:            #define THROWABLE_CLASS_NAME        L"java/lang/Throwable"
256:            #define CLINIT_METHOD_NAME                      L"<clinit>"
257:            #define CLINIT_METHOD_DESCRIPTOR        L"()V"
258:            #define INIT_METHOD_NAME                        L"<init>"
259:            #define INIT_METHOD_DESCRIPTOR      L"()V"
260:            #define MAIN_METHOD_NAME                        L"main"
261:            #define MAIN_METHOD_DESCRIPTOR          L"([Ljava/lang/String;)V"
262:            #define THREAD_GROUP_CLASS_NAME     L"java/lang/ThreadGroup"
263:            #define RUN_METHOD_NAME             L"run"
264:            #define RUN_METHOD_DESCRIPTOR       L"()V"
265:            // The following six macros are needed to construct the "interned" String object
```

Footnotes:

1: *defs.h:208*
2: *defs.h:233*
3: *defs.h:203*
4: *defs.h:236*
5: *defs.h:233*
6: *defs.h:233*
7: *util.cpp:49*
8: *defs.h:224*

```
266:        // without calling the Java constructor
267:        #define STRING_CLASS_VALUE_NAME        L"value"
268:        #define STRING_CLASS_VALUE_DESCRIPTOR  L"[C"
269:        #define STRING_CLASS_OFFSET_NAME       L"offset"
270:        #define STRING_CLASS_OFFSET_DESCRIPTOR L"I"
271:        #define STRING_CLASS_COUNT_NAME        L"count"
272:        #define STRING_CLASS_COUNT_DESCRIPTOR  L"I"
273:        // The following macros needed to assign the ThreadGroup field of the bootstrap class
274:        #define THREAD_GROUP_GROUP_NAME        L"group"
275:        #define THREAD_GROUP_GROUP_DESCRIPTOR  L"Ljava/lang/ThreadGroup;"
276:        #define THREAD_PRIORITY_NAME           L"priority"
277:        #define THREAD_PRIORITY_DESCRIPTOR     L"I"
278:        // The following macros needed to assign I/O streams in the "initializeSystemClass" call
279:        #define SYSTEM_CLASS_NAME            L"java/lang/System"
280:        #define SYSTEM_OUT_NAME              L"out"
281:        #define SYSTEM_OUT_DESCRIPTOR        L"Ljava/io/PrintStream;"
282:        #define SYSTEM_ERR_NAME              L"err"
283:        #define SYSTEM_ERR_DESCRIPTOR        L"Ljava/io/PrintStream;"
284:        #define SYSTEM_IN_NAME               L"out"
285:        #define SYSTEM_IN_DESCRIPTOR         L"Ljava/io/InputStream;"
286:
287:        // Predefined attribure names as they appear in the Constant Pool (as defined in JVM spec)
288:
289:        #define CONSTANT_VALUE_ATTRIBUTE_NAME       L"ConstantValue"
290:        #define CODE_ATTRIBUTE_NAME                 L"Code"
291:        #define EXCEPTIONS_ATTRIBUTE_NAME           L"Exceptions"
292:        #define INNER_CLASSES_ATTRIBUTE_NAME        L"InnerClasses"
293:        #define SYNTHETIC_ATTRIBUTE_NAME            L"Synthetic"
294:        #define SOURCE_FILE_ATTRIBUTE_NAME          L"SourceFile"
295:        #define LINE_NUMBER_TABLE_ATTRIBUTE_NAME    L"LineNumberTable"
296:        #define LOCAL_VARIABLE_TABLE_ATTRIBUTE_NAME L"LocalVariableTable"
297:        #define DEPRECATED_ATTRIBUTE_NAME           L"Deprecated"
298:
299:        // Base type character internal representation (as defined in JVM spec)
300:
301:        #define BYTE_INTERNAL_REP      L"B" //  represents byte (signed byte)
302:        #define CHAR_INTERNAL_REP      L"C" //  represents char  (Unicode character)
303:        #define DOUBLE_INTERNAL_REP    L"D" //  represents double  (double-precision floating-point value)
304:        #define FLOAT_INTERNAL_REP     L"F" //  represents float  (single-precision floating-point value)
305:        #define INT_INTERNAL_REP       L"I" //  represents int  (integer)
306:        #define LONG_INTERNAL_REP      L"J" //  represents long  (long integer)
307:        #define CLASSNAME_INTERNAL_REP L"L" //  represents reference  (a prefix for an instance of class)
308:        #define SHORT_INTERNAL_REP     L"S" //  represents short  (signed short)
309:        #define BOOL_INTERNAL_REP      L"Z" //  represents boolean  (true or false)
310:        #define ARRAY_DIM_INTERNAL_REP L"[" //  represents reference (one array dimension)
311:
312:        // The fully qualified name of a primitive type is the keyword for that primitive type,
313:        // namely, boolean, char, byte, short, int, long, float, or double.
314:
315:        #define BOOLEAN_TYPE_NAME "boolean"
316:        #define CHAR_TYPE_NAME    "char"
317:        #define BYTE_TYPE_NAME    "byte"
318:        #define SHORT_TYPE_NAME   "short"
```

```
319:          #define INT_TYPE_NAME     "int"
320:          #define LONG_TYPE_NAME    "long"
321:          #define FLOAT_TYPE_NAME   "float"
322:          #define DOUBLE_TYPE_NAME  "double"
323:
324:          // Kinds of cp_info entries (as defined in JVM spec)
325:
326:          #define CONSTANT_Class              7
327:          #define CONSTANT_Fieldref           9
328:          #define CONSTANT_Methodref          10
329:          #define CONSTANT_InterfaceMethodref 11
330:          #define CONSTANT_String             8
331:          #define CONSTANT_Integer            3
332:          #define CONSTANT_Float              4
333:          #define CONSTANT_Long               5
334:          #define CONSTANT_Double             6
335:          #define CONSTANT_NameAndType        12
336:          #define CONSTANT_Utf8               1
337:
338:          // Array types (as defined in JVM spec)
339:
340:          #define T_BOOLEAN  4
341:          #define T_CHAR     5
342:          #define T_FLOAT    6
343:          #define T_DOUBLE   7
344:          #define T_BYTE     8
345:          #define T_SHORT    9
346:          #define T_INT      10
347:          #define T_LONG     11
348:
349:
350:          // Access flags for the Class file format structures (as defined in JVM spec)
351:
352:
353:          // Access flags for class
354:          #define ACC_PUBLIC       0x0001 // Declared public; may be accessed from outside its package.
355:          #define ACC_FINAL        0x0010 // Declared final; no subclasses allowed.
356:          #define ACC_SUPER        0x0020 // Treat superclass methods specially when invoked by the invokespecial i
              nstruction.
357:          #define ACC_INTERFACE    0x0200 // Is an interface, not a class.
358:          #define ACC_ABSTRACT     0x0400 // Declared abstract; may not be instantiated.
359:
360:          // Access flags for field
361:          #define ACC_PUBLIC[1]       0x0001 // Declared public; may be accessed from outside its package.
362:          #define ACC_PRIVATE      0x0002 // Declared private; usable only within the defining class.
363:          #define ACC_PROTECTED    0x0004 // Declared protected; may be accessed within subclasses.
364:          #define ACC_STATIC       0x0008 // Declared static.
365:          #define ACC_FINAL[2]        0x0010 // Declared final; no further assignment after initialization.
366:          #define ACC_VOLATILE     0x0040 // Declared volatile; cannot be cached.
367:          #define ACC_TRANSIENT    0x0080 // Declared transient; not written or read by a persistent object manager
              .
368:
369:          // Access flags for method
```

Footnotes:

1: *defs.h:354*
2: *defs.h:355*

```
370:        #define ACC_PUBLIC[1]       0x0001 // Declared public; may be accessed from outside its package.
371:        #define ACC_PRIVATE[2]      0x0002 // Declared private; accessible only within the defining class.
372:        #define ACC_PROTECTED[3]    0x0004 // Declared protected; may be accessed within subclasses.
373:        #define ACC_STATIC[4]       0x0008 // Declared static.
374:        #define ACC_FINAL[5]        0x0010 // Declared final; may not be overridden.
375:        #define ACC_SYNCHRONIZED 0x0020 // Declared synchronized; invocation is wrapped in a monitor lock.
376:        #define ACC_NATIVE       0x0100 // Declared native; implemented in a language other than Java.
377:        #define ACC_ABSTRACT[6]     0x0400 // Declared abstract; no implementation is provided.
378:        #define ACC_STRICT       0x0800 // Declared strictfp; floating-point mode is FP-strict
379:
380:        // Access flags for inner class
381:        #define ACC_PUBLIC[1]       0x0001 // Marked or implicitly public in source.
382:        #define ACC_PRIVATE[2]      0x0002 // Marked private in source.
383:        #define ACC_PROTECTED[3]    0x0004 // Marked protected in source.
384:        #define ACC_STATIC[4]       0x0008 // Marked or implicitly static in source.
385:        #define ACC_FINAL[5]        0x0010 // Marked final in source.
386:        #define ACC_INTERFACE[7]    0x0200 // Was an interface in source.
387:        #define ACC_ABSTRACT[6]     0x0400 // Marked or implicitly abstract in source.
388:
389:
390:        #define POSITIVE_INFINITY 0 // TODO
391:        #define NEGATIVE_INFINITY 0 // TOTO
392:        #define NaN 0 // TODO
393:
394:        // Thread priorities
395:
396:        enum Priority
397:        {
398:                MIN_PRIORITY  = 0,
399:                NORM_PRIORITY = 1,
400:                MAX_PRIORITY  = 2,
401:                SYS_PRIORITY  = 3, // not defined in JVM spec but used internally for system own threads
402:            PRIORITY_ARRAY_LENGTH
403:        };
404:
405:        // Utility functions
406:
407:        void wchar2ascii[8](wchar_t * wcharstr, char * charstr, unsigned int length);
408:        void ascii2wchar[9](char * charstr, wchar_t * wcharstr);
409:        void slash2backslash[10](char * str);
410:        void slash2underscore[11](char * str);
411:        void dot2slash[12](char * str);
412:        void print_wchar[13](wchar_t * str, unsigned int length, ostream & out = cout);
413:        void swap_bytes8[14](u8[15] * value);
414:        void swap_bytes4[16](u4[17] * value);
415:
416:        Result[18] decipher_method_descriptor[19](char * string, Vector[20]<general_type*> & args, general_type[21] * return_
           type);
417:
418:        void debug_print_code[22](u1[23] * code, unsigned long code_length, ostream & out = cout);
419:
420:        extern ofstream debug_file;
421:
```

Footnotes:

1: *defs.h:354*
2: *defs.h:362*
3: *defs.h:363*
4: *defs.h:364*
5: *defs.h:355*
6: *defs.h:358*
7: *defs.h:357*
8: *util.cpp:8*
9: *util.cpp:16*
10: *util.cpp:24*
11: *util.cpp:30*
12: *util.cpp:36*
13: *util.cpp:42*
14: *memcpy_bigendian.cpp:51*
15: *defs.h:176*
16: *memcpy_bigendian.cpp:42*
17: *defs.h:174*
18: *defs.h:29*
19: *decipher.cpp:113*
20: *vector.h:7*
21: *defs.h:231*
22: *opcodes.cpp:3424*
23: *defs.h:168*

```
422:          #define CLASSPATH "e:\\personal\\myjvm\\rt\\"
423:
424:          // Various debug macros
425:
426:          // define this for "heavy" debugging: every instruction will be printed
427:          // together with the thread id
428:          #define DEBUG_EXECUTION
429:
430:          // more "heavy" debugging: every field, method and new instance is printed
431:          // (works only when DEBUG_EXECUTION is on)
432:          //#define DEBUG_INSTRUCTIONS
433:
434:          // class loader will dump the classes immediately after loading
435:          //#define DEBUG_CLASS_LOADING
436:
437:          // created instances will be displayed in order of creation
438:          // together with their class names and references
439:          //#define DEBUG_INSTANCE_CREATION
440:
441:          // stack snapshots will be printed for each instruction (extremely heavy !)
442:          //#define DEBUG_STACK_SNAPSHOT
443:
444:          // debug Garbage Collector: will print the pass number, the number of deallocated objects
445:          // and the heap state
446:          #define DEBUG_GC
447:
448:          // will print the processing of monitor acquiring and release, wait and notify calls
449:          #define DEBUG_SYNCHRONIZATION
450:
451:          // Constants
452:
453:          #define MAX_HEAP_SIZE  1048576 // 1Mb
454:          #define MAX_STACK_SIZE 10240 // 10Kb (~100 threads with 10Kb stack within 1Mb total stack space)
455:
456:          #endif // DEFS_H
457:
```

```
 1:          #ifndef DLIST_H[1]
 2:          #define DLIST_H
 3:
 4:          #include <stdio.h>
 5:          #include <assert.h>
 6:
 7:          struct node
 8:          {
 9:                  node[2] * next;
10:                  node[2] * prev;
11:                  void * info;
12:                  node(void * _info) : next[3](NULL), prev[4](NULL), info[5](_info)
13:                  {
14:                  }
15:          };
16:
17:          // Double-linked list
18:
19:          class DList
20:          {
21:          private:
22:                  node[2] * first; // points to the very first node
23:                  node[2] * last; // always moves with the new nodes
24:          public:
25:                  DList() : first[6](NULL), last[7](NULL)
26:                  {
27:                  }
28:
29:                  ~DList()
30:                  {
31:                          remove_all_nodes[8]();
32:                  }
33:
34:                  void add_node(void * info)
35:                  {
36:                          node[2] * new_node = new node[9](info[10]);
37:                          if (last[7])
38:                          {
39:                                  last[7]->next[3] = new_node[11];
40:                                  new_node[11]->prev[4] = last[7];
41:                                  last[7] = new_node[11];
42:                          }
43:                          else
44:                          {
45:                                  first[6] = last[7] = new_node[11];
46:                          }
47:                  }
48:
49:                  inline node[2] * get_first_node()
50:                  {
51:                          return first[6];
52:                  }
53:
```

Footnotes:

1: *dlist.h:2*
2: *dlist.h:7*
3: *dlist.h:9*
4: *dlist.h:10*
5: *dlist.h:11*
6: *dlist.h:22*
7: *dlist.h:23*
8: *dlist.h:137*
9: *dlist.h:12*
10: *dlist.h:34*
11: *dlist.h:36*

```
54:                 inline node[1] * get_last_node()
55:                 {
56:                         return last[2];
57:                 }
58:
59:                 inline node[1] * get_next_node(node[1] * current_node)
60:                 {
61:                         return current_node[3]->next[4];
62:                 }
63:
64:                 inline node[1] * get_prev_node(node[1] * current_node)
65:                 {
66:                         return current_node[5]->prev[6];
67:                 }
68:
69:                 node[1] * find_node(void * info)
70:                 {
71:                         node[1] * current = first[7];
72:                         while (current[8])
73:                         {
74:                                 if (current[8]->info[9] == info[10])
75:                                         return current[8];
76:                                 current[8] = current[8]->next[4];
77:                         }
78:                         return NULL;
79:                 }
80:
81:                 void insert_node_after(node[1] * given_node, void * info)
82:                 {
83:                         assert(given_node[11] != 0);
84:
85:                         node[1] * new_node = new node[12](info[13]);
86:
87:                         node[1] * next_node = given_node[11]->next[4];
88:                         given_node[11]->next[4] = new_node[14];
89:                         new_node[14]->prev[6] = given_node[11];
90:                         new_node[14]->next[4] = next_node[15];
91:                         if (next_node[15])
92:                                 next_node[15]->prev[6] = new_node[14];
93:                         if (given_node[11] == last[2])
94:                                 last[2] = new_node[14];
95:                 }
96:
97:
98:                 void insert_node_before(node[1] * given_node, void * info)
99:                 {
100:                        assert(given_node[16] != 0);
101:
102:                        node[1] * new_node = new node[12](info[17]);
103:
104:                        node[1] * prev_node = given_node[16]->prev[6];
105:                        given_node[16]->prev[6] = new_node[18];
106:                        new_node[18]->next[4] = given_node[16];
```

Footnotes:

1: *dlist.h:7*
2: *dlist.h:23*
3: *dlist.h:59*
4: *dlist.h:9*
5: *dlist.h:64*
6: *dlist.h:10*
7: *dlist.h:22*
8: *dlist.h:71*
9: *dlist.h:11*
10: *dlist.h:69*
11: *dlist.h:81*
12: *dlist.h:12*
13: *dlist.h:81*
14: *dlist.h:85*
15: *dlist.h:87*
16: *dlist.h:98*
17: *dlist.h:98*
18: *dlist.h:102*

```
107:                        new_node[1]->prev[2] = prev_node[3];
108:                        if (prev_node[3])
109:                                prev_node[3]->next[4] = new_node[1];
110:                        if (given_node[5] == first[6])
111:                                first[6] = new_node[1];
112:                }
113:
114:                void remove_node(node[7] * given_node)
115:                {
116:                        assert(given_node[8] != 0);
117:
118:                        node[7] * prev_node = given_node[8]->prev[2];
119:                        node[7] * next_node = given_node[8]->next[4];
120:                        if (prev_node[9])
121:                                prev_node[9]->next[4] = next_node[10];
122:                        else
123:                        {
124:                                first[6] = next_node[10];
125:                                next_node[10]->prev[2] = NULL;
126:                        }
127:                        if (next_node[10])
128:                                next_node[10]->prev[2] = prev_node[9];
129:                        else
130:                        {
131:                                last[11] = prev_node[9];
132:                                prev_node[9]->next[4] = NULL;
133:                        }
134:                }
135:
136:                // Note: this function does NOT deallocate the memory pointed by "info" field
137:                void remove_all_nodes()
138:                {
139:                        node[7] * current = first[6];
140:                        while (current[12])
141:                        {
142:                                node[7] * next = current[12]->next[4];
143:                                delete current[12];
144:                                current[12] = next[13];
145:                        }
146:                        first[6] = last[11] = NULL;
147:                }
148:
149:        };
150:        #endif // DLIST_H
```

Footnotes:

1: *dlist.h:102*
2: *dlist.h:10*
3: *dlist.h:104*
4: *dlist.h:9*
5: *dlist.h:98*
6: *dlist.h:22*
7: *dlist.h:7*
8: *dlist.h:114*
9: *dlist.h:118*
10: *dlist.h:119*
11: *dlist.h:23*
12: *dlist.h:139*
13: *dlist.h:142*

```
1:  //-------------------------------------------------------------------
2:  //              HashTable  and  HashTableIterator
3:  //-------------------------------------------------------------------
4:  //
5:  //           < - - - - - - - - - - capacity - - - - - - - - - - ->
6:  //           ------------------------------------------------------
7:  // table --->| null |  null |      | null |  . . .   | null |      |
8:  //           ------------------------------------------------------
9:  //                            |                              |
10: //           ----------------------         ----------------------
11: //           | hash key value next |         | hash key value next |
12: //           ----------------------         ----------------------
13: //                            |                              |
14: //           ----------------------                        null
15: //           | hash key value next |
16: //           ----------------------
17: //                            |
18: //                           null
19: //
20: #ifndef HASHTABLE_H[1]
21: #define HASHTABLE_H
22:
23: #include "string.h"
24: //#include "vector.h"
25:
26: class HashTableIterator[2];
27:
28: class HashTable {
29: protected:
30:     static long HashCode[3](char * p);
31:  public:
32:    struct HashTableEntry {
33:       long hash;
34:       char *  key;
35:       void *  value;
36:       HashTableEntry[4] *next;
37:
38:       HashTableEntry(char * k, void * v) : value[5](v)
39:       {
40:           key[6] = new char[strlen(k[7])+1];
41:           strcpy(key[6],k[7]);
42:       }
43:    };
44:
45:     typedef HashTableIterator[2] iterator;
46:     typedef HashTable[8]::HashTableEntry[4] entry;
47:
48:    // Capacity the initial number of buckets.
49:    // FillPercent is usually a number between 0.0 and 1.0
50:    // (but you can use it in range 0..10.0 or
51:    //  use 0.75 and set table limit 16000 in function Rehash)
52:    HashTable[9](int Capacity=101, float FillPercent=0.75);
53:    ~HashTable(){ remove_all[10](); delete [] table[11];  }
```

Footnotes:

1: *hashtable.h:21*
2: *hashtable.h:82*
3: *hashtable.cpp:14*
4: *hashtable.h:32*
5: *hashtable.h:35*
6: *hashtable.h:34*
7: *hashtable.h:38*
8: *hashtable.h:28*
9: *hashtable.cpp:4*
10: *hashtable.cpp:91*
11: *hashtable.h:67*

```
54:
55:            int insert[1](char * k, void * v);
56:            int remove[2](char * k);
57:            void*  find[3](char * k) const;
58:
59:            long size() const { return count[4]; }
60:            void remove_all[5]();
61:
62:            // For hash table iterator.
63:            HashTableEntry[6]** get_table() { return table[7];   }
64:            int  Capacity() const       { return capacity[8];}
65:
66:          protected:
67:            HashTableEntry[6] **table; // The hash table data.
68:            int   capacity;         // Length of allocated memory (table-> array of Entry)
69:            long  count;            // The total number of entries in the hash table.
70:            long  threshold;        // Rehashes the table when count exceeds this threshold.
71:            float fill_percent;     // The filling percent for the hashTable.
72:
73:            void  rehash[9]();         // Rehashes the content of the table into a bigger table.
74:         };
75:
76:
77:
78:
79:         //---------------------------------------------------------------------
80:         //                    Hash table iterator
81:         //---------------------------------------------------------------------
82:         class HashTableIterator {
83:            int index;
84:            HashTable[10]::HashTableEntry[6] *entry;
85:            HashTable[10]   &hashtable;
86:
87:          public:
88:            HashTableIterator(HashTable[10] &ht) : hashtable[11](ht) { reset[12](); }
89:            void reset[12]();
90:
91:            HashTable[10]::HashTableEntry[6]* next[13]();
92:         };
93:
94:
95:
96:         class MultiHashTable : public HashTable[10] {
97:         public:
98:             int insert[14](char * k, void * v);
99:             void * pop[15](char * k);
100:        };
101:
102:
103:
104:        #endif
```

Footnotes:

1: *hashtable.cpp:23*
2: *hashtable.cpp:54*
3: *hashtable.cpp:79*
4: *hashtable.h:69*
5: *hashtable.cpp:91*
6: *hashtable.h:32*
7: *hashtable.h:67*
8: *hashtable.h:68*
9: *hashtable.cpp:109*
10: *hashtable.h:28*
11: *hashtable.h:85*
12: *hashtable.cpp:143*
13: *hashtable.cpp:153*
14: *hashtable.cpp:171*
15: *hashtable.cpp:192*

```
1:  #ifndef JNI_WRAPPERS_H[1]
2:  #define JNI_WRAPPERS_H
3:
4:  #include "jni.h"
5:  #include "ObjectData.h"
6:
7:  // The base class of this hierarchy - jobject_wrapper - inherits from
8:  // both InstanceData class and _jobject - the dummy class defined in the
9:  // JNI standard header. Thus, every class in the hierachy has the InstanceData
10: // as its superclass and the corresponding superclass of the JNI dummy hierarchy.
11: // This allows implicit passing of InstanceData as a parameter to
12: // all the JNI methods.
13: // For the parameters of type jclass InstanceData will represent an instance
14: // of the java.lang.Class class (created once per a type)
15:
16: class jobject_wrapper : public _jobject, public InstanceData[2] {};
17: class jclass_wrapper : public _jclass, public jobject_wrapper[3] {};
18: class jthrowable_wrapper : public _jthrowable, public jobject_wrapper[3] {};
19: class jstring_wrapper : public _jstring, public jobject_wrapper[3] {};
20: class jarray_wrapper : public _jarray, public jobject_wrapper[3] {};
21: class jbooleanArray_wrapper : public _jbooleanArray, public jarray_wrapper[4] {};
22: class jbyteArray_wrapper : public _jbyteArray, public jarray_wrapper[4] {};
23: class jcharArray_wrapper : public _jcharArray, public jarray_wrapper[4] {};
24: class jshortArray_wrapper : public _jshortArray, public jarray_wrapper[4] {};
25: class jintArray_wrapper : public _jintArray, public jarray_wrapper[4] {};
26: class jlongArray_wrapper : public _jlongArray, public jarray_wrapper[4] {};
27: class jfloatArray_wrapper : public _jfloatArray, public jarray_wrapper[4] {};
28: class jdoubleArray_wrapper : public _jdoubleArray, public jarray_wrapper[4] {};
29: class jobjectArray_wrapper : public _jobjectArray, public jarray_wrapper[4] {};
30:
31:
32: // These functions initially called by the NativeHandler to register
33: // the "proper" registerNatives function
34:
35: void Java_java_lang_Object_register[5](JNIEnv * env);
36: void Java_java_lang_Class_register[6](JNIEnv * env);
37: void Java_java_lang_Runtime_register(JNIEnv * env);
38: void Java_java_lang_System_register[7](JNIEnv * env);
39: void Java_java_lang_Thread_register[8](JNIEnv * env);
40:
41: void Java_java_security_AccessController_register[9](JNIEnv * env);
42: void Java_java_io_FileInputStream_register[10](JNIEnv * env);
43: void Java_java_io_FileOutputStream_register[11](JNIEnv * env);
44: void Java_java_io_FileDescriptor_register[12](JNIEnv * env);
45: void Java_java_lang_ClassLoader_register[13](JNIEnv * env);
46:
47:
48: #endif JNI_WRAPPERS
```

Footnotes:

1: *jni_wrappers.h:2*
2: *ObjectData.h:113*
3: *jni_wrappers.h:16*
4: *jni_wrappers.h:20*
5: *java_lang_Object.cpp:161*
6: *java_lang_Class.cpp:321*
7: *java_lang_System.cpp:154*
8: *java_lang_Thread.cpp:187*
9: *java_security_AccessController.cpp:74*
10: *java_io_FileInputStream.cpp:78*
11: *java_io_FileOutputStream.cpp:100*
12: *java_io_FileDescriptor.cpp:36*
13: *java_lang_ClassLoader.cpp:65*

```
1:  #include "defs.h"
2:
3:
4:  /*****************************************************************************************
5:
6:                                  Instruction Set Summary
7:                                  -----------------------
8:
9:  A Java virtual machine instruction consists of a one-byte opcode specifying the operation
10: to be performed, followed by zero or more operands supplying arguments or data that are used
11: by the operation. Many instructions have no operands and consist only of an opcode.
12:
13: Ignoring exceptions, the inner loop of a Java virtual machine interpreter is effectively:
14:
15: do {
16:      fetch an opcode;
17:      if (operands) fetch operands;
18:      execute the action for the opcode;
19: } while (there is more to do);
20:
21: (See method "step" of the class Thread implemented in "Thread.cpp" for the detailed
22: explanations of the interpreter loop)
23:
24: The number and size of the operands are determined by the opcode. If an operand is more than
25: one byte in size, then it is stored in big-endian order-high-order byte first.
26: For example, an unsigned 16-bit index into the local variables is stored as
27: two unsigned bytes, byte1 and byte2, such that its value is  (byte1 << 8) | byte2.
28:
29: The bytecode instruction stream is only single-byte aligned.
30: The two exceptions are the tableswitch and lookupswitch instructions,
31: which are padded to force internal alignment of some of their operands on 4-byte boundaries.
32:
33: *****************************************************************************************/
34:
35: #ifndef OPCODES_H[1]
36: #define OPCODES_H
37:
38: // All the opcode mnemonics as they are defined in the JVM specification
39:
40: #define NOP                     (0x00)
41: #define ACONST_NULL     (0x01)
42: #define ICONST_M1       (0x02)
43: #define ICONST_0        (0x03)
44: #define ICONST_1        (0x04)
45: #define ICONST_2        (0x05)
46: #define ICONST_3        (0x06)
47: #define ICONST_4        (0x07)
48: #define ICONST_5        (0x08)
49: #define LCONST_0        (0x09)
50: #define LCONST_1        (0x0a)
51: #define FCONST_0        (0x0b)
52: #define FCONST_1        (0x0c)
53: #define FCONST_2        (0x0d)
```

Footnotes:

1: *opcodes.h:36*

```
54:         #define DCONST_0        (0x0e)
55:         #define DCONST_1        (0x0f)
56:         #define BIPUSH          (0x10)
57:         #define SIPUSH          (0x11)
58:         #define LDC                     (0x12)
59:         #define LDC_W       (0x13)
60:         #define LDC2_W      (0x14)
61:         #define ILOAD       (0x15)
62:         #define LLOAD       (0x16)
63:         #define FLOAD       (0x17)
64:         #define DLOAD       (0x18)
65:         #define ALOAD       (0x19)
66:         #define ILOAD_0         (0x1a)
67:         #define ILOAD_1         (0x1b)
68:         #define ILOAD_2         (0x1c)
69:         #define ILOAD_3         (0x1d)
70:         #define LLOAD_0         (0x1e)
71:         #define LLOAD_1         (0x1f)
72:         #define LLOAD_2         (0x20)
73:         #define LLOAD_3         (0x21)
74:         #define FLOAD_0         (0x22)
75:         #define FLOAD_1         (0x23)
76:         #define FLOAD_2         (0x24)
77:         #define FLOAD_3         (0x25)
78:         #define DLOAD_0         (0x26)
79:         #define DLOAD_1         (0x27)
80:         #define DLOAD_2         (0x28)
81:         #define DLOAD_3         (0x29)
82:         #define ALOAD_0         (0x2a)
83:         #define ALOAD_1         (0x2b)
84:         #define ALOAD_2         (0x2c)
85:         #define ALOAD_3         (0x2d)
86:         #define IALOAD          (0x2e)
87:         #define LALOAD      (0x2f)
88:         #define FALOAD      (0x30)
89:         #define DALOAD      (0x31)
90:         #define AALOAD      (0x32)
91:         #define BALOAD      (0x33)
92:         #define CALOAD      (0x34)
93:         #define SALOAD      (0x35)
94:         #define ISTORE      (0x36)
95:         #define LSTORE      (0x37)
96:         #define FSTORE      (0x38)
97:         #define DSTORE      (0x39)
98:         #define ASTORE      (0x3a)
99:         #define ISTORE_0    (0x3b)
100:        #define ISTORE_1    (0x3c)
101:        #define ISTORE_2    (0x3d)
102:        #define ISTORE_3    (0x3e)
103:        #define LSTORE_0    (0x3f)
104:        #define LSTORE_1    (0x40)
105:        #define LSTORE_2    (0x41)
106:        #define LSTORE_3    (0x42)
```

```
107:          #define FSTORE_0     (0x43)
108:          #define FSTORE_1     (0x44)
109:          #define FSTORE_2     (0x45)
110:          #define FSTORE_3     (0x46)
111:          #define DSTORE_0     (0x47)
112:          #define DSTORE_1     (0x48)
113:          #define DSTORE_2     (0x49)
114:          #define DSTORE_3     (0x4a)
115:          #define ASTORE_0     (0x4b)
116:          #define ASTORE_1     (0x4c)
117:          #define ASTORE_2     (0x4d)
118:          #define ASTORE_3     (0x4e)
119:          #define IASTORE      (0x4f)
120:          #define LASTORE      (0x50)
121:          #define FASTORE      (0x51)
122:          #define DASTORE      (0x52)
123:          #define AASTORE      (0x53)
124:          #define BASTORE      (0x54)
125:          #define CASTORE      (0x55)
126:          #define SASTORE      (0x56)
127:          #define POP          (0x57)
128:          #define POP2         (0x58)
129:          #define DUP          (0x59)
130:          #define DUP_X1       (0x5a)
131:          #define DUP_X2       (0x5b)
132:          #define DUP2         (0x5c)
133:          #define DUP2_X1      (0x5d)
134:          #define DUP2_X2      (0x5e)
135:          #define SWAP         (0x5f)
136:          #define IADD         (0x60)
137:          #define LADD         (0x61)
138:          #define FADD         (0x62)
139:          #define DADD         (0x63)
140:          #define ISUB         (0x64)
141:          #define LSUB         (0x65)
142:          #define FSUB         (0x66)
143:          #define DSUB         (0x67)
144:          #define IMUL         (0x68)
145:          #define LMUL         (0x69)
146:          #define FMUL         (0x6a)
147:          #define DMUL         (0x6b)
148:          #define IDIV         (0x6c)
149:          #define LDIV         (0x6d)
150:          #define FDIV         (0x6e)
151:          #define DDIV         (0x6f)
152:          #define IREM         (0x70)
153:          #define LREM         (0x71)
154:          #define FREM         (0x72)
155:          #define DREM         (0x73)
156:          #define INEG         (0x74)
157:          #define LNEG         (0x75)
158:          #define FNEG         (0x76)
159:          #define DNEG         (0x77)
```

```
160:          #define ISHL         (0x78)
161:          #define LSHL         (0x79)
162:          #define ISHR         (0x7a)
163:          #define LSHR         (0x7b)
164:          #define IUSHR        (0x7c)
165:          #define LUSHR        (0x7d)
166:          #define IAND         (0x7e)
167:          #define LAND         (0x7f)
168:          #define IOR          (0x80)
169:          #define LOR          (0x81)
170:          #define IXOR         (0x82)
171:          #define LXOR         (0x83)
172:          #define IINC         (0x84)
173:          #define I2L          (0x85)
174:          #define I2F          (0x86)
175:          #define I2D          (0x87)
176:          #define L2I          (0x88)
177:          #define L2F          (0x89)
178:          #define L2D          (0x8a)
179:          #define F2I          (0x8b)
180:          #define F2L          (0x8c)
181:          #define F2D          (0x8d)
182:          #define D2I          (0x8e)
183:          #define D2L          (0x8f)
184:          #define D2F          (0x90)
185:          #define I2B          (0x91)
186:          #define I2C          (0x92)
187:          #define I2S          (0x93)
188:          #define LCMP         (0x94)
189:          #define FCMPL        (0x95)
190:          #define FCMPG        (0x96)
191:          #define DCMPL        (0x97)
192:          #define DCMPG        (0x98)
193:          #define IFEQ         (0x99)
194:          #define IFNE         (0x9a)
195:          #define IFLT         (0x9b)
196:          #define IFGE         (0x9c)
197:          #define IFGT         (0x9d)
198:          #define IFLE         (0x9e)
199:          #define IF_ICMPEQ    (0x9f)
200:          #define IF_ICMPNE    (0xa0)
201:          #define IF_ICMPLT    (0xa1)
202:          #define IF_ICMPGE    (0xa2)
203:          #define IF_ICMPGT    (0xa3)
204:          #define IF_ICMPLE    (0xa4)
205:          #define IF_ACMPEQ    (0xa5)
206:          #define IF_ACMPNE    (0xa6)
207:          #define GOTO         (0xa7)
208:          #define JSR          (0xa8)
209:          #define RET          (0xa9)
210:          #define TABLESWITCH  (0xaa)
211:          #define LOOKUPSWITCH (0xab)
212:          #define IRETURN      (0xac)
```

```
213:        #define LRETURN      (0xad)
214:        #define FRETURN      (0xae)
215:        #define DRETURN      (0xaf)
216:        #define ARETURN      (0xb0)
217:        #define RETURN       (0xb1)
218:        #define GETSTATIC    (0xb2)
219:        #define PUTSTATIC    (0xb3)
220:        #define GETFIELD     (0xb4)
221:        #define PUTFIELD     (0xb5)
222:        #define INVOKEVIRTUAL (0xb6)
223:        #define INVOKESPECIAL (0xb7)
224:        #define INVOKESTATIC  (0xb8)
225:        #define INVOKEINTERFACE (0xb9)
226:        #define XXXUNUSEDXXX1 (0xba)
227:        #define NEW          (0xbb)
228:        #define NEWARRAY     (0xbc)
229:        #define ANEWARRAY    (0xbd)
230:        #define ARRAYLENGTH  (0xbe)
231:        #define ATHROW       (0xbf)
232:        #define CHECKCAST    (0xc0)
233:        #define INSTANCEOF   (0xc1)
234:        #define MONITORENTER (0xc2)
235:        #define MONITOREXIT  (0xc3)
236:        #define WIDE         (0xc4)
237:        #define MULTIANEWARRAY (0xc5)
238:        #define IFNULL       (0xc6)
239:        #define IFNONNULL    (0xc7)
240:        #define GOTO_W       (0xc8)
241:        #define JSR_W        (0xc9)
242:
243:        // Reserved opcodes
244:
245:        #define BREAKPOINT   (0xca)
246:        #define IMPDEP1      (0xfe)
247:        #define IMPDEP2      (0xff)
248:
249:        class Thread[1];
250:
251:        // General instruction function prototype
252:        typedef InstructionResult[2] execute_instruction(Thread *);
253:
254:        struct Opcode
255:        {
256:                char * mnemonic;
257:                int argument_number; // how many bytes are the arguments of this command
258:                                     // -1 means variable-length instruction
259:                execute_instruction[3] * exec_function;
260:        };
261:
262:        extern Opcode[4] opcodes[5][];
263:
264:        //-----------------------------------------------------------------------------------
265:        // Instruction function prototypes
```

Footnotes:

1: *Thread.h:17*
2: *defs.h:120*
3: *opcodes.h:252*
4: *opcodes.h:254*
5: *opcodes.cpp:3215*

```
266:          //-----------------------------------------------------------------------------------------
267:
268:          // Load and store instructions (see "loadstore.cpp")
269:
270:          InstructionResult[1] bipush[2](Thread[3] * thread);
271:          InstructionResult[1] sipush[4](Thread[3] * thread);
272:          InstructionResult[1] aconst_null[5](Thread[3] * thread);
273:          InstructionResult[1] ldc[6](Thread[3] * thread);
274:          InstructionResult[1] ldc_w[7](Thread[3] * thread);
275:          InstructionResult[1] ldc2_w[8](Thread[3] * thread);
276:          InstructionResult[1] aconst_null[5](Thread[3] * thread);
277:          InstructionResult[1] iload_0[9](Thread[3] * thread);
278:          InstructionResult[1] iload_1[10](Thread[3] * thread);
279:          InstructionResult[1] iload_2[11](Thread[3] * thread);
280:          InstructionResult[1] iload_3[12](Thread[3] * thread);
281:          InstructionResult[1] iload[13](Thread[3] * thread);
282:          InstructionResult[1] lload_0[14](Thread[3] * thread);
283:          InstructionResult[1] lload_1[15](Thread[3] * thread);
284:          InstructionResult[1] lload_2[16](Thread[3] * thread);
285:          InstructionResult[1] lload_3[17](Thread[3] * thread);
286:          InstructionResult[1] lload[18](Thread[3] * thread);
287:          InstructionResult[1] fload_0[19](Thread[3] * thread);
288:          InstructionResult[1] fload_1[20](Thread[3] * thread);
289:          InstructionResult[1] fload_2[21](Thread[3] * thread);
290:          InstructionResult[1] fload_3[22](Thread[3] * thread);
291:          InstructionResult[1] fload[23](Thread[3] * thread);
292:          InstructionResult[1] dload_0[24](Thread[3] * thread);
293:          InstructionResult[1] dload_1[25](Thread[3] * thread);
294:          InstructionResult[1] dload_2[26](Thread[3] * thread);
295:          InstructionResult[1] dload_3[27](Thread[3] * thread);
296:          InstructionResult[1] dload[28](Thread[3] * thread);
297:          InstructionResult[1] aload_0[29](Thread[3] * thread);
298:          InstructionResult[1] aload_1[30](Thread[3] * thread);
299:          InstructionResult[1] aload_2[31](Thread[3] * thread);
300:          InstructionResult[1] aload_3[32](Thread[3] * thread);
301:          InstructionResult[1] aload[33](Thread[3] * thread);
302:          InstructionResult[1] iconst_0[34](Thread[3] * thread);
303:          InstructionResult[1] iconst_1[35](Thread[3] * thread);
304:          InstructionResult[1] iconst_2[36](Thread[3] * thread);
305:          InstructionResult[1] iconst_3[37](Thread[3] * thread);
306:          InstructionResult[1] iconst_4[38](Thread[3] * thread);
307:          InstructionResult[1] iconst_5[39](Thread[3] * thread);
308:          InstructionResult[1] iconst_m1[40](Thread[3] * thread);
309:          InstructionResult[1] lconst_0[41](Thread[3] * thread);
310:          InstructionResult[1] lconst_1[42](Thread[3] * thread);
311:          InstructionResult[1] fconst_0[43](Thread[3] * thread);
312:          InstructionResult[1] fconst_1[44](Thread[3] * thread);
313:          InstructionResult[1] fconst_2[45](Thread[3] * thread);
314:          InstructionResult[1] dconst_0[46](Thread[3] * thread);
315:          InstructionResult[1] dconst_1[47](Thread[3] * thread);
316:          InstructionResult[1] istore(Thread[3] * thread);
317:          InstructionResult[1] istore_0[48](Thread[3] * thread);
318:          InstructionResult[1] istore_1[49](Thread[3] * thread);
```

Footnotes:

1: *defs.h:120*
2: *opcodes.cpp:466*
3: *Thread.h:17*
4: *opcodes.cpp:474*
5: *opcodes.cpp:25*
6: *opcodes.cpp:1601*
7: *opcodes.cpp:1608*
8: *opcodes.cpp:1617*
9: *opcodes.cpp:40*
10: *opcodes.cpp:44*
11: *opcodes.cpp:48*
12: *opcodes.cpp:52*
13: *opcodes.cpp:56*
14: *opcodes.cpp:71*
15: *opcodes.cpp:75*
16: *opcodes.cpp:79*
17: *opcodes.cpp:83*
18: *opcodes.cpp:87*
19: *opcodes.cpp:103*
20: *opcodes.cpp:107*
21: *opcodes.cpp:111*
22: *opcodes.cpp:115*
23: *opcodes.cpp:119*
24: *opcodes.cpp:134*
25: *opcodes.cpp:138*
26: *opcodes.cpp:142*
27: *opcodes.cpp:146*
28: *opcodes.cpp:150*
29: *opcodes.cpp:165*
30: *opcodes.cpp:169*
31: *opcodes.cpp:173*
32: *opcodes.cpp:177*
33: *opcodes.cpp:181*
34: *opcodes.cpp:195*
35: *opcodes.cpp:199*
36: *opcodes.cpp:203*
37: *opcodes.cpp:207*
38: *opcodes.cpp:211*
39: *opcodes.cpp:215*
40: *opcodes.cpp:219*
41: *opcodes.cpp:231*
42: *opcodes.cpp:235*
43: *opcodes.cpp:248*
44: *opcodes.cpp:252*
45: *opcodes.cpp:256*
46: *opcodes.cpp:268*
47: *opcodes.cpp:272*
48: *opcodes.cpp:298*
49: *opcodes.cpp:302*

```
319:          InstructionResult[1] istore_2[2](Thread[3] * thread);
320:          InstructionResult[1] istore_3[4](Thread[3] * thread);
321:          InstructionResult[1] lstore(Thread[3] * thread);
322:          InstructionResult[1] lstore_0[5](Thread[3] * thread);
323:          InstructionResult[1] lstore_1[6](Thread[3] * thread);
324:          InstructionResult[1] lstore_2[7](Thread[3] * thread);
325:          InstructionResult[1] lstore_3[8](Thread[3] * thread);
326:          InstructionResult[1] fstore(Thread[3] * thread);
327:          InstructionResult[1] fstore_0[9](Thread[3] * thread);
328:          InstructionResult[1] fstore_1[10](Thread[3] * thread);
329:          InstructionResult[1] fstore_2[11](Thread[3] * thread);
330:          InstructionResult[1] fstore_3[12](Thread[3] * thread);
331:          InstructionResult[1] dstore(Thread[3] * thread);
332:          InstructionResult[1] dstore_0[13](Thread[3] * thread);
333:          InstructionResult[1] dstore_1[14](Thread[3] * thread);
334:          InstructionResult[1] dstore_2[15](Thread[3] * thread);
335:          InstructionResult[1] dstore_3[16](Thread[3] * thread);
336:          InstructionResult[1] astore(Thread[3] * thread);
337:          InstructionResult[1] astore_0[17](Thread[3] * thread);
338:          InstructionResult[1] astore_1[18](Thread[3] * thread);
339:          InstructionResult[1] astore_2[19](Thread[3] * thread);
340:          InstructionResult[1] astore_3[20](Thread[3] * thread);
341:          InstructionResult[1] bipush[21](Thread[3] * thread);
342:          InstructionResult[1] sipush[22](Thread[3] * thread);
343:
344:          // Method invokation and return instructions (see "method.cpp")
345:
346:          InstructionResult[1] invokevirtual[23](Thread[3] * thread);
347:          InstructionResult[1] invokestatic[24](Thread[3] * thread);
348:          InstructionResult[1] invokespecial[25](Thread[3] * thread);
349:          InstructionResult[1] invokeinterface[26](Thread[3] * thread);
350:          InstructionResult[1] _return[27](Thread[3] * thread);
351:          InstructionResult[1] ireturn[28](Thread[3] * thread);
352:          InstructionResult[1] lreturn[29](Thread[3] * thread);
353:          InstructionResult[1] freturn[30](Thread[3] * thread);
354:          InstructionResult[1] dreturn[31](Thread[3] * thread);
355:          InstructionResult[1] areturn[32](Thread[3] * thread);
356:
357:
358:          #endif // OPCODES_H
359:
```

Footnotes:

1: *defs.h:120*
2: *opcodes.cpp:306*
3: *Thread.h:17*
4: *opcodes.cpp:310*
5: *opcodes.cpp:337*
6: *opcodes.cpp:341*
7: *opcodes.cpp:345*
8: *opcodes.cpp:349*
9: *opcodes.cpp:375*
10: *opcodes.cpp:379*
11: *opcodes.cpp:383*
12: *opcodes.cpp:387*
13: *opcodes.cpp:412*
14: *opcodes.cpp:416*
15: *opcodes.cpp:420*
16: *opcodes.cpp:424*
17: *opcodes.cpp:449*
18: *opcodes.cpp:453*
19: *opcodes.cpp:457*
20: *opcodes.cpp:461*
21: *opcodes.cpp:466*
22: *opcodes.cpp:474*
23: *opcodes.cpp:2235*
24: *opcodes.cpp:2254*
25: *opcodes.cpp:2273*
26: *opcodes.cpp:2292*
27: *opcodes.cpp:2319*
28: *opcodes.cpp:2325*
29: *opcodes.cpp:2331*
30: *opcodes.cpp:2337*
31: *opcodes.cpp:2343*
32: *opcodes.cpp:2349*

```
1:          /*
2:           *  "Java" C++Vector
3:           */
4:          #ifndef VECTOR_H[1]
5:          #define VECTOR_H
6:
7:          template <class T> class Vector {
8:            protected:
9:              T      *elementData;
10:             int    Capacity;          // size of elementData
11:
12:             int elementCount;         // number of element in the vector
13:             int capacityIncrement;    // capacity increment, double its if 0
14:
15:           public:
16:
17:             Vector(int _initialCapacity = 10, int _capacityIncrement = 0) :
18:                Capacity[2](_initialCapacity[3]), capacityIncrement[4](_capacityIncrement[5]),
19:                elementCount[6](0)  {  elementData[7] = new T[Capacity[2]];  }
20:
21:             ~Vector() { delete [] elementData[7]; }
22:
23:             int  size() const { return elementCount[6]; }
24:             bool is_empty() const { return elementCount[6]==0; }
25:             int  capacity() const { return Capacity[2]; }
26:
27:             T& operator[](int index){
28:                         return elementData[7][index[8]];
29:             }
30:
31:             T operator[](int index) const {
32:                         return elementData[7][index[9]];
33:             }
34:
35:
36:             const T&   element_at(int index){
37:               return elementData[7][index[10]];
38:             }
39:
40:             void add_element(const T& obj){
41:                ensure_capacity[11](elementCount[6]+1);
42:                elementData[7][elementCount[6]++] = obj[12];
43:             }
44:
45:             void set_element_at(const T& obj, int index){
46:                elementData[7][index[13]] = obj[14];
47:             }
48:
49:             void remove_element_at(int index){
50:
51:                for(int i = index[15] ; i < elementCount[6]-1 ; i++)
52:                   elementData[7][i[16]] = elementData[7][i[16]+1];
53:
```

Footnotes:

1: *vector.h:5*
2: *vector.h:10*
3: *vector.h:17*
4: *vector.h:13*
5: *vector.h:17*
6: *vector.h:12*
7: *vector.h:9*
8: *vector.h:27*
9: *vector.h:31*
10: *vector.h:36*
11: *vector.h:96*
12: *vector.h:40*
13: *vector.h:45*
14: *vector.h:45*
15: *vector.h:49*
16: *vector.h:51*

```
54:                elementCount[1]--;
55:             }
56:
57:             void insert_element_at(const T& obj, int index){
58:
59:                ensure_capacity[2](elementCount[1] + 1);
60:
61:                for (int i = elementCount[1]; i > index[3] ; i--)
62:                    elementData[4][i[5]] = elementData[4][i[5]-1];
63:
64:                elementCount[1]++;
65:                elementData[4][index[3]] = obj[6];
66:             }
67:
68:             void remove_all_elements(){  elementCount[1] = 0; }
69:
70:             const T&   first_element(){
71:               return elementData[4][0];
72:             }
73:
74:             const T&   last_element(){
75:                return elementData[4][elementCount[1] - 1];
76:             }
77:
78:             //Trims the capacity of this vector to be the vector's current size.
79:             //An application can use this operation to minimize the storage of a vector.
80:             void trim_to_size(){
81:                if(elementCount[1] < Capacity[7]) {
82:                   T *oldElementData = elementData[4];
83:                   elementData[4] = new T[elementCount[1]];
84:                   for(int i=0 ; i < elementCount[1] ; i++)
85:                      elementData[4][i[8]] = oldElementData[9][i[8]];
86:
87:                   Capacity[7] = elementCount[1];
88:                   delete [] oldElementData[9];
89:                }
90:             }
91:
92:
93:             // Increases the capacity of this vector, if necessary, to ensure
94:             // that it can hold at least the number of components specified by
95:             // the minimum capacity argument.
96:             void ensure_capacity(int minCapacity) {
97:                if(minCapacity[10] > Capacity[7]) {
98:                   T *oldData = elementData[4];
99:
100:                  Capacity[7] = (capacityIncrement[11] > 0) ?
101:                           (Capacity[7] + capacityIncrement[11]) : (Capacity[7] * 2);
102:
103:                  if(Capacity[7] < minCapacity[10])
104:                     Capacity[7] = minCapacity[10];
105:
106:                  elementData[4] = new T[Capacity[7]];
```

Footnotes:

1: *vector.h:12*
2: *vector.h:96*
3: *vector.h:57*
4: *vector.h:9*
5: *vector.h:61*
6: *vector.h:57*
7: *vector.h:10*
8: *vector.h:84*
9: *vector.h:82*
10: *vector.h:96*
11: *vector.h:13*

```
107:
108:                    for(int i=0 ; i < elementCount[1] ; i++)
109:                       elementData[2][i[3]] = oldData[4][i[3]];
110:
111:                    delete [] oldData[4];
112:                 }
113:              }
114:
115:
116:              // Sets the size of this vector. If the new size is greater than the
117:              // current size, new items are added (by default constructor)
118:              // to the end of the vector.
119:              // If the new size is less than the current size, all
120:              // components at index newSize and greater are discarded.
121:              void set_size(int newSize){
122:                 if( newSize[5] > elementCount[1] )
123:                    ensureCapacity(newSize[5]);
124:                 else
125:                    for(int i=newSize[5] ; i < elementCount[1] ; i++)
126:                       elementData[2][i[6]] = T();
127:
128:                 elementCount[1] = newSize[5];
129:              }
130:
131:
132:              void copy_into(T obj[]) const {
133:                          int i = elementCount[1];
134:                          while( i[7]-- > 0 )
135:                                  obj[8][i[7]] = elementData[2][i[7]];
136:                   }
137:           };
138:
139:
140:
141:
142:           template <class T> class Queue : public Vector[9]<T> {
143:
144:              public:
145:
146:                   void push(const T& obj) {
147:                       add_element[10](obj[11]);
148:                   }
149:
150:                   void pop() {
151:                       remove_element_at[12](0);
152:                   }
153:
154:                   const T& front() {
155:                       return element_at[13](0);
156:                   }
157:
158:           };
159:
```

Footnotes:

1: *vector.h:12*
2: *vector.h:9*
3: *vector.h:108*
4: *vector.h:98*
5: *vector.h:121*
6: *vector.h:125*
7: *vector.h:133*
8: *vector.h:132*
9: *vector.h:7*
10: *vector.h:40*
11: *vector.h:146*
12: *vector.h:49*
13: *vector.h:36*

```
160:
161:          #endif
```

```
 1:
 2:          #include <iostream.h>
 3:          #include <assert.h>
 4:          #include <memory.h>
 5:
 6:          #include "defs.h"
 7:          #include "AttributeInfo.h"
 8:          #include "CodeManager.h"
 9:
10:
11:          Result[1] ConstantValue_attribute[2]::read(u1[3] *& stream_pointer)
12:          {
13:                  // Read the attribute length (should be always 2)
14:
15:                  memcpy_u4[4](&attribute_length[5], stream_pointer[6]);
16:                  stream_pointer[6] += sizeof(u4[7]);
17:
18:                  memcpy_u2[8](&constantvalue_index[9], stream_pointer[6]);
19:                  stream_pointer[6] += sizeof(u2[10]);
20:
21:                  return Success[11];
22:          }
23:
24:          void ConstantValue_attribute[2]::debug_print(ostream & out)
25:          {
26:                  out[12] << "      ConstantValue_attribute:" << endl;
27:                  out[12] << "         constantvalue_index: " << constantvalue_index[9] << endl;
28:          }
29:
30:          Result[1] Code_attribute[13]::read(u1[3] *& stream_pointer)
31:          {
32:                  int i;
33:
34:                  // Read the attribute length
35:
36:                  memcpy_u4[4](&attribute_length[5], stream_pointer[14]);
37:                  stream_pointer[14] += sizeof(u4[7]);
38:
39:                  memcpy_u2[8](&max_stack[15], stream_pointer[14]);
40:                  stream_pointer[14] += sizeof(u2[10]);
41:
42:                  memcpy_u2[8](&max_locals[16], stream_pointer[14]);
43:                  stream_pointer[14] += sizeof(u2[10]);
44:
45:                  memcpy_u4[4](&code_length[17], stream_pointer[14]);
46:                  stream_pointer[14] += sizeof(u4[7]);
47:
48:                  // Allocate code array
49:
50:                  code[18] = new u1[3][code_length[17]];
51:                  assert(code[18] != 0);
52:
53:                  // Read code
```

Footnotes:

1: *defs.h:29*
2: *AttributeInfo.h:47*
3: *defs.h:168*
4: *memcpy_bigendian.cpp:27*
5: *AttributeInfo.h:22*
6: *AttributeInfo.cpp:11*
7: *defs.h:174*
8: *memcpy_bigendian.cpp:10*
9: *AttributeInfo.h:49*
10: *defs.h:172*
11: *defs.h:33*
12: *AttributeInfo.cpp:24*
13: *AttributeInfo.h:55*
14: *AttributeInfo.cpp:30*
15: *AttributeInfo.h:65*
16: *AttributeInfo.h:66*
17: *AttributeInfo.h:67*
18: *AttributeInfo.h:68*

```
54:
55:                 memcpy(code[1],stream_pointer[2],code_length[3]);
56:                 stream_pointer[2] += code_length[3];
57:
58:                 memcpy_u2[4](&exception_table_length[5], stream_pointer[2]);
59:                 stream_pointer[2] += sizeof(u2[6]);
60:
61:                 exception_table[7] = new exception_table_entry[8][exception_table_length[5]];
62:
63:                 // Read exceptions table
64:
65:                 for (i[9] = 0; i[9] < exception_table_length[5]; i[9]++)
66:                 {
67:                         memcpy_u2[4](&(exception_table[7][i[9]].start_pc[10]), stream_pointer[2]);
68:                         stream_pointer[2] += sizeof(u2[6]);
69:                         memcpy_u2[4](&(exception_table[7][i[9]].end_pc[11]), stream_pointer[2]);
70:                         stream_pointer[2] += sizeof(u2[6]);
71:                         memcpy_u2[4](&(exception_table[7][i[9]].handler_pc[12]), stream_pointer[2]);
72:                         stream_pointer[2] += sizeof(u2[6]);
73:                         memcpy_u2[4](&(exception_table[7][i[9]].catch_type[13]), stream_pointer[2]);
74:                         stream_pointer[2] += sizeof(u2[6]);
75:                 }
76:
77:                 // Read attributes count
78:
79:                 memcpy_u2[4](&attributes_count[14], stream_pointer[2]);
80:                 stream_pointer[2] += sizeof(u2[6]);
81:
82:                 // For now, just skip the necessary number of bytes in the input stream
83:
84:                 for (i[9] = 0; i[9] < attributes_count[14]; i[9]++)
85:                 {
86:                         // Skip the attribute_name_index
87:                         u2[6] tmp_attribute_name_index;
88:                         memcpy_u2[4](&tmp_attribute_name_index[15], stream_pointer[2]);
89:                         stream_pointer[2] += sizeof(u2[6]);
90:
91:                         u4[16] tmp_attribute_length;
92:                         memcpy_u4[17](&tmp_attribute_length[18], stream_pointer[2]);
93:                         stream_pointer[2] += sizeof(u4[16]);
94:
95:                         // Skip the attribute_length bytes in the input stream
96:                         stream_pointer[2] += tmp_attribute_length[18];
97:
98:                 }
99:
100:                return Success[19];
101:        }
102:
103:        void Code_attribute[20]::debug_print(ostream & out)
104:        {
105:                out[21] << "      Code_attribute:" << endl;
106:                out[21] << "         attribute_name_index:" << endl;
```

Footnotes:

1: *AttributeInfo.h:68*
2: *AttributeInfo.cpp:30*
3: *AttributeInfo.h:67*
4: *memcpy_bigendian.cpp:10*
5: *AttributeInfo.h:69*
6: *defs.h:172*
7: *AttributeInfo.h:70*
8: *AttributeInfo.h:57*
9: *AttributeInfo.cpp:32*
10: *AttributeInfo.h:59*
11: *AttributeInfo.h:60*
12: *AttributeInfo.h:61*
13: *AttributeInfo.h:62*
14: *AttributeInfo.h:71*
15: *AttributeInfo.cpp:87*
16: *defs.h:174*
17: *memcpy_bigendian.cpp:27*
18: *AttributeInfo.cpp:91*
19: *defs.h:33*
20: *AttributeInfo.h:55*
21: *AttributeInfo.cpp:103*

```
107:                out[1] << "         max_stack: " << max_stack[2] << endl;
108:                out[1] << "         max_locals: " << max_locals[3] << endl;
109:                out[1] << "         code_length: " << (unsigned int)code_length[4] << endl;
110:                out[1] << "         code: " << endl;
111:                debug_print_code[5](code[6], code_length[4], out[1]);
112:
113:                // Print exceptions
114:        }
115:
116:
117:        Result[7] Exceptions_attribute[8]::read(u1[9] *& stream_pointer)
118:        {
119:                // Read the attribute length
120:
121:                memcpy_u4[10](&attribute_length[11], stream_pointer[12]);
122:                stream_pointer[12] += sizeof(u4[13]);
123:
124:                memcpy_u2[14](&number_of_exceptions[15], stream_pointer[12]);
125:                stream_pointer[12] += sizeof(u2[16]);
126:
127:                // Allocate exception index table
128:
129:                exception_index_table[17] = new u2[16][number_of_exceptions[15]];
130:
131:                // Read the exception index table
132:
133:                for (int i = 0; i < number_of_exceptions[15]; i++)
134:                {
135:                        memcpy_u2[14](&(exception_index_table[17][i[18]]), stream_pointer[12]);
136:                        stream_pointer[12] += sizeof(u2[16]);
137:                }
138:
139:                return Success[19];
140:        }
141:
142:        void Exceptions_attribute[8]::debug_print(ostream & out)
143:        {
144:                out[20] << "      Exceptions_attribute:" << endl;
145:        }
146:
147:
148:        Result[7] InnerClasses_attribute[21]::read(u1[9] *& stream_pointer)
149:        {
150:                // Read the attribute length
151:
152:                memcpy_u4[10](&attribute_length[11], stream_pointer[22]);
153:                stream_pointer[22] += sizeof(u4[13]);
154:
155:                memcpy_u2[14](&number_of_classes[23], stream_pointer[22]);
156:                stream_pointer[22] += sizeof(u2[16]);
157:
158:                // Allocate inner classes info table
159:
```

Footnotes:

1: *AttributeInfo.cpp:103*
2: *AttributeInfo.h:65*
3: *AttributeInfo.h:66*
4: *AttributeInfo.h:67*
5: *opcodes.cpp:3424*
6: *AttributeInfo.h:68*
7: *defs.h:29*
8: *AttributeInfo.h:78*
9: *defs.h:168*
10: *memcpy_bigendian.cpp:27*
11: *AttributeInfo.h:22*
12: *AttributeInfo.cpp:117*
13: *defs.h:174*
14: *memcpy_bigendian.cpp:10*
15: *AttributeInfo.h:80*
16: *defs.h:172*
17: *AttributeInfo.h:81*
18: *AttributeInfo.cpp:133*
19: *defs.h:33*
20: *AttributeInfo.cpp:142*
21: *AttributeInfo.h:87*
22: *AttributeInfo.cpp:148*
23: *AttributeInfo.h:97*

```
160:                    classes[1] = new classes_entry[2][number_of_classes[3]];
161:
162:                    // Read inner classes info
163:
164:                    for (int i = 0; i < number_of_classes[3]; i++)
165:                    {
166:                            memcpy_u2[4](&(classes[1][i[5]].inner_class_info_index[6]), stream_pointer[7]);
167:                            stream_pointer[7] += sizeof(u2[8]);
168:                            memcpy_u2[4](&(classes[1][i[5]].outer_class_info_index[9]), stream_pointer[7]);
169:                            stream_pointer[7] += sizeof(u2[8]);
170:                            memcpy_u2[4](&(classes[1][i[5]].inner_name_index[10]), stream_pointer[7]);
171:                            stream_pointer[7] += sizeof(u2[8]);
172:                            memcpy_u2[4](&(classes[1][i[5]].inner_class_access_flags[11]), stream_pointer[7]);
173:                            stream_pointer[7] += sizeof(u2[8]);
174:                    }
175:
176:                    return Success[12];
177:            }
178:
179:            void InnerClasses_attribute[13]::debug_print(ostream & out)
180:            {
181:                    out[14] << "      InnerClasses_attribute:" << endl;
182:            }
183:
184:            Result[15] Synthetic_attribute[16]::read(u1[17] *& stream_pointer)
185:            {
186:                    // Read the attribute length (must be zero)
187:
188:                    memcpy_u4[18](&attribute_length[19], stream_pointer[20]);
189:                    stream_pointer[20] += sizeof(u4[21]);
190:
191:                    return Success[12];
192:            }
193:
194:            void Synthetic_attribute[16]::debug_print(ostream & out)
195:            {
196:                    out[22] << "      Synthetic_attribute:" << endl;
197:            }
198:
199:            Result[15] SourceFile_attribute[23]::read(u1[17] *& stream_pointer)
200:            {
201:                    // Read the attribute length (must be 2)
202:
203:                    memcpy_u4[18](&attribute_length[19], stream_pointer[24]);
204:                    stream_pointer[24] += sizeof(u4[21]);
205:
206:                    memcpy_u2[4](&sourcefile_index[25], stream_pointer[24]);
207:                    stream_pointer[24] += sizeof(u2[8]);
208:
209:                    return Success[12];
210:            }
211:
212:            void SourceFile_attribute[23]::debug_print(ostream & out)
```

Footnotes:

1: *AttributeInfo.h:98*
2: *AttributeInfo.h:89*
3: *AttributeInfo.h:97*
4: *memcpy_bigendian.cpp:10*
5: *AttributeInfo.cpp:164*
6: *AttributeInfo.h:91*
7: *AttributeInfo.cpp:148*
8: *defs.h:172*
9: *AttributeInfo.h:92*
10: *AttributeInfo.h:93*
11: *AttributeInfo.h:94*
12: *defs.h:33*
13: *AttributeInfo.h:87*
14: *AttributeInfo.cpp:179*
15: *defs.h:29*
16: *AttributeInfo.h:104*
17: *defs.h:168*
18: *memcpy_bigendian.cpp:27*
19: *AttributeInfo.h:22*
20: *AttributeInfo.cpp:184*
21: *defs.h:174*
22: *AttributeInfo.cpp:194*
23: *AttributeInfo.h:110*
24: *AttributeInfo.cpp:199*
25: *AttributeInfo.h:112*

```
213:        {
214:                out[1] << "      SourceFile_attribute:" << endl;
215:        }
216:
217:
218:        Result[2] LineNumberTable_attribute[3]::read(u1[4] *& stream_pointer)
219:        {
220:                // Read the attribute length
221:
222:                memcpy_u4[5](&attribute_length[6], stream_pointer[7]);
223:                stream_pointer[7] += sizeof(u4[8]);
224:
225:                memcpy_u2[9](&line_number_table_length[10], stream_pointer[7]);
226:                stream_pointer[7] += sizeof(u2[11]);
227:
228:                // Allocate line number info table
229:
230:                line_number_table[12] = new line_number_table_entry[13][line_number_table_length[10]];
231:
232:                // Read line number table
233:
234:                for (int i = 0; i < line_number_table_length[10]; i++)
235:                {
236:                        memcpy_u2[9](&(line_number_table[12][i[14]].start_pc[15]), stream_pointer[7]);
237:                        stream_pointer[7] += sizeof(u2[11]);
238:                        memcpy_u2[9](&(line_number_table[12][i[14]].line_number[16]), stream_pointer[7]);
239:                        stream_pointer[7] += sizeof(u2[11]);
240:                }
241:
242:                return Success[17];
243:        }
244:
245:        void LineNumberTable_attribute[3]::debug_print(ostream & out)
246:        {
247:                out[18] << "      LineNumberTable_attribute:" << endl;
248:        }
249:
250:
251:        Result[2] LocalVariableTable_attribute[19]::read(u1[4] *& stream_pointer)
252:        {
253:                // Read the attribute length
254:
255:                memcpy_u4[5](&attribute_length[6], stream_pointer[20]);
256:                stream_pointer[20] += sizeof(u4[8]);
257:
258:                memcpy_u2[9](&local_variable_table_length[21], stream_pointer[20]);
259:                stream_pointer[20] += sizeof(u2[11]);
260:
261:                // Allocate local variable info table
262:
263:                local_variable_table[22] = new local_variable_table_entry[23][local_variable_table_length[21]];
264:
265:                // Read local variable info table
```

Footnotes:

1: *AttributeInfo.cpp:212*
2: *defs.h:29*
3: *AttributeInfo.h:118*
4: *defs.h:168*
5: *memcpy_bigendian.cpp:27*
6: *AttributeInfo.h:22*
7: *AttributeInfo.cpp:218*
8: *defs.h:174*
9: *memcpy_bigendian.cpp:10*
10: *AttributeInfo.h:126*
11: *defs.h:172*
12: *AttributeInfo.h:127*
13: *AttributeInfo.h:120*
14: *AttributeInfo.cpp:234*
15: *AttributeInfo.h:122*
16: *AttributeInfo.h:123*
17: *defs.h:33*
18: *AttributeInfo.cpp:245*
19: *AttributeInfo.h:133*
20: *AttributeInfo.cpp:251*
21: *AttributeInfo.h:144*
22: *AttributeInfo.h:145*
23: *AttributeInfo.h:135*

```
266:
267:                    for (int i = 0; i < local_variable_table_length[1]; i++)
268:                    {
269:                            memcpy_u2[2](&(local_variable_table[3][i[4]].start_pc[5]), stream_pointer[6]);
270:                            stream_pointer[6] += sizeof(u2[7]);
271:                            memcpy_u2[2](&(local_variable_table[3][i[4]].length[8]), stream_pointer[6]);
272:                            stream_pointer[6] += sizeof(u2[7]);
273:                            memcpy_u2[2](&(local_variable_table[3][i[4]].name_index[9]), stream_pointer[6]);
274:                            stream_pointer[6] += sizeof(u2[7]);
275:                            memcpy_u2[2](&(local_variable_table[3][i[4]].descriptor_index[10]), stream_pointer[6]);
276:                            stream_pointer[6] += sizeof(u2[7]);
277:                            memcpy_u2[2](&(local_variable_table[3][i[4]].index[11]), stream_pointer[6]);
278:                            stream_pointer[6] += sizeof(u2[7]);
279:                    }
280:
281:                    return Success[12];
282:            }
283:
284:            void LocalVariableTable_attribute[13]::debug_print(ostream & out)
285:            {
286:                    out[14] << "      LocalVariableTable_attribute:" << endl;
287:            }
288:
289:
290:            Result[15] Deprecated_attribute[16]::read(u1[17] *& stream_pointer)
291:            {
292:                    // Read the attribute length (must be zero)
293:
294:                    memcpy_u4[18](&attribute_length[19], stream_pointer[20]);
295:                    stream_pointer[20] += sizeof(u4[21]);
296:
297:                    return Success[12];
298:            }
299:
300:            void Deprecated_attribute[16]::debug_print(ostream & out)
301:            {
302:                    out[22] << "      Deprecated_attribute:" << endl;
303:            }
304:
305:
306:            Result[15] Unrecognized_attribute[23]::read(u1[17] *& stream_pointer)
307:            {
308:                    // Read the attribute length (must be zero)
309:
310:                    memcpy_u4[18](&attribute_length[19], stream_pointer[24]);
311:                    stream_pointer[24] += sizeof(u4[21]);
312:
313:                    // Skip the attribute_length bytes in the input stream
314:
315:                    stream_pointer[24] += attribute_length[19];
316:
317:                    return Success[12];
318:            }
```

Footnotes:

1: *AttributeInfo.h:144*
2: *memcpy_bigendian.cpp:10*
3: *AttributeInfo.h:145*
4: *AttributeInfo.cpp:267*
5: *AttributeInfo.h:137*
6: *AttributeInfo.cpp:251*
7: *defs.h:172*
8: *AttributeInfo.h:138*
9: *AttributeInfo.h:139*
10: *AttributeInfo.h:140*
11: *AttributeInfo.h:141*
12: *defs.h:33*
13: *AttributeInfo.h:133*
14: *AttributeInfo.cpp:284*
15: *defs.h:29*
16: *AttributeInfo.h:151*
17: *defs.h:168*
18: *memcpy_bigendian.cpp:27*
19: *AttributeInfo.h:22*
20: *AttributeInfo.cpp:290*
21: *defs.h:174*
22: *AttributeInfo.cpp:300*
23: *AttributeInfo.h:159*
24: *AttributeInfo.cpp:306*

```
319:
320:          void Unrecognized_attribute[1]::debug_print(ostream & out)
321:          {
322:                  out[2] << "      Unrecognized_attribute:" << endl;
323:          }
324:
325:
```

Footnotes:
1: *AttributeInfo.h:159*
2: *AttributeInfo.cpp:320*

```
 1:
 2:          #include <string.h>
 3:          #include <iostream.h>
 4:          #include <wchar.h>
 5:
 6:          #include "ClassFile.h"
 7:          #include "AttributeInfo.h"
 8:          #include "ClassLoader.h"
 9:          #include "ObjectData.h"
10:          #include "opcodes.h"
11:          #include "Thread.h"
12:          #include "HandlePool.h"
13:
14:          Result[1] ReadAttributeInfoEntry(wchar_t * attribute_name, int length, u1[2] *& stream_pointer, attribute_info[3]
              *& ai)
15:          {
16:                          if (!wmemcmp(attribute_name[4],CONSTANT_VALUE_ATTRIBUTE_NAME[5],length[6]))
17:                                  ai[7] = new ConstantValue_attribute[8];
18:                          else if (!wmemcmp(attribute_name[4],CODE_ATTRIBUTE_NAME[9],length[6]))
19:                                  ai[7] = new Code_attribute[10];
20:                          else if (!wmemcmp(attribute_name[4],EXCEPTIONS_ATTRIBUTE_NAME[11],length[6]))
21:                                  ai[7] = new Exceptions_attribute[12];
22:                          else if (!wmemcmp(attribute_name[4],INNER_CLASSES_ATTRIBUTE_NAME[13],length[6]))
23:                                  ai[7] = new InnerClasses_attribute[14];
24:                          else if (!wmemcmp(attribute_name[4],SYNTHETIC_ATTRIBUTE_NAME[15],length[6]))
25:                                  ai[7] = new Synthetic_attribute[16];
26:                          else if (!wmemcmp(attribute_name[4],SOURCE_FILE_ATTRIBUTE_NAME[17],length[6]))
27:                                  ai[7] = new SourceFile_attribute[18];
28:                          else if (!wmemcmp(attribute_name[4],LINE_NUMBER_TABLE_ATTRIBUTE_NAME[19],length[6]))
29:                                  ai[7] = new LineNumberTable_attribute[20];
30:                          else if (!wmemcmp(attribute_name[4],LOCAL_VARIABLE_TABLE_ATTRIBUTE_NAME[21],length[6]))
31:                                  ai[7] = new LocalVariableTable_attribute[22];
32:                          else if (!wmemcmp(attribute_name[4],DEPRECATED_ATTRIBUTE_NAME[23],length[6]))
33:                                  ai[7] = new Deprecated_attribute[24];
34:                          else
35:                                  ai[7] = new Unrecognized_attribute[25];
36:
37:                          ai[7]->read(stream_pointer[26]);
38:
39:                          return Success[27];
40:          }
41:
42:          Result[1] field_method_info[28]::read(u1[2] *& stream_pointer)
43:          {
44:
45:                  memcpy_u2[29](&access_flags[30], stream_pointer[31]);
46:                  stream_pointer[31] += sizeof(u2[32]);
47:
48:                  memcpy_u2[29](&name_index[33], stream_pointer[31]);
49:                  stream_pointer[31] += sizeof(u2[32]);
50:
51:                  memcpy_u2[29](&descriptor_index[34], stream_pointer[31]);
52:                  stream_pointer[31] += sizeof(u2[32]);
```

Footnotes:

1: *defs.h:29*
2: *defs.h:168*
3: *AttributeInfo.h:15*
4: *ClassFile.cpp:14*
5: *defs.h:289*
6: *ClassFile.cpp:14*
7: *ClassFile.cpp:14*
8: *AttributeInfo.h:47*
9: *defs.h:290*
10: *AttributeInfo.h:55*
11: *defs.h:291*
12: *AttributeInfo.h:78*
13: *defs.h:292*
14: *AttributeInfo.h:87*
15: *defs.h:293*
16: *AttributeInfo.h:104*
17: *defs.h:294*
18: *AttributeInfo.h:110*
19: *defs.h:295*
20: *AttributeInfo.h:118*
21: *defs.h:296*
22: *AttributeInfo.h:133*
23: *defs.h:297*
24: *AttributeInfo.h:151*
25: *AttributeInfo.h:159*
26: *ClassFile.cpp:14*
27: *defs.h:33*
28: *ClassFile.h:30*
29: *memcpy_bigendian.cpp:10*
30: *ClassFile.h:37*
31: *ClassFile.cpp:42*
32: *defs.h:172*
33: *ClassFile.h:42*
34: *ClassFile.h:47*

```
53:
54:             memcpy_u2[1](&attributes_count[2], stream_pointer[3]);
55:             stream_pointer[3] += sizeof(u2[4]);
56:
57:             // Allocate attribute_info table
58:
59:             attributes[5] = new attribute_info[6]*[attributes_count[2]];
60:
61:             // Read attribute_info table
62:
63:             for (int i = 0; i < attributes_count[2] ; i++)
64:             {
65:                     attribute_info[7] * ai;
66:                     u2[4] attribute_name_index; // index of the Utf8 attribute name in the Constant Pool
67:
68:                     memcpy_u2[1](&attribute_name_index[8],stream_pointer[3]);
69:                     stream_pointer[3] += sizeof(u2[4]);
70:
71:                     CONSTANT_Utf8_info[9] * utf8Info = (CONSTANT_Utf8_info[9]*)class_file[10]->constant_pool[11]->get_it
        em_at_index[12](attribute_name_index[8]);
72:
73:                     wchar_t * attribute_name;
74:                     Result[13] result = utf8Info[14]->get_string[15](attribute_name[16]);
75:                     if (result[17] != Success[18])
76:                     {
77:                             print_error[19](result[17]);
78:                             return ClassLoaderCannotReadFieldMethodInfo[20];
79:                     }
80:
81:                     result[17] = ReadAttributeInfoEntry[21](attribute_name[16], utf8Info[14]->length[22], stream_pointer[3], a
        i[23]);
82:                     ai[23]->attribute_name_index[24] = attribute_name_index[8];
83:
84:                     if (result[17] != Success[18])
85:                     {
86:                             print_error[19](result[17]);
87:                             return ClassLoaderCannotReadFieldMethodInfo[20];
88:                     }
89:
90:                     ai[23]->set_class_file[25](class_file[10]);
91:                     ai[23]->set_parent[26](this);
92:                     attributes[5][i[27]] = ai[23];
93:
94:             }
95:
96:             return Success[18];
97:
98:     }
99:
100:    void field_method_info[28]::debug_print(ostream & out)
101:    {
102:            CONSTANT_Utf8_info[9] * name = (CONSTANT_Utf8_info[9]*)class_file[10]->constant_pool[11]->get_item_at_index[12]
        (name_index[29]);
```

Footnotes:

1: *memcpy_bigendian.cpp:10*
2: *ClassFile.h:50*
3: *ClassFile.cpp:42*
4: *defs.h:172*
5: *ClassFile.h:53*
6: *AttributeInfo.h:25*
7: *AttributeInfo.h:15*
8: *ClassFile.cpp:66*
9: *ConstantPool.h:186*
10: *ClassFile.h:56*
11: *ClassFile.h:159*
12: *ConstantPool.h:299*
13: *defs.h:29*
14: *ClassFile.cpp:71*
15: *ConstantPool.cpp:639*
16: *ClassFile.cpp:73*
17: *ClassFile.cpp:74*
18: *defs.h:33*
19: *errors.cpp:35*
20: *defs.h:46*
21: *ClassFile.cpp:14*
22: *ConstantPool.h:188*
23: *ClassFile.cpp:65*
24: *AttributeInfo.h:21*
25: *AttributeInfo.h:32*
26: *AttributeInfo.h:36*
27: *ClassFile.cpp:63*
28: *ClassFile.h:30*
29: *ClassFile.h:42*

```
103:                wchar_t * namestr;
104:                name[1]->get_string[2](namestr[3]);
105:                CONSTANT_Utf8_info[4] * descriptor = (CONSTANT_Utf8_info[4]*)class_file[5]->constant_pool[6]->get_item_at_i
        ndex[7](descriptor_index[8]);
106:                wchar_t * descstr;
107:                descriptor[9]->get_string[2](descstr[10]);
108:
109:                wchar_t * class_name;
110:                u2[11] length;
111:                class_file[5]->get_full_class_name[12](class_name[13], length[14]);
112:                out[15] << "   CLASS: "; print_wchar[16](class_name[13], length[14], out[15]); out[15] << endl;
113:                delete [] class_name[13];
114:
115:            out[15] << "   ACCESS FLAGS: " << access_flags[17] << endl;
116:            out[15] << "   NAME INDEX: " << name_index[18] << "("; print_wchar[16](namestr[3], name[1]->length[19], out[15]);
117:                out[15] << ")" << endl;
118:            out[15] << "   DESCRIPTOR INDEX: " << descriptor_index[8] << "("; print_wchar[16](descstr[10], descriptor[9]->lengt
        h[19], out[15]);
119:                out[15] << ")" << endl;
120:            out[15] << "   ATTRIBUTES COUNT: " << attributes_count[20] << endl;
121:                out[15] << "ATTRIBUTES:" << endl;
122:
123:                delete [] namestr[3];
124:                delete [] descstr[10];
125:
126:                for (int i = 0; i < attributes_count[20]; i++)
127:                {
128:                        attributes[21][i[22]]->debug_print(out[15]);
129:                }
130:        }
131:
132:        void field_info[23]::debug_print(ostream & out)
133:        {
134:                out[24] << "FIELD INFO:" << endl;
135:                field_method_info[25]::debug_print[26](out[24]);
136:                out[24] << "OFFSET:" << offset[27] << endl;
137:        }
138:
139:        void method_info[28]::debug_print(ostream & out)
140:        {
141:                out[29] << "METHOD INFO:" << endl;
142:                field_method_info[25]::debug_print[26](out[29]);
143:
144:                Code_attribute[30] * code_attribute = get_code_attribute[31]();
145:                if (! code_attribute[32])
146:                        return;
147:                out[29] << "EXCEPTION TABLE:" << endl;
148:                for (int i = 0; i < code_attribute[32]->exception_table_length[33]; i++)
149:                {       out[29] << "   Exception #" << i[34] << ":" << endl;
150:                        Code_attribute[30]::exception_table_entry[35] entry = code_attribute[32]->exception_table[36][i[34]];
151:                        out[29] << "      start_pc: " << entry[37].start_pc[38] << endl;
152:                        out[29] << "      end_pc: " << entry[37].end_pc[39] << endl;
153:                        out[29] << "      handler_pc: " << entry[37].handler_pc[40] << endl;
```

Footnotes:

1: *ClassFile.cpp:102*
2: *ConstantPool.cpp:639*
3: *ClassFile.cpp:103*
4: *ConstantPool.h:186*
5: *ClassFile.h:56*
6: *ClassFile.h:159*
7: *ConstantPool.h:299*
8: *ClassFile.h:47*
9: *ClassFile.cpp:105*
10: *ClassFile.cpp:106*
11: *defs.h:172*
12: *ClassFile.cpp:442*
13: *ClassFile.cpp:109*
14: *ClassFile.cpp:110*
15: *ClassFile.cpp:100*
16: *util.cpp:42*
17: *ClassFile.h:37*
18: *ClassFile.h:42*
19: *ConstantPool.h:188*
20: *ClassFile.h:50*
21: *ClassFile.h:53*
22: *ClassFile.cpp:126*
23: *ClassFile.h:68*
24: *ClassFile.cpp:132*
25: *ClassFile.h:30*
26: *ClassFile.cpp:100*
27: *ClassFile.h:79*
28: *ClassFile.h:92*
29: *ClassFile.cpp:139*
30: *AttributeInfo.h:55*
31: *ClassFile.cpp:230*
32: *ClassFile.cpp:144*
33: *AttributeInfo.h:69*
34: *ClassFile.cpp:148*
35: *AttributeInfo.h:57*
36: *AttributeInfo.h:70*
37: *ClassFile.cpp:150*
38: *AttributeInfo.h:59*
39: *AttributeInfo.h:60*
40: *AttributeInfo.h:61*

```
154:                    if (entry[1].catch_type[2] == 0)
155:                            continue;
156:                    CONSTANT_Class_info[3] * class_info =
157:                            (CONSTANT_Class_info[3]*)class_file[4]->constant_pool[5]->get_item_at_index[6](entry[1].catch_type[2]);
158:                    CONSTANT_Utf8_info[7] * utf8_info =
159:                            (CONSTANT_Utf8_info[7]*)class_file[4]->constant_pool[5]->get_item_at_index[6](class_info[8]->name_index[9]);
160:                    wchar_t * class_name;
161:                    utf8_info[10]->get_string[11](class_name[12]);
162:                    out[13] << "      catch_type: "; print_wchar[14](class_name[12], wcslen(class_name[12]), out[13]); out[13] << endl;
163:                    delete [] class_name[12];
164:            }
165:    }
166:
167:    // Note that method_name must be deleted after usage
168:    void method_info[15]::get_method_name(wchar_t *& method_name, u2[16] & method_name_length)
169:    {
170:            CONSTANT_Utf8_info[7] * utf8_name =
171:                    (CONSTANT_Utf8_info[7]*)class_file[4]->constant_pool[5]->get_item_at_index[6](name_index[17]);
172:            utf8_name[18]->get_string[11](method_name[19]);
173:            method_name_length[20] = utf8_name[18]->length[21];
174:    }
175:
176:    // Note that descriptor must be deleted after usage
177:    void method_info[15]::get_method_descriptor(wchar_t *& descriptor, u2[16] & descriptor_length)
178:    {
179:            CONSTANT_Utf8_info[7] * utf8_descriptor =
180:                    (CONSTANT_Utf8_info[7]*)class_file[4]->constant_pool[5]->get_item_at_index[6](descriptor_index[22]);
181:            utf8_descriptor[23]->get_string[11](descriptor[24]);
182:            descriptor_length[25] = utf8_descriptor[23]->length[21];
183:    }
184:
185:    int method_info[15]::is_equal_signature(method_info[15] * mi)
186:    {
187:            // Compare method names
188:            CONSTANT_Utf8_info[7] * my_utf8_name =
189:                    (CONSTANT_Utf8_info[7]*)class_file[4]->constant_pool[5]->get_item_at_index[6](name_index[17]);
190:            CONSTANT_Utf8_info[7] * his_utf8_name =
191:                    (CONSTANT_Utf8_info[7]*)mi[26]->class_file[4]->constant_pool[5]->get_item_at_index[6](mi[26]->name_index[17]);
192:
193:            if (*my_utf8_name[27] != *his_utf8_name[28])
194:                    return 0;
195:
196:            // Then, compare descriptors
197:            CONSTANT_Utf8_info[7] * my_utf8_descriptor =
198:                    (CONSTANT_Utf8_info[7]*)class_file[4]->constant_pool[5]->get_item_at_index[6](descriptor_index[22]);
199:            CONSTANT_Utf8_info[7] * his_utf8_descriptor =
200:                    (CONSTANT_Utf8_info[7]*)mi[26]->class_file[4]->constant_pool[5]->get_item_at_index[6](mi[26]->descriptor_index[22]);
201:
```

Footnotes:

1: *ClassFile.cpp:150*
2: *AttributeInfo.h:62*
3: *ConstantPool.h:58*
4: *ClassFile.h:56*
5: *ClassFile.h:159*
6: *ConstantPool.h:299*
7: *ConstantPool.h:186*
8: *ClassFile.cpp:156*
9: *ConstantPool.h:60*
10: *ClassFile.cpp:158*
11: *ConstantPool.cpp:639*
12: *ClassFile.cpp:160*
13: *ClassFile.cpp:139*
14: *util.cpp:42*
15: *ClassFile.h:92*
16: *defs.h:172*
17: *ClassFile.h:42*
18: *ClassFile.cpp:170*
19: *ClassFile.cpp:168*
20: *ClassFile.cpp:168*
21: *ConstantPool.h:188*
22: *ClassFile.h:47*
23: *ClassFile.cpp:179*
24: *ClassFile.cpp:177*
25: *ClassFile.cpp:177*
26: *ClassFile.cpp:185*
27: *ClassFile.cpp:188*
28: *ClassFile.cpp:190*

```
202:                    if (*my_utf8_descriptor[1] != *his_utf8_descriptor[2])
203:                            return 0;
204:
205:                    return 1; // the signatures are equal
206:            }
207:
208:
209:            int method_info[3]::is_initializer()
210:            {
211:                    CONSTANT_Utf8_info[4] * utf8_name =
212:                            (CONSTANT_Utf8_info[4]*)class_file[5]->constant_pool[6]->get_item_at_index[7](name_index[8]);
213:
214:                    wchar_t * name;
215:                    utf8_name[9]->get_string[10](name[11]);
216:
217:                    // It's enough to compare names only - no Java method can be called "<init>"
218:                    if (!wcscmp(name[11], INIT_METHOD_NAME[12]))
219:                    {
220:                            delete [] name[11];
221:                            return 1;
222:                    }
223:                    delete [] name[11];
224:                    return 0;
225:            }
226:
227:            // TODO: In better implementation we should keep the code attribute once it is read
228:            // by the initial loading process; but the number of attributes is so small
229:            // that this one-time search is actually insignificant
230:            Code_attribute[13] * method_info[3]::get_code_attribute()
231:            {
232:                    for (int i = 0; i < attributes_count[14]; i++)
233:                    {
234:                            CONSTANT_Utf8_info[4] * utf8_info =
235:                                    (CONSTANT_Utf8_info[4]*)class_file[5]->constant_pool[6]->get_item_at_index[7](attributes[15][i[16]
        ]->attribute_name_index[17]);
236:                            if (!utf8_info[18])
237:                                    continue; // Nevermind
238:
239:                            wchar_t * string;
240:                            Result[19] result = utf8_info[18]->get_string[10](string[20]);
241:                            if (result[21] != Success[22])
242:                                    continue; // Nevermind
243:
244:                            if (!wmemcmp(string[20],CODE_ATTRIBUTE_NAME[23],utf8_info[18]->length[24]))
245:                            {
246:                                    delete [] string[20];
247:                                    return (Code_attribute[13]*)attributes[15][i[16]];
248:                            }
249:                            delete [] string[20];
250:                    }
251:
252:                    // Code attribute is not found
253:                    return NULL;
```

Footnotes:

1: *ClassFile.cpp:197*
2: *ClassFile.cpp:199*
3: *ClassFile.h:92*
4: *ConstantPool.h:186*
5: *ClassFile.h:56*
6: *ClassFile.h:159*
7: *ConstantPool.h:299*
8: *ClassFile.h:42*
9: *ClassFile.cpp:211*
10: *ConstantPool.cpp:639*
11: *ClassFile.cpp:214*
12: *defs.h:258*
13: *AttributeInfo.h:55*
14: *ClassFile.h:50*
15: *ClassFile.h:53*
16: *ClassFile.cpp:232*
17: *AttributeInfo.h:21*
18: *ClassFile.cpp:234*
19: *defs.h:29*
20: *ClassFile.cpp:239*
21: *ClassFile.cpp:240*
22: *defs.h:33*
23: *defs.h:290*
24: *ConstantPool.h:188*

```
254:        }
255:
256:        void ClassFile[1]::debug_print(ostream & out)
257:        {
258:                int i;
259:
260:                out[2] << "MAGIC NUMBER: " << (unsigned int)magic[3] << endl;
261:            out[2] << "MINOR VERSION: " << minor_version[4] << endl;
262:            out[2] << "MAJOR VERSION: " << major_version[5] << endl;
263:            out[2] << "CP COUNT: " << constant_pool_count[6] << endl;
264:            constant_pool[7]->debug_print[8](out[2]); // number of items in CP = <constant_pool_count-1>
265:            out[2] << "ACCESS FLAGS: " << access_flags[9] << endl;
266:            out[2] << "THIS CLASS: " << this_class[10] << endl;
267:            out[2] << "SUPER CLASS: " << super_class[11] << endl;
268:            out[2] << "INTERFACES COUNT: " << interfaces_count[12] << endl;
269:
270:                for (i[13] = 0; i[13] < interfaces_count[12]; i[13]++)
271:                {
272:                        out[2] << "INTERACE #" << i[13] << ": " << interfaces[14][i[13]] << endl;
273:                }
274:
275:                for (i[13] = 0; i[13] < fields_count[15]; i[13]++)
276:                {
277:                        out[2] << "--------------------" << endl;
278:                        out[2] << "FIELD #" << i[13] << ":" << endl;
279:                        fields[16][i[13]]->debug_print[17](out[2]);
280:                }
281:
282:                for (i[13] = 0; i[13] < methods_count[18]; i[13]++)
283:                {
284:                        out[2] << "--------------------" << endl;
285:                        out[2] << "METHOD #" << i[13] << ":" << endl;
286:                        methods[19][i[13]]->debug_print[20](out[2]);
287:                }
288:
289:        /*
290:            u2 attributes_count;
291:            AttributeInfo * attributes; // array of length attributes_count
292:        */
293:        }
294:
295:        // This is the point from where the whole Class File is read
296:        Result[21] ClassFile[1]::read(u1[22] *& stream_pointer)
297:        {
298:                int i;
299:
300:                // Read Magic Number
301:
302:                memcpy_u4[23](&magic[3], stream_pointer[24]);
303:                if (magic[3] != MAGIC_NUMBER[25]) // Quick verification
304:                {
305:                        return ClassLoaderErrorWrongMagicNumber[26];
306:                }
```

Footnotes:

1: *ClassFile.h:136*
2: *ClassFile.cpp:256*
3: *ClassFile.h:143*
4: *ClassFile.h:147*
5: *ClassFile.h:148*
6: *ClassFile.h:153*
7: *ClassFile.h:159*
8: *ConstantPool.cpp:876*
9: *ClassFile.h:163*
10: *ClassFile.h:167*
11: *ClassFile.h:178*
12: *ClassFile.h:181*
13: *ClassFile.cpp:258*
14: *ClassFile.h:187*
15: *ClassFile.h:191*
16: *ClassFile.h:197*
17: *ClassFile.cpp:132*
18: *ClassFile.h:200*
19: *ClassFile.h:210*
20: *ClassFile.cpp:139*
21: *defs.h:29*
22: *defs.h:168*
23: *memcpy_bigendian.cpp:27*
24: *ClassFile.cpp:296*
25: *defs.h:250*
26: *defs.h:44*

```
307:                    stream_pointer[1] += sizeof(u4[2]);
308:
309:                    // Read Minor Version
310:
311:                    memcpy_u2[3](&minor_version[4], stream_pointer[1]);
312:                    stream_pointer[1] += sizeof(u2[5]);
313:
314:                    // Read Major Version
315:
316:                    memcpy_u2[3](&major_version[6], stream_pointer[1]);
317:                    stream_pointer[1] += sizeof(u2[5]);
318:
319:                    // Read Constant Pool Count
320:
321:                    memcpy_u2[3](&constant_pool_count[7], stream_pointer[1]);
322:                    stream_pointer[1] += sizeof(u2[5]);
323:
324:                    // Allocate Constant Pool
325:
326:                    constant_pool[8] = new ConstantPool[9](this, constant_pool_count[7] - 1);
327:
328:                    // Read Constant Pool
329:
330:                    Result[10] result = constant_pool[8]->read[11](stream_pointer[1]);
331:
332:                    if (result[12] != Success[13])
333:                            return result[12];
334:
335:                    // Read Access Flags
336:
337:                    memcpy_u2[3](&access_flags[14], stream_pointer[1]);
338:                    stream_pointer[1] += sizeof(u2[5]);
339:
340:                    // Read This Class
341:
342:                    memcpy_u2[3](&this_class[15], stream_pointer[1]);
343:                    stream_pointer[1] += sizeof(u2[5]);
344:
345:                    // Read Super Class
346:
347:                    memcpy_u2[3](&super_class[16], stream_pointer[1]);
348:                    stream_pointer[1] += sizeof(u2[5]);
349:
350:                    // Read Interfaces Count
351:
352:                    memcpy_u2[3](&interfaces_count[17], stream_pointer[1]);
353:                    stream_pointer[1] += sizeof(u2[5]);
354:
355:                    // Allocate interfaces array
356:
357:                    interfaces[18] = new u2[5][interfaces_count[17]];
358:
359:                    // Read interfaces
```

Footnotes:

1: *ClassFile.cpp:296*
2: *defs.h:174*
3: *memcpy_bigendian.cpp:10*
4: *ClassFile.h:147*
5: *defs.h:172*
6: *ClassFile.h:148*
7: *ClassFile.h:153*
8: *ClassFile.h:159*
9: *ConstantPool.h:290*
10: *defs.h:29*
11: *ConstantPool.cpp:846*
12: *ClassFile.cpp:330*
13: *defs.h:33*
14: *ClassFile.h:163*
15: *ClassFile.h:167*
16: *ClassFile.h:178*
17: *ClassFile.h:181*
18: *ClassFile.h:187*

```
360:
361:                    for (i[1] = 0; i[1] < interfaces_count[2]; i[1]++)
362:                    {
363:                            memcpy_u2[3](&interfaces[4][i[1]],stream_pointer[5]);
364:                            stream_pointer[5] += sizeof(u2[6]);
365:                    }
366:
367:                    // Read Fields count
368:
369:                    memcpy_u2[3](&fields_count[7], stream_pointer[5]);
370:                    stream_pointer[5] += sizeof(u2[6]);
371:
372:                    // Allocate Fields array
373:
374:                    fields[8] = new field_info[9]*[fields_count[7]];
375:
376:                    // Read Fields array
377:
378:                    for (i[1] = 0; i[1] < fields_count[7]; i[1]++)
379:                    {
380:                            fields[8][i[1]] = new field_info[9](this);
381:                            fields[8][i[1]]->read[10](stream_pointer[5]);
382:                    }
383:
384:                    // Read Methods count
385:
386:                    memcpy_u2[3](&methods_count[11], stream_pointer[5]);
387:                    stream_pointer[5] += sizeof(u2[6]);
388:
389:                    // Allocate Methods array
390:
391:                    methods[12] = new method_info[13]*[methods_count[11]];
392:
393:                    // Read Methods array
394:
395:                    for (i[1] = 0; i[1] < methods_count[11]; i[1]++)
396:                    {
397:                            methods[12][i[1]] = new method_info[13](this);
398:                            methods[12][i[1]]->read[10](stream_pointer[5]);
399:                    }
400:
401:                    // Read Attributes count
402:
403:                    memcpy_u2[3](&attributes_count[14], stream_pointer[5]);
404:                    stream_pointer[5] += sizeof(u2[6]);
405:
406:                    // Allocate attribute_info table
407:
408:                    attributes[15] = new attribute_info[16]*[attributes_count[14]];
409:
410:                    // Read attribute_info table
411:
412:                    for (i[1] = 0; i[1] < attributes_count[14] ; i[1]++)
```

Footnotes:

1: *ClassFile.cpp:298*
2: *ClassFile.h:181*
3: *memcpy_bigendian.cpp:10*
4: *ClassFile.h:187*
5: *ClassFile.cpp:296*
6: *defs.h:172*
7: *ClassFile.h:191*
8: *ClassFile.h:197*
9: *ClassFile.h:83*
10: *ClassFile.cpp:42*
11: *ClassFile.h:200*
12: *ClassFile.h:210*
13: *ClassFile.h:99*
14: *ClassFile.h:213*
15: *ClassFile.h:217*
16: *AttributeInfo.h:25*

```
413:                {
414:                        attribute_info[1] * ai;
415:                        u2[2] attribute_name_index; // index of the Utf8 attribute name in the Constant Pool
416:
417:                        memcpy_u2[3](&attribute_name_index[4],stream_pointer[5]);
418:                        stream_pointer[5] += sizeof(u2[2]);
419:
420:                        CONSTANT_Utf8_info[6] * utf8Info = (CONSTANT_Utf8_info[6]*)constant_pool[7]->get_item_at_index[8](a
        ttribute_name_index[4]);
421:
422:                        wchar_t * attribute_name;
423:                        Result[9] result = utf8Info[10]->get_string[11](attribute_name[12]);
424:                        if (result[13] != Success[14]) {
425:                                return result[13];
426:                        }
427:
428:                        result[13] = ReadAttributeInfoEntry[15](attribute_name[12], utf8Info[10]->length[16], stream_pointer[5], a
        i[17]);
429:        //              delete [] attribute_name;
430:
431:                        if (result[13] == Success[14])
432:                                return result[13];
433:
434:                        attributes[18][i[19]] = ai[17];
435:
436:                }
437:
438:                return Success[14];
439:        }
440:
441:        // Note, that class_name must be deleted after the usage
442:        void ClassFile[20]::get_full_class_name(wchar_t *& class_name, u2[2] & class_name_length)
443:        {
444:                u2[2] utf8_name_index = ((CONSTANT_Class_info[21]*)constant_pool[7]->get_item_at_index[8](this_class[22]))->na
        me_index[23];
445:                CONSTANT_Utf8_info[6] * utf8_info = (CONSTANT_Utf8_info[6]*)constant_pool[7]->get_item_at_index[8](utf8_nam
        e_index[24]);
446:                utf8_info[25]->get_string[11](class_name[26]);
447:                class_name_length[27] = utf8_info[25]->length[16];
448:        }
449:
450:
451:        // Returns the private, final, static, or initializer method
452:        // having the specified name and descriptor;
453:        // The method will be represented by a direct link to the method_info
454:        // of this method.
455:        // This function will be used by the invokespecial, invokestatic and invokeinterface
456:        // instructions
457:        Result[9] ClassFile[20]::get_method(CONSTANT_Utf8_info[6] * method_name,
458:                                                                CONSTANT_Utf8_info[6] * descriptor,
459:                                                               method_info[28] *& minfo)
460:        {
461:                // Check all methods
```

Footnotes:

1: *AttributeInfo.h:15*
2: *defs.h:172*
3: *memcpy_bigendian.cpp:10*
4: *ClassFile.cpp:415*
5: *ClassFile.cpp:296*
6: *ConstantPool.h:186*
7: *ClassFile.h:159*
8: *ConstantPool.h:299*
9: *defs.h:29*
10: *ClassFile.cpp:420*
11: *ConstantPool.cpp:639*
12: *ClassFile.cpp:422*
13: *ClassFile.cpp:423*
14: *defs.h:33*
15: *ClassFile.cpp:14*
16: *ConstantPool.h:188*
17: *ClassFile.cpp:414*
18: *ClassFile.h:217*
19: *ClassFile.cpp:298*
20: *ClassFile.h:136*
21: *ConstantPool.h:58*
22: *ClassFile.h:167*
23: *ConstantPool.h:60*
24: *ClassFile.cpp:444*
25: *ClassFile.cpp:445*
26: *ClassFile.cpp:442*
27: *ClassFile.cpp:442*
28: *ClassFile.h:92*

```
462:                for (int i = 0; i < methods_count[1]; i++)
463:                {
464:                        method_info[2] * mi = (method_info[2]*)methods[3][i[4]];
465:                        // Get method's name
466:                        CONSTANT_Utf8_info[5] * utf8_name =
467:                        (CONSTANT_Utf8_info[5]*)constant_pool[6]->get_item_at_index[7](mi[8]->name_index[9]);
468:                        if (!utf8_name[10])
469:                                continue; // Never mind, just continue
470:
471:                        if (*method_name[11] != *utf8_name[10])
472:                                continue;
473:                        else
474:                        {
475:                                // Now compare the descriptors
476:
477:                                // Get method's name
478:                                CONSTANT_Utf8_info[5] * utf8_descriptor =
479:                                (CONSTANT_Utf8_info[5]*)constant_pool[6]->get_item_at_index[7](mi[8]->descriptor_index[12]);
480:                                if (!utf8_descriptor[13])
481:                                        continue;
482:
483:                                if (*descriptor[14] != *utf8_descriptor[13])
484:                                        continue;
485:
486:                                // Method cannot be abstract
487:                                if (mi[8]->access_flags[15] & ACC_ABSTRACT[16])
488:                                        return VM_ERROR_AbstractMethodError[17];
489:
490:                                // We've got it !
491:                                minfo[18] = mi[8];
492:
493:                                return Success[19];
494:                        }
495:                }
496:
497:                if (super_class[20] == 0) // we are in java.lang.Object
498:                        return VM_ERROR_NoSuchMethodError[21];
499:
500:                // We have not found the method in the referenced class;
501:                // try to look into the superclass
502:                CONSTANT_Class_info[22] * super_class_entry = (CONSTANT_Class_info[22]*)constant_pool[6]->get_item_at_ind
          ex[7](super_class[20]);
503:                // NOTE: since the current class is already resolved, its superclass must be resolved too;
504:                assert(super_class_entry[23]->is_resolved[24]()); // should NEVER fail
505:
506:                // This call will recursively investigate ALL the superclasses
507:                Result[25] result = super_class_entry[23]->resolved_class_file[26]->get_method[27](method_name[11], descriptor[14],
           minfo[18]);
508:                if (result[28] == Success[19])
509:                        // We've found the method in the superclass; nothing more to do
510:                        return Success[19];
511:
512:
```

Footnotes:

1: *ClassFile.h:200*
2: *ClassFile.h:92*
3: *ClassFile.h:210*
4: *ClassFile.cpp:462*
5: *ConstantPool.h:186*
6: *ClassFile.h:159*
7: *ConstantPool.h:299*
8: *ClassFile.cpp:464*
9: *ClassFile.h:42*
10: *ClassFile.cpp:466*
11: *ClassFile.cpp:457*
12: *ClassFile.h:47*
13: *ClassFile.cpp:478*
14: *ClassFile.cpp:458*
15: *ClassFile.h:37*
16: *defs.h:358*
17: *defs.h:104*
18: *ClassFile.cpp:459*
19: *defs.h:33*
20: *ClassFile.h:178*
21: *defs.h:102*
22: *ConstantPool.h:58*
23: *ClassFile.cpp:502*
24: *ConstantPool.h:48*
25: *defs.h:29*
26: *ConstantPool.h:71*
27: *ClassFile.cpp:457*
28: *ClassFile.cpp:507*

```
513:                    // The method was not found anywhere
514:                    return VM_ERROR_NoSuchMethodError[1];
515:
516:                    return Success[2];
517:            }
518:
519:            // Returns the non-private, non-final, non-static, non-initializer method
520:            // having the specified name and descriptor;
521:            // The method will be represented by an offset in the method table of this class.
522:            // This function will be used by the invokevirtual instruction
523:            Result[3] ClassFile[4]::get_virtual_method(CONSTANT_Utf8_info[5] * method_name,
524:                                                       CONSTANT_Utf8_info[5] * descriptor,
525:                                                       int & moffset)
526:            {
527:                    // Check all methods in the method table
528:                    for (unsigned int offset = 0; offset < method_table_size[6]; offset++)
529:                    {
530:                            method_info[7] * mi = method_table[8][offset[9]];
531:
532:                            // Get method's name
533:                            // Note, that we are looking into the constant pool of the class file where
534:                            // this method is defined
535:                            CONSTANT_Utf8_info[5] * utf8_name =
536:                            (CONSTANT_Utf8_info[5]*)mi[10]->class_file[11]->constant_pool[12]->get_item_at_index[13](mi[10]->name_inde
            x[14]);
537:                            if (!utf8_name[15])
538:                                    continue; // Never mind, just continue
539:
540:                            if (*method_name[16] != *utf8_name[15])
541:                                    continue;
542:                            else
543:                            {
544:                                    // Now compare the descriptors
545:
546:                                    // Get descriptor
547:                                    // Note, that we are looking into the constant pool of the class file where
548:                                    // this method is defined
549:                                    CONSTANT_Utf8_info[5] * utf8_descriptor =
550:                                    (CONSTANT_Utf8_info[5]*)mi[10]->class_file[11]->constant_pool[12]->get_item_at_index[13](mi[10]->d
            escriptor_index[17]);
551:                                    if (!utf8_descriptor[18])
552:                                            continue;
553:
554:                                    if (*descriptor[19] != *utf8_descriptor[18])
555:                                            continue;
556:
557:                                    // We've got it !
558:                                    moffset[20] = offset[9];
559:
560:                                    return Success[2];
561:                            }
562:                    }
563:
```

Footnotes:

1: *defs.h:102*
2: *defs.h:33*
3: *defs.h:29*
4: *ClassFile.h:136*
5: *ConstantPool.h:186*
6: *ClassFile.h:232*
7: *ClassFile.h:92*
8: *ClassFile.h:231*
9: *ClassFile.cpp:528*
10: *ClassFile.cpp:530*
11: *ClassFile.h:56*
12: *ClassFile.h:159*
13: *ConstantPool.h:299*
14: *ClassFile.h:42*
15: *ClassFile.cpp:535*
16: *ClassFile.cpp:523*
17: *ClassFile.h:47*
18: *ClassFile.cpp:549*
19: *ClassFile.cpp:524*
20: *ClassFile.cpp:525*

```
564:                    // The method was not found anywhere
565:                    return VM_ERROR_NoSuchMethodError[1];
566:            }
567:
568:
569:            // Given method's name and descriptor the function will return the code attribute
570:            // for this method.
571:            // (This function is used to find "main" and "<clinit>" methods)
572:            Result[2] ClassFile[3]::get_method_by_name(wchar_t * method_name, wchar_t * descriptor,
573:                                                                                    Code_attribute[4]
                 *& code_attribute)
574:            {
575:                    // For all methods
576:                    for (int i = 0; i < methods_count[5]; i++)
577:                    {
578:                            method_info[6] * mi = (method_info[6]*)methods[7][i[8]];
579:                            // Get method's name
580:                            CONSTANT_Utf8_info[9] * utf8_name =
581:                            (CONSTANT_Utf8_info[9]*)constant_pool[10]->get_item_at_index[11](mi[12]->name_index[13]);
582:                            if (!utf8_name[14])
583:                                    continue; // Never mind, just continue
584:
585:                            wchar_t * string;
586:                            Result[2] result = utf8_name[14]->get_string[15](string[16]);
587:                            if (result[17] != Success[18])
588:                                    continue;
589:
590:                            if (!wcscmp(method_name[19], string[16]))
591:                            {
592:                                    delete [] string[16];
593:                                    // Now compare the descriptors
594:
595:                                    // Get descriptor
596:                                    CONSTANT_Utf8_info[9] * utf8_descriptor =
597:                                            (CONSTANT_Utf8_info[9]*)constant_pool[10]->get_item_at_index[11](mi[12]->descriptor_
            index[20]);
598:                                    if (!utf8_name[14])
599:                                            continue; // Never mind, just continue
600:
601:                                    Result[2] result = utf8_descriptor[21]->get_string[15](string[16]);
602:                                    if (result[22] != Success[18])
603:                                            continue;
604:
605:                                    if (!wcscmp(descriptor[23], string[16]))
606:                                    {
607:                                            delete [] string[16];
608:                                            // Descriptors also match - get method's code
609:                                            code_attribute[24] = mi[12]->get_code_attribute[25]();
610:                                            return Success[18];
611:                                    }
612:
613:                                    delete [] string[16];
614:                                    continue;
```

Footnotes:

1: *defs.h:102*
2: *defs.h:29*
3: *ClassFile.h:136*
4: *AttributeInfo.h:55*
5: *ClassFile.h:200*
6: *ClassFile.h:92*
7: *ClassFile.h:210*
8: *ClassFile.cpp:576*
9: *ConstantPool.h:186*
10: *ClassFile.h:159*
11: *ConstantPool.h:299*
12: *ClassFile.cpp:578*
13: *ClassFile.h:42*
14: *ClassFile.cpp:580*
15: *ConstantPool.cpp:639*
16: *ClassFile.cpp:585*
17: *ClassFile.cpp:586*
18: *defs.h:33*
19: *ClassFile.cpp:572*
20: *ClassFile.h:47*
21: *ClassFile.cpp:596*
22: *ClassFile.cpp:601*
23: *ClassFile.cpp:572*
24: *ClassFile.cpp:573*
25: *ClassFile.cpp:230*

```
615:                        }
616:
617:                        delete [] string[1];
618:
619:                } // for
620:
621:                return Failure[2];
622:        }
623:
624:        // Returns the field having the specified name and descriptor;
625:        // If the field is not found in this class the function will try to find it
626:        // recursively in all the superclasses of this class
627:        Result[3] ClassFile[4]::get_field(CONSTANT_Utf8_info[5] * field_name, CONSTANT_Utf8_info[5] * field_descriptor, fie
           ld_info[6] *& finfo)
628:        {
629:                // Check all fields
630:                for (int i = 0; i < fields_count[7]; i++)
631:                {
632:                        field_info[6] * fi = (field_info[6]*)fields[8][i[9]];
633:                        // Get field's name
634:                        CONSTANT_Utf8_info[5] * utf8_name =
635:                        (CONSTANT_Utf8_info[5]*)constant_pool[10]->get_item_at_index[11](fi[12]->name_index[13]);
636:                        if (!utf8_name[14])
637:                                continue; // Never mind, just continue
638:
639:                        if (*field_name[15] != *utf8_name[14])
640:                                continue;
641:                        else
642:                        {
643:                                // Now compare the descriptors
644:
645:                                // Get field's name
646:                                CONSTANT_Utf8_info[5] * utf8_descriptor =
647:                                (CONSTANT_Utf8_info[5]*)constant_pool[10]->get_item_at_index[11](fi[12]->descriptor_index[16]);
648:                                if (!utf8_descriptor[17])
649:                                        continue;
650:
651:                                if (*field_descriptor[18] != *utf8_descriptor[17])
652:                                        continue;
653:
654:                                // We've got it !
655:                                finfo[19] = fi[12];
656:
657:                                return Success[20];
658:                        }
659:                }
660:
661:                if (super_class[21] == 0) // we are in java.lang.Object
662:                        return VM_ERROR_NoSuchFieldError[22];
663:
664:                // Try to look into all the superinterfaces of the current class
665:
666:                // for all superinterfaces
```

Footnotes:

1: *ClassFile.cpp:585*
2: *defs.h:34*
3: *defs.h:29*
4: *ClassFile.h:136*
5: *ConstantPool.h:186*
6: *ClassFile.h:68*
7: *ClassFile.h:191*
8: *ClassFile.h:197*
9: *ClassFile.cpp:630*
10: *ClassFile.h:159*
11: *ConstantPool.h:299*
12: *ClassFile.cpp:632*
13: *ClassFile.h:42*
14: *ClassFile.cpp:634*
15: *ClassFile.cpp:627*
16: *ClassFile.h:47*
17: *ClassFile.cpp:646*
18: *ClassFile.cpp:627*
19: *ClassFile.cpp:627*
20: *defs.h:33*
21: *ClassFile.h:178*
22: *defs.h:103*

```
667:                for (i = 0; i < interfaces_count[1]; i++)
668:                {
669:                        CONSTANT_Class_info[2] * super_interface_entry = (CONSTANT_Class_info[2]*)constant_pool[3]->get_i
        tem_at_index[4](interfaces[5][i]);
670:                        // NOTE: since the current class is already resolved, all its superinterfaces must be res
        olved too;
671:                        assert(super_interface_entry[6]->is_resolved[7]()); // should NEVER happen
672:
673:                        // This call will recursively investigate ALL the superclasses
674:                        Result[8] result = super_interface_entry[6]->class_file[9]->get_field[10](field_name[11], field_descri
        ptor[12], finfo[13]);
675:                        if (result[14] == Success[15])
676:                                // We've found the field in the superinterface; nothing more to do
677:                                return Success[15];
678:                }
679:
680:                // We have not found the field in the referenced class and its interfaces;
681:                // try to look into the superclass
682:                CONSTANT_Class_info[2] * super_class_entry = (CONSTANT_Class_info[2]*)constant_pool[3]->get_item_at_index[4]
        (super_class[16]);
683:                // NOTE: since the current class is already resolved, its superclass must be resolved too;
684:                assert(super_class_entry[17]->is_resolved[7]()); // should NEVER fail
685:
686:                // This call will recursively investigate ALL the superclasses
687:                Result[8] result = super_class_entry[17]->class_file[9]->get_field[10](field_name[11], field_descriptor[12], finfo[13]
        );
688:                if (result[18] == Success[15])
689:                        // We've found the field in the superclass; nothing more to do
690:                        return Success[15];
691:
692:                // The field was not found anywhere
693:                return VM_ERROR_NoSuchFieldError[19];
694:
695:                return Success[15];
696:        }
697:
698:        Result[8] ClassFile[20]::get_field_by_name(wchar_t * field_name, wchar_t * descriptor,
699:                                                   field_info[21] *& finfo)
700:        {
701:                // Check all fields
702:                for (int i = 0; i < fields_count[22]; i++)
703:                {
704:                        field_info[21] * fi = (field_info[21]*)fields[23][i[24]];
705:
706:                        // Get field's name
707:                        CONSTANT_Utf8_info[25] * utf8_name =
708:                        (CONSTANT_Utf8_info[25]*)constant_pool[3]->get_item_at_index[4](fi[26]->name_index[27]);
709:                        if (!utf8_name[28])
710:                                continue; // Never mind, just continue
711:
712:                        wchar_t * wname;
713:                        utf8_name[28]->get_string[29](wname[30]);
714:
```

Footnotes:

1: *ClassFile.h:181*
2: *ConstantPool.h:58*
3: *ClassFile.h:159*
4: *ConstantPool.h:299*
5: *ClassFile.h:187*
6: *ClassFile.cpp:669*
7: *ConstantPool.h:48*
8: *defs.h:29*
9: *ConstantPool.h:31*
10: *ClassFile.cpp:627*
11: *ClassFile.cpp:627*
12: *ClassFile.cpp:627*
13: *ClassFile.cpp:627*
14: *ClassFile.cpp:674*
15: *defs.h:33*
16: *ClassFile.h:178*
17: *ClassFile.cpp:682*
18: *ClassFile.cpp:687*
19: *defs.h:103*
20: *ClassFile.h:136*
21: *ClassFile.h:68*
22: *ClassFile.h:191*
23: *ClassFile.h:197*
24: *ClassFile.cpp:702*
25: *ConstantPool.h:186*
26: *ClassFile.cpp:704*
27: *ClassFile.h:42*
28: *ClassFile.cpp:707*
29: *ConstantPool.cpp:639*
30: *ClassFile.cpp:712*

```
715:                        if (wcscmp(wname[1], field_name[2]))
716:                        {
717:                                delete [] wname[1];
718:                                continue;
719:                        }
720:                        else
721:                        {
722:                                // Now compare the descriptors
723:
724:                                // Get field's name
725:                                CONSTANT_Utf8_info[3] * utf8_descriptor =
726:                                (CONSTANT_Utf8_info[3]*)constant_pool[4]->get_item_at_index[5](fi[6]->descriptor_index[7]);
727:
728:                                wchar_t * wdescriptor;
729:                                utf8_descriptor[8]->get_string[9](wdescriptor[10]);
730:
731:                                if (wcscmp(wdescriptor[10],descriptor[11]))
732:                                {
733:                                        delete wdescriptor[10];
734:                                        continue;
735:                                }
736:
737:                                // We've got it !
738:                                finfo[12] = fi[6];
739:
740:                                delete [] wname[1];
741:                                delete [] wdescriptor[10];
742:                                return Success[13];
743:                        }
744:                }
745:
746:                if (super_class[14] == 0) // we are in java.lang.Object
747:                        return VM_ERROR_NoSuchFieldError[15];
748:
749:                // Try to look into all the superinterfaces of the current class
750:
751:                // for all superinterfaces
752:                for (i = 0; i < interfaces_count[16]; i++)
753:                {
754:                        CONSTANT_Class_info[17] * super_interface_entry = (CONSTANT_Class_info[17]*)constant_pool[4]->get
        _item_at_index[5](interfaces[18][i]);
755:                        // NOTE: since the current class is already resolved, all its superinterfaces must be res
        olved too;
756:                        assert(super_interface_entry[19]->is_resolved[20]()); // should NEVER happen
757:
758:                        // This call will recursively investigate ALL the superclasses
759:                        Result[21] result = super_interface_entry[19]->resolved_class_file[22]->get_field_by_name[23](field_n
        ame[2], descriptor[11], finfo[12]);
760:                        if (result[24] == Success[13])
761:                                // We've found the field in the superinterface; nothing more to do
762:                                return Success[13];
763:                }
764:
```

Footnotes:

1: *ClassFile.cpp:712*
2: *ClassFile.cpp:698*
3: *ConstantPool.h:186*
4: *ClassFile.h:159*
5: *ConstantPool.h:299*
6: *ClassFile.cpp:704*
7: *ClassFile.h:47*
8: *ClassFile.cpp:725*
9: *ConstantPool.cpp:639*
10: *ClassFile.cpp:728*
11: *ClassFile.cpp:698*
12: *ClassFile.cpp:699*
13: *defs.h:33*
14: *ClassFile.h:178*
15: *defs.h:103*
16: *ClassFile.h:181*
17: *ConstantPool.h:58*
18: *ClassFile.h:187*
19: *ClassFile.cpp:754*
20: *ConstantPool.h:48*
21: *defs.h:29*
22: *ConstantPool.h:71*
23: *ClassFile.cpp:698*
24: *ClassFile.cpp:759*

```
765:                    // We have not found the field in the referenced class and its interfaces;
766:                    // try to look into the superclass
767:                    CONSTANT_Class_info[1] * super_class_entry = (CONSTANT_Class_info[1]*)constant_pool[2]->get_item_at_index[3]
            (super_class[4]);
768:                    // NOTE: since the current class is already resolved, its superclass must be resolved too;
769:                    assert(super_class_entry[5]->is_resolved[6]()); // should NEVER fail
770:
771:                    // This call will recursively investigate ALL the superclasses
772:                    Result[7] result = super_class_entry[5]->resolved_class_file[8]->get_field_by_name[9](field_name[10], descript
            or[11], finfo[12]);
773:                    if (result[13] == Success[14])
774:                            // We've found the field in the superclass; nothing more to do
775:                            return Success[14];
776:
777:                    // The field was not found anywhere
778:                    return VM_ERROR_NoSuchFieldError[15];
779:
780:                    return Success[14];
781:            }
782:
783:
784:            ClassFile[16]::ClassFile(ClassLoader[17] * _class_loader) : class_loader[18](_class_loader),
785:                    class_data[19](NULL), method_table[20](NULL), method_table_size[21](0), class_instance[22](NULL)
786:            {
787:                    class_heap_manager[23] = class_loader[18]->get_jvm[24]()->get_class_heap_manager[25]();
788:                    instance_heap_manager[26] = class_loader[18]->get_jvm[24]()->get_instance_heap_manager[27]();
789:            }
790:
791:            ClassFile[16]::~ClassFile()
792:            {
793:                    if (class_data[19])
794:                            delete class_data[19];
795:            }
796:
797:
798:            // After JVM has loaded, verified and prepared the class, it is ready to be initialized,
799:            // i.e. its <clinit> method should be called.
800:            // Note, that the <clinit> method is never called by any bytecodes;
801:            // it can be only invoked by the JVM itself.
802:            // Note also, that the <clinit> method must be executed "in the middle" of the current
803:            // instruction (during the new type resolution process), i.e. before the current instruction
804:            // is finished
805:            Result[7] ClassFile[16]::initialize()
806:            {
807:                    // Get the <clinit> method
808:                    Code_attribute[28] * code_attribute = NULL;
809:                    Result[7] result = get_method_by_name[29](CLINIT_METHOD_NAME[30], CLINIT_METHOD_DESCRIPTOR[31], code_attribu
            te[32]);
810:                    if (result[33] == Failure[34])
811:                            return Success[14]; // there is just no <clinit> method in this class
812:
813:                    assert(code_attribute[32] != NULL);
814:
```

Footnotes:

1: *ConstantPool.h:58*
2: *ClassFile.h:159*
3: *ConstantPool.h:299*
4: *ClassFile.h:178*
5: *ClassFile.cpp:767*
6: *ConstantPool.h:48*
7: *defs.h:29*
8: *ConstantPool.h:71*
9: *ClassFile.cpp:698*
10: *ClassFile.cpp:698*
11: *ClassFile.cpp:698*
12: *ClassFile.cpp:699*
13: *ClassFile.cpp:772*
14: *defs.h:33*
15: *defs.h:103*
16: *ClassFile.h:136*
17: *ClassLoader.h:14*
18: *ClassFile.h:222*
19: *ClassFile.h:227*
20: *ClassFile.h:231*
21: *ClassFile.h:232*
22: *ClassFile.h:243*
23: *ClassFile.h:235*
24: *ClassLoader.h:98*
25: *VirtualMachine.h:331*
26: *ClassFile.h:237*
27: *VirtualMachine.h:332*
28: *AttributeInfo.h:55*
29: *ClassFile.cpp:572*
30: *defs.h:256*
31: *defs.h:257*
32: *ClassFile.cpp:808*
33: *ClassFile.cpp:809*
34: *defs.h:34*

```
815:                    // The main problem now is that we have to feed the JVM execution engine with the <clinit>
816:                    // method's code and execute it before current instruction is finished.
817:                    // For this purpose we will construct the special "fake" Thread with its own "execution engine"
818:                    // that will execute the <clinit>'s code together with all the possible methods that
819:                    // will be encountered during this method execution
820:                    VirtualMachine[1] * vm = this->class_loader[2]->get_jvm[3]();
821:                    Thread[4] * clinit_thread;
822:                    if (vm[5]->current_thread[6])
823:                            clinit_thread[7] = vm[5]->run_thread[8](code_attribute[9], vm[5]->current_thread[6]->thread_instance[10]);
824:                    else
825:                            clinit_thread[7] = vm[5]->run_thread[8](code_attribute[9], NULL);
826:
827:                    // This is our "tiny execution engine"
828:                    while (!clinit_thread[7]->is_dead[11]())
829:                    {
830:                            result[12] = clinit_thread[7]->step[13]();
831:                            if (result[12] != Success[14])
832:                            {
833:                                    print_error[15](result[12]);
834:                                    vm[5]->abnormal_termination[16]();
835:                                    //return ExecutionCannotExecuteStaticInitializer;
836:                            }
837:                    }
838:
839:                    return Success[14];
840:            }
841:
842:            /*
843:            Result ClassFile::initialize()
844:            {
845:                    // Get the <clinit> method
846:                    Code_attribute * code_attribute = NULL;
847:                    Result result = get_method_by_name(CLINIT_METHOD_NAME, CLINIT_METHOD_DESCRIPTOR, code_attribute);
848:                    if (result == Failure)
849:                            return Success; // there is just no <clinit> method in this class
850:
851:                    assert(code_attribute != NULL);
852:
853:                    // The main problem now is that we have to feed the JVM execution engine with the <clinit>
854:                    // method's code and execute it before current instruction is finished.
855:                    // For this purpose we will construct the special "fake" Thread with its own "execution engine"
856:                    // that will execute the <clinit>'s code together with all the possible methods that
857:                    // will be encountered during this method execution
858:                    VirtualMachine * vm = this->class_loader->get_jvm();
859:                    Thread * new_thread = vm->run_thread(code_attribute, vm->current_thread->thread_instance);
860:
861:                    vm->supercede(vm->current_thread, new_thread);
862:
863:                    return Success;
864:            }
865:            */
866:
867:
```

Footnotes:

1: *VirtualMachine.h:255*
2: *ClassFile.h:222*
3: *ClassLoader.h:98*
4: *Thread.h:17*
5: *ClassFile.cpp:820*
6: *VirtualMachine.h:287*
7: *ClassFile.cpp:821*
8: *VirtualMachine.cpp:287*
9: *ClassFile.cpp:808*
10: *Thread.h:162*
11: *Thread.h:172*
12: *ClassFile.cpp:809*
13: *Thread.cpp:907*
14: *defs.h:33*
15: *errors.cpp:35*
16: *VirtualMachine.cpp:143*

```
868:        // This will allocate a new class instance on the object heap;
869:        // The function returns new object's reference or null (0) in case of failure
870:        word[1] ClassFile[2]::create_new_instance()
871:        {
872:                InstanceData[3] * instance = new InstanceData[4](this);
873:                Result[5] result = instance[6]->create[7]();
874:                if (result[8] != Success[9])
875:                {
876:                        print_error[10](result[8]);
877:                        return ExecutionCannotCreateNewInstance[11];
878:                }
879:
880:                // Put the newely created instance into the Handle Pool and get
881:                // its abstract reference
882:                HandlePool[12] * hp = class_loader[13]->get_jvm[14]()->get_handle_pool[15]();
883:                word[1] reference = hp[16]->put_instance[17](instance[6]);
884:
885:        #ifdef DEBUG_INSTANCE_CREATION
886:                wchar_t * class_name;
887:                u2[18] length;
888:                get_full_class_name[19](class_name,length);
889:                debug_file << "CLASS: "; print_wchar[20](class_name,length,debug_file); debug_file <<  "; INSTANCE:
        " << (int)reference[21] << "; data start = " << instance[6]->data_start[22] << endl;
890:                delete [] class_name;
891:        #endif // DEBUG_INSTANCE_CREATION
892:
893:
894:                return reference[21];
895:        }
896:
897:        // Note, that the superclass' methods will be stored higher than the subclass' methods
898:        // (i.e. with lower offsets)
899:        Result[5] ClassFile[2]::get_class_methods(ClassFile[2] * cf, Vector[23]<method_info*> & methods)
900:        {
901:                if (cf[24]->super_class[25] > 0)
902:                {
903:                        // This is not the java.lang.Object class
904:
905:                        // Get this class' superclass
906:                        CONSTANT_Class_info[26] * super_class_entry =
907:                                (CONSTANT_Class_info[26]*)cf[24]->constant_pool[27]->get_item_at_index[28](cf[24]->super_class[25])
        ;
908:
909:                        // At this time the superclass must be already resolved
910:                        assert(super_class_entry[29]->resolved_class_file[30] != NULL);
911:                        // this is not the java.lang.Object class - get the superclass' methods recursively
912:                        // "me" is false, don't count the private fields
913:                        Result[5] result = get_class_methods[31](super_class_entry[29]->resolved_class_file[30], methods[32]);
914:                        if (result[33] != Success[9])
915:                        {
916:                                print_error[10](result[33]);
917:                                return PreparationCannotBuildMethodTable[34];
918:                        }
```

Footnotes:

1: *defs.h:195*
2: *ClassFile.h:136*
3: *ObjectData.h:113*
4: *ObjectData.h:127*
5: *defs.h:29*
6: *ClassFile.cpp:872*
7: *ObjectData.cpp:236*
8: *ClassFile.cpp:873*
9: *defs.h:33*
10: *errors.cpp:35*
11: *defs.h:74*
12: *HandlePool.h:13*
13: *ClassFile.h:222*
14: *ClassLoader.h:98*
15: *VirtualMachine.h:336*
16: *ClassFile.cpp:882*
17: *HandlePool.cpp:12*
18: *defs.h:172*
19: *ClassFile.cpp:442*
20: *util.cpp:42*
21: *ClassFile.cpp:883*
22: *ObjectData.h:71*
23: *vector.h:7*
24: *ClassFile.cpp:899*
25: *ClassFile.h:178*
26: *ConstantPool.h:58*
27: *ClassFile.h:159*
28: *ConstantPool.h:299*
29: *ClassFile.cpp:906*
30: *ConstantPool.h:71*
31: *ClassFile.cpp:899*
32: *ClassFile.cpp:899*
33: *ClassFile.cpp:913*
34: *defs.h:68*

```
919:                } // else - this is the java.lang.Object class - don't go any higher
920:
921:                // Now get all the non-private, non-static, non-final methods of this class
922:
923:                // For all methods of this class
924:                for (int i = 0; i < cf[1]->methods_count[2]; i++)
925:                {
926:                        method_info[3] * mi = (method_info[3]*)cf[1]->methods[4][i[5]];
927:
928:                        if ((mi[6]->access_flags[7] & ACC_STATIC[8]) ||
929:                            (mi[6]->access_flags[7] & ACC_PRIVATE[9]))
930:                        //      || (mi->access_flags & ACC_ABSTRACT))
931:                                continue; // do nothing
932:
933:                        if (mi[6]->is_initializer[10]())
934:                                continue; // do nothing
935:
936:                        // Now we have to check whether a method with the same signature is
937:                        // defined in the class that is higher in the class hierarchy;
938:                        // If so, it must be replaced by the method of the lower class
939:                        int replaced = 0;
940:                        for (int j = 0; j < methods[11].size[12](); j++)
941:                        {
942:                                method_info[3] * cur_mi = methods[11].element_at[13](j[14]);
943:                                if (mi[6]->is_equal_signature[15](cur_mi[16]))
944:                                {
945:                                        // replace method with the same signature
946:                                        // on the same offset with the new method_info
947:                                        methods[11].set_element_at[17](mi[6], j[14]);
948:                                        replaced[18] = 1;
949:                                }
950:                        }
951:
952:                        if (!replaced[18])
953:                                methods[11].add_element[19](mi[6]);
954:                }
955:
956:                return Success[20];
957:        }
958:
959:        // This function will build the method table of the class containing all the
960:        // non-private, non-static methods of this class and all its superclasses.
961:        // The method table will be used by the invokevirtual instruction (i.e. for
962:        // dynamically bound methods);
963:        // Note, that the offsets of the methods are always the same for all the classes
964:        // along the inheritance tree. This makes calling the proper virtual function easy.
965:        Result[21] ClassFile[22]::build_method_table()
966:        {
967:                // To accumulate the information about the methods
968:                Vector[23]<method_info*> methods;
969:
970:                // Go get my methods (and all of my superclasses' methods)
971:                Result[21] result = get_class_methods[24](this, methods[25]);
```

Footnotes:

1: *ClassFile.cpp:899*
2: *ClassFile.h:200*
3: *ClassFile.h:92*
4: *ClassFile.h:210*
5: *ClassFile.cpp:924*
6: *ClassFile.cpp:926*
7: *ClassFile.h:37*
8: *defs.h:364*
9: *defs.h:362*
10: *ClassFile.cpp:209*
11: *ClassFile.cpp:899*
12: *vector.h:23*
13: *vector.h:36*
14: *ClassFile.cpp:940*
15: *ClassFile.cpp:185*
16: *ClassFile.cpp:942*
17: *vector.h:45*
18: *ClassFile.cpp:939*
19: *vector.h:40*
20: *defs.h:33*
21: *defs.h:29*
22: *ClassFile.h:136*
23: *vector.h:17*
24: *ClassFile.cpp:899*
25: *ClassFile.cpp:968*

```
972:                    if (result[1] != Success[2])
973:                    {
974:                            print_error[3](result[1]);
975:                            return PreparationCannotBuildMethodTable[4];
976:                    }
977:
978:                    // Now we have collected all the class' and superclasses' virtual methods
979:                    // (To keep an array is more economic than a Vector)
980:                    method_table_size[5] = methods[6].size[7]();
981:                    method_table[8] = new method_info[9]*[method_table_size[5]];
982:                    for (int i = 0; i < method_table_size[5]; i++)
983:                    {
984:                            method_table[8][i[10]] = methods[6].element_at[11](i[10]);
985:                    }
986:
987:                    return Success[2];
988:            }
989:
990:
991:            // Allocates an array of reference values on the instance heap of the given length
992:            word[12] ArrayClassFile[13]::create_new_instance(u4[14] count)
993:            {
994:                    InstanceData[15] * instance = new ArrayInstanceData[16](this);
995:                    Result[17] result = ((ArrayInstanceData[18]*)instance[19])->create[20](count[21]);
996:                    if (result[22] != Success[2])
997:                    {
998:                            print_error[3](result[22]);
999:                            return ExecutionCannotCreateNewInstance[23];
1000:                   }
1001:
1002:                   // Put the newely created instance into the Handle Pool and get
1003:                   // its abstract reference
1004:                   HandlePool[24] * hp = class_loader[25]->get_jvm[26]()->get_handle_pool[27]();
1005:                   word[12] reference = hp[28]->put_instance[29](instance[19]);
1006:
1007:                   return reference[30];
1008:           }
1009:
1010:
1011:           // Allocates an array of primitive values on the instance heap of the given length
1012:           word[12] PrimitiveArrayClassFile[31]::create_new_instance(u4[14] count)
1013:           {
1014:                   InstanceData[15] * instance = new PrimitiveArrayInstanceData[32](this);
1015:                   Result[17] result = ((PrimitiveArrayInstanceData[33]*)instance[34])->create[35](count[36]);
1016:                   if (result[37] != Success[2])
1017:                   {
1018:                           print_error[3](result[37]);
1019:                           return ExecutionCannotCreateNewInstance[23];
1020:                   }
1021:
1022:                   // Put the newely created instance into the Handle Pool and get
1023:                   // its abstract reference
1024:                   HandlePool[24] * hp = class_loader[25]->get_jvm[26]()->get_handle_pool[27]();
```

Footnotes:

1: *ClassFile.cpp:971*
2: *defs.h:33*
3: *errors.cpp:35*
4: *defs.h:68*
5: *ClassFile.h:232*
6: *ClassFile.cpp:968*
7: *vector.h:23*
8: *ClassFile.h:231*
9: *ClassFile.h:99*
10: *ClassFile.cpp:982*
11: *vector.h:36*
12: *defs.h:195*
13: *ClassFile.h:341*
14: *defs.h:174*
15: *ObjectData.h:113*
16: *ObjectData.h:160*
17: *defs.h:29*
18: *ObjectData.h:155*
19: *ClassFile.cpp:994*
20: *ObjectData.cpp:303*
21: *ClassFile.cpp:992*
22: *ClassFile.cpp:995*
23: *defs.h:74*
24: *HandlePool.h:13*
25: *ClassFile.h:222*
26: *ClassLoader.h:98*
27: *VirtualMachine.h:336*
28: *ClassFile.cpp:1004*
29: *HandlePool.cpp:12*
30: *ClassFile.cpp:1005*
31: *ClassFile.h:370*
32: *ObjectData.h:177*
33: *ObjectData.h:174*
34: *ClassFile.cpp:1014*
35: *ObjectData.cpp:358*
36: *ClassFile.cpp:1012*
37: *ClassFile.cpp:1015*

```
1025:                   word[1] reference = hp[2]->put_instance[3](instance[4]);
1026:
1027:                   return reference[5];
1028:           }
1029:
1030:           int ClassFile[6]::is_Object()
1031:           {
1032:                   wchar_t * class_name;
1033:                   u2[7] length;
1034:                   get_full_class_name[8](class_name[9], length[10]);
1035:                   int result = !wcscmp(class_name[9], OBJECT_CLASS_NAME[11]);
1036:                   delete [] class_name[9];
1037:                   return result[12];
1038:           }
1039:
1040:           // Returns true if "this" class implements "that" class
1041:           int ClassFile[6]::implements(ClassFile[6] * that)
1042:           {
1043:                   if (this == that[13])
1044:                           return 1;
1045:
1046:                   // For all "this" superinterfaces
1047:                   for (int i = 0; i < interfaces_count[14]; i++)
1048:                   {
1049:                           CONSTANT_Class_info[15] * super_interface_entry = (CONSTANT_Class_info[15]*)constant_pool[16]->get
              _item_at_index[17](interfaces[18][i[19]]);
1050:                           assert(super_interface_entry[20] != NULL);
1051:                           assert(super_interface_entry[20]->is_resolved[21]());
1052:                           if (this == super_interface_entry[20]->resolved_class_file[22])
1053:                                   return 1;
1054:                           return super_interface_entry[20]->resolved_class_file[22]->implements[23](that[13]);
1055:                   }
1056:
1057:                   return 0;
1058:           }
1059:
1060:           // Returns true if "this" class inherits from "that" class
1061:           int ClassFile[6]::inherits(ClassFile[6] * that)
1062:           {
1063:                   if (this == that[24])
1064:                           return 1;
1065:
1066:                   if (super_class[25] == 0)
1067:                           return 0; // this is the Object class
1068:
1069:                   // Get "this" superclass
1070:                   CONSTANT_Class_info[15] * super_class_entry = (CONSTANT_Class_info[15]*)constant_pool[16]->get_item_at_ind
              ex[17](super_class[25]);
1071:                   assert(super_class_entry[26] != NULL);
1072:                   assert(super_class_entry[26]->is_resolved[21]());
1073:                   if (this == super_class_entry[26]->resolved_class_file[22])
1074:                           return 1;
1075:
```

Footnotes:

1: *defs.h:195*
2: *ClassFile.cpp:1024*
3: *HandlePool.cpp:12*
4: *ClassFile.cpp:1014*
5: *ClassFile.cpp:1025*
6: *ClassFile.h:136*
7: *defs.h:172*
8: *ClassFile.cpp:442*
9: *ClassFile.cpp:1032*
10: *ClassFile.cpp:1033*
11: *defs.h:252*
12: *ClassFile.cpp:1035*
13: *ClassFile.cpp:1041*
14: *ClassFile.h:181*
15: *ConstantPool.h:58*
16: *ClassFile.h:159*
17: *ConstantPool.h:299*
18: *ClassFile.h:187*
19: *ClassFile.cpp:1047*
20: *ClassFile.cpp:1049*
21: *ConstantPool.h:48*
22: *ConstantPool.h:71*
23: *ClassFile.cpp:1041*
24: *ClassFile.cpp:1061*
25: *ClassFile.h:178*
26: *ClassFile.cpp:1070*

```
1076:                   return super_class_entry[1]->resolved_class_file[2]->inherits[3](that[4]);
1077:           }
1078:
1079:           // This function determines whether an instance of "that_class" is an instance
1080:           // of "this" class according to the JVM rules
1081:           int ClassFile[5]::instance_of(ClassFile[5] * that)
1082:           {
1083:                   // First of all, return true if these are the same classes
1084:                   if (this == that[6])
1085:                           return 1;
1086:
1087:                   // Further, "this" class is denoted by S and "that" class is denoted by T
1088:
1089:                   // If S is an ordinary (nonarray) class, then:
1090:                   // If T is an interface type, then S must implement interface T.
1091:                   // If T is a class type, then S must be the same class as T or a subclass of T.
1092:
1093:                   if (!this->is_array[7]() && !(this->access_flags[8] & ACC_INTERFACE[9]))
1094:                   {
1095:                           if (that[6]->access_flags[8] & ACC_INTERFACE[9])
1096:                                   return this->implements[10](that[6]);
1097:                           else
1098:                                   return this->inherits[3](that[6]);
1099:                   }
1100:
1101:                   // If S is an interface type, then:
1102:                   // If T is an interface type, then T must be the same interface as S,
1103:                   // or a superinterface of S.
1104:                   // If T is a class type, then T must be Object.
1105:
1106:                   if (!this->is_array[7]() && (this->access_flags[8] & ACC_INTERFACE[9]))
1107:                   {
1108:                           if (that[6]->access_flags[8] & ACC_INTERFACE[9])
1109:                                   return this->inherits[3](that[6]);
1110:                           else
1111:                                   return that[6]->is_Object[11]();
1112:                   }
1113:
1114:                   // Now we know that S is an array
1115:
1116:                   return 0;
1117:           }
1118:
1119:
1120:           // Prepare the "fake" boot class and method for the JVM boot.
1121:           // (Note that the UTF8 string will use the special constructor to make the entry
1122:           // without reading it from the bytestream but rather making the UTF8 string given
1123:           // the wide-character string)
1124:           Result[12] ClassFile[5]::make_bootable(char * main_class_ascii_name, Code_attribute[13] *& code_attribute, Instan
           ceData[14] *& boot_instance)
1125:           {
1126:                   wchar_t * main_class_name = new wchar_t[strlen(main_class_ascii_name[15])+1];
1127:                   ascii2wchar[16](main_class_ascii_name[15], main_class_name[17]);
```

Footnotes:

1: *ClassFile.cpp:1070*
2: *ConstantPool.h:71*
3: *ClassFile.cpp:1061*
4: *ClassFile.cpp:1061*
5: *ClassFile.h:136*
6: *ClassFile.cpp:1081*
7: *ClassFile.h:336*
8: *ClassFile.h:163*
9: *defs.h:357*
10: *ClassFile.cpp:1041*
11: *ClassFile.cpp:1030*
12: *defs.h:29*
13: *AttributeInfo.h:55*
14: *ObjectData.h:113*
15: *ClassFile.cpp:1124*
16: *util.cpp:16*
17: *ClassFile.cpp:1126*

```
1128:
1129:                 /* Below is the layout of the artificially constructed "bootable" class file
1130:
1131:                 Bootstrap "Constant Pool"
1132:                 ------------------------
1133:
1134:                 #1 CONSTANT_Methodref_info:
1135:                    class_index #5 (main class)
1136:                    name_and_type_index #2 ("main" method)
1137:
1138:             #2 CONSTANT_NameAndType_info:
1139:                    name_index #3
1140:                    descriptor_index #4
1141:
1142:             #3 CONSTANT_Utf8_info:
1143:                    bytes: "main"
1144:                    length: length of "main"
1145:
1146:                 #4 CONSTANT_Utf8_info:
1147:                    bytes: "([Ljava/lang/String;)V"
1148:                    length: length of descriptor
1149:
1150:                 #5 CONSTANT_Class_info: // main class
1151:                    name_index #6
1152:
1153:                 #6 CONSTANT_Utf8_info:
1154:                    bytes: <main class name>
1155:                    length: length of <main class name>
1156:
1157:                 #7 CONSTANT_Class_info: // boot class
1158:                    name_index #8
1159:
1160:                 #8 CONSTANT_Utf8_info:
1161:                    bytes: <dummy boot class name>
1162:                    length: length of <boot class name>
1163:
1164:                 #9 CONSTANT_Methodref_info:
1165:                    class_index #7 (boot class)
1166:                    name_and_type_index #10 (boot method)
1167:
1168:             #10 CONSTANT_NameAndType_info:
1169:                    name_index #11
1170:                    descriptor_index #12
1171:
1172:             #11 CONSTANT_Utf8_info:
1173:                    bytes: "$__boot_method__$"
1174:                    length: length of name
1175:
1176:                 #12 CONSTANT_Utf8_info:
1177:                    bytes: "()V"
1178:                    length: length of descriptor
1179:
1180:                 #13 CONSTANT_Class_info: // java.lang.Thread class
```

```
1181:                    name_index #14
1182:
1183:                #14 CONSTANT_Utf8_info:
1184:                   bytes: <Thread class name>
1185:                   length: length of <Thread class name>
1186:
1187:                #15 CONSTANT_Methodref_info:
1188:                   class_index #19 (java.lang.System)
1189:                   name_and_type_index #16 (initializeSystemClass method)
1190:
1191:             #16 CONSTANT_NameAndType_info:
1192:                   name_index #17
1193:                   descriptor_index #18
1194:
1195:             #17 CONSTANT_Utf8_info:
1196:                   bytes: "initializeSystemClass"
1197:                   length: length of name
1198:
1199:                #18 CONSTANT_Utf8_info:
1200:                   bytes: "()V"
1201:                   length: length of descriptor
1202:
1203:                #19 CONSTANT_Class_info: // java.lang.System class
1204:                   name_index #20
1205:
1206:                #20 CONSTANT_Utf8_info:
1207:                   bytes: <System class name>
1208:                   length: length of <System class name>
1209:
1210:                ---------------------------------------
1211:
1212:           method_info:
1213:                access_flags: ACC_STATIC
1214:                name_index: #3
1215:                descriptor_index: #4
1216:                attributes_count: 1
1217:                    attributes:
1218:                       Code_attribute:
1219:                      max_stack: 0
1220:                      max_locals: 0
1221:                      code_length: 3
1222:                      code:
1223:                                    ACONST_NULL
1224:                                        INVOKESTATIC 0 1 // points to entry #1
1225:                                        RETURN
1226:                      exception_table_length: 0
1227:                      attributes_count: 0
1228:
1229:                    ---------------------------------------
1230:
1231:                */
1232:
1233:                // Construct the Constant Pool
```

```
1234:
1235:                   CONSTANT_Methodref_info[1] * entry1 = new CONSTANT_Methodref_info[2](this);
1236:                   entry1[3]->class_index[4] = 5;
1237:                   entry1[3]->name_and_type_index[5] = 2;
1238:
1239:                   CONSTANT_NameAndType_info[6] * entry2 = new CONSTANT_NameAndType_info[7](this);
1240:                   entry2[8]->name_index[9] = 3;
1241:                   entry2[8]->descriptor_index[10] = 4;
1242:
1243:                   CONSTANT_Utf8_info[11] * entry3 = new CONSTANT_Utf8_info[12](this, MAIN_METHOD_NAME[13]);
1244:
1245:                   CONSTANT_Utf8_info[11] * entry4 = new CONSTANT_Utf8_info[12](this, MAIN_METHOD_DESCRIPTOR[14]);
1246:
1247:                   CONSTANT_Class_info[15] * entry5 = new CONSTANT_Class_info[16](this);
1248:                   entry5[17]->name_index[18] = 6;
1249:
1250:                   CONSTANT_Utf8_info[11] * entry6 = new CONSTANT_Utf8_info[12](this, main_class_name[19]);
1251:
1252:                   CONSTANT_Class_info[15] * entry7 = new CONSTANT_Class_info[16](this);
1253:                   entry7[20]->name_index[18] = 8;
1254:
1255:                   CONSTANT_Utf8_info[11] * entry8 = new CONSTANT_Utf8_info[12](this, L"$__boot_class__$");
1256:
1257:                   CONSTANT_Methodref_info[1] * entry9 = new CONSTANT_Methodref_info[2](this);
1258:                   entry9[21]->class_index[4] = 7;
1259:                   entry9[21]->name_and_type_index[5] = 10;
1260:
1261:                   CONSTANT_NameAndType_info[6] * entry10 = new CONSTANT_NameAndType_info[7](this);
1262:                   entry10[22]->name_index[9] = 11;
1263:                   entry10[22]->descriptor_index[10] = 12;
1264:
1265:                   CONSTANT_Utf8_info[11] * entry11 = new CONSTANT_Utf8_info[12](this, L"$__boot_method__$");
1266:
1267:                   CONSTANT_Utf8_info[11] * entry12 = new CONSTANT_Utf8_info[12](this, L"()V");
1268:
1269:                   CONSTANT_Class_info[15] * entry13 = new CONSTANT_Class_info[16](this);
1270:                   entry13[23]->name_index[18] = 14;
1271:
1272:                   CONSTANT_Utf8_info[11] * entry14 = new CONSTANT_Utf8_info[12](this, L"java/lang/Thread");
1273:
1274:                   CONSTANT_Methodref_info[1] * entry15 = new CONSTANT_Methodref_info[2](this);
1275:                   entry15[24]->class_index[4] = 19;
1276:                   entry15[24]->name_and_type_index[5] = 16;
1277:
1278:                   CONSTANT_NameAndType_info[6] * entry16 = new CONSTANT_NameAndType_info[7](this);
1279:                   entry16[25]->name_index[9] = 17;
1280:                   entry16[25]->descriptor_index[10] = 18;
1281:
1282:                   CONSTANT_Utf8_info[11] * entry17 = new CONSTANT_Utf8_info[12](this, L"initializeSystemClass");
1283:
1284:                   CONSTANT_Utf8_info[11] * entry18 = new CONSTANT_Utf8_info[12](this, L"()V");
1285:
1286:                   CONSTANT_Class_info[15] * entry19 = new CONSTANT_Class_info[16](this);
```

Footnotes:

1: *ConstantPool.h:98*
2: *ConstantPool.h:107*
3: *ClassFile.cpp:1235*
4: *ConstantPool.h:102*
5: *ConstantPool.h:104*
6: *ConstantPool.h:258*
7: *ConstantPool.h:264*
8: *ClassFile.cpp:1239*
9: *ConstantPool.h:260*
10: *ConstantPool.h:261*
11: *ConstantPool.h:186*
12: *ConstantPool.h:194*
13: *defs.h:260*
14: *defs.h:261*
15: *ConstantPool.h:58*
16: *ConstantPool.h:63*
17: *ClassFile.cpp:1247*
18: *ConstantPool.h:60*
19: *ClassFile.cpp:1126*
20: *ClassFile.cpp:1252*
21: *ClassFile.cpp:1257*
22: *ClassFile.cpp:1261*
23: *ClassFile.cpp:1269*
24: *ClassFile.cpp:1274*
25: *ClassFile.cpp:1278*

```
1287:                 entry19[1]->name_index[2] = 20;
1288:
1289:                 CONSTANT_Utf8_info[3] * entry20 = new CONSTANT_Utf8_info[4](this, L"java/lang/System");
1290:
1291:                 ConstantPool[5] * cp = new ConstantPool[6](this, 20);
1292:                 cp[7]->table[8][1] = entry1[9];
1293:                 cp[7]->table[8][2] = entry2[10];
1294:                 cp[7]->table[8][3] = entry3[11];
1295:                 cp[7]->table[8][4] = entry4[12];
1296:                 cp[7]->table[8][5] = entry5[13];
1297:                 cp[7]->table[8][6] = entry6[14];
1298:                 cp[7]->table[8][7] = entry7[15];
1299:                 cp[7]->table[8][8] = entry8[16];
1300:                 cp[7]->table[8][9] = entry9[17];
1301:                 cp[7]->table[8][10] = entry10[18];
1302:                 cp[7]->table[8][11] = entry11[19];
1303:                 cp[7]->table[8][12] = entry12[20];
1304:                 cp[7]->table[8][13] = entry13[21];
1305:                 cp[7]->table[8][14] = entry14[22];
1306:                 cp[7]->table[8][15] = entry15[23];
1307:                 cp[7]->table[8][16] = entry16[24];
1308:                 cp[7]->table[8][17] = entry17[25];
1309:                 cp[7]->table[8][18] = entry18[26];
1310:                 cp[7]->table[8][19] = entry19[1];
1311:                 cp[7]->table[8][20] = entry20[27];
1312:
1313:             // Construct code attribute
1314:
1315:                 code_attribute[28] = new Code_attribute[29];
1316:                 code_attribute[28]->class_file[30] = this;
1317:                 code_attribute[28]->max_stack[31] = 1; // for the reference to String[] args
1318:                 code_attribute[28]->max_locals[32] = 0;
1319:                 code_attribute[28]->code_length[33] = 8;
1320:                 code_attribute[28]->exception_table_length[34] = 0;
1321:                 code_attribute[28]->attributes_count[35] = 0;
1322:                 code_attribute[28]->code[36] = new u1[37][code_attribute[28]->code_length[33]];
1323:
1324:                 // Construct code
1325:
1326:
1327:                 code_attribute[28]->code[36][0] = INVOKESTATIC[38]; // call the "initializeSystemClass" method
1328:                 code_attribute[28]->code[36][1] = 0;
1329:                 code_attribute[28]->code[36][2] = 15; // constant pool entry #15
1330:                 code_attribute[28]->code[36][3] = ACONST_NULL[39]; // tmp - push null reference to String[] args
1331:                 code_attribute[28]->code[36][4] = INVOKESTATIC[38]; // call the "main" method
1332:                 code_attribute[28]->code[36][5] = 0;
1333:                 code_attribute[28]->code[36][6] = 1; // constant pool entry #1
1334:                 code_attribute[28]->code[36][7] = RETURN[40]; // "return" from the JVM run
1335:
1336:         /*
1337:                 code_attribute->code[0] = ACONST_NULL; // tmp - push null reference to String[] args
1338:                 code_attribute->code[1] = INVOKESTATIC; // call the "main" method
1339:                 code_attribute->code[2] = 0;
```

Footnotes:

1: *ClassFile.cpp:1286*
2: *ConstantPool.h:60*
3: *ConstantPool.h:186*
4: *ConstantPool.h:194*
5: *ConstantPool.h:274*
6: *ConstantPool.h:290*
7: *ClassFile.cpp:1291*
8: *ConstantPool.h:284*
9: *ClassFile.cpp:1235*
10: *ClassFile.cpp:1239*
11: *ClassFile.cpp:1243*
12: *ClassFile.cpp:1245*
13: *ClassFile.cpp:1247*
14: *ClassFile.cpp:1250*
15: *ClassFile.cpp:1252*
16: *ClassFile.cpp:1255*
17: *ClassFile.cpp:1257*
18: *ClassFile.cpp:1261*
19: *ClassFile.cpp:1265*
20: *ClassFile.cpp:1267*
21: *ClassFile.cpp:1269*
22: *ClassFile.cpp:1272*
23: *ClassFile.cpp:1274*
24: *ClassFile.cpp:1278*
25: *ClassFile.cpp:1282*
26: *ClassFile.cpp:1284*
27: *ClassFile.cpp:1289*
28: *ClassFile.cpp:1124*
29: *AttributeInfo.h:55*
30: *AttributeInfo.h:43*
31: *AttributeInfo.h:65*
32: *AttributeInfo.h:66*
33: *AttributeInfo.h:67*
34: *AttributeInfo.h:69*
35: *AttributeInfo.h:71*
36: *AttributeInfo.h:68*
37: *defs.h:168*
38: *opcodes.h:224*
39: *opcodes.h:41*
40: *opcodes.h:217*

```
1340:                 code_attribute->code[3] = 1; // constant pool entry #1
1341:                 code_attribute->code[4] = RETURN; // "return" from the JVM run
1342:         */
1343:
1344:                 // Construct the "main" method info
1345:
1346:                 method_info[1] * minfo = new method_info[2](this);
1347:                 minfo[3]->access_flags[4] = ACC_STATIC[5];
1348:                 minfo[3]->name_index[6] = 3;
1349:                 minfo[3]->descriptor_index[7] = 4;
1350:             minfo[3]->attributes_count[8] = 1;
1351:                 minfo[3]->attributes[9] = new attribute_info[10]*[1];
1352:                 minfo[3]->attributes[9][0] = code_attribute[11];
1353:
1354:                 code_attribute[11]->field_method_parent[12] = minfo[3];
1355:
1356:                 // Construct the class file
1357:
1358:                 this->constant_pool[13] = cp[14];
1359:                 this->this_class[15] = 7;
1360:                 this->fields_count[16] = 0;
1361:                 this->methods_count[17] = 1;
1362:                 this->methods[18] = new method_info[2]*[1];
1363:                 this->methods[18][0] = minfo[3];
1364:                 this->super_class[19] = 13; // the boot class is inherited from the Thread class
1365:                 this->interfaces_count[20] = 0;
1366:
1367:
1368:                 // After we have created the class file for the bootstrap class this is the time
1369:                 // to "define" it in terms of the class loader, to create an instance of this class
1370:                 // and to do some additional set up
1371:
1372:                 // This is the special case of the "define_class" function of the class loader,
1373:                 // since our artificial boot class is not read from the bytestream but rather
1374:                 // assembled manually
1375:                 Result[21] result = class_loader[22]->define_boot_class[23](this);
1376:                 if (result[24] != Success[25])
1377:                 {
1378:                         print_error[26](result[24]);
1379:                         return BootableClassCannotCreate[27];
1380:                 }
1381:
1382:                 // Create an instance of the bootstrap class
1383:                 word[28] boot_objectref = create_new_instance[29]();
1384:                 boot_instance[30] = class_loader[22]->get_jvm[31]()->get_handle_pool[32]()->get_instance[33](boot_objectref[34]);
1385:
1386:                 // Prepare the system ThreadGroup and assign it to the bootstrap thread's "group" field
1387:                 ClassFile[35] * system_thread_group_cf =
1388:                         class_loader[22]->load_class[36](THREAD_GROUP_CLASS_NAME[37],wcslen(THREAD_GROUP_CLASS_NAME[37]));
1389:                 if (system_thread_group_cf[38] == NULL)
1390:                         return BootableClassCannotCreate[27];
1391:                 word[28] system_thread_group_objectref = system_thread_group_cf[38]->create_new_instance[29]();
1392:
```

Footnotes:

1: *ClassFile.h:92*
2: *ClassFile.h:99*
3: *ClassFile.cpp:1346*
4: *ClassFile.h:37*
5: *defs.h:364*
6: *ClassFile.h:42*
7: *ClassFile.h:47*
8: *ClassFile.h:50*
9: *ClassFile.h:53*
10: *AttributeInfo.h:25*
11: *ClassFile.cpp:1124*
12: *AttributeInfo.h:44*
13: *ClassFile.h:159*
14: *ClassFile.cpp:1291*
15: *ClassFile.h:167*
16: *ClassFile.h:191*
17: *ClassFile.h:200*
18: *ClassFile.h:210*
19: *ClassFile.h:178*
20: *ClassFile.h:181*
21: *defs.h:29*
22: *ClassFile.h:222*
23: *ClassLoader.cpp:468*
24: *ClassFile.cpp:1375*
25: *defs.h:33*
26: *errors.cpp:35*
27: *defs.h:38*
28: *defs.h:195*
29: *ClassFile.cpp:870*
30: *ClassFile.cpp:1124*
31: *ClassLoader.h:98*
32: *VirtualMachine.h:336*
33: *HandlePool.cpp:25*
34: *ClassFile.cpp:1383*
35: *ClassFile.h:136*
36: *ClassLoader.cpp:18*
37: *defs.h:262*
38: *ClassFile.cpp:1387*

```
1393:                   field_info[1] * finfo = NULL;
1394:                   result[2] = get_field_by_name[3](THREAD_GROUP_GROUP_NAME[4], THREAD_GROUP_GROUP_DESCRIPTOR[5], finfo[6]);
1395:                   if (result[2] != Success[7])
1396:                   {
1397:                           print_error[8](result[2]);
1398:                           return BootableClassCannotCreate[9];
1399:                   }
1400:                   assert(finfo[6] != NULL);
1401:                   memcpy(boot_instance[10]->data_start[11]->address[12] + finfo[6]->offset[13], &system_thread_group_objectref[14],
          sizeof(word[15]));
1402:
1403:                   // Assign the priority of the bootstrap thread to be the NORM_PRIORITY
1404:                   su4[16] norm_priority = NORM_PRIORITY[17];
1405:                   result[2] = get_field_by_name[3](THREAD_PRIORITY_NAME[18], THREAD_PRIORITY_DESCRIPTOR[19], finfo[6]);
1406:                   if (result[2] != Success[7])
1407:                   {
1408:                           print_error[8](result[2]);
1409:                           return BootableClassCannotCreate[9];
1410:                   }
1411:                   assert(finfo[6] != NULL);
1412:                   memcpy(boot_instance[10]->data_start[11]->address[12] + finfo[6]->offset[13], &norm_priority[20], sizeof(su4[16]));
1413:
1414:
1415:                   return Success[7];
1416:           }
```

Footnotes:

1: *ClassFile.h:68*
2: *ClassFile.cpp:1375*
3: *ClassFile.cpp:698*
4: *defs.h:274*
5: *defs.h:275*
6: *ClassFile.cpp:1393*
7: *defs.h:33*
8: *errors.cpp:35*
9: *defs.h:38*
10: *ClassFile.cpp:1124*
11: *ObjectData.h:71*
12: *HeapManager.h:46*
13: *ClassFile.h:79*
14: *ClassFile.cpp:1391*
15: *defs.h:195*
16: *defs.h:175*
17: *defs.h:399*
18: *defs.h:276*
19: *defs.h:277*
20: *ClassFile.cpp:1404*

```
 1:
 2:          #include <stdlib.h>
 3:          #include <iostream.h>
 4:
 5:          #include "defs.h"
 6:          #include "ClassFile.h"
 7:          #include "ClassLoader.h"
 8:          #include "ObjectData.h"
 9:          #include "HandlePool.h"
10:
11:
12:          ClassLoader[1]::~ClassLoader()
13:          {
14:                  // TODO: Iterate through the hashtable and delete all the class files loaded
15:                  // by this class loader
16:          }
17:
18:          ClassFile[2] * ClassLoader[1]::load_class(wchar_t * class_name, unsigned int class_name_length)
19:          {
20:                  ClassFile[2] * class_file;
21:
22:                  // First ask the parent class loader to load the class
23:                  // (if this one is not the bootstrap class loader)
24:                  if (!parent_class_loader[3])
25:                  {
26:                          // This is the bootstrap class loader, so try to load the class
27:                          u1[4] * buffer;
28:                          u2[5] length;
29:
30:                          char ascii_class_name[MAX_FILE_PATH_NAME[6]];
31:                          wchar2ascii[7](class_name[8], ascii_class_name[9], class_name_length[10]);
32:
33:                          Result[11] result = open_file[12](ascii_class_name[9], buffer[13], length[14]);
34:                          if (result[15] != Success[16])
35:                          {
36:                                  print_error[17](result[15]);
37:                                  return NULL;
38:                          }
39:
40:                          // The define_class() function will recursively load all the
41:                          // superclasses of this class
42:                          result[15] = define_class[18](buffer[13], length[14], class_file[19]);
43:                          if (result[15] != Success[16])
44:                          {
45:                                  print_error[17](result[15]);
46:                                  return NULL;
47:                          }
48:
49:                          // at this point class_file should be completely initialized (read into memory)
50:                          delete [] buffer[13];
51:
52:                          return class_file[19];
53:
```

Footnotes:

1: *ClassLoader.h:14*
2: *ClassFile.h:136*
3: *ClassLoader.h:39*
4: *defs.h:168*
5: *defs.h:172*
6: *defs.h:203*
7: *util.cpp:8*
8: *ClassLoader.cpp:18*
9: *ClassLoader.cpp:30*
10: *ClassLoader.cpp:18*
11: *defs.h:29*
12: *ClassLoader.cpp:247*
13: *ClassLoader.cpp:27*
14: *ClassLoader.cpp:28*
15: *ClassLoader.cpp:33*
16: *defs.h:33*
17: *errors.cpp:35*
18: *ClassLoader.cpp:70*
19: *ClassLoader.cpp:20*

```
54:                     }
55:                     else // this is the user-defined class loader. Ask parent class to load the class file
56:                     {
57:                             class_file[1] = parent_class_loader[2]->load_class[3](class_name[4], class_name_length[5]);
58:                             // The information about the actual "defining class loader"
59:                             // is stored in the class_file
60:                             return class_file[1];
61:                     }
62:             }
63:
64:             /*
65:             * NOTE: size is of type u2 (as required by JVM limits)
66:             * TODO: the pass-one verification is needed. Ensure that the desired type name
67:             * is actually declared as the name of the type in the passed array
68:             */
69:
70:             Result[6] ClassLoader[7]::define_class(u1[8] * stream, u2[9] size, ClassFile[10] *& cf)
71:             {
72:                     // This class loader will be THE class loader of the class file being loaded
73:                     cf[11] = new ClassFile[12](this);
74:
75:                     u1[8] * stream_pointer = stream[13];
76:
77:                     Result[6] result = cf[11]->read[14](stream_pointer[15]);
78:                     if (result[16] != Success[17])
79:                     {
80:                             print_error[18](result[16]);
81:                             return ClassLoaderCannotLoadClass[19];
82:                     }
83:
84:                     // The class was successfully loaded. Put it in this class loader's namespace
85:                     // (do not delete the class_name because it will be used as a key in the hashtable)
86:                     wchar_t * class_name;
87:                     u2[9] class_name_length;
88:                     cf[11]->get_full_class_name[20](class_name[21], class_name_length[22]);
89:
90:                     // Meanwhile, the key of the hashtable will be the ASCII value of the string
91:                     char * ascii_class_name = new char[class_name_length[22]];
92:                     wchar2ascii[23](class_name[21], ascii_class_name[24], class_name_length[22]);
93:                     this_namespace[25].insert[26](ascii_class_name[24], cf[11]);
94:                     // Do not delete the key of the hashtable !
95:
96:                     // Once the given class is loaded it is time to load its superclass
97:                     // (and then superclass' superclass and so on until the java.lang.Object class)
98:                     // In this recursive process the superclass (and all the higher superclasses)
99:                     // will be defined in this class' class loader namespace
100:                    // But, prior to do it, check if this class is not java.lang.Object
101:
102:            #ifdef DEBUG_CLASS_LOADING
103:                    debug_file << "====================================================" << endl;
104:                    debug_file << "DEFINED: "; print_wchar[27](class_name[21], class_name_length[22], debug_file);
105:                    debug_file << endl;
106:                    cf[11]->debug_print[28](debug_file);
```

Footnotes:

1: *ClassLoader.cpp:20*
2: *ClassLoader.h:39*
3: *ClassLoader.cpp:18*
4: *ClassLoader.cpp:18*
5: *ClassLoader.cpp:18*
6: *defs.h:29*
7: *ClassLoader.h:14*
8: *defs.h:168*
9: *defs.h:172*
10: *ClassFile.h:136*
11: *ClassLoader.cpp:70*
12: *ClassFile.cpp:784*
13: *ClassLoader.cpp:70*
14: *ClassFile.cpp:296*
15: *ClassLoader.cpp:75*
16: *ClassLoader.cpp:77*
17: *defs.h:33*
18: *errors.cpp:35*
19: *defs.h:43*
20: *ClassFile.cpp:442*
21: *ClassLoader.cpp:86*
22: *ClassLoader.cpp:87*
23: *util.cpp:8*
24: *ClassLoader.cpp:91*
25: *ClassLoader.h:34*
26: *hashtable.cpp:23*
27: *util.cpp:42*
28: *ClassFile.cpp:256*

```
107:        #endif
108:
109:                if (wmemcmp(class_name[1], OBJECT_CLASS_NAME[2], class_name_length[3]))
110:                {
111:                        result[4] = load_superclass[5](cf[6]);
112:                        if (result[4] != Success[7])
113:                        {
114:                                print_error[8](result[4]);
115:                                return ClassLoaderCannotLoadSuperclass[9];
116:                        }
117:
118:                        // Now, it's time to load all the superinterfaces of this type
119:                        // As with the superclass, all superinterfaces will be recursively resolved
120:
121:                        result[4] = load_superinterfaces[10](cf[6]);
122:                        if (result[4] != Success[7])
123:                        {
124:                                print_error[8](result[4]);
125:                                return ClassLoaderCannotLoadSuperinterfaces[11];
126:                        }
127:
128:                }
129:
130:                // Last step of the loading process
131:                //--------------------------------------------------------------------------------
132:                // Instantiate the java.lang.Class class that will be associated with this class'
133:                // internal structures
134:                if (!vm[12]->boot_process[13]) // it is impossible during the boot process
135:                {
136:                        ClassFile[14] * class_class = vm[12]->bootstrap_class_loader[15]->get_class[16](CLASS_CLASS_NAME[17], wc
        slen(CLASS_CLASS_NAME[17]));
137:                        if (!class_class[18])
138:                                class_class[18] = vm[12]->bootstrap_class_loader[15]->load_class[19](CLASS_CLASS_NAME[17], wcsle
        n(CLASS_CLASS_NAME[17]));
139:                        assert(class_class[18] != NULL);
140:                        word[20] class_class_ref = class_class[18]->create_new_instance[21]();
141:                        cf[6]->class_instance[22] = vm[12]->get_handle_pool[23]()->get_instance[24](class_class_ref[25]);
142:                        // TODO: assign the ClassFile to the java.lang.Class special field
143:                }
144:
145:                // Step 2a - verify the type
146:                //--------------------------------------------------------------------------------
147:
148:                // Auxiliary step
149:                // Build the method table of the class
150:                result[4] = cf[6]->build_method_table[26]();
151:                if (result[4] != Success[7])
152:                {
153:                        print_error[8](result[4]);
154:                        return ClassLoaderCannotPrepareClass[27];
155:                }
156:
157:                // Step 2b - prepare the type
```

Footnotes:

1: *ClassLoader.cpp:86*
2: *defs.h:252*
3: *ClassLoader.cpp:87*
4: *ClassLoader.cpp:77*
5: *ClassLoader.cpp:294*
6: *ClassLoader.cpp:70*
7: *defs.h:33*
8: *errors.cpp:35*
9: *defs.h:48*
10: *ClassLoader.cpp:316*
11: *defs.h:49*
12: *ClassLoader.h:27*
13: *VirtualMachine.h:302*
14: *ClassFile.h:136*
15: *VirtualMachine.h:281*
16: *ClassLoader.cpp:187*
17: *defs.h:253*
18: *ClassLoader.cpp:136*
19: *ClassLoader.cpp:18*
20: *defs.h:195*
21: *ClassFile.cpp:870*
22: *ClassFile.h:243*
23: *VirtualMachine.h:336*
24: *HandlePool.cpp:25*
25: *ClassLoader.cpp:140*
26: *ClassFile.cpp:965*
27: *defs.h:50*

```
158:                    //-------------------------------------------------------------------------------
159:                    // During this step JVM allocates memory for the class (static) variables
160:                    // and sets them to default initial values. The class variables are not
161:                    // initialized to their proper initial values until the initialization phase.
162:                    // No Java code is executed during the preparation step
163:
164:                    cf[1]->class_data[2] = new ClassData[3](cf[1]);
165:                    result[4] = cf[1]->class_data[2]->prepare[5]();
166:                    if (result[4] != Success[6])
167:                    {
168:                            print_error[7](result[4]);
169:                            return ClassLoaderCannotPrepareClass[8];
170:                    }
171:
172:                    // Step 2c (optional) - resolve the type
173:                    //-------------------------------------------------------------------------------
174:
175:                    // Step 3 - initialize the type
176:                    //-------------------------------------------------------------------------------
177:                    // This function will be called on the superclass at the end of the
178:                    // recursive call before the call on the subclass. Thus, it is in correspondence with
179:                    // the rule that the superclasses must be initialized before their subclasses
180:                    result[4] = cf[1]->initialize[9]();
181:
182:
183:                    return Success[6];
184:            }
185:
186:            // Returns NULL if this type has not been loaded by this class loader
187:            ClassFile[10] * ClassLoader[11]::get_class(wchar_t * class_name, u2[12] class_name_length)
188:            {
189:                    char * ascii_class_name = new char[class_name_length[13]];
190:                    wchar2ascii[14](class_name[15], ascii_class_name[16], class_name_length[13]);
191:                    ClassFile[10] * class_file = (ClassFile[10]*)this_namespace[17].find[18](ascii_class_name[16]);
192:                    // TODO //delete [] ascii_class_name; --- ?
193:                    return class_file[19];
194:            }
195:
196:            // If the class is not loaded yet the function will load it and return
197:            // the instance data of the java.lang.Class instance corresponding to this class.
198:            // If class name is the name of the primitive type (like "int") the special
199:            // class for this type will be created
200:            InstanceData[20] * ClassLoader[11]::find_class(char * class_name)
201:            {
202:                    wchar_t * wname = new wchar_t[strlen(class_name[21])+1];
203:
204:                    // Check whether the type is already defined
205:                    ascii2wchar[22](class_name[21], wname[23]);
206:                    ClassFile[10] * cf = get_class[24](wname[23], wcslen(wname[23]));
207:                    if (cf[25])
208:                    {
209:                            assert(cf[25]->class_instance[26] != NULL);
210:                            return cf[25]->class_instance[26];
```

Footnotes:

1: *ClassLoader.cpp:70*
2: *ClassFile.h:227*
3: *ObjectData.h:98*
4: *ClassLoader.cpp:77*
5: *ObjectData.cpp:41*
6: *defs.h:33*
7: *errors.cpp:35*
8: *defs.h:50*
9: *ClassFile.cpp:805*
10: *ClassFile.h:136*
11: *ClassLoader.h:14*
12: *defs.h:172*
13: *ClassLoader.cpp:187*
14: *util.cpp:8*
15: *ClassLoader.cpp:187*
16: *ClassLoader.cpp:189*
17: *ClassLoader.h:34*
18: *hashtable.cpp:79*
19: *ClassLoader.cpp:191*
20: *ObjectData.h:113*
21: *ClassLoader.cpp:200*
22: *util.cpp:16*
23: *ClassLoader.cpp:202*
24: *ClassLoader.cpp:187*
25: *ClassLoader.cpp:206*
26: *ClassFile.h:243*

```
211:                }
212:
213:                // The type is not defined yet, we have to find and define it.
214:
215:                // First, check whether this request is for the primitive type class
216:
217:                if (!strcmp(BOOLEAN_TYPE_NAME[1],class_name[2]) ||
218:                    !strcmp(CHAR_TYPE_NAME[3],class_name[2]) ||
219:                        !strcmp(BYTE_TYPE_NAME[4],class_name[2]) ||
220:                        !strcmp(SHORT_TYPE_NAME[5],class_name[2]) ||
221:                        !strcmp(INT_TYPE_NAME[6],class_name[2]) ||
222:                        !strcmp(LONG_TYPE_NAME[7],class_name[2]) ||
223:                        !strcmp(FLOAT_TYPE_NAME[8],class_name[2]) ||
224:                        !strcmp(DOUBLE_TYPE_NAME[9],class_name[2]))
225:                {
226:                        cf[10] = create_primitive_type[11](class_name[2]);
227:                        assert (cf[10]->class_instance[12] != NULL);
228:
229:                        delete [] wname[13];
230:                        return cf[10]->class_instance[12];
231:                }
232:
233:                // Otherwise, just load the class
234:
235:                cf[10] = load_class[14](wname[13], wcslen(wname[13]));
236:                assert (cf[10]->class_instance[12] != NULL);
237:
238:                delete [] wname[13];
239:                return cf[10]->class_instance[12];
240:        }
241:
242:
243:        #define READ_BLOCK 1024
244:        #define MAX_CLASSFILE_SIZE 65535 // the size within the u2 as required by the JVM spec
245:
246:        // NOTE: the function will get the filepath WITHOUT the ".class" extension !
247:        Result[15] ClassLoader[16]::open_file(char * filepath, u1[17] *& buffer, u2[18] & length )
248:        {
249:                buffer[19] = new u1[17][MAX_CLASSFILE_SIZE[20]];
250:                u1[17] * pointer = buffer[19];
251:
252:                char full_filepath[MAX_FILE_PATH_NAME[21]];
253:                strcpy(full_filepath[22], filepath[23]);
254:                strcat(full_filepath[22],".class");
255:
256:                FILE * file = fopen(full_filepath[22],"rb");
257:                if (!file[24])
258:                {
259:                        // Try to open the file from the CLASSPATH directory
260:                        strcpy(full_filepath[22], CLASSPATH[25]);
261:                        strcat(full_filepath[22], filepath[23]);
262:                        strcat(full_filepath[22],".class");
263:                        slash2backslash[26](full_filepath[22]);
```

Footnotes:

1: *defs.h:315*
2: *ClassLoader.cpp:200*
3: *defs.h:316*
4: *defs.h:317*
5: *defs.h:318*
6: *defs.h:319*
7: *defs.h:320*
8: *defs.h:321*
9: *defs.h:322*
10: *ClassLoader.cpp:206*
11: *ClassLoader.cpp:434*
12: *ClassFile.h:243*
13: *ClassLoader.cpp:202*
14: *ClassLoader.cpp:18*
15: *defs.h:29*
16: *ClassLoader.h:14*
17: *defs.h:168*
18: *defs.h:172*
19: *ClassLoader.cpp:247*
20: *ClassLoader.cpp:244*
21: *defs.h:203*
22: *ClassLoader.cpp:252*
23: *ClassLoader.cpp:247*
24: *ClassLoader.cpp:256*
25: *defs.h:422*
26: *util.cpp:24*

```
264:
265:                    file[1] = fopen(full_filepath[2], "rb");
266:
267:                    if (!file[1])
268:                            return ClassFileNotFoundError[3];
269:            }
270:
271:            size_t size, total_size = 0;
272:
273:            while (1)
274:            {
275:                    size[4] = fread((void*)pointer[5], sizeof(u1[6]), READ_BLOCK[7], file[1]);
276:                    if (size[4] == READ_BLOCK[7])
277:                    {
278:                            pointer[5] += READ_BLOCK[7];
279:                            total_size[8] += size[4];
280:                    }
281:                    else
282:                    {
283:                            total_size[8] += size[4];
284:                            break;
285:                    }
286:            }
287:
288:            length[9] = total_size[8];
289:
290:            return Success[10];
291:    }
292:
293:    // Loads, defines and resolves the superclass of the given class file
294:    Result[11] ClassLoader[12]::load_superclass(ClassFile[13] * cf)
295:    {
296:            // Loading a superclass means resolving just another symbolic reference
297:            // (namely the "super_class" entry of this "cf" class)
298:            CONSTANT_Class_info[14] * super_class_entry = (CONSTANT_Class_info[14]*)cf[15]->constant_pool[16]->get_item_a
    t_index[17](cf[15]->super_class[18]);
299:            if (super_class_entry[19]->is_resolved[20]())
300:            {
301:                    return Success[10]; // the superclass entry is already resolved, threre is nothing to do
302:            }
303:            else
304:            {
305:                    // This call will recursively load the superclasses of this superclass
306:                    Result[11] result = super_class_entry[19]->resolve[21]();
307:                    if (result[22] != Success[10])
308:                    {
309:                            print_error[23](result[22]);
310:                            return ClassLoaderCannotLoadSuperclass[24];
311:                    }
312:            }
313:            return Success[10];
314:    }
315:
```

Footnotes:

1: *ClassLoader.cpp:256*
2: *ClassLoader.cpp:252*
3: *defs.h:52*
4: *ClassLoader.cpp:271*
5: *ClassLoader.cpp:250*
6: *defs.h:168*
7: *ClassLoader.cpp:243*
8: *ClassLoader.cpp:271*
9: *ClassLoader.cpp:247*
10: *defs.h:33*
11: *defs.h:29*
12: *ClassLoader.h:14*
13: *ClassFile.h:136*
14: *ConstantPool.h:58*
15: *ClassLoader.cpp:294*
16: *ClassFile.h:159*
17: *ConstantPool.h:299*
18: *ClassFile.h:178*
19: *ClassLoader.cpp:298*
20: *ConstantPool.h:48*
21: *ConstantPool.cpp:22*
22: *ClassLoader.cpp:306*
23: *errors.cpp:35*
24: *defs.h:48*

```
316:  Result[1] ClassLoader[2]::load_superinterfaces(ClassFile[3] * cf)
317:  {
318:          // For all superinterfaces
319:          for (int i = 0; i < cf[4]->interfaces_count[5]; i++)
320:          {
321:                  CONSTANT_Class_info[6] * super_interface_entry = (CONSTANT_Class_info[6]*)cf[4]->constant_pool[7]->
      get_item_at_index[8](cf[4]->interfaces[9][i[10]]);
322:                  if (super_interface_entry[11]->is_resolved[12]())
323:                  {
324:                          return Success[13]; // the superinterface entry is already resolved, threre is nothi
      ng to do
325:                  }
326:                  else
327:                  {
328:                          // This call will recursively load the superclasses of this superclass
329:                          Result[1] result = super_interface_entry[11]->resolve[14]();
330:                          if (result[15] != Success[13])
331:                          {
332:                                  print_error[16](result[15]);
333:                                  return ClassLoaderCannotLoadSuperinterfaces[17];
334:                          }
335:                  }
336:          }
337:
338:          return Success[13];
339:  }
340:
341:  Result[1] ClassLoader[2]::define_primitive_array_class(u1[18] array_type, ClassFile[3] *& cf)
342:  {
343:          // Create the name for the primitive array type
344:          wchar_t * type_name = new wchar_t[3];
345:          wcscpy(type_name[19],ARRAY_DIM_INTERNAL_REP[20]);
346:
347:          switch (array_type[21])
348:          {
349:          case T_BOOLEAN[22]: wcscat(type_name[19],BOOL_INTERNAL_REP[23]); break;
350:          case T_CHAR[24]: wcscat(type_name[19],CHAR_INTERNAL_REP[25]); break;
351:          case T_FLOAT[26]: wcscat(type_name[19],FLOAT_INTERNAL_REP[27]); break;
352:          case T_DOUBLE[28]: wcscat(type_name[19],DOUBLE_INTERNAL_REP[29]); break;
353:          case T_BYTE[30]: wcscat(type_name[19],BYTE_INTERNAL_REP[31]); break;
354:          case T_SHORT[32]: wcscat(type_name[19],SHORT_INTERNAL_REP[33]); break;
355:          case T_INT[34]: wcscat(type_name[19],INT_INTERNAL_REP[35]); break;
356:          case T_LONG[36]: wcscat(type_name[19],LONG_INTERNAL_REP[37]); break;
357:          default:
358:                  return ClassLoaderCannotDefinePrimitiveArrayType[38];
359:          }
360:
361:          unsigned int type_name_length = wcslen(type_name[19]);//sizeof(type_name)/sizeof(wchar_t);
362:
363:          // Check whether this class is already defined
364:          cf[39] = get_class[40](type_name[19], type_name_length[41]);
365:
366:          if (cf[39])
```

Footnotes:

1: *defs.h:29*
2: *ClassLoader.h:14*
3: *ClassFile.h:136*
4: *ClassLoader.cpp:316*
5: *ClassFile.h:181*
6: *ConstantPool.h:58*
7: *ClassFile.h:159*
8: *ConstantPool.h:299*
9: *ClassFile.h:187*
10: *ClassLoader.cpp:319*
11: *ClassLoader.cpp:321*
12: *ConstantPool.h:48*
13: *defs.h:33*
14: *ConstantPool.cpp:22*
15: *ClassLoader.cpp:329*
16: *errors.cpp:35*
17: *defs.h:49*
18: *defs.h:168*
19: *ClassLoader.cpp:344*
20: *defs.h:310*
21: *ClassLoader.cpp:341*
22: *defs.h:340*
23: *defs.h:309*
24: *defs.h:341*
25: *defs.h:302*
26: *defs.h:342*
27: *defs.h:304*
28: *defs.h:343*
29: *defs.h:303*
30: *defs.h:344*
31: *defs.h:301*
32: *defs.h:345*
33: *defs.h:308*
34: *defs.h:346*
35: *defs.h:305*
36: *defs.h:347*
37: *defs.h:306*
38: *defs.h:51*
39: *ClassLoader.cpp:341*
40: *ClassLoader.cpp:187*
41: *ClassLoader.cpp:361*

```
367:                        return Success[1]; // the class is already defined
368:
369:                // Create a new class for this array of primitive values
370:
371:                // This class loader will be THE class loader of the class file created
372:                cf[2] = new PrimitiveArrayClassFile[3](this, array_type[4], type_name[5], type_name_length[6]);
373:
374:                // Meanwhile, the key of the hashtable will be the ASCII value of the string
375:                char * ascii_type_name = new char[type_name_length[6]];
376:                wchar2ascii[7](type_name[5], ascii_type_name[8], type_name_length[6]);
377:                this_namespace[9].insert[10](ascii_type_name[8], cf[2]);
378:                // Do not delete the key of the hashtable !
379:
380:                delete [] type_name[5];
381:
382:                // TODO: create a new instance of the java.lang.Class
383:
384:                return Success[1];
385:        }
386:
387:        Result[11] ClassLoader[12]::define_array_class(ClassFile[13] * array_type, ClassFile[13] *& cf)
388:        {
389:                wchar_t * array_type_class_name;
390:                u2[14] array_type_class_name_length;
391:                array_type[15]->get_full_class_name[16](array_type_class_name[17], array_type_class_name_length[18]);
392:
393:                // Create the name for the reference array type
394:                wchar_t * type_name = new wchar_t[array_type_class_name_length[18]+3];
395:                wcscpy(type_name[19],ARRAY_DIM_INTERNAL_REP[20]);
396:                wcscat(type_name[19],array_type_class_name[17]);
397:                wcscat(type_name[19],L";");
398:
399:                unsigned int type_name_length = wcslen(type_name[19]);
400:
401:                // Check whether this class is already defined
402:                cf[21] = get_class[22](type_name[19], type_name_length[23]);
403:
404:                if (cf[21])
405:                        return Success[1]; // the class is already defined
406:
407:                // Create a new class for this array of reference values
408:
409:                // This class loader will be THE class loader of the class file created
410:                cf[21] = new ArrayClassFile[24](this, array_type[15], type_name[19], type_name_length[23]);
411:
412:                // Meanwhile, the key of the hashtable will be the ASCII value of the string
413:                char * ascii_type_name = new char[type_name_length[23]];
414:                wchar2ascii[7](type_name[19], ascii_type_name[25], type_name_length[23]);
415:                this_namespace[9].insert[10](ascii_type_name[25], cf[21]);
416:                // Do not delete the key of the hashtable !
417:
418:                delete [] type_name[19];
419:
```

Footnotes:

1: *defs.h:33*
2: *ClassLoader.cpp:341*
3: *ClassFile.h:375*
4: *ClassLoader.cpp:341*
5: *ClassLoader.cpp:344*
6: *ClassLoader.cpp:361*
7: *util.cpp:8*
8: *ClassLoader.cpp:375*
9: *ClassLoader.h:34*
10: *hashtable.cpp:23*
11: *defs.h:29*
12: *ClassLoader.h:14*
13: *ClassFile.h:136*
14: *defs.h:172*
15: *ClassLoader.cpp:387*
16: *ClassFile.cpp:442*
17: *ClassLoader.cpp:389*
18: *ClassLoader.cpp:390*
19: *ClassLoader.cpp:394*
20: *defs.h:310*
21: *ClassLoader.cpp:387*
22: *ClassLoader.cpp:187*
23: *ClassLoader.cpp:399*
24: *ClassFile.h:349*
25: *ClassLoader.cpp:413*

```
420:                if (!vm[1]->boot_process[2]) // it is impossible during the boot process
421:                {
422:                        ClassFile[3] * class_class = vm[1]->bootstrap_class_loader[4]->get_class[5](CLASS_CLASS_NAME[6], wcsl
        en(CLASS_CLASS_NAME[6]));
423:                        if (!class_class[7])
424:                                class_class[7] = vm[1]->bootstrap_class_loader[4]->load_class[8](CLASS_CLASS_NAME[6], wcslen(
        CLASS_CLASS_NAME[6]));
425:                        assert(class_class[7] != NULL);
426:                        word[9] class_class_ref = class_class[7]->create_new_instance[10]();
427:                        cf[11]->class_instance[12] = vm[1]->get_handle_pool[13]()->get_instance[14](class_class_ref[15]);
428:                        // TODO:
429:                }
430:
431:                return Success[16];
432:        }
433:
434:        ClassFile[3] * ClassLoader[17]::create_primitive_type(char * class_name)
435:        {
436:                ClassFile[3] * cf = new ClassFile[18](this);
437:                cf[19]->constant_pool_count[20] = 0;
438:                this_namespace[21].insert[22](class_name[23], cf[19]);
439:
440:                if (!vm[1]->boot_process[2]) // it is impossible during the boot process
441:                {
442:                        ClassFile[3] * class_class = vm[1]->bootstrap_class_loader[4]->get_class[5](CLASS_CLASS_NAME[6], wcsl
        en(CLASS_CLASS_NAME[6]));
443:                        if (!class_class[24])
444:                                class_class[24] = vm[1]->bootstrap_class_loader[4]->load_class[8](CLASS_CLASS_NAME[6], wcslen(
        CLASS_CLASS_NAME[6]));
445:                        assert(class_class[24] != NULL);
446:                        word[9] class_class_ref = class_class[24]->create_new_instance[10]();
447:                        cf[19]->class_instance[12] = vm[1]->get_handle_pool[13]()->get_instance[14](class_class_ref[25]);
448:                        // TODO:
449:                }
450:
451:                return cf[19];
452:        }
453:
454:        void ClassLoader[17]::dump_namespace()
455:        {
456:                cout << "Class Loader dump start" << endl;
457:                HashTableIterator[26] iterator(this_namespace[21]);
458:                HashTable[27]::HashTableEntry[28] * entry = iterator[29].next[30]();
459:                while (entry[31])
460:                {
461:                        cout << "    " << entry[31]->key[32] << endl;
462:                        entry[31] = iterator[29].next[30]();
463:                }
464:                cout << "Class Loader dump end" << endl;
465:        }
466:
467:
468:        Result[33] ClassLoader[17]::define_boot_class(ClassFile[3] * cf)
```

Footnotes:

1: *ClassLoader.h:27*
2: *VirtualMachine.h:302*
3: *ClassFile.h:136*
4: *VirtualMachine.h:281*
5: *ClassLoader.cpp:187*
6: *defs.h:253*
7: *ClassLoader.cpp:422*
8: *ClassLoader.cpp:18*
9: *defs.h:195*
10: *ClassFile.cpp:870*
11: *ClassLoader.cpp:387*
12: *ClassFile.h:243*
13: *VirtualMachine.h:336*
14: *HandlePool.cpp:25*
15: *ClassLoader.cpp:426*
16: *defs.h:33*
17: *ClassLoader.h:14*
18: *ClassFile.cpp:784*
19: *ClassLoader.cpp:436*
20: *ClassFile.h:153*
21: *ClassLoader.h:34*
22: *hashtable.cpp:23*
23: *ClassLoader.cpp:434*
24: *ClassLoader.cpp:442*
25: *ClassLoader.cpp:446*
26: *hashtable.h:88*
27: *hashtable.h:28*
28: *hashtable.h:32*
29: *ClassLoader.cpp:457*
30: *hashtable.cpp:153*
31: *ClassLoader.cpp:458*
32: *hashtable.h:34*
33: *defs.h:29*

```
469:        {
470:
471:                // The class was successfully loaded. Put it in this class loader's namespace
472:                // (do not delete the class_name because it will be used as a key in the hashtable)
473:                wchar_t * class_name;
474:                u2[1] class_name_length;
475:                cf[2]->get_full_class_name[3](class_name[4], class_name_length[5]);
476:
477:                // Meanwhile, the key of the hashtable will be the ASCII value of the string
478:                char * ascii_class_name = new char[class_name_length[5]];
479:                wchar2ascii[6](class_name[4], ascii_class_name[7], class_name_length[5]);
480:                this_namespace[8].insert[9](ascii_class_name[7], cf[2]);
481:                // Do not delete the key of the hashtable !
482:
483:                // Once the given class is loaded it is time to load its superclass
484:                // (and then superclass' superclass and so on until the java.lang.Object class)
485:                // In this recursive process the superclass (and all the higher superclasses)
486:                // will be defined in this class' class loader namespace
487:                // But, prior to do it, check if this class is not java.lang.Object
488:
489:                if (wmemcmp(class_name[4], OBJECT_CLASS_NAME[10], class_name_length[5]))
490:                {
491:                        Result[11] result = load_superclass[12](cf[2]);
492:                        if (result[13] != Success[14])
493:                        {
494:                                print_error[15](result[13]);
495:                                return ClassLoaderCannotLoadSuperclass[16];
496:                        }
497:
498:                        // Now, it's time to load all the superinterfaces of this type
499:                        // As with the superclass, all superinterfaces will be recursively resolved
500:
501:                        result[13] = load_superinterfaces[17](cf[2]);
502:                        if (result[13] != Success[14])
503:                        {
504:                                print_error[15](result[13]);
505:                                return ClassLoaderCannotLoadSuperinterfaces[18];
506:                        }
507:
508:                }
509:
510:                // Last step of the loading process
511:                //--------------------------------------------------------------------------
512:                // Instantiate the java.lang.Class class that will be associated with this class'
513:                // internal structures
514:                /*
515:                ClassFile * class_class = vm->bootstrap_class_loader->get_class(CLASS_CLASS_NAME, wcslen(CLASS_CL
        ASS_NAME));
516:                if (!class_class)
517:                        class_class = vm->bootstrap_class_loader->load_class(CLASS_CLASS_NAME, wcslen(CLASS_CLASS
        _NAME));
518:                assert(class_class != NULL);
519:                word class_class_ref = class_class->create_new_instance();
```

Footnotes:

1: *defs.h:172*
2: *ClassLoader.cpp:468*
3: *ClassFile.cpp:442*
4: *ClassLoader.cpp:473*
5: *ClassLoader.cpp:474*
6: *util.cpp:8*
7: *ClassLoader.cpp:478*
8: *ClassLoader.h:34*
9: *hashtable.cpp:23*
10: *defs.h:252*
11: *defs.h:29*
12: *ClassLoader.cpp:294*
13: *ClassLoader.cpp:491*
14: *defs.h:33*
15: *errors.cpp:35*
16: *defs.h:48*
17: *ClassLoader.cpp:316*
18: *defs.h:49*

```
520:                    cf->class_instance = vm->get_handle_pool()->get_instance(class_class_ref);
521:                    */
522:                    // Step 2a - verify the type
523:                    //-------------------------------------------------------------------------------
524:
525:                    // Auxiliary step
526:                    // Build the method table of the class
527:                    Result[1] result = cf[2]->build_method_table[3]();
528:                    if (result[4] != Success[5])
529:                    {
530:                            print_error[6](result[4]);
531:                            return ClassLoaderCannotPrepareClass[7];
532:                    }
533:
534:                    // Step 2b - prepare the type
535:                    //-------------------------------------------------------------------------------
536:                    // During this step JVM allocates memory for the class (static) variables
537:                    // and sets them to default initial values. The class variables are not
538:                    // initialized to their proper initial values until the initialization phase.
539:                    // No Java code is executed during the preparation step
540:
541:                    cf[2]->class_data[8] = new ClassData[9](cf[2]);
542:                    result[4] = cf[2]->class_data[8]->prepare[10]();
543:                    if (result[4] != Success[5])
544:                    {
545:                            print_error[6](result[4]);
546:                            return ClassLoaderCannotPrepareClass[7];
547:                    }
548:
549:                    // Step 2c (optional) - resolve the type
550:                    //-------------------------------------------------------------------------------
551:
552:                    // Step 3 - initialize the type
553:                    //-------------------------------------------------------------------------------
554:                    // This function will be called on the superclass at the end of the
555:                    // recursive call before the call on the subclass. Thus, it is in correspondence with
556:                    // the rule that the superclasses must be initialized before their subclasses
557:                    result[4] = cf[2]->initialize[11]();
558:
559:
560:                    return Success[5];
561:            }
```

Footnotes:

1: *defs.h:29*
2: *ClassLoader.cpp:468*
3: *ClassFile.cpp:965*
4: *ClassLoader.cpp:527*
5: *defs.h:33*
6: *errors.cpp:35*
7: *defs.h:50*
8: *ClassFile.h:227*
9: *ObjectData.h:98*
10: *ObjectData.cpp:41*
11: *ClassFile.cpp:805*

```
 1:
 2:          #include <iostream.h>
 3:
 4:          #include "CodeManager.h"
 5:          #include "opcodes.h"
 6:
 7:          void CodeManager[1]::debug_print(u1[2] * code, unsigned long code_length)
 8:          {
 9:                  unsigned long index = 0;
10:
11:                  while (index[3] < code_length[4])
12:                  {
13:                          u1[2] opcode = code[5][index[3]];
14:                          cout << opcodes[6][opcode[7]].mnemonic[8] << endl;
15:                          // TMP
16:                          if (opcodes[6][opcode[7]].argument_number[9] < 0)
17:                                  return;
18:                          // TMP
19:                          for (int i = 0; i < opcodes[6][opcode[7]].argument_number[9]; i++)
20:                          {
21:                                  index[3]++;
22:                                  cout << "   arg #" << (i[10]+1) << ": " << code[5][index[3]] << endl;
23:                          }
24:
25:                          // move forward to the next command
26:                          index[3]++;
27:                  }
28:          }
```

Footnotes:

1: *CodeManager.h:13*
2: *defs.h:168*
3: *CodeManager.cpp:9*
4: *CodeManager.cpp:7*
5: *CodeManager.cpp:7*
6: *opcodes.cpp:3215*
7: *CodeManager.cpp:13*
8: *opcodes.h:256*
9: *opcodes.h:257*
10: *CodeManager.cpp:19*

```
 1:
 2:          #include <math.h>
 3:          #include <stdlib.h>
 4:          #include <string.h>
 5:          #include <iostream.h>
 6:
 7:          #include "ConstantPool.h"
 8:          #include "ClassFile.h"
 9:          #include "ClassLoader.h"
10:          #include "ObjectData.h"
11:          #include "HandlePool.h"
12:
13:          Result[1] CONSTANT_Class_info[2]::read(u1[3] *& stream_pointer)
14:          {
15:                  memcpy_u2[4](&name_index[5], stream_pointer[6]);
16:                  stream_pointer[6] += sizeof(u2[7]);
17:
18:                  return Success[8];
19:          }
20:
21:
22:          Result[1] CONSTANT_Class_info[2]::resolve()
23:          {
24:          /* TODO:
25:                  VirtualMachine * vm = class_file->class_loader->get_jvm();
26:                  Thread * current_thread = vm->current_thread;
27:                  if (current_thread)
28:                          vm->yield(current_thread);
29:          */
30:
31:                  // Load the type and any supertypes
32:                  //-------------------------------------------------------------------------
33:
34:                  // Get this class' name
35:                  ConstantPool[9] * cp = class_file[10]->constant_pool[11];
36:                  wchar_t * resolved_class_name;
37:                  CONSTANT_Utf8_info[12] * utf8_info = (CONSTANT_Utf8_info[12]*)cp[13]->get_item_at_index[14](name_index[5]);
38:
39:                  Result[1] result = utf8_info[15]->get_string[16](resolved_class_name[17]);
40:                  if (result[18] != Success[8])
41:                  {
42:                          print_error[19](result[18]);
43:                          return ResolutionCannotResolveClassName[20];
44:                  }
45:
46:                  // Ask the parent class' class loader whether it has already loaded this type
47:                  ClassFile[21] * class_file_to_resolve = class_file[10]->class_loader[22]->get_class[23](resolved_class_name[17],
           utf8_info[15]->length[24]);
48:                  if (class_file_to_resolve[25])
49:                  {
50:                          // This type is already loaded by the same class loader
51:                          // change symbolic reference to the real reference to this class file
52:                          resolved_class_file[26] = class_file_to_resolve[25];
```

Footnotes:

1: *defs.h:29*
2: *ConstantPool.h:58*
3: *defs.h:168*
4: *memcpy_bigendian.cpp:10*
5: *ConstantPool.h:60*
6: *ConstantPool.cpp:13*
7: *defs.h:172*
8: *defs.h:33*
9: *ConstantPool.h:274*
10: *ConstantPool.h:31*
11: *ClassFile.h:159*
12: *ConstantPool.h:186*
13: *ConstantPool.cpp:35*
14: *ConstantPool.h:299*
15: *ConstantPool.cpp:37*
16: *ConstantPool.cpp:639*
17: *ConstantPool.cpp:36*
18: *ConstantPool.cpp:39*
19: *errors.cpp:35*
20: *defs.h:57*
21: *ClassFile.h:136*
22: *ClassFile.h:222*
23: *ClassLoader.cpp:187*
24: *ConstantPool.h:188*
25: *ConstantPool.cpp:47*
26: *ConstantPool.h:71*

```
53:                             resolved[1] = 1;
54:
55:                             delete [] resolved_class_name[2];
56:                     }
57:                     else // The class was not yet loaded by the parent's class loader
58:                     {
59:                             // Ask the parent's class' class loader to load the class.
60:                             // In the process of loading all the superclasses and superinterfaces
61:                             // of this class will be recursively loaded as well (in case they are not loaded yet)
62:                             class_file_to_resolve[3] = class_file[4]->class_loader[5]->load_class[6](resolved_class_name[2], utf8
        _info[7]->length[8]);
63:                             if (!class_file_to_resolve[3])
64:                             {
65:                                     delete [] resolved_class_name[2];
66:                                     return ResolutionCannotLoadClassFromClassName[9];
67:                             }
68:
69:                             // Class is loaded successfully,
70:                             // change symbolic reference to the real reference to this class file
71:                             resolved_class_file[10] = class_file_to_resolve[3];
72:                             resolved[1] = 1;
73:
74:                             delete [] resolved_class_name[2];
75:                     }
76:
77:                     // Check access permission
78:                     //-------------------------------------------------------------------------------
79:                     // After loading is complete check that the referencing type has permission
80:                     // to access the referenced type
81:
82:
83:                     return Success[11];
84:             }
85:
86:             void CONSTANT_Class_info[12]::debug_print(ostream & out)
87:             {
88:                     CONSTANT_Utf8_info[13] * class_name = (CONSTANT_Utf8_info[13]*)class_file[4]->constant_pool[14]->get_item_at
        _index[15](name_index[16]);
89:                     wchar_t * cname;
90:                     class_name[17]->get_string[18](cname[19]);
91:
92:                     out[20] << "CONSTANT_Class_info:" << endl;
93:                     out[20] << "      name_index: " << name_index[16] << "(";
94:                     print_wchar[21](cname[19], class_name[17]->length[8], out[20]);
95:                     out[20] << ")" << endl;
96:                     delete [] cname[19];
97:             }
98:
99:
100:            Result[22] CONSTANT_Long_info[23]::read(u1[24] *& stream_pointer)
101:            {
102:
103:                    memcpy_u4[25](&high_bytes[26], stream_pointer[27]);
```

Footnotes:

1: *ConstantPool.h:24*
2: *ConstantPool.cpp:36*
3: *ConstantPool.cpp:47*
4: *ConstantPool.h:31*
5: *ClassFile.h:222*
6: *ClassLoader.cpp:18*
7: *ConstantPool.cpp:37*
8: *ConstantPool.h:188*
9: *defs.h:58*
10: *ConstantPool.h:71*
11: *defs.h:33*
12: *ConstantPool.h:58*
13: *ConstantPool.h:186*
14: *ClassFile.h:159*
15: *ConstantPool.h:299*
16: *ConstantPool.h:60*
17: *ConstantPool.cpp:88*
18: *ConstantPool.cpp:639*
19: *ConstantPool.cpp:89*
20: *ConstantPool.cpp:86*
21: *util.cpp:42*
22: *defs.h:29*
23: *ConstantPool.h:231*
24: *defs.h:168*
25: *memcpy_bigendian.cpp:27*
26: *ConstantPool.h:233*
27: *ConstantPool.cpp:100*

```
104:                stream_pointer[1] += sizeof(u4[2]);
105:                memcpy_u4[3](&low_bytes[4], stream_pointer[1]);
106:                stream_pointer[1] += sizeof(u4[2]);
107:
108:                return Success[5];
109:        }
110:
111:        su8[6] CONSTANT_Long_info[7]::get_long_value()
112:        {
113:
114:        /* The following idioms should be used to divide 64-bit signed integer into high and low
115:           bytes and assemble them back together:
116:
117:                signed __int64 a = -123456789123456789; // example number
118:                // Divide into two halves
119:                unsigned __int32 high = a >> 32;       // Note, that high bytes are UNSIGNED
120:                unsigned __int32 low = a & 0xFFFFFFFF; // Note that low bytes are UNSIGNED
121:                // To print 64-bit integer use %I64d (for signed) ot %I64u (for unsigned)
122:                printf("original: %I64d\n",a);
123:                printf("high bytes: %u\n",high);
124:                printf("low bytes: %u\n",low);
125:                // Assemble the number back
126:                // Note, that the conversion of high bytes into signed __int64 is necessary !
127:                signed __int64 back = (((signed __int64)high) << 32) + low;
128:                printf("original: %I64d\n",back);
129:        */
130:
131:                return (((su8[6])high_bytes[8]) << 32) + low_bytes[4];
132:        }
133:
134:        Result[9] CONSTANT_Long_info[7]::resolve()
135:        {
136:                return Success[5];
137:        }
138:
139:        void CONSTANT_Long_info[7]::debug_print(ostream & out)
140:        {
141:                out[10] << "CONSTANT_Long_info: " << (long)get_long_value[11]() << endl;
142:        }
143:
144:        Result[9] CONSTANT_Float_info[12]::read(u1[13] *& stream_pointer)
145:        {
146:                memcpy_u4[3](&bytes[14], stream_pointer[15]);
147:                stream_pointer[15] += sizeof(u4[2]);
148:
149:                return Success[5];
150:        }
151:
152:        float CONSTANT_Float_info[12]::get_float_value()
153:        {
154:                su4[16] bits = (su4[16]) bytes[14];
155:
156:                if (0x7f800000 == bits[17])
```

Footnotes:

1: *ConstantPool.cpp:100*
2: *defs.h:174*
3: *memcpy_bigendian.cpp:27*
4: *ConstantPool.h:234*
5: *defs.h:33*
6: *defs.h:177*
7: *ConstantPool.h:231*
8: *ConstantPool.h:233*
9: *defs.h:29*
10: *ConstantPool.cpp:139*
11: *ConstantPool.cpp:111*
12: *ConstantPool.h:219*
13: *defs.h:168*
14: *ConstantPool.h:221*
15: *ConstantPool.cpp:144*
16: *defs.h:175*
17: *ConstantPool.cpp:154*

```
157:                        return POSITIVE_INFINITY[1];
158:                if (0xff800000 == bits[2])
159:                        return NEGATIVE_INFINITY[3];
160:                if ( (bits[2] >= 0x7f800001 && bits[2] <= 0x7fffffff) ||
161:                         (bits[2] >= 0xff800001 && bits[2] <= 0xffffffff) )
162:                        return NaN[4];
163:
164:            su4[5] s = ((bits[2] >> 31) == 0) ? 1 : -1;
165:            su4[5] e = ((bits[2] >> 23) & 0xff);
166:            su4[5] m = (e[6] == 0) ?
167:                                (bits[2] & 0x7fffff) << 1 :
168:                                (bits[2] & 0x7fffff) | 0x800000;
169:
170:                return float( s[7] * m[8] * pow(2, (e[6]-150) ) );
171:        }
172:
173:        Result[9] CONSTANT_Float_info[10]::resolve()
174:        {
175:                return Success[11];
176:        }
177:
178:        void CONSTANT_Float_info[10]::debug_print(ostream & out)
179:        {
180:                out[12] << "CONSTANT_Float_info: " << get_float_value[13]() << endl;
181:        }
182:
183:        Result[9] CONSTANT_Double_info[14]::read(u1[15] *& stream_pointer)
184:        {
185:                memcpy_u4[16](&high_bytes[17], stream_pointer[18]);
186:                stream_pointer[18] += sizeof(u4[19]);
187:                memcpy_u4[16](&low_bytes[20], stream_pointer[18]);
188:                stream_pointer[18] += sizeof(u4[19]);
189:
190:                return Success[11];
191:        }
192:
193:        double CONSTANT_Double_info[14]::get_double_value(su8[21] bits)
194:        {
195:                if (0x7ff0000000000000L == bits[22])
196:                        return POSITIVE_INFINITY[1];
197:                if (0xfff0000000000000L == bits[22])
198:                        return NEGATIVE_INFINITY[3];
199:                if ( (bits[22] >= 0x7ff0000000000001L && bits[22] <= 0x7fffffffffffffffL) ||
200:                         (bits[22] >= 0xfff0000000000001L && bits[22] <= 0xffffffffffffffffL) )
201:                        return NaN[4];
202:
203:                su4[5] s = ((bits[22] >> 63) == 0) ? 1 : -1;
204:            su4[5] e = (su4[5])((bits[22] >> 52) & 0x7ffL);
205:            su8[21] m = (e[23] == 0) ?
206:                        (bits[22] & 0xfffffffffffffL) << 1 :
207:                        (bits[22] & 0xfffffffffffffL) | 0x10000000000000L;
208:
209:                return double( s[24] * m[25] * pow(2, (e[23]-1075) ) );
```

Footnotes:

1: *defs.h:390*
2: *ConstantPool.cpp:154*
3: *defs.h:391*
4: *defs.h:392*
5: *defs.h:175*
6: *ConstantPool.cpp:165*
7: *ConstantPool.cpp:164*
8: *ConstantPool.cpp:166*
9: *defs.h:29*
10: *ConstantPool.h:219*
11: *defs.h:33*
12: *ConstantPool.cpp:178*
13: *ConstantPool.cpp:152*
14: *ConstantPool.h:244*
15: *defs.h:168*
16: *memcpy_bigendian.cpp:27*
17: *ConstantPool.h:246*
18: *ConstantPool.cpp:183*
19: *defs.h:174*
20: *ConstantPool.h:247*
21: *defs.h:177*
22: *ConstantPool.cpp:193*
23: *ConstantPool.cpp:204*
24: *ConstantPool.cpp:203*
25: *ConstantPool.cpp:205*

```
210:        }
211:
212:        double CONSTANT_Double_info[1]::get_double_value()
213:        {
214:                su8[2] bits = su8[2]( ((su8[2])high_bytes[3] << 32) + low_bytes[4] );
215:                return get_double_value[5](bits[6]);
216:        }
217:
218:        Result[7] CONSTANT_Double_info[1]::resolve()
219:        {
220:                return Success[8];
221:        }
222:
223:        void CONSTANT_Double_info[1]::debug_print(ostream & out)
224:        {
225:                out[9] << "CONSTANT_Double_info: " << get_double_value[5]() << endl;
226:        }
227:
228:        Result[7] CONSTANT_Fieldref_info[10]::read(u1[11] *& stream_pointer)
229:        {
230:                memcpy_u2[12](&class_index[13], stream_pointer[14]);
231:                stream_pointer[14] += sizeof(u2[15]);
232:                memcpy_u2[12](&name_and_type_index[16], stream_pointer[14]);
233:                stream_pointer[14] += sizeof(u2[15]);
234:
235:                return Success[8];
236:        }
237:
238:        Result[7] CONSTANT_Fieldref_info[10]::resolve()
239:        {
240:                // Before resolving the field try to resolve the class this field belongs to
241:                CONSTANT_Class_info[17] * class_info = (CONSTANT_Class_info[17]*)class_file[18]->constant_pool[19]->get_item_
        at_index[20](class_index[13]);
242:                if (!class_info[21])
243:                        return ResolutionCannotResolveFieldref[22];
244:
245:                if (!class_info[21]->is_resolved[23]())
246:                {
247:                        // Referenced class is not resolved yet; resolve it first
248:                        Result[7] result = class_info[21]->resolve[24]();
249:                        if (result[25] != Success[8])
250:                        {
251:                                print_error[26](result[25]);
252:                                return ResolutionCannotResolveFieldref[22];
253:                        }
254:                }
255:
256:                // Now the referenced class is completely resolved
257:                ClassFile[27] * referenced_class = class_info[21]->resolved_class_file[28];
258:                assert(referenced_class[29] != NULL);
259:
260:                // Get fields's name and type info
261:                CONSTANT_NameAndType_info[30] * nat_info =
```

Footnotes:

1: *ConstantPool.h:244*
2: *defs.h:177*
3: *ConstantPool.h:246*
4: *ConstantPool.h:247*
5: *ConstantPool.cpp:212*
6: *ConstantPool.cpp:214*
7: *defs.h:29*
8: *defs.h:33*
9: *ConstantPool.cpp:223*
10: *ConstantPool.h:74*
11: *defs.h:168*
12: *memcpy_bigendian.cpp:10*
13: *ConstantPool.h:78*
14: *ConstantPool.cpp:228*
15: *defs.h:172*
16: *ConstantPool.h:80*
17: *ConstantPool.h:58*
18: *ConstantPool.h:31*
19: *ClassFile.h:159*
20: *ConstantPool.h:299*
21: *ConstantPool.cpp:241*
22: *defs.h:61*
23: *ConstantPool.h:48*
24: *ConstantPool.cpp:22*
25: *ConstantPool.cpp:248*
26: *errors.cpp:35*
27: *ClassFile.h:136*
28: *ConstantPool.h:71*
29: *ConstantPool.cpp:257*
30: *ConstantPool.h:258*

```
262:                    (CONSTANT_NameAndType_info[1]*)class_file[2]->constant_pool[3]->get_item_at_index[4](name_and_type_
        index[5]);
263:            if (!nat_info[6])
264:                    return ResolutionCannotResolveFieldref[7];
265:
266:            // Get field's name
267:            CONSTANT_Utf8_info[8] * utf8_name =
268:                    (CONSTANT_Utf8_info[8]*)class_file[2]->constant_pool[3]->get_item_at_index[4](nat_info[6]->name_index[9]
        );
269:            if (!utf8_name[10])
270:                    return ResolutionCannotResolveFieldref[7];
271:            // Get field's descriptor
272:            CONSTANT_Utf8_info[8] * utf8_descriptor =
273:                    (CONSTANT_Utf8_info[8]*)class_file[2]->constant_pool[3]->get_item_at_index[4](nat_info[6]->descriptor
        _index[11]);
274:            if (!utf8_descriptor[12])
275:                    return ResolutionCannotResolveFieldref[7];
276:
277:            Result[13] result = referenced_class[14]->get_field[15](utf8_name[10], utf8_descriptor[12], finfo[16]);
278:            if (result[17] != Success[18])
279:            {
280:                    print_error[19](result[17]);
281:                    return ResolutionCannotResolveFieldref[7];
282:            }
283:
284:            // finfo field is now pointing to the field_info structure in the referenced class
285:            // and the resolution process is complete
286:            resolved[20] = 1;
287:
288:            return Success[18];
289:    }
290:
291:    void CONSTANT_Fieldref_info[21]::debug_print(ostream & out)
292:    {
293:            out[22] << "CONSTANT_Fieldref_info:" << endl;
294:            CONSTANT_Class_info[23] * class_info = (CONSTANT_Class_info[23]*)class_file[2]->constant_pool[3]->get_item_a
        t_index[4](class_index[24]);
295:            CONSTANT_Utf8_info[8] * class_name = (CONSTANT_Utf8_info[8]*)class_file[2]->constant_pool[3]->get_item_at_i
        ndex[4](class_info[25]->name_index[26]);
296:            wchar_t * cname;
297:            class_name[27]->get_string[28](cname[29]);
298:
299:            CONSTANT_NameAndType_info[1] * nat_info = (CONSTANT_NameAndType_info[1]*)class_file[2]->constant_pool[3]->g
        et_item_at_index[4](name_and_type_index[5]);
300:            CONSTANT_Utf8_info[8] * method_name = (CONSTANT_Utf8_info[8]*)class_file[2]->constant_pool[3]->get_item_at_
        index[4](nat_info[30]->name_index[9]);
301:            wchar_t * mname;
302:            method_name[31]->get_string[28](mname[32]);
303:            CONSTANT_Utf8_info[8] * descriptor = (CONSTANT_Utf8_info[8]*)class_file[2]->constant_pool[3]->get_item_at_i
        ndex[4](nat_info[30]->descriptor_index[11]);
304:            wchar_t * dname;
305:            descriptor[33]->get_string[28](dname[34]);
306:
```

Footnotes:

1: *ConstantPool.h:258*
2: *ConstantPool.h:31*
3: *ClassFile.h:159*
4: *ConstantPool.h:299*
5: *ConstantPool.h:80*
6: *ConstantPool.cpp:261*
7: *defs.h:61*
8: *ConstantPool.h:186*
9: *ConstantPool.h:260*
10: *ConstantPool.cpp:267*
11: *ConstantPool.h:261*
12: *ConstantPool.cpp:272*
13: *defs.h:29*
14: *ConstantPool.cpp:257*
15: *ClassFile.cpp:627*
16: *ConstantPool.h:90*
17: *ConstantPool.cpp:277*
18: *defs.h:33*
19: *errors.cpp:35*
20: *ConstantPool.h:24*
21: *ConstantPool.h:74*
22: *ConstantPool.cpp:291*
23: *ConstantPool.h:58*
24: *ConstantPool.h:78*
25: *ConstantPool.cpp:294*
26: *ConstantPool.h:60*
27: *ConstantPool.cpp:295*
28: *ConstantPool.cpp:639*
29: *ConstantPool.cpp:296*
30: *ConstantPool.cpp:299*
31: *ConstantPool.cpp:300*
32: *ConstantPool.cpp:301*
33: *ConstantPool.cpp:303*
34: *ConstantPool.cpp:304*

```
307:                    out[1] << "      class_index: " << class_index[2] << "(";
308:                    print_wchar[3](cname[4],class_name[5]->length[6],out[1]);
309:                    out[1] << ")" << endl;
310:
311:                    out[1] << "      name_and_type_index: " << name_and_type_index[7] << "(";
312:                    print_wchar[3](mname[8],method_name[9]->length[6],out[1]);
313:                    print_wchar[3](dname[10],descriptor[11]->length[6],out[1]);
314:                    out[1] << ")" << endl;
315:
316:                    delete [] cname[4];
317:                    delete [] mname[8];
318:                    delete [] dname[10];
319:            }
320:
321:            Result[12] CONSTANT_Methodref_info[13]::read(u1[14] *& stream_pointer)
322:            {
323:                    memcpy_u2[15](&class_index[16], stream_pointer[17]);
324:                    stream_pointer[17] += sizeof(u2[18]);
325:                    memcpy_u2[15](&name_and_type_index[19], stream_pointer[17]);
326:                    stream_pointer[17] += sizeof(u2[18]);
327:
328:                    return Success[20];
329:            }
330:
331:
332:            // Will be used by invokevirtual instruction;
333:            // The result of this function is the "moffset" field set
334:            Result[12] CONSTANT_Methodref_info[13]::resolve_virtual()
335:            {
336:                    ClassFile[21] * referenced_class;
337:                    CONSTANT_Utf8_info[22] * utf8_name;
338:                    CONSTANT_Utf8_info[22] * utf8_descriptor;
339:
340:                    Result[12] result = start_resolution[23](referenced_class[24], utf8_name[25], utf8_descriptor[26]);
341:
342:                    if (result[27] != Success[20])
343:                            return result[27];
344:
345:                    result[27] = referenced_class[24]->get_virtual_method[28](utf8_name[25], utf8_descriptor[26], moffset[29]);
346:
347:                    if (result[27] != Success[20])
348:                    {
349:                            print_error[30](result[27]);
350:                            return ResolutionCannotResolveMethodref[31];
351:                    }
352:
353:                    // Mark this methodref_info as resolved
354:                    resolved[32] = 1;
355:
356:                    return Success[20];
357:            }
358:
359:            // Will be used by invokestatic and invokespecial instructions
```

Footnotes:

1: *ConstantPool.cpp:291*
2: *ConstantPool.h:78*
3: *util.cpp:42*
4: *ConstantPool.cpp:296*
5: *ConstantPool.cpp:295*
6: *ConstantPool.h:188*
7: *ConstantPool.h:80*
8: *ConstantPool.cpp:301*
9: *ConstantPool.cpp:300*
10: *ConstantPool.cpp:304*
11: *ConstantPool.cpp:303*
12: *defs.h:29*
13: *ConstantPool.h:98*
14: *defs.h:168*
15: *memcpy_bigendian.cpp:10*
16: *ConstantPool.h:102*
17: *ConstantPool.cpp:321*
18: *defs.h:172*
19: *ConstantPool.h:104*
20: *defs.h:33*
21: *ClassFile.h:136*
22: *ConstantPool.h:186*
23: *ConstantPool.cpp:386*
24: *ConstantPool.cpp:336*
25: *ConstantPool.cpp:337*
26: *ConstantPool.cpp:338*
27: *ConstantPool.cpp:340*
28: *ClassFile.cpp:523*
29: *ConstantPool.h:130*
30: *errors.cpp:35*
31: *defs.h:59*
32: *ConstantPool.h:24*

```
360:          // The result of this function is the "minfo" field set
361:          Result[1] CONSTANT_Methodref_info[2]::resolve_nonvirtual()
362:          {
363:                  ClassFile[3] * referenced_class;
364:                  CONSTANT_Utf8_info[4] * utf8_name;
365:                  CONSTANT_Utf8_info[4] * utf8_descriptor;
366:
367:                  Result[1] result = start_resolution[5](referenced_class[6], utf8_name[7], utf8_descriptor[8]);
368:                  if (result[9] != Success[10])
369:                          return result[9];
370:
371:                  result[9] = referenced_class[6]->get_method[11](utf8_name[7], utf8_descriptor[8], minfo[12]);
372:
373:                  if (result[9] != Success[10])
374:                  {
375:                          print_error[13](result[9]);
376:                          return ResolutionCannotResolveMethodref[14];
377:                  }
378:
379:                  // Mark this methodref_info as resolved
380:                  resolved[15] = 1;
381:
382:                  return Success[10];
383:          }
384:
385:
386:          Result[1] CONSTANT_Methodref_info[2]::start_resolution(ClassFile[3] *& referenced_class,
387:                                                                                                  CONSTANT
          _Utf8_info[4] *& utf8_name,
388:                                                                                                  CONSTANT
          _Utf8_info[4] *& utf8_descriptor)
389:          {
390:                  // Before resolving the method try to resolve the class this method belongs to
391:                  CONSTANT_Class_info[16] * class_info = (CONSTANT_Class_info[16]*)class_file[17]->constant_pool[18]->get_item_
          at_index[19](class_index[20]);
392:                  if (!class_info[21])
393:                          return ResolutionCannotResolveMethodref[14];
394:
395:                  if (!class_info[21]->is_resolved[22]())
396:                  {
397:                          // Referenced class is not resolved yet; resolve it first
398:                          Result[1] result = class_info[21]->resolve[23]();
399:                          if (result[24] != Success[10])
400:                          {
401:                                  print_error[13](result[24]);
402:                                  return ResolutionCannotResolveMethodref[14];
403:                          }
404:                  }
405:
406:                  // Now the referenced class is completely resolved
407:                  referenced_class[25] = class_info[21]->resolved_class_file[26];
408:
409:                  // Class cannot be an interface
```

Footnotes:

1: *defs.h:29*
2: *ConstantPool.h:98*
3: *ClassFile.h:136*
4: *ConstantPool.h:186*
5: *ConstantPool.cpp:386*
6: *ConstantPool.cpp:363*
7: *ConstantPool.cpp:364*
8: *ConstantPool.cpp:365*
9: *ConstantPool.cpp:367*
10: *defs.h:33*
11: *ClassFile.cpp:457*
12: *ConstantPool.h:125*
13: *errors.cpp:35*
14: *defs.h:59*
15: *ConstantPool.h:24*
16: *ConstantPool.h:58*
17: *ConstantPool.h:31*
18: *ClassFile.h:159*
19: *ConstantPool.h:299*
20: *ConstantPool.h:102*
21: *ConstantPool.cpp:391*
22: *ConstantPool.h:48*
23: *ConstantPool.cpp:22*
24: *ConstantPool.cpp:398*
25: *ConstantPool.cpp:386*
26: *ConstantPool.h:71*

```
410:                    if (referenced_class[1]->access_flags[2] & ACC_INTERFACE[3])
411:                            return VM_ERROR_IncompatibleClassChangeError[4];
412:
413:                    // Get method's name and type info
414:                    CONSTANT_NameAndType_info[5] * nat_info =
415:                            (CONSTANT_NameAndType_info[5]*)class_file[6]->constant_pool[7]->get_item_at_index[8](name_and_type_
            index[9]);
416:                    if (!nat_info[10])
417:                            return ResolutionCannotResolveMethodref[11];
418:
419:                    // Get method's name
420:                    utf8_name[12] =
421:                            (CONSTANT_Utf8_info[13]*)class_file[6]->constant_pool[7]->get_item_at_index[8](nat_info[10]->name_inde
            x[14]);
422:                    if (!utf8_name[12])
423:                            return ResolutionCannotResolveMethodref[11];
424:                    // Get method's descriptor
425:                    utf8_descriptor[15] =
426:                            (CONSTANT_Utf8_info[13]*)class_file[6]->constant_pool[7]->get_item_at_index[8](nat_info[10]->descripto
            r_index[16]);
427:                    if (!utf8_descriptor[15])
428:                            return ResolutionCannotResolveMethodref[11];
429:
430:                    return Success[17];
431:            }
432:
433:
434:            // Returns the total size (in words) of all of the method arguments.
435:            // This function is used in the invokation instructions to determine the depth
436:            // of the overlapped memory block between two stack frames
437:            unsigned int CONSTANT_Methodref_info[18]::get_arguments_size()
438:            {
439:                    if (total_arguments_size[19] >= 0)
440:                            return (unsigned int)total_arguments_size[19];
441:
442:                    Vector[20]<general_type*> args;
443:                    general_type[21] return_type;
444:                    get_arguments[22](args[23], &return_type[24]);
445:
446:                    total_arguments_size[19] = 0;
447:                    for (int i = 0; i < args[23].size[25](); i++)
448:                    {
449:                            general_type[26] * cur_arg = args[23].element_at[27](i[28]);
450:                            if ( (cur_arg[29]->kind[30] == general_type[26]::Basic[31] && cur_arg[29]->type[32] == Long[33]) ||
451:                                    (cur_arg[29]->kind[30] == general_type[26]::Basic[31] && cur_arg[29]->type[32] == Double[34]) )
452:                                    total_arguments_size[19] += 2;
453:                            else
454:                                    total_arguments_size[19] += 1;
455:                    }
456:                    return (unsigned int)total_arguments_size[19];
457:            }
458:
            Result[35] CONSTANT_Methodref_info[18]::get_arguments(Vector[36]<general_type*>& args, general_type[26] * return_type
```

Footnotes:

1: *ConstantPool.cpp:386*
2: *ClassFile.h:163*
3: *defs.h:357*
4: *defs.h:101*
5: *ConstantPool.h:258*
6: *ConstantPool.h:31*
7: *ClassFile.h:159*
8: *ConstantPool.h:299*
9: *ConstantPool.h:104*
10: *ConstantPool.cpp:414*
11: *defs.h:59*
12: *ConstantPool.cpp:387*
13: *ConstantPool.h:186*
14: *ConstantPool.h:260*
15: *ConstantPool.cpp:388*
16: *ConstantPool.h:261*
17: *defs.h:33*
18: *ConstantPool.h:98*
19: *ConstantPool.h:137*
20: *vector.h:17*
21: *defs.h:239*
22: *ConstantPool.cpp:459*
23: *ConstantPool.cpp:442*
24: *ConstantPool.cpp:443*
25: *vector.h:23*
26: *defs.h:231*
27: *vector.h:36*
28: *ConstantPool.cpp:447*
29: *ConstantPool.cpp:449*
30: *defs.h:233*
31: *defs.h:233*
32: *defs.h:237*
33: *defs.h:215*
34: *defs.h:212*
35: *defs.h:29*
36: *vector.h:7*

```
460:        )
461:        {
462:                // Get method's name and type info
463:                CONSTANT_NameAndType_info[1] * nat_info =
                            (CONSTANT_NameAndType_info[1]*)class_file[2]->constant_pool[3]->get_item_at_index[4](name_and_type_
            index[5]);
464:                if (!nat_info[6])
465:                        return Failure[7];
466:                // Get method's descriptor
467:                CONSTANT_Utf8_info[8] * utf8_descriptor =
468:                        (CONSTANT_Utf8_info[8]*)class_file[2]->constant_pool[3]->get_item_at_index[4](nat_info[6]->descriptor
            _index[9]);
469:                if (!utf8_descriptor[10])
470:                        return Failure[7];
471:
472:                wchar_t * wstring;
473:                utf8_descriptor[10]->get_string[11](wstring[12]);
474:                // Meanwhile ASCII only method names are good enough
475:                char * string = new char[utf8_descriptor[10]->length[13]+1];
476:                wchar2ascii[14](wstring[12], string[15], utf8_descriptor[10]->length[13]);
477:                delete [] wstring[12];
478:
479:                Result[16] result = decipher_method_descriptor[17](string[15], args[18], return_type[19]);
480:                if (result[20] != Success[21])
481:                {
482:                        delete [] string[15];
483:                        print_error[22](result[20]);
484:                        return Failure[7];
485:                }
486:
487:                delete [] string[15];
488:                return Success[21];
489:        }
490:
491:        void CONSTANT_Methodref_info[23]::debug_print(ostream & out)
492:        {
493:                CONSTANT_Class_info[24] * class_info = (CONSTANT_Class_info[24]*)class_file[2]->constant_pool[3]->get_item_a
            t_index[4](class_index[25]);
494:                CONSTANT_Utf8_info[8] * class_name = (CONSTANT_Utf8_info[8]*)class_file[2]->constant_pool[3]->get_item_at_i
            ndex[4](class_info[26]->name_index[27]);
495:                wchar_t * cname;
496:                class_name[28]->get_string[11](cname[29]);
497:
498:                CONSTANT_NameAndType_info[1] * nat_info = (CONSTANT_NameAndType_info[1]*)class_file[2]->constant_pool[3]->g
            et_item_at_index[4](name_and_type_index[5]);
499:                CONSTANT_Utf8_info[8] * method_name = (CONSTANT_Utf8_info[8]*)class_file[2]->constant_pool[3]->get_item_at_
            index[4](nat_info[30]->name_index[31]);
500:                wchar_t * mname;
501:                method_name[32]->get_string[11](mname[33]);
502:                CONSTANT_Utf8_info[8] * descriptor = (CONSTANT_Utf8_info[8]*)class_file[2]->constant_pool[3]->get_item_at_i
            ndex[4](nat_info[30]->descriptor_index[9]);
503:                wchar_t * dname;
504:                descriptor[34]->get_string[11](dname[35]);
```

Footnotes:

1: *ConstantPool.h:258*
2: *ConstantPool.h:31*
3: *ClassFile.h:159*
4: *ConstantPool.h:299*
5: *ConstantPool.h:104*
6: *ConstantPool.cpp:462*
7: *defs.h:34*
8: *ConstantPool.h:186*
9: *ConstantPool.h:261*
10: *ConstantPool.cpp:467*
11: *ConstantPool.cpp:639*
12: *ConstantPool.cpp:472*
13: *ConstantPool.h:188*
14: *util.cpp:8*
15: *ConstantPool.cpp:475*
16: *defs.h:29*
17: *decipher.cpp:113*
18: *ConstantPool.cpp:459*
19: *ConstantPool.cpp:459*
20: *ConstantPool.cpp:479*
21: *defs.h:33*
22: *errors.cpp:35*
23: *ConstantPool.h:98*
24: *ConstantPool.h:58*
25: *ConstantPool.h:102*
26: *ConstantPool.cpp:493*
27: *ConstantPool.h:60*
28: *ConstantPool.cpp:494*
29: *ConstantPool.cpp:495*
30: *ConstantPool.cpp:498*
31: *ConstantPool.h:260*
32: *ConstantPool.cpp:499*
33: *ConstantPool.cpp:500*
34: *ConstantPool.cpp:502*
35: *ConstantPool.cpp:503*

```
505:
506:                    out[1] << "CONSTANT_Methodref_info:" << endl;
507:                    out[1] << "      class_index: " << class_index[2] << "(";
508:                    print_wchar[3](cname[4],class_name[5]->length[6],out[1]);
509:                    out[1] << ")" << endl;
510:
511:                    out[1] << "      name_and_type_index: " << name_and_type_index[7] << "(";
512:                    print_wchar[3](mname[8],method_name[9]->length[6],out[1]);
513:                    print_wchar[3](dname[10],descriptor[11]->length[6],out[1]);
514:                    out[1] << ")" << endl;
515:
516:                    delete [] cname[4];
517:                    delete [] mname[8];
518:                    delete [] dname[10];
519:            }
520:
521:            Result[12] CONSTANT_InterfaceMethodref_info[13]::read(u1[14] *& stream_pointer)
522:            {
523:                    memcpy_u2[15](&class_index[16], stream_pointer[17]);
524:                    stream_pointer[17] += sizeof(u2[18]);
525:                    memcpy_u2[15](&name_and_type_index[19], stream_pointer[17]);
526:                    stream_pointer[17] += sizeof(u2[18]);
527:
528:                    return Success[20];
529:            }
530:
531:            Result[12] CONSTANT_InterfaceMethodref_info[13]::resolve_on(InstanceData[21] * id, method_info[22] *& minfo)
532:            {
533:                    // Before resolving the method try to resolve the class this method belongs to
534:                    CONSTANT_Class_info[23] * class_info = (CONSTANT_Class_info[23]*)class_file[24]->constant_pool[25]->get_item_
        at_index[26](class_index[16]);
535:                    if (!class_info[27])
536:                            return ResolutionCannotResolveInterfaceMethodref[28];
537:
538:                    if (!class_info[27]->is_resolved[29]())
539:                    {
540:                            // Referenced class is not resolved yet; resolve it first
541:                            Result[12] result = class_info[27]->resolve[30]();
542:                            if (result[31] != Success[20])
543:                            {
544:                                    print_error[32](result[31]);
545:                                    return ResolutionCannotResolveInterfaceMethodref[28];
546:                            }
547:                    }
548:
549:                    // Now the referenced class is completely resolved
550:                    ClassFile[33] * referenced_class = class_info[27]->resolved_class_file[34];
551:
552:                    // Class cannot be an interface
553:                    if (! (referenced_class[35]->access_flags[36] & ACC_INTERFACE[37]) )
554:                            return VM_ERROR_IncompatibleClassChangeError[38];
555:
556:                    // Get method's name and type info
```

Footnotes:

1: *ConstantPool.cpp:491*
2: *ConstantPool.h:102*
3: *util.cpp:42*
4: *ConstantPool.cpp:495*
5: *ConstantPool.cpp:494*
6: *ConstantPool.h:188*
7: *ConstantPool.h:104*
8: *ConstantPool.cpp:500*
9: *ConstantPool.cpp:499*
10: *ConstantPool.cpp:503*
11: *ConstantPool.cpp:502*
12: *defs.h:29*
13: *ConstantPool.h:156*
14: *defs.h:168*
15: *memcpy_bigendian.cpp:10*
16: *ConstantPool.h:158*
17: *ConstantPool.cpp:521*
18: *defs.h:172*
19: *ConstantPool.h:159*
20: *defs.h:33*
21: *ObjectData.h:113*
22: *ClassFile.h:92*
23: *ConstantPool.h:58*
24: *ConstantPool.h:31*
25: *ClassFile.h:159*
26: *ConstantPool.h:299*
27: *ConstantPool.cpp:534*
28: *defs.h:60*
29: *ConstantPool.h:48*
30: *ConstantPool.cpp:22*
31: *ConstantPool.cpp:541*
32: *errors.cpp:35*
33: *ClassFile.h:136*
34: *ConstantPool.h:71*
35: *ConstantPool.cpp:550*
36: *ClassFile.h:163*
37: *defs.h:357*
38: *defs.h:101*

```
557:                CONSTANT_NameAndType_info[1] * nat_info =
558:                        (CONSTANT_NameAndType_info[1]*)class_file[2]->constant_pool[3]->get_item_at_index[4](name_and_type_
        index[5]);
559:                if (!nat_info[6])
560:                        return ResolutionCannotResolveInterfaceMethodref[7];
561:
562:                // Get method's name
563:                CONSTANT_Utf8_info[8] * utf8_name =
564:                        (CONSTANT_Utf8_info[8]*)class_file[2]->constant_pool[3]->get_item_at_index[4](nat_info[6]->name_index[9]
        );
565:                if (!utf8_name[10])
566:                        return ResolutionCannotResolveInterfaceMethodref[7];
567:                // Get method's descriptor
568:                CONSTANT_Utf8_info[8] * utf8_descriptor =
569:                        (CONSTANT_Utf8_info[8]*)class_file[2]->constant_pool[3]->get_item_at_index[4](nat_info[6]->descriptor
        _index[11]);
570:                if (!utf8_descriptor[12])
571:                        return ResolutionCannotResolveInterfaceMethodref[7];
572:
573:                ClassFile[13] * instance_class_file = id[14]->class_file[15];
574:                Result[16] result = instance_class_file[17]->get_method[18](utf8_name[10], utf8_descriptor[12], minfo[19]);
575:                if (result[20] != Success[21])
576:                {
577:                        print_error[22](result[20]);
578:                        return ResolutionCannotResolveInterfaceMethodref[7];
579:                }
580:
581:                return Success[21];
582:        }
583:
584:        void CONSTANT_InterfaceMethodref_info[23]::debug_print(ostream & out)
585:        {
586:                out[24] << "CONSTANT_InterfaceMethodref_info:" << endl;
587:                out[24] << "      class_index: " << class_index[25] << endl;
588:                out[24] << "      name_and_type_index: " << name_and_type_index[5] << endl;
589:        }
590:
591:
592:        Result[16] CONSTANT_String_info[26]::read(u1[27] *& stream_pointer)
593:        {
594:                memcpy_u2[28](&string_index[29], stream_pointer[30]);
595:                stream_pointer[30] += sizeof(u2[31]);
596:
597:                return Success[21];
598:        }
599:
600:        Result[16] CONSTANT_String_info[26]::resolve()
601:        {
602:                CONSTANT_Utf8_info[8] * utf8_info_string =
603:                        (CONSTANT_Utf8_info[8]*)class_file[2]->constant_pool[3]->get_item_at_index[4](string_index[29]);
604:                if (!utf8_info_string[32])
605:                        return ResolutionCannotInternStringObject[33];
606:
```

Footnotes:

1: *ConstantPool.h:258*
2: *ConstantPool.h:31*
3: *ClassFile.h:159*
4: *ConstantPool.h:299*
5: *ConstantPool.h:159*
6: *ConstantPool.cpp:557*
7: *defs.h:60*
8: *ConstantPool.h:186*
9: *ConstantPool.h:260*
10: *ConstantPool.cpp:563*
11: *ConstantPool.h:261*
12: *ConstantPool.cpp:568*
13: *ClassFile.h:136*
14: *ConstantPool.cpp:531*
15: *ObjectData.h:74*
16: *defs.h:29*
17: *ConstantPool.cpp:573*
18: *ClassFile.cpp:457*
19: *ConstantPool.cpp:531*
20: *ConstantPool.cpp:574*
21: *defs.h:33*
22: *errors.cpp:35*
23: *ConstantPool.h:156*
24: *ConstantPool.cpp:584*
25: *ConstantPool.h:158*
26: *ConstantPool.h:170*
27: *defs.h:168*
28: *memcpy_bigendian.cpp:10*
29: *ConstantPool.h:172*
30: *ConstantPool.cpp:592*
31: *defs.h:172*
32: *ConstantPool.cpp:602*
33: *defs.h:64*

```
607:                    HandlePool[1] * hp = class_file[2]->class_loader[3]->get_jvm[4]()->get_handle_pool[5]();
608:                    assert(hp[6] != NULL);
609:
610:                    // "Intern" this string (the function will simply return a reference to this string object
611:                    // if it is already interned)
612:                    interned_string_reference[7] = hp[6]->intern_string[8](utf8_info_string[9]);
613:                    if (interned_string_reference[7] == null[10])
614:                            return ResolutionCannotInternStringObject[11];
615:
616:                    resolved[12] = 1;
617:
618:                    return Success[13];
619:            }
620:
621:            void CONSTANT_String_info[14]::debug_print(ostream & out)
622:            {
623:                    out[15] << "CONSTANT_String_info:" << endl;
624:                    out[15] << "      string_index: " << string_index[16] << endl;
625:            }
626:
627:
628:            Result[17] CONSTANT_Utf8_info[18]::read(u1[19] *& stream_pointer)
629:            {
630:                    memcpy_u2[20](&length[21], stream_pointer[22]);
631:                    stream_pointer[22] += sizeof(u2[23]);
632:                    bytes[24] = new u1[19][length[21]];
633:                    memcpy(bytes[24], stream_pointer[22], length[21]);
634:                    stream_pointer[22] += length[21];
635:
636:                    return Success[13];
637:            }
638:
639:            Result[17] CONSTANT_Utf8_info[18]::get_string(wchar_t *& output)
640:            {
641:                    u1[19] b;
642:                    int index = 0;
643:                    unsigned char first, second, third;
644:                    output[25] = new wchar_t[length[21]+1]; // maximal possible length; should be wchar
645:                    int output_index = 0;
646:
647:                    while (index[26] < length[21])
648:                    {
649:                            b[27] = bytes[24][index[26]];
650:
651:                            if ((b[27] >> 7) == 0x0) // one byte case
652:                            {
653:                                    first[28] = b[27] & 0x7F;
654:                                    output[25][output_index[29]++] = first[28];
655:                                    index[26]++;
656:                            }
657:                            else if ((b[27] >> 5) == 0x6) // two bytes case
658:                            {
659:                                    first[28] = b[27] & 0x1F;
```

Footnotes:

1: *HandlePool.h:13*
2: *ConstantPool.h:31*
3: *ClassFile.h:222*
4: *ClassLoader.h:98*
5: *VirtualMachine.h:336*
6: *ConstantPool.cpp:607*
7: *ConstantPool.h:183*
8: *HandlePool.cpp:49*
9: *ConstantPool.cpp:602*
10: *defs.h:201*
11: *defs.h:64*
12: *ConstantPool.h:24*
13: *defs.h:33*
14: *ConstantPool.h:170*
15: *ConstantPool.cpp:621*
16: *ConstantPool.h:172*
17: *defs.h:29*
18: *ConstantPool.h:186*
19: *defs.h:168*
20: *memcpy_bigendian.cpp:10*
21: *ConstantPool.h:188*
22: *ConstantPool.cpp:628*
23: *defs.h:172*
24: *ConstantPool.h:189*
25: *ConstantPool.cpp:639*
26: *ConstantPool.cpp:642*
27: *ConstantPool.cpp:641*
28: *ConstantPool.cpp:643*
29: *ConstantPool.cpp:645*

```
660:                                    if (++index[1] >= length[2])
661:                                            return Utf8Error[3]; // no second byte
662:                                    b[4] = bytes[5][index[1]];
663:                                    if ((b[4] >> 6) == 0x2) // check second byte
664:                                    {
665:                                            second[6] = b[4] & 0x3F;
666:                                            output[7][output_index[8]++] = ((first[9] << 6) || second[6]);
667:                                            index[1]++;
668:                                    }
669:                                    else
670:                                            return Utf8Error[3]; // second byte is wrong
671:                            }
672:                            else if ((b[4] >> 4) == 0xE) // three bytes case
673:                            {
674:                                    first[9] = b[4] & 0xF;
675:                                    if (++index[1] >= length[2])
676:                                            return Utf8Error[3]; // no second byte
677:                                    b[4] = bytes[5][index[1]];
678:                                    if ((b[4] >> 6) == 0x2) // check second byte
679:                                    {
680:                                            second[6] = b[4] & 0x3F;
681:                                            if (++index[1] >= length[2])
682:                                                    return Utf8Error[3]; // no third byte
683:                                            b[4] = bytes[5][index[1]];
684:                                            if ((b[4] >> 6) == 0x2) // check third byte
685:                                            {
686:                                                    third[10] = b[4] & 0x3F;
687:                                                    output[7][output_index[8]++] = (first[9] << 12) || (second[6] << 6) || thir
        d[10];
688:                                                    index[1]++;
689:                                            }
690:                                            else
691:                                                    return Utf8Error[3]; // third byte is wrong
692:                                    }
693:                                    else
694:                                            return Utf8Error[3]; // second byte is wrong
695:                            }
696:                            else
697:                                    return Utf8Error[3]; // first byte is wrong
698:                    }
699:
700:                    // Put the terminating null
701:                    output[7][output_index[8]] = 0;
702:
703:                    return Success[11];
704:            }
705:
706:            // Special constructor ot make the UTF8 string from the given wide-char string.
707:            // The wide-char string is supposed to contain ascii characters only
708:            CONSTANT_Utf8_info[12]::CONSTANT_Utf8_info(ClassFile[13] * _class_file, wchar_t * wstring) : ConstantPoolEntry[14]
        (_class_file, CONSTANT_Utf8[15])
709:            {
710:                    length[2] = wcslen(wstring[16]);
```

Footnotes:

1: *ConstantPool.cpp:642*
2: *ConstantPool.h:188*
3: *defs.h:40*
4: *ConstantPool.cpp:641*
5: *ConstantPool.h:189*
6: *ConstantPool.cpp:643*
7: *ConstantPool.cpp:639*
8: *ConstantPool.cpp:645*
9: *ConstantPool.cpp:643*
10: *ConstantPool.cpp:643*
11: *defs.h:33*
12: *ConstantPool.h:186*
13: *ClassFile.h:136*
14: *ConstantPool.h:35*
15: *defs.h:336*
16: *ConstantPool.cpp:708*

```
711:                    bytes[1] = new u1[2][length[3]];
712:                    for (int i = 0; i < length[3]; i++)
713:                    {
714:                            char ch = (char)(wstring[4][i[5]] & 0xFF);
715:                            bytes[1][i[5]] = ch[6] & 0x7F; // make sure the first bit is 0
716:                    }
717:            }
718:
719:            Result[7] CONSTANT_Utf8_info[8]::resolve()
720:            {
721:                    return Success[9];
722:            }
723:
724:            int CONSTANT_Utf8_info[8]::operator==(const CONSTANT_Utf8_info[8] & utf8_info)
725:            {
726:                    if (length[3] != utf8_info[10].length[3])
727:                            return 0;
728:                    return !memcmp(bytes[1],utf8_info[10].bytes[1],length[3]);
729:            }
730:
731:            int CONSTANT_Utf8_info[8]::operator!=(const CONSTANT_Utf8_info[8] & utf8_info)
732:            {
733:                    return ! (*this == utf8_info[11]);
734:            }
735:
736:
737:            void CONSTANT_Utf8_info[8]::debug_print(ostream & out)
738:            {
739:                    wchar_t * output;
740:                    get_string[12](output[13]);
741:
742:                    out[14] << "CONSTANT_Utf8_info:" << endl;
743:                    out[14] << "      length: " << length[3] << endl;
744:                    //out << "      bytes: " << bytes << endl;
745:                    out[14] << "      display view (ASCII only): [";
746:                    for (int i = 0; i < length[3]; i++)
747:                    {
748:                            char c = (char)(output[13][i[15]] & 0xFF);
749:                            out[14] << c[16];
750:                    }
751:                    out[14] << "]" << endl;
752:
753:                    delete [] output[13];
754:            }
755:
756:            Result[7] CONSTANT_Integer_info[17]::read(u1[2] *& stream_pointer)
757:            {
758:                    memcpy_u4[18](&bytes[19], stream_pointer[20]);
759:                    stream_pointer[20] += sizeof(u4[21]);
760:
761:                    return Success[9];
762:            }
763:
```

Footnotes:

1: *ConstantPool.h:189*
2: *defs.h:168*
3: *ConstantPool.h:188*
4: *ConstantPool.cpp:708*
5: *ConstantPool.cpp:712*
6: *ConstantPool.cpp:714*
7: *defs.h:29*
8: *ConstantPool.h:186*
9: *defs.h:33*
10: *ConstantPool.cpp:724*
11: *ConstantPool.cpp:731*
12: *ConstantPool.cpp:639*
13: *ConstantPool.cpp:739*
14: *ConstantPool.cpp:737*
15: *ConstantPool.cpp:746*
16: *ConstantPool.cpp:748*
17: *ConstantPool.h:204*
18: *memcpy_bigendian.cpp:27*
19: *ConstantPool.h:206*
20: *ConstantPool.cpp:756*
21: *defs.h:174*

```
764:        Result[1] CONSTANT_Integer_info[2]::resolve()
765:        {
766:                return Success[3];
767:        }
768:
769:        void CONSTANT_Integer_info[2]::debug_print(ostream & out)
770:        {
771:                out[4] << "CONSTANT_Integer_info: " << get_int_value[5]() << endl;
772:        }
773:
774:
775:        Result[1] CONSTANT_NameAndType_info[6]::read(u1[7] *& stream_pointer)
776:        {
777:                memcpy_u2[8](&name_index[9], stream_pointer[10]);
778:                stream_pointer[10] += sizeof(u2[11]);
779:                memcpy_u2[8](&descriptor_index[12], stream_pointer[10]);
780:                stream_pointer[10] += sizeof(u2[11]);
781:
782:                return Success[3];
783:        }
784:
785:        Result[1] CONSTANT_NameAndType_info[6]::resolve()
786:        {
787:                return Success[3];
788:        }
789:
790:        void CONSTANT_NameAndType_info[6]::debug_print(ostream & out)
791:        {
792:                out[13] << "CONSTANT_NameAndType_info:" << endl;
793:                out[13] << "      name_index: " << name_index[9] << endl;
794:                out[13] << "      descriptor_index: " << descriptor_index[12] << endl;
795:        }
796:
797:
798:        Result[1] ConstantPool[14]::read_constant_pool_entry(u1[7] tag, u1[7] *& stream_pointer, ConstantPoolEntry[15] *& cpe)
799:        {
800:                        switch (tag[16])
801:                        {
802:                        case CONSTANT_Class[17]:
803:                                cpe[18] = new CONSTANT_Class_info[19](class_file[20]);
804:                                break;
805:                        case CONSTANT_Fieldref[21]:
806:                                cpe[18] = new CONSTANT_Fieldref_info[22](class_file[20]);
807:                                break;
808:                        case CONSTANT_Methodref[23]:
809:                                cpe[18] = new CONSTANT_Methodref_info[24](class_file[20]);
810:                                break;
811:                        case CONSTANT_InterfaceMethodref[25]:
812:                                cpe[18] = new CONSTANT_InterfaceMethodref_info[26](class_file[20]);
813:                                break;
814:                        case CONSTANT_String[27]:
815:                                cpe[18] = new CONSTANT_String_info[28](class_file[20]);
816:                                break;
```

Footnotes:

1: *defs.h:29*
2: *ConstantPool.h:204*
3: *defs.h:33*
4: *ConstantPool.cpp:769*
5: *ConstantPool.h:209*
6: *ConstantPool.h:258*
7: *defs.h:168*
8: *memcpy_bigendian.cpp:10*
9: *ConstantPool.h:260*
10: *ConstantPool.cpp:775*
11: *defs.h:172*
12: *ConstantPool.h:261*
13: *ConstantPool.cpp:790*
14: *ConstantPool.h:274*
15: *ConstantPool.h:20*
16: *ConstantPool.cpp:798*
17: *defs.h:326*
18: *ConstantPool.cpp:798*
19: *ConstantPool.h:63*
20: *ConstantPool.h:282*
21: *defs.h:327*
22: *ConstantPool.h:83*
23: *defs.h:328*
24: *ConstantPool.h:107*
25: *defs.h:329*
26: *ConstantPool.h:162*
27: *defs.h:330*
28: *ConstantPool.h:175*

```
817:                        case CONSTANT_Integer[1]:
818:                                cpe[2] = new CONSTANT_Integer_info[3](class_file[4]);
819:                                break;
820:                        case CONSTANT_Float[5]:
821:                                cpe[2] = new CONSTANT_Float_info[6](class_file[4]);
822:                                break;
823:                        case CONSTANT_Long[7]:
824:                                cpe[2] = new CONSTANT_Long_info[8](class_file[4]);
825:                                break;
826:                        case CONSTANT_Double[9]:
827:                                cpe[2] = new CONSTANT_Double_info[10](class_file[4]);
828:                                break;
829:                        case CONSTANT_NameAndType[11]:
830:                                cpe[2] = new CONSTANT_NameAndType_info[12](class_file[4]);
831:                                break;
832:                        case CONSTANT_Utf8[13]:
833:                                cpe[2] = new CONSTANT_Utf8_info[14](class_file[4]);
834:                                break;
835:                        default:
836:                                cpe[2] = NULL;
837:                                return ClassLoaderErrorWrongConstantPoolTag[15];
838:                        }
839:
840:                        cpe[2]->read(stream_pointer[16]);
841:
842:                        return Success[17];
843:        }
844:
845:
846:        Result[18] ConstantPool[19]::read(u1[20] *& stream_pointer)
847:        {
848:                for (int i = 1; i <= num_of_items[21];)
849:                {
850:                        u1[20] tag;
851:                        memcpy(&tag[22], stream_pointer[23], sizeof(u1[20]));
852:                        stream_pointer[23] += sizeof(u1[20]);
853:
854:                        ConstantPoolEntry[24] * cpe;
855:
856:                        Result[18] result = read_constant_pool_entry[25](tag[22], stream_pointer[23], cpe[26]);
857:                        if (result[27] != Success[17])
858:                                return result[27];
859:
860:                        table[28][i[29]] = cpe[26];
861:
862:                        if (tag[22] == CONSTANT_Long[7] ||
863:                                tag[22] == CONSTANT_Double[9])
864:                        {
865:                                i[29] += 2; // Long and Double constants occupy two entries of CP
866:                        }
867:                        else
868:                        {
869:                                i[29] += 1; // In normal case move forward to the next CP entry
```

Footnotes:

1: *defs.h:331*
2: *ConstantPool.cpp:798*
3: *ConstantPool.h:213*
4: *ConstantPool.h:282*
5: *defs.h:332*
6: *ConstantPool.h:225*
7: *defs.h:333*
8: *ConstantPool.h:238*
9: *defs.h:334*
10: *ConstantPool.h:252*
11: *defs.h:335*
12: *ConstantPool.h:264*
13: *defs.h:336*
14: *ConstantPool.h:194*
15: *defs.h:45*
16: *ConstantPool.cpp:798*
17: *defs.h:33*
18: *defs.h:29*
19: *ConstantPool.h:274*
20: *defs.h:168*
21: *ConstantPool.h:283*
22: *ConstantPool.cpp:850*
23: *ConstantPool.cpp:846*
24: *ConstantPool.h:20*
25: *ConstantPool.cpp:798*
26: *ConstantPool.cpp:854*
27: *ConstantPool.cpp:856*
28: *ConstantPool.h:284*
29: *ConstantPool.cpp:848*

```
870:                            }
871:                    }
872:
873:                    return Success[1];
874:            }
875:
876:            void ConstantPool[2]::debug_print(ostream & out)
877:            {
878:                    out[3] << "CONSTANT POOL:" << endl;
879:                    out[3] << "--------------------------------" << endl;
880:
881:                    for (int i = 1; i <= num_of_items[4];)
882:                    {
883:                            out[3] << "CP ENTRY # " << i[5] << endl;
884:                            table[6][i[5]]->debug_print(out[3]);
885:                            if (table[6][i[5]]->tag[7] == CONSTANT_Long[8] ||
886:                                    table[6][i[5]]->tag[7] == CONSTANT_Double[9])
887:                            {
888:                                    i[5] += 2; // Long and Double constants occupy two entries of CP
889:                            }
890:                            else
891:                            {
892:                                    i[5] += 1; // In normal case move forward to the next CP entry
893:                            }
894:
895:                    }
896:
897:                    out[3] << "--------------------------------" << endl;
898:            }
```

Footnotes:

1: *defs.h:33*
2: *ConstantPool.h:274*
3: *ConstantPool.cpp:876*
4: *ConstantPool.h:283*
5: *ConstantPool.cpp:881*
6: *ConstantPool.h:284*
7: *ConstantPool.h:28*
8: *defs.h:333*
9: *defs.h:334*

```
 1:  #include <assert.h>
 2:  #include <iostream.h>
 3:
 4:  #include "GarbageCollector.h"
 5:  #include "VirtualMachine.h"
 6:  #include "ObjectData.h"
 7:  #include "HandlePool.h"
 8:  #include "ClassLoader.h"
 9:  #include "ClassFile.h"
10:
11:  Result[1] GarbageCollector[2]::step()
12:  {
13:          run_counter[3]++;
14:
15:  #ifdef DEBUG_GC[4]
16:          cout << "[GC] pass: " << run_counter[3] << endl;
17:  #endif // DEBUG_GC
18:
19:          HandlePool[5] * handle_pool = vm[6]->get_handle_pool[7]();
20:
21:          //------------------------------------------------------------------------
22:          // "Mark" process
23:          //------------------------------------------------------------------------
24:
25:          // Traverse all reachable instances and mark their memory chunks with
26:          // the current run counter
27:
28:          HashTableIterator[8] iterator(handle_pool[9]->roots[10]);
29:          HashTable[11]::HashTableEntry[12] * hte;
30:          while (hte[13]=iterator[14].next[15]())
31:          {
32:                  // Get root
33:                  InstanceData[16] * id = (InstanceData[16]*)hte[13]->value[17];
34:                  assert(id[18] != NULL);
35:                  memory_chunk[19] * chunk = id[18]->data_start[20];
36:                  if (!chunk[21])
37:                          continue;
38:
39:                  chunk[21]->mark[22](run_counter[3]);
40:
41:                  // Recursively visit all the fields of type reference
42:                  // and mark them as reachable too
43:                  mark_reference_fields[23](id[18]);
44:          }
45:
46:          // Traverse all reachable class data memory chunks
47:
48:          Vector[24]<ClassLoader*> * class_loaders = vm[6]->get_class_loaders[25]();
49:          for (int i = 0; i < class_loaders[26]->size[27](); i++)
50:          {
51:                  ClassLoader[28] * class_loader = class_loaders[26]->element_at[29](i[30]);
52:                  HashTable[11] * current_namespace = class_loader[31]->get_namespace[32]();
53:
```

Footnotes:

1: *defs.h:29*
2: *GarbageCollector.h:19*
3: *GarbageCollector.h:24*
4: *defs.h:446*
5: *HandlePool.h:13*
6: *Thread.h:32*
7: *VirtualMachine.h:336*
8: *hashtable.h:88*
9: *GarbageCollector.cpp:19*
10: *HandlePool.h:37*
11: *hashtable.h:28*
12: *hashtable.h:32*
13: *GarbageCollector.cpp:29*
14: *GarbageCollector.cpp:28*
15: *hashtable.cpp:153*
16: *ObjectData.h:113*
17: *hashtable.h:35*
18: *GarbageCollector.cpp:33*
19: *HeapManager.h:44*
20: *ObjectData.h:71*
21: *GarbageCollector.cpp:35*
22: *HeapManager.h:70*
23: *GarbageCollector.cpp:141*
24: *vector.h:7*
25: *VirtualMachine.h:338*
26: *GarbageCollector.cpp:48*
27: *vector.h:23*
28: *ClassLoader.h:14*
29: *vector.h:36*
30: *GarbageCollector.cpp:49*
31: *GarbageCollector.cpp:51*
32: *ClassLoader.h:93*

```
54:                         // Traverse all class files in the current class loader's namespace
55:                         HashTableIterator[1] iterator(*current_namespace[2]);
56:                         HashTable[3]::HashTableEntry[4] * hte;
57:                         while (hte[5]=iterator[6].next[7]())
58:                         {
59:                                 ClassFile[8] * cf = (ClassFile[8]*)hte[5]->value[9];
60:                                 assert(cf[10] != NULL);
61:                                 ClassData[11] * cd = cf[10]->class_data[12];
62:                                 assert(cd[13] != NULL);
63:                                 memory_chunk[14] * chunk = cd[13]->data_start[15];
64:                                 if (!chunk[16])
65:                                         continue;
66:
67:                                 chunk[16]->mark[17](run_counter[18]);
68:                         }
69:                 }
70:
71:                 //------------------------------------------------------------------------
72:                 // "Sweep" process
73:                 //------------------------------------------------------------------------
74:
75:                 unsigned long deallocated_instance_data;
76:                 unsigned long deallocated_class_data;
77:
78:                 // Traverse all memory chunks and deallocate those whose mark counter is less
79:                 // than the current garbage collector's run counter (which means these chunks
80:                 // were not reachable during this pass)
81:                 deallocated_instance_data[19] = vm[20]->get_instance_heap_manager[21]()->sweep[22](run_counter[18]);
82:
83:                 // If two heaps are different, "sweep" the class heap too;
84:                 // otherwise do nothing
85:                 if (vm[20]->get_class_heap_manager[23]() != vm[20]->get_instance_heap_manager[21]())
86:                         deallocated_class_data[24] = vm[20]->get_class_heap_manager[23]()->sweep[22](run_counter[18]);
87:
88:         #ifdef DEBUG_GC[25]
89:                 cout << "[GC] pass: " << "deallocated " << deallocated_instance_data[19] << " objects" << endl;
90:                 if (deallocated_instance_data[19] > 0)
91:                         vm[20]->get_instance_heap_manager[21]()->dump[26]();
92:         #endif // DEBUG_GC
93:
94:                 return Success[27];
95:         }
96:
97:
98:         void GarbageCollector[28]::mark_reference_fields(ClassFile[8] * cf, InstanceData[29] * id)
99:         {
100:                HandlePool[30] * handle_pool = vm[20]->get_handle_pool[31]();
101:
102:                // For all fields of this instance's class
103:
104:                for (int i = 0; i < cf[32]->fields_count[33]; i++)
105:                {
106:                        field_info[34] * finfo = cf[32]->fields[35][i[36]];
```

Footnotes:

1: *hashtable.h:88*
2: *GarbageCollector.cpp:52*
3: *hashtable.h:28*
4: *hashtable.h:32*
5: *GarbageCollector.cpp:56*
6: *GarbageCollector.cpp:55*
7: *hashtable.cpp:153*
8: *ClassFile.h:136*
9: *hashtable.h:35*
10: *GarbageCollector.cpp:59*
11: *ObjectData.h:92*
12: *ClassFile.h:227*
13: *GarbageCollector.cpp:61*
14: *HeapManager.h:44*
15: *ObjectData.h:71*
16: *GarbageCollector.cpp:63*
17: *HeapManager.h:70*
18: *GarbageCollector.h:24*
19: *GarbageCollector.cpp:75*
20: *Thread.h:32*
21: *VirtualMachine.h:332*
22: *HeapManager.cpp:265*
23: *VirtualMachine.h:331*
24: *GarbageCollector.cpp:76*
25: *defs.h:446*
26: *HeapManager.cpp:290*
27: *defs.h:33*
28: *GarbageCollector.h:19*
29: *ObjectData.h:113*
30: *HandlePool.h:13*
31: *VirtualMachine.h:336*
32: *GarbageCollector.cpp:98*
33: *ClassFile.h:191*
34: *ClassFile.h:68*
35: *ClassFile.h:197*
36: *GarbageCollector.cpp:104*

```
107:                        assert(finfo[1] != NULL);
108:
109:                        if (finfo[1]->access_flags[2] & ACC_STATIC[3])
110:                                continue; // nothing to do here. TODO: check
111:
112:                        if (!(finfo[1]->type[4] == Reference[5]) && !(finfo[1]->type[4] == Array[6]))
113:                                continue;
114:
115:                        u1[7] * field_start = id[8]->data_start[9]->address[10] + finfo[1]->offset[11];
116:                        word[12] reference;
117:                        memcpy(&reference[13], field_start[14], sizeof(word[12]));
118:                        if (reference[13] == null[15])
119:                                continue;
120:
121:                        InstanceData[16] * field_id = handle_pool[17]->get_instance[18](reference[13]);
122:                        assert(field_id[19] != NULL);
123:
124:                        // Mark the instance itself
125:                        memory_chunk[20] * chunk = field_id[19]->data_start[9];
126:                        if (!chunk[21])
127:                                continue;
128:
129:                        // To avoid cycle reference problem check whether this instance was already marked
130:                        if (chunk[21]->mark_counter[22] == run_counter[23])
131:                                continue; // already visited
132:
133:                        // Otherwise, mark this memory chunk and visit all its fields
134:                        chunk[21]->mark[24](run_counter[23]);
135:
136:                        // Recursively mark all this instance fields
137:                        mark_reference_fields[25](field_id[19]);
138:                }
139:        }
140:
141:        void GarbageCollector[26]::mark_reference_fields(InstanceData[16] * id)
142:        {
143:                HandlePool[27] * handle_pool = vm[28]->get_handle_pool[29]();
144:
145:                ClassFile[30] * cf = id[31]->class_file[32];
146:                assert (cf[33] != NULL);
147:
148:                // Mark all fields belonging to this class;
149:                // this call will recursively call this variant of "mark_reference_fields"
150:                // given instance data residing in the fields of the class
151:                mark_reference_fields[25](cf[33],id[31]);
152:
153:                if (cf[33]->super_class[34] == 0) // we are in java.lang.Object
154:                        return;
155:
156:                // Look into all the superinterfaces of the current class
157:                for (int i = 0; i < cf[33]->interfaces_count[35]; i++)
158:                {
                        CONSTANT_Class_info[36] * super_interface_entry = (CONSTANT_Class_info[36]*)cf[33]->constant_pool[37]
```

Footnotes:

1: *GarbageCollector.cpp:106*
2: *ClassFile.h:37*
3: *defs.h:364*
4: *ClassFile.h:72*
5: *defs.h:216*
6: *defs.h:219*
7: *defs.h:168*
8: *GarbageCollector.cpp:98*
9: *ObjectData.h:71*
10: *HeapManager.h:46*
11: *ClassFile.h:79*
12: *defs.h:195*
13: *GarbageCollector.cpp:116*
14: *GarbageCollector.cpp:115*
15: *defs.h:201*
16: *ObjectData.h:113*
17: *GarbageCollector.cpp:100*
18: *HandlePool.cpp:25*
19: *GarbageCollector.cpp:121*
20: *HeapManager.h:44*
21: *GarbageCollector.cpp:125*
22: *HeapManager.h:55*
23: *GarbageCollector.h:24*
24: *HeapManager.h:70*
25: *GarbageCollector.cpp:141*
26: *GarbageCollector.h:19*
27: *HandlePool.h:13*
28: *Thread.h:32*
29: *VirtualMachine.h:336*
30: *ClassFile.h:136*
31: *GarbageCollector.cpp:141*
32: *ObjectData.h:74*
33: *GarbageCollector.cpp:145*
34: *ClassFile.h:178*
35: *ClassFile.h:181*
36: *ConstantPool.h:58*
37: *ClassFile.h:159*

```
        ->get_item_at_index[1](cf[2]->interfaces[3][i[4]]);
160:                    // NOTE: since the current class is already resolved, all its superinterfaces must be res
        olved too;
161:                    assert(super_interface_entry[5]->is_resolved[6]()); // should NEVER happen
162:
163:                    // This call will recursively investigate all superinterfaces
164:                    mark_reference_fields[7](super_interface_entry[5]->resolved_class_file[8],id[9]);
165:            }
166:
167:            // Look into the superclass
168:            CONSTANT_Class_info[10] * super_class_entry = (CONSTANT_Class_info[10]*)cf[2]->constant_pool[11]->get_item_a
        t_index[1](cf[2]->super_class[12]);
169:            // NOTE: since the current class is already resolved, its superclass must be resolved too;
170:            assert(super_class_entry[13]->is_resolved[6]()); // should NEVER fail
171:
172:            // This call will recursively investigate all the superclasses
173:            mark_reference_fields[7](super_class_entry[13]->resolved_class_file[8],id[9]);
174:    }
175:
159:
```

Footnotes:

1: *ConstantPool.h:299*
2: *GarbageCollector.cpp:145*
3: *ClassFile.h:187*
4: *GarbageCollector.cpp:157*
5: *GarbageCollector.cpp:159*
6: *ConstantPool.h:48*
7: *GarbageCollector.cpp:141*
8: *ConstantPool.h:71*
9: *GarbageCollector.cpp:141*
10: *ConstantPool.h:58*
11: *ClassFile.h:159*
12: *ClassFile.h:178*
13: *GarbageCollector.cpp:168*

```
 1:          #include <stdlib.h>
 2:          #include <memory.h>
 3:
 4:          #include "defs.h"
 5:          #include "HandlePool.h"
 6:          #include "ObjectData.h"
 7:          #include "Thread.h"
 8:          #include "ClassLoader.h"
 9:
10:
11:          // Stores the new instance in the pool and returns the abstract handle (reference) for it
12:          word[1] HandlePool[2]::put_instance(InstanceData[3] * id)
13:          {
14:                  char key[11]; // the length of string "4294967296"
15:                  word[1] reference = counter[4]++;
16:                  ltoa(reference[5], key[6], 10);
17:                  pool[7].insert[8](key[6], (void*)id[9]);
18:                  id[9]->reference[10] = reference[5];
19:
20:                  return reference[5];
21:          }
22:
23:          // Given a reference returns the native pointer to the instance data;
24:          // NOTE: returning NULL does not mean that this is the null reference !
25:          InstanceData[3] * HandlePool[2]::get_instance(word[1] reference)
26:          {
27:                  char key[11]; // the length of string "4294967296"
28:                  ltoa(reference[11], key[12], 10);
29:                  return (InstanceData[3]*)pool[7].find[13](key[12]);
30:          }
31:
32:          // Given an instance data of a String object returns the character representation
33:          // of the interned string
34:          char * HandlePool[2]::get_interned_string(InstanceData[3] * id)
35:          {
36:                  HashTableIterator[14] iterator(interned_strings[15]);
37:                  HashTable[16]::HashTableEntry[17] * hte;
38:                  while (hte[18]=iterator[19].next[20]())
39:                  {
40:                          InstanceData[3] * current_id = (InstanceData[3]*)hte[18]->value[21];
41:                          assert(current_id[22] != NULL);
42:                          if (current_id[22]->get_reference[23]() == id[24]->get_reference[23]())
43:                                  return hte[18]->key[25];
44:                  }
45:
46:                  return NULL;
47:          }
48:
49:          word[1] HandlePool[2]::intern_string(CONSTANT_Utf8_info[26] * utf8_info_string)
50:          {
51:                  char * ascii_string = new char[utf8_info_string[27]->length[28]];
52:                  wchar_t * wstring;
53:                  utf8_info_string[27]->get_string[29](wstring[30]);
```

Footnotes:

1: *defs.h:195*
2: *HandlePool.h:13*
3: *ObjectData.h:113*
4: *HandlePool.h:23*
5: *HandlePool.cpp:15*
6: *HandlePool.cpp:14*
7: *HandlePool.h:21*
8: *hashtable.cpp:23*
9: *HandlePool.cpp:12*
10: *ObjectData.h:121*
11: *HandlePool.cpp:25*
12: *HandlePool.cpp:27*
13: *hashtable.cpp:79*
14: *hashtable.h:88*
15: *HandlePool.h:30*
16: *hashtable.h:28*
17: *hashtable.h:32*
18: *HandlePool.cpp:37*
19: *HandlePool.cpp:36*
20: *hashtable.cpp:153*
21: *hashtable.h:35*
22: *HandlePool.cpp:40*
23: *ObjectData.h:142*
24: *HandlePool.cpp:34*
25: *hashtable.h:34*
26: *ConstantPool.h:186*
27: *HandlePool.cpp:49*
28: *ConstantPool.h:188*
29: *ConstantPool.cpp:639*
30: *HandlePool.cpp:52*

```
54:                 wchar2ascii[1](wstring[2], ascii_string[3], utf8_info_string[4]->length[5]);
55:         // TODO:        delete [] wstring;
56:
57:                 InstanceData[6] * id = (InstanceData[6]*)interned_strings[7].find[8](ascii_string[3]);
58:                 if (id[9])
59:                 {
60:                         // The string is already interned
61:                         return id[9]->reference[10];
62:                 }
63:
64:                 // Create a new java.lang.String object
65:                 ClassFile[11] * string_class_file =
66:                         vm[12]->get_bootstrap_class_loader[13]()->get_class[14](STRING_CLASS_NAME[15], sizeof(STRING_CLASS_NA
        ME[15])/sizeof(wchar_t));
67:
68:             if (!string_class_file[16])
69:                 {
70:                         // ATTENTION: in the first attempt to intern a string literal the String class file
71:                         // is not defined yet, so the function will try to load it.
72:                         // But construction of the String type itself (namely, its <clinit> constructor)
73:                         // will try to intern another string literal ("" - an empty string).
74:                         // To avoid endless recursion it is important that at the moment the function tries
75:                         // to intern String's own string the String class file will be already defined
76:                         // and this "if" will not be executing
77:
78:                         // java.lang.String class is not defined yet
79:                         string_class_file[16] =
80:                         vm[12]->get_bootstrap_class_loader[13]()->load_class[17](STRING_CLASS_NAME[15], sizeof(STRING_CLASS_N
        AME[15])/sizeof(wchar_t));
81:                         if (!string_class_file[16])
82:                                 return null[18];
83:                 }
84:
85:                 // Create a new instance of type String
86:                 word[19] reference = string_class_file[16]->create_new_instance[20]();
87:                 if (reference[21] == null[18])
88:                         return null[18];
89:
90:                 // Now comes the trick: we have to artificially initialize this String's character array
91:                 // without calling any constructors
92:
93:                 // first, create the array that will correspond to the String's "char value[]" field:
94:                 // 1) create the "char[]" type if not yet created
95:                 ClassFile[11] * char_array_class_file;
96:                 vm[12]->get_bootstrap_class_loader[13]()->define_primitive_array_class[22](T_CHAR[23], char_array_class_file[24]
        );
97:
98:                 // 2) now create the instance of this char array with the appropriate length
99:                 word[19] char_array_ref = ((PrimitiveArrayClassFile[25]*)char_array_class_file[24])->create_new_instance[26](
        utf8_info_string[4]->length[5]);
100:                 if (char_array_ref[27] == null[18])
101:                         return null[18];
102:
```

Footnotes:

1: *util.cpp:8*
2: *HandlePool.cpp:52*
3: *HandlePool.cpp:51*
4: *HandlePool.cpp:49*
5: *ConstantPool.h:188*
6: *ObjectData.h:113*
7: *HandlePool.h:30*
8: *hashtable.cpp:79*
9: *HandlePool.cpp:57*
10: *ObjectData.h:121*
11: *ClassFile.h:136*
12: *HandlePool.h:18*
13: *VirtualMachine.h:334*
14: *ClassLoader.cpp:187*
15: *defs.h:254*
16: *HandlePool.cpp:65*
17: *ClassLoader.cpp:18*
18: *defs.h:201*
19: *defs.h:195*
20: *ClassFile.cpp:870*
21: *HandlePool.cpp:86*
22: *ClassLoader.cpp:341*
23: *defs.h:341*
24: *HandlePool.cpp:95*
25: *ClassFile.h:370*
26: *ClassFile.cpp:1012*
27: *HandlePool.cpp:99*

```
103:                    // 3) get the instance of this array
104:                    InstanceData[1] * char_array_id = get_instance[2](char_array_ref[3]);
105:                    u1[4] * data_start = (u1[4]*)(char_array_id[5]->data_start[6]->address[7]);
106:
107:                    // 4) fill the array with the characters of the string
108:                    for (int i = 0; i < utf8_info_string[8]->length[9]; i++)
109:                    {
110:                            memcpy((data_start[10]+i[11]*sizeof(wchar_t)),&wstring[12][i[11]],sizeof(wchar_t));
111:                            //data_start[i] = (u1)ascii_string[i];
112:                    }
113:
114:                    // 5) assign this array's reference to the appropriate field "value"
115:                    // of the created String object
116:                    InstanceData[1] * string_id = get_instance[2](reference[13]);
117:                    assert(string_id[14] != NULL);
118:                    field_info[15] * finfo = NULL;
119:                    string_id[14]->class_file[16]->get_field_by_name[17](STRING_CLASS_VALUE_NAME[18], STRING_CLASS_VALUE_DESCRIPT
            OR[19], finfo[20]);
120:                    assert(finfo[20] != NULL);
121:                    memcpy(string_id[14]->data_start[6]->address[7] + finfo[20]->offset[21], &char_array_ref[3], sizeof(word[22]));
122:
123:                    // 6) assign the "offset" field of the String with 0
124:                    string_id[14]->class_file[16]->get_field_by_name[17](STRING_CLASS_OFFSET_NAME[23], STRING_CLASS_OFFSET_DESCRI
            PTOR[24], finfo[20]);
125:                    assert(finfo[20] != NULL);
126:                    int offset_value = 0;
127:                    memcpy(string_id[14]->data_start[6]->address[7] + finfo[20]->offset[21], &offset_value[25], sizeof(offset_value[25]))
            ;
128:
129:                    // 7) finally, assign the "count" field of the string to the actual length
130:                    string_id[14]->class_file[16]->get_field_by_name[17](STRING_CLASS_COUNT_NAME[26], STRING_CLASS_COUNT_DESCRIPT
            OR[27], finfo[20]);
131:                    assert(finfo[20] != NULL);
132:                    int count_value = utf8_info_string[8]->length[9];
133:                    memcpy(string_id[14]->data_start[6]->address[7] + finfo[20]->offset[21], &count_value[28], sizeof(count_value[28]));
134:
135:                    // the string is initialized !
136:
137:                    // "intern" this string
138:                    interned_strings[29].insert[30](ascii_string[31], string_id[14]);
139:
140:                    return reference[13];
141:            }
142:
143:            // Called when a stack frame pushes a reference to its operand stack or stores
144:            // a reference into its local variable
145:            void HandlePool[32]::put_root(InstanceData[1] * root)
146:            {
147:                    char key[11]; // the length of string "4294967296"
148:                    ltoa(root[33]->get_reference[34](), key[35], 10);
149:                    roots[36].insert[37](key[35], root[33]);
150:            }
151:
```

Footnotes:

1: *ObjectData.h:113*
2: *HandlePool.cpp:25*
3: *HandlePool.cpp:99*
4: *defs.h:168*
5: *HandlePool.cpp:104*
6: *ObjectData.h:71*
7: *HeapManager.h:46*
8: *HandlePool.cpp:49*
9: *ConstantPool.h:188*
10: *HandlePool.cpp:105*
11: *HandlePool.cpp:108*
12: *HandlePool.cpp:52*
13: *HandlePool.cpp:86*
14: *HandlePool.cpp:116*
15: *ClassFile.h:68*
16: *ObjectData.h:74*
17: *ClassFile.cpp:698*
18: *defs.h:267*
19: *defs.h:268*
20: *HandlePool.cpp:118*
21: *ClassFile.h:79*
22: *defs.h:195*
23: *defs.h:269*
24: *defs.h:270*
25: *HandlePool.cpp:126*
26: *defs.h:271*
27: *defs.h:272*
28: *HandlePool.cpp:132*
29: *HandlePool.h:30*
30: *hashtable.cpp:23*
31: *HandlePool.cpp:51*
32: *HandlePool.h:13*
33: *HandlePool.cpp:145*
34: *ObjectData.h:142*
35: *HandlePool.cpp:147*
36: *HandlePool.h:37*
37: *hashtable.cpp:171*

```
152:        // Called when a stack frame is destroyed together with all its "roots".
153:        // This causes an overhead when returning from a method but it pays back
154:        // when garbage collector is working
155:        void HandlePool[1]::remove_roots(Vector[2]<InstanceData *> * frame_roots)
156:        {
157:                for (int i = 0; i < frame_roots[3]->size[4](); i++)
158:                {
159:                        word[5] reference = frame_roots[3]->element_at[6](i[7])->get_reference();
160:                        char key[11]; // the length of string "4294967296"
161:                        ltoa(reference[8], key[9], 10);
162:                        roots[10].remove[11](key[9]);
163:                }
164:        }
165:
166:
```

Footnotes:

1: *HandlePool.h:13*
2: *vector.h:7*
3: *HandlePool.cpp:155*
4: *vector.h:23*
5: *defs.h:195*
6: *vector.h:36*
7: *HandlePool.cpp:157*
8: *HandlePool.cpp:159*
9: *HandlePool.cpp:160*
10: *HandlePool.h:37*
11: *hashtable.cpp:54*

```
 1:
 2:          #include <memory.h>
 3:          #include <stdlib.h>
 4:          #include <iostream.h>
 5:
 6:          #include "HeapManager.h"
 7:          #include "dlist.h"
 8:
 9:          void memory_chunk[1]::debug_print()
10:          {
11:                  if (free[2])
12:                          cout << "   FREE ";
13:                  else
14:                          cout << "   OCCUPIED ";
15:                  cout << size[3] << endl;
16:          }
17:
18:          // Initial size of the heap to allocate at once
19:          Result[4] HeapManager[5]::init(unsigned long bytes)
20:          {
21:                  // heap_start will never move
22:                  heap_start[6] = malloc(bytes[7]);
23:                  if (heap_start[6] == NULL)
24:                          return HeapErrorCannotAllocateInitialSize[8];
25:
26:                  // put first memory chunk into the list. It will contain the whole heap
27:                  memory_chunk[1] * chunk = new memory_chunk[9]((u1[10]*)heap_start[6], bytes[7], 1);
28:                  chunk_list[11].add_node[12](chunk[13]);
29:
30:                  heap_size[14] = bytes[7];
31:
32:                  return Success[15];
33:          }
34:
35:          Result[4] HeapManager[5]::clean_up()
36:          {
37:                  node[16] * current = chunk_list[11].get_first_node[17]();
38:                  while (current[18])
39:                  {
40:                          delete (memory_chunk[1]*)(current[18]->info[19]);
41:                          current[18] = chunk_list[11].get_next_node[20](current[18]);
42:                  }
43:
44:                  chunk_list[11].remove_all_nodes[21]();
45:
46:                  memory_chunk[1] * chunk = new memory_chunk[9]((u1[10]*)heap_start[6], heap_size[14], 1);
47:                  chunk_list[11].add_node[12](chunk[22]);
48:
49:                  return Success[15];
50:          }
51:
52:          HeapManager[5]::~HeapManager()
53:          {
```

Footnotes:

1: *HeapManager.h:44*
2: *HeapManager.h:47*
3: *HeapManager.h:48*
4: *defs.h:29*
5: *HeapManager.h:81*
6: *HeapManager.h:87*
7: *HeapManager.cpp:19*
8: *defs.h:54*
9: *HeapManager.h:60*
10: *defs.h:168*
11: *HeapManager.h:88*
12: *dlist.h:34*
13: *HeapManager.cpp:27*
14: *HeapManager.h:86*
15: *defs.h:33*
16: *dlist.h:7*
17: *dlist.h:49*
18: *HeapManager.cpp:37*
19: *dlist.h:11*
20: *dlist.h:59*
21: *dlist.h:137*
22: *HeapManager.cpp:46*

```
54:                 clean_up[1]();
55:                 // free the memory
56:                 free(heap_start[2]);
57:         }
58:
59:         // This function will use the "first fit" algorithm to allocate the object.
60:         Result[3] HeapManager[4]::alloc(unsigned long bytes, memory_chunk[5] *& new_one)
61:         {
62:                 node[6] * current = chunk_list[7].get_first_node[8]();
63:                 while (1)
64:                 {
65:                         memory_chunk[5] * chunk = (memory_chunk[5]*)current[9]->info[10];
66:                         if (!chunk[11]->free[12])
67:                         {
68:                                 current[9] = chunk_list[7].get_next_node[13](current[9]);
69:                                 continue;
70:                         }
71:                         else // this chunk is free
72:                         {
73:                                 if (chunk[11]->size[14] >= bytes[15])
74:                                 {
75:                                         // there is enough room to allocate the new object
76:                                         new_one[16] = new memory_chunk[17](chunk[11]->address[18], bytes[15], 0);
77:                                         chunk_list[7].insert_node_before[19](current[9], new_one[16]);
78:
79:                                         // decrease the size of the free chunk
80:                                         chunk[11]->size[14] -= bytes[15];
81:                                         // and point it to the new address
82:                                         chunk[11]->address[18] += bytes[15];
83:
84:                                         // check whether the free chunk disappeared
85:                                         if (chunk[11]->size[14] == 0)
86:                                         {
87:                                                 chunk_list[7].remove_node[20](current[9]);
88:                                         }
89:                                         // we've done
90:                                         return Success[21];
91:                                 }
92:                                 else // there is not enough room
93:                                 {
94:                                         current[9] = chunk_list[7].get_next_node[13](current[9]);
95:
96:                                         // check whether the heap is over
97:                                         if (!current[9])
98:                                         {
99:                                                 // there is no free chunk left that is suitable for this
100:                                                // allocation. Try to rearrange the heap
101:                                                if (just_in_case_defragment[22](bytes[15]) == Success[21])
102:                                                        // and start from the beginning
103:                                                        current[9] = chunk_list[7].get_first_node[8]();
104:                                                else
105:                                                        // the heap was defragmented but there is still no space
106:                                                        return HeapErrorCannotAllocateMemory[23];
```

Footnotes:

1: *HeapManager.cpp:35*
2: *HeapManager.h:87*
3: *defs.h:29*
4: *HeapManager.h:81*
5: *HeapManager.h:44*
6: *dlist.h:7*
7: *HeapManager.h:88*
8: *dlist.h:49*
9: *HeapManager.cpp:62*
10: *dlist.h:11*
11: *HeapManager.cpp:65*
12: *HeapManager.h:47*
13: *dlist.h:59*
14: *HeapManager.h:48*
15: *HeapManager.cpp:60*
16: *HeapManager.cpp:60*
17: *HeapManager.h:60*
18: *HeapManager.h:46*
19: *dlist.h:98*
20: *dlist.h:114*
21: *defs.h:33*
22: *HeapManager.cpp:199*
23: *defs.h:55*

```
107:                                    }
108:                                    continue;
109:                            }
110:                    }
111:
112:            }
113:    }
114:
115:    // Note: no two free memory chunks can be found together in the chunk list,
116:    // they must be merged in the dealloc function
117:    Result[1] HeapManager[2]::dealloc(memory_chunk[3] * this_chunk)
118:    {
119:            memory_chunk[3] * prev_chunk;
120:            memory_chunk[3] * next_chunk;
121:
122:            node[4] * to_delete = chunk_list[5].find_node[6](this_chunk[7]);
123:            if (to_delete[8] == NULL)
124:                    return HeapErrorCannotFindMemoryChunkToDeallocate[9];
125:
126:            node[4] * prev_node = to_delete[8]->prev[10];
127:            node[4] * next_node = to_delete[8]->next[11];
128:
129:            if (prev_node[12])
130:                    prev_chunk[13] = (memory_chunk[3]*)prev_node[12]->info[14];
131:            if (next_node[15])
132:                    next_chunk[16] = (memory_chunk[3]*)next_node[15]->info[14];
133:
134:            this_chunk[7]->free[17] = 1; // in any case
135:
136:            if (prev_node[12] && next_node[15]) // if both previous and next chunks exist
137:            {
138:                    if (!prev_chunk[13]->free[17])
139:                    {
140:                            // prev is occupied - proceed to check next
141:
142:                            if (next_chunk[16]->free[17])
143:                            {
144:                                    // next chunk is free - merge current and next chunks
145:                                    this_chunk[7]->size[18] += next_chunk[16]->size[18];
146:                                    chunk_list[5].remove_node[19](next_node[15]);
147:                                    delete next_node[15]->info[14];
148:                            }
149:                            // if next chunk is not free there is nothing to do
150:                    }
151:                    else
152:                    {
153:                            // prev is free - merge two chunks, delete currentNode
154:                            // and proceed to check next
155:                            prev_chunk[13]->size[18] += this_chunk[7]->size[18];
156:                            chunk_list[5].remove_node[19](to_delete[8]);
157:                            delete to_delete[8]->info[14];
158:
159:                            if (next_chunk[16]->free[17])
```

Footnotes:

1: *defs.h:29*
2: *HeapManager.h:81*
3: *HeapManager.h:44*
4: *dlist.h:7*
5: *HeapManager.h:88*
6: *dlist.h:69*
7: *HeapManager.cpp:117*
8: *HeapManager.cpp:122*
9: *defs.h:56*
10: *dlist.h:10*
11: *dlist.h:9*
12: *HeapManager.cpp:126*
13: *HeapManager.cpp:119*
14: *dlist.h:11*
15: *HeapManager.cpp:127*
16: *HeapManager.cpp:120*
17: *HeapManager.h:47*
18: *HeapManager.h:48*
19: *dlist.h:114*

```
160:                                    {
161:                                            // next chunk is free - merge all these chunks (prev and current already
        merged)
162:                                            // and delete next node
163:                                            prev_chunk[1]->size[2] += next_chunk[3]->size[2];
164:                                            chunk_list[4].remove_node[5](next_node[6]);
165:                                            delete next_node[6]->info[7];
166:                                    }
167:                                    // if next chunk is not free there is nothing to do
168:                            }
169:                    }
170:
171:                    else if (prev_node[8])     // only previous chunk exists
172:                    {
173:                            if (prev_chunk[1]->free[9])
174:                            {
175:                                    // prev is free - merge two chunks, delete current node
176:                                    prev_chunk[1]->size[2] += this_chunk[10]->size[2];
177:                                    chunk_list[4].remove_node[5](to_delete[11]);
178:                                    delete to_delete[11]->info[7];
179:                            }
180:                            // if the previous chunk is occupied there is nothing to do
181:                    }
182:
183:                    else if (next_node[6]) // only next chunk exists
184:                    {
185:                            if (next_chunk[3]->free[9])
186:                            {
187:                                    // next is free - merge two chunks and delete next node
188:                                    this_chunk[10]->size[2] += next_chunk[3]->size[2];
189:                                    chunk_list[4].remove_node[5](next_node[6]);
190:                                    delete next_node[6]->info[7];
191:                            }
192:                            // if the next chunk is occupied there is nothing to do
193:                    }
194:                    // if no previous or next chunks exist there is nothing to do
195:
196:                    return Success[12];
197:            }
198:
199:            Result[13] HeapManager[14]::just_in_case_defragment(unsigned long bytes_needed)
200:            {
201:                    node[15] * current_node = chunk_list[4].get_first_node[16]();
202:
203:                    while (1)
204:                    {
205:                            memory_chunk[17] * current_chunk = (memory_chunk[17]*)current_node[18]->info[7];
206:                            if (current_chunk[19]->free[9])
207:                            {
208:                                    node[15] * next_node = chunk_list[4].get_next_node[20](current_node[18]);
209:                                    // if this chunk is not the last one - remove it from the list
210:                                    if (next_node[21])
211:                                            chunk_list[4].remove_node[5](current_node[18]);
```

Footnotes:

1: *HeapManager.cpp:119*
2: *HeapManager.h:48*
3: *HeapManager.cpp:120*
4: *HeapManager.h:88*
5: *dlist.h:114*
6: *HeapManager.cpp:127*
7: *dlist.h:11*
8: *HeapManager.cpp:126*
9: *HeapManager.h:47*
10: *HeapManager.cpp:117*
11: *HeapManager.cpp:122*
12: *defs.h:33*
13: *defs.h:29*
14: *HeapManager.h:81*
15: *dlist.h:7*
16: *dlist.h:49*
17: *HeapManager.h:44*
18: *HeapManager.cpp:201*
19: *HeapManager.cpp:205*
20: *dlist.h:59*
21: *HeapManager.cpp:208*

```
212:
213:                                    // squeeze this free place - relocate all the following occupied chunks into it
214:                                    while (1)
215:                                    {
216:                                            if (!next_node[1])
217:                                            {
218:                                                    if (current_chunk[2]->size[3] >= bytes_needed[4])
219:                                                            return Success[5];
220:                                                    else
221:                                                            return Failure[6];
222:                                            }
223:                                            // otherwise get the next chunk
224:                                            memory_chunk[7] * next_chunk = (memory_chunk[7]*)next_node[1]->info[8];
225:                                            // relocate chunk back
226:                                            u1[9] * saved_address = next_chunk[10]->address[11];
227:                                            next_chunk[10]->address[11] -= current_chunk[2]->size[3];
228:
229:                                            if (next_chunk[10]->free[12])
230:                                            {
231:                                                    // if the chunk is free update its size and check the free space
232:                                                    next_chunk[10]->size[3] += current_chunk[2]->size[3];
233:                                                    if (next_chunk[10]->size[3] >= bytes_needed[4])
234:                                                            return Success[5]; // stop relocation
235:                                                    else
236:                                                    {
237:                                                            // there is still not enough room -
238:                                                            // continue the process from this place
239:                                                            current_node[13] = next_node[1];
240:                                                            break;
241:                                                    }
242:                                            }
243:                                            else // next chunk is not free - relocate memory block
244:                                                    memcpy(next_chunk[10]->address[11], saved_address[14], next_chunk[10]->size[3])
        ;
245:
246:                                            next_node[1] = chunk_list[15].get_next_node[16](next_node[1]);
247:
248:                                    } // relocating while
249:
250:                            }
251:                            else // the current chunk is not free - nothing to squeeze, continue search
252:                            {
253:                                    current_node[13] = chunk_list[15].get_next_node[16](current_node[13]);
254:                                    if (!current_node[13])
255:                                            return Failure[6];
256:                            }
257:
258:                    } // big while
259:            }
260:
261:            void HeapManager[17]::total_defragment()
262:            {
263:            }
```

Footnotes:

1: *HeapManager.cpp:208*
2: *HeapManager.cpp:205*
3: *HeapManager.h:48*
4: *HeapManager.cpp:199*
5: *defs.h:33*
6: *defs.h:34*
7: *HeapManager.h:44*
8: *dlist.h:11*
9: *defs.h:168*
10: *HeapManager.cpp:224*
11: *HeapManager.h:46*
12: *HeapManager.h:47*
13: *HeapManager.cpp:201*
14: *HeapManager.cpp:226*
15: *HeapManager.h:88*
16: *dlist.h:59*
17: *HeapManager.h:81*

```
264:
265:        unsigned long HeapManager[1]::sweep(unsigned long mark_counter)
266:        {
267:                // Gather all occupied memory chunks whose mark counter is less than
268:                // the mark counter given by the garbage collector
269:                Vector[2]<memory_chunk*> unreachable_chunks(1000);
270:
271:                node[3] * current = chunk_list[4].get_first_node[5]();
272:                while (current[6])
273:                {
274:                        memory_chunk[7] * chunk = (memory_chunk[7]*)current[6]->info[8];
275:                        if (!chunk[9]->free[10] && chunk[9]->mark_counter[11] < mark_counter[12])
276:                                unreachable_chunks[13].add_element[14](chunk[9]);
277:                        current[6] = chunk_list[4].get_next_node[15](current[6]);
278:                }
279:
280:                // Deallocate unreachable chunks
281:                for (int i = 0; i < unreachable_chunks[13].size[16](); i++)
282:                {
283:                        memory_chunk[7] * chunk = unreachable_chunks[13].element_at[17](i[18]);
284:                        dealloc[19](chunk[20]);
285:                }
286:
287:                return unreachable_chunks[13].size[16]();
288:        }
289:
290:        void HeapManager[1]::dump()
291:        {
292:                int i = 1;
293:                node[3] * current = chunk_list[4].get_first_node[5]();
294:                while (current[21])
295:                {
296:                        memory_chunk[7] * chunk = (memory_chunk[7]*)current[21]->info[8];
297:                        cout << "CHUNK #" << i[22]++ << ":" << endl;
298:                        chunk[23]->debug_print[24]();
299:                        current[21] = chunk_list[4].get_next_node[15](current[21]);
300:                }
301:        }
```

Footnotes:

1: *HeapManager.h:81*
2: *vector.h:17*
3: *dlist.h:7*
4: *HeapManager.h:88*
5: *dlist.h:49*
6: *HeapManager.cpp:271*
7: *HeapManager.h:44*
8: *dlist.h:11*
9: *HeapManager.cpp:274*
10: *HeapManager.h:47*
11: *HeapManager.h:55*
12: *HeapManager.cpp:265*
13: *HeapManager.cpp:269*
14: *vector.h:40*
15: *dlist.h:59*
16: *vector.h:23*
17: *vector.h:36*
18: *HeapManager.cpp:281*
19: *HeapManager.cpp:117*
20: *HeapManager.cpp:283*
21: *HeapManager.cpp:293*
22: *HeapManager.cpp:292*
23: *HeapManager.cpp:296*
24: *HeapManager.cpp:9*

```
1:          #include <assert.h>
2:          #include <iostream.h>
3:
4:          #include "NativeHandler.h"
5:          #include "VirtualMachine.h"
6:          #include "Thread.h"
7:          #include "ClassFile.h"
8:          #include "ObjectData.h"
9:          #include "HandlePool.h"
10:         #include "ClassLoader.h"
11:         #include "jni_wrappers.h"
12:
13:         // Native method includes for java.lang package
14:         #include "java_lang_Object.h"
15:         #include "java_lang_Class.h"
16:         #include "java_lang_System.h"
17:         #include "java_lang_Runtime.h"
18:         #include "java_lang_Thread.h"
19:
20:         // This is the definition of the Java Native Interface variable
21:         static JNINativeInterface_ jni_native_interface;
22:
23:         // This is the definition of the Java Native Environment variable
24:         // JNIEnv uses pointers to functions defined in JNINativeInterface
25:         // in C++ manner
26:         static JNIEnv             jni_env;
27:
28:         // Implementation of the Java Native Interface functions
29:         jint JNICALL GetJavaVM(JNIEnv *env, JavaVM **vm)
30:         {
31:                 assert(env[1]->functions->reserved0 != NULL);
32:                 *vm[2] = (JavaVM*)env[1]->functions->reserved0;
33:
34:                 return JNI_OK;
35:         }
36:
37:         jclass JNICALL GetObjectClass(JNIEnv *env, jobject obj)
38:         {
39:                 jobject_wrapper[3] * id = (jobject_wrapper[3]*)obj[4];
40:                 assert(id[5] != NULL);
41:
42:                 ClassFile[6] * cf = id[5]->class_file[7];
43:                 assert(cf[8] != NULL);
44:                 assert (cf[8]->class_instance[9] != NULL);
45:
46:                 return (jclass)cf[8]->class_instance[9];
47:         }
48:
49:         jint JNICALL ThrowNew(JNIEnv *env, jclass clazz, const char *msg)
50:         {
51:                 // Meanwhile, just print the message
52:                 cout << "[Native code exception:] " << msg[10] << endl;
53:                 return JNI_OK;
```

Footnotes:

1: *NativeHandler.cpp:29*
2: *NativeHandler.cpp:29*
3: *jni_wrappers.h:16*
4: *NativeHandler.cpp:37*
5: *NativeHandler.cpp:39*
6: *ClassFile.h:136*
7: *ObjectData.h:74*
8: *NativeHandler.cpp:42*
9: *ClassFile.h:243*
10: *NativeHandler.cpp:49*

```
54:         }
55:
56:         jsize JNICALL GetStringUTFLength(JNIEnv *env, jstring str)
57:         {
58:                 JavaVM * jvm = (JavaVM*)env[1]->functions->reserved0;
59:                 VirtualMachine[2] * vm = (VirtualMachine[2]*)jvm[3];
60:                 HandlePool[4] * hp = vm[5]->get_handle_pool[6]();
61:                 char * ascii_string = hp[7]->get_interned_string[8]((InstanceData[9]*)str[10]);
62:                 if (!ascii_string[11])
63:                         return 0;
64:                 return strlen(ascii_string[11]);
65:         }
66:
67:         const char * JNICALL GetStringUTFChars(JNIEnv *env, jstring str, jboolean *isCopy)
68:         {
69:                 JavaVM * jvm = (JavaVM*)env[12]->functions->reserved0;
70:                 VirtualMachine[2] * vm = (VirtualMachine[2]*)jvm[13];
71:                 HandlePool[4] * hp = vm[14]->get_handle_pool[6]();
72:                 char * ascii_string = hp[15]->get_interned_string[8]((InstanceData[9]*)str[16]);
73:                 return ascii_string[17];
74:         }
75:
76:         void JNICALL ReleaseStringUTFChars(JNIEnv *env, jstring str, const char* chars)
77:         {
78:         }
79:
80:         jclass JNICALL FindClass(JNIEnv *env, const char * name)
81:         {
82:                 JavaVM * jvm = (JavaVM*)env[18]->functions->reserved0;
83:                 VirtualMachine[2] * vm = (VirtualMachine[2]*)jvm[19];
84:                 ClassLoader[20] * bootstrap_class_loader = vm[21]->get_bootstrap_class_loader[22]();
85:
86:                 // This is the instance of the java.lang.Class type
87:                 InstanceData[9] * id = bootstrap_class_loader[23]->find_class[24]((char*)name[25]);
88:                 assert(id[26] != NULL);
89:
90:                 return (jclass)id[26];
91:         }
92:
93:
94:         #define ASSIGN_FUNCTION(f) jni_native_interface_ptr->f = f;
95:
96:         NativeHandler[27]::NativeHandler(VirtualMachine[2] * _vm) : vm[28](_vm)
97:         {
98:                 jni_native_interface_ptr[29] = &jni_native_interface[30];
99:                 jni_env_ptr[31] = &jni_env[32];
100:
101:                // Assign our JVM to the reserved0 field of the JNI environment
102:                jni_native_interface_ptr[29]->reserved0 = (void*)vm[28];
103:
104:                // Assign "functions" field of the jni_env
105:                jni_env_ptr[31]->functions = jni_native_interface_ptr[29];
106:
```

Footnotes:

1: *NativeHandler.cpp:56*
2: *VirtualMachine.h:255*
3: *NativeHandler.cpp:58*
4: *HandlePool.h:13*
5: *NativeHandler.cpp:59*
6: *VirtualMachine.h:336*
7: *NativeHandler.cpp:60*
8: *HandlePool.cpp:34*
9: *ObjectData.h:113*
10: *NativeHandler.cpp:56*
11: *NativeHandler.cpp:61*
12: *NativeHandler.cpp:67*
13: *NativeHandler.cpp:69*
14: *NativeHandler.cpp:70*
15: *NativeHandler.cpp:71*
16: *NativeHandler.cpp:67*
17: *NativeHandler.cpp:72*
18: *NativeHandler.cpp:80*
19: *NativeHandler.cpp:82*
20: *ClassLoader.h:14*
21: *NativeHandler.cpp:83*
22: *VirtualMachine.h:334*
23: *NativeHandler.cpp:84*
24: *ClassLoader.cpp:200*
25: *NativeHandler.cpp:80*
26: *NativeHandler.cpp:87*
27: *NativeHandler.h:13*
28: *NativeHandler.h:22*
29: *NativeHandler.h:17*
30: *NativeHandler.cpp:21*
31: *NativeHandler.h:19*
32: *NativeHandler.cpp:26*

```
107:                    // Assign all function implementations
108:                    ASSIGN_FUNCTION[1](GetJavaVM)
109:                    ASSIGN_FUNCTION[1](GetObjectClass)
110:                    ASSIGN_FUNCTION[1](FindClass)
111:                    ASSIGN_FUNCTION[1](ThrowNew)
112:                    ASSIGN_FUNCTION[1](GetStringUTFLength)
113:                    ASSIGN_FUNCTION[1](GetStringUTFChars)
114:                    ASSIGN_FUNCTION[1](ReleaseStringUTFChars)
115:                    ASSIGN_FUNCTION[1](FindClass)
116:            }
117:
118:            // This function will register the proper "registerNatives" functions for all known class
119:            // implementations (such as java.lang package)
120:            void NativeHandler[2]::init()
121:            {
122:                    // Proper "registerNatives"
123:                    Java_java_lang_Object_register[3](jni_env_ptr[4]);
124:                    Java_java_lang_Thread_register[5](jni_env_ptr[4]);
125:                    Java_java_lang_System_register[6](jni_env_ptr[4]);
126:                    Java_java_lang_Class_register[7](jni_env_ptr[4]);
127:
128:                    // Others
129:                    Java_java_security_AccessController_register[8](jni_env_ptr[4]);
130:                    Java_java_io_FileInputStream_register[9](jni_env_ptr[4]);
131:                    Java_java_io_FileOutputStream_register[10](jni_env_ptr[4]);
132:                    Java_java_io_FileDescriptor_register[11](jni_env_ptr[4]);
133:                    Java_java_lang_ClassLoader_register[12](jni_env_ptr[4]);
134:            }
135:
136:            void NativeHandler[2]::register_method(char * name, char * descriptor, void * func_ptr)
137:            {
138:                    JNINativeMethod * method = new JNINativeMethod;
139:                    method[13]->name = name[14];
140:                    method[13]->signature = descriptor[15];
141:                    method[13]->fnPtr = func_ptr[16];
142:
143:                    table[17].insert[18](name[14], method[13]);
144:            }
145:
146:            // Suppose the full_name has enough space
147:            void NativeHandler[2]::get_jni_compliant_method_name(method_info[19] * mi, char * full_name)
148:            {
149:                    // We have to form the method name according to the JNI spec.
150:                    // For example: Java_java_lang_Object_hashCode
151:
152:                    strcpy(full_name[20], "Java_");
153:
154:                    wchar_t * w_method_name;
155:                    u2[21] method_name_length;
156:                    wchar_t * w_class_name;
157:                    u2[21] class_name_length;
158:
159:                    ClassFile[22] * cf = mi[23]->class_file[24];
```

Footnotes:

1: *NativeHandler.cpp:94*
2: *NativeHandler.h:13*
3: *java_lang_Object.cpp:161*
4: *NativeHandler.h:19*
5: *java_lang_Thread.cpp:187*
6: *java_lang_System.cpp:154*
7: *java_lang_Class.cpp:321*
8: *java_security_AccessController.cpp:74*
9: *java_io_FileInputStream.cpp:78*
10: *java_io_FileOutputStream.cpp:100*
11: *java_io_FileDescriptor.cpp:36*
12: *java_lang_ClassLoader.cpp:65*
13: *NativeHandler.cpp:138*
14: *NativeHandler.cpp:136*
15: *NativeHandler.cpp:136*
16: *NativeHandler.cpp:136*
17: *NativeHandler.h:25*
18: *hashtable.cpp:23*
19: *ClassFile.h:92*
20: *NativeHandler.cpp:147*
21: *defs.h:172*
22: *ClassFile.h:136*
23: *NativeHandler.cpp:147*
24: *ClassFile.h:56*

```
160:                    assert(cf[1] != NULL);
161:                    cf[1]->get_full_class_name[2](w_class_name[3], class_name_length[4]);
162:                    char * class_name = new char[class_name_length[4]+1];
163:                    wchar2ascii[5](w_class_name[3], class_name[6], class_name_length[4]);
164:                    slash2underscore[7](class_name[6]);
165:
166:                    strcat(full_name[8], class_name[6]);
167:                    strcat(full_name[8], "_");
168:
169:                    mi[9]->get_method_name[10](w_method_name[11], method_name_length[12]);
170:                    char * method_name = new char[method_name_length[12]+1];
171:                    wchar2ascii[5](w_method_name[11], method_name[13], method_name_length[12]);
172:                    strcat(full_name[8], method_name[13]);
173:
174:                    delete [] w_class_name[3];
175:                    delete [] w_method_name[11];
176:                    delete [] class_name[6];
177:                    delete [] method_name[13];
178:            }
179:
180:            // This function has to use assembley because a method signature cannot be
181:            // formed dynamically;
182:            // Parameters to the native method will be taken from the local variables of the
183:            // Java stack and pushed onto the native stack; then the method will be called.
184:            // Its return value (if any) will be taken from the eax register and pushed into the
185:            // Java stack.
186:            // This way of processing makes the native method invocation transparent from the Java stack
187:            // point of view - parameter passing and return is performed in exactly the same way
188:            // as with the Java method invocation
189:            Result[14] NativeHandler[15]::run_method(Thread[16] * thread, method_info[17] * mi, InstanceData[18] * id)
190:            {
191:                    char full_name[MAX_JNI_METHOD_NAME[19]];
192:                    get_jni_compliant_method_name[20](mi[21], full_name[22]);
193:
194:            ////////////////////////////////////////////////////////////////////////////// TODO:
195:                    if (!strcmp(full_name[22],"Java_java_security_AccessController_doPrivileged"))
196:                    {
197:                                            thread[23]->current_stack_frame[24]->push_reference[25](null[26]);
198:                                            return thread[23]->return_reference_from_method[27]();
199:                    }
200:
201:                    if (!strcmp(full_name[22],"Java_java_lang_Throwable_fillInStackTrace"))
202:                    {
203:                                            thread[23]->current_stack_frame[24]->push_reference[25](null[26]);
204:                                            return thread[23]->return_reference_from_method[27]();
205:                    }
206:            //////////////////////////////////////////////////////////////////////////////
207:
208:            #ifdef DEBUG_EXECUTION[28]
209:            //      cout << "================================================" << endl;
210:            //      cout << "Entering native method " << full_name << endl;
211:            //      cout << "================================================" << endl;
212:                    debug_file << "================================================" << endl;
```

Footnotes:

1: *NativeHandler.cpp:159*
2: *ClassFile.cpp:442*
3: *NativeHandler.cpp:156*
4: *NativeHandler.cpp:157*
5: *util.cpp:8*
6: *NativeHandler.cpp:162*
7: *util.cpp:30*
8: *NativeHandler.cpp:147*
9: *NativeHandler.cpp:147*
10: *ClassFile.cpp:168*
11: *NativeHandler.cpp:154*
12: *NativeHandler.cpp:155*
13: *NativeHandler.cpp:170*
14: *defs.h:29*
15: *NativeHandler.h:13*
16: *Thread.h:17*
17: *ClassFile.h:92*
18: *ObjectData.h:113*
19: *defs.h:204*
20: *NativeHandler.cpp:147*
21: *NativeHandler.cpp:189*
22: *NativeHandler.cpp:191*
23: *NativeHandler.cpp:189*
24: *Thread.h:148*
25: *Stack.cpp:187*
26: *defs.h:201*
27: *Thread.cpp:306*
28: *defs.h:428*

```
213:                debug_file << "Entering native method " << full_name[1] << endl;
214:                debug_file << "================================================" << endl;
215:        #endif // DEBUG_EXECUTION
216:
217:
218:                JNINativeMethod * method = (JNINativeMethod*)table[2].find[3](full_name[1]);
219:                if (!method[4])
220:                        return ExecutionNativeMethodNotFound[5];
221:
222:                void * method_pointer = method[4]->fnPtr;
223:                assert(method_pointer[6] != NULL);
224:
225:                JNIEnv * env_pointer = jni_env_ptr[7]; // to pass as a parameter to a method call
226:
227:                // First, we must know what types of parameters do we have
228:                Vector[8]<general_type*> args;
229:                general_type[9] return_type;
230:                Result[10] result = decipher_method_descriptor[11](method[4]->signature, args[12], &return_type[13]);
231:                if (result[14] != Success[15])
232:                {
233:                        print_error[16](result[14]);
234:                        return ExecutionCannotExecuteNativeMethod[17];
235:                }
236:
237:                stack_frame[18] * frame = thread[19]->current_stack_frame[20];
238:                assert(frame[21] != NULL);
239:
240:                word[22] objectref = null[23];
241:                if (id[24])
242:                        // this is not a static method; object reference is stored
243:                        // in local variable 0
244:                        objectref[25] = frame[21]->get_reference[26](0);
245:
246:                // Now, get the arguments from the local variables of the current stack frame
247:                int start = (objectref[25] == null[23] ? 0 : 1);
248:                int end = (objectref[25] == null[23] ? args[12].size[27]()-1 : args[12].size[27]());
249:                int args_size = args[12].size[27]()-1;
250:
251:                jbyte byte_value;
252:                jchar char_value;
253:                jint int_value;
254:                jboolean bool_value;
255:                jshort short_value;
256:                jdouble double_value;
257:                jfloat float_value;
258:                jlong long_value;
259:                word[22] reference_value;
260:                jobject_wrapper[28] * obj;
261:                u4[29] high_bytes, low_bytes;
262:
263:                unsigned int bytes_to_pop = 0;
264:
265:                // Push the values of the parameters from the local variables of the Java stack
```

Footnotes:

1: *NativeHandler.cpp:191*
2: *NativeHandler.h:25*
3: *hashtable.cpp:79*
4: *NativeHandler.cpp:218*
5: *defs.h:76*
6: *NativeHandler.cpp:222*
7: *NativeHandler.h:19*
8: *vector.h:17*
9: *defs.h:239*
10: *defs.h:29*
11: *decipher.cpp:113*
12: *NativeHandler.cpp:228*
13: *NativeHandler.cpp:229*
14: *NativeHandler.cpp:230*
15: *defs.h:33*
16: *errors.cpp:35*
17: *defs.h:77*
18: *Stack.h:129*
19: *NativeHandler.cpp:189*
20: *Thread.h:148*
21: *NativeHandler.cpp:237*
22: *defs.h:195*
23: *defs.h:201*
24: *NativeHandler.cpp:189*
25: *NativeHandler.cpp:240*
26: *Stack.cpp:396*
27: *vector.h:23*
28: *jni_wrappers.h:16*
29: *defs.h:174*

```
266:                    // onto the native stack
267:                    int i,j;
268:                    for (i[1] = end[2], j[3] = args_size[4]; i[1] >= start[5]; i[1]--, j[3]--)
269:                    {
270:                            general_type[6] * field = (general_type[6]*)args[7].element_at[8](j[3]);
271:                            switch (field[9]->kind[10])
272:                            {
273:                            case general_type[6]::Basic[11]:
274:                                    switch (field[9]->type[12])
275:                                    {
276:                                    case Byte[13]:
277:                                            byte_value[14] = (jbyte)frame[15]->get_int[16](i[1]);
278:
279:                                            __asm push byte_value[14]
280:
281:                            bytes_to_pop[17] += sizeof(jbyte);
282:                                            break;
283:                                    case Char[18]:
284:                                            char_value[19] = (jchar)frame[15]->get_int[16](i[1]);
285:
286:                                            __asm push char_value[19]
287:
288:                            bytes_to_pop[17] += sizeof(jchar);
289:                                            break;
290:                                    case Int[20]:
291:                                            int_value[21] = (jint)frame[15]->get_int[16](i[1]);
292:
293:                                            __asm push int_value[21]
294:
295:                                            bytes_to_pop[17] += sizeof(jint);
296:                                            break;
297:                                    case Boolean[22]:
298:                                            bool_value[23] = (jboolean)frame[15]->get_int[16](i[1]);
299:
300:                                            __asm push bool_value[23]
301:
302:                            bytes_to_pop[17] += sizeof(jboolean);
303:                                            break;
304:                                    case Short[24]:
305:                                            short_value[25] = (jshort)frame[15]->get_int[16](i[1]);
306:
307:                                            _asm push short_value[25]
308:
309:                                            bytes_to_pop[17] += sizeof(jshort);
310:                                            break;
311:                                    case Double[26]:
312:                                            double_value[27] = (jdouble)frame[15]->get_double[28](i[1]);
313:
314:                                            __asm push double_value[27]
315:
316:                            bytes_to_pop[17] += sizeof(jdouble);
317:                                            break;
318:                                    case Float[29]:
```

Footnotes:

1: *NativeHandler.cpp:267*
2: *NativeHandler.cpp:248*
3: *NativeHandler.cpp:267*
4: *NativeHandler.cpp:249*
5: *NativeHandler.cpp:247*
6: *defs.h:231*
7: *NativeHandler.cpp:228*
8: *vector.h:36*
9: *NativeHandler.cpp:270*
10: *defs.h:233*
11: *defs.h:233*
12: *defs.h:237*
13: *defs.h:210*
14: *NativeHandler.cpp:251*
15: *NativeHandler.cpp:237*
16: *Stack.cpp:357*
17: *NativeHandler.cpp:263*
18: *defs.h:211*
19: *NativeHandler.cpp:252*
20: *defs.h:214*
21: *NativeHandler.cpp:253*
22: *defs.h:218*
23: *NativeHandler.cpp:254*
24: *defs.h:217*
25: *NativeHandler.cpp:255*
26: *defs.h:212*
27: *NativeHandler.cpp:256*
28: *Stack.cpp:383*
29: *defs.h:213*

```
319:                                    float_value[1] = (jfloat)frame[2]->get_float[3](i[4]);
320:
321:                                    __asm push float_value[1]
322:
323:                            bytes_to_pop[5] += sizeof(jfloat);
324:                                    break;
325:                            case Long[6]:
326:                                    long_value[7] = (jlong)frame[2]->get_long[8](i[4]);
327:
328:                                    // Divide long value into two halves and push them both onto the stack
329:                                    high_bytes[9] = long_value[7] >> 32;
330:                                    low_bytes[10] = long_value[7] & 0xFFFFFFFF;
331:
332:                                    __asm {
333:                                            push high_bytes[9]
334:                                            push low_bytes[10]
335:                                    }
336:
337:                            bytes_to_pop[5] += sizeof(jlong);
338:                                    break;
339:                            default:
340:                                    return ExecutionCannotExecuteNativeMethod[11];
341:                            }
342:                            break;
343:                    case general_type[12]::Reference[13]: case general_type[12]::Array[14]:
344:                            reference_value[15] = frame[2]->get_reference[16](i[4]);
345:                            obj[17] = (jobject_wrapper[18]*)vm[19]->get_handle_pool[20]()->get_instance[21](reference_value[15]
            );
346:
347:                                    __asm push obj[17]
348:
349:                                    break;
350:                            default:
351:                                    return ExecutionCannotExecuteNativeMethod[11];
352:                            }
353:                    }
354:
355:                    // Push the object onto the native stack if needed
356:                    if (objectref[22] != null[23])
357:                    {
358:                            obj[17] = (jobject_wrapper[18]*)vm[19]->get_handle_pool[20]()->get_instance[21](objectref[22]);
359:
360:                            __asm push obj[17]
361:
362:                    bytes_to_pop[5] += sizeof(jobject_wrapper[18]);
363:                    }
364:                    else // The method is static - push Class object instead
365:                    {
366:                            InstanceData[24] * class_instance = mi[25]->class_file[26]->class_instance[27];
367:                            //assert(class_instance != NULL);
368:
369:                            __asm push class_instance
370:                    }
```

Footnotes:

1: *NativeHandler.cpp:257*
2: *NativeHandler.cpp:237*
3: *Stack.cpp:376*
4: *NativeHandler.cpp:267*
5: *NativeHandler.cpp:263*
6: *defs.h:215*
7: *NativeHandler.cpp:258*
8: *Stack.cpp:364*
9: *NativeHandler.cpp:261*
10: *NativeHandler.cpp:261*
11: *defs.h:77*
12: *defs.h:231*
13: *defs.h:233*
14: *defs.h:233*
15: *NativeHandler.cpp:259*
16: *Stack.cpp:396*
17: *NativeHandler.cpp:260*
18: *jni_wrappers.h:16*
19: *NativeHandler.h:22*
20: *VirtualMachine.h:336*
21: *HandlePool.cpp:25*
22: *NativeHandler.cpp:240*
23: *defs.h:201*
24: *ObjectData.h:113*
25: *NativeHandler.cpp:189*
26: *ClassFile.h:56*
27: *ClassFile.h:243*

```
371:
372:
373:            // Push the JNI environment onto the native stack
374:            __asm push env_pointer[1]
375:
376:            bytes_to_pop[2] += sizeof(void*);
377:
378:            // Call the method
379:            __asm call method_pointer[3]
380:
381:            // Note, that because all native functions are declared as "JNICALL" (i.e. __stdcall)
382:            // there is no need to clean up the stack after the function returns;
383:            // otherwise, the following command would be needed:
384:            //   __asm add esp, bytes_to_pop
385:
386:                 // Check whether we are expecting any return type
387:                 if (return_type[4].type == Void[5])
388:                         return thread[6]->return_from_method[7](); // return immediately
389:
390:                 // Special care should be taken when a function returns long
391:                 if (return_type[4].type[8] == Long[9])
392:                 {
393:                         u4[10] high_bytes;
394:                         u4[10] low_bytes;
395:
396:                         __asm {
397:                                 mov low_bytes[11], eax  // low bytes are passed in the eax register
398:                                 mov high_bytes[12], edx // high bytes are passed in the edx register
399:                         }
400:
401:                         su8[13] long_result = ( ((su8[13])high_bytes[14]) << 32) + low_bytes[15];
402:                         frame[16]->push_long[17](long_result[18]);
403:                         return thread[6]->return_long_from_method[19]();
404:                 }
405:
406:                 // If the return value is Boolean the result will be placed in register AL
407:                 if (return_type[4].type[8] == Boolean[20])
408:                 {
409:                         u1[21] bool_result;
410:
411:                         __asm mov bool_result, al
412:
413:                         frame[16]->push_int((su4[22])bool_result);
414:                         return thread[6]->return_int_from_method[23]();
415:                 }
416:
417:
418:            word[24] method_result;
419:
420:            // Store the value returned from the method
421:            __asm mov method_result[25], eax
422:
423:            // Finally, push the value returned from the method onto the Java stack
```

Footnotes:

1: *NativeHandler.cpp:225*
2: *NativeHandler.cpp:263*
3: *NativeHandler.cpp:222*
4: *NativeHandler.cpp:229*
5: *defs.h:220*
6: *NativeHandler.cpp:189*
7: *Thread.cpp:142*
8: *defs.h:237*
9: *defs.h:215*
10: *defs.h:174*
11: *NativeHandler.cpp:261*
12: *NativeHandler.cpp:261*
13: *defs.h:177*
14: *NativeHandler.cpp:393*
15: *NativeHandler.cpp:394*
16: *NativeHandler.cpp:237*
17: *Stack.cpp:153*
18: *NativeHandler.cpp:401*
19: *Thread.cpp:204*
20: *defs.h:218*
21: *defs.h:168*
22: *defs.h:175*
23: *Thread.cpp:170*
24: *defs.h:195*
25: *NativeHandler.cpp:418*

```
424:            // and return from method immediately using the regular return functions
425:                 switch (return_type[1].kind)
426:                 {
427:                         case general_type[2]::Basic:
428:                                 switch (return_type[1].type)
429:                                 {
430:                                 case Byte[3]: case Char[4]: case Int[5]: case Short[6]:
431:                                         frame[7]->push_int((su4[8])method_result[9]);
432:                                         return thread[10]->return_int_from_method();
433:                                 case Double[11]:
434:                                         frame[7]->push_double((double)method_result[9]);
435:                                         return thread[10]->return_double_from_method();
436:                                 case Float[12]:
437:                                         frame[7]->push_float((float)method_result[9]);
438:                                         return thread[10]->return_float_from_method();
439:                                 default:
440:                                         return ExecutionCannotExecuteNativeMethod[13];
441:                                 }
442:                                 break;
443:                                 case general_type[2]::Reference[14]: case general_type[2]::Array[15]:
444:                                         obj[16] = (jobject_wrapper[17]*)method_result[9];
445:                                         // Extract internal reference representation from
446:                                         // the returned object
447:                                         reference_value[18] =
448:                                                 (obj[16] == NULL ? null[19] : ((InstanceData[20]*)obj[16])->get_reference());
449:                                         frame[7]->push_reference(reference_value[18]);
450:                                         return thread[10]->return_reference_from_method();
451:                                         break;
452:                                 default:
453:                                         return ExecutionCannotExecuteNativeMethod[13];
454:                 }
455:
456:                 return Success[21];
457:         }
```

Footnotes:

1: *NativeHandler.cpp:229*
2: *defs.h:231*
3: *defs.h:210*
4: *defs.h:211*
5: *defs.h:214*
6: *defs.h:217*
7: *NativeHandler.cpp:237*
8: *defs.h:175*
9: *NativeHandler.cpp:418*
10: *NativeHandler.cpp:189*
11: *defs.h:212*
12: *defs.h:213*
13: *defs.h:77*
14: *defs.h:216*
15: *defs.h:219*
16: *NativeHandler.cpp:260*
17: *jni_wrappers.h:16*
18: *NativeHandler.cpp:259*
19: *defs.h:201*
20: *ObjectData.h:113*
21: *defs.h:33*

```
 1:
 2:          #include <stdlib.h>
 3:          #include <memory.h>
 4:
 5:          #include "defs.h"
 6:          #include "ObjectData.h"
 7:          #include "Thread.h"
 8:          #include "ClassLoader.h"
 9:
10:          // Default values for all the basic types
11:
12:          static su1[1] DEFAULT_BYTE = (u1[2])0;
13:          static u2[3] DEFAULT_CHAR = 0;//'\u0000';
14:          static u4[4] DEFAULT_DOUBLE[2] = { 0.0f, 0.0f };
15:          static su4[5] DEFAULT_FLOAT = 0.0f;
16:          static su4[5] DEFAULT_INT = 0;
17:          static u4[4] DEFAULT_LONG[2] = { 0L, 0L };
18:          static word[6] DEFAULT_REFERENCE = (word[6])null[7];
19:          static su2[8] DEFAULT_SHORT = (u2[3])0;
20:          static u4[4] DEFAULT_BOOLEAN = (u4[4])0;
21:          static word[6] DEFAULT_ARRAY = (word[6])null[7];
22:
23:
24:          basic_type[9] BasicTypes[] =
25:          {
26:                  { Byte[10], sizeof(u1[2]), &DEFAULT_BYTE[11] },
27:                  { Char[12], sizeof(u2[3]), &DEFAULT_CHAR[13] },
28:                  { Double[14], sizeof(u4[4][2]), &DEFAULT_DOUBLE[15] },
29:                  { Float[16], sizeof(u4[4]), &DEFAULT_FLOAT[17] },
30:                  { Int[18], sizeof(u4[4]), &DEFAULT_INT[19] },
31:                  { Long[20], sizeof(u4[4][2]), &DEFAULT_LONG[21] },
32:                  { Reference[22], sizeof(word[6]), &DEFAULT_REFERENCE[23] },
33:                  { Short[24], sizeof(u4[4]), &DEFAULT_SHORT[25] },
34:                  //{ Boolean, sizeof(u1), &DEFAULT_BOOLEAN },
35:                  { Boolean[26], sizeof(u4[4]), &DEFAULT_BOOLEAN[27] },
36:                  { Array[28], sizeof(word[6]), &DEFAULT_ARRAY[29] }
37:          };
38:
39:
40:
41:          Result[30] ClassData[31]::prepare()
42:          {
43:                  // Point class file to its class data
44:                  class_file[32]->class_data[33] = this;
45:
46:                  // Allocate maximal possible storage for temporary variable offsets holding;
47:                  variable_offset[34] * variables = new variable_offset[34][class_file[32]->fields_count[35]];
48:                  int variable_count = 0;
49:                  int current_offset = 0;
50:
51:                  // For all fields of this class
52:                  for (int i = 0; i < class_file[32]->fields_count[35]; i++)
53:                  {
```

Footnotes:

1: *defs.h:169*
2: *defs.h:168*
3: *defs.h:172*
4: *defs.h:174*
5: *defs.h:175*
6: *defs.h:195*
7: *defs.h:201*
8: *defs.h:173*
9: *defs.h:224*
10: *defs.h:210*
11: *ObjectData.cpp:12*
12: *defs.h:211*
13: *ObjectData.cpp:13*
14: *defs.h:212*
15: *ObjectData.cpp:14*
16: *defs.h:213*
17: *ObjectData.cpp:15*
18: *defs.h:214*
19: *ObjectData.cpp:16*
20: *defs.h:215*
21: *ObjectData.cpp:17*
22: *defs.h:216*
23: *ObjectData.cpp:18*
24: *defs.h:217*
25: *ObjectData.cpp:19*
26: *defs.h:218*
27: *ObjectData.cpp:20*
28: *defs.h:219*
29: *ObjectData.cpp:21*
30: *defs.h:29*
31: *ObjectData.h:92*
32: *ObjectData.h:74*
33: *ClassFile.h:227*
34: *ObjectData.h:46*
35: *ClassFile.h:191*

```
54:                             field_info[1] * fi = (field_info[1]*)class_file[2]->fields[3][i[4]];
55:
56:                             // Check whether the current field is static
57:                             if (! (fi[5]->access_flags[6] & ACC_STATIC[7]))
58:                                     continue;
59:                             // Check whether the current field is constant (final)
60:                             // TODO: check what about FINAL
61:                             //if (fi->access_flags & ACC_FINAL)
62:                             //      continue; // final fields don't belong here, they appear in the Constant Pool
63:
64:                             // This field is static and not final - get its descriptor
65:                             CONSTANT_Utf8_info[8] * utf8_info = (CONSTANT_Utf8_info[8]*)class_file[2]->constant_pool[9]->get_it
        em_at_index[10](fi[5]->descriptor_index[11]);
66:                             if (!utf8_info[12])
67:                             {
68:                                     delete [] variables[13];
69:                                     return PreparationNoFieldDescriptorFound[14];
70:                             }
71:
72:                             // Decipher the variable's type and store the variable in the temporary array
73:                             BasicType[15] type = get_variable_type[16](utf8_info[12]);
74:                             if (type[17] == UnrecognizedType[18])
75:                             {
76:                                     delete [] variables[13];
77:                                     return PreparationUnrecognizedBasicType[19];
78:                             }
79:
80:                             variables[13][variable_count[20]].size[21] = BasicTypes[22][type[17]].size[23];
81:                             variables[13][variable_count[20]].offset[24] = current_offset[25];
82:                             variables[13][variable_count[20]].type[26] = type[17];
83:
84:                             // Store the offset in the corresponding field
85:                             fi[5]->offset[27] = current_offset[25];
86:                             fi[5]->type[28] = type[17];
87:
88:                             // Prepare for the next field
89:                             current_offset[25] += BasicTypes[22][type[17]].size[23];
90:
91:                             variable_count[20]++;
92:                     }
93:
94:                     if (current_offset[25] == 0)
95:                     {
96:                             // This class does not have any staitc fields
97:                             return Success[29];
98:                     }
99:
100:                    // At this point current_offset contains the total size of all the static variables
101:                    // Allocate the class data block on the class heap
102:                    Result[30] result = class_file[2]->class_heap_manager[31]->alloc[32](current_offset[25], data_start[33]);
103:                    if (result[34] != Success[29])
104:                    {
105:                            print_error[35](result[34]);
```

Footnotes:

1: *ClassFile.h:68*
2: *ObjectData.h:74*
3: *ClassFile.h:197*
4: *ObjectData.cpp:52*
5: *ObjectData.cpp:54*
6: *ClassFile.h:37*
7: *defs.h:364*
8: *ConstantPool.h:186*
9: *ClassFile.h:159*
10: *ConstantPool.h:299*
11: *ClassFile.h:47*
12: *ObjectData.cpp:65*
13: *ObjectData.cpp:47*
14: *defs.h:65*
15: *defs.h:208*
16: *ObjectData.cpp:124*
17: *ObjectData.cpp:73*
18: *defs.h:221*
19: *defs.h:66*
20: *ObjectData.cpp:48*
21: *ObjectData.h:49*
22: *ObjectData.cpp:24*
23: *defs.h:227*
24: *ObjectData.h:48*
25: *ObjectData.cpp:49*
26: *ObjectData.h:50*
27: *ClassFile.h:79*
28: *ClassFile.h:72*
29: *defs.h:33*
30: *defs.h:29*
31: *ClassFile.h:235*
32: *HeapManager.cpp:60*
33: *ObjectData.h:71*
34: *ObjectData.cpp:102*
35: *errors.cpp:35*

```
106:                        delete [] variables[1];
107:                        return PreparationCannotAllocateClassData[2];
108:                }
109:
110:                // Assign all the fields with their default values
111:                for (i = 0; i < variable_count[3]; i++)
112:                {
113:                        u1[4] * field_memory = data_start[5]->address[6] + variables[1][i].offset[7];
114:                        void * default_value = BasicTypes[8][variables[1][i].type[9]].default_value[10];
115:                        int size = BasicTypes[8][variables[1][i].type[9]].size[11];
116:
117:                        memcpy(field_memory[12], default_value[13], size[14]);
118:                }
119:                delete [] variables[1];
120:
121:                return Success[15];
122:        }
123:
124:        BasicType[16] ObjectData[17]::get_variable_type(CONSTANT_Utf8_info[18] * utf8_info)
125:        {
126:                BasicType[16] type;
127:
128:                wchar_t * string;
129:                Result[19] result = utf8_info[20]->get_string[21](string[22]);
130:                if (result[23] != Success[15])
131:                {
132:                        delete [] string[22];
133:                        print_error[24](result[23]);
134:                        return UnrecognizedType[25];
135:                }
136:
137:                // TODO: compare simple bytes
138:
139:                if (!wmemcmp(string[22], BYTE_INTERNAL_REP[26], 1))
140:                        type[27] = Byte[28];
141:                else if (!wmemcmp(string[22], CHAR_INTERNAL_REP[29], 1))
142:                        type[27] = Char[30];
143:                else if (!wmemcmp(string[22], DOUBLE_INTERNAL_REP[31], 1))
144:                        type[27] = Double[32];
145:                else if (!wmemcmp(string[22], FLOAT_INTERNAL_REP[33], 1))
146:                        type[27] = Float[34];
147:                else if (!wmemcmp(string[22], INT_INTERNAL_REP[35], 1))
148:                        type[27] = Int[36];
149:                else if (!wmemcmp(string[22], LONG_INTERNAL_REP[37], 1))
150:                        type[27] = Long[38];
151:                else if (!wmemcmp(string[22], SHORT_INTERNAL_REP[39], 1))
152:                        type[27] = Short[40];
153:                else if (!wmemcmp(string[22], BOOL_INTERNAL_REP[41], 1))
154:                        type[27] = Boolean[42];
155:                else if (!wmemcmp(string[22], CLASSNAME_INTERNAL_REP[43], 1))
156:                        type[27] = Reference[44];
157:                else if (!wmemcmp(string[22], ARRAY_DIM_INTERNAL_REP[45], 1))
158:                        type[27] = Array[46];
```

Footnotes:

1: *ObjectData.cpp:47*
2: *defs.h:67*
3: *ObjectData.cpp:48*
4: *defs.h:168*
5: *ObjectData.h:71*
6: *HeapManager.h:46*
7: *ObjectData.h:48*
8: *ObjectData.cpp:24*
9: *ObjectData.h:50*
10: *defs.h:228*
11: *defs.h:227*
12: *ObjectData.cpp:113*
13: *ObjectData.cpp:114*
14: *ObjectData.cpp:115*
15: *defs.h:33*
16: *defs.h:208*
17: *ObjectData.h:56*
18: *ConstantPool.h:186*
19: *defs.h:29*
20: *ObjectData.cpp:124*
21: *ConstantPool.cpp:639*
22: *ObjectData.cpp:128*
23: *ObjectData.cpp:129*
24: *errors.cpp:35*
25: *defs.h:221*
26: *defs.h:301*
27: *ObjectData.cpp:126*
28: *defs.h:210*
29: *defs.h:302*
30: *defs.h:211*
31: *defs.h:303*
32: *defs.h:212*
33: *defs.h:304*
34: *defs.h:213*
35: *defs.h:305*
36: *defs.h:214*
37: *defs.h:306*
38: *defs.h:215*
39: *defs.h:308*
40: *defs.h:217*
41: *defs.h:309*
42: *defs.h:218*
43: *defs.h:307*
44: *defs.h:216*
45: *defs.h:310*
46: *defs.h:219*

```
159:                    else
160:                            type[1] = UnrecognizedType[2];
161:
162:                    delete [] string[3];
163:                    return type[1];
164:            }
165:
166:
167:            // "variables" will accumulate field information of all the superclasses
168:            Result[4] InstanceData[5]::get_class_fields(ClassFile[6] * cf, Vector[7]<variable_offset*> * variables, int me)
169:            {
170:                    if (cf[8]->super_class[9] > 0)
171:                    {
172:                            // This is not the java.lang.Object class
173:
174:                            // Get this class' superclass
175:                            CONSTANT_Class_info[10] * super_class_entry =
176:                                    (CONSTANT_Class_info[10]*)cf[8]->constant_pool[11]->get_item_at_index[12](cf[8]->super_class[9]);
177:
178:                            // At this time the superclass must be already resolved
179:                            assert(super_class_entry[13]->resolved_class_file[14] != NULL);
180:                            // this is not the java.lang.Object class - get the superclass' fields recursively
181:                            // "me" is false, don't count the private fields
182:                            Result[4] result = get_class_fields[15](super_class_entry[13]->resolved_class_file[14], variables[16],
            );
183:                            if (result[17] != Success[18])
184:                            {
185:                                    print_error[19](result[17]);
186:                                    return InstanceCreationFailure[20];
187:                            }
188:                    } // else - this is the java.lang.Object class - don't go any higher
189:
190:                    // Now get all the fields of this class
191:
192:                    // For all fields of this class
193:                    for (int i = 0; i < cf[8]->fields_count[21]; i++)
194:                    {
195:                            field_info[22] * fi = (field_info[22]*)cf[8]->fields[23][i[24]];
196:
197:                            // Check whether the current field is static
198:                            if (fi[25]->access_flags[26] & ACC_STATIC[27])
199:                                    continue; // do nothing
200:                            // Check whether the current field is constant (final)
201:                            // TODO:
202:                            //if (fi->access_flags & ACC_FINAL)
203:                            //      continue; // final fields don't belong here, they appear in the Constant Pool
204:
205:                            // This field is not static, final or private (for superclass) - get its descriptor
206:                            CONSTANT_Utf8_info[28] * utf8_info = (CONSTANT_Utf8_info[28]*)cf[8]->constant_pool[11]->get_item_at_
            index[12](fi[25]->descriptor_index[29]);
207:                            if (!utf8_info[30])
208:                                    return InstanceCreationNoFieldDescriptorFound[31];
209:
```

Footnotes:

1: *ObjectData.cpp:126*
2: *defs.h:221*
3: *ObjectData.cpp:128*
4: *defs.h:29*
5: *ObjectData.h:113*
6: *ClassFile.h:136*
7: *vector.h:7*
8: *ObjectData.cpp:168*
9: *ClassFile.h:178*
10: *ConstantPool.h:58*
11: *ClassFile.h:159*
12: *ConstantPool.h:299*
13: *ObjectData.cpp:175*
14: *ConstantPool.h:71*
15: *ObjectData.cpp:168*
16: *ObjectData.cpp:168*
17: *ObjectData.cpp:182*
18: *defs.h:33*
19: *errors.cpp:35*
20: *defs.h:72*
21: *ClassFile.h:191*
22: *ClassFile.h:68*
23: *ClassFile.h:197*
24: *ObjectData.cpp:193*
25: *ObjectData.cpp:195*
26: *ClassFile.h:37*
27: *defs.h:364*
28: *ConstantPool.h:186*
29: *ClassFile.h:47*
30: *ObjectData.cpp:206*
31: *defs.h:69*

```
210:                        // Decipher the variable's type and store the variable in the temporary array
211:                        BasicType[1] type = get_variable_type[2](utf8_info[3]);
212:                        if (type[4] == UnrecognizedType[5])
213:                                return InstanceCreationUnrecognizedBasicType[6];
214:
215:                        variable_offset[7] * prev = (variables[8]->is_empty[9]() ? NULL : variables[8]->last_element[10]());
216:                        variable_offset[7] * vo = new variable_offset[7];
217:                        vo[11]->size[12] = BasicTypes[13][type[4]].size[14];
218:                        vo[11]->offset[15] = (prev[16] ? (prev[16]->offset[15] + prev[16]->size[12]) : 0);
219:                        vo[11]->type[17] = type[4];
220:                        variables[8]->add_element[18](vo[11]);
221:
222:                        // Store the offset in the corresponding field
223:                        // NOTE: throughout the tree of superclasses and subclasses, offsets of the
224:                        // class fields remain invariant
225:                        fi[19]->offset[20] = vo[11]->offset[15];
226:                        fi[19]->type[21] = type[4];
227:                }
228:
229:                return Success[22];
230:        }
231:
232:        // This function will create the new instance of the given class.
233:        // It will look for all the instance variables of the class and all the
234:        // instance variables of its superclasses and initialize the necessary
235:        // memory block to hold them;
236:        Result[23] InstanceData[24]::create()
237:        {
238:                // To accumulate the information about the variables
239:                Vector[25]<variable_offset*> * variables = new Vector[26]<variable_offset*>;
240:
241:                // Go get my fields (and all of my superclasses' fields)
242:                // "me" is true, so get the private variables too (obsolete !)
243:                Result[23] result = get_class_fields[27](class_file[28], variables[29], 1);
244:                if (result[30] != Success[22])
245:                {
246:                        print_error[31](result[30]);
247:                        free_vector[32](variables[29]);
248:                        return InstanceCreationFailure[33];
249:                }
250:
251:                // If there are no fields - just return
252:                if (variables[29]->size[34]() == 0)
253:                {
254:                        free_vector[32](variables[29]);
255:                        return Success[22];
256:                }
257:
258:                // At this point the last member of "variables" contains the last offset and the
259:                // last size of the last field in the hierarchy.
260:                // We can allocate the instance data block on the instance heap
261:                variable_offset[7] * last_var = variables[29]->last_element[10]();
262:                u4[35] total_size = last_var[36]->offset[15] + last_var[36]->size[12];
```

Footnotes:

1: *defs.h:208*
2: *ObjectData.cpp:124*
3: *ObjectData.cpp:206*
4: *ObjectData.cpp:211*
5: *defs.h:221*
6: *defs.h:71*
7: *ObjectData.h:46*
8: *ObjectData.cpp:168*
9: *vector.h:24*
10: *vector.h:74*
11: *ObjectData.cpp:216*
12: *ObjectData.h:49*
13: *ObjectData.cpp:24*
14: *defs.h:227*
15: *ObjectData.h:48*
16: *ObjectData.cpp:215*
17: *ObjectData.h:50*
18: *vector.h:40*
19: *ObjectData.cpp:195*
20: *ClassFile.h:79*
21: *ClassFile.h:72*
22: *defs.h:33*
23: *defs.h:29*
24: *ObjectData.h:113*
25: *vector.h:7*
26: *vector.h:17*
27: *ObjectData.cpp:168*
28: *ObjectData.h:74*
29: *ObjectData.cpp:239*
30: *ObjectData.cpp:243*
31: *errors.cpp:35*
32: *ObjectData.cpp:290*
33: *defs.h:72*
34: *vector.h:23*
35: *defs.h:174*
36: *ObjectData.cpp:261*

```
263:
264:                result[1] = class_file[2]->instance_heap_manager[3]->alloc[4](total_size[5], data_start[6]);
265:                if (result[1] != Success[7])
266:                {
267:                        print_error[8](result[1]);
268:                        free_vector[9](variables[10]);
269:                        return InstanceCreationCannotAllocateInstanceData[11];
270:                }
271:
272:                // Assign all the fields with their default values
273:                for (int i = 0; i < variables[10]->size[12](); i++)
274:                {
275:                        variable_offset[13] * vo = variables[10]->element_at[14](i[15]);
276:                        u1[16] * field_memory = data_start[6]->address[17] + vo[18]->offset[19];
277:                        void * default_value = BasicTypes[20][vo[18]->type[21]].default_value[22];
278:                        int size = BasicTypes[20][vo[18]->type[21]].size[23];
279:
280:                        memcpy(field_memory[24], default_value[25], size[26]);
281:                }
282:
283:                free_vector[9](variables[10]);
284:
285:                return Success[7];
286:        }
287:
288:
289:        // Will delete all vectors data and the vector itself
290:        void InstanceData[27]::free_vector(Vector[28]<variable_offset*> * variables)
291:        {
292:                for (int i = 0; i < variables[29]->size[12](); i++)
293:                {
294:                        variable_offset[13] * vo = variables[29]->element_at[14](i[30]);
295:                        delete vo[31];
296:                }
297:
298:                delete variables[29];
299:        }
300:
301:        // This will create an array of the reference type values of the given length
302:        // and initialize all the elements to null
303:        Result[32] ArrayInstanceData[33]::create(u4[34] count)
304:        {
305:                array_length[35] = count[36];
306:
307:                u4[34] total_size = BasicTypes[20][Reference[37]].size[23] * count[36]; // our element is of type reference ("wo
        rd" size)
308:
309:                // Allocate necessary space on the heap
310:                Result[32] result = class_file[2]->instance_heap_manager[3]->alloc[4](total_size[38], data_start[6]);
311:                if (result[39] != Success[7])
312:                {
313:                        print_error[8](result[39]);
314:                        return InstanceCreationCannotAllocateInstanceData[11];
```

Footnotes:

1: *ObjectData.cpp:243*
2: *ObjectData.h:74*
3: *ClassFile.h:237*
4: *HeapManager.cpp:60*
5: *ObjectData.cpp:262*
6: *ObjectData.h:71*
7: *defs.h:33*
8: *errors.cpp:35*
9: *ObjectData.cpp:290*
10: *ObjectData.cpp:239*
11: *defs.h:70*
12: *vector.h:23*
13: *ObjectData.h:46*
14: *vector.h:36*
15: *ObjectData.cpp:273*
16: *defs.h:168*
17: *HeapManager.h:46*
18: *ObjectData.cpp:275*
19: *ObjectData.h:48*
20: *ObjectData.cpp:24*
21: *ObjectData.h:50*
22: *defs.h:228*
23: *defs.h:227*
24: *ObjectData.cpp:276*
25: *ObjectData.cpp:277*
26: *ObjectData.cpp:278*
27: *ObjectData.h:113*
28: *vector.h:7*
29: *ObjectData.cpp:290*
30: *ObjectData.cpp:292*
31: *ObjectData.cpp:294*
32: *defs.h:29*
33: *ObjectData.h:155*
34: *defs.h:174*
35: *ObjectData.h:158*
36: *ObjectData.cpp:303*
37: *defs.h:216*
38: *ObjectData.cpp:307*
39: *ObjectData.cpp:310*

```
315:                    }
316:
317:                    // Assign all array elements with null references
318:                    for (int i = 0; i < count[1]; i++)
319:                    {
320:                            word[2] * field_memory = (word[2]*)(data_start[3]->address[4] + sizeof(word[2])*i[5]);
321:                            *field_memory[6] = null[7];
322:                            //memcpy(field_memory, &BasicTypes[Reference].default_value, BasicTypes[Reference].size);
323:                    }
324:
325:                    return Success[8];
326:            }
327:
328:            void InstanceData[9]::debug_print(ostream & out)
329:            {
330:                    out[10] << "REFERENCE: " << (unsigned long)reference[11] << " of class ";
331:                    wchar_t * wname;
332:                    u2[12] length;
333:                    class_file[13]->get_full_class_name[14](wname[15],length[16]);
334:                    print_wchar[17](wname[15],length[16],out[10]);
335:                    delete [] wname[15];
336:                    out[10] << endl;
337:            }
338:
339:            BasicType[18] PrimitiveArrayInstanceData[19]::get_internal_type(u1[20] array_type)
340:            {
341:                    switch (array_type[21])
342:                    {
343:                    case T_BOOLEAN[22]: return Boolean[23];
344:                    case T_CHAR[24]: return Char[25];
345:                    case T_FLOAT[26]: return Float[27];
346:                    case T_DOUBLE[28]: return Double[29];
347:                    case T_BYTE[30]: return Byte[31];
348:                    case T_SHORT[32]:return Short[33];
349:                    case T_INT[34]:return Int[35];
350:                    case T_LONG[36]:return Long[37];
351:                    default:return UnrecognizedType[38];
352:                    }
353:                    return UnrecognizedType[38];
354:            }
355:
356:            // This will create and initialize to the default values an array of the
357:            // primitive type values of the given length
358:            Result[39] PrimitiveArrayInstanceData[19]::create(u4[40] count)
359:            {
360:                    array_length[41] = count[42];
361:                    u1[20] array_type = ((PrimitiveArrayClassFile[43]*)class_file[13])->get_array_type[44]();
362:
363:                    BasicType[18] internal_type = get_internal_type[45](array_type[46]);
364:                    if (internal_type[47] == UnrecognizedType[38])
365:                            return InstanceCreationFailure[48];
366:
367:                    u4[40] total_size = BasicTypes[49][internal_type[47]].size[50] * count[42];
```

Footnotes:

1: *ObjectData.cpp:303*
2: *defs.h:195*
3: *ObjectData.h:71*
4: *HeapManager.h:46*
5: *ObjectData.cpp:318*
6: *ObjectData.cpp:320*
7: *defs.h:201*
8: *defs.h:33*
9: *ObjectData.h:113*
10: *ObjectData.cpp:328*
11: *ObjectData.h:121*
12: *defs.h:172*
13: *ObjectData.h:74*
14: *ClassFile.cpp:442*
15: *ObjectData.cpp:331*
16: *ObjectData.cpp:332*
17: *util.cpp:42*
18: *defs.h:208*
19: *ObjectData.h:174*
20: *defs.h:168*
21: *ObjectData.cpp:339*
22: *defs.h:340*
23: *defs.h:218*
24: *defs.h:341*
25: *defs.h:211*
26: *defs.h:342*
27: *defs.h:213*
28: *defs.h:343*
29: *defs.h:212*
30: *defs.h:344*
31: *defs.h:210*
32: *defs.h:345*
33: *defs.h:217*
34: *defs.h:346*
35: *defs.h:214*
36: *defs.h:347*
37: *defs.h:215*
38: *defs.h:221*
39: *defs.h:29*
40: *defs.h:174*
41: *ObjectData.h:158*
42: *ObjectData.cpp:358*
43: *ClassFile.h:370*
44: *ClassFile.h:395*
45: *ObjectData.cpp:339*
46: *ObjectData.cpp:361*
47: *ObjectData.cpp:363*
48: *defs.h:72*
49: *ObjectData.cpp:24*
50: *defs.h:227*

```
368:
369:                    // Allocate necessary space on the heap
370:                    Result[1] result = class_file[2]->instance_heap_manager[3]->alloc[4](total_size[5], data_start[6]);
371:                    if (result[7] != Success[8])
372:                    {
373:                            print_error[9](result[7]);
374:                            return InstanceCreationCannotAllocateInstanceData[10];
375:                    }
376:
377:                    // Assign all array elements with their default values
378:                    for (int i = 0; i < count[11]; i++)
379:                    {
380:                            u1[12] * field_memory = data_start[6]->address[13] + BasicTypes[14][internal_type[15]].size[16]*i[17];
381:                            void * default_value = BasicTypes[14][internal_type[15]].default_value[18];
382:                            memcpy(field_memory[19], default_value[20], BasicTypes[14][internal_type[15]].size[16]);
383:                    }
384:
385:                    return Success[8];
386:            }
387:
388:            Result[1] ArrayInstanceData[21]::copy_from(InstanceData[22] * src, u4[23] src_offset, u4[23] dst_offset, u4[23] length)
389:            {
390:                    // TODO
391:                    return Success[8];
392:            }
393:
394:            int PrimitiveArrayInstanceData[24]::is_convertable(BasicType[25] from_type)
395:            {
396:                    // TODO
397:                    return 1;
398:            }
399:
400:            Result[1] PrimitiveArrayInstanceData[24]::copy_from(InstanceData[22] * src, u4[23] src_offset, u4[23] dst_offset, u4[23] l
            ength)
401:            {
402:                    if (!src[26])
403:                            return VM_ERROR_NullPointerException[27];
404:
405:                    if (dst_offset[28] < 0 || dst_offset[28] >= array_length[29] ||
406:                            src_offset[30] < 0 || src_offset[30] > ((PrimitiveArrayInstanceData[24]*)src[26])->get_array_length[31](
            ) ||
407:                            length[32] < 0 ||
408:                            dst_offset[28] + length[32] > array_length[29] ||
409:                            src_offset[30] + length[32] > ((PrimitiveArrayInstanceData[24]*)src[26])->get_array_length[31]())
410:                                    return VM_ERROR_ArrayIndexOutOfBoundsException[33];
411:
412:                    PrimitiveArrayClassFile[34] * src_cf = (PrimitiveArrayClassFile[34]*)src[26]->class_file[2];
413:                    PrimitiveArrayClassFile[34] * dst_cf = (PrimitiveArrayClassFile[34]*)class_file[2];
414:
415:                    BasicType[25] dst_internal_type = get_internal_type[35](dst_cf[36]->get_array_type[37]());
416:                    if (dst_internal_type[38] == UnrecognizedType[39]) // cannot happen
417:                            return VM_ERROR_ArrayStoreExceptionClass[40];
418:
```

Footnotes:

1: *defs.h:29*
2: *ObjectData.h:74*
3: *ClassFile.h:237*
4: *HeapManager.cpp:60*
5: *ObjectData.cpp:367*
6: *ObjectData.h:71*
7: *ObjectData.cpp:370*
8: *defs.h:33*
9: *errors.cpp:35*
10: *defs.h:70*
11: *ObjectData.cpp:358*
12: *defs.h:168*
13: *HeapManager.h:46*
14: *ObjectData.cpp:24*
15: *ObjectData.cpp:363*
16: *defs.h:227*
17: *ObjectData.cpp:378*
18: *defs.h:228*
19: *ObjectData.cpp:380*
20: *ObjectData.cpp:381*
21: *ObjectData.h:155*
22: *ObjectData.h:113*
23: *defs.h:174*
24: *ObjectData.h:174*
25: *defs.h:208*
26: *ObjectData.cpp:400*
27: *defs.h:105*
28: *ObjectData.cpp:400*
29: *ObjectData.h:158*
30: *ObjectData.cpp:400*
31: *ObjectData.h:163*
32: *ObjectData.cpp:400*
33: *defs.h:108*
34: *ClassFile.h:370*
35: *ObjectData.cpp:339*
36: *ObjectData.cpp:413*
37: *ClassFile.h:395*
38: *ObjectData.cpp:415*
39: *defs.h:221*
40: *defs.h:109*

```
419:                    u1[1] src_array_type = src_cf[2]->get_array_type[3]();
420:                    BasicType[4] src_internal_type = get_internal_type[5](src_array_type[6]);
421:                    if (src_internal_type[7] == UnrecognizedType[8])
422:                            return VM_ERROR_ArrayStoreExceptionClass[9];
423:
424:                    if (!is_convertable[10](src_internal_type[7])) // from src internal type to own internal type
425:                            return VM_ERROR_ArrayStoreExceptionClass[9];
426:
427:                    u4[11] src_offset_address = src_offset[12]*BasicTypes[13][src_internal_type[7]].size[14];
428:                    u4[11] dst_offset_address = dst_offset[15]*BasicTypes[13][dst_internal_type[16]].size[14];
429:
430:                    for (int i = 0; i < length[17]; i++)
431:                    {
432:                            u1[1] * src_element_address =
433:                                    src[18]->data_start[19]->address[20] + src_offset_address[21] + BasicTypes[13][src_internal_type[7]].size[14]*i[22];
434:                            u1[1] * dst_element_address =
435:                                    data_start[19]->address[20] + dst_offset_address[23] + BasicTypes[13][dst_internal_type[16]].size[14]*i[22];
436:
437:                            memcpy(dst_element_address[24], src_element_address[25], BasicTypes[13][src_internal_type[7]].size[14]);
438:                    }
439:
440:            }
441:
442:
443:
444:
445:            //------------------------------ monitor ------------------------------
446:
447:            void monitor[26]::acquire(Thread[27] * thread)
448:            {
449:
450:                    debug_file << "---------------------------------------" << endl;
451:                    debug_file << thread[28]->get_id[29]() << " trying to acquire " << this << endl;
452:
453:                    if (!owner[30]) // if the monitor does not have an owner
454:                    {
455:                            // acquire the monitor
456:                            counter[31] = 1;
457:                            owner[30] = thread[28];
458:                            // continue execution
459:
460:                            debug_file << thread[28]->get_id[29]() << " success " << endl;
461:                            return;
462:                    }
463:
464:                    if (thread[28] == owner[30])
465:                    {
466:                            // increment the number of times this monitor was acquired by the same thread
467:                            counter[31]++;
468:                            // continue running
```

Footnotes:

1: *defs.h:168*
2: *ObjectData.cpp:412*
3: *ClassFile.h:395*
4: *defs.h:208*
5: *ObjectData.cpp:339*
6: *ObjectData.cpp:419*
7: *ObjectData.cpp:420*
8: *defs.h:221*
9: *defs.h:109*
10: *ObjectData.cpp:394*
11: *defs.h:174*
12: *ObjectData.cpp:400*
13: *ObjectData.cpp:24*
14: *defs.h:227*
15: *ObjectData.cpp:400*
16: *ObjectData.cpp:415*
17: *ObjectData.cpp:400*
18: *ObjectData.cpp:400*
19: *ObjectData.h:71*
20: *HeapManager.h:46*
21: *ObjectData.cpp:427*
22: *ObjectData.cpp:430*
23: *ObjectData.cpp:428*
24: *ObjectData.cpp:434*
25: *ObjectData.cpp:432*
26: *ObjectData.h:14*
27: *Thread.h:17*
28: *ObjectData.cpp:447*
29: *Thread.h:82*
30: *ObjectData.h:17*
31: *ObjectData.h:19*

```
469:
470:                    debug_file << thread[1]->get_id[2]() << " success - already owner; counter=" << counter[3] << end
        l;
471:
472:                    // This small patch is done for the special case that the thread ackquires
473:                    // the lock after being awakened from the "wait" call
474:            /*      if (thread->lock_counter > 0)
475:                    {
476:                            counter = thread->lock_counter;
477:
478:                            debug_file << thread->get_id() << " success - awakened from wait; counter=" << co
        unter << endl;
479:
480:                            thread->lock_counter = 0;
481:                    }
482:            */
483:            }
484:            else
485:            {
486:                    debug_file << thread[1]->get_id[2]() << " failed; going to sleep " << endl;
487:
488:                    // another thread owns this monitor -
489:                    // put this thread into the waiting set of the monitor
490:                    waiting_set[4].add_element[5](thread[1]);
491:                    // set thread's status as Waiting
492:                    thread[1]->set_waiting[6]();
493:                    // yield
494:            }
495:    }
496:
497:    // returns false if the thread attempting to release this monitor
498:    // is not its owner
499:    int monitor[7]::release(Thread[8] * thread)
500:    {
501:            debug_file << "----------------------------------------" << endl;
502:            debug_file << thread[9]->get_id[2]() << " is going to release " << this << endl;
503:
504:            if (!owner[10])
505:                    return 0;
506:
507:            if (thread[9] != owner[10])
508:                    return 0;
509:
510:            if (--counter[3] == 0)
511:            {
512:                    debug_file << "success; monitor released" << endl;
513:                    // monitor released
514:                    owner[10] = NULL;
515:
516:                    // release next thread that is waiting for this monitor if any
517:                    if (waiting_set[4].size[11]() == 0)
518:                            return 1; // nobody waits for the monitor
519:
```

Footnotes:

1: *ObjectData.cpp:447*
2: *Thread.h:82*
3: *ObjectData.h:19*
4: *ObjectData.h:21*
5: *vector.h:40*
6: *Thread.h:167*
7: *ObjectData.h:14*
8: *Thread.h:17*
9: *ObjectData.cpp:499*
10: *ObjectData.h:17*
11: *vector.h:23*

```
520:                        // otherwise, get the next waiting thread (in the queue order)
521:                        Thread[1] * candidate = waiting_set[2].first_element[3]();
522:                        assert(candidate[4] != NULL);
523:                        waiting_set[2].remove_element_at[5](0);
524:
525:                        // make the winner thread to be the new owner of the monitor
526:                        counter[6] = 1;
527:                        owner[7] = candidate[4];
528:
529:                        debug_file << "the new owner is " << candidate[4]->get_id[8]() << endl;
530:
531:                        // mark this thread suitable for execution
532:                        candidate[4]->set_running[9]();
533:                }
534:
535:                        debug_file << "success, but not yet released; counter=" << counter[6] << endl;
536:                return 1;
537:        }
538:
539:
540:
```

Footnotes:

1: *Thread.h:17*
2: *ObjectData.h:21*
3: *vector.h:70*
4: *ObjectData.cpp:521*
5: *vector.h:49*
6: *ObjectData.h:19*
7: *ObjectData.h:17*
8: *Thread.h:82*
9: *Thread.h:166*

```
1:  #include <memory.h>
2:  #include <iostream.h>
3:  #include <assert.h>
4:
5:  #include "Thread.h"
6:  #include "Stack.h"
7:  #include "AttributeInfo.h"
8:  #include "ClassFile.h"
9:  #include "opcodes.h"
10: #include "HandlePool.h"
11:
12:
13: Stack[1]::Stack(Thread[2] * _owner_thread) : owner_thread[3](_owner_thread)
14: {
15:         owner_thread[3]->get_jvm[4]()->get_stack_heap_manager[5]()->alloc[6](MAX_STACK_SIZE[7], stack_start[8]);
16: }
17:
18: Stack[1]::~Stack()
19: {
20:         owner_thread[3]->get_jvm[4]()->get_stack_heap_manager[5]()->dealloc[9](stack_start[8]);
21: }
22:
23: // Creates and pushes a new stack frame on top of the stack;
24: // returns pointer to the newely created stack frame
25: stack_frame[10] * Stack[1]::push_frame(u2[11] max_locals, u2[11] max_stack)
26: {
27:         word[12] * next_frame_start;
28:         if (frames[13].size[14]() > 0)
29:         {
30:                 stack_frame[10] * top_frame = frames[13].last_element[15]();
31:                 next_frame_start[16] = top_frame[17]->frame_start[18] + top_frame[17]->size[19];
32:         }
33:         else
34:                 next_frame_start[16] = (word[12]*)stack_start[8]->address[20];
35:
36:         stack_frame[10] * frame = new stack_frame[21](owner_thread[3], (u1[22]*)next_frame_start[16], max_locals[23], max_
    stack[24]);
37:         frames[13].add_element[25](frame[26]);
38:
39:         return frame[26];
40: }
41:
42: // This function will be used for the frames created by a method invokation;
43: // The operand stack of the calling frame will become the local variables
44: // of the frame being created
45: stack_frame[10] * Stack[1]::push_overlapped_frame(word[12] * next_frame_start)
46: {
47:         stack_frame[10] * frame = new stack_frame[21](owner_thread[3], (u1[22]*)next_frame_start[27]);
48:         frames[13].add_element[25](frame[28]);
49:
50:         return frame[28];
51: }
52:
```

Footnotes:

1: *Stack.h:264*
2: *Thread.h:17*
3: *Stack.h:269*
4: *Thread.h:179*
5: *VirtualMachine.h:333*
6: *HeapManager.cpp:60*
7: *defs.h:454*
8: *Stack.h:272*
9: *HeapManager.cpp:117*
10: *Stack.h:129*
11: *defs.h:172*
12: *defs.h:195*
13: *Stack.h:275*
14: *vector.h:23*
15: *vector.h:74*
16: *Stack.cpp:27*
17: *Stack.cpp:30*
18: *Stack.h:142*
19: *Stack.h:140*
20: *HeapManager.h:46*
21: *Stack.cpp:94*
22: *defs.h:168*
23: *Stack.cpp:25*
24: *Stack.cpp:25*
25: *vector.h:40*
26: *Stack.cpp:36*
27: *Stack.cpp:45*
28: *Stack.cpp:47*

```
53:         // Pops the top stack frame off the stack and destroys it
54:         void Stack[1]::pop_frame()
55:         {
56:                 stack_frame[2] * frame = frames[3].last_element[4]();
57:                 frames[3].remove_element_at[5](frames[3].size[6]()-1);
58:
59:                 // We now must delete all the references in this frame's operand stack
60:                 // and local variables from the global roots repository
61:                 frame[7]->handle_pool[8]->remove_roots[9](&(frame[7]->roots[10]));
62:
63:                 delete frame[7];
64:         }
65:
66:         // Returns the topmost frame on the stack;
67:         // If the stack is empty returns NULL
68:         stack_frame[2] * Stack[1]::get_top_frame()
69:         {
70:                 if (frames[3].is_empty[11]())
71:                         return NULL;
72:                 return frames[3].last_element[4]();
73:         }
74:
75:         void Stack[1]::debug_print(ostream & out)
76:         {
77:                 out[12] << endl << endl;
78:                 out[12] << "=== STACK SNAPSHOT ===" << endl;
79:
80:                 for (int i = frames[3].size[6]()-1; i >= 0; i--)
81:                 {
82:                         out[12] << "FRAME #" << frames[3].size[6]()-i[13]-1 << endl;
83:                         stack_frame[2] * frame = frames[3].element_at[14](i[13]);
84:                         frame[15]->debug_print[16](out[12]);
85:                 }
86:
87:                 out[12] << "======================" << endl;
88:                 out[12] << endl << endl;
89:         }
90:
91:         ////////////////////// stack_frame //////////////////////////
92:
93:         // Note, that max_locals and max_stack are measured in words
94:         stack_frame[2]::stack_frame(Thread[17] * _owner, u1[18] * _frame_start, u2[19] max_locals, u2[19] max_stack) :
95:                 owner_thread[20](_owner[21]), current_code_attribute[22](NULL), current_pc_register[23](0),
96:                 monitor_to_release[24](NULL), words_to_pop[25](0)
97:         {
98:             handle_pool[8] = owner_thread[20]->get_jvm[26]()->get_handle_pool[27]();
99:
100:                size[28] = max_locals[29] + max_stack[30];
101:                frame_start[31] = (word[32]*)_frame_start[33];
102:
103:                // Local variables area starts from the beginning of the stack frame
104:                locals_start[34] = frame_start[31];
105:
```

Footnotes:

1: *Stack.h:264*
2: *Stack.h:129*
3: *Stack.h:275*
4: *vector.h:74*
5: *vector.h:49*
6: *vector.h:23*
7: *Stack.cpp:56*
8: *Stack.h:132*
9: *HandlePool.cpp:155*
10: *Stack.h:138*
11: *vector.h:24*
12: *Stack.cpp:75*
13: *Stack.cpp:80*
14: *vector.h:36*
15: *Stack.cpp:83*
16: *Stack.cpp:410*
17: *Thread.h:17*
18: *defs.h:168*
19: *defs.h:172*
20: *Stack.h:131*
21: *Stack.cpp:94*
22: *Stack.h:153*
23: *Stack.h:156*
24: *Stack.h:160*
25: *Stack.h:166*
26: *Thread.h:179*
27: *VirtualMachine.h:336*
28: *Stack.h:140*
29: *Stack.cpp:94*
30: *Stack.cpp:94*
31: *Stack.h:142*
32: *defs.h:195*
33: *Stack.cpp:94*
34: *Stack.h:144*

```
106:                 // Operand stack will begin right after the local variables area;
107:                 // use some pointer arithmetic here
108:                 operand_stack_start[1] = frame_start[2] + max_locals[3];
109:                 operand_stack_top[4] = 0;
110:         }
111:
112:         // This constructor will be used for the frames created by a method invokation;
113:         // The operand stack of the calling frame will become the local variables
114:         // of the frame being created.
115:         // This function must be used in conjunction with define_size()
116:         stack_frame[5]::stack_frame(Thread[6] * _owner, u1[7] * _frame_start) :
117:                         owner_thread[8](_owner[9]), current_code_attribute[10](NULL), current_pc_register[11](0),
118:                         monitor_to_release[12](NULL), words_to_pop[13](0)
119:         {
120:             handle_pool[14] = owner_thread[8]->get_jvm[15]()->get_handle_pool[16]();
121:
122:                 size[17] = 0; // NOTE: size is undefined; it must be defined when possible
123:                 frame_start[2] = (word[18]*)_frame_start[19];
124:
125:                 // Local variables area strats from the beginning of the stack frame
126:                 locals_start[20] = frame_start[2];
127:
128:                 // NOTE: operand stack is undefined; it must be defined when possible
129:                 operand_stack_start[1] = NULL;
130:                 operand_stack_top[4] = 0;
131:         }
132:
133:         // Determine size and operand stack of the uppermost frame stack
134:         // (used in conjunction with the constructor for overlapped frames)
135:         void stack_frame[5]::define_size(u2[21] max_locals, u2[21] max_stack)
136:         {
137:                 size[17] = max_locals[22] + max_stack[23];
138:
139:                 // Operand stack will begin right after the local variables area;
140:                 // use some pointer arithmetic here
141:                 operand_stack_start[1] = frame_start[2] + max_locals[22];
142:                 operand_stack_top[4] = 0;
143:         }
144:
145:         // Basic operations over the operand stack
146:
147:         void stack_frame[5]::push_int(su4[24] value)
148:         {
149:                 memcpy(operand_stack_start[1] + operand_stack_top[4], &value[25], sizeof(su4[24]));
150:                 operand_stack_top[4]++;
151:         }
152:
153:         void stack_frame[5]::push_long(su8[26] value)
154:         {
155:                 // Note, that high_bytes and low_bytes are unsigned
156:                 u4[27] low_bytes = value[28] & 0xFFFFFFFF;
157:                 u4[27] high_bytes = value[28] >> 32;
158:
```

Footnotes:

1: *Stack.h:146*
2: *Stack.h:142*
3: *Stack.cpp:94*
4: *Stack.h:149*
5: *Stack.h:129*
6: *Thread.h:17*
7: *defs.h:168*
8: *Stack.h:131*
9: *Stack.cpp:116*
10: *Stack.h:153*
11: *Stack.h:156*
12: *Stack.h:160*
13: *Stack.h:166*
14: *Stack.h:132*
15: *Thread.h:179*
16: *VirtualMachine.h:336*
17: *Stack.h:140*
18: *defs.h:195*
19: *Stack.cpp:116*
20: *Stack.h:144*
21: *defs.h:172*
22: *Stack.cpp:135*
23: *Stack.cpp:135*
24: *defs.h:175*
25: *Stack.cpp:147*
26: *defs.h:177*
27: *defs.h:174*
28: *Stack.cpp:153*

```
159:                memcpy(operand_stack_start[1] + operand_stack_top[2], &low_bytes[3], sizeof(u4[4]));
160:                operand_stack_top[2]++;
161:                memcpy(operand_stack_start[1] + operand_stack_top[2], &high_bytes[5], sizeof(u4[4]));
162:                operand_stack_top[2]++;
163:        }
164:
165:        void stack_frame[6]::push_float(float value)
166:        {
167:                memcpy(operand_stack_start[1] + operand_stack_top[2], &value[7], sizeof(float));
168:                operand_stack_top[2]++;
169:        }
170:
171:        void stack_frame[6]::push_double(double value)
172:        {
173:
174:                u8[8] converter;
175:                memcpy(&converter[9],&value[10],sizeof(double));
176:
177:                // Note, that high_bytes and low_bytes are unsigned
178:                u4[4] low_bytes = converter[9] & 0xFFFFFFFF;
179:                u4[4] high_bytes = converter[9] >> 32;
180:
181:                memcpy(operand_stack_start[1] + operand_stack_top[2], &low_bytes[11], sizeof(u4[4]));
182:                operand_stack_top[2]++;
183:                memcpy(operand_stack_start[1] + operand_stack_top[2], &high_bytes[12], sizeof(u4[4]));
184:                operand_stack_top[2]++;
185:        }
186:
187:        void stack_frame[6]::push_reference(word[13] value)
188:        {
189:                operand_stack_start[1][operand_stack_top[2]++] = value[14];
190:
191:                // Note, that once a reference is pushed onto the operand stack it
192:                // is automatically added to this stack frame "roots" and to the global
193:                // roots repository
194:                if (value[14] == null[15])
195:                        return;
196:                InstanceData[16] * id = handle_pool[17]->get_instance[18](value[14]);
197:                roots[19].add_element[20](id[21]);
198:                handle_pool[17]->put_root[22](id[21]);
199:        }
200:
201:        void stack_frame[6]::push_returnType(word[13] value)
202:        {
203:                operand_stack_start[1][operand_stack_top[2]++] = value[23];
204:        }
205:
206:        su4[24] stack_frame[6]::pop_int()
207:        {
208:                su4[24] result;
209:                --operand_stack_top[2];
210:                memcpy(&result[25], operand_stack_start[1]+operand_stack_top[2], sizeof(int));
211:                return result[25];
```

Footnotes:

1: *Stack.h:146*
2: *Stack.h:149*
3: *Stack.cpp:156*
4: *defs.h:174*
5: *Stack.cpp:157*
6: *Stack.h:129*
7: *Stack.cpp:165*
8: *defs.h:176*
9: *Stack.cpp:174*
10: *Stack.cpp:171*
11: *Stack.cpp:178*
12: *Stack.cpp:179*
13: *defs.h:195*
14: *Stack.cpp:187*
15: *defs.h:201*
16: *ObjectData.h:113*
17: *Stack.h:132*
18: *HandlePool.cpp:25*
19: *Stack.h:138*
20: *vector.h:40*
21: *Stack.cpp:196*
22: *HandlePool.cpp:145*
23: *Stack.cpp:201*
24: *defs.h:175*
25: *Stack.cpp:208*

```
212:        }
213:
214:        su8[1] stack_frame[2]::pop_long()
215:        {
216:                u4[3] low_bytes, high_bytes;
217:
218:                --operand_stack_top[4];
219:                memcpy(&high_bytes[5], operand_stack_start[6]+operand_stack_top[4], sizeof(u4[3]));
220:                --operand_stack_top[4];
221:                memcpy(&low_bytes[7], operand_stack_start[6]+operand_stack_top[4], sizeof(u4[3]));
222:
223:                // Note that high_bytes is converted into signed 8-byte integer before shifting
224:                return ( ((su8[1])high_bytes[5]) << 32) + low_bytes[7];
225:        }
226:
227:        float stack_frame[2]::pop_float()
228:        {
229:                float result;
230:                --operand_stack_top[4];
231:                memcpy(&result[8], operand_stack_start[6]+operand_stack_top[4], sizeof(float));
232:                return result[8];
233:        }
234:
235:        double stack_frame[2]::pop_double()
236:        {
237:                u4[3] low_bytes, high_bytes;
238:
239:                --operand_stack_top[4];
240:                memcpy(&high_bytes[9], operand_stack_start[6]+operand_stack_top[4], sizeof(u4[3]));
241:                --operand_stack_top[4];
242:                memcpy(&low_bytes[10], operand_stack_start[6]+operand_stack_top[4], sizeof(u4[3]));
243:
244:                // Note that high_bytes is converted into signed 8-byte integer before shifting
245:                 su8[1] bits = ( ((su8[1])high_bytes[9]) << 32) + low_bytes[10];
246:                 return CONSTANT_Double_info[11]::get_double_value[12](bits[13]);
247:        }
248:
249:        word[14] stack_frame[2]::pop_reference()
250:        {
251:                return operand_stack_start[6][--operand_stack_top[4]];
252:        }
253:
254:        word[14] stack_frame[2]::pop_returnType()
255:        {
256:                return operand_stack_start[6][--operand_stack_top[4]];
257:        }
258:
259:        void stack_frame[2]::pop_word()
260:        {
261:                --operand_stack_top[4];
262:        }
263:
264:        void stack_frame[2]::pop2_word()
```

Footnotes:

1: *defs.h:177*
2: *Stack.h:129*
3: *defs.h:174*
4: *Stack.h:149*
5: *Stack.cpp:216*
6: *Stack.h:146*
7: *Stack.cpp:216*
8: *Stack.cpp:229*
9: *Stack.cpp:237*
10: *Stack.cpp:237*
11: *ConstantPool.h:244*
12: *ConstantPool.cpp:212*
13: *Stack.cpp:245*
14: *defs.h:195*

```
265:        {
266:                operand_stack_top[1] -= 2;
267:        }
268:
269:        void stack_frame[2]::dup_word()
270:        {
271:                word[3] value = operand_stack_start[4][operand_stack_top[1]-1];
272:                operand_stack_start[4][operand_stack_top[1]++] = value[5];
273:        }
274:
275:        void stack_frame[2]::dup2_word()
276:        {
277:                word[3] value1 = operand_stack_start[4][operand_stack_top[1]-1];
278:                word[3] value2 = operand_stack_start[4][operand_stack_top[1]-2];
279:                operand_stack_start[4][operand_stack_top[1]++] = value2[6];
280:                operand_stack_start[4][operand_stack_top[1]++] = value1[7];
281:        }
282:
283:        void stack_frame[2]::dup_x1()
284:        {
285:                word[3] value1 = operand_stack_start[4][operand_stack_top[1]-1];
286:                word[3] value2 = operand_stack_start[4][operand_stack_top[1]-2];
287:                operand_stack_start[4][operand_stack_top[1]-2] = value1[8];
288:                operand_stack_start[4][operand_stack_top[1]-1] = value2[9];
289:                operand_stack_start[4][operand_stack_top[1]++] = value1[8];
290:        }
291:
292:        void stack_frame[2]::swap_word()
293:        {
294:                word[3] value1 = operand_stack_start[4][--operand_stack_top[1]];
295:                word[3] value2 = operand_stack_start[4][--operand_stack_top[1]];
296:                operand_stack_start[4][operand_stack_top[1]++] = value1[10];
297:                operand_stack_start[4][operand_stack_top[1]++] = value2[11];
298:        }
299:
300:        void stack_frame[2]::pop_words(unsigned int n)
301:        {
302:                operand_stack_top[1] -= n[12];
303:        }
304:
305:        void stack_frame[2]::store_int(su4[13] value, u1[14] index)
306:        {
307:                memcpy((locals_start[15]+index[16]), &value[17], sizeof(su4[13]));
308:        }
309:
310:        void stack_frame[2]::store_long(su8[18] value, u1[14] index)
311:        {
312:                // Note, that high_bytes and low_bytes are unsigned
313:                u4[19] low_bytes = value[20] & 0xFFFFFFFF;
314:                u4[19] high_bytes = value[20] >> 32;
315:
316:                memcpy((locals_start[15]+index[21]), &low_bytes[22], sizeof(u4[19]));
317:                memcpy((locals_start[15]+index[21]+1), &high_bytes[23], sizeof(u4[19]));
```

Footnotes:

1: *Stack.h:149*
2: *Stack.h:129*
3: *defs.h:195*
4: *Stack.h:146*
5: *Stack.cpp:271*
6: *Stack.cpp:278*
7: *Stack.cpp:277*
8: *Stack.cpp:285*
9: *Stack.cpp:286*
10: *Stack.cpp:294*
11: *Stack.cpp:295*
12: *Stack.cpp:300*
13: *defs.h:175*
14: *defs.h:168*
15: *Stack.h:144*
16: *Stack.cpp:305*
17: *Stack.cpp:305*
18: *defs.h:177*
19: *defs.h:174*
20: *Stack.cpp:310*
21: *Stack.cpp:310*
22: *Stack.cpp:313*
23: *Stack.cpp:314*

```
318:        }
319:
320:        void stack_frame[1]::store_float(float value, u1[2] index)
321:        {
322:                memcpy((locals_start[3]+index[4]), &value[5], sizeof(float));
323:        }
324:
325:        void stack_frame[1]::store_double(double value, u1[2] index)
326:        {
327:                u8[6] converter;
328:                memcpy(&converter[7],&value[8],sizeof(double));
329:
330:                // Note, that high_bytes and low_bytes are unsigned
331:                u4[9] low_bytes = converter[7] & 0xFFFFFFFF;
332:                u4[9] high_bytes = converter[7] >> 32;
333:
334:                memcpy((locals_start[3]+index[10]), &low_bytes[11], sizeof(u4[9]));
335:                memcpy((locals_start[3]+index[10]+1), &high_bytes[12], sizeof(u4[9]));
336:        }
337:
338:        void stack_frame[1]::store_reference(word[13] value, u1[2] index)
339:        {
340:                memcpy((locals_start[3]+index[14]), &value[15], sizeof(word[13]));
341:
342:                // Note, that once a reference is pushed onto the operand stack it
343:                // is automatically added to this stack frame "roots" and to the global
344:                // roots repository
345:                if (value[15] == null[16])
346:                        return;
347:                InstanceData[17] * id = handle_pool[18]->get_instance[19](value[15]);
348:                roots[20].add_element[21](id[22]);
349:                handle_pool[18]->put_root[23](id[22]);
350:        }
351:
352:        void stack_frame[1]::store_returnType(word[13] value, u1[2] index)
353:        {
354:                memcpy((locals_start[3]+index[24]), &value[25], sizeof(word[13]));
355:        }
356:
357:        su4[26] stack_frame[1]::get_int(u1[2] index)
358:        {
359:                su4[26] result;
360:                memcpy(&result[27], (locals_start[3]+index[28]), sizeof(su4[26]));
361:                return result[27];
362:        }
363:
364:        su8[29] stack_frame[1]::get_long(u1[2] index)
365:        {
366:                u4[9] low_bytes;
367:                u4[9] high_bytes;
368:
369:                memcpy(&low_bytes[30], (locals_start[3]+index[31]), sizeof(u4[9]));
370:                memcpy(&high_bytes[32], (locals_start[3]+index[31]+1), sizeof(u4[9]));
```

Footnotes:

1: *Stack.h:129*
2: *defs.h:168*
3: *Stack.h:144*
4: *Stack.cpp:320*
5: *Stack.cpp:320*
6: *defs.h:176*
7: *Stack.cpp:327*
8: *Stack.cpp:325*
9: *defs.h:174*
10: *Stack.cpp:325*
11: *Stack.cpp:331*
12: *Stack.cpp:332*
13: *defs.h:195*
14: *Stack.cpp:338*
15: *Stack.cpp:338*
16: *defs.h:201*
17: *ObjectData.h:113*
18: *Stack.h:132*
19: *HandlePool.cpp:25*
20: *Stack.h:138*
21: *vector.h:40*
22: *Stack.cpp:347*
23: *HandlePool.cpp:145*
24: *Stack.cpp:352*
25: *Stack.cpp:352*
26: *defs.h:175*
27: *Stack.cpp:359*
28: *Stack.cpp:357*
29: *defs.h:177*
30: *Stack.cpp:366*
31: *Stack.cpp:364*
32: *Stack.cpp:367*

```
371:
372:                    // Note that high_bytes is converted into signed 8-byte integer before shifting
373:                    return ( ((su8[1])high_bytes[2]) << 32) + low_bytes[3];
374:            }
375:
376:            float stack_frame[4]::get_float(u1[5] index)
377:            {
378:                    float result;
379:                    memcpy(&result[6], (locals_start[7]+index[8]), sizeof(float));
380:                    return result[6];
381:            }
382:
383:            double stack_frame[4]::get_double(u1[5] index)
384:            {
385:                    u4[9] low_bytes;
386:                    u4[9] high_bytes;
387:
388:                    memcpy(&low_bytes[10], (locals_start[7]+index[11]), sizeof(u4[9]));
389:                    memcpy(&high_bytes[12], (locals_start[7]+index[11]+1), sizeof(u4[9]));
390:
391:                    // Note that high_bytes is converted into signed 8-byte integer before shifting
392:                    su8[1] bits = ( ((su8[1])high_bytes[12]) << 32) + low_bytes[10];
393:                    return CONSTANT_Double_info[13]::get_double_value[14](bits[15]);
394:            }
395:
396:            word[16] stack_frame[4]::get_reference(u1[5] index)
397:            {
398:                    word[16] result;
399:                    memcpy(&result[17], (locals_start[7]+index[18]), sizeof(word[16]));
400:                    return result[17];
401:            }
402:
403:            word[16] stack_frame[4]::get_returnType(u1[5] index)
404:            {
405:                    word[16] result;
406:                    memcpy(&result[19], (locals_start[7]+index[20]), sizeof(word[16]));
407:                    return result[19];
408:            }
409:
410:            void stack_frame[4]::debug_print(ostream & out)
411:            {
412:                    out[21] << "--- STACK FRAME SNAPSHOT ---" << endl;
413:                    out[21] << "    frame start            = " << frame_start[22] << endl;
414:                    out[21] << "    frame size             = " << size[23] - (operand_stack_start[24]-locals_start[7]) << endl;
415:                    out[21] << "    locals start           = " << locals_start[7] << endl;
416:                    out[21] << "    local variables number = " << operand_stack_start[24]-locals_start[7] << endl;
417:                    out[21] << "    operand stack start    = " << operand_stack_start[24] << endl;
418:                    out[21] << "    stack pointer          = " << operand_stack_top[25] << endl;
419:                    if (current_code_attribute[26])
420:                    {
421:                            out[21] << "    current class : ";
422:                            wchar_t * class_name;
423:                            u2[27] length;
```

Footnotes:

1: *defs.h:177*
2: *Stack.cpp:367*
3: *Stack.cpp:366*
4: *Stack.h:129*
5: *defs.h:168*
6: *Stack.cpp:378*
7: *Stack.h:144*
8: *Stack.cpp:376*
9: *defs.h:174*
10: *Stack.cpp:385*
11: *Stack.cpp:383*
12: *Stack.cpp:386*
13: *ConstantPool.h:244*
14: *ConstantPool.cpp:212*
15: *Stack.cpp:392*
16: *defs.h:195*
17: *Stack.cpp:398*
18: *Stack.cpp:396*
19: *Stack.cpp:405*
20: *Stack.cpp:403*
21: *Stack.cpp:410*
22: *Stack.h:142*
23: *Stack.h:140*
24: *Stack.h:146*
25: *Stack.h:149*
26: *Stack.h:153*
27: *defs.h:172*

```
424:                            current_code_attribute[1]->class_file[2]->get_full_class_name[3](class_name[4], length[5]);
425:                            print_wchar[6](class_name[4], length[5], out[7]); out[7] << endl;
426:                            delete [] class_name[4];
427:
428:                            out[7] << "    current method : ";
429:                            method_info[8] * mi = (method_info[8]*)current_code_attribute[1]->field_method_parent[9];
430:                            wchar_t * method_name, * descriptor;
431:                            u2[10] method_name_length, descriptor_length;
432:                            mi[11]->get_method_name[12](method_name[13], method_name_length[14]);
433:                            mi[11]->get_method_descriptor[15](descriptor[16], descriptor_length[17]);
434:                            print_wchar[6](method_name[13], method_name_length[14], out[7]);
435:                            print_wchar[6](descriptor[16], descriptor_length[17], out[7]);
436:                            delete [] method_name[13];
437:                            delete [] descriptor[16];
438:
439:                            out[7] << endl;
440:                            out[7] << "    code executed so far : " << endl;
441:                            unsigned long index = 0;
442:                            u1[18] * code = current_code_attribute[1]->code[19];
443:                            while (index[20] < current_pc_register[21])
444:                            {
445:                                    u1[18] opcode = code[22][index[20]++];
446:                                    out[7] << "        " << opcodes[23][opcode[24]].mnemonic[25] << endl;
447:                                    if (opcodes[23][opcode[24]].argument_number[26] < 0)
448:                                    {
449:                                            out[7] << "variable length command -- breaking..." << endl;
450:                                            break;
451:                                    }
452:                                    for (int i = 0; i < opcodes[23][opcode[24]].argument_number[26]; i++)
453:                                    {
454:                                            index[20]++;
455:                                            //out << "   arg #" << (i+1) << ": " << code[index] << endl;
456:                                    }
457:                            }
458:
459:                    }
460:                    out[7] << "----------------------------" << endl;
461:            }
```

Footnotes:

1: *Stack.h:153*
2: *AttributeInfo.h:43*
3: *ClassFile.cpp:442*
4: *Stack.cpp:422*
5: *Stack.cpp:423*
6: *util.cpp:42*
7: *Stack.cpp:410*
8: *ClassFile.h:92*
9: *AttributeInfo.h:44*
10: *defs.h:172*
11: *Stack.cpp:429*
12: *ClassFile.cpp:168*
13: *Stack.cpp:430*
14: *Stack.cpp:431*
15: *ClassFile.cpp:177*
16: *Stack.cpp:430*
17: *Stack.cpp:431*
18: *defs.h:168*
19: *AttributeInfo.h:68*
20: *Stack.cpp:441*
21: *Stack.h:156*
22: *Stack.cpp:442*
23: *opcodes.cpp:3215*
24: *Stack.cpp:445*
25: *opcodes.h:256*
26: *opcodes.h:257*

```
 1:          #include <assert.h>
 2:          #include <stdio.h>
 3:          #include <iostream.h>
 4:
 5:          #include "Thread.h"
 6:          #include "opcodes.h"
 7:          #include "ClassLoader.h"
 8:          #include "ClassFile.h"
 9:          #include "ConstantPool.h"
10:          #include "ObjectData.h"
11:          #include "NativeHandler.h"
12:          #include "HandlePool.h"
13:
14:
15:          // Start the new thread with the given method's code attribute
16:          // as an initial thread's method
17:          Result[1] Thread[2]::start(Code_attribute[3] * code_attribute, InstanceData[4] * _thread_instance)
18:          {
19:                  current_code_attribute[5] = code_attribute[6];
20:
21:                  // Point the Program Counter register to the first instruction
22:                  pc_register[7] = 0;
23:
24:                  // Determine the current entities
25:                  current_class_file[8] = code_attribute[6]->class_file[9];
26:                  assert(current_class_file[8] != NULL);
27:                  current_cp[10] = current_class_file[8]->constant_pool[11];
28:                  assert(current_cp[10] != NULL);
29:
30:                  current_code_length[12] = code_attribute[6]->code_length[13];
31:                  current_code[14] = code_attribute[6]->code[15];
32:
33:                  if (current_code_length[12] == 0)
34:                          return Success[16]; // thread finished
35:
36:                  assert(current_code[14] != NULL);
37:
38:                  thread_instance[17] = _thread_instance[18];
39:
40:                  // Create a stack for this new thread
41:                  stack[19] = new Stack[20](this);
42:
43:                  // Push the stack frame of the current method onto the stack
44:                  current_stack_frame[21] = stack[19]->push_frame[22](current_code_attribute[5]->max_locals[23], current_code_att
          ribute[5]->max_stack[24]);
45:
46:                  // Mark this thread as suitable for running
47:                  set_running[25]();
48:
49:                  // Put this thread's instance reference into the local variable 0
50:                  if (thread_instance[17] != NULL)
51:                          current_stack_frame[21]->store_reference[26](thread_instance[17]->get_reference[27](),0);
52:
```

Footnotes:

1: *defs.h:29*
2: *Thread.h:17*
3: *AttributeInfo.h:55*
4: *ObjectData.h:113*
5: *Thread.h:155*
6: *Thread.cpp:17*
7: *Thread.h:159*
8: *Thread.h:149*
9: *AttributeInfo.h:43*
10: *Thread.h:150*
11: *ClassFile.h:159*
12: *Thread.h:152*
13: *AttributeInfo.h:67*
14: *Thread.h:151*
15: *AttributeInfo.h:68*
16: *defs.h:33*
17: *Thread.h:162*
18: *Thread.cpp:17*
19: *Thread.h:42*
20: *Stack.cpp:13*
21: *Thread.h:148*
22: *Stack.cpp:25*
23: *AttributeInfo.h:66*
24: *AttributeInfo.h:65*
25: *Thread.h:166*
26: *Stack.cpp:338*
27: *ObjectData.h:142*

```
53:                     return Success[1];
54:             }
55:
56:             void Thread[2]::destroy()
57:             {
58:                     // Stack's destructor will deallocate memory area on the heap
59:                     // allocated to this thread's stack
60:                     if (stack[3])
61:                             delete stack[3];
62:             }
63:
64:             // This function switches the contexts between the stack frame of the calling method
65:             // and the stack frame of the new method being called
66:             void Thread[2]::context_switch_up(ClassFile[4] * new_method_class_file,
67:                                                             Code_attribute[5] * new_code_attribute,
68:                                                             stack_frame[6] * new_stack_frame)
69:             {
70:                     // Save current entities in the previous stack frame
71:                     current_stack_frame[7]->current_code_attribute[8] = current_code_attribute[9];
72:                     current_stack_frame[7]->current_pc_register[10] = pc_register[11];
73:
74:                     // Prepare all the current entities of this thread
75:                     // Current class becomes the one whose method is going to be executed
76:                     current_class_file[12] = new_method_class_file[13];
77:                     current_cp[14] = current_class_file[12]->constant_pool[15];
78:                     current_code_attribute[9] = new_code_attribute[16];
79:                     if (current_code_attribute[9])
80:                     {
81:                             current_code[17] = current_code_attribute[9]->code[18];
82:                             current_code_length[19] = current_code_attribute[9]->code_length[20];
83:                             // Define size and the operand stack start of the new stack frame
84:                             new_stack_frame[21]->define_size[22](current_code_attribute[9]->max_locals[23], current_code_attribu
           te[9]->max_stack[24]);
85:                     }
86:                     else
87:                     {
88:                             // The code attribute is NULL in case of a native method
89:                             current_code[17] = NULL;
90:                             current_code_length[19] = 0;
91:                             // Define size and the operand stack start of the new stack frame
92:                             new_stack_frame[21]->define_size[22](0,0);
93:                     }
94:
95:                     // Make the new stack frame current
96:                     current_stack_frame[7] = new_stack_frame[21];
97:
98:                     // Set the program counter to point to the first instruction in the new method
99:                     pc_register[11] = 0;
100:
101:                    // We are ready to run the invoked method (the next "step" instruction
102:                    // will get the first opcode of the newely invoked method)
103:            }
104:
```

Footnotes:

1: *defs.h:33*
2: *Thread.h:17*
3: *Thread.h:42*
4: *ClassFile.h:136*
5: *AttributeInfo.h:55*
6: *Stack.h:129*
7: *Thread.h:148*
8: *Stack.h:153*
9: *Thread.h:155*
10: *Stack.h:156*
11: *Thread.h:159*
12: *Thread.h:149*
13: *Thread.cpp:66*
14: *Thread.h:150*
15: *ClassFile.h:159*
16: *Thread.cpp:67*
17: *Thread.h:151*
18: *AttributeInfo.h:68*
19: *Thread.h:152*
20: *AttributeInfo.h:67*
21: *Thread.cpp:68*
22: *Stack.cpp:135*
23: *AttributeInfo.h:66*
24: *AttributeInfo.h:65*

```
105:        // Switch contexts between the method finishing the execution and the method
106:        // that called it
107:        void Thread[1]::context_switch_down(stack_frame[2] * prev_stack_frame)
108:        {
109:                current_code_attribute[3] = prev_stack_frame[4]->current_code_attribute[5];
110:                current_class_file[6] = current_code_attribute[3]->class_file[7];
111:                current_cp[8] = current_class_file[6]->constant_pool[9];
112:
113:                current_code[10] = current_code_attribute[3]->code[11];
114:                current_code_length[12] = current_code_attribute[3]->code_length[13];
115:
116:                // Make the topmost stack frame current
117:                current_stack_frame[14] = prev_stack_frame[4];
118:
119:                // Set the program counter to point to the instruction following the invokeXXX
120:                // instruction of the method that becomes current
121:                pc_register[15] = prev_stack_frame[4]->current_pc_register[16];
122:
123:                // We are ready to run the next instruction of the method that becomes current
124:        }
125:
126:        Result[17] Thread[1]::release_monitor()
127:        {
128:                if (!current_stack_frame[14]->monitor_to_release[18])
129:                        return Success[19];
130:                else
131:                {
132:                        if (!current_stack_frame[14]->monitor_to_release[18]->release[20](this))
133:                                return VM_ERROR_IllegalMonitorStateException[21];
134:                        else
135:                        {
136:                                current_stack_frame[14]->monitor_to_release[18] = NULL;
137:                                return Success[19];
138:                        }
139:                }
140:        }
141:
142:        Result[17] Thread[1]::return_from_method()
143:        {
144:                // Release the monitor acquired by this method (if any)
145:                Result[17] result = release_monitor[22]();
146:                if (result[23] != Success[19])
147:                        return result[23];
148:
149:                // Remove the current frame stack from the stack
150:                stack[24]->pop_frame[25]();
151:                // Get the previous stack frame
152:                stack_frame[2] * prev_stack_frame = stack[24]->get_top_frame[26]();
153:                if (!prev_stack_frame[27])
154:                {
155:                        // Thread finished running
156:                        set_dead[28]();
157:                        return Success[19];
```

Footnotes:

1: *Thread.h:17*
2: *Stack.h:129*
3: *Thread.h:155*
4: *Thread.cpp:107*
5: *Stack.h:153*
6: *Thread.h:149*
7: *AttributeInfo.h:43*
8: *Thread.h:150*
9: *ClassFile.h:159*
10: *Thread.h:151*
11: *AttributeInfo.h:68*
12: *Thread.h:152*
13: *AttributeInfo.h:67*
14: *Thread.h:148*
15: *Thread.h:159*
16: *Stack.h:156*
17: *defs.h:29*
18: *Stack.h:160*
19: *defs.h:33*
20: *ObjectData.cpp:499*
21: *defs.h:106*
22: *Thread.cpp:126*
23: *Thread.cpp:145*
24: *Thread.h:42*
25: *Stack.cpp:54*
26: *Stack.cpp:68*
27: *Thread.cpp:152*
28: *Thread.h:165*

```
158:                }
159:
160:                // Pop all the parameters that were pushed by the previous frame into its operand stack
161:                // for the method which is now returns
162:                prev_stack_frame[1]->pop_words[2](prev_stack_frame[1]->words_to_pop[3]);
163:
164:                // Make the topmost frame current
165:                context_switch_down[4](prev_stack_frame[1]);
166:
167:                return Success[5];
168:        }
169:
170:        Result[6] Thread[7]::return_int_from_method()
171:        {
172:                // Release the monitor acquired by this method (if any)
173:                Result[6] result = release_monitor[8]();
174:                if (result[9] != Success[5])
175:                        return result[9];
176:
177:                // Pop an integer value from the current stack frame
178:                su4[10] int_value = current_stack_frame[11]->pop_int[12]();
179:
180:                // Remove the current frame stack from the stack
181:                stack[13]->pop_frame[14]();
182:                // Get the previous stack frame
183:                stack_frame[15] * prev_stack_frame = stack[13]->get_top_frame[16]();
184:                if (!prev_stack_frame[17])
185:                {
186:                        // Thread finished running
187:                        set_dead[18]();
188:                        return Success[5];
189:                }
190:
191:                // Pop all the parameters that were pushed by the previous frame into its operand stack
192:                // for the method which is now returns
193:                prev_stack_frame[17]->pop_words[2](prev_stack_frame[17]->words_to_pop[3]);
194:
195:                // Push the returned value onto the previous stack frame
196:                prev_stack_frame[17]->push_int[19](int_value[20]);
197:
198:                // Make the topmost frame current
199:                context_switch_down[4](prev_stack_frame[17]);
200:
201:                return Success[5];
202:        }
203:
204:        Result[6] Thread[7]::return_long_from_method()
205:        {
206:                // Release the monitor acquired by this method (if any)
207:                Result[6] result = release_monitor[8]();
208:                if (result[21] != Success[5])
209:                        return result[21];
210:
```

Footnotes:

1: *Thread.cpp:152*
2: *Stack.cpp:300*
3: *Stack.h:166*
4: *Thread.cpp:107*
5: *defs.h:33*
6: *defs.h:29*
7: *Thread.h:17*
8: *Thread.cpp:126*
9: *Thread.cpp:173*
10: *defs.h:175*
11: *Thread.h:148*
12: *Stack.cpp:206*
13: *Thread.h:42*
14: *Stack.cpp:54*
15: *Stack.h:129*
16: *Stack.cpp:68*
17: *Thread.cpp:183*
18: *Thread.h:165*
19: *Stack.cpp:147*
20: *Thread.cpp:178*
21: *Thread.cpp:207*

```
211:                    // Pop a long value from the current stack frame
212:                    su8[1] long_value = current_stack_frame[2]->pop_long[3]();
213:
214:                    // Remove the current frame stack from the stack
215:                    stack[4]->pop_frame[5]();
216:                    // Get the previous stack frame
217:                    stack_frame[6] * prev_stack_frame = stack[4]->get_top_frame[7]();
218:                    if (!prev_stack_frame[8])
219:                    {
220:                            // Thread finished running
221:                            set_dead[9]();
222:                            return Success[10];
223:                    }
224:
225:                    // Pop all the parameters that were pushed by the previous frame into its operand stack
226:                    // for the method which is now returns
227:                    prev_stack_frame[8]->pop_words[11](prev_stack_frame[8]->words_to_pop[12]);
228:
229:                    // Push the returned value onto the previous stack frame
230:                    prev_stack_frame[8]->push_long[13](long_value[14]);
231:
232:                    // Make the topmost frame current
233:                    context_switch_down[15](prev_stack_frame[8]);
234:
235:                    return Success[10];
236:            }
237:
238:            Result[16] Thread[17]::return_float_from_method()
239:            {
240:                    // Release the monitor acquired by this method (if any)
241:                    Result[16] result = release_monitor[18]();
242:                    if (result[19] != Success[10])
243:                            return result[19];
244:
245:                    // Pop a float value from the current stack frame
246:                    float float_value = current_stack_frame[2]->pop_float[20]();
247:
248:                    // Remove the current frame stack from the stack
249:                    stack[4]->pop_frame[5]();
250:                    // Get the previous stack frame
251:                    stack_frame[6] * prev_stack_frame = stack[4]->get_top_frame[7]();
252:                    if (!prev_stack_frame[21])
253:                    {
254:                            // Thread finished running
255:                            set_dead[9]();
256:                            return Success[10];
257:                    }
258:
259:                    // Pop all the parameters that were pushed by the previous frame into its operand stack
260:                    // for the method which is now returns
261:                    prev_stack_frame[21]->pop_words[11](prev_stack_frame[21]->words_to_pop[12]);
262:
263:                    // Push the returned value onto the previous stack frame
```

Footnotes:

1: *defs.h:177*
2: *Thread.h:148*
3: *Stack.cpp:214*
4: *Thread.h:42*
5: *Stack.cpp:54*
6: *Stack.h:129*
7: *Stack.cpp:68*
8: *Thread.cpp:217*
9: *Thread.h:165*
10: *defs.h:33*
11: *Stack.cpp:300*
12: *Stack.h:166*
13: *Stack.cpp:153*
14: *Thread.cpp:212*
15: *Thread.cpp:107*
16: *defs.h:29*
17: *Thread.h:17*
18: *Thread.cpp:126*
19: *Thread.cpp:241*
20: *Stack.cpp:227*
21: *Thread.cpp:251*

```
264:                    prev_stack_frame[1]->push_float[2](float_value[3]);
265:
266:                    // Make the topmost frame current
267:                    context_switch_down[4](prev_stack_frame[1]);
268:
269:                    return Success[5];
270:            }
271:
272:            Result[6] Thread[7]::return_double_from_method()
273:            {
274:                    // Release the monitor acquired by this method (if any)
275:                    Result[6] result = release_monitor[8]();
276:                    if (result[9] != Success[5])
277:                            return result[9];
278:
279:                    // Pop a double value from the current stack frame
280:                    double double_value = current_stack_frame[10]->pop_double[11]();
281:
282:                    // Remove the current frame stack from the stack
283:                    stack[12]->pop_frame[13]();
284:                    // Get the previous stack frame
285:                    stack_frame[14] * prev_stack_frame = stack[12]->get_top_frame[15]();
286:                    if (!prev_stack_frame[16])
287:                    {
288:                            // Thread finished running
289:                            set_dead[17]();
290:                            return Success[5];
291:                    }
292:
293:                    // Pop all the parameters that were pushed by the previous frame into its operand stack
294:                    // for the method which is now returns
295:                    prev_stack_frame[16]->pop_words[18](prev_stack_frame[16]->words_to_pop[19]);
296:
297:                    // Push the returned value onto the previous stack frame
298:                    prev_stack_frame[16]->push_double[20](double_value[21]);
299:
300:                    // Make the topmost frame current
301:                    context_switch_down[4](prev_stack_frame[16]);
302:
303:                    return Success[5];
304:            }
305:
306:            Result[6] Thread[7]::return_reference_from_method()
307:            {
308:                    // Release the monitor acquired by this method (if any)
309:                    Result[6] result = release_monitor[8]();
310:                    if (result[22] != Success[5])
311:                            return result[22];
312:
313:                    // Pop a reference value from the current stack frame
314:                    word[23] ref_value = current_stack_frame[10]->pop_int[24]();
315:
316:                    // Remove the current frame stack from the stack
```

Footnotes:

1: *Thread.cpp:251*
2: *Stack.cpp:165*
3: *Thread.cpp:246*
4: *Thread.cpp:107*
5: *defs.h:33*
6: *defs.h:29*
7: *Thread.h:17*
8: *Thread.cpp:126*
9: *Thread.cpp:275*
10: *Thread.h:148*
11: *Stack.cpp:235*
12: *Thread.h:42*
13: *Stack.cpp:54*
14: *Stack.h:129*
15: *Stack.cpp:68*
16: *Thread.cpp:285*
17: *Thread.h:165*
18: *Stack.cpp:300*
19: *Stack.h:166*
20: *Stack.cpp:171*
21: *Thread.cpp:280*
22: *Thread.cpp:309*
23: *defs.h:195*
24: *Stack.cpp:206*

```
317:                    stack[1]->pop_frame[2]();
318:                    // Get the previous stack frame
319:                    stack_frame[3] * prev_stack_frame = stack[1]->get_top_frame[4]();
320:                    if (!prev_stack_frame[5])
321:                    {
322:                            // Thread finished running
323:                            set_dead[6]();
324:                            return Success[7];
325:                    }
326:
327:                    // Pop all the parameters that were pushed by the previous frame into its operand stack
328:                    // for the method which is now returns
329:                    prev_stack_frame[5]->pop_words[8](prev_stack_frame[5]->words_to_pop[9]);
330:
331:                    // Push the returned value onto the previous stack frame
332:                    prev_stack_frame[5]->push_reference[10](ref_value[11]);
333:
334:                    // Make the topmost frame current
335:                    context_switch_down[12](prev_stack_frame[5]);
336:
337:                    return Success[7];
338:            }
339:
340:
341:            // Is called by the invokevirtual instruction
342:            Result[13] Thread[14]::invoke_instance_method(CONSTANT_Methodref_info[15] * entry)
343:            {
344:                    // Check whether this CP entry is already resolved; resolve if needed
345:                    if (!entry[16]->is_resolved[17]())
346:                    {
347:                            Result[13] result = entry[16]->resolve_virtual[18]();
348:                            if (result[19] != Success[7])
349:                            {
350:                                    print_error[20](result[19]);
351:                                    return ThreadCannotInvokeInstanceMethod[21];
352:                            }
353:                    }
354:
355:                    if (entry[16]->moffset[22] < 0) // should never happen
356:                            return ThreadCannotInvokeInstanceMethod[21];
357:
358:                    // Now we have to calculate the depth of the overlapped area between two stack frames
359:
360:                    // Determine the size (in words) of all the arguments of this method
361:                    unsigned int arg_size_words = entry[16]->get_arguments_size[23]();
362:
363:                    // Determine the depth of the new stack frame (the address of the beginning of the
364:                    // new stack frame)
365:                    word[24] * new_stack_frame_start =
366:                            current_stack_frame[25]->operand_stack_start[26] + current_stack_frame[25]->operand_stack_top[27] - a
            rg_size_words[28];
367:                    // One more word down the stack to get the address of the object reference
368:                    new_stack_frame_start[29] -= 1;
```

Footnotes:

1: *Thread.h:42*
2: *Stack.cpp:54*
3: *Stack.h:129*
4: *Stack.cpp:68*
5: *Thread.cpp:319*
6: *Thread.h:165*
7: *defs.h:33*
8: *Stack.cpp:300*
9: *Stack.h:166*
10: *Stack.cpp:187*
11: *Thread.cpp:314*
12: *Thread.cpp:107*
13: *defs.h:29*
14: *Thread.h:17*
15: *ConstantPool.h:98*
16: *Thread.cpp:342*
17: *ConstantPool.h:48*
18: *ConstantPool.cpp:334*
19: *Thread.cpp:347*
20: *errors.cpp:35*
21: *defs.h:83*
22: *ConstantPool.h:130*
23: *ConstantPool.cpp:437*
24: *defs.h:195*
25: *Thread.h:148*
26: *Stack.h:146*
27: *Stack.h:149*
28: *Thread.cpp:361*
29: *Thread.cpp:365*

```
369:
370:                    // This variable will remember how many words we have to pop from the calling method's
371:                    // frame upon return
372:                    current_stack_frame[1]->words_to_pop[2] = arg_size_words[3] + 1;
373:
374:                    // Push new stack frame onto the stack so that the operand stack of the
375:                    // calling method's frame will become the local variables storage
376:                    // of the frame of the method being called.
377:                    // This approach facilitates the process of passing parameters to the new method
378:                    stack_frame[4] * new_stack_frame =
379:                            stack[5]->push_overlapped_frame[6](new_stack_frame_start[7]);
380:
381:                    // After creating the new stack frame this way we can be sure that
382:                    // the reference to the object upon which the method should be called
383:                    // is located in the local variable 0 of the new stack frame
384:                    // (the following parameters are located in the local variables 1,2, and so on)
385:                    word[8] objectref = new_stack_frame[9]->get_reference[10](0);
386:
387:                    // Get the class of the object upon which the method must be called
388:                    InstanceData[11] * id = vm[12]->get_handle_pool[13]()->get_instance[14](objectref[15]);
389:                    if (id[16] == null[17])
390:                    {
391:                            // Get rid of the allocated stack frame
392:                            stack[5]->pop_frame[18]();
393:                            raise_exception[19](VM_ERROR_NullPointerException[20]);
394:                            return ThreadCannotInvokeInstanceMethod[21];
395:                    }
396:
397:                    // To obtain the method's offset we will look into the method table of the object
398:                    // upon which the method is executed
399:                    ClassFile[22] * object_class_file = id[16]->class_file[23];
400:                    assert(object_class_file[24] != NULL);
401:
402:                    // Now we have the ClassFile of the object that the given method should run upon
403:                    // and the offset of this method within this ClassFile's method table;
404:                    // Go get the code attribute of the method
405:                    assert(object_class_file[24]->method_table_size[25] > 0);
406:                    assert(entry[26]->moffset[27] < object_class_file[24]->method_table_size[25]);
407:
408:                    method_info[28] * mi = object_class_file[24]->method_table[29][entry[26]->moffset[27]];
409:                    assert(mi[30] != NULL);
410:
411:                    // The current class file is the class file where current method is defined
412:                    ClassFile[22] * new_method_class_file = mi[30]->class_file[31];
413:                    assert(new_method_class_file[32] != NULL);
414:
415:                    Code_attribute[33] * new_code_attribute = mi[30]->get_code_attribute[34]();
416:                    // Note that the new_code_attribute can be NULL in case of a native method !
417:
418:                    // switch contexts between two frames (going up the stack)
419:                    context_switch_up[35](new_method_class_file[32], new_code_attribute[36], new_stack_frame[9]);
420:
421:                    // After everything is set up, we will check whether this method is synchronized.
```

Footnotes:

1: *Thread.h:148*
2: *Stack.h:166*
3: *Thread.cpp:361*
4: *Stack.h:129*
5: *Thread.h:42*
6: *Stack.cpp:45*
7: *Thread.cpp:365*
8: *defs.h:195*
9: *Thread.cpp:378*
10: *Stack.cpp:396*
11: *ObjectData.h:113*
12: *Thread.h:32*
13: *VirtualMachine.h:336*
14: *HandlePool.cpp:25*
15: *Thread.cpp:385*
16: *Thread.cpp:388*
17: *defs.h:201*
18: *Stack.cpp:54*
19: *Thread.cpp:776*
20: *defs.h:105*
21: *defs.h:83*
22: *ClassFile.h:136*
23: *ObjectData.h:74*
24: *Thread.cpp:399*
25: *ClassFile.h:232*
26: *Thread.cpp:342*
27: *ConstantPool.h:130*
28: *ClassFile.h:92*
29: *ClassFile.h:231*
30: *Thread.cpp:408*
31: *ClassFile.h:56*
32: *Thread.cpp:412*
33: *AttributeInfo.h:55*
34: *ClassFile.cpp:230*
35: *Thread.cpp:66*
36: *Thread.cpp:415*

```
422:                // If yes, the thread will try to acquire the monitor associated with the instance
423:                // the method is called upon.
424:                // In case the monitor is already acquired by another thread, the current one will be
425:                // set as Waiting; once it will be chosen by the monitor to run again the following
426:                // instruction to execute will be the first instruction of the new method
427:                if (mi[1]->access_flags[2] & ACC_SYNCHRONIZED[3])
428:                {
429:                        id[4]->lock[5].acquire[6](this);
430:                        current_stack_frame[7]->monitor_to_release[8] = &id[4]->lock[5];
431:                }
432:                else
433:                        current_stack_frame[7]->monitor_to_release[8] = NULL;
434:
435:                // We are completely unaware at this moment whether the thread succeeded to
436:                // acquire the monitor and continue running or it has failed and put asleep
437:
438:                // As the last step of the invokation process check whether the method to be invoked
439:                // is native. If yes, take special care of its invokation.
440:                // Note, that in the case the method is native there will be no "return" instruction
441:                // at the end of it, that's why we have to deal with the return functionality after
442:                // the native method returns
443:                if (mi[1]->access_flags[2] & ACC_NATIVE[9])
444:                {
445:                        Result[10] result = invoke_native_method[11](mi[1],id[4]);
446:                        if (result[12] != Success[13])
447:                        {
448:                                print_error[14](result[12]);
449:                                return ExecutionCannotExecuteNativeMethod[15];
450:                        }
451:                }
452:
453:                return Success[13];
454:        }
455:
456:        // Is called by invokestatic instruction
457:        Result[10] Thread[16]::invoke_static_method(CONSTANT_Methodref_info[17] * entry)
458:        {
459:                // Check whether this CP entry is already resolved; resolve if needed
460:                if (!entry[18]->is_resolved[19]())
461:                {
462:                        Result[10] result = entry[18]->resolve_nonvirtual[20]();
463:                        if (result[21] != Success[13])
464:                        {
465:                                print_error[14](result[21]);
466:                                return ThreadCannotInvokeStaticMethod[22];
467:                        }
468:                }
469:
470:                // Now we have to calculate the depth of the overlapped area between two stack frames
471:
472:                // Determine the size (in words) of all the arguments of this method
473:                unsigned int arg_size_words = entry[18]->get_arguments_size[23]();
474:
```

Footnotes:

1: *Thread.cpp:408*
2: *ClassFile.h:37*
3: *defs.h:375*
4: *Thread.cpp:388*
5: *ObjectData.h:77*
6: *ObjectData.cpp:447*
7: *Thread.h:148*
8: *Stack.h:160*
9: *defs.h:376*
10: *defs.h:29*
11: *Thread.cpp:737*
12: *Thread.cpp:445*
13: *defs.h:33*
14: *errors.cpp:35*
15: *defs.h:77*
16: *Thread.h:17*
17: *ConstantPool.h:98*
18: *Thread.cpp:457*
19: *ConstantPool.h:48*
20: *ConstantPool.cpp:361*
21: *Thread.cpp:462*
22: *defs.h:84*
23: *ConstantPool.cpp:437*

```
475:                    // This variable will remember how many words we have to pop from the calling method's
476:                    // frame upon return
477:                    current_stack_frame[1]->words_to_pop[2] = arg_size_words[3];
478:
479:                    // Determine the depth of the new stack frame (the address of the beginning of the
480:                    // new stack frame); no object reference this time
481:                    word[4] * new_stack_frame_start =
482:                            current_stack_frame[1]->operand_stack_start[5] + current_stack_frame[1]->operand_stack_top[6] - arg
        _size_words[3];
483:
484:                    // Push new stack frame onto the stack so that the operand stack of the
485:                    // calling method's frame will become the local variables storage
486:                    // of the frame of the method being called.
487:                    // This approach facilitates the process of passing parameters to the new method
488:                    stack_frame[7] * new_stack_frame =
489:                            stack[8]->push_overlapped_frame[9](new_stack_frame_start[10]);
490:
491:                    // Go get the code attribute of the method
492:                    method_info[11] * mi = entry[12]->minfo[13];
493:                    assert(mi[14] != NULL);
494:
495:                    Code_attribute[15] * new_code_attribute = mi[14]->get_code_attribute[16]();
496:                    // Note that the new_code_attribute can be NULL in case of a native method !
497:
498:                    // The current class file becomes the CLASS FILE OF THE METHOD being invoked
499:                    // and NOT the class file of the objectref it is invoked upon
500:                    ClassFile[17] * new_method_class_file = mi[14]->class_file[18];
501:                    assert(new_method_class_file[19] != NULL);
502:
503:                    // switch contexts between two frames (going up the stack)
504:                    context_switch_up[20](new_method_class_file[19], new_code_attribute[21], new_stack_frame[22]);
505:
506:                    // After everything is set up, we will check whether this method is synchronized.
507:                    // If yes, the thread will try to acquire the monitor associated with the class
508:                    // the method is called upon.
509:                    // In case the monitor is already acquired by another thread, the current one will be
510:                    // set as Waiting; once it will be chosen by the monitor to run again the following
511:                    // instruction to execute will be the first instruction of the new method
512:                    if (mi[14]->access_flags[23] & ACC_SYNCHRONIZED[24])
513:                    {
514:                            new_method_class_file[19]->class_data[25]->lock[26].acquire[27](this);
515:                            current_stack_frame[1]->monitor_to_release[28] = &new_method_class_file[19]->class_data[25]->lock[26];
516:                    }
517:                    else
518:                            current_stack_frame[1]->monitor_to_release[28] = NULL;
519:
520:                    // We are completely unaware at this moment whether the thread succeeded to
521:                    // acquire the monitor and continue running or it has failed and put asleep
522:
523:                    // As the last step of the invokation process check whether the method to be invoked
524:                    // is native. If yes, take special care of its invokation.
525:                    // Note, that in the case the method is native there will be no "return" instruction
526:                    // at the end of it, that's why we have to deal with the return functionality after
```

Footnotes:

1: *Thread.h:148*
2: *Stack.h:166*
3: *Thread.cpp:473*
4: *defs.h:195*
5: *Stack.h:146*
6: *Stack.h:149*
7: *Stack.h:129*
8: *Thread.h:42*
9: *Stack.cpp:45*
10: *Thread.cpp:481*
11: *ClassFile.h:92*
12: *Thread.cpp:457*
13: *ConstantPool.h:125*
14: *Thread.cpp:492*
15: *AttributeInfo.h:55*
16: *ClassFile.cpp:230*
17: *ClassFile.h:136*
18: *ClassFile.h:56*
19: *Thread.cpp:500*
20: *Thread.cpp:66*
21: *Thread.cpp:495*
22: *Thread.cpp:488*
23: *ClassFile.h:37*
24: *defs.h:375*
25: *ClassFile.h:227*
26: *ObjectData.h:77*
27: *ObjectData.cpp:447*
28: *Stack.h:160*

```
527:                    // the native method returns
528:                    if (mi[1]->access_flags[2] & ACC_NATIVE[3])
529:                    {
530:                            Result[4] result = invoke_native_method[5](mi[1]);
531:                            if (result[6] != Success[7])
532:                            {
533:                                    print_error[8](result[6]);
534:                                    return ExecutionCannotExecuteNativeMethod[9];
535:                            }
536:                    }
537:
538:                    return Success[7];
539:            }
540:
541:            // Is called by invokespecial instruction
542:            Result[4] Thread[10]::invoke_special_method(CONSTANT_Methodref_info[11] * entry)
543:            {
544:                    // Check whether this CP entry is already resolved; resolve if needed
545:                    if (!entry[12]->is_resolved[13]())
546:                    {
547:                            Result[4] result = entry[12]->resolve_nonvirtual[14]();
548:                            if (result[15] != Success[7])
549:                            {
550:                                    print_error[8](result[15]);
551:                                    return ThreadCannotInvokeSpecialMethod[16];
552:                            }
553:                    }
554:
555:                    // Now we have to calculate the depth of the overlapped area between two stack frames
556:
557:                    // Determine the size (in words) of all the arguments of this method
558:                    unsigned int arg_size_words = entry[12]->get_arguments_size[17]();
559:
560:                    // Determine the depth of the new stack frame (the address of the beginning of the
561:                    // new stack frame)
562:                    word[18] * new_stack_frame_start =
563:                            current_stack_frame[19]->operand_stack_start[20] + current_stack_frame[19]->operand_stack_top[21] - a
            rg_size_words[22];
564:                    // One more word down the stack to get the address of the object reference
565:                    new_stack_frame_start[23] -= 1;
566:
567:                    // This variable will remember how many words we have to pop from the calling method's
568:                    // frame upon return
569:                    current_stack_frame[19]->words_to_pop[24] = arg_size_words[22] + 1;
570:
571:                    // Push new stack frame onto the stack so that the operand stack of the
572:                    // calling method's frame will become the local variables storage
573:                    // of the frame of the method being called.
574:                    // This approach facilitates the process of passing parameters to the new method
575:                    stack_frame[25] * new_stack_frame =
576:                            stack[26]->push_overlapped_frame[27](new_stack_frame_start[23]);
577:
578:                    // After creating the new stack frame this way we can be sure that
```

Footnotes:

1: *Thread.cpp:492*
2: *ClassFile.h:37*
3: *defs.h:376*
4: *defs.h:29*
5: *Thread.cpp:737*
6: *Thread.cpp:530*
7: *defs.h:33*
8: *errors.cpp:35*
9: *defs.h:77*
10: *Thread.h:17*
11: *ConstantPool.h:98*
12: *Thread.cpp:542*
13: *ConstantPool.h:48*
14: *ConstantPool.cpp:361*
15: *Thread.cpp:547*
16: *defs.h:85*
17: *ConstantPool.cpp:437*
18: *defs.h:195*
19: *Thread.h:148*
20: *Stack.h:146*
21: *Stack.h:149*
22: *Thread.cpp:558*
23: *Thread.cpp:562*
24: *Stack.h:166*
25: *Stack.h:129*
26: *Thread.h:42*
27: *Stack.cpp:45*

```
579:                    // the reference to the object upon which the method should be called
580:                    // is located in the local variable 0 of the new stack frame
581:                    // (the following parameters are located in the local variables 1,2, and so on)
582:                    word[1] objectref = new_stack_frame[2]->get_reference[3](0);
583:
584:                    // Get the class of the object upon which the method must be called
585:                    InstanceData[4] * id = vm[5]->get_handle_pool[6]()->get_instance[7](objectref[8]);
586:                    if (id[9] == null[10])
587:                    {
588:                            // Get rid of the allocated stack frame
589:                            stack[11]->pop_frame[12]();
590:                            raise_exception[13](VM_ERROR_NullPointerException[14]);
591:                            return ThreadCannotInvokeInstanceMethod[15];
592:                    }
593:
594:                    // Go get the code attribute of the method
595:                    method_info[16] * mi = entry[17]->minfo[18];
596:                    assert(mi[19] != NULL);
597:
598:                    // Here is the main difference between invokevirtual and invokespecial instructions:
599:                    // We look into the statically binded method instead of the virtual method's table
600:                    Code_attribute[20] * new_code_attribute = mi[19]->get_code_attribute[21]();
601:                    // Note that the new_code_attribute can be NULL in case of a native method !
602:
603:                    ClassFile[22] * new_method_class_file = mi[19]->class_file[23];
604:                    assert(new_method_class_file[24] != NULL);
605:
606:                    // switch contexts between two frames (going up the stack)
607:                    context_switch_up[25](new_method_class_file[24], new_code_attribute[26], new_stack_frame[2]);
608:
609:                    // After everything is set up, we will check whether this method is synchronized.
610:                    // If yes, the thread will try to acquire the monitor associated with the instance
611:                    // the method is called upon.
612:                    // In case the monitor is already acquired by another thread, the current one will be
613:                    // set as Waiting; once it will be chosen by the monitor to run again the following
614:                    // instruction to execute will be the first instruction of the new method
615:                    if (mi[19]->access_flags[27] & ACC_SYNCHRONIZED[28])
616:                    {
617:                            id[9]->lock[29].acquire[30](this);
618:                            current_stack_frame[31]->monitor_to_release[32] = &id[9]->lock[29];
619:                    }
620:                    else
621:                            current_stack_frame[31]->monitor_to_release[32] = NULL;
622:
623:                    // We are completely unaware at this moment whether the thread succeeded to
624:                    // acquire the monitor and continue running or it has failed and put asleep
625:
626:                    // As the last step of the invokation process check whether the method to be invoked
627:                    // is native. If yes, take special care of its invokation.
628:                    // Note, that in the case the method is native there will be no "return" instruction
629:                    // at the end of it, that's why we have to deal with the return functionality after
630:                    // the native method returns
631:                    if (mi[19]->access_flags[27] & ACC_NATIVE[33])
```

Footnotes:

1: *defs.h:195*
2: *Thread.cpp:575*
3: *Stack.cpp:396*
4: *ObjectData.h:113*
5: *Thread.h:32*
6: *VirtualMachine.h:336*
7: *HandlePool.cpp:25*
8: *Thread.cpp:582*
9: *Thread.cpp:585*
10: *defs.h:201*
11: *Thread.h:42*
12: *Stack.cpp:54*
13: *Thread.cpp:776*
14: *defs.h:105*
15: *defs.h:83*
16: *ClassFile.h:92*
17: *Thread.cpp:542*
18: *ConstantPool.h:125*
19: *Thread.cpp:595*
20: *AttributeInfo.h:55*
21: *ClassFile.cpp:230*
22: *ClassFile.h:136*
23: *ClassFile.h:56*
24: *Thread.cpp:603*
25: *Thread.cpp:66*
26: *Thread.cpp:600*
27: *ClassFile.h:37*
28: *defs.h:375*
29: *ObjectData.h:77*
30: *ObjectData.cpp:447*
31: *Thread.h:148*
32: *Stack.h:160*
33: *defs.h:376*

```
632:                    {
633:                            Result[1] result = invoke_native_method[2](mi[3],id[4]);
634:                            if (result[5] != Success[6])
635:                            {
636:                                    print_error[7](result[5]);
637:                                    return ExecutionCannotExecuteNativeMethod[8];
638:                            }
639:                    }
640:
641:                    return Success[6];
642:            }
643:
644:            Result[1] Thread[9]::invoke_interface_method(CONSTANT_InterfaceMethodref_info[10] * entry, u1[11] args_count)
645:            {
646:                    int arg_size_words = args_count[12]-1;
647:                    // Determine the depth of the new stack frame (the address of the beginning of the
648:                    // new stack frame)
649:                    word[13] * new_stack_frame_start =
650:                            current_stack_frame[14]->operand_stack_start[15] + current_stack_frame[14]->operand_stack_top[16] - a
            rg_size_words[17];
651:                    // One more word down the stack to get the address of the object reference
652:                    new_stack_frame_start[18] -= 1;
653:
654:                    // This variable will remember how many words we have to pop from the calling method's
655:                    // frame upon return
656:                    current_stack_frame[14]->words_to_pop[19] = arg_size_words[17] + 1;
657:
658:                    // Push new stack frame onto the stack so that the operand stack of the
659:                    // calling method's frame will become the local variables storage
660:                    // of the frame of the method being called.
661:                    // This approach facilitates the process of passing parameters to the new method
662:                    stack_frame[20] * new_stack_frame =
663:                            stack[21]->push_overlapped_frame[22](new_stack_frame_start[18]);
664:
665:                    // After creating the new stack frame this way we can be sure that
666:                    // the reference to the object upon which the method should be called
667:                    // is located in the local variable 0 of the new stack frame
668:                    // (the following parameters are located in the local variables 1,2, and so on)
669:                    word[13] objectref = new_stack_frame[23]->get_reference[24](0);
670:
671:                    // Get the class of the object upon which the method must be called
672:                    InstanceData[25] * id = vm[26]->get_handle_pool[27]()->get_instance[28](objectref[29]);
673:                    if (id[30] == null[31])
674:                    {
675:                            // Get rid of the allocated stack frame
676:                            stack[21]->pop_frame[32]();
677:                            raise_exception[33](VM_ERROR_NullPointerException[34]);
678:                            return ThreadCannotInvokeInterfaceMethod[35];
679:                    }
680:
681:                    // Now we can resolve the method based on the instance information
682:
683:                    method_info[36] * mi;
```

Footnotes:

1: *defs.h:29*
2: *Thread.cpp:737*
3: *Thread.cpp:595*
4: *Thread.cpp:585*
5: *Thread.cpp:633*
6: *defs.h:33*
7: *errors.cpp:35*
8: *defs.h:77*
9: *Thread.h:17*
10: *ConstantPool.h:156*
11: *defs.h:168*
12: *Thread.cpp:644*
13: *defs.h:195*
14: *Thread.h:148*
15: *Stack.h:146*
16: *Stack.h:149*
17: *Thread.cpp:646*
18: *Thread.cpp:649*
19: *Stack.h:166*
20: *Stack.h:129*
21: *Thread.h:42*
22: *Stack.cpp:45*
23: *Thread.cpp:662*
24: *Stack.cpp:396*
25: *ObjectData.h:113*
26: *Thread.h:32*
27: *VirtualMachine.h:336*
28: *HandlePool.cpp:25*
29: *Thread.cpp:669*
30: *Thread.cpp:672*
31: *defs.h:201*
32: *Stack.cpp:54*
33: *Thread.cpp:776*
34: *defs.h:105*
35: *defs.h:86*
36: *ClassFile.h:92*

```
684:                    // We have to resolve the entry each time it is used
685:                    Result[1] result = entry[2]->resolve_on[3](id[4], mi[5]);
686:                    if (result[6] != Success[7])
687:                    {
688:                            print_error[8](result[6]);
689:                            return ThreadCannotInvokeInterfaceMethod[9];
690:                    }
691:                    assert(mi[5] != NULL);
692:
693:                    Code_attribute[10] * new_code_attribute = mi[5]->get_code_attribute[11]();
694:                    // Note that the new_code_attribute can be NULL in case of a native method !
695:
696:                    ClassFile[12] * new_method_class_file = mi[5]->class_file[13];
697:                    assert(new_method_class_file[14] != NULL);
698:
699:                    // switch contexts between two frames (going up the stack)
700:                    context_switch_up[15](new_method_class_file[14], new_code_attribute[16], new_stack_frame[17]);
701:
702:                    // After everything is set up, we will check whether this method is synchronized.
703:                    // If yes, the thread will try to acquire the monitor associated with the instance
704:                    // the method is called upon.
705:                    // In case the monitor is already acquired by another thread, the current one will be
706:                    // set as Waiting; once it will be chosen by the monitor to run again the following
707:                    // instruction to execute will be the first instruction of the new method
708:                    if (mi[5]->access_flags[18] & ACC_SYNCHRONIZED[19])
709:                    {
710:                            id[4]->lock[20].acquire[21](this);
711:                            current_stack_frame[22]->monitor_to_release[23] = &id[4]->lock[20];
712:                    }
713:                    else
714:                            current_stack_frame[22]->monitor_to_release[23] = NULL;
715:
716:                    // We are completely unaware at this moment whether the thread succeeded to
717:                    // acquire the monitor and continue running or it has failed and put asleep
718:
719:                    // As the last step of the invokation process check whether the method to be invoked
720:                    // is native. If yes, take special care of its invokation.
721:                    // Note, that in the case the method is native there will be no "return" instruction
722:                    // at the end of it, that's why we have to deal with the return functionality after
723:                    // the native method returns
724:                    if (mi[5]->access_flags[18] & ACC_NATIVE[24])
725:                    {
726:                            Result[1] result = invoke_native_method[25](mi[5],id[4]);
727:                            if (result[26] != Success[7])
728:                            {
729:                                    print_error[8](result[26]);
730:                                    return ExecutionCannotExecuteNativeMethod[27];
731:                            }
732:                    }
733:
734:                    return Success[7];
735:            }
736:
```

Footnotes:

1: *defs.h:29*
2: *Thread.cpp:644*
3: *ConstantPool.cpp:531*
4: *Thread.cpp:672*
5: *Thread.cpp:683*
6: *Thread.cpp:685*
7: *defs.h:33*
8: *errors.cpp:35*
9: *defs.h:86*
10: *AttributeInfo.h:55*
11: *ClassFile.cpp:230*
12: *ClassFile.h:136*
13: *ClassFile.h:56*
14: *Thread.cpp:696*
15: *Thread.cpp:66*
16: *Thread.cpp:693*
17: *Thread.cpp:662*
18: *ClassFile.h:37*
19: *defs.h:375*
20: *ObjectData.h:77*
21: *ObjectData.cpp:447*
22: *Thread.h:148*
23: *Stack.h:160*
24: *defs.h:376*
25: *Thread.cpp:737*
26: *Thread.cpp:726*
27: *defs.h:77*

```
737:        Result[1] Thread[2]::invoke_native_method(method_info[3] * mi, InstanceData[4] * id)
738:        {
739:                // mark thread as runnning native code
740:                set_native[5]();
741:
742:                // Go into the "real world"
743:                Result[1] result = vm[6]->get_native_handler[7]()->run_method[8](this, mi[9], id[10]);
744:                if (result[11] != Success[12])
745:                {
746:                        print_error[13](result[11]);
747:                        return ExecutionCannotExecuteNativeMethod[14];
748:                }
749:
750:                // mark thread as running Java code if its state has not become waiting
751:                // (for example as a result of the "wait" function call)
752:                if (is_native[15]())
753:                        set_running[16]();
754:
755:                // Surprisingly, all the rest is done be the NativeHandler itself
756:                // including the return from the method, pushing result onto the stack,
757:                // releasing monitor etc.
758:
759:                return Success[12];
760:        }
761:
762:        // This function is used when the class of the exception is user defined
763:        // or not known from the exceptions array;
764:        // Called from "athrow" instruction - that is, it is user initiated exception
765:        void Thread[2]::raise_exception(ClassFile[17] * exception_cf)
766:        {
767:                // Create a new instance of the raised exception in order to put its reference
768:                // on the stack frame later
769:                word[18] ref = exception_cf[19]->create_new_instance[20]();
770:                raised_exception_instance[21] = vm[6]->get_handle_pool[22]()->get_instance[23](ref[24]);
771:        }
772:
773:        // This function is used for the predefined exceptions;
774:        // It is called from all the instructions where an exceptional situation occur;
775:        // That is, it is a JVM-initiated exception
776:        void Thread[2]::raise_exception(Result[1] exception_code)
777:        {
778:                // The exception flag must be the proper JVM exception, not an internal message
779:                assert((exception_code[25] > VM_ERROR_START[26]) && (exception_code[25] < VM_ERROR_END[27]));
780:                // Display a message
781:                print_error[13](exception_code[25]);
782:
783:                // We will create the class file of the raised exception given its code
784:                int exception_index = exception_code[25] - VM_ERROR_START[26] - 1;
785:                wchar_t * exception_class_name = ExceptionClasses[28][exception_index[29]].exception_class_name[30];
786:
787:                // If the class of the raised exception is not loaded yet - load it
788:                ClassFile[17] * exception_cf =
                                vm[6]->get_bootstrap_class_loader[31]()->get_class[32](exception_class_name[33], wcslen(exce
```

Footnotes:

1: *defs.h:29*
2: *Thread.h:17*
3: *ClassFile.h:92*
4: *ObjectData.h:113*
5: *Thread.h:168*
6: *Thread.h:32*
7: *VirtualMachine.h:337*
8: *NativeHandler.cpp:189*
9: *Thread.cpp:737*
10: *Thread.cpp:737*
11: *Thread.cpp:743*
12: *defs.h:33*
13: *errors.cpp:35*
14: *defs.h:77*
15: *Thread.h:175*
16: *Thread.h:166*
17: *ClassFile.h:136*
18: *defs.h:195*
19: *Thread.cpp:765*
20: *ClassFile.cpp:870*
21: *Thread.h:134*
22: *VirtualMachine.h:336*
23: *HandlePool.cpp:25*
24: *Thread.cpp:769*
25: *Thread.cpp:776*
26: *defs.h:91*
27: *defs.h:116*
28: *errors.cpp:7*
29: *Thread.cpp:784*
30: *defs.h:152*
31: *VirtualMachine.h:334*
32: *ClassLoader.cpp:187*
33: *Thread.cpp:785*

```
        ption_class_name[1]));
790:            if (! exception_cf[2])
791:            {
792:                    exception_cf[2] =
793:                            vm[3]->get_bootstrap_class_loader[4]()->load_class[5](exception_class_name[1], wcslen(excep
        tion_class_name[1]));
794:                    if (! exception_cf[2])
795:                    {
796:                            raised_exception_instance[6] = NULL;
797:                            return;
798:                    }
799:            }
800:
801:            // Create a new instance of the raised exception in order to put its reference
802:            // on the stack frame later
803:            raise_exception[7](exception_cf[2]);
804:    }
805:
806:    // This function is called when the exception handler is defined in the body
807:    // of the same method where an exception is thrown
808:    void Thread[8]::handle_exception_locally(word[9] handler_pc)
809:    {
810:            assert(raised_exception_instance[6] != NULL);
811:
812:            // Push the instance of the raised exception onto the stack
813:            current_stack_frame[10]->push_reference[11](raised_exception_instance[6]->get_reference[12]());
814:
815:            // Clean up the exception and activate the exception handler
816:            raised_exception_instance[6] = NULL;
817:            pc_register[13] = handler_pc[14];
818:
819:            // Now the program counter is set to the exception handler code,
820:            // so the next instruction executed by this thread will be the first
821:            // instruction of the exception handler code
822:    }
823:
824:    Result[15] Thread[8]::treat_exception()
825:    {
826:            assert(raised_exception_instance[6] != NULL);
827:
828:            // Iterate through the current method's exception table looking for the appropriate
829:            // entry corresponding to the raised exception
830:            for (int i = 0; i < current_code_attribute[16]->exception_table_length[17]; i++)
831:            {
832:                    Code_attribute[18]::exception_table_entry[19] entry = current_code_attribute[16]->exception_table[20]
        [i[21]];
833:
834:                    // Check whether the raised exception is within the range declared in the entry
835:                    // (Note that "end_pc" is always one more than the exception scope)
836:                    if (current_stack_frame[10]->instruction_pc_register[22] >= entry[23].start_pc[24] && current_stack_f
        rame[10]->instruction_pc_register[22] < entry[23].end_pc[25])
837:                    {
880:                            // Note: in case of the "finally" clause without any declared "catch" block
```

Footnotes:

1: *Thread.cpp:785*
2: *Thread.cpp:788*
3: *Thread.h:32*
4: *VirtualMachine.h:334*
5: *ClassLoader.cpp:18*
6: *Thread.h:134*
7: *Thread.cpp:776*
8: *Thread.h:17*
9: *defs.h:195*
10: *Thread.h:148*
11: *Stack.cpp:187*
12: *ObjectData.h:142*
13: *Thread.h:159*
14: *Thread.cpp:808*
15: *defs.h:29*
16: *Thread.h:155*
17: *AttributeInfo.h:69*
18: *AttributeInfo.h:55*
19: *AttributeInfo.h:57*
20: *AttributeInfo.h:70*
21: *Thread.cpp:830*
22: *Stack.h:178*
23: *Thread.cpp:832*
24: *AttributeInfo.h:59*
25: *AttributeInfo.h:60*

```
839:                                    // Java compiler will insert the "catch" block that will just visit the "finally"
840:                                    // block and rethrow the same exception. In this case the exception table
841:                                    // entry will have a type of the exception equal to 0, which should mean -
842:                                    // catch everything
843:                                    if (entry[1].catch_type[2] == 0)
844:                                    {
845:                                            handle_exception_locally[3](entry[1].handler_pc[4]);
846:                                            return Success[5];
847:                                    }
848:
849:                                    // Get the class file of the exception in the corresponding exception table entry
850:                                    CONSTANT_Class_info[6] * exception_class_info_entry =
851:                                            (CONSTANT_Class_info[6]*)current_cp[7]->get_item_at_index[8](entry[1].catch_type[2]);
852:                                    if (!exception_class_info_entry[9])
853:                                            return ExecutionWrongConstantPoolEntryIndex[10];
854:
855:                                    // Check whether this entry is already resolved; if not - resolve it
856:                                    if (!exception_class_info_entry[9]->is_resolved[11]())
857:                                    {
858:                                            Result[12] result = exception_class_info_entry[9]->resolve[13]();
859:                                            if (result[14] != Success[5])
860:                                                    return ExecutionCannotResolveExceptionClass[15];
861:                                    }
862:
863:                                    // The last check is whether the actual thrown exception class is equal
864:                                    // or is a subclass of the class declared in the exception table
865:                                    ClassFile[16] * raised_exception_cf = raised_exception_instance[17]->class_file[18];
866:                                    if (raised_exception_cf[19]->inherits[20](exception_class_info_entry[9]->resolved_class_f
        ile[21]))
867:                                    {
868:                                            handle_exception_locally[3](entry[1].handler_pc[4]);
869:                                            return Success[5];
870:                                    }
871:
872:                            } // if the pc is in the current entry's range
873:                    } // for all the exception table entries
874:
875:                    // At this point we know that no exception handler is specified for the exception
876:                    // raised in the body of this method.
877:                    // We have to pop the current stack frame and return to the method that called
878:                    // this method. But instead of continuing execution normally in the previous method
879:                    // we will throw the same exception again which will cause the thread to go through
880:                    // the same process of exception treatment;
881:                    // Note: throwing the exception is simply done by the fact that
882:                    // the "raised_exception_instance" is not cleaned up
883:                    Result[12] result = return_from_method[22]();
884:                    if (result[23] != Success[5])
885:                    {
886:                            print_error[24](result[23]);
887:                            return Failure[25];
888:                    }
889:
890:                    return Success[5];
```

Footnotes:

1: *Thread.cpp:832*
2: *AttributeInfo.h:62*
3: *Thread.cpp:808*
4: *AttributeInfo.h:61*
5: *defs.h:33*
6: *ConstantPool.h:58*
7: *Thread.h:150*
8: *ConstantPool.h:299*
9: *Thread.cpp:850*
10: *defs.h:80*
11: *ConstantPool.h:48*
12: *defs.h:29*
13: *ConstantPool.cpp:22*
14: *Thread.cpp:858*
15: *defs.h:81*
16: *ClassFile.h:136*
17: *Thread.h:134*
18: *ObjectData.h:74*
19: *Thread.cpp:865*
20: *ClassFile.cpp:1061*
21: *ConstantPool.h:71*
22: *Thread.cpp:142*
23: *Thread.cpp:883*
24: *errors.cpp:35*
25: *defs.h:34*

```
891:        }
892:
893:        // A single step of execution:
894:        // --------------------------
895:        // 1)  notify JVM about the new current thread
896:        // 2a) check whether the previous method has thrown any exceptions
897:        // 2b) if yes, try to catch or rethrow it further
898:        // 3)  clean up the raised exception
899:        // 4)  save the program counter of the instruction that is going to be executed
900:        // 5)  fetch the opcode
901:        // 6)  increment the program counter
902:        // 7)  execute the instruction
903:        // 8)  check whether an internal error has occured
904:        // 9a) check whether an exception has been raised
905:        // 9b) if yes, try to catch or rethrow it further
906:        // ---------------------------
907:        Result[1] Thread[2]::step()
908:        {
909:                // Let JVM know that myself is the currently executing thread
910:                vm[3]->current_thread[4] = this;
911:
912:                // Check the raised exception BEFORE the instruction execution;
913:                // This is the case when we have abruptly returned from the previous method
914:                // due to the exception thrown but not handled in it
915:                if (raised_exception_instance[5])
916:                {
917:                        // If an exception was raised during the execution of the previous method,
918:                        // try to catch or rethrow it
919:                        Result[1] result = treat_exception[6]();
920:                        if (result[7] != Success[8])
921:                        {
922:                                print_error[9](result[7]);
923:                                return ExecutionCannotExecuteInstruction[10];
924:                        }
925:
926:                        return Success[8];
927:                }
928:
929:        #ifdef DEBUG_STACK_SNAPSHOT
930:                stack[11]->debug_print[12](debug_file);
931:        #endif // DEBUG_STACK_SNAPSHOT
932:
933:                // Save the program counter of the instruction that is going to be executed
934:                // in the current stack frame (it may be needed for the exception check)
935:                current_stack_frame[13]->instruction_pc_register[14] = pc_register[15];
936:
937:                // Fetch the instruction and increment the PC
938:                u1[16] opcode = current_code[17][pc_register[15]++];
939:
940:                // Clean up the exception
941:                raised_exception_instance[5] = NULL;
942:
943:        #ifdef DEBUG_EXECUTION[18]
```

Footnotes:

1: *defs.h:29*
2: *Thread.h:17*
3: *Thread.h:32*
4: *VirtualMachine.h:287*
5: *Thread.h:134*
6: *Thread.cpp:824*
7: *Thread.cpp:919*
8: *defs.h:33*
9: *errors.cpp:35*
10: *defs.h:75*
11: *Thread.h:42*
12: *Stack.cpp:75*
13: *Thread.h:148*
14: *Stack.h:178*
15: *Thread.h:159*
16: *defs.h:168*
17: *Thread.h:151*
18: *defs.h:428*

```
944:                debug_print_instruction[1](opcode[2],debug_file);
945:                void * tmp = opcodes[3][opcode[2]].mnemonic[4];
946:        #endif // DEBUG_EXECUTION
947:
948:                // Execute the instruction
949:                InstructionResult[5] result = (*opcodes[3][opcode[2]].exec_function[6])(this);
950:
951:                // Check the result of the instruction internally
952:                if (result[7] != InstructionSuccess[8])
953:                {
954:                        // Dispaly the internal message
955:                        print_instruction_error[9](result[7]);
956:                        //return ExecutionCannotExecuteInstruction;
957:                }
958:
959:                // Check the raised exception AFTER the instruction execution
960:                if (raised_exception_instance[10])
961:                {
962:                        // If an exception was raised during the execution of this instruction,
963:                        // try to catch or rethrow it
964:                        Result[11] result = treat_exception[12]();
965:                        if (result[13] != Success[14])
966:                        {
967:                                print_error[15](result[13]);
968:                                return ExecutionCannotExecuteInstruction[16];
969:                        }
970:                }
971:
972:                return Success[14];
973:        }
974:
975:        // Is called from the java.lang.Object's native implementation of "wait"
976:        Result[11] Thread[17]::wait(InstanceData[18] * id)
977:        {
978:
979:                debug_file << "Thread " << this->get_id[19]() << " issuing WAIT on monitor " << &id[20]->lock[21] << " of
        " << (int)id[20]->get_reference[22]() << endl;
980:
981:                // Make sure that this thread owns the object's monitor
982:                if (id[20]->lock[21].owner[23] != this)
983:                {
984:                        raise_exception[24](VM_ERROR_IllegalMonitorStateException[25]);
985:                        return Failure[26];
986:                }
987:
988:                // Insert the current thread to this object's waiting set
989:                id[20]->waiting_set[27].add_element[28](this);
990:                // Perform N unlock operations on the object to relinquish the lock on it;
991:                // (first, remember the number N in the thread's memory)
992:                lock_counter[29] = id[20]->lock[21].counter[30];
993:
994:
995:        debug_file << "LOCK_COUNTER to remember = " << lock_counter[29] << endl;
```

Footnotes:

1: *Thread.cpp:1085*
2: *Thread.cpp:938*
3: *opcodes.cpp:3215*
4: *opcodes.h:256*
5: *defs.h:120*
6: *opcodes.h:259*
7: *Thread.cpp:949*
8: *defs.h:122*
9: *errors.cpp:84*
10: *Thread.h:134*
11: *defs.h:29*
12: *Thread.cpp:824*
13: *Thread.cpp:964*
14: *defs.h:33*
15: *errors.cpp:35*
16: *defs.h:75*
17: *Thread.h:17*
18: *ObjectData.h:113*
19: *Thread.h:82*
20: *Thread.cpp:976*
21: *ObjectData.h:77*
22: *ObjectData.h:142*
23: *ObjectData.h:17*
24: *Thread.cpp:776*
25: *defs.h:106*
26: *defs.h:34*
27: *ObjectData.h:80*
28: *vector.h:40*
29: *Thread.h:145*
30: *ObjectData.h:19*

```
996:
997:                    id[1]->lock[2].counter[3] = 0;
998:                    id[1]->lock[2].owner[4] = NULL;
999:
1000:                   // We cannot use the monitor's release function here because it will
1001:                   // unlock only one of the locks made by this thread-owner
1002:
1003:                   // Release next thread that is waiting for this monitor if any
1004:                   // (Note, that the waiting set here is the monitor's waiting set,
1005:                   // not the object's waiting set !)
1006:                   if (id[1]->lock[2].waiting_set[5].size[6]() > 0)
1007:                   {
1008:                           Thread[7] * candidate = id[1]->lock[2].waiting_set[5].first_element[8]();
1009:                           assert(candidate[9] != NULL);
1010:                           id[1]->lock[2].waiting_set[5].remove_element_at[10](0);
1011:
1012:                           // make the winner thread to be the new owner of the monitor
1013:                           id[1]->lock[2].counter[3] = 1;
1014:                           id[1]->lock[2].owner[4] = candidate[9];
1015:
1016:                           // mark this thread suitable for execution
1017:                           candidate[9]->set_running[11]();
1018:                   }
1019:
1020:                   // Put the thread asleep
1021:                   set_waiting[12]();
1022:
1023:                   return Success[13];
1024:           }
1025:
1026:           // Is called from the java.lang.Object's native implementation of "notify"
1027:           Result[14] Thread[7]::notify(InstanceData[15] * id)
1028:           {
1029:
1030:                   debug_file << "Thread " << this->get_id[16]() << " issuing NOTIFY on monitor of " << (int)id[17]->get_r
       eference[18]() << endl;
1031:
1032:                   // Make sure that this thread owns the object's monitor
1033:                   if (id[17]->lock[2].owner[4] != this)
1034:                   {
1035:                           raise_exception[19](VM_ERROR_IllegalMonitorStateException[20]);
1036:                           return Failure[21];
1037:                   }
1038:
1039:                   // Pick up any thread that is in the waiting set of this object
1040:                   // (Note, that the waiting set of the object is used here,
1041:                   // not the the object monitor's waiting set)
1042:                   if (id[17]->waiting_set[22].size[6]() > 0)
1043:                   {
1044:                           Thread[7] * candidate = id[17]->waiting_set[22].first_element[8]();
1045:                           assert(candidate[23] != NULL);
1046:                           id[17]->waiting_set[22].remove_element_at[10](0);
1047:
```

Footnotes:

1: *Thread.cpp:976*
2: *ObjectData.h:77*
3: *ObjectData.h:19*
4: *ObjectData.h:17*
5: *ObjectData.h:21*
6: *vector.h:23*
7: *Thread.h:17*
8: *vector.h:70*
9: *Thread.cpp:1008*
10: *vector.h:49*
11: *Thread.h:166*
12: *Thread.h:167*
13: *defs.h:33*
14: *defs.h:29*
15: *ObjectData.h:113*
16: *Thread.h:82*
17: *Thread.cpp:1027*
18: *ObjectData.h:142*
19: *Thread.cpp:776*
20: *defs.h:106*
21: *defs.h:34*
22: *ObjectData.h:80*
23: *Thread.cpp:1044*

```
1048:                           // Now the running candidate should compete with other threads to
1049:                           // acquire the lock again
1050:                           id[1]->lock[2].acquire[3](candidate[4]);
1051:                   }
1052:
1053:                   return Success[5];
1054:           }
1055:
1056:           // Is called from the java.lang.Object's native implementation of "notifyAll"
1057:           Result[6] Thread[7]::notifyAll(InstanceData[8] * id)
1058:           {
1059:                   // Make sure that this thread owns the object's monitor
1060:                   if (id[9]->lock[2].owner[10] != this)
1061:                   {
1062:                           raise_exception[11](VM_ERROR_IllegalMonitorStateException[12]);
1063:                           return Failure[13];
1064:                   }
1065:
1066:                   // Release all threads that are in the waiting set of this object
1067:                   // (Note, that the waiting set of the object is used here,
1068:                   // not the the object monitor's waiting set)
1069:                   for(int i = 0; i < id[9]->waiting_set[14].size[15](); i++)
1070:                   {
1071:                           Thread[7] * candidate = id[9]->waiting_set[14].element_at[16](i[17]);
1072:                           assert(candidate[18] != NULL);
1073:
1074:                           // Now the running candidate should compete with the threads to
1075:                           // acquire the lock again
1076:                           id[9]->lock[2].acquire[3](candidate[18]);
1077:                   }
1078:
1079:                   // Clean up the waiting set
1080:                   id[9]->waiting_set[14].remove_all_elements[19]();
1081:
1082:                   return Success[5];
1083:           }
1084:
1085:           void Thread[7]::debug_print_instruction(u1[20] opcode, ostream & out)
1086:           {
1087:                   //      if (!debug)
1088:                   //              return;
1089:
1090:                           out[21] << "[" << id[22] << "] pc="; out[21].operator<<((unsigned int)pc_register[23]-1);
1091:                           out[21] << " " << opcodes[24][opcode[25]].mnemonic[26];
1092:                           wchar_t * class_name;
1093:                           u2[27] class_name_length;
1094:                           current_class_file[28]->get_full_class_name[29](class_name[30], class_name_length[31]);
1095:                           out[21] << " class: ";
1096:                           print_wchar[32](class_name[30], class_name_length[31], out[21]);
1097:                           delete [] class_name[30];
1098:
1099:                           method_info[33] * minfo = (method_info[33]*)current_code_attribute[34]->field_method_parent[35];
1100:                           wchar_t * mname, * dname;
```

Footnotes:

1: *Thread.cpp:1027*
2: *ObjectData.h:77*
3: *ObjectData.cpp:447*
4: *Thread.cpp:1044*
5: *defs.h:33*
6: *defs.h:29*
7: *Thread.h:17*
8: *ObjectData.h:113*
9: *Thread.cpp:1057*
10: *ObjectData.h:17*
11: *Thread.cpp:776*
12: *defs.h:106*
13: *defs.h:34*
14: *ObjectData.h:80*
15: *vector.h:23*
16: *vector.h:36*
17: *Thread.cpp:1069*
18: *Thread.cpp:1071*
19: *vector.h:68*
20: *defs.h:168*
21: *Thread.cpp:1085*
22: *Thread.h:35*
23: *Thread.h:159*
24: *opcodes.cpp:3215*
25: *Thread.cpp:1085*
26: *opcodes.h:256*
27: *defs.h:172*
28: *Thread.h:149*
29: *ClassFile.cpp:442*
30: *Thread.cpp:1092*
31: *Thread.cpp:1093*
32: *util.cpp:42*
33: *ClassFile.h:92*
34: *Thread.h:155*
35: *AttributeInfo.h:44*

```
1101:                           u2[1] mname_length, dname_length;
1102:                           minfo[2]->get_method_name[3](mname[4], mname_length[5]);
1103:                           minfo[2]->get_method_descriptor[6](dname[7], dname_length[8]);
1104:                           out[9] << " method: ";
1105:                           print_wchar[10](mname[4], mname_length[5], out[9]);
1106:                           out[9] << "[";
1107:                           print_wchar[10](dname[7], dname_length[8], out[9]);
1108:                           out[9] << "]" << endl;
1109:
1110:                           delete [] mname[4];
1111:                           delete [] dname[7];
1112:           }
1113:
1114:
```

Footnotes:

1: *defs.h:172*
2: *Thread.cpp:1099*
3: *ClassFile.cpp:168*
4: *Thread.cpp:1100*
5: *Thread.cpp:1101*
6: *ClassFile.cpp:177*
7: *Thread.cpp:1100*
8: *Thread.cpp:1101*
9: *Thread.cpp:1085*
10: *util.cpp:42*

```
 1:
 2:          #include <iostream.h>
 3:          #include <stdio.h>
 4:
 5:          #include "VirtualMachine.h"
 6:          #include "HeapManager.h"
 7:          #include "GarbageCollector.h"
 8:          #include "ClassLoader.h"
 9:          #include "ClassFile.h"
10:          #include "Thread.h"
11:          #include "ObjectData.h"
12:          #include "NativeHandler.h"
13:          #include "HandlePool.h"
14:
15:          /*
16:          * The design is such that all the objects are created in the VM c'tor
17:          * given a pointer to VM, but are initialized each one in its own time
18:          * by the init() function.
19:          */
20:
21:          VirtualMachine[1]::VirtualMachine()
22:          {
23:                  p_ConstantManager[2] = new ConstantManager[3](this);
24:                  heap_manager[4] = new HeapManager[5](this);
25:                  stack_heap_manager[6] = new HeapManager[5](this);
26:
27:                  class_loaders[7] = new Vector[8]<ClassLoader*>;
28:
29:                  // Passing NULL as a parent class loader means this will be
30:                  // the bootstrap class loader
31:                  bootstrap_class_loader[9] = new ClassLoader[10](NULL, this);
32:                  class_loaders[7]->add_element[11](bootstrap_class_loader[9]);
33:
34:                  execution_pool[12] = new ExecutionPool[13](this);
35:
36:                  // Create garbage collector thread;
37:                  // Initially it will run at the lowest priority.
38:                  // Garbage collector's id will be always 1
39:                  garbage_collector_thread[14] = new GarbageCollector[15](this, execution_pool[12]->get_next_thread_id[16]());
40:
41:                  handle_pool[17] = new HandlePool[18](this);
42:
43:                  native_handler[19] = new NativeHandler[20](this);
44:
45:                  primary_thread[21] = current_thread[22] = NULL;
46:
47:                  boot_process[23] = 1;
48:          }
49:
50:          /*
51:          * The design is such that all the objects are destroyed in the VM d'tor
52:          * but the real finalization is done in the previous call to each one's
53:          * destroy() function.
```

Footnotes:

1: *VirtualMachine.h:255*
2: *VirtualMachine.h:258*
3: *VirtualMachine.h:403*
4: *VirtualMachine.h:263*
5: *HeapManager.h:100*
6: *VirtualMachine.h:267*
7: *VirtualMachine.h:298*
8: *vector.h:17*
9: *VirtualMachine.h:281*
10: *ClassLoader.h:42*
11: *vector.h:40*
12: *VirtualMachine.h:291*
13: *VirtualMachine.cpp:319*
14: *VirtualMachine.h:278*
15: *GarbageCollector.h:36*
16: *VirtualMachine.h:389*
17: *VirtualMachine.h:295*
18: *HandlePool.h:42*
19: *VirtualMachine.h:270*
20: *NativeHandler.cpp:96*
21: *VirtualMachine.h:284*
22: *VirtualMachine.h:287*
23: *VirtualMachine.h:302*

```
54:         */
55:
56:         VirtualMachine[1]::~VirtualMachine()
57:         {
58:                 delete p_ConstantManager[2];
59:                 delete heap_manager[3];
60:                 delete stack_heap_manager[4];
61:                 delete garbage_collector_thread[5];
62:                 delete bootstrap_class_loader[6];
63:         //      delete primary_thread;
64:                 delete execution_pool[7];
65:                 delete handle_pool[8];
66:                 delete native_handler[9];
67:                 delete class_loaders[10];
68:         }
69:
70:         Result[11] VirtualMachine[1]::init()
71:         {
72:                 cout << "JVM started..." << endl;
73:
74:                 // All the initialization stuff
75:
76:                 // Initialize memory manager for the heap
77:                 Result[11] result = heap_manager[3]->init[12](MAX_HEAP_SIZE[13]); // 1Mb
78:                 if (result[14] != Success[15])
79:                 {
80:                         print_error[16](result[14]);
81:                         return InitializationCannotAllocateHeap[17];
82:                 }
83:
84:                 // Initialize memory manager for the stack
85:                 result[14] = stack_heap_manager[4]->init[12](MAX_HEAP_SIZE[13]); // 1Mb
86:                 if (result[14] != Success[15])
87:                 {
88:                         print_error[16](result[14]);
89:                         return InitializationCannotAllocateStack[18];
90:                 }
91:
92:                 // Register all known native methods - TODO: TMP
93:                 native_handler[9]->init[19]();
94:
95:                 return Success[15];
96:         }
97:
98:         Result[11] VirtualMachine[1]::shut_down()
99:         {
100:                // All the cleanup stuff
101:
102:                // Debug:
103:                // heap_manager->dump();
104:
105:                cout << "JVM ended..." << endl;
106:
```

Footnotes:

1: *VirtualMachine.h:255*
2: *VirtualMachine.h:258*
3: *VirtualMachine.h:263*
4: *VirtualMachine.h:267*
5: *VirtualMachine.h:278*
6: *VirtualMachine.h:281*
7: *VirtualMachine.h:291*
8: *VirtualMachine.h:295*
9: *VirtualMachine.h:270*
10: *VirtualMachine.h:298*
11: *defs.h:29*
12: *HeapManager.cpp:19*
13: *defs.h:453*
14: *VirtualMachine.cpp:77*
15: *defs.h:33*
16: *errors.cpp:35*
17: *defs.h:36*
18: *defs.h:37*
19: *NativeHandler.cpp:120*

```
107:                    return Success[1];
108:            }
109:
110:            // To start the primary thread we need to feed it with some code to execute.
111:            // This code will contain the call to the static method "main" of the main class
112:            Result[2] VirtualMachine[3]::start(char * filepath)
113:            {
114:                    // Add garbage collector thread to execution pool with lowest priority
115:                    execution_pool[4]->add_thread[5](garbage_collector_thread[6], MIN_PRIORITY[7]);
116:
117:                    // Imitate call of the static method that will invoke "main" method
118:                    // of the main class
119:                    ClassFile[8] * boot_class_file = new ClassFile[9](bootstrap_class_loader[10]);
120:                    Code_attribute[11] * boot_method;
121:                    InstanceData[12] * boot_instance;
122:                    Result[2] result = boot_class_file[13]->make_bootable[14](filepath[15], boot_method[16], boot_instance[17]);
123:                    assert(boot_method[16] != NULL);
124:                    assert(boot_instance[17] != NULL);
125:
126:                    // Boot process is finished
127:                    boot_process[18] = 0;
128:
129:                    // The "boot" method will call the "main" method of the main class
130:                    // Add primary thread to execution pool with normal priority
131:                    primary_thread[19] = run_thread[20](boot_method[16], boot_instance[17]);
132:
133:                    primary_thread[19]->debug[21] = 1; // debug this thread if compiled with debug flag
134:
135:                    // Run virtual machine
136:                    result[22] = run[23]();
137:                    if (result[22] != Success[1])
138:                            cout << "JVM ended abnormally" << endl;
139:
140:                    return Success[1];
141:            }
142:
143:            void VirtualMachine[3]::abnormal_termination(InstanceData[12] * exception_instance)
144:            {
145:                            bootstrap_class_loader[10]->dump_namespace[24]();
146:                            current_thread[25]->get_stack[26]()->debug_print[27]();
147:                            /*
148:                            cout << "Stack heap dump" << endl;
149:                            stack_heap_manager->dump();
150:                            cout << "Object heap dump" << endl;
151:                            heap_manager->dump();
152:                            */
153:                            cout <<"Exiting...";
154:                            exit(0);
155:            }
156:
157:
158:            // Wait for "by_thread" to be dead, then continue
159:            void VirtualMachine[3]::supercede(Thread[28] * thread, Thread[28] * by_thread)
```

Footnotes:

1: *defs.h:33*
2: *defs.h:29*
3: *VirtualMachine.h:255*
4: *VirtualMachine.h:291*
5: *VirtualMachine.cpp:344*
6: *VirtualMachine.h:278*
7: *defs.h:398*
8: *ClassFile.h:136*
9: *ClassFile.cpp:784*
10: *VirtualMachine.h:281*
11: *AttributeInfo.h:55*
12: *ObjectData.h:113*
13: *VirtualMachine.cpp:119*
14: *ClassFile.cpp:1124*
15: *VirtualMachine.cpp:112*
16: *VirtualMachine.cpp:120*
17: *VirtualMachine.cpp:121*
18: *VirtualMachine.h:302*
19: *VirtualMachine.h:284*
20: *VirtualMachine.cpp:287*
21: *Thread.h:184*
22: *VirtualMachine.cpp:122*
23: *VirtualMachine.cpp:258*
24: *ClassLoader.cpp:454*
25: *VirtualMachine.h:287*
26: *Thread.h:182*
27: *Stack.cpp:75*
28: *Thread.h:17*

```
160:         {
161:                 // Mark myself as a superceded thread
162:                 thread[1]->set_superceded[2]();
163:
164:                 // Give other threads to run
165:                 while (1)
166:                 {
167:                         // Pick up another candidate to run
168:                         Thread[3] * next_thread = execution_pool[4]->resched[5]();
169:                         assert(next_thread[6] != NULL); // this cannot happen - we have at least the current thread
170:
171:                         // If the execution engine has picked up the yielding thread as the next candidate
172:                         // we must return and accomplish the step that has been postponed by calling
173:                         // the "supercede" function
174:                         if (next_thread[6]->get_id[7]() == thread[1]->get_id[7]())
175:                         {
176:                                 assert(thread[1]->is_superceded[8]()); // this thread must be in "yield" state
177:
178:                                 // Only if by_thread is dead (not anything else !) the current thread
179:                                 // can continue execution
180:                                 if (by_thread[9]->is_dead[10]())
181:                                 {
182:                                         // Be running again
183:                                         thread[1]->set_running[11]();
184:                                         current_thread[12] = thread[1]; // current again
185:                                         return;
186:                                 }
187:                                 else
188:                                         continue;
189:                         }
190:
191:                         // If this is another thread we have to check that it is not in the "yield" state
192:                         if (next_thread[6]->is_yield[13]() || next_thread[6]->is_superceded[8]())
193:                                 // Try to choose another thread, this one is itself in the "yield" state
194:                                 // and is in the middle of incomplete "step", so no other steps could be
195:                                 // performed
196:                                 continue;
197:
198:                         // Otherwise, continue to execute another thread
199:                         Result[14] result = next_thread[6]->step[15]();
200:
201:                         // Check the result of the last instruction executed
202:                         if (result[16] != Success[17])
203:                         {
204:                                 abnormal_termination[18]();
205:                                 return;
206:                         }
207:                 }
208:         }
209:
210:
211:
212:
```

Footnotes:

1: *VirtualMachine.cpp:159*
2: *Thread.h:170*
3: *Thread.h:17*
4: *VirtualMachine.h:291*
5: *VirtualMachine.cpp:382*
6: *VirtualMachine.cpp:168*
7: *Thread.h:82*
8: *Thread.h:177*
9: *VirtualMachine.cpp:159*
10: *Thread.h:172*
11: *Thread.h:166*
12: *VirtualMachine.h:287*
13: *Thread.h:176*
14: *defs.h:29*
15: *Thread.cpp:907*
16: *VirtualMachine.cpp:199*
17: *defs.h:33*
18: *VirtualMachine.cpp:143*

```
213:        void VirtualMachine[1]::yield(Thread[2] * thread)
214:        {
215:                // Mark myself as yielding thread
216:                thread[3]->set_yield[4]();
217:
218:                // Give other threads to run
219:                while (1)
220:                {
221:                        // Pick up another candidate to run
222:                        Thread[2] * next_thread = execution_pool[5]->resched[6]();
223:                        assert(next_thread[7] != NULL); // this cannot happen - we have at least the current thread
224:
225:                        // If the execution engine has picked up the yielding thread as the next candidate
226:                        // we must return and accomplish the step that has been postponed by calling
227:                        // the "yield" function
228:                        if (next_thread[7]->get_id[8]() == thread[3]->get_id[8]())
229:                        {
230:                                assert(thread[3]->is_yield[9]()); // this thread must be in "yield" state
231:
232:                                // Be running again
233:                                thread[3]->set_running[10]();
234:                                current_thread[11] = thread[3]; // current again
235:                                return;
236:                        }
237:
238:                        // If this is another thread we have to check that it is not in the "yield" state
239:                        if (next_thread[7]->is_yield[9]() || next_thread[7]->is_superceded[12]())
240:                                // Try to choose another thread, this one is itself in the "yield" state
241:                                // and is in the middle of incomplete "step", so no other steps could be
242:                                // performed
243:                                continue;
244:
245:                        // Otherwise, continue to execute another thread
246:                        Result[13] result = next_thread[7]->step[14]();
247:
248:                        // Check the result of the last instruction executed
249:                        if (result[15] != Success[16])
250:                        {
251:                                abnormal_termination[17]();
252:                                return;
253:                        }
254:                }
255:        }
256:
257:        // Main execution loop of the JVM
258:        Result[13] VirtualMachine[1]::run()
259:        {
260:                // Forever
261:                while (1)
262:                {
263:                        Thread[2] * thread = execution_pool[5]->resched[6]();
264:                        if (!thread[18])
265:                        {
```

Footnotes:

1: *VirtualMachine.h:255*
2: *Thread.h:17*
3: *VirtualMachine.cpp:213*
4: *Thread.h:169*
5: *VirtualMachine.h:291*
6: *VirtualMachine.cpp:382*
7: *VirtualMachine.cpp:222*
8: *Thread.h:82*
9: *Thread.h:176*
10: *Thread.h:166*
11: *VirtualMachine.h:287*
12: *Thread.h:177*
13: *defs.h:29*
14: *Thread.cpp:907*
15: *VirtualMachine.cpp:246*
16: *defs.h:33*
17: *VirtualMachine.cpp:143*
18: *VirtualMachine.cpp:263*

```
266:                                    // We don't have any running thread - sleep ?
267:                                    //continue;
268:                                    // For the time being - just finish the execution
269:                                    break;
270:                            }
271:
272:                            Result[1] result = thread[2]->step[3]();
273:
274:                            // Check the result of the last instruction executed
275:                            if (result[4] != Success[5])
276:                            {
277:                                    abnormal_termination[6]();
278:                                    return Failure[7];
279:                            }
280:                    }
281:
282:                    return Success[5];
283:            }
284:
285:
286:            // Adds a new, ready to run thread to the VM with the given priority
287:            Thread[8] * VirtualMachine[9]::run_thread(Code_attribute[10] * code_attribute,
288:                                                                                    InstanceData[11] * thread_instance,
289:                                                                                    Priority[12] priority)
290:            {
291:                    assert(code_attribute[13] != NULL);
292:
293:                    // Create a new thread of execution;
294:                    Thread[8] * thread = new Thread[14](this, execution_pool[15]->get_next_thread_id[16]());
295:                    // Initialize thread with the given method's code and the instance
296:                    // which is of type java.lang.Thread associated with this thread
297:                    thread[17]->start[18](code_attribute[13], thread_instance[19]);
298:                    // Add thread to the execution pool with given priority
299:                    execution_pool[15]->add_thread[20](thread[17], priority[21]);
300:
301:                    // Thread is ready to run; its first instruction will be chosen eventually
302:                    // by the execution engine
303:                    return thread[17];
304:            }
305:
306:
307:            // This function is called by the registerNatives function of all the
308:            // native class implementations
309:            void VirtualMachine[9]::register_method(char * name, char * descriptor, void * func_ptr)
310:            {
311:                    native_handler[22]->register_method[23](name[24], descriptor[25], func_ptr[26]);
312:            }
313:
314:
315:            //-----------------------------------------------------------------------------
316:            // ExecutionPool
317:            //-----------------------------------------------------------------------------
318:
```

Footnotes:

1: *defs.h:29*
2: *VirtualMachine.cpp:263*
3: *Thread.cpp:907*
4: *VirtualMachine.cpp:272*
5: *defs.h:33*
6: *VirtualMachine.cpp:143*
7: *defs.h:34*
8: *Thread.h:17*
9: *VirtualMachine.h:255*
10: *AttributeInfo.h:55*
11: *ObjectData.h:113*
12: *defs.h:396*
13: *VirtualMachine.cpp:287*
14: *Thread.h:73*
15: *VirtualMachine.h:291*
16: *VirtualMachine.h:389*
17: *VirtualMachine.cpp:294*
18: *Thread.cpp:17*
19: *VirtualMachine.cpp:288*
20: *VirtualMachine.cpp:344*
21: *VirtualMachine.cpp:289*
22: *VirtualMachine.h:270*
23: *NativeHandler.cpp:136*
24: *VirtualMachine.cpp:309*
25: *VirtualMachine.cpp:309*
26: *VirtualMachine.cpp:309*

```
319:  ExecutionPool[1]::ExecutionPool(VirtualMachine[2] * _vm) : vm[3](_vm), id_counter[4](0)
320:  {
321:          threads[5][MIN_PRIORITY[6]]  = new Vector[7]<Thread*>;
322:          threads[5][NORM_PRIORITY[8]] = new Vector[7]<Thread*>;
323:          threads[5][MAX_PRIORITY[9]]  = new Vector[7]<Thread*>;
324:          threads[5][SYS_PRIORITY[10]]  = new Vector[7]<Thread*>;
325:
326:          current_index[11][MIN_PRIORITY[6]]  = 0;
327:          current_index[11][NORM_PRIORITY[8]] = 0;
328:          current_index[11][MAX_PRIORITY[9]]  = 0;
329:          current_index[11][SYS_PRIORITY[10]]  = 0;
330:
331:          current_priority[12] = SYS_PRIORITY[10];
332:  }
333:
334:  ExecutionPool[1]::~ExecutionPool()
335:  {
336:          // TODO: delete the threads first
337:
338:          delete threads[5][MIN_PRIORITY[6]];
339:          delete threads[5][NORM_PRIORITY[8]];
340:          delete threads[5][MAX_PRIORITY[9]];
341:          delete threads[5][SYS_PRIORITY[10]];
342:  }
343:
344:  void ExecutionPool[1]::add_thread(Thread[13] * thread, Priority[14] priority)
345:  {
346:          // Set this thread's priority
347:          thread[15]->set_priority[16](priority[17]);
348:          // Add thread to its priority vector
349:          threads[5][priority[17]]->add_element(thread[15]);
350:  }
351:
352:  int ExecutionPool[1]::threads_number()
353:  {
354:          return threads[5][MIN_PRIORITY[6]]->size() +
355:                     threads[5][NORM_PRIORITY[8]]->size() +
356:                     threads[5][MAX_PRIORITY[9]]->size() +
357:                     threads[5][SYS_PRIORITY[10]]->size();
358:  }
359:
360:  void ExecutionPool[1]::change_priority(Thread[13] * thread, Priority[14] new_priority)
361:  {
362:          Vector[18]<Thread*> * vector = threads[5][thread[19]->get_priority[20]()];
363:          for (int i = 0; i < vector[21]->size[22](); i++)
364:          {
365:                  Thread[13] * t = vector[21]->element_at[23](i[24]);
366:                  if (t[25]->get_id[26]() != thread[19]->get_id[26]())
367:                          continue;
368:                  // Thread found, remove it from the current priority vector
369:                  vector[21]->remove_element_at[27](i[24]);
370:                  // Insert into the new priority vector
371:                  threads[5][new_priority[28]]->add_element(thread[19]);
```

Footnotes:

1: *VirtualMachine.h:355*
2: *VirtualMachine.h:255*
3: *VirtualMachine.h:360*
4: *VirtualMachine.h:361*
5: *VirtualMachine.h:368*
6: *defs.h:398*
7: *vector.h:17*
8: *defs.h:399*
9: *defs.h:400*
10: *defs.h:401*
11: *VirtualMachine.h:373*
12: *VirtualMachine.h:371*
13: *Thread.h:17*
14: *defs.h:396*
15: *VirtualMachine.cpp:344*
16: *Thread.h:39*
17: *VirtualMachine.cpp:344*
18: *vector.h:7*
19: *VirtualMachine.cpp:360*
20: *Thread.h:181*
21: *VirtualMachine.cpp:362*
22: *vector.h:23*
23: *vector.h:36*
24: *VirtualMachine.cpp:363*
25: *VirtualMachine.cpp:365*
26: *Thread.h:82*
27: *vector.h:49*
28: *VirtualMachine.cpp:360*

```
372:                            // Change thread's priority
373:                            thread[1]->set_priority[2](new_priority[3]);
374:                    }
375:            }
376:
377:            // Pick up the next thread eligible for execution;
378:            // The search is based on the priority queues: we start searching from the queue
379:            // of the highest priority threads, then (in case no candidates are found) lower the
380:            // priority and so on.
381:            // The function also responsible for destroying the dead threads
382:            Thread[4] * ExecutionPool[5]::resched()
383:            {
384:            ////////////////////
385:            //      if (threads_number() == 1)
386:                                    // We don't need the garbage collector to run alone
387:            //                      return NULL;
388:            ///////////////////////////////////
389:
390:                    // NOTE: current_index always points to the thread that is to be executed,
391:                    // it will be advanced after the execution of the current thread
392:
393:                    // First, check that the current priority level is not empty
394:                    while (threads[6][current_priority[7]]->is_empty())
395:                    {
396:                            // Go to the lower priority
397:                            current_priority[7]--;
398:
399:                            // Check whether the only running thread is the garbage collector
400:                            if ((current_priority[7] == MIN_PRIORITY[8]) && (threads_number[9]() == 1))
401:                                    // We don't need the garbage collector to run alone
402:                                    return NULL;
403:
404:                            if (current_priority[7] < MIN_PRIORITY[8])
405:                                    // We don't have any thread that is ready to run !
406:                                    return NULL;
407:                    }
408:
409:                    // Check all threads of this priority level
410:                    for (int i = 0; i < threads[6][current_priority[7]]->size(); i++)
411:                    {
412:                            // Wrap the current_index of the corresponding priority if needed
413:                            if (current_index[10][current_priority[7]] >= threads[6][current_priority[7]]->size())
414:                                    current_index[10][current_priority[7]] = 0;
415:
416:                            // Get the candidate thread
417:                            Thread[4] * thread = threads[6][current_priority[7]]->element_at(current_index[10][current_priority[7]]);
418:
419:                            // Now we have to check whether current_index (that was advanced in the
420:                            // previous step) is still pointing to the thread that is not dead
421:                            // (the thread may be marked as dead in the previous step)
422:                            if (thread[11]->is_dead[12]())
423:                            {
```

Footnotes:

1: *VirtualMachine.cpp:360*
2: *Thread.h:39*
3: *VirtualMachine.cpp:360*
4: *Thread.h:17*
5: *VirtualMachine.h:355*
6: *VirtualMachine.h:368*
7: *VirtualMachine.h:371*
8: *defs.h:398*
9: *VirtualMachine.cpp:352*
10: *VirtualMachine.h:373*
11: *VirtualMachine.cpp:417*
12: *Thread.h:172*

```
424:                                    // Check whether the dead thread has raised any exception before dying
425:                                    if (thread[1]->raised_exception_instance[2])
426:                                    {
427:                                            // Notify JVM about it
428:                                            vm[3]->abnormal_termination[4](thread[1]->raised_exception_instance[2]);
429:                                            return NULL;
430:                                    }
431:                                    // Deallocate resources allocated by this thread
432:                                    thread[1]->destroy[5]();
433:                                    // Remove this thread from the list
434:                                    threads[6][current_priority[7]]->remove_element_at(current_index[8][current_priority[7]]);
435:                                    // Advance the counter
436:                                    current_index[8][current_priority[7]]++;
437:                                    // Call the function recursively to get another candidate
438:                                    return resched[9]();
439:                            }
440:                            // Check whether the current thread is ready to run
441:                            else if (thread[1]->is_running[10]() || thread[1]->is_yield[11]() || thread[1]->is_superceded[12]())
442:                            {
443:                                    // Prepare for the next cycle:
444:                                    // 1) switch to the next thread on this priority level
445:                                    current_index[8][current_priority[7]]++;
446:                                    // 2) we always start checking at the highest priority
447:                                    current_priority[7] = SYS_PRIORITY[13];
448:
449:                                    return thread[1];
450:                            }
451:                            // Otherwise, check another candidate
452:                            else
453:                                    current_index[8][current_priority[7]]++;
454:                    }
455:
456:                    return NULL;
457:            }
```

Footnotes:

1: *VirtualMachine.cpp:417*
2: *Thread.h:134*
3: *VirtualMachine.h:360*
4: *VirtualMachine.cpp:143*
5: *Thread.cpp:56*
6: *VirtualMachine.h:368*
7: *VirtualMachine.h:371*
8: *VirtualMachine.h:373*
9: *VirtualMachine.cpp:382*
10: *Thread.h:173*
11: *Thread.h:176*
12: *Thread.h:177*
13: *defs.h:401*

```
/*****************************************************************************************

 Decipher.cpp
 ------------

 The functions of this file decipher the method and field descriptors given their
 internal representaion in the class file constant pool.
 This representation and the descriptor grammar is shown below.

*****************************************************************************************/

#include "defs.h"
#include "vector.h"

/*****************************************************************************************
                                        Descriptors
                                                                              -----------

A descriptor is a string representing the type of a field or method. Descriptors are
represented in the class file format using UTF-8 strings and thus may be drawn,
where not further constrained, from the entire Unicode character set.

Field Descriptors
-----------------
A field descriptor represents the type of a class, instance, or local variable.
It is a series of characters generated by the grammar:

FieldDescriptor: FieldType
ComponentType: FieldType
FieldType: BaseType | ObjectType | ArrayType
BaseType: B | C | D | F | I | J | S | Z
ObjectType: L <classname> ;
ArrayType: [ ComponentType

The characters of BaseType, the L and ; of ObjectType, and the [ of ArrayType are all
ASCII characters. The <classname> represents a fully qualified class or interface name.
The interpretation of the field types is shown below:

BaseType      Character Type  Interpretation
---------------------------------------
B             byte            signed byte
C             char            Unicode character
D             double          double-precision floating-point value
F             float           single-precision floating-point value
I             int             integer
J             long            long integer
L<classname>; reference       an instance of class <classname>
S             short           signed short
Z             boolean         true or false
[             reference       one array dimension


For example, the descriptor of an instance variable of type int is simply I.
```

```
54:         The descriptor of an instance variable of type Object is Ljava/lang/Object;.
55:         Note that the internal form of the fully qualified name for class Object is used.
56:         The descriptor of an instance variable that is a multidimensional double array,
57:
58:             double d[][][];
59:
60:         is
61:
62:             [[[D
63:
64:         Method Descriptors
65:         ------------------
66:
67:         A method descriptor represents the parameters that the method takes and the value that
68:         it returns:
69:
70:         MethodDescriptor: ( ParameterDescriptor* ) ReturnDescriptor
71:
72:         A parameter descriptor represents a parameter passed to a method:
73:
74:         ParameterDescriptor: FieldType
75:
76:         A return descriptor represents the type of the value returned from a method.
77:         It is a series of characters generated by the grammar:
78:
79:         ReturnDescriptor: FieldType | V
80:
81:         The character V indicates that the method returns no value (its return type is void).
82:         A method descriptor is valid only if it represents method parameters with a total length
83:         of 255 or less, where that length includes the contribution for this in the case of instance
84:         or interface method invocations. The total length is calculated by summing the contributions
85:         of the individual parameters, where a parameter of type long or double contributes two units
86:         to the length and a parameter of any other type contributes one unit.
87:
88:         For example, the method descriptor for the method
89:
90:             Object mymethod(int i, double d, Thread t)
91:
92:         is
93:
94:             (IDLjava/lang/Thread;)Ljava/lang/Object;
95:
96:         Note that internal forms of the fully qualified names of Thread and Object are used in
97:         the method descriptor.
98:         The method descriptor for mymethod is the same whether mymethod is a class or an instance
99:         method. Although an instance method is passed this, a reference to the current class instance,
100:        in addition to its intended parameters, that fact is not reflected in the method descriptor.
101:        (A reference to this is not passed to a class method.) The reference to this is passed
102:        implicitly by the method invocation instructions of the Java virtual machine used to invoke
103:        instance methods.
104:
105:        ********************************************************************************/
106:
```

```
107:
108:        Result[1] read_basic_type[2](char ch, int & index, general_type[3] * field);
109:        Result[1] read_array[4](char * string, int & index, general_type[3] * field);
110:        Result[1] read_reference_type[5](char * string, int & index, general_type[3] * field);
111:        Result[1] read_field[6](char * string, int & index, general_type[3] * field);
112:
113:        Result[1] decipher_method_descriptor(char * string, Vector[7]<general_type*> & args, general_type[3] * return_typ
            e)
114:        {
115:                int index = 0;
116:
117:                if (string[8][index[9]++] != '(')
118:                        return MethodDescriptorWrongFormat[10];
119:
120:                while (1)
121:                {
122:                        general_type[3] * field = new general_type[11]; // dimension is initialized to 0, kind to None
123:
124:                        Result[1] result = read_field[6](string[8], index[9], field[12]);
125:                        if (result[13] != Success[14])
126:                                return MethodDescriptorWrongFormat[10];
127:
128:                        if (field[12]->kind[15] != general_type[3]::None[16])
129:                                args[17].add_element[18](field[12]);
130:
131:                        if (string[8][index[9]] == ')')
132:                                break;
133:                }
134:
135:                index[9]++;
136:                Result[1] result = read_field[6](string[8], index[9], return_type[19]);
137:                if (result[20] != Success[14])
138:                        return MethodDescriptorWrongFormat[10];
139:
140:                return Success[14];
141:        }
142:
143:        Result[1] read_field(char * string, int & index, general_type[3] * field)
144:        {
145:                Result[1] result;
146:
147:                char ch = string[21][index[22]++];
148:                if (ch[23] == '[')
149:                {
150:                        result[24] = read_array[4](string[21], index[22], field[25]);
151:                        field[25]->kind[15] = general_type[3]::Array[26];
152:                        return result[24];
153:                }
154:                else if (ch[23] == 'L')
155:                {
156:                        result[24] = read_reference_type[5](string[21], index[22], field[25]);
157:                        field[25]->kind[15] = field[25]->array_kind[27] = general_type[3]::Reference[28];
158:                        return result[24];
```

Footnotes:

1: *defs.h:29*
2: *decipher.cpp:173*
3: *defs.h:231*
4: *decipher.cpp:194*
5: *decipher.cpp:202*
6: *decipher.cpp:143*
7: *vector.h:7*
8: *decipher.cpp:113*
9: *decipher.cpp:115*
10: *defs.h:87*
11: *defs.h:239*
12: *decipher.cpp:122*
13: *decipher.cpp:124*
14: *defs.h:33*
15: *defs.h:233*
16: *defs.h:233*
17: *decipher.cpp:113*
18: *vector.h:40*
19: *decipher.cpp:113*
20: *decipher.cpp:136*
21: *decipher.cpp:143*
22: *decipher.cpp:143*
23: *decipher.cpp:147*
24: *decipher.cpp:145*
25: *decipher.cpp:143*
26: *defs.h:233*
27: *defs.h:234*
28: *defs.h:233*

```
159:                }
160:                else if (ch[1] == ')')
161:                {
162:                        --index[2];
163:                        return Success[3];
164:                }
165:                else
166:                {
167:                        result[4] = read_basic_type[5](ch[1], index[2], field[6]);
168:                        field[6]->kind[7] = field[6]->array_kind[8] = general_type[9]::Basic[10];
169:                        return result[4];
170:                }
171:        }
172:
173:        Result[11] read_basic_type(char ch, int & index, general_type[9] * field)
174:        {
175:                field[12]->kind[7] = general_type[9]::Basic[10];
176:
177:                switch (ch[13])
178:                {
179:                case 'B':  field[12]->type[14] = Byte[15]; break;
180:                case 'C':  field[12]->type[14] = Char[16]; break;
181:                case 'D':  field[12]->type[14] = Double[17]; break;
182:                case 'F':  field[12]->type[14] = Float[18]; break;
183:                case 'I':  field[12]->type[14] = Int[19]; break;
184:                case 'J':  field[12]->type[14] = Long[20]; break;
185:                case 'S':  field[12]->type[14] = Short[21]; break;
186:                case 'Z':  field[12]->type[14] = Boolean[22]; break;
187:            case 'V':  field[12]->type[14] = Void[23]; break;
188:                default: return Failure[24];
189:                }
190:
191:                return Success[3];
192:        }
193:
194:        Result[11] read_array(char * string, int & index, general_type[9] * field)
195:        {
196:                field[25]->kind[7] = general_type[9]::Array[26];
197:                field[25]->dimension[27]++;
198:
199:                return read_field[28](string[29], index[30], field[25]);
200:        }
201:
202:        Result[11] read_reference_type(char * string, int & index, general_type[9] * field)
203:        {
204:                field[31]->kind[7] = general_type[9]::Reference[32];
205:                char ch;
206:                int counter = 0;
207:
208:                while ( (ch[33]=string[34][index[35]++]) != ';' )
209:                        field[31]->qualified_name[36][counter[37]++] = ch[33];
210:
211:                field[31]->qualified_name[36][counter[37]] = 0;
```

Footnotes:

1: *decipher.cpp:147*
2: *decipher.cpp:143*
3: *defs.h:33*
4: *decipher.cpp:145*
5: *decipher.cpp:173*
6: *decipher.cpp:143*
7: *defs.h:233*
8: *defs.h:234*
9: *defs.h:231*
10: *defs.h:233*
11: *defs.h:29*
12: *decipher.cpp:173*
13: *decipher.cpp:173*
14: *defs.h:237*
15: *defs.h:210*
16: *defs.h:211*
17: *defs.h:212*
18: *defs.h:213*
19: *defs.h:214*
20: *defs.h:215*
21: *defs.h:217*
22: *defs.h:218*
23: *defs.h:220*
24: *defs.h:34*
25: *decipher.cpp:194*
26: *defs.h:233*
27: *defs.h:236*
28: *decipher.cpp:143*
29: *decipher.cpp:194*
30: *decipher.cpp:194*
31: *decipher.cpp:202*
32: *defs.h:233*
33: *decipher.cpp:205*
34: *decipher.cpp:202*
35: *decipher.cpp:202*
36: *defs.h:235*
37: *decipher.cpp:206*

```
212:
213:                    return Success[1];
214:            }
```

Footnotes:

1: *defs.h:33*

```
 1:
 2:          #include <iostream.h>
 3:
 4:          #include "defs.h"
 5:
 6:
 7:          exception_class[1] ExceptionClasses[] =
 8:          {
 9:                  { VM_ERROR_NoClassDefFoundError[2], L"java/lang/NoClassDefFoundError" },
10:                  { VM_ERROR_ClassNotFoundError[3], L"java/lang/ClassNotFoundError" },
11:                  { VM_ERROR_ClassFormatError[4], L"java/lang/ClassFormatError" },
12:                  { VM_ERROR_UnsupportedClassVersionError[5], L"java/lang/UnsupportedClassVersionError" },
13:                  { VM_ERROR_LinkageError[6], L"java/lang/LinkageError" },
14:                  { VM_ERROR_ClassCircularityError[7], L"java/lang/ClassCircularityError" },
15:                  { VM_ERROR_IllegalAccessError[8], L"java/lang/IllegalAccessError" },
16:                  { VM_ERROR_VerifyError[9], L"java/lang/VerifyError" },
17:                  { VM_ERROR_IncompatibleClassChangeError[10], L"java/lang/IncompatibleClassChangeError" },
18:                  { VM_ERROR_NoSuchMethodError[11], L"java/lang/NoSuchMethodError" },
19:                  { VM_ERROR_NoSuchFieldError[12], L"java/lang/NoSuchFieldError" },
20:                  { VM_ERROR_AbstractMethodError[13], L"java/lang/AbstractMethodError" },
21:                  { VM_ERROR_NullPointerException[14], L"java/lang/NullPointerException" },
22:                  { VM_ERROR_IllegalMonitorStateException[15], L"java/lang/IllegalMonitorStateException" },
23:                  { VM_ERROR_NegativeArraySizeException[16], L"java/lang/NegativeArraySizeException" },
24:                  { VM_ERROR_ArrayIndexOutOfBoundsException[17], L"java/lang/ArrayIndexOutOfBoundsException" },
25:                  { VM_ERROR_ArrayStoreExceptionClass[18], L"java/lang/ArrayStoreExceptionClass" },
26:                  { VM_ERROR_ArrayStoreException[19], L"java/lang/ArrayStoreException" },
27:                  { VM_ERROR_ClassCastException[20], L"java/lang/ClassCastException" },
28:                  { VM_ERROR_ArithmeticException[21], L"java/lang/ArithmeticException" },
29:                  { VM_ERROR_END[22], L"" }
30:          };
31:
32:          #define ERROR_STRING(a) #a
33:
34:
35:          void print_error(Result[23] result)
36:          {
37:                  switch (result[24])
38:                  {
39:                  case Success[25]:
40:                          cout << "Success" << endl;
41:                          break;
42:                  case Failure[26]:
43:                          cout << "Failure" << endl;
44:                          break;
45:                  case HeapInitializationError[27]:
46:                          cout << "HeapInitializationError" << endl;
47:                          break;
48:                  case GarbageCollectorInitializationError[28]:
49:                          cout << "GarbageCollectorInitializationError" << endl;
50:                          break;
51:                  case Utf8Error[29]:
52:                          cout << "Utf8Error" << endl;
53:                          break;
```

Footnotes:

1: *defs.h:149*
2: *defs.h:93*
3: *defs.h:94*
4: *defs.h:95*
5: *defs.h:96*
6: *defs.h:97*
7: *defs.h:98*
8: *defs.h:99*
9: *defs.h:100*
10: *defs.h:101*
11: *defs.h:102*
12: *defs.h:103*
13: *defs.h:104*
14: *defs.h:105*
15: *defs.h:106*
16: *defs.h:107*
17: *defs.h:108*
18: *defs.h:109*
19: *defs.h:110*
20: *defs.h:111*
21: *defs.h:112*
22: *defs.h:116*
23: *defs.h:29*
24: *errors.cpp:35*
25: *defs.h:33*
26: *defs.h:34*
27: *defs.h:35*
28: *defs.h:39*
29: *defs.h:40*

```
54:                     case ClassLoaderErrorWrongMagicNumber[1]:
55:                             cout << "ClassLoaderErrorWrongMagicNumber" << endl;
56:                             break;
57:                     case ClassLoaderErrorWrongConstantPoolTag[2]:
58:                             cout << "ClassLoaderErrorWrongConstantPoolTag" << endl;
59:                             break;
60:                     case ClassLoaderErrorUnknownAttribute[3]:
61:                             cout << "ClassLoaderErrorUnknownAttribute" << endl;
62:                             break;
63:                     case ClassFileNotFoundError[4]:
64:                             cout << "ClassFileNotFoundError" << endl;
65:                             break;
66:                     case ClassFileReadError[5]:
67:                             cout << "ClassFileReadError" << endl;
68:                             break;
69:                     case HeapErrorCannotAllocateInitialSize[6]:
70:                             cout << "HeapErrorCannotAllocateInitialSize" << endl;
71:                             break;
72:                     case HeapErrorCannotAllocateMemory[7]:
73:                             cout << "HeapErrorCannotAllocateMemory" << endl;
74:                             break;
75:                     case HeapErrorCannotFindMemoryChunkToDeallocate[8]:
76:                             cout << "HeapErrorCannotFindMemoryChunkToDeallocate" << endl;
77:                             break;
78:                     default:
79:                             cout << "Unknown error" << endl;
80:
81:                     }
82:             }
83:
84:             void print_instruction_error(InstructionResult[9] result)
85:             {
86:                     cout << "Instruction error" << endl;
87:             }
```

Footnotes:

1: *defs.h:44*
2: *defs.h:45*
3: *defs.h:47*
4: *defs.h:52*
5: *defs.h:53*
6: *defs.h:54*
7: *defs.h:55*
8: *defs.h:56*
9: *defs.h:120*

```
 1:
 2:          #include "hashtable.h"
 3:
 4:          HashTable[1]::HashTable(int Capacity, float FillPercent) :
 5:           count[2](0L), capacity[3](Capacity[4]),  fill_percent[5](FillPercent[6]) {
 6:
 7:             table[7] = new HashTableEntry[8]* [capacity[3]];
 8:
 9:             for(int i = capacity[3] ; i-- > 0 ; )
10:                table[7][i[9]] = 0;
11:             threshold[10] = (long)(capacity[3] * fill_percent[5]);
12:          }
13:
14:           long HashTable[1]::HashCode(char * p) {
15:               long h=0L;
16:               for(char *s=p[11] ; *s != 0 ; s++){
17:                   h[12] = h[12]*37 + *s[13];  // or  h ^= *s;  h = _lrotl(h, 1);
18:               }
19:               return h[12];
20:           }
21:
22:
23:           int HashTable[1]::insert(char * key, void * value){
24:             // Find record in the hashtable.
25:             long hash = HashCode[14](key[15]);
26:             int  index = (int) ((hash[16] & 0x7FFFFFFFL) % capacity[3]);
27:             for(HashTableEntry[17] *e = table[7][index[18]] ; e != 0 ; e = e->next[19]) {
28:                if(e[20]->hash[21] == hash[16] && !strcmp(e[20]->key[22],key[15])){
29:                   e[20]->value[23] = value[24];  // replace value.
30:                   return 1;
31:                }
32:             }
33:
34:             // Rehash the table if the threshold is exceeded.
35:             if(count[2] >= threshold[10]){
36:                rehash[25]();
37:                index[18] = (int)((hash[16] & 0x7FFFFFFFL) % capacity[3]);
38:             }
39:
40:             // Create the new entry.
41:             e = new HashTableEntry[8](key[15], value[24]);
42:
43:             e->hash  = hash[16];
44:             e->next  = table[7][index[18]];
45:             table[7][index[18]] = e;
46:             count[2]++;
47:             return 0;
48:          }
49:
50:          /*
51:          * Delete the element corresponding to the key. Return 1.
52:          * Does nothing if the key isn't present.       Return 0.
```

Footnotes:

1: *hashtable.h:28*
2: *hashtable.h:69*
3: *hashtable.h:68*
4: *hashtable.cpp:4*
5: *hashtable.h:71*
6: *hashtable.cpp:4*
7: *hashtable.h:67*
8: *hashtable.h:38*
9: *hashtable.cpp:9*
10: *hashtable.h:70*
11: *hashtable.cpp:14*
12: *hashtable.cpp:15*
13: *hashtable.cpp:16*
14: *hashtable.cpp:14*
15: *hashtable.cpp:23*
16: *hashtable.cpp:25*
17: *hashtable.h:32*
18: *hashtable.cpp:26*
19: *hashtable.h:36*
20: *hashtable.cpp:27*
21: *hashtable.h:33*
22: *hashtable.h:34*
23: *hashtable.h:35*
24: *hashtable.cpp:23*
25: *hashtable.cpp:109*

```
53:          */
54:          int HashTable[1]::remove(char * key) {
55:            long hash = HashCode[2](key[3]);
56:            int  index = (int)((hash[4] & 0x7FFFFFFFL) % capacity[5]);
57:
58:            HashTableEntry[6] *e, *prev;
59:            for(e[7] = table[8][index[9]], prev[10] = 0 ; e[7] != 0 ; prev[10] = e[7], e[7] = e[7]->next[11]){
60:               if(e[7]->hash[12] == hash[4] && !strcmp(e[7]->key[13],key[3])){
61:                  if(prev[10] != 0){
62:                     prev[10]->next[11] = e[7]->next[11];
63:                  } else {
64:                     table[8][index[9]] = e[7]->next[11];
65:                  }
66:                  delete e[7];
67:                  count[14]--;
68:                  return 1;
69:               }
70:            }
71:            return 0;
72:          }
73:
74:
75:          /* Find the value associated with a key in the hashtable.
76:          * return pointer to the element for the key or null if the key
77:          * is not defined in the hash table.
78:          */
79:          void* HashTable[1]::find(char * key) const {
80:            long hash = HashCode[2](key[15]);
81:            int  index = (int)((hash[16] & 0x7FFFFFFFL) % capacity[5]);
82:            for(HashTableEntry[6] *e = table[8][index[17]] ; e != 0 ; e = e->next[11])
83:               if(e[18]->hash[12] == hash[16] &&  !strcmp(e[18]->key[13],key[15]))
84:                  return e[18]->value[19];
85:            return 0;
86:          }
87:
88:
89:          /* Delete all elements from HashTable. Table array remains
90:          */
91:          void HashTable[1]::remove_all() {
92:            for(int i = capacity[5] ; i-- > 0 ; ){
93:               for(HashTableEntry[6] *e = table[8][i[20]] ; e != 0 ; ){
94:                   HashTableEntry[6] * tmp = e[21];
95:                   e[21] = e[21]->next[11];
96:                  delete [] tmp[22]->key[13];
97:                  delete tmp[22];
98:               }
99:               table[8][i[20]] = 0;
100:           }
101:           count[14] = 0;
102:         }
103:
104:
105:         /* Rehashes the content of the table into a bigger table.
```

Footnotes:

1: *hashtable.h:28*
2: *hashtable.cpp:14*
3: *hashtable.cpp:54*
4: *hashtable.cpp:55*
5: *hashtable.h:68*
6: *hashtable.h:32*
7: *hashtable.cpp:58*
8: *hashtable.h:67*
9: *hashtable.cpp:56*
10: *hashtable.cpp:58*
11: *hashtable.h:36*
12: *hashtable.h:33*
13: *hashtable.h:34*
14: *hashtable.h:69*
15: *hashtable.cpp:79*
16: *hashtable.cpp:80*
17: *hashtable.cpp:81*
18: *hashtable.cpp:82*
19: *hashtable.h:35*
20: *hashtable.cpp:92*
21: *hashtable.cpp:93*
22: *hashtable.cpp:94*

```
106:          * This is method called automatically when the hashtable's
107:          * size exeeds a threshold.
108:          */
109:           void HashTable[1]::rehash() {
110:             HashTableEntry[2] **oldTable = table[3];
111:             int oldCapacity = capacity[4];
112:
113:          /* !!!!!!!!!!!! for long memory model in Windows only  !!!!!!!!!!!!!!*/
114:          /* table->array of pointers can't be bigger than 64K, sizeof pointer 4 */
115:             if( capacity[4]*2 + 1 > 0xFFFF/4 ){
116:                threshold[5] = 10000000L;
117:                return;
118:             }
119:          /* !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!*/
120:
121:             capacity[4] = capacity[4]*2 + 1;
122:             table[3] = new HashTableEntry[6]* [capacity[4]];
123:
124:             for(int i = capacity[4] ; i-- > 0 ; )
125:                table[3][i[7]] = 0;
126:             threshold[5] = (long)(capacity[4] * fill_percent[8]);
127:
128:             HashTableEntry[2] *e, *eNext;
129:             for(i = oldCapacity[9] ; i-- > 0 ; ){
130:                for( e[10] = oldTable[11][i] ; e[10] != 0 ; e[10] = eNext[12] ){
131:                   eNext[12] = e[10]->next[13];
132:                   int index = (int)((e[10]->hash[14] & 0x7FFFFFFFL) % capacity[4]);
133:                   e[10]->next[13] = table[3][index[15]];
134:                   table[3][index[15]] = e[10];
135:                }
136:             }
137:             delete [] oldTable[11];
138:          }
139:
140:
141:          /*   Sets the iterator to the beginning
142:          */
143:           void HashTableIterator[16]::reset(){
144:              for( entry[17] = 0, index[18]=0 ; index[18] < hashtable[19].Capacity[20]() ; index[18]++ )
145:                  if( (hashtable[19].get_table[21]())[index[18]] != 0 ){
146:                      entry[17] = (hashtable[19].get_table[21]())[index[18]];
147:                      return;
148:                  }
149:          }
150:
151:          /*   Returns next element of the iteration
152:          */
153:           HashTable[1]::HashTableEntry[2]* HashTableIterator[16]::next(){
154:              if(entry[17]==0) return 0;
155:              HashTable[1]::HashTableEntry[2] *old_entry = entry[17];
156:              if(entry[17]->next[13] != 0)
157:                  entry[17] = entry[17]->next[13];
158:              else {
```

Footnotes:

1: *hashtable.h:28*
2: *hashtable.h:32*
3: *hashtable.h:67*
4: *hashtable.h:68*
5: *hashtable.h:70*
6: *hashtable.h:38*
7: *hashtable.cpp:124*
8: *hashtable.h:71*
9: *hashtable.cpp:111*
10: *hashtable.cpp:128*
11: *hashtable.cpp:110*
12: *hashtable.cpp:128*
13: *hashtable.h:36*
14: *hashtable.h:33*
15: *hashtable.cpp:132*
16: *hashtable.h:82*
17: *hashtable.h:84*
18: *hashtable.h:83*
19: *hashtable.h:85*
20: *hashtable.h:64*
21: *hashtable.h:63*

```
159:                 for( entry[1]=0, index[2]++ ; index[2] < hashtable[3].Capacity[4]() ; index[2]++ )
160:                     if( hashtable[3].get_table[5]()[index[2]] != 0 ){
161:                         entry[1] = (hashtable[3].get_table[5]())[index[2]];
162:                         break;
163:                     }
164:                 }
165:                 return old_entry[6];
166:             }
167:
168:
169:             ////////////////////////////////////////////////////////////////////////////////////////////////////
170:
171:             int MultiHashTable[7]::insert(char * key, void * value){
172:                // Find record in the hashtable.
173:                long hash = HashCode[8](key[9]);
174:                int  index = (int) ((hash[10] & 0x7FFFFFFFL) % capacity[11]);
175:
176:                // Rehash the table if the threshold is exceeded.
177:                if(count[12] >= threshold[13]){
178:                   rehash[14]();
179:                   index[15] = (int)((hash[10] & 0x7FFFFFFFL) % capacity[11]);
180:                }
181:
182:                // Create the new entry.
183:                HashTableEntry[16] * e = new HashTableEntry[17](key[9], value[18]);
184:
185:                e[19]->hash[20]  = hash[10];
186:                e[19]->next[21]  = table[22][index[15]];
187:                table[22][index[15]] = e[19];
188:                count[12]++;
189:                return 0;
190:             }
191:
192:             void * MultiHashTable[7]::pop(char * key)
193:             {
194:                 void * value = find[23](key[24]);
195:                 remove[25](key[24]);
196:                 return value[26];
197:             }
198:
199:             /*
200:
201:             int CSIDMultiHashTable::Insert(char * key, void * v)
202:             {
203:                // Find record in the hashtable.
204:                long hash = HashCode(key);
205:                int  index = (int) ((hash & 0x7FFFFFFFL) % capacity);
206:                for(HashTableEntry *e = table[index] ; e != 0 ; e = e->next) {
207:                   if(e->hash == hash && !strcmp(e->key,key)){
208:                       if (!e->value) {
209:                           e->value = new CSIDVector<void*>;
210:                       }
211:                       ((CSIDVector<void*>*)e->value)->addElement(v);
```

Footnotes:

1: *hashtable.h:84*
2: *hashtable.h:83*
3: *hashtable.h:85*
4: *hashtable.h:64*
5: *hashtable.h:63*
6: *hashtable.cpp:155*
7: *hashtable.h:96*
8: *hashtable.cpp:14*
9: *hashtable.cpp:171*
10: *hashtable.cpp:173*
11: *hashtable.h:68*
12: *hashtable.h:69*
13: *hashtable.h:70*
14: *hashtable.cpp:109*
15: *hashtable.cpp:174*
16: *hashtable.h:32*
17: *hashtable.h:38*
18: *hashtable.cpp:171*
19: *hashtable.cpp:183*
20: *hashtable.h:33*
21: *hashtable.h:36*
22: *hashtable.h:67*
23: *hashtable.cpp:79*
24: *hashtable.cpp:192*
25: *hashtable.cpp:54*
26: *hashtable.cpp:194*

```
212:                 return 1;
213:              }
214:           }
215:
216:           // Rehash the table if the threshold is exceeded.
217:           if(count >= threshold){
218:              Rehash();
219:              index = (int)((hash & 0x7FFFFFFFL) % capacity);
220:           }
221:
222:           // Create the new entry.
223:           e = new HashTableEntry(key, v);
224:
225:           e->hash  = hash;
226:           e->next  = table[index];
227:           table[index] = e;
228:           count++;
229:           return 0;
230:        }
231:
232:        CSIDVector<void*> * CSIDMultiHashTable::Find(char * k) const
233:        {
234:            CSIDVector<void*> * v = (CSIDVector<void*>*)HashTable::Find(k);
235:            return v;
236:        }
237:
238:        void * CSIDMultiHashTable::Pop(char * key)
239:        {
240:           long hash = HashCode(key);
241:           int  index = (int)((hash & 0x7FFFFFFFL) % capacity);
242:           for(HashTableEntry *e = table[index] ; e != 0 ; e = e->next)
243:               if(e->hash == hash &&  !strcmp(e->key,key) && e->value) {
244:                    CSIDVector<void*> * v = (CSIDVector<void*>*)e->value;
245:                    if (v->size() == 0)
246:                        return 0;
247:                    void * elem = v->elementAt(0);
248:                    v->removeElementAt(0);
249:                    return elem;
250:               }
251:           return 0;
252:        }
253:
254:        */
```

```
 1:
 2:          #include <memory.h>
 3:
 4:          #include "defs.h"
 5:
 6:          // Copying memory from big-endian byte stream into a short type variable will cause
 7:          // the short contain the wrong value.
 8:          // We reverse the bytes of the destination variable, so they are stored in the little-endian
 9:          // order representing the proper short value on the Intel architecture
10:          void memcpy_u2(u2[1] * dst, const u1[2] * src)
11:          {
12:                  memcpy(dst[3],src[4],sizeof(u2[1]));
13:
14:          #ifdef JVM_FOR_INTEL[5]
15:
16:                  u1[2] high_byte = ((*dst[3]) & 0xFF00) >> 8;
17:                  u1[2] low_byte = (*dst[3]) & 0xFF;
18:                  *dst[3] = (low_byte[6] << 8) | high_byte[7];
19:
20:          #endif
21:          }
22:
23:          // Copying memory from big-endian byte stream into a integer type variable will cause
24:          // the integer contain the wrong value.
25:          // We reverse the bytes of the destination variable, so they are stored in the little-endian
26:          // order representing the proper integer value on the Intel architecture
27:          void memcpy_u4(u4[8] * dst, const u1[2] * src)
28:          {
29:                  memcpy(dst[9],src[10],sizeof(u4[8]));
30:
31:          #ifdef JVM_FOR_INTEL[5]
32:
33:                  u1[2] byte1 = ((*dst[9]) & 0xFF000000) >> 24;
34:                  u1[2] byte2 = ((*dst[9]) & 0xFF0000) >> 16;
35:                  u1[2] byte3 = ((*dst[9]) & 0xFF00) >> 8;
36:                  u1[2] byte4 = ((*dst[9]) & 0xFF);
37:                  *dst[9] = (byte4[11] << 24) | (byte3[12] << 16) | (byte2[13] << 8) | byte1[14];
38:
39:          #endif
40:          }
41:
42:          void swap_bytes4(u4[8] * value)
43:          {
44:                  u1[2] byte1 = ((*value[15]) & 0xFF000000) >> 24;
45:                  u1[2] byte2 = ((*value[15]) & 0xFF0000) >> 16;
46:                  u1[2] byte3 = ((*value[15]) & 0xFF00) >> 8;
47:                  u1[2] byte4 = ((*value[15]) & 0xFF);
48:                  *value[15] = (byte4[16] << 24) | (byte3[17] << 16) | (byte2[18] << 8) | byte1[19];
49:          }
50:
51:          void swap_bytes8(u8[20] * value)
52:          {
53:                  u1[2] byte1 = ((*value[21]) & 0xFF00000000000000) >> 56;
```

Footnotes:

1: *defs.h:172*
2: *defs.h:168*
3: *memcpy_bigendian.cpp:10*
4: *memcpy_bigendian.cpp:10*
5: *defs.h:15*
6: *memcpy_bigendian.cpp:17*
7: *memcpy_bigendian.cpp:16*
8: *defs.h:174*
9: *memcpy_bigendian.cpp:27*
10: *memcpy_bigendian.cpp:27*
11: *memcpy_bigendian.cpp:36*
12: *memcpy_bigendian.cpp:35*
13: *memcpy_bigendian.cpp:34*
14: *memcpy_bigendian.cpp:33*
15: *memcpy_bigendian.cpp:42*
16: *memcpy_bigendian.cpp:47*
17: *memcpy_bigendian.cpp:46*
18: *memcpy_bigendian.cpp:45*
19: *memcpy_bigendian.cpp:44*
20: *defs.h:176*
21: *memcpy_bigendian.cpp:51*

```
54:                 u1[1] byte2 = ((*value[2]) & 0xFF000000000000) >> 48;
55:                 u1[1] byte3 = ((*value[2]) & 0xFF0000000000) >> 40;
56:                 u1[1] byte4 = ((*value[2]) & 0xFF00000000) >> 32;
57:                 u1[1] byte5 = ((*value[2]) & 0xFF000000) >> 24;
58:                 u1[1] byte6 = ((*value[2]) & 0xFF0000) >> 16;
59:                 u1[1] byte7 = ((*value[2]) & 0xFF00) >> 8;
60:                 u1[1] byte8 = ((*value[2]) & 0xFF);
61:                 *value[2] = (byte8[3] << 56) | (byte7[4] << 48) | (byte6[5] << 40) | (byte5[6] << 32) | (byte4[7] << 24) | (byte
           3[8] << 16) | (byte2[9] << 8) | byte1[10];
62:         }
63:
```

Footnotes:

1: *defs.h:168*
2: *memcpy_bigendian.cpp:51*
3: *memcpy_bigendian.cpp:60*
4: *memcpy_bigendian.cpp:59*
5: *memcpy_bigendian.cpp:58*
6: *memcpy_bigendian.cpp:57*
7: *memcpy_bigendian.cpp:56*
8: *memcpy_bigendian.cpp:55*
9: *memcpy_bigendian.cpp:54*
10: *memcpy_bigendian.cpp:53*

```
 1:
 2:          #include <memory.h>
 3:          #include <assert.h>
 4:          #include <iostream.h>
 5:
 6:          #include "opcodes.h"
 7:          #include "Stack.h"
 8:          #include "Thread.h"
 9:          #include "ConstantPool.h"
10:          #include "ClassFile.h"
11:          #include "defs.h"
12:          #include "ObjectData.h"
13:          #include "ClassLoader.h"
14:          #include "HandlePool.h"
15:
16:          InstructionResult[1] nop(Thread[2] * thread)
17:          {
18:                  return InstructionSuccess[3];
19:          }
20:
21:          //-------------------------------------------------------------------------------
22:          // Local variable operations
23:          //-------------------------------------------------------------------------------
24:
25:          InstructionResult[1] aconst_null(Thread[2] * thread)
26:          {
27:                  thread[4]->current_stack_frame[5]->push_reference[6](null[7]);
28:                  return InstructionSuccess[3];
29:          }
30:
31:          inline static InstructionResult[1] _iload_n(Thread[2] * thread, u1[8] n)
32:          {
33:                  stack_frame[9] * frame = thread[10]->current_stack_frame[5];
34:                  su4[11] int_value = frame[12]->get_int[13](n[14]);
35:                  frame[12]->push_int[15](int_value[16]);
36:
37:                  return InstructionSuccess[3];
38:          }
39:
40:          InstructionResult[1] iload_0(Thread[2] * thread)
41:          {
42:                  return _iload_n[17](thread[18], 0);
43:          }
44:          InstructionResult[1] iload_1(Thread[2] * thread)
45:          {
46:                  return _iload_n[17](thread[19], 1);
47:          }
48:          InstructionResult[1] iload_2(Thread[2] * thread)
49:          {
50:                  return _iload_n[17](thread[20], 2);
51:          }
52:          InstructionResult[1] iload_3(Thread[2] * thread)
53:          {
```

Footnotes:

1: *defs.h:120*
2: *Thread.h:17*
3: *defs.h:122*
4: *opcodes.cpp:25*
5: *Thread.h:148*
6: *Stack.cpp:187*
7: *defs.h:201*
8: *defs.h:168*
9: *Stack.h:129*
10: *opcodes.cpp:31*
11: *defs.h:175*
12: *opcodes.cpp:33*
13: *Stack.cpp:357*
14: *opcodes.cpp:31*
15: *Stack.cpp:147*
16: *opcodes.cpp:34*
17: *opcodes.cpp:31*
18: *opcodes.cpp:40*
19: *opcodes.cpp:44*
20: *opcodes.cpp:48*

```
54:                 return _iload_n[1](thread[2], 3);
55:         }
56:         InstructionResult[3] iload(Thread[4] * thread)
57:         {
58:                 u1[5] index = thread[6]->current_code[7][thread[6]->pc_register[8]++];
59:                 return _iload_n[1](thread[6], index[9]);
60:         }
61:
62:         inline static InstructionResult[3] _lload_n(Thread[4] * thread, u1[5] n)
63:         {
64:                 stack_frame[10] * frame = thread[11]->current_stack_frame[12];
65:                 su8[13] long_value = frame[14]->get_long[15](n[16]);
66:                 frame[14]->push_long[17](long_value[18]);
67:
68:                 return InstructionSuccess[19];
69:         }
70:
71:         InstructionResult[3] lload_0(Thread[4] * thread)
72:         {
73:                 return _lload_n[20](thread[21], 0);
74:         }
75:         InstructionResult[3] lload_1(Thread[4] * thread)
76:         {
77:                 return _lload_n[20](thread[22], 1);
78:         }
79:         InstructionResult[3] lload_2(Thread[4] * thread)
80:         {
81:                 return _lload_n[20](thread[23], 2);
82:         }
83:         InstructionResult[3] lload_3(Thread[4] * thread)
84:         {
85:                 return _lload_n[20](thread[24], 3);
86:         }
87:         InstructionResult[3] lload(Thread[4] * thread)
88:         {
89:                 u1[5] index = thread[25]->current_code[7][thread[25]->pc_register[8]++];
90:                 return _lload_n[20](thread[25], index[26]);
91:         }
92:
93:
94:         inline static InstructionResult[3] _fload_n(Thread[4] * thread, u1[5] n)
95:         {
96:                 stack_frame[10] * frame = thread[27]->current_stack_frame[12];
97:                 float float_value = frame[28]->get_float[29](n[30]);
98:                 frame[28]->push_float[31](float_value[32]);
99:
100:                return InstructionSuccess[19];
101:        }
102:
103:        InstructionResult[3] fload_0(Thread[4] * thread)
104:        {
105:                return _fload_n[33](thread[34], 0);
106:        }
```

Footnotes:

1: *opcodes.cpp:31*
2: *opcodes.cpp:52*
3: *defs.h:120*
4: *Thread.h:17*
5: *defs.h:168*
6: *opcodes.cpp:56*
7: *Thread.h:151*
8: *Thread.h:159*
9: *opcodes.cpp:58*
10: *Stack.h:129*
11: *opcodes.cpp:62*
12: *Thread.h:148*
13: *defs.h:177*
14: *opcodes.cpp:64*
15: *Stack.cpp:364*
16: *opcodes.cpp:62*
17: *Stack.cpp:153*
18: *opcodes.cpp:65*
19: *defs.h:122*
20: *opcodes.cpp:62*
21: *opcodes.cpp:71*
22: *opcodes.cpp:75*
23: *opcodes.cpp:79*
24: *opcodes.cpp:83*
25: *opcodes.cpp:87*
26: *opcodes.cpp:89*
27: *opcodes.cpp:94*
28: *opcodes.cpp:96*
29: *Stack.cpp:376*
30: *opcodes.cpp:94*
31: *Stack.cpp:165*
32: *opcodes.cpp:97*
33: *opcodes.cpp:94*
34: *opcodes.cpp:103*

```
107:        InstructionResult[1] fload_1(Thread[2] * thread)
108:        {
109:                return _fload_n[3](thread[4], 1);
110:        }
111:        InstructionResult[1] fload_2(Thread[2] * thread)
112:        {
113:                return _fload_n[3](thread[5], 2);
114:        }
115:        InstructionResult[1] fload_3(Thread[2] * thread)
116:        {
117:                return _fload_n[3](thread[6], 3);
118:        }
119:        InstructionResult[1] fload(Thread[2] * thread)
120:        {
121:                u1[7] index = thread[8]->current_code[9][thread[8]->pc_register[10]++];
122:                return _fload_n[3](thread[8], index[11]);
123:        }
124:
125:        inline static InstructionResult[1] _dload_n(Thread[2] * thread, u1[7] n)
126:        {
127:                stack_frame[12] * frame = thread[13]->current_stack_frame[14];
128:                double double_value = frame[15]->get_double[16](n[17]);
129:                frame[15]->push_double[18](double_value[19]);
130:
131:                return InstructionSuccess[20];
132:        }
133:
134:        InstructionResult[1] dload_0(Thread[2] * thread)
135:        {
136:                return _dload_n[21](thread[22], 0);
137:        }
138:        InstructionResult[1] dload_1(Thread[2] * thread)
139:        {
140:                return _dload_n[21](thread[23], 1);
141:        }
142:        InstructionResult[1] dload_2(Thread[2] * thread)
143:        {
144:                return _dload_n[21](thread[24], 2);
145:        }
146:        InstructionResult[1] dload_3(Thread[2] * thread)
147:        {
148:                return _dload_n[21](thread[25], 3);
149:        }
150:        InstructionResult[1] dload(Thread[2] * thread)
151:        {
152:                u1[7] index = thread[26]->current_code[9][thread[26]->pc_register[10]++];
153:                return _dload_n[21](thread[26], index[27]);
154:        }
155:
156:        inline static InstructionResult[1] _aload_n(Thread[2] * thread, u1[7] n)
157:        {
158:                stack_frame[12] * frame = thread[28]->current_stack_frame[14];
159:                word[29] reference_value = frame[30]->get_reference[31](n[32]);
```

Footnotes:

1: *defs.h:120*
2: *Thread.h:17*
3: *opcodes.cpp:94*
4: *opcodes.cpp:107*
5: *opcodes.cpp:111*
6: *opcodes.cpp:115*
7: *defs.h:168*
8: *opcodes.cpp:119*
9: *Thread.h:151*
10: *Thread.h:159*
11: *opcodes.cpp:121*
12: *Stack.h:129*
13: *opcodes.cpp:125*
14: *Thread.h:148*
15: *opcodes.cpp:127*
16: *Stack.cpp:383*
17: *opcodes.cpp:125*
18: *Stack.cpp:171*
19: *opcodes.cpp:128*
20: *defs.h:122*
21: *opcodes.cpp:125*
22: *opcodes.cpp:134*
23: *opcodes.cpp:138*
24: *opcodes.cpp:142*
25: *opcodes.cpp:146*
26: *opcodes.cpp:150*
27: *opcodes.cpp:152*
28: *opcodes.cpp:156*
29: *defs.h:195*
30: *opcodes.cpp:158*
31: *Stack.cpp:396*
32: *opcodes.cpp:156*

```
160:                frame[1]->push_reference[2](reference_value[3]);
161:
162:                return InstructionSuccess[4];
163:        }
164:
165:        InstructionResult[5] aload_0(Thread[6] * thread)
166:        {
167:                return _aload_n[7](thread[8], 0);
168:        }
169:        InstructionResult[5] aload_1(Thread[6] * thread)
170:        {
171:                return _aload_n[7](thread[9], 1);
172:        }
173:        InstructionResult[5] aload_2(Thread[6] * thread)
174:        {
175:                return _aload_n[7](thread[10], 2);
176:        }
177:        InstructionResult[5] aload_3(Thread[6] * thread)
178:        {
179:                return _aload_n[7](thread[11], 3);
180:        }
181:        InstructionResult[5] aload(Thread[6] * thread)
182:        {
183:                u1[12] index = thread[13]->current_code[14][thread[13]->pc_register[15]++];
184:                return _aload_n[7](thread[13], index[16]);
185:        }
186:
187:
188:        inline static InstructionResult[5] _iconst_n(Thread[6] * thread, signed short n)
189:        {
190:                thread[17]->current_stack_frame[18]->push_int[19](n[20]);
191:
192:                return InstructionSuccess[4];
193:        }
194:
195:        InstructionResult[5] iconst_0(Thread[6] * thread)
196:        {
197:                return _iconst_n[21](thread[22], 0);
198:        }
199:        InstructionResult[5] iconst_1(Thread[6] * thread)
200:        {
201:                return _iconst_n[21](thread[23], 1);
202:        }
203:        InstructionResult[5] iconst_2(Thread[6] * thread)
204:        {
205:                return _iconst_n[21](thread[24], 2);
206:        }
207:        InstructionResult[5] iconst_3(Thread[6] * thread)
208:        {
209:                return _iconst_n[21](thread[25], 3);
210:        }
211:        InstructionResult[5] iconst_4(Thread[6] * thread)
212:        {
```

Footnotes:

1: *opcodes.cpp:158*
2: *Stack.cpp:187*
3: *opcodes.cpp:159*
4: *defs.h:122*
5: *defs.h:120*
6: *Thread.h:17*
7: *opcodes.cpp:156*
8: *opcodes.cpp:165*
9: *opcodes.cpp:169*
10: *opcodes.cpp:173*
11: *opcodes.cpp:177*
12: *defs.h:168*
13: *opcodes.cpp:181*
14: *Thread.h:151*
15: *Thread.h:159*
16: *opcodes.cpp:183*
17: *opcodes.cpp:188*
18: *Thread.h:148*
19: *Stack.cpp:147*
20: *opcodes.cpp:188*
21: *opcodes.cpp:188*
22: *opcodes.cpp:195*
23: *opcodes.cpp:199*
24: *opcodes.cpp:203*
25: *opcodes.cpp:207*

```
213:                return _iconst_n[1](thread[2], 4);
214:        }
215:        InstructionResult[3] iconst_5(Thread[4] * thread)
216:        {
217:                return _iconst_n[1](thread[5], 5);
218:        }
219:        InstructionResult[3] iconst_m1(Thread[4] * thread)
220:        {
221:                return _iconst_n[1](thread[6], -1);
222:        }
223:
224:        inline static InstructionResult[3] _lconst_n(Thread[4] * thread, signed short n)
225:        {
226:                thread[7]->current_stack_frame[8]->push_long[9]((su4[10])n[11]);
227:
228:                return InstructionSuccess[12];
229:        }
230:
231:        InstructionResult[3] lconst_0(Thread[4] * thread)
232:        {
233:                return _lconst_n[13](thread[14], 0);
234:        }
235:        InstructionResult[3] lconst_1(Thread[4] * thread)
236:        {
237:                return _lconst_n[13](thread[15], 1);
238:        }
239:
240:
241:        inline static InstructionResult[3] _fconst_n(Thread[4] * thread, float n)
242:        {
243:                thread[16]->current_stack_frame[8]->push_float[17](n[18]);
244:
245:                return InstructionSuccess[12];
246:        }
247:
248:        InstructionResult[3] fconst_0(Thread[4] * thread)
249:        {
250:                return _fconst_n[19](thread[20], 0.0f);
251:        }
252:        InstructionResult[3] fconst_1(Thread[4] * thread)
253:        {
254:                return _fconst_n[19](thread[21], 1.0f);
255:        }
256:        InstructionResult[3] fconst_2(Thread[4] * thread)
257:        {
258:                return _fconst_n[19](thread[22], 2.0f);
259:        }
260:
261:        inline static InstructionResult[3] _dconst_n(Thread[4] * thread, double n)
262:        {
263:                thread[23]->current_stack_frame[8]->push_double[24](n[25]);
264:
265:                return InstructionSuccess[12];
```

Footnotes:

1: *opcodes.cpp:188*
2: *opcodes.cpp:211*
3: *defs.h:120*
4: *Thread.h:17*
5: *opcodes.cpp:215*
6: *opcodes.cpp:219*
7: *opcodes.cpp:224*
8: *Thread.h:148*
9: *Stack.cpp:153*
10: *defs.h:175*
11: *opcodes.cpp:224*
12: *defs.h:122*
13: *opcodes.cpp:224*
14: *opcodes.cpp:231*
15: *opcodes.cpp:235*
16: *opcodes.cpp:241*
17: *Stack.cpp:165*
18: *opcodes.cpp:241*
19: *opcodes.cpp:241*
20: *opcodes.cpp:248*
21: *opcodes.cpp:252*
22: *opcodes.cpp:256*
23: *opcodes.cpp:261*
24: *Stack.cpp:171*
25: *opcodes.cpp:261*

```
266:        }
267:
268:        InstructionResult[1] dconst_0(Thread[2] * thread)
269:        {
270:                return _dconst_n[3](thread[4], 0.0);
271:        }
272:        InstructionResult[1] dconst_1(Thread[2] * thread)
273:        {
274:                return _dconst_n[3](thread[5], 1.0);
275:        }
276:
277:        inline static InstructionResult[1] istore(Thread[2] * thread)
278:        {
279:                // Get the byte as the unsigned index of the local variable
280:                u1[6] index = thread[7]->current_code[8][thread[7]->pc_register[9]++];
281:
282:                stack_frame[10] * frame = thread[7]->current_stack_frame[11];
283:                su4[12] int_value = frame[13]->pop_int[14]();
284:                frame[13]->store_int[15](int_value[16], index[17]);
285:
286:                return InstructionSuccess[18];
287:        }
288:
289:        inline static InstructionResult[1] _istore_n(Thread[2] * thread, u1[6] n)
290:        {
291:                stack_frame[10] * frame = thread[19]->current_stack_frame[11];
292:                su4[12] int_value = frame[20]->pop_int[14]();
293:                frame[20]->store_int[15](int_value[21], n[22]);
294:
295:                return InstructionSuccess[18];
296:        }
297:
298:        InstructionResult[1] istore_0(Thread[2] * thread)
299:        {
300:                return _istore_n[23](thread[24], 0);
301:        }
302:        InstructionResult[1] istore_1(Thread[2] * thread)
303:        {
304:                return _istore_n[23](thread[25], 1);
305:        }
306:        InstructionResult[1] istore_2(Thread[2] * thread)
307:        {
308:                return _istore_n[23](thread[26], 2);
309:        }
310:        InstructionResult[1] istore_3(Thread[2] * thread)
311:        {
312:                return _istore_n[23](thread[27], 3);
313:        }
314:
315:
316:        inline static InstructionResult[1] lstore(Thread[2] * thread)
317:        {
318:                // Get the byte as the unsigned index of the local variable
```

Footnotes:

1: *defs.h:120*
2: *Thread.h:17*
3: *opcodes.cpp:261*
4: *opcodes.cpp:268*
5: *opcodes.cpp:272*
6: *defs.h:168*
7: *opcodes.cpp:277*
8: *Thread.h:151*
9: *Thread.h:159*
10: *Stack.h:129*
11: *Thread.h:148*
12: *defs.h:175*
13: *opcodes.cpp:282*
14: *Stack.cpp:206*
15: *Stack.cpp:305*
16: *opcodes.cpp:283*
17: *opcodes.cpp:280*
18: *defs.h:122*
19: *opcodes.cpp:289*
20: *opcodes.cpp:291*
21: *opcodes.cpp:292*
22: *opcodes.cpp:289*
23: *opcodes.cpp:289*
24: *opcodes.cpp:298*
25: *opcodes.cpp:302*
26: *opcodes.cpp:306*
27: *opcodes.cpp:310*

```
319:                    u1[1] index = thread[2]->current_code[3][thread[2]->pc_register[4]++];
320:
321:                    stack_frame[5] * frame = thread[2]->current_stack_frame[6];
322:                    su8[7] long_value = frame[8]->pop_long[9]();
323:                    frame[8]->store_long[10](long_value[11], index[12]);
324:
325:                    return InstructionSuccess[13];
326:            }
327:
328:            static InstructionResult[14] _lstore_n(Thread[15] * thread, u1[1] n)
329:            {
330:                    stack_frame[5] * frame = thread[16]->current_stack_frame[6];
331:                    su8[7] long_value = frame[17]->pop_long[9]();
332:                    frame[17]->store_long[10](long_value[18], n[19]);
333:
334:                    return InstructionSuccess[13];
335:            }
336:
337:            InstructionResult[14] lstore_0(Thread[15] * thread)
338:            {
339:                    return _lstore_n[20](thread[21], 0);
340:            }
341:            InstructionResult[14] lstore_1(Thread[15] * thread)
342:            {
343:                    return _lstore_n[20](thread[22], 1);
344:            }
345:            InstructionResult[14] lstore_2(Thread[15] * thread)
346:            {
347:                    return _lstore_n[20](thread[23], 2);
348:            }
349:            InstructionResult[14] lstore_3(Thread[15] * thread)
350:            {
351:                    return _lstore_n[20](thread[24], 3);
352:            }
353:
354:
355:            inline static InstructionResult[14] fstore(Thread[15] * thread)
356:            {
357:                    // Get the byte as the unsigned index of the local variable
358:                    u1[1] index = thread[25]->current_code[3][thread[25]->pc_register[4]++];
359:
360:                    stack_frame[5] * frame = thread[25]->current_stack_frame[6];
361:                    float float_value = frame[26]->pop_float[27]();
362:                    frame[26]->store_float[28](float_value[29], index[30]);
363:
364:                    return InstructionSuccess[13];
365:            }
366:
367:            inline static InstructionResult[14] _fstore_n(Thread[15] * thread, u1[1] n)
368:            {
369:                    stack_frame[5] * frame = thread[31]->current_stack_frame[6];
370:                    float float_value = frame[32]->pop_float[27]();
371:                    frame[32]->store_float[28](float_value[33], n[34]);
```

Footnotes:

1: *defs.h:168*
2: *opcodes.cpp:316*
3: *Thread.h:151*
4: *Thread.h:159*
5: *Stack.h:129*
6: *Thread.h:148*
7: *defs.h:177*
8: *opcodes.cpp:321*
9: *Stack.cpp:214*
10: *Stack.cpp:310*
11: *opcodes.cpp:322*
12: *opcodes.cpp:319*
13: *defs.h:122*
14: *defs.h:120*
15: *Thread.h:17*
16: *opcodes.cpp:328*
17: *opcodes.cpp:330*
18: *opcodes.cpp:331*
19: *opcodes.cpp:328*
20: *opcodes.cpp:328*
21: *opcodes.cpp:337*
22: *opcodes.cpp:341*
23: *opcodes.cpp:345*
24: *opcodes.cpp:349*
25: *opcodes.cpp:355*
26: *opcodes.cpp:360*
27: *Stack.cpp:227*
28: *Stack.cpp:320*
29: *opcodes.cpp:361*
30: *opcodes.cpp:358*
31: *opcodes.cpp:367*
32: *opcodes.cpp:369*
33: *opcodes.cpp:370*
34: *opcodes.cpp:367*

```
372:
373:            return InstructionSuccess[1];
374:    }
375:    InstructionResult[2] fstore_0(Thread[3] * thread)
376:    {
377:            return _fstore_n[4](thread[5], 0);
378:    }
379:    InstructionResult[2] fstore_1(Thread[3] * thread)
380:    {
381:            return _fstore_n[4](thread[6], 1);
382:    }
383:    InstructionResult[2] fstore_2(Thread[3] * thread)
384:    {
385:            return _fstore_n[4](thread[7], 2);
386:    }
387:    InstructionResult[2] fstore_3(Thread[3] * thread)
388:    {
389:            return _fstore_n[4](thread[8], 3);
390:    }
391:
392:    inline static InstructionResult[2] dstore(Thread[3] * thread)
393:    {
394:            // Get the byte as the unsigned index of the local variable
395:            u1[9] index = thread[10]->current_code[11][thread[10]->pc_register[12]++];
396:
397:            stack_frame[13] * frame = thread[10]->current_stack_frame[14];
398:            double double_value = frame[15]->pop_double[16]();
399:            frame[15]->store_double[17](double_value[18], index[19]);
400:
401:            return InstructionSuccess[1];
402:    }
403:
404:    inline static InstructionResult[2] _dstore_n(Thread[3] * thread, u1[9] n)
405:    {
406:            stack_frame[13] * frame = thread[20]->current_stack_frame[14];
407:            double double_value = frame[21]->pop_double[16]();
408:            frame[21]->store_double[17](double_value[22], n[23]);
409:
410:            return InstructionSuccess[1];
411:    }
412:    InstructionResult[2] dstore_0(Thread[3] * thread)
413:    {
414:            return _dstore_n[24](thread[25], 0);
415:    }
416:    InstructionResult[2] dstore_1(Thread[3] * thread)
417:    {
418:            return _dstore_n[24](thread[26], 1);
419:    }
420:    InstructionResult[2] dstore_2(Thread[3] * thread)
421:    {
422:            return _dstore_n[24](thread[27], 2);
423:    }
424:    InstructionResult[2] dstore_3(Thread[3] * thread)
```

Footnotes:

1: *defs.h:122*
2: *defs.h:120*
3: *Thread.h:17*
4: *opcodes.cpp:367*
5: *opcodes.cpp:375*
6: *opcodes.cpp:379*
7: *opcodes.cpp:383*
8: *opcodes.cpp:387*
9: *defs.h:168*
10: *opcodes.cpp:392*
11: *Thread.h:151*
12: *Thread.h:159*
13: *Stack.h:129*
14: *Thread.h:148*
15: *opcodes.cpp:397*
16: *Stack.cpp:235*
17: *Stack.cpp:325*
18: *opcodes.cpp:398*
19: *opcodes.cpp:395*
20: *opcodes.cpp:404*
21: *opcodes.cpp:406*
22: *opcodes.cpp:407*
23: *opcodes.cpp:404*
24: *opcodes.cpp:404*
25: *opcodes.cpp:412*
26: *opcodes.cpp:416*
27: *opcodes.cpp:420*

```
425:        {
426:                return _dstore_n[1](thread[2], 3);
427:        }
428:
429:        inline static InstructionResult[3] astore(Thread[4] * thread)
430:        {
431:                // Get the byte as the unsigned index of the local variable
432:                u1[5] index = thread[6]->current_code[7][thread[6]->pc_register[8]++];
433:
434:                stack_frame[9] * frame = thread[6]->current_stack_frame[10];
435:                word[11] ref_value = frame[12]->pop_reference[13]();
436:                frame[12]->store_reference[14](ref_value[15], index[16]);
437:
438:                return InstructionSuccess[17];
439:        }
440:
441:        inline static InstructionResult[3] _astore_n(Thread[4] * thread, u1[5] n)
442:        {
443:                stack_frame[9] * frame = thread[18]->current_stack_frame[10];
444:                word[11] reference_value = frame[19]->pop_reference[13]();
445:                frame[19]->store_reference[14](reference_value[20], n[21]);
446:
447:                return InstructionSuccess[17];
448:        }
449:        InstructionResult[3] astore_0(Thread[4] * thread)
450:        {
451:                return _astore_n[22](thread[23], 0);
452:        }
453:        InstructionResult[3] astore_1(Thread[4] * thread)
454:        {
455:                return _astore_n[22](thread[24], 1);
456:        }
457:        InstructionResult[3] astore_2(Thread[4] * thread)
458:        {
459:                return _astore_n[22](thread[25], 2);
460:        }
461:        InstructionResult[3] astore_3(Thread[4] * thread)
462:        {
463:                return _astore_n[22](thread[26], 3);
464:        }
465:
466:        InstructionResult[3] bipush(Thread[4] * thread)
467:        {
468:                // Get the byte as the signed one and push it as a signed integer
469:                su1[27] byte = thread[28]->current_code[7][thread[28]->pc_register[8]++];
470:                thread[28]->current_stack_frame[10]->push_int[29]((su4[30])byte[31]);
471:                return InstructionSuccess[17];
472:        }
473:
474:        InstructionResult[3] sipush(Thread[4] * thread)
475:        {
476:                // Get the byte as the signed one and push it as a signed integer
477:                u1[5] byte1 = thread[32]->current_code[7][thread[32]->pc_register[8]++];
```

Footnotes:

1: *opcodes.cpp:404*
2: *opcodes.cpp:424*
3: *defs.h:120*
4: *Thread.h:17*
5: *defs.h:168*
6: *opcodes.cpp:429*
7: *Thread.h:151*
8: *Thread.h:159*
9: *Stack.h:129*
10: *Thread.h:148*
11: *defs.h:195*
12: *opcodes.cpp:434*
13: *Stack.cpp:249*
14: *Stack.cpp:338*
15: *opcodes.cpp:435*
16: *opcodes.cpp:432*
17: *defs.h:122*
18: *opcodes.cpp:441*
19: *opcodes.cpp:443*
20: *opcodes.cpp:444*
21: *opcodes.cpp:441*
22: *opcodes.cpp:441*
23: *opcodes.cpp:449*
24: *opcodes.cpp:453*
25: *opcodes.cpp:457*
26: *opcodes.cpp:461*
27: *defs.h:169*
28: *opcodes.cpp:466*
29: *Stack.cpp:147*
30: *defs.h:175*
31: *opcodes.cpp:469*
32: *opcodes.cpp:474*

```
478:                u1[1] byte2 = thread[2]->current_code[3][thread[2]->pc_register[4]++];
479:                su2[5] short_value = (byte1[6] << 8) | byte2[7];
480:                thread[2]->current_stack_frame[8]->push_int[9]((su4[10])short_value[11]);
481:                return InstructionSuccess[12];
482:        }
483:
484:        //-----------------------------------------------------------------------------
485:        // Arithmetic operations
486:        //-----------------------------------------------------------------------------
487:
488:        InstructionResult[13] imul(Thread[14] * thread)
489:        {
490:                stack_frame[15] * frame = thread[16]->current_stack_frame[8];
491:                su4[10] value2 = frame[17]->pop_int[18]();
492:                su4[10] value1 = frame[17]->pop_int[18]();
493:                frame[17]->push_int[9](value1[19] * value2[20]);
494:
495:                // According to JVM spec: despite the fact that overflow may occur,
496:                // execution of an imul instruction never throws a runtime exception.
497:                return InstructionSuccess[12];
498:        }
499:
500:        InstructionResult[13] iadd(Thread[14] * thread)
501:        {
502:                stack_frame[15] * frame = thread[21]->current_stack_frame[8];
503:                su4[10] value2 = frame[22]->pop_int[18]();
504:                su4[10] value1 = frame[22]->pop_int[18]();
505:                frame[22]->push_int[9](value1[23] + value2[24]);
506:
507:                // According to JVM spec: despite the fact that overflow may occur,
508:                // execution of an iadd instruction never throws a runtime exception.
509:                return InstructionSuccess[12];
510:        }
511:
512:        InstructionResult[13] isub(Thread[14] * thread)
513:        {
514:                stack_frame[15] * frame = thread[25]->current_stack_frame[8];
515:                su4[10] value2 = frame[26]->pop_int[18]();
516:                su4[10] value1 = frame[26]->pop_int[18]();
517:                frame[26]->push_int[9](value1[27] - value2[28]);
518:
519:                // According to JVM spec: despite the fact that overflow may occur,
520:                // execution of an iadd instruction never throws a runtime exception.
521:                return InstructionSuccess[12];
522:        }
523:
524:        InstructionResult[13] idiv(Thread[14] * thread)
525:        {
526:                stack_frame[15] * frame = thread[29]->current_stack_frame[8];
527:                su4[10] value2 = frame[30]->pop_int[18]();
528:                su4[10] value1 = frame[30]->pop_int[18]();
529:                if (value2[31] == 0)
530:                {
```

Footnotes:

1: *defs.h:168*
2: *opcodes.cpp:474*
3: *Thread.h:151*
4: *Thread.h:159*
5: *defs.h:173*
6: *opcodes.cpp:477*
7: *opcodes.cpp:478*
8: *Thread.h:148*
9: *Stack.cpp:147*
10: *defs.h:175*
11: *opcodes.cpp:479*
12: *defs.h:122*
13: *defs.h:120*
14: *Thread.h:17*
15: *Stack.h:129*
16: *opcodes.cpp:488*
17: *opcodes.cpp:490*
18: *Stack.cpp:206*
19: *opcodes.cpp:492*
20: *opcodes.cpp:491*
21: *opcodes.cpp:500*
22: *opcodes.cpp:502*
23: *opcodes.cpp:504*
24: *opcodes.cpp:503*
25: *opcodes.cpp:512*
26: *opcodes.cpp:514*
27: *opcodes.cpp:516*
28: *opcodes.cpp:515*
29: *opcodes.cpp:524*
30: *opcodes.cpp:526*
31: *opcodes.cpp:527*

```
531:                        thread[1]->raise_exception[2](VM_ERROR_ArithmeticException[3]);
532:                        return InstructionErrorDivisionByZero[4];
533:                }
534:                frame[5]->push_int[6](value1[7] / value2[8]);
535:
536:                // According to JVM spec: despite the fact that overflow may occur,
537:                // execution of an iadd instruction never throws a runtime exception.
538:                return InstructionSuccess[9];
539:        }
540:
541:        InstructionResult[10] irem(Thread[11] * thread)
542:        {
543:                stack_frame[12] * frame = thread[13]->current_stack_frame[14];
544:                su4[15] value2 = frame[16]->pop_int[17]();
545:                su4[15] value1 = frame[16]->pop_int[17]();
546:                if (value2[18] == 0)
547:                {
548:                        thread[13]->raise_exception[2](VM_ERROR_ArithmeticException[3]);
549:                        return InstructionErrorDivisionByZero[4];
550:                }
551:                su4[15] result = value1[19] - (value1[19]/value2[18])*value2[18];
552:                frame[16]->push_int[6](result[20]);
553:
554:                return InstructionSuccess[9];
555:        }
556:
557:        InstructionResult[10] iushr(Thread[11] * thread)
558:        {
559:                stack_frame[12] * frame = thread[21]->current_stack_frame[14];
560:                su4[15] value2 = frame[22]->pop_int[17]();
561:                su4[15] value1 = frame[22]->pop_int[17]();
562:                su4[15] res;
563:                u4[23] s = value2[24] & 0x1F; // low 5 bits of value2
564:                if (value1[25] > 0)
565:                        res[26] = value1[25] >> s[27]; // shift right with zero extension
566:                else {
567:                        // to compensate sign extension: (2 << ~s) cancels the propagation of sign bit
568:                        su4[15] not_s = ~s[27];
569:                        u4[23] compensation;
570:                        if (not_s[28] < 0)
571:                                compensation[29] = 0;
572:                        else
573:                                compensation[29] = 2 << not_s[28];
574:                        su4[15] value = value1[25] >> s[27];
575:                        res[26] = value[30] + compensation[29];
576:                }
577:                frame[22]->push_int[6](res[26]);
578:
579:                return InstructionSuccess[9];
580:        }
581:
582:        InstructionResult[10] ishr(Thread[11] * thread)
583:        {
```

Footnotes:

1: *opcodes.cpp:524*
2: *Thread.cpp:776*
3: *defs.h:112*
4: *defs.h:124*
5: *opcodes.cpp:526*
6: *Stack.cpp:147*
7: *opcodes.cpp:528*
8: *opcodes.cpp:527*
9: *defs.h:122*
10: *defs.h:120*
11: *Thread.h:17*
12: *Stack.h:129*
13: *opcodes.cpp:541*
14: *Thread.h:148*
15: *defs.h:175*
16: *opcodes.cpp:543*
17: *Stack.cpp:206*
18: *opcodes.cpp:544*
19: *opcodes.cpp:545*
20: *opcodes.cpp:551*
21: *opcodes.cpp:557*
22: *opcodes.cpp:559*
23: *defs.h:174*
24: *opcodes.cpp:560*
25: *opcodes.cpp:561*
26: *opcodes.cpp:562*
27: *opcodes.cpp:563*
28: *opcodes.cpp:568*
29: *opcodes.cpp:569*
30: *opcodes.cpp:574*

```
584:                    stack_frame[1] * frame = thread[2]->current_stack_frame[3];
585:                    su4[4] value2 = frame[5]->pop_int[6]();
586:                    su4[4] value1 = frame[5]->pop_int[6]();
587:                    su4[4] s = value2[7] & 0x1F; // low 5 bits of value2
588:                    su4[4] res = value1[8] >> s[9]; // shift right with sign extension
589:                    frame[5]->push_int[10](res[11]);
590:
591:                    return InstructionSuccess[12];
592:            }
593:
594:            InstructionResult[13] ishl(Thread[14] * thread)
595:            {
596:                    stack_frame[1] * frame = thread[15]->current_stack_frame[3];
597:                    su4[4] value2 = frame[16]->pop_int[6]();
598:                    su4[4] value1 = frame[16]->pop_int[6]();
599:                    su4[4] s = value2[17] & 0x1F; // low 5 bits of value2
600:                    su4[4] res = value1[18] << s[19]; // shift left
601:                    frame[16]->push_int[10](res[20]);
602:
603:                    return InstructionSuccess[12];
604:            }
605:
606:
607:            InstructionResult[13] ixor(Thread[14] * thread)
608:            {
609:                    stack_frame[1] * frame = thread[21]->current_stack_frame[3];
610:                    su4[4] value2 = frame[22]->pop_int[6]();
611:                    su4[4] value1 = frame[22]->pop_int[6]();
612:                    su4[4] res = value1[23] ^ value2[24];
613:                    frame[22]->push_int[10](res[25]);
614:
615:                    return InstructionSuccess[12];
616:            }
617:
618:            InstructionResult[13] iand(Thread[14] * thread)
619:            {
620:                    stack_frame[1] * frame = thread[26]->current_stack_frame[3];
621:                    su4[4] value2 = frame[27]->pop_int[6]();
622:                    su4[4] value1 = frame[27]->pop_int[6]();
623:                    su4[4] res = value1[28] & value2[29];
624:                    frame[27]->push_int[10](res[30]);
625:
626:                    return InstructionSuccess[12];
627:            }
628:
629:            InstructionResult[13] ior(Thread[14] * thread)
630:            {
631:                    stack_frame[1] * frame = thread[31]->current_stack_frame[3];
632:                    su4[4] value2 = frame[32]->pop_int[6]();
633:                    su4[4] value1 = frame[32]->pop_int[6]();
634:                    su4[4] res = value1[33] | value2[34];
635:                    frame[32]->push_int[10](res[35]);
636:
```

Footnotes:

1: *Stack.h:129*
2: *opcodes.cpp:582*
3: *Thread.h:148*
4: *defs.h:175*
5: *opcodes.cpp:584*
6: *Stack.cpp:206*
7: *opcodes.cpp:585*
8: *opcodes.cpp:586*
9: *opcodes.cpp:587*
10: *Stack.cpp:147*
11: *opcodes.cpp:588*
12: *defs.h:122*
13: *defs.h:120*
14: *Thread.h:17*
15: *opcodes.cpp:594*
16: *opcodes.cpp:596*
17: *opcodes.cpp:597*
18: *opcodes.cpp:598*
19: *opcodes.cpp:599*
20: *opcodes.cpp:600*
21: *opcodes.cpp:607*
22: *opcodes.cpp:609*
23: *opcodes.cpp:611*
24: *opcodes.cpp:610*
25: *opcodes.cpp:612*
26: *opcodes.cpp:618*
27: *opcodes.cpp:620*
28: *opcodes.cpp:622*
29: *opcodes.cpp:621*
30: *opcodes.cpp:623*
31: *opcodes.cpp:629*
32: *opcodes.cpp:631*
33: *opcodes.cpp:633*
34: *opcodes.cpp:632*
35: *opcodes.cpp:634*

```
637:                return InstructionSuccess[1];
638:        }
639:
640:        InstructionResult[2] ineg(Thread[3] * thread)
641:        {
642:                stack_frame[4] * frame = thread[5]->current_stack_frame[6];
643:                su4[7] value = frame[8]->pop_int[9]();
644:                frame[8]->push_int[10](-value[11]);
645:
646:                return InstructionSuccess[1];
647:        }
648:
649:        InstructionResult[2] lmul(Thread[3] * thread)
650:        {
651:                stack_frame[4] * frame = thread[12]->current_stack_frame[6];
652:                su8[13] value2 = frame[14]->pop_long[15]();
653:                su8[13] value1 = frame[14]->pop_long[15]();
654:                frame[14]->push_long[16](value1[17] * value2[18]);
655:
656:                // According to JVM spec: despite the fact that overflow may occur,
657:                // execution of an imul instruction never throws a runtime exception.
658:                return InstructionSuccess[1];
659:        }
660:
661:        InstructionResult[2] ladd(Thread[3] * thread)
662:        {
663:                stack_frame[4] * frame = thread[19]->current_stack_frame[6];
664:                su8[13] value2 = frame[20]->pop_long[15]();
665:                su8[13] value1 = frame[20]->pop_long[15]();
666:                frame[20]->push_long[16](value1[21] + value2[22]);
667:
668:                // According to JVM spec: despite the fact that overflow may occur,
669:                // execution of an iadd instruction never throws a runtime exception.
670:                return InstructionSuccess[1];
671:        }
672:
673:        InstructionResult[2] lsub(Thread[3] * thread)
674:        {
675:                stack_frame[4] * frame = thread[23]->current_stack_frame[6];
676:                su8[13] value2 = frame[24]->pop_long[15]();
677:                su8[13] value1 = frame[24]->pop_long[15]();
678:                frame[24]->push_long[16](value1[25] - value2[26]);
679:
680:                // According to JVM spec: despite the fact that overflow may occur,
681:                // execution of an iadd instruction never throws a runtime exception.
682:                return InstructionSuccess[1];
683:        }
684:
685:        InstructionResult[2] ldiv(Thread[3] * thread)
686:        {
687:                stack_frame[4] * frame = thread[27]->current_stack_frame[6];
688:                su8[13] value2 = frame[28]->pop_long[15]();
689:                su8[13] value1 = frame[28]->pop_long[15]();
```

Footnotes:

1: *defs.h:122*
2: *defs.h:120*
3: *Thread.h:17*
4: *Stack.h:129*
5: *opcodes.cpp:640*
6: *Thread.h:148*
7: *defs.h:175*
8: *opcodes.cpp:642*
9: *Stack.cpp:206*
10: *Stack.cpp:147*
11: *opcodes.cpp:643*
12: *opcodes.cpp:649*
13: *defs.h:177*
14: *opcodes.cpp:651*
15: *Stack.cpp:214*
16: *Stack.cpp:153*
17: *opcodes.cpp:653*
18: *opcodes.cpp:652*
19: *opcodes.cpp:661*
20: *opcodes.cpp:663*
21: *opcodes.cpp:665*
22: *opcodes.cpp:664*
23: *opcodes.cpp:673*
24: *opcodes.cpp:675*
25: *opcodes.cpp:677*
26: *opcodes.cpp:676*
27: *opcodes.cpp:685*
28: *opcodes.cpp:687*

```
690:                    if (value2[1] == 0)
691:                    {
692:                            thread[2]->raise_exception[3](VM_ERROR_ArithmeticException[4]);
693:                            return InstructionErrorDivisionByZero[5];
694:                    }
695:                    frame[6]->push_long[7](value1[8] / value2[1]);
696:
697:                    // According to JVM spec: despite the fact that overflow may occur,
698:                    // execution of an iadd instruction never throws a runtime exception.
699:                    return InstructionSuccess[9];
700:            }
701:
702:            InstructionResult[10] lrem(Thread[11] * thread)
703:            {
704:                    stack_frame[12] * frame = thread[13]->current_stack_frame[14];
705:                    su8[15] value2 = frame[16]->pop_long[17]();
706:                    su8[15] value1 = frame[16]->pop_long[17]();
707:                    if (value2[18] == 0)
708:                    {
709:                            thread[13]->raise_exception[3](VM_ERROR_ArithmeticException[4]);
710:                            return InstructionErrorDivisionByZero[5];
711:                    }
712:                    su8[15] result = value1[19] - (value1[19]/value2[18])*value2[18];
713:                    frame[16]->push_long[7](result[20]);
714:
715:                    return InstructionSuccess[9];
716:            }
717:
718:            InstructionResult[10] lxor(Thread[11] * thread)
719:            {
720:                    stack_frame[12] * frame = thread[21]->current_stack_frame[14];
721:                    su8[15] value2 = frame[22]->pop_long[17]();
722:                    su8[15] value1 = frame[22]->pop_long[17]();
723:                    su8[15] res = value1[23] ^ value2[24];
724:                    frame[22]->push_long[7](res[25]);
725:
726:                    return InstructionSuccess[9];
727:            }
728:
729:            InstructionResult[10] land(Thread[11] * thread)
730:            {
731:                    stack_frame[12] * frame = thread[26]->current_stack_frame[14];
732:                    su8[15] value2 = frame[27]->pop_long[17]();
733:                    su8[15] value1 = frame[27]->pop_long[17]();
734:                    su8[15] res = value1[28] & value2[29];
735:                    frame[27]->push_long[7](res[30]);
736:
737:                    return InstructionSuccess[9];
738:            }
739:
740:            InstructionResult[10] lor(Thread[11] * thread)
741:            {
742:                    stack_frame[12] * frame = thread[31]->current_stack_frame[14];
```

Footnotes:

1: *opcodes.cpp:688*
2: *opcodes.cpp:685*
3: *Thread.cpp:776*
4: *defs.h:112*
5: *defs.h:124*
6: *opcodes.cpp:687*
7: *Stack.cpp:153*
8: *opcodes.cpp:689*
9: *defs.h:122*
10: *defs.h:120*
11: *Thread.h:17*
12: *Stack.h:129*
13: *opcodes.cpp:702*
14: *Thread.h:148*
15: *defs.h:177*
16: *opcodes.cpp:704*
17: *Stack.cpp:214*
18: *opcodes.cpp:705*
19: *opcodes.cpp:706*
20: *opcodes.cpp:712*
21: *opcodes.cpp:718*
22: *opcodes.cpp:720*
23: *opcodes.cpp:722*
24: *opcodes.cpp:721*
25: *opcodes.cpp:723*
26: *opcodes.cpp:729*
27: *opcodes.cpp:731*
28: *opcodes.cpp:733*
29: *opcodes.cpp:732*
30: *opcodes.cpp:734*
31: *opcodes.cpp:740*

```
743:                    su8[1] value2 = frame[2]->pop_long[3]();
744:                    su8[1] value1 = frame[2]->pop_long[3]();
745:                    su8[1] res = value1[4] | value2[5];
746:                    frame[2]->push_long[6](res[7]);
747:
748:                    return InstructionSuccess[8];
749:            }
750:
751:            InstructionResult[9] lneg(Thread[10] * thread)
752:            {
753:                    stack_frame[11] * frame = thread[12]->current_stack_frame[13];
754:                    su8[1] value = frame[14]->pop_long[3]();
755:                    frame[14]->push_long[6](-value[15]);
756:
757:                    return InstructionSuccess[8];
758:            }
759:
760:
761:            InstructionResult[9] fmul(Thread[10] * thread)
762:            {
763:                    stack_frame[11] * frame = thread[16]->current_stack_frame[13];
764:                    float value2 = frame[17]->pop_float[18]();
765:                    float value1 = frame[17]->pop_float[18]();
766:                    frame[17]->push_float[19](value1[20] * value2[21]);
767:
768:                    // According to JVM spec: despite the fact that overflow may occur,
769:                    // execution of an imul instruction never throws a runtime exception.
770:                    return InstructionSuccess[8];
771:            }
772:
773:            InstructionResult[9] fadd(Thread[10] * thread)
774:            {
775:                    stack_frame[11] * frame = thread[22]->current_stack_frame[13];
776:                    float value2 = frame[23]->pop_float[18]();
777:                    float value1 = frame[23]->pop_float[18]();
778:                    frame[23]->push_float[19](value1[24] + value2[25]);
779:
780:                    // According to JVM spec: despite the fact that overflow may occur,
781:                    // execution of an iadd instruction never throws a runtime exception.
782:                    return InstructionSuccess[8];
783:            }
784:
785:            InstructionResult[9] fsub(Thread[10] * thread)
786:            {
787:                    stack_frame[11] * frame = thread[26]->current_stack_frame[13];
788:                    float value2 = frame[27]->pop_float[18]();
789:                    float value1 = frame[27]->pop_float[18]();
790:                    frame[27]->push_float[19](value1[28] - value2[29]);
791:
792:                    // According to JVM spec: despite the fact that overflow may occur,
793:                    // execution of an iadd instruction never throws a runtime exception.
794:                    return InstructionSuccess[8];
795:            }
```

Footnotes:

1: *defs.h:177*
2: *opcodes.cpp:742*
3: *Stack.cpp:214*
4: *opcodes.cpp:744*
5: *opcodes.cpp:743*
6: *Stack.cpp:153*
7: *opcodes.cpp:745*
8: *defs.h:122*
9: *defs.h:120*
10: *Thread.h:17*
11: *Stack.h:129*
12: *opcodes.cpp:751*
13: *Thread.h:148*
14: *opcodes.cpp:753*
15: *opcodes.cpp:754*
16: *opcodes.cpp:761*
17: *opcodes.cpp:763*
18: *Stack.cpp:227*
19: *Stack.cpp:165*
20: *opcodes.cpp:765*
21: *opcodes.cpp:764*
22: *opcodes.cpp:773*
23: *opcodes.cpp:775*
24: *opcodes.cpp:777*
25: *opcodes.cpp:776*
26: *opcodes.cpp:785*
27: *opcodes.cpp:787*
28: *opcodes.cpp:789*
29: *opcodes.cpp:788*

```
796:
797:        InstructionResult[1] fdiv(Thread[2] * thread)
798:        {
799:                stack_frame[3] * frame = thread[4]->current_stack_frame[5];
800:                float value2 = frame[6]->pop_float[7]();
801:                float value1 = frame[6]->pop_float[7]();
802:                if (value2[8] == 0)
803:                {
804:                        thread[4]->raise_exception[9](VM_ERROR_ArithmeticException[10]);
805:                        return InstructionErrorDivisionByZero[11];
806:                }
807:                frame[6]->push_float[12](value1[13] / value2[8]);
808:
809:                // According to JVM spec: despite the fact that overflow may occur,
810:                // execution of an iadd instruction never throws a runtime exception.
811:                return InstructionSuccess[14];
812:        }
813:
814:        InstructionResult[1] dmul(Thread[2] * thread)
815:        {
816:                stack_frame[3] * frame = thread[15]->current_stack_frame[5];
817:                double value2 = frame[16]->pop_double[17]();
818:                double value1 = frame[16]->pop_double[17]();
819:                frame[16]->push_double[18](value1[19] * value2[20]);
820:
821:                // According to JVM spec: despite the fact that overflow may occur,
822:                // execution of an imul instruction never throws a runtime exception.
823:                return InstructionSuccess[14];
824:        }
825:
826:        InstructionResult[1] dadd(Thread[2] * thread)
827:        {
828:                stack_frame[3] * frame = thread[21]->current_stack_frame[5];
829:                double value2 = frame[22]->pop_double[17]();
830:                double value1 = frame[22]->pop_double[17]();
831:                frame[22]->push_double[18](value1[23] + value2[24]);
832:
833:                // According to JVM spec: despite the fact that overflow may occur,
834:                // execution of an iadd instruction never throws a runtime exception.
835:                return InstructionSuccess[14];
836:        }
837:
838:        InstructionResult[1] dsub(Thread[2] * thread)
839:        {
840:                stack_frame[3] * frame = thread[25]->current_stack_frame[5];
841:                double value2 = frame[26]->pop_double[17]();
842:                double value1 = frame[26]->pop_double[17]();
843:                frame[26]->push_double[18](value1[27] - value2[28]);
844:
845:                // According to JVM spec: despite the fact that overflow may occur,
846:                // execution of an iadd instruction never throws a runtime exception.
847:                return InstructionSuccess[14];
848:        }
```

Footnotes:

1: *defs.h:120*
2: *Thread.h:17*
3: *Stack.h:129*
4: *opcodes.cpp:797*
5: *Thread.h:148*
6: *opcodes.cpp:799*
7: *Stack.cpp:227*
8: *opcodes.cpp:800*
9: *Thread.cpp:776*
10: *defs.h:112*
11: *defs.h:124*
12: *Stack.cpp:165*
13: *opcodes.cpp:801*
14: *defs.h:122*
15: *opcodes.cpp:814*
16: *opcodes.cpp:816*
17: *Stack.cpp:235*
18: *Stack.cpp:171*
19: *opcodes.cpp:818*
20: *opcodes.cpp:817*
21: *opcodes.cpp:826*
22: *opcodes.cpp:828*
23: *opcodes.cpp:830*
24: *opcodes.cpp:829*
25: *opcodes.cpp:838*
26: *opcodes.cpp:840*
27: *opcodes.cpp:842*
28: *opcodes.cpp:841*

```
849:
850:        InstructionResult[1] ddiv(Thread[2] * thread)
851:        {
852:                stack_frame[3] * frame = thread[4]->current_stack_frame[5];
853:                double value2 = frame[6]->pop_double[7]();
854:                double value1 = frame[6]->pop_double[7]();
855:                if (value2[8] == 0)
856:                {
857:                        thread[4]->raise_exception[9](VM_ERROR_ArithmeticException[10]);
858:                        return InstructionErrorDivisionByZero[11];
859:                }
860:                frame[6]->push_double[12](value1[13] / value2[8]);
861:
862:                // According to JVM spec: despite the fact that overflow may occur,
863:                // execution of an iadd instruction never throws a runtime exception.
864:                return InstructionSuccess[14];
865:        }
866:
867:        InstructionResult[1] iinc(Thread[2] * thread)
868:        {
869:                stack_frame[3] * frame = thread[15]->current_stack_frame[5];
870:                // Get the index of the local variable
871:                u1[16] index = thread[15]->current_code[17][thread[15]->pc_register[18]++];
872:                // Get the signed byte constant
873:                su1[19] constant = thread[15]->current_code[17][thread[15]->pc_register[18]++];
874:                // Sign-extend to an integer
875:                su4[20] int_constant = (su4[20])constant[21];
876:                // Get the local variable
877:                su4[20] var = frame[22]->get_int[23](index[24]);
878:                // Store the incremented value
879:                frame[22]->store_int[25](var[26]+int_constant[27], index[24]);
880:
881:                return InstructionSuccess[14];
882:        }
883:
884:        //-----------------------------------------------------------------------------
885:        // Stack operations
886:        //-----------------------------------------------------------------------------
887:
888:        InstructionResult[1] pop(Thread[2] * thread)
889:        {
890:                thread[28]->current_stack_frame[5]->pop_word[29]();
891:                return InstructionSuccess[14];
892:        }
893:
894:        InstructionResult[1] pop2(Thread[2] * thread)
895:        {
896:                thread[30]->current_stack_frame[5]->pop2_word[31]();
897:                return InstructionSuccess[14];
898:        }
899:
900:        InstructionResult[1] dup(Thread[2] * thread)
901:        {
```

Footnotes:

1: *defs.h:120*
2: *Thread.h:17*
3: *Stack.h:129*
4: *opcodes.cpp:850*
5: *Thread.h:148*
6: *opcodes.cpp:852*
7: *Stack.cpp:235*
8: *opcodes.cpp:853*
9: *Thread.cpp:776*
10: *defs.h:112*
11: *defs.h:124*
12: *Stack.cpp:171*
13: *opcodes.cpp:854*
14: *defs.h:122*
15: *opcodes.cpp:867*
16: *defs.h:168*
17: *Thread.h:151*
18: *Thread.h:159*
19: *defs.h:169*
20: *defs.h:175*
21: *opcodes.cpp:873*
22: *opcodes.cpp:869*
23: *Stack.cpp:357*
24: *opcodes.cpp:871*
25: *Stack.cpp:305*
26: *opcodes.cpp:877*
27: *opcodes.cpp:875*
28: *opcodes.cpp:888*
29: *Stack.cpp:259*
30: *opcodes.cpp:894*
31: *Stack.cpp:264*

```
902:                thread[1]->current_stack_frame[2]->dup_word[3]();
903:                return InstructionSuccess[4];
904:        }
905:
906:        InstructionResult[5] dup2(Thread[6] * thread)
907:        {
908:                thread[7]->current_stack_frame[2]->dup2_word[8]();
909:                return InstructionSuccess[4];
910:        }
911:
912:        InstructionResult[5] swap(Thread[6] * thread)
913:        {
914:                thread[9]->current_stack_frame[2]->swap_word[10]();
915:                return InstructionSuccess[4];
916:        }
917:
918:        InstructionResult[5] dup_x1(Thread[6] * thread)
919:        {
920:                thread[11]->current_stack_frame[2]->dup_x1[12]();
921:                return InstructionSuccess[4];
922:        }
923:
924:        //-----------------------------------------------------------------------------
925:        // Conversion operations
926:        //-----------------------------------------------------------------------------
927:
928:        InstructionResult[5] i2f(Thread[6] * thread)
929:        {
930:                stack_frame[13] * frame = thread[14]->current_stack_frame[2];
931:                su4[15] int_value = frame[16]->pop_int[17]();
932:                // Convert to float
933:                float float_value = (float)int_value[18];
934:                frame[16]->push_float[19](float_value[20]);
935:
936:                return InstructionSuccess[4];
937:        }
938:
939:        InstructionResult[5] i2b(Thread[6] * thread)
940:        {
941:                stack_frame[13] * frame = thread[21]->current_stack_frame[2];
942:                su4[15] int_value = frame[22]->pop_int[17]();
943:                // Truncate to byte
944:                u1[23] byte = (u1[23])int_value[24];
945:                // Sign-extend to int
946:                int_value[24] = (su4[15])byte[25];
947:                frame[22]->push_int[26](int_value[24]);
948:
949:                return InstructionSuccess[4];
950:        }
951:
952:        InstructionResult[5] i2l(Thread[6] * thread)
953:        {
954:                stack_frame[13] * frame = thread[27]->current_stack_frame[2];
```

Footnotes:

1: *opcodes.cpp:900*
2: *Thread.h:148*
3: *Stack.cpp:269*
4: *defs.h:122*
5: *defs.h:120*
6: *Thread.h:17*
7: *opcodes.cpp:906*
8: *Stack.cpp:275*
9: *opcodes.cpp:912*
10: *Stack.cpp:292*
11: *opcodes.cpp:918*
12: *Stack.cpp:283*
13: *Stack.h:129*
14: *opcodes.cpp:928*
15: *defs.h:175*
16: *opcodes.cpp:930*
17: *Stack.cpp:206*
18: *opcodes.cpp:931*
19: *Stack.cpp:165*
20: *opcodes.cpp:933*
21: *opcodes.cpp:939*
22: *opcodes.cpp:941*
23: *defs.h:168*
24: *opcodes.cpp:942*
25: *opcodes.cpp:944*
26: *Stack.cpp:147*
27: *opcodes.cpp:952*

```
955:                su4[1] int_value = frame[2]->pop_int[3]();
956:                // Sign-extend to long
957:                su8[4] long_value = (su8[4])int_value[5];
958:                frame[2]->push_long[6](long_value[7]);
959:
960:                return InstructionSuccess[8];
961:        }
962:
963:        InstructionResult[9] i2s(Thread[10] * thread)
964:        {
965:                stack_frame[11] * frame = thread[12]->current_stack_frame[13];
966:                su4[1] int_value = frame[14]->pop_int[3]();
967:                // Truncate to short
968:                su2[15] short_value = (su2[15])int_value[16];
969:                // Push sign-extended int result back onto stack
970:                frame[14]->push_int[17]((su4[1])short_value[18]);
971:
972:                return InstructionSuccess[8];
973:        }
974:
975:        InstructionResult[9] l2i(Thread[10] * thread)
976:        {
977:                stack_frame[11] * frame = thread[19]->current_stack_frame[13];
978:                su8[4] long_value = frame[20]->pop_long[21]();
979:                // Truncate to low 32 bits
980:                // (Note, that the result may not have the same sign as long_value)
981:                su4[1] int_value = long_value[22] & 0xFFFFFFFF;
982:                frame[20]->push_int[17](int_value[23]);
983:
984:                return InstructionSuccess[8];
985:        }
986:
987:
988:        InstructionResult[9] f2i(Thread[10] * thread)
989:        {
990:                stack_frame[11] * frame = thread[24]->current_stack_frame[13];
991:                float float_value = frame[25]->pop_float[26]();
992:                // Convert to int
993:                su4[1] int_value = (int)float_value[27];
994:                frame[25]->push_int[17](int_value[28]);
995:
996:                return InstructionSuccess[8];
997:        }
998:
999:        InstructionResult[9] f2d(Thread[10] * thread)
1000:       {
1001:               stack_frame[11] * frame = thread[29]->current_stack_frame[13];
1002:               float float_value = frame[30]->pop_float[26]();
1003:               // Convert to double
1004:               double double_value = (double)float_value[31];
1005:               frame[30]->push_double[32](double_value[33]);
1006:
1007:               return InstructionSuccess[8];
```

Footnotes:

1: *defs.h:175*
2: *opcodes.cpp:954*
3: *Stack.cpp:206*
4: *defs.h:177*
5: *opcodes.cpp:955*
6: *Stack.cpp:153*
7: *opcodes.cpp:957*
8: *defs.h:122*
9: *defs.h:120*
10: *Thread.h:17*
11: *Stack.h:129*
12: *opcodes.cpp:963*
13: *Thread.h:148*
14: *opcodes.cpp:965*
15: *defs.h:173*
16: *opcodes.cpp:966*
17: *Stack.cpp:147*
18: *opcodes.cpp:968*
19: *opcodes.cpp:975*
20: *opcodes.cpp:977*
21: *Stack.cpp:214*
22: *opcodes.cpp:978*
23: *opcodes.cpp:981*
24: *opcodes.cpp:988*
25: *opcodes.cpp:990*
26: *Stack.cpp:227*
27: *opcodes.cpp:991*
28: *opcodes.cpp:993*
29: *opcodes.cpp:999*
30: *opcodes.cpp:1001*
31: *opcodes.cpp:1002*
32: *Stack.cpp:171*
33: *opcodes.cpp:1004*

```
1008:         }
1009:
1010:         InstructionResult[1] d2f(Thread[2] * thread)
1011:         {
1012:                 stack_frame[3] * frame = thread[4]->current_stack_frame[5];
1013:                 double double_value = frame[6]->pop_double[7]();
1014:                 // Convert to float
1015:                 float float_value = (float)double_value[8];
1016:                 frame[6]->push_float[9](float_value[10]);
1017:
1018:                 return InstructionSuccess[11];
1019:         }
1020:
1021:         InstructionResult[1] instanceof(Thread[2] * thread)
1022:         {
1023:                 stack_frame[3] * frame = thread[12]->current_stack_frame[5];
1024:
1025:                 // Make a Constant Pool index
1026:                 u1[13] indexbyte1 = thread[12]->current_code[14][thread[12]->pc_register[15]++];
1027:                 u1[13] indexbyte2 = thread[12]->current_code[14][thread[12]->pc_register[15]++];
1028:                 u2[16] index = (indexbyte1[17] << 8) | indexbyte2[18];
1029:
1030:                 CONSTANT_Class_info[19] * entry =
1031:                         (CONSTANT_Class_info[19]*)thread[12]->current_cp[20]->get_item_at_index[21](index[22]);
1032:                 if (!entry[23])
1033:                         return InstructionErrorWrongConstantPoolEntryIndex[24];
1034:
1035:                 // Check whether this entry is already resolved; if not - resolve it
1036:                 if (!entry[23]->is_resolved[25]())
1037:                 {
1038:                         Result[26] result = entry[23]->resolve[27]();
1039:                         if (result[28] != Success[29])
1040:                         {
1041:                                 thread[12]->raise_exception[30](result[28]);
1042:                                 return InstructionErrorCannotExecuteInstanceof[31];
1043:                         }
1044:                 }
1045:
1046:                 // Get the instance that is checked against the resolved class
1047:                 word[32] objectref = frame[33]->pop_reference[34]();
1048:                 if (objectref[35] == null[36])
1049:                 {
1050:                         frame[33]->push_int[37](0); // instance is null - the result is 0
1051:                         return InstructionSuccess[11];
1052:                 }
1053:
1054:                 // Instance is not null
1055:                 InstanceData[38] * id = thread[12]->get_jvm[39]()->get_handle_pool[40]()->get_instance[41](objectref[35]);
1056:                 assert(id[42] != NULL);
1057:                 ClassFile[43] * cf = id[42]->class_file[44];
1058:                 assert(cf[45] != NULL);
1059:
1060:                 if (cf[45]->instance_of[46](entry[23]->resolved_class_file[47]))
```

Footnotes:

1: *defs.h:120*
2: *Thread.h:17*
3: *Stack.h:129*
4: *opcodes.cpp:1010*
5: *Thread.h:148*
6: *opcodes.cpp:1012*
7: *Stack.cpp:235*
8: *opcodes.cpp:1013*
9: *Stack.cpp:165*
10: *opcodes.cpp:1015*
11: *defs.h:122*
12: *opcodes.cpp:1021*
13: *defs.h:168*
14: *Thread.h:151*
15: *Thread.h:159*
16: *defs.h:172*
17: *opcodes.cpp:1026*
18: *opcodes.cpp:1027*
19: *ConstantPool.h:58*
20: *Thread.h:150*
21: *ConstantPool.h:299*
22: *opcodes.cpp:1028*
23: *opcodes.cpp:1030*
24: *defs.h:125*
25: *ConstantPool.h:48*
26: *defs.h:29*
27: *ConstantPool.cpp:22*
28: *opcodes.cpp:1038*
29: *defs.h:33*
30: *Thread.cpp:776*
31: *defs.h:144*
32: *defs.h:195*
33: *opcodes.cpp:1023*
34: *Stack.cpp:249*
35: *opcodes.cpp:1047*
36: *defs.h:201*
37: *Stack.cpp:147*
38: *ObjectData.h:113*
39: *Thread.h:179*
40: *VirtualMachine.h:336*
41: *HandlePool.cpp:25*
42: *opcodes.cpp:1055*
43: *ClassFile.h:136*
44: *ObjectData.h:74*
45: *opcodes.cpp:1057*
46: *ClassFile.cpp:1081*
47: *ConstantPool.h:71*

```
1061:                   frame[1]->push_int[2](1);
1062:           else
1063:                   frame[1]->push_int[2](0);
1064:
1065:           return InstructionSuccess[3];
1066:   }
1067:
1068:   InstructionResult[4] checkcast(Thread[5] * thread)
1069:   {
1070:           stack_frame[6] * frame = thread[7]->current_stack_frame[8];
1071:
1072:           // Make a Constant Pool index
1073:           u1[9] indexbyte1 = thread[7]->current_code[10][thread[7]->pc_register[11]++];
1074:           u1[9] indexbyte2 = thread[7]->current_code[10][thread[7]->pc_register[11]++];
1075:           u2[12] index = (indexbyte1[13] << 8) | indexbyte2[14];
1076:
1077:           CONSTANT_Class_info[15] * entry =
1078:                   (CONSTANT_Class_info[15]*)thread[7]->current_cp[16]->get_item_at_index[17](index[18]);
1079:           if (!entry[19])
1080:                   return InstructionErrorWrongConstantPoolEntryIndex[20];
1081:
1082:           // Check whether this entry is already resolved; if not - resolve it
1083:           if (!entry[19]->is_resolved[21]())
1084:           {
1085:                   Result[22] result = entry[19]->resolve[23]();
1086:                   if (result[24] != Success[25])
1087:                   {
1088:                           thread[7]->raise_exception[26](result[24]);
1089:                           return InstructionErrorCannotExecuteInstanceof[27];
1090:                   }
1091:           }
1092:
1093:           // Get the instance that is checked against the resolved class
1094:           word[28] objectref = frame[29]->pop_reference[30]();
1095:           frame[29]->push_reference[31](objectref[32]); // push the reference back - stack must be unchanged
1096:           if (objectref[32] == null[33])
1097:           {
1098:                   return InstructionSuccess[3];
1099:           }
1100:
1101:           // Instance is not null
1102:           InstanceData[34] * id = thread[7]->get_jvm[35]()->get_handle_pool[36]()->get_instance[37](objectref[32]);
1103:           assert(id[38] != NULL);
1104:           ClassFile[39] * cf = id[38]->class_file[40];
1105:           assert(cf[41] != NULL);
1106:
1107:           if (!cf[41]->instance_of[42](entry[19]->resolved_class_file[43]))
1108:           {
1109:                   thread[7]->raise_exception[26](VM_ERROR_ClassCastException[44]);
1110:                   return InstructionErrorClassCastException[45];
1111:           }
1112:
1113:           return InstructionSuccess[3];
```

Footnotes:

1: *opcodes.cpp:1023*
2: *Stack.cpp:147*
3: *defs.h:122*
4: *defs.h:120*
5: *Thread.h:17*
6: *Stack.h:129*
7: *opcodes.cpp:1068*
8: *Thread.h:148*
9: *defs.h:168*
10: *Thread.h:151*
11: *Thread.h:159*
12: *defs.h:172*
13: *opcodes.cpp:1073*
14: *opcodes.cpp:1074*
15: *ConstantPool.h:58*
16: *Thread.h:150*
17: *ConstantPool.h:299*
18: *opcodes.cpp:1075*
19: *opcodes.cpp:1077*
20: *defs.h:125*
21: *ConstantPool.h:48*
22: *defs.h:29*
23: *ConstantPool.cpp:22*
24: *opcodes.cpp:1085*
25: *defs.h:33*
26: *Thread.cpp:776*
27: *defs.h:144*
28: *defs.h:195*
29: *opcodes.cpp:1070*
30: *Stack.cpp:249*
31: *Stack.cpp:187*
32: *opcodes.cpp:1094*
33: *defs.h:201*
34: *ObjectData.h:113*
35: *Thread.h:179*
36: *VirtualMachine.h:336*
37: *HandlePool.cpp:25*
38: *opcodes.cpp:1102*
39: *ClassFile.h:136*
40: *ObjectData.h:74*
41: *opcodes.cpp:1104*
42: *ClassFile.cpp:1081*
43: *ConstantPool.h:71*
44: *defs.h:111*
45: *defs.h:145*

```
1114:           }
1115:
1116:           //-----------------------------------------------------------------------------
1117:           // Branch operations
1118:           //-----------------------------------------------------------------------------
1119:
1120:           InstructionResult[1] _goto(Thread[2] * thread)
1121:           {
1122:                   u1[3] branchbyte1 = thread[4]->current_code[5][thread[4]->pc_register[6]++];
1123:                   u1[3] branchbyte2 = thread[4]->current_code[5][thread[4]->pc_register[6]++];
1124:                   // NOTE: branchoffset is signed
1125:                   su2[7] branchoffset = (branchbyte1[8] << 8) | branchbyte2[9];
1126:                   // Step back to the opcode of this instruction
1127:                   thread[4]->pc_register[6] -= 3;
1128:                   // Add the offset
1129:                   thread[4]->pc_register[6] += branchoffset[10];
1130:
1131:                   // Check that the next instruction address is within this method's code
1132:                   if (thread[4]->pc_register[6] >= thread[4]->current_code_length[11] || thread[4]->pc_register[6] < 0)
1133:                           return InstructionErrorBranchOutsideMethodCode[12];
1134:
1135:                   return InstructionSuccess[13];
1136:           }
1137:
1138:           InstructionResult[1] _goto_w(Thread[2] * thread)
1139:           {
1140:                   u1[3] branchbyte1 = thread[14]->current_code[5][thread[14]->pc_register[6]++];
1141:                   u1[3] branchbyte2 = thread[14]->current_code[5][thread[14]->pc_register[6]++];
1142:                   u1[3] branchbyte3 = thread[14]->current_code[5][thread[14]->pc_register[6]++];
1143:                   u1[3] branchbyte4 = thread[14]->current_code[5][thread[14]->pc_register[6]++];
1144:                   // NOTE: branchoffset is signed
1145:                   su4[15] branchoffset = (branchbyte1[16] << 24) | (branchbyte2[17] << 16) | (branchbyte3[18] << 8) | branchbyt
           e4[19];
1146:                   // Step back to the opcode of this instruction
1147:                   thread[14]->pc_register[6] -= 5;
1148:                   // Add the offset
1149:                   thread[14]->pc_register[6] += branchoffset[20];
1150:
1151:                   // Check that the next instruction address is within this method's code
1152:                   if (thread[14]->pc_register[6] >= thread[14]->current_code_length[11] || thread[14]->pc_register[6] < 0)
1153:                           return InstructionErrorBranchOutsideMethodCode[12];
1154:
1155:                   return InstructionSuccess[13];
1156:           }
1157:
1158:           // Helper enumeration for the comparisons instructions
1159:           enum IfCond {
1160:                   eq, ne, lt, le, gt, ge
1161:           };
1162:
1163:           static InstructionResult[1] _if_cond(Thread[2] * thread, IfCond[21] cond)
1164:           {
1165:                   u1[3] branchbyte1 = thread[22]->current_code[5][thread[22]->pc_register[6]++];
```

Footnotes:

1: *defs.h:120*
2: *Thread.h:17*
3: *defs.h:168*
4: *opcodes.cpp:1120*
5: *Thread.h:151*
6: *Thread.h:159*
7: *defs.h:173*
8: *opcodes.cpp:1122*
9: *opcodes.cpp:1123*
10: *opcodes.cpp:1125*
11: *Thread.h:152*
12: *defs.h:127*
13: *defs.h:122*
14: *opcodes.cpp:1138*
15: *defs.h:175*
16: *opcodes.cpp:1140*
17: *opcodes.cpp:1141*
18: *opcodes.cpp:1142*
19: *opcodes.cpp:1143*
20: *opcodes.cpp:1145*
21: *opcodes.cpp:1159*
22: *opcodes.cpp:1163*

```
1166:                   u1[1] branchbyte2 = thread[2]->current_code[3][thread[2]->pc_register[4]++];
1167:
1168:                   // Pop an integer value from the stack to compare against zero
1169:                   su4[5] int_value = thread[2]->current_stack_frame[6]->pop_int[7]();
1170:                   int satisfied = 0;
1171:                   switch (cond[8])
1172:                   {
1173:                   case eq[9]: if (int_value[10] == 0) satisfied[11] = 1; break;
1174:                   case ne[12]: if (int_value[10] != 0) satisfied[11] = 1; break;
1175:                   case lt[13]: if (int_value[10] < 0)  satisfied[11] = 1; break;
1176:                   case le[14]: if (int_value[10] <= 0) satisfied[11] = 1; break;
1177:                   case gt[15]: if (int_value[10] > 0)  satisfied[11] = 1; break;
1178:                   case ge[16]: if (int_value[10] >= 0) satisfied[11] = 1; break;
1179:                   }
1180:
1181:                   if (!satisfied[11])
1182:                           // continue execution from the next instruction
1183:                           return InstructionSuccess[17];
1184:
1185:                   // NOTE: branchoffset is signed
1186:                   su2[18] branchoffset = (branchbyte1[19] << 8) | branchbyte2[20];
1187:                   // Step back to the opcode of this instruction
1188:                   thread[2]->pc_register[4] -= 3;
1189:                   // Add the offset
1190:                   thread[2]->pc_register[4] += branchoffset[21];
1191:
1192:                   // Check that the next instruction address is within this method's code
1193:                   if (thread[2]->pc_register[4] >= thread[2]->current_code_length[22] || thread[2]->pc_register[4] < 0)
1194:                           return InstructionErrorBranchOutsideMethodCode[23];
1195:
1196:                   return InstructionSuccess[17];
1197:           }
1198:
1199:           InstructionResult[24] ifeq(Thread[25] * thread)
1200:           {
1201:                   return _if_cond[26](thread[27], eq[9]);
1202:           }
1203:
1204:           InstructionResult[24] ifne(Thread[25] * thread)
1205:           {
1206:                   return _if_cond[26](thread[28], ne[12]);
1207:           }
1208:
1209:           InstructionResult[24] iflt(Thread[25] * thread)
1210:           {
1211:                   return _if_cond[26](thread[29], lt[13]);
1212:           }
1213:
1214:           InstructionResult[24] ifle(Thread[25] * thread)
1215:           {
1216:                   return _if_cond[26](thread[30], le[14]);
1217:           }
1218:
```

Footnotes:

1: *defs.h:168*
2: *opcodes.cpp:1163*
3: *Thread.h:151*
4: *Thread.h:159*
5: *defs.h:175*
6: *Thread.h:148*
7: *Stack.cpp:206*
8: *opcodes.cpp:1163*
9: *opcodes.cpp:1160*
10: *opcodes.cpp:1169*
11: *opcodes.cpp:1170*
12: *opcodes.cpp:1160*
13: *opcodes.cpp:1160*
14: *opcodes.cpp:1160*
15: *opcodes.cpp:1160*
16: *opcodes.cpp:1160*
17: *defs.h:122*
18: *defs.h:173*
19: *opcodes.cpp:1165*
20: *opcodes.cpp:1166*
21: *opcodes.cpp:1186*
22: *Thread.h:152*
23: *defs.h:127*
24: *defs.h:120*
25: *Thread.h:17*
26: *opcodes.cpp:1163*
27: *opcodes.cpp:1199*
28: *opcodes.cpp:1204*
29: *opcodes.cpp:1209*
30: *opcodes.cpp:1214*

```
1219:         InstructionResult[1] ifgt(Thread[2] * thread)
1220:         {
1221:                 return _if_cond[3](thread[4], gt[5]);
1222:         }
1223:
1224:         InstructionResult[1] ifge(Thread[2] * thread)
1225:         {
1226:                 return _if_cond[3](thread[6], ge[7]);
1227:         }
1228:
1229:         static InstructionResult[1] _if_icmp_cond(Thread[2] * thread, IfCond[8] cond)
1230:         {
1231:                 u1[9] branchbyte1 = thread[10]->current_code[11][thread[10]->pc_register[12]++];
1232:                 u1[9] branchbyte2 = thread[10]->current_code[11][thread[10]->pc_register[12]++];
1233:
1234:                 // Pop two integers from the stack and compare them
1235:                 // (note that value2 is on top of the stack)
1236:                 su4[13] int_value2 = thread[10]->current_stack_frame[14]->pop_int[15]();
1237:                 su4[13] int_value1 = thread[10]->current_stack_frame[14]->pop_int[15]();
1238:
1239:                 int satisfied = 0;
1240:                 switch (cond[16])
1241:                 {
1242:                 case eq[17]: if (int_value1[18] == int_value2[19]) satisfied[20] = 1; break;
1243:                 case ne[21]: if (int_value1[18] != int_value2[19]) satisfied[20] = 1; break;
1244:                 case lt[22]: if (int_value1[18] < int_value2[19])  satisfied[20] = 1; break;
1245:                 case le[23]: if (int_value1[18] <= int_value2[19]) satisfied[20] = 1; break;
1246:                 case gt[5]: if (int_value1[18] > int_value2[19])  satisfied[20] = 1; break;
1247:                 case ge[7]: if (int_value1[18] >= int_value2[19]) satisfied[20] = 1; break;
1248:                 }
1249:
1250:                 if (!satisfied[20])
1251:                         // continue execution from the next instruction
1252:                         return InstructionSuccess[24];
1253:
1254:                 // NOTE: branchoffset is signed
1255:                 su2[25] branchoffset = (branchbyte1[26] << 8) | branchbyte2[27];
1256:                 // Step back to the opcode of this instruction
1257:                 thread[10]->pc_register[12] -= 3;
1258:                 // Add the offset
1259:                 thread[10]->pc_register[12] += branchoffset[28];
1260:
1261:                 // Check that the next instruction address is within this method's code
1262:                 if (thread[10]->pc_register[12] >= thread[10]->current_code_length[29] || thread[10]->pc_register[12] < 0)
1263:                         return InstructionErrorBranchOutsideMethodCode[30];
1264:
1265:                 return InstructionSuccess[24];
1266:         }
1267:
1268:         InstructionResult[1] if_icmpeq(Thread[2] * thread)
1269:         {
1270:                 return _if_icmp_cond[31](thread[32], eq[17]);
1271:         }
```

Footnotes:

1: *defs.h:120*
2: *Thread.h:17*
3: *opcodes.cpp:1163*
4: *opcodes.cpp:1219*
5: *opcodes.cpp:1160*
6: *opcodes.cpp:1224*
7: *opcodes.cpp:1160*
8: *opcodes.cpp:1159*
9: *defs.h:168*
10: *opcodes.cpp:1229*
11: *Thread.h:151*
12: *Thread.h:159*
13: *defs.h:175*
14: *Thread.h:148*
15: *Stack.cpp:206*
16: *opcodes.cpp:1229*
17: *opcodes.cpp:1160*
18: *opcodes.cpp:1237*
19: *opcodes.cpp:1236*
20: *opcodes.cpp:1239*
21: *opcodes.cpp:1160*
22: *opcodes.cpp:1160*
23: *opcodes.cpp:1160*
24: *defs.h:122*
25: *defs.h:173*
26: *opcodes.cpp:1231*
27: *opcodes.cpp:1232*
28: *opcodes.cpp:1255*
29: *Thread.h:152*
30: *defs.h:127*
31: *opcodes.cpp:1229*
32: *opcodes.cpp:1268*

```
1272:
1273:           InstructionResult[1] if_icmpne(Thread[2] * thread)
1274:           {
1275:                   return _if_icmp_cond[3](thread[4], ne[5]);
1276:           }
1277:
1278:           InstructionResult[1] if_icmplt(Thread[2] * thread)
1279:           {
1280:                   return _if_icmp_cond[3](thread[6], lt[7]);
1281:           }
1282:
1283:           InstructionResult[1] if_icmple(Thread[2] * thread)
1284:           {
1285:                   return _if_icmp_cond[3](thread[8], le[9]);
1286:           }
1287:
1288:           InstructionResult[1] if_icmpgt(Thread[2] * thread)
1289:           {
1290:                   return _if_icmp_cond[3](thread[10], gt[11]);
1291:           }
1292:
1293:           InstructionResult[1] if_icmpge(Thread[2] * thread)
1294:           {
1295:                   return _if_icmp_cond[3](thread[12], ge[13]);
1296:           }
1297:
1298:           // Helper for funcions if_acmpeq and if_acmpne
1299:           InstructionResult[1] _if_acmp_cond(Thread[2] * thread, IfCond[14] cond)
1300:           {
1301:                   u1[15] branchbyte1 = thread[16]->current_code[17][thread[16]->pc_register[18]++];
1302:                   u1[15] branchbyte2 = thread[16]->current_code[17][thread[16]->pc_register[18]++];
1303:
1304:                   // Pop two references from the stack and compare them
1305:                   // (note that value2 is on top of the stack)
1306:                   word[19] ref_value2 = thread[16]->current_stack_frame[20]->pop_reference[21]();
1307:                   word[19] ref_value1 = thread[16]->current_stack_frame[20]->pop_reference[21]();
1308:
1309:                   int satisfied = 0;
1310:                   switch (cond[22])
1311:                   {
1312:                   case eq[23]: if (ref_value1[24] == ref_value2[25]) satisfied[26] = 1; break;
1313:                   case ne[5]: if (ref_value1[24] != ref_value2[25]) satisfied[26] = 1; break;
1314:                   }
1315:
1316:                   if (!satisfied[26])
1317:                           // continue execution from the next instruction
1318:                           return InstructionSuccess[27];
1319:
1320:                   // NOTE: branchoffset is signed
1321:                   su2[28] branchoffset = (branchbyte1[29] << 8) | branchbyte2[30];
1322:                   // Step back to the opcode of this instruction
1323:                   thread[16]->pc_register[18] -= 3;
1324:                   // Add the offset
```

Footnotes:

1: *defs.h:120*
2: *Thread.h:17*
3: *opcodes.cpp:1229*
4: *opcodes.cpp:1273*
5: *opcodes.cpp:1160*
6: *opcodes.cpp:1278*
7: *opcodes.cpp:1160*
8: *opcodes.cpp:1283*
9: *opcodes.cpp:1160*
10: *opcodes.cpp:1288*
11: *opcodes.cpp:1160*
12: *opcodes.cpp:1293*
13: *opcodes.cpp:1160*
14: *opcodes.cpp:1159*
15: *defs.h:168*
16: *opcodes.cpp:1299*
17: *Thread.h:151*
18: *Thread.h:159*
19: *defs.h:195*
20: *Thread.h:148*
21: *Stack.cpp:249*
22: *opcodes.cpp:1299*
23: *opcodes.cpp:1160*
24: *opcodes.cpp:1307*
25: *opcodes.cpp:1306*
26: *opcodes.cpp:1309*
27: *defs.h:122*
28: *defs.h:173*
29: *opcodes.cpp:1301*
30: *opcodes.cpp:1302*

```
1325:                   thread[1]->pc_register[2] += branchoffset[3];
1326:
1327:                   // Check that the next instruction address is within this method's code
1328:                   if (thread[1]->pc_register[2] >= thread[1]->current_code_length[4] || thread[1]->pc_register[2] < 0)
1329:                           return InstructionErrorBranchOutsideMethodCode[5];
1330:
1331:                   return InstructionSuccess[6];
1332:           }
1333:
1334:           InstructionResult[7] if_acmpeq(Thread[8] * thread)
1335:           {
1336:                   return _if_acmp_cond[9](thread[10], eq[11]);
1337:           }
1338:
1339:           InstructionResult[7] if_acmpne(Thread[8] * thread)
1340:           {
1341:                   return _if_acmp_cond[9](thread[12], ne[13]);
1342:           }
1343:
1344:
1345:           InstructionResult[7] ifnull(Thread[8] * thread)
1346:           {
1347:                   u1[14] branchbyte1 = thread[15]->current_code[16][thread[15]->pc_register[2]++];
1348:                   u1[14] branchbyte2 = thread[15]->current_code[16][thread[15]->pc_register[2]++];
1349:
1350:                   word[17] ref_value = thread[15]->current_stack_frame[18]->pop_reference[19]();
1351:                   if (ref_value[20] != null[21])
1352:                           return InstructionSuccess[6];
1353:
1354:                   // NOTE: branchoffset is signed
1355:                   su2[22] branchoffset = (branchbyte1[23] << 8) | branchbyte2[24];
1356:                   // Step back to the opcode of this instruction
1357:                   thread[15]->pc_register[2] -= 3;
1358:                   // Add the offset
1359:                   thread[15]->pc_register[2] += branchoffset[25];
1360:
1361:                   // Check that the next instruction address is within this method's code
1362:                   if (thread[15]->pc_register[2] >= thread[15]->current_code_length[4] || thread[15]->pc_register[2] < 0)
1363:                           return InstructionErrorBranchOutsideMethodCode[5];
1364:
1365:                   return InstructionSuccess[6];
1366:           }
1367:
1368:           InstructionResult[7] ifnonnull(Thread[8] * thread)
1369:           {
1370:                   u1[14] branchbyte1 = thread[26]->current_code[16][thread[26]->pc_register[2]++];
1371:                   u1[14] branchbyte2 = thread[26]->current_code[16][thread[26]->pc_register[2]++];
1372:
1373:                   word[17] ref_value = thread[26]->current_stack_frame[18]->pop_reference[19]();
1374:                   if (ref_value[27] == null[21])
1375:                           return InstructionSuccess[6];
1376:
1377:                   // NOTE: branchoffset is signed
```

Footnotes:

1: *opcodes.cpp:1299*
2: *Thread.h:159*
3: *opcodes.cpp:1321*
4: *Thread.h:152*
5: *defs.h:127*
6: *defs.h:122*
7: *defs.h:120*
8: *Thread.h:17*
9: *opcodes.cpp:1299*
10: *opcodes.cpp:1334*
11: *opcodes.cpp:1160*
12: *opcodes.cpp:1339*
13: *opcodes.cpp:1160*
14: *defs.h:168*
15: *opcodes.cpp:1345*
16: *Thread.h:151*
17: *defs.h:195*
18: *Thread.h:148*
19: *Stack.cpp:249*
20: *opcodes.cpp:1350*
21: *defs.h:201*
22: *defs.h:173*
23: *opcodes.cpp:1347*
24: *opcodes.cpp:1348*
25: *opcodes.cpp:1355*
26: *opcodes.cpp:1368*
27: *opcodes.cpp:1373*

```
1378:                   su2[1] branchoffset = (branchbyte1[2] << 8) | branchbyte2[3];
1379:                   // Step back to the opcode of this instruction
1380:                   thread[4]->pc_register[5] -= 3;
1381:                   // Add the offset
1382:                   thread[4]->pc_register[5] += branchoffset[6];
1383:
1384:                   // Check that the next instruction address is within this method's code
1385:                   if (thread[4]->pc_register[5] >= thread[4]->current_code_length[7] || thread[4]->pc_register[5] < 0)
1386:                           return InstructionErrorBranchOutsideMethodCode[8];
1387:
1388:                   return InstructionSuccess[9];
1389:           }
1390:
1391:
1392:           static su4[10] _make_offset(u1[11] byte1, u1[11] byte2, u1[11] byte3, u1[11] byte4)
1393:           {
1394:                   return (byte1[12] << 24) | (byte2[13] << 16) | (byte3[14] << 8) | byte4[15];
1395:           }
1396:
1397:           InstructionResult[16] tableswitch(Thread[17] * thread)
1398:           {
1399:                   // Save adress of the tableswitch instruction
1400:                   word[18] instruction_address = thread[19]->pc_register[5]-1;
1401:
1402:                   // Skip padding bytes
1403:                   while ((thread[19]->pc_register[5] % 4) > 0)
1404:                           thread[19]->pc_register[5]++;
1405:
1406:                   // Make default offset
1407:                   u1[11] defaultbyte1 = thread[19]->current_code[20][thread[19]->pc_register[5]++];
1408:                   u1[11] defaultbyte2 = thread[19]->current_code[20][thread[19]->pc_register[5]++];
1409:                   u1[11] defaultbyte3 = thread[19]->current_code[20][thread[19]->pc_register[5]++];
1410:                   u1[11] defaultbyte4 = thread[19]->current_code[20][thread[19]->pc_register[5]++];
1411:                   su4[10] _default = _make_offset[21](defaultbyte1[22],defaultbyte2[23],defaultbyte3[24],defaultbyte4[25]);
1412:
1413:                   // Make low endpoint
1414:                   u1[11] lowbyte1 = thread[19]->current_code[20][thread[19]->pc_register[5]++];
1415:                   u1[11] lowbyte2 = thread[19]->current_code[20][thread[19]->pc_register[5]++];
1416:                   u1[11] lowbyte3 = thread[19]->current_code[20][thread[19]->pc_register[5]++];
1417:                   u1[11] lowbyte4 = thread[19]->current_code[20][thread[19]->pc_register[5]++];
1418:                   su4[10] low = _make_offset[21](lowbyte1[26],lowbyte2[27],lowbyte3[28],lowbyte4[29]);
1419:
1420:                   // Make high endpoint
1421:                   u1[11] highbyte1 = thread[19]->current_code[20][thread[19]->pc_register[5]++];
1422:                   u1[11] highbyte2 = thread[19]->current_code[20][thread[19]->pc_register[5]++];
1423:                   u1[11] highbyte3 = thread[19]->current_code[20][thread[19]->pc_register[5]++];
1424:                   u1[11] highbyte4 = thread[19]->current_code[20][thread[19]->pc_register[5]++];
1425:                   su4[10] high = _make_offset[21](highbyte1[30],highbyte2[31],highbyte3[32],highbyte4[33]);
1426:
1427:                   // Jump table is strated right after this
1428:                   word[18] jump_table_start = thread[19]->pc_register[5];
1429:
1430:                   su4[10] index = thread[19]->current_stack_frame[34]->pop_int[35]();
```

Footnotes:

1: *defs.h:173*
2: *opcodes.cpp:1370*
3: *opcodes.cpp:1371*
4: *opcodes.cpp:1368*
5: *Thread.h:159*
6: *opcodes.cpp:1378*
7: *Thread.h:152*
8: *defs.h:127*
9: *defs.h:122*
10: *defs.h:175*
11: *defs.h:168*
12: *opcodes.cpp:1392*
13: *opcodes.cpp:1392*
14: *opcodes.cpp:1392*
15: *opcodes.cpp:1392*
16: *defs.h:120*
17: *Thread.h:17*
18: *defs.h:195*
19: *opcodes.cpp:1397*
20: *Thread.h:151*
21: *opcodes.cpp:1392*
22: *opcodes.cpp:1407*
23: *opcodes.cpp:1408*
24: *opcodes.cpp:1409*
25: *opcodes.cpp:1410*
26: *opcodes.cpp:1414*
27: *opcodes.cpp:1415*
28: *opcodes.cpp:1416*
29: *opcodes.cpp:1417*
30: *opcodes.cpp:1421*
31: *opcodes.cpp:1422*
32: *opcodes.cpp:1423*
33: *opcodes.cpp:1424*
34: *Thread.h:148*
35: *Stack.cpp:206*

```
1431:                   if (index[1] < low[2] || index[1] > high[3])
1432:                           // Jump
1433:                           thread[4]->pc_register[5] = (instruction_address[6] + _default[7]);
1434:                   else
1435:                   {
1436:                           // Make an offset from the bytes found at the calculated jump table index
1437:                           u4[8] jump_table_index = jump_table_start[9] + (index[1] - low[2])*4;
1438:                           u1[10] offsetbyte1 = thread[4]->current_code[11][jump_table_index[12]];
1439:                           u1[10] offsetbyte2 = thread[4]->current_code[11][jump_table_index[12] + 1];
1440:                           u1[10] offsetbyte3 = thread[4]->current_code[11][jump_table_index[12] + 2];
1441:                           u1[10] offsetbyte4 = thread[4]->current_code[11][jump_table_index[12] + 3];
1442:                           su4[13] offset = _make_offset[14](offsetbyte1[15],offsetbyte2[16],offsetbyte3[17],offsetbyte4[18]);
1443:                           // Jump
1444:                           thread[4]->pc_register[5] = (instruction_address[6] + offset[19]);
1445:                   }
1446:
1447:                   // Check that the next instruction address is within this method's code
1448:                   if (thread[4]->pc_register[5] >= thread[4]->current_code_length[20] || thread[4]->pc_register[5] < 0)
1449:                           return InstructionErrorBranchOutsideMethodCode[21];
1450:
1451:                   return InstructionSuccess[22];
1452:           }
1453:
1454:           //-----------------------------------------------------------------------------
1455:           // Comparison operations
1456:           //-----------------------------------------------------------------------------
1457:
1458:           /*****************************************************************************
1459:           Format
1460:           ------
1461:           LCMP
1462:
1463:           Operand Stack
1464:           -------------
1465:           ..., value1, value2  -> ..., result
1466:
1467:           Description
1468:           -----------
1469:           Both value1 and value2 must be of type long.
1470:           They are both popped from the operand stack, and a signed integer comparison is performed.
1471:           If value1 is greater than value2, the int value 1 is pushed onto the operand stack.
1472:           If value1 is equal to value2, the int value 0 is pushed onto the operand stack.
1473:           If value1 is less than value2, the int value -1 is pushed onto the operand stack.
1474:           *****************************************************************************/
1475:           InstructionResult[23] lcmp(Thread[24] * thread)
1476:           {
1477:                   stack_frame[25] * frame = thread[26]->current_stack_frame[27];
1478:                   su8[28] value2 = frame[29]->pop_long[30]();
1479:                   su8[28] value1 = frame[29]->pop_long[30]();
1480:
1481:                   if (value1[31] > value2[32])
1482:                           frame[29]->push_int[33](1);
1483:                   else if (value1[31] == value2[32])
```

Footnotes:

1: *opcodes.cpp:1430*
2: *opcodes.cpp:1418*
3: *opcodes.cpp:1425*
4: *opcodes.cpp:1397*
5: *Thread.h:159*
6: *opcodes.cpp:1400*
7: *opcodes.cpp:1411*
8: *defs.h:174*
9: *opcodes.cpp:1428*
10: *defs.h:168*
11: *Thread.h:151*
12: *opcodes.cpp:1437*
13: *defs.h:175*
14: *opcodes.cpp:1392*
15: *opcodes.cpp:1438*
16: *opcodes.cpp:1439*
17: *opcodes.cpp:1440*
18: *opcodes.cpp:1441*
19: *opcodes.cpp:1442*
20: *Thread.h:152*
21: *defs.h:127*
22: *defs.h:122*
23: *defs.h:120*
24: *Thread.h:17*
25: *Stack.h:129*
26: *opcodes.cpp:1475*
27: *Thread.h:148*
28: *defs.h:177*
29: *opcodes.cpp:1477*
30: *Stack.cpp:214*
31: *opcodes.cpp:1479*
32: *opcodes.cpp:1478*
33: *Stack.cpp:147*

```
1484:                           frame[1]->push_int[2](0);
1485:                   else if (value1[3] < value2[4])
1486:                           frame[1]->push_int[2](-1);
1487:
1488:                   return InstructionSuccess[5];
1489:           }
1490:
1491:           // Helper function for fcmpl and fcmpg
1492:           InstructionResult[6] _fcmp(Thread[7] * thread, int unit)
1493:           {
1494:                   stack_frame[8] * frame = thread[9]->current_stack_frame[10];
1495:                   float value2 = frame[11]->pop_float[12]();
1496:                   float value1 = frame[11]->pop_float[12]();
1497:
1498:                   if (value1[13] > value2[14])
1499:                           frame[11]->push_int[2](1);
1500:                   else if (value1[13] == value2[14])
1501:                           frame[11]->push_int[2](0);
1502:                   else if (value1[13] < value2[14])
1503:                           frame[11]->push_int[2](-1);
1504:                   else // at least one of the values is NaN
1505:                           frame[11]->push_int[2](unit[15]);
1506:
1507:                   return InstructionSuccess[5];
1508:           }
1509:
1510:           InstructionResult[6] fcmpl(Thread[7] * thread)
1511:           {
1512:                   return _fcmp[16](thread[17], -1);
1513:           }
1514:
1515:           InstructionResult[6] fcmpg(Thread[7] * thread)
1516:           {
1517:                   return _fcmp[16](thread[18], 1);
1518:           }
1519:
1520:           // Helper function for dcmpl and dcmpg
1521:           InstructionResult[6] _dcmp(Thread[7] * thread, int unit)
1522:           {
1523:                   stack_frame[8] * frame = thread[19]->current_stack_frame[10];
1524:                   double value2 = frame[20]->pop_double[21]();
1525:                   double value1 = frame[20]->pop_double[21]();
1526:
1527:                   if (value1[22] > value2[23])
1528:                           frame[20]->push_int[2](1);
1529:                   else if (value1[22] == value2[23])
1530:                           frame[20]->push_int[2](0);
1531:                   else if (value1[22] < value2[23])
1532:                           frame[20]->push_int[2](-1);
1533:                   else // at least one of the values is NaN
1534:                           frame[20]->push_int[2](unit[24]);
1535:
1536:                   return InstructionSuccess[5];
```

Footnotes:

1: *opcodes.cpp:1477*
2: *Stack.cpp:147*
3: *opcodes.cpp:1479*
4: *opcodes.cpp:1478*
5: *defs.h:122*
6: *defs.h:120*
7: *Thread.h:17*
8: *Stack.h:129*
9: *opcodes.cpp:1492*
10: *Thread.h:148*
11: *opcodes.cpp:1494*
12: *Stack.cpp:227*
13: *opcodes.cpp:1496*
14: *opcodes.cpp:1495*
15: *opcodes.cpp:1492*
16: *opcodes.cpp:1492*
17: *opcodes.cpp:1510*
18: *opcodes.cpp:1515*
19: *opcodes.cpp:1521*
20: *opcodes.cpp:1523*
21: *Stack.cpp:235*
22: *opcodes.cpp:1525*
23: *opcodes.cpp:1524*
24: *opcodes.cpp:1521*

```
1537:           }
1538:
1539:           InstructionResult[1] dcmpl(Thread[2] * thread)
1540:           {
1541:                   return _dcmp[3](thread[4], -1);
1542:           }
1543:
1544:           InstructionResult[1] dcmpg(Thread[2] * thread)
1545:           {
1546:                   return _dcmp[3](thread[5], 1);
1547:           }
1548:
1549:           //----------------------------------------------------------------------------------------
1550:           // Constant Pool operations
1551:           //----------------------------------------------------------------------------------------
1552:
1553:           // Helper function for ldc and ldc_w
1554:           InstructionResult[1] _ldc(Thread[2] * thread, u2[6] index)
1555:           {
1556:                   float fvalue;
1557:                   int ivalue;
1558:                   word[7] reference;
1559:                   CONSTANT_String_info[8] * string_entry;
1560:
1561:                   ConstantPoolEntry[9] * entry = thread[10]->current_cp[11]->get_item_at_index[12](index[13]);
1562:                   if (!entry[14])
1563:                           return InstructionErrorWrongConstantPoolEntryIndex[15];
1564:
1565:                   switch (entry[14]->tag[16])
1566:                   {
1567:                   case CONSTANT_Float[17]:
1568:                           fvalue[18] = ((CONSTANT_Float_info[19]*)entry[14])->get_float_value[20]();
1569:                           thread[10]->current_stack_frame[21]->push_float[22](fvalue[18]);
1570:                           break;
1571:                   case CONSTANT_Integer[23]:
1572:                           ivalue[24] = ((CONSTANT_Integer_info[25]*)entry[14])->get_int_value[26]();
1573:                           thread[10]->current_stack_frame[21]->push_int[27](ivalue[24]);
1574:                           break;
1575:                   case CONSTANT_String[28]:
1576:                           // Get the reference from the interned String collection
1577:                           string_entry[29] = (CONSTANT_String_info[8]*)entry[14];
1578:                           if (!string_entry[29]->is_resolved[30]())
1579:                           {
1580:                                   Result[31] result = string_entry[29]->resolve[32]();
1581:                                   if (result[33] != Success[34])
1582:                                   {
1583:                                           thread[10]->raise_exception[35](result[33]);
1584:                                           return InstructionErrorCannotGetInternedString[36];
1585:                                   }
1586:                           }
1587:                           reference[37] = string_entry[29]->interned_string_reference[38];
1588:                           if (reference[37] == null[39])
1589:                           {
```

Footnotes:

1: *defs.h:120*
2: *Thread.h:17*
3: *opcodes.cpp:1521*
4: *opcodes.cpp:1539*
5: *opcodes.cpp:1544*
6: *defs.h:172*
7: *defs.h:195*
8: *ConstantPool.h:170*
9: *ConstantPool.h:20*
10: *opcodes.cpp:1554*
11: *Thread.h:150*
12: *ConstantPool.h:299*
13: *opcodes.cpp:1554*
14: *opcodes.cpp:1561*
15: *defs.h:125*
16: *ConstantPool.h:28*
17: *defs.h:332*
18: *opcodes.cpp:1556*
19: *ConstantPool.h:219*
20: *ConstantPool.cpp:152*
21: *Thread.h:148*
22: *Stack.cpp:165*
23: *defs.h:331*
24: *opcodes.cpp:1557*
25: *ConstantPool.h:204*
26: *ConstantPool.h:209*
27: *Stack.cpp:147*
28: *defs.h:330*
29: *opcodes.cpp:1559*
30: *ConstantPool.h:48*
31: *defs.h:29*
32: *ConstantPool.cpp:600*
33: *opcodes.cpp:1580*
34: *defs.h:33*
35: *Thread.cpp:776*
36: *defs.h:142*
37: *opcodes.cpp:1558*
38: *ConstantPool.h:183*
39: *defs.h:201*

```
1590:                                   thread[1]->raise_exception[2](VM_ERROR_NullPointerException[3]);
1591:                                   return InstructionErrorNullPointerException[4];
1592:                           }
1593:                           thread[1]->current_stack_frame[5]->push_reference[6](reference[7]);
1594:                           break;
1595:                   default:
1596:                           return InstructionErrorWrongConstantPoolEntryType[8];
1597:                   }
1598:                   return InstructionSuccess[9];
1599:           }
1600:
1601:           InstructionResult[10] ldc(Thread[11] * thread)
1602:           {
1603:                   u1[12] index = thread[13]->current_code[14][thread[13]->pc_register[15]++];
1604:
1605:                   return _ldc[16](thread[13], (u2[17])index[18]);
1606:           }
1607:
1608:           InstructionResult[10] ldc_w(Thread[11] * thread)
1609:           {
1610:                   u1[12] indexbyte1 = thread[19]->current_code[14][thread[19]->pc_register[15]++];
1611:                   u1[12] indexbyte2 = thread[19]->current_code[14][thread[19]->pc_register[15]++];
1612:                   u2[17] index = (indexbyte1[20] << 8) | indexbyte2[21];
1613:
1614:                   return _ldc[16](thread[19], index[22]);
1615:           }
1616:
1617:           InstructionResult[10] ldc2_w(Thread[11] * thread)
1618:           {
1619:                   double dvalue;
1620:                   su8[23] lvalue;
1621:
1622:                   // Make a Constant Pool index
1623:                   u1[12] indexbyte1 = thread[24]->current_code[14][thread[24]->pc_register[15]++];
1624:                   u1[12] indexbyte2 = thread[24]->current_code[14][thread[24]->pc_register[15]++];
1625:                   u2[17] index = (indexbyte1[25] << 8) | indexbyte2[26];
1626:
1627:                   ConstantPoolEntry[27] * entry = thread[24]->current_cp[28]->get_item_at_index[29](index[30]);
1628:                   if (!entry[31])
1629:                           return InstructionErrorWrongConstantPoolEntryIndex[32];
1630:
1631:                   switch (entry[31]->tag[33])
1632:                   {
1633:                   case CONSTANT_Double[34]:
1634:                           dvalue[35] = ((CONSTANT_Double_info[36]*)entry[31])->get_double_value[37]();
1635:                           thread[24]->current_stack_frame[5]->push_double[38](dvalue[35]);
1636:                           break;
1637:                   case CONSTANT_Long[39]:
1638:                           lvalue[40] = ((CONSTANT_Long_info[41]*)entry[31])->get_long_value[42]();
1639:                           thread[24]->current_stack_frame[5]->push_long[43](lvalue[40]);
1640:                           break;
1641:                   default:
1642:                           return InstructionErrorWrongConstantPoolEntryType[8];
```

Footnotes:

1: *opcodes.cpp:1554*
2: *Thread.cpp:776*
3: *defs.h:105*
4: *defs.h:138*
5: *Thread.h:148*
6: *Stack.cpp:187*
7: *opcodes.cpp:1558*
8: *defs.h:126*
9: *defs.h:122*
10: *defs.h:120*
11: *Thread.h:17*
12: *defs.h:168*
13: *opcodes.cpp:1601*
14: *Thread.h:151*
15: *Thread.h:159*
16: *opcodes.cpp:1554*
17: *defs.h:172*
18: *opcodes.cpp:1603*
19: *opcodes.cpp:1608*
20: *opcodes.cpp:1610*
21: *opcodes.cpp:1611*
22: *opcodes.cpp:1612*
23: *defs.h:177*
24: *opcodes.cpp:1617*
25: *opcodes.cpp:1623*
26: *opcodes.cpp:1624*
27: *ConstantPool.h:20*
28: *Thread.h:150*
29: *ConstantPool.h:299*
30: *opcodes.cpp:1625*
31: *opcodes.cpp:1627*
32: *defs.h:125*
33: *ConstantPool.h:28*
34: *defs.h:334*
35: *opcodes.cpp:1619*
36: *ConstantPool.h:244*
37: *ConstantPool.cpp:212*
38: *Stack.cpp:171*
39: *defs.h:333*
40: *opcodes.cpp:1620*
41: *ConstantPool.h:231*
42: *ConstantPool.cpp:111*
43: *Stack.cpp:153*

```
1643:                   }
1644:                   return InstructionSuccess[1];
1645:           }
1646:
1647:           //-----------------------------------------------------------------------------
1648:           // Field operations
1649:           //-----------------------------------------------------------------------------
1650:
1651:           InstructionResult[2] getstatic(Thread[3] * thread)
1652:           {
1653:                   // Make a Constant Pool index
1654:                   u1[4] indexbyte1 = thread[5]->current_code[6][thread[5]->pc_register[7]++];
1655:                   u1[4] indexbyte2 = thread[5]->current_code[6][thread[5]->pc_register[7]++];
1656:                   u2[8] index = (indexbyte1[9] << 8) | indexbyte2[10];
1657:
1658:                   // Get the CONSTANT_Fieldref_info entry from the Constant Pool
1659:                   CONSTANT_Fieldref_info[11] * entry = (CONSTANT_Fieldref_info[11]*)thread[5]->current_cp[12]->get_item_at_ind
           ex[13](index[14]);
1660:                   if (!index[14])
1661:                           return InstructionErrorWrongConstantPoolEntryIndex[15];
1662:
1663:                   // Check whether this entry is already resolved; if not - resolve it
1664:                   if (!entry[16]->is_resolved[17]())
1665:                   {
1666:                           Result[18] result = entry[16]->resolve[19]();
1667:                           if (result[20] != Success[21])
1668:                           {
1669:                                   thread[5]->raise_exception[22](result[20]);
1670:                                   return InstructionErrorCannotGetStaticField[23];
1671:                           }
1672:                   }
1673:
1674:                   // TODO: check access permissions
1675:
1676:                   // Check that the field is static
1677:                   if (! (entry[16]->finfo[24]->access_flags[25] & ACC_STATIC[26]))
1678:                   {
1679:                           thread[5]->raise_exception[22](VM_ERROR_IncompatibleClassChangeError[27]);
1680:                           return InstructionErrorCannotGetStaticField[23];
1681:                   }
1682:
1683:                   // Get the value stored in the field from the corresponding offset
1684:                   // in the referenced class' static data
1685:                   field_info[28] * finfo = entry[16]->finfo[24];
1686:                   assert(finfo[29] != NULL);
1687:                   assert(finfo[29]->class_file[30]->class_data[31] != NULL);
1688:                   assert(finfo[29]->class_file[30]->class_data[31]->data_start[32] != NULL);
1689:                   u1[4] * field_start = finfo[29]->class_file[30]->class_data[31]->data_start[32]->address[33] + finfo[29]->offset[34];
1690:
1691:                   // Push the field's value onto the operand stack;
1692:                   // Note, that the proper number of bytes to copy
1693:                   // will be taken from the memory by the corresponding function
1694:                   su1[35] byte_value;
```

Footnotes:

1: *defs.h:122*
2: *defs.h:120*
3: *Thread.h:17*
4: *defs.h:168*
5: *opcodes.cpp:1651*
6: *Thread.h:151*
7: *Thread.h:159*
8: *defs.h:172*
9: *opcodes.cpp:1654*
10: *opcodes.cpp:1655*
11: *ConstantPool.h:74*
12: *Thread.h:150*
13: *ConstantPool.h:299*
14: *opcodes.cpp:1656*
15: *defs.h:125*
16: *opcodes.cpp:1659*
17: *ConstantPool.h:48*
18: *defs.h:29*
19: *ConstantPool.cpp:238*
20: *opcodes.cpp:1666*
21: *defs.h:33*
22: *Thread.cpp:776*
23: *defs.h:134*
24: *ConstantPool.h:90*
25: *ClassFile.h:37*
26: *defs.h:364*
27: *defs.h:101*
28: *ClassFile.h:68*
29: *opcodes.cpp:1685*
30: *ClassFile.h:56*
31: *ClassFile.h:227*
32: *ObjectData.h:71*
33: *HeapManager.h:46*
34: *ClassFile.h:79*
35: *defs.h:169*

```
1695:                   switch (finfo[1]->type[2])
1696:                   {
1697:                   case Byte[3]:
1698:                           byte_value[4] = *(su1[5]*)field_start[6];
1699:                           thread[7]->current_stack_frame[8]->push_int[9]((su4[10])byte_value[4]);
1700:                           break;
1701:                   case Int[11]: case Char[12]: case Short[13]: case Boolean[14]:
1702:                           thread[7]->current_stack_frame[8]->push_int[9](*(su4[10]*)field_start[6]);
1703:                           break;
1704:                   case Double[15]:
1705:                           thread[7]->current_stack_frame[8]->push_double[16](*(double*)field_start[6]);
1706:                           break;
1707:                   case Float[17]:
1708:                           thread[7]->current_stack_frame[8]->push_float[18](*(float*)field_start[6]);
1709:                           break;
1710:                   case Long[19]:
1711:                           thread[7]->current_stack_frame[8]->push_long[20](*(su8[21]*)field_start[6]);
1712:                           break;
1713:                   case Array[22]: case Reference[23]:
1714:                           thread[7]->current_stack_frame[8]->push_reference[24](*(word[25]*)field_start[6]);
1715:                           break;
1716:                   default: return InstructionErrorCannotGetStaticField[26];
1717:                   }
1718:
1719:           #ifdef DEBUG_EXECUTION[27]
1720:                   #ifdef DEBUG_INSTRUCTIONS
1721:                   finfo[1]->debug_print[28](debug_file);
1722:                   debug_file << "VALUE OF INSTANCE FIELD IS: ";
1723:                   switch (finfo[1]->type[2])
1724:                   {
1725:                   case Int[11]: case Byte[3]: case Char[12]: case Short[13]: case Boolean[14]:
1726:                           debug_file << *(su4[10]*)field_start[6]; break;
1727:                   case Double[15]:
1728:                           debug_file << *(double*)field_start[6]; break;
1729:                   case Float[17]:
1730:                           debug_file << *(float*)field_start[6]; break;
1731:                   case Long[19]:
1732:                           debug_file << *(long*)field_start[6]; break;
1733:                   case Array[22]: case Reference[23]:
1734:                           debug_file << *(long*)field_start[6]; break;
1735:                   }
1736:                   debug_file << endl;
1737:                   #endif
1738:           #endif
1739:
1740:                   return InstructionSuccess[29];
1741:           }
1742:
1743:           InstructionResult[30] putstatic(Thread[31] * thread)
1744:           {
1745:                   // Make a Constant Pool index
1746:                   u1[32] indexbyte1 = thread[33]->current_code[34][thread[33]->pc_register[35]++];
1747:                   u1[32] indexbyte2 = thread[33]->current_code[34][thread[33]->pc_register[35]++];
```

Footnotes:

1: *opcodes.cpp:1685*
2: *ClassFile.h:72*
3: *defs.h:210*
4: *opcodes.cpp:1694*
5: *defs.h:169*
6: *opcodes.cpp:1689*
7: *opcodes.cpp:1651*
8: *Thread.h:148*
9: *Stack.cpp:147*
10: *defs.h:175*
11: *defs.h:214*
12: *defs.h:211*
13: *defs.h:217*
14: *defs.h:218*
15: *defs.h:212*
16: *Stack.cpp:171*
17: *defs.h:213*
18: *Stack.cpp:165*
19: *defs.h:215*
20: *Stack.cpp:153*
21: *defs.h:177*
22: *defs.h:219*
23: *defs.h:216*
24: *Stack.cpp:187*
25: *defs.h:195*
26: *defs.h:134*
27: *defs.h:428*
28: *ClassFile.cpp:132*
29: *defs.h:122*
30: *defs.h:120*
31: *Thread.h:17*
32: *defs.h:168*
33: *opcodes.cpp:1743*
34: *Thread.h:151*
35: *Thread.h:159*

```
1748:                   u2[1] index = (indexbyte1[2] << 8) | indexbyte2[3];
1749:
1750:                   // Get the CONSTANT_Fieldref_info entry from the Constant Pool
1751:                   CONSTANT_Fieldref_info[4] * entry = (CONSTANT_Fieldref_info[4]*)thread[5]->current_cp[6]->get_item_at_index[7]
         (index[8]);
1752:                   if (!index[8])
1753:                           return InstructionErrorWrongConstantPoolEntryIndex[9];
1754:
1755:                   // Check whether this entry is already resolved; if not - resolve it
1756:                   if (!entry[10]->is_resolved[11]())
1757:                   {
1758:                           Result[12] result = entry[10]->resolve[13]();
1759:                           if (result[14] != Success[15])
1760:                           {
1761:                                   thread[5]->raise_exception[16](result[14]);
1762:                                   return InstructionErrorCannotPutStaticField[17];
1763:                           }
1764:                   }
1765:
1766:                   // TODO: check access permissions
1767:
1768:                   // Check that the field is static
1769:                   if (! (entry[10]->finfo[18]->access_flags[19] & ACC_STATIC[20]))
1770:                   {
1771:                           thread[5]->raise_exception[16](VM_ERROR_IncompatibleClassChangeError[21]);
1772:                           return InstructionErrorCannotPutStaticField[17];
1773:                   }
1774:
1775:                   // Get the value stored in the field from the corresponding offset
1776:                   // in the referenced class' static data
1777:                   field_info[22] * finfo = entry[10]->finfo[18];
1778:                   assert(finfo[23] != NULL);
1779:                   assert(finfo[23]->class_file[24]->class_data[25] != NULL);
1780:                   assert(finfo[23]->class_file[24]->class_data[25]->data_start[26] != NULL);
1781:                   u1[27] * field_start = finfo[23]->class_file[24]->class_data[25]->data_start[26]->address[28] + finfo[23]->offset[29];
1782:
1783:                   // Pop the value from the operand stack;
1784:                   // The type of the value must correspond to the type of the field
1785:                   // Then, copy the value into the field
1786:                   su4[30] int_value;
1787:                   double double_value;
1788:                   float float_value;
1789:                   su8[31] long_value;
1790:                   word[32] ref_value;
1791:                   switch (finfo[23]->type[33])
1792:                   {
1793:                           case Int[34]:
1794:                                   int_value[35] = thread[5]->current_stack_frame[36]->pop_int[37]();
1795:                                   memcpy(field_start[38], &int_value[35], BasicTypes[39][Int[34]].size[40]);
1796:                                   break;
1797:                           case Byte[41]:
1798:                                   int_value[35] = thread[5]->current_stack_frame[36]->pop_int[37]();
1799:                                   memcpy(field_start[38], (su1[42]*)(&int_value[35]), BasicTypes[39][Byte[41]].size[40]);
```

Footnotes:

1: *defs.h:172*
2: *opcodes.cpp:1746*
3: *opcodes.cpp:1747*
4: *ConstantPool.h:74*
5: *opcodes.cpp:1743*
6: *Thread.h:150*
7: *ConstantPool.h:299*
8: *opcodes.cpp:1748*
9: *defs.h:125*
10: *opcodes.cpp:1751*
11: *ConstantPool.h:48*
12: *defs.h:29*
13: *ConstantPool.cpp:238*
14: *opcodes.cpp:1758*
15: *defs.h:33*
16: *Thread.cpp:776*
17: *defs.h:135*
18: *ConstantPool.h:90*
19: *ClassFile.h:37*
20: *defs.h:364*
21: *defs.h:101*
22: *ClassFile.h:68*
23: *opcodes.cpp:1777*
24: *ClassFile.h:56*
25: *ClassFile.h:227*
26: *ObjectData.h:71*
27: *defs.h:168*
28: *HeapManager.h:46*
29: *ClassFile.h:79*
30: *defs.h:175*
31: *defs.h:177*
32: *defs.h:195*
33: *ClassFile.h:72*
34: *defs.h:214*
35: *opcodes.cpp:1786*
36: *Thread.h:148*
37: *Stack.cpp:206*
38: *opcodes.cpp:1781*
39: *defs.h:246*
40: *defs.h:227*
41: *defs.h:210*
42: *defs.h:169*

```
1800:                                   break;
1801:                           case Char[1]:
1802:                                   int_value[2] = thread[3]->current_stack_frame[4]->pop_int[5]();
1803:                                   memcpy(field_start[6], (u2[7]*)(&int_value[2]), BasicTypes[8][Char[1]].size[9]);
1804:                                   break;
1805:                           case Short[10]:
1806:                                   int_value[2] = thread[3]->current_stack_frame[4]->pop_int[5]();
1807:                                   memcpy(field_start[6], (su2[11]*)(&int_value[2]), BasicTypes[8][Short[10]].size[9]);
1808:                                   break;
1809:                           case Boolean[12]:
1810:                                   int_value[2] = thread[3]->current_stack_frame[4]->pop_int[5]();
1811:                                   memcpy(field_start[6], &int_value[2], BasicTypes[8][Boolean[12]].size[9]);
1812:                                   break;
1813:                           case Double[13]:
1814:                                   double_value[14] = thread[3]->current_stack_frame[4]->pop_double[15]();
1815:                                   memcpy(field_start[6], &double_value[14], BasicTypes[8][Double[13]].size[9]);
1816:                                   break;
1817:                           case Float[16]:
1818:                                   float_value[17] = thread[3]->current_stack_frame[4]->pop_float[18]();
1819:                                   memcpy(field_start[6], &float_value[17], BasicTypes[8][Float[16]].size[9]);
1820:                                   break;
1821:                           case Long[19]:
1822:                                   long_value[20] = thread[3]->current_stack_frame[4]->pop_long[21]();
1823:                                   memcpy(field_start[6], &long_value[20], BasicTypes[8][Long[19]].size[9]);
1824:                                   break;
1825:                           case Array[22]: case Reference[23]:
1826:                                   ref_value[24] = thread[3]->current_stack_frame[4]->pop_reference[25]();
1827:                                   memcpy(field_start[6], &ref_value[24], sizeof(word[26]));
1828:                                   break;
1829:                           default: return InstructionErrorCannotPutStaticField[27];
1830:                   }
1831:
1832:           #ifdef DEBUG_EXECUTION[28]
1833:                   #ifdef DEBUG_INSTRUCTIONS
1834:                   finfo[29]->debug_print[30](debug_file);
1835:                   debug_file << "VALUE ASSIGNED TO CLASS FIELD: ";
1836:                   switch (finfo[29]->type[31])
1837:                   {
1838:                   case Int[32]: case Byte[33]: case Char[1]: case Short[10]: case Boolean[12]:
1839:                                   debug_file << int_value[2]; break;
1840:                           case Double[13]:
1841:                                   debug_file << double_value[14]; break;
1842:                           case Float[16]:
1843:                                   debug_file << float_value[17]; break;
1844:                           case Long[19]:
1845:                                   debug_file << (long)long_value[20]; break;
1846:                           case Array[22]: case Reference[23]:
1847:                                   debug_file << (long)ref_value[24]; break;
1848:                   }
1849:                   debug_file << endl;
1850:                   #endif
1851:           #endif
1852:
```

Footnotes:

1: *defs.h:211*
2: *opcodes.cpp:1786*
3: *opcodes.cpp:1743*
4: *Thread.h:148*
5: *Stack.cpp:206*
6: *opcodes.cpp:1781*
7: *defs.h:172*
8: *defs.h:246*
9: *defs.h:227*
10: *defs.h:217*
11: *defs.h:173*
12: *defs.h:218*
13: *defs.h:212*
14: *opcodes.cpp:1787*
15: *Stack.cpp:235*
16: *defs.h:213*
17: *opcodes.cpp:1788*
18: *Stack.cpp:227*
19: *defs.h:215*
20: *opcodes.cpp:1789*
21: *Stack.cpp:214*
22: *defs.h:219*
23: *defs.h:216*
24: *opcodes.cpp:1790*
25: *Stack.cpp:249*
26: *defs.h:195*
27: *defs.h:135*
28: *defs.h:428*
29: *opcodes.cpp:1777*
30: *ClassFile.cpp:132*
31: *ClassFile.h:72*
32: *defs.h:214*
33: *defs.h:210*

```
1853:                   return InstructionSuccess[1];
1854:           }
1855:
1856:           InstructionResult[2] getfield(Thread[3] * thread)
1857:           {
1858:                   // Make a Constant Pool index
1859:                   u1[4] indexbyte1 = thread[5]->current_code[6][thread[5]->pc_register[7]++];
1860:                   u1[4] indexbyte2 = thread[5]->current_code[6][thread[5]->pc_register[7]++];
1861:                   u2[8] index = (indexbyte1[9] << 8) | indexbyte2[10];
1862:
1863:                   // Get the CONSTANT_Fieldref_info entry from the Constant Pool
1864:                   CONSTANT_Fieldref_info[11] * entry = (CONSTANT_Fieldref_info[11]*)thread[5]->current_cp[12]->get_item_at_ind
           ex[13](index[14]);
1865:                   if (!entry[15])
1866:                           return InstructionErrorWrongConstantPoolEntryIndex[16];
1867:
1868:                   // Check whether this entry is already resolved; if not - resolve it
1869:                   if (!entry[15]->is_resolved[17]())
1870:                   {
1871:                           Result[18] result = entry[15]->resolve[19]();
1872:                           if (result[20] != Success[21])
1873:                           {
1874:                                   thread[5]->raise_exception[22](result[20]);
1875:                                   return InstructionErrorCannotGetField[23];
1876:                           }
1877:                   }
1878:
1879:                   // Check that the field is not static
1880:                   if (entry[15]->finfo[24]->access_flags[25] & ACC_STATIC[26])
1881:                   {
1882:                           thread[5]->raise_exception[22](VM_ERROR_IncompatibleClassChangeError[27]);
1883:                           return InstructionErrorCannotGetField[23];
1884:                   }
1885:
1886:                   // TODO: check access permissions
1887:
1888:                   // Pop the object reference from the operand stack
1889:                   word[28] objectref = thread[5]->current_stack_frame[29]->pop_reference[30]();
1890:                   if (objectref[31] == null[32])
1891:                   {
1892:                           thread[5]->raise_exception[22](VM_ERROR_NullPointerException[33]);
1893:                           return InstructionErrorNullPointerException[34];
1894:                   }
1895:                   // Get the class of the object whose field is to be accessed
1896:                   InstanceData[35] * id = thread[5]->get_jvm[36]()->get_handle_pool[37]()->get_instance[38](objectref[31]);
1897:                   assert(id[39] != NULL);
1898:
1899:                   // Get the value stored in the field from the corresponding offset
1900:                   // in the referenced instance memory block
1901:                   field_info[40] * finfo = entry[15]->finfo[24];
1902:                   assert(finfo[41] != NULL);
1903:                   assert(id[39]->data_start[42]);
1904:                   u1[4] * field_start = id[39]->data_start[42]->address[43] + finfo[41]->offset[44];
```

Footnotes:

1: *defs.h:122*
2: *defs.h:120*
3: *Thread.h:17*
4: *defs.h:168*
5: *opcodes.cpp:1856*
6: *Thread.h:151*
7: *Thread.h:159*
8: *defs.h:172*
9: *opcodes.cpp:1859*
10: *opcodes.cpp:1860*
11: *ConstantPool.h:74*
12: *Thread.h:150*
13: *ConstantPool.h:299*
14: *opcodes.cpp:1861*
15: *opcodes.cpp:1864*
16: *defs.h:125*
17: *ConstantPool.h:48*
18: *defs.h:29*
19: *ConstantPool.cpp:238*
20: *opcodes.cpp:1871*
21: *defs.h:33*
22: *Thread.cpp:776*
23: *defs.h:136*
24: *ConstantPool.h:90*
25: *ClassFile.h:37*
26: *defs.h:364*
27: *defs.h:101*
28: *defs.h:195*
29: *Thread.h:148*
30: *Stack.cpp:249*
31: *opcodes.cpp:1889*
32: *defs.h:201*
33: *defs.h:105*
34: *defs.h:138*
35: *ObjectData.h:113*
36: *Thread.h:179*
37: *VirtualMachine.h:336*
38: *HandlePool.cpp:25*
39: *opcodes.cpp:1896*
40: *ClassFile.h:68*
41: *opcodes.cpp:1901*
42: *ObjectData.h:71*
43: *HeapManager.h:46*
44: *ClassFile.h:79*

```
1905:
1906:                   // Push the field's value onto the operand stack;
1907:                   // Note, that the proper number of bytes to copy
1908:                   // will be taken from the memory by the corresponding function
1909:                   su1[1] byte_value;
1910:                   switch (finfo[2]->type[3])
1911:                   {
1912:                   case Byte[4]:
1913:                                   byte_value[5] = *(su1[1]*)field_start[6];
1914:                                   thread[7]->current_stack_frame[8]->push_int[9]((su4[10])byte_value[5]);
1915:                                   break;
1916:                   case Int[11]: case Char[12]: case Short[13]: case Boolean[14]:
1917:                                   thread[7]->current_stack_frame[8]->push_int[9](*(su4[10]*)field_start[6]);
1918:                                   break;
1919:                           case Double[15]:
1920:                                   thread[7]->current_stack_frame[8]->push_double[16](*(double*)field_start[6]);
1921:                                   break;
1922:                           case Float[17]:
1923:                                   thread[7]->current_stack_frame[8]->push_float[18](*(float*)field_start[6]);
1924:                                   break;
1925:                           case Long[19]:
1926:                                   thread[7]->current_stack_frame[8]->push_long[20](*(su8[21]*)field_start[6]);
1927:                                   break;
1928:                           case Array[22]: case Reference[23]:
1929:                                   thread[7]->current_stack_frame[8]->push_reference[24](*(word[25]*)field_start[6]);
1930:                                   break;
1931:                           default: return InstructionErrorCannotGetField[26];
1932:                   }
1933:
1934:           #ifdef DEBUG_EXECUTION[27]
1935:                   #ifdef DEBUG_INSTRUCTIONS
1936:                   finfo[2]->debug_print[28](debug_file);
1937:                   debug_file << "VALUE OF INSTANCE FIELD IS: ";
1938:                   switch (finfo[2]->type[3])
1939:                   {
1940:                   case Int[11]: case Byte[4]: case Char[12]: case Short[13]: case Boolean[14]:
1941:                                   debug_file << *(su4[10]*)field_start[6]; break;
1942:                           case Double[15]:
1943:                                   debug_file << *(double*)field_start[6]; break;
1944:                           case Float[17]:
1945:                                   debug_file << *(float*)field_start[6]; break;
1946:                           case Long[19]:
1947:                                   debug_file << *(long*)field_start[6]; break;
1948:                           case Array[22]: case Reference[23]:
1949:                                   debug_file << *(long*)field_start[6]; break;
1950:                   }
1951:                   debug_file << endl;
1952:                   #endif
1953:           #endif
1954:
1955:                   return InstructionSuccess[29];
1956:           }
1957:
```

Footnotes:

1: *defs.h:169*
2: *opcodes.cpp:1901*
3: *ClassFile.h:72*
4: *defs.h:210*
5: *opcodes.cpp:1909*
6: *opcodes.cpp:1904*
7: *opcodes.cpp:1856*
8: *Thread.h:148*
9: *Stack.cpp:147*
10: *defs.h:175*
11: *defs.h:214*
12: *defs.h:211*
13: *defs.h:217*
14: *defs.h:218*
15: *defs.h:212*
16: *Stack.cpp:171*
17: *defs.h:213*
18: *Stack.cpp:165*
19: *defs.h:215*
20: *Stack.cpp:153*
21: *defs.h:177*
22: *defs.h:219*
23: *defs.h:216*
24: *Stack.cpp:187*
25: *defs.h:195*
26: *defs.h:136*
27: *defs.h:428*
28: *ClassFile.cpp:132*
29: *defs.h:122*

```
1958:          InstructionResult[1] putfield(Thread[2] * thread)
1959:          {
1960:                  // Make a Constant Pool index
1961:                  u1[3] indexbyte1 = thread[4]->current_code[5][thread[4]->pc_register[6]++];
1962:                  u1[3] indexbyte2 = thread[4]->current_code[5][thread[4]->pc_register[6]++];
1963:                  u2[7] index = (indexbyte1[8] << 8) | indexbyte2[9];
1964:
1965:                  // Get the CONSTANT_Fieldref_info entry from the Constant Pool
1966:                  CONSTANT_Fieldref_info[10] * entry = (CONSTANT_Fieldref_info[10]*)thread[4]->current_cp[11]->get_item_at_ind
          ex[12](index[13]);
1967:                  if (!entry[14])
1968:                          return InstructionErrorWrongConstantPoolEntryIndex[15];
1969:
1970:                  // Check whether this entry is already resolved; if not - resolve it
1971:                  if (!entry[14]->is_resolved[16]())
1972:                  {
1973:                          Result[17] result = entry[14]->resolve[18]();
1974:                          if (result[19] != Success[20])
1975:                          {
1976:                                  thread[4]->raise_exception[21](result[19]);
1977:                                  return InstructionErrorCannotPutField[22];
1978:                          }
1979:                  }
1980:
1981:                  field_info[23] * finfo = entry[14]->finfo[24];
1982:                  assert(finfo[25] != NULL);
1983:
1984:                  // Check that the field is not static
1985:                  if (finfo[25]->access_flags[26] & ACC_STATIC[27])
1986:                  {
1987:                          thread[4]->raise_exception[21](VM_ERROR_IncompatibleClassChangeError[28]);
1988:                          return InstructionErrorCannotPutField[22];
1989:                  }
1990:
1991:                  // TODO: check access permissions
1992:
1993:                  // Pop the value from the operand stack;
1994:                  // The type of the value must correspond to the type of the field
1995:                  su4[29] int_value;
1996:                  double double_value;
1997:                  float float_value;
1998:                  su8[30] long_value;
1999:                  word[31] ref_value;
2000:                  switch (finfo[25]->type[32])
2001:                  {
2002:                  case Int[33]: case Byte[34]: case Char[35]: case Short[36]: case Boolean[37]:
2003:                                  int_value[38] = thread[4]->current_stack_frame[39]->pop_int[40]();
2004:                                  break;
2005:                          case Double[41]:
2006:                                  double_value[42] = thread[4]->current_stack_frame[39]->pop_double[43]();
2007:                                  break;
2008:                          case Float[44]:
2009:                                  float_value[45] = thread[4]->current_stack_frame[39]->pop_float[46]();
```

Footnotes:

1: *defs.h:120*
2: *Thread.h:17*
3: *defs.h:168*
4: *opcodes.cpp:1958*
5: *Thread.h:151*
6: *Thread.h:159*
7: *defs.h:172*
8: *opcodes.cpp:1961*
9: *opcodes.cpp:1962*
10: *ConstantPool.h:74*
11: *Thread.h:150*
12: *ConstantPool.h:299*
13: *opcodes.cpp:1963*
14: *opcodes.cpp:1966*
15: *defs.h:125*
16: *ConstantPool.h:48*
17: *defs.h:29*
18: *ConstantPool.cpp:238*
19: *opcodes.cpp:1973*
20: *defs.h:33*
21: *Thread.cpp:776*
22: *defs.h:137*
23: *ClassFile.h:68*
24: *ConstantPool.h:90*
25: *opcodes.cpp:1981*
26: *ClassFile.h:37*
27: *defs.h:364*
28: *defs.h:101*
29: *defs.h:175*
30: *defs.h:177*
31: *defs.h:195*
32: *ClassFile.h:72*
33: *defs.h:214*
34: *defs.h:210*
35: *defs.h:211*
36: *defs.h:217*
37: *defs.h:218*
38: *opcodes.cpp:1995*
39: *Thread.h:148*
40: *Stack.cpp:206*
41: *defs.h:212*
42: *opcodes.cpp:1996*
43: *Stack.cpp:235*
44: *defs.h:213*
45: *opcodes.cpp:1997*
46: *Stack.cpp:227*

```
2010:                                   break;
2011:                           case Long[1]:
2012:                                   long_value[2] = thread[3]->current_stack_frame[4]->pop_long[5]();
2013:                                   break;
2014:                           case Array[6]: case Reference[7]:
2015:                                   ref_value[8] = thread[3]->current_stack_frame[4]->pop_reference[9]();
2016:                                   break;
2017:                           default: return InstructionErrorCannotPutField[10];
2018:                   }
2019:
2020:
2021:                   // Now, pop the object reference from the operand stack
2022:                   word[11] objectref = thread[3]->current_stack_frame[4]->pop_reference[9]();
2023:                   if (objectref[12] == null[13])
2024:                   {
2025:                           thread[3]->raise_exception[14](VM_ERROR_NullPointerException[15]);
2026:                           return InstructionErrorNullPointerException[16];
2027:                   }
2028:
2029:                   // Get the class of the object whose field is to be accessed
2030:                   InstanceData[17] * id = thread[3]->get_jvm[18]()->get_handle_pool[19]()->get_instance[20](objectref[12]);
2031:                   assert(id[21] != NULL);
2032:
2033:                   // Get the memory corresponding to the field of the given instance
2034:                   // from the appropriate offset in the referenced instance memory block
2035:                   assert(id[21]->data_start[22]);
2036:                   u1[23] * field_start = id[21]->data_start[22]->address[24] + finfo[25]->offset[26];
2037:
2038:                   // Store the value popped from the stack into the memory
2039:                   // corresponding to the field of the given instance
2040:                   switch (finfo[25]->type[27])
2041:                   {
2042:                           case Int[28]:
2043:                                   memcpy(field_start[29], &int_value[30], BasicTypes[31][Int[28]].size[32]);
2044:                                   break;
2045:                           case Byte[33]:
2046:                                   memcpy(field_start[29], &int_value[30], BasicTypes[31][Byte[33]].size[32]);
2047:                                   break;
2048:                           case Char[34]:
2049:                                   memcpy(field_start[29], &int_value[30], BasicTypes[31][Char[34]].size[32]);
2050:                                   break;
2051:                           case Short[35]:
2052:                                   memcpy(field_start[29], &int_value[30], BasicTypes[31][Short[35]].size[32]);
2053:                                   break;
2054:                           case Boolean[36]:
2055:                                   memcpy(field_start[29], &int_value[30], BasicTypes[31][Boolean[36]].size[32]);
2056:                                   break;
2057:                           case Double[37]:
2058:                                   memcpy(field_start[29], &double_value[38], BasicTypes[31][Double[37]].size[32]);
2059:                                   break;
2060:                           case Float[39]:
2061:                                   memcpy(field_start[29], &float_value[40], BasicTypes[31][Float[39]].size[32]);
2062:                                   break;
```

Footnotes:

1: *defs.h:215*
2: *opcodes.cpp:1998*
3: *opcodes.cpp:1958*
4: *Thread.h:148*
5: *Stack.cpp:214*
6: *defs.h:219*
7: *defs.h:216*
8: *opcodes.cpp:1999*
9: *Stack.cpp:249*
10: *defs.h:137*
11: *defs.h:195*
12: *opcodes.cpp:2022*
13: *defs.h:201*
14: *Thread.cpp:776*
15: *defs.h:105*
16: *defs.h:138*
17: *ObjectData.h:113*
18: *Thread.h:179*
19: *VirtualMachine.h:336*
20: *HandlePool.cpp:25*
21: *opcodes.cpp:2030*
22: *ObjectData.h:71*
23: *defs.h:168*
24: *HeapManager.h:46*
25: *opcodes.cpp:1981*
26: *ClassFile.h:79*
27: *ClassFile.h:72*
28: *defs.h:214*
29: *opcodes.cpp:2036*
30: *opcodes.cpp:1995*
31: *defs.h:246*
32: *defs.h:227*
33: *defs.h:210*
34: *defs.h:211*
35: *defs.h:217*
36: *defs.h:218*
37: *defs.h:212*
38: *opcodes.cpp:1996*
39: *defs.h:213*
40: *opcodes.cpp:1997*

```
2063:                         case Long[1]:
2064:                                 memcpy(field_start[2], &long_value[3], BasicTypes[4][Long[1]].size[5]);
2065:                                 break;
2066:                         case Array[6]: case Reference[7]:
2067:                                 memcpy(field_start[2], &ref_value[8], sizeof(word[9]));
2068:                                 break;
2069:                         default: return InstructionErrorCannotPutField[10];
2070:                 }
2071:
2072:         #ifdef DEBUG_EXECUTION[11]
2073:                 #ifdef DEBUG_INSTRUCTIONS
2074:                 finfo[12]->debug_print[13](debug_file);
2075:
2076:                 debug_file << "VALUE ASSIGNED TO INSTANCE FIELD: ";
2077:                 switch (finfo[12]->type[14])
2078:                 {
2079:                 case Int[15]: case Byte[16]: case Char[17]: case Short[18]: case Boolean[19]:
2080:                                 debug_file << int_value[20]; break;
2081:                         case Double[21]:
2082:                                 debug_file << double_value[22]; break;
2083:                         case Float[23]:
2084:                                 debug_file << float_value[24]; break;
2085:                         case Long[1]:
2086:                                 debug_file << (long)long_value[3]; break;
2087:                         case Array[6]: case Reference[7]:
2088:                                 debug_file << (long)ref_value[8]; break;
2089:                 }
2090:                 debug_file << endl;
2091:                 #endif
2092:         #endif
2093:
2094:
2095:                 return InstructionSuccess[25];
2096:         }
2097:
2098:
2099:         //-------------------------------------------------------------------------------
2100:         // Heap operations
2101:         //-------------------------------------------------------------------------------
2102:
2103:         // NOTE: this operation might involve the massive resolution tree of the types
2104:         // and cause a significant delay in this thread processing
2105:         InstructionResult[26] _new(Thread[27] * thread)
2106:         {
2107:                 // Make a Constant Pool index
2108:                 u1[28] indexbyte1 = thread[29]->current_code[30][thread[29]->pc_register[31]++];
2109:                 u1[28] indexbyte2 = thread[29]->current_code[30][thread[29]->pc_register[31]++];
2110:                 u2[32] index = (indexbyte1[33] << 8) | indexbyte2[34];
2111:
2112:                 // Get the CONSTANT_Class_info entry from the Constant Pool
2113:                 CONSTANT_Class_info[35] * entry = (CONSTANT_Class_info[35]*)thread[29]->current_cp[36]->get_item_at_index[37](in
dex[38]);
2114:                 if (!entry[39])
```

Footnotes:

1: *defs.h:215*
2: *opcodes.cpp:2036*
3: *opcodes.cpp:1998*
4: *defs.h:246*
5: *defs.h:227*
6: *defs.h:219*
7: *defs.h:216*
8: *opcodes.cpp:1999*
9: *defs.h:195*
10: *defs.h:137*
11: *defs.h:428*
12: *opcodes.cpp:1981*
13: *ClassFile.cpp:132*
14: *ClassFile.h:72*
15: *defs.h:214*
16: *defs.h:210*
17: *defs.h:211*
18: *defs.h:217*
19: *defs.h:218*
20: *opcodes.cpp:1995*
21: *defs.h:212*
22: *opcodes.cpp:1996*
23: *defs.h:213*
24: *opcodes.cpp:1997*
25: *defs.h:122*
26: *defs.h:120*
27: *Thread.h:17*
28: *defs.h:168*
29: *opcodes.cpp:2105*
30: *Thread.h:151*
31: *Thread.h:159*
32: *defs.h:172*
33: *opcodes.cpp:2108*
34: *opcodes.cpp:2109*
35: *ConstantPool.h:58*
36: *Thread.h:150*
37: *ConstantPool.h:299*
38: *opcodes.cpp:2110*
39: *opcodes.cpp:2113*

```
2115:                   return InstructionErrorWrongConstantPoolEntryIndex[1];
2116:
2117:           // Check whether this entry is already resolved; if not - resolve it
2118:           if (!entry[2]->is_resolved[3]())
2119:           {
2120:                   Result[4] result = entry[2]->resolve[5]();
2121:                   if (result[6] != Success[7])
2122:                   {
2123:                           thread[8]->raise_exception[9](result[6]);
2124:                           return InstructionErrorCannotCreateInstance[10];
2125:                   }
2126:           }
2127:
2128:           assert(entry[2]->resolved_class_file[11] != NULL);
2129:
2130:           // Create the new instance of this class
2131:           word[12] objectref = entry[2]->resolved_class_file[11]->create_new_instance[13]();
2132:           if (objectref[14] == null[15])
2133:                   return InstructionErrorCannotCreateInstance[10];
2134:
2135:           // Push the reference to the object just created onto the stack
2136:           thread[8]->current_stack_frame[16]->push_reference[17](objectref[14]);
2137:
2138:           // NOTE: the constructor of the instance being created will be called as the
2139:           // following (in a while) invokespecial instruction
2140:
2141:   #ifdef DEBUG_EXECUTION[18]
2142:           #ifdef DEBUG_INSTRUCTIONS
2143:           InstanceData[19] * id = thread[8]->get_jvm[20]()->get_handle_pool()->get_instance(objectref[14]);
2144:           id->debug_print(debug_file);
2145:           #endif
2146:   #endif
2147:
2148:           return InstructionSuccess[21];
2149:   }
2150:
2151:   InstructionResult[22] anewarray(Thread[23] * thread)
2152:   {
2153:           su4[24] count = thread[25]->current_stack_frame[16]->pop_int[26]();
2154:           if (count[27] < 0)
2155:           {
2156:                   thread[25]->raise_exception[9](VM_ERROR_NegativeArraySizeException[28]);
2157:                   return InstructionErrorCannotCreateArray[29];
2158:           }
2159:
2160:           // Make a Constant Pool index
2161:           u1[30] indexbyte1 = thread[25]->current_code[31][thread[25]->pc_register[32]++];
2162:           u1[30] indexbyte2 = thread[25]->current_code[31][thread[25]->pc_register[32]++];
2163:           u2[33] index = (indexbyte1[34] << 8) | indexbyte2[35];
2164:
2165:           // Get the CONSTANT_Class_info entry from the Constant Pool
2166:           CONSTANT_Class_info[36] * entry = (CONSTANT_Class_info[36]*)thread[25]->current_cp[37]->get_item_at_index[38](in
      dex[39]);
```

Footnotes:

1: *defs.h:125*
2: *opcodes.cpp:2113*
3: *ConstantPool.h:48*
4: *defs.h:29*
5: *ConstantPool.cpp:22*
6: *opcodes.cpp:2120*
7: *defs.h:33*
8: *opcodes.cpp:2105*
9: *Thread.cpp:776*
10: *defs.h:128*
11: *ConstantPool.h:71*
12: *defs.h:195*
13: *ClassFile.cpp:870*
14: *opcodes.cpp:2131*
15: *defs.h:201*
16: *Thread.h:148*
17: *Stack.cpp:187*
18: *defs.h:428*
19: *ObjectData.h:113*
20: *Thread.h:179*
21: *defs.h:122*
22: *defs.h:120*
23: *Thread.h:17*
24: *defs.h:175*
25: *opcodes.cpp:2151*
26: *Stack.cpp:206*
27: *opcodes.cpp:2153*
28: *defs.h:107*
29: *defs.h:129*
30: *defs.h:168*
31: *Thread.h:151*
32: *Thread.h:159*
33: *defs.h:172*
34: *opcodes.cpp:2161*
35: *opcodes.cpp:2162*
36: *ConstantPool.h:58*
37: *Thread.h:150*
38: *ConstantPool.h:299*
39: *opcodes.cpp:2163*

```
2167:                   if (!entry[1])
2168:                           return InstructionErrorWrongConstantPoolEntryIndex[2];
2169:
2170:                   // Check whether this entry is already resolved; if not - resolve it
2171:                   if (!entry[1]->is_resolved[3]())
2172:                   {
2173:                           Result[4] result = entry[1]->resolve[5]();
2174:                           if (result[6] != Success[7])
2175:                           {
2176:                                   thread[8]->raise_exception[9](result[6]);
2177:                                   return InstructionErrorCannotCreateArray[10];
2178:                           }
2179:                   }
2180:
2181:                   assert(entry[1]->resolved_class_file[11] != NULL);
2182:
2183:                   // Create the new instance of the array of reference values.
2184:                   // (The name of the class is the descriptor of the array type, for exmple
2185:                   // for String[] the name of the class is "[java/lang/String;")
2186:                   ClassFile[12] * array_class_file;
2187:                   // Note that the class loader of this array type will be the class loader
2188:                   // that has loaded the type of the array element
2189:                   assert(entry[1]->resolved_class_file[11]->class_loader[13] != NULL);
2190:                   entry[1]->resolved_class_file[11]->class_loader[13]->define_array_class[14](entry[1]->resolved_class_file[11], arr
          ay_class_file[15]);
2191:
2192:                   word[16] objectref = ((ArrayClassFile[17]*)array_class_file[15])->create_new_instance[18](count[19]);
2193:                   if (objectref[20] == null[21])
2194:                           return InstructionErrorCannotCreateArray[10];
2195:
2196:                   // Push the reference to the object just created onto the stack
2197:                   thread[8]->current_stack_frame[22]->push_reference[23](objectref[20]);
2198:
2199:                   return InstructionSuccess[24];
2200:           }
2201:
2202:           InstructionResult[25] newarray(Thread[26] * thread)
2203:           {
2204:                   su4[27] count = thread[28]->current_stack_frame[22]->pop_int[29]();
2205:                   if (count[30] < 0)
2206:                   {
2207:                           thread[28]->raise_exception[9](VM_ERROR_NegativeArraySizeException[31]);
2208:                           return InstructionErrorCannotCreateArray[10];
2209:                   }
2210:
2211:                   u1[32] array_type = thread[28]->current_code[33][thread[28]->pc_register[34]++];
2212:
2213:                   // Create the new instance of the array of primitive values.
2214:                   // (The name of the class is the descriptor of the array type, for exmple
2215:                   // for int[] the name of the class is "[I")
2216:                   // The classes for arrays of primitive types will always be created by the bootstrap
2217:                   // class loaders
2218:                   ClassFile[12] * array_class_file;
```

Footnotes:

1: *opcodes.cpp:2166*
2: *defs.h:125*
3: *ConstantPool.h:48*
4: *defs.h:29*
5: *ConstantPool.cpp:22*
6: *opcodes.cpp:2173*
7: *defs.h:33*
8: *opcodes.cpp:2151*
9: *Thread.cpp:776*
10: *defs.h:129*
11: *ConstantPool.h:71*
12: *ClassFile.h:136*
13: *ClassFile.h:222*
14: *ClassLoader.cpp:387*
15: *opcodes.cpp:2186*
16: *defs.h:195*
17: *ClassFile.h:341*
18: *ClassFile.cpp:992*
19: *opcodes.cpp:2153*
20: *opcodes.cpp:2192*
21: *defs.h:201*
22: *Thread.h:148*
23: *Stack.cpp:187*
24: *defs.h:122*
25: *defs.h:120*
26: *Thread.h:17*
27: *defs.h:175*
28: *opcodes.cpp:2202*
29: *Stack.cpp:206*
30: *opcodes.cpp:2204*
31: *defs.h:107*
32: *defs.h:168*
33: *Thread.h:151*
34: *Thread.h:159*

```
2219:                   thread[1]->get_jvm[2]()->get_bootstrap_class_loader[3]()->define_primitive_array_class[4](array_type[5], arra
                y_class_file[6]);
2220:
2221:                   word[7] objectref = ((PrimitiveArrayClassFile[8]*)array_class_file[6])->create_new_instance[9](count[10]);
2222:                   if (objectref[11] == null[12])
2223:                           return InstructionErrorCannotCreateArray[13];
2224:
2225:                   // Push the reference to the object just created onto the stack
2226:                   thread[1]->current_stack_frame[14]->push_reference[15](objectref[11]);
2227:
2228:                   return InstructionSuccess[16];
2229:           }
2230:
2231:           //--------------------------------------------------------------------------------
2232:           // Method invocation and return operations
2233:           //--------------------------------------------------------------------------------
2234:
2235:           InstructionResult[17] invokevirtual(Thread[18] * thread)
2236:           {
2237:                   // Make a Constant Pool index
2238:                   u1[19] indexbyte1 = thread[20]->current_code[21][thread[20]->pc_register[22]++];
2239:                   u1[19] indexbyte2 = thread[20]->current_code[21][thread[20]->pc_register[22]++];
2240:                   u2[23] index = (indexbyte1[24] << 8) | indexbyte2[25];
2241:
2242:                   // Get the CONSTANT_Methodref_info entry from the Constant Pool
2243:                   CONSTANT_Methodref_info[26] * entry = (CONSTANT_Methodref_info[26]*)thread[20]->current_cp[27]->get_item_at_i
                ndex[28](index[29]);
2244:                   if (!entry[30])
2245:                           return InstructionErrorWrongConstantPoolEntryIndex[31];
2246:
2247:                   Result[32] result = thread[20]->invoke_instance_method[33](entry[30]);
2248:                   if (result[34] != Success[35])
2249:                           return InstructionErrorCannotInvokeVirtual[36];
2250:
2251:                   return InstructionSuccess[16];
2252:           }
2253:
2254:           InstructionResult[17] invokestatic(Thread[18] * thread)
2255:           {
2256:                   // Make a Constant Pool index
2257:                   u1[19] indexbyte1 = thread[37]->current_code[21][thread[37]->pc_register[22]++];
2258:                   u1[19] indexbyte2 = thread[37]->current_code[21][thread[37]->pc_register[22]++];
2259:                   u2[23] index = (indexbyte1[38] << 8) | indexbyte2[39];
2260:
2261:                   // Get the CONSTANT_Methodref_info entry from the Constant Pool
2262:                   CONSTANT_Methodref_info[26] * entry = (CONSTANT_Methodref_info[26]*)thread[37]->current_cp[27]->get_item_at_i
                ndex[28](index[40]);
2263:                   if (!entry[41])
2264:                           return InstructionErrorWrongConstantPoolEntryIndex[31];
2265:
2266:                   Result[32] result = thread[37]->invoke_static_method[42](entry[41]);
2267:                   if (result[43] != Success[35])
2268:                           return InstructionErrorCannotInvokeStatic[44];
```

Footnotes:

1: *opcodes.cpp:2202*
2: *Thread.h:179*
3: *VirtualMachine.h:334*
4: *ClassLoader.cpp:341*
5: *opcodes.cpp:2211*
6: *opcodes.cpp:2218*
7: *defs.h:195*
8: *ClassFile.h:370*
9: *ClassFile.cpp:1012*
10: *opcodes.cpp:2204*
11: *opcodes.cpp:2221*
12: *defs.h:201*
13: *defs.h:129*
14: *Thread.h:148*
15: *Stack.cpp:187*
16: *defs.h:122*
17: *defs.h:120*
18: *Thread.h:17*
19: *defs.h:168*
20: *opcodes.cpp:2235*
21: *Thread.h:151*
22: *Thread.h:159*
23: *defs.h:172*
24: *opcodes.cpp:2238*
25: *opcodes.cpp:2239*
26: *ConstantPool.h:98*
27: *Thread.h:150*
28: *ConstantPool.h:299*
29: *opcodes.cpp:2240*
30: *opcodes.cpp:2243*
31: *defs.h:125*
32: *defs.h:29*
33: *Thread.cpp:342*
34: *opcodes.cpp:2247*
35: *defs.h:33*
36: *defs.h:133*
37: *opcodes.cpp:2254*
38: *opcodes.cpp:2257*
39: *opcodes.cpp:2258*
40: *opcodes.cpp:2259*
41: *opcodes.cpp:2262*
42: *Thread.cpp:457*
43: *opcodes.cpp:2266*
44: *defs.h:132*

```
2269:
2270:                   return InstructionSuccess[1];
2271:           }
2272:
2273:           InstructionResult[2] invokespecial(Thread[3] * thread)
2274:           {
2275:                   // Make a Constant Pool index
2276:                   u1[4] indexbyte1 = thread[5]->current_code[6][thread[5]->pc_register[7]++];
2277:                   u1[4] indexbyte2 = thread[5]->current_code[6][thread[5]->pc_register[7]++];
2278:                   u2[8] index = (indexbyte1[9] << 8) | indexbyte2[10];
2279:
2280:                   // Get the CONSTANT_Methodref_info entry from the Constant Pool
2281:                   CONSTANT_Methodref_info[11] * entry = (CONSTANT_Methodref_info[11]*)thread[5]->current_cp[12]->get_item_at_i
           ndex[13](index[14]);
2282:                   if (!entry[15])
2283:                           return InstructionErrorWrongConstantPoolEntryIndex[16];
2284:
2285:                   Result[17] result = thread[5]->invoke_special_method[18](entry[15]);
2286:                   if (result[19] != Success[20])
2287:                           return InstructionErrorCannotInvokeSpecial[21];
2288:
2289:                   return InstructionSuccess[1];
2290:           }
2291:
2292:           InstructionResult[2] invokeinterface(Thread[3] * thread)
2293:           {
2294:                   // Make a Constant Pool index
2295:                   u1[4] indexbyte1 = thread[22]->current_code[6][thread[22]->pc_register[7]++];
2296:                   u1[4] indexbyte2 = thread[22]->current_code[6][thread[22]->pc_register[7]++];
2297:                   u2[8] index = (indexbyte1[23] << 8) | indexbyte2[24];
2298:
2299:                   // "count" is kept by historical reasons but we will make a use of it;
2300:                   // in fact, this number allows us not to resolve the CONSTANT_InterfaceMethodref
2301:                   // just to get the number of arguments, but rather to resolve it together with
2302:                   // the given object reference. This object reference can be retrieved from the
2303:                   // frame's local variables by the help of the "count" parameter
2304:                   u1[4] count = thread[22]->current_code[6][thread[22]->pc_register[7]++];
2305:                   u1[4] nothing = thread[22]->current_code[6][thread[22]->pc_register[7]++]; // just skip the fourth operand
2306:
2307:                   // Get the CONSTANT_InterfaceMethodref_info entry from the Constant Pool
2308:                   CONSTANT_InterfaceMethodref_info[25] * entry = (CONSTANT_InterfaceMethodref_info[25]*)thread[22]->current_
           cp[12]->get_item_at_index[13](index[26]);
2309:                   if (!entry[27])
2310:                           return InstructionErrorWrongConstantPoolEntryIndex[16];
2311:
2312:                   Result[17] result = thread[22]->invoke_interface_method[28](entry[27], count[29]);
2313:                   if (result[30] != Success[20])
2314:                           return InstructionErrorCannotInvokeInterface[31];
2315:
2316:                   return InstructionSuccess[1];
2317:           }
2318:
2319:           InstructionResult[2] _return(Thread[3] * thread)
```

Footnotes:

1: *defs.h:122*
2: *defs.h:120*
3: *Thread.h:17*
4: *defs.h:168*
5: *opcodes.cpp:2273*
6: *Thread.h:151*
7: *Thread.h:159*
8: *defs.h:172*
9: *opcodes.cpp:2276*
10: *opcodes.cpp:2277*
11: *ConstantPool.h:98*
12: *Thread.h:150*
13: *ConstantPool.h:299*
14: *opcodes.cpp:2278*
15: *opcodes.cpp:2281*
16: *defs.h:125*
17: *defs.h:29*
18: *Thread.cpp:542*
19: *opcodes.cpp:2285*
20: *defs.h:33*
21: *defs.h:131*
22: *opcodes.cpp:2292*
23: *opcodes.cpp:2295*
24: *opcodes.cpp:2296*
25: *ConstantPool.h:156*
26: *opcodes.cpp:2297*
27: *opcodes.cpp:2308*
28: *Thread.cpp:644*
29: *opcodes.cpp:2304*
30: *opcodes.cpp:2312*
31: *defs.h:130*

```
2320:         {
2321:                 Result[1] result = thread[2]->return_from_method[3]();
2322:                 return InstructionSuccess[4];
2323:         }
2324:
2325:         InstructionResult[5] ireturn(Thread[6] * thread)
2326:         {
2327:                 Result[1] result = thread[7]->return_int_from_method[8]();
2328:                 return InstructionSuccess[4];
2329:         }
2330:
2331:         InstructionResult[5] lreturn(Thread[6] * thread)
2332:         {
2333:                 Result[1] result = thread[9]->return_long_from_method[10]();
2334:                 return InstructionSuccess[4];
2335:         }
2336:
2337:         InstructionResult[5] freturn(Thread[6] * thread)
2338:         {
2339:                 Result[1] result = thread[11]->return_float_from_method[12]();
2340:                 return InstructionSuccess[4];
2341:         }
2342:
2343:         InstructionResult[5] dreturn(Thread[6] * thread)
2344:         {
2345:                 Result[1] result = thread[13]->return_double_from_method[14]();
2346:                 return InstructionSuccess[4];
2347:         }
2348:
2349:         InstructionResult[5] areturn(Thread[6] * thread)
2350:         {
2351:                 Result[1] result = thread[15]->return_reference_from_method[16]();
2352:                 return InstructionSuccess[4];
2353:         }
2354:
2355:         //-----------------------------------------------------------------------------
2356:         // Exception handling
2357:         //-----------------------------------------------------------------------------
2358:
2359:         InstructionResult[5] athrow(Thread[6] * thread)
2360:         {
2361:                 word[17] objectref = thread[18]->current_stack_frame[19]->pop_reference[20]();
2362:                 // If the reference is null throw the NullPointerException
2363:                 if (objectref[21] == null[22])
2364:                 {
2365:                         thread[18]->raise_exception[23](VM_ERROR_NullPointerException[24]);
2366:                         return InstructionSuccess[4];
2367:                 }
2368:
2369:                 InstanceData[25] * id = thread[18]->get_jvm[26]()->get_handle_pool[27]()->get_instance[28](objectref[21]);
2370:                 assert(id[29] != NULL);
2371:
2372:                 // Load the java.lang.Throwable class if not yet loaded
```

Footnotes:

1: *defs.h:29*
2: *opcodes.cpp:2319*
3: *Thread.cpp:142*
4: *defs.h:122*
5: *defs.h:120*
6: *Thread.h:17*
7: *opcodes.cpp:2325*
8: *Thread.cpp:170*
9: *opcodes.cpp:2331*
10: *Thread.cpp:204*
11: *opcodes.cpp:2337*
12: *Thread.cpp:238*
13: *opcodes.cpp:2343*
14: *Thread.cpp:272*
15: *opcodes.cpp:2349*
16: *Thread.cpp:306*
17: *defs.h:195*
18: *opcodes.cpp:2359*
19: *Thread.h:148*
20: *Stack.cpp:249*
21: *opcodes.cpp:2361*
22: *defs.h:201*
23: *Thread.cpp:776*
24: *defs.h:105*
25: *ObjectData.h:113*
26: *Thread.h:179*
27: *VirtualMachine.h:336*
28: *HandlePool.cpp:25*
29: *opcodes.cpp:2369*

```
2373:                   ClassFile[1] * throwable_cf = thread[2]->get_jvm[3]()->get_bootstrap_class_loader[4]()->get_class[5](THROWABL
                E_CLASS_NAME[6], wcslen(THROWABLE_CLASS_NAME[6]));
2374:                   if (!throwable_cf[7])
2375:                           throwable_cf[7] = thread[2]->get_jvm[3]()->get_bootstrap_class_loader[4]()->load_class[8](THROWABLE_C
                LASS_NAME[6], wcslen(THROWABLE_CLASS_NAME[6]));
2376:                   assert(throwable_cf[7] != NULL);
2377:
2378:                   // Make sure that the class of the given exception instance inherits from the Throwable class
2379:                   if (! id[9]->class_file[10]->inherits[11](throwable_cf[7]))
2380:                           return InstructionErrorExceptionIsNotThrowableSubclass[12];
2381:
2382:                   // "Artificially" raise the exception - all the rest will be automatically done
2383:                   thread[2]->raise_exception[13](id[9]->class_file[10]);
2384:
2385:                   return InstructionSuccess[14];
2386:           }
2387:
2388:           InstructionResult[15] jsr(Thread[16] * thread)
2389:           {
2390:                   u1[17] indexbyte1 = thread[18]->current_code[19][thread[18]->pc_register[20]++];
2391:                   u1[17] indexbyte2 = thread[18]->current_code[19][thread[18]->pc_register[20]++];
2392:                   su2[21] offset = (indexbyte1[22] << 8) | indexbyte2[23];
2393:
2394:                   // Push the next opcode address onto the stack frame as a returnAddress
2395:                   thread[18]->current_stack_frame[24]->push_returnType[25](thread[18]->pc_register[20]);
2396:
2397:                   // Continue execution from the given offset
2398:                   // Step back to the opcode of this instruction
2399:                   thread[18]->pc_register[20] -= 3;
2400:                   // Add the offset
2401:                   thread[18]->pc_register[20] += offset[26];
2402:
2403:                   // Check that the next instruction address is within this method's code
2404:                   if (thread[18]->pc_register[20] >= thread[18]->current_code_length[27] || thread[18]->pc_register[20] < 0)
2405:                           return InstructionErrorBranchOutsideMethodCode[28];
2406:
2407:                   return InstructionSuccess[14];
2408:           }
2409:
2410:           InstructionResult[15] ret(Thread[16] * thread)
2411:           {
2412:                   u1[17] indexbyte = thread[29]->current_code[19][thread[29]->pc_register[20]++];
2413:                   thread[29]->pc_register[20] = thread[29]->current_stack_frame[24]->get_returnType[30](indexbyte[31]);
2414:
2415:                   return InstructionSuccess[14];
2416:           }
2417:
2418:           //-------------------------------------------------------------------------------
2419:           // Synchronization operations
2420:           //-------------------------------------------------------------------------------
2421:
2422:           InstructionResult[15] monitorenter(Thread[16] * thread)
2423:           {
```

Footnotes:

1: *ClassFile.h:136*
2: *opcodes.cpp:2359*
3: *Thread.h:179*
4: *VirtualMachine.h:334*
5: *ClassLoader.cpp:187*
6: *defs.h:255*
7: *opcodes.cpp:2373*
8: *ClassLoader.cpp:18*
9: *opcodes.cpp:2369*
10: *ObjectData.h:74*
11: *ClassFile.cpp:1061*
12: *defs.h:146*
13: *Thread.cpp:776*
14: *defs.h:122*
15: *defs.h:120*
16: *Thread.h:17*
17: *defs.h:168*
18: *opcodes.cpp:2388*
19: *Thread.h:151*
20: *Thread.h:159*
21: *defs.h:173*
22: *opcodes.cpp:2390*
23: *opcodes.cpp:2391*
24: *Thread.h:148*
25: *Stack.cpp:201*
26: *opcodes.cpp:2392*
27: *Thread.h:152*
28: *defs.h:127*
29: *opcodes.cpp:2410*
30: *Stack.cpp:403*
31: *opcodes.cpp:2412*

```
2424:                   // Pop the object reference from the operand stack
2425:                   word[1] objectref = thread[2]->current_stack_frame[3]->pop_reference[4]();
2426:                   if (objectref[5] == null[6])
2427:                   {
2428:                           thread[2]->raise_exception[7](VM_ERROR_NullPointerException[8]);
2429:                           return InstructionErrorNullPointerException[9];
2430:                   }
2431:
2432:                   // Get the object whose monitor is to be acquired
2433:                   InstanceData[10] * id = thread[2]->get_jvm[11]()->get_handle_pool[12]()->get_instance[13](objectref[5]);
2434:                   assert(id[14] != NULL);
2435:
2436:                   // Try to acquire the monitor associated with this object
2437:                   id[14]->lock[15].acquire[16](thread[2]);
2438:
2439:                   // We are completely unaware at this moment whether the thread succeeded to
2440:                   // acquire the monitor and continue running or it has failed and put asleep
2441:
2442:                   return InstructionSuccess[17];
2443:           }
2444:
2445:           InstructionResult[18] monitorexit(Thread[19] * thread)
2446:           {
2447:                   // Pop the object reference from the operand stack
2448:                   word[1] objectref = thread[20]->current_stack_frame[3]->pop_reference[4]();
2449:                   if (objectref[21] == null[6])
2450:                   {
2451:                           thread[20]->raise_exception[7](VM_ERROR_NullPointerException[8]);
2452:                           return InstructionErrorNullPointerException[9];
2453:                   }
2454:
2455:                   // Get the object whose monitor is to be released
2456:                   InstanceData[10] * id = thread[20]->get_jvm[11]()->get_handle_pool[12]()->get_instance[13](objectref[21]);
2457:                   assert(id[22] != NULL);
2458:
2459:                   // Try to acquire the monitor associated with this object
2460:                   if (!id[22]->lock[15].release[23](thread[20]))
2461:                   {
2462:                           // This thread does not own the monitor it is trying to release
2463:                           thread[20]->raise_exception[7](VM_ERROR_IllegalMonitorStateException[24]);
2464:                           return InstructionErrorMonitorNotOwned[25];
2465:                   }
2466:
2467:
2468:                   return InstructionSuccess[17];
2469:           }
2470:
2471:           //-----------------------------------------------------------------------------
2472:           // Array operations
2473:           //-----------------------------------------------------------------------------
2474:
2475:           InstructionResult[18] iaload(Thread[19] * thread)
2476:           {
```

Footnotes:

1: *defs.h:195*
2: *opcodes.cpp:2422*
3: *Thread.h:148*
4: *Stack.cpp:249*
5: *opcodes.cpp:2425*
6: *defs.h:201*
7: *Thread.cpp:776*
8: *defs.h:105*
9: *defs.h:138*
10: *ObjectData.h:113*
11: *Thread.h:179*
12: *VirtualMachine.h:336*
13: *HandlePool.cpp:25*
14: *opcodes.cpp:2433*
15: *ObjectData.h:77*
16: *ObjectData.cpp:447*
17: *defs.h:122*
18: *defs.h:120*
19: *Thread.h:17*
20: *opcodes.cpp:2445*
21: *opcodes.cpp:2448*
22: *opcodes.cpp:2456*
23: *ObjectData.cpp:499*
24: *defs.h:106*
25: *defs.h:139*

```
2477:                   stack_frame[1] * frame = thread[2]->current_stack_frame[3];
2478:
2479:                   // Pop the index
2480:                   su4[4] index = frame[5]->pop_int[6]();
2481:
2482:                   // Pop the array reference
2483:                   word[7] arrayref = frame[5]->pop_reference[8]();
2484:                   if (arrayref[9] == null[10])
2485:                   {
2486:                           thread[2]->raise_exception[11](VM_ERROR_NullPointerException[12]);
2487:                           return InstructionErrorNullPointerException[13];
2488:                   }
2489:
2490:                   // Get the array object
2491:                   InstanceData[14] * id = thread[2]->get_jvm[15]()->get_handle_pool[16]()->get_instance[17](arrayref[9]);
2492:                   assert(id[18] != NULL);
2493:
2494:                   // Check the array bounds
2495:                   u4[19] array_length = ((PrimitiveArrayInstanceData[20]*)id[18])->get_array_length[21]();
2496:                   if (index[22] < 0 || index[22] >= array_length[23])
2497:                   {
2498:                           thread[2]->raise_exception[11](VM_ERROR_ArrayIndexOutOfBoundsException[24]);
2499:                           return InstructionErrorWrongArrayIndex[25];
2500:                   }
2501:
2502:                   // Check that the array type is integer
2503:                   ClassFile[26] * cf = id[18]->class_file[27];
2504:                   if (((PrimitiveArrayClassFile[28]*)cf[29])->get_array_type[30]() != T_INT[31])
2505:                           return InstructionErrorWrongArrayType[32];
2506:
2507:                   // Get the value of the element
2508:                   su4[4] int_value;
2509:                   unsigned short element_size = BasicTypes[33][Int[34]].size[35];
2510:                   memcpy(&int_value[36], id[18]->data_start[37]->address[38] + index[22]*element_size[39], element_size[39]);
2511:
2512:                   // Push the value onto the stack
2513:                   frame[5]->push_int[40](int_value[36]);
2514:
2515:                   return InstructionSuccess[41];
2516:           }
2517:
2518:           InstructionResult[42] iastore(Thread[43] * thread)
2519:           {
2520:                   stack_frame[1] * frame = thread[44]->current_stack_frame[3];
2521:
2522:                   // Pop the value
2523:                   su4[4] int_value = frame[45]->pop_int[6]();
2524:
2525:                   // Pop the index
2526:                   su4[4] index = frame[45]->pop_int[6]();
2527:
2528:                   // Pop the array reference
2529:                   word[7] arrayref = frame[45]->pop_reference[8]();
```

Footnotes:

1: *Stack.h:129*
2: *opcodes.cpp:2475*
3: *Thread.h:148*
4: *defs.h:175*
5: *opcodes.cpp:2477*
6: *Stack.cpp:206*
7: *defs.h:195*
8: *Stack.cpp:249*
9: *opcodes.cpp:2483*
10: *defs.h:201*
11: *Thread.cpp:776*
12: *defs.h:105*
13: *defs.h:138*
14: *ObjectData.h:113*
15: *Thread.h:179*
16: *VirtualMachine.h:336*
17: *HandlePool.cpp:25*
18: *opcodes.cpp:2491*
19: *defs.h:174*
20: *ObjectData.h:174*
21: *ObjectData.h:163*
22: *opcodes.cpp:2480*
23: *opcodes.cpp:2495*
24: *defs.h:108*
25: *defs.h:140*
26: *ClassFile.h:136*
27: *ObjectData.h:74*
28: *ClassFile.h:370*
29: *opcodes.cpp:2503*
30: *ClassFile.h:395*
31: *defs.h:346*
32: *defs.h:141*
33: *defs.h:246*
34: *defs.h:214*
35: *defs.h:227*
36: *opcodes.cpp:2508*
37: *ObjectData.h:71*
38: *HeapManager.h:46*
39: *opcodes.cpp:2509*
40: *Stack.cpp:147*
41: *defs.h:122*
42: *defs.h:120*
43: *Thread.h:17*
44: *opcodes.cpp:2518*
45: *opcodes.cpp:2520*

```
2530:                   if (arrayref[1] == null[2])
2531:                   {
2532:                           thread[3]->raise_exception[4](VM_ERROR_NullPointerException[5]);
2533:                           return InstructionErrorNullPointerException[6];
2534:                   }
2535:
2536:                   // Get the array object
2537:                   InstanceData[7] * id = thread[3]->get_jvm[8]()->get_handle_pool[9]()->get_instance[10](arrayref[1]);
2538:                   assert(id[11] != NULL);
2539:
2540:                   // Check the array bounds
2541:                   u4[12] array_length = ((PrimitiveArrayInstanceData[13]*)id[11])->get_array_length[14]();
2542:                   if (index[15] < 0 || index[15] >= array_length[16])
2543:                   {
2544:                           thread[3]->raise_exception[4](VM_ERROR_ArrayIndexOutOfBoundsException[17]);
2545:                           return InstructionErrorWrongArrayIndex[18];
2546:                   }
2547:
2548:                   // Check that the array type is integer
2549:                   ClassFile[19] * cf = id[11]->class_file[20];
2550:                   if (((PrimitiveArrayClassFile[21]*)cf[22])->get_array_type[23]() != T_INT[24])
2551:                           return InstructionErrorWrongArrayType[25];
2552:
2553:                   // Assign the value to the array element
2554:                   unsigned short element_size = BasicTypes[26][Int[27]].size[28];
2555:                   memcpy(id[11]->data_start[29]->address[30] + index[15]*element_size[31], &int_value[32], element_size[31]);
2556:
2557:                   return InstructionSuccess[33];
2558:           }
2559:
2560:           InstructionResult[34] saload(Thread[35] * thread)
2561:           {
2562:                   stack_frame[36] * frame = thread[37]->current_stack_frame[38];
2563:
2564:                   // Pop the index
2565:                   su4[39] index = frame[40]->pop_int[41]();
2566:
2567:                   // Pop the array reference
2568:                   word[42] arrayref = frame[40]->pop_reference[43]();
2569:                   if (arrayref[44] == null[2])
2570:                   {
2571:                           thread[37]->raise_exception[4](VM_ERROR_NullPointerException[5]);
2572:                           return InstructionErrorNullPointerException[6];
2573:                   }
2574:
2575:                   // Get the array object
2576:                   InstanceData[7] * id = thread[37]->get_jvm[8]()->get_handle_pool[9]()->get_instance[10](arrayref[44]);
2577:                   assert(id[45] != NULL);
2578:
2579:                   // Check the array bounds
2580:                   u4[12] array_length = ((PrimitiveArrayInstanceData[13]*)id[45])->get_array_length[14]();
2581:                   if (index[46] < 0 || index[46] >= array_length[47])
2582:                   {
```

Footnotes:

1: *opcodes.cpp:2529*
2: *defs.h:201*
3: *opcodes.cpp:2518*
4: *Thread.cpp:776*
5: *defs.h:105*
6: *defs.h:138*
7: *ObjectData.h:113*
8: *Thread.h:179*
9: *VirtualMachine.h:336*
10: *HandlePool.cpp:25*
11: *opcodes.cpp:2537*
12: *defs.h:174*
13: *ObjectData.h:174*
14: *ObjectData.h:163*
15: *opcodes.cpp:2526*
16: *opcodes.cpp:2541*
17: *defs.h:108*
18: *defs.h:140*
19: *ClassFile.h:136*
20: *ObjectData.h:74*
21: *ClassFile.h:370*
22: *opcodes.cpp:2549*
23: *ClassFile.h:395*
24: *defs.h:346*
25: *defs.h:141*
26: *defs.h:246*
27: *defs.h:214*
28: *defs.h:227*
29: *ObjectData.h:71*
30: *HeapManager.h:46*
31: *opcodes.cpp:2554*
32: *opcodes.cpp:2523*
33: *defs.h:122*
34: *defs.h:120*
35: *Thread.h:17*
36: *Stack.h:129*
37: *opcodes.cpp:2560*
38: *Thread.h:148*
39: *defs.h:175*
40: *opcodes.cpp:2562*
41: *Stack.cpp:206*
42: *defs.h:195*
43: *Stack.cpp:249*
44: *opcodes.cpp:2568*
45: *opcodes.cpp:2576*
46: *opcodes.cpp:2565*
47: *opcodes.cpp:2580*

```
2583:                           thread[1]->raise_exception[2](VM_ERROR_ArrayIndexOutOfBoundsException[3]);
2584:                           return InstructionErrorWrongArrayIndex[4];
2585:                   }
2586:
2587:                   // Check that the array type is short
2588:                   ClassFile[5] * cf = id[6]->class_file[7];
2589:                   if (((PrimitiveArrayClassFile[8]*)cf[9])->get_array_type[10]() != T_SHORT[11])
2590:                           return InstructionErrorWrongArrayType[12];
2591:
2592:                   // Get the value of the element
2593:                   su2[13] short_value;
2594:                   unsigned short element_size = BasicTypes[14][Short[15]].size[16];
2595:                   memcpy(&short_value[17], id[6]->data_start[18]->address[19] + index[20]*element_size[21], element_size[21]);
2596:
2597:                   // Push the value onto the stack
2598:                   frame[22]->push_int[23]((su4[24])short_value[17]);
2599:
2600:                   return InstructionSuccess[25];
2601:           }
2602:
2603:           InstructionResult[26] sastore(Thread[27] * thread)
2604:           {
2605:                   stack_frame[28] * frame = thread[29]->current_stack_frame[30];
2606:
2607:                   // Pop the value
2608:                   su2[13] short_value = (su2[13])frame[31]->pop_int[32]();
2609:
2610:                   // Pop the index
2611:                   su4[24] index = frame[31]->pop_int[32]();
2612:
2613:                   // Pop the array reference
2614:                   word[33] arrayref = frame[31]->pop_reference[34]();
2615:                   if (arrayref[35] == null[36])
2616:                   {
2617:                           thread[29]->raise_exception[2](VM_ERROR_NullPointerException[37]);
2618:                           return InstructionErrorNullPointerException[38];
2619:                   }
2620:
2621:                   // Get the array object
2622:                   InstanceData[39] * id = thread[29]->get_jvm[40]()->get_handle_pool[41]()->get_instance[42](arrayref[35]);
2623:                   assert(id[43] != NULL);
2624:
2625:                   // Check the array bounds
2626:                   u4[44] array_length = ((PrimitiveArrayInstanceData[45]*)id[43])->get_array_length[46]();
2627:                   if (index[47] < 0 || index[47] >= array_length[48])
2628:                   {
2629:                           thread[29]->raise_exception[2](VM_ERROR_ArrayIndexOutOfBoundsException[3]);
2630:                           return InstructionErrorWrongArrayIndex[4];
2631:                   }
2632:
2633:                   // Check that the array type is short
2634:                   ClassFile[5] * cf = id[43]->class_file[7];
2635:                   if (((PrimitiveArrayClassFile[8]*)cf[49])->get_array_type[10]() != T_SHORT[11])
```

Footnotes:

1: *opcodes.cpp:2560*
2: *Thread.cpp:776*
3: *defs.h:108*
4: *defs.h:140*
5: *ClassFile.h:136*
6: *opcodes.cpp:2576*
7: *ObjectData.h:74*
8: *ClassFile.h:370*
9: *opcodes.cpp:2588*
10: *ClassFile.h:395*
11: *defs.h:345*
12: *defs.h:141*
13: *defs.h:173*
14: *defs.h:246*
15: *defs.h:217*
16: *defs.h:227*
17: *opcodes.cpp:2593*
18: *ObjectData.h:71*
19: *HeapManager.h:46*
20: *opcodes.cpp:2565*
21: *opcodes.cpp:2594*
22: *opcodes.cpp:2562*
23: *Stack.cpp:147*
24: *defs.h:175*
25: *defs.h:122*
26: *defs.h:120*
27: *Thread.h:17*
28: *Stack.h:129*
29: *opcodes.cpp:2603*
30: *Thread.h:148*
31: *opcodes.cpp:2605*
32: *Stack.cpp:206*
33: *defs.h:195*
34: *Stack.cpp:249*
35: *opcodes.cpp:2614*
36: *defs.h:201*
37: *defs.h:105*
38: *defs.h:138*
39: *ObjectData.h:113*
40: *Thread.h:179*
41: *VirtualMachine.h:336*
42: *HandlePool.cpp:25*
43: *opcodes.cpp:2622*
44: *defs.h:174*
45: *ObjectData.h:174*
46: *ObjectData.h:163*
47: *opcodes.cpp:2611*
48: *opcodes.cpp:2626*
49: *opcodes.cpp:2634*

```
2636:                           return InstructionErrorWrongArrayType[1];
2637:
2638:                   // Assign the value to the array element
2639:                   unsigned short element_size = BasicTypes[2][Short[3]].size[4];
2640:                   memcpy(id[5]->data_start[6]->address[7] + index[8]*element_size[9], &short_value[10], element_size[9]);
2641:
2642:                   return InstructionSuccess[11];
2643:           }
2644:
2645:
2646:           InstructionResult[12] laload(Thread[13] * thread)
2647:           {
2648:                   stack_frame[14] * frame = thread[15]->current_stack_frame[16];
2649:
2650:                   // Pop the index
2651:                   su4[17] index = frame[18]->pop_int[19]();
2652:
2653:                   // Pop the array reference
2654:                   word[20] arrayref = frame[18]->pop_reference[21]();
2655:                   if (arrayref[22] == null[23])
2656:                   {
2657:                           thread[15]->raise_exception[24](VM_ERROR_NullPointerException[25]);
2658:                           return InstructionErrorNullPointerException[26];
2659:                   }
2660:
2661:                   // Get the array object
2662:                   InstanceData[27] * id = thread[15]->get_jvm[28]()->get_handle_pool[29]()->get_instance[30](arrayref[22]);
2663:                   assert(id[31] != NULL);
2664:
2665:                   // Check the array bounds
2666:                   u4[32] array_length = ((PrimitiveArrayInstanceData[33]*)id[31])->get_array_length[34]();
2667:                   if (index[35] < 0 || index[35] >= array_length[36])
2668:                   {
2669:                           thread[15]->raise_exception[24](VM_ERROR_ArrayIndexOutOfBoundsException[37]);
2670:                           return InstructionErrorWrongArrayIndex[38];
2671:                   }
2672:
2673:                   // Check that the array type is long
2674:                   ClassFile[39] * cf = id[31]->class_file[40];
2675:                   if (((PrimitiveArrayClassFile[41]*)cf[42])->get_array_type[43]() != T_LONG[44])
2676:                           return InstructionErrorWrongArrayType[1];
2677:
2678:                   // Get the value of the element
2679:                   su8[45] long_value;
2680:                   unsigned short element_size = BasicTypes[2][Long[46]].size[4];
2681:                   memcpy(&long_value[47], id[31]->data_start[6]->address[7] + index[35]*element_size[48], element_size[48]);
2682:
2683:                   // Push the value onto the stack
2684:                   frame[18]->push_long[49](long_value[47]);
2685:
2686:                   return InstructionSuccess[11];
2687:           }
2688:
```

Footnotes:

1: *defs.h:141*
2: *defs.h:246*
3: *defs.h:217*
4: *defs.h:227*
5: *opcodes.cpp:2622*
6: *ObjectData.h:71*
7: *HeapManager.h:46*
8: *opcodes.cpp:2611*
9: *opcodes.cpp:2639*
10: *opcodes.cpp:2608*
11: *defs.h:122*
12: *defs.h:120*
13: *Thread.h:17*
14: *Stack.h:129*
15: *opcodes.cpp:2646*
16: *Thread.h:148*
17: *defs.h:175*
18: *opcodes.cpp:2648*
19: *Stack.cpp:206*
20: *defs.h:195*
21: *Stack.cpp:249*
22: *opcodes.cpp:2654*
23: *defs.h:201*
24: *Thread.cpp:776*
25: *defs.h:105*
26: *defs.h:138*
27: *ObjectData.h:113*
28: *Thread.h:179*
29: *VirtualMachine.h:336*
30: *HandlePool.cpp:25*
31: *opcodes.cpp:2662*
32: *defs.h:174*
33: *ObjectData.h:174*
34: *ObjectData.h:163*
35: *opcodes.cpp:2651*
36: *opcodes.cpp:2666*
37: *defs.h:108*
38: *defs.h:140*
39: *ClassFile.h:136*
40: *ObjectData.h:74*
41: *ClassFile.h:370*
42: *opcodes.cpp:2674*
43: *ClassFile.h:395*
44: *defs.h:347*
45: *defs.h:177*
46: *defs.h:215*
47: *opcodes.cpp:2679*
48: *opcodes.cpp:2680*
49: *Stack.cpp:153*

```
2689:           InstructionResult[1] lastore(Thread[2] * thread)
2690:           {
2691:                   stack_frame[3] * frame = thread[4]->current_stack_frame[5];
2692:
2693:                   // Pop the value
2694:                   su8[6] long_value = frame[7]->pop_long[8]();
2695:
2696:                   // Pop the index
2697:                   su4[9] index = frame[7]->pop_int[10]();
2698:
2699:                   // Pop the array reference
2700:                   word[11] arrayref = frame[7]->pop_reference[12]();
2701:                   if (arrayref[13] == null[14])
2702:                   {
2703:                           thread[4]->raise_exception[15](VM_ERROR_NullPointerException[16]);
2704:                           return InstructionErrorNullPointerException[17];
2705:                   }
2706:
2707:                   // Get the array object
2708:                   InstanceData[18] * id = thread[4]->get_jvm[19]()->get_handle_pool[20]()->get_instance[21](arrayref[13]);
2709:                   assert(id[22] != NULL);
2710:
2711:                   // Check the array bounds
2712:                   u4[23] array_length = ((PrimitiveArrayInstanceData[24]*)id[22])->get_array_length[25]();
2713:                   if (index[26] < 0 || index[26] >= array_length[27])
2714:                   {
2715:                           thread[4]->raise_exception[15](VM_ERROR_ArrayIndexOutOfBoundsException[28]);
2716:                           return InstructionErrorWrongArrayIndex[29];
2717:                   }
2718:
2719:                   // Check that the array type is integer
2720:                   ClassFile[30] * cf = id[22]->class_file[31];
2721:                   if (((PrimitiveArrayClassFile[32]*)cf[33])->get_array_type[34]() != T_LONG[35])
2722:                           return InstructionErrorWrongArrayType[36];
2723:
2724:                   // Assign the value to the array element
2725:                   unsigned short element_size = BasicTypes[37][Long[38]].size[39];
2726:                   memcpy(id[22]->data_start[40]->address[41] + index[26]*element_size[42], &long_value[43], element_size[42]);
2727:
2728:                   return InstructionSuccess[44];
2729:           }
2730:
2731:
2732:           InstructionResult[1] castore(Thread[2] * thread)
2733:           {
2734:                   stack_frame[3] * frame = thread[45]->current_stack_frame[5];
2735:
2736:                   // Pop the value and truncate it to char (unsigned 16-bit)
2737:                   u2[46] char_value = (u2[46])frame[47]->pop_int[10]();
2738:
2739:                   // Pop the index
2740:                   su4[9] index = frame[47]->pop_int[10]();
2741:
```

Footnotes:

1: *defs.h:120*
2: *Thread.h:17*
3: *Stack.h:129*
4: *opcodes.cpp:2689*
5: *Thread.h:148*
6: *defs.h:177*
7: *opcodes.cpp:2691*
8: *Stack.cpp:214*
9: *defs.h:175*
10: *Stack.cpp:206*
11: *defs.h:195*
12: *Stack.cpp:249*
13: *opcodes.cpp:2700*
14: *defs.h:201*
15: *Thread.cpp:776*
16: *defs.h:105*
17: *defs.h:138*
18: *ObjectData.h:113*
19: *Thread.h:179*
20: *VirtualMachine.h:336*
21: *HandlePool.cpp:25*
22: *opcodes.cpp:2708*
23: *defs.h:174*
24: *ObjectData.h:174*
25: *ObjectData.h:163*
26: *opcodes.cpp:2697*
27: *opcodes.cpp:2712*
28: *defs.h:108*
29: *defs.h:140*
30: *ClassFile.h:136*
31: *ObjectData.h:74*
32: *ClassFile.h:370*
33: *opcodes.cpp:2720*
34: *ClassFile.h:395*
35: *defs.h:347*
36: *defs.h:141*
37: *defs.h:246*
38: *defs.h:215*
39: *defs.h:227*
40: *ObjectData.h:71*
41: *HeapManager.h:46*
42: *opcodes.cpp:2725*
43: *opcodes.cpp:2694*
44: *defs.h:122*
45: *opcodes.cpp:2732*
46: *defs.h:172*
47: *opcodes.cpp:2734*

```
2742:                   // Pop the array reference
2743:                   word[1] arrayref = frame[2]->pop_reference[3]();
2744:                   if (arrayref[4] == null[5])
2745:                   {
2746:                           thread[6]->raise_exception[7](VM_ERROR_NullPointerException[8]);
2747:                           return InstructionErrorNullPointerException[9];
2748:                   }
2749:
2750:                   // Get the array object
2751:                   InstanceData[10] * id = thread[6]->get_jvm[11]()->get_handle_pool[12]()->get_instance[13](arrayref[4]);
2752:                   assert(id[14] != NULL);
2753:
2754:                   // Check the array bounds
2755:                   u4[15] array_length = ((PrimitiveArrayInstanceData[16]*)id[14])->get_array_length[17]();
2756:                   if (index[18] < 0 || index[18] >= array_length[19])
2757:                   {
2758:                           thread[6]->raise_exception[7](VM_ERROR_ArrayIndexOutOfBoundsException[20]);
2759:                           return InstructionErrorWrongArrayIndex[21];
2760:                   }
2761:
2762:                   // Check that the array type is char
2763:                   ClassFile[22] * cf = id[14]->class_file[23];
2764:                   if (((PrimitiveArrayClassFile[24]*)cf[25])->get_array_type[26]() != T_CHAR[27])
2765:                           return InstructionErrorWrongArrayType[28];
2766:
2767:                   // Assign the value to the array element
2768:                   unsigned short element_size = BasicTypes[29][Char[30]].size[31];
2769:                   memcpy(id[14]->data_start[32]->address[33] + index[18]*element_size[34], &char_value[35], element_size[34]);
2770:
2771:                   return InstructionSuccess[36];
2772:           }
2773:
2774:           InstructionResult[37] caload(Thread[38] * thread)
2775:           {
2776:                   stack_frame[39] * frame = thread[40]->current_stack_frame[41];
2777:
2778:                   // Pop the index
2779:                   su4[42] index = frame[43]->pop_int[44]();
2780:
2781:                   // Pop the array reference
2782:                   word[1] arrayref = frame[43]->pop_reference[3]();
2783:                   if (arrayref[45] == null[5])
2784:                   {
2785:                           thread[40]->raise_exception[7](VM_ERROR_NullPointerException[8]);
2786:                           return InstructionErrorNullPointerException[9];
2787:                   }
2788:
2789:                   // Get the array object
2790:                   InstanceData[10] * id = thread[40]->get_jvm[11]()->get_handle_pool[12]()->get_instance[13](arrayref[45]);
2791:                   assert(id[46] != NULL);
2792:
2793:                   // Check the array bounds
2794:                   u4[15] array_length = ((PrimitiveArrayInstanceData[16]*)id[46])->get_array_length[17]();
```

Footnotes:

1: *defs.h:195*
2: *opcodes.cpp:2734*
3: *Stack.cpp:249*
4: *opcodes.cpp:2743*
5: *defs.h:201*
6: *opcodes.cpp:2732*
7: *Thread.cpp:776*
8: *defs.h:105*
9: *defs.h:138*
10: *ObjectData.h:113*
11: *Thread.h:179*
12: *VirtualMachine.h:336*
13: *HandlePool.cpp:25*
14: *opcodes.cpp:2751*
15: *defs.h:174*
16: *ObjectData.h:174*
17: *ObjectData.h:163*
18: *opcodes.cpp:2740*
19: *opcodes.cpp:2755*
20: *defs.h:108*
21: *defs.h:140*
22: *ClassFile.h:136*
23: *ObjectData.h:74*
24: *ClassFile.h:370*
25: *opcodes.cpp:2763*
26: *ClassFile.h:395*
27: *defs.h:341*
28: *defs.h:141*
29: *defs.h:246*
30: *defs.h:211*
31: *defs.h:227*
32: *ObjectData.h:71*
33: *HeapManager.h:46*
34: *opcodes.cpp:2768*
35: *opcodes.cpp:2737*
36: *defs.h:122*
37: *defs.h:120*
38: *Thread.h:17*
39: *Stack.h:129*
40: *opcodes.cpp:2774*
41: *Thread.h:148*
42: *defs.h:175*
43: *opcodes.cpp:2776*
44: *Stack.cpp:206*
45: *opcodes.cpp:2782*
46: *opcodes.cpp:2790*

```
2795:                   if (index[1] < 0 || index[1] >= array_length[2])
2796:                   {
2797:                           thread[3]->raise_exception[4](VM_ERROR_ArrayIndexOutOfBoundsException[5]);
2798:                           return InstructionErrorWrongArrayIndex[6];
2799:                   }
2800:
2801:                   // Check that the array type is char
2802:                   ClassFile[7] * cf = id[8]->class_file[9];
2803:                   if (((PrimitiveArrayClassFile[10]*)cf[11])->get_array_type[12]() != T_CHAR[13])
2804:                           return InstructionErrorWrongArrayType[14];
2805:
2806:                   // Get the value of the element
2807:                   u2[15] char_value;
2808:
2809:                   //BasicType source_type = ((PrimitiveArrayInstanceData*)id)->get_internal_type(((PrimitiveArrayCl
      assFile*)cf)->get_array_type());
2810:                   unsigned short element_size = BasicTypes[16][Char[17]].size[18];
2811:                   memcpy(&char_value[19], id[8]->data_start[20]->address[21] + index[1]*element_size[22], element_size[22]);
2812:
2813:
2814:                   // Push the value onto the stack (as a zero-extended integer)
2815:                   frame[23]->push_int[24]((u4[25])char_value[19]);
2816:
2817:                   return InstructionSuccess[26];
2818:           }
2819:
2820:           InstructionResult[27] baload(Thread[28] * thread)
2821:           {
2822:                   stack_frame[29] * frame = thread[30]->current_stack_frame[31];
2823:
2824:                   // Pop the index
2825:                   su4[32] index = frame[33]->pop_int[34]();
2826:
2827:                   // Pop the array reference
2828:                   word[35] arrayref = frame[33]->pop_reference[36]();
2829:                   if (arrayref[37] == null[38])
2830:                   {
2831:                           thread[30]->raise_exception[4](VM_ERROR_NullPointerException[39]);
2832:                           return InstructionErrorNullPointerException[40];
2833:                   }
2834:
2835:                   // Get the array object
2836:                   InstanceData[41] * id = thread[30]->get_jvm[42]()->get_handle_pool[43]()->get_instance[44](arrayref[37]);
2837:                   assert(id[45] != NULL);
2838:
2839:                   // Check the array bounds
2840:                   u4[25] array_length = ((PrimitiveArrayInstanceData[46]*)id[45])->get_array_length[47]();
2841:                   if (index[48] < 0 || index[48] >= array_length[49])
2842:                   {
2843:                           thread[30]->raise_exception[4](VM_ERROR_ArrayIndexOutOfBoundsException[5]);
2844:                           return InstructionErrorWrongArrayIndex[6];
2845:                   }
2846:
```

Footnotes:

1: *opcodes.cpp:2779*
2: *opcodes.cpp:2794*
3: *opcodes.cpp:2774*
4: *Thread.cpp:776*
5: *defs.h:108*
6: *defs.h:140*
7: *ClassFile.h:136*
8: *opcodes.cpp:2790*
9: *ObjectData.h:74*
10: *ClassFile.h:370*
11: *opcodes.cpp:2802*
12: *ClassFile.h:395*
13: *defs.h:341*
14: *defs.h:141*
15: *defs.h:172*
16: *defs.h:246*
17: *defs.h:211*
18: *defs.h:227*
19: *opcodes.cpp:2807*
20: *ObjectData.h:71*
21: *HeapManager.h:46*
22: *opcodes.cpp:2810*
23: *opcodes.cpp:2776*
24: *Stack.cpp:147*
25: *defs.h:174*
26: *defs.h:122*
27: *defs.h:120*
28: *Thread.h:17*
29: *Stack.h:129*
30: *opcodes.cpp:2820*
31: *Thread.h:148*
32: *defs.h:175*
33: *opcodes.cpp:2822*
34: *Stack.cpp:206*
35: *defs.h:195*
36: *Stack.cpp:249*
37: *opcodes.cpp:2828*
38: *defs.h:201*
39: *defs.h:105*
40: *defs.h:138*
41: *ObjectData.h:113*
42: *Thread.h:179*
43: *VirtualMachine.h:336*
44: *HandlePool.cpp:25*
45: *opcodes.cpp:2836*
46: *ObjectData.h:174*
47: *ObjectData.h:163*
48: *opcodes.cpp:2825*
49: *opcodes.cpp:2840*

```
2847:                   ClassFile[1] * cf = id[2]->class_file[3];
2848:                   su4[4] int_value;
2849:                   if (((PrimitiveArrayClassFile[5]*)cf[6])->get_array_type[7]() == T_BYTE[8])
2850:                   {
2851:                           su1[9] byte_value;
2852:                           // Get the value of the element
2853:                           unsigned short element_size = BasicTypes[10][Byte[11]].size[12];
2854:                           memcpy(&byte_value[13], id[2]->data_start[14]->address[15] + index[16]*element_size[17], element_size[17]);
2855:                           // Sign-extend to int
2856:                           int_value[18] = (su4[4])byte_value[13];
2857:                   }
2858:                   else if (((PrimitiveArrayClassFile[5]*)cf[6])->get_array_type[7]() == T_BOOLEAN[19])
2859:                   {
2860:                           su1[9] bool_value;
2861:                           // Get the value of the element
2862:                           unsigned short element_size = BasicTypes[10][Boolean[20]].size[12];
2863:                           memcpy(&bool_value[21], id[2]->data_start[14]->address[15] + index[16]*element_size[22], element_size[22]);
2864:                           // Zero-extend to int
2865:                           int_value[18] = (su4[4])(u4[23])bool_value[21];
2866:                   }
2867:                   else
2868:                           return InstructionErrorWrongArrayType[24];
2869:
2870:                   // Push the value onto the stack
2871:                   frame[25]->push_int[26](int_value[18]);
2872:
2873:                   return InstructionSuccess[27];
2874:           }
2875:
2876:           InstructionResult[28] bastore(Thread[29] * thread)
2877:           {
2878:                   stack_frame[30] * frame = thread[31]->current_stack_frame[32];
2879:
2880:                   // Pop the value
2881:                   su4[4] int_value = frame[33]->pop_int[34]();
2882:
2883:                   // Pop the index
2884:                   su4[4] index = frame[33]->pop_int[34]();
2885:
2886:                   // Pop the array reference
2887:                   word[35] arrayref = frame[33]->pop_reference[36]();
2888:                   if (arrayref[37] == null[38])
2889:                   {
2890:                           thread[31]->raise_exception[39](VM_ERROR_NullPointerException[40]);
2891:                           return InstructionErrorNullPointerException[41];
2892:                   }
2893:
2894:                   // Get the array object
2895:                   InstanceData[42] * id = thread[31]->get_jvm[43]()->get_handle_pool[44]()->get_instance[45](arrayref[37]);
2896:                   assert(id[46] != NULL);
2897:
2898:                   // Check the array bounds
2899:                   u4[23] array_length = ((PrimitiveArrayInstanceData[47]*)id[46])->get_array_length[48]();
```

Footnotes:

1: *ClassFile.h:136*
2: *opcodes.cpp:2836*
3: *ObjectData.h:74*
4: *defs.h:175*
5: *ClassFile.h:370*
6: *opcodes.cpp:2847*
7: *ClassFile.h:395*
8: *defs.h:344*
9: *defs.h:169*
10: *defs.h:246*
11: *defs.h:210*
12: *defs.h:227*
13: *opcodes.cpp:2851*
14: *ObjectData.h:71*
15: *HeapManager.h:46*
16: *opcodes.cpp:2825*
17: *opcodes.cpp:2853*
18: *opcodes.cpp:2848*
19: *defs.h:340*
20: *defs.h:218*
21: *opcodes.cpp:2860*
22: *opcodes.cpp:2862*
23: *defs.h:174*
24: *defs.h:141*
25: *opcodes.cpp:2822*
26: *Stack.cpp:147*
27: *defs.h:122*
28: *defs.h:120*
29: *Thread.h:17*
30: *Stack.h:129*
31: *opcodes.cpp:2876*
32: *Thread.h:148*
33: *opcodes.cpp:2878*
34: *Stack.cpp:206*
35: *defs.h:195*
36: *Stack.cpp:249*
37: *opcodes.cpp:2887*
38: *defs.h:201*
39: *Thread.cpp:776*
40: *defs.h:105*
41: *defs.h:138*
42: *ObjectData.h:113*
43: *Thread.h:179*
44: *VirtualMachine.h:336*
45: *HandlePool.cpp:25*
46: *opcodes.cpp:2895*
47: *ObjectData.h:174*
48: *ObjectData.h:163*

```
2900:                   if (index[1] < 0 || index[1] >= array_length[2])
2901:                   {
2902:                           thread[3]->raise_exception[4](VM_ERROR_ArrayIndexOutOfBoundsException[5]);
2903:                           return InstructionErrorWrongArrayIndex[6];
2904:                   }
2905:
2906:                   ClassFile[7] * cf = id[8]->class_file[9];
2907:
2908:                   // Check whether the array type is byte
2909:                   if (((PrimitiveArrayClassFile[10]*)cf[11])->get_array_type[12]() == T_BYTE[13])
2910:                   {
2911:                           su1[14] byte_value = (su1[14])int_value[15]; // truncate the value to a byte
2912:                           // Assign the value to the array element
2913:                           unsigned short element_size = BasicTypes[16][Byte[17]].size[18];
2914:                           memcpy(id[8]->data_start[19]->address[20] + index[1]*element_size[21], &byte_value[22], element_size[21]);
2915:                   }
2916:                   // Check whether the array type is byte
2917:                   else if (((PrimitiveArrayClassFile[10]*)cf[11])->get_array_type[12]() == T_BOOLEAN[23])
2918:                   {
2919:                           u1[24] bool_value = int_value[15] & 0xFF; // get low bytes (zero-extended)
2920:                           // Assign the value to the array element
2921:                           unsigned short element_size = BasicTypes[16][Boolean[25]].size[18];
2922:                           memcpy(id[8]->data_start[19]->address[20] + index[1]*element_size[26], &bool_value[27], element_size[26]);
2923:                   }
2924:                   else
2925:                           return InstructionErrorWrongArrayType[28];
2926:
2927:                   return InstructionSuccess[29];
2928:           }
2929:
2930:           InstructionResult[30] daload(Thread[31] * thread)
2931:           {
2932:                   stack_frame[32] * frame = thread[33]->current_stack_frame[34];
2933:
2934:                   // Pop the index
2935:                   su4[35] index = frame[36]->pop_int[37]();
2936:
2937:                   // Pop the array reference
2938:                   word[38] arrayref = frame[36]->pop_reference[39]();
2939:                   if (arrayref[40] == null[41])
2940:                   {
2941:                           thread[33]->raise_exception[4](VM_ERROR_NullPointerException[42]);
2942:                           return InstructionErrorNullPointerException[43];
2943:                   }
2944:
2945:                   // Get the array object
2946:                   InstanceData[44] * id = thread[33]->get_jvm[45]()->get_handle_pool[46]()->get_instance[47](arrayref[40]);
2947:                   assert(id[48] != NULL);
2948:
2949:                   // Check the array bounds
2950:                   u4[49] array_length = ((PrimitiveArrayInstanceData[50]*)id[48])->get_array_length[51]();
2951:                   if (index[52] < 0 || index[52] >= array_length[53])
2952:                   {
```

Footnotes:

1: *opcodes.cpp:2884*
2: *opcodes.cpp:2899*
3: *opcodes.cpp:2876*
4: *Thread.cpp:776*
5: *defs.h:108*
6: *defs.h:140*
7: *ClassFile.h:136*
8: *opcodes.cpp:2895*
9: *ObjectData.h:74*
10: *ClassFile.h:370*
11: *opcodes.cpp:2906*
12: *ClassFile.h:395*
13: *defs.h:344*
14: *defs.h:169*
15: *opcodes.cpp:2881*
16: *defs.h:246*
17: *defs.h:210*
18: *defs.h:227*
19: *ObjectData.h:71*
20: *HeapManager.h:46*
21: *opcodes.cpp:2913*
22: *opcodes.cpp:2911*
23: *defs.h:340*
24: *defs.h:168*
25: *defs.h:218*
26: *opcodes.cpp:2921*
27: *opcodes.cpp:2919*
28: *defs.h:141*
29: *defs.h:122*
30: *defs.h:120*
31: *Thread.h:17*
32: *Stack.h:129*
33: *opcodes.cpp:2930*
34: *Thread.h:148*
35: *defs.h:175*
36: *opcodes.cpp:2932*
37: *Stack.cpp:206*
38: *defs.h:195*
39: *Stack.cpp:249*
40: *opcodes.cpp:2938*
41: *defs.h:201*
42: *defs.h:105*
43: *defs.h:138*
44: *ObjectData.h:113*
45: *Thread.h:179*
46: *VirtualMachine.h:336*
47: *HandlePool.cpp:25*
48: *opcodes.cpp:2946*
49: *defs.h:174*
50: *ObjectData.h:174*
51: *ObjectData.h:163*
52: *opcodes.cpp:2935*
53: *opcodes.cpp:2950*

```
2953:                   thread[1]->raise_exception[2](VM_ERROR_ArrayIndexOutOfBoundsException[3]);
2954:                   return InstructionErrorWrongArrayIndex[4];
2955:           }
2956:
2957:           // Check that the array type is double
2958:           ClassFile[5] * cf = id[6]->class_file[7];
2959:           if (((PrimitiveArrayClassFile[8]*)cf[9])->get_array_type[10]() != T_DOUBLE[11])
2960:                   return InstructionErrorWrongArrayType[12];
2961:
2962:           // Get the value of the element
2963:           double double_value;
2964:           unsigned short element_size = BasicTypes[13][Double[14]].size[15];
2965:           memcpy(&double_value[16], id[6]->data_start[17]->address[18] + index[19]*element_size[20], element_size[20]);
2966:
2967:           // Push the value onto the stack
2968:           frame[21]->push_double[22](double_value[16]);
2969:
2970:           return InstructionSuccess[23];
2971:   }
2972:
2973:   InstructionResult[24] dastore(Thread[25] * thread)
2974:   {
2975:           stack_frame[26] * frame = thread[27]->current_stack_frame[28];
2976:
2977:           // Pop the value
2978:           double double_value = frame[29]->pop_double[30]();
2979:
2980:           // Pop the index
2981:           su4[31] index = frame[29]->pop_int[32]();
2982:
2983:           // Pop the array reference
2984:           word[33] arrayref = frame[29]->pop_reference[34]();
2985:           if (arrayref[35] == null[36])
2986:           {
2987:                   thread[27]->raise_exception[2](VM_ERROR_NullPointerException[37]);
2988:                   return InstructionErrorNullPointerException[38];
2989:           }
2990:
2991:           // Get the array object
2992:           InstanceData[39] * id = thread[27]->get_jvm[40]()->get_handle_pool[41]()->get_instance[42](arrayref[35]);
2993:           assert(id[43] != NULL);
2994:
2995:           // Check the array bounds
2996:           u4[44] array_length = ((PrimitiveArrayInstanceData[45]*)id[43])->get_array_length[46]();
2997:           if (index[47] < 0 || index[47] >= array_length[48])
2998:           {
2999:                   thread[27]->raise_exception[2](VM_ERROR_ArrayIndexOutOfBoundsException[3]);
3000:                   return InstructionErrorWrongArrayIndex[4];
3001:           }
3002:
3003:           // Check that the array type is double
3004:           ClassFile[5] * cf = id[43]->class_file[7];
3005:           if (((PrimitiveArrayClassFile[8]*)cf[49])->get_array_type[10]() != T_DOUBLE[11])
```

Footnotes:

1: *opcodes.cpp:2930*
2: *Thread.cpp:776*
3: *defs.h:108*
4: *defs.h:140*
5: *ClassFile.h:136*
6: *opcodes.cpp:2946*
7: *ObjectData.h:74*
8: *ClassFile.h:370*
9: *opcodes.cpp:2958*
10: *ClassFile.h:395*
11: *defs.h:343*
12: *defs.h:141*
13: *defs.h:246*
14: *defs.h:212*
15: *defs.h:227*
16: *opcodes.cpp:2963*
17: *ObjectData.h:71*
18: *HeapManager.h:46*
19: *opcodes.cpp:2935*
20: *opcodes.cpp:2964*
21: *opcodes.cpp:2932*
22: *Stack.cpp:171*
23: *defs.h:122*
24: *defs.h:120*
25: *Thread.h:17*
26: *Stack.h:129*
27: *opcodes.cpp:2973*
28: *Thread.h:148*
29: *opcodes.cpp:2975*
30: *Stack.cpp:235*
31: *defs.h:175*
32: *Stack.cpp:206*
33: *defs.h:195*
34: *Stack.cpp:249*
35: *opcodes.cpp:2984*
36: *defs.h:201*
37: *defs.h:105*
38: *defs.h:138*
39: *ObjectData.h:113*
40: *Thread.h:179*
41: *VirtualMachine.h:336*
42: *HandlePool.cpp:25*
43: *opcodes.cpp:2992*
44: *defs.h:174*
45: *ObjectData.h:174*
46: *ObjectData.h:163*
47: *opcodes.cpp:2981*
48: *opcodes.cpp:2996*
49: *opcodes.cpp:3004*

```
3006:                           return InstructionErrorWrongArrayType[1];
3007:
3008:                   // Assign the value to the array element
3009:                   unsigned short element_size = BasicTypes[2][Double[3]].size[4];
3010:                   memcpy(id[5]->data_start[6]->address[7] + index[8]*element_size[9], &double_value[10], element_size[9]);
3011:
3012:                   return InstructionSuccess[11];
3013:           }
3014:
3015:           InstructionResult[12] faload(Thread[13] * thread)
3016:           {
3017:                   stack_frame[14] * frame = thread[15]->current_stack_frame[16];
3018:
3019:                   // Pop the index
3020:                   su4[17] index = frame[18]->pop_int[19]();
3021:
3022:                   // Pop the array reference
3023:                   word[20] arrayref = frame[18]->pop_reference[21]();
3024:                   if (arrayref[22] == null[23])
3025:                   {
3026:                           thread[15]->raise_exception[24](VM_ERROR_NullPointerException[25]);
3027:                           return InstructionErrorNullPointerException[26];
3028:                   }
3029:
3030:                   // Get the array object
3031:                   InstanceData[27] * id = thread[15]->get_jvm[28]()->get_handle_pool[29]()->get_instance[30](arrayref[22]);
3032:                   assert(id[31] != NULL);
3033:
3034:                   // Check the array bounds
3035:                   u4[32] array_length = ((PrimitiveArrayInstanceData[33]*)id[31])->get_array_length[34]();
3036:                   if (index[35] < 0 || index[35] >= array_length[36])
3037:                   {
3038:                           thread[15]->raise_exception[24](VM_ERROR_ArrayIndexOutOfBoundsException[37]);
3039:                           return InstructionErrorWrongArrayIndex[38];
3040:                   }
3041:
3042:                   // Check that the array type is float
3043:                   ClassFile[39] * cf = id[31]->class_file[40];
3044:                   if (((PrimitiveArrayClassFile[41]*)cf[42])->get_array_type[43]() != T_FLOAT[44])
3045:                           return InstructionErrorWrongArrayType[1];
3046:
3047:                   // Get the value of the element
3048:                   float float_value;
3049:                   unsigned short element_size = BasicTypes[2][Float[45]].size[4];
3050:                   memcpy(&float_value[46], id[31]->data_start[6]->address[7] + index[35]*element_size[47], element_size[47]);
3051:
3052:                   // Push the value onto the stack
3053:                   frame[18]->push_float[48](float_value[46]);
3054:
3055:                   return InstructionSuccess[11];
3056:           }
3057:
3058:           InstructionResult[12] fastore(Thread[13] * thread)
```

Footnotes:

1: *defs.h:141*
2: *defs.h:246*
3: *defs.h:212*
4: *defs.h:227*
5: *opcodes.cpp:2992*
6: *ObjectData.h:71*
7: *HeapManager.h:46*
8: *opcodes.cpp:2981*
9: *opcodes.cpp:3009*
10: *opcodes.cpp:2978*
11: *defs.h:122*
12: *defs.h:120*
13: *Thread.h:17*
14: *Stack.h:129*
15: *opcodes.cpp:3015*
16: *Thread.h:148*
17: *defs.h:175*
18: *opcodes.cpp:3017*
19: *Stack.cpp:206*
20: *defs.h:195*
21: *Stack.cpp:249*
22: *opcodes.cpp:3023*
23: *defs.h:201*
24: *Thread.cpp:776*
25: *defs.h:105*
26: *defs.h:138*
27: *ObjectData.h:113*
28: *Thread.h:179*
29: *VirtualMachine.h:336*
30: *HandlePool.cpp:25*
31: *opcodes.cpp:3031*
32: *defs.h:174*
33: *ObjectData.h:174*
34: *ObjectData.h:163*
35: *opcodes.cpp:3020*
36: *opcodes.cpp:3035*
37: *defs.h:108*
38: *defs.h:140*
39: *ClassFile.h:136*
40: *ObjectData.h:74*
41: *ClassFile.h:370*
42: *opcodes.cpp:3043*
43: *ClassFile.h:395*
44: *defs.h:342*
45: *defs.h:213*
46: *opcodes.cpp:3048*
47: *opcodes.cpp:3049*
48: *Stack.cpp:165*

```
3059:         {
3060:                 stack_frame[1] * frame = thread[2]->current_stack_frame[3];
3061:
3062:                 // Pop the value
3063:                 float float_value = frame[4]->pop_float[5]();
3064:
3065:                 // Pop the index
3066:                 su4[6] index = frame[4]->pop_int[7]();
3067:
3068:                 // Pop the array reference
3069:                 word[8] arrayref = frame[4]->pop_reference[9]();
3070:                 if (arrayref[10] == null[11])
3071:                 {
3072:                         thread[2]->raise_exception[12](VM_ERROR_NullPointerException[13]);
3073:                         return InstructionErrorNullPointerException[14];
3074:                 }
3075:
3076:                 // Get the array object
3077:                 InstanceData[15] * id = thread[2]->get_jvm[16]()->get_handle_pool[17]()->get_instance[18](arrayref[10]);
3078:                 assert(id[19] != NULL);
3079:
3080:                 // Check the array bounds
3081:                 u4[20] array_length = ((PrimitiveArrayInstanceData[21]*)id[19])->get_array_length[22]();
3082:                 if (index[23] < 0 || index[23] >= array_length[24])
3083:                 {
3084:                         thread[2]->raise_exception[12](VM_ERROR_ArrayIndexOutOfBoundsException[25]);
3085:                         return InstructionErrorWrongArrayIndex[26];
3086:                 }
3087:
3088:                 // Check that the array type is float
3089:                 ClassFile[27] * cf = id[19]->class_file[28];
3090:                 if (((PrimitiveArrayClassFile[29]*)cf[30])->get_array_type[31]() != T_FLOAT[32])
3091:                         return InstructionErrorWrongArrayType[33];
3092:
3093:                 // Assign the value to the array element
3094:                 unsigned short element_size = BasicTypes[34][Float[35]].size[36];
3095:                 memcpy(id[19]->data_start[37]->address[38] + index[23]*element_size[39], &float_value[40], element_size[39]);
3096:
3097:                 return InstructionSuccess[41];
3098:         }
3099:
3100:         InstructionResult[42] aaload(Thread[43] * thread)
3101:         {
3102:                 stack_frame[1] * frame = thread[44]->current_stack_frame[3];
3103:
3104:                 // Pop the index
3105:                 su4[6] index = frame[45]->pop_int[7]();
3106:
3107:                 // Pop the array reference
3108:                 word[8] arrayref = frame[45]->pop_reference[9]();
3109:                 if (arrayref[46] == null[11])
3110:                 {
3111:                         thread[44]->raise_exception[12](VM_ERROR_NullPointerException[13]);
```

Footnotes:

1: *Stack.h:129*
2: *opcodes.cpp:3058*
3: *Thread.h:148*
4: *opcodes.cpp:3060*
5: *Stack.cpp:227*
6: *defs.h:175*
7: *Stack.cpp:206*
8: *defs.h:195*
9: *Stack.cpp:249*
10: *opcodes.cpp:3069*
11: *defs.h:201*
12: *Thread.cpp:776*
13: *defs.h:105*
14: *defs.h:138*
15: *ObjectData.h:113*
16: *Thread.h:179*
17: *VirtualMachine.h:336*
18: *HandlePool.cpp:25*
19: *opcodes.cpp:3077*
20: *defs.h:174*
21: *ObjectData.h:174*
22: *ObjectData.h:163*
23: *opcodes.cpp:3066*
24: *opcodes.cpp:3081*
25: *defs.h:108*
26: *defs.h:140*
27: *ClassFile.h:136*
28: *ObjectData.h:74*
29: *ClassFile.h:370*
30: *opcodes.cpp:3089*
31: *ClassFile.h:395*
32: *defs.h:342*
33: *defs.h:141*
34: *defs.h:246*
35: *defs.h:213*
36: *defs.h:227*
37: *ObjectData.h:71*
38: *HeapManager.h:46*
39: *opcodes.cpp:3094*
40: *opcodes.cpp:3063*
41: *defs.h:122*
42: *defs.h:120*
43: *Thread.h:17*
44: *opcodes.cpp:3100*
45: *opcodes.cpp:3102*
46: *opcodes.cpp:3108*

```
3112:                   return InstructionErrorNullPointerException[1];
3113:           }
3114:
3115:           // Get the array object
3116:           InstanceData[2] * id = thread[3]->get_jvm[4]()->get_handle_pool[5]()->get_instance[6](arrayref[7]);
3117:           assert(id[8] != NULL);
3118:
3119:           // Check the array bounds
3120:           u4[9] array_length = ((ArrayInstanceData[10]*)id[8])->get_array_length[11]();
3121:           if (index[12] < 0 || index[12] >= array_length[13])
3122:           {
3123:                   thread[3]->raise_exception[14](VM_ERROR_ArrayIndexOutOfBoundsException[15]);
3124:                   return InstructionErrorWrongArrayIndex[16];
3125:           }
3126:
3127:           // Get the value of the element
3128:           word[17] reference_value;
3129:           memcpy(&reference_value[18], id[8]->data_start[19]->address[20] + index[12]*sizeof(word[17]), sizeof(word[17]));
3130:
3131:           // Push the value onto the stack
3132:           frame[21]->push_reference[22](reference_value[18]);
3133:
3134:           return InstructionSuccess[23];
3135:   }
3136:
3137:   InstructionResult[24] aastore(Thread[25] * thread)
3138:   {
3139:           stack_frame[26] * frame = thread[27]->current_stack_frame[28];
3140:
3141:           // Pop the value
3142:           word[17] reference_value = frame[29]->pop_reference[30]();
3143:
3144:           // Pop the index
3145:           su4[31] index = frame[29]->pop_int[32]();
3146:
3147:           // Pop the array reference
3148:           word[17] arrayref = frame[29]->pop_reference[30]();
3149:           if (arrayref[33] == null[34])
3150:           {
3151:                   thread[27]->raise_exception[14](VM_ERROR_NullPointerException[35]);
3152:                   return InstructionErrorNullPointerException[1];
3153:           }
3154:
3155:           // Get the array object
3156:           InstanceData[2] * array_id = thread[27]->get_jvm[4]()->get_handle_pool[5]()->get_instance[6](arrayref[33]);
3157:           assert(array_id[36] != NULL);
3158:   /*      TODO:
3159:           // Check that the reference value is assigment compatible with the type
3160:           // of this array
3161:           if (reference_value != null)
3162:           {
3163:                   InstanceData * value_id = thread->get_jvm()->get_handle_pool()->get_instance(reference_va
   lue);
```

Footnotes:

1: *defs.h:138*
2: *ObjectData.h:113*
3: *opcodes.cpp:3100*
4: *Thread.h:179*
5: *VirtualMachine.h:336*
6: *HandlePool.cpp:25*
7: *opcodes.cpp:3108*
8: *opcodes.cpp:3116*
9: *defs.h:174*
10: *ObjectData.h:155*
11: *ObjectData.h:163*
12: *opcodes.cpp:3105*
13: *opcodes.cpp:3120*
14: *Thread.cpp:776*
15: *defs.h:108*
16: *defs.h:140*
17: *defs.h:195*
18: *opcodes.cpp:3128*
19: *ObjectData.h:71*
20: *HeapManager.h:46*
21: *opcodes.cpp:3102*
22: *Stack.cpp:187*
23: *defs.h:122*
24: *defs.h:120*
25: *Thread.h:17*
26: *Stack.h:129*
27: *opcodes.cpp:3137*
28: *Thread.h:148*
29: *opcodes.cpp:3139*
30: *Stack.cpp:249*
31: *defs.h:175*
32: *Stack.cpp:206*
33: *opcodes.cpp:3148*
34: *defs.h:201*
35: *defs.h:105*
36: *opcodes.cpp:3156*

```
3164:                           assert(value_id != NULL);
3165:                           if (!array_id->is_assignment_compatible(value_id))
3166:                           {
3167:                                   thread->raise_exception(VM_ERROR_ArrayStoreException);
3168:                                   return InstructionErrorAssignmentIncompatible;
3169:                           }
3170:                   }
3171:           */
3172:                   // Check the array bounds
3173:                   u4[1] array_length = ((ArrayInstanceData[2]*)array_id[3])->get_array_length[4]();
3174:                   if (index[5] < 0 || index[5] >= array_length[6])
3175:                   {
3176:                           thread[7]->raise_exception[8](VM_ERROR_ArrayIndexOutOfBoundsException[9]);
3177:                           return InstructionErrorWrongArrayIndex[10];
3178:                   }
3179:
3180:                   // Assign the value to the array element
3181:                   memcpy(array_id[3]->data_start[11]->address[12] + index[5]*sizeof(word[13]), &reference_value[14], sizeof(word[13]));
3182:
3183:                   return InstructionSuccess[15];
3184:           }
3185:
3186:           InstructionResult[16] arraylength(Thread[17] * thread)
3187:           {
3188:                   stack_frame[18] * frame = thread[19]->current_stack_frame[20];
3189:                   // Pop the array reference
3190:                   word[13] arrayref = frame[21]->pop_reference[22]();
3191:                   if (arrayref[23] == null[24])
3192:                   {
3193:                           thread[19]->raise_exception[8](VM_ERROR_NullPointerException[25]);
3194:                           return InstructionErrorNullPointerException[26];
3195:                   }
3196:
3197:                   // Get the array object
3198:                   InstanceData[27] * id = thread[19]->get_jvm[28]()->get_handle_pool[29]()->get_instance[30](arrayref[23]);
3199:                   assert(id[31] != NULL);
3200:
3201:                   u4[1] array_length = ((PrimitiveArrayInstanceData[32]*)id[31])->get_array_length[4]();
3202:
3203:                   // Push array length
3204:                   frame[21]->push_int[33]((su4[34])array_length[35]);
3205:
3206:                   return InstructionSuccess[15];
3207:           }
3208:
3209:           //-----------------------------------------------------------------------------
3210:           // Opcodes array
3211:           // Index of this array corresponds to the value of the opcode thus making retrieving easy
3212:           //-----------------------------------------------------------------------------
3213:
3214:
3215:           Opcode[36] opcodes[] = {
3216:                   { "NOP", 0, nop[37] },
```

Footnotes:

1: *defs.h:174*
2: *ObjectData.h:155*
3: *opcodes.cpp:3156*
4: *ObjectData.h:163*
5: *opcodes.cpp:3145*
6: *opcodes.cpp:3173*
7: *opcodes.cpp:3137*
8: *Thread.cpp:776*
9: *defs.h:108*
10: *defs.h:140*
11: *ObjectData.h:71*
12: *HeapManager.h:46*
13: *defs.h:195*
14: *opcodes.cpp:3142*
15: *defs.h:122*
16: *defs.h:120*
17: *Thread.h:17*
18: *Stack.h:129*
19: *opcodes.cpp:3186*
20: *Thread.h:148*
21: *opcodes.cpp:3188*
22: *Stack.cpp:249*
23: *opcodes.cpp:3190*
24: *defs.h:201*
25: *defs.h:105*
26: *defs.h:138*
27: *ObjectData.h:113*
28: *Thread.h:179*
29: *VirtualMachine.h:336*
30: *HandlePool.cpp:25*
31: *opcodes.cpp:3198*
32: *ObjectData.h:174*
33: *Stack.cpp:147*
34: *defs.h:175*
35: *opcodes.cpp:3201*
36: *opcodes.h:254*
37: *opcodes.cpp:16*

```
3217:                 { "ACONST_NULL", 0, aconst_null[1] },
3218:                 { "ICONST_M1", 0, iconst_m1[2] },
3219:                 { "ICONST_0", 0, iconst_0[3] },
3220:                 { "ICONST_1", 0, iconst_1[4] },
3221:                 { "ICONST_2", 0, iconst_2[5] },
3222:                 { "ICONST_3", 0, iconst_3[6] },
3223:                 { "ICONST_4", 0, iconst_4[7] },
3224:                 { "ICONST_5", 0, iconst_5[8] },
3225:                 { "LCONST_0", 0, lconst_0[9] },
3226:                 { "LCONST_1", 0, lconst_1[10] },
3227:                 { "FCONST_0", 0, fconst_0[11] },
3228:                 { "FCONST_1", 0, fconst_1[12] },
3229:                 { "FCONST_2", 0, fconst_2[13] },
3230:                 { "DCONST_0", 0, dconst_0[14] },
3231:                 { "DCONST_1", 0, dconst_1[15] },
3232:                 { "BIPUSH",      0, bipush[16] },
3233:                 { "SIPUSH",      2, sipush[17] },
3234:                 { "LDC", 1, ldc[18] },
3235:                 { "LDC_W", 2, ldc_w[19] },
3236:                 { "LDC2_W", 2, ldc2_w[20] },
3237:                 { "ILOAD", 1, iload[21] },
3238:                 { "LLOAD", 1, lload[22] },
3239:                 { "FLOAD", 1, fload[23] },
3240:                 { "DLOAD", 1, dload[24] },
3241:                 { "ALOAD", 1, aload[25] },
3242:                 { "ILOAD_0", 0, iload_0[26] },
3243:                 { "ILOAD_1", 0, iload_1[27] },
3244:                 { "ILOAD_2", 0, iload_2[28] },
3245:                 { "ILOAD_3", 0, iload_3[29] },
3246:                 { "LLOAD_0", 0, lload_0[30] },
3247:                 { "LLOAD_1", 0, lload_1[31] },
3248:                 { "LLOAD_2", 0, lload_2[32] },
3249:                 { "LLOAD_3", 0, lload_3[33] },
3250:                 { "FLOAD_0", 0, fload_0[34] },
3251:                 { "FLOAD_1", 0, fload_1[35] },
3252:                 { "FLOAD_2", 0, fload_2[36] },
3253:                 { "FLOAD_3", 0, fload_3[37] },
3254:                 { "DLOAD_0", 0, dload_0[38] },
3255:                 { "DLOAD_1", 0, dload_1[39] },
3256:                 { "DLOAD_2", 0, dload_2[40] },
3257:                 { "DLOAD_3", 0, dload_3[41] },
3258:                 { "ALOAD_0", 0, aload_0[42] },
3259:                 { "ALOAD_1", 0, aload_1[43] },
3260:                 { "ALOAD_2", 0, aload_2[44] },
3261:                 { "ALOAD_3", 0, aload_3[45] },
3262:                 { "IALOAD", 0, iaload[46] },
3263:                 { "LALOAD", 0, laload[47] },
3264:                 { "FALOAD", 0, faload[48] },
3265:                 { "DALOAD", 0, daload[49] },
3266:                 { "AALOAD", 0, aaload[50] },
3267:                 { "BALOAD", 0, baload[51] },
3268:                 { "CALOAD", 0, caload[52] },
3269:                 { "SALOAD", 0, saload[53] },
```

Footnotes:

1: *opcodes.cpp:25*
2: *opcodes.cpp:219*
3: *opcodes.cpp:195*
4: *opcodes.cpp:199*
5: *opcodes.cpp:203*
6: *opcodes.cpp:207*
7: *opcodes.cpp:211*
8: *opcodes.cpp:215*
9: *opcodes.cpp:231*
10: *opcodes.cpp:235*
11: *opcodes.cpp:248*
12: *opcodes.cpp:252*
13: *opcodes.cpp:256*
14: *opcodes.cpp:268*
15: *opcodes.cpp:272*
16: *opcodes.cpp:466*
17: *opcodes.cpp:474*
18: *opcodes.cpp:1601*
19: *opcodes.cpp:1608*
20: *opcodes.cpp:1617*
21: *opcodes.cpp:56*
22: *opcodes.cpp:87*
23: *opcodes.cpp:119*
24: *opcodes.cpp:150*
25: *opcodes.cpp:181*
26: *opcodes.cpp:40*
27: *opcodes.cpp:44*
28: *opcodes.cpp:48*
29: *opcodes.cpp:52*
30: *opcodes.cpp:71*
31: *opcodes.cpp:75*
32: *opcodes.cpp:79*
33: *opcodes.cpp:83*
34: *opcodes.cpp:103*
35: *opcodes.cpp:107*
36: *opcodes.cpp:111*
37: *opcodes.cpp:115*
38: *opcodes.cpp:134*
39: *opcodes.cpp:138*
40: *opcodes.cpp:142*
41: *opcodes.cpp:146*
42: *opcodes.cpp:165*
43: *opcodes.cpp:169*
44: *opcodes.cpp:173*
45: *opcodes.cpp:177*
46: *opcodes.cpp:2475*
47: *opcodes.cpp:2646*
48: *opcodes.cpp:3015*
49: *opcodes.cpp:2930*
50: *opcodes.cpp:3100*
51: *opcodes.cpp:2820*
52: *opcodes.cpp:2774*
53: *opcodes.cpp:2560*

```
3270:                 { "ISTORE", 1, istore[1] },
3271:                 { "LSTORE", 1, lstore[2] },
3272:                 { "FSTORE", 1, fstore[3] },
3273:                 { "DSTORE", 1, dstore[4] },
3274:                 { "ASTORE", 1, astore[5] },
3275:                 { "ISTORE_0", 0, istore_0[6] },
3276:                 { "ISTORE_1", 0, istore_1[7] },
3277:                 { "ISTORE_2", 0, istore_2[8] },
3278:                 { "ISTORE_3", 0, istore_3[9] },
3279:                 { "LSTORE_0", 0, lstore_0[10] },
3280:                 { "LSTORE_1", 0, lstore_1[11] },
3281:                 { "LSTORE_2", 0, lstore_2[12] },
3282:                 { "LSTORE_3", 0, lstore_3[13] },
3283:                 { "FSTORE_0", 0, fstore_0[14] },
3284:                 { "FSTORE_1", 0, fstore_1[15] },
3285:                 { "FSTORE_2", 0, fstore_2[16] },
3286:                 { "FSTORE_3", 0, fstore_3[17] },
3287:                 { "DSTORE_0", 0, dstore_0[18] },
3288:                 { "DSTORE_1", 0, dstore_1[19] },
3289:                 { "DSTORE_2", 0, dstore_2[20] },
3290:                 { "DSTORE_3", 0, dstore_3[21] },
3291:                 { "ASTORE_0", 0, astore_0[22] },
3292:                 { "ASTORE_1", 0, astore_1[23] },
3293:                 { "ASTORE_2", 0, astore_2[24] },
3294:                 { "ASTORE_3", 0, astore_3[25] },
3295:                 { "IASTORE", 0, iastore[26] },
3296:                 { "LASTORE", 0, lastore[27] },
3297:                 { "FASTORE", 0, fastore[28] },
3298:                 { "DASTORE", 0, dastore[29] },
3299:                 { "AASTORE", 0, aastore[30] },
3300:                 { "BASTORE", 0, bastore[31] },
3301:                 { "CASTORE", 0, castore[32] },
3302:                 { "SASTORE", 0, sastore[33] },
3303:                 { "POP", 0, pop[34] },
3304:                 { "POP2", 0, pop2[35] },
3305:                 { "DUP", 0, dup[36] },
3306:                 { "DUP_X1", 0, dup_x1[37] },
3307:                 { "DUP_X2", 0 },
3308:                 { "DUP2", 0, dup2[38] },
3309:                 { "DUP2_X1", 0 },
3310:                 { "DUP2_X2", 0 },
3311:                 { "SWAP", 0, swap[39] },
3312:                 { "IADD", 0, iadd[40] },
3313:                 { "LADD", 0, ladd[41] },
3314:                 { "FADD", 0, fadd[42] },
3315:                 { "DADD", 0, dadd[43] },
3316:                 { "ISUB", 0, isub[44] },
3317:                 { "LSUB", 0, lsub[45] },
3318:                 { "FSUB", 0, fsub[46] },
3319:                 { "DSUB", 0, dsub[47] },
3320:                 { "IMUL", 0, imul[48] },
3321:                 { "LMUL", 0, lmul[49] },
3322:                 { "FMUL", 0, fmul[50] },
```

Footnotes:

1: *opcodes.cpp:277*
2: *opcodes.cpp:316*
3: *opcodes.cpp:355*
4: *opcodes.cpp:392*
5: *opcodes.cpp:429*
6: *opcodes.cpp:298*
7: *opcodes.cpp:302*
8: *opcodes.cpp:306*
9: *opcodes.cpp:310*
10: *opcodes.cpp:337*
11: *opcodes.cpp:341*
12: *opcodes.cpp:345*
13: *opcodes.cpp:349*
14: *opcodes.cpp:375*
15: *opcodes.cpp:379*
16: *opcodes.cpp:383*
17: *opcodes.cpp:387*
18: *opcodes.cpp:412*
19: *opcodes.cpp:416*
20: *opcodes.cpp:420*
21: *opcodes.cpp:424*
22: *opcodes.cpp:449*
23: *opcodes.cpp:453*
24: *opcodes.cpp:457*
25: *opcodes.cpp:461*
26: *opcodes.cpp:2518*
27: *opcodes.cpp:2689*
28: *opcodes.cpp:3058*
29: *opcodes.cpp:2973*
30: *opcodes.cpp:3137*
31: *opcodes.cpp:2876*
32: *opcodes.cpp:2732*
33: *opcodes.cpp:2603*
34: *opcodes.cpp:888*
35: *opcodes.cpp:894*
36: *opcodes.cpp:900*
37: *opcodes.cpp:918*
38: *opcodes.cpp:906*
39: *opcodes.cpp:912*
40: *opcodes.cpp:500*
41: *opcodes.cpp:661*
42: *opcodes.cpp:773*
43: *opcodes.cpp:826*
44: *opcodes.cpp:512*
45: *opcodes.cpp:673*
46: *opcodes.cpp:785*
47: *opcodes.cpp:838*
48: *opcodes.cpp:488*
49: *opcodes.cpp:649*
50: *opcodes.cpp:761*

```
3323:                 { "DMUL", 0, dmul[1] },
3324:                 { "IDIV", 0, idiv[2] },
3325:                 { "LDIV", 0, ldiv[3] },
3326:                 { "FDIV", 0, fdiv[4] },
3327:                 { "DDIV", 0, ddiv[5] },
3328:                 { "IREM", 0, irem[6] },
3329:                 { "LREM", 0, lrem[7] },
3330:                 { "FREM", 0 },
3331:                 { "DREM", 0 },
3332:                 { "INEG", 0, ineg[8] },
3333:                 { "LNEG", 0, lneg[9] },
3334:                 { "FNEG", 0 },
3335:                 { "DNEG", 0 },
3336:                 { "ISHL", 0, ishl[10] },
3337:                 { "LSHL", 0 },
3338:                 { "ISHR", 0, ishr[11] },
3339:                 { "LSHR", 0 },
3340:                 { "IUSHR", 0, iushr[12] },
3341:                 { "LUSHR", 0 },
3342:                 { "IAND", 0, iand[13] },
3343:                 { "LAND", 0, land[14] },
3344:                 { "IOR", 0, ior[15] },
3345:                 { "LOR", 0, lor[16] },
3346:                 { "IXOR", 0, ixor[17] },
3347:                 { "LXOR", 0, lxor[18] },
3348:                 { "IINC", 2, iinc[19] },
3349:                 { "I2L", 0, i2l[20] },
3350:                 { "I2F", 0, i2f[21] },
3351:                 { "I2D", 0 },
3352:                 { "L2I", 0, l2i[22] },
3353:                 { "L2F", 0 },
3354:                 { "L2D", 0 },
3355:                 { "F2I", 0, f2i[23] },
3356:                 { "F2L", 0 },
3357:                 { "F2D", 0, f2d[24] },
3358:                 { "D2I", 0 },
3359:                 { "D2L", 0 },
3360:                 { "D2F", 0, d2f[25] },
3361:                 { "I2B", 0, i2b[26] },
3362:                 { "I2C", 0 },
3363:                 { "I2S", 0, i2s[27] },
3364:                 { "LCMP", 0, lcmp[28] },
3365:                 { "FCMPL", 0, fcmpl[29] },
3366:                 { "FCMPG", 0, fcmpg[30] },
3367:                 { "DCMPL", 0, dcmpl[31] },
3368:                 { "DCMPG", 0, dcmpg[32] },
3369:                 { "IFEQ", 2, ifeq[33] },
3370:                 { "IFNE", 2, ifne[34] },
3371:                 { "IFLT", 2, iflt[35] },
3372:                 { "IFGE", 2, ifge[36] },
3373:                 { "IFGT", 2, ifgt[37] },
3374:                 { "IFLE", 2, ifle[38] },
3375:                 { "IF_ICMPEQ", 2, if_icmpeq[39] },
```

Footnotes:

1: *opcodes.cpp:814*
2: *opcodes.cpp:524*
3: *opcodes.cpp:685*
4: *opcodes.cpp:797*
5: *opcodes.cpp:850*
6: *opcodes.cpp:541*
7: *opcodes.cpp:702*
8: *opcodes.cpp:640*
9: *opcodes.cpp:751*
10: *opcodes.cpp:594*
11: *opcodes.cpp:582*
12: *opcodes.cpp:557*
13: *opcodes.cpp:618*
14: *opcodes.cpp:729*
15: *opcodes.cpp:629*
16: *opcodes.cpp:740*
17: *opcodes.cpp:607*
18: *opcodes.cpp:718*
19: *opcodes.cpp:867*
20: *opcodes.cpp:952*
21: *opcodes.cpp:928*
22: *opcodes.cpp:975*
23: *opcodes.cpp:988*
24: *opcodes.cpp:999*
25: *opcodes.cpp:1010*
26: *opcodes.cpp:939*
27: *opcodes.cpp:963*
28: *opcodes.cpp:1475*
29: *opcodes.cpp:1510*
30: *opcodes.cpp:1515*
31: *opcodes.cpp:1539*
32: *opcodes.cpp:1544*
33: *opcodes.cpp:1199*
34: *opcodes.cpp:1204*
35: *opcodes.cpp:1209*
36: *opcodes.cpp:1224*
37: *opcodes.cpp:1219*
38: *opcodes.cpp:1214*
39: *opcodes.cpp:1268*

```
3376:                   { "IF_ICMPNE", 2, if_icmpne[1] },
3377:                   { "IF_ICMPLT", 2, if_icmplt[2] },
3378:                   { "IF_ICMPGE", 2, if_icmpge[3] },
3379:                   { "IF_ICMPGT", 2, if_icmpgt[4] },
3380:                   { "IF_ICMPLE", 2, if_icmple[5] },
3381:                   { "IF_ACMPEQ", 2, if_acmpeq[6] },
3382:                   { "IF_ACMPNE", 2, if_acmpne[7] },
3383:                   { "GOTO", 2, _goto[8] },
3384:                   { "JSR", 2, jsr[9] },
3385:                   { "RET", 1, ret[10] },
3386:                   { "TABLESWITCH", -1, tableswitch[11] },
3387:                   { "LOOKUPSWITCH", -1 },
3388:                   { "IRETURN", 0, ireturn[12] },
3389:                   { "LRETURN", 0, lreturn[13] },
3390:                   { "FRETURN", 0, freturn[14] },
3391:                   { "DRETURN", 0, dreturn[15] },
3392:                   { "ARETURN", 0, areturn[16] },
3393:                   { "RETURN", 0, _return[17] },
3394:                   { "GETSTATIC", 2, getstatic[18] },
3395:                   { "PUTSTATIC", 2, putstatic[19] },
3396:                   { "GETFIELD", 2, getfield[20] },
3397:                   { "PUTFIELD", 2, putfield[21] },
3398:                   { "INVOKEVIRTUAL", 2, invokevirtual[22] },
3399:                   { "INVOKESPECIAL", 2, invokespecial[23] },
3400:                   { "INVOKESTATIC", 2, invokestatic[24] },
3401:                   { "INVOKEINTERFACE", 4, invokeinterface[25] },
3402:                   { "XXXUNUSEDXXX1", -1, nop[26] },
3403:                   { "NEW", 2, _new[27] },
3404:                   { "NEWARRAY", 1, newarray[28] },
3405:                   { "ANEWARRAY", 2, anewarray[29] },
3406:                   { "ARRAYLENGTH", 0, arraylength[30] },
3407:                   { "ATHROW", 0, athrow[31] },
3408:                   { "CHECKCAST", 2, checkcast[32] },
3409:                   { "INSTANCEOF", 2, instanceof[33] },
3410:                   { "MONITORENTER", 0, monitorenter[34] },
3411:                   { "MONITOREXIT", 0, monitorexit[35] },
3412:                   { "WIDE", -1 },
3413:                   { "MULTIANEWARRAY", 3 },
3414:                   { "IFNULL", 2, ifnull[36] },
3415:                   { "IFNONNULL", 2, ifnonnull[37] },
3416:                   { "GOTO_W", 4, _goto_w[38] },
3417:                   { "JSR_W", 4 },
3418:                   { "BREAKPOINT", 0 },
3419:                   { "IMPDEP1", -1 },
3420:                   { "IMPDEP2"  -1 }
3421:           };
3422:
3423:
3424:           void debug_print_code(u1[39] * code, unsigned long code_length, ostream & out)
3425:           {
3426:                   unsigned long index = 0;
3427:
3428:                   while (index[40] < code_length[41])
```

Footnotes:

1: *opcodes.cpp:1273*
2: *opcodes.cpp:1278*
3: *opcodes.cpp:1293*
4: *opcodes.cpp:1288*
5: *opcodes.cpp:1283*
6: *opcodes.cpp:1334*
7: *opcodes.cpp:1339*
8: *opcodes.cpp:1120*
9: *opcodes.cpp:2388*
10: *opcodes.cpp:2410*
11: *opcodes.cpp:1397*
12: *opcodes.cpp:2325*
13: *opcodes.cpp:2331*
14: *opcodes.cpp:2337*
15: *opcodes.cpp:2343*
16: *opcodes.cpp:2349*
17: *opcodes.cpp:2319*
18: *opcodes.cpp:1651*
19: *opcodes.cpp:1743*
20: *opcodes.cpp:1856*
21: *opcodes.cpp:1958*
22: *opcodes.cpp:2235*
23: *opcodes.cpp:2273*
24: *opcodes.cpp:2254*
25: *opcodes.cpp:2292*
26: *opcodes.cpp:16*
27: *opcodes.cpp:2105*
28: *opcodes.cpp:2202*
29: *opcodes.cpp:2151*
30: *opcodes.cpp:3186*
31: *opcodes.cpp:2359*
32: *opcodes.cpp:1068*
33: *opcodes.cpp:1021*
34: *opcodes.cpp:2422*
35: *opcodes.cpp:2445*
36: *opcodes.cpp:1345*
37: *opcodes.cpp:1368*
38: *opcodes.cpp:1138*
39: *defs.h:168*
40: *opcodes.cpp:3426*
41: *opcodes.cpp:3424*

```
3429:                   {
3430:                           u1[1] opcode = code[2][index[3]];
3431:                           out[4] << "[" << index[3] << "] ";
3432:                           out[4] << opcodes[5][opcode[6]].mnemonic[7] << endl;
3433:                           // TMP
3434:                           if (opcodes[5][opcode[6]].argument_number[8] < 0)
3435:                                   return;
3436:                           // TMP
3437:                           for (int i = 0; i < opcodes[5][opcode[6]].argument_number[8]; i++)
3438:                           {
3439:                                   index[3]++;
3440:                                   out[4] << "   arg #" << (i[9]+1) << /*": " << code[index] <<*/ endl;
3441:                           }
3442:
3443:                           // move forward to the next command
3444:                           index[3]++;
3445:                   }
3446:           }
3447:
```

Footnotes:

1: *defs.h:168*
2: *opcodes.cpp:3424*
3: *opcodes.cpp:3426*
4: *opcodes.cpp:3424*
5: *opcodes.cpp:3215*
6: *opcodes.cpp:3430*
7: *opcodes.h:256*
8: *opcodes.h:257*
9: *opcodes.cpp:3437*

```
 1:          #include <iostream.h>
 2:
 3:          #include "defs.h"
 4:
 5:
 6:          // Convert the wide-char string into a char string assuming that the wide-char string
 7:          // consists of ASCII characters only
 8:          void wchar2ascii(wchar_t * wcharstr, char * charstr, unsigned int length)
 9:          {
10:                  for (int i = 0; i < length[1]; i++)
11:                          charstr[2][i[3]] = (char)(wcharstr[4][i[3]] & 0xFF); // truncate the higher byte
12:
13:                  charstr[2][i] = 0;
14:          }
15:
16:          void ascii2wchar(char * charstr, wchar_t * wcharstr)
17:          {
18:                  for (int i = 0; charstr[5][i]; i++)
19:                          wcharstr[6][i[7]] = charstr[5][i[7]]; // copy the lower byte
20:
21:                  wcharstr[6][i] = 0;
22:          }
23:
24:          void slash2backslash(char * str)
25:          {
26:                  for (int i = 0; str[8][i]; i++)
27:                          str[8][i[9]] = (str[8][i[9]] == '/' ? '\\' : str[8][i[9]]);
28:          }
29:
30:          void slash2underscore(char * str)
31:          {
32:                  for (int i = 0; str[10][i]; i++)
33:                          str[10][i[11]] = (str[10][i[11]] == '/' ? '_' : str[10][i[11]]);
34:          }
35:
36:          void dot2slash(char * str)
37:          {
38:                  for (int i = 0; str[12][i]; i++)
39:                          str[12][i[13]] = (str[12][i[13]] == '.' ? '/' : str[12][i[13]]);
40:          }
41:
42:          void print_wchar(wchar_t * str, unsigned int length, ostream & out)
43:          {
44:                  for (int i = 0; i < length[14]; i++)
45:                          out[15] << (char)(str[16][i[17]] & 0xFF); // truncate the higher byte
46:                  //cout << endl;
47:          }
48:
49:          void general_type[18]::debug_print()
50:          {
51:                  switch (kind[19])
52:                  {
53:                  case general_type[18]::Basic[20]:
```

Footnotes:

1: *util.cpp:8*
2: *util.cpp:8*
3: *util.cpp:10*
4: *util.cpp:8*
5: *util.cpp:16*
6: *util.cpp:16*
7: *util.cpp:18*
8: *util.cpp:24*
9: *util.cpp:26*
10: *util.cpp:30*
11: *util.cpp:32*
12: *util.cpp:36*
13: *util.cpp:38*
14: *util.cpp:42*
15: *util.cpp:42*
16: *util.cpp:42*
17: *util.cpp:44*
18: *defs.h:231*
19: *defs.h:233*
20: *defs.h:233*

```
54:                         cout << "BASIC: ";
55:                         switch (type[1])
56:                         {
57:                         case Byte[2]: cout << "Byte" << endl; break;
58:                         case Char[3]: cout << "Char" << endl; break;
59:                         case Double[4]: cout << "Double" << endl; break;
60:                         case Float[5]: cout << "Float" << endl; break;
61:                         case Int[6]: cout << "Int" << endl; break;
62:                         case Long[7]: cout << "Long" << endl; break;
63:                         case Short[8]: cout << "Short" << endl; break;
64:                         case Boolean[9]: cout << "Boolean" << endl; break;
65:                         case Void[10]: cout << "Void" << endl; break;
66:                         default: cout << "Unknown" << endl;
67:                         }
68:                         break;
69:                         case general_type[11]::Array[12]:
70:                                 cout << "ARRAY [" << dimension[13] << "] of type " << endl;
71:                                 if (array_kind[14] == general_type[11]::Basic[15])
72:                                 {
73:                                         switch (type[1])
74:                                         {
75:                                         case Byte[2]: cout << "Byte" << endl; break;
76:                                         case Char[3]: cout << "Char" << endl; break;
77:                                         case Double[4]: cout << "Double" << endl; break;
78:                                         case Float[5]: cout << "Float" << endl; break;
79:                                         case Int[6]: cout << "Int" << endl; break;
80:                                         case Long[7]: cout << "Long" << endl; break;
81:                                         case Short[8]: cout << "Short" << endl; break;
82:                                         case Boolean[9]: cout << "Boolean" << endl; break;
83:                                         case Void[10]: cout << "Void" << endl; break;
84:                                         default: cout << "Unknown" << endl;
85:                                         }
86:                                 }
87:                                 else
88:                                         cout << "reference on " <<  qualified_name[16] << endl;
89:                         break;
90:                 case general_type[11]::Reference[17]:
91:                         cout << "REFERENCE on " << qualified_name[16] << endl;
92:                         break;
93:                 }
94:         }
```

Footnotes:

1: *defs.h:237*
2: *defs.h:210*
3: *defs.h:211*
4: *defs.h:212*
5: *defs.h:213*
6: *defs.h:214*
7: *defs.h:215*
8: *defs.h:217*
9: *defs.h:218*
10: *defs.h:220*
11: *defs.h:231*
12: *defs.h:233*
13: *defs.h:236*
14: *defs.h:234*
15: *defs.h:233*
16: *defs.h:235*
17: *defs.h:233*

```
 1:          #include <assert.h>
 2:
 3:          #include "jni.h"
 4:          #include "jni_wrappers.h"
 5:          #include "VirtualMachine.h"
 6:          #include "ObjectData.h"
 7:          #include "ClassFile.h"
 8:
 9:          #ifdef __cplusplus
10:          extern "C" {
11:          #endif
12:          /* Inaccessible static: in */
13:          /* Inaccessible static: out */
14:          /* Inaccessible static: err */
15:          /*
16:           * Class:     java_io_FileDescriptor
17:           * Method:    sync
18:           * Signature: ()V
19:           */
20:          JNIEXPORT void JNICALL Java_java_io_FileDescriptor_sync
21:            (JNIEnv *, jobject);
22:
23:          /*
24:           * Class:     java_io_FileDescriptor
25:           * Method:    initIDs
26:           * Signature: ()V
27:           */
28:          JNIEXPORT void JNICALL Java_java_io_FileDescriptor_initIDs(JNIEnv *, jclass)
29:          {
30:          }
31:
32:          #ifdef __cplusplus
33:          }
34:          #endif
35:
36:          void Java_java_io_FileDescriptor_register(JNIEnv * env)
37:          {
38:                  JavaVM * vm;
39:                  env[1]->GetJavaVM(&vm[2]);
40:                  VirtualMachine[3] * my_jvm = (VirtualMachine[3]*)vm[2];
41:                  my_jvm[4]->register_method[5]("Java_java_io_FileDescriptor_initIDs", "()V", Java_java_io_FileDescriptor_initIDs[6]);
42:          }
43:
```

Footnotes:

1: *java_io_FileDescriptor.cpp:36*
2: *java_io_FileDescriptor.cpp:38*
3: *VirtualMachine.h:255*
4: *java_io_FileDescriptor.cpp:40*
5: *VirtualMachine.cpp:309*
6: *java_io_FileDescriptor.cpp:28*

```
 1:          #include <assert.h>
 2:
 3:          #include "jni.h"
 4:          #include "jni_wrappers.h"
 5:          #include "VirtualMachine.h"
 6:          #include "ObjectData.h"
 7:          #include "ClassFile.h"
 8:
 9:
10:          #ifdef __cplusplus
11:          extern "C" {
12:          #endif
13:          #undef java_io_FileInputStream_SKIP_BUFFER_SIZE[1]
14:          #define java_io_FileInputStream_SKIP_BUFFER_SIZE 2048L
15:          /* Inaccessible static: skipBuffer */
16:          /*
17:           * Class:     java_io_FileInputStream
18:           * Method:    open
19:           * Signature: (Ljava/lang/String;)V
20:           */
21:          JNIEXPORT void JNICALL Java_java_io_FileInputStream_open
22:            (JNIEnv *, jobject, jstring);
23:
24:          /*
25:           * Class:     java_io_FileInputStream
26:           * Method:    read
27:           * Signature: ()I
28:           */
29:          JNIEXPORT jint JNICALL Java_java_io_FileInputStream_read
30:            (JNIEnv *, jobject);
31:
32:          /*
33:           * Class:     java_io_FileInputStream
34:           * Method:    readBytes
35:           * Signature: ([BII)I
36:           */
37:          JNIEXPORT jint JNICALL Java_java_io_FileInputStream_readBytes
38:            (JNIEnv *, jobject, jbyteArray, jint, jint);
39:
40:          /*
41:           * Class:     java_io_FileInputStream
42:           * Method:    skip
43:           * Signature: (J)J
44:           */
45:          JNIEXPORT jlong JNICALL Java_java_io_FileInputStream_skip
46:            (JNIEnv *, jobject, jlong);
47:
48:          /*
49:           * Class:     java_io_FileInputStream
50:           * Method:    available
51:           * Signature: ()I
52:           */
53:          JNIEXPORT jint JNICALL Java_java_io_FileInputStream_available
```

Footnotes:

1: *java_io_FileInputStream.cpp:14*

```
54:             (JNIEnv *, jobject);
55:
56:         /*
57:          * Class:     java_io_FileInputStream
58:          * Method:    close
59:          * Signature: ()V
60:          */
61:         JNIEXPORT void JNICALL Java_java_io_FileInputStream_close
62:           (JNIEnv *, jobject);
63:
64:         /*
65:          * Class:     java_io_FileInputStream
66:          * Method:    initIDs
67:          * Signature: ()V
68:          */
69:         JNIEXPORT void JNICALL Java_java_io_FileInputStream_initIDs(JNIEnv *, jclass)
70:         {
71:         }
72:
73:         #ifdef __cplusplus
74:         }
75:         #endif
76:
77:
78:         void Java_java_io_FileInputStream_register(JNIEnv * env)
79:         {
80:                 JavaVM * vm;
81:                 env[1]->GetJavaVM(&vm[2]);
82:                 VirtualMachine[3] * my_jvm = (VirtualMachine[3]*)vm[2];
83:                 my_jvm[4]->register_method[5]("Java_java_io_FileInputStream_initIDs", "()V", Java_java_io_FileInputStr
        eam_initIDs[6]);
84:         }
85:
```

Footnotes:

1: *java_io_FileInputStream.cpp:78*
2: *java_io_FileInputStream.cpp:80*
3: *VirtualMachine.h:255*
4: *java_io_FileInputStream.cpp:82*
5: *VirtualMachine.cpp:309*
6: *java_io_FileInputStream.cpp:69*

```
 1:          #include <assert.h>
 2:          #include <iostream.h>
 3:
 4:          #include "jni.h"
 5:          #include "jni_wrappers.h"
 6:          #include "VirtualMachine.h"
 7:          #include "ObjectData.h"
 8:          #include "ClassFile.h"
 9:          #include "Thread.h"
10:
11:          #ifdef __cplusplus
12:          extern "C" {
13:          #endif
14:          /*
15:           * Class:     java_io_FileOutputStream
16:           * Method:    open
17:           * Signature: (Ljava/lang/String;)V
18:           */
19:          JNIEXPORT void JNICALL Java_java_io_FileOutputStream_open
20:            (JNIEnv *, jobject, jstring);
21:
22:          /*
23:           * Class:     java_io_FileOutputStream
24:           * Method:    openAppend
25:           * Signature: (Ljava/lang/String;)V
26:           */
27:          JNIEXPORT void JNICALL Java_java_io_FileOutputStream_openAppend
28:            (JNIEnv *, jobject, jstring);
29:
30:          /*
31:           * Class:     java_io_FileOutputStream
32:           * Method:    write
33:           * Signature: (I)V
34:           */
35:          JNIEXPORT void JNICALL Java_java_io_FileOutputStream_write(JNIEnv *, jobject , jint byte)
36:          {
37:                  cout << "PRINTING (BYTE):" << endl;
38:                  cout << byte[1];
39:          }
40:
41:          /*
42:           * Class:     java_io_FileOutputStream
43:           * Method:    writeBytes
44:           * Signature: ([BII)V
45:           */
46:          JNIEXPORT void JNICALL Java_java_io_FileOutputStream_writeBytes(JNIEnv * env, jobject, jbyteArray array,
          jint offset, jint length)
47:          {
48:
49:                  // DEBUG ONLY:
50:                  JavaVM * vm;
51:                  env[2]->GetJavaVM(&vm[3]);
52:                  VirtualMachine[4] * my_jvm = (VirtualMachine[4]*)vm[3];
```

Footnotes:

1: *java_io_FileOutputStream.cpp:35*
2: *java_io_FileOutputStream.cpp:46*
3: *java_io_FileOutputStream.cpp:50*
4: *VirtualMachine.h:255*

```
53:                 Thread[1] * cur_thread = my_jvm[2]->current_thread[3];
54:                 cout << "[" << cur_thread[4]->get_id[5]() << "] ";
55:                 ////////////////////////
56:
57:                 InstanceData[6] * id = (InstanceData[6]*)array[7];
58:                 if (id[8] == NULL)
59:                 {
60:                         // TODO: raise exception
61:                         return;
62:                 }
63:
64:
65:                 for (int count = 0, i = 0;;i++)
66:                 {
67:                         u2[9] * byte = (u2[9]*)(id[8]->data_start[10]->address[11] + offset[12] + i[13]*sizeof(u2[9]));
68:                         if (++count[14] > length[15]) break;
69:                         u1[16] low = (*byte[17]) & 0xFF;
70:                         cout << low[18]; cout.flush();
71:                         if (++count[14] > length[15]) break;
72:                         u1[16] high = (*byte[17]) >> 8;
73:                         cout << high[19]; cout.flush();
74:                 }
75:
76:                 //cout << endl;
77:         }
78:
79:         /*
80:          * Class:     java_io_FileOutputStream
81:          * Method:    close
82:          * Signature: ()V
83:          */
84:         JNIEXPORT void JNICALL Java_java_io_FileOutputStream_close
85:           (JNIEnv *, jobject);
86:
87:         /*
88:          * Class:     java_io_FileOutputStream
89:          * Method:    initIDs
90:          * Signature: ()V
91:          */
92:         JNIEXPORT void JNICALL Java_java_io_FileOutputStream_initIDs(JNIEnv *, jclass)
93:         {
94:         }
95:
96:         #ifdef __cplusplus
97:         }
98:         #endif
99:
100:        void Java_java_io_FileOutputStream_register(JNIEnv * env)
101:        {
102:                JavaVM * vm;
103:                env[20]->GetJavaVM(&vm[21]);
104:                VirtualMachine[22] * my_jvm = (VirtualMachine[22]*)vm[21];
                   my_jvm[23]->register_method[24]("Java_java_io_FileOutputStream_initIDs", "()V", Java_java_io_FileOutput
```

Footnotes:

1: *Thread.h:17*
2: *java_io_FileOutputStream.cpp:52*
3: *VirtualMachine.h:287*
4: *java_io_FileOutputStream.cpp:53*
5: *Thread.h:82*
6: *ObjectData.h:113*
7: *java_io_FileOutputStream.cpp:46*
8: *java_io_FileOutputStream.cpp:57*
9: *defs.h:172*
10: *ObjectData.h:71*
11: *HeapManager.h:46*
12: *java_io_FileOutputStream.cpp:46*
13: *java_io_FileOutputStream.cpp:65*
14: *java_io_FileOutputStream.cpp:65*
15: *java_io_FileOutputStream.cpp:46*
16: *defs.h:168*
17: *java_io_FileOutputStream.cpp:67*
18: *java_io_FileOutputStream.cpp:69*
19: *java_io_FileOutputStream.cpp:72*
20: *java_io_FileOutputStream.cpp:100*
21: *java_io_FileOutputStream.cpp:102*
22: *VirtualMachine.h:255*
23: *java_io_FileOutputStream.cpp:104*
24: *VirtualMachine.cpp:309*

```
        Stream_initIDs[1]);
106:            my_jvm[2]->register_method[3]("Java_java_io_FileOutputStream_writeBytes","([BII)V",Java_java_io_FileOu
        tputStream_writeBytes[4]);
107:            my_jvm[2]->register_method[3]("Java_java_io_FileOutputStream_write","(I)V",Java_java_io_FileOutputStre
        am_write[5]);
108:        }
109:
110:
111:
105:
```

Footnotes:

1: *java_io_FileOutputStream.cpp:92*
2: *java_io_FileOutputStream.cpp:104*
3: *VirtualMachine.cpp:309*
4: *java_io_FileOutputStream.cpp:46*
5: *java_io_FileOutputStream.cpp:35*

```
 1:          #include <assert.h>
 2:
 3:          #include "jni.h"
 4:          #include "jni_wrappers.h"
 5:          #include "VirtualMachine.h"
 6:          #include "ObjectData.h"
 7:          #include "ClassFile.h"
 8:          #include "HandlePool.h"
 9:          #include "ClassLoader.h"
10:
11:          #ifdef __cplusplus
12:          extern "C" {
13:          #endif
14:
15:
16:          /*
17:           * Class:     java_lang_Class
18:           * Method:    forName0
19:           * Signature: (Ljava/lang/String;ZLjava/lang/ClassLoader;)Ljava/lang/Class;
20:           */
21:          JNIEXPORT jclass JNICALL Java_java_lang_Class_forName0(JNIEnv * env, jclass, jstring string_class_name, j
             boolean initialize, jobject class_loader)
22:          {
23:                  // TODO: For the time being we ignore the initialize and the class_loader parameters
24:                  JavaVM * vm;
25:                  env[1]->GetJavaVM(&vm[2]);
26:                  VirtualMachine[3] * my_jvm = (VirtualMachine[3]*)vm[2];
27:                  ClassLoader[4] * cl = my_jvm[5]->get_bootstrap_class_loader[6]();
28:                  assert(cl[7] != NULL);
29:
30:                  InstanceData[8] * id = (InstanceData[8]*)string_class_name[9];
31:                  char * class_name = my_jvm[5]->get_handle_pool[10]()->get_interned_string[11](id[12]);
32:                  if (class_name[13] == NULL)
33:                          return null[14];
34:                  char * tmp = new char[strlen(class_name[13])+1];
35:                  dot2slash[15](class_name[13]);
36:
37:                  InstanceData[8] * class_id = cl[7]->find_class[16](class_name[13]);
38:                  return (jclass)class_id[17];
39:          }
40:
41:          /*
42:           * Class:     java_lang_Class
43:           * Method:    newInstance0
44:           * Signature: ()Ljava/lang/Object;
45:           */
46:          JNIEXPORT jobject JNICALL Java_java_lang_Class_newInstance0(JNIEnv *, jobject)
47:          {
48:                  return 0;
49:          }
50:
51:          /*
52:           * Class:     java_lang_Class
```

Footnotes:

1: *java_lang_Class.cpp:21*
2: *java_lang_Class.cpp:24*
3: *VirtualMachine.h:255*
4: *ClassLoader.h:14*
5: *java_lang_Class.cpp:26*
6: *VirtualMachine.h:334*
7: *java_lang_Class.cpp:27*
8: *ObjectData.h:113*
9: *java_lang_Class.cpp:21*
10: *VirtualMachine.h:336*
11: *HandlePool.cpp:34*
12: *java_lang_Class.cpp:30*
13: *java_lang_Class.cpp:31*
14: *defs.h:201*
15: *util.cpp:36*
16: *ClassLoader.cpp:200*
17: *java_lang_Class.cpp:37*

```
 * Method:    isInstance
 * Signature: (Ljava/lang/Object;)Z
 */
JNIEXPORT jboolean JNICALL Java_java_lang_Class_isInstance(JNIEnv *, jobject, jobject)
{
        return 0;
}

/*
 * Class:     java_lang_Class
 * Method:    isAssignableFrom
 * Signature: (Ljava/lang/Class;)Z
 */
JNIEXPORT jboolean JNICALL Java_java_lang_Class_isAssignableFrom(JNIEnv *, jobject, jclass)
{
        return 0;
}

/*
 * Class:     java_lang_Class
 * Method:    isInterface
 * Signature: ()Z
 */
JNIEXPORT jboolean JNICALL Java_java_lang_Class_isInterface(JNIEnv *, jobject)
{
        return 0;
}

/*
 * Class:     java_lang_Class
 * Method:    isArray
 * Signature: ()Z
 */
JNIEXPORT jboolean JNICALL Java_java_lang_Class_isArray(JNIEnv *, jobject)
{
        return 0;
}

/*
 * Class:     java_lang_Class
 * Method:    isPrimitive
 * Signature: ()Z
 */
JNIEXPORT jboolean JNICALL Java_java_lang_Class_isPrimitive(JNIEnv *, jobject)
{
        return 0;
}

/*
 * Class:     java_lang_Class
 * Method:    getName
 * Signature: ()Ljava/lang/String;
 */
```

```
106:          JNIEXPORT jstring JNICALL Java_java_lang_Class_getName(JNIEnv *, jobject)
107:          {
108:                  return 0;
109:          }
110:
111:          /*
112:           * Class:     java_lang_Class
113:           * Method:    getClassLoader0
114:           * Signature: ()Ljava/lang/ClassLoader;
115:           */
116:          JNIEXPORT jobject JNICALL Java_java_lang_Class_getClassLoader0(JNIEnv *, jobject)
117:          {
118:                  return 0;
119:          }
120:
121:          /*
122:           * Class:     java_lang_Class
123:           * Method:    getSuperclass
124:           * Signature: ()Ljava/lang/Class;
125:           */
126:          JNIEXPORT jclass JNICALL Java_java_lang_Class_getSuperclass(JNIEnv *, jobject)
127:          {
128:                  return 0;
129:          }
130:
131:          /*
132:           * Class:     java_lang_Class
133:           * Method:    getInterfaces
134:           * Signature: ()[Ljava/lang/Class;
135:           */
136:          JNIEXPORT jobjectArray JNICALL Java_java_lang_Class_getInterfaces(JNIEnv *, jobject)
137:          {
138:                  return 0;
139:          }
140:
141:          /*
142:           * Class:     java_lang_Class
143:           * Method:    getComponentType
144:           * Signature: ()Ljava/lang/Class;
145:           */
146:          JNIEXPORT jclass JNICALL Java_java_lang_Class_getComponentType(JNIEnv *, jobject)
147:          {
148:                  return 0;
149:          }
150:
151:          /*
152:           * Class:     java_lang_Class
153:           * Method:    getModifiers
154:           * Signature: ()I
155:           */
156:          JNIEXPORT jint JNICALL Java_java_lang_Class_getModifiers(JNIEnv *, jobject)
157:          {
158:                  return 0;
```

```
159:        }
160:
161:        /*
162:         * Class:     java_lang_Class
163:         * Method:    getSigners
164:         * Signature: ()[Ljava/lang/Object;
165:         */
166:        JNIEXPORT jobjectArray JNICALL Java_java_lang_Class_getSigners(JNIEnv *, jobject)
167:        {
168:                return 0;
169:        }
170:
171:        /*
172:         * Class:     java_lang_Class
173:         * Method:    setSigners
174:         * Signature: ([Ljava/lang/Object;)V
175:         */
176:        JNIEXPORT void JNICALL Java_java_lang_Class_setSigners(JNIEnv * env, jobject, jobjectArray)
177:        {
178:                JavaVM * vm;
179:                env[1]->GetJavaVM(&vm[2]);
180:                VirtualMachine[3] * my_jvm = (VirtualMachine[3]*)vm[2];
181:        }
182:
183:        /*
184:         * Class:     java_lang_Class
185:         * Method:    getDeclaringClass
186:         * Signature: ()Ljava/lang/Class;
187:         */
188:        JNIEXPORT jclass JNICALL Java_java_lang_Class_getDeclaringClass(JNIEnv *, jobject)
189:        {
190:                return 0;
191:        }
192:
193:        /*
194:         * Class:     java_lang_Class
195:         * Method:    getProtectionDomain0
196:         * Signature: ()Ljava/security/ProtectionDomain;
197:         */
198:        JNIEXPORT jobject JNICALL Java_java_lang_Class_getProtectionDomain0(JNIEnv *, jobject)
199:        {
200:                return 0;
201:        }
202:
203:        /*
204:         * Class:     java_lang_Class
205:         * Method:    setProtectionDomain0
206:         * Signature: (Ljava/security/ProtectionDomain;)V
207:         */
208:        JNIEXPORT void JNICALL Java_java_lang_Class_setProtectionDomain0(JNIEnv *, jobject, jobject)
209:        {
210:        }
211:
```

Footnotes:

1: *java_lang_Class.cpp:176*
2: *java_lang_Class.cpp:178*
3: *VirtualMachine.h:255*

```
212:        /*
213:         * Class:     java_lang_Class
214:         * Method:    getPrimitiveClass
215:         * Signature: (Ljava/lang/String;)Ljava/lang/Class;
216:         */
217:        JNIEXPORT jclass JNICALL Java_java_lang_Class_getPrimitiveClass(JNIEnv * env, jclass, jstring type_name)
218:        {
219:                // From the JVM spec:
220:                // The fully qualified name of a primitive type is the keyword for that primitive type,
221:                // namely, boolean, char, byte, short, int, long, float, or double.
222:
223:            const char * ascii_type_name = env[1]->GetStringUTFChars(type_name[2], 0);
224:                // Note that the primitive type class will always be "found" (or created on demand)
225:                jclass primitive_class = env[1]->FindClass(ascii_type_name[3]);
226:            env[1]->ReleaseStringUTFChars(type_name[2], ascii_type_name[3]);
227:
228:                return primitive_class[4];
229:        }
230:
231:        /*
232:         * Class:     java_lang_Class
233:         * Method:    getFields0
234:         * Signature: (I)[Ljava/lang/reflect/Field;
235:         */
236:        JNIEXPORT jobjectArray JNICALL Java_java_lang_Class_getFields0(JNIEnv *, jobject, jint)
237:        {
238:                return 0;
239:        }
240:
241:        /*
242:         * Class:     java_lang_Class
243:         * Method:    getMethods0
244:         * Signature: (I)[Ljava/lang/reflect/Method;
245:         */
246:        JNIEXPORT jobjectArray JNICALL Java_java_lang_Class_getMethods0(JNIEnv *, jobject, jint)
247:        {
248:                return 0;
249:        }
250:
251:        /*
252:         * Class:     java_lang_Class
253:         * Method:    getConstructors0
254:         * Signature: (I)[Ljava/lang/reflect/Constructor;
255:         */
256:        JNIEXPORT jobjectArray JNICALL Java_java_lang_Class_getConstructors0(JNIEnv *, jobject, jint)
257:        {
258:                return 0;
259:        }
260:
261:        /*
262:         * Class:     java_lang_Class
263:         * Method:    getField0
264:         * Signature: (Ljava/lang/String;I)Ljava/lang/reflect/Field;
```

Footnotes:

1: *java_lang_Class.cpp:217*
2: *java_lang_Class.cpp:217*
3: *java_lang_Class.cpp:223*
4: *java_lang_Class.cpp:225*

```
265:  */
266: JNIEXPORT jobject JNICALL Java_java_lang_Class_getField0(JNIEnv *, jobject, jstring, jint)
267: {
268:         return 0;
269: }
270:
271: /*
272:  * Class:     java_lang_Class
273:  * Method:    getMethod0
274:  * Signature: (Ljava/lang/String;[Ljava/lang/Class;I)Ljava/lang/reflect/Method;
275:  */
276: JNIEXPORT jobject JNICALL Java_java_lang_Class_getMethod0(JNIEnv *, jobject, jstring, jobjectArray, jint)
277: {
278:         return 0;
279: }
280:
281: /*
282:  * Class:     java_lang_Class
283:  * Method:    getConstructor0
284:  * Signature: ([Ljava/lang/Class;I)Ljava/lang/reflect/Constructor;
285:  */
286: JNIEXPORT jobject JNICALL Java_java_lang_Class_getConstructor0(JNIEnv *, jobject, jobjectArray, jint)
287: {
288:         return 0;
289: }
290:
291: /*
292:  * Class:     java_lang_Class
293:  * Method:    getDeclaredClasses0
294:  * Signature: ()[Ljava/lang/Class;
295:  */
296: JNIEXPORT jobjectArray JNICALL Java_java_lang_Class_getDeclaredClasses0 (JNIEnv *, jobject)
297: {
298:         return 0;
299: }
300:
301: /*
302:  * Class:     java_lang_Class
303:  * Method:    registerNatives
304:  * Signature: ()V
305:  */
306: JNIEXPORT void JNICALL Java_java_lang_Class_registerNatives(JNIEnv * env, jclass)
307: {
308:         JavaVM * vm;
309:         env[1]->GetJavaVM(&vm[2]);
310:         VirtualMachine[3] * my_jvm = (VirtualMachine[3]*)vm[2];
311:         my_jvm[4]->register_method[5]("Java_java_lang_Class_getPrimitiveClass", "(Ljava/lang/String;)Ljava/lang/Class;", Java_java_lang_Class_getPrimitiveClass[6]);
312:         my_jvm[4]->register_method[5]("Java_java_lang_Class_setSigners","([Ljava/lang/Object;)V",Java_java_lang_Class_setSigners[7]);
313:         my_jvm[4]->register_method[5]("Java_java_lang_Class_forName0","(Ljava/lang/String;ZLjava/lang/ClassLoader;)Ljava/lang/Class;",Java_java_lang_Class_forName0[8]);
314: }
```

Footnotes:

1: *java_lang_Class.cpp:306*
2: *java_lang_Class.cpp:308*
3: *VirtualMachine.h:255*
4: *java_lang_Class.cpp:310*
5: *VirtualMachine.cpp:309*
6: *java_lang_Class.cpp:217*
7: *java_lang_Class.cpp:176*
8: *java_lang_Class.cpp:21*

```
315:
316:          #ifdef __cplusplus
317:          }
318:          #endif
319:
320:          // This function initially called by the NativeHandler to register the "proper" registerNatives function
321:          void Java_java_lang_Class_register(JNIEnv * env)
322:          {
323:                  JavaVM * vm;
324:                  env[1]->GetJavaVM(&vm[2]);
325:                  VirtualMachine[3] * my_jvm = (VirtualMachine[3]*)vm[2];
326:                  my_jvm[4]->register_method[5]("Java_java_lang_Class_registerNatives", "()V", Java_java_lang_Class_regi
          sterNatives[6]);
327:          }
328:
329:
```

Footnotes:

1: *java_lang_Class.cpp:321*
2: *java_lang_Class.cpp:323*
3: *VirtualMachine.h:255*
4: *java_lang_Class.cpp:325*
5: *VirtualMachine.cpp:309*
6: *java_lang_Class.cpp:306*

```
1:  #include <assert.h>
2:
3:  #include "jni.h"
4:  #include "jni_wrappers.h"
5:  #include "VirtualMachine.h"
6:  #include "ObjectData.h"
7:  #include "ClassFile.h"
8:
9:  #ifdef __cplusplus
10: extern "C" {
11: #endif
12: /*
13:  * Class:     java_lang_ClassLoader
14:  * Method:    defineClass0
15:  * Signature: (Ljava/lang/String;[BIILjava/security/ProtectionDomain;)Ljava/lang/Class;
16:  */
17: JNIEXPORT jclass JNICALL Java_java_lang_ClassLoader_defineClass0
18:   (JNIEnv *, jobject, jstring, jbyteArray, jint, jint, jobject);
19:
20: /*
21:  * Class:     java_lang_ClassLoader
22:  * Method:    resolveClass0
23:  * Signature: (Ljava/lang/Class;)V
24:  */
25: JNIEXPORT void JNICALL Java_java_lang_ClassLoader_resolveClass0
26:   (JNIEnv *, jobject, jclass);
27:
28: /*
29:  * Class:     java_lang_ClassLoader
30:  * Method:    findBootstrapClass
31:  * Signature: (Ljava/lang/String;)Ljava/lang/Class;
32:  */
33: JNIEXPORT jclass JNICALL Java_java_lang_ClassLoader_findBootstrapClass
34:   (JNIEnv *, jobject, jstring);
35:
36: /*
37:  * Class:     java_lang_ClassLoader
38:  * Method:    findLoadedClass
39:  * Signature: (Ljava/lang/String;)Ljava/lang/Class;
40:  */
41: JNIEXPORT jclass JNICALL Java_java_lang_ClassLoader_findLoadedClass
42:   (JNIEnv *, jobject, jstring);
43:
44: /*
45:  * Class:     java_lang_ClassLoader
46:  * Method:    getCallerClassLoader
47:  * Signature: ()Ljava/lang/ClassLoader;
48:  */
49: JNIEXPORT jobject JNICALL Java_java_lang_ClassLoader_getCallerClassLoader(JNIEnv * env, jclass caller)
50: {
51:         JavaVM * vm;
52:         env[1]->GetJavaVM(&vm[2]);
53:         VirtualMachine[3] * my_jvm = (VirtualMachine[3]*)vm[2];
```

Footnotes:

1: *java_lang_ClassLoader.cpp:49*
2: *java_lang_ClassLoader.cpp:51*
3: *VirtualMachine.h:255*

```
54:          /*
55:                  InstanceData * id = (InstanceData*)caller;
56:                  return id->class_file->class_loader;
57:          */
58:                  return null[1];
59:          }
60:
61:          #ifdef __cplusplus
62:          }
63:          #endif
64:
65:          void Java_java_lang_ClassLoader_register(JNIEnv * env)
66:          {
67:                  JavaVM * vm;
68:                  env[2]->GetJavaVM(&vm[3]);
69:                  VirtualMachine[4] * my_jvm = (VirtualMachine[4]*)vm[3];
70:                  my_jvm[5]->register_method[6]("Java_java_lang_ClassLoader_getCallerClassLoader", "()Ljava/lang/ClassLo
          ader;", Java_java_lang_ClassLoader_getCallerClassLoader[7]);
71:          }
72:
73:
```

Footnotes:

1: *defs.h:201*
2: *java_lang_ClassLoader.cpp:65*
3: *java_lang_ClassLoader.cpp:67*
4: *VirtualMachine.h:255*
5: *java_lang_ClassLoader.cpp:69*
6: *VirtualMachine.cpp:309*
7: *java_lang_ClassLoader.cpp:49*

```
 1:          #include <assert.h>
 2:
 3:          #include "jni.h"
 4:          #include "jni_wrappers.h"
 5:          #include "VirtualMachine.h"
 6:          #include "ObjectData.h"
 7:          #include "ClassFile.h"
 8:          #include "Thread.h"
 9:
10:          #ifdef __cplusplus
11:          extern "C" {
12:          #endif
13:
14:          // Implementations of the java.lang.Object's native methods
15:
16:          /*
17:           * Class:     java_lang_Object
18:           * Method:    getClass
19:           * Signature: ()Ljava/lang/Class;
20:           */
21:          JNIEXPORT jclass JNICALL Java_java_lang_Object_getClass
22:            (JNIEnv * env, jobject obj)
23:          {
24:                  jobject_wrapper[1] * id = (jobject_wrapper[1]*)obj[2];
25:                  assert(id[3] != NULL);
26:
27:                  ClassFile[4] * cf = id[3]->class_file[5];
28:                  assert(cf[6] != NULL);
29:                  assert (cf[6]->class_instance[7] != NULL);
30:
31:                  return (jclass)cf[6]->class_instance[7];
32:          }
33:
34:          /*
35:           * Class:     java_lang_Object
36:           * Method:    hashCode
37:           * Signature: ()I
38:           */
39:          JNIEXPORT jint JNICALL Java_java_lang_Object_hashCode
40:            (JNIEnv * env, jobject obj)
41:          {
42:                  jobject_wrapper[1] * id = (jobject_wrapper[1]*)obj[8];
43:                  assert(id[9] != NULL);
44:
45:                  word[10] ref = id[9]->get_reference[11]();
46:                  if (ref[12] == null[13])
47:                          return 0;
48:
49:                  char buf[11]; // the length of string "4294967296"
50:                  ltoa(ref[12], buf[14], 10);
51:
52:                  long h=0L;
53:                  for (char * s = buf[14] ; *s != 0 ; s++)
```

Footnotes:

1: *jni_wrappers.h:16*
2: *java_lang_Object.cpp:22*
3: *java_lang_Object.cpp:24*
4: *ClassFile.h:136*
5: *ObjectData.h:74*
6: *java_lang_Object.cpp:27*
7: *ClassFile.h:243*
8: *java_lang_Object.cpp:40*
9: *java_lang_Object.cpp:42*
10: *defs.h:195*
11: *ObjectData.h:142*
12: *java_lang_Object.cpp:45*
13: *defs.h:201*
14: *java_lang_Object.cpp:49*

```
54:                     {
55:                             h[1] = h[1]*37 + *s[2];  // or  h ^= *s;  h = _lrotl(h, 1);
56:                     }
57:                     return h[1];
58:             }
59:
60:             /*
61:              * Class:     java_lang_Object
62:              * Method:    clone
63:              * Signature: ()Ljava/lang/Object;
64:              */
65:             JNIEXPORT jobject JNICALL Java_java_lang_Object_clone
66:               (JNIEnv * env, jobject obj)
67:             {
68:                     JavaVM * vm;
69:                     env[3]->GetJavaVM(&vm[4]);
70:                     VirtualMachine[5] * my_jvm = (VirtualMachine[5]*)vm[4];
71:
72:                     return NULL;
73:             }
74:
75:             /*
76:              * Class:     java_lang_Object
77:              * Method:    notify
78:              * Signature: ()V
79:              */
80:             JNIEXPORT void JNICALL Java_java_lang_Object_notify
81:               (JNIEnv * env, jobject obj)
82:             {
83:                     JavaVM * vm;
84:                     env[6]->GetJavaVM(&vm[7]);
85:                     VirtualMachine[5] * my_jvm = (VirtualMachine[5]*)vm[7];
86:
87:                     Thread[8] * thread = my_jvm[9]->current_thread[10];
88:                     assert(thread[11] != NULL);
89:                     InstanceData[12] * id = (InstanceData[12]*)obj[13];
90:                     assert(id[14] != NULL);
91:
92:                     thread[11]->notify[15](id[14]);
93:
94:             }
95:
96:             /*
97:              * Class:     java_lang_Object
98:              * Method:    notifyAll
99:              * Signature: ()V
100:             */
101:            JNIEXPORT void JNICALL Java_java_lang_Object_notifyAll
102:              (JNIEnv * env, jobject obj)
103:            {
104:                    JavaVM * vm;
105:                    env[16]->GetJavaVM(&vm[17]);
106:                    VirtualMachine[5] * my_jvm = (VirtualMachine[5]*)vm[17];
```

Footnotes:

1: *java_lang_Object.cpp:52*
2: *java_lang_Object.cpp:53*
3: *java_lang_Object.cpp:66*
4: *java_lang_Object.cpp:68*
5: *VirtualMachine.h:255*
6: *java_lang_Object.cpp:81*
7: *java_lang_Object.cpp:83*
8: *Thread.h:17*
9: *java_lang_Object.cpp:85*
10: *VirtualMachine.h:287*
11: *java_lang_Object.cpp:87*
12: *ObjectData.h:113*
13: *java_lang_Object.cpp:81*
14: *java_lang_Object.cpp:89*
15: *Thread.cpp:1027*
16: *java_lang_Object.cpp:102*
17: *java_lang_Object.cpp:104*

```
107:
108:                    Thread[1] * thread = my_jvm[2]->current_thread[3];
109:                    assert(thread[4] != NULL);
110:                    InstanceData[5] * id = (InstanceData[5]*)obj[6];
111:                    assert(id[7] != NULL);
112:
113:                    thread[4]->notifyAll[8](id[7]);
114:            }
115:
116:            /*
117:             * Class:     java_lang_Object
118:             * Method:    wait
119:             * Signature: (J)V
120:             */
121:            JNIEXPORT void JNICALL Java_java_lang_Object_wait
122:              (JNIEnv * env, jobject obj, jlong millis)
123:            {
124:                    // TODO: millis
125:
126:                    JavaVM * vm;
127:                    env[9]->GetJavaVM(&vm[10]);
128:                    VirtualMachine[11] * my_jvm = (VirtualMachine[11]*)vm[10];
129:
130:                    Thread[1] * thread = my_jvm[12]->current_thread[3];
131:                    assert(thread[13] != NULL);
132:                    InstanceData[5] * id = (InstanceData[5]*)obj[14];
133:                    assert(id[15] != NULL);
134:
135:                    thread[13]->wait[16](id[15]);
136:            }
137:
138:            /*
139:             * Class:     java_lang_Object
140:             * Method:    registerNatives
141:             * Signature: ()V
142:             */
143:            JNIEXPORT void JNICALL Java_java_lang_Object_registerNatives(JNIEnv * env, jclass clazz)
144:            {
145:                    JavaVM * vm;
146:                    env[17]->GetJavaVM(&vm[18]);
147:                    VirtualMachine[11] * my_jvm = (VirtualMachine[11]*)vm[18];
148:                    my_jvm[19]->register_method[20]("Java_java_lang_Object_getClass", "()Ljava/lang/Class;", Java_java_lang
            _Object_getClass[21]);
149:                    my_jvm[19]->register_method[20]("Java_java_lang_Object_hashCode", "()I", Java_java_lang_Object_hashCode[22]
            );
150:                    my_jvm[19]->register_method[20]("Java_java_lang_Object_clone","()Ljava/lang/Object;",Java_java_lang_Obj
            ect_clone[23]);
151:                    my_jvm[19]->register_method[20]("Java_java_lang_Object_notify","()V",Java_java_lang_Object_notify[24]);
152:                    my_jvm[19]->register_method[20]("Java_java_lang_Object_notifyAll","()V",Java_java_lang_Object_notifyAll[25]
            );
153:                    my_jvm[19]->register_method[20]("Java_java_lang_Object_wait","(J)V",Java_java_lang_Object_wait[26]);
154:            }
155:
```

Footnotes:

1: *Thread.h:17*
2: *java_lang_Object.cpp:106*
3: *VirtualMachine.h:287*
4: *java_lang_Object.cpp:108*
5: *ObjectData.h:113*
6: *java_lang_Object.cpp:102*
7: *java_lang_Object.cpp:110*
8: *Thread.cpp:1057*
9: *java_lang_Object.cpp:122*
10: *java_lang_Object.cpp:126*
11: *VirtualMachine.h:255*
12: *java_lang_Object.cpp:128*
13: *java_lang_Object.cpp:130*
14: *java_lang_Object.cpp:122*
15: *java_lang_Object.cpp:132*
16: *Thread.cpp:976*
17: *java_lang_Object.cpp:143*
18: *java_lang_Object.cpp:145*
19: *java_lang_Object.cpp:147*
20: *VirtualMachine.cpp:309*
21: *java_lang_Object.cpp:21*
22: *java_lang_Object.cpp:39*
23: *java_lang_Object.cpp:65*
24: *java_lang_Object.cpp:80*
25: *java_lang_Object.cpp:101*
26: *java_lang_Object.cpp:121*

```
156:        #ifdef __cplusplus
157:        }
158:        #endif
159:
160:        // This function initially called by the NativeHandler to register the "proper" registerNatives function
161:        void Java_java_lang_Object_register(JNIEnv * env)
162:        {
163:                JavaVM * vm;
164:                env[1]->GetJavaVM(&vm[2]);
165:                VirtualMachine[3] * my_jvm = (VirtualMachine[3]*)vm[2];
166:                my_jvm[4]->register_method[5]("Java_java_lang_Object_registerNatives", "()V", Java_java_lang_Object_re
        gisterNatives[6]);
167:        }
```

Footnotes:

1: *java_lang_Object.cpp:161*
2: *java_lang_Object.cpp:163*
3: *VirtualMachine.h:255*
4: *java_lang_Object.cpp:165*
5: *VirtualMachine.cpp:309*
6: *java_lang_Object.cpp:143*

```
 1:          #include <assert.h>
 2:          #include <sys/types.h>
 3:          #include <sys/timeb.h>
 4:
 5:          #include "jni.h"
 6:          #include "jni_wrappers.h"
 7:          #include "VirtualMachine.h"
 8:          #include "ObjectData.h"
 9:          #include "ClassFile.h"
10:          #include "ClassLoader.h"
11:
12:          #ifdef __cplusplus
13:          extern "C" {
14:          #endif
15:
16:          /*
17:           * Class:     java_lang_System
18:           * Method:    setIn0
19:           * Signature: (Ljava/io/InputStream;)V
20:           */
21:          JNIEXPORT void JNICALL Java_java_lang_System_setIn0(JNIEnv *, jclass, jobject)
22:          {
23:          }
24:
25:          /*
26:           * Class:     java_lang_System
27:           * Method:    setOut0
28:           * Signature: (Ljava/io/PrintStream;)V
29:           */
30:          JNIEXPORT void JNICALL Java_java_lang_System_setOut0(JNIEnv * env, jclass, jobject obj)
31:          {
32:                  JavaVM * vm;
33:                  env[1]->GetJavaVM(&vm[2]);
34:                  VirtualMachine[3] * my_jvm = (VirtualMachine[3]*)vm[2];
35:
36:                  ClassFile[4] * system_cf = my_jvm[5]->get_bootstrap_class_loader[6]()->get_class[7](SYSTEM_CLASS_NAME[8], wcs
          len(SYSTEM_CLASS_NAME[8]));
37:                  assert(system_cf[9] != NULL);
38:
39:                  InstanceData[10] * stream_id = (InstanceData[10]*)obj[11];
40:                  assert(stream_id[12] != NULL);
41:
42:                  field_info[13] * finfo = NULL;
43:                  Result[14] result = system_cf[9]->get_field_by_name[15](SYSTEM_OUT_NAME[16], SYSTEM_OUT_DESCRIPTOR[17], finfo[18])
          ;
44:                  if (result[19] != Success[20])
45:                  {
46:                          // raise exception
47:                          return;
48:                  }
49:                  assert(finfo[18] != NULL);
50:
51:                  // Copy the output stream reference in the "out" field of the java.lang.System class
```

Footnotes:

1: *java_lang_System.cpp:30*
2: *java_lang_System.cpp:32*
3: *VirtualMachine.h:255*
4: *ClassFile.h:136*
5: *java_lang_System.cpp:34*
6: *VirtualMachine.h:334*
7: *ClassLoader.cpp:187*
8: *defs.h:279*
9: *java_lang_System.cpp:36*
10: *ObjectData.h:113*
11: *java_lang_System.cpp:30*
12: *java_lang_System.cpp:39*
13: *ClassFile.h:68*
14: *defs.h:29*
15: *ClassFile.cpp:698*
16: *defs.h:280*
17: *defs.h:281*
18: *java_lang_System.cpp:42*
19: *java_lang_System.cpp:43*
20: *defs.h:33*

```
52:                 u1[1] * field_data = system_cf[2]->class_data[3]->data_start[4]->address[5] + finfo[6]->offset[7];
53:                 word[8] objectref = stream_id[9]->get_reference[10]();
54:                 memcpy(field_data[11], &objectref[12], sizeof(word[8]));
55:         }
56:
57:         /*
58:          * Class:     java_lang_System
59:          * Method:    setErr0
60:          * Signature: (Ljava/io/PrintStream;)V
61:          */
62:         JNIEXPORT void JNICALL Java_java_lang_System_setErr0(JNIEnv *, jclass, jobject)
63:         {
64:         }
65:
66:         /*
67:          * Class:     java_lang_System
68:          * Method:    currentTimeMillis
69:          * Signature: ()J
70:          */
71:         JNIEXPORT jlong JNICALL Java_java_lang_System_currentTimeMillis(JNIEnv *, jclass)
72:         {
73:                 _timeb timebuf;
74:                 _ftime(&timebuf[13]);
75:                 return (jlong)timebuf[13].time*1000;
76:         }
77:
78:         /*
79:          * Class:     java_lang_System
80:          * Method:    arraycopy
81:          * Signature: (Ljava/lang/Object;ILjava/lang/Object;II)V
82:          */
83:         JNIEXPORT void JNICALL Java_java_lang_System_arraycopy
84:           (JNIEnv * env, jclass, jobject src, jint src_offset, jobject dst, jint dst_offset, jint length)
85:         {
86:                 if (!src[14] || !dst[15])
87:                         return; // TODO: throw exception
88:
89:                 InstanceData[16] * array_src_id = (ArrayInstanceData[17]*)src[14];
90:                 InstanceData[16] * array_dst_id = (ArrayInstanceData[17]*)dst[15];
91:
92:                 Result[18] result = array_dst_id[19]->copy_from[20](array_src_id[21], src_offset[22], dst_offset[23], length[24]);
93:                 if (result[25] != Success[26])
94:                         return; // TODO: throw exception
95:         }
96:
97:         /*
98:          * Class:     java_lang_System
99:          * Method:    identityHashCode
100:         * Signature: (Ljava/lang/Object;)I
101:         */
102:        JNIEXPORT jint JNICALL Java_java_lang_System_identityHashCode
103:          (JNIEnv *, jclass, jobject);
104:
```

Footnotes:

1: *defs.h:168*
2: *java_lang_System.cpp:36*
3: *ClassFile.h:227*
4: *ObjectData.h:71*
5: *HeapManager.h:46*
6: *java_lang_System.cpp:42*
7: *ClassFile.h:79*
8: *defs.h:195*
9: *java_lang_System.cpp:39*
10: *ObjectData.h:142*
11: *java_lang_System.cpp:52*
12: *java_lang_System.cpp:53*
13: *java_lang_System.cpp:73*
14: *java_lang_System.cpp:84*
15: *java_lang_System.cpp:84*
16: *ObjectData.h:113*
17: *ObjectData.h:155*
18: *defs.h:29*
19: *java_lang_System.cpp:90*
20: *ObjectData.h:146*
21: *java_lang_System.cpp:89*
22: *java_lang_System.cpp:84*
23: *java_lang_System.cpp:84*
24: *java_lang_System.cpp:84*
25: *java_lang_System.cpp:92*
26: *defs.h:33*

```
105:          /*
106:           * Class:     java_lang_System
107:           * Method:    initProperties
108:           * Signature: (Ljava/util/Properties;)Ljava/util/Properties;
109:           */
110:          JNIEXPORT jobject JNICALL Java_java_lang_System_initProperties(JNIEnv *, jclass, jobject)
111:          {
112:                  return null[1];
113:          }
114:
115:          /*
116:           * Class:     java_lang_System
117:           * Method:    mapLibraryName
118:           * Signature: (Ljava/lang/String;)Ljava/lang/String;
119:           */
120:          JNIEXPORT jstring JNICALL Java_java_lang_System_mapLibraryName
121:            (JNIEnv *, jclass, jstring);
122:
123:          /*
124:           * Class:     java_lang_System
125:           * Method:    getCallerClass
126:           * Signature: ()Ljava/lang/Class;
127:           */
128:          JNIEXPORT jclass JNICALL Java_java_lang_System_getCallerClass
129:            (JNIEnv *, jclass);
130:
131:          /*
132:           * Class:     java_lang_System
133:           * Method:    registerNatives
134:           * Signature: ()V
135:           */
136:          JNIEXPORT void JNICALL Java_java_lang_System_registerNatives(JNIEnv * env, jclass clazz)
137:          {
138:                  JavaVM * vm;
139:                  env[2]->GetJavaVM(&vm[3]);
140:                  VirtualMachine[4] * my_jvm = (VirtualMachine[4]*)vm[3];
141:                  my_jvm[5]->register_method[6]("Java_java_lang_System_arraycopy", "(Ljava/lang/Object;ILjava/lang/Objec
          t;II)V", Java_java_lang_System_arraycopy[7]);
142:                  my_jvm[5]->register_method[6]("Java_java_lang_System_currentTimeMillis","()J",Java_java_lang_System_cu
          rrentTimeMillis[8]);
143:                  my_jvm[5]->register_method[6]("Java_java_lang_System_initProperties","(Ljava/util/Properties;)Ljava/ut
          il/Properties;",Java_java_lang_System_initProperties[9]);
144:                  my_jvm[5]->register_method[6]("Java_java_lang_System_setIn0","(Ljava/io/InputStream;)V",Java_java_lang
          _System_setIn0[10]);
145:                  my_jvm[5]->register_method[6]("Java_java_lang_System_setOut0","(Ljava/io/PrintStream;)V",Java_java_lan
          g_System_setOut0[11]);
146:                  my_jvm[5]->register_method[6]("Java_java_lang_System_setErr0","(Ljava/io/PrintStream;)V",Java_java_lan
          g_System_setErr0[12]);
147:          }
148:
149:          #ifdef __cplusplus
150:          }
151:          #endif
```

Footnotes:

1: *defs.h:201*
2: *java_lang_System.cpp:136*
3: *java_lang_System.cpp:138*
4: *VirtualMachine.h:255*
5: *java_lang_System.cpp:140*
6: *VirtualMachine.cpp:309*
7: *java_lang_System.cpp:83*
8: *java_lang_System.cpp:71*
9: *java_lang_System.cpp:110*
10: *java_lang_System.cpp:21*
11: *java_lang_System.cpp:30*
12: *java_lang_System.cpp:62*

```
152:
153:        // This function initially called by the NativeHandler to register the "proper" registerNatives function
154:        void Java_java_lang_System_register(JNIEnv * env)
155:        {
156:                JavaVM * vm;
157:                env[1]->GetJavaVM(&vm[2]);
158:                VirtualMachine[3] * my_jvm = (VirtualMachine[3]*)vm[2];
159:                my_jvm[4]->register_method[5]("Java_java_lang_System_registerNatives", "()V", Java_java_lang_System_re
        gisterNatives[6]);
160:        }
```

Footnotes:

1: *java_lang_System.cpp:154*
2: *java_lang_System.cpp:156*
3: *VirtualMachine.h:255*
4: *java_lang_System.cpp:158*
5: *VirtualMachine.cpp:309*
6: *java_lang_System.cpp:136*

```
 1:          #include <assert.h>
 2:
 3:          #include "jni.h"
 4:          #include "jni_wrappers.h"
 5:          #include "VirtualMachine.h"
 6:          #include "ObjectData.h"
 7:          #include "ClassFile.h"
 8:          #include "Thread.h"
 9:
10:          #ifdef __cplusplus
11:          extern "C" {
12:          #endif
13:
14:          /*
15:           * Class:     java_lang_Thread
16:           * Method:    currentThread
17:           * Signature: ()Ljava/lang/Thread;
18:           */
19:          JNIEXPORT jobject JNICALL Java_java_lang_Thread_currentThread(JNIEnv * env, jclass)
20:          {
21:                  JavaVM * vm;
22:                  env[1]->GetJavaVM(&vm[2]);
23:                  VirtualMachine[3] * my_jvm = (VirtualMachine[3]*)vm[2];
24:
25:                  Thread[4] * thread = my_jvm[5]->current_thread[6];
26:                  assert(thread[7] != NULL);
27:
28:                  InstanceData[8] * id = thread[7]->thread_instance[9];
29:                  assert(id[10] != NULL);
30:                  return (jobject)id[10];
31:          }
32:
33:          /*
34:           * Class:     java_lang_Thread
35:           * Method:    yield
36:           * Signature: ()V
37:           */
38:          JNIEXPORT void JNICALL Java_java_lang_Thread_yield(JNIEnv *, jclass)
39:          {
40:                  // In our implementation this function does nothing
41:          }
42:
43:          /*
44:           * Class:     java_lang_Thread
45:           * Method:    sleep
46:           * Signature: (J)V
47:           */
48:          JNIEXPORT void JNICALL Java_java_lang_Thread_sleep(JNIEnv *, jclass, jlong)
49:          {
50:          }
51:
52:          /*
53:           * Class:     java_lang_Thread
```

Footnotes:

1: *java_lang_Thread.cpp:19*
2: *java_lang_Thread.cpp:21*
3: *VirtualMachine.h:255*
4: *Thread.h:17*
5: *java_lang_Thread.cpp:23*
6: *VirtualMachine.h:287*
7: *java_lang_Thread.cpp:25*
8: *ObjectData.h:113*
9: *Thread.h:162*
10: *java_lang_Thread.cpp:28*

```
54:          * Method:    start
55:          * Signature: ()V
56:          */
57:         JNIEXPORT void JNICALL Java_java_lang_Thread_start(JNIEnv * env, jobject thread_instance)
58:         {
59:                 JavaVM * vm;
60:                 env[1]->GetJavaVM(&vm[2]);
61:                 VirtualMachine[3] * my_jvm = (VirtualMachine[3]*)vm[2];
62:
63:                 InstanceData[4] * id = (InstanceData[4]*)thread_instance[5];
64:                 assert(id[6] != NULL);
65:                 ClassFile[7] * cf = id[6]->class_file[8];
66:                 assert (cf[9] != NULL);
67:
68:                 Code_attribute[10] * code_attribute;
69:                 cf[9]->get_method_by_name[11](RUN_METHOD_NAME[12], RUN_METHOD_DESCRIPTOR[13], code_attribute[14]);
70:                 assert(code_attribute[14] != NULL);
71:
72:                 // Create the new thread and run it
73:                 my_jvm[15]->run_thread[16](code_attribute[14], id[6]);
74:         }
75:
76:         /*
77:          * Class:     java_lang_Thread
78:          * Method:    isInterrupted
79:          * Signature: (Z)Z
80:          */
81:         JNIEXPORT jboolean JNICALL Java_java_lang_Thread_isInterrupted(JNIEnv *, jobject, jboolean)
82:         {
83:                 return 0;
84:         }
85:
86:         /*
87:          * Class:     java_lang_Thread
88:          * Method:    isAlive
89:          * Signature: ()Z
90:          */
91:         JNIEXPORT jboolean JNICALL Java_java_lang_Thread_isAlive(JNIEnv *, jobject)
92:         {
93:                 return (jboolean)0;
94:         }
95:
96:         /*
97:          * Class:     java_lang_Thread
98:          * Method:    countStackFrames
99:          * Signature: ()I
100:          */
101:         JNIEXPORT jint JNICALL Java_java_lang_Thread_countStackFrames(JNIEnv * env, jobject)
102:         {
103:                 JavaVM * vm;
104:                 env[17]->GetJavaVM(&vm[18]);
105:                 VirtualMachine[3] * my_jvm = (VirtualMachine[3]*)vm[18];
106:
```

Footnotes:

1: *java_lang_Thread.cpp:57*
2: *java_lang_Thread.cpp:59*
3: *VirtualMachine.h:255*
4: *ObjectData.h:113*
5: *java_lang_Thread.cpp:57*
6: *java_lang_Thread.cpp:63*
7: *ClassFile.h:136*
8: *ObjectData.h:74*
9: *java_lang_Thread.cpp:65*
10: *AttributeInfo.h:55*
11: *ClassFile.cpp:572*
12: *defs.h:263*
13: *defs.h:264*
14: *java_lang_Thread.cpp:68*
15: *java_lang_Thread.cpp:61*
16: *VirtualMachine.cpp:287*
17: *java_lang_Thread.cpp:101*
18: *java_lang_Thread.cpp:103*

```
107:                Thread[1] * thread = my_jvm[2]->current_thread[3];
108:                assert(thread[4] != NULL);
109:
110:                return (jint)thread[4]->get_stack[5]()->count_stack_frames[6]();
111:        }
112:
113:        /*
114:         * Class:     java_lang_Thread
115:         * Method:    setPriority0
116:         * Signature: (I)V
117:         */
118:        JNIEXPORT void JNICALL Java_java_lang_Thread_setPriority0(JNIEnv * env, jobject thread_instance, jint pri
            ority)
119:        {
120:                InstanceData[7] * id = (InstanceData[7]*)thread_instance[8];
121:                // TODO
122:        }
123:
124:        /*
125:         * Class:     java_lang_Thread
126:         * Method:    stop0
127:         * Signature: (Ljava/lang/Object;)V
128:         */
129:        JNIEXPORT void JNICALL Java_java_lang_Thread_stop0(JNIEnv *, jobject, jobject)
130:        {
131:        }
132:
133:        /*
134:         * Class:     java_lang_Thread
135:         * Method:    suspend0
136:         * Signature: ()V
137:         */
138:        JNIEXPORT void JNICALL Java_java_lang_Thread_suspend0(JNIEnv *, jobject)
139:        {
140:        }
141:
142:        /*
143:         * Class:     java_lang_Thread
144:         * Method:    resume0
145:         * Signature: ()V
146:         */
147:        JNIEXPORT void JNICALL Java_java_lang_Thread_resume0(JNIEnv *, jobject)
148:        {
149:        }
150:
151:        /*
152:         * Class:     java_lang_Thread
153:         * Method:    interrupt0
154:         * Signature: ()V
155:         */
156:        JNIEXPORT void JNICALL Java_java_lang_Thread_interrupt0(JNIEnv *, jobject)
157:        {
158:        }
```

Footnotes:

1: *Thread.h:17*
2: *java_lang_Thread.cpp:105*
3: *VirtualMachine.h:287*
4: *java_lang_Thread.cpp:107*
5: *Thread.h:182*
6: *Stack.h:297*
7: *ObjectData.h:113*
8: *java_lang_Thread.cpp:118*

```
159:
160:        /*
161:         * Class:     java_lang_Thread
162:         * Method:    registerNatives
163:         * Signature: ()V
164:         */
165:        JNIEXPORT void JNICALL Java_java_lang_Thread_registerNatives(JNIEnv * env, jclass)
166:        {
167:                JavaVM * vm;
168:                env[1]->GetJavaVM(&vm[2]);
169:                VirtualMachine[3] * my_jvm = (VirtualMachine[3]*)vm[2];
170:
171:                my_jvm[4]->register_method[5]("Java_java_lang_Thread_currentThread", "()Ljava/lang/Thread;", Java_java
        _lang_Thread_currentThread[6]);
172:                my_jvm[4]->register_method[5]("Java_java_lang_Thread_yield", "()V",Java_java_lang_Thread_yield[7] );
173:                my_jvm[4]->register_method[5]("Java_java_lang_Thread_sleep", "(J)V",Java_java_lang_Thread_sleep[8] );
174:                my_jvm[4]->register_method[5]("Java_java_lang_Thread_start", "()V",Java_java_lang_Thread_start[9] );
175:                my_jvm[4]->register_method[5]("Java_java_lang_Thread_isInterrupted", "(Z)Z",Java_java_lang_Thread_isIn
        terrupted[10] );
176:                my_jvm[4]->register_method[5]("Java_java_lang_Thread_isAlive", "()Z",Java_java_lang_Thread_isAlive[11] );
177:                my_jvm[4]->register_method[5]("Java_java_lang_Thread_countStackFrames", "()I",Java_java_lang_Thread_co
        untStackFrames[12] );
178:                my_jvm[4]->register_method[5]("Java_java_lang_Thread_setPriority0", "(I)V",Java_java_lang_Thread_setPr
        iority0[13] );
179:
180:        }
181:
182:        #ifdef __cplusplus
183:        }
184:        #endif
185:
186:        // This function initially called by the NativeHandler to register the "proper" registerNatives function
187:        void Java_java_lang_Thread_register(JNIEnv * env)
188:        {
189:                JavaVM * vm;
190:                env[14]->GetJavaVM(&vm[15]);
191:                VirtualMachine[3] * my_jvm = (VirtualMachine[3]*)vm[15];
192:                my_jvm[16]->register_method[5]("Java_java_lang_Thread_registerNatives", "()V", Java_java_lang_Thread_re
        gisterNatives[17]);
193:        }
```

Footnotes:

1: *java_lang_Thread.cpp:165*
2: *java_lang_Thread.cpp:167*
3: *VirtualMachine.h:255*
4: *java_lang_Thread.cpp:169*
5: *VirtualMachine.cpp:309*
6: *java_lang_Thread.cpp:19*
7: *java_lang_Thread.cpp:38*
8: *java_lang_Thread.cpp:48*
9: *java_lang_Thread.cpp:57*
10: *java_lang_Thread.cpp:81*
11: *java_lang_Thread.cpp:91*
12: *java_lang_Thread.cpp:101*
13: *java_lang_Thread.cpp:118*
14: *java_lang_Thread.cpp:187*
15: *java_lang_Thread.cpp:189*
16: *java_lang_Thread.cpp:191*
17: *java_lang_Thread.cpp:165*

```
 1: #include <assert.h>
 2:
 3: #include "jni.h"
 4: #include "jni_wrappers.h"
 5: #include "VirtualMachine.h"
 6: #include "ObjectData.h"
 7: #include "ClassFile.h"
 8:
 9: #ifdef __cplusplus
10: extern "C" {
11: #endif
12: /*
13:  * Class:     java_security_AccessController
14:  * Method:    doPrivileged
15:  * Signature: (Ljava/security/PrivilegedAction;)Ljava/lang/Object;
16:  */
17: JNIEXPORT jobject JNICALL Java_java_security_AccessController_doPrivileged__Ljava_security_PrivilegedActi
    on_2
18:   (JNIEnv *, jclass, jobject);
19:
20: /*
21:  * Class:     java_security_AccessController
22:  * Method:    doPrivileged
23:  * Signature: (Ljava/security/PrivilegedAction;Ljava/security/AccessControlContext;)Ljava/lang/Object;
24:  */
25: JNIEXPORT jobject JNICALL Java_java_security_AccessController_doPrivileged__Ljava_security_PrivilegedActi
    on_2Ljava_security_AccessControlContext_2
26:   (JNIEnv *, jclass, jobject, jobject);
27:
28: /*
29:  * Class:     java_security_AccessController
30:  * Method:    doPrivileged
31:  * Signature: (Ljava/security/PrivilegedExceptionAction;)Ljava/lang/Object;
32:  */
33: JNIEXPORT jobject JNICALL Java_java_security_AccessController_doPrivileged__Ljava_security_PrivilegedExce
    ptionAction_2
34:   (JNIEnv *, jclass, jobject);
35:
36: /*
37:  * Class:     java_security_AccessController
38:  * Method:    doPrivileged
39:  * Signature: (Ljava/security/PrivilegedExceptionAction;Ljava/security/AccessControlContext;)Ljava/lang/O
    bject;
40:  */
41: JNIEXPORT jobject JNICALL Java_java_security_AccessController_doPrivileged__Ljava_security_PrivilegedExce
    ptionAction_2Ljava_security_AccessControlContext_2
42:   (JNIEnv *, jclass, jobject, jobject);
43:
44: /*
45:  * Class:     java_security_AccessController
46:  * Method:    getStackAccessControlContext
47:  * Signature: ()Ljava/security/AccessControlContext;
48:  */
```

```
49:  JNIEXPORT jobject JNICALL Java_java_security_AccessController_getStackAccessControlContext(JNIEnv *, jcla
     ss)
50:  {
51:          return null[1];
52:  }
53:
54:  /*
55:   * Class:     java_security_AccessController
56:   * Method:    getInheritedAccessControlContext
57:   * Signature: ()Ljava/security/AccessControlContext;
58:   */
59:  JNIEXPORT jobject JNICALL Java_java_security_AccessController_getInheritedAccessControlContext
60:    (JNIEnv *, jclass);
61:
62:  // TODO: fake function ////////////////////////////////////////
63:  JNIEXPORT jobject JNICALL Java_java_security_AccessController_doPrivileged(JNIEnv *, jclass, jobject, job
     ject)
64:  {
65:          return 0;
66:  }
67:  // TODO: fake function ////////////////////////////////////////
68:
69:  #ifdef __cplusplus
70:  }
71:  #endif
72:
73:
74:  void Java_java_security_AccessController_register(JNIEnv * env)
75:  {
76:          JavaVM * vm;
77:          env[2]->GetJavaVM(&vm[3]);
78:          VirtualMachine[4] * my_jvm = (VirtualMachine[4]*)vm[3];
79:          my_jvm[5]->register_method[6]("Java_java_security_AccessController_getStackAccessControlContext", "()L
     java/security/AccessControlContext;", Java_java_security_AccessController_getStackAccessControlContext[7]);
80:          my_jvm[5]->register_method[6]("Java_java_security_AccessController_doPrivileged","(Ljava/security/Priv
     ilegedExceptionAction;)Ljava/lang/Object;",Java_java_security_AccessController_doPrivileged[8]);
81:  }
```

Footnotes:

1: *defs.h:201*
2: *java_security_AccessController.cpp:74*
3: *java_security_AccessController.cpp:76*
4: *VirtualMachine.h:255*
5: *java_security_AccessController.cpp:78*
6: *VirtualMachine.cpp:309*
7: *java_security_AccessController.cpp:49*
8: *java_security_AccessController.cpp:63*

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y