Università di Pisa
Dipartimento di Informatica

Technical Report: TR-08-19

Design and implementation of the @Java system

Giacomo A. Galilei
Dipartimento di Informatica
Università di Pisa

Address: Largo B. Pontecorvo 3, 56127 Pisa, Italy. Tel: +39 050 2212700 Fax: +39 050 2212726

Design and implementation of the @Java system

Giacomo A. Galilei
Dipartimento di Informatica
Università di Pisa

Abstract

Annotations are a recent feature introduced in languages such as Java, C#, and other languages of the .NET family, which allow programmers to attach arbitrary, structured and typed metadata to their code. These languages run on top of so-called *virtual execution environments*, e.g. the JVM for Java, and the CLR for .NET languages, which allow for the run-time generation of executable code. In this report we explore how annotations and the dynamic code generation capability can be used together to provide programmers with high-level methods for dynamic generation and modification of an application's code — at run-time. The report describes the framework @Java system, the Java libraries it is constituted by, and the @Java language, which is an extension to Java allowing annotation of arbitrary statements. The strategy we developed consists in parsing a source file written using @Java language to produce a Java 5 compatible source file. Once compiled, information about annotated statements, such as their bytecode instructions, can be recovered at run-time. We developed two libraries, one that works at low level to do generic bytecode engineering called JDAsm, and one at high level called JCodeBrick. Together, these two libraries allow type-safe and totally symbolic runtime code modification and generation without any need to explicitly address bytecode instructions, letting a programmer to easily manipulate existing classes or to synthesize new ones, by inserting, deleting or extruding pieces of code.

After an overview of the Annotations in Section 2, we address the @Java language and its parser in Section 3. Then a full description of the two libraries for bytecode engineering and code manipulation follows in Section 4 and 5. We introduce then in Section 6 example cases where @Java System can be useful and we give a particular case study in Section 7. The last Section offers conclusions and ideas for future work. Finally the API documentation for the two presented libraries is given, completing the report.

Chapter 1

Introduction

The concept of *metadata*, which is data describing other data, is one of the mainstay in computer science, and has been used in a large variety of contexts, from defining database schema, to structuring digital annotations of medieval manuscripts. In this report, we are mostly interested in *program* metadata, i.e. data describing programs. The concept of program metadata arises naturally in those languages where programs are data, e.g. LISP [18]. In these languages, the normal ways to describe relationships about different pieces of data can be used equally well to annotate programs with metadata. However, program metadata are common in more traditional languages as well, although mostly in a limited way.

The various incarnation of the concept of program metadata can be characterized by five features:

- **content:** what kind of information is carried by a metadata element
- **author:** who (person or tool) assigns a value to a metadata element
- **lifetime:** when a metadata element is attached to a program element, and when (if ever) it is discarded
- **location:** where is the metadata stored (e.g., together with the code, or in a separate location)
- **target:** to which program elements can a metadata element be attached

Historically, program metadata has been used, in a *ad hoc* fashion, to convey specific type of information across various tools, systems or activities in the development cycle, or across long time spans to different persons working on a system. For example, traditional forms of *comments* can be interpreted as free-form metadata, attached to a specific lexical position in the source code when the code is written by the programmer, and discarded upon compilation. Table 1.1 lists some forms of metadata which are commonly used in programming and system development practice.

The real weakness of these historical forms is the fact that each metadata type is defined in a different way, is set and processed by specific tools, and in most cases has no associated notion of *validity* of the content (e.g., there is no

Metadata	Content	Author	Lifetime	Location	Target
comments	free text	programmer	source	source file	any lexical position as permitted by the language grammar
typing	types & signatures	compiler	source	source file	variables, functions, objects
compilation directive (e.g.#pragma)	instructions to the compiler	programmer	compile time	source	module
debugging symbols	symbol name, address, size, attributes	compiler	object	object file	module
identification tags	config tags, version numbers	compiler	object	object file	module or executable
API docs (e.g. JavaDoc)	API specifications	programmer	source, deploy to developers	source file (as comments)	functions, methods
Interface definitions	signatures	programmer	deploy to developers	IDL file (CORBA), WSDL file (web services)	functions (CORBA), methods (web services)
Versioning info (e.g., CVS)	release tags	revision control system	configuration release cycle	versioned source file	any lexical position
intellectual rights management (license info)	legal terms of use	programmer, lawyer	source (legal validity extends to executable)	source file (as comments)	module (typical), code fragment
development cycle control	links to design, rationale, tests, approvals, reviews, etc.	program manager	source to deploy	source file (as comments) or external management database	module, unit

Table 1.1: Characterization of some historical forms of metadata.

way to guarantee that a comment specifying some legal terms will be consistent with a predefined policy).

In recent times, the notion of program metadata has gained full citizenship both in the design of languages (e.g., C# [9], Java [16]) and in the corresponding execution environments (.NET CLR [10], JVM [17]). These new forms sport important differences w.r.t. the historical forms we discussed above:

- they are **general purpose**, i.e. the schema for their content can be defined by the programmer, and the mechanism to set and retrieve the content is not specific to a particular schema;
- they can be applied to **generic program elements**, with the programmer being able to declare specific restrictions about which class of elements can be designated as targets
- they have **customizable lifetime and location**, encompassing all the range from source-only metadata (as comments) to run-time metadata (as typing information with reflection).

Another interesting development linked to the mainstream adoption of virtual execution environments is the comeback of modifiable code. With the exception of quasi-quotation mechanisms [8] in certain interpreted languages like LISP [18] or MetaML [19], the possibility of modifying the running code of an application has been ruled out in language design and by operating systems (usually with the assistance of hardware devices, e.g. by using an MMU to forbid writing in memory pages containing executable code) since the seventies, on the ground of security concerns.

However, the ability to synthesize, configure, customize or adapt the running code of an application at run-time, possibly without even requiring a shutdown of the application itself, is invaluable in many circumstances, as we will see in Section 6.

While it is certainly true that allowing the uncontrolled modification of executable code is unacceptable in terms of security, very prone to introducing bugs and potentially disastrous side effects, and can easily be abused or bring an entire application to an abrupt termination, the type safety of .NET IL [10] and of JVM bytecode [17] has allowed a safer approach to the issue. In fact, the standard library in .NET explicitly include means to generate IL code on the fly, and the class loading mechanism in the JVM provides similar (albeit less programmer-friendly) features to the same effect.

Both these approaches require however that the programmer synthesizes IL or bytecode fragments "by hand", listing instruction after instruction the contents of the fragment. In short, the programmer is required to be proficient in both a high-level language (e.g., C# or Java) to write the main bulk of the application in, and in IL or bytecode, in order to write high-level code which will emit at runtime the sequence of instructions appropriate to accomplish the task at hand. Given that programmers who can efficiently write assembly code are increasingly difficult to find (as a consequence of the demise of machine-code programming), this requirement is too stringent for most scenarios.

One solution which has been proposed (and implemented) has been to provide programmatic access to the compiler for the high-level language, so that applications can generate the source code for a high-level class, then ask the

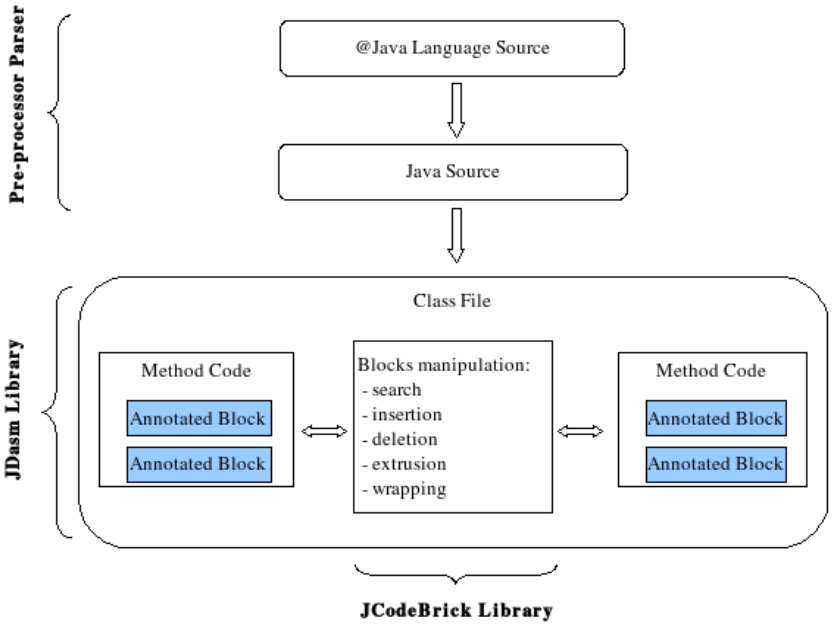

Figure 1.1: The @Java system: first an extended Java source is parsed to obtain a Java 5 conformed source; then two layer of abstraction in code manipulation are given: one more generic used for basic class file manipulation made by JDAsm library, and one more specific for code fragment operation made by the library JCodeBrick

compiler to compile it, obtain a reference to the compiled class, and finally load it and invoke some of its methods (see, among others, [5]). However, this method suffers from a number of inconveniences, including a huge performance hit (both in time and space, as the whole compiler for the high-level language needs to be loaded and executed even for a small fragment), the difficulty of programmatically generating the source code, and the possibility of introducing errors which would cause the compilation of the fragment to fail at run-time.

In this technical report we will introduce a different approach that uses program metadata to drive in a semi-declarative way the run-time synthesis of executable code. The software we developed takes the name of *@Java system* [14] and it is divided in three parts (see Figure 1): a preprocessor parser for an extension of the Java language which offers the possibility to annotate fragment of code with the Java Annotations, a generic code manipulation library called *JDAsm* [15], and a library called *JCodeBrick* [14] which uses JDAsm for the manipulation of annotated fragments.

We will refer to Java and the JVM throughout the report, but the main ideas can be applied to .NET languages as well, as in part already done in [4, 12]. Section 2 will briefly introduce Java 5 *Annotations*, and is followed in Section 3 by a presentation of the `@Java` language we defined to extend Java 5 Annotations and the preprocessor parser. Section 4 presents the low-level code manipulation library JDAsm, while 5 presents the high-level library JCodeBrick and the operations we have defined for `@Java`. Section 6 discusses a number of

applications for dynamic bytecode manipulation through annotations, while in Section 7 we give a real study case. Section 8 offers some conclusions and ideas for future work. Finally the API documentation for the two presented libraries is given, completing the report.

Chapter 2

Java Annotations

2.1 The annotations model in Java 5

The *Annotations*[1] introduced in Java 5 allow programmers to associate metadata to specific program elements. These metadata are characterized by an identifier (akin to a class or interface name) and by a signature (or schema), akin to the fields of a class, where each field has an identifier and a value. Custom Annotation types are declared with a syntax similar to that of a class, through the `@interface` keyword. Only field of basic types, String, Class, Enum, Annotation, or arrays of the same are allowed, and default values for them can be defined in the declaration of the Annotation type (see example in Figure 2.1-a.). More precisely, Annotation support in Java includes:

- a syntax to declare Annotation types (Figure 2.1-a.);
- a syntax to annotate program elements with instances of Annotation types (Figure 2.1-b.);
- an API and library to inspect through reflection the annotations associated to program elements;
- a format specification, stating how annotations are stored in .class files;
- a tool (called `apt`, annotation processor tool) for generic processing of annotations in source code at pre-compile time;
- an API and associate library for generic programmatic processing of annotations through the `apt` facilities.

Since Annotation declarations are themselves program elements, Annotations can be annotated as well. Two of these meta-annotations (i.e., program meta-meta-data, data describing how data about the program should be interpreted) are of particular relevance for our purposes. The first is the *retention policy* of an annotation type, allowing the programmer to define its life time: source only (and discarded upon compilation), source and .class (and discarded upon class loading), or runtime (preserved in the running system). The second

[1] In the following we will use the term Annotation, with a capital A, to refer to the specific form of annotations as used in Java.

```
/* a. an Annotation type to record the link between
   requirements and code */

public @interface Requirement {
   String id();
   String complianceStatement();
   String certifiedBy() default "John Doe, program manager";
   String date();
};


/* b. and its application to a method */
...
@Requirement(id="M1541", certifiedBy="Paul", date="12/5/2008")
public void applyStyle(TSpan span, Style s)
{
   ...
}
```

Figure 2.1: An example of Annotation declaration and use.

is the *target* of an annotation type, allowing the programmer to define to which program elements it can be attached: other annotations, constructors, fields, local variables, methods, packages, formal parameters, types.

It can be easily seen how with the ability to define the name, schema, lifetime, location and target of each custom annotation, all the features of our characterization of annotations from Section 1 have been placed under control of the programmer.

2.2 Limitations of the Java 5 annotation model

While the annotation model presented in the previous section is sufficiently comprehensive for the vast majority of applications, it suffers one major drawback for the purpose of dynamic code manipulation: a too coarse granularity level. In fact, while the target granularity for data is a single field, parameter, or local variables, the target granularity for code is a single method. This choice is reasonable in consideration of the fact that methods are the smallest code elements which can be found in class signatures[2], but any code manipulation system that can only manipulate entire methods could be expressed more easily by using typed "function pointers" (e.g., the delegate model in C#), and would not be suitable for fine grained optimization or configuration. We will discuss why fine grained manipulation is useful in Section 6.

Another minor limitation is that, unlike C#, only a single instance of a given annotation type can be applied to a given target, even when the Annotation fields would be different. For example, with reference to Figure 2.1, we cannot place multiple `Requirement` annotations on a method, to signify that it satisfies several requirements at once. This limitation (for which we could not find

[2]But notice that the same principle has not been applied to data, in that local variables are not visible in class signatures, while all other possible targets are.

a documented design rationale, and that apparently could be easily lifted by extending a few Reflection API methods) can be overcome by using array-typed fields, at the cost of some complication in the code. In our example, we could have used a `String` array for the `id`, `certifiedBy`, `complianceStatement` and `date` fields. This, however, is a solution which is somewhat contrived, more error-prone and less general than one could desire.

It should be noted that both these limitations have been identified in other contexts as well. For example, there is ongoing work on allowing annotations on each usage of a type which are being discussed for adoption in Java 7 [11] (the same document cites allowing multiple instances of an annotation on the same target as an example of possible developments).

Chapter 3

The @Java language

Following the example set in [4], we propose to extend the Java language in order to allow Annotations to be placed on code fragments inside a method, or more precisely, on any statement[1]. The resulting language, called @Java, can be reduced to Java 5 by a preprocessor, which serves as compiler for the language.

3.1 Syntax extension

@Java differs from Java by a single syntax rule, namely

```
Statement ::= Annotations Statement
```

We refer here to the Java 5 grammar as presented in [16]§18.1; a more concrete definition which exploits lookahead to optimize parsing time in the implementation is provided in Chapter 3.3, which allows annotations to be placed in front of any statement.

A typical example of a fragment of @Java code is presented in Figure 3.1, where a statement annotation @Parallel is used to indicate that all iterations of a for loop could be executed in parallel.

[1]The same technique could be applied to (sub-)expressions, but the usefulness of such micro-granularity does not appear to be worth the additional complexity in practical applications. Also, since in Java expressions can be turned into statements by appending a ";" as statement terminator, most typical cases are covered already by our approach.

```
int i;
double t=0;
for (i=0; i<a.length; i++)
   t+=a[i];
@Parallel for (i=0; i<a.length; i++)
   a[i] /= t;
```

Figure 3.1: An example of statement annotation in @Java.

3.2 Compilation strategy

The strategy we adopted is based upon a source-to-source conversion carried out by a preprocessor parser.

Definition 3.2.1. *The conversion from an* `@Java` *source to a Java 5 source is performed applying these two rules*

1. an annotated statement of the form $A(\vec{v})\ S$ or $A\ S$, where A is an Annotation and S is a statement, is replaced with a block of the form $\{\ K_b(k)\ S\ K_e(k)\ \}$ where $K_b(k)$ and $K_e(k)$ are special statements which serve as markers (these will be discussed in the following), and k is a unique identifier for the statement annotation instance.

2. an Annotation is generated for the method containing the annotated statement, with two fields: an array of identifiers *ids* and, in parallel, an array of Annotations *anns*. The contents of these arrays are initialized so that, for each index i, $id[i] = k$ and $anns[i] = A(\vec{v})$ (or A if the form of annotation without arguments was used).

Thus, statement annotations are lifted to the method (a legal target according to Java), stored in an array of Annotations (to overcome the problem with multiple instances of the same Annotation type on a single target), and linked by its index i to the unique identifier k in the parallel array of identifiers, which is also part of the method annotation. k in its turn is used to link to the marker statements $K_b(k)$ and $K_e(k)$. These statements must be such that:

Property 3.2.2. *their presence does not alter the semantics of the program*

This can can be obtained by using as markers method calls to non-final, static methods of a special dummy class, with empty bodies. Since their bodies are empty calling these methods does not alter the semantics.

Property 3.2.3. *they can be localized in compiled bytecode, together with their unique key k*

This can be obtained by using k as the only argument of the method calls. The method call sequence consisting of a `iconst_n`, or `bipush`, or `sipush`, or `ldc`, or `ldc_w` instruction to push k on the stack, followed by a `invokestatic` instruction to a distinguished method is easily identifiable in the code.

Property 3.2.4. *they cannot be optimized away or otherwise corrupted by any Java compiler.*

Since the dummy class could be changed after compilation of the invocation, the compiler cannot optimize away the call by inlining the body.

Through this strategy we can state the following:

Theorem 3.2.5. *Is it possible to retrieve which statements were annotated with which Annotations from the .class data produced by the Java compiler*

This is true by setting the appropriate retention policy to the method annotation (recall Section 2.1), and letting its value to become accessible at run-time through the reflection. And again with the reflection, will be possible to have access at the bytecode of the method in search of the special sequence representing the markers.

Theorem 3.2.6. *The conversion of a legal* `@Java` *results in a legal Java 5 program.*

By removing the annotations from the code, which could be easily performed by visiting the syntax tree of the `@Java` program and skipping the annotations nodes corresponding to the grammar rule above, adding them to a method, and adding two method calls will produce a source that will be accepted by the grammar of Java 5, though it is not always true that a Java compiler will accept it as valid input.

Few special conditions could prevent the compilation to have success: statement annotations in fact cannot be applied to `return`, `throw`, `break` and `continue` statements, or block statements that unconditionally end with one of these, because in that case, the $K_e(k)$ end markers would be flagged as unreachable code by the Java compiler.

Other compilation strategies could result suitable for our purpose; for example we could modify a Java open source compiler to make it to add custom attributes to the methods of our interest. Following this route we could reach an even finer grained manipulation, and overcome the above presented limitations.

A few observations are in order. First, it should be noted that the bytecode sequence is easily identifiable, but not unique. A similar snippet, consisting of a push followed by a method call, could also be generated in the course of evaluating an expression like $k + o.M()$, where it would be followed by an `add` instruction. However, since we define only a version of the method M with a single argument k, cases like the above would be flagged as errors by the compiler, so the risk if erroneously identifying the bytecode fragments for the $K_b(k)$ and $K_e(k)$ sequences is minimal, and in practice confined to hand-crafted bytecode.

Second, the method calls $K_e(k)$ and $K_b(k)$ do not alter the functional semantics of the program, but they could alter its performance, and possibly adversely impact the meeting of non-functional requirements, since method invocation add a small performance penalty. However, while the Java compiler cannot inline or optimize away the method calls, an adaptive optimizing JIT compiler can, and usually will, so in practice even non-functional semantics is preserved.

Third, since method calls in Java can have side effects, and the compiler cannot be sure which body will be executed for non-final methods (as already discussed above), it is extremely unlikely that even an aggressively optimizing compiler will move code across $K_b(k)$ and $K_e(k)$ borders, so we can rely on the fact that the compiled bytecode contained between $K_b(k)$ and $K_e(k)$ markers is indeed the complete and only code for the annotated statement S.

Figure 3.2 shows an example of how an `@Java` code fragment is translated to Java by the `@Java` precompiler. As a side note, observe how the compilation scheme can be applied to the empty statement `;` (e.g., `@Pos;`), hence `@Java` annotations can be used to assign symbolic names to specific positions in the source code.

3.3 The preprocessor parser

The first step of `@Java` system consists in transforming a source written in `@Java` language into another source which conforms to the standard of Java 5. This

```
public void M()
{
  ...
  @A while (...)  {
    cnt++;
    @B(c=1) for (T i:  coll) {
      ...
    }
  }
}
```

```
import jcodebrick.Fragment;
import jcodebrick.MultiA;
...
@MultiA(
  ids={1,2},
  value={@A,@B(c=1)}
)
public void M()
{
   ...
   Fragment.begin(1);
   while (...)  {
     cnt++;
     Fragment.begin(2);
     for (T i:  coll) {
       ...
     }
     Fragment.end(2);
   }
   Fragment.end(1);
}
```

Figure 3.2: An example of the source-to-source translation performed by the `@Java` compiler. On the left, the source `@Java` code; on the right, the result of the translation.

precompilation passage is obtained through the use of a source-to-source parser.

3.3.1 Definitions

Let $\mathbb{C}$ be the domain of Java classes, $\mathbb{ML}$ the domain of a method set, $\mathbb{M}$ the domain of the methods, $\mathbb{SL}$ the domain of a statement list, $\mathbb{S}$ the domain of the statements, $\mathbb{AL}$ the domain of an annotation set, $\mathbb{A}$ the domain of the annotations, $\mathbb{IL}$ the domain of an id list, con id $\in \mathbb{N}$; let

$$met\colon \mathbb{C} \rightarrow \mathbb{ML}$$

be the function that given a generic class $c \in \mathbb{C}$ returns a list of all the methods $m_i \in \mathbb{M}$ declared in c, defined as

$$met(c) = \langle m_1, m_2, \ldots, m_n \rangle$$

Let

$$stat\colon \mathbb{M} \rightarrow \mathbb{SL}$$

be the function that given a generic method $m \in \mathbb{M}$ returns a list of all the statements $s_i \in \mathbb{S}$ declared in m, defined as

$$stat(m) = \langle s_1, s_2, \ldots, s_n \rangle$$

Let

$$isann\colon \mathbb{S} \to \mathbb{B}$$

be a boolean function that given a generic statement $s \in \mathbb{S}$ is defined as

$$isann(s) = \begin{cases} true & \text{if } s \text{ is an annotated statement} \\ false & \text{otherwise} \end{cases}$$

For simplicity, we give also an overloaded version of $isann$ that works on a arbitrary method $m \in \mathbb{M}$ instead that working on statements.

$$isann\colon \mathbb{M} \to \mathbb{B}$$

$$isann(m) = \begin{cases} true & \text{if } \exists\, s \in stat(m) \quad \text{s.t. } isann(s) \\ false & \text{otherwise} \end{cases}$$

The purpose of the parser is to convert a file with an extended version of Java source, say it S_e, into another file whose source is Java 5 compliant, say it S_s. We define the transformation between S_e into S_s as the operation π working on the class c defined in S_e:

$$\pi\colon \mathbb{C} \to \mathbb{C}$$
$$\pi(c) = c'$$

where if

$$met(c) = \langle m_1, \ldots, m_n \rangle$$

then the resulting c' is a class such that

$$met(c') = \langle m'_1, \ldots, m'_n \rangle \quad \text{s.t. } \forall m' \in met(c') \;\; \neg isann(m')$$
$$m'_i = \begin{cases} m_i & \text{if } \neg isann(m_i) \\ mod(m_i) & \text{if } isann(m_i) \end{cases}$$

The function

$$mod\colon \mathbb{M} \to \mathbb{M}$$

is the one that converts a `@Java` language method m in a Java 5 method m'.

$$mod(m) = m'$$

To define it we use two variables $al \in \mathbb{AL}$ and $idl \in \mathbb{IL}$ to list the annotations found in the given method and to list their unique id. Given $m \in \mathbb{M}$ such that

$$stat(m) = \langle s_1, \ldots, s_n \rangle$$
$$s_i = \begin{cases} \bar{s}_i & \text{if } \neg isann(s_i) \\ a_i \cdot \bar{s}_i, a_i \in \mathbb{A} & \text{otherwise} \end{cases}$$

then mod returns m' such that

$$m' = A(\vec{v}) \cdot m'' \quad \text{with } A \in \mathbb{A}$$
$$stat(m'') = \langle s'_1, \ldots, s'_n \rangle$$

$$s'_i = \begin{cases} \bar{s_i} & \text{if } \neg isann(s_i) \\ K_b(q) \cdot s_i \cdot K_e(q), & \text{otherwise} \end{cases}$$

with $q =$ unique id, and if $isann(s_i)$ then

$$idl = idl \cup q$$

$$al = al \cup a_i$$

Finally

$$\vec{v} = \langle idl, al \rangle$$

3.3.2 Implementations

To create the preprocessor parser, the parser generator JavaCC [1] has been used. Briefly, JavaCC uses a grammar file (with the extension ".jj") to generate several files which compiled originate a Java parser. Some of these files are standard and automatically generated if they are not found in the working directory; it is possible to modify the behavior of the generated parser by customizing some of them, and for the reason we will explain later, one of this modification has been found essential.

The grammar file of Java 5 is distributed by JavaCC; we alter the file both to extend the grammar to include `@Java` language, and to customize the class object representing the final Java parser.

Since JavaCC does not offer the possibility to build a parse-tree[2] of the parsed source, we choose an in-line approach: during the scanning, the parser we create, records the offsets, in terms of byte from the beginning of the source, of hot points. Such points are related to the first and the last character of the declaration of the annotation, the first and the last character of the annotated statement, and the first character of the declaration of the method.

Three classes have been declared with the parser:

1. Annotation: the representation of an annotation. It's a data structure used to store all the relevant offsets found while parsing the source.

2. JavaParser: the parser itself, autogenerated by JavaCC.

3. JavaParserCommon: a singleton with some utility methods and the vector of all the annotation found.

To resolve the correct byte offset we used the class `Token` offered by JavaCC. It represents the next token being parsed from the source stream, and let us to known the right offset just calling its methods `beginLine` and `beginColumn`. Unfortunately the latter suffers of some limitation: any *tab* character found in the stream is treated as an 8-characters wide column; that is why we customize the `Token` class adding the method `setTabSize`(int) which let to control the spacing of the *tab* character: as soon as `JavaParser` is created `setTabSize`(1) is called. Finally, through `Annotation.convertLineColumnToByteOffset`, a static methods we defined, we are able to resolve the exact starting byte offset of the given `Token`. The ending offset is resolved with the corresponding methods `endLine` and `endColumn`.

[2]This is achieved by other library included in JavaCC called *JJTree* and *JTB*

The modifications of the grammar

For our purpose there are only two modifications to the grammar of Java 5: in the rule *ClassOrInterfaceBodyDeclaration* we mark the beginning of the method we are going to parse: if any statement annotation will be found in the following method, then we already know where to make it lift to.

The other modification concerns the rule *Statement*, which becomes *Statement ::= Annotations Statement*. In this case we remember the first and the last offset of the token of the annotation, and the first and the last offset of the annotated statement.

Build time

When the source stream has reached the end, we are in the situation in which the parser object knows any relevant offset for our purpose, and therefore it is ready to emit the target source assigning the right *id*s to the lifted annotations and placing the right markers at the beginning and at the ending of the annotated statement. This is done by the method `Build` in the class `JavaParserCommon` :

```
destSource = '';
cursor = 0;
foreach annotated-method m found in the source:
  destSource += origSource.substr( cursor, an.method.begin )
  cursor = an.method.begin
  anList = getAnnotationListFromMethod( m )
  foreach an in anList
    an.id = new unique id
  outputSource += createMultiAnnotation( anList )

  do {
    nextAnB = nextAnBegin( anList, cursor )
    nextAnE = nextAnEnd( anList, cursor )

    if( nextAnB.bodyBegin < nextAnE.bodyEnd ) {
      destSource += origSoruce( cursor, nextAnB.bodyBegin )
      destSource += beginMarker( nextAnB.id )
      cursor = nextAnB.bodyBegin
    } else
    if( nextAnB.bodyEnd < nextAnE.bodyBegin ) {
      destSource += origSoruce( cursor, nextAnE.bodyEnd )
      destSource += endMarker( nextAnE.id )
      cursor = nextAnE.bodyEnd
    }
  } while( other annotation exists )

destSource += origSource( cursor, end )
```

Chapter 4

Bytecode Engineering through JDAsm

The `@Java` *system* is based upon two level of abstraction: one at low level provided by the library *JDAsm*, and one at high level provided by *JCodeBrick*.

JDAsm [15, 13] is similar in spirit to other code manipulation libraries like BCEL [2] or JavaAssist [6, 5], and it has been developed with the goal of offering ease of use, and at the same time good performances and completeness of operations. In fact, JDAsm offers the possibility to read already compiled class file or create new ones from scratch, and lets to modify any part of the class file structure, and furthermore offers some advanced feature like an inner bytecode verifier or instruction pattern searcher.

4.1 Structure

The structure of the library reflects the class file one [17]§4; for each entity in the class file, there exists a Java class that represents it and offers both read and write access to it. The main class is `DClass`; it acts as the container for classes like `DConstantPool` , `DMethod` and `DAttribute` (Figure 4.1).

Consider an already compiled Java class file s, and a `DClass` d used to create a new target class t; using JDAsm can be summarized in these steps:

1. Through the constructors of `DClass` one can either create an empty d or load s making d to reflect its structure.

2. The structure can be read and modified as needed using the appropriate class

3. Finally d is ready to be built to create the byte array representing t. Optionally t can be saved in a file or just loaded in the JVM and executed

The initialization is achieved by calling the constructor of `DClass` with no arguments (empty d) or with the class name (to fill d with an existing class file). In the latter case, the byte array of s is sequentially read, byte after byte, by each appropriate class to build the class tree representing s (Figure 4.2). In this tree, say T, also obtainable from scratch adding components to d, we

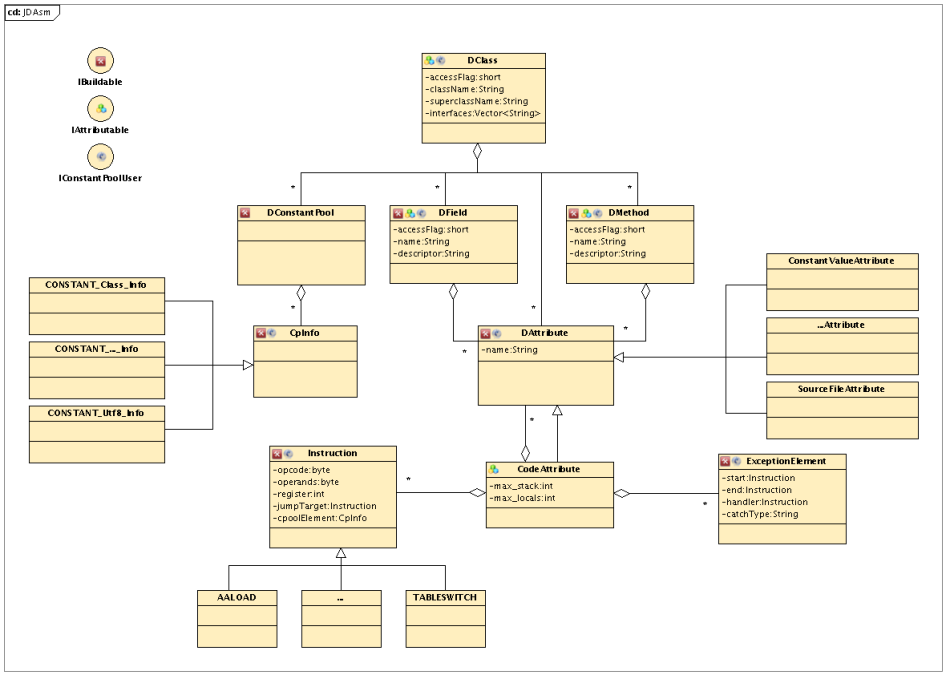

Figure 4.1: JDAsm class diagram

identify the root with the class `DClass`, and the intermediate nodes by such classes that represents methods, fields, constant pool and class-level attributes. Further on, a second level of attributes, either attached to a field or to a method, can be considered as childs of those classes in T. And so on, for instance, for instructions or attributes of the code attribute of one method.

The inspection and modification of a `DClass` is performed by using the respective class:

- `DConstantPool` allows reading and modify the constant pool of the class, even if this behavior can be totally managed automatically (Section 4.2)
- `DField` allows reading and modifing the fields of the class (Section 4.3)
- `DMethod` allows reading and modifing the methods of the class. In particular one can inspect, create or modify the code attribute of the method, and its bytecode (Section 4.3, 4.5)
- `DAttribute` is the super class of all the attribute known by the JVM. Through its interface one can inspect, create or modify an attribute. (Section 4.4)

4.2 Constant pool

The constant pool is an array of constants which takes place in the class file structure soon after the header bytes (Figure 4.2); it contains all the constants

```
ClassFile {
    u4 magic;
    u2 minor_version;
    u2 major_version;
    u2 constant_pool_count;
        cp_info constant_pool[constant_pool_count-1];
    u2 access_flags;
    u2 this_class;
    u2 super_class;
    u2 interfaces_count;
    u2 interfaces[interfaces_count];
    u2 fields_count;
        field_info fields[fields_count]  {
            u2 access_flags;
            u2 name_index;
            u2 descriptor_index;
            u2 attributes_count;
                attribute_info attributes[attributes_count] {
                    ...
                }
        }
    u2 methods_count;
        method_info methods[methods_count] {
            u2 access_flags;
            u2 name_index;
            u2 descriptor_index;
            u2 attributes_count;
                attribute_info attributes[attributes_count] {
                    ...
                    Code_attribute {
                        u2 attribute_name_index;
                        u4 attribute_length;
                        u2 max_stack;
                        u2 max_locals;
                        u4 code_length;
                        u1 code[code_length];
                        u2 exception_table_length;
                        {
                            u2 start_pc;
                            u2 end_pc;
                            u2  handler_pc;
                            u2  catch_type;
                        } exception_table[exception_table_length];
                        u2 attributes_count;
                            attribute_info attributes[attributes_count] {
                                ...
                            }
                    }
                }
        }
    u2 attributes_count;
        attribute_info attributes[attributes_count] {
            ...
        }
}
```

Figure 4.2: Java class file structure

used in the class, either numerical or literal. JDAsm defines eleven classes for managing every type of constant known by JVM:

- `CONSTANT_Class_Info`
- `CONSTANT_Double_Info`
- `CONSTANT_FieldRef_Info`
- `CONSTANT_Float_Info`
- `CONSTANT_Integer_Info`
- `CONSTANT_InterfaceMethodRef_Info`
- `CONSTANT_Long_Info`
- `CONSTANT_MethodRef_Info`
- `CONSTANT_NameAndType_Info`
- `CONSTANT_String_Info`
- `CONSTANT_Utf8_Info`

The class which collects them all and acts like their container is `DContantPool`. To the contrary of other bytecode engineering library [5, 2], JDAsm allows any entity of class file to survive in an independent context due to a *delayed constant pool linkage* mechanism: in fact the constant pool is automatically filled by JDAsm at *build time* by traversing the structure tree and asking to each entity met for its constant pool usage. Let p be a `DConstantPool` object, and let n an arbitrary node in the tree structure T (also a leaf). Consider these function:

$$\mathit{childs}(\mathit{tree}, \mathit{node}) = \langle n_1, \ldots, n_n \rangle \quad | \; n_i \text{ is child of } \mathit{node} \text{ in the given } \mathit{tree}$$

$$\mathit{child}(\mathit{tree}, \mathit{node}, k) = n_k \quad | \; n_k \text{ is the k-th child of } \mathit{node} \text{ in the given } \mathit{tree}$$

$$\mathit{constants}(\mathit{node}) = \langle c_1, \ldots, c_n \rangle \quad | \; c_i \text{ is the i-th constant that } \mathit{node} \text{ wants to link in } p$$

$$\mathit{ctype}(c) = \begin{cases} \text{Class_Info} \\ \text{Double_Info} \\ \text{FieldRef_Info} \\ \text{Float_Info} \\ \text{Integer_Info} \\ \text{InterfaceMethodRef_Info} \\ \text{Long_Info} \\ \text{MethodRef_Info} \\ \text{NameAndType_Info} \\ \text{String_Info} \\ \text{Utf8_Info} \end{cases}$$

The *delayed constant pool linkage* mechanism is implemented through the Java interface *IConstantPoolUser* which defines the following methods to be implemented for each node n which uses constants that need to be stored in p.

- `constantPoolUserChildSize()` $= |\mathit{childs}(T, n)|$
- `constantPoolUserChild(int k)` $= \mathit{child}(T, n, k)$

- `constantPoolValueSize()` $= |constants(n)|$
- `getConstantValue(int k)` $= constants(n)_k$
- `getConstantValueType(int k)` $= ctype(constants(n)_k)$
- `setConstantValueIndex(int k, short cpool_index)` $= n$ remembers that the constant k has index *cpool_index* in p, and it will use this value when needed in the final byte array representing the class file.

At *build time* p performs these steps:

```
function fillConstantPool( n ) {

  for i from 1 to n.constantPoolValueSize()
     val = n.getConstantValue(i)
     typ = n.getConstantValueType(i)
     if( p already contains <val,typ> )
        n.setConstantValueIndex(i, the index of <val,typ> )
     else
        idx = add new constant <val,typ> and the the relative index
        n.setConstantValueIndex(i, idx )

  for i from 1 to n.constantPoolUserChildSize()
     fillConstantPool( n.constantPoolUserChild( i ) )
}

fillConstantPool( root of T )
```

The class `DConstantPool` offers also a way to insert manually constants in the pool. In doing this, a boolean flag, will inform p to not discard such constants even if they have not been met in T

4.3 Fields and Methods

The classes `DFields` and `DMethod` are the ones designed to represent fields and methods of a Java class. They are very similar to each other in the contents and in the way they are going to write data into the final array of bytes. They offers the same methods for read and modify their own access flags, name and descriptor[1]. Both fields and methods are entities that are *IAttributable*, and for this they implement such interface which allows to read, add and remove attributes (Section 4.4) to and from the entity.

4.4 Attributes

JDAsm implements every attribute known to the JVM in a independent class with the same common parent class `DAttribute`. These classes are:

- `CodeAttribute`

[1]The descriptor is the only thing that significantly changes, as specified in [17]§4

- `ConstantValueAttribute`
- `CustomAttribute`
- `DeprecatedAttribute`
- `ExceptionsAttribute`
- `InnerClassesAttribute`
- `LineNumberTableAttribute`
- `LocalVariableTableAttribute`
- `LocalVariableTypeTableAttribute`
- `SignatureAttribute`
- `SourceDebugExtensionAttribute`
- `SourceFileAttribute`
- `StackMapTableAttribute`
- `SyntheticAttribute`

Each one of this class reflects the meaning of the same name attribute specified in [17], and offers the proper methods to inspect and modify its content. A special consideration is done for the `CustomAttribute` which allows the user to insert custom data to any *IAttributable* entity; furthermore, even if all `DAttribute` are *IConstantPoolUser*, the `CustomAttribute` due to its unknown structure and general purpose contents, can not benefit by the *delayed constant pool linkage* mechanism; such linkage, if needed, can however be done manually.

In the Java class file the only entities allowed to have attributes are Classes, Methods, Fields and Attributes; let $a \in \mathbb{A}$ be a generic attribute and let $AL = \langle a_1, \ldots, a_n \rangle$ be the attribute list of such entities. Even if an attribute can be very different from another one, they all are similar in the fact that they must have a identifying name; let $\nu \colon \mathbb{A} \to \mathit{String}$ be the function the returns the name of an attribute. JDAsm make the classes `DClass`, `DMethod`, `DFields` and `DAttribute` to implement the `IAttributable` interface which declares the following methods:

- `attributeCount()` $= |AL|$
- `addAttribute(`DAttribute attr`)` $: AL = AL \cup attr$
- `removeAttribute(`DAttribute attr`)` $: AL = AL \setminus attr$
- `removeAttribute(`int idx`)` $: AL = AL \setminus a_{idx}$
- `removeAttribute(`String name`)` $: AL = AL \setminus a_k \mid \nu(a_k) =$be name
- `getAttribute(`int idx`)\,`$= a_{idx} \in AL$
- `getAttribute(`String name`)` $= a_k \in AL \mid \nu(a_k) =$ name
- `getAttribtues()` $= AL$
- `hasAttribute(`String name`)` $= \begin{cases} \mathit{true} & \exists\, a_k \in AL \mid \nu(a_k) = \text{name} \\ \mathit{false} & \text{otherwise} \end{cases}$

4.5 Code Attribute

The `CodeAttribute` acts like the container of all the instructions of a method; it offers methods to read, append, insert and remove bytecode to and from it; it furthermore realizes the bytecode verifier to check the bytecode consistency, and automatically computes the *max stack* and the *max locals* values.

The instructions in Java can be grouped in five groups not disjoint according to their functionalities:

- Instructions that access the *Local Variable Array*; Every method of a Java class stores the local variables into the *local variable array*. We use $\mathcal{L} \subset \mathbb{N}$ to indicate it, treating a variable just as the index of its position in $\mathcal{L}$; since the variables are stored in $\mathcal{L}$ in increasing order starting from index 0, it will be $\mathcal{L} = [0, \ldots, n)$. The instructions included in this group are:

  `iload, lload, fload, dload, aload, iload_0, iload_1, iload_2, iload_3, lload_0, lload_1, lload_2, lload_3, fload_0, fload_1, fload_2, fload_3, dload_0, dload_1, dload_2, dload_3, aload_0, aload_1, aload_2, aload_3, istore, lstore, fstore, dstore, astore, istore_0, istore_1, istore_2, istore_3, lstore_0, lstore_1, lstore_2, lstore_3, fstore_0, fstore_1, fstore_2, fstore_3, dstore_0, dstore_1, dstore_2, dstore_3, astore_0, astore_1, astore_2, astore_3, iinc, ret`

- Instructions that use an integer value as *operand* of their operation; such operand is reported within the bytecode. The instructions included in this group are:

  `bipush sipush iinc newarray multianewarray`

- Instructions that use the *stack* for pushing or popping a value. We use $\mathcal{S}$ to refer to it. The instructions included in this group are:

  `aconst_null, iconst_m1, iconst_0, iconst_1, iconst_2, iconst_3, iconst_4, iconst_5, lconst_0, lconst_1, fconst_0, fconst_1, fconst_2, dconst_0, dconst_1, bipush, sipush, ldc, ldc_w, ldc2_w, iload, lload, fload, dload, aload, iload_0, iload_1, iload_2, iload_3, lload_0, lload_1, lload_2, lload_3, fload_0, fload_1, fload_2, fload_3, dload_0, dload_1, dload_2, dload_3, aload_0, aload_1, aload_2, aload_3, istore, lstore, fstore, dstore, astore, istore_0, istore_1, istore_2, istore_3, lstore_0, lstore_1, lstore_2, lstore_3, fstore_0, fstore_1, fstore_2, fstore_3, dstore_0, dstore_1, dstore_2, dstore_3, astore_0, astore_1, astore_2, astore_3, iinc, ret, iaload, laload, faload, daload, aaload, baload, caload, saload, iastore, lastore, fastore, dastore, aastore, bastore, castore, sastore, pop, pop2, dup, dup_x1, dup_x2, dup2, dup2_x1, dup2_x2, swap, iadd, ladd, fadd, dadd, isub, lsub, fsub, dsub, imul, lmul, fmul, dmul, idiv, ldiv, fdiv, ddiv, irem, lrem, frem, drem, ineg, lneg, fneg, dneg, ishl, lshl, ishr, lshr, iushr, lushr, iand, land, ior, lor, ixor, lxor, i2l, i2f, i2d, l2i, l2f, l2d, f2i, f2l, f2d, d2i, d2l, d2f,`

`i2b, i2c, i2s, lcmp, fcmpl, fcmpg, dcmpl, dcmpg, tableswitch, lookupswitch, arraylength, checkcast, instanceof, monitorenter, monitorexit, ifeq, ifne, iflt, ifge, ifgt, ifle, if_icmpeq, if_icmpne, if_icmplt, if_icmpge, if_icmpgt, if_icmple, if_acmpeq, if_acmpne, jsr, jsr_w, ireturn, lreturn, freturn, dreturn, areturn, athrow, getstatic, putstatic, getfield, putfield, invokevirtual, invokespecial, invokestatic, invokeinterface, new, newarray, anewarray, multianewarray, ifnull, ifnonnull`

- Instructions that use a *constant pool* index as operand of its operation. The instructions included in this group are:

  `anewarray, checkcast, getfield, getstatic, instanceof, ldc, ldc_w, ldc2_w, invokeinterface, invokespecial, invokevirtual, invokestatic, putfield, putstatic, multianewarray, new`

- Instructions that make the control flow of execution to branch (*jump instructions*). The instructions included in this group are:

  `ifeq, ifne, iflt, ifge, ifgt, ifle, if_icmpeq, if_icmpne, if_icmplt, if_icmpge, if_icmpgt, if_icmple, if_acmpeq, if_acmpne, goto, jsr, ret, goto_w, jsr_w, ifnull, ifnonnull`

We introduce now the domain of the Java instruction $\mathbb{I}$ which includes all the five groups seen above, the domain $\mathbb{T}$ of the types recognized by the JVM plus an empty type, and the domain of list of types $\mathbb{TL}$.

$$\mathbb{T} = \begin{cases} \texttt{void} \\ \texttt{int} \\ \texttt{long} \\ \texttt{float} \\ \texttt{double} \\ \texttt{reference}\ \text{(object)} \\ \texttt{address} \end{cases}$$

we define the follows:

$$ti\colon \mathbb{I} \to \mathbb{T}$$

as the function that returns the type of the operand that the specified instruction uses;

$$tl\colon \mathcal{L} \times \mathbb{N} \to \mathbb{T}$$

as the function that returns the type of the variable hold in the specified position of $\mathcal{L}$; and

$$ts\colon \mathcal{S} \to \mathbb{T}$$

as the function that returns the type of the variable which is at the top of $\mathcal{S}$. Moreover we define the following functions:

$$rv\colon \mathbb{I} \to \mathcal{P}(\mathcal{L})$$

$$wv\colon \mathbb{I} \to \mathcal{P}(\mathcal{L})$$

that given an instruction, return the register read (rv) and written (wv);

$$puv\colon \mathbb{I} \to \mathbb{TL}$$

$$pov\colon \mathbb{I} \to \mathbb{TL}$$

that given an instruction, return the types of the variables pushed (puv) and popped (pov) to and from the stack;

$$\sigma\colon \mathbb{I} \to \mathbb{N}$$

that given an instruction, returns its length in the bytecode

Besides having the instruction list of a method, the `CodeAttribute` holds the exception list of such instructions; we define an exception in the domain $\mathbb{E}$ as the tuple $exc = \langle ExcType, j, k, h\rangle$ with its type, the indexes j and k as delimiters of the scope of the *try* block and the index h of the first instruction of the *catch* block. This information is held in the field `exception_table`.

Let $i \in \mathbb{I}$ be the instance of a generic instruction, we use $IL = \langle i_1, \ldots, i_n\rangle \in \mathbb{IL}$ to indicate a generic instruction list; in the same way, let $e \in mathbbE$ be the instance of a generic exception, we use $EL = \langle e_1, \ldots, e_n\rangle$ to indicate a generic exception list.

We modeled the `CodeAttribute` of a generic method m as the tuple $\langle$ *IL, EL, max_stack, max_locals* $\rangle$, where $max_stack \in \mathbb{N}$ is the maximum dimension that $\mathcal{S}$ can reach during the execution of m, and the $max_locals \in \mathbb{N}$ is the dimension of $\mathcal{L}$ that m will use. To modify the instruction list IL and the exception list EL the following methods are available:

- `getCodeSize()` $= |IL|$
- `getByteCodeSize()` $= \sum_{k=1}^{n} \sigma(i_k)$
- `getInstruction(int idx)` $= i_{idx}$
- `getInstructionAtOffset(int offset)` $= i_k \mid \sum_{j=1}^{k} \sigma(i_j) = \text{offset}$
- `getFirstIstruction()` $= i_1$
- `getLastInstruction()` $= i_n$
- `addCode(Instruction instr, int idx)` :
  $IL = \langle i_1, \ldots, i_{idx}, \ldots, i_n\rangle \to IL = \langle i_1, \ldots, \text{instr}, i_{idx}, \ldots, i_n\rangle$
- `addCode(Instruction instr)` : $IL = IL \cup \text{instr}$
- `replaceInstruction(int idx, Instruction instr)` :
  $IL = \langle i_1, \ldots, i_{idx-1}, i_{idx}, \ldots, i_n\rangle \to IL = \langle i_1, \ldots, i_{idx-1}, instr, \ldots, i_n, \rangle$
- `removeCode( int from, int to )` :
  $IL = \langle i_1, \ldots, i_{from-1}, i_{from}, \ldots, i_{to}, \ldots, i_n\rangle \to IL = \langle i_1, \ldots, i_{from-1}, i_{to}, \ldots, i_n, \rangle$
- `getExceptionSize()` $= |EL|$
- `getException(int idx)` $= e_{idx} \in EL$

- `getExceptions()` $= EL$
- `addException( ExceptionElement ex)` : $EL = EL \cup ex$
- `removeException( int idx )` : $EL = EL \setminus e_{idx}$

Furthermore JDAsm uses a specific class for each kind of instruction i; each of these classes is child of a common abstract class called `Instruction` that implements the methods for retrieving all kind of information about i, like the number of operands, their size, the index of the register in $\mathcal{L}$ used, what type should be contained in the register before the execution and what type will be there after the execution, what is the stack request of types and the stack response after the execution, whatever is the target of the jump (when i is a jump of course) and other information about the use of the constant pool.

Few considerations are in order. Our implementation makes `CodeAttribute` have an *HashMap* for a fast indexing of the instruction through their offset. Anyway, this index must be built before being accessed, since it can be modified by any modification operation. Such re-indexing is necessary for the methods that uses σ. We make an instruction to remember its real offset in the bytecode, and make the `CodeAttribute` to have a global boolean flag that holds `true` if and only if all the offset held by the instructions are valid. We state that every operation that adds or removes or replaces instruction will set such flag to `false`. There are two main reasons for why we want to build the index of all the instruction by their offset, and one of this is for fast access through the hashmap. But this would not be sufficient. The main reason is that some instruction can modify its size according to the offset of other instruction. This is the case of jump instruction, where, if the jump offset is bigger than the one representable by 2 bytes, then some changes must occur. In particular, `goto` becomes `goto_w`, `jsr` becomes `jsr_w`, and all the other conditional jump instruction `if<cond>+K` become the sequence `if<cond>+2, goto+2, goto_w+K`. A similar problem occurs for the opcode `ldc` which load a constant from the pool into the stack. Such instruction is two bytes long, the first is the bytecode (0x12), and the second is the index into the constant pool (unsigned byte). Another version, the three bytes long `ldc_w`, let to index the constant through 2 bytes (unsigned short). Since the constant pool is automatically filled only during the building, it can happened that a `ldc` instruction must become a `ldc_w` if it was pointing at a constant whose index is greater than 256. JDAsm automatically makes all this changes on the fly in its re-indexing function if needed.

Once the `CodeAttribute` has been completed, and before the *build time*, JDAsm, by default[2], execute a bytecode verifier to validate the bytecode.

Definition 4.5.1. *A bytecode is said to be valid if the following properties are met:*

1. *Branches must be within the bounds of the code array for the method.*
2. *The targets of all control-flow instructions are each the start of an instruction: branches into the middle of an instruction are not allowed.*

[2]This feature can be turned off to improve performances

3. *No instruction can access or modify a local variable at an index greater than or equal to the number of local variables that its method indicates it allocates.*

4. *All references to the constant pool must be to an entry of the appropriate type*

5. *The code does not end in the middle of an instruction.*

6. *Execution cannot fall off the end of the code*

7. *For each exception handler, the starting and ending point of code protected by the handler must be at the beginning of an instruction or, in the case of the ending point, immediately past the end of the code. The starting point must be before the ending point. The exception handler code must start at a valid instruction.*

   Furthermore, at any given point in the program, no matter what code path is taken to reach that point, the following is true:

8. *The operand stack is always the same size and contains the same types of values*

9. *No local variable is accessed unless it is known to contain a value of an appropriate type*

10. *Methods are invoked with the appropriate arguments*

11. *Fields are assigned only using values of appropriate types*

12. *All opcodes have appropriate type arguments on the operand stack and in the local variable array*

Since JDAsm uses object instances for the instructions, where it is needed to reference a specific instruction, JDAsm uses the Java references, thus making property 7 already verified. Properties 2 and 5 are verified due to the atomic mechanism that JDAsm uses to create the bytecode array, in which any instruction knows how to represent itself in it. The re-indexing function as specified, makes the property 4 already verified too.

The implementation of a bytecode verifier into JDAsm has been necessary to automatically compute the `max_stack` and the `max_locals` fields of the `CodeAttribute`. For this purpose, the execution of the control flow is virtually done, in such a way the verifier can inform the user about the validity of the bytecode.

Let $LS_k = \langle t_1, \ldots, t_n \rangle$ be the state of $\mathcal{L}$ and $SS_k = \langle t_1, \ldots, t_n \rangle$ be the state of $\mathcal{S}$ with $t_i \in \mathbb{T}$ when the instruction k is executed. The following functions are defined for LS:

$$get(m) = t_m \text{ with } LS_k = \langle t_1, \ldots, t_m, \ldots, t_n \rangle$$

$$set(m, t) \colon LS_k = \langle t_1, \ldots, t_n \rangle \rightarrow LS_k = \langle t_1, \ldots, t_{m-1}, t, t_{m+1}, \ldots, t_n \rangle$$

which return the i-th type contained in $\mathcal{L}$ at instant k, and that makes the given position to have the specified type; and the following are defined over SS:

$$pop = SS_{k_n}, \text{ with } SS_k = \langle t_1, \ldots, t_n \rangle \rightarrow SS_k = \langle t_1, \ldots, t_{n-1} \rangle$$
$$push(t)\colon SS_k = \langle t_1, \ldots, t_n \rangle \rightarrow SS_k = \langle t_1, \ldots, t_n, t \rangle$$

To perform the verification, for each instruction i_j, we hold this information: $\langle LS_j, SS_j \rangle$. The verification process is only doable after that the `CodeAttribute` has been attached to a method m. It begins at the first instruction; let $p = \langle t_1, \ldots, t_n \rangle$ with $t_i \in \mathbb{T}$ be the list of types of the argument that m takes, then:

$$LS_1 = \begin{cases} \{\texttt{reference}\} \cup p & \text{if } m \text{ is not static} \\ p & \text{otherwise} \end{cases}$$
$$SS_1 = \emptyset$$

After reaching the generic instruction k, first the verifier controls that any read access to $\mathcal{L}$ and to $\mathcal{S}$ is consistent with LS_k and SS_k respectively, verifying properties 3, 9 and 12.

Definition 4.5.2. *Read access of instruction k to LS_k is consistent if:*

$$LS_k.get(rv(i_k)) = ti(i_k)$$

Definition 4.5.3. *Read access of instruction k to SS_k is consistent if:*

$$\forall\, t \in pov(i_k), SS_k.pop() = t$$

The next step consists in computing the new values for LS and SS:

$$LS_{k+1} = \begin{cases} LS_k & \text{if } wv(i_k) = \emptyset \\ \bar{LS} \quad | \; \bar{LS} = LS_k.set(wv(i_k), ti(i_k)) & \text{otherwise} \end{cases}$$

$$SS_{k+1} = \begin{cases} SS_k & \text{if } puv(i_k) = \emptyset \wedge pov(i_k) = \emptyset \\ \bar{SS} \mid \forall t \in pov(i_k), & \\ \qquad SS_k.pop(), \forall t \in puv(i_k), SS_k.push(t) & \text{otherwise} \end{cases}$$

Since code can contain jumps, the verifier may have already visited the instruction $k+1$ and can already have created LS_{k+1} and SS_{k+1}. If such values already exists, the verifier will check that they are identical to the ones just computed, verifying the property 8. If instruction $k+1$ is not yet visited instead, then LS_{k+1} and SS_{k+1} will be associated to it, and properties 1, 6, 10 and 11 are checked:

- If i_{k+1} is a jump instruction, it must have set a destination (1)
- Either i_{k+1} is not the last instruction of the method or it is a return instruction (6)
- If i_{k+1} is a method call instruction or a field access instruction, than SS_{k+1} must contains at its top the right types (10,11)

Once the whole code has been verified and accepted, all the data structures to automatically compute the value for *max_stack* and *max_locals* are already been created. In particular:

$$\text{max_locals} = max \text{ cardinality of } LS_k \;\forall\, k$$
$$\text{max_stack} = max \text{ cardinality of } SS_k \;\forall\, k$$

4.6 Advanced Features

JDAsm makes available same features aimed at simplifying the manipulation of code for the `@Java` *system*. One of these allows to define a search pattern for the bytecode of a method. Through the classes:

- `InstructionSearcher`
- `InstructionSearcherOption`

one can specify a sequence of instructions put in *or* or in *and*. Through the options is then possible to specify for every single instruction the *opcode*, the used registers of $\mathcal{L}$, the used types in $\mathcal{S}$, a specific value of the operand or a range of values, a specific constant pointed into the *c*onstant pool, and all the other parameters. Once configured, the function `InstructionSearcher.search()` returns a set of all the values of ranges of instructions that match the given pattern.

Another utility cares for the *super constructor wrapping*. JDAsm can change the super constructor invocation to call the constructor of a given class name. This is available only for such `CodeAttribute` which refers to a class constructor, and the wrap consists checking the old super class name of current class and in making the proper instruction `invokespecial` to call the constructor of a new given super class instead of invoking the constructor of of the old class. In this operation, the argument constructor are pushed onto the stack according to the method descriptor of the new super constructor. JCodeBrick will intensively use this feature as well as the next one (Section 5.4).

The *self subclassing* is another interesting feature used by `@Java` *system*. Let $c \in \mathbb{C}$ and $s \in \mathbb{C}$ be two arbitrary Java classes with c which extends s. Through *self subclassing* we create another class, say c', which is perfectly identical to c with the exception that c' extends c. In doing this operation, a *super constructor wrapping* is necessary (Section 5.4).

4.7 Build time

The *build time* is delayed and performed once for all when all the modification are done. Let c be the class that is going to be built, then the procedure begins with the crossover of the nodes of the tree T of c, according to the model defined by the interface `IConstantPoolUser` (Section 4.2), to build the constant pool exclusively based upon the constants defined by the nodes of T. As reaction of this first passage, every node gets back from the constant pool the numerical index that can be used to point to the proper declared constant. The construction of the dynamic sized byte array b occurs after. b is created to reflect the content of c as recognized by the JVM, i.e. the content is formatted to be a legal Java class file. Similarly at what happened for the model of tree defined by the interface *IConstantPoolUser*, the interface `IBuildable` declares the following methods to be implemented by all the nodes which represents a part of the class file:

1. `public int getBuildLength()` - This method must returns the amount of byte required in b that the IBuildable node needs to write its content.

The value must be inclusive of all the children in T, and this can be computed by making a depth recursion.

2. `public int build(byte[] tofill, int atindex )` - In pair with the method above, this one takes as argument the byte array b to fill, and the index at which the node is requested to begin the filling. As before, the method must make recursion for all its children in T. The return value r is the total amount of bytes inserted by it and its children, in such a way that the next call of this method will use as second argument the value `atindex`$+r$.

3. `public int build(OutputStream output)` - An alternatively method ask the node to fill b managed by an `OutputStream`, which automatically increase its size according to its needs. As all the other methods, this method must fill b recursively too.

At the moment, the build process is performed using only the third method, since the procedure base on methods 1-2 requires two passages, one to statically compute the size of the array, and one to fill it.

The filling of b starts with the class header; then the constant pool object `DConstantPool` writes down all its constants; then the class access flags and name, and the super class name follow; then an iteration over all the declared interface is done; it follows the insertion of all the fields with their attributes, all the methods with their attributes, and finally all the attributes of c are written down. The resulting byte array can be either written into a file to create a Java class file, or can be loaded into the JVM with a specific method of JDAsm `toClass()`.

4.8 Benchmarks

Given that one of the major advantages of our proposal over previous research is the ability to perform code manipulation at runtime, we are particularly concerned about the performances.

We have compared the execution times of typical `@Java` operations using different libraries for bytecode engineering. In particularly, JDAsm performances have been compared to that of BCEL [2] and JavaAssist [6, 7], using the latter both at source level and at bytecode level. In particular, we have measured the performances of the three libraries in the synthesis of a new Java class (as in our build operation), containing a single "Hello world" method.

The experimental results, obtained by averaging 20 runs of the equivalent generating code for the three libraries are shown in Figure 4.3. As can be seen, JDAsm is substantially faster than both BCEL and JavaAssist in source mode, and offers performances comparable (and slightly better) with those of JavaAssist used in bytecode mode, but with the advantage of being able to compose the method symbolically, rather than having to handle each individual bytecode instruction.

Library	*Time*
BCEL	172ms
JavaAssist (source level)	188ms
JavaAssist (bytecode level)	78ms
JDAsm	62ms

Figure 4.3: Execution times for the class synthesis benchmark.

Chapter 5

Manipulating annotated code: JCodeBrick

As we have seen in section 3, the statement annotations introduced by `@Java` can be used in three capacities:

- to express metadata about program fragment, serving all the needs we introduced in Section 1 (but with a finer granularity, so that metadata can be more precisely attached to code w.r.t. the standard model of Java 5);
- to assign symbolic names to specific positions in the source code, with a single-statement granularity; such symbolic references will be available also at runtime, in executable code.
- to assign symbolic names to code fragments, both in source and corresponding bytecode, and again available at runtime.

We will not discuss in this report applications of the first role that statement annotations can serve, focusing instead of using the other two roles for *dynamic bytecode manipulation*.

In fact, given the availability at runtime of a system of symbolic names for places and fragments, established in the source code (or even programmatically, in more contrived cases), and coupling that with the dynamic class loading system provided by the JVM, it becomes possible to insert, delete or move around parts of the program, and immediately execute the resulting code.

In the following sections, we will describe the operations offered by the library, together with the formal definition and our implementation of them.

5.1 Notation and definitions

Recalling the notation introduced in Section 4, and especially in the sub section 4.5, we extend it now to treat the methods and their bytecode. Let $\mathbb{M}_C$ be the set of all the methods of a Java class C, and let $i \in \mathbb{I}$ be the instance of a generic instruction, we use $IL = \langle i_1, \ldots, i_n \rangle \in \mathbb{IL}$ to indicate a generic instruction list (either the body of a method or just a part of it). Then we define the following:

$$\mu \colon \mathbb{M}_C \to \mathbb{IL}$$

as the function that given a method $m \in \mathbb{M}_C$ returns all its bytecode as a list of instructions; with a slight abuse of notation we will write $IL \subseteq m$ to indicate that IL is a sublist of $\mu(m)$. We use the function ι to retrieve the index of an instruction i in an instruction list:

$$\iota \colon \mathbb{IL} \times \mathbb{I} \to \mathbb{N}$$

to simplify the notation, we will overload ι as follows:

$$\iota \colon \mathbb{M}_C \times \mathbb{I} \to \mathbb{N}$$

$$\iota(m, i) = \iota(\mu(m), i)$$

The set of local variables referred to by an instruction or an instruction list (again, overloading the notation for simplicity) is defined as

$$loc(i) = rv(i) \cup wv(i)$$

$$loc(IL) = \bigcup_{i \in IL} loc(i)$$

Let α be a statement annotation inserted in the source code to mark a statement (typically a block statement) inside a method $m \in \mathbb{M}_C$. Then we define a *Fragment* f as the section of bytecode of m identified by the triple $r = \langle id, \alpha, m \rangle$, where the id is the unique identifier generated by the pre-compilation parser.

Definition 5.1.1. *A fragment is the smallest part of code that the user can manipulate by moving it and deleting it.*

It is defined as:

$$f = \langle i_b, i_e, r \rangle \text{ where } i_b \in r.m,\ i_e \in r.m,\ b < e$$

Lemma 5.1.2. *A fragment is delimited by two markers called starting marker, immediately preceding i_b, and ending marker, immediately following i_e.*

We call the starting marker K_b^f and the ending marker K_e^f. Each marker is a two-instruction sequence, $K_{b_1}^f$ and $K_{b_2}^f$, $K_{e_1}^f$ and $K_{e_2}^f$, which are the result of compiling the marker method calls inserted by the `@Java` compiler in place of a statement annotation. They include:

- An instruction $K_{b_1}^f = K_{e_1}^f$ to push onto the stack the value of $f.r.id$
- A static call to an empty method, one for any $K_{b_2}^f$ and another one for any $K_{e_2}^f$

Between the markers, f includes $l >= 0$ inner instructions, and we use this function to get them:

$$\nu(f) = IL$$

Thus, given a method $m \in \mathbb{M}_C$ of n instructions, and a fragment f of length l in m, we will have

$$\mu(m) = \langle i_1, \ldots, K_{b_1}^f, K_{b_2}^f, i_j, \ldots, i_{j+l}, K_{e_1}^f, K_{e_2}^f, \ldots, i_n \rangle$$

Definition 5.1.3. *A fragment is* valid *if it does not contain any jump instruction targeting an instruction outside of the fragment, with the exception that a jump immediately after the end of the fragment (i.e., to the first instruction following the last instruction in f) is considered valid.*

This condition excludes as valid fragments any part of code which contains a `break` or `continue` instruction which would continue the execution to locations not included in the fragment, and, depending on the compilation scheme used by the Java compiler, certain statements with `return` or `throw` clauses embedded in an outer `try-catch-finally` statement (in all these cases, the compiled code would include a jump to the code for the `finally` clause). In the following, we concern ourselves only with valid fragments.

Lemma 5.1.4. *Multiple fragments in a method never overlay each other and are always correctly nested, i.e. they are either disjoint, or one is entirely contained in the other.*

This is guaranteed by the grammar of `@Java`; for instance, given two fragments f' and f'' (appearing in this order) in the same method m, their markers K'_s, K'_e, K''_s,K''_e, and the index in the $IL \subseteq m$ of such markers, $a = \iota(m, K'_s)$, $b = \iota(m, K'_e)$, $c = \iota(m, K''_s)$, $d = \iota(m, K''_e)$, then either $a < b < c < d$ or $a < c < d < b$.

Lemma 5.1.5. *Given b as the bytecode of a valid fragment f, it always exists at least one branch in the control flow of b that allow the execution to fall off the end of b.*

This is in part guaranteed by the bytecode verifier at compilation time. Since we add the K_e at the end of f, the verifier will complain if it found K_e to be unreachable code. As already discussed in Section 3.2, this behavior implies the impossibility to have fragments with a `return` instruction without having jump instruction before it that branches the execution flow.

Lemma 5.1.6. *A fragment can have zero or more* `return instruction`*, but at most only of one type.*

This is fully guaranteed by the bytecode verifier and by the Java compiler.

5.2 Basic implementation

The JCodeBrick library declares `CbClass` as the main class for representing the Java class that is going to be modified. The constructor takes as argument a reference to a Java class, in such a way it can be able to load its structure as a `DClass` class. As soon as this structure is loaded, a cycle over all the declared methods is performed in search of special annotated method: the annotation looked at is the *@MultiAnnotation* created by the parser as defined in Section 3.3.

The class `Fragment` is the one that represents a fragment as defined before; for each *@MultiAnnotation* found, a set of `Fragment` is stored, and every one will hold information about the unique index that link such fragment with the portion of code through the markers. For a fast indexing, `CbClass` uses different *hashmaps* to access a fragment directly by its name[1] and/or by its method.

[1] with a fragment name we intend the annotation name

Once the class is loaded, and the fragments references are too, the user is ready to declare its operation over them; a lazy evaluation strategy will apply the queued operations of insertion and deletion in order to generate the modified class.

For such operations, one class `BrickOperaton` is created for any insertion or deletion declared and appended to a vector owned by the parent `CbClass`. The constructor of `BrickOperation` differentiates for the type of operation, and generally includes all the references needed to perform it, such as the involved fragments, the insert position and all the possible free variables with not free variables mapping (Section 5.3.2), and the class from which the deletion must be applied (Section 5.3.3).

5.3 Operations

We define four operations over fragments:

- op_{src}, to *search* and retrieve fragments;
- op_{ins}, to *insert* a fragment at the start or end of another fragment;
- op_{del}, to *delete* a fragment a fragment from the code in which it appears;
- op_{xtr}, to *extrude* a fragment, and execute it outside its context.

In the following, we provide a full formal definition for the operations above.

5.3.1 Search

Through the search operations the user is able to retrieve and get a reference to the fragments declared in a Class C. The operation is offered in several overloaded forms, allowing searches according to different criteria. Remembering that $r = \langle id, \alpha, m \rangle$, then we have a lot of overloaded operations, in which almost all forms take a class or a single method as argument, and then more arguments to specify which annotated fragments in the class should be retrieved.

The first one is the simplest: we retrieve the fragment with the given unique id. Anyway, it should be noted that for the programmer there is not a priori knowledge about the relation between a declared fragment with its future id (which is assigned automatically by the parser in the precompilation time).

$$op_{src} \colon \mathbb{C} \times \mathbb{N} \to F$$

defined as:

$$op(c, id) = \bar{f} \quad | \; \bar{f}.r.id = id, \bar{f}.r.m \in met(c)$$

The implementation refers to a method named `getFragmentById(int id)` of the class `CbClass`, which simply cycles over all the fragments to test which particular fragment has the specified id, a returning it over a match.

Other methods let the user to specify one single fragment mode in detail, i.e. by giving the annotation name and/or its method. Since this pair of values refers to a set of result, the index in this set is also requested, or intended to be zero (the first occurrence) if omitted:

$$op_{src}\colon \mathbb{A} \times \mathbb{M} \times \mathbb{N} \to \mathbb{F}$$
$$op(\alpha, m, k) = f_k \in \langle f_1, \ldots, f_n \rangle \mid f_i.r.\alpha = \alpha, f_i.r.m = m$$

$$op_{src}\colon \mathbb{A} \times \mathbb{N} \to \mathbb{F}$$
$$op(\alpha, k) = f_k \in \langle f_1, \ldots, f_n \rangle \mid f_i.r.\alpha = \alpha$$

$$op_{src}\colon \mathbb{A} \times \mathbb{M} \times \mathbb{N} \to \mathbb{F}$$
$$op(\alpha, m) = f_1 \in \langle f_1, \ldots, f_n \rangle \mid f_i.r.\alpha = \alpha, f_i.r.m = m$$

As corresponding `CbClass` defines

$$\texttt{getFragment}(\texttt{Stringname}, \texttt{Methodmethod}, \texttt{intn})$$

which cycle over all the fragments in search of the n-th item that matches, and the following:

- `getFragment(String name,int n) {return this.getFragment(name,null, n);}`
- `getFragment(String name) {return this.getFragment(name,0);}`

More generalized methods of retrieving let the user to get all the set of fragment matching the given argument. In this case we have:

$$op_{src}\colon \mathbb{A} \times \mathbb{M} \to \langle f_1, \ldots, f_n \rangle$$
$$op(\alpha, m) = \langle f_1, \ldots, f_n \rangle \mid f_i.r.\alpha = \alpha, f_i.r.m = m$$

$$op_{src}\colon \mathbb{A} \to \langle f_1, \ldots, f_n \rangle$$
$$op(\alpha, m) = \langle f_1, \ldots, f_n \rangle \mid f_i.r.\alpha = \alpha$$

$$op_{src}\colon \mathbb{M} \to \langle f_1, \ldots, f_n \rangle$$
$$op(\alpha, m) = \langle f_1, \ldots, f_n \rangle \mid f_i.r.m = m$$

The `CbClass` offers the following:

- `Vector<Fragment> getFragmentsVector(String name, Method method)` that uses the hashmap to easily return all the specified fragments.
- `Fragment[] getFragments( String name, Method method )` as above, except that it returns the fragment as an array.
- `Vector<Fragment> getFragmentsVector(String name)`
  `{return this.fragments_by_name.get(name); }`
- `Fragment[] getFragments(Annotation annotation, Method method)` that transforms the vector returned by the above function into an array and returns it.
- `Fragment[] getFragments(Annotation annotation)`
  `{return this.getFragments(annotation,null);}`
- `Fragment[] getFragments(String name)`
  `{return this.getFragments(name,null);}`

Finally, also functions that return all the fragment defined for a class are given:

$$op_{src}\colon \mathbb{C} \to \langle f_1, \ldots, f_n \rangle$$
$$op(c) = \langle f_1, \ldots, f_n \rangle \mid f_i.r.m \in met(c)$$

implemented in `CbClass` in:

- `Vector<Fragment> getFragmentsVector() {return this.fragments;}`
- `Fragment[] getFragments()` that converts the resulting vector of the method above into an array and returns it.

5.3.2 Insertion

Through the insertion operation the user can inject the bytecode of a source fragment f_s into a specific position in a method m of a class C. The destination position is related to a destination fragment f_d, and it can be one of *before_start*, *after_start*, *before_end*, *after_end*, which indicate, respectively, that f_s is to be inserted before the starting marker of f_d, after the starting marker of f_d, before the ending marker of f_d, and after the ending marker of f_d. The possibility of inserting code inside and outside the destination markers has consequences in concatenated operations that involve the destination fragment f_d more than once. For instance, given four fragments A, B, T, Z, by inserting A into T in position $before_start$, then inserting B into T in position $after_start$, and finally inserting T into Z, the code of B will be carried into Z through T, but not the code of A, which has been inserted outside the markers of T.

Given a fragment f, let $IL = \nu(f)$ be its instruction list. IL can use and modify local variables, so we need to consider the source method $m_s = f_s.r.m$, the destination method $m_d = f_d.r.m$ and their respective local variables. Any variable has its own scope; the following function:

$$scope(IL, v) = (j, k) \quad | \, v \in \mathbb{V} \quad j, k \in \mathbb{N}$$

is defined to return the pair j and k as the boundary index of the instructions in IL where the scope of v is valid (this information is provided by the Java compiler among the metadata carried with Java classes, in the table `LocalVariableTableAttribute`).

Given an instructions list IL, a *free variable* $v' \in loc(IL)$ is a variable whose scope is defined outside IL:

$$v' \in loc(\nu(f_s)) \quad | \, (j, k) = scope(\mu(m_s), v'), j < \iota(m, k^f_{s_1}) \wedge k > \iota(m, k^f_{s_2})$$

When we want to deal with insertion of a source fragment f_s that uses the free variable v', we need the user to specify a valid mapping among all the free variables in f_s with the variables in m_d whose scope covers the insertion point. We use a function to get the subset of $loc(IL)$ of all the free variables in IL:

$$floc(IL) = \{v \in loc(IL) \mid v \text{ is free}\}$$

Let $V_m = \{v_{s_1} \rightarrow v_{d_1}, \ldots, v_{s_n} \rightarrow v_{d_n}\}$ be a user defined mapping that associates to any free variable v_{s_i} of f_s a valid variable v_{d_i} of f_d (valid variables are those that are in-scope at the insertion point and have the appropriate type; the mapping is specified by name in the implementation for ease of use, but here we will only refer to the variable indexes), then we define the operation of *insertion* as the function that given a source fragment f_s, a destination fragment f_d, a position p and a mapping V_m, inserts the new fragment in the same method m_d of f_d, and returns m'_d to indicate that the instruction list IL of m_d has been modified.

$$op_{ins} \colon \mathbb{F} \times \mathbb{F} \times P \times \mathbb{V}_m \rightarrow \mathbb{M}_C$$

Other aspects have to be considered in addition to free variable mapping in implementing this operation. In particular, there are cases where the insertion cannot be performed in a type-safe way. If the source bytecode contains a

return instruction, we have to check that the return type is compatible with the return type of the destination method. To model this, we introduce the following functions:

$$ret\colon \mathbb{I} \rightarrow Type$$

$$ret\colon \mathbb{M}_C \rightarrow Type$$

defined as:

$$ret(i) = \begin{cases} t & \text{if } i \text{ is the } \texttt{RETURN} \text{ instruction for type } t \\ \emptyset & \text{otherwise} \end{cases}$$

$$ret(m) = \bigcup_{i \in \mu(m)} ret(i)$$

with $|ret(m)| \leq 1$, that is, since we are working on an already loaded class, guaranteed by the bytecode verifier.

If $\exists i \in \nu(f_s)$ such that $ret(i) \neq \emptyset \wedge ret(i) \neq ret(m_d)$ (i.e., a return instruction whose type differs from that of the method it is being injected into), then the fragment is not compatible with the method and the insert operation fails returning an error. As we have already seen, free variables are renumbered through the user-supplied mapping V_m; all other variables need to have their index shifted so that they do not conflict with the local variables of f_d. Since all variables in m_d use their own index into the local variable array $\mathcal{L}$ and the variables $V \in loc(\nu(f_s))$ with $V \notin floc(\nu(f_s))$ might use the same indexes, to avoid the risk of overlaying the two sets, we compute the higher index h used by m_d and add h to any index used in V.

Furthermore we consider the possibility that $IL = \nu(f_d)$ is included inside a *try-catch* block. Since we cannot determine by looking at IL alone if its instructions can raise an exception, we conservatively assume that they can, and surround the inserted code with a brand new *try-catch* block that will catch any exception, and handle it by throwing a new `RuntimeException` (having the original exception in its `cause` field) in the catch block.

It should be noted that our choice is not the only possible one. Another possibility would be to update the signature of m_d to accommodate for the additional exceptions which could be raised by the inserted fragment. This choice however would violate the API contract between the method and its callers, and make seamless replacement of code difficult, while our approach, based on the unchecked `RuntimeException`, does not suffer from this difficulty.

We define an exception as the tuple $exc = \langle ExcType, j, k, h \rangle$ with its type, the indexes j and k as delimiters of the scope of the *try* block and the index h of the first instruction of the *catch* block. This information is held in the Java class file into the `exception_table` field of the `Code_attribute` for the method. We will indicate with $et(m)$ the exception table of a method m, according to its `Code_attribute`, containing metadata about the type and indexes of all try/catch blocks, and with $te(m)$ the set of $ExcType$s thrown by a method m, according to its signature.

The set of exception types which might be thrown by a fragment $f = \langle i_b, i_e, \langle id, \alpha, m \rangle \rangle$ is defined as follows:

$$tc(f) = te(m) \cup \{ET \mid \langle ET, j, k, h \rangle \in et(m) \wedge j \leq b \wedge e \leq k\}$$

```
        goto end
catch: new jcodebrick/FragmentRTE
        dup_x1
        swap
        invokespecial jcodebrick/FragmentRTE."<init>":(Ljava/lang/Throwable;)V
        athrow
end:
```

Figure 5.1: The IL code for the `catch` blocks appended at the end of fragments for the insertion operation.

The code that will be inserted at the end of f_s in case we have to add the *catch* block will be that shown in Figure 5.1; we will denote that instruction list with IL_{RT}.

With the above definitions, we say that an insertion operation $op_{ins}(f_s, f_d, p, V_m)$ is *valid* if the following conditions are met:

1. $floc(f_s) = domain(V_m)$;
2. $\forall v \in range(V_m), scope(\nu(f_d), v) = (j, k) \implies j \leq ip(f_d, p) \leq k$;
3. $\forall (v \to w) \in V_m, type(v) = type(w)$;
4. $\forall i \in \nu(f_s), ret(i) = \emptyset \vee ret(i) = ret(f_d.m)$.

where $domain(m)$ and $range(m)$ are, respectively, the set of keys and of values in a mapping m; $ip(f, p)$ returns the index of the insertion point for a fragment f with a position p (it will be the index of the begin or end marker of f, depending on p), and $type(v)$ is the VM type of a variable v.

An invalid insertion operation results in an `InvalidBuildException` being thrown at build time, and the operation is aborted. If the operation is valid, the insertion proceeds as follows.

First, the local variables in the IL associated with the source fragments are renumbered, to avoid clashes with the variable already used in the destination fragment. Then, free variables are mapped according to V_m, and finally a *try-catch* block is added, if needed, to capture and turn into `RuntimeException` all exception thrown by the source fragment which are not handled in the destination method. Formally, this process is described in the following.

Let $h = \max_{v \in loc(\mu(f_d.m))}(v)$ be the index of the highest-numbered local variable in the destination method. Then a new instruction list $IL' = \langle i'_1, \ldots, i'_n \rangle \cdot \theta$ is obtained by copying and modifying the instruction list of the source fragment $IL = \langle i_1, \ldots, i_n \rangle$ in such a way that

$$i'_j = \begin{cases} i_j \left[{v+h}/{v} \right] & \text{if } v \in loc(i_j) \text{ and } v \text{ is not free} \\ i_j \left[{w}/{v} \right] & \text{if } v \in loc(i_j) \text{ and } (v \to w) \in V_m \\ i_j & \text{otherwise} \end{cases}$$

and

$$\theta = \begin{cases} IL_{RT} & \text{if } tc(f_s) \setminus tc(f_d) \neq \emptyset \\ \langle\rangle & \text{otherwise} \end{cases}$$

The resulting method m'_d will be such that its instruction list will be updated to insert IL' at the location specified by p and f_d; its exception table is updated

to include the possible addition of *try-catch* blocks for the inserted fragment; and its `LocalVariableTableAttribute` is updated to include the new local variables carried into the method by the inserted fragment. In all other respects (e.g., signature, `throws` clause, debug attributes, etc.) m'_d is identical to m_d.

The definition for the case when $p = \mathit{before_start}$ follows, where if

$$\mu(m_d) = \alpha \cdot \langle K^{f_d}_{b_1}, K^{f_d}_{b_2} \rangle \cdot \beta \cdot \langle K^{f_d}_{e_1}, K^{f_d}_{e_2} \rangle \cdot \gamma$$

then the result of the insertion is m'_d such that

$$\mu(m'_d) = \alpha \cdot \langle K'^{f_s}_{b_1}, K^{f_s}_{b_2} \rangle \cdot IL' \cdot \langle K'^{f_s}_{e_1}, K^{f_s}_{e_2} \rangle \cdot \langle K^{f_d}_{b_1}, K^{f_d}_{b_2} \rangle \cdot \beta \cdot \langle K^{f_d}_{e_1}, K^{f_d}_{e_2} \rangle \cdot \gamma$$

and

$$et(m'_d) = et(m_d) \cup E$$

where $K'^{f_s}_{b_1}, K'^{f_s}_{e_1}$ are similar to $K^{f_s}_{b_1}, K^{f_s}_{e_1}$, respectively, except in that they have a fresh unique *id* (a larger id may require a different opcode), and

$$\begin{aligned} E \quad = \{ \quad & (ET, j, k, k) \mid ET \in tc(f_s) \setminus tc(f_d), \\ & j \text{ is the initial index of } IL' \text{ in } \mu(m'_d), \\ & k \text{ is the index of the } \texttt{catch} \text{ label from } \theta \text{ in } \mu(m'_d) \qquad \} \end{aligned}$$

Similarly, we define the left cases. When $p = \mathit{after_start}$ we have that the result of the insertion is m'_d such that

$$\mu(m'_d) = \alpha \cdot \langle K^{f_d}_{b_1}, K^{f_d}_{b_2} \rangle \cdot \langle K'^{f_s}_{b_1}, K^{f_s}_{b_2} \rangle \cdot IL' \cdot \langle K'^{f_s}_{e_1}, K^{f_s}_{e_2} \rangle \cdot \beta \cdot \langle K^{f_d}_{e_1}, K^{f_d}_{e_2} \rangle \cdot \gamma$$

with $p = \mathit{before_end}$:

$$\mu(m'_d) = \alpha \cdot \langle K^{f_d}_{b_1}, K^{f_d}_{b_2} \rangle \cdot \beta \cdot \langle K'^{f_s}_{b_1}, K^{f_s}_{b_2} \rangle \cdot IL' \cdot \langle K'^{f_s}_{e_1}, K^{f_s}_{e_2} \rangle \cdot \langle K^{f_d}_{e_1}, K^{f_d}_{e_2} \rangle \cdot \gamma$$

with $p = \mathit{after_end}$:

$$\mu(m'_d) = \alpha \cdot \langle K^{f_d}_{b_1}, K^{f_d}_{b_2} \rangle \cdot \beta \cdot \langle K^{f_d}_{e_1}, K^{f_d}_{e_2} \rangle \cdot \langle K'^{f_s}_{b_1}, K^{f_s}_{b_2} \rangle \cdot IL' \cdot \langle K'^{f_s}_{e_1}, K^{f_s}_{e_2} \rangle \cdot \gamma$$

Again, in all the three cases above we have:

$$et(m'_d) = et(m_d) \cup E$$

As a final technicality, the `max_stack`, `max_locals`, `code_length`, `code`, `exception_table_length`, `exception_table`, `attribute_info` of the `Code_attribute` for m'_d are updated as needed, and a copy of the Annotation for f_s with the new fresh *id* used in $K'^{f_s}_{b_1}$ and $K'^{f_s}_{e_1}$ is added to the annotations for m'_d (Section 5.4).

Early implementation

The insertion is allowed through the class `Fragment` with the method:

```
int insertFragment( InsertionPosition where, Fragment what, String[] variableMapping )
```

The method is call on an instance of the destination fragment and takes as argument the insertion point, the source fragment, and the variables mapping defined as:

$v[i] \rightarrow v[i+1]$ with i even, $v[i]$ a free variable, and $v[i+1]$ a not free variable

The implementation strategy schedules the insertion operation to be executed at build time once for all at the end (Section 5.4). Therefore at declaration time the system just enqueue a new `BrickOperation` into the parent `CbClass` operation vector: this parent is intended to be the class that the declares the destination fragment. Even if the operation is not soon executed, the new fresh unique id is soon assigned computed as the higher known id used by the library plus 1; such id is returned at the end of the method.

5.3.3 Deletion

To delete a fragment f from a method m means to re-emit the bytecode of m without the instructions delimited by K_b^f and K_e^f. We define three types of deletion: *delete_without_markers*, *delete_with_markers*, *delete_only_markers* where respectively the bytecode included by f is deleted but the markers are not, the bytecode is deleted and the markers are too, and only the markers are deleted while the bytecode included in f is left untouched.

The operation of deletion is defined as the function that, given a method m, a fragment f in m, and a type t of deletion, returns m' which is identical to m except that part or all of $\mu(m)$ is not present in it anymore:

$$op_{del} \colon \mathbb{M}_C \times \mathbb{F} \times T \rightarrow \mathbb{M}_{C'}$$

Since the `@Java` compiler inserts fragment markers only at the begin and at the end of a statement, we are guaranteed that a deletion cannot overlap a *try-catch* block nor the scope for a variable, and that the corresponding fragment cannot contain an instruction which is a target from an external jump instruction. Furthermore, since the Java compiler always adds an explicit `return` instruction at the end of a void method, we are assured that the return type from a method's code cannot be changed by a deletion. Hence, a deletion does not need any structural change to a method.

We define the three types of operation according to t considering a source method like the following:

$$\mu(m) = \alpha \cdot \langle K_{b_1}^f, K_{b_2}^f \rangle \cdot \beta \cdot \langle K_{e_1}^f, K_{e_2}^f \rangle \cdot \gamma$$

When $t =$ *delete_without_markers*, then the result of the deletion of f is m'_d such that

$$\mu(m') = \alpha \cdot \langle K_{b_1}^f, K_{b_2}^f, K_{e_1}^f, K_{e_2}^f \rangle \cdot \gamma$$

for $t =$ *delete_with_markers*:

$$\mu(m') = \alpha \cdot \gamma$$

and for $t =$ *delete_only_markers*:

$$\mu(m') = \alpha \cdot \beta \cdot \gamma$$

We also need to remove from the exception table all the *try-catch* blocks which were entirely contained in the removed fragment f, and possibly compact the local variable table by removing all variables whose scope was entirely within f. Again, addresses in $\mu(m')$ are renumbered, and the various `Code_attribute` fields are recomputed as needed (Section 5.4).

Early implementation

As seen for the insertion operation, the implementation strategy makes the deletion to be scheduled to be executed at build time. At the moment of user declaration a new instance of `BrickOperation` is appended to the operation vector of the parent `CbClass`.

The suitable methods must be called on an instance of the class `Fragment` that represents the fragment one want to delete. Two different methods are implemented:

```
void delete(boolean withMarker)
```

that let the user to define a deletion operation with $t = \textit{delete_with_markers}$ or $t = \textit{delete_without_markers}$, and

```
void deleteMarkers()
```

for the operation with $t = \textit{delete_only_markers}$. In the `BrickOperation` the source class and the fragment are stored for a later reference (Section 5.4).

5.3.4 Extrusion

The extrusion operation makes it possible to execute the code of a fragment as a self-sufficient method, outside of its original context. The result of the operation is a new class, containing a single static method `exec` (and the default empty constructor), whose body is the $IL = \nu(f)$.

$$op_{xtr} \colon \mathbb{F} \to \mathbb{C}$$

$$op_{xtr}(f) = \bar{c} \text{ with } met(\bar{c}) = \langle \bar{z}, \bar{m} \rangle$$

$$\bar{z} = \text{ the default empty constructor}$$

$$\bar{m} \mid \mu(\bar{m}) = K_b^f(k) \cdot \nu(f) \cdot K_e^f(k) \cdot \rho$$

with a fresh k, and ρ as one or two return instruction as explained below.

Some operations are done in order to synthesize the signature of the method `exec`. The return type of the method is determined accordingly to the T`return` instruction found in IL. Through lemma 5.1.6 it is:

$$|ret(f.r.m)| \leq 1$$

Then the return type of `exec` will be t such that:

$$t = \begin{cases} \texttt{void} & \text{if } ret(f.r.m) = \emptyset \\ ret(f.r.m) & \text{otherwise} \end{cases}$$

The bytecode verifier also guarantees the absence of unreachable code in the source fragment (lemma 5.1.5), i.e. we are guaranteed that the execution flows off the end of the fragment bytecode. To accomplish the definition 4.5.1, property 6, and to emit a valid bytecode, a new `return` instruction is appended at the end of the IL, and this is what we called ρ. Furthermore, when $t \neq$ `void`, ρ cannot be one single return instruction since the default value for t (see [17]§2.5.1) must be pushed onto the stack before; in this cases, ρ will be the sequence $\langle \rho_1, \rho_2 \rangle$ with:

- ρ_1 as the pushing instruction:
  - `iconst_0` when $t =$ `int`[2]
  - `fconst_0` when $t =$ `float`
  - `lconst_0` when $t =$ `long`
  - `dconst_0` when $t =$ `double`
  - `aconst_null` when $t =$ `reference`
- ρ_2 as the returning instruction:
  - `ireturn` when $t =$ `int`
  - `freturn` when $t =$ `float`
  - `lreturn` when $t =$ `long`
  - `dreturn` when $t =$ `double`
  - `areturn` when $t =$ `reference`

To determinate the signature of `exec` other checks concerns the individuation of all the free local variables in f and their lifting to the arguments of the method. Such method will have $k = |floc(\nu(f))|$ arguments of appropriate [3] type. Consider the function

$$arg\colon \mathbb{M} \to \mathbb{TL}$$

$$arg(m) = \langle t_1, \ldots, t_n \rangle$$

that given a method returns its argument types, then we have that:

$$\forall\, v \in floc(\nu(f)),\, type(arg(\bar{m})) = type(v)$$

Nevertheless the method can have one additional argument, in first position, in the case that $\bar{m}$ is not declared to be static: since $\bar{m}$,when not declared static, is supposed to work with fields and methods of an instance of the class of which $\bar{c}$ is part of, the first argument will be used as "this" reference by the bytecode.

According to [17] in not static methods, the first register of $\mathcal{L}$ is always occupied by a reference to the an instance of the current object, what is called `this` in the Java language. When we create `exec` we do not make it to be static, but simply request as its first argument a reference to an instance of what the user intend to use as "current" object. In such a way, a register shifting is not required for those instructions that use the *local variable array* in position zero.

A register renumbering is required for those instructions instead that use the free variable. Since those variable are lifted to the method argument, their register index must be translated in the interval between 0 (or 1 if the method is not static) and $k-1$ (k), when all the other local variable must occupy higher indexes $\geq k$ ($> k$). More formally:

$$args(\bar{m}) = \phi \cdot \chi \cdot \psi$$

[2]This also is intended to be the default instruction for Java types: `boolean`, `byte`, `char` and `short`

[3]Notice that types inferred this way may differ from those in the source code; for example, `short` local variables will be promoted to `int` when lifted as arguments, according to the standard type conversion rules of the JVM [17]§3.11.1.

$$\phi = \begin{cases} \emptyset & \text{if } f.r.m \text{ is static} \\ \texttt{reference} & \text{if } f.r.m \text{ is not static} \end{cases}$$

$$\chi = \langle t_1, \ldots, t_n \rangle \mid t_i = v_i \text{ with } v \in floc(\nu(f))$$

$$\psi = \langle t_i, \ldots, t_n \rangle \mid t_i = v_i \text{ with } v \in loc(\nu(f)) \setminus floc(\nu(f))$$

Moreover we observe that when $f.r.m$ is not static, there can occur some access to fields or methods that has been not declared public. In this case the bytecode verifier will throw an access violation exception informing the user about the impossibility of the requested operation.

As final step, the exception table for $\bar{m}$ is computed by looking at the exception table found for $f.m$, in such a way that:

$$te(\texttt{exec}) = tc(f)$$

which indicates that any exception which is declared to be thrown by the source method, or caught by a `try-catch` surrounding f, is added to the `throws` clause of `exec`.

Implementation

The implementation passes through the class `BrickedMethod` which propose a constructor based on the source fragment f:

```
BrickedMethod( Fragment f )
```

All the above operation are performed in order: a cycle through all the instruction is executed to individuate the possible return type; the default `void` return type is set if no return instruction are found. Since the instructions in JDAsm are instances of different object (with the same parent class), this task can be achieved by simply performing the following check.

```
foreach instruction instr in f.code
  if( instr instanceof ARETURN ) type = reference
  if( instr instanceof IRETURN ) type = int
  if( instr instanceof FRETURN ) type = float
  if( instr instanceof DRETURN ) type = double
  if( instr instanceof LRETURN ) type = long
```

Then the method `getFreeVariables()` of f is called in order to obtain the list of all the free variables. Its implementation is based on the presence of the `LocalVariableTableAttribute` of the `CodeAttribute`.

```
mcode = f.getDMethod().getCode()
fcode = f.getCode()
lvt = fcode.getLocalVariableTableAttribute();
foreach element e in lvt
  if( e.start_pc > fcode.firstInstructionOffset() )
    then the store instruction that initialize this variable
    is inside the fragment (or the variable is completely
    outside: e.end_pc > fcode.lastInstructionOffset()): this
    is not a free variable
  else if( e.end_pc < fcode.firstInstructionOffset() )
```

```
    then the scope of this variable ends before the code of
    the fragment: this is not a free variable
  else
    r = register used
    foreach instruction instr in fcode
      if instr uses r
        e is a free variable
```

With this information the prototype of the new method is built to create m as an object of the class `DMethod` of the JDAsm library. Then the creation of the code attribute of m follows:

```
m.code = get fragment code
frees = get fragment free variables
k = count(frees)
foreach instruction instr in m.code
  if instr uses register
    r = instr.getRegister()
    if r is the j-th entry of frees
      make instr to use register j
    else
    if r is the j-th entry of not free variables
      make instr to use register k+j

if t = return type is void
  m.code.add( return )
else
  m.code.add( default value pushing instruction )
  m.code.add( t-return )
```

The exception table is finally filled with all the appropriate exceptions both with the exception table of the code of f and with the exception thrown by $f.r.m$.

The last step of the extrusion operation includes the possibility to executed the brand new method just created: a d `DClass` is created and an empty constructor is added to it through the appropriate method:

```
void addEmptyConstructor()
```

which substantially create a `DMethod` with the signature

```
public void <init> ()
```

and to its bytecode appends the following three instructions:

1. `aload_0`
2. `invokespecial( superclassName + ‘‘ <init> ()V’’ )`
3. `return`

Finally it statically set the max locals and the max stack size to be 1. Note that the `invokespecial` instruction initialize an empty super constructor since the super constructor of the `DClass` created for the extrusion is `Object`. Further the created `DMethod` m is added to d.

The build operation of d is called at this point (see the next chapter for how it works in detail), and the resulting bytecode is ready to be loaded into the JVM through the apposite methods of `BrickedMethod`:

```
Object Invoke( Object...  args )
```

which use the Java reflection to invoke the resulting Java method with the specified arguments. Note that the result of the invocation is an instance of type `Object`, meaning an instance of the returned value if the return type is not void, or `null` if it is void.

5.4 Build Operation

As we introduced in Section 5.2, we let the user to declare a set of operations over discovered fragments of a class, and then we use a lazy evaluation strategy: the requested operation of insertion and deletion are queued and not evaluated until a `build` operation is invoked; at that point, the queued operations are applied in order, and a new class is generated in-memory hosting the resulting code.

Let $\mathbb{O}$ be the domain of the operations declared for a class $c \in \mathbb{C}$ and $\mathbb{OL}$ the domain of a list of such operations. We define $op \in \mathbb{O}$ the tuple $\langle t, f_s, f_d, p, v, \rangle$ with

$$t = \begin{cases} \text{insertion} \\ \text{deletion_with_marker} \\ \text{deletion_without_marker} \\ \text{deletion_maker} \end{cases}$$

$f_s \in \mathbb{F}$ as the source fragment, and only in case that $t =$ insertion, $f_d \in \mathbb{F}$ as the destination fragment, p as the position of the insertion and v as the variable mapping.

To increase performance, since no intermediate code or classes have to be generated in the course of the manipulation, this lazy strategy offers an opportunity for optimizing the operation queue (e.g., all operations modifying a fragment which is later deleted can be skipped altogether) before the actual build[4].

Given $opl \in \mathbb{OL} = \langle op_1, \ldots, op_n \rangle$ defined by the user to be applied to c, such optimizations try to determinate the operations which have no effect, in order to put them out of opl. A typical situation could be this: if at operation i we insert fragment A inside B, at operation j with $i < j$ we delete fragment B, and no insert operations involving B as source fragment are declared between i and j, then we can assume that the operation i will have no effect in the final result, and we can put such operation off of the opl.

More formally we can state that:

[4]The current implementation does not apply any optimization; these issues are scheduled for future work.

Theorem 5.4.1. *An operation op_i is idempotent if it modifies, either by inserting or removing a fragment, the source fragment $\bar{f}$ of a subsequently deletion operation op_j and no inner insertion*[5] *operation op_k with $i < k < j$ use $\bar{f}$ as source fragment.*

In fact all the operation involving $\bar{f} = op_j.f_s$ either with $\bar{f}' = op_i.f_s$ with $\nu\bar{f}' \subset \nu\bar{f}$ if op_i is a deletion, or with $\bar{f} = op_i.f_d$ if op_i is an inner insertion, will result in no effect when the $\bar{f}$ is deleted from the manipulated class. It will result in a side effect instead if an insertion operation op_k use $\bar{f} = op_k.f_s = op_i.f_s$ since the new code inserted in $op_k.f_d$ will be the one relative to $\bar{f}$ before the application of op_i minus the code deleted by op_i.

Corollary 5.4.2. *If the build process skips all the idempotent operation the resulting class does not change*

and this is what the optimizations point to.

Implementation

The build operation begins with the invocation of the method `build()` of the class `CbClass`. Given d a `DClass()` representing the manipulated Java class before the application of the *opl*, the first step is to use JDAsm features of *self subclassing* and *super constructor wrapping* (Section 4.6) to produce a `DClass()` d' which is child of d. This is implemented as follows:

```
newSuperName = d.getName()
oldSuperName = d.getSuperclassName()
newName = newSuperName + "_" + counter++
setClassName( newName )
setSuperclassName( newSuperName )

foreach method m in code of d
if m.isConstructor()
 change the method descriptor of m accordingly to newName
 m.wrapSuperConstructor(oldSuperName, newSuperName
```

Subsequently all the operations $op \in opl$ are applied in order to produce a final d' representing the manipulated class with the manipulated bytecode.

Theorem 5.4.3. *The altered bytecode produced by JCodeBrick after the manipulation is a valid bytecode*

Since it is created with JDAsm, and the inner bytecode verifier will test for a valid bytecode before emitting it (Section 4.5), the produced class and methods are valid.

JDAsm plays a fundamental role in the application of all the operations described in Section 5.3.2,5.3.4, and finally it can perform the class loading to let the user to use the modified class, which, due to the *self subclassing* feature, can be used, in a Java program, as the parent class itself when accessing methods and fields.

[5] With inner insertion we intend $p = \{\text{after_start}, \text{before_end}\}$

5.5 Examples

Let us consider an application which has to perform frequently some check on given conditions. These checks can be very thorough and complex, and computationally expensive, but in most cases a more basic and more efficient approximation might be sufficient, depending on environmental conditions. As will be described in Section 6, we envision a situation where the checks have to be performed in real-time, so we do not want to pay the penalty for an indirect method call each time, and decide to use runtime code manipulation instead.

The method performing the checks could be as follows:

@Java *Source*	@Java *compiled code*

`class C1 {`
`...`
`public void m()`
`{`
`  ...`
`  @ComplexChecks {`
`  /* complex check code */`
`  }`
`  ...`
`}`
`...`
`}`
	`import jcodebrick.Fragment;`
	`import jcodebrick.MultiA;`
	
	`class C1 {`
	`...`
	`@MultiA(`
	`  ids={1},`
	`  value={@ComplexChecks} )`
	`public void m()`
	`{`
	`  ...`
	`  {`
	`  Fragment.begin(1);`
	`  /* complex checks code */`
	`  Fragment.end(1);`
	`  }`
	`  ...`
	`}`
	`...`
	`}`

The compiled @Java code will be in turn compiled by the Java compiler into the following bytecode:

```
@MultiA{ids={1}, value={@ComplexChecks}}
method m():
   Code:
     // Initial method code
     ...
     // Starting marker K_b
     iconst_1
     invokestatic jcodebrick/Fragment.begin
     ...
       // complex checks code
     ...
     // Ending marker K_e
     iconst_1
     invokestatic jcodebrick/Fragment.end
     ...
     // More method code
     ...
     // End of method
     return
```

The code to replace at runtime the complex checks fragment with the basic checks one, and invoke the modified method, could be as follows:

```
CbClass c = new CbClass( C1.class );
Fragment complex = c.getFragment("ComplexChecks");
Fragment basic = c.getFragment("BasicChecks");
...
complex.insertFragment(Fragment.BEFORE_START, basic);
complex.delete();
C1 cc=(C1)c.build().newInstance();
cc.m();
```

The bytecode of the modified method `m` is obtained through these two steps:

After the insertion	After the deletion

`Code:`
`// Initial method code`
`...`
`iconst_4`
`invokestatic jcodebrick/Fragment.begin`
`...`
`// Basic Fragment code`
`...`
`iconst_4`
`invokestatic jcodebrick/Fragment.end`
`iconst_1`
`invokestatic jcodebrick/Fragment.begin`
`...`
`// Complex Fragment code`
`...`
`iconst_1`
`invokestatic jcodebrick/Fragment.end`
`...`
`// More method code`
`...`
`// End of method`
`return`
	`Code:`
	`// Initial method code`
	`...`
	`iconst_4`
	`invokestatic jcodebrick/Fragment.begin`
	`...`
	`// Basic Fragment code`
	`...`
	`iconst_4`
	`invokestatic jcodebrick/Fragment.end`
	`...`
	`// More method code`
	`...`
	`// End of method`
	`return`

Chapter 6

Applications

The ability to modify the running code of an application in a structured, symbolic and type-safe way, while leaving the programmer able to express code fragments in source form, opens the way to a vast number of novel applications. In the following we will only list a few examples, serving as conceptual scenarios but with no aim of completeness. Before going into the details, it is worth remarking that similar techniques have been used already in the past, albeit typically in an ad hoc fashion, and often at the source level (e.g., classical aspect-oriented programming), or at program installation time (e.g., configuration-selecting installers, as in a OS installer that only installs drivers needed for the actual hardware). In contrast, our proposed technique is totally general, annotation can be used both at the source level and at the bytecode level, and operations can be performed at any stage of the life cycle of the application, even while the application is running and without requiring a restart.

6.1 Logging

At installation time, a program could contain statements whose purpose is to compute and log to some external file certain values describing the state of the application during its execution, as a way of monitoring its performances and correctness. After monitoring the system's logs for a while, it can be determined that the system is behaving correctly, and that there is no longer a need for a detailed log.

Current logging frameworks (e.g., Log4J [3]) can enable or disable the output to the log file dynamically, but cannot avoid computing the values, which might be costly or have other undesired side effects. In contrast, with `@Java` the logging statements (or blocks) can be marked with an annotation such as `@Log(level)`, and when it is determined that logging is no longer required, all the logging blocks below a given severity level can be removed from the running code, thus avoiding any associated computation and possibly improving performances significantly.

As a related example, the `@Log` fragments could be removed leaving the markers in place, and stored in a data structure, together with a reference to their original location. This way, it becomes also possible to reinstate them if at a later time logging is desired again.

6.2 Environment-based reconfiguration

It is often the case that a system has to react differently to certain events based on changing environment conditions. For example, a heavy-load dispatcher for a web server farm could operate normally under standard operating conditions, while monitoring the response times of the system. If these become too high, it could install in its running code a fragment to monitor incoming requests especially to identify denial-of-service attacks (this might entail maintaining and updating complex data structures, to perform pattern matching on the requests data and to identify sets of IP addresses from which a potential distributed-DoS attack is coming). If no DoS attack is recognized, the dispatcher would go back to the standard dispatch code. On the other hand, if such an attack is identified, the dispatcher could further substitute its request-dispatching code with a more precise, but less efficient, version which would guard against requests coming from potential DoS sources. The assumption here is that the more precise dispatching code, rejecting DoS requests upfront, will save processing costs later on in the requests handling chain. If, after some time, it is determined that the DoS attack has ended, the original, optimistic but faster code can be replaced again inside the dispatcher.

In a more flexible implementation, both attack-detecting code and hardened dispatching code could be loaded dynamically based on the type of attack, thus making the system able to detect and respond optimally to different threats.

Similar behavior could be obtained by calling virtual, abstract or interface methods to perform the monitoring, detection and dispatching functions, and switching to different implementations of the same when appropriate. However, this standard technique would leave several method invocations in place even when they are not needed, which might be undesirable for a very high-performance system. On the contrary, with `@Java` the mutable code is substituted in-place, with no need for indirection, thus guaranteeing better performances both in the optimistic case and in the hardened one.

6.3 Dynamic optimization

A numeric application could include some heavy computation, which could be performed either in floating point (e.g., using `double`s) or in fixed point (e.g., using `int`s and then scaling the results by a fixed amount). At install time, the application could measure the performances of both, and then insert into its own computation code the version which offers better performances.

Again, similar results could be obtained by guarding the computation with an `if` statement, or by calling a method, but if the variable fragment has to be executed a relevant number of times (which is not uncommon, e.g. with large matrix operations), the cumulative cost of evaluating the flags or calling the methods, multiplied by millions or billions of invocations, could become significant. In contrast, with `@Java` the insertion of the proper fragment in-line would be performed only once, regardless of the number of times the fragment is run.

It is also worth remarking that the choice between different versions of a code fragment could be done dynamically, possibly switching between multiple versions based on external conditions. For example, using a floating point version

can be too costly if another numerical application is running concurrently (e.g., due to the need of storing and retrieving all the FPU registers at every context switch), but may be more convenient otherwise, so the application could periodically re-check the performances of the various versions of the code available, and choose a different one to execute based on current performances (again, saving on indirection costs as the chosen fragment would be inserted in-line).

6.4 Adaptable declarative security

The native security model in Java is *operational*, meaning that code performing a protected function has to call specific methods to check whether the caller has the right permission to invoke the given function. This might be inconvenient and error-prone, and moreover the entire security model of an application is wired-in once the application is written and compiled[1].

With `@Java`, a programmer can mark relevant sections of code with annotations such as `@GrantPermission(perm)`, `@AcquirePermission(perm)` and `@RequirePermission(perm)`, thus moving to a declarative model instead. One of the advantages is that in `@Java` permission-related annotations can be placed on statements and blocks, thus providing finer control over which sections of the code are critical (and satisfying Denning's principles). Another advantage is that the operational code needed to actually grant, acquire and check permissions can be injected at the appropriate places automatically, and – moreover – it can be changed, at runtime, to suit different security models as appropriate from time to time.

6.5 Parallelization

In parallel applications, it is customary to use dialects of common programming languages extended with keywords used to declare properties relevant for the parallel execution of the code. This approach typically requires custom compilers, which produce parallelism-handling code based on the custom keywords.

As we have seen in Figure 3.1, we could use a `@Parallel` annotation placed on a `for` statement to declare that the iterations of the `for` are independent and could potentially be executed in parallel. Then, an application could inject in those places code to actually realize the parallelism, choosing whatever implementation is more appropriate for the JVM/OS/hardware combination the program is running on (e.g., no parallelism at all, or creating a certain number of threads or processes based on how many CPUs are available on the machine, etc.).

Even more interesting, with the emergence of virtualization systems, it is becoming increasingly common that an application can be run on a virtualized server, and in that case the server could be dynamically reconfigured to allocate or simulate a variable number of CPUs - in which case, the application can react by changing its parallelization strategy and injecting different thread-handling fragments at `@Parallel` locations.

[1]The Java security model provides for that by externalizing policy decisions in a text file which can be edited by the user, but with limited flexibility, essentially implementing a source-based permission policy.

Chapter 7

Case Study

To better understand the usefulness of @Java we proceed now through concrete example told in details. Our example application must perform continuously mathematical operations with arrays of 1024 integers that it receives in input. For any of them, it must execute 500.000 times a convulsion operation, which consists in making the value at position i to get closer to the one at position $i + 1$ by adding or increasing the value i by its one hundredth.

The application is designed to pass most of its time in doing mathematical operations, and since its installation targets are hosts with very different architectures, the application comes with two routines to make requested operations:

a) one make computation over integers

b) the other one uses floating point

The routine a works with data in its original format making over it the needed computations; the routine b instead is optimized for such *CPU*s that, having for instance a dedicated *FPU* unit, can execute operation in floating point faster than the one over integers. The latter routine, since the input data are integers, executes first a conversion from "int" to "float" of the elements of the array, then performs the convulsion, and then converts back the result into integers again.

The application, at install-time, uses @Java to generate part of the code that will be executed at run-time, choosing which one of the two routines results to be the faster in the target host, and injecting the code of elected one directly into the object class.

As first thing, the application provides a dummy class with an ad hoc method that will never be called directly, but which will be used instead as container to

define the body of the two routines:

```
public class ComputationsFormulae
{
public void computationConvulsion()
{
int len = 0;
int[] values = null;
int convulsionNumber = 1;

@IntegerComputation
for( int k = 0; k < convulsionNumber; k++ ) {
  for( int i = 0; i < len - 1; i++ ) {
if( values[i] < values[i+1] ) {
  values[i] += (values[i]/100);
} else {
  values[i] -= (values[i]/100);
}
  }
}

@FloatComputation
{
  float[] fvalues = new int[len];
  for( int i = 0; i < len; i++ )
fvalues[i] = values[i];

  for( int k = 0; k < convulsionNumber; k++ ) {
for( int i = 0; i < len - 1; i++ ) {
  if( fvalues[i] < values[i+1] ) {
fvalues[i] += (fvalues[i]/100);
  } else {
fvalues[i] -= (fvalues[i]/100);
  }
}
  }

  for( int i = 0; i < len; i++ )
values[i] = (int)fvalues[i];
}
}
}
```

The class so defined is then parsed to generate the relative Java 5 source and

then is compiled, obtaining the following bytecode:

```
// * Declaration and initialization of the variables
0 iconst_0
1 istore_1                // * free variable "int len", index #1
2 aconst_null
3 checkcast #102 <[I>
6 astore_2                // * free variable "int[] values", index #2
7 iconst_0
8 istore_3                // * free variable "int convulsionNumber", index #3

// * Begin of the block annotated with @IntegerComputation
9 iconst_4
10 invokestatic #25 <jcodebrick/Fragment.Begin>
13 iconst_0                // * Creation of "k", not free
14 istore 4                // * Initialization of "k", index #4
16 goto 83 (+67)           // * cycle beginning
..                         // * Code of the computation.. the index #1, #2 and #3
..                         // * of the free variables are used
80 iinc 4 by 1             // * increment of "k", index #4
83 iload 4
85 iload_3
86 if_icmplt 19 (-67)      // * test of "k" and "convulsionNumber"
89 iconst_4
90 invokestatic #37 <jcodebrick/Fragment.End>
// * End of the block annotated with @IntegerComputation

// * Begin of the block annotated with @FloatComputation
93 iconst_5
94 invokestatic #25 <jcodebrick/Fragment.Begin>
97 iload_1                 // * the new array of converted values
98 newarray 6 (float)      // * will have index #1
..
127 iconst_0                // * the variable "k" used as index of the cycle
128 istore 5                // * here occupies the index #5
130 goto 204 (+74)          // * cycle beginning
..
201 iinc 5 by 1
204 iload 5
206 iload_3
207 if_icmplt 133 (-74)     // * test of "k" and "convulsionNumber"
210 iconst_5
211 invokestatic #37 <jcodebrick/Fragment.End>
// * End of the block annotated with @FloatComputation

214 return
```

Also the main class that will do the real computation is writtin with the `@Java` *language* extension and must be parsed to support an intra-code annotation as

insertion point:

```
class MainComputation {
..
public void compute( ) {
..
int len = ..;
int[] values = ..;
int convulsions = ..;
..
@Computation;    // * here will be injected the code of the choosen routine
..
}
..
}
```

At the moment of installation the application tests the two routines of the class `ComputationsFormulae` with the extrusion operation:

```
..
int arrayLength = 1024;
int convulsionNumber = 500000;
int[][] vals = getTestArrays();
..
Fragment f_int = routineClass.getFragment("IntegerComputation");
BrickedMethod m_int = new BrickedMethod(f_int);
double startTime_int = new GregorianCalendar().getTimeInMillis();
for( int r = 0; r < ntimes; r++ )
m_int.Invoke( arrayLength, vals[r], convulsionNumber );
double integerTime = (new GregorianCalendar().getTimeInMillis()) - startTime_int;
..
Fragment f_flt = routineClass.getFragment("FloatComputation");
BrickedMethod m_flt = new BrickedMethod(f_flt);
double startTime_flt = new GregorianCalendar().getTimeInMillis();
for( int r = 0; r < ntimes; r++ )
m_flt.Invoke( arrayLength, vals[r], convulsionNumber );
double floatTime = (new GregorianCalendar().getTimeInMillis()) - startTime_flt;
```

After having analyzed the following output:

```
CbClass created in 188.0 ms
Tested 5 arrays of 1024 elements
Time spent by integer computations: 39813.0 ms
Time spent by floating point computations: 33703.0 ms
```

the application figures out that the current CPU is optimized to perform calculations in floating point since, despite the two cycles to convert values from "int" to "float" and viceversa, the routine b resulted faster; hence, the following step is to create a modified copy of the class `MainComputation` injecting the fast code of b directly in the method of the real computation:

```
CbClass cb = new CbClass( MainComputation.class);
Fragment dst = cb.getFragment("Computation");
int newId = dst.insertFragment( InsertionPosition.AFTER_START, f_flt, new String[]{
"len", "len", "values", "values", "convulsionNumber", "convulsions"});
cb.removeInsertedFragment(newId, true );
src.deleteMarker();
cb.build().writeToFile( installationDir );
```

If we take a look at the generated bytecode we can follow the automatic map-

pings done by the @Java *system* of the registers index:

```
..
102 iload_1             // * the index of "len" is left the same
103 newarray 6 (float)  // * as in the annotated block: #1
..
132 iconst_0            // * the index of "k" is changed instead
133 istore 9            // * from #5 to #9
135 goto 209 (+74)      // * cycle beginning
..
206 iinc 9 by 1
209 iload 9
211 iload_3             // * also the variable "convulsion" occupies
212 if_icmplt 138 (-74) // * again register number #3
.. other operations: the markers have been deleted ..
```

The class above defined will be a copied class, and by convention, it will be named `MainComputation_0`. Once the creation is ended, all the files are correctly installed and the program is ready to be runned with the following starting routine:

```
Class<?> executor = Class.forName("MainComputation_0");
MainComputation r = (MainComputation)executor .newInstance();
r.compute();
```

This example, even if quite simple, shows an interesting approach at the exploitation of the features of the system, aimed to slightly optimize calculations: the inline expansion of the proper routine in fact saves a lot of instructions in terms of tests and method calls, that in quantity in the order of some milion, can make the sensible performance improvement to be appreciated.

Chapter 8

Conclusions and future work

In this technical report we described `@Java`, a variant of Java which permits to manipulate an application code at runtime in a structured, symbolic and type-safe way, by using annotations placed on single statements or blocks to define code fragments and locations in the code.

While sharing similarities in its scope with traditional aspect-oriented programming techniques, our contribution places a greater emphasis on the possibility of manipulating the code at run-time, whereas aspect weaving is typically performed at compile-time only. This important distinction opens the way to a number of applications for which standard AOP techniques are not flexible enough.

The techniques we presented, building on top of execution technology provided by virtual execution environments and on novel language features such as custom annotations, change in a fundamental way the notion of lifecycle of a program. Whereas customarily writing, compiling, linking, shipping, deploying, installing, loading and running a program were considered completely distinct phases, the ability to identify and process annotations both in source and in object (.class) form, and at runtime in executable code, in a sense blends this phases. Now, program code can be written at run-time; compilation can execute user-provided code based on annotations found in source files, an installer can manipulate the object code that has been deployed based on a specific machine architecture, etc.

The `@Java` language and its code-manipulation capabilities are a contribution towards reaching this vision, in which code manipulation and program re-writing is a substantial part of execution. The language itself could be extended to address annotation of (sub-)expressions, to cover cases where one might want to manipulate, say, a `new` expression, or a method invocation. We intend to address this issue as part of future work.

More work is also needed in two other directions: (i) on the application side, by providing run-time support and case studies for common needs (e.g., logging, security, parallelism), and (ii) on the technological side, by providing more flexible and more efficient implementations of the code-manipulation primitives we have defined.

The `@Java` source-to-source compiler and the associated JDAsm code manipulation library have been released as open source, and are currently available, respectively, at `http://at-java.sourceforge.net` and `http://jdasm.sourceforge.net`.

Bibliography

[1] Java Compiler Compiler [tm] (JavaCC [tm]) - The Java Parser Generator. `http://https://javacc.dev.java.net/`.

[2] Apache Software Foundation. Bcel: Bytecode engineering library. `http://jakarta.apache.org/bcel`.

[3] Apache Software Foundation. Apache log4j, 2007. `http://logging.apache.org/log4j`.

[4] W. Cazzola, A. Cisternino, and D. Colombo. [a]C#: C# with a customizable code annotation mechanism. In *Proceedings of the 10th Annual ACM Symposium on Applied Computing (SAC'05)*, pages 1274–1278, Santa Fe, New Mexico, USA, Mar. 2005. ACM Press.

[5] S. Chiba. JavaAssist. `http://www.csg.is.titech.ac.jp/~chiba/javaassist`.

[6] S. Chiba. Load-time structural reflection in Java. In *Proc. of ECOOP 2000*, volume 1850 of *LNCS*, pages 313–336. Springer-Verlag, 2000.

[7] S. Chiba and M. Nishizawa. An easy-to-use toolkit for efficient java bytecode translators. In *Proc. of 2nd Int'l Conf. on Generative Programming and Component Engineering (GPCE '03)*, volume 2830 of *LNCS*, pages 364–376. Springer-Verlag, 2003.

[8] A. Cisternino and V. Gervasi. Meta-programming without quasi-quotation. In *Proceedings of the 2nd MetaOCaml Workshop*, Tallin, Estonia, Sept. 2005.

[9] ECMA. *Standard ECMA-334 – C# Language Specification*. European Computer Manufacturer Association, Geneva, 4th edition, 2006.

[10] ECMA. *Standard ECMA-335 – Common Language Infrastructure (CLI)*. European Computer Manufacturer Association, Geneva, 4th edition, 2006.

[11] M. D. Ernst. JSR 308: Annotations on Java types, 2007. (updated on March 2008).

[12] A. K. G. Attardi, A. Cisternino. CodeBricks: code fragments as building blocks. *SIGPLAN Notices*, 38(10):306–314, Oct. 2003.

[13] G. Galilei. Applicazioni delle annotazioni alla manipolazione a runtime di codice su macchine virtuali. Master's thesis, University of Pisa, 2007. (in Italian).

[14] Giacomo A. Galilei. @java system, 2008. `http://at-java.sourceforge.net`.

[15] Giacomo A. Galilei. Jdasm, a code manipulation library, 2008. `http://jdasm.sourceforge.net`.

[16] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, 3rd edition, 2005.

[17] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2nd edition, 1999.

[18] G. L. Steele. *Common LISP – The language*. Digital Press, 2nd edition, 1990.

[19] W. Taha and T. Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1–2):211–242, 2000.

Appendix A

JDAsm API Documentation

A.1 Package org.ldp.jdasm

Classes

DClass 63

This is the main class of JDasm project: it is a disassembled representation of a java class.

You can create a DClass from scratch or open an existing java class as a DClass to disassemble, read, modify it and finally create a new class ready to be instanced and used.

DClass is the representation of java class file, so you can perform almost any operation you want to do in a class file: e.g.

DConstantPool 73

The constant pool of the java class file.

In the JDasm project the constant pool is mainly automatically managed; it means that whenever a field, a method, a code is added (or removed) the constant pool will be automatically updated at the build time (not before!).

DField 76

This class represents a java class field.

DMethod 82

This class represents a java class method.

DAttribute 90

This class is the super class of all the attributes.

A.1.1 Classes

A.1.2 Class DClass

This is the main class of JDasm project: it is a disassembled representation of a java class.

You can create a DClass from scratch or open an existing java class as a DClass to disassemble, read, modify it and finally create a new class ready to be instanced and used.

DClass is the representation of java class file, so you can perform almost any operation you want to do in a class file: e.g. the class fields management is allowed through the class DField , while the methods management (bytecode inclusive) is performed due to the DMethod class.

When you get done with editing, you are able to build a real java class: the bytecode is autogenerated as the class Constant Pool is automatically build.

Declaration

public class DClass
extends org.ldp.jdasm.attribute.Attributable

Constructors

- *DClass*
  `public` **DClass**`( )`
  - **Usage**
    - Constructs an empty DClass.
- *DClass*
  `public` **DClass**`( java.lang.Class` **clazz** `)`
  - **Usage**
    - Constructs a DClass reading a java class.
- *DClass*
  `public` **DClass**`( java.io.InputStream` **is** `)`
  - **Usage**
    - Constructs a DClass reading the bytecode from an input stream.
- *DClass*
  `public` **DClass**`( java.lang.String` **name** `)`
  - **Usage**
    - Constructs an empty DClass with given name and public accessFlags; superclass is also set to Object

Methods

- *addEmptyConstructor*
  `public void` **addEmptyConstructor**`( )`
  - **Usage**
    * Adds an empty constructor which calls the super constructor and returns.

      The added method will be public, named `<init>` and its descriptor will be `()V`. Such new inserted method will call an identical method of the same class of this DClass's super constructor.

- *addField*
  `public DField` **addField**`( org.ldp.jdasm.DField` **field** `)`
  - **Usage**
    * Adds a DField; if a field with the same name already exists, it is replaced.
  - **Returns** - the inserted DField

- *addField*
  `public DField` **addField**`( java.lang.String` **code** `)`
  - **Usage**
    * Adds a java field to this class; the field is obtained by parsing the code passed as argument. If a field with the same name already exists, it is replaced.
  - **Returns** - the inserted DField
  - **See Also**
    * `org.ldp.jdasm.DField.DField(String)`

- *addInterface*
  `public void` **addInterface**`( java.lang.String` **name** `)`
  - **Usage**
    * Adds an interface with given name

- *addMethod*
  `public DMethod` **addMethod**`( org.ldp.jdasm.DMethod` **method** `)`
  - **Usage**
    * Adds a DMethod; if a method with the same signature already exists, it is replaced.
  - **Returns** - the inserted DMethod

- *addMethod*
  `public DMethod` **addMethod**`( java.lang.String` **code** `)`
  - **Usage**
    * Adds a java method to this class; the method is obtained by parsing the code passed as argument. If a method with the same signature already exists, it is replaced.
  - **Returns** - the inserted DMethod
  - **See Also**

    * `org.ldp.jdasm.DMethod.DMethod(String)`

- *build*
  `public byte` **build**`( )`
  - **Usage**
    * Constructs the java bytecode representing this DClass and returns it
  - **See Also**
    * `org.ldp.jdasm.DClass.toBytecode()`
    * `org.ldp.jdasm.DClass.getComputedBytecode()`

- *classFileInfo*
  `public String` **classFileInfo**`( )`
  - **Usage**
    * Returns a printable String with this DField information. It uses a default indentation level of 2 white spaces

---

- *classFileInfo*
  `public String` **classFileInfo**`( int` **indent_level** `)`
  - **Usage**
    * Returns a printable String with this DClass information.
  - **Parameters**
    * `indent_level` - The indentation level (in white-spaces) for each generated line

---

- *constantPoolUserChildSize*
  `public int` **constantPoolUserChildSize**`( )`
  - **Usage**
    * Returns the number of IConstantPoolUser instances of this class.

- *constantPoolValueSize*
  `public int` **constantPoolValueSize**`( )`
  - **Usage**
    * Returns the number of values that this constant pool user wants to insert into the constant pool.

- *fieldCount*
  `public int` **fieldCount**`( )`
  - **Usage**
    * Returns the number of fields for this class

- *getAccessFlagDescription*
  `public static String` **getAccessFlagDescription**`( short` **flag** `)`
  - **Usage**
    * Returns a printable String of the access permission qualifiers passed as argument.
  - **Parameters**
    * `flag` - The access permission qualifier typical of each DClass class.

- *getAccessFlags*
  `public short` **getAccessFlags**`( )`
  - **Usage**
    - Get the access permission qualifier for this class.

- *getClassName*
  `public String` **getClassName**`( )`
  - **Usage**
    - Get the class name.

- *getComputedBytecode*
  `public byte` **getComputedBytecode**`( )`
  - **Usage**
    - Returns the bytecode that is already built. If no build has been done yet, then it returns null

---

- *getConstantPool*
  `public DConstantPool` **getConstantPool**`( )`

- *getConstantPoolUserChild*
  `public AConstantPoolUser` **getConstantPoolUserChild**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th IConstantPoolUser instance of this class.

- *getConstantValue*
  `public Object` **getConstantValue**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th value that this constant pool user wants to insert into the constant pool.
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *getConstantValueType*
  `public int` **getConstantValueType**`( int` **idx** `)`
  - **Usage**
    - Returns the value type of the idx-th value that this constant pool user wants to insert into the constant pool. The list of the types is declared in DConstantPool

---

- *getField*
  `public DField` **getField**`( java.lang.String` **name** `)`
  - **Usage**
    - Get the field with the given name.
  - **Returns** - a DField with matched name, or null if no such DField exists

---

- *getFields*
  `public DField` **getFields**`( )`
  - **Usage**
    - Returns all the fields

- *getFullyQualifiedClassName*
  `public String` **getFullyQualifiedClassName**`( )`
  - **Usage**
    - Get the fully qualified class name.

- *getFullyQualifiedSuperClassName*
  `public String` **getFullyQualifiedSuperClassName**`( )`
  - **Usage**
    - Get the fully qualified super class name.

- *getInterface*
  `public String` **getInterface**`( int` **idx** `)`
  - **Usage**
    - Get the interface name at specified index

- *getInterfaces*
  `public String` **getInterfaces**`( )`
  - **Usage**
    - Get all the interface names

- *getMagic*
  `public int` **getMagic**`( )`
  - **Usage**
    - Get the magic number for this class file (default is 0xCAFEBABE).

- *getMajorVersion*
  `public short` **getMajorVersion**`( )`
  - **Usage**
    - Get the major number for this class file (default is 49).

- *getMethod*
  `public DMethod` **getMethod**`( java.lang.reflect.Method` **method** `)`
  - **Usage**
    - Get the method associated to the given reflective `method`.

      If you load this DClass from a java class, you can use the reflection to inspect the methods inside the original class and use this method to get the correspondent DMethod. Returns null if no dmethod is found.

- *getMethod*
  `public DMethod` **getMethod**`( java.lang.String` **signature** `)`
  - **Usage**
    - Get the method with the given signature.
  - **Returns** - a DMethod with matched signature, or null if no such DMethod exists
  - **See Also**
    - `org.ldp.jdasm.DMethod.DMethod(String)`

    - `org.ldp.jdasm.DMethod.getSignature()`

- *getMethods*
  `public DMethod` **getMethods**`( )`
  - **Usage**
    - Returns all the methods

- *getMinorVersion*
  `public short` **getMinorVersion**`( )`
  - **Usage**
    - Get the minor number for this class file (default is 0).

- *getSuperclassName*
  `public String` **getSuperclassName**`( )`
  - **Usage**
    - Get the name of the super class.

- *interfaceCount*
  `public int` **interfaceCount**`( )`
  - **Usage**
    - Returns the number of interfaces held

- *isBuildCustomAttribute*
  `public static boolean` **isBuildCustomAttribute**`( )`
  - **Usage**
    - Returns the buildCustomAttribute flag for this DClass. When loaded from a class, they probably have references into the constant table that cannot be updated without the attribute knowledge. So false (default) is a safe choice.

- *makeChildOfItself*
  `public void` **makeChildOfItself**`( java.lang.String` **newname** `)`
  - **Parameters**
    - `newname` -
  - **Exceptions**
    - `org.ldp.jdasm.exception.MethodDescriptorException` - if some of the constructor methods have errors in their types specification, and JDasm cannot create their method descriptor.

- *methodCount*
  `public int` **methodCount**`( )`
  - **Usage**
    - Returns the number of methods for this class

- *removeField*
  `public boolean` **removeField**`( org.ldp.jdasm.DField` **field** `)`
  - **Usage**
    - Removes the given field (match is done on the name) from the list.
  - **Parameters**

    - `field` - A DField with the name of the field you want to remove.
  - **Returns** - true if a field is removed (if it was present), false it it is not present in the list.

- *removeField*
  `public boolean` **removeField**`( java.lang.String` **name** `)`
  - **Usage**
    - Removes the field with the given name.
  - **Returns** - true if a field is removed (if it was present), false it it is not present in the list.

- *removeInterface*
  `public void` **removeInterface**`( int` **idx** `)`
  - **Usage**
    - Removes the interface at given index

- *removeInterface*
  `public boolean` **removeInterface**`( java.lang.String` **name** `)`
  - **Usage**
    - Removes the interface with given name
  - **Returns** - true if the interface is found in the list and removed, false if it is not present

---

- *removeMethod*
  `public boolean` **removeMethod**`( org.ldp.jdasm.DMethod` **method** `)`
  - **Usage**
    - Removes the given method (match is done on the signature) from the list.
  - **Parameters**
    - `method` - A DMethod with the signature of the method you want to remove.
  - **Returns** - true if a method is removed (if it was present), false it it is not present in the list.

- *removeMethod*
  `public boolean` **removeMethod**`( java.lang.String` **signature** `)`
  - **Usage**
    - Removes the method with the given method signature
  - **Returns** - true if a method is removed (if it was present), false it it is not present in the list.
  - **See Also**
    - `org.ldp.jdasm.DMethod.DMethod(String)`
    - `org.ldp.jdasm.DMethod.getSignature()`

- *setAccessFlags*
  `public void` **setAccessFlags**`( short` **accessFlags** `)`
  - **Usage**
    - Get the access permission qualifier for this class.

- *setBuildCustomAttribute*
  `public static void` **setBuildCustomAttribute**`( boolean` **buildCustomAttribute** `)`
  - **Usage**
    * Sets the buildCustomAttribute flag for this DClass. When loaded from a class, they probably have references into the constant table that cannot be updated without the attribute knowledge. So false (default) is a safe choice. If you want to build custom attribute pass `true` here. If you want to use custom attribute with constant pool reference you can add manually constants you need and tell the constant pool to build the unlinked constants too.
  - **See Also**
    * `org.ldp.jdasm.DConstantPool.addConstant(org.ldp.jdasm.constantpool.CpInfo)` (in A.1.3, page 74)
    * `org.ldp.jdasm.DConstantPool.setKeepUnlinkedConstant(boolean)` (in A.1.3, page 76)

- *setClassName*
  `public void` **setClassName**`( java.lang.String` **className** `)`
  - **Usage**
    * Set the class name.

- *setConstantValueIndex*
  `public void` **setConstantValueIndex**`( int` **idx**`, short` **cpool_index** `)`
  - **Usage**
    * The constant pool tells the user that the value retrieved with AConstantPoolUser#getConstantValue(int) at index "idx" has received the "cpool_index" index in the constant pool.
  - **Parameters**
    * `idx` - the index of the value mapped with the ones returned by getConstantValue(int)
    * `cpool_index` - the index of the value in the constant pool

- *setMagic*
  `public void` **setMagic**`( int` **magic** `)`
  - **Usage**
    * Set the magic number for this class file (default is 0xCAFEBABE).

- *setMajorVersion*
  `public void` **setMajorVersion**`( short` **majorVersion** `)`
  - **Usage**
    * Set the major number for this class file (default is 49).

- *setMinorVersion*
  `public void` **setMinorVersion**`( short` **minorVersion** `)`
  - **Usage**
    * Set the minor number for this class file (default is 0).

- *setSuperclassName*
  `public void` **setSuperclassName**`( java.lang.String` **superclassName** `)`
  - **Usage**
    - Set the name of the super class. Remember that when a class X is supposed to be child of a super parent class Y, then X is also supposed to have a call to the Y constructor in any of X constructor methods
- *showClassInfo*
  `public void` **showClassInfo**`( )`
  - **Usage**
    - Prints on stdout the info returned by DClass#classFileInfo()
- *toBytecode*
  `public byte` **toBytecode**`( )`
  - **Usage**
    - Returns this DClass converted to java bytecode. If no build has been done yet, then this method calls DClass#build() before, otherwise the already computed bytecode is used.
- *toClass*
  `public Class` **toClass**`( )`
  - **Usage**
    - Convert the bytecode into a class. If no build has been done yet, then this method calls DClass#build() before, otherwise the already computed bytecode is used.
  - **Returns** - The java Class representing the DClass on success, or null if something goes wrong
- *writeClassToFile*
  `public boolean` **writeClassToFile**`( java.lang.String` **filename** `)`
  - **Usage**
    - We give a fast method to write the class to a file. No exception are thrown, it just returns true on success, false on error (error print on output).
  - **Parameters**
    - `filename` - the path where to write the class. If it is a directory, then the file name will be the class name plus ".class"

Methods inherited from class
`org.ldp.jdasm.attribute.Attributable`

(in A.3.2, page 136)
- *addAttribute*
  `public DAttribute` **addAttribute**`( org.ldp.jdasm.DAttribute` **attr** `)`
  - **Usage**
    - Adds a DAttribute.
  - **Returns** - the inserted DAttribute

- *attributeCount*
  `public int` **attributeCount**`( )`
  - **Usage**
    - Returns the number of attribute for this class
- *getAttribute*
  `public DAttribute` **getAttribute**`( int` **idx** `)`
  - **Usage**
    - Get the DAttribute at specified position
- *getAttribute*
  `public DAttribute` **getAttribute**`( java.lang.String` **name** `)`
  - **Usage**
    - Returns the DAttribute with specified name, or null if it doesn't exist.
- *getAttributes*
  `public DAttribute` **getAttributes**`( )`
  - **Usage**
    - Returns all the attributes
- *hasAttribute*
  `public boolean` **hasAttribute**`( java.lang.String` **name** `)`
  - **Usage**
    - Returns true if an attribute with given name exists.
- *removeAttribute*
  `public boolean` **removeAttribute**`( org.ldp.jdasm.DAttribute` **attr** `)`
  - **Usage**
    - Removes the specified DAttribute (match on equals())
  - **Returns** - true if the attribute is removed (if it was present), false it it is not present in the list.
- *removeAttribute*
  `public void` **removeAttribute**`( int` **idx** `)`
  - **Usage**
    - Removes the DAttribute at specified position
- *removeAttribute*
  `public boolean` **removeAttribute**`( java.lang.String` **name** `)`
  - **Usage**
    - Removes the DAttribute with specified name. Returns true if it is removed (if it was present), false if it is not present in the list.

Methods inherited from class `org.ldp.jdasm.constantpool.AConstantPoolUser`

(in A.2.4, page 97)

- *constantPoolUserChildSize*
  `public abstract int` **constantPoolUserChildSize**`( )`
  - **Usage**
    - Returns the number of IConstantPoolUser instances of this class.
- *constantPoolValueSize*
  `public abstract int` **constantPoolValueSize**`( )`
  - **Usage**
    - Returns the number of values that this constant pool user wants to insert into the constant pool.

- *getConstantPoolUserChild*
  `public abstract AConstantPoolUser` **getConstantPoolUserChild**`( int` **idx** `)`
  - **Usage**
    * Returns the idx-th IConstantPoolUser instance of this class.
- *getConstantValue*
  `public abstract Object` **getConstantValue**`( int` **idx** `)`
  - **Usage**
    * Returns the idx-th value that this constant pool user wants to insert into the constant pool.
  - **Exceptions**
    * `java.lang.Exception` - if the value is not ready (since the control is done only at build time)
- *getConstantValueType*
  `public abstract int` **getConstantValueType**`( int` **idx** `)`
  - **Usage**
    * Returns the value type of the idx-th value that this constant pool user wants to insert into the constant pool. The list of the types is declared in DConstantPool
  - **Exceptions**
    * `java.lang.Exception` - if the value is not ready (since the control is done only at build time)
- *setConstantValueIndex*
  `public abstract void` **setConstantValueIndex**`( int` **idx**`, short` **cpool_index** `)`
  - **Usage**
    * The constant pool tells the user that the value retrieved with AConstantPoolUser#getConstantValue(int) at index "idx" has received the "cpool_index" index in the constant pool.
  - **Parameters**
    * `idx` - the index of the value mapped with the ones returned by getConstantValue(int)
    * `cpool_index` - the index of the value in the constant pool

A.1.3 Class DConstantPool

The constant pool of the java class file.

In the JDasm project the constant pool is mainly automatically managed; it means that whenever a field, a method, a code is added (or removed) the constant pool will be automatically updated at the build time (not before!). At each build, the constant pool is populated with the new entry, while the old ones will be removed. To do this the constant pool explore the main AConstantPoolUser (that is the DClass) and recursively its child (see DConstantPool#linkConstants(AConstantPoolUser))

Declaration

public class DConstantPool
extends java.lang.Object
implements org.ldp.jdasm.constantpool.IBuildable

Constructors

- *DConstantPool*
  `public` **DConstantPool**`( )`
  - **Usage**
    - Creates an empty DConstantPool.

- *DConstantPool*
  `public` **DConstantPool**`( java.io.InputStream` **is** `)`
  - **Usage**
    - Creates a DConstantPool reading it from an input stream. The input stream's cursor must point to the beginning of the constant pool (it's entries count), and it is made to advance until the end (the last entry). All the constants are loaded.

Methods

- *addConstant*
  `public short` **addConstant**`( org.ldp.jdasm.constantpool.CpInfo` **info** `)`
  - **Usage**
    - Adds the `info` constant to the list and returns its index. If such constant already exists it is not inserted again, and the present's one index is returned;

- *build*
  `public int` **build**`( byte []` **tofill**`, int` **atindex** `)`
  - **Usage**
    - It builds the content of this class into a final byte array. Such array is passed as first argument, and the index from where start to insert bytes is the second argument.
  - **Parameters**
    - `tofill` - the array to fill
    - `atindex` - the index from where start to fill
  - **Returns** - the number of byte inserted

- *build*
  `public int` **build**`( java.io.OutputStream` **output** `)`
  - **Usage**
    - It builds the content of this class into an output stream. This method is a faster version of int) since it doesn't need to call the IBuildable#getBuildLength() before.
  - **Parameters**
    - `output` - the stream to write onto
  - **Returns** - the number of byte inserted
  - **Exceptions**
    - `on` - error while writing onto the stream

- *classFileInfo*
  `public String` **classFileInfo**`( int` **indent_level** `)`

- **Usage**
  - Returns a printable String with this DClass information.
- **Parameters**
  - `indent_level` - The indentation level (in white-spaces) for each generated line

---

- *clear*
  `public void` **clear**`( )`
  - **Usage**
    - Clears all the entries in the constant pool

- *getBuildLength*
  `public int` **getBuildLength**`( )`
  - **Usage**
    - Returns the length of portion of byte array the IBuildable class is going to create. If this class has instances of other IBuildable classes, then it will return the whole sum of all the various getBuildLength().

- *getConstant*
  `public CpInfo` **getConstant**`( byte` **idx** `)`
  - **Usage**
    - Returns the constant pool element at index `idx`. Such index is in the form of the constant pool, so it goes from 1 to n, instead of classical 0 to n-1.

- *getConstant*
  `public short` **getConstant**`( org.ldp.jdasm.constantpool.CpInfo` **info** `)`
  - **Usage**
    - Returns the index of given constant `info</info>in this constant pool if it is present. It returns -1 if the constant is not present`
  - **Parameters**
    - `info` - the constant to check
  - **Returns** - the index of given constant, or -1 if not present

---

- *getConstant*
  `public CpInfo` **getConstant**`( short` **idx** `)`
  - **Usage**
    - Returns the constant pool element at index `idx`. Such index is in the form of the constant pool, so it goes from 1 to n, instead of classical 0 to n-1.

- *isKeepUnlinkedConstant*
  `public boolean` **isKeepUnlinkedConstant**`( )`
  - **Usage**
    - Returns the value indicating if the constant pool at build time will discard all the constants that are not requested by the other parts of the parent DClass to be inserted in the final bytecode. The default value for a DConstantPool is false.

- *linkConstants*
  `public void` **linkConstants**`(`
  `org.ldp.jdasm.constantpool.AConstantPoolUser` **user** `)`
  - **Usage**
    * This function clear the internal state of the constant pool and rebuild a new one reading recursively the ConstantPoolUser and its children. After this call any IConstantPoolUser reachable from `user` will have constant pool index set.

- *setKeepUnlinkedConstant*
  `public void` **setKeepUnlinkedConstant**`( boolean` **keepUnlinkedConstant** `)`
  - **Usage**
    * Sets the value indicating if the constant pool at build time will discard all the constants that are not requested by the other parts of the parent DClass to be inserted in the final bytecode. The default value for a DConstantPool is false.

- *size*
  `public int` **size**`( )`
  - **Usage**
    * Returns the constants in the pool

A.1.4 Class DField

This class represents a java class field. It can have an access permission qualifier (such as "private", "public", ...) a type and a name. It can also have a list of attributes.

The attributes that a standard jvm permits are: ConstantValueAttribute , SyntheticAttribute , DeprecatedAttribute
The internal class do not store link with a *ConstantPool* until a build is done, so any change is fast and a link to a DClass or to a DConstantPool is not needed at construction time

Declaration

public class DField
extends org.ldp.jdasm.attribute.Attributable
implements org.ldp.jdasm.constantpool.IBuildable

Constructors

- *DField*
  `public` **DField**`( )`
  - **Usage**
    * Creates an empty DField.

- *DField*
  `public` **DField**`( java.io.InputStream` **is**`,`
  `org.ldp.jdasm.DConstantPool` **cpool** `)`

- **Usage**
  - Creates a DField attribute reading it from an input stream whose cursor must point to the beginning of the field. The constant pool needs to have all constant values used by this field and its attribute.

- *DField*
  `public` **DField**`( short` **access_flags**`, java.lang.String` **type**`, java.lang.String` **name** `)`
  - **Usage**
    - Creates a DField with given arguments.
  - **Parameters**
    - `access_flags` - (static, final, public, private, protected, ...)
    - `type` - the name of the class type (int, float, MySpecialObject[], ...)
    - `name` - the name of the field

- *DField*
  `public` **DField**`( java.lang.String` **code** `)`
  - **Usage**
    - Creates a DField from a java code string.

      It recognizes strings like: *<blockquote>"public int foo"*

      "static private java.lang.String bar"</blockquote> Throws an CodeParseException when an invalid string is found

Methods

- *build*
  `public int` **build**`( byte []` **tofill**`, int` **atindex** `)`
  - **Usage**
    - It builds the content of this class into a final byte array. Such array is passed as first argument, and the index from where start to insert bytes is the second argument.
  - **Parameters**
    - `tofill` - the array to fill
    - `atindex` - the index from where start to fill
  - **Returns** - the number of byte inserted

- *build*
  `public int` **build**`( java.io.OutputStream` **output** `)`
  - **Usage**
    - It builds the content of this class into an output stream. This method is a faster version of int) since it doesn't need to call the IBuildable#getBuildLength() before.
  - **Parameters**
    - `output` - the stream to write onto
  - **Returns** - the number of byte inserted
  - **Exceptions**
    - `on` - error while writing onto the stream

- *classFileInfo*
  `public String` **classFileInfo**`( int` **indent_level** `)`
  - **Usage**
    - Returns a printable String with this DField information.
  - **Parameters**
    - `indent_level` - The indentation level (in white-spaces) for each generated line

---

- *constantPoolUserChildSize*
  `public int` **constantPoolUserChildSize**`( )`
  - **Usage**
    - Returns the number of IConstantPoolUser instances of this class.

- *constantPoolValueSize*
  `public int` **constantPoolValueSize**`( )`
  - **Usage**
    - Returns the number of values that this constant pool user wants to insert into the constant pool.

- *equals*
  `public boolean` **equals**`( java.lang.Object` **o** `)`
  - **Usage**
    - We match a DField that has the same name and type, or a String if it is equals to the name of the field.

- *getAccessFlagDescription*
  `public static String` **getAccessFlagDescription**`( short` **flag** `)`
  - **Usage**
    - Returns a printable String of the access permission qualifiers passed as argument.
  - **Parameters**
    - `flag` - The access permission qualifier typical of each DField class.

- *getAccessFlags*
  `public short` **getAccessFlags**`( )`
  - **Usage**
    - Get the access permission qualifier.

- *getBuildLength*
  `public int` **getBuildLength**`( )`
  - **Usage**
    - Returns the length of portion of byte array the IBuildable class is going to create. If this class has instances of other IBuildable classes, then it will return the whole sum of all the various getBuildLength().

- *getConstantPoolUserChild*
  `public AConstantPoolUser` **getConstantPoolUserChild**`( int` **idx** `)`

- **Usage**
  - Returns the idx-th IConstantPoolUser instance of this class.

- *getConstantValue*
  `public Object` **getConstantValue**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th value that this constant pool user wants to insert into the constant pool.
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *getConstantValueType*
  `public int` **getConstantValueType**`( int` **idx** `)`
  - **Usage**
    - Returns the value type of the idx-th value that this constant pool user wants to insert into the constant pool. The list of the types is declared in DConstantPool

---

- *getFieldDescriptor*
  `public String` **getFieldDescriptor**`( )`
  - **Usage**
    - Returns the FieldDescriptor for this DField as specified for the java class file.
  - **See Also**
    - `org.ldp.jdasm.constantpool.TypeDescriptor.getFieldDescriptorByString(String)`

- *getName*
  `public String` **getName**`( )`
  - **Usage**
    - Get the name of the field.

- *getType*
  `public String` **getType**`( )`
  - **Usage**
    - Get the type of the field.

- *isAccessFlagsSet*
  `public boolean` **isAccessFlagsSet**`( )`
  - **Usage**
    - Returns true if at least one access permission qualifier has been set.

- *isNameSet*
  `public boolean` **isNameSet**`( )`
  - **Usage**
    - Returns true if the name of this field has been set.

- *isTypeSet*
  `public boolean` **isTypeSet**`( )`

  - **Usage**
    * Returns true if the type of this field has been set.

- *setAccessFlags*
  `public void` **setAccessFlags**`( short` **accessFlags** `)`
  - **Usage**
    * Set the access permission qualifier.

- *setConstantValueIndex*
  `public void` **setConstantValueIndex**`( int` **idx**`, short` **cpool_index** `)`
  - **Usage**
    * The constant pool tells the user that the value retrieved with AConstantPoolUser#getConstantValue(int) at index "idx" has received the "cpool_index" index in the constant pool.
  - **Parameters**
    * `idx` - the index of the value mapped with the ones returned by getConstantValue(int)
    * `cpool_index` - the index of the value in the constant pool

- *setName*
  `public void` **setName**`( java.lang.String` **name** `)`
  - **Usage**
    * Set the name of the field.

- *setType*
  `public void` **setType**`( java.lang.String` **type** `)`
  - **Usage**
    * Set the type of the field. If you use a class type then you need a fully qualifying name (java.lang.String is ok, but String is not)

Methods inherited from class
`org.ldp.jdasm.attribute.Attributable`

(in A.3.2, page 136)

- *addAttribute*
  `public DAttribute` **addAttribute**`( org.ldp.jdasm.DAttribute` **attr** `)`
  - **Usage**
    * Adds a DAttribute.
  - **Returns** - the inserted DAttribute

- *attributeCount*
  `public int` **attributeCount**`( )`
  - **Usage**
    * Returns the number of attribute for this class

- *getAttribute*
  `public DAttribute` **getAttribute**`( int` **idx** `)`
  - **Usage**
    * Get the DAttribute at specified position

- *getAttribute*
  `public DAttribute` **getAttribute**`( java.lang.String` **name** `)`

  - **Usage**
    - Returns the DAttribute with specified name, or null if it doesn't exist.

- *getAttributes*
  `public DAttribute` **getAttributes**`( )`
  - **Usage**
    - Returns all the attributes

- *hasAttribute*
  `public boolean` **hasAttribute**`( java.lang.String` **name** `)`
  - **Usage**
    - Returns true if an attribute with given name exists.

- *removeAttribute*
  `public boolean` **removeAttribute**`( org.ldp.jdasm.DAttribute` **attr** `)`
  - **Usage**
    - Removes the specified DAttribute (match on equals())
  - **Returns** - true if the attribute is removed (if it was present), false it it is not present in the list.

- *removeAttribute*
  `public void` **removeAttribute**`( int` **idx** `)`
  - **Usage**
    - Removes the DAttribute at specified position

- *removeAttribute*
  `public boolean` **removeAttribute**`( java.lang.String` **name** `)`
  - **Usage**
    - Removes the DAttribute with specified name. Returns true if it is removed (if it was present), false if it is not present in the list.

Methods inherited from class `org.ldp.jdasm.constantpool.AConstantPoolUser`

(in A.2.4, page 97)

- *constantPoolUserChildSize*
  `public abstract int` **constantPoolUserChildSize**`( )`
  - **Usage**
    - Returns the number of IConstantPoolUser instances of this class.

- *constantPoolValueSize*
  `public abstract int` **constantPoolValueSize**`( )`
  - **Usage**
    - Returns the number of values that this constant pool user wants to insert into the constant pool.

- *getConstantPoolUserChild*
  `public abstract AConstantPoolUser` **getConstantPoolUserChild**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th IConstantPoolUser instance of this class.

- *getConstantValue*
  `public abstract Object` **getConstantValue**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th value that this constant pool user wants to insert into the constant pool.
  - **Exceptions**

    * `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *getConstantValueType*
  `public abstract int` **getConstantValueType**`( int` **idx** `)`
  - **Usage**
    * Returns the value type of the idx-th value that this constant pool user wants to insert into the constant pool. The list of the types is declared in DConstantPool
  - **Exceptions**
    * `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *setConstantValueIndex*
  `public abstract void` **setConstantValueIndex**`( int` **idx**`, short` **cpool_index** `)`
  - **Usage**
    * The constant pool tells the user that the value retrieved with AConstantPoolUser#getConstantValue(int) at index "idx" has received the "cpool_index" index in the constant pool.
  - **Parameters**
    * `idx` - the index of the value mapped with the ones returned by getConstantValue(int)
    * `cpool_index` - the index of the value in the constant pool

A.1.5 Class DMethod

This class represents a java class method. It can have an access permission qualifier (such as "private", "public", ...) a type, a name and a list of argument, each with its own type and name. It can also have a list of attributes.

The attributes that a standard jvm permits are: CodeAttribute , ExceptionsAttribute
The internal class do not store link with a *ConstantPool* until a build is done, so any change is fast and a link to a DClass or to a DConstantPool is not needed at construction time

Declaration

public class DMethod
extends org.ldp.jdasm.attribute.Attributable
implements org.ldp.jdasm.constantpool.IBuildable

Constructors

- *DMethod*
  `public` **DMethod**`( )`
  - **Usage**
    * Creates an empty DMethod.

- *DMethod*
  `public` **DMethod**`( java.io.InputStream` **is**`, org.ldp.jdasm.DConstantPool` **cpool** `)`
  - **Usage**

    * Creates a DMethod attribute reading it from an input stream whose cursor must point to the beginning of the method. The constant pool needs to have all constant values used by this method and its attribute.

- *DMethod*
  `public` **DMethod**`( short` **access_flags**`, java.lang.String` **returnType**`, java.lang.String` **name** `)`
  - **Usage**
    * Creates a DMethod with given arguments.

      For a list of argument you can call DMethod#addArgument(String)
  - **Parameters**
    * `access_flags` - (static, final, public, private, protected, ...)
    * `returnType` - the name of the class type (int, float, MySpecialObject[], ...)
    * `name` - the name of the field

- *DMethod*
  `public` **DMethod**`( java.lang.String` **code** `)`
  - **Usage**
    * Creates a DMethod from a java code string.

      It recognizes strings like: *<blockquote>"public int foo()"*

      "static private java.lang.String bar(int, float[])"</blockquote>
      Note: The special constructor method is called `<init>`

      Note: The initialization method of a class is called `<clinit>`

      Throws an CodeParseException when an invalid string is found

Methods

- *addArgument*
  `public void` **addArgument**`( java.lang.String` **code** `)`
  - **Usage**
    * Adds a single argument to the method from given code. It accepts a string of java code in the form like these:

      <blockquote>*"int foo"*

      "MyObject[] bar"</blockquote>It throws a CodeParseException when an error occurs while parsing the java code

---

- *addArgument*
  `public void` **addArgument**`( java.lang.String` **type**`, java.lang.String` **name** `)`
  - **Usage**
    * Adds a single argument of given type and name.

- *argumentCount*
  `public int` **argumentCount**`( )`
  - **Usage**
    - Returns the number of argument this method is going to accept.

- *build*
  `public int` **build**`( byte [] tofill, int` **atindex** `)`
  - **Usage**
    - It builds the content of this class into a final byte array. Such array is passed as first argument, and the index from where start to insert bytes is the second argument.
  - **Parameters**
    - `tofill` - the array to fill
    - `atindex` - the index from where start to fill
  - **Returns** - the number of byte inserted

- *build*
  `public int` **build**`( java.io.OutputStream` **output** `)`
  - **Usage**
    - It builds the content of this class into an output stream. This method is a faster version of int) since it doesn't need to call the IBuildable#getBuildLength() before.
  - **Parameters**
    - `output` - the stream to write onto
  - **Returns** - the number of byte inserted
  - **Exceptions**
    - `on` - error while writing onto the stream

- *classFileInfo*
  `public String` **classFileInfo**`( int` **indent_level** `)`
  - **Usage**
    - Returns a printable String with this DMethod information.
  - **Parameters**
    - `indent_level` - The indentation level (in white-spaces) for each generated line

---

- *constantPoolUserChildSize*
  `public int` **constantPoolUserChildSize**`( )`
  - **Usage**
    - Returns the number of IConstantPoolUser instances of this class.

- *constantPoolValueSize*
  `public int` **constantPoolValueSize**`( )`
  - **Usage**
    - Returns the number of values that this constant pool user wants to insert into the constant pool.

- *equals*
  `public boolean` **equals**`( java.lang.Object` **o** `)`
  - **Usage**

    - We match a DMethod that has the same signature, or a String if it is equals to the signature of the method.

- *getAccessFlagDescription*
  `public static String` **getAccessFlagDescription**`( short` **flag** `)`
  - **Usage**
    - Returns a printable String of the access permission qualifiers passed as argument.
  - **Parameters**
    - `flag` - The access permission qualifier typical of each DMethod class.

- *getAccessFlags*
  `public short` **getAccessFlags**`( )`
  - **Usage**
    - Get the access permission qualifier for this method.

- *getArgumentString*
  `public static String` **getArgumentString**`( java.util.Vector` **args** `)`
  - **Usage**
    - Returns a printable String for the argument of the DMethod.

- *getBuildLength*
  `public int` **getBuildLength**`( )`
  - **Usage**
    - Returns the length of portion of byte array the IBuildable class is going to create. If this class has instances of other IBuildable classes, then it will return the whole sum of all the various getBuildLength().

- *getCodeAttribute*
  `public CodeAttribute` **getCodeAttribute**`( )`
  - **Usage**
    - Returns the CodeAttribute associated to this method. If this method doesn't have any, it will be created before being returned.

- *getCodeAttribute*
  `public CodeAttribute` **getCodeAttribute**`( boolean` **create** `)`
  - **Usage**
    - Returns the CodeAttribute associated to this method. You can specify the behavior in case this method doesn't have the code attribute: if `create` is true a new code attribute is created and returned, if it is false then null is returned and no actions are done..

- *getConstantPoolUserChild*
  `public AConstantPoolUser` **getConstantPoolUserChild**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th IConstantPoolUser instance of this class.

- *getConstantValue*
  `public Object` **getConstantValue**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th value that this constant pool user wants to insert into the constant pool.
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *getConstantValueType*
  `public int` **getConstantValueType**`( int` **idx** `)`
  - **Usage**
    - Returns the value type of the idx-th value that this constant pool user wants to insert into the constant pool. The list of the types is declared in DConstantPool

- *getMethodDescriptor*
  `public String` **getMethodDescriptor**`( )`
  - **Usage**
    - Returns the MethodDescriptor for this DMethod as specified for the java class file.
  - **Exceptions**
    - `org.ldp.jdasm.exception.MethodDescriptorException` - if the specified types are not parsable as types
  - **See Also**
    - `org.ldp.jdasm.constantpool.TypeDescriptor.getFieldDescriptorByString(String)`

- *getName*
  `public String` **getName**`( )`
  - **Usage**
    - Get the name of the method.

- *getReturnType*
  `public String` **getReturnType**`( )`
  - **Usage**
    - Get the return type of the method. It's something like "int" "double[]" "mypackage.MyObject"

- *getSignature*
  `public String` **getSignature**`( )`
  - **Usage**
    - Returns a string signature representing this method univocally. It is in the form "{name}{descriptor}", something like "<init>()V"

- *isAccessFlagsSet*
  `public boolean` **isAccessFlagsSet**`( )`
  - **Usage**
    - Returns true if at least one access permission qualifier has been set.

- *isConstructor*
  `public boolean` **isConstructor**`( )`
  - **Usage**
    - Returns True if this method's name is <`init`>.
- *isNameSet*
  `public boolean` **isNameSet**`( )`
  - **Usage**
    - Returns true if the name of this method has been set.
- *isReturnTypeSet*
  `public boolean` **isReturnTypeSet**`( )`
  - **Usage**
    - Returns true if the return type of this method has been set.
- *removeArgument*
  `public void` **removeArgument**`( int` **idx** `)`
  - **Usage**
    - Removes argument at given index.
- *removeArgument*
  `public boolean` **removeArgument**`( java.lang.String` **code** `)`
  - **Usage**
    - Removes argument given as java code. Throws CodeParseException if an error occurs while parsing the code
  - **Returns** - true if argument is found in the list and removed, false if it is not present
  - **See Also**
    - `org.ldp.jdasm.DMethod.addArgument(String)`
- *removeArgument*
  `public boolean` **removeArgument**`( java.lang.String` **type**`, java.lang.String` **name** `)`
  - **Usage**
    - Removes argument of given type and name
  - **See Also**
    - `org.ldp.jdasm.DMethod.addArgument(String, String)`
- *setAccessFlags*
  `public void` **setAccessFlags**`( short` **accessFlags** `)`
  - **Usage**
    - Set the access permission qualifier for this method.
- *setConstantValueIndex*
  `public void` **setConstantValueIndex**`( int` **idx**`, short` **cpool_index** `)`
  - **Usage**
    - The constant pool tells the user that the value retrieved with AConstantPoolUser#getConstantValue(int) at index "idx" has received the "cpool_index" index in the constant pool.

- **Parameters**
  - `idx` - the index of the value mapped with the ones returned by getConstantValue(int)
  - `cpool_index` - the index of the value in the constant pool

- *setName*
  `public void` **setName**`( java.lang.String` **name** `)`
  - **Usage**
    - Set the name of the method.

- *setReturnType*
  `public void` **setReturnType**`( java.lang.String` **type** `)`
  - **Usage**
    - Set the return type of the method. If you use a class type then you need a fully qualifying name (java.lang.String is ok, but String is not)

Methods inherited from class
`org.ldp.jdasm.attribute.Attributable`

(in A.3.2, page 136)

- *addAttribute*
  `public DAttribute` **addAttribute**`( org.ldp.jdasm.DAttribute` **attr** `)`
  - **Usage**
    - Adds a DAttribute.
  - **Returns** - the inserted DAttribute

- *attributeCount*
  `public int` **attributeCount**`( )`
  - **Usage**
    - Returns the number of attribute for this class

- *getAttribute*
  `public DAttribute` **getAttribute**`( int` **idx** `)`
  - **Usage**
    - Get the DAttribute at specified position

- *getAttribute*
  `public DAttribute` **getAttribute**`( java.lang.String` **name** `)`
  - **Usage**
    - Returns the DAttribute with specified name, or null if it doesn't exist.

- *getAttributes*
  `public DAttribute` **getAttributes**`( )`
  - **Usage**
    - Returns all the attributes

- *hasAttribute*
  `public boolean` **hasAttribute**`( java.lang.String` **name** `)`
  - **Usage**
    - Returns true if an attribute with given name exists.

- *removeAttribute*
  `public boolean` **removeAttribute**`( org.ldp.jdasm.DAttribute` **attr** `)`
  - **Usage**

    - Removes the specified DAttribute (match on equals())
  - **Returns** - true if the attribute is removed (if it was present), false it it is not present in the list.

- *removeAttribute*
  `public void` **removeAttribute**`( int` **idx** `)`
  - **Usage**
    - Removes the DAttribute at specified position

- *removeAttribute*
  `public boolean` **removeAttribute**`( java.lang.String` **name** `)`
  - **Usage**
    - Removes the DAttribute with specified name. Returns true if it is removed (if it was present), false if it is not present in the list.

Methods inherited from class `org.ldp.jdasm.constantpool.AConstantPoolUser`

(in A.2.4, page 97)

- *constantPoolUserChildSize*
  `public abstract int` **constantPoolUserChildSize**`( )`
  - **Usage**
    - Returns the number of IConstantPoolUser instances of this class.

- *constantPoolValueSize*
  `public abstract int` **constantPoolValueSize**`( )`
  - **Usage**
    - Returns the number of values that this constant pool user wants to insert into the constant pool.

- *getConstantPoolUserChild*
  `public abstract AConstantPoolUser` **getConstantPoolUserChild**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th IConstantPoolUser instance of this class.

- *getConstantValue*
  `public abstract Object` **getConstantValue**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th value that this constant pool user wants to insert into the constant pool.
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *getConstantValueType*
  `public abstract int` **getConstantValueType**`( int` **idx** `)`
  - **Usage**
    - Returns the value type of the idx-th value that this constant pool user wants to insert into the constant pool. The list of the types is declared in DConstantPool
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *setConstantValueIndex*
  `public abstract void` **setConstantValueIndex**`( int` **idx**`, short` **cpool_index** `)`
  - **Usage**

    - The constant pool tells the user that the value retrieved with AConstantPoolUser#getConstantValue(int) at index "idx" has received the "cpool_index" index in the constant pool.
  - **Parameters**
    - `idx` - the index of the value mapped with the ones returned by getConstantValue(int)
    - `cpool_index` - the index of the value in the constant pool

A.1.6 Class DAttribute

This class is the super class of all the attributes. It contains the common methods for all the attributes, such manage the constant pool addressing of the name: since each attribute needs a name its constructor wants the name as argument.

Declaration

public abstract class DAttribute
extends org.ldp.jdasm.attribute.Attributable
implements org.ldp.jdasm.constantpool.IBuildable

Constructors

- *DAttribute*
  `public` **DAttribute**`( java.lang.String` **name** `)`
  - **Usage**
    - The constructor of the attribute. It needs the name of the attribute.

- *DAttribute*
  `public` **DAttribute**`( java.lang.String` **name**`, int` **length** `)`
  - **Usage**
    - The constructor of the attribute. You can specify the name and the length of the attribute, if it is a fixed-length attribute. The length is intended to be the length of the subclass attribute

Methods

- *build*
  `public int` **build**`( byte []` **tofill**`, int` **atindex** `)`
  - **Usage**
    - It builds the content of this class into a final byte array. Such array is passed as first argument, and the index from where start to insert bytes is the second argument.
  - **Parameters**
    - `tofill` - the array to fill
    - `atindex` - the index from where start to fill
  - **Returns** - the number of byte inserted

- *build*
  `public int` **build**`( java.io.OutputStream` **output** `)`

- **Usage**
  - It builds the content of this class into an output stream. This method is a faster version of int) since it doesn't need to call the IBuildable#getBuildLength() before.
- **Parameters**
  - `output` - the stream to write onto
- **Returns** - the number of byte inserted
- **Exceptions**
  - `on` - error while writing onto the stream

- *classFileInfo*
  `public String` **classFileInfo**`( int` **indent_level** `)`
  - **Usage**
    - Returns a printable info string

- *constantPoolUserChildSize*
  `public int` **constantPoolUserChildSize**`( )`
  - **Usage**
    - Returns the number of IConstantPoolUser instances of this class.

- *constantPoolValueSize*
  `public int` **constantPoolValueSize**`( )`
  - **Usage**
    - Returns the number of values that this constant pool user wants to insert into the constant pool.

- *getBuildLength*
  `public int` **getBuildLength**`( )`
  - **Usage**
    - Returns the length of portion of byte array the IBuildable class is going to create. If this class has instances of other IBuildable classes, then it will return the whole sum of all the various getBuildLength().

- *getConstantPoolUserChild*
  `public AConstantPoolUser` **getConstantPoolUserChild**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th IConstantPoolUser instance of this class.

- *getConstantValue*
  `public Object` **getConstantValue**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th value that this constant pool user wants to insert into the constant pool.
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *getConstantValueType*
  `public int` **getConstantValueType**`( int` **idx** `)`

- **Usage**
  - Returns the value type of the idx-th value that this constant pool user wants to insert into the constant pool. The list of the types is declared in DConstantPool
- **Exceptions**
  - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *getDAttributeFromInputStream*
  `public static DAttribute` **getDAttributeFromInputStream**`( java.io.InputStream` **is**`, org.ldp.jdasm.DConstantPool` **cpool** `)`

  - **Usage**
    - This function read an Attribute from an input stream whose cursor must point to the first byte (which specify the index into the constant pool of the name).

- *getDAttributeFromInputStream*
  `public static DAttribute` **getDAttributeFromInputStream**`( java.io.InputStream` **is**`, org.ldp.jdasm.DConstantPool` **cpool**`, org.ldp.jdasm.attribute.CodeAttribute` **code** `)`

  - **Usage**
    - This function read an Attribute from an input stream whose cursor must point to the first byte (which specify the index into the constant pool of the name). If the attribute's parent is a CodeAttribute, it must be specified as argument to allow the LineNumberTableAttribute to have link to instructions.

- *getName*
  `public String` **getName**`( )`

  - **Usage**
    - Get the name of the attribute.

- *setConstantValueIndex*
  `public void` **setConstantValueIndex**`( int` **idx**`, short` **cpool_index** `)`

  - **Usage**
    - The constant pool tells the user that the value retrieved with AConstantPoolUser#getConstantValue(int) at index "idx" has received the "cpool_index" index in the constant pool.
  - **Parameters**
    - `idx` - the index of the value mapped with the ones returned by getConstantValue(int)
    - `cpool_index` - the index of the value in the constant pool

- *setName*
  `public void` **setName**`( java.lang.String` **name** `)`

  - **Usage**
    - Set the name of the attribute.

Methods inherited from class
`org.ldp.jdasm.attribute.Attributable`

(in A.3.2, page 136)

- *addAttribute*
  `public DAttribute` **addAttribute**`( org.ldp.jdasm.DAttribute` **attr** `)`
  - **Usage**
    - Adds a DAttribute.
  - **Returns** - the inserted DAttribute

- *attributeCount*
  `public int` **attributeCount**`( )`
  - **Usage**
    - Returns the number of attribute for this class

- *getAttribute*
  `public DAttribute` **getAttribute**`( int` **idx** `)`
  - **Usage**
    - Get the DAttribute at specified position

- *getAttribute*
  `public DAttribute` **getAttribute**`( java.lang.String` **name** `)`
  - **Usage**
    - Returns the DAttribute with specified name, or null if it doesn't exist.

- *getAttributes*
  `public DAttribute` **getAttributes**`( )`
  - **Usage**
    - Returns all the attributes

- *hasAttribute*
  `public boolean` **hasAttribute**`( java.lang.String` **name** `)`
  - **Usage**
    - Returns true if an attribute with given name exists.

- *removeAttribute*
  `public boolean` **removeAttribute**`( org.ldp.jdasm.DAttribute` **attr** `)`
  - **Usage**
    - Removes the specified DAttribute (match on equals())
  - **Returns** - true if the attribute is removed (if it was present), false it it is not present in the list.

- *removeAttribute*
  `public void` **removeAttribute**`( int` **idx** `)`
  - **Usage**
    - Removes the DAttribute at specified position

- *removeAttribute*
  `public boolean` **removeAttribute**`( java.lang.String` **name** `)`
  - **Usage**
    - Removes the DAttribute with specified name. Returns true if it is removed (if it was present), false if it is not present in the list.

Methods inherited from class
`org.ldp.jdasm.constantpool.AConstantPoolUser`

(in A.2.4, page 97)

- *constantPoolUserChildSize*
  `public abstract int` **constantPoolUserChildSize**`( )`
  - **Usage**
    - Returns the number of IConstantPoolUser instances of this class.
- *constantPoolValueSize*
  `public abstract int` **constantPoolValueSize**`( )`
  - **Usage**
    - Returns the number of values that this constant pool user wants to insert into the constant pool.
- *getConstantPoolUserChild*
  `public abstract AConstantPoolUser` **getConstantPoolUserChild**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th IConstantPoolUser instance of this class.
- *getConstantValue*
  `public abstract Object` **getConstantValue**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th value that this constant pool user wants to insert into the constant pool.
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)
- *getConstantValueType*
  `public abstract int` **getConstantValueType**`( int` **idx** `)`
  - **Usage**
    - Returns the value type of the idx-th value that this constant pool user wants to insert into the constant pool. The list of the types is declared in DConstantPool
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)
- *setConstantValueIndex*
  `public abstract void` **setConstantValueIndex**`( int` **idx**`, short` **cpool_index** `)`
  - **Usage**
    - The constant pool tells the user that the value retrieved with AConstantPoolUser#getConstantValue(int) at index "idx" has received the "cpool_index" index in the constant pool.
  - **Parameters**
    - `idx` - the index of the value mapped with the ones returned by getConstantValue(int)
    - `cpool_index` - the index of the value in the constant pool

A.2 Package org.ldp.jdasm.constantpool

Interfaces

IBuildable 96

A common interface for any class that will be build to form the java class file.

Classes

AConstantPoolUser 97

A common interface for any class that uses the constant pool.

CONSTANT_Class_Info 98

The class representing a CONSTANT_Class_Info in the constant pool.

CONSTANT_Double_Info 101

The class representing a CONSTANT_Double_Info in the constant pool.

CONSTANT_FieldRef_Info 103

The class representing a CONSTANT_FieldRef_Info in the constant pool.

It refers to two other elements into the same constant pool; the first one is a CONSTANT_Class_Info representing the name of the class of the object, while the other is a constant of type CONSTANT_NameAndType_Info representing the name of the object and its descriptor.

CONSTANT_Float_Info 107

The class representing a CONSTANT_Float_Info in the constant pool.

CONSTANT_Integer_Info 109

The class representing a CONSTANT_Integer_Info in the constant pool.

CONSTANT_InterfaceMethodRef_Info 111

The class representing a CONSTANT_InterfaceMethodRef_Info in the constant pool.

It refers to two other elements into the same constant pool; the first one is a CONSTANT_Class_Info representing the name of the class of the object, while the other is a constant of type CONSTANT_NameAndType_Info representing the name of the object and its descriptor.

CONSTANT_Long_Info 115

The class representing a CONSTANT_Long_Info in the constant pool.

CONSTANT_MethodRef_Info 117

The class representing a CONSTANT_MethodRef_Info in the constant pool.

It refers to two other elements into the same constant pool; the first one is a CONSTANT_Class_Info representing the name of the class of the object, while the other is a constant of type CONSTANT_NameAndType_Info representing the name of the object and its descriptor.

CONSTANT_NameAndType_Info 121

The class representing a CONSTANT_NameAndType_Info in the constant pool.

It refers to two string into the same constant pool; the first one is a string representing the name of the object, while the other is a string representing the name of the type of the object.

CONSTANT_String_Info 125

The class representing a CONSTANT_String_Info in the constant pool.

CONSTANT_Utf8_Info 127

The class representing a CONSTANT_Utf8_Info in the constant pool.

CpInfo 129

CpInfo is the abstract superclass of all the constants pool classes.

TypeDescriptor 131

...no description...

A.2.1 Interfaces

A.2.2 Interface IBuildable

A common interface for any class that will be build to form the java class file. Almost any class that represents a structure in the java class file will known as build itself, and any of these will be asked to give the length in bytes of its building and to fill the final byte[] array through int)

Declaration

```
public interface IBuildable
```

Methods

- *build*
  `public int` **build**`( byte [] tofill, int  atindex )`
  - **Usage**
    - It builds the content of this class into a final byte array. Such array is passed as first argument, and the index from where start to insert bytes is the second argument.
  - **Parameters**
    - `tofill` - the array to fill
    - `atindex` - the index from where start to fill
  - **Returns** - the number of byte inserted

- *build*
  `public int` **build**`( java.io.OutputStream  output )`
  - **Usage**
    - It builds the content of this class into an output stream. This method is a faster version of int) since it doesn't need to call the IBuildable#getBuildLength() before.
  - **Parameters**
    - `output` - the stream to write onto
  - **Returns** - the number of byte inserted
  - **Exceptions**
    - `on` - error while writing onto the stream

- *getBuildLength*
  `public int` **getBuildLength**`( )`
  - **Usage**
    - Returns the length of portion of byte array the IBuildable class is going to create. If this class has instances of other IBuildable classes, then it will return the whole sum of all the various getBuildLength().

A.2.3 Classes

A.2.4 Class AConstantPoolUser

A common interface for any class that uses the constant pool. Almost any class that represents a class file structure uses the class file constant pool. For any variable that need a place in the constant pool the main class is asked to give its value in an Object (through AConstantPoolUser#getConstantValue(int)) and its type (through AConstantPoolUser#getConstantValueType(int)). After this, the constant pool will assign an index value to the main class through short) .

Since any AConstantPoolUser can have instances of classes that are AConstantPoolUser too, at build moment any class will be asked to give a reference to these AConstantPoolUser through AConstantPoolUser#getConstantPoolUserChild(int) making the constant pool filling recursive

Declaration

public abstract class AConstantPoolUser
extends java.lang.Object

Constructors

- *AConstantPoolUser*
  `public` **AConstantPoolUser**`( )`

Methods

- *constantPoolUserChildSize*
  `public abstract int` **constantPoolUserChildSize**`( )`
  - **Usage**
    - Returns the number of IConstantPoolUser instances of this class.

- *constantPoolValueSize*
  `public abstract int` **constantPoolValueSize**`( )`
  - **Usage**
    - Returns the number of values that this constant pool user wants to insert into the constant pool.

- *getConstantPoolUserChild*
  `public abstract AConstantPoolUser` **getConstantPoolUserChild**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th IConstantPoolUser instance of this class.

- *getConstantValue*
  `public abstract Object` **getConstantValue**`( int` **idx** `)`
  - **Usage**

    - Returns the idx-th value that this constant pool user wants to insert into the constant pool.
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *getConstantValueType*
  `public abstract int` **getConstantValueType**`( int` **idx** `)`
  - **Usage**
    - Returns the value type of the idx-th value that this constant pool user wants to insert into the constant pool. The list of the types is declared in DConstantPool
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *setConstantValueIndex*
  `public abstract void` **setConstantValueIndex**`( int` **idx**`, short` **cpool_index** `)`
  - **Usage**
    - The constant pool tells the user that the value retrieved with AConstantPoolUser#getConstantValue(int) at index "idx" has received the "cpool_index" index in the constant pool.
  - **Parameters**
    - `idx` - the index of the value mapped with the ones returned by getConstantValue(int)
    - `cpool_index` - the index of the value in the constant pool

A.2.5 Class CONSTANT_Class_Info

The class representing a CONSTANT_Class_Info in the constant pool. It refers to a string into the same constant pool.

Declaration

public class CONSTANT_Class_Info
extends org.ldp.jdasm.constantpool.CpInfo

Constructors

- *CONSTANT_Class_Info*
  `public` **CONSTANT_Class_Info**`( java.io.InputStream` **is** `)`
  - **Usage**
    - Creates a CONSTANT_Class_Info reading its content by an input stream. The cursor of the input stream must point to the beginning of the constant after the byte that specify the type. All the constant pool reference are saved (as short value).

- *CONSTANT_Class_Info*
  `public` **CONSTANT_Class_Info**`( java.lang.String` **className** `)`
  - **Usage**
    - Creates a CONSTANT_Class_Info with given class name.

Methods

- *addToConstantPool*
  `public short` **addToConstantPool**`( org.ldp.jdasm.DConstantPool` **pool** `)`
  - **Usage**
    - It overrides the main CpInfo#addToConstantPool(DConstantPool) because it needs to add a Utf8 string for the name

---

- *build*
  `public int` **build**`( byte []` **tofill**`, int` **atindex** `)`
  - **Usage**
    - It builds the content of this class into a final byte array. Such array is passed as first argument, and the index from where start to insert bytes is the second argument.
  - **Parameters**
    - `tofill` - the array to fill
    - `atindex` - the index from where start to fill
  - **Returns** - the number of byte inserted

- *build*
  `public int` **build**`( java.io.OutputStream` **output** `)`
  - **Usage**
    - It builds the content of this class into an output stream. This method is a faster version of int) since it doesn't need to call the IBuildable#getBuildLength() before.
  - **Parameters**
    - `output` - the stream to write onto
  - **Returns** - the number of byte inserted
  - **Exceptions**
    - `on` - error while writing onto the stream

- *copyValueFromReference*
  `public void` **copyValueFromReference**`(`
  `org.ldp.jdasm.DConstantPool` **pool** `)`
  - **Usage**
    - Converts the references to other constants in the pool from short index to a value (String).

- *equals*
  `public boolean` **equals**`( java.lang.Object` **o** `)`
  - **Usage**
    - We match when `o` has the same class name

- *getBuildLength*
  `public int` **getBuildLength**`( )`
  - **Usage**
    - Returns the length of portion of byte array the IBuildable class is going to create. If this class has instances of other IBuildable classes, then it will return the whole sum of all the various getBuildLength().

- *getClassName*
  `public String` **getClassName**`( )`
  - **Usage**
    - Get the class name associated to this CONSTANT_Class_Info.
- *getConstantPoolClassNameStringIndex*
  `public short` **getConstantPoolClassNameStringIndex**`( )`
  - **Usage**
    - Get the class name string index into the constant pool.
- *setClassName*
  `public void` **setClassName**`( java.lang.String` **className** `)`
  - **Usage**
    - Set the class name associated to this CONSTANT_Class_Info.
- *setConstantPoolClassNameStringIndex*
  `public void` **setConstantPoolClassNameStringIndex**`( short` **cpool_classNameString** `)`
  - **Usage**
    - Set the class name string index into the constant pool.
- *toString*
  `public String` **toString**`( )`
  - **Usage**
    - Returns a short description of constant values

Methods inherited from class `org.ldp.jdasm.constantpool.CpInfo`

(in A.2.16, page 129)

- *addToConstantPool*
  `public short` **addToConstantPool**`( org.ldp.jdasm.DConstantPool` **pool** `)`
  - **Usage**
    - Adds this constant to the constant pool. If a constant has references to other constants, then here it creates such constants classes and add them before adding itself. In such way, it can know the index of its references.
- *copyValueFromReference*
  `public void` **copyValueFromReference**`( org.ldp.jdasm.DConstantPool` **pool** `)`
  - **Usage**
    - Converts the references to other constants in the pool from short index to a value (String). When read the constant pool from a file, all the constants has a reference number but hasn't the real value referenced. All the constant classes that uses references will overload this method to copy the referenced constant value into its own value.
- *getTag*
  `public byte` **getTag**`( )`
  - **Usage**
    - Get the tag (the type) of this CpInfo.
  - **See Also**

    * `org.ldp.jdasm.DConstantPool` (in A.1.3, page 73)

- *hashCode*
  `public int` **hashCode**`( )`
  - **Usage**
    * Returns the hashcode according to inner value. This is used to hashmapping constants

A.2.6 Class CONSTANT_Double_Info

The class representing a CONSTANT_Double_Info in the constant pool. It holds a double constant value.

Declaration

public class CONSTANT_Double_Info
extends org.ldp.jdasm.constantpool.CpInfo

Constructors

- *CONSTANT_Double_Info*
  `public` **CONSTANT_Double_Info**`( double` **value** `)`
  - **Usage**
    * Creates a CONSTANT_Double_Info with given value.
- *CONSTANT_Double_Info*
  `public` **CONSTANT_Double_Info**`( java.lang.Double` **value** `)`
  - **Usage**
    * Creates a CONSTANT_Double_Info with given value.
- *CONSTANT_Double_Info*
  `public` **CONSTANT_Double_Info**`( java.io.InputStream` **is** `)`
  - **Usage**
    * Creates a CONSTANT_Double_Info reading its content by an input stream. The cursor of the input stream must point to the beginning of the constant after the byte that specify the type. The constant pool value are saved.

Methods

- *build*
  `public int` **build**`( byte []` **tofill**`, int` **atindex** `)`
  - **Usage**
    * It builds the content of this class into a final byte array. Such array is passed as first argument, and the index from where start to insert bytes is the second argument.
  - **Parameters**
    * `tofill` - the array to fill
    * `atindex` - the index from where start to fill
  - **Returns** - the number of byte inserted

- *build*
  `public int` **build**`( java.io.OutputStream` **output** `)`
  - **Usage**
    - It builds the content of this class into an output stream. This method is a faster version of int) since it doesn't need to call the IBuildable#getBuildLength() before.
  - **Parameters**
    - `output` - the stream to write onto
  - **Returns** - the number of byte inserted
  - **Exceptions**
    - `on` - error while writing onto the stream

- *equals*
  `public boolean` **equals**`( java.lang.Object` **o** `)`
  - **Usage**
    - We match when `o` has the same value

- *getBuildLength*
  `public int` **getBuildLength**`( )`
  - **Usage**
    - Returns the length of portion of byte array the IBuildable class is going to create. If this class has instances of other IBuildable classes, then it will return the whole sum of all the various getBuildLength().

- *getValue*
  `public double` **getValue**`( )`
  - **Usage**
    - Get the integer value associated to this CONSTANT_Double_Info

- *setValue*
  `public void` **setValue**`( double` **value** `)`
  - **Usage**
    - Set the integer value associated to this CONSTANT_Double_Info

- *toString*
  `public String` **toString**`( )`
  - **Usage**
    - Returns a short description of constant values

Methods inherited from class `org.ldp.jdasm.constantpool.CpInfo`

(in A.2.16, page 129)

- *addToConstantPool*
  `public short` **addToConstantPool**`( org.ldp.jdasm.DConstantPool` **pool** `)`
  - **Usage**

    - Adds this constant to the constant pool. If a constant has references to other constants, then here it creates such constants classes and add them before adding itself. In such way, it can know the index of its references.

- *copyValueFromReference*
  `public void` **copyValueFromReference**`( org.ldp.jdasm.DConstantPool` **pool** `)`
  - **Usage**
    - Converts the references to other constants in the pool from short index to a value (String). When read the constant pool from a file, all the constants has a reference number but hasn't the real value referenced. All the constant classes that uses references will overload this method to copy the referenced constant value into its own value.

- *getTag*
  `public byte` **getTag**`( )`
  - **Usage**
    - Get the tag (the type) of this CpInfo.
  - **See Also**
    - `org.ldp.jdasm.DConstantPool` (in A.1.3, page 73)

- *hashCode*
  `public int` **hashCode**`( )`
  - **Usage**
    - Returns the hashcode according to inner value. This is used to hashmapping constants

A.2.7 Class CONSTANT_FieldRef_Info

The class representing a CONSTANT_FieldRef_Info in the constant pool.

It refers to two other elements into the same constant pool; the first one is a CONSTANT_Class_Info representing the name of the class of the object, while the other is a constant of type CONSTANT_NameAndType_Info representing the name of the object and its descriptor.

Declaration

public class CONSTANT_FieldRef_Info
extends org.ldp.jdasm.constantpool.CpInfo

Constructors

- *CONSTANT_FieldRef_Info*
  `public` **CONSTANT_FieldRef_Info**`( java.io.InputStream` **is** `)`
  - **Usage**
    - Creates a CONSTANT_FieldRef_Info reading its content by an input stream. The cursor of the input stream must point to the beginning of the constant after the byte that specify the type. All the constant pool reference are saved (as short value).

- *CONSTANT_FieldRef_Info*
  `public` **CONSTANT_FieldRef_Info**`( java.lang.String` **value** `)`

- **Usage**
  - Creates a CONSTANT_FieldRef_Info from a string in the form "CLASSNAME NAME DESCRIPTOR".
- **See Also**
  - `org.ldp.jdasm.constantpool.TypeDescriptor` (in A.2.17, page 131)

- *CONSTANT_FieldRef_Info*
  `public` **CONSTANT_FieldRef_Info**`( java.lang.String` **className**`, java.lang.String` **name**`, java.lang.String` **descriptor** `)`
  - **Usage**
    - Creates a CONSTANT_FieldRef_Info with given class name, name and descriptor

Methods

- *addToConstantPool*
  `public short` **addToConstantPool**`( org.ldp.jdasm.DConstantPool` **pool** `)`
  - **Usage**
    - It overrides the main CpInfo#addToConstantPool(DConstantPool) because it needs to add two other constants. These constants are a CONSTANT_Class_Info and a CONSTANT_NameAndType_Info .

- *build*
  `public int` **build**`( byte []` **tofill**`, int` **atindex** `)`
  - **Usage**
    - It builds the content of this class into a final byte array. Such array is passed as first argument, and the index from where start to insert bytes is the second argument.
  - **Parameters**
    - `tofill` - the array to fill
    - `atindex` - the index from where start to fill
  - **Returns** - the number of byte inserted

- *build*
  `public int` **build**`( java.io.OutputStream` **output** `)`
  - **Usage**
    - It builds the content of this class into an output stream. This method is a faster version of int) since it doesn't need to call the IBuildable#getBuildLength() before.
  - **Parameters**
    - `output` - the stream to write onto
  - **Returns** - the number of byte inserted
  - **Exceptions**
    - `on` - error while writing onto the stream

- *copyValueFromReference*
  `public void` **copyValueFromReference**`(`
  `org.ldp.jdasm.DConstantPool` **pool** `)`
  - **Usage**
    * Converts the references to other constants in the pool from short index to a value (String).

- *equals*
  `public boolean` **equals**`( java.lang.Object` **o** `)`
  - **Usage**
    * We match when `o` has the same class name, name and descriptor

---

- *getBuildLength*
  `public int` **getBuildLength**`( )`
  - **Usage**
    * Returns the length of portion of byte array the IBuildable class is going to create. If this class has instances of other IBuildable classes, then it will return the whole sum of all the various getBuildLength().

- *getClassName*
  `public String` **getClassName**`( )`
  - **Usage**
    * Get the class name associated to this CONSTANT_FieldRef_Info.

- *getConstantPoolClassNameStringIndex*
  `public short` **getConstantPoolClassNameStringIndex**`( )`
  - **Usage**
    * Get the class name index of the constant pool.

- *getConstantPoolNameAndTypeIndex*
  `public short` **getConstantPoolNameAndTypeIndex**`( )`
  - **Usage**
    * Get the name and type index of the constant pool.

- *getDescriptor*
  `public String` **getDescriptor**`( )`
  - **Usage**
    * Get the descriptor associated to this CONSTANT_FieldRef_Info.

- *getName*
  `public String` **getName**`( )`
  - **Usage**
    * Get the name associated to this CONSTANT_FieldRef_Info.

- *setClassName*
  `public void` **setClassName**`( java.lang.String` **className** `)`
  - **Usage**

    - Set the class name associated to this CONSTANT_FieldRef_Info.

- *setConstantPoolClassNameIndex*
  `public void` **setConstantPoolClassNameIndex**`( short` **cpool_classNameIndex** `)`
  - **Usage**
    - Set the class name index of the constant pool.

- *setConstantPoolNameAndTypeIndex*
  `public void` **setConstantPoolNameAndTypeIndex**`( short` **cpool_nameAndTypeIndex** `)`
  - **Usage**
    - Set the name and type index of the constant pool.

- *setDescriptor*
  `public void` **setDescriptor**`( java.lang.String` **descriptor** `)`
  - **Usage**
    - Set the descriptor associated to this CONSTANT_FieldRef_Info.

- *setName*
  `public void` **setName**`( java.lang.String` **name** `)`
  - **Usage**
    - Set the name associated to this CONSTANT_FieldRef_Info.

- *toString*
  `public String` **toString**`( )`
  - **Usage**
    - Returns a short description of constant values

Methods inherited from class `org.ldp.jdasm.constantpool.CpInfo`

(in A.2.16, page 129)

- *addToConstantPool*
  `public short` **addToConstantPool**`( org.ldp.jdasm.DConstantPool` **pool** `)`
  - **Usage**
    - Adds this constant to the constant pool. If a constant has references to other constants, then here it creates such constants classes and add them before adding itself. In such way, it can know the index of its references.

- *copyValueFromReference*
  `public void` **copyValueFromReference**`( org.ldp.jdasm.DConstantPool` **pool** `)`
  - **Usage**
    - Converts the references to other constants in the pool from short index to a value (String). When read the constant pool from a file, all the constants has a reference number but hasn't the real value referenced. All the constant classes that uses references will overload this method to copy the referenced constant value into its own value.

- *getTag*
  `public byte` **getTag**`( )`

- **Usage**
  - Get the tag (the type) of this CpInfo.
- **See Also**
  - `org.ldp.jdasm.DConstantPool` (in A.1.3, page 73)

- *hashCode*
  `public int` **hashCode**`( )`
  - **Usage**
    - Returns the hashcode according to inner value. This is used to hashmapping constants

A.2.8 Class CONSTANT_Float_Info

The class representing a CONSTANT_Float_Info in the constant pool. It holds a float constant value.

Declaration

public class CONSTANT_Float_Info
extends org.ldp.jdasm.constantpool.CpInfo

Constructors

- *CONSTANT_Float_Info*
  `public` **CONSTANT_Float_Info**`( float` **value** `)`
  - **Usage**
    - Creates a CONSTANT_Float_Info with given value.

- *CONSTANT_Float_Info*
  `public` **CONSTANT_Float_Info**`( java.lang.Float` **value** `)`
  - **Usage**
    - Creates a CONSTANT_Float_Info with given value.

- *CONSTANT_Float_Info*
  `public` **CONSTANT_Float_Info**`( java.io.InputStream` **is** `)`
  - **Usage**
    - Creates a CONSTANT_Float_Info reading its content by an input stream. The cursor of the input stream must point to the beginning of the constant after the byte that specify the type. The constant pool value are saved.

Methods

- *build*
  `public int` **build**`( byte [] tofill, int` **atindex** `)`
  - **Usage**
    - It builds the content of this class into a final byte array. Such array is passed as first argument, and the index from where start to insert bytes is the second argument.

- **Parameters**
  - `tofill` - the array to fill
  - `atindex` - the index from where start to fill
- **Returns** - the number of byte inserted

- *build*
  `public int` **build**`( java.io.OutputStream` **output** `)`
  - **Usage**
    - It builds the content of this class into an output stream. This method is a faster version of int) since it doesn't need to call the IBuildable#getBuildLength() before.
  - **Parameters**
    - `output` - the stream to write onto
  - **Returns** - the number of byte inserted
  - **Exceptions**
    - `on` - error while writing onto the stream

- *equals*
  `public boolean` **equals**`( java.lang.Object` **o** `)`
  - **Usage**
    - We match when `o` has the same value

- *getBuildLength*
  `public int` **getBuildLength**`( )`
  - **Usage**
    - Returns the length of portion of byte array the IBuildable class is going to create. If this class has instances of other IBuildable classes, then it will return the whole sum of all the various getBuildLength().

- *getValue*
  `public float` **getValue**`( )`
  - **Usage**
    - Get the integer value associated to this CONSTANT_Float_Info

---

- *setValue*
  `public void` **setValue**`( float` **value** `)`
  - **Usage**
    - Set the integer value associated to this CONSTANT_Float_Info

---

- *toString*
  `public String` **toString**`( )`
  - **Usage**
    - Returns a short description of constant values

METHODS INHERITED FROM CLASS `org.ldp.jdasm.constantpool.CpInfo`

(in A.2.16, page 129)

- *addToConstantPool*
  `public short` **addToConstantPool**`( org.ldp.jdasm.DConstantPool` **pool** `)`
  - **Usage**
    - Adds this constant to the constant pool. If a constant has references to other constants, then here it creates such constants classes and add them before adding itself. In such way, it can know the index of its references.
- *copyValueFromReference*
  `public void` **copyValueFromReference**`( org.ldp.jdasm.DConstantPool` **pool** `)`
  - **Usage**
    - Converts the references to other constants in the pool from short index to a value (String). When read the constant pool from a file, all the constants has a reference number but hasn't the real value referenced. All the constant classes that uses references will overload this method to copy the referenced constant value into its own value.
- *getTag*
  `public byte` **getTag**`( )`
  - **Usage**
    - Get the tag (the type) of this CpInfo.
  - **See Also**
    - `org.ldp.jdasm.DConstantPool` (in A.1.3, page 73)
- *hashCode*
  `public int` **hashCode**`( )`
  - **Usage**
    - Returns the hashcode according to inner value. This is used to hashmapping constants

A.2.9 CLASS CONSTANT_Integer_Info

The class representing a CONSTANT_Integer_Info in the constant pool. It holds a int constant value.

DECLARATION

public class CONSTANT_Integer_Info
extends org.ldp.jdasm.constantpool.CpInfo

CONSTRUCTORS

- *CONSTANT_Integer_Info*
  `public` **CONSTANT_Integer_Info**`( java.io.InputStream` **is** `)`
  - **Usage**
    - Creates a CONSTANT_Integer_Info reading its content by an input stream. The cursor of the input stream must point to the beginning of the constant after the byte that specify the type. The constant pool value are saved.

- *CONSTANT_Integer_Info*
  `public` **CONSTANT_Integer_Info**`( int` **value** `)`
  - **Usage**
    - Creates a CONSTANT_Integer_Info with given value.
- *CONSTANT_Integer_Info*
  `public` **CONSTANT_Integer_Info**`( java.lang.Integer` **value** `)`
  - **Usage**
    - Creates a CONSTANT_Integer_Info with given value.

Methods

- *build*
  `public int` **build**`( byte []` **tofill**`, int` **atindex** `)`
  - **Usage**
    - It builds the content of this class into a final byte array. Such array is passed as first argument, and the index from where start to insert bytes is the second argument.
  - **Parameters**
    - `tofill` - the array to fill
    - `atindex` - the index from where start to fill
  - **Returns** - the number of byte inserted
- *build*
  `public int` **build**`( java.io.OutputStream` **output** `)`
  - **Usage**
    - It builds the content of this class into an output stream. This method is a faster version of int) since it doesn't need to call the IBuildable#getBuildLength() before.
  - **Parameters**
    - `output` - the stream to write onto
  - **Returns** - the number of byte inserted
  - **Exceptions**
    - `on` - error while writing onto the stream
- *equals*
  `public boolean` **equals**`( java.lang.Object` **o** `)`
  - **Usage**
    - We match when `o` has the same value
- *getBuildLength*
  `public int` **getBuildLength**`( )`
  - **Usage**
    - Returns the length of portion of byte array the IBuildable class is going to create. If this class has instances of other IBuildable classes, then it will return the whole sum of all the various getBuildLength().
- *getValue*
  `public int` **getValue**`( )`

  - **Usage**
    * Get the integer value associated to this CONSTANT_Integer_Info

- *setValue*
  `public void` **setValue**`( int` **value** `)`
  - **Usage**
    * Set the integer value associated to this CONSTANT_Integer_Info

- *toString*
  `public String` **toString**`( )`
  - **Usage**
    * Returns a short description of constant values

METHODS INHERITED FROM CLASS `org.ldp.jdasm.constantpool.CpInfo`

(in A.2.16, page 129)

- *addToConstantPool*
  `public short` **addToConstantPool**`( org.ldp.jdasm.DConstantPool` **pool** `)`
  - **Usage**
    * Adds this constant to the constant pool. If a constant has references to other constants, then here it creates such constants classes and add them before adding itself. In such way, it can know the index of its references.

- *copyValueFromReference*
  `public void` **copyValueFromReference**`( org.ldp.jdasm.DConstantPool` **pool** `)`
  - **Usage**
    * Converts the references to other constants in the pool from short index to a value (String). When read the constant pool from a file, all the constants has a reference number but hasn't the real value referenced. All the constant classes that uses references will overload this method to copy the referenced constant value into its own value.

- *getTag*
  `public byte` **getTag**`( )`
  - **Usage**
    * Get the tag (the type) of this CpInfo.
  - **See Also**
    * `org.ldp.jdasm.DConstantPool` (in A.1.3, page 73)

- *hashCode*
  `public int` **hashCode**`( )`
  - **Usage**
    * Returns the hashcode according to inner value. This is used to hashmapping constants

A.2.10 CLASS CONSTANT_InterfaceMethodRef_Info

The class representing a CONSTANT_InterfaceMethodRef_Info in the constant pool.

It refers to two other elements into the same constant pool; the first one is a CONSTANT_Class_Info representing the name of the class of the object, while the other is a constant of type CONSTANT_NameAndType_Info representing the name of the object and its descriptor.

Declaration

public class CONSTANT_InterfaceMethodRef_Info
extends org.ldp.jdasm.constantpool.CpInfo

Constructors

- *CONSTANT_InterfaceMethodRef_Info*
  `public` **CONSTANT_InterfaceMethodRef_Info(** `java.io.InputStream` **is**)
  - **Usage**
    - Creates a CONSTANT_InterfaceMethodRef_Info reading its content by an input stream. The cursor of the input stream must point to the beginning of the constant after the byte that specify the type. All the constant pool reference are saved (as short value).
- *CONSTANT_InterfaceMethodRef_Info*
  `public` **CONSTANT_InterfaceMethodRef_Info(** `java.lang.String` **value**)
  - **Usage**
    - Creates a CONSTANT_InterfaceMethodRef_Info from a string in the form "CLASSNAME NAME DESCRIPTOR".
  - **See Also**
    - `org.ldp.jdasm.constantpool.TypeDescriptor` (in A.2.17, page 131)
- *CONSTANT_InterfaceMethodRef_Info*
  `public` **CONSTANT_InterfaceMethodRef_Info(** `java.lang.String` **className**, `java.lang.String` **name**, `java.lang.String` **descriptor**)
  - **Usage**
    - Creates a CONSTANT_InterfaceMethodRef_Info with given class name, name and descriptor

Methods

- *addToConstantPool*
  `public short` **addToConstantPool(** `org.ldp.jdasm.DConstantPool` **pool**)
  - **Usage**
    - It overrides the main CpInfo#addToConstantPool(DConstantPool) because it needs to add two other constants. These constants are a CONSTANT_Class_Info and a CONSTANT_NameAndType_Info .
- *build*
  `public int` **build(** `byte []` **tofill**, `int` **atindex**)
  - **Usage**

    * It builds the content of this class into a final byte array. Such array is passed as first argument, and the index from where start to insert bytes is the second argument.
  - **Parameters**
    * `tofill` - the array to fill
    * `atindex` - the index from where start to fill
  - **Returns** - the number of byte inserted

- *build*
  `public int` **build**`( java.io.OutputStream` **output** `)`
  - **Usage**
    * It builds the content of this class into an output stream. This method is a faster version of int) since it doesn't need to call the IBuildable#getBuildLength() before.
  - **Parameters**
    * `output` - the stream to write onto
  - **Returns** - the number of byte inserted
  - **Exceptions**
    * `on` - error while writing onto the stream

- *copyValueFromReference*
  `public void` **copyValueFromReference**`(`
  `org.ldp.jdasm.DConstantPool` **pool** `)`
  - **Usage**
    * Converts the references to other constants in the pool from short index to a value (String).

- *equals*
  `public boolean` **equals**`( java.lang.Object` **o** `)`
  - **Usage**
    * We match when `o` has the same class name, name and descriptor

---

- *getBuildLength*
  `public int` **getBuildLength**`( )`
  - **Usage**
    * Returns the length of portion of byte array the IBuildable class is going to create. If this class has instances of other IBuildable classes, then it will return the whole sum of all the various getBuildLength().

- *getClassName*
  `public String` **getClassName**`( )`
  - **Usage**
    * Get the class name associated to this CONSTANT_InterfaceMethodRef_Info.

- *getConstantPoolClassNameStringIndex*
  `public short` **getConstantPoolClassNameStringIndex**`( )`
  - **Usage**
    * Get the class name index of the constant pool.

- *getConstantPoolNameAndTypeIndex*
  `public short` **getConstantPoolNameAndTypeIndex**`( )`
  - **Usage**
    - Get the name and type index of the constant pool.

- *getDescriptor*
  `public String` **getDescriptor**`( )`
  - **Usage**
    - Get the descriptor associated to this CONSTANT_InterfaceMethodRef_Info.

- *getName*
  `public String` **getName**`( )`
  - **Usage**
    - Get the name associated to this CONSTANT_InterfaceMethodRef_Info.

- *setClassName*
  `public void` **setClassName**`( java.lang.String` **className** `)`
  - **Usage**
    - Set the class name associated to this CONSTANT_InterfaceMethodRef_Info.

- *setConstantPoolClassNameIndex*
  `public void` **setConstantPoolClassNameIndex**`( short` **cpool_classNameIndex** `)`
  - **Usage**
    - Set the class name index of the constant pool.

- *setConstantPoolNameAndTypeIndex*
  `public void` **setConstantPoolNameAndTypeIndex**`( short` **cpool_nameAndTypeIndex** `)`
  - **Usage**
    - Set the name and type index of the constant pool.

- *setDescriptor*
  `public void` **setDescriptor**`( java.lang.String` **descriptor** `)`
  - **Usage**
    - Set the descriptor associated to this CONSTANT_InterfaceMethodRef_Info.

- *setName*
  `public void` **setName**`( java.lang.String` **name** `)`
  - **Usage**
    - Set the name associated to this CONSTANT_InterfaceMethodRef_Info.

- *toString*
  `public String` **toString**`( )`
  - **Usage**
    - Returns a short description of constant values

METHODS INHERITED FROM CLASS `org.ldp.jdasm.constantpool.CpInfo`

(in A.2.16, page 129)

- *addToConstantPool*
  `public short` **addToConstantPool**`( org.ldp.jdasm.DConstantPool` **pool** `)`
  - **Usage**
    - Adds this constant to the constant pool. If a constant has references to other constants, then here it creates such constants classes and add them before adding itself. In such way, it can know the index of its references.
- *copyValueFromReference*
  `public void` **copyValueFromReference**`( org.ldp.jdasm.DConstantPool` **pool** `)`
  - **Usage**
    - Converts the references to other constants in the pool from short index to a value (String). When read the constant pool from a file, all the constants has a reference number but hasn't the real value referenced. All the constant classes that uses references will overload this method to copy the referenced constant value into its own value.
- *getTag*
  `public byte` **getTag**`( )`
  - **Usage**
    - Get the tag (the type) of this CpInfo.
  - **See Also**
    - `org.ldp.jdasm.DConstantPool` (in A.1.3, page 73)
- *hashCode*
  `public int` **hashCode**`( )`
  - **Usage**
    - Returns the hashcode according to inner value. This is used to hashmapping constants

A.2.11 CLASS CONSTANT_Long_Info

The class representing a CONSTANT_Long_Info in the constant pool. It holds a long constant value.

DECLARATION

public class CONSTANT_Long_Info
extends org.ldp.jdasm.constantpool.CpInfo

CONSTRUCTORS

- *CONSTANT_Long_Info*
  `public` **CONSTANT_Long_Info**`( java.io.InputStream` **is** `)`
  - **Usage**
    - Creates a CONSTANT_Long_Info reading its content by an input stream. The cursor of the input stream must point to the beginning of the constant after the byte that specify the type. The constant pool value are saved.

- *CONSTANT_Long_Info*
  `public` **CONSTANT_Long_Info**`( long` **value** `)`
  - **Usage**
    * Creates a CONSTANT_Long_Info with given value.

- *CONSTANT_Long_Info*
  `public` **CONSTANT_Long_Info**`( java.lang.Long` **value** `)`
  - **Usage**
    * Creates a CONSTANT_Long_Info with given value.

Methods

- *build*
  `public int` **build**`( byte []` **tofill**`, int` **atindex** `)`
  - **Usage**
    * It builds the content of this class into a final byte array. Such array is passed as first argument, and the index from where start to insert bytes is the second argument.
  - **Parameters**
    * `tofill` - the array to fill
    * `atindex` - the index from where start to fill
  - **Returns** - the number of byte inserted

- *build*
  `public int` **build**`( java.io.OutputStream` **output** `)`
  - **Usage**
    * It builds the content of this class into an output stream. This method is a faster version of int) since it doesn't need to call the IBuildable#getBuildLength() before.
  - **Parameters**
    * `output` - the stream to write onto
  - **Returns** - the number of byte inserted
  - **Exceptions**
    * `on` - error while writing onto the stream

- *equals*
  `public boolean` **equals**`( java.lang.Object` **o** `)`
  - **Usage**
    * We match when `o` has the same value

- *getBuildLength*
  `public int` **getBuildLength**`( )`
  - **Usage**
    * Returns the length of portion of byte array the IBuildable class is going to create. If this class has instances of other IBuildable classes, then it will return the whole sum of all the various getBuildLength().

- *getValue*
  `public long` **getValue**`( )`

  - **Usage**
    * Get the integer value associated to this CONSTANT_Long_Info

- *setValue*
  `public void` **setValue**`( long` **value** `)`
  - **Usage**
    * Set the integer value associated to this CONSTANT_Long_Info

- *toString*
  `public String` **toString**`( )`
  - **Usage**
    * Returns a short description of constant values

METHODS INHERITED FROM CLASS `org.ldp.jdasm.constantpool.CpInfo`

(in A.2.16, page 129)

- *addToConstantPool*
  `public short` **addToConstantPool**`( org.ldp.jdasm.DConstantPool` **pool** `)`
  - **Usage**
    * Adds this constant to the constant pool. If a constant has references to other constants, then here it creates such constants classes and add them before adding itself. In such way, it can know the index of its references.

- *copyValueFromReference*
  `public void` **copyValueFromReference**`( org.ldp.jdasm.DConstantPool` **pool** `)`
  - **Usage**
    * Converts the references to other constants in the pool from short index to a value (String). When read the constant pool from a file, all the constants has a reference number but hasn't the real value referenced. All the constant classes that uses references will overload this method to copy the referenced constant value into its own value.

- *getTag*
  `public byte` **getTag**`( )`
  - **Usage**
    * Get the tag (the type) of this CpInfo.
  - **See Also**
    * `org.ldp.jdasm.DConstantPool` (in A.1.3, page 73)

- *hashCode*
  `public int` **hashCode**`( )`
  - **Usage**
    * Returns the hashcode according to inner value. This is used to hashmapping constants

A.2.12 CLASS CONSTANT_MethodRef_Info

The class representing a CONSTANT_MethodRef_Info in the constant pool.

It refers to two other elements into the same constant pool; the first one is a CONSTANT_Class_Info representing the name of the class of the object, while the other is a constant of type CONSTANT_NameAndType_Info representing the name of the object and its descriptor.

Declaration

public class CONSTANT_MethodRef_Info
extends org.ldp.jdasm.constantpool.CpInfo

Constructors

- *CONSTANT_MethodRef_Info*
  `public` **CONSTANT_MethodRef_Info**`( java.io.InputStream` **is** `)`
  - **Usage**
    - Creates a CONSTANT_MethodRef_Info reading its content by an input stream. The cursor of the input stream must point to the beginning of the constant after the byte that specify the type. All the constant pool reference are saved (as short value).

- *CONSTANT_MethodRef_Info*
  `public` **CONSTANT_MethodRef_Info**`( java.lang.String` **value** `)`
  - **Usage**
    - Creates a CONSTANT_MethodRef_Info from a string in the form "CLASSNAME NAME DESCRIPTOR"
  - **See Also**
    - `org.ldp.jdasm.constantpool.TypeDescriptor` (in A.2.17, page 131)

- *CONSTANT_MethodRef_Info*
  `public` **CONSTANT_MethodRef_Info**`( java.lang.String` **className**`, java.lang.String` **name**`, java.lang.String` **descriptor** `)`
  - **Usage**
    - Creates a CONSTANT_MethodRef_Info with given class name, name and descriptor

Methods

- *addToConstantPool*
  `public short` **addToConstantPool**`( org.ldp.jdasm.DConstantPool` **pool** `)`
  - **Usage**
    - It overrides the main CpInfo#addToConstantPool(DConstantPool) because it needs to add two other constants. These constants are a CONSTANT_Class_Info and a CONSTANT_NameAndType_Info .

- *build*
  `public int` **build**`( byte [] tofill, int` **atindex** `)`
  - **Usage**

    - It builds the content of this class into a final byte array. Such array is passed as first argument, and the index from where start to insert bytes is the second argument.
  - **Parameters**
    - `tofill` - the array to fill
    - `atindex` - the index from where start to fill
  - **Returns** - the number of byte inserted

- *build*
  `public int` **build**`( java.io.OutputStream` **output** `)`
  - **Usage**
    - It builds the content of this class into an output stream. This method is a faster version of int) since it doesn't need to call the IBuildable#getBuildLength() before.
  - **Parameters**
    - `output` - the stream to write onto
  - **Returns** - the number of byte inserted
  - **Exceptions**
    - `on` - error while writing onto the stream

- *copyValueFromReference*
  `public void` **copyValueFromReference**`(`
  `org.ldp.jdasm.DConstantPool` **pool** `)`
  - **Usage**
    - Converts the references to other constants in the pool from short index to a value (String).

- *equals*
  `public boolean` **equals**`( java.lang.Object` **o** `)`
  - **Usage**
    - We match when `o` has the same class name, name and descriptor

---

- *getBuildLength*
  `public int` **getBuildLength**`( )`
  - **Usage**
    - Returns the length of portion of byte array the IBuildable class is going to create. If this class has instances of other IBuildable classes, then it will return the whole sum of all the various getBuildLength().

- *getClassName*
  `public String` **getClassName**`( )`
  - **Usage**
    - Get the class name associated to this CONSTANT_MethodRef_Info.

- *getConstantPoolClassNameStringIndex*
  `public short` **getConstantPoolClassNameStringIndex**`( )`
  - **Usage**
    - Get the class name index of the constant pool.

- *getConstantPoolNameAndTypeIndex*
  `public short` **getConstantPoolNameAndTypeIndex**`( )`
  - **Usage**
    - Get the name and type index of the constant pool.

- *getDescriptor*
  `public String` **getDescriptor**`( )`
  - **Usage**
    - Get the descriptor associated to this CONSTANT_MethodRef_Info.

- *getName*
  `public String` **getName**`( )`
  - **Usage**
    - Get the name associated to this CONSTANT_MethodRef_Info.

- *setClassName*
  `public void` **setClassName**`( java.lang.String` **className** `)`
  - **Usage**
    - Set the class name associated to this CONSTANT_MethodRef_Info.

- *setConstantPoolClassNameIndex*
  `public void` **setConstantPoolClassNameIndex**`( short` **cpool_classNameIndex** `)`
  - **Usage**
    - Set the class name index of the constant pool.

- *setConstantPoolNameAndTypeIndex*
  `public void` **setConstantPoolNameAndTypeIndex**`( short` **cpool_nameAndTypeIndex** `)`
  - **Usage**
    - Set the name and type index of the constant pool.

- *setDescriptor*
  `public void` **setDescriptor**`( java.lang.String` **descriptor** `)`
  - **Usage**
    - Set the descriptor associated to this CONSTANT_MethodRef_Info.

- *setName*
  `public void` **setName**`( java.lang.String` **name** `)`
  - **Usage**
    - Set the name associated to this CONSTANT_MethodRef_Info.

- *toString*
  `public String` **toString**`( )`
  - **Usage**
    - Returns a short description of constant values

METHODS INHERITED FROM CLASS `org.ldp.jdasm.constantpool.CpInfo`

(in A.2.16, page 129)

- *addToConstantPool*
  `public short` **addToConstantPool**`( org.ldp.jdasm.DConstantPool` **pool** `)`
  - **Usage**
    * Adds this constant to the constant pool. If a constant has references to other constants, then here it creates such constants classes and add them before adding itself. In such way, it can know the index of its references.
- *copyValueFromReference*
  `public void` **copyValueFromReference**`( org.ldp.jdasm.DConstantPool` **pool** `)`
  - **Usage**
    * Converts the references to other constants in the pool from short index to a value (String). When read the constant pool from a file, all the constants has a reference number but hasn't the real value referenced. All the constant classes that uses references will overload this method to copy the referenced constant value into its own value.
- *getTag*
  `public byte` **getTag**`( )`
  - **Usage**
    * Get the tag (the type) of this CpInfo.
  - **See Also**
    * `org.ldp.jdasm.DConstantPool` (in A.1.3, page 73)
- *hashCode*
  `public int` **hashCode**`( )`
  - **Usage**
    * Returns the hashcode according to inner value. This is used to hashmapping constants

A.2.13 CLASS CONSTANT_NameAndType_Info

The class representing a CONSTANT_NameAndType_Info in the constant pool.

It refers to two string into the same constant pool; the first one is a string representing the name of the object, while the other is a string representing the name of the type of the object.

DECLARATION

public class CONSTANT_NameAndType_Info
extends org.ldp.jdasm.constantpool.CpInfo

CONSTRUCTORS

- *CONSTANT_NameAndType_Info*
  `public` **CONSTANT_NameAndType_Info**`( java.io.InputStream` **is** `)`

- **Usage**
  - Creates a CONSTANT_NameAndType_Info reading its content by an input stream. The cursor of the input stream must point to the beginning of the constant after the byte that specify the type. All the constant pool reference are saved (as short value).

- *CONSTANT_NameAndType_Info*
  `public` **CONSTANT_NameAndType_Info**`( java.lang.String` **value** `)`
  - **Usage**
    - Creates a CONSTANT_NameAndType_Info from a string in the form "NAME DESCRIPTOR"
  - **See Also**
    - `org.ldp.jdasm.constantpool.TypeDescriptor` (in A.2.17, page 131)

- *CONSTANT_NameAndType_Info*
  `public` **CONSTANT_NameAndType_Info**`( java.lang.String` **name**`, java.lang.String` **descriptor** `)`
  - **Usage**
    - Creates a CONSTANT_NameAndType_Info with given name and descriptor

Methods

- *addToConstantPool*
  `public short` **addToConstantPool**`( org.ldp.jdasm.DConstantPool` **pool** `)`
  - **Usage**
    - It overrides the main CpInfo#addToConstantPool(DConstantPool) because it needs to add two Utf8 strings, for the name and for the descriptor

---

- *build*
  `public int` **build**`( byte []` **tofill**`, int` **atindex** `)`
  - **Usage**
    - It builds the content of this class into a final byte array. Such array is passed as first argument, and the index from where start to insert bytes is the second argument.
  - **Parameters**
    - `tofill` - the array to fill
    - `atindex` - the index from where start to fill
  - **Returns** - the number of byte inserted

- *build*
  `public int` **build**`( java.io.OutputStream` **output** `)`
  - **Usage**
    - It builds the content of this class into an output stream. This method is a faster version of int) since it doesn't need to call the IBuildable#getBuildLength() before.
  - **Parameters**

    - `output` - the stream to write onto
  - **Returns** - the number of byte inserted
  - **Exceptions**
    - `on` - error while writing onto the stream

- *copyValueFromReference*
  `public void` **copyValueFromReference**`(`
  `org.ldp.jdasm.DConstantPool` **pool** `)`
  - **Usage**
    - Converts the references to other constants in the pool from short index to a value (String).

- *equals*
  `public boolean` **equals**`( java.lang.Object` **o** `)`
  - **Usage**
    - We match when `o` has the same name and descriptor

- *getBuildLength*
  `public int` **getBuildLength**`( )`
  - **Usage**
    - Returns the length of portion of byte array the IBuildable class is going to create. If this class has instances of other IBuildable classes, then it will return the whole sum of all the various getBuildLength().

- *getConstantPoolDescriptorStringIndex*
  `public short` **getConstantPoolDescriptorStringIndex**`( )`
  - **Usage**
    - Get the descriptor string index of the constant pool

- *getConstantPoolNameStringIndex*
  `public short` **getConstantPoolNameStringIndex**`( )`
  - **Usage**
    - Get the name string index of the constant pool

- *getDescriptor*
  `public String` **getDescriptor**`( )`
  - **Usage**
    - Get the descriptor associated to this CONSTANT_NameAndType_Info

---

- *getName*
  `public String` **getName**`( )`
  - **Usage**
    - Get the name associated to this CONSTANT_NameAndType_Info

---

- *setConstantPoolDescriptorStringIndex*
  `public void` **setConstantPoolDescriptorStringIndex**`( short` **cpool_descriptorStringIndex** `)`
  - **Usage**
    - Set the descriptor string index of the constant pool

- *setConstantPoolNameStringIndex*
  `public void` **setConstantPoolNameStringIndex**`( short` **cpool_nameStringIndex** `)`
  - **Usage**
    * Set the name string index of the constant pool
- *setDescriptor*
  `public void` **setDescriptor**`( java.lang.String` **descriptor** `)`
  - **Usage**
    * Set the descriptor associated to this CONSTANT_NameAndType_Info
- *setName*
  `public void` **setName**`( java.lang.String` **name** `)`
  - **Usage**
    * Set the name associated to this CONSTANT_NameAndType_Info
- *toString*
  `public String` **toString**`( )`
  - **Usage**
    * Returns a short description of constant values

Methods inherited from class `org.ldp.jdasm.constantpool.CpInfo`

(in A.2.16, page 129)

- *addToConstantPool*
  `public short` **addToConstantPool**`( org.ldp.jdasm.DConstantPool` **pool** `)`
  - **Usage**
    * Adds this constant to the constant pool. If a constant has references to other constants, then here it creates such constants classes and add them before adding itself. In such way, it can know the index of its references.
- *copyValueFromReference*
  `public void` **copyValueFromReference**`( org.ldp.jdasm.DConstantPool` **pool** `)`
  - **Usage**
    * Converts the references to other constants in the pool from short index to a value (String). When read the constant pool from a file, all the constants has a reference number but hasn't the real value referenced. All the constant classes that uses references will overload this method to copy the referenced constant value into its own value.
- *getTag*
  `public byte` **getTag**`( )`
  - **Usage**
    * Get the tag (the type) of this CpInfo.
  - **See Also**
    * `org.ldp.jdasm.DConstantPool` (in A.1.3, page 73)
- *hashCode*
  `public int` **hashCode**`( )`
  - **Usage**
    * Returns the hashcode according to inner value. This is used to hashmapping constants

A.2.14 Class CONSTANT_String_Info

The class representing a CONSTANT_String_Info in the constant pool. It refers to a string into the same constant pool.

Declaration

public class CONSTANT_String_Info
extends org.ldp.jdasm.constantpool.CpInfo

Constructors

- *CONSTANT_String_Info*
  `public` **CONSTANT_String_Info**`( java.io.InputStream` **is** `)`
  - **Usage**
    - Creates a CONSTANT_String_Info reading its content by an input stream. The cursor of the input stream must point to the beginning of the constant after the byte that specify the type. All the constant pool reference are saved (as short value).
- *CONSTANT_String_Info*
  `public` **CONSTANT_String_Info**`( java.lang.String` **value** `)`
  - **Usage**
    - Creates a CONSTANT_String_Info with given string value.

Methods

- *addToConstantPool*
  `public short` **addToConstantPool**`( org.ldp.jdasm.DConstantPool` **pool** `)`
  - **Usage**
    - it overrides the main CpInfo#addToConstantPool(DConstantPool) because it needs to add a Utf8 string for the value
- *build*
  `public int` **build**`( byte []` **tofill**`, int` **atindex** `)`
  - **Usage**
    - It builds the content of this class into a final byte array. Such array is passed as first argument, and the index from where start to insert bytes is the second argument.
  - **Parameters**
    - `tofill` - the array to fill
    - `atindex` - the index from where start to fill
  - **Returns** - the number of byte inserted
- *build*
  `public int` **build**`( java.io.OutputStream` **output** `)`
  - **Usage**

    - It builds the content of this class into an output stream. This method is a faster version of int) since it doesn't need to call the IBuildable#getBuildLength() before.
  - **Parameters**
    - `output` - the stream to write onto
  - **Returns** - the number of byte inserted
  - **Exceptions**
    - `on` - error while writing onto the stream

- *copyValueFromReference*
  `public void` **copyValueFromReference**`(`
  `org.ldp.jdasm.DConstantPool` **pool** `)`
  - **Usage**
    - Converts the references to other constants in the pool from short index to a value (String).

- *equals*
  `public boolean` **equals**`( java.lang.Object` **o** `)`
  - **Usage**
    - We match when `o` has the same string value

- *getBuildLength*
  `public int` **getBuildLength**`( )`
  - **Usage**
    - Returns the length of portion of byte array the IBuildable class is going to create. If this class has instances of other IBuildable classes, then it will return the whole sum of all the various getBuildLength().

- *getConstantPoolValueStringIndex*
  `public short` **getConstantPoolValueStringIndex**`( )`
  - **Usage**
    - Get the value string index into the constant pool

- *getValue*
  `public String` **getValue**`( )`
  - **Usage**
    - Get the string value associated to this CONSTANT_String_Info

---

- *setConstantPoolValueStringIndex*
  `public void` **setConstantPoolValueStringIndex**`( short` **cpool_valueStringIndex** `)`
  - **Usage**
    - Set the value string index into the constant pool

- *setValue*
  `public void` **setValue**`( java.lang.String` **value** `)`
  - **Usage**
    - Set the string value associated to this CONSTANT_String_Info

---

- *toString*
  `public String` **toString**`( )`
  - **Usage**
    - Returns a short description of constant values

METHODS INHERITED FROM CLASS `org.ldp.jdasm.constantpool.CpInfo`

(in A.2.16, page 129)

- *addToConstantPool*
  `public short` **addToConstantPool**`( org.ldp.jdasm.DConstantPool` **pool** `)`
  - **Usage**
    * Adds this constant to the constant pool. If a constant has references to other constants, then here it creates such constants classes and add them before adding itself. In such way, it can know the index of its references.
- *copyValueFromReference*
  `public void` **copyValueFromReference**`( org.ldp.jdasm.DConstantPool` **pool** `)`
  - **Usage**
    * Converts the references to other constants in the pool from short index to a value (String). When read the constant pool from a file, all the constants has a reference number but hasn't the real value referenced. All the constant classes that uses references will overload this method to copy the referenced constant value into its own value.
- *getTag*
  `public byte` **getTag**`( )`
  - **Usage**
    * Get the tag (the type) of this CpInfo.
  - **See Also**
    * `org.ldp.jdasm.DConstantPool` (in A.1.3, page 73)
- *hashCode*
  `public int` **hashCode**`( )`
  - **Usage**
    * Returns the hashcode according to inner value. This is used to hashmapping constants

A.2.15 CLASS CONSTANT_Utf8_Info

The class representing a CONSTANT_Utf8_Info in the constant pool. It holds a string constant value.

DECLARATION

public class CONSTANT_Utf8_Info
extends org.ldp.jdasm.constantpool.CpInfo

CONSTRUCTORS

- *CONSTANT_Utf8_Info*
  `public` **CONSTANT_Utf8_Info**`( java.io.InputStream` **is** `)`
  - **Usage**
    * Creates a CONSTANT_Utf8_Info reading its content by an input stream. The cursor of the input stream must point to the beginning of the constant after the byte that specify the type. The constant pool value are saved.

- *CONSTANT_Utf8_Info*
  `public` **CONSTANT_Utf8_Info**`( java.lang.String` **value** `)`
  - **Usage**
    - Creates a CONSTANT_Utf8_Info with given string.

Methods

- *build*
  `public int` **build**`( byte []` **tofill**`, int` **atindex** `)`
  - **Usage**
    - It builds the content of this class into a final byte array. Such array is passed as first argument, and the index from where start to insert bytes is the second argument.
  - **Parameters**
    - `tofill` - the array to fill
    - `atindex` - the index from where start to fill
  - **Returns** - the number of byte inserted
- *build*
  `public int` **build**`( java.io.OutputStream` **output** `)`
  - **Usage**
    - It builds the content of this class into an output stream. This method is a faster version of int) since it doesn't need to call the IBuildable#getBuildLength() before.
  - **Parameters**
    - `output` - the stream to write onto
  - **Returns** - the number of byte inserted
  - **Exceptions**
    - `on` - error while writing onto the stream
- *equals*
  `public boolean` **equals**`( java.lang.Object` **o** `)`
  - **Usage**
    - We match when `o` has the same value
- *getBuildLength*
  `public int` **getBuildLength**`( )`
  - **Usage**
    - Returns the length of portion of byte array the IBuildable class is going to create. If this class has instances of other IBuildable classes, then it will return the whole sum of all the various getBuildLength().
- *getBuiltValue*
  `public byte` **getBuiltValue**`( )`
  - **Usage**
    - Get the constant pool representation of the corresponding value
- *getValue*
  `public String` **getValue**`( )`

- **Usage**
  - Get the string value associated to this CONSTANT_Utf8_Info

---

- *setValue*
  `public void` **setValue**`( java.lang.String` **value** `)`
  - **Usage**
    - Set the string value associated to this CONSTANT_Utf8_Info

---

- *toString*
  `public String` **toString**`( )`
  - **Usage**
    - Returns a short description of constant values

METHODS INHERITED FROM CLASS `org.ldp.jdasm.constantpool.CpInfo`

(in A.2.16, page 129)

- *addToConstantPool*
  `public short` **addToConstantPool**`( org.ldp.jdasm.DConstantPool` **pool** `)`
  - **Usage**
    - Adds this constant to the constant pool. If a constant has references to other constants, then here it creates such constants classes and add them before adding itself. In such way, it can know the index of its references.

- *copyValueFromReference*
  `public void` **copyValueFromReference**`( org.ldp.jdasm.DConstantPool` **pool** `)`
  - **Usage**
    - Converts the references to other constants in the pool from short index to a value (String). When read the constant pool from a file, all the constants has a reference number but hasn't the real value referenced. All the constant classes that uses references will overload this method to copy the referenced constant value into its own value.

- *getTag*
  `public byte` **getTag**`( )`
  - **Usage**
    - Get the tag (the type) of this CpInfo.
  - **See Also**
    - `org.ldp.jdasm.DConstantPool` (in A.1.3, page 73)

- *hashCode*
  `public int` **hashCode**`( )`
  - **Usage**
    - Returns the hashcode according to inner value. This is used to hashmapping constants

A.2.16 CLASS CpInfo

CpInfo is the abstract superclass of all the constants pool classes. It just adds the tag management for all the constants and two empty method that should be overloaded from classes that uses references instead of values.

Declaration

public abstract class CpInfo
extends java.lang.Object
implements IBuildable

Constructors

- *CpInfo*
  `public` **CpInfo**`( byte` **tag** `)`
  - **Usage**
    - The Constant Info constructor requires the tag value.
  - **See Also**
    - `org.ldp.jdasm.DConstantPool` (in A.1.3, page 73)

Methods

- *addToConstantPool*
  `public short` **addToConstantPool**`( org.ldp.jdasm.DConstantPool` **pool** `)`
  - **Usage**
    - Adds this constant to the constant pool. If a constant has references to other constants, then here it creates such constants classes and add them before adding itself. In such way, it can know the index of its references.

- *copyValueFromReference*
  `public void` **copyValueFromReference**`( org.ldp.jdasm.DConstantPool` **pool** `)`
  - **Usage**
    - Converts the references to other constants in the pool from short index to a value (String). When read the constant pool from a file, all the constants has a reference number but hasn't the real value referenced. All the constant classes that uses references will overload this method to copy the referenced constant value into its own value.

- *getTag*
  `public byte` **getTag**`( )`
  - **Usage**
    - Get the tag (the type) of this CpInfo.
  - **See Also**
    - `org.ldp.jdasm.DConstantPool` (in A.1.3, page 73)

- *hashCode*
  `public int` **hashCode**`( )`
  - **Usage**
    - Returns the hashcode according to inner value. This is used to hashmapping constants

A.2.17 Class **TypeDescriptor**

Declaration

public class TypeDescriptor
extends java.lang.Object

Constructors

- *TypeDescriptor*
  `public` **TypeDescriptor**`( )`

Methods

- *getFieldDescriptorByString*
  `public static String` **getFieldDescriptorByString**`(`
  `java.lang.String` **type** `)`
  - **Usage**
    * Converts the human readable stringed `type` into the java class file internal representation of such type. Examples:

```
"int"                => "I"
"double[]"           => "[D"
"long[][]"           => "[[[J"
"my.pack.MyObject"   => "Lmy/pack/MyObject;"
```

- *getMethodArgumentByDescriptor*
  `public static String` **getMethodArgumentByDescriptor**`(`
  `java.lang.String` **descr** `)`
  - **Usage**
    * Get the descriptor in the form "(IDLmy/pack/MyObject;[I)D" and returns "IDLmy/pack/MyObject;[I" .

- *getMethodDescriptor*
  `public static String` **getMethodDescriptor**`( java.lang.String` **type**`, java.util.Vector` **args** `)`
  - **Usage**
    * Returns the method descriptor starting from the return type and its argument.
  - **Parameters**
    * `type` - The return type of the method (e.g.: "int")
    * `args` - The arguments of the method (e.g.: { "int[] arg0", "my.pack.MyObject arg1" })
  - **Returns** - The descriptor, e.g.: "([ILmy/pack/MyObject;)I"
  - **Exceptions**
    * `org.ldp.jdasm.exception.MethodDescriptorException` - if errors occur while parsing the types or argument strings

- *getMethodReturnTypeAndArgumentByDescriptor*
  `public static String` **getMethodReturnTypeAndArgumentByDescriptor(**
  `java.lang.String` **descriptor,** `java.util.Vector` **args)**
  - **Usage**
    - Gets the method descriptor and analyzes it to determinate the return type and the arguments.
  - **Parameters**
    - `descriptor` - The method descriptor in the form "(IDLmy/pack/MyObject;[I)D"
    - `args` - The vector is filled (in output) with {"int arg0", "double arg1", "long arg2", "my.pack.MyObject arg3", "int[] arg4"}
  - **Returns** - the return type ("double" in the example given)
  - **Exceptions**
    - `org.ldp.jdasm.exception.MethodDescriptorException` - if errors occur while parsing the descriptor string

---

- *getMethodReturnTypeByDescriptor*
  `public static String` **getMethodReturnTypeByDescriptor(**
  `java.lang.String` **descr)**
  - **Usage**
    - Get the descriptor in the form "(IDLmy/pack/MyObject;[I)D" and returns the return type "D" .

- *getNextFieldTypeLengthInDescriptor*
  `public static int` **getNextFieldTypeLengthInDescriptor(**
  `java.lang.String` **descriptor)**
  - **Usage**
    - Returns the length of the first type descriptor found in the given descriptor.

      E.g.: the descriptor describing these types "int double[] MyObject long[][]" is "ID[LMyObject;[[J". With this descriptor, the method returns the length of the first type found, that is "I" of integer: it returns 1.

```
String argument = "ID[LMyObject;[[J";
argument = argument.substring( getNextFieldTypeLengthInDescriptor(argument) );
.. arguments now is "D[LMyObject;[[J" ..
argument = argument.substring( getNextFieldTypeLengthInDescriptor(argument) );
.. arguments now is "[LMyObject;[[J" ..
argument = argument.substring( getNextFieldTypeLengthInDescriptor(argument) );
.. arguments now is "[[J" ..
argument = argument.substring( getNextFieldTypeLengthInDescriptor(argument) );
.. arguments now is empty.
```

  - **Exceptions**
    - `org.ldp.jdasm.exception.FieldDescriptorException` - if the first character do not recognize a valid type (it is not one of these "BCDFIJLSZ["

- *getStringByFieldDescriptor*
  `public static String` **getStringByFieldDescriptor(**
  `java.lang.String` **descriptor)**

– **Usage**

  * Returns the name of the class representing the given descriptor.

```
"I"                    => "int"
"[D"                   => "double[]"
"[[[J"                 => "long[][][]"
"Lmy/pack/MyObject;"   => "my.pack.MyObject"
```

A.3 Package org.ldp.jdasm.attribute

Classes

Attributable 136

This class comes to establish a convention for all the class files structures that can have attributes.

In fact they all will expose this methods to easy manipulate their attributes.

CodeAttribute 138

The class representing either the java class file CodeAttribute, and a container for the instruction of a method.

ConstantValueAttribute 152

This class represents an Attribute of the java class file.

The attribute is intended to store a constant value which can be of one of the primitive types.

CustomAttribute 159

This class represents a custom Attribute of the java class file.

You can create your own attribute by specifying its name.

DeprecatedAttribute 165

This class represents an Attribute of the java class file.

The Deprecated attribute is an optional attribute of DClass , DField , and DMethod classes.

A class, interface, method, or field may be marked using a Deprecated attribute to indicate that the class, interface, method, or field has been superseded.

ExceptionsAttribute 170

This class represents an Attribute of the java class file.

The Exceptions attribute is an attribute used in the attributes table of a DMethod class.

The Exceptions attribute indicates which checked exceptions a method may throw; there may be at most one Exceptions attribute in each DMethod.

InnerClassesAttribute 176

This class represents an Attribute of the java class file.

This attribute uses an inner class InnerClassesElement as element of its table.

InnerClassesElement 182

Class used inside the InnerClassesAttribute .

LineNumberTableAttribute 185

This class represents an Attribute of the java class file.

LocalVariableTableAttribute 191

This class represents an Attribute of the java class file.

This attribute uses the class LocalVariableTableElement as element of its table: each of these element has information about the starting instruction, the length in byte of the variable's scope, the name of the variable and its descriptor..

LocalVariableTableElement 198

Class used inside the LocalVariableTableAttribute and inside LocalVariableTypeTableAttribute classes: it has informations about the name and descriptor or signature of the variable, start_pc and length of the scope, index into the local variable array.

LocalVariableTypeTableAttribute 199

This class represents an Attribute of the java class file.

This attribute uses the class LocalVariableTableElement as element of its table: each of these element has information about the starting instruction, the length in byte of the variable's scope, the name of the variable and its signature..

SignatureAttribute 205

This class represents an Attribute of the java class file.

This attribute is intended to hold the signature of a class (if it is an attribute of DClass), of a method (if it is an attribute of DMethod) or of a field (if it is an attribute of DField).

SourceDebugExtensionAttribute 211

This class represents an Attribute of the java class file.

This attribute (attribute of DClass) holds a string in UFT-8 format with no terminating zero byte.

SourceFileAttribute 217

This class represents an Attribute of the java class file.

This attribute is intended to hold the name of the source file from which this class file was compiled.

StackMapTableAttribute 223

Unimplemented Attribute.

SyntheticAttribute 223

This class represents an Attribute of the java class file.

A class member that does not appear in the source code must be marked using a Synthetic attribute.

A.3.1 Classes

A.3.2 Class Attributable

This class comes to establish a convention for all the class files structures that can have attributes.

In fact they all will expose this methods to easy manipulate their attributes.

Declaration

public abstract class Attributable
extends org.ldp.jdasm.constantpool.AConstantPoolUser

Constructors

- *Attributable*
  `public` **Attributable**`( )`

Methods

- *addAttribute*
  `public DAttribute` **addAttribute**`( org.ldp.jdasm.DAttribute` **attr** `)`
  - **Usage**
    - Adds a DAttribute.
  - **Returns** - the inserted DAttribute

- *attributeCount*
  `public int` **attributeCount**`( )`
  - **Usage**
    - Returns the number of attribute for this class

- *getAttribute*
  `public DAttribute` **getAttribute**`( int` **idx** `)`
  - **Usage**
    - Get the DAttribute at specified position

- *getAttribute*
  `public DAttribute` **getAttribute**`( java.lang.String` **name** `)`
  - **Usage**
    - Returns the DAttribute with specified name, or null if it doesn't exist.

- *getAttributes*
  `public DAttribute` **getAttributes**`( )`
  - **Usage**
    - Returns all the attributes

- *hasAttribute*
  `public boolean` **hasAttribute**`( java.lang.String` **name** `)`
  - **Usage**
    - Returns true if an attribute with given name exists.

- *removeAttribute*
  `public boolean` **removeAttribute**`( org.ldp.jdasm.DAttribute` **attr** `)`
  - **Usage**
    - Removes the specified DAttribute (match on equals())
  - **Returns** - true if the attribute is removed (if it was present), false it it is not present in the list.

- *removeAttribute*
  `public void` **removeAttribute**`( int` **idx** `)`
  - **Usage**
    - Removes the DAttribute at specified position

- *removeAttribute*
  `public boolean` **removeAttribute**`( java.lang.String` **name** `)`
  - **Usage**
    - Removes the DAttribute with specified name. Returns true if it is removed (if it was present), false if it is not present in the list.

Methods inherited from class
`org.ldp.jdasm.constantpool.AConstantPoolUser`

(in A.2.4, page 97)

- *constantPoolUserChildSize*
  `public abstract int` **constantPoolUserChildSize**`( )`
  - **Usage**
    - Returns the number of IConstantPoolUser instances of this class.

- *constantPoolValueSize*
  `public abstract int` **constantPoolValueSize**`( )`
  - **Usage**
    - Returns the number of values that this constant pool user wants to insert into the constant pool.

- *getConstantPoolUserChild*
  `public abstract AConstantPoolUser` **getConstantPoolUserChild**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th IConstantPoolUser instance of this class.

- *getConstantValue*
  `public abstract Object` **getConstantValue**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th value that this constant pool user wants to insert into the constant pool.
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *getConstantValueType*
  `public abstract int` **getConstantValueType**`( int` **idx** `)`
  - **Usage**
    - Returns the value type of the idx-th value that this constant pool user wants to insert into the constant pool. The list of the types is declared in DConstantPool
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *setConstantValueIndex*
  `public abstract void` **setConstantValueIndex**`( int` **idx**`, short` **cpool_index** `)`
  - **Usage**
    - The constant pool tells the user that the value retrieved with AConstantPoolUser#getConstantValue(int) at index "idx" has received the "cpool_index" index in the constant pool.
  - **Parameters**
    - `idx` - the index of the value mapped with the ones returned by getConstantValue(int)
    - `cpool_index` - the index of the value in the constant pool

A.3.3 Class CodeAttribute

The class representing either the java class file CodeAttribute, and a container for the instruction of a method. In this class you will find all the API to interact with the code.

The class provides an inner computation of the states of the local variable array and the stack for each instruction instant, offering the possibility to automatically computing of stack and local variable array size, and offering some other consistency features while inserting or removing code. Both automatic max stack and locals computing, and code support can be turned of to improve the performance with CodeAttribute#setCodeSupportEnabled(boolean) and CodeAttribute#setAutomaticMaxStackAndLocals(boolean) .

As constant pool user, this class keep the counter of all the instruction that need a link into the cpool at build time. for efficient reason methods like CodeAttribute#getConstantValue(int) , CodeAttribute#getConstantValueType(int) and short) uses a internal counter to cycle through the vector of instruction: this mean that the index is ignored if much greater than DAttribute super class DAttribute#constantPoolValueSize() , and at each call you get a reference to the next suitable instruction.

Declaration

public class CodeAttribute
extends org.ldp.jdasm.DAttribute

Constructors

- *CodeAttribute*
  `public` **CodeAttribute**`( )`

- **Usage**
  - Default constructor. It creates an empty CodeAttribute with empty code.

- *CodeAttribute*
  `public` **CodeAttribute**`( java.io.InputStream` **is**`, org.ldp.jdasm.DConstantPool` **cpool** `)`
  - **Usage**
    - Creates a CodeAttribute attribute reading it from an input stream whose cursor must point to the beginning of the attribute just after the two bytes that reference the name in the constant pool. The constant pool needs to have all the string constant values used by this attribute.

Methods

- *addCode*
  `public void` **addCode**`( org.ldp.jdasm.attribute.CodeAttribute` **code** `)`
  - **Usage**
    - Appends all the instruction of the given CodeAttribute to this CodeAttribute code. Exceptions from the exceptions table are also added (nothing's copied).

- *addCode*
  `public void` **addCode**`( org.ldp.jdasm.attribute.CodeAttribute` **code**`, int` **idx** `)`
  - **Usage**
    - Adds all the instruction of the given CodeAttribute to this CodeAttribute at specific position.

- *addCode*
  `public Instruction` **addCode**`( org.ldp.jdasm.attribute.instruction.Instruction` **instr** `)`
  - **Usage**
    - Adds a specific instruction `instr` at the end of the list.

- *addCode*
  `public Instruction` **addCode**`( org.ldp.jdasm.attribute.instruction.Instruction` **instr**`, int` **idx** `)`
  - **Usage**
    - Adds a specific instruction `instr` at specific index `idx`. Negative value of `idx` are meant to be relative to the count of the instructions: using idx = -1 will insert the new instruction next to the last.

- *addCode*
  `public void` **addCode**`( java.lang.String` **code** `)`
  - **Usage**

    * Parses the `code` passed as argument to get a certain number of instruction that are appended at the end of the list. For how the string `code` must be, refer to CodeAttribute#addCode(String,int)

- *addCode*
  `public void` **addCode**`( java.lang.String` **code**`, int` **idx** `)`
  - **Usage**
    * Parses the `code` passed as argument to get a certain number of instruction that are added to the list at position `idx`.

- *addException*
  `public void` **addException**`(`
  `org.ldp.jdasm.attribute.instruction.ExceptionElement` **exception** `)`
  - **Usage**
    * Adds a specific exception.

- *addValidationError*
  `public void` **addValidationError**`(`
  `org.ldp.jdasm.attribute.instruction.Instruction` **instr**`,`
  `java.lang.String` **validationError** `)`
  - **Usage**
    * Adds the given `validationError` at the error string.
  - **Parameters**
    * `instr` - The instruction that raised the error

- *build*
  `public int` **build**`( byte []` **tofill**`, int` **atindex** `)`
  - **Usage**
    * It builds the content of this class into a final byte array. Such array is passed as first argument, and the index from where start to insert bytes is the second argument.
  - **Parameters**
    * `tofill` - the array to fill
    * `atindex` - the index from where start to fill
  - **Returns** - the number of byte inserted

- *build*
  `public int` **build**`( java.io.OutputStream` **output** `)`
  - **Usage**
    * It builds the content of this class into an output stream. This method is a faster version of int) since it doesn't need to call the IBuildable#getBuildLength() before.
  - **Parameters**
    * `output` - the stream to write onto
  - **Returns** - the number of byte inserted
  - **Exceptions**
    * `on` - error while writing onto the stream

- *buildInstructionIndex*
  `public void` **buildInstructionIndex**`( )`

  - **Usage**
    * Sets the real offset to each instruction by computing each instruction size. It also change the opcode LDC into LDC_W if needed, and GOTO in GOTO_W if needed. Finally the value `codelen` is filled with right value.

- *buildSupport*
  `public void` **buildSupport**`( )`

- *classFileInfo*
  `public String` **classFileInfo**`( int` **indent_level** `)`
  - **Usage**
    * Returns a printable info string

- *clearValidationErrors*
  `public void` **clearValidationErrors**`( )`
  - **Usage**
    * Clears the validation error bit.

- *constantPoolUserChildSize*
  `public int` **constantPoolUserChildSize**`( )`
  - **Usage**
    * Returns the number of IConstantPoolUser instances of this class.

- *constantPoolValueSize*
  `public int` **constantPoolValueSize**`( )`
  - **Usage**
    * Returns the number of values that this constant pool user wants to insert into the constant pool.

- *getBuildLength*
  `public int` **getBuildLength**`( )`
  - **Usage**
    * Returns the length of portion of byte array the IBuildable class is going to create. If this class has instances of other IBuildable classes, then it will return the whole sum of all the various getBuildLength().

- *getByteCodeSize*
  `public int` **getByteCodeSize**`( )`
  - **Usage**
    * Returns the length in byte of this byte code.

- *getCodeSize*
  `public int` **getCodeSize**`( )`
  - **Usage**
    * Returns the number of instruction.

- *getConstantPoolUserChild*
  `public AConstantPoolUser` **getConstantPoolUserChild**`( int` **idx** `)`

- **Usage**
  - Returns the idx-th IConstantPoolUser instance of this class.

- *getConstantValue*
  `public Object` **getConstantValue**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th value that this constant pool user wants to insert into the constant pool.
  - **Parameters**
    - `idx` - for efficient reason this parameter is mainly ignored.. see detail in CodeAttribute
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *getConstantValueType*
  `public int` **getConstantValueType**`( int` **idx** `)`
  - **Usage**
    - Returns the value type of the idx-th value that this constant pool user wants to insert into the constant pool. The list of the types is declared in DConstantPool
  - **Parameters**
    - `idx` - for efficient reason this parameter is mainly ignored.. see detail in CodeAttribute
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *getException*
  `public ExceptionElement` **getException**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th exception.

- *getExceptions*
  `public ExceptionElement` **getExceptions**`( )`
  - **Usage**
    - Returns all the exceptions.

- *getExceptionSize*
  `public int` **getExceptionSize**`( )`
  - **Usage**
    - Returns the number of try-catch block.

- *getFirstInstruction*
  `public Instruction` **getFirstInstruction**`( )`
  - **Usage**
    - Returns the first instruction, or null it hasn't any.

- *getInstruction*
  `public Instruction` **getInstruction**`( int` **idx** `)`
  - **Usage**

    - Returns the idx-th instruction.

- *getInstructionAtOffset*
  `public Instruction` **getInstructionAtOffset**`( int` **offset** `)`
  - **Usage**
    - Returns the instruction that is at specified byte offset. Returns null if at `offset` does not correspond any instruction.

- *getLastInstruction*
  `public Instruction` **getLastInstruction**`( )`
  - **Usage**
    - Returns the last instruction, or null it hasn't any.

- *getMaxLocals*
  `public int` **getMaxLocals**`( )`
  - **Usage**
    - Returns the max use of stack. The value is always computed and kept updated (unless automatic update is set off).
  - **See Also**
    - `org.ldp.jdasm.attribute.CodeAttribute.setAutomaticMaxStackAndLocals(boolean)` (in A.3.3, page 146)

- *getMaxStack*
  `public int` **getMaxStack**`( )`
  - **Usage**
    - Returns the max use of stack. The value is always computed and kept updated (unless automatic update is set off).
  - **See Also**
    - `org.ldp.jdasm.attribute.CodeAttribute.setAutomaticMaxStackAndLocals(boolean)` (in A.3.3, page 146)

- *getMethodDescriptor*
  `public String` **getMethodDescriptor**`( )`
  - **Usage**
    - Get the descriptor of the method that encapsulates this CodeAttribute. If no information about the encapsulater method has been given with boolean) this always returns null.
  - **See Also**
    - `org.ldp.jdasm.attribute.CodeAttribute.setMethodDescriptor(String,boolean)`

- *getMethodStaticFlag*
  `public boolean` **getMethodStaticFlag**`( )`
  - **Usage**
    - Returns true if the method that encapsulate this CodeAttribute is static. If no information about the encapsulater method has been given with boolean) this always returns false.
  - **See Also**
    - `org.ldp.jdasm.attribute.CodeAttribute.setMethodDescriptor(String,boolean)`

- *getRangedCodeAttribute*
  `public CodeAttribute` **getRangedCodeAttribute**`(`
  `org.ldp.jdasm.attribute.instruction.InstructionRange` **range**
  `)`
  - **Usage**
    * Returns a part of code delimited by the given `range`.
  - **See Also**
    * `org.ldp.jdasm.attribute.CodeAttribute.getRangedCodeAttribute(int, int)`

- *getRangedCodeAttribute*
  `public CodeAttribute` **getRangedCodeAttribute**`( int` **from**`,`
  `int` **to** `)`
  - **Usage**
    * Returns a part of code delimited by the given range.

      The new CodeAttribute will have `to - from` instructions and no attributes at all. If one of ranged instruction is inside a try-catch block, then the new CodeAttribute will have an ExceptionElement that will include this instruction; if the catching instruction is not present in selected code its value will be `null`, so be care and always check the exceptions.

      Same thing happens whenever a jump instruction has a target that falls outside the selected code: it's target will be set to `null`.
  - **Parameters**
    * `from` - Inclusive: the index of the first instruction of the range
    * `to` - Exclusive: The index of the first instruction after the range

---

- *getValiationError*
  `public String` **getValiationError**`( )`
  - **Usage**
    * Returns the string containing the validation error. Returns null if no error has been reported.

- *hasValidationError*
  `public boolean` **hasValidationError**`( )`
  - **Usage**
    * Returns true if a validation error has been reported during the validation.

- *isAutomaticMaxStackAndLocals*
  `public boolean` **isAutomaticMaxStackAndLocals**`( )`
  - **Usage**
    * Returns true if CodeAttribute is handling max_stack and max_locals in automatic way; false otherwise.

- *isCodeSupportEnabled*
  `public boolean` **isCodeSupportEnabled**`( )`

- **Usage**
  - Returns true if CodeAttribute builds the support structures which help to analyze the code.

- *removeCode*
  `public void` **`removeCode`**`(`
  `org.ldp.jdasm.attribute.instruction.InstructionRange` **`range`**
  `)`
  - **Usage**
    - Removes part of the code. The part is indicated by the InstructionRange range.

- *removeCode*
  `public void` **`removeCode`**`( int` **`from`**`, int` **`to`** `)`
  - **Usage**
    - Removes part of the code. The part is starting at instruction `from` (inclusive) and ends at instruction `to` (exclusive)
  - **Parameters**
    - `from` - The first instruction to remove
    - `to` - The first instruction that wont be removed

- *removeException*
  `public void` **`removeException`**`( int` **`index`** `)`
  - **Usage**
    - Remove the exception at specified `index`.

- *removeInstructionFromAttribute*
  `public void` **`removeInstructionFromAttribute`**`(`
  `org.ldp.jdasm.attribute.instruction.Instruction` **`instr`** `)`
  - **Usage**
    - Tells to any direct attribute of type LineNumberTableAttribute , LocalVariableTableAttribute and LocalVariableTypeTableAttribute that the given instruction is no longer part of this CodeAttribute. Any entry in these attributes containing this instruction will be deleted.
  - **See Also**
    - `org.ldp.jdasm.attribute.LineNumberTableAttribute.removeInstruction(Instruction)`
    - `org.ldp.jdasm.attribute.LocalVariableTableAttribute.removeInstruction(Instruction, CodeAttribute)`
    - `org.ldp.jdasm.attribute.LocalVariableTypeTableAttribute.removeInstruction(Instruction, CodeAttribute)`

- *replaceInstruction*
  `public void` **`replaceInstruction`**`( int` **`idx`**`,`
  `org.ldp.jdasm.attribute.instruction.Instruction` **`instr`** `)`
  - **Usage**
    - Replace the instruction at index `idx` with the given instruction. If any jump was targeting the old instruction, then the given instruction will be designed as the new target. The attributes that use the old instruction will lose the reference.

- *setAutomaticMaxStackAndLocals*
  `public void` **setAutomaticMaxStackAndLocals**`( boolean` **automaticMaxStackAndLocals** `)`
  - **Usage**
    * Set whenever CodeAttribute has to handle max_stack and max_locals value automatically. If you put this to manual (by passing `false`) don't forget to update max locals and stack values before a build, or the class wont be able to be loaded.
  - **Parameters**
    * `automaticMaxStackAndLocals` - if True, this call will implicitly turn on the build support by calling CodeAttribute#setCodeSupportEnabled(boolean) .

- *setCodeSupportEnabled*
  `public void` **setCodeSupportEnabled**`( boolean` **withCodeSupport** `)`
  - **Usage**
    * Set whenever CodeAttribute has to build the support structures. With such structures, CodeAttribute keep informations about the local variable array and the stack state for each instruction. Doing this makes CodeAttribute enable to compute max locals and stacks, and help the code consistency while inserting or removing code, but it needs more computation.

- *setConstantValueIndex*
  `public void` **setConstantValueIndex**`( int` **idx**`, short` **cpool_index** `)`
  - **Usage**
    * The constant pool tells the user that the value retrieved with AConstantPoolUser#getConstantValue(int) at index "idx" has received the "cpool_index" index in the constant pool.
  - **Parameters**
    * `idx` - the index of the value mapped with the ones returned by getConstantValue(int).

      for efficient reason this parameter is mainly ignored.. see detail in CodeAttribute
    * `cpool_index` - the index of the value in the constant pool

- *setMaxLocals*
  `public void` **setMaxLocals**`( int` **maxLocals** `)`
  - **Usage**
    * Set manually the value of max_stack. Using this will implicitly turn off the automatic update of max_stack and max_locals.
  - **See Also**
    * `org.ldp.jdasm.attribute.CodeAttribute.setAutomaticMaxStackAndLocals(boolean)` (in A.3.3, page 146)
    * `org.ldp.jdasm.attribute.CodeAttribute.setMaxStack(int)` (in A.3.3, page 147)

- *setMaxStack*
  `public void` **setMaxStack**`( int` **maxStack** `)`
  - **Usage**
    * Set manually the value of max_stack. Using this will implicitly turn off the automatic update of max_stack and max_locals.
  - **See Also**
    * `org.ldp.jdasm.attribute.CodeAttribute.setAutomaticMaxStackAndLocals(boolean)` (in A.3.3, page 146)
    * `org.ldp.jdasm.attribute.CodeAttribute.setMaxLocals(int)` (in A.3.3, page 146)

- *setMethodDescriptor*
  `public void` **setMethodDescriptor**`( java.lang.String` **methodDescriptor**`, boolean` **isStaticMethod** `)`
  - **Usage**
    * Set the descriptor of the method that encapsulates this CodeAttribute. This descriptor is necessary to build a correct CodeAttribute support (manually or automatically with CodeAttribute#buildSupport()) because of the number and type of argument of this method that will be put into the local variable array at the beginning of the code.
  - **Parameters**
    * `methodDescriptor` - the descriptor string, something like "(Ljava/lang/String;ID)V)"
    * `isStaticMethod` - must be true if the method that encapsulates this CodeAttribute is a static method
  - **See Also**
    * `org.ldp.jdasm.attribute.CodeAttribute.getMethodDescriptor()`

- *wrapSuperConstructor*
  `public void` **wrapSuperConstructor**`( java.lang.String` **oldSuperConstructorClassName**`, java.lang.String` **newConstructorClassName** `)`
  - **Usage**
    * Changes the super constructor invocation to call the constructor of the given class name `newConstructorClassName`.

      To do this the method descriptor for this code attribute must be set (see boolean)) and this CodeAttribute must represent a constructor method CodeAttribute.

      Consider this CodeAttribute method descriptor as example: `<init>(ILjava/lang/String;)V`.

      Then the wrap consists in making the INVOKESPECIAL actual call (to `oldSuperConstructorClassName`) to invoke another constructor of the class `newConstructorClassName` with the same descriptor. Then all the arguments received by this method are passed to the new super constructor.

The example above will generate these instructions:

```
ALOAD_0
ILOAD_1
ALOAD_2
INVOKESPECIAL   newConstructorClassName <init> (ILjava/lang/String;)V
```

replacing these old instructions:

```
ALOAD_0
.. any other LOADS ..
INVOKESPECIAL   oldSuperConstructorClassName <init> oldSuperConstructorDescriptor
```

- **Exceptions**
  - `org.ldp.jdasm.exception.MethodDescriptorException` - if the method descriptor for this CodeAttribute is not set, or if it's impossible to find any INVOKESPECIAL call to the super constructor.

Methods inherited from class `org.ldp.jdasm.DAttribute`

(in A.1.6, page 90)

- *build*
  `public int` **build**`( byte [] tofill, int` **atindex** `)`
  - **Usage**
    - It builds the content of this class into a final byte array. Such array is passed as first argument, and the index from where start to insert bytes is the second argument.
  - **Parameters**
    - `tofill` - the array to fill
    - `atindex` - the index from where start to fill
  - **Returns** - the number of byte inserted

- *build*
  `public int` **build**`( java.io.OutputStream` **output** `)`
  - **Usage**
    - It builds the content of this class into an output stream. This method is a faster version of int) since it doesn't need to call the IBuildable#getBuildLength() before.
  - **Parameters**
    - `output` - the stream to write onto
  - **Returns** - the number of byte inserted
  - **Exceptions**
    - `on` - error while writing onto the stream

- *classFileInfo*
  `public String` **classFileInfo**`( int` **indent_level** `)`
  - **Usage**
    - Returns a printable info string

- *constantPoolUserChildSize*
  `public int` **constantPoolUserChildSize**`( )`
  - **Usage**
    - Returns the number of IConstantPoolUser instances of this class.

- *constantPoolValueSize*
  `public int` **constantPoolValueSize**`( )`
  - **Usage**
    - Returns the number of values that this constant pool user wants to insert into the constant pool.

- *getBuildLength*
  `public int` **getBuildLength**`( )`
  - **Usage**
    - Returns the length of portion of byte array the IBuildable class is going to create. If this class has instances of other IBuildable classes, then it will return the whole sum of all the various getBuildLength().

- *getConstantPoolUserChild*
  `public AConstantPoolUser` **getConstantPoolUserChild**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th IConstantPoolUser instance of this class.

- *getConstantValue*
  `public Object` **getConstantValue**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th value that this constant pool user wants to insert into the constant pool.
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *getConstantValueType*
  `public int` **getConstantValueType**`( int` **idx** `)`
  - **Usage**
    - Returns the value type of the idx-th value that this constant pool user wants to insert into the constant pool. The list of the types is declared in DConstantPool
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *getDAttributeFromInputStream*
  `public static DAttribute` **getDAttributeFromInputStream**`(`
  `java.io.InputStream` **is**`, org.ldp.jdasm.DConstantPool` **cpool** `)`
  - **Usage**
    - This function read an Attribute from an input stream whose cursor must point to the first byte (which specify the index into the constant pool of the name).

- *getDAttributeFromInputStream*
  `public static DAttribute` **getDAttributeFromInputStream**`(`
  `java.io.InputStream` **is**`, org.ldp.jdasm.DConstantPool` **cpool**`,`
  `org.ldp.jdasm.attribute.CodeAttribute` **code** `)`
  - **Usage**
    - This function read an Attribute from an input stream whose cursor must point to the first byte (which specify the index into the constant pool of the name). If the attribute's parent is a CodeAttribute, it must be specified as argument to allow the LineNumberTableAttribute to have link to instructions.

- *getName*
  `public String` **getName**`( )`
  - **Usage**
    - Get the name of the attribute.

- *setConstantValueIndex*
  `public void` **setConstantValueIndex**`( int` **idx**`, short` **cpool_index** `)`
  - **Usage**
    - The constant pool tells the user that the value retrieved with AConstantPoolUser#getConstantValue(int) at index "idx" has received the "cpool_index" index in the constant pool.
  - **Parameters**
    - `idx` - the index of the value mapped with the ones returned by getConstantValue(int)
    - `cpool_index` - the index of the value in the constant pool

- *setName*
  `public void` **setName**`( java.lang.String` **name** `)`
  - **Usage**
    - Set the name of the attribute.

Methods inherited from class `org.ldp.jdasm.attribute.Attributable`

(in A.3.2, page 136)

- *addAttribute*
  `public DAttribute` **addAttribute**`( org.ldp.jdasm.DAttribute` **attr** `)`
  - **Usage**
    - Adds a DAttribute.
  - **Returns** - the inserted DAttribute

- *attributeCount*
  `public int` **attributeCount**`( )`
  - **Usage**
    - Returns the number of attribute for this class

- *getAttribute*
  `public DAttribute` **getAttribute**`( int` **idx** `)`
  - **Usage**
    - Get the DAttribute at specified position

- *getAttribute*
  `public DAttribute` **getAttribute**`( java.lang.String` **name** `)`
  - **Usage**
    - Returns the DAttribute with specified name, or null if it doesn't exist.

- *getAttributes*
  `public DAttribute` **getAttributes**`( )`
  - **Usage**
    - Returns all the attributes

- *hasAttribute*
  `public boolean` **hasAttribute**`( java.lang.String` **name** `)`
  - **Usage**
    - Returns true if an attribute with given name exists.

- *removeAttribute*
  `public boolean` **removeAttribute**`( org.ldp.jdasm.DAttribute` **attr** `)`
  - **Usage**
    - Removes the specified DAttribute (match on equals())

  - **Returns** - true if the attribute is removed (if it was present), false it it is not present in the list.

- *removeAttribute*
  `public void` **removeAttribute**`( int` **idx** `)`
  - **Usage**
    * Removes the DAttribute at specified position

- *removeAttribute*
  `public boolean` **removeAttribute**`( java.lang.String` **name** `)`
  - **Usage**
    * Removes the DAttribute with specified name. Returns true if it is removed (if it was present), false if it is not present in the list.

Methods inherited from class
`org.ldp.jdasm.constantpool.AConstantPoolUser`

(in A.2.4, page 97)

- *constantPoolUserChildSize*
  `public abstract int` **constantPoolUserChildSize**`( )`
  - **Usage**
    * Returns the number of IConstantPoolUser instances of this class.

- *constantPoolValueSize*
  `public abstract int` **constantPoolValueSize**`( )`
  - **Usage**
    * Returns the number of values that this constant pool user wants to insert into the constant pool.

- *getConstantPoolUserChild*
  `public abstract AConstantPoolUser` **getConstantPoolUserChild**`( int` **idx** `)`
  - **Usage**
    * Returns the idx-th IConstantPoolUser instance of this class.

- *getConstantValue*
  `public abstract Object` **getConstantValue**`( int` **idx** `)`
  - **Usage**
    * Returns the idx-th value that this constant pool user wants to insert into the constant pool.
  - **Exceptions**
    * `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *getConstantValueType*
  `public abstract int` **getConstantValueType**`( int` **idx** `)`
  - **Usage**
    * Returns the value type of the idx-th value that this constant pool user wants to insert into the constant pool. The list of the types is declared in DConstantPool
  - **Exceptions**
    * `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *setConstantValueIndex*
  `public abstract void` **setConstantValueIndex**`( int` **idx**`, short` **cpool_index** `)`
  - **Usage**

    - The constant pool tells the user that the value retrieved with AConstantPoolUser#getConstantValue(int) at index "idx" has received the "cpool_index" index in the constant pool.
  - **Parameters**
    - `idx` - the index of the value mapped with the ones returned by getConstantValue(int)
    - `cpool_index` - the index of the value in the constant pool

A.3.4 Class **ConstantValueAttribute**

This class represents an Attribute of the java class file.

The attribute is intended to store a constant value which can be of one of the primitive types. The value is stored in an object of type Object, and its type can be retrieved by the ConstantValueAttribute#getType()

Declaration

public class ConstantValueAttribute
extends org.ldp.jdasm.DAttribute

Constructors

- *ConstantValueAttribute*
  `public` **ConstantValueAttribute**`( )`
  - **Usage**
    - Creates an empty ConstantValue attribute.

- *ConstantValueAttribute*
  `public` **ConstantValueAttribute**`( boolean` **value** `)`
  - **Usage**
    - Creates a ConstantValue attribute with a boolean value.

- *ConstantValueAttribute*
  `public` **ConstantValueAttribute**`( byte` **value** `)`
  - **Usage**
    - Creates a ConstantValue attribute with a byte value.

- *ConstantValueAttribute*
  `public` **ConstantValueAttribute**`( char` **value** `)`
  - **Usage**
    - Creates a ConstantValue attribute with a char value.

- *ConstantValueAttribute*
  `public` **ConstantValueAttribute**`( double` **value** `)`
  - **Usage**
    - Creates a ConstantValue attribute with a double value.

- *ConstantValueAttribute*
  `public` **ConstantValueAttribute**`( float` **value** `)`

- **Usage**
  - Creates a ConstantValue attribute with a float value.

- *ConstantValueAttribute*
  `public` **ConstantValueAttribute**`( java.io.InputStream` **is**`, org.ldp.jdasm.DConstantPool` **cpool** `)`
  - **Usage**
    - Creates a ConstantValue attribute reading it from an input stream whose cursor must point to the beginning of the attribute just after the two bytes that reference the name in the constant pool.

- *ConstantValueAttribute*
  `public` **ConstantValueAttribute**`( int` **value** `)`
  - **Usage**
    - Creates a ConstantValue attribute with a integer value.

- *ConstantValueAttribute*
  `public` **ConstantValueAttribute**`( long` **value** `)`
  - **Usage**
    - Creates a ConstantValue attribute with a long value.

- *ConstantValueAttribute*
  `public` **ConstantValueAttribute**`( short` **value** `)`
  - **Usage**
    - Creates a ConstantValue attribute with a short value.

- *ConstantValueAttribute*
  `public` **ConstantValueAttribute**`( java.lang.String` **value** `)`
  - **Usage**
    - Creates a ConstantValue attribute with a String value.

Methods

- *build*
  `public int` **build**`( byte [] tofill, int` **atindex** `)`
  - **Usage**
    - It builds the content of this class into a final byte array. Such array is passed as first argument, and the index from where start to insert bytes is the second argument.
  - **Parameters**
    - `tofill` - the array to fill
    - `atindex` - the index from where start to fill
  - **Returns** - the number of byte inserted

- *build*
  `public int` **build**`( java.io.OutputStream` **output** `)`
  - **Usage**
    - It builds the content of this class into an output stream. This method is a faster version of int) since it doesn't need to call the IBuildable#getBuildLength() before.

- **Parameters**
  - `output` - the stream to write onto
- **Returns** - the number of byte inserted
- **Exceptions**
  - `on` - error while writing onto the stream

- *classFileInfo*
  `public String` **classFileInfo**`( int` **indent_level** `)`
  - **Usage**
    - Returns a printable info string

- *constantPoolValueSize*
  `public int` **constantPoolValueSize**`( )`
  - **Usage**
    - Returns the number of values that this constant pool user wants to insert into the constant pool.

- *getBuildLength*
  `public int` **getBuildLength**`( )`
  - **Usage**
    - Returns the length of portion of byte array the IBuildable class is going to create. If this class has instances of other IBuildable classes, then it will return the whole sum of all the various getBuildLength().

- *getConstantValue*
  `public Object` **getConstantValue**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th value that this constant pool user wants to insert into the constant pool.
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *getConstantValueType*
  `public int` **getConstantValueType**`( int` **idx** `)`
  - **Usage**
    - Returns the value type of the idx-th value that this constant pool user wants to insert into the constant pool. The list of the types is declared in DConstantPool
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *getType*
  `public short` **getType**`( )`
  - **Usage**
    - Retrieve the type of the value stored in this ConstantValue attribute.

- *getValue*
  `public Object` **getValue**`( )`

- **Usage**
  - Retrieves the constant stored value.

    The retrieved Object can be of type Long, Float, Double, Integer, Character, Short, Byte, Boolean or String according to the type retrievable with ConstantValueAttribute#getType()

- *setConstantValueIndex*
  `public void` **setConstantValueIndex**`( int` **idx**`, short` **cpool_index** `)`
  - **Usage**
    - The constant pool tells the user that the value retrieved with AConstantPoolUser#getConstantValue(int) at index "idx" has received the "cpool_index" index in the constant pool.
  - **Parameters**
    - `idx` - the index of the value mapped with the ones returned by getConstantValue(int)
    - `cpool_index` - the index of the value in the constant pool

- *setValue*
  `public void` **setValue**`( boolean` **value** `)`
  - **Usage**
    - Change the value of this attribute to a boolean value.

- *setValue*
  `public void` **setValue**`( byte` **value** `)`
  - **Usage**
    - Change the value of this attribute to a byte value.

- *setValue*
  `public void` **setValue**`( char` **value** `)`
  - **Usage**
    - Change the value of this attribute to a char value.

- *setValue*
  `public void` **setValue**`( double` **value** `)`
  - **Usage**
    - Change the value of this attribute to a double value.

- *setValue*
  `public void` **setValue**`( float` **value** `)`
  - **Usage**
    - Change the value of this attribute to a float value.

- *setValue*
  `public void` **setValue**`( int` **value** `)`
  - **Usage**
    - Change the value of this attribute to a integer value.

- *setValue*
  `public void` **setValue**`( long` **value** `)`
  - **Usage**

    - Change the value of this attribute to a long value.

- *setValue*
  `public void` **setValue**`( short` **value** `)`
  - **Usage**
    - Change the value of this attribute to a short value.

- *setValue*
  `public void` **setValue**`( java.lang.String` **value** `)`
  - **Usage**
    - Change the value of this attribute to a string value.

Methods inherited from class `org.ldp.jdasm.DAttribute`

(in A.1.6, page 90)

- *build*
  `public int` **build**`( byte [] tofill, int` **atindex** `)`
  - **Usage**
    - It builds the content of this class into a final byte array. Such array is passed as first argument, and the index from where start to insert bytes is the second argument.
  - **Parameters**
    - `tofill` - the array to fill
    - `atindex` - the index from where start to fill
  - **Returns** - the number of byte inserted

- *build*
  `public int` **build**`( java.io.OutputStream` **output** `)`
  - **Usage**
    - It builds the content of this class into an output stream. This method is a faster version of int) since it doesn't need to call the IBuildable#getBuildLength() before.
  - **Parameters**
    - `output` - the stream to write onto
  - **Returns** - the number of byte inserted
  - **Exceptions**
    - `on` - error while writing onto the stream

- *classFileInfo*
  `public String` **classFileInfo**`( int` **indent_level** `)`
  - **Usage**
    - Returns a printable info string

- *constantPoolUserChildSize*
  `public int` **constantPoolUserChildSize**`( )`
  - **Usage**
    - Returns the number of IConstantPoolUser instances of this class.

- *constantPoolValueSize*
  `public int` **constantPoolValueSize**`( )`
  - **Usage**
    - Returns the number of values that this constant pool user wants to insert into the constant pool.

- *getBuildLength*
  `public int` **getBuildLength**`( )`
  - **Usage**

    * Returns the length of portion of byte array the IBuildable class is going to create. If this class has instances of other IBuildable classes, then it will return the whole sum of all the various getBuildLength().

- *getConstantPoolUserChild*
  `public AConstantPoolUser` **getConstantPoolUserChild**`( int` **idx** `)`
  - **Usage**
    * Returns the idx-th IConstantPoolUser instance of this class.

- *getConstantValue*
  `public Object` **getConstantValue**`( int` **idx** `)`
  - **Usage**
    * Returns the idx-th value that this constant pool user wants to insert into the constant pool.
  - **Exceptions**
    * `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *getConstantValueType*
  `public int` **getConstantValueType**`( int` **idx** `)`
  - **Usage**
    * Returns the value type of the idx-th value that this constant pool user wants to insert into the constant pool. The list of the types is declared in DConstantPool
  - **Exceptions**
    * `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *getDAttributeFromInputStream*
  `public static DAttribute` **getDAttributeFromInputStream**`( java.io.InputStream` **is**`, org.ldp.jdasm.DConstantPool` **cpool** `)`
  - **Usage**
    * This function read an Attribute from an input stream whose cursor must point to the first byte (which specify the index into the constant pool of the name).

- *getDAttributeFromInputStream*
  `public static DAttribute` **getDAttributeFromInputStream**`( java.io.InputStream` **is**`, org.ldp.jdasm.DConstantPool` **cpool**`, org.ldp.jdasm.attribute.CodeAttribute` **code** `)`
  - **Usage**
    * This function read an Attribute from an input stream whose cursor must point to the first byte (which specify the index into the constant pool of the name). If the attribute's parent is a CodeAttribute, it must be specified as argument to allow the LineNumberTableAttribute to have link to instructions.

- *getName*
  `public String` **getName**`( )`
  - **Usage**
    * Get the name of the attribute.

- *setConstantValueIndex*
  `public void` **setConstantValueIndex**`( int` **idx**`, short` **cpool_index** `)`
  - **Usage**
    * The constant pool tells the user that the value retrieved with AConstantPoolUser#getConstantValue(int) at index "idx" has received the "cpool_index" index in the constant pool.
  - **Parameters**
    * `idx` - the index of the value mapped with the ones returned by getConstantValue(int)

    - `cpool_index` - the index of value in the constant pool

- *setName*
  `public void` **setName**`( java.lang.String` **name** `)`
  - **Usage**
    - Set the name of the attribute.

Methods inherited from class
`org.ldp.jdasm.attribute.Attributable`

(in A.3.2, page 136)

- *addAttribute*
  `public DAttribute` **addAttribute**`( org.ldp.jdasm.DAttribute` **attr** `)`
  - **Usage**
    - Adds a DAttribute.
  - **Returns** - the inserted DAttribute
- *attributeCount*
  `public int` **attributeCount**`( )`
  - **Usage**
    - Returns the number of attribute for this class
- *getAttribute*
  `public DAttribute` **getAttribute**`( int` **idx** `)`
  - **Usage**
    - Get the DAttribute at specified position
- *getAttribute*
  `public DAttribute` **getAttribute**`( java.lang.String` **name** `)`
  - **Usage**
    - Returns the DAttribute with specified name, or null if it doesn't exist.
- *getAttributes*
  `public DAttribute` **getAttributes**`( )`
  - **Usage**
    - Returns all the attributes
- *hasAttribute*
  `public boolean` **hasAttribute**`( java.lang.String` **name** `)`
  - **Usage**
    - Returns true if an attribute with given name exists.
- *removeAttribute*
  `public boolean` **removeAttribute**`( org.ldp.jdasm.DAttribute` **attr** `)`
  - **Usage**
    - Removes the specified DAttribute (match on equals())
  - **Returns** - true if the attribute is removed (if it was present), false it it is not present in the list.
- *removeAttribute*
  `public void` **removeAttribute**`( int` **idx** `)`
  - **Usage**
    - Removes the DAttribute at specified position
- *removeAttribute*
  `public boolean` **removeAttribute**`( java.lang.String` **name** `)`
  - **Usage**
    - Removes the DAttribute with specified name. Returns true if it is removed (if it was present), false if it is not present in the list.

Methods inherited from class
`org.ldp.jdasm.constantpool.AConstantPoolUser`

(in A.2.4, page 97)

- *constantPoolUserChildSize*
  `public abstract int` **constantPoolUserChildSize**`( )`
  - **Usage**
    - Returns the number of IConstantPoolUser instances of this class.
- *constantPoolValueSize*
  `public abstract int` **constantPoolValueSize**`( )`
  - **Usage**
    - Returns the number of values that this constant pool user wants to insert into the constant pool.
- *getConstantPoolUserChild*
  `public abstract AConstantPoolUser` **getConstantPoolUserChild**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th IConstantPoolUser instance of this class.
- *getConstantValue*
  `public abstract Object` **getConstantValue**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th value that this constant pool user wants to insert into the constant pool.
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)
- *getConstantValueType*
  `public abstract int` **getConstantValueType**`( int` **idx** `)`
  - **Usage**
    - Returns the value type of the idx-th value that this constant pool user wants to insert into the constant pool. The list of the types is declared in DConstantPool
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)
- *setConstantValueIndex*
  `public abstract void` **setConstantValueIndex**`( int` **idx**`, short` **cpool_index** `)`
  - **Usage**
    - The constant pool tells the user that the value retrieved with AConstantPoolUser#getConstantValue(int) at index "idx" has received the "cpool_index" index in the constant pool.
  - **Parameters**
    - `idx` - the index of the value mapped with the ones returned by getConstantValue(int)
    - `cpool_index` - the index of the value in the constant pool

A.3.5 Class CustomAttribute

This class represents a custom Attribute of the java class file.

You can create your own attribute by specifying its name. It is seen as a fully customizable array of byte.

Declaration

public class CustomAttribute
extends org.ldp.jdasm.DAttribute

Constructors

- *CustomAttribute*
  `public` **CustomAttribute**`( java.lang.String` **name** `)`
  - **Usage**
    * Creates a CustomAttribute with specified name. This will implicitly call DClass#setBuildCustomAttribute(boolean) and turn on the build of custom attributes

- *CustomAttribute*
  `public` **CustomAttribute**`( java.lang.String` **name**`, java.io.InputStream` **is** `)`
  - **Usage**
    * Creates a CustomAttribute attribute reading it from an input stream whose cursor must point to the beginning of the attribute just after the two bytes that reference the name in the constant pool. This will implicitly call DClass#setBuildCustomAttribute(boolean) and turn off the build of custom attributes

Methods

- *build*
  `public int` **build**`( byte []` **tofill**`, int` **atindex** `)`
  - **Usage**
    * It builds the content of this class into a final byte array. Such array is passed as first argument, and the index from where start to insert bytes is the second argument. If the class has been read from a java class file, it can happen that many unrecognized attribute has been loaded as custom attribute. Such attribute can have references into the constant pool, references that cannot be updated by jdasm (not recognizing the attribute), so by default custom attribute are not build. If instead, the constructor a custom attribute CustomAttribute#CustomAttribute(String) is called, then the build is automatically turned on. You can always change this behavior with DClass#setBuildCustomAttribute(boolean)
  - **Parameters**
    * `tofill` - the array to fill
    * `atindex` - the index from where start to fill
  - **Returns** - the number of byte inserted

- *build*
  `public int` **build**`( java.io.OutputStream` **output** `)`
  - **Usage**

    - It builds the content of this class into an output stream. This method is a faster version of int) since it doesn't need to call the IBuildable#getBuildLength() before.
  - **Parameters**
    - `output` - the stream to write onto
  - **Returns** - the number of byte inserted
  - **Exceptions**
    - `on` - error while writing onto the stream

- *classFileInfo*
  `public String` **classFileInfo**`( int` **indent_level** `)`
  - **Usage**
    - Returns a printable info string

- *constantPoolUserChildSize*
  `public int` **constantPoolUserChildSize**`( )`
  - **Usage**
    - Returns the number of IConstantPoolUser instances of this class.

- *constantPoolValueSize*
  `public int` **constantPoolValueSize**`( )`
  - **Usage**
    - Returns the number of values that this constant pool user wants to insert into the constant pool.

- *getBuildLength*
  `public int` **getBuildLength**`( )`
  - **Usage**
    - Returns the length of portion of byte array the IBuildable class is going to create. If this class has instances of other IBuildable classes, then it will return the whole sum of all the various getBuildLength(). If the class has been read from a java class file, it can happen that many unrecognized attribute has been loaded as custom attribute. Such attribute can have references into the constant pool, references that cannot be updated by jdasm (not recognizing the attribute), so by default custom attribute are not build. If instead, the constructor a custom attribute CustomAttribute#CustomAttribute(String) is called, then the build is automatically turned on. You can always change this behavior with DClass#setBuildCustomAttribute(boolean)

Methods inherited from class `org.ldp.jdasm.DAttribute`

(in A.1.6, page 90)

- *build*
  `public int` **build**`( byte [] tofill, int` **atindex** `)`
  - **Usage**
    - It builds the content of this class into a final byte array. Such array is passed as first argument, and the index from where start to insert bytes is the second argument.
  - **Parameters**

    - `tofill` - the array to fill
    - `atindex` - the index from where start to fill
  - **Returns** - the number of byte inserted

- *build*
  `public int` **build**`( java.io.OutputStream` **output** `)`
  - **Usage**
    - It builds the content of this class into an output stream. This method is a faster version of int) since it doesn't need to call the IBuildable#getBuildLength() before.
  - **Parameters**
    - `output` - the stream to write onto
  - **Returns** - the number of byte inserted
  - **Exceptions**
    - `on` - error while writing onto the stream

- *classFileInfo*
  `public String` **classFileInfo**`( int` **indent_level** `)`
  - **Usage**
    - Returns a printable info string

- *constantPoolUserChildSize*
  `public int` **constantPoolUserChildSize**`( )`
  - **Usage**
    - Returns the number of IConstantPoolUser instances of this class.

- *constantPoolValueSize*
  `public int` **constantPoolValueSize**`( )`
  - **Usage**
    - Returns the number of values that this constant pool user wants to insert into the constant pool.

- *getBuildLength*
  `public int` **getBuildLength**`( )`
  - **Usage**
    - Returns the length of portion of byte array the IBuildable class is going to create. If this class has instances of other IBuildable classes, then it will return the whole sum of all the various getBuildLength().

- *getConstantPoolUserChild*
  `public AConstantPoolUser` **getConstantPoolUserChild**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th IConstantPoolUser instance of this class.

- *getConstantValue*
  `public Object` **getConstantValue**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th value that this constant pool user wants to insert into the constant pool.
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *getConstantValueType*
  `public int` **getConstantValueType**`( int` **idx** `)`
  - **Usage**
    - Returns the value type of the idx-th value that this constant pool user wants to insert into the constant pool. The list of the types is declared in DConstantPool
  - **Exceptions**

    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *getDAttributeFromInputStream*
  `public static DAttribute` **getDAttributeFromInputStream**`( java.io.InputStream` **is**`, org.ldp.jdasm.DConstantPool` **cpool** `)`
  - **Usage**
    - This function read an Attribute from an input stream whose cursor must point to the first byte (which specify the index into the constant pool of the name).

- *getDAttributeFromInputStream*
  `public static DAttribute` **getDAttributeFromInputStream**`( java.io.InputStream` **is**`, org.ldp.jdasm.DConstantPool` **cpool**`, org.ldp.jdasm.attribute.CodeAttribute` **code** `)`
  - **Usage**
    - This function read an Attribute from an input stream whose cursor must point to the first byte (which specify the index into the constant pool of the name). If the attribute's parent is a CodeAttribute, it must be specified as argument to allow the LineNumberTableAttribute to have link to instructions.

- *getName*
  `public String` **getName**`( )`
  - **Usage**
    - Get the name of the attribute.

- *setConstantValueIndex*
  `public void` **setConstantValueIndex**`( int` **idx**`, short` **cpool_index** `)`
  - **Usage**
    - The constant pool tells the user that the value retrieved with AConstantPoolUser#getConstantValue(int) at index "idx" has received the "cpool_index" index in the constant pool.
  - **Parameters**
    - `idx` - the index of the value mapped with the ones returned by getConstantValue(int)
    - `cpool_index` - the index of the value in the constant pool

- *setName*
  `public void` **setName**`( java.lang.String` **name** `)`
  - **Usage**
    - Set the name of the attribute.

Methods inherited from class `org.ldp.jdasm.attribute.Attributable`

(in A.3.2, page 136)

- *addAttribute*
  `public DAttribute` **addAttribute**`( org.ldp.jdasm.DAttribute` **attr** `)`
  - **Usage**
    - Adds a DAttribute.
  - **Returns** - the inserted DAttribute

- *attributeCount*
  `public int` **attributeCount**`( )`
  - **Usage**
    - Returns the number of attribute for this class

- *getAttribute*
  `public DAttribute` **getAttribute**`( int` **idx** `)`
  - **Usage**
    - Get the DAttribute at specified position

- *getAttribute*
  `public DAttribute` **getAttribute**`( java.lang.String` **name** `)`
  - **Usage**
    - Returns the DAttribute with specified name, or null if it doesn't exist.

- *getAttributes*
  `public DAttribute` **getAttributes**`( )`
  - **Usage**
    - Returns all the attributes

- *hasAttribute*
  `public boolean` **hasAttribute**`( java.lang.String` **name** `)`
  - **Usage**
    - Returns true if an attribute with given name exists.

- *removeAttribute*
  `public boolean` **removeAttribute**`( org.ldp.jdasm.DAttribute` **attr** `)`
  - **Usage**
    - Removes the specified DAttribute (match on equals())
  - **Returns** - true if the attribute is removed (if it was present), false it it is not present in the list.

- *removeAttribute*
  `public void` **removeAttribute**`( int` **idx** `)`
  - **Usage**
    - Removes the DAttribute at specified position

- *removeAttribute*
  `public boolean` **removeAttribute**`( java.lang.String` **name** `)`
  - **Usage**
    - Removes the DAttribute with specified name. Returns true if it is removed (if it was present), false if it is not present in the list.

Methods inherited from class `org.ldp.jdasm.constantpool.AConstantPoolUser`

(in A.2.4, page 97)

- *constantPoolUserChildSize*
  `public abstract int` **constantPoolUserChildSize**`( )`
  - **Usage**
    - Returns the number of IConstantPoolUser instances of this class.

- *constantPoolValueSize*
  `public abstract int` **constantPoolValueSize**`( )`
  - **Usage**
    - Returns the number of values that this constant pool user wants to insert into the constant pool.

- *getConstantPoolUserChild*
  `public abstract AConstantPoolUser` **getConstantPoolUserChild**`( int` **idx** `)`
  - **Usage**

    - Returns the idx-th IConstantPoolUser instance of this class.

- *getConstantValue*
  `public abstract Object` **getConstantValue**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th value that this constant pool user wants to insert into the constant pool.
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *getConstantValueType*
  `public abstract int` **getConstantValueType**`( int` **idx** `)`
  - **Usage**
    - Returns the value type of the idx-th value that this constant pool user wants to insert into the constant pool. The list of the types is declared in DConstantPool
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *setConstantValueIndex*
  `public abstract void` **setConstantValueIndex**`( int` **idx**`, short` **cpool_index** `)`
  - **Usage**
    - The constant pool tells the user that the value retrieved with AConstantPoolUser#getConstantValue(int) at index "idx" has received the "cpool_index" index in the constant pool.
  - **Parameters**
    - `idx` - the index of the value mapped with the ones returned by getConstantValue(int)
    - `cpool_index` - the index of the value in the constant pool

A.3.6 Class DeprecatedAttribute

This class represents an Attribute of the java class file.

The Deprecated attribute is an optional attribute of DClass , DField , and DMethod classes.

A class, interface, method, or field may be marked using a Deprecated attribute to indicate that the class, interface, method, or field has been superseded.

Declaration

```
public class DeprecatedAttribute
extends org.ldp.jdasm.DAttribute
```

Constructors

- *DeprecatedAttribute*
  `public` **DeprecatedAttribute**`( )`
  - **Usage**
    - Creates a Deprecated attribute.

- *DeprecatedAttribute*
  `public` **DeprecatedAttribute**`( java.io.InputStream` **is**`, org.ldp.jdasm.DConstantPool` **cpool** `)`
  - **Usage**
    * Creates a DeprecatedAttribute attribute reading it from an input stream whose cursor must point to the beginning of the attribute just after the two bytes that reference the name in the constant pool.

METHODS INHERITED FROM CLASS `org.ldp.jdasm.DAttribute`

(in A.1.6, page 90)

- *build*
  `public int` **build**`( byte [] tofill, int` **atindex** `)`
  - **Usage**
    * It builds the content of this class into a final byte array. Such array is passed as first argument, and the index from where start to insert bytes is the second argument.
  - **Parameters**
    * `tofill` - the array to fill
    * `atindex` - the index from where start to fill
  - **Returns** - the number of byte inserted

- *build*
  `public int` **build**`( java.io.OutputStream` **output** `)`
  - **Usage**
    * It builds the content of this class into an output stream. This method is a faster version of int) since it doesn't need to call the IBuildable#getBuildLength() before.
  - **Parameters**
    * `output` - the stream to write onto
  - **Returns** - the number of byte inserted
  - **Exceptions**
    * `on` - error while writing onto the stream

- *classFileInfo*
  `public String` **classFileInfo**`( int` **indent_level** `)`
  - **Usage**
    * Returns a printable info string

- *constantPoolUserChildSize*
  `public int` **constantPoolUserChildSize**`( )`
  - **Usage**
    * Returns the number of IConstantPoolUser instances of this class.

- *constantPoolValueSize*
  `public int` **constantPoolValueSize**`( )`
  - **Usage**
    * Returns the number of values that this constant pool user wants to insert into the constant pool.

- *getBuildLength*
  `public int` **getBuildLength**`( )`
  - **Usage**
    * Returns the length of portion of byte array the IBuildable class is going to create. If this class has instances of other IBuildable classes, then it will return the whole sum of all the various getBuildLength().

- *getConstantPoolUserChild*
  `public AConstantPoolUser` **getConstantPoolUserChild**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th IConstantPoolUser instance of this class.
- *getConstantValue*
  `public Object` **getConstantValue**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th value that this constant pool user wants to insert into the constant pool.
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)
- *getConstantValueType*
  `public int` **getConstantValueType**`( int` **idx** `)`
  - **Usage**
    - Returns the value type of the idx-th value that this constant pool user wants to insert into the constant pool. The list of the types is declared in DConstantPool
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)
- *getDAttributeFromInputStream*
  `public static DAttribute` **getDAttributeFromInputStream**`( java.io.InputStream` **is**`, org.ldp.jdasm.DConstantPool` **cpool** `)`
  - **Usage**
    - This function read an Attribute from an input stream whose cursor must point to the first byte (which specify the index into the constant pool of the name).
- *getDAttributeFromInputStream*
  `public static DAttribute` **getDAttributeFromInputStream**`( java.io.InputStream` **is**`, org.ldp.jdasm.DConstantPool` **cpool**`, org.ldp.jdasm.attribute.CodeAttribute` **code** `)`
  - **Usage**
    - This function read an Attribute from an input stream whose cursor must point to the first byte (which specify the index into the constant pool of the name). If the attribute's parent is a CodeAttribute, it must be specified as argument to allow the LineNumberTableAttribute to have link to instructions.
- *getName*
  `public String` **getName**`( )`
  - **Usage**
    - Get the name of the attribute.
- *setConstantValueIndex*
  `public void` **setConstantValueIndex**`( int` **idx**`, short` **cpool_index** `)`
  - **Usage**
    - The constant pool tells the user that the value retrieved with AConstantPoolUser#getConstantValue(int) at index "idx" has received the "cpool_index" index in the constant pool.
  - **Parameters**
    - `idx` - the index of the value mapped with the ones returned by getConstantValue(int)
    - `cpool_index` - the index of the value in the constant pool
- *setName*
  `public void` **setName**`( java.lang.String` **name** `)`
  - **Usage**
    - Set the name of the attribute.

Methods inherited from class
`org.ldp.jdasm.attribute.Attributable`

(in A.3.2, page 136)

- *addAttribute*
  `public DAttribute` **addAttribute**`( org.ldp.jdasm.DAttribute` **attr** `)`
  - **Usage**
    - Adds a DAttribute.
  - **Returns** - the inserted DAttribute

- *attributeCount*
  `public int` **attributeCount**`( )`
  - **Usage**
    - Returns the number of attribute for this class

- *getAttribute*
  `public DAttribute` **getAttribute**`( int` **idx** `)`
  - **Usage**
    - Get the DAttribute at specified position

- *getAttribute*
  `public DAttribute` **getAttribute**`( java.lang.String` **name** `)`
  - **Usage**
    - Returns the DAttribute with specified name, or null if it doesn't exist.

- *getAttributes*
  `public DAttribute` **getAttributes**`( )`
  - **Usage**
    - Returns all the attributes

- *hasAttribute*
  `public boolean` **hasAttribute**`( java.lang.String` **name** `)`
  - **Usage**
    - Returns true if an attribute with given name exists.

- *removeAttribute*
  `public boolean` **removeAttribute**`( org.ldp.jdasm.DAttribute` **attr** `)`
  - **Usage**
    - Removes the specified DAttribute (match on equals())
  - **Returns** - true if the attribute is removed (if it was present), false it it is not present in the list.

- *removeAttribute*
  `public void` **removeAttribute**`( int` **idx** `)`
  - **Usage**
    - Removes the DAttribute at specified position

- *removeAttribute*
  `public boolean` **removeAttribute**`( java.lang.String` **name** `)`
  - **Usage**
    - Removes the DAttribute with specified name. Returns true if it is removed (if it was present), false if it is not present in the list.

Methods inherited from class
`org.ldp.jdasm.constantpool.AConstantPoolUser`

(in A.2.4, page 97)

- *constantPoolUserChildSize*
  `public abstract int` **constantPoolUserChildSize**`( )`
  - **Usage**
    - Returns the number of IConstantPoolUser instances of this class.
- *constantPoolValueSize*
  `public abstract int` **constantPoolValueSize**`( )`
  - **Usage**
    - Returns the number of values that this constant pool user wants to insert into the constant pool.
- *getConstantPoolUserChild*
  `public abstract AConstantPoolUser` **getConstantPoolUserChild**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th IConstantPoolUser instance of this class.
- *getConstantValue*
  `public abstract Object` **getConstantValue**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th value that this constant pool user wants to insert into the constant pool.
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)
- *getConstantValueType*
  `public abstract int` **getConstantValueType**`( int` **idx** `)`
  - **Usage**
    - Returns the value type of the idx-th value that this constant pool user wants to insert into the constant pool. The list of the types is declared in DConstantPool
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)
- *setConstantValueIndex*
  `public abstract void` **setConstantValueIndex**`( int` **idx**`, short` **cpool_index** `)`
  - **Usage**
    - The constant pool tells the user that the value retrieved with AConstantPoolUser#getConstantValue(int) at index "idx" has received the "cpool_index" index in the constant pool.
  - **Parameters**
    - `idx` - the index of the value mapped with the ones returned by getConstantValue(int)
    - `cpool_index` - the index of the value in the constant pool

A.3.7 Class **ExceptionsAttribute**

This class represents an Attribute of the java class file.

The Exceptions attribute is an attribute used in the attributes table of a DMethod class.

The Exceptions attribute indicates which checked exceptions a method may throw; there may be at most one Exceptions attribute in each DMethod. In its internals an ExceptionAttribute is a container of a list of class names

Declaration

public class ExceptionsAttribute
extends org.ldp.jdasm.DAttribute

Constructors

- *ExceptionsAttribute*
  `public` **ExceptionsAttribute**`( )`
  - **Usage**
    - Creates a Exceptions attribute.

- *ExceptionsAttribute*
  `public` **ExceptionsAttribute**`( java.io.InputStream` **is**`, org.ldp.jdasm.DConstantPool` **cpool** `)`
  - **Usage**
    - Creates an ExceptionsAttribute attribute reading it from an input stream whose cursor must point to the beginning of the attribute just after the two bytes that reference the name in the constant pool. The constant pool needs to have all the string constant values used by this attribute.

Methods

- *addException*
  `public void` **addException**`( java.lang.String` **className** `)`
  - **Usage**
    - Adds an exception to the list.
  - **Parameters**
    - `className` - the name of the class the method can throw

- *build*
  `public int` **build**`( byte []` **tofill**`, int` **atindex** `)`
  - **Usage**
    - It builds the content of this class into a final byte array. Such array is passed as first argument, and the index from where start to insert bytes is the second argument.
  - **Parameters**

    - `tofill` - the array to fill
    - `atindex` - the index from where start to fill
  - **Returns** - the number of byte inserted

- *build*
  `public int` **build**`( java.io.OutputStream` **output** `)`
  - **Usage**
    - It builds the content of this class into an output stream. This method is a faster version of int) since it doesn't need to call the IBuildable#getBuildLength() before.
  - **Parameters**
    - `output` - the stream to write onto
  - **Returns** - the number of byte inserted
  - **Exceptions**
    - `on` - error while writing onto the stream

- *classFileInfo*
  `public String` **classFileInfo**`( int` **indent_level** `)`
  - **Usage**
    - Returns a printable info string

- *constantPoolValueSize*
  `public int` **constantPoolValueSize**`( )`
  - **Usage**
    - Returns the number of values that this constant pool user wants to insert into the constant pool.

- *exceptionCount*
  `public int` **exceptionCount**`( )`
  - **Usage**
    - Returns the number of class names in the list.

- *getBuildLength*
  `public int` **getBuildLength**`( )`
  - **Usage**
    - Returns the length of portion of byte array the IBuildable class is going to create. If this class has instances of other IBuildable classes, then it will return the whole sum of all the various getBuildLength().

- *getConstantValue*
  `public Object` **getConstantValue**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th value that this constant pool user wants to insert into the constant pool.
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *getConstantValueType*
  `public int` **getConstantValueType**`( int` **idx** `)`
  - **Usage**

    - Returns the value type of the idx-th value that this constant pool user wants to insert into the constant pool. The list of the types is declared in DConstantPool
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *getExceptions*
  `public String` **getExceptions**`( )`
  - **Usage**
    - Returns the name of all the exceptions.

- *getExceptions*
  `public String` **getExceptions**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th name of the exception.

- *removeException*
  `public boolean` **removeException**`( java.lang.String` **className** `)`
  - **Usage**
    - Removes an exception from the list.
  - **Parameters**
    - `className` - the name of the class throwable
  - **Returns** - true if such class is removed, false if it is not present in the list

---

- *setConstantValueIndex*
  `public void` **setConstantValueIndex**`( int` **idx**`, short` **cpool_index** `)`
  - **Usage**
    - The constant pool tells the user that the value retrieved with AConstantPoolUser#getConstantValue(int) at index "idx" has received the "cpool_index" index in the constant pool.
  - **Parameters**
    - `idx` - the index of the value mapped with the ones returned by getConstantValue(int)
    - `cpool_index` - the index of the value in the constant pool

Methods inherited from class `org.ldp.jdasm.DAttribute`

(in A.1.6, page 90)

- *build*
  `public int` **build**`( byte [] tofill, int` **atindex** `)`
  - **Usage**
    - It builds the content of this class into a final byte array. Such array is passed as first argument, and the index from where start to insert bytes is the second argument.
  - **Parameters**
    - `tofill` - the array to fill
    - `atindex` - the index from where start to fill
  - **Returns** - the number of byte inserted

- *build*
  `public int` **build**`( java.io.OutputStream` **output** `)`
  - **Usage**
    - It builds the content of this class into an output stream. This method is a faster version of int) since it doesn't need to call the IBuildable#getBuildLength() before.
  - **Parameters**
    - `output` - the stream to write onto
  - **Returns** - the number of byte inserted
  - **Exceptions**
    - `on` - error while writing onto the stream

- *classFileInfo*
  `public String` **classFileInfo**`( int` **indent_level** `)`
  - **Usage**
    - Returns a printable info string

- *constantPoolUserChildSize*
  `public int` **constantPoolUserChildSize**`( )`
  - **Usage**
    - Returns the number of IConstantPoolUser instances of this class.

- *constantPoolValueSize*
  `public int` **constantPoolValueSize**`( )`
  - **Usage**
    - Returns the number of values that this constant pool user wants to insert into the constant pool.

- *getBuildLength*
  `public int` **getBuildLength**`( )`
  - **Usage**
    - Returns the length of portion of byte array the IBuildable class is going to create. If this class has instances of other IBuildable classes, then it will return the whole sum of all the various getBuildLength().

- *getConstantPoolUserChild*
  `public AConstantPoolUser` **getConstantPoolUserChild**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th IConstantPoolUser instance of this class.

- *getConstantValue*
  `public Object` **getConstantValue**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th value that this constant pool user wants to insert into the constant pool.
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *getConstantValueType*
  `public int` **getConstantValueType**`( int` **idx** `)`
  - **Usage**
    - Returns the value type of the idx-th value that this constant pool user wants to insert into the constant pool. The list of the types is declared in DConstantPool
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *getDAttributeFromInputStream*
  `public static DAttribute` **getDAttributeFromInputStream(** `java.io.InputStream` **is,** `org.ldp.jdasm.DConstantPool` **cpool**)
  - **Usage**
    - This function read an Attribute from an input stream whose cursor must point to the first byte (which specify the index into the constant pool of the name).

- *getDAttributeFromInputStream*
  `public static DAttribute` **getDAttributeFromInputStream(** `java.io.InputStream` **is,** `org.ldp.jdasm.DConstantPool` **cpool,** `org.ldp.jdasm.attribute.CodeAttribute` **code**)
  - **Usage**
    - This function read an Attribute from an input stream whose cursor must point to the first byte (which specify the index into the constant pool of the name). If the attribute's parent is a CodeAttribute, it must be specified as argument to allow the LineNumberTableAttribute to have link to instructions.

- *getName*
  `public String` **getName(**)
  - **Usage**
    - Get the name of the attribute.

- *setConstantValueIndex*
  `public void` **setConstantValueIndex(** `int` **idx,** `short` **cpool_index**)
  - **Usage**
    - The constant pool tells the user that the value retrieved with AConstantPoolUser#getConstantValue(int) at index "idx" has received the "cpool_index" index in the constant pool.
  - **Parameters**
    - `idx` - the index of the value mapped with the ones returned by getConstantValue(int)
    - `cpool_index` - the index of the value in the constant pool

- *setName*
  `public void` **setName(** `java.lang.String` **name**)
  - **Usage**
    - Set the name of the attribute.

Methods inherited from class
`org.ldp.jdasm.attribute.Attributable`

(in A.3.2, page 136)

- *addAttribute*
  `public DAttribute` **addAttribute(** `org.ldp.jdasm.DAttribute` **attr**)
  - **Usage**
    - Adds a DAttribute.
  - **Returns** - the inserted DAttribute

- *attributeCount*
  `public int` **attributeCount(**)
  - **Usage**
    - Returns the number of attribute for this class

- *getAttribute*
  `public DAttribute` **getAttribute(** `int` **idx**)

  - **Usage**
    - Get the DAttribute at specified position

- *getAttribute*
  `public DAttribute` **getAttribute**`( java.lang.String` **name** `)`
  - **Usage**
    - Returns the DAttribute with specified name, or null if it doesn't exist.

- *getAttributes*
  `public DAttribute` **getAttributes**`( )`
  - **Usage**
    - Returns all the attributes

- *hasAttribute*
  `public boolean` **hasAttribute**`( java.lang.String` **name** `)`
  - **Usage**
    - Returns true if an attribute with given name exists.

- *removeAttribute*
  `public boolean` **removeAttribute**`( org.ldp.jdasm.DAttribute` **attr** `)`
  - **Usage**
    - Removes the specified DAttribute (match on equals())
  - **Returns** - true if the attribute is removed (if it was present), false it it is not present in the list.

- *removeAttribute*
  `public void` **removeAttribute**`( int` **idx** `)`
  - **Usage**
    - Removes the DAttribute at specified position

- *removeAttribute*
  `public boolean` **removeAttribute**`( java.lang.String` **name** `)`
  - **Usage**
    - Removes the DAttribute with specified name. Returns true if it is removed (if it was present), false if it is not present in the list.

Methods inherited from class `org.ldp.jdasm.constantpool.AConstantPoolUser`

(in A.2.4, page 97)

- *constantPoolUserChildSize*
  `public abstract int` **constantPoolUserChildSize**`( )`
  - **Usage**
    - Returns the number of IConstantPoolUser instances of this class.

- *constantPoolValueSize*
  `public abstract int` **constantPoolValueSize**`( )`
  - **Usage**
    - Returns the number of values that this constant pool user wants to insert into the constant pool.

- *getConstantPoolUserChild*
  `public abstract AConstantPoolUser` **getConstantPoolUserChild**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th IConstantPoolUser instance of this class.

- *getConstantValue*
  `public abstract Object` **getConstantValue**`( int` **idx** `)`

  - **Usage**
    * Returns the idx-th value that this constant pool user wants to insert into the constant pool.
  - **Exceptions**
    * `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *getConstantValueType*
  `public abstract int` **getConstantValueType**`( int` **idx** `)`
  - **Usage**
    * Returns the value type of the idx-th value that this constant pool user wants to insert into the constant pool. The list of the types is declared in DConstantPool
  - **Exceptions**
    * `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *setConstantValueIndex*
  `public abstract void` **setConstantValueIndex**`( int` **idx**`, short` **cpool_index** `)`
  - **Usage**
    * The constant pool tells the user that the value retrieved with AConstantPoolUser#getConstantValue(int) at index "idx" has received the "cpool_index" index in the constant pool.
  - **Parameters**
    * `idx` - the index of the value mapped with the ones returned by getConstantValue(int)
    * `cpool_index` - the index of the value in the constant pool

A.3.8 Class InnerClassesAttribute

This class represents an Attribute of the java class file.

This attribute uses an inner class InnerClassesElement as element of its table.

Declaration

```
public class InnerClassesAttribute
extends org.ldp.jdasm.DAttribute
```

Constructors

- *InnerClassesAttribute*
  `public` **InnerClassesAttribute**`( )`
  - **Usage**
    * Creates an empty LineNumberTable attribute.

- *InnerClassesAttribute*
  `public` **InnerClassesAttribute**`( java.io.InputStream` **is**`, org.ldp.jdasm.DConstantPool` **cpool** `)`
  - **Usage**
    * Creates a InnerClassesAttribute attribute reading it from an input stream whose cursor must point to the beginning of the attribute just after the two bytes that reference the name in the constant pool. The constant pool needs to have all the string constant values used by this attribute.

Methods

- *add*
  `public void` **add**`( org.ldp.jdasm.attribute.InnerClassesElement` **item** `)`
  - **Usage**
    - Add a InnerClassesElement element to the table

- *build*
  `public int` **build**`( byte []` **tofill**`, int` **atindex** `)`
  - **Usage**
    - It builds the content of this class into a final byte array. Such array is passed as first argument, and the index from where start to insert bytes is the second argument.
  - **Parameters**
    - `tofill` - the array to fill
    - `atindex` - the index from where start to fill
  - **Returns** - the number of byte inserted

- *build*
  `public int` **build**`( java.io.OutputStream` **output** `)`
  - **Usage**
    - It builds the content of this class into an output stream. This method is a faster version of int) since it doesn't need to call the IBuildable#getBuildLength() before.
  - **Parameters**
    - `output` - the stream to write onto
  - **Returns** - the number of byte inserted
  - **Exceptions**
    - `on` - error while writing onto the stream

- *classFileInfo*
  `public String` **classFileInfo**`( int` **indent_level** `)`
  - **Usage**
    - Returns a printable info string

- *clearElements*
  `public void` **clearElements**`( )`
  - **Usage**
    - Clears all the items.

- *constantPoolUserChildSize*
  `public int` **constantPoolUserChildSize**`( )`
  - **Usage**
    - Returns the number of IConstantPoolUser instances of this class.

- *elementAt*
  `public InnerClassesElement` **elementAt**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th InnerClassesElement of the table

- *getBuildLength*
  `public int` **getBuildLength**`( )`
  - **Usage**
    - Returns the length of portion of byte array the IBuildable class is going to create. If this class has instances of other IBuildable classes, then it will return the whole sum of all the various getBuildLength().
- *getConstantPoolUserChild*
  `public AConstantPoolUser` **getConstantPoolUserChild**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th IConstantPoolUser instance of this class.
- *getElementSize*
  `public int` **getElementSize**`( )`
  - **Usage**
    - Returns the number of items in this attribute.

Methods inherited from class `org.ldp.jdasm.DAttribute`

(in A.1.6, page 90)

- *build*
  `public int` **build**`( byte []` **tofill**`, int` **atindex** `)`
  - **Usage**
    - It builds the content of this class into a final byte array. Such array is passed as first argument, and the index from where start to insert bytes is the second argument.
  - **Parameters**
    - `tofill` - the array to fill
    - `atindex` - the index from where start to fill
  - **Returns** - the number of byte inserted
- *build*
  `public int` **build**`( java.io.OutputStream` **output** `)`
  - **Usage**
    - It builds the content of this class into an output stream. This method is a faster version of int) since it doesn't need to call the IBuildable#getBuildLength() before.
  - **Parameters**
    - `output` - the stream to write onto
  - **Returns** - the number of byte inserted
  - **Exceptions**
    - `on` - error while writing onto the stream
- *classFileInfo*
  `public String` **classFileInfo**`( int` **indent_level** `)`
  - **Usage**
    - Returns a printable info string
- *constantPoolUserChildSize*
  `public int` **constantPoolUserChildSize**`( )`
  - **Usage**
    - Returns the number of IConstantPoolUser instances of this class.

- *constantPoolValueSize*
  `public int` **constantPoolValueSize**`( )`
  - **Usage**
    - Returns the number of values that this constant pool user wants to insert into the constant pool.
- *getBuildLength*
  `public int` **getBuildLength**`( )`
  - **Usage**
    - Returns the length of portion of byte array the IBuildable class is going to create. If this class has instances of other IBuildable classes, then it will return the whole sum of all the various getBuildLength().
- *getConstantPoolUserChild*
  `public AConstantPoolUser` **getConstantPoolUserChild**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th IConstantPoolUser instance of this class.
- *getConstantValue*
  `public Object` **getConstantValue**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th value that this constant pool user wants to insert into the constant pool.
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)
- *getConstantValueType*
  `public int` **getConstantValueType**`( int` **idx** `)`
  - **Usage**
    - Returns the value type of the idx-th value that this constant pool user wants to insert into the constant pool. The list of the types is declared in DConstantPool
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)
- *getDAttributeFromInputStream*
  `public static DAttribute` **getDAttributeFromInputStream**`(`
  `java.io.InputStream` **is**`, org.ldp.jdasm.DConstantPool` **cpool** `)`
  - **Usage**
    - This function read an Attribute from an input stream whose cursor must point to the first byte (which specify the index into the constant pool of the name).
- *getDAttributeFromInputStream*
  `public static DAttribute` **getDAttributeFromInputStream**`(`
  `java.io.InputStream` **is**`, org.ldp.jdasm.DConstantPool` **cpool**`,`
  `org.ldp.jdasm.attribute.CodeAttribute` **code** `)`
  - **Usage**
    - This function read an Attribute from an input stream whose cursor must point to the first byte (which specify the index into the constant pool of the name). If the attribute's parent is a CodeAttribute, it must be specified as argument to allow the LineNumberTableAttribute to have link to instructions.
- *getName*
  `public String` **getName**`( )`
  - **Usage**
    - Get the name of the attribute.

- *setConstantValueIndex*
  `public void` **setConstantValueIndex**`( int` **idx**`, short` **cpool_index** `)`
  - **Usage**
    * The constant pool tells the user that the value retrieved with AConstantPoolUser#getConstantValue(int) at index "idx" has received the "cpool_index" index in the constant pool.
  - **Parameters**
    * `idx` - the index of the value mapped with the ones returned by getConstantValue(int)
    * `cpool_index` - the index of the value in the constant pool
- *setName*
  `public void` **setName**`( java.lang.String` **name** `)`
  - **Usage**
    * Set the name of the attribute.

Methods inherited from class
`org.ldp.jdasm.attribute.Attributable`

(in A.3.2, page 136)
- *addAttribute*
  `public DAttribute` **addAttribute**`( org.ldp.jdasm.DAttribute` **attr** `)`
  - **Usage**
    * Adds a DAttribute.
  - **Returns** - the inserted DAttribute
- *attributeCount*
  `public int` **attributeCount**`( )`
  - **Usage**
    * Returns the number of attribute for this class
- *getAttribute*
  `public DAttribute` **getAttribute**`( int` **idx** `)`
  - **Usage**
    * Get the DAttribute at specified position
- *getAttribute*
  `public DAttribute` **getAttribute**`( java.lang.String` **name** `)`
  - **Usage**
    * Returns the DAttribute with specified name, or null if it doesn't exist.
- *getAttributes*
  `public DAttribute` **getAttributes**`( )`
  - **Usage**
    * Returns all the attributes
- *hasAttribute*
  `public boolean` **hasAttribute**`( java.lang.String` **name** `)`
  - **Usage**
    * Returns true if an attribute with given name exists.
- *removeAttribute*
  `public boolean` **removeAttribute**`( org.ldp.jdasm.DAttribute` **attr** `)`
  - **Usage**
    * Removes the specified DAttribute (match on equals())

  - **Returns** - true if the attribute is removed (if it was present), false it it is not present in the list.

- *removeAttribute*
  `public void` **removeAttribute**`( int` **idx** `)`
  - **Usage**
    * Removes the DAttribute at specified position

- *removeAttribute*
  `public boolean` **removeAttribute**`( java.lang.String` **name** `)`
  - **Usage**
    * Removes the DAttribute with specified name. Returns true if it is removed (if it was present), false if it is not present in the list.

Methods inherited from class
`org.ldp.jdasm.constantpool.AConstantPoolUser`

(in A.2.4, page 97)

- *constantPoolUserChildSize*
  `public abstract int` **constantPoolUserChildSize**`( )`
  - **Usage**
    * Returns the number of IConstantPoolUser instances of this class.

- *constantPoolValueSize*
  `public abstract int` **constantPoolValueSize**`( )`
  - **Usage**
    * Returns the number of values that this constant pool user wants to insert into the constant pool.

- *getConstantPoolUserChild*
  `public abstract AConstantPoolUser` **getConstantPoolUserChild**`( int` **idx** `)`
  - **Usage**
    * Returns the idx-th IConstantPoolUser instance of this class.

- *getConstantValue*
  `public abstract Object` **getConstantValue**`( int` **idx** `)`
  - **Usage**
    * Returns the idx-th value that this constant pool user wants to insert into the constant pool.
  - **Exceptions**
    * `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *getConstantValueType*
  `public abstract int` **getConstantValueType**`( int` **idx** `)`
  - **Usage**
    * Returns the value type of the idx-th value that this constant pool user wants to insert into the constant pool. The list of the types is declared in DConstantPool
  - **Exceptions**
    * `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *setConstantValueIndex*
  `public abstract void` **setConstantValueIndex**`( int` **idx**`, short` **cpool_index** `)`
  - **Usage**

    * The constant pool tells the user that the value retrieved with AConstantPoolUser#getConstantValue(int) at index "idx" has received the "cpool_index" index in the constant pool.
  - **Parameters**
    * `idx` - the index of the value mapped with the ones returned by getConstantValue(int)
    * `cpool_index` - the index of the value in the constant pool

A.3.9 Class InnerClassesElement

Class used inside the InnerClassesAttribute .

Declaration

public class InnerClassesElement
extends org.ldp.jdasm.constantpool.AConstantPoolUser

Constructors

- *InnerClassesElement*
  `public` **InnerClassesElement**`( )`
  - **Usage**
    * Default empty constructor.

- *InnerClassesElement*
  `public` **InnerClassesElement**`( java.io.InputStream` **is**`, org.ldp.jdasm.DConstantPool` **cpool** `)`
  - **Usage**
    * Creates a InnerClassesElement attribute reading it from an input stream whose cursor must point to the beginning of the attribute just after the two bytes that reference the name in the constant pool. The constant pool needs to have all the string constant values used by this attribute.

- *InnerClassesElement*
  `public` **InnerClassesElement**`( java.lang.String` **innerClass**`, java.lang.String` **outerClass**`, java.lang.String` **innerName**`, short` **accessFlag** `)`
  - **Usage**
    * Constructs a InnerClassesElement with all values specified.

Methods

- *classFileInfo*
  `public String` **classFileInfo**`( int` **indent_level** `)`
  - **Usage**
    * Returns a printable info string

- *constantPoolUserChildSize*
  `public int` **constantPoolUserChildSize**`( )`

  - **Usage**
    * Returns the number of IConstantPoolUser instances of this class.

- *constantPoolValueSize*
  `public int` **constantPoolValueSize**`( )`
  - **Usage**
    * Returns the number of values that this constant pool user wants to insert into the constant pool.

- *getAccessFlag*
  `public short` **getAccessFlag**`( )`
  - **Usage**
    * Get access flags.

- *getConstantPoolUserChild*
  `public AConstantPoolUser` **getConstantPoolUserChild**`( int` **idx** `)`
  - **Usage**
    * Returns the idx-th IConstantPoolUser instance of this class.

- *getConstantValue*
  `public Object` **getConstantValue**`( int` **idx** `)`
  - **Usage**
    * Returns the idx-th value that this constant pool user wants to insert into the constant pool.
  - **Exceptions**
    * `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *getConstantValueType*
  `public int` **getConstantValueType**`( int` **idx** `)`
  - **Usage**
    * Returns the value type of the idx-th value that this constant pool user wants to insert into the constant pool. The list of the types is declared in DConstantPool
  - **Exceptions**
    * `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *getInnerClass*
  `public String` **getInnerClass**`( )`
  - **Usage**
    * Get the inner class name or null.

- *getInnerName*
  `public String` **getInnerName**`( )`
  - **Usage**
    * Get the inner name or null if anonymous.

- *getOuterClass*
  `public String` **getOuterClass**`( )`

- **Usage**
  - Set the outer class name or null.

- *setAccessFlag*
  `public void` **setAccessFlag**`( short` **accessFlag** `)`
  - **Usage**
    - Set access flags.

- *setConstantValueIndex*
  `public void` **setConstantValueIndex**`( int` **idx**`, short` **cpool_index** `)`
  - **Usage**
    - The constant pool tells the user that the value retrieved with AConstantPoolUser#getConstantValue(int) at index "idx" has received the "cpool_index" index in the constant pool.
  - **Parameters**
    - `idx` - the index of the value mapped with the ones returned by getConstantValue(int)
    - `cpool_index` - the index of the value in the constant pool

- *setInnerClass*
  `public void` **setInnerClass**`( java.lang.String` **innerClass** `)`
  - **Usage**
    - Set the inner class name or null.

- *setInnerName*
  `public void` **setInnerName**`( java.lang.String` **innerName** `)`
  - **Usage**
    - Set the inner name or null if anonymous.

- *setOuterClass*
  `public void` **setOuterClass**`( java.lang.String` **outerClass** `)`
  - **Usage**
    - Get the outer class name or null.

Methods inherited from class
`org.ldp.jdasm.constantpool.AConstantPoolUser`

(in A.2.4, page 97)

- *constantPoolUserChildSize*
  `public abstract int` **constantPoolUserChildSize**`( )`
  - **Usage**
    - Returns the number of IConstantPoolUser instances of this class.

- *constantPoolValueSize*
  `public abstract int` **constantPoolValueSize**`( )`
  - **Usage**
    - Returns the number of values that this constant pool user wants to insert into the constant pool.

- *getConstantPoolUserChild*
  `public abstract AConstantPoolUser` **getConstantPoolUserChild**`( int` **idx** `)`

  - **Usage**
    - Returns the idx-th IConstantPoolUser instance of this class.

- *getConstantValue*
  `public abstract Object` **getConstantValue**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th value that this constant pool user wants to insert into the constant pool.
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *getConstantValueType*
  `public abstract int` **getConstantValueType**`( int` **idx** `)`
  - **Usage**
    - Returns the value type of the idx-th value that this constant pool user wants to insert into the constant pool. The list of the types is declared in DConstantPool
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *setConstantValueIndex*
  `public abstract void` **setConstantValueIndex**`( int` **idx**`, short` **cpool_index** `)`
  - **Usage**
    - The constant pool tells the user that the value retrieved with AConstantPoolUser#getConstantValue(int) at index "idx" has received the "cpool_index" index in the constant pool.
  - **Parameters**
    - `idx` - the index of the value mapped with the ones returned by getConstantValue(int)
    - `cpool_index` - the index of the value in the constant pool

A.3.10 Class LineNumberTableAttribute

This class represents an Attribute of the java class file.

Declaration

public class LineNumberTableAttribute
extends org.ldp.jdasm.DAttribute

Constructors

- *LineNumberTableAttribute*
  `public` **LineNumberTableAttribute**`( )`
  - **Usage**
    - Creates an empty LineNumberTable attribute.

- *LineNumberTableAttribute*
  `public` **LineNumberTableAttribute**`( java.io.InputStream` **is**`, org.ldp.jdasm.DConstantPool` **cpool**`, org.ldp.jdasm.attribute.CodeAttribute` **code** `)`

- **Usage**
  - Creates a LineNumberTableAttribute attribute reading it from an input stream whose cursor must point to the beginning of the attribute just after the two bytes that reference the name in the constant pool. The constant pool needs to have all the string constant values used by this attribute.

    In order to have a link with instructions, this constructor needs the filled CodeAttribute it refers to.

Methods

- *addLineNumber*
  `public void` **addLineNumber**`(`
  `org.ldp.jdasm.attribute.instruction.Instruction` **start_pc**`,`
  `int` **line_number** `)`
  - **Usage**
    - Adds a line number information for specified start pc.

- *build*
  `public int` **build**`( byte [] tofill, int` **atindex** `)`
  - **Usage**
    - It builds the content of this class into a final byte array. Such array is passed as first argument, and the index from where start to insert bytes is the second argument.
  - **Parameters**
    - `tofill` - the array to fill
    - `atindex` - the index from where start to fill
  - **Returns** - the number of byte inserted

- *build*
  `public int` **build**`( java.io.OutputStream` **output** `)`
  - **Usage**
    - It builds the content of this class into an output stream. This method is a faster version of int) since it doesn't need to call the IBuildable#getBuildLength() before.
  - **Parameters**
    - `output` - the stream to write onto
  - **Returns** - the number of byte inserted
  - **Exceptions**
    - `on` - error while writing onto the stream

- *classFileInfo*
  `public String` **classFileInfo**`( int` **indent_level** `)`
  - **Usage**
    - Returns a printable info string

- *clearElements*
  `public void` **clearElements**`( )`
  - **Usage**
    - Clears all the items.

- *getBuildLength*
  `public int` **getBuildLength**`( )`
  - **Usage**
    - Returns the length of portion of byte array the IBuildable class is going to create. If this class has instances of other IBuildable classes, then it will return the whole sum of all the various getBuildLength().

- *getElementSize*
  `public int` **getElementSize**`( )`
  - **Usage**
    - Returns the number of items in this attribute.

- *getLineNumber*
  `public short` **getLineNumber**`( int` **idx** `)`
  - **Usage**
    - Get the idx-th item's line number

- *getStartPc*
  `public Instruction` **getStartPc**`( int` **idx** `)`
  - **Usage**
    - Get the idx-th item's start pc

- *removeInstruction*
  `public boolean` **removeInstruction**`(`
  `org.ldp.jdasm.attribute.instruction.Instruction` **instruction**
  `)`
  - **Usage**
    - Search for given `instruction`: if it's found, its line information is removed and true is returned, otherwise false is returned.

Methods inherited from class `org.ldp.jdasm.DAttribute`

(in A.1.6, page 90)

- *build*
  `public int` **build**`( byte []` **tofill**`, int` **atindex** `)`
  - **Usage**
    - It builds the content of this class into a final byte array. Such array is passed as first argument, and the index from where start to insert bytes is the second argument.
  - **Parameters**
    - `tofill` - the array to fill
    - `atindex` - the index from where start to fill
  - **Returns** - the number of byte inserted

- *build*
  `public int` **build**`( java.io.OutputStream` **output** `)`
  - **Usage**
    - It builds the content of this class into an output stream. This method is a faster version of int) since it doesn't need to call the IBuildable#getBuildLength() before.
  - **Parameters**
    - `output` - the stream to write onto

  - **Returns** - the number of byte inserted
  - **Exceptions**
    * `on` - error while writing onto the stream

- *classFileInfo*
  `public String` **classFileInfo**`( int` **indent_level** `)`
  - **Usage**
    * Returns a printable info string

- *constantPoolUserChildSize*
  `public int` **constantPoolUserChildSize**`( )`
  - **Usage**
    * Returns the number of IConstantPoolUser instances of this class.

- *constantPoolValueSize*
  `public int` **constantPoolValueSize**`( )`
  - **Usage**
    * Returns the number of values that this constant pool user wants to insert into the constant pool.

- *getBuildLength*
  `public int` **getBuildLength**`( )`
  - **Usage**
    * Returns the length of portion of byte array the IBuildable class is going to create. If this class has instances of other IBuildable classes, then it will return the whole sum of all the various getBuildLength().

- *getConstantPoolUserChild*
  `public AConstantPoolUser` **getConstantPoolUserChild**`( int` **idx** `)`
  - **Usage**
    * Returns the idx-th IConstantPoolUser instance of this class.

- *getConstantValue*
  `public Object` **getConstantValue**`( int` **idx** `)`
  - **Usage**
    * Returns the idx-th value that this constant pool user wants to insert into the constant pool.
  - **Exceptions**
    * `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *getConstantValueType*
  `public int` **getConstantValueType**`( int` **idx** `)`
  - **Usage**
    * Returns the value type of the idx-th value that this constant pool user wants to insert into the constant pool. The list of the types is declared in DConstantPool
  - **Exceptions**
    * `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *getDAttributeFromInputStream*
  `public static DAttribute` **getDAttributeFromInputStream**`(`
  `java.io.InputStream` **is**`, org.ldp.jdasm.DConstantPool` **cpool** `)`
  - **Usage**
    * This function read an Attribute from an input stream whose cursor must point to the first byte (which specify the index into the constant pool of the name).

- *getDAttributeFromInputStream*
  `public static DAttribute` **getDAttributeFromInputStream**`( java.io.InputStream` **is**`, org.ldp.jdasm.DConstantPool` **cpool**`, org.ldp.jdasm.attribute.CodeAttribute` **code** `)`
  - **Usage**
    - This function read an Attribute from an input stream whose cursor must point to the first byte (which specify the index into the constant pool of the name). If the attribute's parent is a CodeAttribute, it must be specified as argument to allow the LineNumberTableAttribute to have link to instructions.

- *getName*
  `public String` **getName**`( )`
  - **Usage**
    - Get the name of the attribute.

- *setConstantValueIndex*
  `public void` **setConstantValueIndex**`( int` **idx**`, short` **cpool_index** `)`
  - **Usage**
    - The constant pool tells the user that the value retrieved with AConstantPoolUser#getConstantValue(int) at index "idx" has received the "cpool_index" index in the constant pool.
  - **Parameters**
    - `idx` - the index of the value mapped with the ones returned by getConstantValue(int)
    - `cpool_index` - the index of the value in the constant pool

- *setName*
  `public void` **setName**`( java.lang.String` **name** `)`
  - **Usage**
    - Set the name of the attribute.

Methods inherited from class
`org.ldp.jdasm.attribute.Attributable`

(in A.3.2, page 136)

- *addAttribute*
  `public DAttribute` **addAttribute**`( org.ldp.jdasm.DAttribute` **attr** `)`
  - **Usage**
    - Adds a DAttribute.
  - **Returns** - the inserted DAttribute

- *attributeCount*
  `public int` **attributeCount**`( )`
  - **Usage**
    - Returns the number of attribute for this class

- *getAttribute*
  `public DAttribute` **getAttribute**`( int` **idx** `)`
  - **Usage**
    - Get the DAttribute at specified position

- *getAttribute*
  `public DAttribute` **getAttribute**`( java.lang.String` **name** `)`
  - **Usage**
    - Returns the DAttribute with specified name, or null if it doesn't exist.

- *getAttributes*
  `public DAttribute` **getAttributes**`( )`
  - **Usage**
    - Returns all the attributes

- *hasAttribute*
  `public boolean` **hasAttribute**`( java.lang.String` **name** `)`
  - **Usage**
    - Returns true if an attribute with given name exists.

- *removeAttribute*
  `public boolean` **removeAttribute**`( org.ldp.jdasm.DAttribute` **attr** `)`
  - **Usage**
    - Removes the specified DAttribute (match on equals())
  - **Returns** - true if the attribute is removed (if it was present), false it it is not present in the list.

- *removeAttribute*
  `public void` **removeAttribute**`( int` **idx** `)`
  - **Usage**
    - Removes the DAttribute at specified position

- *removeAttribute*
  `public boolean` **removeAttribute**`( java.lang.String` **name** `)`
  - **Usage**
    - Removes the DAttribute with specified name. Returns true if it is removed (if it was present), false if it is not present in the list.

Methods inherited from class `org.ldp.jdasm.constantpool.AConstantPoolUser`

(in A.2.4, page 97)

- *constantPoolUserChildSize*
  `public abstract int` **constantPoolUserChildSize**`( )`
  - **Usage**
    - Returns the number of IConstantPoolUser instances of this class.

- *constantPoolValueSize*
  `public abstract int` **constantPoolValueSize**`( )`
  - **Usage**
    - Returns the number of values that this constant pool user wants to insert into the constant pool.

- *getConstantPoolUserChild*
  `public abstract AConstantPoolUser` **getConstantPoolUserChild**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th IConstantPoolUser instance of this class.

- *getConstantValue*
  `public abstract Object` **getConstantValue**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th value that this constant pool user wants to insert into the constant pool.
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *getConstantValueType*
  `public abstract int` **getConstantValueType**`( int` **idx** `)`
  - **Usage**
    - Returns the value type of the idx-th value that this constant pool user wants to insert into the constant pool. The list of the types is declared in DConstantPool
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *setConstantValueIndex*
  `public abstract void` **setConstantValueIndex**`( int` **idx**`, short` **cpool_index** `)`
  - **Usage**
    - The constant pool tells the user that the value retrieved with AConstantPoolUser#getConstantValue(int) at index "idx" has received the "cpool_index" index in the constant pool.
  - **Parameters**
    - `idx` - the index of the value mapped with the ones returned by getConstantValue(int)
    - `cpool_index` - the index of the value in the constant pool

A.3.11 Class LocalVariableTableAttribute

This class represents an Attribute of the java class file.

This attribute uses the class LocalVariableTableElement as element of its table: each of these element has information about the starting instruction, the length in byte of the variable's scope, the name of the variable and its descriptor.. all mapped to an index into the local variable array. If you really want to use this manually to add debugging information to your class see Instruction#getInstructionOffset() to correctly fit in `start_pc` and `length` fields.

Declaration

public class LocalVariableTableAttribute
extends org.ldp.jdasm.DAttribute

Constructors

- *LocalVariableTableAttribute*
  `public` **LocalVariableTableAttribute**`( )`
  - **Usage**
    - Creates an empty LocalVariableTable attribute.

- *LocalVariableTableAttribute*
  `public` **LocalVariableTableAttribute**`( java.io.InputStream` **is**`, org.ldp.jdasm.DConstantPool` **cpool**`, org.ldp.jdasm.attribute.CodeAttribute` **code** `)`
  - **Usage**

    * Creates a LocalVariableTableAttribute attribute reading it from an input stream whose cursor must point to the beginning of the attribute just after the two bytes that reference the name in the constant pool. The constant pool needs to have all the string constant values used by this attribute. In order to have a link with instructions, this constructor needs the filled CodeAttribute it refers to.

Methods

- *add*
  `public void` **add**`(`
  `org.ldp.jdasm.attribute.LocalVariableTableElement` **item** `)`
  - **Usage**
    * Add a LocalVariableTableElement element to the table.

- *build*
  `public int` **build**`( byte []` **tofill**`, int` **atindex** `)`
  - **Usage**
    * It builds the content of this class into a final byte array. Such array is passed as first argument, and the index from where start to insert bytes is the second argument.
  - **Parameters**
    * `tofill` - the array to fill
    * `atindex` - the index from where start to fill
  - **Returns** - the number of byte inserted

- *build*
  `public int` **build**`( java.io.OutputStream` **output** `)`
  - **Usage**
    * It builds the content of this class into an output stream. This method is a faster version of int) since it doesn't need to call the IBuildable#getBuildLength() before.
  - **Parameters**
    * `output` - the stream to write onto
  - **Returns** - the number of byte inserted
  - **Exceptions**
    * `on` - error while writing onto the stream

- *classFileInfo*
  `public String` **classFileInfo**`( int` **indent_level** `)`
  - **Usage**
    * Returns a printable info string

- *clearElements*
  `public void` **clearElements**`( )`
  - **Usage**
    * Clears all the items.

- *constantPoolValueSize*
  `public int` **constantPoolValueSize**`( )`

  - **Usage**
    * Returns the number of values that this constant pool user wants to insert into the constant pool.

- *getBuildLength*
  `public int` **getBuildLength**`( )`
  - **Usage**
    * Returns the length of portion of byte array the IBuildable class is going to create. If this class has instances of other IBuildable classes, then it will return the whole sum of all the various getBuildLength().

- *getConstantValue*
  `public Object` **getConstantValue**`( int` **idx** `)`
  - **Usage**
    * Returns the idx-th value that this constant pool user wants to insert into the constant pool.
  - **Exceptions**
    * `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *getConstantValueType*
  `public int` **getConstantValueType**`( int` **idx** `)`
  - **Usage**
    * Returns the value type of the idx-th value that this constant pool user wants to insert into the constant pool. The list of the types is declared in DConstantPool
  - **Exceptions**
    * `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *getElement*
  `public LocalVariableTableElement` **getElement**`( int` **idx** `)`
  - **Usage**
    * Returns the idx-th LocalVariableTableElement of the table.

- *getElements*
  `public LocalVariableTableElement` **getElements**`( )`
  - **Usage**
    * Returns all the LocalVariableTableElement of the table.

- *getElementSize*
  `public int` **getElementSize**`( )`
  - **Usage**
    * Returns the number of items in this attribute.

- *removeInstruction*
  `public boolean` **removeInstruction**`(`
  `org.ldp.jdasm.attribute.instruction.Instruction` **instruction**`,`
  `org.ldp.jdasm.attribute.CodeAttribute` **code** `)`
  - **Usage**

    - Search for given `instruction` as starting of a local variable information: if it's found, this information is removed and true is returned, otherwise false is returned. If this instruction is found to be an end scope instruction, some adjustment is done: the new ending instruction becomes the previous one (`code` is used for this purpose).

- *setConstantValueIndex*
  `public void` **setConstantValueIndex**`( int` **idx**`, short` **cpool_index** `)`
  - **Usage**
    - The constant pool tells the user that the value retrieved with AConstantPoolUser#getConstantValue(int) at index "idx" has received the "cpool_index" index in the constant pool.
  - **Parameters**
    - `idx` - the index of the value mapped with the ones returned by getConstantValue(int)
    - `cpool_index` - the index of the value in the constant pool

METHODS INHERITED FROM CLASS `org.ldp.jdasm.DAttribute`

(in A.1.6, page 90)

- *build*
  `public int` **build**`( byte []` **tofill**`, int` **atindex** `)`
  - **Usage**
    - It builds the content of this class into a final byte array. Such array is passed as first argument, and the index from where start to insert bytes is the second argument.
  - **Parameters**
    - `tofill` - the array to fill
    - `atindex` - the index from where start to fill
  - **Returns** - the number of byte inserted

- *build*
  `public int` **build**`( java.io.OutputStream` **output** `)`
  - **Usage**
    - It builds the content of this class into an output stream. This method is a faster version of int) since it doesn't need to call the IBuildable#getBuildLength() before.
  - **Parameters**
    - `output` - the stream to write onto
  - **Returns** - the number of byte inserted
  - **Exceptions**
    - `on` - error while writing onto the stream

- *classFileInfo*
  `public String` **classFileInfo**`( int` **indent_level** `)`
  - **Usage**
    - Returns a printable info string

- *constantPoolUserChildSize*
  `public int` **constantPoolUserChildSize**`( )`
  - **Usage**
    - Returns the number of IConstantPoolUser instances of this class.

- *constantPoolValueSize*
  `public int` **constantPoolValueSize**`( )`

  - **Usage**
    - Returns the number of values that this constant pool user wants to insert into the constant pool.

- *getBuildLength*
  `public int` **getBuildLength**`( )`
  - **Usage**
    - Returns the length of portion of byte array the IBuildable class is going to create. If this class has instances of other IBuildable classes, then it will return the whole sum of all the various getBuildLength().

- *getConstantPoolUserChild*
  `public AConstantPoolUser` **getConstantPoolUserChild**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th IConstantPoolUser instance of this class.

- *getConstantValue*
  `public Object` **getConstantValue**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th value that this constant pool user wants to insert into the constant pool.
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *getConstantValueType*
  `public int` **getConstantValueType**`( int` **idx** `)`
  - **Usage**
    - Returns the value type of the idx-th value that this constant pool user wants to insert into the constant pool. The list of the types is declared in DConstantPool
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *getDAttributeFromInputStream*
  `public static DAttribute` **getDAttributeFromInputStream**`(`
  `java.io.InputStream` **is**`, org.ldp.jdasm.DConstantPool` **cpool** `)`
  - **Usage**
    - This function read an Attribute from an input stream whose cursor must point to the first byte (which specify the index into the constant pool of the name).

- *getDAttributeFromInputStream*
  `public static DAttribute` **getDAttributeFromInputStream**`(`
  `java.io.InputStream` **is**`, org.ldp.jdasm.DConstantPool` **cpool**`,`
  `org.ldp.jdasm.attribute.CodeAttribute` **code** `)`
  - **Usage**
    - This function read an Attribute from an input stream whose cursor must point to the first byte (which specify the index into the constant pool of the name). If the attribute's parent is a CodeAttribute, it must be specified as argument to allow the LineNumberTableAttribute to have link to instructions.

- *getName*
  `public String` **getName**`( )`
  - **Usage**
    - Get the name of the attribute.

- *setConstantValueIndex*
  `public void` **setConstantValueIndex**`( int` **idx**`, short` **cpool_index** `)`

- **Usage**
  - The constant pool tells the user that the value retrieved with AConstantPoolUser#getConstantValue(int) at index "idx" has received the "cpool_index" index in the constant pool.
- **Parameters**
  - `idx` - the index of the value mapped with the ones returned by getConstantValue(int)
  - `cpool_index` - the index of the value in the constant pool

- *setName*
  `public void` **setName**`( java.lang.String` **name** `)`
  - **Usage**
    - Set the name of the attribute.

Methods inherited from class
`org.ldp.jdasm.attribute.Attributable`

(in A.3.2, page 136)

- *addAttribute*
  `public DAttribute` **addAttribute**`( org.ldp.jdasm.DAttribute` **attr** `)`
  - **Usage**
    - Adds a DAttribute.
  - **Returns** - the inserted DAttribute

- *attributeCount*
  `public int` **attributeCount**`( )`
  - **Usage**
    - Returns the number of attribute for this class

- *getAttribute*
  `public DAttribute` **getAttribute**`( int` **idx** `)`
  - **Usage**
    - Get the DAttribute at specified position

- *getAttribute*
  `public DAttribute` **getAttribute**`( java.lang.String` **name** `)`
  - **Usage**
    - Returns the DAttribute with specified name, or null if it doesn't exist.

- *getAttributes*
  `public DAttribute` **getAttributes**`( )`
  - **Usage**
    - Returns all the attributes

- *hasAttribute*
  `public boolean` **hasAttribute**`( java.lang.String` **name** `)`
  - **Usage**
    - Returns true if an attribute with given name exists.

- *removeAttribute*
  `public boolean` **removeAttribute**`( org.ldp.jdasm.DAttribute` **attr** `)`
  - **Usage**
    - Removes the specified DAttribute (match on equals())
  - **Returns** - true if the attribute is removed (if it was present), false it it is not present in the list.

- *removeAttribute*
  `public void` **removeAttribute**`( int` **idx** `)`
  - **Usage**
    - Removes the DAttribute at specified position
- *removeAttribute*
  `public boolean` **removeAttribute**`( java.lang.String` **name** `)`
  - **Usage**
    - Removes the DAttribute with specified name. Returns true if it is removed (if it was present), false if it is not present in the list.

Methods inherited from class
`org.ldp.jdasm.constantpool.AConstantPoolUser`

(in A.2.4, page 97)

- *constantPoolUserChildSize*
  `public abstract int` **constantPoolUserChildSize**`( )`
  - **Usage**
    - Returns the number of IConstantPoolUser instances of this class.
- *constantPoolValueSize*
  `public abstract int` **constantPoolValueSize**`( )`
  - **Usage**
    - Returns the number of values that this constant pool user wants to insert into the constant pool.
- *getConstantPoolUserChild*
  `public abstract AConstantPoolUser` **getConstantPoolUserChild**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th IConstantPoolUser instance of this class.
- *getConstantValue*
  `public abstract Object` **getConstantValue**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th value that this constant pool user wants to insert into the constant pool.
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)
- *getConstantValueType*
  `public abstract int` **getConstantValueType**`( int` **idx** `)`
  - **Usage**
    - Returns the value type of the idx-th value that this constant pool user wants to insert into the constant pool. The list of the types is declared in DConstantPool
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)
- *setConstantValueIndex*
  `public abstract void` **setConstantValueIndex**`( int` **idx**`, short` **cpool_index** `)`
  - **Usage**
    - The constant pool tells the user that the value retrieved with AConstantPoolUser#getConstantValue(int) at index "idx" has received the "cpool_index" index in the constant pool.
  - **Parameters**
    - `idx` - the index of the value mapped with the ones returned by getConstantValue(int)
    - `cpool_index` - the index of the value in the constant pool

A.3.12 Class **LocalVariableTableElement**

Class used inside the LocalVariableTableAttribute and inside LocalVariableTypeTableAttribute classes: it has informations about the name and descriptor or signature of the variable, start_pc and length of the scope, index into the local variable array.

Declaration

public class LocalVariableTableElement
extends java.lang.Object

Constructors

- *LocalVariableTableElement*
  `public` **LocalVariableTableElement**`( )`
  - **Usage**
    - Default empty constructor.
- *LocalVariableTableElement*
  `public` **LocalVariableTableElement**`( java.io.InputStream` **is**`, org.ldp.jdasm.DConstantPool` **cpool**`, org.ldp.jdasm.attribute.CodeAttribute` **code** `)`
  - **Usage**
    - Creates a LocalVariableTableElement attribute reading it from an input stream whose cursor must point to the beginning of the attribute just after the two bytes that reference the name in the constant pool. The constant pool needs to have all the string constant values used by this attribute. In order to have a link with instructions, this constructor needs the filled CodeAttribute it refers to.
- *LocalVariableTableElement*
  `public` **LocalVariableTableElement**`( org.ldp.jdasm.attribute.instruction.Instruction` **start_pc**`, org.ldp.jdasm.attribute.instruction.Instruction` **end_pc**`, java.lang.String` **name**`, java.lang.String` **descriptor**`, int` **index** `)`
  - **Usage**
    - Constructs a LocalVariableTableElement with all values specified.
  - **Parameters**
    - `end_pc` - it is the last inclusive instruction this scope is valid until.

Methods

- *classFileInfo*
  `public String` **classFileInfo**`( int` **indent_level** `)`
  - **Usage**
    - Returns a printable info string

A.3.13 Class **LocalVariableTypeTableAttribute**

This class represents an Attribute of the java class file.

This attribute uses the class LocalVariableTableElement as element of its table: each of these element has information about the starting instruction, the length in byte of the variable's scope, the name of the variable and its signature.. all mapped to an index into the local variable array. This attribute differs from LocalVariableTableAttribute in that it provides signature information rather than descriptor information. This difference is only significant for variable whose type is a generic reference type. Such variables will appear in both tables, while variables of other types will appear only in LocalVariableTableAttribute . Nevertheless, the LocalVariableTableElement class is used both in LocalVariableTableAttribute and in LocalVariableTypeTableAttribute

Declaration

public class LocalVariableTypeTableAttribute
extends org.ldp.jdasm.DAttribute

Constructors

- *LocalVariableTypeTableAttribute*
  `public` **LocalVariableTypeTableAttribute**`( )`
  - **Usage**
    * Creates an empty LocalVariableTypeAttribute attribute.

- *LocalVariableTypeTableAttribute*
  `public` **LocalVariableTypeTableAttribute**`( java.io.InputStream` **is**`, org.ldp.jdasm.DConstantPool` **cpool**`, org.ldp.jdasm.attribute.CodeAttribute` **code** `)`
  - **Usage**
    * Creates a LocalVariableTypeTableAttribute attribute reading it from an input stream whose cursor must point to the beginning of the attribute just after the two bytes that reference the name in the constant pool. The constant pool needs to have all the string constant values used by this attribute. In order to have a link with instructions, this constructor needs the filled CodeAttribute it refers to.

Methods

- *add*
  `public void` **add**`( org.ldp.jdasm.attribute.LocalVariableTableElement` **item** `)`
  - **Usage**
    * Add a LocalVariableTableElement element to the table

- *build*
  `public int` **build**`( byte []` **tofill**`, int` **atindex** `)`

- **Usage**
  - It builds the content of this class into a final byte array. Such array is passed as first argument, and the index from where start to insert bytes is the second argument.
- **Parameters**
  - `tofill` - the array to fill
  - `atindex` - the index from where start to fill
- **Returns** - the number of byte inserted

- *build*
  `public int` **build**`( java.io.OutputStream` **output** `)`
  - **Usage**
    - It builds the content of this class into an output stream. This method is a faster version of int) since it doesn't need to call the IBuildable#getBuildLength() before.
  - **Parameters**
    - `output` - the stream to write onto
  - **Returns** - the number of byte inserted
  - **Exceptions**
    - `on` - error while writing onto the stream

- *classFileInfo*
  `public String` **classFileInfo**`( int` **indent_level** `)`
  - **Usage**
    - Returns a printable info string

- *clearElements*
  `public void` **clearElements**`( )`
  - **Usage**
    - Clears all the items.

- *constantPoolValueSize*
  `public int` **constantPoolValueSize**`( )`
  - **Usage**
    - Returns the number of values that this constant pool user wants to insert into the constant pool.

- *elementAt*
  `public LocalVariableTableElement` **elementAt**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th LocalVariableTableElement of the table

---

- *getBuildLength*
  `public int` **getBuildLength**`( )`
  - **Usage**
    - Returns the length of portion of byte array the IBuildable class is going to create. If this class has instances of other IBuildable classes, then it will return the whole sum of all the various getBuildLength().

- *getConstantValue*
  `public Object` **getConstantValue**`( int` **idx** `)`

- **Usage**
  - Returns the idx-th value that this constant pool user wants to insert into the constant pool.
- **Exceptions**
  - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *getConstantValueType*
  `public int` **getConstantValueType**`( int` **idx** `)`
  - **Usage**
    - Returns the value type of the idx-th value that this constant pool user wants to insert into the constant pool. The list of the types is declared in DConstantPool
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *getElementSize*
  `public int` **getElementSize**`( )`
  - **Usage**
    - Returns the number of items in this attribute.

- *removeInstruction*
  `public boolean` **removeInstruction**`(`
  `org.ldp.jdasm.attribute.instruction.Instruction` **instruction**`,`
  `org.ldp.jdasm.attribute.CodeAttribute` **code** `)`
  - **Usage**
    - Search for given `instruction` as starting of a local variable information: if it's found, this information is removed and true is returned, otherwise false is returned. If this instruction is found to be an end scope instruction, some adjustment is done: the new ending instruction becomes the previous one (`code` is used for this purpose).

- *setConstantValueIndex*
  `public void` **setConstantValueIndex**`( int` **idx**`, short` **cpool_index** `)`
  - **Usage**
    - The constant pool tells the user that the value retrieved with AConstantPoolUser#getConstantValue(int) at index "idx" has received the "cpool_index" index in the constant pool.
  - **Parameters**
    - `idx` - the index of the value mapped with the ones returned by getConstantValue(int)
    - `cpool_index` - the index of the value in the constant pool

Methods inherited from class `org.ldp.jdasm.DAttribute`

(in A.1.6, page 90)

- *build*
  `public int` **build**`( byte [] tofill, int` **atindex** `)`
  - **Usage**

    - It builds the content of this class into a final byte array. Such array is passed as first argument, and the index from where start to insert bytes is the second argument.
  - **Parameters**
    - `tofill` - the array to fill
    - `atindex` - the index from where start to fill
  - **Returns** - the number of byte inserted

- *build*
  `public int` **build**`( java.io.OutputStream` **output** `)`
  - **Usage**
    - It builds the content of this class into an output stream. This method is a faster version of int) since it doesn't need to call the IBuildable#getBuildLength() before.
  - **Parameters**
    - `output` - the stream to write onto
  - **Returns** - the number of byte inserted
  - **Exceptions**
    - `on` - error while writing onto the stream

- *classFileInfo*
  `public String` **classFileInfo**`( int` **indent_level** `)`
  - **Usage**
    - Returns a printable info string

- *constantPoolUserChildSize*
  `public int` **constantPoolUserChildSize**`( )`
  - **Usage**
    - Returns the number of IConstantPoolUser instances of this class.

- *constantPoolValueSize*
  `public int` **constantPoolValueSize**`( )`
  - **Usage**
    - Returns the number of values that this constant pool user wants to insert into the constant pool.

- *getBuildLength*
  `public int` **getBuildLength**`( )`
  - **Usage**
    - Returns the length of portion of byte array the IBuildable class is going to create. If this class has instances of other IBuildable classes, then it will return the whole sum of all the various getBuildLength().

- *getConstantPoolUserChild*
  `public AConstantPoolUser` **getConstantPoolUserChild**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th IConstantPoolUser instance of this class.

- *getConstantValue*
  `public Object` **getConstantValue**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th value that this constant pool user wants to insert into the constant pool.
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *getConstantValueType*
  `public int` **getConstantValueType**`( int` **idx** `)`
  - **Usage**

    - Returns the value type of the idx-th value that this constant pool user wants to insert into the constant pool. The list of the types is declared in DConstantPool
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *getDAttributeFromInputStream*
  `public static DAttribute` **getDAttributeFromInputStream(** `java.io.InputStream` **is,** `org.ldp.jdasm.DConstantPool` **cpool**)
  - **Usage**
    - This function read an Attribute from an input stream whose cursor must point to the first byte (which specify the index into the constant pool of the name).

- *getDAttributeFromInputStream*
  `public static DAttribute` **getDAttributeFromInputStream(** `java.io.InputStream` **is,** `org.ldp.jdasm.DConstantPool` **cpool,** `org.ldp.jdasm.attribute.CodeAttribute` **code**)
  - **Usage**
    - This function read an Attribute from an input stream whose cursor must point to the first byte (which specify the index into the constant pool of the name). If the attribute's parent is a CodeAttribute, it must be specified as argument to allow the LineNumberTableAttribute to have link to instructions.

- *getName*
  `public String` **getName( )**
  - **Usage**
    - Get the name of the attribute.

- *setConstantValueIndex*
  `public void` **setConstantValueIndex(** `int` **idx,** `short` **cpool_index**)
  - **Usage**
    - The constant pool tells the user that the value retrieved with AConstantPoolUser#getConstantValue(int) at index "idx" has received the "cpool_index" index in the constant pool.
  - **Parameters**
    - `idx` - the index of the value mapped with the ones returned by getConstantValue(int)
    - `cpool_index` - the index of the value in the constant pool

- *setName*
  `public void` **setName(** `java.lang.String` **name**)
  - **Usage**
    - Set the name of the attribute.

Methods inherited from class `org.ldp.jdasm.attribute.Attributable`

(in A.3.2, page 136)

- *addAttribute*
  `public DAttribute` **addAttribute(** `org.ldp.jdasm.DAttribute` **attr**)
  - **Usage**
    - Adds a DAttribute.
  - **Returns** - the inserted DAttribute

- *attributeCount*
  `public int` **attributeCount( )**

  - **Usage**
    - Returns the number of attribute for this class

- *getAttribute*
  `public DAttribute` **getAttribute**`( int` **idx** `)`
  - **Usage**
    - Get the DAttribute at specified position

- *getAttribute*
  `public DAttribute` **getAttribute**`( java.lang.String` **name** `)`
  - **Usage**
    - Returns the DAttribute with specified name, or null if it doesn't exist.

- *getAttributes*
  `public DAttribute` **getAttributes**`( )`
  - **Usage**
    - Returns all the attributes

- *hasAttribute*
  `public boolean` **hasAttribute**`( java.lang.String` **name** `)`
  - **Usage**
    - Returns true if an attribute with given name exists.

- *removeAttribute*
  `public boolean` **removeAttribute**`( org.ldp.jdasm.DAttribute` **attr** `)`
  - **Usage**
    - Removes the specified DAttribute (match on equals())
  - **Returns** - true if the attribute is removed (if it was present), false it it is not present in the list.

- *removeAttribute*
  `public void` **removeAttribute**`( int` **idx** `)`
  - **Usage**
    - Removes the DAttribute at specified position

- *removeAttribute*
  `public boolean` **removeAttribute**`( java.lang.String` **name** `)`
  - **Usage**
    - Removes the DAttribute with specified name. Returns true if it is removed (if it was present), false if it is not present in the list.

Methods inherited from class
`org.ldp.jdasm.constantpool.AConstantPoolUser`

(in A.2.4, page 97)

- *constantPoolUserChildSize*
  `public abstract int` **constantPoolUserChildSize**`( )`
  - **Usage**
    - Returns the number of IConstantPoolUser instances of this class.

- *constantPoolValueSize*
  `public abstract int` **constantPoolValueSize**`( )`
  - **Usage**
    - Returns the number of values that this constant pool user wants to insert into the constant pool.

- *getConstantPoolUserChild*
  `public abstract AConstantPoolUser` **getConstantPoolUserChild**`( int` **idx** `)`

  - **Usage**
    - Returns the idx-th IConstantPoolUser instance of this class.

- *getConstantValue*
  `public abstract Object` **getConstantValue**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th value that this constant pool user wants to insert into the constant pool.
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *getConstantValueType*
  `public abstract int` **getConstantValueType**`( int` **idx** `)`
  - **Usage**
    - Returns the value type of the idx-th value that this constant pool user wants to insert into the constant pool. The list of the types is declared in DConstantPool
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *setConstantValueIndex*
  `public abstract void` **setConstantValueIndex**`( int` **idx**`, short` **cpool_index** `)`
  - **Usage**
    - The constant pool tells the user that the value retrieved with AConstantPoolUser#getConstantValue(int) at index "idx" has received the "cpool_index" index in the constant pool.
  - **Parameters**
    - `idx` - the index of the value mapped with the ones returned by getConstantValue(int)
    - `cpool_index` - the index of the value in the constant pool

A.3.14 Class SignatureAttribute

This class represents an Attribute of the java class file.

This attribute is intended to hold the signature of a class (if it is an attribute of DClass), of a method (if it is an attribute of DMethod) or of a field (if it is an attribute of DField).

Declaration

public class SignatureAttribute
extends org.ldp.jdasm.DAttribute

Constructors

- *SignatureAttribute*
  `public` **SignatureAttribute**`( )`
  - **Usage**
    - Creates an empty SignatureAttribute attribute.

- *SignatureAttribute*
  `public` **SignatureAttribute**`( java.io.InputStream` **is**`, org.ldp.jdasm.DConstantPool` **cpool** `)`
  - **Usage**
    * Creates a Signature attribute reading it from an input stream whose cursor must point to the beginning of the attribute just after the two bytes that reference the name in the constant pool.

- *SignatureAttribute*
  `public` **SignatureAttribute**`( java.lang.String` **signatureValue** `)`
  - **Usage**
    * Creates a SignatureAttribute attribute with given signature value.

Methods

- *build*
  `public int` **build**`( byte []` **tofill**`, int` **atindex** `)`
  - **Usage**
    * It builds the content of this class into a final byte array. Such array is passed as first argument, and the index from where start to insert bytes is the second argument.
  - **Parameters**
    * `tofill` - the array to fill
    * `atindex` - the index from where start to fill
  - **Returns** - the number of byte inserted

- *build*
  `public int` **build**`( java.io.OutputStream` **output** `)`
  - **Usage**
    * It builds the content of this class into an output stream. This method is a faster version of int) since it doesn't need to call the IBuildable#getBuildLength() before.
  - **Parameters**
    * `output` - the stream to write onto
  - **Returns** - the number of byte inserted
  - **Exceptions**
    * `on` - error while writing onto the stream

- *classFileInfo*
  `public String` **classFileInfo**`( int` **indent_level** `)`
  - **Usage**
    * Returns a printable info string

- *constantPoolValueSize*
  `public int` **constantPoolValueSize**`( )`
  - **Usage**
    * Returns the number of values that this constant pool user wants to insert into the constant pool.

- *getBuildLength*
  `public int` **getBuildLength**`( )`
  - **Usage**
    * Returns the length of portion of byte array the IBuildable class is going to create. If this class has instances of other IBuildable classes, then it will return the whole sum of all the various getBuildLength().
- *getConstantValue*
  `public Object` **getConstantValue**`( int` **idx** `)`
  - **Usage**
    * Returns the idx-th value that this constant pool user wants to insert into the constant pool.
  - **Exceptions**
    * `java.lang.Exception` - if the value is not ready (since the control is done only at build time)
- *getConstantValueType*
  `public int` **getConstantValueType**`( int` **idx** `)`
  - **Usage**
    * Returns the value type of the idx-th value that this constant pool user wants to insert into the constant pool. The list of the types is declared in DConstantPool
  - **Exceptions**
    * `java.lang.Exception` - if the value is not ready (since the control is done only at build time)
- *getSignatureValue*
  `public String` **getSignatureValue**`( )`
  - **Usage**
    * Get the value of the signature in this attribute
- *setConstantValueIndex*
  `public void` **setConstantValueIndex**`( int` **idx**`, short` **cpool_index** `)`
  - **Usage**
    * The constant pool tells the user that the value retrieved with AConstantPoolUser#getConstantValue(int) at index "idx" has received the "cpool_index" index in the constant pool.
  - **Parameters**
    * `idx` - the index of the value mapped with the ones returned by getConstantValue(int)
    * `cpool_index` - the index of the value in the constant pool
- *setSignatureValue*
  `public void` **setSignatureValue**`( java.lang.String` **signatureValue** `)`
  - **Usage**
    * Set the value of the signature in this attribute

Methods inherited from class `org.ldp.jdasm.DAttribute`

(in A.1.6, page 90)

- *build*
  `public int` **build**`( byte [] tofill, int` **atindex** `)`
  - **Usage**
    - It builds the content of this class into a final byte array. Such array is passed as first argument, and the index from where start to insert bytes is the second argument.
  - **Parameters**
    - `tofill` - the array to fill
    - `atindex` - the index from where start to fill
  - **Returns** - the number of byte inserted
- *build*
  `public int` **build**`( java.io.OutputStream` **output** `)`
  - **Usage**
    - It builds the content of this class into an output stream. This method is a faster version of int) since it doesn't need to call the IBuildable#getBuildLength() before.
  - **Parameters**
    - `output` - the stream to write onto
  - **Returns** - the number of byte inserted
  - **Exceptions**
    - `on` - error while writing onto the stream
- *classFileInfo*
  `public String` **classFileInfo**`( int` **indent_level** `)`
  - **Usage**
    - Returns a printable info string
- *constantPoolUserChildSize*
  `public int` **constantPoolUserChildSize**`( )`
  - **Usage**
    - Returns the number of IConstantPoolUser instances of this class.
- *constantPoolValueSize*
  `public int` **constantPoolValueSize**`( )`
  - **Usage**
    - Returns the number of values that this constant pool user wants to insert into the constant pool.
- *getBuildLength*
  `public int` **getBuildLength**`( )`
  - **Usage**
    - Returns the length of portion of byte array the IBuildable class is going to create. If this class has instances of other IBuildable classes, then it will return the whole sum of all the various getBuildLength().
- *getConstantPoolUserChild*
  `public AConstantPoolUser` **getConstantPoolUserChild**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th IConstantPoolUser instance of this class.
- *getConstantValue*
  `public Object` **getConstantValue**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th value that this constant pool user wants to insert into the constant pool.

- **Exceptions**
  - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *getConstantValueType*
  `public int` **getConstantValueType**`( int` **idx** `)`
  - **Usage**
    - Returns the value type of the idx-th value that this constant pool user wants to insert into the constant pool. The list of the types is declared in DConstantPool
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *getDAttributeFromInputStream*
  `public static DAttribute` **getDAttributeFromInputStream**`(`
  `java.io.InputStream` **is**`, org.ldp.jdasm.DConstantPool` **cpool** `)`
  - **Usage**
    - This function read an Attribute from an input stream whose cursor must point to the first byte (which specify the index into the constant pool of the name).

- *getDAttributeFromInputStream*
  `public static DAttribute` **getDAttributeFromInputStream**`(`
  `java.io.InputStream` **is**`, org.ldp.jdasm.DConstantPool` **cpool**`,`
  `org.ldp.jdasm.attribute.CodeAttribute` **code** `)`
  - **Usage**
    - This function read an Attribute from an input stream whose cursor must point to the first byte (which specify the index into the constant pool of the name). If the attribute's parent is a CodeAttribute, it must be specified as argument to allow the LineNumberTableAttribute to have link to instructions.

- *getName*
  `public String` **getName**`( )`
  - **Usage**
    - Get the name of the attribute.

- *setConstantValueIndex*
  `public void` **setConstantValueIndex**`( int` **idx**`, short` **cpool_index** `)`
  - **Usage**
    - The constant pool tells the user that the value retrieved with AConstantPoolUser#getConstantValue(int) at index "idx" has received the "cpool_index" index in the constant pool.
  - **Parameters**
    - `idx` - the index of the value mapped with the ones returned by getConstantValue(int)
    - `cpool_index` - the index of the value in the constant pool

- *setName*
  `public void` **setName**`( java.lang.String` **name** `)`
  - **Usage**
    - Set the name of the attribute.

Methods inherited from class `org.ldp.jdasm.attribute.Attributable`

(in A.3.2, page 136)

- *addAttribute*
  `public DAttribute` **addAttribute**`( org.ldp.jdasm.DAttribute` **attr** `)`

- **Usage**
  - Adds a DAttribute.
- **Returns** - the inserted DAttribute

- *attributeCount*
  `public int` **attributeCount**`( )`
  - **Usage**
    - Returns the number of attribute for this class

- *getAttribute*
  `public DAttribute` **getAttribute**`( int` **idx** `)`
  - **Usage**
    - Get the DAttribute at specified position

- *getAttribute*
  `public DAttribute` **getAttribute**`( java.lang.String` **name** `)`
  - **Usage**
    - Returns the DAttribute with specified name, or null if it doesn't exist.

- *getAttributes*
  `public DAttribute` **getAttributes**`( )`
  - **Usage**
    - Returns all the attributes

- *hasAttribute*
  `public boolean` **hasAttribute**`( java.lang.String` **name** `)`
  - **Usage**
    - Returns true if an attribute with given name exists.

- *removeAttribute*
  `public boolean` **removeAttribute**`( org.ldp.jdasm.DAttribute` **attr** `)`
  - **Usage**
    - Removes the specified DAttribute (match on equals())
  - **Returns** - true if the attribute is removed (if it was present), false it it is not present in the list.

- *removeAttribute*
  `public void` **removeAttribute**`( int` **idx** `)`
  - **Usage**
    - Removes the DAttribute at specified position

- *removeAttribute*
  `public boolean` **removeAttribute**`( java.lang.String` **name** `)`
  - **Usage**
    - Removes the DAttribute with specified name. Returns true if it is removed (if it was present), false if it is not present in the list.

Methods inherited from class `org.ldp.jdasm.constantpool.AConstantPoolUser`

(in A.2.4, page 97)

- *constantPoolUserChildSize*
  `public abstract int` **constantPoolUserChildSize**`( )`
  - **Usage**
    - Returns the number of IConstantPoolUser instances of this class.

- *constantPoolValueSize*
  `public abstract int` **constantPoolValueSize**`( )`
  - **Usage**
    - Returns the number of values that this constant pool user wants to insert into the constant pool.
- *getConstantPoolUserChild*
  `public abstract AConstantPoolUser` **getConstantPoolUserChild**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th IConstantPoolUser instance of this class.
- *getConstantValue*
  `public abstract Object` **getConstantValue**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th value that this constant pool user wants to insert into the constant pool.
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)
- *getConstantValueType*
  `public abstract int` **getConstantValueType**`( int` **idx** `)`
  - **Usage**
    - Returns the value type of the idx-th value that this constant pool user wants to insert into the constant pool. The list of the types is declared in DConstantPool
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)
- *setConstantValueIndex*
  `public abstract void` **setConstantValueIndex**`( int` **idx**`, short` **cpool_index** `)`
  - **Usage**
    - The constant pool tells the user that the value retrieved with AConstantPoolUser#getConstantValue(int) at index "idx" has received the "cpool_index" index in the constant pool.
  - **Parameters**
    - `idx` - the index of the value mapped with the ones returned by getConstantValue(int)
    - `cpool_index` - the index of the value in the constant pool

A.3.15 Class SourceDebugExtensionAttribute

This class represents an Attribute of the java class file.

This attribute (attribute of DClass) holds a string in UFT-8 format with no terminating zero byte. The string will be interpreted as extended debugging information. The content of this string has no semantic effect on the jvm.

Declaration

```
public class SourceDebugExtensionAttribute
extends org.ldp.jdasm.DAttribute
```

Constructors

- *SourceDebugExtensionAttribute*
  `public` **SourceDebugExtensionAttribute**`( )`
  - **Usage**
    - Creates an empty SourceDebugExtensionAttribute.

- *SourceDebugExtensionAttribute*
  `public` **SourceDebugExtensionAttribute**`( byte [] content )`
  - **Usage**
    - Creates a SourceDebugExtensionAttribute with specific content.

- *SourceDebugExtensionAttribute*
  `public` **SourceDebugExtensionAttribute**`( java.io.InputStream` **is** `)`
  - **Usage**
    - Creates a SourceDebugExtensionAttribute attribute reading it from an input stream whose cursor must point to the beginning of the attribute just after the two bytes that reference the name in the constant pool.

Methods

- *build*
  `public int` **build**`( byte []` **tofill**`, int` **atindex** `)`
  - **Usage**
    - It builds the content of this class into a final byte array. Such array is passed as first argument, and the index from where start to insert bytes is the second argument.
  - **Parameters**
    - `tofill` - the array to fill
    - `atindex` - the index from where start to fill
  - **Returns** - the number of byte inserted

- *build*
  `public int` **build**`( java.io.OutputStream` **output** `)`
  - **Usage**
    - It builds the content of this class into an output stream. This method is a faster version of int) since it doesn't need to call the IBuildable#getBuildLength() before.
  - **Parameters**
    - `output` - the stream to write onto
  - **Returns** - the number of byte inserted
  - **Exceptions**
    - `on` - error while writing onto the stream

- *classFileInfo*
  `public String` **classFileInfo**`( int` **indent_level** `)`
  - **Usage**

    - Returns a printable info string

- *getBuildLength*
  `public int` **getBuildLength**`( )`
  - **Usage**
    - Returns the length of portion of byte array the IBuildable class is going to create. If this class has instances of other IBuildable classes, then it will return the whole sum of all the various getBuildLength().

- *getDebugInfo*
  `public byte` **getDebugInfo**`( )`
  - **Usage**
    - Get the content of this SourceDebugExtensionAttribute.

- *setDebugInfo*
  `public void` **setDebugInfo**`( byte [] info )`
  - **Usage**
    - Set the content of this SourceDebugExtensionAttribute.

METHODS INHERITED FROM CLASS `org.ldp.jdasm.DAttribute`

(in A.1.6, page 90)

- *build*
  `public int` **build**`( byte [] tofill, int` **atindex** `)`
  - **Usage**
    - It builds the content of this class into a final byte array. Such array is passed as first argument, and the index from where start to insert bytes is the second argument.
  - **Parameters**
    - `tofill` - the array to fill
    - `atindex` - the index from where start to fill
  - **Returns** - the number of byte inserted

- *build*
  `public int` **build**`( java.io.OutputStream` **output** `)`
  - **Usage**
    - It builds the content of this class into an output stream. This method is a faster version of int) since it doesn't need to call the IBuildable#getBuildLength() before.
  - **Parameters**
    - `output` - the stream to write onto
  - **Returns** - the number of byte inserted
  - **Exceptions**
    - `on` - error while writing onto the stream

- *classFileInfo*
  `public String` **classFileInfo**`( int` **indent_level** `)`
  - **Usage**
    - Returns a printable info string

- *constantPoolUserChildSize*
  `public int` **constantPoolUserChildSize**`( )`
  - **Usage**
    - Returns the number of IConstantPoolUser instances of this class.

- *constantPoolValueSize*
  `public int` **constantPoolValueSize**`( )`
  - **Usage**
    - Returns the number of values that this constant pool user wants to insert into the constant pool.
- *getBuildLength*
  `public int` **getBuildLength**`( )`
  - **Usage**
    - Returns the length of portion of byte array the IBuildable class is going to create. If this class has instances of other IBuildable classes, then it will return the whole sum of all the various getBuildLength().
- *getConstantPoolUserChild*
  `public AConstantPoolUser` **getConstantPoolUserChild**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th IConstantPoolUser instance of this class.
- *getConstantValue*
  `public Object` **getConstantValue**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th value that this constant pool user wants to insert into the constant pool.
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)
- *getConstantValueType*
  `public int` **getConstantValueType**`( int` **idx** `)`
  - **Usage**
    - Returns the value type of the idx-th value that this constant pool user wants to insert into the constant pool. The list of the types is declared in DConstantPool
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)
- *getDAttributeFromInputStream*
  `public static DAttribute` **getDAttributeFromInputStream**`(`
  `java.io.InputStream` **is**`, org.ldp.jdasm.DConstantPool` **cpool** `)`
  - **Usage**
    - This function read an Attribute from an input stream whose cursor must point to the first byte (which specify the index into the constant pool of the name).
- *getDAttributeFromInputStream*
  `public static DAttribute` **getDAttributeFromInputStream**`(`
  `java.io.InputStream` **is**`, org.ldp.jdasm.DConstantPool` **cpool**`,`
  `org.ldp.jdasm.attribute.CodeAttribute` **code** `)`
  - **Usage**
    - This function read an Attribute from an input stream whose cursor must point to the first byte (which specify the index into the constant pool of the name). If the attribute's parent is a CodeAttribute, it must be specified as argument to allow the LineNumberTableAttribute to have link to instructions.
- *getName*
  `public String` **getName**`( )`
  - **Usage**
    - Get the name of the attribute.

- *setConstantValueIndex*
  `public void` **setConstantValueIndex**`( int` **idx**`, short` **cpool_index** `)`
  - **Usage**
    - The constant pool tells the user that the value retrieved with AConstantPoolUser#getConstantValue(int) at index "idx" has received the "cpool_index" index in the constant pool.
  - **Parameters**
    - `idx` - the index of the value mapped with the ones returned by getConstantValue(int)
    - `cpool_index` - the index of the value in the constant pool
- *setName*
  `public void` **setName**`( java.lang.String` **name** `)`
  - **Usage**
    - Set the name of the attribute.

Methods inherited from class
`org.ldp.jdasm.attribute.Attributable`

(in A.3.2, page 136)

- *addAttribute*
  `public DAttribute` **addAttribute**`( org.ldp.jdasm.DAttribute` **attr** `)`
  - **Usage**
    - Adds a DAttribute.
  - **Returns** - the inserted DAttribute
- *attributeCount*
  `public int` **attributeCount**`( )`
  - **Usage**
    - Returns the number of attribute for this class
- *getAttribute*
  `public DAttribute` **getAttribute**`( int` **idx** `)`
  - **Usage**
    - Get the DAttribute at specified position
- *getAttribute*
  `public DAttribute` **getAttribute**`( java.lang.String` **name** `)`
  - **Usage**
    - Returns the DAttribute with specified name, or null if it doesn't exist.
- *getAttributes*
  `public DAttribute` **getAttributes**`( )`
  - **Usage**
    - Returns all the attributes
- *hasAttribute*
  `public boolean` **hasAttribute**`( java.lang.String` **name** `)`
  - **Usage**
    - Returns true if an attribute with given name exists.
- *removeAttribute*
  `public boolean` **removeAttribute**`( org.ldp.jdasm.DAttribute` **attr** `)`
  - **Usage**
    - Removes the specified DAttribute (match on equals())

  - **Returns** - true if the attribute is removed (if it was present), false it it is not present in the list.

- *removeAttribute*
  `public void` **removeAttribute**`( int` **idx** `)`
  - **Usage**
    * Removes the DAttribute at specified position

- *removeAttribute*
  `public boolean` **removeAttribute**`( java.lang.String` **name** `)`
  - **Usage**
    * Removes the DAttribute with specified name. Returns true if it is removed (if it was present), false if it is not present in the list.

Methods inherited from class
`org.ldp.jdasm.constantpool.AConstantPoolUser`

(in A.2.4, page 97)

- *constantPoolUserChildSize*
  `public abstract int` **constantPoolUserChildSize**`( )`
  - **Usage**
    * Returns the number of IConstantPoolUser instances of this class.

- *constantPoolValueSize*
  `public abstract int` **constantPoolValueSize**`( )`
  - **Usage**
    * Returns the number of values that this constant pool user wants to insert into the constant pool.

- *getConstantPoolUserChild*
  `public abstract AConstantPoolUser` **getConstantPoolUserChild**`( int` **idx** `)`
  - **Usage**
    * Returns the idx-th IConstantPoolUser instance of this class.

- *getConstantValue*
  `public abstract Object` **getConstantValue**`( int` **idx** `)`
  - **Usage**
    * Returns the idx-th value that this constant pool user wants to insert into the constant pool.
  - **Exceptions**
    * `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *getConstantValueType*
  `public abstract int` **getConstantValueType**`( int` **idx** `)`
  - **Usage**
    * Returns the value type of the idx-th value that this constant pool user wants to insert into the constant pool. The list of the types is declared in DConstantPool
  - **Exceptions**
    * `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *setConstantValueIndex*
  `public abstract void` **setConstantValueIndex**`( int` **idx**`, short` **cpool_index** `)`
  - **Usage**

    * The constant pool tells the user that the value retrieved with AConstantPoolUser#getConstantValue(int) at index "idx" has received the "cpool_index" index in the constant pool.
  - **Parameters**
    * `idx` - the index of the value mapped with the ones returned by getConstantValue(int)
    * `cpool_index` - the index of the value in the constant pool

A.3.16 Class SourceFileAttribute

This class represents an Attribute of the java class file.

This attribute is intended to hold the name of the source file from which this class file was compiled.

Declaration

public class SourceFileAttribute
extends org.ldp.jdasm.DAttribute

Constructors

- *SourceFileAttribute*
  `public` **SourceFileAttribute**`( )`
  - **Usage**
    * Creates an empty SourceFile attribute.
- *SourceFileAttribute*
  `public` **SourceFileAttribute**`( java.io.InputStream` **is**`, org.ldp.jdasm.DConstantPool` **cpool** `)`
  - **Usage**
    * Creates a SourceFileAttribute attribute reading it from an input stream whose cursor must point to the beginning of the attribute just after the two bytes that reference the name in the constant pool.
- *SourceFileAttribute*
  `public` **SourceFileAttribute**`( java.lang.String` **sourceFileName** `)`
  - **Usage**
    * Creates a SourceFile attribute with given source file name.

Methods

- *build*
  `public int` **build**`( byte []` **tofill**`, int` **atindex** `)`
  - **Usage**
    * It builds the content of this class into a final byte array. Such array is passed as first argument, and the index from where start to insert bytes is the second argument.

- **Parameters**
  - `tofill` - the array to fill
  - `atindex` - the index from where start to fill
- **Returns** - the number of byte inserted

- *build*
  `public int` **build**`( java.io.OutputStream` **output** `)`
  - **Usage**
    - It builds the content of this class into an output stream. This method is a faster version of int) since it doesn't need to call the IBuildable#getBuildLength() before.
  - **Parameters**
    - `output` - the stream to write onto
  - **Returns** - the number of byte inserted
  - **Exceptions**
    - `on` - error while writing onto the stream

- *classFileInfo*
  `public String` **classFileInfo**`( int` **indent_level** `)`
  - **Usage**
    - Returns a printable info string

- *constantPoolValueSize*
  `public int` **constantPoolValueSize**`( )`
  - **Usage**
    - Returns the number of values that this constant pool user wants to insert into the constant pool.

- *getBuildLength*
  `public int` **getBuildLength**`( )`
  - **Usage**
    - Returns the length of portion of byte array the IBuildable class is going to create. If this class has instances of other IBuildable classes, then it will return the whole sum of all the various getBuildLength().

- *getConstantValue*
  `public Object` **getConstantValue**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th value that this constant pool user wants to insert into the constant pool.
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *getConstantValueType*
  `public int` **getConstantValueType**`( int` **idx** `)`
  - **Usage**
    - Returns the value type of the idx-th value that this constant pool user wants to insert into the constant pool. The list of the types is declared in DConstantPool
  - **Exceptions**

    * `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *getSourceFileName*
  `public String` **getSourceFileName**`( )`
  - **Usage**
    * Get the name of the source file hold in this attribute

- *setConstantValueIndex*
  `public void` **setConstantValueIndex**`( int` **idx**`, short` **cpool_index** `)`
  - **Usage**
    * The constant pool tells the user that the value retrieved with AConstantPoolUser#getConstantValue(int) at index "idx" has received the "cpool_index" index in the constant pool.
  - **Parameters**
    * `idx` - the index of the value mapped with the ones returned by getConstantValue(int)
    * `cpool_index` - the index of the value in the constant pool

- *setSourceFileName*
  `public void` **setSourceFileName**`( java.lang.String` **sourceFileName** `)`
  - **Usage**
    * Set the name of the source file hold in this attribute

Methods inherited from class `org.ldp.jdasm.DAttribute`

(in A.1.6, page 90)

- *build*
  `public int` **build**`( byte [] tofill, int` **atindex** `)`
  - **Usage**
    * It builds the content of this class into a final byte array. Such array is passed as first argument, and the index from where start to insert bytes is the second argument.
  - **Parameters**
    * `tofill` - the array to fill
    * `atindex` - the index from where start to fill
  - **Returns** - the number of byte inserted

- *build*
  `public int` **build**`( java.io.OutputStream` **output** `)`
  - **Usage**
    * It builds the content of this class into an output stream. This method is a faster version of int) since it doesn't need to call the IBuildable#getBuildLength() before.
  - **Parameters**
    * `output` - the stream to write onto
  - **Returns** - the number of byte inserted
  - **Exceptions**
    * `on` - error while writing onto the stream

- *classFileInfo*
  `public String` **classFileInfo**`( int` **indent_level** `)`
  - **Usage**

    - Returns a printable info string

- *constantPoolUserChildSize*
  `public int` **constantPoolUserChildSize**`( )`
  - **Usage**
    - Returns the number of IConstantPoolUser instances of this class.

- *constantPoolValueSize*
  `public int` **constantPoolValueSize**`( )`
  - **Usage**
    - Returns the number of values that this constant pool user wants to insert into the constant pool.

- *getBuildLength*
  `public int` **getBuildLength**`( )`
  - **Usage**
    - Returns the length of portion of byte array the IBuildable class is going to create. If this class has instances of other IBuildable classes, then it will return the whole sum of all the various getBuildLength().

- *getConstantPoolUserChild*
  `public AConstantPoolUser` **getConstantPoolUserChild**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th IConstantPoolUser instance of this class.

- *getConstantValue*
  `public Object` **getConstantValue**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th value that this constant pool user wants to insert into the constant pool.
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *getConstantValueType*
  `public int` **getConstantValueType**`( int` **idx** `)`
  - **Usage**
    - Returns the value type of the idx-th value that this constant pool user wants to insert into the constant pool. The list of the types is declared in DConstantPool
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *getDAttributeFromInputStream*
  `public static DAttribute` **getDAttributeFromInputStream**`(`
  `java.io.InputStream` **is**`, org.ldp.jdasm.DConstantPool` **cpool** `)`
  - **Usage**
    - This function read an Attribute from an input stream whose cursor must point to the first byte (which specify the index into the constant pool of the name).

- *getDAttributeFromInputStream*
  `public static DAttribute` **getDAttributeFromInputStream**`(`
  `java.io.InputStream` **is**`, org.ldp.jdasm.DConstantPool` **cpool**`,`
  `org.ldp.jdasm.attribute.CodeAttribute` **code** `)`
  - **Usage**
    - This function read an Attribute from an input stream whose cursor must point to the first byte (which specify the index into the constant pool of the name). If the attribute's parent is a CodeAttribute, it must be specified as argument to allow the LineNumberTableAttribute to have link to instructions.

- *getName*
  `public String` **getName**`( )`
  - **Usage**
    - Get the name of the attribute.
- *setConstantValueIndex*
  `public void` **setConstantValueIndex**`( int` **idx**`, short` **cpool_index** `)`
  - **Usage**
    - The constant pool tells the user that the value retrieved with AConstantPoolUser#getConstantValue(int) at index "idx" has received the "cpool_index" index in the constant pool.
  - **Parameters**
    - `idx` - the index of the value mapped with the ones returned by getConstantValue(int)
    - `cpool_index` - the index of the value in the constant pool
- *setName*
  `public void` **setName**`( java.lang.String` **name** `)`
  - **Usage**
    - Set the name of the attribute.

Methods inherited from class `org.ldp.jdasm.attribute.Attributable`

(in A.3.2, page 136)

- *addAttribute*
  `public DAttribute` **addAttribute**`( org.ldp.jdasm.DAttribute` **attr** `)`
  - **Usage**
    - Adds a DAttribute.
  - **Returns** - the inserted DAttribute
- *attributeCount*
  `public int` **attributeCount**`( )`
  - **Usage**
    - Returns the number of attribute for this class
- *getAttribute*
  `public DAttribute` **getAttribute**`( int` **idx** `)`
  - **Usage**
    - Get the DAttribute at specified position
- *getAttribute*
  `public DAttribute` **getAttribute**`( java.lang.String` **name** `)`
  - **Usage**
    - Returns the DAttribute with specified name, or null if it doesn't exist.
- *getAttributes*
  `public DAttribute` **getAttributes**`( )`
  - **Usage**
    - Returns all the attributes
- *hasAttribute*
  `public boolean` **hasAttribute**`( java.lang.String` **name** `)`
  - **Usage**
    - Returns true if an attribute with given name exists.

- *removeAttribute*
  `public boolean` **removeAttribute**`( org.ldp.jdasm.DAttribute` **attr** `)`
  - **Usage**
    - Removes the specified DAttribute (match on equals())
  - **Returns** - true if the attribute is removed (if it was present), false it it is not present in the list.

- *removeAttribute*
  `public void` **removeAttribute**`( int` **idx** `)`
  - **Usage**
    - Removes the DAttribute at specified position

- *removeAttribute*
  `public boolean` **removeAttribute**`( java.lang.String` **name** `)`
  - **Usage**
    - Removes the DAttribute with specified name. Returns true if it is removed (if it was present), false if it is not present in the list.

Methods inherited from class `org.ldp.jdasm.constantpool.AConstantPoolUser`

(in A.2.4, page 97)

- *constantPoolUserChildSize*
  `public abstract int` **constantPoolUserChildSize**`( )`
  - **Usage**
    - Returns the number of IConstantPoolUser instances of this class.

- *constantPoolValueSize*
  `public abstract int` **constantPoolValueSize**`( )`
  - **Usage**
    - Returns the number of values that this constant pool user wants to insert into the constant pool.

- *getConstantPoolUserChild*
  `public abstract AConstantPoolUser` **getConstantPoolUserChild**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th IConstantPoolUser instance of this class.

- *getConstantValue*
  `public abstract Object` **getConstantValue**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th value that this constant pool user wants to insert into the constant pool.
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *getConstantValueType*
  `public abstract int` **getConstantValueType**`( int` **idx** `)`
  - **Usage**
    - Returns the value type of the idx-th value that this constant pool user wants to insert into the constant pool. The list of the types is declared in DConstantPool
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *setConstantValueIndex*
  `public abstract void` **setConstantValueIndex**`( int` **idx**`, short` **cpool_index** `)`
  - **Usage**
    * The constant pool tells the user that the value retrieved with AConstantPoolUser#getConstantValue(int) at index "idx" has received the "cpool_index" index in the constant pool.
  - **Parameters**
    * `idx` - the index of the value mapped with the ones returned by getConstantValue(int)
    * `cpool_index` - the index of the value in the constant pool

A.3.17 Class StackMapTableAttribute

Unimplemented Attribute.

Declaration

public class StackMapTableAttribute
extends java.lang.Object

Constructors

- *StackMapTableAttribute*
  `public` **StackMapTableAttribute**`( )`

A.3.18 Class SyntheticAttribute

This class represents an Attribute of the java class file.

A class member that does not appear in the source code must be marked using a Synthetic attribute.

Declaration

public class SyntheticAttribute
extends org.ldp.jdasm.DAttribute

Constructors

- *SyntheticAttribute*
  `public` **SyntheticAttribute**`( )`
  - **Usage**
    * Creates a Synthetic attribute.

- *SyntheticAttribute*
  `public` **SyntheticAttribute**`( java.io.InputStream` **is**`, org.ldp.jdasm.DConstantPool` **cpool** `)`

- **Usage**
  - Creates a SyntheticAttribute attribute reading it from an input stream whose cursor must point to the beginning of the attribute just after the two bytes that reference the name in the constant pool.

Methods inherited from class `org.ldp.jdasm.DAttribute`

(in A.1.6, page 90)

- *build*
  `public int` **build**`( byte [] tofill, int` **atindex** `)`
  - **Usage**
    - It builds the content of this class into a final byte array. Such array is passed as first argument, and the index from where start to insert bytes is the second argument.
  - **Parameters**
    - `tofill` - the array to fill
    - `atindex` - the index from where start to fill
  - **Returns** - the number of byte inserted

- *build*
  `public int` **build**`( java.io.OutputStream` **output** `)`
  - **Usage**
    - It builds the content of this class into an output stream. This method is a faster version of int) since it doesn't need to call the IBuildable#getBuildLength() before.
  - **Parameters**
    - `output` - the stream to write onto
  - **Returns** - the number of byte inserted
  - **Exceptions**
    - `on` - error while writing onto the stream

- *classFileInfo*
  `public String` **classFileInfo**`( int` **indent_level** `)`
  - **Usage**
    - Returns a printable info string

- *constantPoolUserChildSize*
  `public int` **constantPoolUserChildSize**`( )`
  - **Usage**
    - Returns the number of IConstantPoolUser instances of this class.

- *constantPoolValueSize*
  `public int` **constantPoolValueSize**`( )`
  - **Usage**
    - Returns the number of values that this constant pool user wants to insert into the constant pool.

- *getBuildLength*
  `public int` **getBuildLength**`( )`
  - **Usage**
    - Returns the length of portion of byte array the IBuildable class is going to create. If this class has instances of other IBuildable classes, then it will return the whole sum of all the various getBuildLength().

- *getConstantPoolUserChild*
  `public AConstantPoolUser` **getConstantPoolUserChild**`( int` **idx** `)`
  - **Usage**

    - Returns the idx-th IConstantPoolUser instance of this class.

- *getConstantValue*
  `public Object` **getConstantValue**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th value that this constant pool user wants to insert into the constant pool.
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *getConstantValueType*
  `public int` **getConstantValueType**`( int` **idx** `)`
  - **Usage**
    - Returns the value type of the idx-th value that this constant pool user wants to insert into the constant pool. The list of the types is declared in DConstantPool
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *getDAttributeFromInputStream*
  `public static DAttribute` **getDAttributeFromInputStream**`(`
  `java.io.InputStream` **is**`, org.ldp.jdasm.DConstantPool` **cpool** `)`
  - **Usage**
    - This function read an Attribute from an input stream whose cursor must point to the first byte (which specify the index into the constant pool of the name).

- *getDAttributeFromInputStream*
  `public static DAttribute` **getDAttributeFromInputStream**`(`
  `java.io.InputStream` **is**`, org.ldp.jdasm.DConstantPool` **cpool**`,`
  `org.ldp.jdasm.attribute.CodeAttribute` **code** `)`
  - **Usage**
    - This function read an Attribute from an input stream whose cursor must point to the first byte (which specify the index into the constant pool of the name). If the attribute's parent is a CodeAttribute, it must be specified as argument to allow the LineNumberTableAttribute to have link to instructions.

- *getName*
  `public String` **getName**`( )`
  - **Usage**
    - Get the name of the attribute.

- *setConstantValueIndex*
  `public void` **setConstantValueIndex**`( int` **idx**`, short` **cpool_index** `)`
  - **Usage**
    - The constant pool tells the user that the value retrieved with AConstantPoolUser#getConstantValue(int) at index "idx" has received the "cpool_index" index in the constant pool.
  - **Parameters**
    - `idx` - the index of the value mapped with the ones returned by getConstantValue(int)
    - `cpool_index` - the index of the value in the constant pool

- *setName*
  `public void` **setName**`( java.lang.String` **name** `)`
  - **Usage**
    - Set the name of the attribute.

Methods inherited from class `org.ldp.jdasm.attribute.Attributable`

(in A.3.2, page 136)

- *addAttribute*
  `public DAttribute` **addAttribute**`( org.ldp.jdasm.DAttribute` **attr** `)`
  - **Usage**
    - Adds a DAttribute.
  - **Returns** - the inserted DAttribute
- *attributeCount*
  `public int` **attributeCount**`( )`
  - **Usage**
    - Returns the number of attribute for this class
- *getAttribute*
  `public DAttribute` **getAttribute**`( int` **idx** `)`
  - **Usage**
    - Get the DAttribute at specified position
- *getAttribute*
  `public DAttribute` **getAttribute**`( java.lang.String` **name** `)`
  - **Usage**
    - Returns the DAttribute with specified name, or null if it doesn't exist.
- *getAttributes*
  `public DAttribute` **getAttributes**`( )`
  - **Usage**
    - Returns all the attributes
- *hasAttribute*
  `public boolean` **hasAttribute**`( java.lang.String` **name** `)`
  - **Usage**
    - Returns true if an attribute with given name exists.
- *removeAttribute*
  `public boolean` **removeAttribute**`( org.ldp.jdasm.DAttribute` **attr** `)`
  - **Usage**
    - Removes the specified DAttribute (match on equals())
  - **Returns** - true if the attribute is removed (if it was present), false it it is not present in the list.
- *removeAttribute*
  `public void` **removeAttribute**`( int` **idx** `)`
  - **Usage**
    - Removes the DAttribute at specified position
- *removeAttribute*
  `public boolean` **removeAttribute**`( java.lang.String` **name** `)`
  - **Usage**
    - Removes the DAttribute with specified name. Returns true if it is removed (if it was present), false if it is not present in the list.

Methods inherited from class
`org.ldp.jdasm.constantpool.AConstantPoolUser`

(in A.2.4, page 97)

- *constantPoolUserChildSize*
  `public abstract int` **constantPoolUserChildSize**`( )`
  - **Usage**
    - Returns the number of IConstantPoolUser instances of this class.

- *constantPoolValueSize*
  `public abstract int` **constantPoolValueSize**`( )`
  - **Usage**
    - Returns the number of values that this constant pool user wants to insert into the constant pool.

- *getConstantPoolUserChild*
  `public abstract AConstantPoolUser` **getConstantPoolUserChild**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th IConstantPoolUser instance of this class.

- *getConstantValue*
  `public abstract Object` **getConstantValue**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th value that this constant pool user wants to insert into the constant pool.
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *getConstantValueType*
  `public abstract int` **getConstantValueType**`( int` **idx** `)`
  - **Usage**
    - Returns the value type of the idx-th value that this constant pool user wants to insert into the constant pool. The list of the types is declared in DConstantPool
  - **Exceptions**
    - `java.lang.Exception` - if the value is not ready (since the control is done only at build time)

- *setConstantValueIndex*
  `public abstract void` **setConstantValueIndex**`( int` **idx**`, short` **cpool_index** `)`
  - **Usage**
    - The constant pool tells the user that the value retrieved with AConstantPoolUser#getConstantValue(int) at index "idx" has received the "cpool_index" index in the constant pool.
  - **Parameters**
    - `idx` - the index of the value mapped with the ones returned by getConstantValue(int)
    - `cpool_index` - the index of the value in the constant pool

Appendix B

JCodeBrick API Documentation

B.1 Package org.ldp.jcodebrick

Classes

BrickedClass 229

BrickedClass is intended to make easy the instantiation of manipulated CbClasses.

BrickedMethod 231

This class allows you to invoke the code contained by a fragment.

BrickOperation 233

BrickOperation.InsertionPosition 234

Position of fragment insertions. You can insert a fragment A in four different positions relative to another fragment B.

```
BEFORE_START: you insert A before the B beginner marker
AFTER_START: you insert A after the B beginner marker
BEFORE_END: you insert A before the B ending marker
AFTER_END: you insert A after the B ending marker
```

BuildException 235

This exception is raised when the build process fails for some reason.

CbClass 237

...no description...

Fragment 240

...no description...

LocalVariableTableNotPresentException 245

When a needed LocalVariableTableAttribute is not found this exception is raised.

MultiAnnotation 246

Class that comes in help to parse java classes through the reflection in search of Annotation suitable from JCodeBrick.

B.1.1 Classes

B.1.2 Class BrickedClass

BrickedClass is intended to make easy the instantiation of manipulated CbClasses. It provides several overload of newInstance method

Declaration

public class BrickedClass
extends java.lang.Object

Constructors

- *BrickedClass*
  `public` **BrickedClass**`( java.lang.String` **realName**`, java.lang.Class` **c**`, byte []` **bytecode** `)`
  - **Usage**
    - Construct the BrickedClass on a real class name and an already loaded Class
  - **Parameters**
    - `realName` - the name of the real class before being manipulated
    - `c` - the Class this BrickedClass refers to

Methods

- *getClazz*
  `public Class` **getClazz**`( )`
  - **Usage**
    - Returns the java class represented by this BrickedClass.
- *getName*
  `public String` **getName**`( )`
  - **Usage**
    - Returns the name of the java class represented by this BrickedClass. This name wont be the name of the class you brick-operate onto, but the new generated name of the generated bricked class.
- *getRealName*
  `public String` **getRealName**`( )`
  - **Usage**
    - Returns the name of the class this BrickedClass refers to (the real one, not the bricked one).
- *newInstance*
  `public Object` **newInstance**`( )`
  - **Usage**
    - It returns an instance of this class for a constructor with no argument

  - **Returns** - an instance of type Class
  - **Exceptions**
    * `java.lang.InstantiationException` -
    * `java.lang.IllegalAccessException` -

- *newInstance*
  `public Object` **newInstance**`( java.lang.Class [] types, java.lang.Object [] args )`
  - **Usage**
    * It returns an instance of this class for a constructor with any kind of argument
  - **Parameters**
    * `types` - an array of N types, with N equal to the size of args
    * `args` - an array of N argument of type specified in types
  - **Returns** - an instance of type Class
  - **Exceptions**
    * `java.lang.SecurityException` -
    * `java.lang.NoSuchMethodException` -
    * `java.lang.IllegalArgumentException` -
    * `java.lang.InstantiationException` -
    * `java.lang.IllegalAccessException` -
    * `java.lang.reflect.InvocationTargetException` -

- *newInstance*
  `public Object` **newInstance**`( java.lang.Object [] args )`
  - **Usage**
    * It returns an instance of this class for a constructor with any kind of argument If the class constructor you want to call takes argument of primitive types (int,float..), then pass object of type Integer, Float.. and they will be automagically converted. If you want to use a constructor that need class Integer, Float.. you cannot use this method, see Object[]) overload
  - **Parameters**
    * `args` - The argument of the constructor you want to use: Integer, Float.. are converted in int, float..
  - **Returns** - an instance of type Class
  - **Exceptions**
    * `java.lang.SecurityException` -
    * `java.lang.IllegalArgumentException` -
    * `java.lang.NoSuchMethodException` -
    * `java.lang.InstantiationException` -
    * `java.lang.IllegalAccessException` -
    * `java.lang.reflect.InvocationTargetException` -

- *toClass*
  `public Class` **toClass**`( )`
  - **Usage**
    * Returns the java class represented by this BrickedClass.

- *writeToFile*
  `public void` **writeToFile**`( java.lang.String` **filename** `)`

B.1.3 Class **BrickedMethod**

This class allows you to invoke the code contained by a fragment.

You build the BrickedMethod with a source fragment, and a new class is created and loaded: the anonymous class which contains the fragment made method. The class is named "anonymousX", where X is an incremental number. The only method this class contains is:

```
public static returnType exec( argument )
```

The returnType type is set by watching inside the fragment code for a return instruction: if no returning instruction is found, then the method is "void"; if, for example, a LRETURN is found, then the method is "long". For any reference (ARETURN), the method is "Object".

The arguments are the free variables: see BrickedMethod#getArgumentSize() , BrickedMethod#getArgumentTypes() , BrickedMethod#getArgumentType(int)

Declaration

public class BrickedMethod
extends java.lang.Object

Constructors

- *BrickedMethod*
  `public` **BrickedMethod**`( org.ldp.jcodebrick.Fragment` **f** `)`
  - **Usage**
    - The default Constructor.

      When you constructs a BrickedMethod the anonymous class that will declare the fragment method is soon build. If this build fails, you'll get an exception. After the constructor call, you will be able to invoke the static method by using BrickedMethod#Invoke(Object...) , or you can get the java Method and use it by your own with BrickedMethod#getMethod()

Methods

- *getArgumentName*
  `public String` **getArgumentName**`( int` **idx** `)`
  - **Usage**
    - Returns the name of the argument at position `idx` that this bricked method takes. You have to specify all the arguments when calling BrickedMethod#Invoke(Object[])

- *getArgumentNames*
  `public String` **getArgumentNames**`( )`
  - **Usage**

    - Returns the name of the arguments this bricked method takes. You have to specify these arguments when calling BrickedMethod#Invoke(Object[])

- *getArgumentSize*
  `public int` **getArgumentSize**`( )`
  - **Usage**
    - Returns the number of arguments this bricked method takes. You have to specify these arguments when calling BrickedMethod#Invoke(Object[])

- *getArgumentType*
  `public String` **getArgumentType**`( int` **idx** `)`
  - **Usage**
    - Returns the type of the argument at position `idx` that this bricked method takes. You have to specify all the arguments when calling BrickedMethod#Invoke(Object[])

- *getArgumentTypes*
  `public String` **getArgumentTypes**`( )`
  - **Usage**
    - Returns the type of the arguments this bricked method takes. You have to specify these arguments when calling BrickedMethod#Invoke(Object[])

- *getDClass*
  `public DClass` **getDClass**`( )`
  - **Usage**
    - Returns the DClass used in this bricked method.

- *getMethod*
  `public Method` **getMethod**`( )`
  - **Usage**
    - Returns the java Method created by this BrickedMethod. Such method is both public and static, and has the name "exec". It's argument list is computed with the free variables list obtained by the fragment (variables that are used inside the block of code, but declared outside)

- *getSourceFragment*
  `public Fragment` **getSourceFragment**`( )`
  - **Usage**
    - Returns the original Fragment that this BrickedMethod has been created with.

- *getThrowableExceptionSize*
  `public int` **getThrowableExceptionSize**`( )`
  - **Usage**
    - Returns the number of exceptions this bricked method can throws.

- *getThrowableExceptionType*
  `public String` **getThrowableExceptionType**`( )`

  - **Usage**
    * Returns the type of the exceptions this bricked method can throws.

- *getThrowableExceptionType*
  `public String` **getThrowableExceptionType**`( int` **idx** `)`
  - **Usage**
    * Returns the type of the exception at position `idx` that this bricked method can throws.

- *Invoke*
  `public Object` **Invoke**`( )`
  - **Usage**
    * Invoke the fragment method with no argument.
  - **See Also**
    * `org.ldp.jcodebrick.BrickedMethod.Invoke(Object...)`

- *Invoke*
  `public Object` **Invoke**`( java.lang.Object [] args )`
  - **Usage**
    * Get the fragment method and invoke it. You can specify the arguments of the method, which will be mapped with the free variables in the block of code represented by the source fragment BrickedMethod#getSourceFragment() . If you want to be able to invoke the method by your self, for example to catch the exception, you can use BrickedMethod#getMethod() and then the Invoke standard reflection api.
  - **Returns** - If the code returned a value, you'll get such value as the return value of this method.
  - **See Also**
    * `org.ldp.jcodebrick.BrickedMethod.getMethod()`

- *methodInfo*
  `public String` **methodInfo**`( )`
  - **Usage**
    * Returns a full description of the bricked method.

- *showInfo*
  `public void` **showInfo**`( )`
  - **Usage**
    * Prints out the description of the bricked method.

B.1.4 Class **BrickOperation**

Declaration

---

public class BrickOperation
extends java.lang.Object

Methods

- *getBrickOperationInfo*
  `public String` **getBrickOperationInfo**`( )`
  - **Usage**
    - Returns an informational printable string.
- *getInsertionPosition*
  `public BrickOperation.InsertionPosition` **getInsertionPosition**`( )`
- *getNewJCodeBrickId*
  `public int` **getNewJCodeBrickId**`( )`
- *getObjectFragment*
  `public Fragment` **getObjectFragment**`( )`
- *getOperationType*
  `public BrickOperation.OperationType` **getOperationType**`( )`
- *getRemoveFromCbClass*
  `public CbClass` **getRemoveFromCbClass**`( )`
- *getSourceFragment*
  `public Fragment` **getSourceFragment**`( )`
- *setInsertionPosition*
  `public void` **setInsertionPosition**`(` `org.ldp.jcodebrick.BrickOperation.InsertionPosition` **position** `)`

B.1.5 Class **BrickOperation.InsertionPosition**

Position of fragment insertions.

You can insert a fragment A in four different positions relative to another fragment B.

```
BEFORE_START: you insert A before the B beginner marker
AFTER_START: you insert A after the B beginner marker
BEFORE_END: you insert A before the B ending marker
AFTER_END: you insert A after the B ending marker
```

Declaration

public static final class BrickOperation.InsertionPosition
extends java.lang.Enum

Methods

- *valueOf*
  `public static BrickOperation.InsertionPosition` **valueOf**`(` `java.lang.String` **name** `)`
- *values*
  `public static BrickOperation.InsertionPosition` **values**`( )`

Methods inherited from class `java.lang.Enum`

- *compareTo*
  `public final int` **compareTo**`( java.lang.Enum` **arg0** `)`
- *equals*
  `public final boolean` **equals**`( java.lang.Object` **arg0** `)`
- *getDeclaringClass*
  `public final Class` **getDeclaringClass**`( )`
- *hashCode*
  `public final int` **hashCode**`( )`
- *name*
  `public final String` **name**`( )`
- *ordinal*
  `public final int` **ordinal**`( )`
- *toString*
  `public String` **toString**`( )`
- *valueOf*
  `public static Enum` **valueOf**`( java.lang.Class` **arg0**`, java.lang.String` **arg1** `)`

B.1.6 Class BuildException

This exception is raised when the build process fails for some reason. In its details you can get all the information about the error.

Declaration

public class BuildException
extends java.lang.Exception

Serializable Fields

- private BrickOperation.OperationType operationType
  –
- private String errorMessage
  –

Constructors

- *BuildException*
  `public` **BuildException**`( )`
  – **Usage**
    * Constructs an empty exception.
- *BuildException*
  `public` **BuildException**`( org.ldp.jcodebrick.BrickOperation.OperationType` **operationType**`, java.lang.String` **errorMessage** `)`
  – **Usage**

    - Constructs the exception with given `operationType` and given `errorMessage`.

- *BuildException*
  `public` **BuildException**`( java.lang.String` **errorMessage** `)`
  - **Usage**
    - Constructs the exception with given `errorMessage`.

Methods

- *getErrorMessage*
  `public String` **getErrorMessage**`( )`
  - **Usage**
    - Returns the message associated to this exception.
- *getOperationType*
  `public BrickOperation.OperationType` **getOperationType**`( )`
  - **Usage**
    - Returns the operation type that raised this exception.
- *setOperationType*
  `public void` **setOperationType**`(`
  `org.ldp.jcodebrick.BrickOperation.OperationType` **optype** `)`
  - **Usage**
    - Set the operation type to this exception.
- *toString*
  `public String` **toString**`( )`
  - **Usage**
    - Returns the message associated to this exception.

Methods inherited from class `java.lang.Exception`

Methods inherited from class `java.lang.Throwable`

- *fillInStackTrace*
  `public synchronized native Throwable` **fillInStackTrace**`( )`
- *getCause*
  `public Throwable` **getCause**`( )`
- *getLocalizedMessage*
  `public String` **getLocalizedMessage**`( )`
- *getMessage*
  `public String` **getMessage**`( )`
- *getStackTrace*
  `public StackTraceElement` **getStackTrace**`( )`
- *initCause*
  `public synchronized Throwable` **initCause**`( java.lang.Throwable` **arg0** `)`
- *printStackTrace*
  `public void` **printStackTrace**`( )`

- *printStackTrace*
  `public void printStackTrace( java.io.PrintStream arg0 )`
- *printStackTrace*
  `public void printStackTrace( java.io.PrintWriter arg0 )`
- *setStackTrace*
  `public void setStackTrace( java.lang.StackTraceElement [] arg0 )`
- *toString*
  `public String toString( )`

B.1.7 Class CbClass

Declaration

public class CbClass
extends java.lang.Object

Constructors

- *CbClass*
  `public CbClass( java.lang.Class c )`
  - **Usage**
    - Constructs a CbClass reading the data from a java class. The class is inspected through the reflection and all the JCodeBrick annotations are loaded and indexed.

Methods

- *build*
  `public BrickedClass build( )`
  - **Usage**
    - Builds the CbClass to create a java class. It returns a BrickedClass , a container of the created java class.
- *getCbClassInfo*
  `public String getCbClassInfo( )`
  - **Usage**
    - Returns an informational printable string.
- *getClazz*
  `public Class getClazz( )`
  - **Usage**
    - Returns the class on which this CbClass has been created.
- *getDClass*
  `public DClass getDClass( )`
  - **Usage**
    - Used by the fragments.

- *getFragment*
  `public Fragment` **getFragment**`( java.lang.String` **name** `)`
  - **Usage**
    - Retrieve the first fragment found with the given name, or null if no fragment is found.

- *getFragment*
  `public Fragment` **getFragment**`( java.lang.String` **name**`, int` **n** `)`
  - **Usage**
    - Retrieve the n-th fragment with the given name searched in any method with no order, or null if no fragment is found.

- *getFragment*
  `public Fragment` **getFragment**`( java.lang.String` **name**`, java.lang.reflect.Method` **method**`, int` **n** `)`
  - **Usage**
    - Retrieve the n-th fragment with the given name in the given method
  - **Parameters**
    - `name` - the name of the annotation the block refers to
    - `method` - if not null, only block in the given method is searched
    - `n` - the n-th fragment found
  - **Returns** - the searched fragment, or null if no fragment is found.

- *getFragmentById*
  `public Fragment` **getFragmentById**`( int` **id** `)`
  - **Usage**
    - Retrieve the fragment with specified id. Returns null if no such fragment is found.

- *getFragments*
  `public Fragment` **getFragments**`( )`
  - **Usage**
    - Retrieve all the fragments found into this class.

- *getFragments*
  `public Fragment` **getFragments**`( java.lang.annotation.Annotation` **annotation** `)`
  - **Usage**
    - Retrieve all the fragments found into this class matching the given annotation.

- *getFragments*
  `public Fragment` **getFragments**`( java.lang.annotation.Annotation` **annotation**`, java.lang.reflect.Method` **method** `)`
  - **Usage**
    - Retrieve all the fragments found into this class matching the given annotation and method
  - **Parameters**
    - `annotation` - the annotation of the block

    * `method` - if Method is not given (null) this argument is ignored

---

- *getFragments*
  `public Fragment` **getFragments**`( java.lang.String` **name** `)`
  - **Usage**
    * Retrieve all the fragments found into this class matching the given name.

- *getFragments*
  `public Fragment` **getFragments**`( java.lang.String` **name**`, java.lang.reflect.Method` **method** `)`
  - **Usage**
    * Retrieve all the fragments found into this class matching the given name and method
  - **Parameters**
    * `name` - the name of the annotation of the block
    * `method` - if Method is not given (null) this argument is ignored

---

- *getFragmentsVector*
  `public Vector` **getFragmentsVector**`( )`
  - **Usage**
    * Retrieve all the fragments found into this class.

- *getFragmentsVector*
  `public Vector` **getFragmentsVector**`( java.lang.String` **name** `)`
  - **Usage**
    * Retrieve all the fragments found into this class matching the given name.

- *getFragmentsVector*
  `public Vector` **getFragmentsVector**`( java.lang.String` **name**`, java.lang.reflect.Method` **method** `)`
  - **Usage**
    * Retrieve all the fragments found into this class matching the given name and method
  - **Parameters**
    * `name` - the name of the annotation of the block
    * `method` - if Method is not given (null) this argument is ignored

---

- *getOperationAt*
  `public BrickOperation` **getOperationAt**`( int` **i** `)`
  - **Usage**
    * Returns the i-th registered operation.
  - **See Also**
    * `org.ldp.jcodebrick.CbClass.operationsSize()`
    * `org.ldp.jcodebrick.CbClass.addOperation(BrickOperation)`

- *operationsSize*
  `public int` **operationsSize**`( )`
  - **Usage**

    * Returns the number of BrickOperation registered to be executed at build time.

- *removeInsertedFragment*
  `public void` **removeInsertedFragment**`( int` **id**`, boolean` **onlyMarkers** `)`
  - **Usage**
    * Removes a fragment previously inserted with Fragment) .

      The new inserted fragment always receives a new unique jcodebrickid (the return value of Fragment)); you have to specify this id in order to remove the new inserted fragment. Consider that if you insert A into B, then B into C, and then you remove A with this function, you will remove A from B and from C.
  - **Parameters**
    * `id` - The jcodebrickid of a fragment previously inserted with Fragment)
    * `onlyMarkers` - if true, only the markers are removed, otherwise both the markers and the code.
  - **See Also**
    * `org.ldp.jcodebrick.Fragment.insertFragment(org.ldp.jcodebrick.BrickOperation.InsertionPo Fragment)`

- *showInfo*
  `public void` **showInfo**`( )`
  - **Usage**
    * Prints on the System.out informations about this CbClass.

B.1.8 Class **Fragment**

Declaration

public class Fragment
extends java.lang.Object

Methods

- *BrickBegin*
  `public static void` **BrickBegin**`( int` **id** `)`
  - **Usage**
    * Does nothing.

      This method is used by the JCodeBrick parser as marker at the begin of a fragment

- *BrickEnd*
  `public static void` **BrickEnd**`( int` **id** `)`
  - **Usage**

    * Does nothing.

      This method is used by the JCodeBrick parser as marker at the end of a fragment

---

- *buildFragmentSearcher*
  `public static InstructionSearcher` **buildFragmentSearcher**`( org.ldp.jdasm.attribute.CodeAttribute` **code**`, int` **jcodebrickid** `)`
  - **Usage**
    * Returns an InstructionSearcher able to find the fragment markers into given CodeAttribute .
  - **Parameters**
    * `code` - The CodeAttribute the fragment markers refers to
    * `jcodebrickid` - The unique id of the fragment markers

- *buildInstructionSearcher*
  `public InstructionSearcher` **buildInstructionSearcher**`( )`
  - **Usage**
    * Returns an InstructionSearcher able to find this fragment markers.

- *delete*
  `public void` **delete**`( )`
  - **Usage**
    * Deletes the fragment and its markers. No further operations that was referring to such markers will be available.
  - **See Also**
    * `org.ldp.jcodebrick.Fragment.undoDelete()`

- *delete*
  `public void` **delete**`( boolean` **withMarker** `)`
  - **Usage**
    * Deletes this fragment and all the code contained inside its marker.
  - **Parameters**
    * `withMarker` - if true, the markers are deleted too, otherwise the markers are left while the code is removed.
  - **See Also**
    * `org.ldp.jcodebrick.Fragment.undoDelete(boolean)`

- *deleteMarker*
  `public void` **deleteMarker**`( )`
  - **Usage**
    * Deletes the marker of this fragment. The fragment becomes untrackable and no further operations will be available on it.
  - **See Also**
    * `org.ldp.jcodebrick.Fragment.delete()`
    * `org.ldp.jcodebrick.Fragment.undoDeleteMarker()`

- *getAnnotation*
  `public Annotation` **getAnnotation**`( )`

- **Usage**
  - Returns the java Annotation associated to this Fragment.

- *getDMethod*
  `public DMethod` **getDMethod**`( )`
  - **Usage**
    - Returns the JDasm DMethod that encloses this Fragment.

- *getFragmentCodeAttribute*
  `public CodeAttribute` **getFragmentCodeAttribute**`( )`
  - **Usage**
    - Returns the code of this fragment.

- *getFragmentInfo*
  `public String` **getFragmentInfo**`( )`
  - **Usage**
    - Returns an informational printable string.

- *getFreeVariable*
  `public Fragment.FreeVariable` **getFreeVariable**`( int` **idx** `)`
  - **Usage**
    - Returns the free variable at given index.
  - **See Also**
    - `org.ldp.jcodebrick.Fragment.getFreeVariableSize()`
    - `org.ldp.jcodebrick.Fragment.getFreeVariables()`

- *getFreeVariables*
  `public Fragment.FreeVariable` **getFreeVariables**`( )`
  - **Usage**
    - Returns all the free variables of this fragment.
  - **See Also**
    - `org.ldp.jcodebrick.Fragment.getFreeVariableSize()`
    - `org.ldp.jcodebrick.Fragment.getFreeVariable(int)` (in B.1.8, page 242)

- *getFreeVariableSize*
  `public int` **getFreeVariableSize**`( )`
  - **Usage**
    - Returns the number variables used in this fragment that are declared somewhere in this fragment method outside the fragment.
  - **See Also**
    - `org.ldp.jcodebrick.Fragment.getFreeVariable(int)` (in B.1.8, page 242)
    - `org.ldp.jcodebrick.Fragment.getFreeVariables()`

- *getJCodebrickid*
  `public int` **getJCodebrickid**`( )`
  - **Usage**
    - Returns the JCodeBrick id associated to this Fragment.

- *getMethod*
  `public Method` **getMethod**`( )`
  - **Usage**
    * Returns the java Method that encloses this Fragment.

- *getMethodCodeAttribute*
  `public CodeAttribute` **getMethodCodeAttribute**`( )`
  - **Usage**
    * Returns the code of the method that encloses this fragment.

- *getName*
  `public String` **getName**`( )`
  - **Usage**
    * Returns the name of this Fragment (that is the annotation name).

- *insertFragment*
  `public int` **insertFragment**`(`
  `org.ldp.jcodebrick.BrickOperation.InsertionPosition` **where**`,`
  `org.ldp.jcodebrick.Fragment` **what** `)`
  - **Usage**
    * This is an overload of Fragment, String[]) that doesn't offer a way to have a variable mapping.
  - **Parameters**
    * `where` - The position the fragment will be insert into.
    * `what` - The fragment that must be inserted close to this one.
  - **Returns** - Returns a new unique jcodebrickid. This id can be used to abort the effect of this operation using CbClass#removeInsertedFragment(int) .
  - **See Also**
    * `org.ldp.jcodebrick.Fragment.insertFragment(org.ldp.jcodebrick.BrickOperation.InsertionPo Fragment, String[])`

- *insertFragment*
  `public int` **insertFragment**`(`
  `org.ldp.jcodebrick.BrickOperation.InsertionPosition` **where**`,`
  `org.ldp.jcodebrick.Fragment` **what**`, java.lang.String []`
  variableMapping `)`
  - **Usage**
    * Insert a fragment in a position relative to this fragment.

      You can insert any fragment in one of this four position:

      ```
      BrickOperation.InsertionPosition.BEFORE_START: the fragment is inserted before the
      BrickOperation.InsertionPosition.AFTER_START: the fragment is inserted after the st
      BrickOperation.InsertionPosition.BEFORE_END: the fragment is inserted before the en
      BrickOperation.InsertionPosition.AFTER_END: the fragment is inserted after the endi
      ```

      The new fragment will have a different two new markers at the beginning and at the end of its position. These markers use a new unique id, so delete operations on the original fragment

wont delete this new inserted one; to delete this fragment as subsequent operation you can call CbClass#removeInsertedFragment(int) using the unique id returned by this method.

When you perform an insertion of a fragment there are few things you should care of.

Try-Catch block: if the object fragment contains a code that was protected by a try block, (but no catch handler is part of the fragment) then it is added with a special try-catch block that catches any exception and raise a new java.lang.RuntimeException.

Returns: in the object fragment code you can find a return instruction whose type is incompatible with the return type of the method. If this situation occurs, this instruction is replaced with a return of the correct type according to the method return type: if it should return void, then a RETURN is added in place of the old return, otherwise two instructions are added: the first will push onto the stack a default value of the correct type (that is false, zero or null for a boolean, any number or any reference), and the second will return the pushed default value.

Variable mapping: you can encounter this situation:

```
..
int n = 10;
..
[Begin of the object Fragment]
.. int q = n / 5; ..
[End of the object Fragment]
..
```

In this case the object fragment uses the variable `n` but it doesn't know where it comes from. You can use a variable mapping to inform it to use another local variable when `n` is should be moved from the local variable array onto the stack. Example:

```
[method of the source Fragment]
..
int k = 10;
..
[point where the object Fragment is inserted]
..
```

Through the variable mapping (`variableMapping` argument) you can specify that the new fragment copied must use `k` instead of `n` when needed.

Other registers: any other register used that can conflict with the original method will be simply shifted to upper values.

– **Parameters**

    * `where` - The position the fragment will be insert into.
    * `what` - The fragment that must be inserted close to this one.
    * `variableMapping` - An array of even strings reporting at `k` position (with `k%2 == 0`) the name of the variable in the object fragment that must be covered by the variable in the source method named as specified at position `k+1`
  - **Returns** - Returns a new unique jcodebrickid. This id can be used to abort the effect of this operation using CbClass#removeInsertedFragment(int) .
  - **See Also**
    * `org.ldp.jcodebrick.Fragment.getUsedVariableSize()`
    * `org.ldp.jcodebrick.Fragment.getFreeVariableNames()`
    * `org.ldp.jcodebrick.Fragment.getUsedVariableTypes()`

- *insertWrap*
  `public void` **insertWrap**`(`
  `org.ldp.jcodebrick.BrickOperation.InsertionPosition` **position**`, java.lang.String` **className**`, java.lang.String` **methodName** `)`
  - **Usage**
    * Insert a call to a static method at specified `position` in the current fragment. The method must be specified by the name of its class and its method name. It must be a static public method, with no arguments and returning void.

- *showInfo*
  `public void` **showInfo**`( )`
  - **Usage**
    * Shows informations about this fragment.

- *toString*
  `public String` **toString**`( )`
  - **Usage**
    * Returns an informational printable string.

B.1.9 Class **LocalVariableTableNotPresentException**

When a needed LocalVariableTableAttribute is not found this exception is raised.

Declaration

public class LocalVariableTableNotPresentException
extends java.lang.Exception

Constructors

- *LocalVariableTableNotPresentException*
  `public` **LocalVariableTableNotPresentException**`( )`

Methods inherited from class `java.lang.Exception`

Methods inherited from class `java.lang.Throwable`

- *fillInStackTrace*
  `public synchronized native Throwable` **fillInStackTrace**`( )`
- *getCause*
  `public Throwable` **getCause**`( )`
- *getLocalizedMessage*
  `public String` **getLocalizedMessage**`( )`
- *getMessage*
  `public String` **getMessage**`( )`
- *getStackTrace*
  `public StackTraceElement` **getStackTrace**`( )`
- *initCause*
  `public synchronized Throwable` **initCause**`( java.lang.Throwable` **arg0** `)`
- *printStackTrace*
  `public void` **printStackTrace**`( )`
- *printStackTrace*
  `public void` **printStackTrace**`( java.io.PrintStream` **arg0** `)`
- *printStackTrace*
  `public void` **printStackTrace**`( java.io.PrintWriter` **arg0** `)`
- *setStackTrace*
  `public void` **setStackTrace**`( java.lang.StackTraceElement []` **arg0** `)`
- *toString*
  `public String` **toString**`( )`

B.1.10 Class MultiAnnotation

Class that comes in help to parse java classes through the reflection in search of Annotation suitable from JCodeBrick. Such annotation are at method level and present two values: one called MultiXX (where XX is the name of the annotation used to mark pieces of code) which is an array of all the annotations used inside the method, and one called jcodebrickids, an array of all the unique id associated to each of such annotations.

Once these annotations are correctly recognized, this class offers method to access them one by one.

Declaration

public class MultiAnnotation
extends java.lang.Object

Constructors

- *MultiAnnotation*
  `public` **MultiAnnotation**`( java.lang.annotation.Annotation []` **ann** `)`
  - **Usage**
    - The constructor takes an array of all the annotation of a method. In these annotation you can find any number of JCodeBrick suitable annotation: This instance will parse them to identify the suitable ones.

Methods

- *fragmentCount*
  `public int` **fragmentCount**`( )`
  - **Usage**
    - Returns the count of the valid annotation found.

- *getAnnotation*
  `public Annotation` **getAnnotation**`( int` **idx** `)`
  - **Usage**
    - Returns the idx-th annotation.

- *getCodebrickid*
  `public int` **getCodebrickid**`( int` **idx** `)`
  - **Usage**
    - Returns the jcodebrickid of idx-th annotation.

- *getUnusedCodeBrickId*
  `public static int` **getUnusedCodeBrickId**`( )`
  - **Usage**
    - Returns an id suitable as jcodebrickid for new fragment insertion. The class ensures that this id is the higher id it has never met, resulting in a unique id over all the loaded fragments.

- *isMultiAnnotation*
  `public static boolean` **isMultiAnnotation**`(`
  `java.lang.annotation.Annotation` **ann** `)`
  - **Usage**
    - Returns true if the given annotation is a JCodeBrick annotation. A JCodeBrick annotation is an annotation originally used to mark a piece of code and then moved by a parser outside the method body, and now associated to the body.

Contents

1 Introduction **1**

2 Java Annotations **6**
2.1 The annotations model in Java 5 6
2.2 Limitations of the Java 5 annotation model 7

3 The @Java language **9**
3.1 Syntax extension 9
3.2 Compilation strategy 10
3.3 The preprocessor parser 11
3.3.1 Definitions 12
3.3.2 Implementations 14

4 Bytecode Engineering through JDAsm **16**
4.1 Structure 16
4.2 Constant pool 17
4.3 Fields and Methods 20
4.4 Attributes 20
4.5 Code Attribute 22
4.6 Advanced Features 28
4.7 Build time 28
4.8 Benchmarks 29

5 Manipulating annotated code: JCodeBrick **31**
5.1 Notation and definitions 31
5.2 Basic implementation 33
5.3 Operations 34
5.3.1 Search 34
5.3.2 Insertion 36
5.3.3 Deletion 40
5.3.4 Extrusion 41
5.4 Build Operation 45
5.5 Examples 47

6 Applications **50**
6.1 Logging 50
6.2 Environment-based reconfiguration 51
6.3 Dynamic optimization 51
6.4 Adaptable declarative security 52
6.5 Parallelization 52

7 Case Study **53**

8 Conclusions and future work **58**

A JDAsm API Documentation **62**
A.1 **Package org.ldp.jdasm** 62
A.1.1 Classes 63
A.1.2 Class **DClass** 63
A.1.3 Class **DConstantPool** 73
A.1.4 Class **DField** 76
A.1.5 Class **DMethod** 82
A.1.6 Class **DAttribute** 90
A.2 **Package org.ldp.jdasm.constantpool** 95
A.2.1 Interfaces 96
A.2.2 Interface **IBuildable** 96
A.2.3 Classes 97
A.2.4 Class **AConstantPoolUser** 97
A.2.5 Class **CONSTANT_Class_Info** 98
A.2.6 Class **CONSTANT_Double_Info** 101
A.2.7 Class **CONSTANT_FieldRef_Info** 103
A.2.8 Class **CONSTANT_Float_Info** 107
A.2.9 Class **CONSTANT_Integer_Info** 109
A.2.10 Class **CONSTANT_InterfaceMethodRef_Info** . . . 111
A.2.11 Class **CONSTANT_Long_Info** 115
A.2.12 Class **CONSTANT_MethodRef_Info** 117
A.2.13 Class **CONSTANT_NameAndType_Info** 121
A.2.14 Class **CONSTANT_String_Info** 125
A.2.15 Class **CONSTANT_Utf8_Info** 127
A.2.16 Class **CpInfo** 129
A.2.17 Class **TypeDescriptor** 131
A.3 **Package org.ldp.jdasm.attribute** 134
A.3.1 Classes 136
A.3.2 Class **Attributable** 136
A.3.3 Class **CodeAttribute** 138
A.3.4 Class **ConstantValueAttribute** 152
A.3.5 Class **CustomAttribute** 159
A.3.6 Class **DeprecatedAttribute** 165
A.3.7 Class **ExceptionsAttribute** 170
A.3.8 Class **InnerClassesAttribute** 176
A.3.9 Class **InnerClassesElement** 182
A.3.10 Class **LineNumberTableAttribute** 185
A.3.11 Class **LocalVariableTableAttribute** 191
A.3.12 Class **LocalVariableTableElement** 198
A.3.13 Class **LocalVariableTypeTableAttribute** 199
A.3.14 Class **SignatureAttribute** 205
A.3.15 Class **SourceDebugExtensionAttribute** 211
A.3.16 Class **SourceFileAttribute** 217
A.3.17 Class **StackMapTableAttribute** 223
A.3.18 Class **SyntheticAttribute** 223

B JCodeBrick API Documentation **228**
B.1 **Package org.ldp.jcodebrick** 228
B.1.1 Classes 229
B.1.2 Class **BrickedClass** 229
B.1.3 Class **BrickedMethod** 231
B.1.4 Class **BrickOperation** 233
B.1.5 Class **BrickOperation.InsertionPosition** 234
B.1.6 Class **BuildException** 235
B.1.7 Class **CbClass** 237
B.1.8 Class **Fragment** 240

B.1.9 Class **LocalVariableTableNotPresentException** . . 245
B.1.10 Class **MultiAnnotation** 246