Barista 2.0-alpha

Reference Manual

http://barista.x9c.fr

July 14, 2011

Abstract

This document presents Barista, the command-line program as well as the underlying Objective Caml library. The purpose of both is to allow easy manipulation of Java™ class files in their compiled (*a.k.a.* bytecode form).

The program allows one to asssemble and disassemble class file, respectively from and to a source format called assembler (which is essentially a human-readable form of the bytecode). The program also allows one to retrieve information about a classfile, by printing either the contents of a class or the control flow of a method.

The Objective Caml library (upon which the program is built) allows more sophisticate analysis and manipulation of class file (and also package and module files). Its primary goal is to be used as the back-end for code generation in the OCaml-Java project (`http://ocamljava.x9c.fr`), which provides a compiler from Objective Caml sources into Java™ class files.

This manual is structured in five chapters, and three appendixes.
After a first chapter providing an overview of the Barista project, the second chapter explains how Barista can be built from sources and details its dependencies. Then, chapter three introduces the Barista program and how it can be executed, while chapter four introduces the the format of so-called *assembler* source file. The last chapter shed light on Barista as an Objective Caml library and provides entry points to its API. Finally, the three appendixes summarize all the instructions recognized by the Barista assembler, classified under three different ways (namely: alphabetically, by categories, and by opcode number) for easy reference.

Contents

Chapter 1

Overview

This chapter presents some elements about the Barista project: its contents and objectives, its evolution, and also its management.

1.1 Current state

Barista is initially an Objective Caml[1] library designed to load, construct, manipulate and save Java™[2] class files. The library supports the whole class file format as defined by Oracle®(formerly Sun®). The Barista library was designed to be used in the OCaml-Java project (available at `http://ocamljava.x9c.fr`) for code generation.

Upon the library, a command-line utility (also named "barista") has been developped: both an assembler and a disassembler for the Java™ platform. In its 2.0-alpha version, Barista supports Java™ 1.7 and needs Objective Caml 3.12.1 to build. Code sample 1 below shows the canonical basic example coded in the Barista assembler; the assembler will turn it into a class file to be run into ja Java™ virtual machine. Chapter 4 describes the format of such assembler sources.
The disassembler does the same work in the opposite direction: it takes the fully qualified name of a Java™ bytecode class file present in the classpath, and transforms it into an assembler source. Two other utility allow to inspect the contents of a bytecode file; it is possible to just print the list of methods of a given class, and also to print the control flow of a given method as a graph.

1.2 Some history

From version 1.0β to 1.4, Barista also featured a Java™ API allowing to use Barista directly from Java™. This API has been removed as of version 2.0. This removal was motivated by the following three reasons, sorted from most to least valuable. First, it introduced a circular dependency between Barista and the rest of the OCaml-Java project, intricating both. Second, it was a burden to manually update the API in order to keep it a Java^{TM} look and feel. Third,

[1]The official Objective Caml website can be reached at `http://caml.inria.fr` and contains the full development suite (compilers, tools, virtual machine, *etc.*) as well as links to third-party contributions.

[2]The official Java™ website can be reached at `http://java.sun.com` where most of official Java™ information can be found.

it was pretty unused; those who actually want to use Barista from Java™ should take a look at the OCaml-Java project, and the ways it provides to interface Objective Caml and Java™ codes.

The 2.0 version of Barista is quite a leap forward: it claims full support for Java™ 1.7 (including serialization, various optimization on produced class files, and handling of package files), and has seen a major overhaul of the API to make it as simple, readable, and type-safe as can be. Of course, this induced backward-incompatible changes which motivated the jump in version numbering from 1.4 to 2.0.

1.3 Glimpse of the assembler syntax

Before dwelling upon technical issues in the next two chapters, here is the classical hello world example as it should be programmed in the Barista assembler (see code sample 1).

Code sample 1 The classical "hello world" in Barista assembler.

```
.class public final pack.Test
.extends java.lang.Object

.method public static void main(java.lang.String[])
        getstatic java.lang.System.out : java.io.PrintStream
        ldc "hello world.\n"
        invokevirtual java.io.PrintStream.println(java.lang.String):void
```

As in most so-called assembly languages, the source format is line-oriented (meaning that in most cases an instruction cannot be split over multiple lines), and *directives* (how to interpret things) are discriminated from *instructions* (what things should be done) by prefixing the directive with a "`.`" character (*i.e.* a dot).

To be able to use the assembler, and to understand this very manual, good familiarity with Java™ in general and with the class file format in particular is assumed. One should refer to the documentation published by Oracle® for more information (a good entry point for the JVM specification being `http://java.sun.com/docs/books/jvms/`).

1.4 Contributions

In order to improve the project, I am primarily looking for testers and bug reporters. Pointing errors in documentation and indicating where it should be enhanced is also very helpful.
Bug reports can be made at `http://bugs.x9c.fr`.
Other requests can be sent to barista@x9c.fr.

Chapter 2

Building Barista from sources

This chapter first lists the various libraries and tools Barista depends upon. Then, the following sections explains the steps to follow in order to build, and then install Barista (both the program and the library) from sources.

2.1 Dependencies

In order to compile the Barista sources, one needs Objective Caml 3.12.1, and GNU make 3.81. Compilation is actually realized by the `ocamlbuild` tool, although launched through the targets of a `Makefile`. Additionally, Barista depends upon the following Objective Caml libraries that should hence be installed before build:

- **bigarray**, **str**, **unix** available in the Objective Caml standard distribution;
- **camlzip** (zip/gzip/jar library), at least version 1.04
  available at: `http://cristal.inria.fr/~xleroy/software.html`;
- **camomile** (Unicode library), at least version 0.8.3
  available at: `http://camomile.sourceforge.net`.

2.2 Configuration

Before the actual compilation could be launched, it is necessary to configure the build process to the actuel machine and system used. To this end, one should execute the `configure` script (*e.g.* by running `sh configure`). This script recognizes the following command-line options, that can be used to supersede the autodetection the script performs:

- `-ocaml-prefix` to indicate where Objective Caml should actually be found (it should hence be the same value as the one passed to Objective Caml's `configure` script through the `-prefix` option);
- `-ocamlfind` to indicate the path to the `ocamlfind`[1] program (setting it to the empty string will disable the use of `ocamlfind`).

Upon sucessful execution, a `Makefile.config` file is produced which contains the information either inferred by or passed to the script.

[1]Findlib, a library manager for Objective Caml, is available at `http://projects.camlcity.org/projects/findlib.html`

2.3 Compilation

Once configured, Barista can be built by calling the `make` program.
The following targets are available:

- `all` compiles all files
- `doc` generates ocamldoc documentations
- `tests` runs tests
- `clean` deletes all produced files (excluding documentation)
- `veryclean` deletes all produced files (including documentation)
- `install` copies executable and library files
- `generate` generates files needed for build

2.4 Installation

Finally, as shown in the previous section, the installation of Barista is done by calling `make install`. However, two important things should be noted about installation.

First, the actual installation path depends on whether `ocamlfind` is enabled: if so, every file will be installed in the `ocamlfind` hierarchy; otherwise, the library will be installed in the .../`lib/ocaml/barista` directory, and the program executables will be installed in the same directory as the Objective Caml compilers.

Second, to install the files to their final destination, the user may need to have to acquire privileged rights; this is usually done through the `sudo` command, leading to a full installation command of `sudo make install`.

Chapter 3

Using Barista as a command-line program

The Barista program exists under two or three forms: `barista.byte`, `barista.native`, and optionally `barista.jar`. The first one is an Objective Caml bytecode file, the second one is a native executable, and the third one is an executable jar file compiled only if the `ocamljava` compiler is present. All three programs should behave the same way, only differing in execution speed.

For convenience, at installation a symbolic link is created from `barista` to either `barista.native` or `barista.byte` in the installation directory. This allows one to only refer to `barista` in order to launch the faster version of the program.

Since version 2.0, Barista uses the concept of *subcommands* (as usually found *e.g.* in version management systems) to let the user indicate which action to perform. As natural in this kind of settings, the command-line switches recognized depend on the subcommand that is passed immediatly after the name of the executable. As an example, this means that invoking Barista to assemble a file is done by the following command line:

```
barista assemble source.j
```

The following sections presents the various available commands. All commands invoving a classpath set by default this value to the value of the environment variable named `CLASSPATH` if it exists, to "`.`" (*a.k.a.* dot) otherwise.

3.1 The `assemble` subcommand (alias: `asm`)

Assembles (*i.e.* compiles) the passed assembler files into Java™ bytecode class files.

The `assemble` subcommand recognizes the following parameters:

- `-cp` ⟨path⟩ add to classpath
- `-classpath` ⟨colon-separated-list⟩ set classpath
- `-compute-stacks` computes stack elements (max_locals, max_stacks, and frames)
- `-d` ⟨path⟩ output path for generated class files

- `-destination` ⟨path⟩ output path for generated class files
- `-optimize` optimize bytecode
- `-target` target version for generated class files (possible values being: `1.0`, `1.1`, `1.2`, `1.3`, `1.4`, `1.5`, `1.6`, `1.7`, `1.8`)

3.2 The `disassemble` subcommand (alias: `dasm`)

Disassembles (*i.e.* decompiles) the passed Java™ bytecode class files into assembler files.

The `disassemble` subcommand recognizes the following parameters:

- `-cp` ⟨path⟩ add to classpath
- `-classpath` ⟨colon-separated-list⟩ set classpath

3.3 The `flow` subcommand

Prints onto the standard output the control flow of the passed methods. The output format is the dot format[1]. The passed methods should follow the format for method signatures as presented in chapter . As an example, printing the control flow for the `toString` method is done by the following command line:

```
barista flow 'java.lang.Object.toString():java.lang.String'
```

It is important to enclose the signature of the method inside quotes, as otherwise the shell whould interpret the parentheses.

The `flow` subcommand recognizes the following parameters:

- `-cp` ⟨path⟩ add to classpath
- `-classpath` ⟨colon-separated-list⟩ set classpath

3.4 The `print` subcommand

Prints the contents of the passed Java™ bytecode class files onto the standard output.

The `print` subcommand recognizes the following parameters:

- `-cp` ⟨path⟩ add to classpath
- `-classpath` ⟨colon-separated-list⟩ set classpath

3.5 The `version` subcommand

Prints the current Barista version onto the standard output.

The `version` subcommand recognizes no parameter.

[1] http://www.graphviz.org/doc/info/lang.html

3.6 The `help` subcommand

The `help` subcommand basically writes on the standard error stream all the information contained in the previous sections of this chapter.

Chapter 4

Programming in Barista assembler language

This chapter describes the format of the source files that the Barista assembler accepts. The very same format is used by the Barista disassembler as its output format. This makes it easy to disassemble a class, modify the disassembled source, and then reassemble it. Each file will be assembled to a single class file.

4.1 Lexical elements

The source files used by the Barista assembler are line-oriented (this means that source elements are analyzed line by line). Over a line, elements are whitespace-separated, meaning that whitespaces are meaningful. The different lexical elements are presented below.

Comments are introduced by the `#` (sharp) character and terminate at the end of the line.

Assembler directives are introduced by the `.` (dot) character followed by a non-empty sequence of lowercase letters or underscores *e.g.* `.class`.

Element attributes are introduced by the `@` (at) character followed by a non-empty capitalized sequence of letters *e.g.* `@ConstantValue`.

Labels are non-empty sequences of letters and digits beginning with a letter and ending with a colon *e.g.* `aLabel3:`.

Integer constants follow the Objective Caml conventions and thus support decimal notation (*e.g.* `15`), hexadecimal notation (*e.g.* `0x0F`), octal notation (*e.g.* `0o17`) and binary notation (*e.g.* `0b1111`). All notations support embedded underscore characters used for increased readability (such underscore characters are ignored). Integer constants can also be defined using the character notation (*e.g.* `'a'`); this notation supports escaped sequence for ASCII characters (*e.g.* `'\t'`, `'\n'`, or `'\\'`) as well as for Unicode character in short (*e.g.* `'\u1234'`) and long formats (*e.g.* `'\U12345678'`).

Floating-point constants follow the Objective Caml conventions and thus consist of three part: an integral part, a decimal part and an optional exponent part (*e.g.* `-1.234e+2`). As for integer constants, embedded underscore characters can be used for readability.

String constants are presented between `"` quotes, and support the escaped sequences presented for integer character constants *e.g.* `"abc\tdef"`.

Class names should be given in *external* format, that is using dots between packages/classes and dollars between inner classes (*e.g.* `pack.Cls$Inn` for an inner-class *Inn* of a class named *Cls* in package *pack*).

Primitive names should appear as they would be in a Java™ source file (*e.g.* `boolean`).

Array types are either a class name or a primitive type name followed by a non-empty list of square brackets `[]` pairs (*e.g.* `int[][]` or `java.lang.String[]`).

Field references are defined by a qualified field name followed by a colon and the field type (*e.g.* `java.lang.String.CASE_SENSITIVE_ORDER:java.util.Comparator`).

Dynamic method references are defined by an unqualified method name followed by the parameters between paretheses, then followed by a colon and the return type
e.g. `toString():java.lang.String`.

Method references are defined by a qualified method name followed by the parameter types between parentheses, then followed by a colon and the return type
e.g. `java.lang.Object.toString():java.lang.String` or `int[].toString():java.lang.String`.

Method signatures are defined by an unqualified method name followed by the parameter types between parentheses *e.g.* `toString()`.

Method types are defined by the parameter types between parentheses, then followed by a colon and the return type *e.g.* `(int,int):void`.

Method handles are defined by a prefix and a suffix separated by a `%` character. The prefix defines the kind of entity pointed by the handle, while the suffix defines an actual reference to the entity. The possible prefix/suffix combinations are:

- `getField` followed by a field reference;
- `getStatic` followed by a field reference;
- `putField` followed by a field reference;
- `putStatic` followed by a field reference;
- `invokeVirtual` followed by a method reference;
- `invokeStatic` followed by a method reference;

- `invokeSpecial` followed by a method reference;
- `newInvokeSpecial` followed by a method reference;
- `invokeInterface` followed by a method reference.

Examples:

- `getStatic%java.lang.String.CASE_SENSITIVE_ORDER:java.util.Comparator`
- `invokeVirtual%java.lang.Object.toString():java.lang.String`

Identifiers are non-empty sequences of letters and digits beginning with a letter *e.g.* `anIdent2`.

Arrows are simply the character sequence `=>`, and are used for the encoding of the `lookupswitch`, and `tableswitch` cases.

Tildes are simply the character `~`, and are used as a separator between stack elements and locals in stack frames.

4.2 Assembler directives

The assembler directives define the elements of a Java™ class as well as their main properties. They are used by the programmer to tell the assembler which elements are part of a class file. The different assembler directives are presented below.

4.2.1 The `class` directive

Written `.class` *flags name* where *flags* is a list of class flags[1] (among `public`, `final`, `super`, `interface`, `abstract`, `synthetic`, `annotation`, `enum`), and *name* is a fully qualified class name. This directive should be the first one of the source file.

4.2.2 The `extends` directive

Written `.extends` *name* where *name* is a fully qualified class name. This directive sets the class parent and should be present even if the parent is `java.lang.Object`; this directive may be missing if and only if the class defined by the source file has no parent (it should only be true for the `java.lang.Object` class).

4.2.3 The `implements` directive

Written `.implements` *name* where *name* is a fully qualified class name (that should be an interface).

[1] One should notice that flags discussed in this document are JVM-level flags, and not Java™ language-level flags. Hence the presence of flag that cannot occur in Java™ sources.

4.2.4 The `field` directive

Written `.field` *flags type name* where *flags* is a list of field flags (among `public`, `private`, `protected`, `static`, `final`, `volatile`, `transient`, `synthetic`, `enum`), *type* is either a primitive, a fully qualified class name or an array type, and *name* is a field name. This directive adds a field to the class.

4.2.5 The `method` directive

Written `.method` *flags returntype signature* where *flags* is a list of method class flags (among `public`, `private`, `protected`, `static`, `final`, `synchronized`, `bridge`, `varargs`, `native`, `abstract`, `strict`, `synthetic`), *returntype* is either a primitive, a fully qualified class name, or an array type, and *signature* is a method signature. This directive adds a method to the class.

4.2.6 The `max_stack` directive

Written `.max_stack` n where n is an integer in the interval from 0 to 65535 (both inclusive). This directive sets the maximum stack size for the current method.

4.2.7 The `max_locals` directive

Written `.max_locals` n where n is an integer in the interval from 0 to 65535 (both inclusive). This directive sets the maximum local size for the current method.

4.2.8 The `catch` directive

Written `.catch` *start end handler* [*name*] where *start*, *end* and *handler* are labels refering to respectively the begin and the end of the protected code area, and to the associated exception handler. The optional *name* is a fully qualified class name giving the name of the exception class to be handled by the code located at handler label; if this name is missing, all exceptions will be caught.

4.2.9 The `frame` directive

Written `.frame` l def where l is the label where the frame definition def applies. This directive allows to specify elements for the `StackMapFrame` attributes of the class file. The frame definition (noted def above) can take of of the following forms:

- `same` to indicate that the frame definition is the same as the previous one;
- `same_locals` t to indicate that the frame definition is the same as the previous one regarding locals and has one element on the stack whose type is t;
- `chop` n to indicate that the frame definition is the same as the previous one, chopped by n elements (n should be `1`, `2`, or `3`);
- `append` t_1 [t_2 [t_3]] indicate that the frame definition is the same as the previous one, with one to three elements appended (whose types are given by the t_i);
- `full` t_1 ... t_n `~` t'_1 ... t'_m to define a frame with no reference to the previous one, the t_i are the types for locals while the t'_j ones are the types for stack elements (the tilde symbol separating the two lists);

The t, t_i, and t'_j are type definitions whose possible values are:

- `top` for the *top* type;
- `int` for the `int` type;
- `float` for the `float` type;
- `long` for the `long` type;
- `double` for the `double` type;
- `null` for the type associated to the *null* value;
- `uninit_this` for the uninitialized *this* reference;
- `uninit` l for an uninitialized reference whose related `new` instruction is located at the offset given by label l;
- a fully-qualified class name for the type associated to this class.

4.3 Attributes

The assembler attributes define the properties of a Java™ element. An attribute is applied to the latest defined Java™ element by a `.class`, `.field` or `.method` directive. (one should notice that some attributes are only used inside a given element kind). The different elements attributes are presented below.

4.3.1 The `ConstantValue` attribute

Written `@ConstantValue` *value* (for fields only), it defines the initial value of the field. *value* should be compatible with the field type and can be a float, an integer or a string. A `false` boolean is coded by a `0` integer while every other values code a `true` boolean

4.3.2 The `Exceptions` attribute

Written `@Exceptions` *non-empty-list* (for methods only), it defines the list of exceptions (as fully qualified class names) that the method can throw

4.3.3 The `InnerClasses` attribute

Written `@InnerClasses` *ic oc n flags* (for classes only), it adds an inner-class information. *ic* is the fully qualified name of the inner-class (or `0` if this information is missing), *oc* is the fully qualified name of the outer-class (or `0` if this information is missing), *n* is the name of the inner-class in the outer-class (or `0` if this information is missing), *flags* is a list of inner-class flags (among `public`, `private`, `protected`, `static`, `final`, `super`, `interface`, `abstract`, `synthetic`, `annotation`, `enum`)

4.3.4 The `EnclosingMethod` attribute

Written `@EnclosingMethod` *name meth* (for classes only), it adds an enlosing-method information. *name* is the fully qualified name of the enclosed class, *meth* is the enclosing method specified as a dynamic method (or `0` if this information is missing)

4.3.5 The `Synthetic` attribute

Written `@Synthetic` (for classes, fields, and methods), it marks the element as synthetic (*i.e.* compiler-generated)

4.3.6 The `Signature` attribute

Written `@Signature` *string* (for classes, fields, and methods), it sets the signature of the element

4.3.7 The `SourceFile` attribute

Written `@SourceFile` *string* (for classes only), it sets the source file name

4.3.8 The `SourceDebugExtension` attribute

Written `@SourceDebugExtension` *string* (for classes only), it sets the source debug extension

4.3.9 The `Deprecated` attribute

Written `@Deprecated` (for classes, fields, and methods), it marks the element as deprecated

4.3.10 The `RuntimeVisibleAnnotations` attribute

Written `@RuntimeVisibleAnnotations` *elems* (for classes, fields, and methods), it adds an annotation to the element (the format of annotations is discussed below)

4.3.11 The `RuntimeInvisibleAnnotations` attribute

Written `@RuntimeInvisibleAnnotations` *elems* (for classes, fields, and methods), it adds an annotation to the element (the format of annotations is discussed below)

4.3.12 The `RuntimeVisibleParameterAnnotations` attribute

Written `@RuntimeVisibleParameterAnnotations` n *elems* (for methods only), it adds an annotation to the element for the parameter at index n (the format of annotations is discussed below)

4.3.13 The `RuntimeInvisibleParameterAnnotations` attribute

Written `@RuntimeInvisibleParameterAnnotations` n *elems* (for methods only), it adds an annotation to the element for the parameter at index n (the format of annotations is discussed below)

4.3.14 The `AnnotationDefault` attribute

Written `@AnnotationDefault` *elems* (for methods only), it adds an annotation default to the element (the format of annotation defaults is discussed below)

4.3.15 The `LineNumberTable` attribute

Written `@LineNumberTable` [n] (for methods only), it maps the current code offset to a line number. The line number is n if provided, otherwise it is the current line of the Barista source file

4.3.16 The `LocalVariableTable` attribute

Written `@LocalVariableTable` *start end id t idx* (for methods only), it adds a type information for a local variable. *id* and *t* are the variable identifier and type, *idx* is its position in the locals, and *start* (inclusive) and *end* (exclusive) are labels defining the portion of code where this variable is defined

4.3.17 The `LocalVariableTypeTable` attribute

Written `@LocalVariableTypeTable` *start end id s idx* (for methods only), it adds a type information for a local variable. *id* and *s* are the variable identifier and signature, *idx* is its position in the locals, and *start* (inclusive) and *end* (exclusive) are labels defining the portion of code where this variable is defined

4.3.18 The `Unknown` attribute

Written `@Unknown` *string*1 *string*2 (for classes, fields, and methods), it adds an unknown (*i.e.* implementation-dependent) attributes to the element. *string*1 is the identifier of the attribute while *string*2 is its value

4.4 Annotations

Annotations with their key-value attributes and possibly embedded annotations form a tree-like structure. The Barista assembler sources being line-oriented, annotations are organized in a way such that lines with the same prefix gives informations for the same subtree.

On a line defining an annotation, the first element should be the class of the annotation. This fully qualified class name is followed by the key identifier and its associated value. The value is itself a couple; the first component is the value type while the second component is the actual value. Code sample 2 shows an annotation (whose class is `pack.AnnotationClass`) with two parameters:

- parameter `a` whose type is *string* and value `"xyz"`;
- parameter `b` whose type is *float* and value `3.14`.

Code sample 2 Annotation example.

```
@RuntimeVisibleAnnotation pack.AnnotationClass a string "xzy"
@RuntimeVisibleAnnotation pack.AnnotationClass b float 3.14
```

The possible types for an annotation attribute are the Java™ primitive types, extended with the following ones:

- `string` for string values (using the format of string constants discussed above);
- `enum` for enum values (defined by the fully qualified name of the enum class followed by the identifier of the enum value);
- `class` for reference to a given class (followed by the fully qualified name of the class);
- `annotation` for embedded annotation (followed by an annotation value using the format explained in this very section);
- an array is not introduced by any keyword, each value being introduced by its index[2] (as an integer). The type of each embedded element should be repeated.

Code sample 3 shows the previous annotation enriched with three values:

- parameter `c` whose type is *array* and value is a two-element array containing `5` and `7`;
- parameter `d` whose type is *annotation* and value `java.lang.Deprecated`;
- parameter `e` whose type is *enum* and value `pack.EnumClass.E1`.

Code sample 3 Annotation example.

```
@RuntimeVisibleAnnotation pack.AnnotationClass a string "xzy"
@RuntimeVisibleAnnotation pack.AnnotationClass b float 3.14
@RuntimeVisibleAnnotation pack.AnnotationClass c 0 int 5
@RuntimeVisibleAnnotation pack.AnnotationClass c 1 int 7
@RuntimeVisibleAnnotation pack.AnnotationClass d annotation java.lang.Deprecated
@RuntimeVisibleAnnotation pack.AnnotationClass e enum pack.EnumClass E1
```

4.5 Annotation defaults

Annotation defaults closely follow the notation used for annotation, except that leading annotation class name as well as attribute name should be omitted. Code sample 4 defines an annotation default whose value is the `"xyz"` string.

Code sample 4 Annotation default example.

```
@AnnotationDefault string "xyz"
```

[2]These indexes do not need to be successive, they are only used to sort the values in order to produce the array.

4.6 Instructions

Instructions may, of course, be used only inside methods. An instruction line may contain either a label, or an instruction (along with its parameters), or both. One should refer to the JVM specification for the list of available instructions; this specification also defines the parameters waited by each instruction. The translation of parameters from the specification to their Barista counterpart is straightforward and only detailed in the appendix (one may also read examples and tests cases for samples). When using the *wide* version of an instruction, one has to use the `wide` keyword before the instruction.

4.6.1 The switch instructions

Almost all instructions are declared on one line but *switch* instructions are multi-line ones. Code sample 5 shows the syntax used by the `tableswitch` and `lookupswitch` instructions. A `tableswitch` instruction accepts three parameters: a label, and lower and upper bounds. The label is the destination for the default case; the following lines define the destination for the lower bound, the lower bound plus one, and so on until the upper bound.
A `lookupswitch` instruction accepts two parameters: a label and a number of matchings. The label is the destination for the default case; the following lines define the matchings in the *value* `=>` *label* form.

Code sample 5 Switch examples.

```
tableswitch default: 0 2
        => zero:
        => one:
        => two:

lookupswitch default: 3
        0 => zero:
        1 => one:
        2 => two:
```

4.6.2 The `invokedynamic` instruction

The `invokedynamic` instruction is also different from the other ones. Although it is written on one single line, it is particular because unlike others it accepts a variable number of arguments.

$$\texttt{invokedynamic}\ bsm\ \ bsa_0 \ldots bsa_n\ \ mr$$

where:

- bsm is a method handle to the bootstrap method;
- bsa_i are arguments to the bootstrap method;
- mr is the method reference to be actually invoked.

4.7 Example

Code sample 6 shows an example of a Barista source file coding a Java™ class named `pack.Test`. This class contains two fields (the constants named `PREFIX` and `SUFFIX`) as well as two methods (`print` and `main`). The `main` method prints the classical hello world message and then iterates over the elements of the passed string array by applying the `print` method to each element. In turn, the `print` method prints each string with the prefix and suffix specified by the `PREFIX` and `SUFFIX` field values.

Code sample 6 Example of a Barista-coded class file.

```
.class public final pack.Test
.extends java.lang.Object

.field private static final java.lang.String PREFIX
        @ConstantValue "  - << "

.field private static final java.lang.String SUFFIX
        @ConstantValue " >>"

.method public static void print(java.lang.String)
        getstatic java.lang.System.out:java.io.PrintStream
        dup
        dup
        getstatic pack.Test.PREFIX: java.lang.String
        invokevirtual java.io.PrintStream.print(java.lang.String): void
        aload_0
        invokevirtual java.io.PrintStream.print(java.lang.String) :void
        getstatic pack.Test.SUFFIX :java.lang.String
        invokevirtual java.io.PrintStream.println(java.lang.String) : void
        return

.method public static void main(java.lang.String[])
        nop
        getstatic java.lang.System.out : java.io.PrintStream
        ldc "hello\t... \n\t... \"world\""
        invokevirtual java.io.PrintStream.println(java.lang.String):void

        iconst_0
        istore_1
        aload_0
        arraylength
        istore_2
loop:
        iload_1
        iload_2
        if_icmpeq end:
        aload_0
        iload_1
        aaload
        invokestatic pack.Test.print(java.lang.String):void
        iinc 1 1
        goto loop:
end:
        return
```

Chapter 5

Using Barista as an Objective Caml library

This chapter sheds light on the way the Barista library is organized, and its main modules. First, it explain how to compile a Barista-based application. Then, the library contents is explored. Finally, a complete example demonstrates a practical use of the library.

5.1 Compilation

In order to use Barista as a library, it is sufficient to link the program with either `baristaLibrary.cma` (bytecode version), `baristaLibrary.cmxa` (native version), or `baristaLibrary.cmja` (Java™-compiled version). These libraries are crafted in such a way that there is only one top-level module called `BaristaLibrary` that contains all the modules described by the `ocamldoc`-generated documentation.

As there are two ways to install Barista (classical installation, and `ocamlfind`-based installation), there are two ways to compile a program which are explained in the following two section.

5.1.1 Under *classical* installation

In order to compile the source files presented in this chapter, one of the following commands should be used (where "`source.ml`" is the program to be compiled):

- `ocamlc -I +barista -I +zip bigarray.cma camomile.cma unix.cma zip.cma str.cma baristaLibrary.cma source.ml`
- `ocamlopt -I +barista -I +zip bigarray.cmxa camomile.cmxa unix.cmxa zip.cmxa str.cmxa baristaLibrary.cmxa source.ml`
- `ocamljava -I +barista -I +zip bigarray.cmja camomile.cmja unix.cmja zip.cmja str.cmja baristaLibrary.cmja source.ml`

5.1.2 Under `ocamlfind` installation

In order to compile the source files presented in this chapter, one of the following commands should be used:

- `ocamlfind ocamlc -package bigarray,unix,str,camomile,zip,barista source.ml`
- `ocamlfind ocamlopt -package bigarray,unix,str,camomile,zip,barista source.ml`
- `ocamlfind ocamljava -package bigarray,unix,str,camomile,zip,barista source.ml`

5.2 Overview of the library

5.2.1 Organization of the source base

Although all modules are packed into one, namely `BaristaLibrary`, the source directory is organized in subsdirectories corresponding to the areas covered by the library.
These directories are:

- `common` that contains various utility modules of general use;
- `helpers` that contains various utility modules for lexing and class loading;
- `classfile` that contains the modules defining class files;
- `analysis` that contains some modules for both analysis and optimization of bytecode;
- `commands`, and `driver` that contain the definition of command-line subcommands and the definition of main program;
- `tools` that contains the actual implementation of command-line subcommands;
- `utf8` that contains a Camomile-based UTF8 implementation.

5.2.2 Description of the main modules

The complete documentation for the library can be generated by issuing `make doc` after a successful compilation; the produced documentation is located in the `ocamldoc` subdirectory. The remainder of this section provides a quick overview of the main modules in the different areas.

The main utility modules in `common` are:

- `Utils` providing Unicode functions (based on the Camomile library);
- `InputStream` providing various implementation of input streams;
- `OutputStream` providing various implementation of output streams;
- `Consts` providing the Unicode constants for the whole project.

The main utility modules in `helpers` are:

- `ClassPath` providing definition for a classpath;
- `ClassLoader` providing functions for loading class, package, and module files.

The main utility modules in `classfile` are:

- `Name` defines the names for the various kinds of Java™ elements;
- `Descriptor` defines the descriptors (*i.e.* type informations) for the various kinds of Java™ elements;
- `Signature` defines the signature (*i.e.* type informations for *generic* types) for the various kinds of Java™ elements;
- `AccessFlag` defines the access flags for the various kinds of Java™ elements;
- `ConstantPool` defines the constant pools used for decoding/encoding of class files.

Additionally, other modules provide types representing the class file elements. Each class file element comes in two flavours: a low-level form (as close as possible to the class file format) and a high-level form (as expressive as can be). Table 5.1 gives the types for these elements.

Element	**Low-level form**	**High-level form**
Annotations	`Annotation.info`	`Annotation.t`
Attributes	`Attribute.into`	`Attribute.t`
Instructions	`ByteCode.t`	`Instruction.t`
Fields	`Field.info`	`Field.t`
Methods	`Method.info`	`Method.t`
Classes	`ClassFile.t`	`ClassDefinition.t`
Packages	`ClassFile.t`	`PackageDefinition.t`
Modules	`ClassFile.t`	`ModuleDefinition.t`

Table 5.1: Mapping of Java™ elements to Barista types.

The main utility modules in `analysis` are:

- `StackState` providing manipulation of locals and operand stacks;
- `ControlFlow` providing construction of the (hyper-)graph representing the control flow of a method;
- `Peephole` providing rewriting rules for *peephole* of instruction list in a control flow graph;
- `Code` providing various functions working on a control flow graph (*e.g.* computation of stack elements, that is `max_locals`, `max_stack`, and stack frames).

The main utility modules in `tools` are:

- `Assembler` providing the function called by the `assemble` subcommand;
- `ClassPrinter` providing the function called by the `print` subcommand;
- `Disassembler` providing the function called by the `disassemble` subcommand;
- `FlowPrinter` providing the function called by the `flow` subcommand.

5.3 Complete example

Code sample 7 shows an example of an Objective Caml program using Barista to produce a file named `Hello.class` containing the definition of a class named `example.Hello`. This class contains only a `main` method that prints “hello.” on the standard output.

Here are the key points of this example, in the order they appear in the source file:

- first, the Barista library and its `Utils` is open (the latter is especially useful to get access to the UTF8-related modules);
- then, some abbreviations are given to functions building UTF8 strings and class, field, and method names;
- from these shorthands, some constants denoting class field and methods are defined;
- `instructions` is bound to what will actually be executed by the `main` method of the class;
- `code` encapsulates `instructions`, as well as stack/local limits and possible exception table and attributes;
- `main_method` is defined as would be in a Java™ source file: flags, name, descriptor (*i.e.* signature), and attributes (here only its code);
- at this point, it is possible to build an actual class definition from flags, name, parent class, parent interfaces, embedded fields, embedded methods, and attributes;
- finally, the class definition is converted into a class file that is written to the file `‘‘Hello.class’’`.

Code sample 7 Using Barista to produce a class file from an Objective Caml program.

```
open BaristaLibrary
open Utils

let utf8 = UTF8.of_string
let utf8_for_class x = Name.make_for_class_from_external (utf8 x)
let utf8_for_field x = Name.make_for_field (utf8 x)
let utf8_for_method x = Name.make_for_method (utf8 x)

let example_Hello = utf8_for_class "example.Hello"
let java_lang_Object = utf8_for_class "java.lang.Object"
let java_lang_System = utf8_for_class "java.lang.System"
let java_lang_String = utf8_for_class "java.lang.String"
let java_io_PrintStream = utf8_for_class "java.io.PrintStream"
let out = utf8_for_field "out"
let println = utf8_for_method "println"
let main = utf8_for_method "main"

let () =
  let instructions = [
    Instruction.GETSTATIC (java_lang_System, out, `Class java_io_PrintStream);
    Instruction.LDC (`String (utf8 "hello."));
    Instruction.INVOKEVIRTUAL (`Class_or_interface java_io_PrintStream,
                               println,
                               ([`Class java_lang_String], `Void));
    Instruction.RETURN;
  ] in
  let code = {
    Attribute.max_stack = u2 2;
    Attribute.max_locals = u2 1;
    Attribute.code = instructions;
    Attribute.exception_table = [];
    Attribute.attributes = [];
  } in
  let main_method =
    Method.Regular { Method.flags = [`Public; `Static];
                     Method.name = main;
                     Method.descriptor = [`Array (`Class java_lang_String)], `Void;
                     Method.attributes = [`Code code] } in
  let hello = {
    ClassDefinition.access_flags = [`Public; `Super];
    ClassDefinition.name = example_Hello;
    ClassDefinition.extends = Some java_lang_Object;
    ClassDefinition.implements = [];
    ClassDefinition.fields = [];
    ClassDefinition.methods = [main_method];
    ClassDefinition.attributes = [];
  } in
  let cf = ClassDefinition.encode hello in
  ClassFile.write cf (OutputStream.make_of_channel (open_out "Hello.class"))
```

Appendix A

Instructions and related parameters

This appendix summarizes, in alphabetical order, the instructions recognized by the Barista assembler. For a complete description, one should refer to the JVM documentation provided by Oracle®.

For various examples of actual uses, one is advised to check either the samples in the `examples` subdirectory, or the sources used as unit tests (they can be found in the `tests` subdirectory of the Barista source distribution).

`aaload` — load reference from array
opcode: `0x32`
no parameter

`aastore` — store into reference array
opcode: `0x53`
no parameter

`aconst_null` — push null
opcode: `0x01`
no parameter

`aload` — load reference from local variable
opcode: `0x19`
parameter #1: local index (unsigned 8-bit integer)

`aload_0` — load reference from local variable
opcode: `0x2A`
no parameter

`aload_1` — load reference from local variable
opcode: `0x2B`
no parameter

`aload_2` — load reference from local variable
opcode: `0x2C`
no parameter

`aload_3` — load reference from local variable
opcode: `0x2D`
no parameter

`anewarray` — create new array of references
opcode: `0xBD`
parameter #1: element type (class name, or array type)

`areturn` — return reference from method
opcode: `0xB0`
no parameter

`arraylength` — get length of array
opcode: `0xBE`
no parameter

`astore` — store reference into local variable
opcode: `0x3A`
parameter #1: local index (unsigned 8-bit integer)

`astore_0` — store reference into local variable
opcode: `0x4B`
no parameter

`astore_1` — store reference into local variable
opcode: `0x4C`
no parameter

`astore_2` — store reference into local variable
opcode: `0x4D`
no parameter

`astore_3` — store reference into local variable
opcode: `0x4E`
no parameter

`athrow` — throw exception or error
opcode: `0xBF`
no parameter

`baload` — load byte or boolean from array
opcode: `0x33`
no parameter

`bastore` — store into byte or boolean array
opcode: `0x54`
no parameter

`bipush` — push byte
opcode: `0x10`
parameter #1: byte value (signed 8-bit integer)

`caload` — load char from array
opcode: `0x34`
no parameter

`castore` — store into char array
opcode: `0x55`
no parameter

`checkcast` — check whether object is of given type
opcode: `0xC0`
parameter #1: type to test against (class name, or array type)

`d2f` — convert double to float
opcode: `0x90`
no parameter

`d2i` — convert double to int
opcode: `0x8E`
no parameter

`d2l` — convert double to long
opcode: `0x8F`
no parameter

`dadd` — add double
opcode: `0x63`
no parameter

`daload` — load double from array
opcode: `0x31`
no parameter

`dastore` — store into double array
opcode: `0x52`
no parameter

`dcmpg` — compare double
opcode: `0x98`
no parameter

`dcmpl` — compare double
opcode: `0x97`
no parameter

`dconst_0` — push double
opcode: `0x0E`
no parameter

`dconst_1` — push double
opcode: `0x0F`
no parameter

`ddiv` — divide double
opcode: `0x6F`
no parameter

`dload` — load double from local variable
opcode: `0x18`
parameter #1: local index (unsigned 8-bit integer)

`dload_0` — load double from local variable
opcode: `0x26`
no parameter

`dload_1` — load double from local variable
opcode: `0x27`
no parameter

`dload_2` — load double from local variable
opcode: `0x28`
no parameter

`dload_3` — load double from local variable
opcode: `0x29`
no parameter

`dmul` — multiply double
opcode: `0x6B`
no parameter

`dneg` — negate double
opcode: `0x77`
no parameter

`drem` — remainder double
opcode: `0x73`
no parameter

`dreturn` — return double from method
opcode: `0xAF`
no parameter

`dstore` — store double into local variable
opcode: `0x39`
parameter #1: local index (unsigned 8-bit integer)

`dstore_0` — store double into local variable
opcode: `0x47`
no parameter

`dstore_1` — store double into local variable
opcode: `0x48`
no parameter

`dstore_2` — store double into local variable
opcode: `0x49`
no parameter

`dstore_3` — store double into local variable
opcode: `0x4A`
no parameter

`dsub` — subtract double
opcode: `0x67`
no parameter

`dup` — duplicate the top operand stack value
opcode: `0x59`
no parameter

`dup2` — duplicate the top one or two operand stack values
opcode: `0x5C`
no parameter

`dup2_x1` — duplicate the top one or two operand stack values and insert two or three values down
opcode: `0x5D`
no parameter

`dup2_x2` — duplicate the top one or two operand stack values and insert two, three, or four values down
opcode: `0x5E`
no parameter

`dup_x1` — duplicate the top operand stack value and insert two values down
opcode: `0x5A`
no parameter

`dup_x2` — duplicate the top operand stack value and insert two or threee values down
opcode: `0x5B`
no parameter

`f2d` — convert float to double
opcode: `0x8D`
no parameter

`f2i` — convert float to int
opcode: `0x8B`
no parameter

`f2l` — convert float to long
opcode: `0x8C`
no parameter

`fadd` — add float
opcode: `0x62`
no parameter

`faload` — load float from array
opcode: `0x30`
no parameter

`fastore` — store into float array
opcode: `0x51`
no parameter

`fcmpg` — compare float
opcode: `0x96`
no parameter

`fcmpl` — compare float
opcode: `0x95`
no parameter

`fconst_0` — push float
opcode: `0x0B`
no parameter

`fconst_1` — push float
opcode: `0x0C`
no parameter

`fconst_2` — push float
opcode: `0x0D`
no parameter

`fdiv` — divide float
opcode: `0x6E`
no parameter

`fload` — load float from local variable
opcode: `0x17`
parameter #1: local index (unsigned 8-bit integer)

`fload_0` — load float from local variable
opcode: `0x22`
no parameter

`fload_1` — load float from local variable
opcode: `0x23`
no parameter

`fload_2` — load float from local variable
opcode: `0x24`
no parameter

`fload_3` — load float from local variable
opcode: `0x25`
no parameter

`fmul` — multiply float
opcode: `0x6A`
no parameter

`fneg` — negate float
opcode: `0x76`
no parameter

`frem` — remainder float
opcode: `0x72`
no parameter

`freturn` — return float from method
opcode: `0xAE`
no parameter

`fstore` — store float into local variable
opcode: `0x38`
parameter #1: local index (unsigned 8-bit integer)

`fstore_0` — store float into local variable
opcode: `0x43`
no parameter

`fstore_1` — store float into local variable
opcode: `0x44`
no parameter

`fstore_2` — store float into local variable
opcode: `0x45`
no parameter

`fstore_3` — store float into local variable
opcode: `0x46`
no parameter

`fsub` — subtract float
opcode: `0x66`
no parameter

`getfield` — fetch field from object
opcode: `0xB4`
parameter #1: field to get (field reference)

`getstatic` — get static field from class
opcode: `0xB2`
parameter #1: field to get (field reference)

`goto` — branch always
opcode: `0xA7`
parameter #1: destination offset (label)

`goto_w` — branch always
opcode: `0xC8`
parameter #1: destination offset (label)

`i2b` — convert int to byte
opcode: `0x91`
no parameter

`i2c` — convert int to char
opcode: `0x92`
no parameter

`i2d` — convert int to double
opcode: `0x87`
no parameter

`i2f` — convert int to float
opcode: `0x86`
no parameter

`i2l` — convert int to long
opcode: `0x85`
no parameter

`i2s` — convert int to short
opcode: `0x93`
no parameter

`iadd` — add int
opcode: `0x60`
no parameter

`iaload` — load int from array
opcode: `0x2E`
no parameter

`iand` — boolean AND int
opcode: `0x7E`
no parameter

`iastore` — store into int array
opcode: `0x4F`
no parameter

`iconst_0` — push int constant
opcode: `0x03`
no parameter

`iconst_1` — push int constant
opcode: `0x04`
no parameter

`iconst_2` — push int constant
opcode: `0x05`
no parameter

`iconst_3` — push int constant
opcode: `0x06`
no parameter

`iconst_4` — push int constant
opcode: `0x07`
no parameter

`iconst_5` — push int constant
opcode: `0x08`
no parameter

`iconst_m1` — push int constant
opcode: `0x02`
no parameter

`idiv` — divide int
opcode: `0x6C`
no parameter

`if_acmpeq` — branch if reference comparison succeeds
opcode: `0xA5`
parameter #1: destination offset (label)

`if_acmpne` — branch if reference comparison succeeds
opcode: `0xA6`
parameter #1: destination offset (label)

`if_icmpeq` — branch if int comparison succeeds
opcode: `0x9F`
parameter #1: destination offset (label)

`if_icmpge` — branch if int comparison succeeds
opcode: `0xA2`
parameter #1: destination offset (label)

`if_icmpgt` — branch if int comparison succeeds
opcode: `0xA3`
parameter #1: destination offset (label)

`if_icmple` — branch if int comparison succeeds
opcode: `0xA4`
parameter #1: destination offset (label)

`if_icmplt` — branch if int comparison succeeds
opcode: `0xA1`
parameter #1: destination offset (label)

`if_icmpne` — branch if int comparison succeeds
opcode: `0xA0`
parameter #1: destination offset (label)

`ifeq` — branch if int comparison with zero succeeds
opcode: `0x99`
parameter #1: destination offset (label)

`ifge` — branch if int comparison with zero succeeds
opcode: `0x9C`
parameter #1: destination offset (label)

`ifgt` — branch if int comparison with zero succeeds
opcode: `0x9D`
parameter #1: destination offset (label)

`ifle` — branch if int comparison with zero succeeds
opcode: `0x9E`
parameter #1: destination offset (label)

`iflt` — branch if int comparison with zero succeeds
opcode: `0x9B`
parameter #1: destination offset (label)

`ifne` — branch if int comparison with zero succeeds
opcode: `0x9A`
parameter #1: destination offset (label)

`ifnonnull` — branch if reference not null
opcode: `0xC7`
parameter #1: destination offset (label)

`ifnull` — branch if reference not null
opcode: `0xC6`
parameter #1: destination offset (label)

`iinc` — increment local variable by constant
opcode: `0x84`
parameter #1: local index (unsigned 8-bit integer)
parameter #2: increment value (signed 8-bit integer)

`iload` — load int from local variable
opcode: `0x15`
parameter #1: local index (unsigned 8-bit integer)

`iload_0` — load int from local variable
opcode: `0x1A`
no parameter

`iload_1` — load int from local variable
opcode: `0x1B`
no parameter

`iload_2` — load int from local variable
opcode: `0x1C`
no parameter

`iload_3` — load int from local variable
opcode: `0x1D`
no parameter

`imul` — multiply int
opcode: `0x68`
no parameter

`ineg` — negate int
opcode: `0x74`
no parameter

`instanceof` — determine if object is of given type
opcode: `0xC1`
parameter #1: type to test against (class name, or array type)

`invokedynamic` — invoke instance method; resolve and dispatch based on class
opcode: `0xBA`
parameter #1: method to invoke (dynamic method reference)

`invokeinterface` — invoke interface method
opcode: `0xB9`
parameter #1: method to invoke (method reference)
parameter #2: parameter count (unsigned 8-bit integer)

`invokespecial` — invoke instance method; special handling for superclass, private, and instance initialization method invocations
opcode: `0xB7`
parameter #1: method to invoke (method reference)

`invokestatic` — invoke a class (static) method
opcode: `0xB8`
parameter #1: method to invoke (method reference)

`invokevirtual` — invoke instance method; dispatch based on class
opcode: `0xB6`
parameter #1: method to invoke (method reference)

`ior` — boolean OR int
opcode: `0x80`
no parameter

`irem` — remainder int
opcode: `0x70`
no parameter

`ireturn` — return int from method
opcode: `0xAC`
no parameter

`ishl` — shift left int
opcode: `0x78`
no parameter

`ishr` — arithmetic shift right int
opcode: `0x7A`
no parameter

`istore` — store int into local variable
opcode: `0x36`
parameter #1: local index (unsigned 8-bit integer)

`istore_0` — store int into local variable
opcode: `0x3B`
no parameter

`istore_1` — store int into local variable
opcode: `0x3C`
no parameter

`istore_2` — store int into local variable
opcode: `0x3D`
no parameter

`istore_3` — store int into local variable
opcode: `0x3E`
no parameter

`isub` — subtract int
opcode: `0x64`
no parameter

`iushr` — logical shift right int
opcode: `0x7C`
no parameter

`ixor` — boolean XOR int
opcode: `0x82`
no parameter

`jsr` — jump subroutine
opcode: `0xA8`
parameter #1: destination offset (label)

`jsr_w` — jump subroutine (wide index)
opcode: `0xC9`
parameter #1: destination offset (label)

`l2d` — convert long to double
opcode: `0x8A`
no parameter

`l2f` — convert long to float
opcode: `0x89`
no parameter

`l2i` — convert long to int
opcode: `0x88`
no parameter

`ladd` — add long
opcode: `0x61`
no parameter

`laload` — load long from array
opcode: `0x2F`
no parameter

`land` — boolean AND long
opcode: `0x7F`
no parameter

`lastore` — store into long array
opcode: `0x50`
no parameter

`lcmp` — compare long
opcode: `0x94`
no parameter

`lconst_0` — push long constant
opcode: `0x09`
no parameter

`lconst_1` — push long constant
opcode: `0x0A`
no parameter

`ldc` — push item from runtime constant pool
opcode: `0x12`
parameter #1: value to be pushed (signed 32-bit integer, or float, or string literal, or class name, or array type, or method type constant, or method handle constant)

`ldc2_w` — push long or double from runtime constant pool (wide index)
opcode: `0x14`
parameter #1: value to be pushed (signed 64-bit integer, or float)

`ldc_w` — push item from runtime constant pool (wide index)
opcode: `0x13`
parameter #1: value to be pushed (signed 32-bit integer, or float, or string literal, or class name, or array type, or method type constant, or method handle constant)

`ldiv` — divide long
opcode: `0x6D`
no parameter

`lload` — load long from local variable
opcode: `0x16`
parameter #1: local index (unsigned 8-bit integer)

`lload_0` — load long from local variable
opcode: `0x1E`
no parameter

`lload_1` — load long from local variable
opcode: `0x1F`
no parameter

`lload_2` — load long from local variable
opcode: `0x20`
no parameter

`lload_3` — load long from local variable
opcode: `0x21`
no parameter

`lmul` — multiply long
opcode: `0x69`
no parameter

`lneg` — negate long
opcode: `0x75`
no parameter

`lookupswitch` — access jump table by key match and jump
opcode: `0xAB`
parameter #1: default destination offset (label)
parameter #2: number of key, offset pairs (signed 32-bit integer)
parameter #3: key, offset pairs (list of (match, label) pairs)

`lor` — boolean OR long
opcode: `0x81`
no parameter

`lrem` — remainder long
opcode: `0x71`
no parameter

`lreturn` — return long from method
opcode: `0xAD`
no parameter

`lshl` — shift left long
opcode: `0x79`
no parameter

`lshr` — arithmetic shift right long
opcode: `0x7B`
no parameter

`lstore` — store long into local variable
opcode: `0x37`
parameter #1: local index (unsigned 8-bit integer)

`lstore_0` — store long into local variable
opcode: `0x3F`
no parameter

`lstore_1` — store long into local variable
opcode: `0x40`
no parameter

`lstore_2` — store long into local variable
opcode: `0x41`
no parameter

`lstore_3` — store long into local variable
opcode: `0x42`
no parameter

`lsub` — subtract long
opcode: `0x65`
no parameter

`lushr` — logical shift right long
opcode: `0x7D`
no parameter

`lxor` — boolean XOR long
opcode: `0x83`
no parameter

`monitorenter` — enter monitor for object
opcode: `0xC2`
no parameter

`monitorexit` — exit monitor for object
opcode: `0xC3`
no parameter

`multianewarray` — create new multidimensional array
opcode: `0xC5`
parameter #1: element type (class name, or array type)
parameter #2: number of dimensions (unsigned 8-bit integer)

`new` — create new object
opcode: `0xBB`
parameter #1: type to create (class name)

`newarray` — create new arrayhandler
opcode: `0xBC`
parameter #1: element type (primitive array type)

`nop` — do nothing
opcode: `0x00`
no parameter

`pop` — pop the top operand stack value
opcode: `0x57`
no parameter

`pop2` — pop the top one or two operand stack values
opcode: `0x58`
no parameter

`putfield` — set field in object
opcode: `0xB5`
parameter #1: field to set (field reference)

`putstatic` — set static field in class
opcode: `0xB3`
parameter #1: field to set (field reference)

`ret` — return from subroutine
opcode: `0xA9`
parameter #1: local index (unsigned 8-bit integer)

`return` — return void from method
opcode: `0xB1`
no parameter

`saload` — load short from array
opcode: `0x35`
no parameter

`sastore` — store into short array
opcode: `0x56`
no parameter

`sipush` — push short
opcode: `0x11`
parameter #1: short value (signed 16-bit integer)

`swap` — swap the top two operand stack values
opcode: `0x5F`
no parameter

`tableswitch` — access jump table by index and jump
opcode: `0xAA`
parameter #1: default destination offset (label)
parameter #2: lower bound (signed 32-bit integer)
parameter #3: higher bound (signed 32-bit integer)
parameter #4: destination offsets (list of labels)

`wide aload` — load reference from local variable
opcode: `0xC4` followed by `0x19`
parameter #1: local index (unsigned 16-bit integer)

`wide astore` — store reference into local variable
opcode: `0xC4` followed by `0x3A`
parameter #1: local index (unsigned 16-bit integer)

`wide dload` — load double from local variable
opcode: `0xC4` followed by `0x18`
parameter #1: local index (unsigned 16-bit integer)

`wide dstore` — store double into local variable
opcode: `0xC4` followed by `0x39`
parameter #1: local index (unsigned 16-bit integer)

`wide fload` — load float from local variable
opcode: `0xC4` followed by `0x17`
parameter #1: local index (unsigned 16-bit integer)

`wide fstore` — store float into local variable
opcode: `0xC4` followed by `0x38`
parameter #1: local index (unsigned 16-bit integer)

`wide iinc` — increment local variable by constant
opcode: `0xC4` followed by `0x84`
parameter #1: local index (unsigned 16-bit integer)
parameter #2: increment value (signed 16-bit integer)

`wide iload` — load int from local variable
opcode: `0xC4` followed by `0x15`
parameter #1: local index (unsigned 16-bit integer)

`wide istore` — store int into local variable
opcode: `0xC4` followed by `0x36`
parameter #1: local index (unsigned 16-bit integer)

`wide lload` — load long from local variable
opcode: `0xC4` followed by `0x16`
parameter #1: local index (unsigned 16-bit integer)

`wide lstore` — store long into local variable
opcode: `0xC4` followed by `0x37`
parameter #1: local index (unsigned 16-bit integer)

`wide ret` — return from subroutine
opcode: `0xC4` followed by `0xA9`
parameter #1: local index (unsigned 16-bit integer)

Appendix B

Instructions by categories

This appendix lists the instructions grouped by categories, allowing easy search for a given operation. Categories overlaps, as a given instruction belongs to several categories; example: `fconst_0` is both related to constants and to floats.

Hyperlinks allow navigation to the previous appendix.

Array creations and accesses

- `aaload` : load reference from array
- `aastore` : store into reference array
- `anewarray` : create new array of references
- `arraylength` : get length of array
- `baload` : load byte or boolean from array
- `bastore` : store into byte or boolean array
- `caload` : load char from array
- `castore` : store into char array
- `daload` : load double from array
- `dastore` : store into double array
- `faload` : load float from array
- `fastore` : store into float array
- `iaload` : load int from array
- `iastore` : store into int array
- `laload` : load long from array
- `lastore` : store into long array
- `multianewarray` : create new multidimensional array
- `newarray` : create new arrayhandler
- `saload` : load short from array
- `sastore` : store into short array

Arithmetic operations

- dadd : add double
- ddiv : divide double
- dmul : multiply double
- dneg : negate double
- drem : remainder double
- dsub : subtract double
- fadd : add float
- fdiv : divide float
- fmul : multiply float
- fneg : negate float
- frem : remainder float
- fsub : subtract float
- iadd : add int
- idiv : divide int
- iinc : increment local variable by constant
- imul : multiply int
- ineg : negate int
- irem : remainder int
- isub : subtract int
- ladd : add long
- ldiv : divide long
- lmul : multiply long
- lneg : negate long
- lrem : remainder long
- lsub : subtract long
- wide iinc : increment local variable by constant

Comparison operations

- dcmpg : compare double
- dcmpl : compare double
- fcmpg : compare float
- fcmpl : compare float
- if_acmpeq : branch if reference comparison succeeds
- if_acmpne : branch if reference comparison succeeds
- if_icmpeq : branch if int comparison succeeds

- `if_icmpge` : branch if int comparison succeeds
- `if_icmpgt` : branch if int comparison succeeds
- `if_icmple` : branch if int comparison succeeds
- `if_icmplt` : branch if int comparison succeeds
- `if_icmpne` : branch if int comparison succeeds
- `ifeq` : branch if int comparison with zero succeeds
- `ifge` : branch if int comparison with zero succeeds
- `ifgt` : branch if int comparison with zero succeeds
- `ifle` : branch if int comparison with zero succeeds
- `iflt` : branch if int comparison with zero succeeds
- `ifne` : branch if int comparison with zero succeeds
- `ifnonnull` : branch if reference not null
- `ifnull` : branch if reference not null
- `lcmp` : compare long
- `lookupswitch` : access jump table by key match and jump

Control flow instructions

- `goto` : branch always
- `goto_w` : branch always
- `if_acmpeq` : branch if reference comparison succeeds
- `if_acmpne` : branch if reference comparison succeeds
- `if_icmpeq` : branch if int comparison succeeds
- `if_icmpge` : branch if int comparison succeeds
- `if_icmpgt` : branch if int comparison succeeds
- `if_icmple` : branch if int comparison succeeds
- `if_icmplt` : branch if int comparison succeeds
- `if_icmpne` : branch if int comparison succeeds
- `ifeq` : branch if int comparison with zero succeeds
- `ifge` : branch if int comparison with zero succeeds
- `ifgt` : branch if int comparison with zero succeeds
- `ifle` : branch if int comparison with zero succeeds
- `iflt` : branch if int comparison with zero succeeds
- `ifne` : branch if int comparison with zero succeeds
- `ifnonnull` : branch if reference not null
- `ifnull` : branch if reference not null
- `jsr` : jump subroutine
- `jsr_w` : jump subroutine (wide index)

- `lookupswitch` : access jump table by key match and jump
- `ret` : return from subroutine
- `tableswitch` : access jump table by index and jump
- `wide ret` : return from subroutine

Conversions

- `checkcast` : check whether object is of given type
- `d2f` : convert double to float
- `d2i` : convert double to int
- `d2l` : convert double to long
- `f2d` : convert float to double
- `f2i` : convert float to int
- `f2l` : convert float to long
- `i2b` : convert int to byte
- `i2c` : convert int to char
- `i2d` : convert int to double
- `i2f` : convert int to float
- `i2l` : convert int to long
- `i2s` : convert int to short
- `l2d` : convert long to double
- `l2f` : convert long to float
- `l2i` : convert long to int

Exception-related instructions

- `athrow` : throw exception or error

Field-related instructions

- `getfield` : fetch field from object
- `getstatic` : get static field from class
- `putfield` : set field in object
- `putstatic` : set static field in class

Constants

- bipush : push byte
- dconst_0 : push double
- dconst_1 : push double
- fconst_0 : push float
- fconst_1 : push float
- fconst_2 : push float
- iconst_0 : push int constant
- iconst_1 : push int constant
- iconst_2 : push int constant
- iconst_3 : push int constant
- iconst_4 : push int constant
- iconst_5 : push int constant
- iconst_m1 : push int constant
- lconst_0 : push long constant
- lconst_1 : push long constant
- sipush : push short

Access to local variables

- aload : load reference from local variable
- aload_0 : load reference from local variable
- aload_1 : load reference from local variable
- aload_2 : load reference from local variable
- aload_3 : load reference from local variable
- astore : store reference into local variable
- astore_0 : store reference into local variable
- astore_1 : store reference into local variable
- astore_2 : store reference into local variable
- astore_3 : store reference into local variable
- dload : load double from local variable
- dload_0 : load double from local variable
- dload_1 : load double from local variable
- dload_2 : load double from local variable
- dload_3 : load double from local variable
- dstore : store double into local variable
- dstore_0 : store double into local variable

- dstore_1 : store double into local variable
- dstore_2 : store double into local variable
- dstore_3 : store double into local variable
- fload : load float from local variable
- fload_0 : load float from local variable
- fload_1 : load float from local variable
- fload_2 : load float from local variable
- fload_3 : load float from local variable
- fstore : store float into local variable
- fstore_0 : store float into local variable
- fstore_1 : store float into local variable
- fstore_2 : store float into local variable
- fstore_3 : store float into local variable
- iinc : increment local variable by constant
- iload : load int from local variable
- iload_0 : load int from local variable
- iload_1 : load int from local variable
- iload_2 : load int from local variable
- iload_3 : load int from local variable
- istore : store int into local variable
- istore_0 : store int into local variable
- istore_1 : store int into local variable
- istore_2 : store int into local variable
- istore_3 : store int into local variable
- lload : load long from local variable
- lload_0 : load long from local variable
- lload_1 : load long from local variable
- lload_2 : load long from local variable
- lload_3 : load long from local variable
- lstore : store long into local variable
- lstore_0 : store long into local variable
- lstore_1 : store long into local variable
- lstore_2 : store long into local variable
- lstore_3 : store long into local variable
- wide aload : load reference from local variable
- wide astore : store reference into local variable
- wide dload : load double from local variable

- `wide dstore :` store double into local variable
- `wide fload :` load float from local variable
- `wide fstore :` store float into local variable
- `wide iinc :` increment local variable by constant
- `wide iload :` load int from local variable
- `wide istore :` store int into local variable
- `wide lload :` load long from local variable
- `wide lstore :` store long into local variable

Logical operations

- `iand :` boolean AND int
- `ior :` boolean OR int
- `ishl :` shift left int
- `ishr :` arithmetic shift right int
- `iushr :` logical shift right int
- `ixor :` boolean XOR int
- `land :` boolean AND long
- `lor :` boolean OR long
- `lshl :` shift left long
- `lshr :` arithmetic shift right long
- `lushr :` logical shift right long
- `lxor :` boolean XOR long

Method calls

- `invokedynamic :` invoke instance method; resolve and dispatch based on class
- `invokeinterface :` invoke interface method
- `invokespecial :` invoke instance method; special handling for superclass, private, and instance initialization method invocations
- `invokestatic :` invoke a class (static) method
- `invokevirtual :` invoke instance method; dispatch based on class

Instructions over monitors

- `monitorenter :` enter monitor for object
- `monitorexit :` exit monitor for object

Method returns

- `areturn` : return reference from method
- `dreturn` : return double from method
- `freturn` : return float from method
- `ireturn` : return int from method
- `lreturn` : return long from method
- `return` : return void from method

Stack manipulation

- `dup` : duplicate the top operand stack value
- `dup2` : duplicate the top one or two operand stack values
- `dup2_x1` : duplicate the top one or two operand stack values and insert two or three values down
- `dup2_x2` : duplicate the top one or two operand stack values and insert two, three, or four values down
- `dup_x1` : duplicate the top operand stack value and insert two values down
- `dup_x2` : duplicate the top operand stack value and insert two or threee values down
- `ldc` : push item from runtime constant pool
- `ldc2_w` : push long or double from runtime constant pool (wide index)
- `ldc_w` : push item from runtime constant pool (wide index)
- `pop` : pop the top operand stack value
- `pop2` : pop the top one or two operand stack values
- `swap` : swap the top two operand stack values

Operations over `booleans`

- `baload` : load byte or boolean from array
- `bastore` : store into byte or boolean array

Operations over `bytes`

- `baload` : load byte or boolean from array
- `bastore` : store into byte or boolean array
- `bipush` : push byte
- `i2b` : convert int to byte

Operations over `chars`

- `caload` : load char from array
- `castore` : store into char array
- `i2c` : convert int to char

Operations over `doubles`

- `d2f` : convert double to float
- `d2i` : convert double to int
- `d2l` : convert double to long
- `dadd` : add double
- `daload` : load double from array
- `dastore` : store into double array
- `dcmpg` : compare double
- `dcmpl` : compare double
- `dconst_0` : push double
- `dconst_1` : push double
- `ddiv` : divide double
- `dload` : load double from local variable
- `dload_0` : load double from local variable
- `dload_1` : load double from local variable
- `dload_2` : load double from local variable
- `dload_3` : load double from local variable
- `dmul` : multiply double
- `dneg` : negate double
- `drem` : remainder double
- `dreturn` : return double from method
- `dstore` : store double into local variable
- `dstore_0` : store double into local variable
- `dstore_1` : store double into local variable
- `dstore_2` : store double into local variable
- `dstore_3` : store double into local variable
- `dsub` : subtract double
- `f2d` : convert float to double
- `i2d` : convert int to double
- `l2d` : convert long to double
- `wide dload` : load double from local variable
- `wide dstore` : store double into local variable

Operations over `floats`

- `d2f` : convert double to float
- `f2d` : convert float to double
- `f2i` : convert float to int
- `f2l` : convert float to long
- `fadd` : add float
- `faload` : load float from array
- `fastore` : store into float array
- `fcmpg` : compare float
- `fcmpl` : compare float
- `fconst_0` : push float
- `fconst_1` : push float
- `fconst_2` : push float
- `fdiv` : divide float
- `fload` : load float from local variable
- `fload_0` : load float from local variable
- `fload_1` : load float from local variable
- `fload_2` : load float from local variable
- `fload_3` : load float from local variable
- `fmul` : multiply float
- `fneg` : negate float
- `frem` : remainder float
- `freturn` : return float from method
- `fstore` : store float into local variable
- `fstore_0` : store float into local variable
- `fstore_1` : store float into local variable
- `fstore_2` : store float into local variable
- `fstore_3` : store float into local variable
- `fsub` : subtract float
- `i2f` : convert int to float
- `l2f` : convert long to float
- `wide fload` : load float from local variable
- `wide fstore` : store float into local variable

Operations over `ints`

- `d2i` : convert double to int
- `f2i` : convert float to int
- `i2b` : convert int to byte
- `i2c` : convert int to char
- `i2d` : convert int to double
- `i2f` : convert int to float
- `i2l` : convert int to long
- `i2s` : convert int to short
- `iadd` : add int
- `iaload` : load int from array
- `iand` : boolean AND int
- `iastore` : store into int array
- `iconst_0` : push int constant
- `iconst_1` : push int constant
- `iconst_2` : push int constant
- `iconst_3` : push int constant
- `iconst_4` : push int constant
- `iconst_5` : push int constant
- `iconst_m1` : push int constant
- `idiv` : divide int
- `if_icmpeq` : branch if int comparison succeeds
- `if_icmpge` : branch if int comparison succeeds
- `if_icmpgt` : branch if int comparison succeeds
- `if_icmple` : branch if int comparison succeeds
- `if_icmplt` : branch if int comparison succeeds
- `if_icmpne` : branch if int comparison succeeds
- `ifeq` : branch if int comparison with zero succeeds
- `ifge` : branch if int comparison with zero succeeds
- `ifgt` : branch if int comparison with zero succeeds
- `ifle` : branch if int comparison with zero succeeds
- `iflt` : branch if int comparison with zero succeeds
- `ifne` : branch if int comparison with zero succeeds
- `iinc` : increment local variable by constant
- `iload` : load int from local variable
- `iload_0` : load int from local variable
- `iload_1` : load int from local variable

- `iload_2` : load int from local variable
- `iload_3` : load int from local variable
- `imul` : multiply int
- `ineg` : negate int
- `ior` : boolean OR int
- `irem` : remainder int
- `ireturn` : return int from method
- `ishl` : shift left int
- `ishr` : arithmetic shift right int
- `istore` : store int into local variable
- `istore_0` : store int into local variable
- `istore_1` : store int into local variable
- `istore_2` : store int into local variable
- `istore_3` : store int into local variable
- `isub` : subtract int
- `iushr` : logical shift right int
- `ixor` : boolean XOR int
- `l2i` : convert long to int
- `lookupswitch` : access jump table by key match and jump
- `tableswitch` : access jump table by index and jump
- `wide iinc` : increment local variable by constant
- `wide iload` : load int from local variable
- `wide istore` : store int into local variable

Operations over `longs`

- `d2l` : convert double to long
- `f2l` : convert float to long
- `i2l` : convert int to long
- `l2d` : convert long to double
- `l2f` : convert long to float
- `l2i` : convert long to int
- `ladd` : add long
- `laload` : load long from array
- `land` : boolean AND long
- `lastore` : store into long array
- `lcmp` : compare long
- `lconst_0` : push long constant

- `lconst_1` : push long constant
- `ldiv` : divide long
- `lload` : load long from local variable
- `lload_0` : load long from local variable
- `lload_1` : load long from local variable
- `lload_2` : load long from local variable
- `lload_3` : load long from local variable
- `lmul` : multiply long
- `lneg` : negate long
- `lor` : boolean OR long
- `lrem` : remainder long
- `lreturn` : return long from method
- `lshl` : shift left long
- `lshr` : arithmetic shift right long
- `lstore` : store long into local variable
- `lstore_0` : store long into local variable
- `lstore_1` : store long into local variable
- `lstore_2` : store long into local variable
- `lstore_3` : store long into local variable
- `lsub` : subtract long
- `lushr` : logical shift right long
- `lxor` : boolean XOR long
- `wide lload` : load long from local variable
- `wide lstore` : store long into local variable

Operations over references

- `aaload` : load reference from array
- `aastore` : store into reference array
- `aconst_null` : push null
- `aload` : load reference from local variable
- `aload_0` : load reference from local variable
- `aload_1` : load reference from local variable
- `aload_2` : load reference from local variable
- `aload_3` : load reference from local variable
- `anewarray` : create new array of references
- `areturn` : return reference from method
- `astore` : store reference into local variable

- `astore_0` : store reference into local variable
- `astore_1` : store reference into local variable
- `astore_2` : store reference into local variable
- `astore_3` : store reference into local variable
- `checkcast` : check whether object is of given type
- `if_acmpeq` : branch if reference comparison succeeds
- `if_acmpne` : branch if reference comparison succeeds
- `ifnonnull` : branch if reference not null
- `ifnull` : branch if reference not null
- `instanceof` : determine if object is of given type
- `monitorenter` : enter monitor for object
- `monitorexit` : exit monitor for object
- `multianewarray` : create new multidimensional array
- `new` : create new object
- `wide aload` : load reference from local variable
- `wide astore` : store reference into local variable

Operations over `shorts`

- `i2s` : convert int to short
- `saload` : load short from array
- `sastore` : store into short array
- `sipush` : push short

Appendix C

Instructions by opcode

This appendix lists the instructions sorted by opcode.

Hyperlinks allow navigation to the first appendix.

0x00 : nop
0x01 : aconst_null
0x02 : iconst_m1
0x03 : iconst_0
0x04 : iconst_1
0x05 : iconst_2
0x06 : iconst_3
0x07 : iconst_4
0x08 : iconst_5
0x09 : lconst_0
0x0A : lconst_1
0x0B : fconst_0
0x0C : fconst_1
0x0D : fconst_2
0x0E : dconst_0
0x0F : dconst_1
0x10 : bipush
0x11 : sipush
0x12 : ldc
0x13 : ldc_w
0x14 : ldc2_w
0x15 : iload
0x16 : lload
0x17 : fload
0x18 : dload
0x19 : aload
0x1A : iload_0
0x1B : iload_1
0x1C : iload_2
0x1D : iload_3
0x1E : lload_0
0x1F : lload_1
0x20 : lload_2
0x21 : lload_3
0x22 : fload_0
0x23 : fload_1
0x24 : fload_2
0x25 : fload_3
0x26 : dload_0
0x27 : dload_1
0x28 : dload_2
0x29 : dload_3
0x2A : aload_0
0x2B : aload_1
0x2C : aload_2
0x2D : aload_3
0x2E : iaload
0x2F : laload
0x30 : faload
0x31 : daload
0x32 : aaload
0x33 : baload
0x34 : caload
0x35 : saload
0x36 : istore
0x37 : lstore
0x38 : fstore
0x39 : dstore
0x3A : astore
0x3B : istore_0
0x3C : istore_1
0x3D : istore_2
0x3E : istore_3
0x3F : lstore_0
0x40 : lstore_1
0x41 : lstore_2
0x42 : lstore_3
0x43 : fstore_0
0x44 : fstore_1
0x45 : fstore_2
0x46 : fstore_3
0x47 : dstore_0
0x48 : dstore_1
0x49 : dstore_2
0x4A : dstore_3
0x4B : astore_0
0x4C : astore_1
0x4D : astore_2
0x4E : astore_3
0x4F : iastore
0x50 : lastore
0x51 : fastore
0x52 : dastore
0x53 : aastore
0x54 : bastore
0x55 : castore
0x56 : sastore
0x57 : pop
0x58 : pop2
0x59 : dup
0x5A : dup_x1
0x5B : dup_x2
0x5C : dup2
0x5D : dup2_x1
0x5E : dup2_x2
0x5F : swap
0x60 : iadd
0x61 : ladd
0x62 : fadd
0x63 : dadd
0x64 : isub
0x65 : lsub

0x66 : fsub
0x67 : dsub
0x68 : imul
0x69 : lmul
0x6A : fmul
0x6B : dmul
0x6C : idiv
0x6D : ldiv
0x6E : fdiv
0x6F : ddiv
0x70 : irem
0x71 : lrem
0x72 : frem
0x73 : drem
0x74 : ineg
0x75 : lneg
0x76 : fneg
0x77 : dneg
0x78 : ishl
0x79 : lshl
0x7A : ishr
0x7B : lshr
0x7C : iushr
0x7D : lushr
0x7E : iand
0x7F : land
0x80 : ior
0x81 : lor
0x82 : ixor
0x83 : lxor
0x84 : iinc
0x85 : i2l
0x86 : i2f
0x87 : i2d
0x88 : l2i
0x89 : l2f
0x8A : l2d
0x8B : f2i
0x8C : f2l
0x8D : f2d
0x8E : d2i
0x8F : d2l
0x90 : d2f
0x91 : i2b
0x92 : i2c
0x93 : i2s
0x94 : lcmp
0x95 : fcmpl
0x96 : fcmpg
0x97 : dcmpl
0x98 : dcmpg
0x99 : ifeq
0x9A : ifne
0x9B : iflt
0x9C : ifge
0x9D : ifgt
0x9E : ifle
0x9F : if_icmpeq
0xA0 : if_icmpne
0xA1 : if_icmplt
0xA2 : if_icmpge
0xA3 : if_icmpgt
0xA4 : if_icmple
0xA5 : if_acmpeq
0xA6 : if_acmpne
0xA7 : goto
0xA8 : jsr
0xA9 : ret
0xAA : tableswitch
0xAB : lookupswitch
0xAC : ireturn
0xAD : lreturn
0xAE : freturn
0xAF : dreturn
0xB0 : areturn
0xB1 : return
0xB2 : getstatic
0xB3 : putstatic
0xB4 : getfield
0xB5 : putfield
0xB6 : invokevirtual
0xB7 : invokespecial
0xB8 : invokestatic
0xB9 : invokeinterface
0xBA : invokedynamic
0xBB : new
0xBC : newarray
0xBD : anewarray
0xBE : arraylength
0xBF : athrow
0xC0 : checkcast
0xC1 : instanceof
0xC2 : monitorenter
0xC3 : monitorexit
0xC5 : multianewarray
0xC6 : ifnull
0xC7 : ifnonnull
0xC8 : goto_w
0xC9 : jsr_w

0xC4 0x15 : wide iload
0xC4 0x16 : wide lload
0xC4 0x17 : wide fload
0xC4 0x18 : wide dload
0xC4 0x19 : wide aload
0xC4 0x36 : wide istore
0xC4 0x37 : wide lstore
0xC4 0x38 : wide fstore
0xC4 0x39 : wide dstore
0xC4 0x3A : wide astore
0xC4 0x84 : wide iinc
0xC4 0xA9 : wide ret