The Bytelang Programming Language Specification

Version 2.0.0

Tomáš Zíma

2014

Number of revision	1
Last update	11-27-2014

The Bytelang programming language was created by Tomáš Zíma (himself) for ~~the fun~~ the sake of science. It is a personal project of the single person, not a project developed by the Oracle Corporation. Keep this in mind in the case you find some bugs (~~what is a matter of time, by the way~~). I am responsible for that, not Oracle. It is not an official project.

Requirements on the reader

This text assumes that the reader has very stong knowledges of the Java programming language, Java bytecode and basic knowledges of the assembly language.

This document is intended to be a handbook with the short and expressive language specification, not a complete Java bytecode guide.

Contents

Chapter 1

Introducing the Bytelang

Bytelang is a low-level programming language for the Java platform. It allows programmers to precisely describe the Class file and all its parts, including, but not limited to, methods' bytecode, attributes and constant pool.

This is particularly handy for such purposes as a testing of JVM, developing the Java bytecode manipulation tools or libraries, using and implementing bytecode instrumentation or developing decompilers.

There is not another sufficient and comfortable way to generate a particular bytecode. The existing possibilities include writing a Class file by hand in a hexadecimal editor, generating it through bytecode instrumentation libraries, such as ASM or Javassit, or modifying existing Class files, but none of these solutions is efficient enough.

Bytelang is a conventional programming language which is effectively combining syntax of the Java programming language and assembly language. This unique combination makes a source code in Bytelang fairly readable and understandable for everyone who has basic knowledges of the Java bytecode.

Since, for example, testing of the JVM may require a developer to deploy and execute an invalid bytecode, Bytelang allows to write and compile such code. At the same time, it allows a developer to prohibit automatic code generation, therefore giving him a chance to take a full control over the generated code.

Chapter 2

Source code file

2.1 File name mask

Name of each Bytelang source code file should match the mask `*.jsm`. It is a common choice to use the mask `*.asm` for assembly language source files and since Bytelang could be also called *Assembly for Java*, letter *a* has been replaced with *j* for Java.

Although it is highly impreffered, using of a different file name mask is also possible and shall not be considered as a violation of this standard.

2.2 Comments

Comments are parts of a source code which are ignored by the compiler and are only useful for programmers to better understand program's behavior.

Comments are allowed to be present at any place of a source code except the string literals and identifiers.

2.2.1 Line comments

Line comment consists of two forward slashes, one immediately following the other. All characters following these two slashes up to the end of the line will be ignored.

Listing 2.1: Using line comments

```
// A fairly complicated class for adding of two
// numbers together!
public class Sum {
        private String foo = "WTF";
        // Calculates a sum of two numbers.
        @set(maxLocals = 5)
        public static int sum(int, int) {
                iload_0 // load a first argument
                iload_1 // load a second argument
                iadd    // calculate a sum
                ireturn // return result!
        }
}
```

2.2.2 Multiline comments

Multiline comment is delimited by a forward slash followed by an asterisk and an asterisk followed by a forward slash. The end delimiter might be located on the same line or any of the following lines. Everything between these two delimiters (included) is ignored by the compiler.

Listing 2.2: Using line comments

```
/*
 * A fairly complicated class for adding of two
 * numbers together.
 */
public class Sum {
        /* Calculates a sum of two numbers. */
        @set(maxLocals = 5)
```

```
        public static int sum(int /*a*/, int /*b*/) {
                iload_0 // load a first argument
                iload_1 // load a second argument
                iadd    // calculate a sum
                ireturn // return result!
        }
}
```

2.3 Encoding

Each source code file must be encoded using `ASCII` coding standard. Using of different coding standards is not officialy supported in the current language version.

Although using of `UTF8` characters in comments usually works, programmer should not rely on it.

2.4 Splitting a source code file

Splitting of a a source code file is not supported by the Bytelang programming language. Each source code file describes exactly one class and all its code must be located in exactly one file.

Chapter 3

Describing a Class file

The Bytelang programming language is dedicated for describing of Java platform Class files as defined by the JVM specification ([JVM]). This chapter provides basic informations about how this goal is achieved in Bytelang. Further chapters provides more detailed informations about the particular subjects.

3.1 Defining a class

Class definition matches basically the same rules as the Java programming language. The general format is shown in the figure 3.1.

Listing 3.1: General class definition rules

```
[accessFlag][propertyModifiers]<classType><className>
[implements <interface1>[, <interface2> [...]]][extends
<parent1>[, <parent2>]] {
      // class body
}
```

In the listing 3.1, mandatory parts are delimited by angle brackets while non-mandatory parts are delimited by square brackets. The only possible access flag is `public`, all possible property modifiers are shown in the table 3.1 and all possible class types are shown in the table 3.2.

`final`	`super`	`abstract`	`synthetic`

Table 3.1: All possible class modifiers

`class`	`interface`	`annotation`	`enum`

Table 3.2: All possible class types

Meaning of all of these flags is explained in the [JVM]. The only exception is a keyword `class`, which does not have any value or meaning. Since you are defining a class, it is obvious that result will be some class. In special cases, this class would be, for example, an interface, so you would use a flag `interface`. Therefore, `class` only exists as a *syntactical sugar* and does not have any other meaning. Listings 3.2, 3.3 and 3.4 shows some examples of valid class definitions in the Bytelang programming language.

Listing 3.2: Simple class definition

```
public class Foo {
}
```

Listing 3.3: Fairly hidden interface definition

```
synthetic final interface Foo {
}
```

Listing 3.4: Crazy class

```
synthetic abstract super class Foo {
}
```

Please note that listing 3.3 shows a code which is semantically invalid and a similar equivalent cannot be written using the Java programming language. However, it is a valid source code in the Bytelang programming language and can be even decompiled by `javap` (a standard Java decompilation tool provided by the JDK).

3.1.1 Inheritance

Class may have (but do not have to) specified a parent using the `extends` keyword. Source code 3.5 shows an example class which inherits from the class `java.lang.Object`.

Listing 3.5: Inheriting from `java.lang.Object`

```
public class Foo extends java.lang.Object {
}
```

It is necessary to always specify a fully qualified name of the class. Stating, for example, only `Object` would be interpreted as a class named `Object` located in the default package.

Definition of a super class might be omitted. In that case, class **does not** have any super class. In the Java programming language, each class will have, if not specified otherwise, a parent `java.lang.Object`. Do not get confused with that!

3.1.2 Implementing interfaces

Class may implement (but do not have to) any number of interfaces. Source code 3.6 shows an example class which implements interface `Boo` and interface `java.io.Serializable`.

Listing 3.6: Implementing interfaces `java.io.Serializable` and `Boo`

```
public class Foo implements java.io.Serializable, Boo {
}
```

It is necessary to always specify a fully qualified name of the class. This code assumes there is an interface called `Boo` located in the default package.

Please remember that Bytelang do not verify that your classes actually exists. It will accept any valid name, but you may not be able to actually execute your code. The same applies to inheritance (see 3.1.1).

3.2 Defining a constant pool

All content of a Class file is heavily bound to the so-called constant pool. Unless programmer forbade it, Bytelang will automatically generate everything what is necessary to deliver a proper Class file as described in the source code.

For example, when you define a class called `Foo`, Bytelang will automatically generated an `UTF8` item containing the string `Foo`, a `CLASS` item referring to the `UTF8` item and set the `CLASS` item as a `classFile.thisClass` item.

This applies to all items which are defined in the source code (class name, method names, field names, super class, implemented interfaces etc). However, when programmer needs to refer to some constant pool items from his code, he would have to define them himself. Defining of new constant pool items is described in the chapter 4.2.

3.3 Defining a field

Field definition matches basically the same rules as the Java programming language. The most significant difference is that all type names must be fully qualified. Listing 3.7 shows a general field definition:

Listing 3.7: General field definition

```
[accessFlag][propertyModifiers]<type><name>;
```

All possible access flags are shown in the table 3.3 and property modifiers in the table 3.4.

Source codes 3.8 and 3.9 shows example field definitions.

Listing 3.8: Fairly simple field definition

```
public class Foo {
        private static int[] mySimpleField;
}
```

public	private	protected

Table 3.3: All possible fields access flags

static	final	volatile
synthetic	enum	transient

Table 3.4: All possible fields property modifiers

Listing 3.9: Fairly simple field definition

```
public class Foo {
        static volatile java.lang.Object mutex;
}
```

Please note that when defining an array, square brackets must be located directly behind the type. It is not possible to place them behind the identifier as Java allows.

Also note that due to the bug in the current Bytelang compiler (version 2.0.0) only one dimension arrays are allowed.

Inline field initialization is not allowed.

3.4 Defining a method

Method definition matches basically the same rules as the Java programming language does. The most significant difference is that all type names must be fully qualified. Listing 3.10 shows a general method definition:

Listing 3.10: General method definition

```
[accessFlag][propertyModifiers]<returnType><methodName>(
[<type>[,<type[,...]]]) {
        // method implementation
}
```

Table 3.5 shows all possible access flags and 3.6 all possible property modifiers.

public	private	protected

Table 3.5: All possible method access flags

static	final	synchronized
bridge	varargs	native
abstract	strict	synthetic

Table 3.6: All possible method property modifiers

Listings 3.11, 3.12, 3.13 shows examples of method definitions:

Listing 3.11: Pretty simple method

```
public class Foo {
        public static int foo() {
                iconst_1
                ireturn
        }
}
```

Listing 3.12: Hidden method

```
public class Foo {
        static synthetic java.lang.Object foo(int) {
                aconst_null
                areturn
        }
}
```

Listing 3.13: Multi-parametric method

```
public class Foo {
        public static int sum(int, int) {
                iload_0
                iload_1
                iadd
                ireturn
        }
}
```

You may notice in the example 3.13 that method's arguments are specified only by a fully qualified name of type. Name of the argument is not present at all because it is not present in the generated bytecode too.

Chapter 4

Commands

Commands are delimited by an at sign (`@`). It uses the same syntax as Java annotations do, but it is something completely different.

Commands are used for defining new values or overriding the default compiler's behaviour (which is particularly useful when you need to generate an invalid Class file for JVM testing purposes).

Command might be applied to a class, to a method or to a field.

4.1 Syntax

Listing 4.1 shows general syntax rules for writing commands.

Listing 4.1: General syntax rules for writing commands

```
<@commandName>([param1=value1[, param2=value2[, ...]])
```

After a name of command, there must **always** be present left and right parenthesis.

Table 4.1: Allowed value types for command arguments

Name	Example
string	@foo(id="myFoo")
integer (dec)	@foo(value=0)
integer (hex)	@foo(value=0xFF)
reference	@foo(value=#someID)
array	@foo(value=[1, 0xFF, 3])

4.1.1 Parameters

Most of the commands will require additional values called *parameters* or *arguments*. Bytelang makes a difference between a mandatory argument (must be always set) and a non-mandatory argument (might not be set).

It is possible to pass only constant values as parameters. All possible types are shown in the table 4.1.

Note: *Decimal and hexadecimal integer are obviously of the same type. It is only written that way to demonstrate Bytelang's possibilites.*

Some commands accept multiple possible types for some arguments. In that case, you may pass value of any of those types and command will automatically convert it.

For example, when defining a constant pool (see 4.2), Bytelang allows you to pass an argument of type `string` to the command `@class`. In that case, Bytelang would automatically generate an item of type `UTF8` and use its index as an argument for the command `@class`.

4.2 Constant pool

Bytelang will automatically generate all constant pool items which are directly present in the source code (names, types etc.). This behaviour might be modified using the `@lock` command (see 4.2.1), which will prevent Bytelang from doing so.

Programmer, however, has to define all constant pool items which he wants to use in the bytecode.

4.2.1 `@lock`

Command `@lock` prevents compiler from generating constant pool items automatically. If source code contains some content, which has not been defined in the constant pool by user, compilation will fail.

Consider the code 4.2. It will be properly compiled and the generated Class file will match your expectations.

Listing 4.2: A code without `@lock`

```
public class Foo {
}
```

However, if you modified the previous code with a `@lock` command (as shown in the 4.3), compiler would reject the compilation (see 4.4).

Listing 4.3: An invalid code with `@lock`

```
@lock()
public class Foo {
}
```

Listing 4.4: Compilation error (inproperly applied `@lock`)

```
$ bytelang Foo.jsm Foo.class
Compilation error: Annotation @lock has been applied,
but there are constant-pool items (CLASS) to be
generated.
```

The error shown in the listing 4.4 notifies you that command `@lock` has been applied, but some constant pool items are missing and therefore compilation cannot continue, because it would result in an incomplete code.

You may fix it by defining all necessary items manually as shown in the listing 4.5.

Listing 4.5: Fixing the code 4.3

```
@lock()
@constantPoolBegin()
        @class(name=#StrClassName)
        @utf8(id="StrClassName", bytes="Foo")
@constantPoolEnd()
public class Foo {
}
```

Command `@lock` should be only used in the case you absolutely need to have a full control over the generated code and you need to prevent compiler from automatically generating any items.

Please note that using the command `@lock` does not affect automatic generation of other items (such as, for example, `classFile.thisClass`). You may, however, override these values manually using a command `@set` (see 4.3).

4.2.2 `@constantPoolBegin`, `@constantPoolEnd`

Definition of custom constant pool items must be delimited by commands `@constantPoolBegin` and `@constantPoolEnd`. Commands, which are used for specifing new constant pool items, will only work in this special context.

It is a good practice to always indent all content between these commands for a better readability.

Look at the code 4.5 for an example of using these two commands.

4.2.3 Defining new constant pool items

Look in the table 4.2 to see a set of all standard commands for defining new constant pool items.

Table 4.2: Commands for a standard constant pool definition

Command	Arguments	
	Name	**Type**
`@class`	`id` `name`	`str` `int, str, ref`

`@fieldref`	`id`
	`class`
	`nameAndType`
		`str`
		`int, str, ref`
		`int, str`

`@methodref`	`id`
	`class`
	`nameAndType`
		`str`
		`int, str, ref`
		`int, str`

`@interfaceMethodref`	`id`
	`class`
	`nameAndType`
		`str`
		`int, str, ref`
		`int, str`

`@string`	`id` `string`	`str` `int, str, ref`
`@integer`	`id` `bytes`	`str` `arr`
`@float`	`id` `bytes`	`str` `arr`

`@long`	`id`
	`highBytes`
	`lowBytes`
		`str`
		`arr`
		`arr`

`@double`	`id`
	`highBytes`
	`lowBytes`
		`str`
		`arr`
		`arr`

`@nameAndType`	`id`
	`name`
	`descriptor`
		`str`
		`int, str, ref`
		`int, str, ref`

`@utf8`	`id`
	`length`
	`bytes`
		`str`
		`int`
		`str, arr`

`@methodHandle`	`id`
	`referenceKind`
	`referenceIndex`
		`str`
		`int`
		`int, ref`

Table 4.3: Aliases for defining constant pool items

Original name	Alias
`@fieldref`	`@fref`
`@methodref`	`@mref`
`@interfaceMethodref`	`@imref`
`@nameAndType`	`@nat`
`@methodHandle`	`@mhandle`
`@methodType`	`@mtype`
`@invokeDynamic`	`@idyn`

Table 4.2: Commands for a standard constant pool definition

Command	Arguments	
	Name	Type
`@methodType`	`id`	`str`
	`descriptor`	`int, ref, str`
`@invokeDynamic`	`id`	`str`
	`bootstrapMethodAttr`	`int`
	`nameAndType`	`int, ref`

Note: In the table 4.2, `str` stands for a string, `int` for an integer, `ref` for a reference and `arr` for an array.

Because some names are too long, there are aliases (as shown in the table 4.3). Alias is an alternative (shorter) name for the same command.

Language shortcuts for defining new constant pool items

Bytelang 2 introduces new commands which might be used as shortcuts for defining some constant pool items much faster. All of these shortcuts are denoted with a letter *f* in the beggining, which stands for either *fake* (because such constant pool item does not actually exist) or *fast*. List of all of these shortcuts is shown in the table 4.4.

Table 4.4: List of all language shortcuts for defining new constant pool items

Command	Arguments	
	Name Meaning	Type

`@ffref`	`id`
	`class`
	`name`
	`type`
		`str`
		`int, str, ref`
		`int, str, ref`
		`int, str, ref`

`@fmref`	`id`
	`class`
	`name`
	`type`
		`str`
		`int, str, ref`
		`int, str, ref`
		`int, str, ref`

`@fimref`	`id`
	`class`
	`name`
	`type`
		`str`
		`int, str, ref`
		`int, str, ref`
		`int, str, ref`

Command `@ffref` is a shortcut for a field reference, `@fmref` for a method reference and `@fimref` for an interface method reference. Source code 4.6 shows an example *Hello World!* application which is written using these shortcuts.

Listing 4.6: Using language shortcuts for defining new constant pool items

```
@constantPoolBegin()
        @string(
                id = "StrHello",
                string = "Hello, I am Bytelang!"
        )
        @ffref(
                id="SystemOut",
                class="java/lang/System",
                name="out",
                type="Ljava/io/PrintStream;"
        )
        @fmref(
                id="println",
```

```
                class="java/io/PrintStream",
                name="println",
                type="(Ljava/lang/String;)V"
        )
@constantPoolEnd()
public class HelloWorld extends java.lang.Object {
        @set(maxStack = 2)
        @set(maxLocals = 1)
        public static void main(java.lang.String[]) {
                getstatic      #SystemOut
                ldc            #StrHello
                invokevirtual  #println
                return
        }
}
```

4.3 `@set`

`@set` is a special command which might be used for override the values present in the Class file, which are automatically generated by the compiler. This command might be used for a class (see the table 4.5), a method (see the table 4.6) or a field (see the table 4.7). These values exactly matches the JVM specification (see [JVM]).

4.4 `@method, @field`

`@method` is a command which might be used for adding new methods. Table 4.8 shows accepted arguments. This command might be used repeatedly.

Rules for the command `@field` are exactly the same, except it is used for defining fields instead of methods.

Code 4.7 shows using of a `@method` and a `@field` commands.

Table 4.5: Values (arguments) of a `@set` command for a class

Name of value	Accepted types
`magic`	`arr`
`version`	`arr, str`
`accessFlags`	`int`
`thisClass`	`int, ref`
`superClass`	`int, ref`
`interfacesCount`	`int`
`fieldsCount`	`int`
`methodsCount`	`int`
`attributesCount`	`int`

Table 4.6: Values (arguments) of a `@set` command for a method

Name of value	Accepted types
`methodAttributesCount`	`int`
`maxStack`	`int`
`maxLocals`	`int`
`codeLength`	`int`

Table 4.7: Values (arguments) of a `@set` command for a field

Name of value	Accepted types
`fieldAttributesCount`	`int`

Table 4.8: Values (arguments) of a `@method` command

Name of value	Accepted types
`accessFlags`	`int`
`name`	`int, ref, str`
`descriptor`	`int, ref, str`

Listing 4.7: Using of `@method` and `@field` commands

```
@method(
        accessFlags=0,
        name="foo",
        descriptor="()V"
)
@method(
        accessFlags=0,
        name="boo",
        descriptor="()V"
)
@field(
        accessFlags=0,
        name="val",
        descriptor="I"
)
@field(
        accessFlags=0,
        name="num",
        descriptor="I"
)
public class Foo {
}
```

4.5 `@interface`

Commands `@interface` might be used for specifying the implemented interfaces. The only possible argument is a `classIndex`, which is allowed to be of a type `integer`, `string` or `reference`. Code 4.8 shows an example usage.

Listing 4.8: Using of a command `@interface`

```
@constantPoolBegin()
        @class(id="boo", name="Boo")
@constantPoolEnd()
```

Table 4.9: Arguments of an `@attribute` command

Name of value	Accepted types
`name`	`int, str`
`length`	`int`
`data`	`arr`

```
@interface(classIndex="java/io/Serializable")
@interface(classIndex=#boo)
public class Foo {
}
```

4.6 `@attribute`

Command `@attribute` is used for adding attributes to classes, fields or methods. Table 4.9 shows all arguments accepted by this command.

Arguments `length` and `data` are not mandatory. The code 4.9 shows how to use this command.

Listing 4.9: Using of attributes

```
@attribute(
        name="foo",
        length=3,
        data=[0xAA, 0xBB, 0xCC]
)
public class Foo {
        @attribute(name="boo")
        public java.lang.String text;

        @attribute(name="goo", data=[1, 2, 3])
        @attribute(name="hoo", length=0)
        public void foo() {
        }
}
```

Chapter 5

Bytecode

The Bytelang programming language supports all instructions specified by the JVM specification (see [JVM]), except the instructions `lookupswitch` and `tableswitch`.

These instructions can be written into the method's body.

Jumps use offsets. Bytelang introduces labels and automatically calculates offsets as a distance from the current instruction to the specified label.

Definition of a label must be always started with a dot in the beginning. The code 5.1 shows an example source code using labels.

Listing 5.1: Using of labels

```
public class Foo {
        public int foo() {
                iconst_0
                ifeq .true
                iconst_1
                ireturn
                .true:
                        iconst_2
                        ireturn
        }
}
```

When calling an instruction with multiple operands, these operands are separated using a comma as shows the code 5.2.

Listing 5.2: Invoking instructions with multiple operands

```
public class Foo {
        public int foo() {
                iconst_0
                istore_0
                iinc 0, 1
        }
}
```

Chapter 6

Example source codes

6.1 Hello, I am Bytelang!

The source code 6.1 shows a simple *Hello, world!* application (the only way how it could be written in the first version of Bytelang).

Listing 6.1: Hello I am Bytelang!

```
@constantPoolBegin()
        @class(
                id = "ClassSystem",
                name = "java/lang/System"
        )
        @nameAndType(
                id = "NTout",
                name = "out",
                descriptor = "Ljava/io/PrintStream;"
        )
        @fieldref(
                id = "SystemOut",
                class = #ClassSystem,
                nameAndType = #NTout
        )
        @string(
```

```
                id = "StrHello",
                string = "Hello, I am Bytelang!"
        )
        @class(
                id = "ClassPrintStream",
                name = "java/io/PrintStream"
        )
        @nameAndType(
                id = "NTprintln",
                name = "println",
                descriptor = "(Ljava/lang/String;)V"
        )
        @methodref(
                id = "println",
                class = #ClassPrintStream,
                nameAndType = #NTprintln
        )
@constantPoolEnd()
public class HelloWorld extends java.lang.Object {
        @set(maxStack = 2)
        @set(maxLocals = 1)
        public static void main(java.lang.String[]) {
                getstatic      #SystemOut
                ldc            #StrHello
                invokevirtual  #println
                return
        }
}
```

6.2 Hello, I am Bytelang! #2

The source code 6.2 shows a simple *Hello, world!* application which uses language shortcuts from Bytelang 2 (see 4.2.3).

Listing 6.2: Hello I am Bytelang! #2

```
@constantPoolBegin()
```

```
        @string(
                id = "StrHello",
                string = "Hello, I am Bytelang!"
        )
        @ffref(
                id="SystemOut",
                class="java/lang/System",
                name="out",
                type="Ljava/io/PrintStream;"
        )
        @fmref(
                id="println",
                class="java/io/PrintStream",
                name="println",
                type="(Ljava/lang/String;)V"
        )
@constantPoolEnd()
public class HelloWorld2 extends java.lang.Object {
        @set(maxStack = 2)
        @set(maxLocals = 1)
        public static void main(java.lang.String[]) {
                getstatic      #SystemOut
                ldc            #StrHello
                invokevirtual  #println
                return
        }
}
```

6.3 Numbers

The source code 6.3 show a simple application which prints numbers from 1 to 15 to the standard output.

Listing 6.3: Numbers

```
@constantPoolBegin()
        @class(
```

```
        id = "ClassSystem",
        name = "java/lang/System"
)
@nameAndType(
        id = "NTout",
        name = "out",
        descriptor = "Ljava/io/PrintStream;"
)
@fieldref(
        id = "SystemOut",
        class = #ClassSystem,
        nameAndType = #NTout
)
@string(
        id = "StrSep",
        string = " "
)
@class(
        id = "ClassPrintStream",
        name = "java/io/PrintStream"
)
@nameAndType(
        id = "NTprintlni",
        name = "println",
        descriptor = "(I)V"
)
@methodref(
        id = "printlni",
        class = #ClassPrintStream,
        nameAndType = #NTprintlni
)
@nameAndType(
        id = "NTprinti",
        name = "print",
        descriptor = "(I)V"
)
@methodref(
        id = "printi",
```

```
                class = #ClassPrintStream,
                nameAndType = #NTprinti
        )
        @nameAndType(
                id = "NTprintlnv",
                name = "println",
                descriptor = "()V"
        )
        @methodref(
                id = "printlnv",
                class = #ClassPrintStream,
                nameAndType = #NTprintlnv
        )
        @nameAndType(
                id = "NTprints",
                name = "print",
                descriptor = "(Ljava/lang/String;)V"
        )
        @methodref(
                id = "prints",
                class = #ClassPrintStream,
                nameAndType = #NTprints
        )
@constantPoolEnd()
public class Numbers extends java.lang.Object {
        @set(maxStack = 15)
        @set(maxLocals = 1)
        public static void main(java.lang.String[]) {
                getstatic #SystemOut

                iconst_1
                istore_0

                .rep:
                        dup
                        iload_0
                        invokevirtual #printi
                        iinc 0, 1
```

```
                iload_0
                bipush 15
                isub
                ifgt .end

                dup
                ldc #StrSep
                invokevirtual #prints
                goto .rep
            .end:
                invokevirtual #printlnv
                return
        }
}
```

Chapter 7

Compiling and running the applications

Compilation might be done using a standard Bytelang compiler. It is a conventional console application, which accepts two arguments: name of the input file (a source code) and name of the output file (a Class file). Example 7.1 shows a compilation process.

Listing 7.1: Using the Bytelang compiler

```
$ bytelang Numbers.jsm Numbers.class
```

Example 7.2 shows how to run a compiled code:

Listing 7.2: Running the compiled code

```
$ java -noverify Numbers
```

You may notice the `noverify` argument passed to `java`. To be able to run your code without this argument, you would have to add a lot of attributes, which are described in the JVM specification (see [JVM]). This is possible, but since Bytelang does not provide a direct support for them and you would have to use a command `@attribute` (with manually filled bytes), it would be quiet difficult.

Bibliography

[JVM] LINDHOLM, Tim, Frank YELLIN, Gilad BRACHA a Alex BUCKLEY. ORACLE AMERICA, Inc. *The Java Virtual Machine Specification: Java SE 7 Edition [online]. Redwood City (California), 2013* `http://docs.oracle.com/javase/specs/jvms/se7/jvms7.pdf`