Programming Language Specification

Version 1.0

Sebastian Nicolai Kaupe

16th June 2015

Contents

1 Introduction

The *educational Java-esque Virtual Machine Programming Language*—*e***JVM** Programming Language for short—is a simple, stack-based programming language meant to help students of Computer Science understand how stack-based machines, such as the **Java Virtual Machine**, operate.

To achieve this, the eJVM Programming Language is largely based on the **IJVM** instruction set developed by Tanenbaum in his book *Structured Computer Organization* as an example for a processor's instruction set. It is a heavily reduced subset of the instruction set used in the Java Virtual Machine, featuring only instructions to work with integer values (and arrays thereof in some versions) and basic jump instructions.

eJVM adapts most of these instructions without changes, but omits all array instructions, applies a few carefully chosen changes on others and adds instructions for input and output handling. This was done to facilitate the easy creation of simple programs by students in order to study the execution of said programs in a stack-based machine, for which arrays are often unnecessary. Furthermore, eJVM specifies its own format for compiled executables, designed to be simple to understand if needed.

While all these changes make the eJVM Programming Language incompatible with the existing IJVM simulators available, this is of no concern as it allows to include features not present in the IJVM language, providing students with a more helpful and easy-to-use environment (such as customizable error messages).

2 Language Specification

This chapter describes the new language. It explains the syntax elements, defines the precise modus operandi of the available instructions and presents the layout of the compiled program files.

2.1 eJVM Program Syntax

eJVM programs are constructed using three elements: directives, instructions and comments. **Directives** organize a program into different parts responsible for different things, like the definition of constants or methods. **Instructions** form the effective actions the program undertakes to achieve its purpose. **Comments** are used to document the program and facilitate its reading by other programmers and may be freely mixed with directives and instructions. The simplistic division program in listing 2.1 shall serve as an example for an eJVM program with its usual elements. Directives are highlighted in blue, comments in green and instructions as well as arguments to them as well as directives remain black, with the exception of error message strings in the `.errors` directive (section 2.2.4), which are highlighted using a red color. This style of highlighting is used throughout this document.

```
1  .program Division
2
3  .constants
4      LINEBREAK       10  ; ASCII-/UTF16-code for linebreak character.
5  .end-constants
6
7  .errors
8      E_DIV_BY_ZERO   "Division by zero!"
9  .end-errors
10
11 .method main()
12     .vars
13         dividend
14         divisor
15         cnt                     ; Number of subtractions (the result).
```

```
16      .end-vars
17          BIPUSH 15               ; Initialize dividend and divisor.
18          ISTORE dividend
19          BIPUSH 5
20          ISTORE divisor
21          ILOAD divisor           ; Check for division by zero.
22          IFEQ err                ; Go to err if dividend is 0.
23  sub:    ILOAD divisor           ; Place divisor and...
24          ILOAD dividend          ; ...dividend on the stack.
25          ISUB                    ; Subtract divisor from current value.
26          DUP                     ; Duplicate value.
27          IFLT end                ; We are done once we are below 0.
28          ISTORE dividend         ; Write new value into local variable.
29          IINC cnt 1              ; Increment subtraction counter.
30          GOTO sub                ; Next iteration.
31  end:    ILOAD cnt               ; Put cnt on stack.
32          INVOKEVIRTUAL printline ; Invoke printline with no as parameter.
33          RETURN;                 ; End program by returning from main().
34  err:    ERR E_DIV_BY_ZERO       ; Abort with error message.
35  .end-method
36
37  ; Prints the argument as a number and appends a line break.
38  .method printline(no)
39          SETOUT NUMBER           ; Switch output to numbers.
40          ILOAD no                ; Put argument on stack.
41          OUT                     ; Print argument out.
42          SETOUT CHAR             ; Switch output to characters.
43          LDC LINEBREAK           ; Load linebreak character (ASCII/UTF-16).
44          OUT                     ; Print linebreak.
45          RETURN
46  .end-method
```

Listing 2.1: A simple eJVM program realizing unsigned integer division

2.1.1 Literals

Numeric literals, such as arguments to instructions or the value of constants (see lines 17 and 4 of listing 2.1 for respective examples), can be written using decimal or hexadecimal numbers. The latter have to be prefixed with an `0x`.

2.1.2 Comments

Comments are initiated by a single semicolon (`;`) and stretch to the end of the line. eJVM assumes UNIX-style line ends, that is, a single line-feed character as defined by ASCII (LF, `0x0A`, `'\n'`). All characters behind the leading semicolon are ignored by the compiler.

2.2 eJVM Directives

2.2.1 Overview

The eJVM Programming Language offers the following directives:

.program **.constants** **.errors**
.method **.vars**

As can be seen, directives always start with a leading dot, followed by the directive's name. Except for the `.program` directive, every directive has a closing counterpart prefixed with `.end-`, followed by the directive's name:

.end-constants **.end-errors**
.end-method **.end-vars**

These closing directives are used to end the block of definitions every directive opens. The `.program` directive is exempt from this rule because interpreting the end of the program file as end of the contained program is sufficient.

2.2.2 .program

The `.program` directive starts a program and provides a name for it. Unlike all other directives, it does *not* have a closing counterpart. The name may take up to 255 bytes of space. eJVM uses UTF-16 in the Big Endian variant to encode all characters, which leads to program names having a maximum of 127 UTF-16 characters (as long as characters from the Basic Multilingual Plane are used that do not take up more than two bytes). The program name *must* start with a character, but may contain white space or numbers after the initial character.

Syntax: `.program Name of the eJVM Program`

2.2.3 .constants

The `.constants` directive opens a block in which program constants may be declared. A program constant is a constant value available under the name used to define it in the `.constants` block and may be accessed from methods using the LDC instruction. Because there is no instruction to change the value of a constant (and it is against a constant's nature to be changed), every constant defined must be provided with a value. The coding conventions for eJVM programs recommend to use all-uppercase letters for constant names.

Syntax:

```
.constants
    CONSTANT_0      0x00
    CONSTANT_1      1
.end-constants
```

2.2.4 .errors

The `.errors` directive opens a block in which error messages may be declared. In case of an error during program execution, these error messages can be used to provide the ERR instruction with an error message it can display to the user, allowing for a more sophisticated error handling as it was possible with IJVM.

An error message declaration consists of the name of the message and the actual message itself, enclosed in quotes. Coding conventions recommend to use all-uppercase letters for the error message name and to prefix it with `E_` to make distinction between constants and error messages more easy (even though it is not possible to use error messages in conjunction with the LDC instruction or constants with the ERR instruction).

Like the program name, error messages are encoded using the big endian version of UTF-16 and may take up to 255 bytes of space.

Syntax:

```
.errors
    E_NAME  "Error message"
.end-errors
```

2.2.5 .method

The `.method` directive is the most complex directive eJVM offers and declares a method in the eJVM program. The opening directive takes the method declaration as an argument. A method declaration consists of a name and an optional parameter list. Like the program name, a method name must start with a character and may take up to 255 bytes of space in UTF-16BE-encoded characters. It is always to be followed by a pair of round brackets. Inside the brackets, method parameters may be declared using arbitrary names (again starting with a character, but they may contains numbers afterwards). If more than one parameter is present, the parameters are separated using commas. The combined number of parameters and local variables (section 2.2.6) must not exceed 255.

Inside the block opened by the `.method` directive, the `.vars` directive may be used to declare variables local to that method (see section 2.2.6 for more about local variables). No other directives may be used inside a `.method` block, but unlike all other blocks opened by a directive, it may contain eJVM instructions. A list of instructions is available in section 2.3.

A method's size is not unlimited. Because eJVM uses 4-byte addresses and 2-byte offsets for jump instructions, any method in an eJVM program may only take up to $2^{16} - 1$ (65535) byte of space after compilation.

To be a valid eJVM program, a program must provide a method named `main` that does not expect any parameters. Furthermore, every method must contain atleast one RETURN or IRETURN instruction to ensure that a method returns to its caller.

Ordering of methods inside of an eJVM program is irrelevant. However, the compiler guarantees that the `main` method is always at the beginning of the compiled program.

Syntax:

```
.method name(param0, param1)
    RETURN
.end-method
```

2.2.6 .vars

The `.vars` directive can only be placed at the beginning of a `.method` block and contains declarations of local variables for that method. A local variable is declared with only its name (following the same rules as method names, see section 2.2.5 for details). During runtime, the eJVM interpreter guarantees that all local variables declared are initialized with the value `0`. The combined number of local variables and parameters (section 2.2.5) for a method must

not exceed 255.

Syntax:

```
.vars
    variable0
    variable1
.end-vars
```

2.3 eJVM Instructions

2.3.1 Overview

The eJVM Programming Language consists of the following instructions:

BIPUSH	**DUP**	**POP**	**SWAP**	**ILOAD**
ISTORE	**LDC**	**IADD**	**ISUB**	**IINC**
IAND	**IOR**	**GOTO**	**IFEQ**	**IFLT**
IF_ICMPEQ	**INVOKEVIRTUAL**	**IRETURN**	**RETURN**	**IN**
OUT	**SETOUT**	**ERR**	**HALT**	**NOP**
WIDE				

These instructions can be further classified into one of six different types:

Stack manipulation instructions modify the values placed upon the program stack, i. e. the method parameters, local variables and working values.

Mathematical instructions conduct mathematical operations using the values placed on the program stack.

Logical instructions conduct logical operations using the values placed on the program stack.

Control flow instructions change a program's control flow, e. g. invoking a method or jumping to another instruction.

I/O instructions perform input and output operations or change properties of the input and output channels.

Control instructions offer limited control over the executing machine itself.

Table 2.1 shows the instructions grouped by their respective type.

Stack manipulation	Mathematical	Logical	Control flow	I/O	Control
BIPUSH	**IADD**	**IAND**	**GOTO**	**IN**	**ERR**
DUP	**ISUB**	**IOR**	**IFEQ**	**OUT**	**HALT**
POP	**IINC**		**IFLT**	**SETOUT**	**NOP**
SWAP			**IF_ICMPEQ**		**WIDE**
ILOAD			**INVOKEVIRTUAL**		
ISTORE			**IRETURN**		
LDC			**RETURN**		

Table 2.1: eJVM instructions by type

2.3.2 Arguments

Instructions may receive none, one or two arguments. Arguments are separated into different types for convenience of reference, as shown by table 2.2. All arguments are compiled into the executable using big-endian order.

Name	Size	Value range	Description
INDEX	1 Byte	0–255	Index value for one of the tables in a compiled eJVM program, e. g. the method table.
OFFSET	2 Byte	-32,768–32,767	Byte offset for jump instructions (relative to the position of said instructions).
LITERAL	2 Byte	-32,768–32,767	Literal number compiled into the executable. This value may also be interpreted as an UTF-16BE-encoded character.

Table 2.2: Instruction argument types

When writing eJVM code to be compiled by the eJVM compiler, INDEX and OFFSET types of arguments are not written as number, instead using the identifier of the accessed variable, constant, method or label. The compiler handles the conversion into a numerical value according to this document.

2.3.3 Stack Layout and Operation

Every instruction's description includes the changes they exert on the stack as well as a formal descriptions of their operation in a simple, mathematical notation. These latter explains the exact operations executed by the instruction on the arguments it receives from or

puts onto the stack.

Usually, operands are working values (see chapter 4 for an explanation of the term) taken from the stack and referred to as in_i, with i being the index of the operand on the stack (starting with 0 for the highest working value). In the same fashion, new working values pushed onto the stack by the instruction are usually named as out_i. The other, unchanged parts of the stack are denoted by an ellipse (. . .) at the left edge of the stack layout description. The ellipse may also be used to denote a range of similar values, e. g. a the entirety of a methods parameter as $p_0 \ldots p_n$.

In some cases, different names may be chosen when the values pushed onto or taken from the stack are not working values (for example, the INVOKEVIRTUAL instruction pushes a new stack frame onto the stack, containing different fields named appropriately).

In addition, their are certain predefined operands used by the descriptions to signify access to certain fields or tables of the eJVM program or process:

VAR: The table of local variables and method parameters.

CONST: The program's tabel of constants.

METHOD: The table of methods provided by the program.

PC: The program counter of the eJVM process, pointing to the next instruction.

FP: The processes frame pointer, pointing to the base of the current method's stack frame.

SP: The stack pointer of the process, pointing to the top of the stack.

INPUT: The input channel associated with the current eJVM process.

OUTPUT: The output channel associated with the current eJVM process.

Access to entries in tables is signified by the use of a C-like array notation: *VAR[i]* indicates access to the i^{th} entry of the local variables table. Entries in the methods table have different fields, access to which will be designated using Dot-syntax, similar to the access to fields in C structs: *METHOD[i].startAddress*.

Assigning a value to the *OUTPUT* channel signifies this value being written into said channel as process output, will assigning *INPUT* to a value signifies that value being read from the input channel.

For the sake of brevity, modifying the stack always implies a change of the stack pointer (*SP*). Therefore, assignment to the stack pointer does not occur in the description, but very much occurs in the virtual machine every time the stack is manipulated.

2.3.4 Stack Manipulation Instructions

BIPUSH Pushes k as a new working value onto the stack.

OpCode: `0x10`

Parameters	
k	LITERAL

Stack Layout:

Before: ...
After: ... out_0

Operation: $out_0 = k$

DUP Duplicates the top-most working value on the stack.

OpCode: `0x59`

Stack Layout:

Before: ... in_0
After: ... out_0 out_1

Operation: $out_0 = out_1 = in_0$

POP Removes the top-most working value from the stack.

OpCode: `0x57`

Stack Layout:

Before: ... in_0
After: ...

SWAP Swaps the two top-most working values on the stack.

OpCode: `0x5F`

Stack Layout:

Before: ... in_0 in_1
After: ... out_0 out_1

Operation: $out_0 = in_1, out_1 = in_0$

ILOAD Loads the local variable at index i and pushes it onto the stack as a new working variable.

OpCode: `0x15`

Parameters	
i	INDEX

Stack Layout:

Before: ...
After: ... out_0

Operation: $out_0 = VAR[i]$

ISTORE Pops the top-most working value from the stack and stores it at the local variable slot at index i.

OpCode: `0x36`

Parameters	
i	INDEX

Stack Layout:

Before: ... in_0
After: ...

Operation: *VAR[i]* = in_0

LDC Loads the value of the constant at index i from the program's constant table and pushes it onto the stack as a new working value.

OpCode: `0x12`

Parameters	
i	INDEX

Stack Layout:

Before: ...
After: ... out_0

Operation: out_0 = *CONST[i]*

2.3.5 Mathematical Instructions

IADD Removes the two top-most working values from the program stack, adds them together and pushes the resulting value back on the program stack.

OpCode: `0x60`

Stack Layout:

Before: ... in_1 in_0
After: ... out_0

Operation: $out_0 = in_1 + in_0$

ISUB Removes the two top-most working values from the program stack, subtracts them and pushes the resulting value back on the program stack.

OpCode: `0x64`

Stack Layout:

Before: ... in_1 in_0
After: ... out_0

Operation: $out_0 = in_1 - in_0$

IINC Increments the stack variable at index i by k.

OpCode: `0x84`

Parameters	
i	INDEX
k	LITERAL

Operation: $VAR[i] = VAR[i] + k$

2.3.6 Logical Instructions

IAND Removes the two top-most working values from the program stack, applies a logical AND to them and pushes the resulting value back on the program stack.

OpCode: `0x7E`

Stack Layout:

Before: ... in_1 in_0
After: ... out_0

Operation: $out_0 = in_0 \wedge in_1$

IOR Removes the two top-most working values from the program stack, applies a logical OR to them and pushes the resulting value back on the program stack.

OpCode: `0x80`

Stack Layout:

Before: ... in_1 in_0
After: ... out_0

Operation: $out_0 = in_0 \vee in_1$

2.3.7 Control Flow Instructions

GOTO Jumps to another instruction by moving the program counter by the number of bytes indicated by its argument *o*.

OpCode: `0xA7`

Parameters	
o	OFFSET

Operation: $PC = PC + o$

IFEQ Pops the topmost working value and compares it to 0. If both are equal, execution jumps to another instruction by moving the program counter by the number of bytes indicated by the instruction's argument *o*.

OpCode: `0x99`

Parameters	
o	OFFSET

Stack Layout:

Before: ... in_0
After: ...

Operation: $PC = PC + o$ if $in_0 == 0$

IFLT Pops the topmost working value and compares it to 0. If the stack value is less than 0, execution jumps to another instruction by moving the program counter by the number of bytes indicated by the instruction's argument *o*.

OpCode: `0x9B`

Parameters	
o	OFFSET

Stack Layout:

Before: ... in_0
After: ...

Operation: $PC = PC + o$ if $in_0 < 0$

IF_ICMPEQ Pops the two topmost working values and compares them. If both are equal, execution jumps to another instruction by moving the program counter by the number of bytes indicated by the instruction's argument *o*.

OpCode: `0x9F`

Parameters	
o	OFFSET

Stack Layout:

Before: ... in_1 in_0
After: ...

Operation: $PC = PC + o$ if $in_0 == in_1$

INVOKEVIRTUAL Invokes the execution of another method of the eJVM program, designated by its index i in the program's methods table. A number of working values equal to the number of method parameters is removed from the stack before a new stack frame is added to the process stack, consisting of the index of the method invoked, the program counter and frame pointer, followed by the parameters taken from the stack and a 0 for every local variable declared inside the method. INVOKEVIRTUAL assumes that the parameters for the invoked method have been pushed onto the stack in order of their declaration.

OpCode: `0xB6`

Parameters	
i	INDEX

Stack Layout:

Before: ... $p_0 \dots p_n$
After: ... i oPC oFP $p_0 \dots p_n$ $locals$

Operation:
$oPC = PC$
$oFP = FP$
$FP = SP$
$PC = METHOD[i].startAddress$
$locals = 0 \forall local_i,\ i \in \{i \in \mathbb{N} \mid 0 \leq i \leq METHOD[i].numberOfLocalVariables\}$

IRETURN Ends execution of the current method by removing the top-most stack frame from the process stack, conserving the top-most working value as return value for the method. If the method returned from is the original invocation of the program's `main` method (identifiable by `MI == 0` and `oFP == 0xFFFFFFFF`), program execution ends.

OpCode: `0xAC`

Stack Layout:

Before: ... MI oPC oFP $p_0 \dots p_n$ $locals$ $w_0 \dots w_n$
After: ... out_0

Operation:
$out_0 = w_n$
$FP = oFP$
$PC = oPC$

RETURN Ends execution of the current method by removing the top-most stack frame from the process stack without placing a return value for the method onto the stack. If the method returned from is the original invocation of the program's `main` method (identifiable by `MI == 0` and `oFP == 0xFFFFFFFF`), program execution ends.

OpCode: `0xB1`

Stack Layout:

Before: ... *MI* *oPC* *oFP* $p_0 \dots p_n$ *locals* $w_0 \dots w_n$
After: ...

Operation:
$FP = oFP$
$PC = oPC$

2.3.8 I/O Instructions

IN Attempts to read in a single character from the input stream connected to the process the eJVM program runs in and places it unto the stack as the top-most working variable. If no character can be read, the instruction blocks the process until a character is available.

OpCode: `0xF0`

Stack Layout:

Before: ...
After: ... out_1

Operation: $out_1 = INPUT$

OUT Pops the top-most working value off the stack and writes it into the output channel associated with the process the eJVM program runs in. Whether the working value is written as numerical value or interpreted as an UTF-16 character before being written depends on the output mode currently used for the process. See SETOUT for more information about output modes.

OpCode: `0xF1`

Stack Layout:

Before: ... in_1
After: ...

Operation: $OUTPUT = in_1$

SETOUT Sets the output mode for the eJVM program. The output mode of a program defines how output send to the output channel via the OUT instruction is formatted before being printed. For this first version of the eJVM Programming Language, two output modes have been defined:

CHAR: The value popped off from the stack is interpreted as a character encoded in the UTF-16BE encoding and printed as such.

NUMBER: The value popped off from the stack is printed as a numerical value, possibly comprised out of several digits, all of which will be printed out.

In the compiled eJVM program, these modes are referred to by their respective byte value (`0` for CHAR and `1` for NUMBER).

OpCode: `0xFA`

Parameters	
i	INDEX

2.3.9 Control Instructions

ERR Halts the execution of the program by the eJVM interpreter, with the reason for the halt indicated by the parameter `i`, which refers to a corresponding error message in the program's error message table.

OpCode: `0xF2`

Parameters	
i	INDEX

HALT Halts the execution of the program by the eJVM interpreter.

OpCode: `0xFF`

NOP Performs no operation.

OpCode: `0x00`

WIDE *(deprecated)* In the original IJVM language, WIDE was used to signalise that the following instruction would be expecting 16 Bit indexes instead of a 8 Bit indexes. Since eJVM does not support wide indexes, the instruction was deprecated and now behaves like an invocation of the NOP instruction.

OpCode: `0xC4`

2.3.10 Differences to IJVM

This section summarizes the differences between eJVM and IJVM to make porting IJVM programs to eJVM code easier. Even though both languages are closely related, eJVM sports a number of differences in comparison with IJVM. Most of those have been introduced to strengthen its ability to be used as an educational language.

I/O instructions: Tanenbaum's original IJVM language does not feature I/O instructions [**tanenbaum:sco**]. However, those instructions are sometimes found in IJVM interpreters[1] in order to provide users with the ability to process input and produce output and have been included into the instruction set of eJVM for the very same purpose.

ERR instruction: eJVM added the ERR instruction to the IJVM instruction set to provide students with an easy mechanism to abort program execution in case of an error. A similar instruction may be found in some IJVM interpreters; however, that version of the ERR instruction does not feature the automatic display of error messages that has been added to the eJVM ERR instruction (and, consequently, does not take an argument in IJVM).

WIDE instruction: The WIDE instruction found in Java byte code and IJVM signalised that the following instruction was using 16 bit indexes instead of 8 bit indexes. eJVM only supports the latter, which is why the WIDE instruction was deprecated and behaves like an invocation of NOP (i. e. does nothing).

LDC and LDC_W: In IJVM, the instruction to load a constant is LDC_W. This instruction has been renamed to LDC in the eJVM language as the _W served no purpose aside from being a hassle to type.

Arrays: In some versions, IJVM features instructions to declare and work with arrays of integers. While arrays can be helpful, many educational programs do not necessarily need arrays. As the primary purpose of eJVM is to help students to understand how stack-based machines work and not necessarily to teach assembly level programming (as is IJVM's purpose), these instruction have been omitted from eJVM.

RETURN instruction: IJVM only has IRETURN to return from a method. eJVM adds the RETURN instruction to make the implementation of methods without return value more easy and to keep the stack clean of unneeded return values.

OpCodes: eJVM's opcodes for instructions are, as far as possible, the same as the byte codes for the corresponding instructions in the Java Programming Language (TODO: Ref). However, both IJVM and eJVM add instructions not present in the Java byte code. For these instructions, eJVM specifies its own opcodes that may be different from those used by IJVM interpreters. This is the case for the OUT, IN, ERR, HALT and SETOUT

[1] http://www.supereasyfree.com/software/simulators/structured-computer-organization-tanenbaum/ijvm-simulator/ijvm-simulator.php

instructions, for which easily readable opcodes where chosen. The opcodes for IN, OUT and ERR were chosen to mimic the standard I/O file descriptors found on UNIX system: `0xF0` for IN, `0xF1` for OUT and `0xF2` for ERR, as ERR now produces an error message. HALT is easily distinguishable from other instructions by the use of `0xFF` as opcode.

.main directive: IJVM interpreters usually expect a method declared using a directive called `.main`. This was replaced for eJVM with the interpreter expecting a normal `.method` named main to strengthen the similarity to programming languages possible already known by students, such as C or Java.

INVOKEVIRTUAL and OBJREF: The original IJVM language required the reference to an object, called OBJREF, to be pushed onto the stack when using the INVOKEVIRTUAL instruction, as the same is done in Java byte code to enable dynamic binding. eJVM omitted this requirement and removed the OBJREF entry from the constant table because it was unneeded for the purpose of teaching stack-based machine behaviour.

3 eJVM Executable File Format

eJVM programs are compiled into their own executable format, designed to be relatively simple to understand and, if need be, to be read using nothing but a hex editor. Every executable starts with a number of fixed-width fields, follows with the often-used tables containing methods and constants before the actual byte code and ends with mostly optional features. The program name is the very last element, placed at the end of the executable file to make it easy to see if some bytes of the executable have somehow been lost (e. g. during a transmission over a network).

Addresses and offsets are always absolute, i. e. expect the program to be loaded into memory at position zero. If this is not the case, it is the virtual machine's responsibility to simulate that it is loaded at memory position zero, that is, to compensate for the misplacement by adding the additional offset onto each address and offset during execution.

eJVM executables are expected to be using the Big Endian system, that is, with the most significant byte stored at the lowest memory address.

3.1 General Structure

Table 3.1 on page 26 lists all elements of an eJVM executable. All length and offset values are in byte. An offset of 13+ indicates that it is a static offset of 13 bytes (due to the fixed-width fields in the beginning of the executable file) plus a number of bytes that depend on each program. The actual offset can be calculated by adding the size of all fields up to the desired field together. The overall size of the executable is calculated as `pnoff + pnsize`.

3.2 Method Table Structure

The method table (field `mtbl` in the executable) contains a 10-byte entry for every method in the eJVM program, containing the start address for the compiled method code, the number of parameters and local variables the method expects and the address of the optional block with debug information. Table 3.2 describes this.

Offset	Length	Name	Description
0	4	magic	Magic number, ASCII-encoded 'eJVM' (`0x654A564D`)
4	1	vno	Number of eJVM version, major and minor (each 4 bit), 1.0 (`0x10`) for this version
5	1	pnsize	Program name length (in bytes)
6	4	pnoff	Byte offset the the first byte of the program's name
10	1	mcnt	Number of methods in the method table (0 to 255)
11	1	ccnt	Number of constants in the constant table (0 to 255)
12	1	errcnt	Number of error messages in the error message table (0 to 255)
13	mcnt * 10	mtbl	The method table, one 10-byte entry per method. `main()` is always the first entry
13+	ccnt * 2	ctbl	Constant table, every constant is two byte
13+	?	text	Compiled method code. Code of `main()` is always placed at the beginning
13+	?	dbgpool	Debug information pool (optional), one entry for every method with information about it
13+	?	errtbl	Error message table, 1-byte message size followed by the UTF-16BE-encoded message itself (optional)
pnoff	pnsize	pname	Program name as declared in the source file, encoded using UTF-16BE

Table 3.1: eJVM executable file structure

It is of note that the combined number of method parameters and local variables must *not* exceed 255. This was done deliberately so that parameters and local variables may be stored in one continuous section and accessed using the same instructions.

3.3 Debug Information Structure

Debug information are an optional feature eJVM programs may make use of. The block of debug information, if compiled into the executable, contains the name for each method as well as the size of its compiled code, as seen in table 3.3.

Name	Length	Description
maddr	4	Address of the method's first instruction in the `text` section of the executable file
pcnt	1	The number of parameters the method expects
lcnt	1	The number of local variables declared for the method
dbgaddr	4	The address of the block with debug information for this method. If the executable was compiled omitting debug information, the value of this field will be 0

Table 3.2: Method table structure

Name	Length	Description
mnsize	1	The size of the method's name (in byte)
msize	2	The size of the compiled method's code
mname	mnsize	The name of the method, encoded using UTF-16BE

Table 3.3: Debug information structure

3.4 Error Message Table Structure

Error messages are stored in a very simple table structure, using just two fields for every message. Table 3.4 explains these fields.

Name	Length	Description
emsize	2	The size of the of the error message (in byte)
emsg	emsize	The actual error message, encoded using UTF-16BE

Table 3.4: Error message table structure

4 Execution of eJVM Programs

As already stated in chapter 3, eJVM executables expect to be loaded into memory at position zero, that is, the beginning of the machine's memory. If this cannot be guaranteed (e. g. because the interpreter supports parallel execution of different eJVM programs), the interpreter has to feign being loaded at address zero to the program.

Memory can be divided into two sections: the loaded program at the beginning (consisting of all the elements described in chapter 3) and the program stack somewhere behind it. eJVM programs work with 32-bit addresses; therefore, limiting the machine's memory to $2^{32} - 1$ bytes is encouraged. A heap section is not needed.

Once a program has been loaded into memory, the interpreter has to set up the pointers needed for execution. The interpreter has to ensure that input and output channels are available and ready for I/O operations done by the program. The **program counter (PC)** is set to point to the first instruction of the `main()` method, located at the beginning of the `text` section of the program (see section 3.1). The **frame pointer (FP)** and the **stack pointer (SP)** are set to the initial address of the stack.

As a stack-based language, eJVM makes heavy use of this memory section. eJVM programs do not make any assumption as to where in memory the stack is located—it is the responsibility of the interpreter to handle the initial stack placement and manage it during execution. However, it is encouraged to place the stack at the end of the memory assigned to the program, as is usually done in real machines. The stack, in that case, grows towards smaller memory addresses.

The stack contains a stack frame for every method invoked. That includes the initial invocation of the `main()` method, the entry point expected to be provided by every eJVM program (see section 2.2.5, paragraph 4). It is expected that execution of a program ends once the initial invocation of `main()` returns.

A stack frame is defined as the part of memory between the frame pointer at its highest and the stack pointer at its lowest memory address, signalising the bottom and top of said stack frame (in that order). The frame contains four or five fields, depending on the existence of method parameters and local variables. Table 4.1 outlines these fields.

Name	Length	Description
mindex	1	The index of the method this stack frame was constructed for in the method table
oldPC	4	The program counter pointing to the next instruction to execute once the method for which this stack frame was constructed returns
oldFP	4	The frame pointer for the previous stack frame
vstack	0 to 510	Optional section containing the method parameters and local variables, the latter being initialized with `0`
wstack	varying	The stack of working values

Table 4.1: Stack frame structure

The entirety of `mindex`, `oldPC` and `oldFP` may also be referred to as 'stack frame preamble'.

As mentioned by section 3.2, method parameters and local variables are stored in the same stack frame element, namely `vstack`, with method parameters being placed in front of local variables. If needed, method parameters can be distinguished from local variables by checking if the index of the accessed variable is between 0 and the number of method parameters as defined by the method information (section 3.2).

Working values are the values placed upon or taken from the stack by instructions. For most instructions, they are the only operands available (with the exception of instructions loading or modifying constants or variables in the `vstack` segment, like LDC or ILOAD). Every time a new working value is placed onto the stack, the stack pointer's value decreases by two byte (when using a downwards-growing stack); every time a working value is removed from the stack, the stack pointer increases value by two byte. The size of the `wstack` segment in table 4.1 has therefore been given as 'varying'.

Every time the INVOKEVIRTUAL instruction is executed by the interpreter, a new stack frame is added to the stack. To achieve this, several actions have to be undertaken: First, the interpreter takes the parameters the invoked method expects from the current stack frame's working values and stores them for later use. Then the new frame is added by pushing the index byte for the invoked method, the current program counter and frame pointer onto the stack, followed by the formerly stored parameters in the correct order (as declared). Afterwards, a two-byte value equal to `0` is pushed onto the stack for every local variable the invoked method declares. At last, frame pointer and stack pointer are set to their new values (FP to the old value of SP before the execution of the INVOKEVIRTUAL instruction, SP to the top of the `vstack`). No working values are placed on the new stack frame by this operation.

Removing a stack frame is done by simply restoring the old values of PC, FP and SP. The stack pointer can be restored by assigning the current value of the frame pointer to it, while the old values of PC and FP have been stored in the stack frame preamble. Removing any value placed upon the stack during method execution is not necessary, as those values are no longer accessible and will be overwritten by the next method invocation. If the method returns a return value, it has to be taken from the `wstack` segment before the stack frame is removed and pushed onto the stack after removal.

The regular end of execution is reached once the initial invocation of the `main()` method returns. The `oldFP` field can be used to detect this by defining a special, otherwise unused value as `oldFP` value for the very first invocation of `main()`. Once a program returns to this value, execution may be safely terminated. When using a stack as encouraged (that is, set at the highest memory address and growing downwards), the highest memory address is a good choice for this special value.

An irregular end of execution may be reached by execution of the ERR and HALT instructions or if the interpreter encounters an unexpected situation (such as an insufficient amount of working values to retrieve parameters for an invoked method from). In case of the ERR and HALT instructions, the execution is to be stopped immediately without any modification to memory, with ERR additionally printing out the error message it received as its argument. Behaviour upon unexpected situations is not defined, but it is encouraged to avoid any modification to stack or program (to allow them to be evaluated) and to provide the user with a meaningful error message.