Research Article

Tomaž Dobravec*

Selected tools for Java class and bytecode inspection in the educational environment

https://doi.org/10.1515/comp-2020-0170
Received Feb 28, 2020; accepted Apr 30, 2020

Abstract: Java is not only a modern, powerful, and frequently used programming language, but together with Java Virtual Machine it represents a novel dynamic approach of writing and executing computer programs. The fact that Java programs are executed in a controlled environment has several important implications that define the nature of the language and makes it different from the traditional C-like languages. Knowing the detailed differences between the two types of languages and execution environments is a part of the holistic education of a computer engineer.

In this paper, we present some behind-the-scene details about the Java Virtual Machine and we show how these details could be used in the educational process to demonstrate the differences and to emphasise the advantages of the dynamic programming approach when compared to the static one. After presenting some information about class files and about the internal structure and operation of the Java Virtual Machine we demonstrate the usage of public domain programs that could be used in the educational process to put these theoretical concepts into practice.

Keywords: java class file, JVM behind-the-scenes, bytecode execution, Java educational tools

1 Introduction

Java is a modern, powerful, robust, secure and frequently used programming language. Its popularity reflects in several different-purpose programming language indices which place Java at the very top of the scoreboard [1, 2, 12]. Considering these facts it is not surprising that Java has became a very popular programming language in the industrial environment. Aside from the practical usage in industry, Java possesses several properties that makes it interesting from an educational point of view [4]. Its most significant pedagogical strength is in the fact that Java programs [7] are executed in a virtual environment (inside Java Virtual Machine [9]) and are therefore protected and controlled. The logic that is used in the Java Virtual Machine (JVM) is similar to the logic that it is used on a typical computer at the hardware level, that is, executing the bytecode on the Java Virtual Machine is similar to executing a machine code on a processor except that Java Virtual Machine has a great advantage in that it controls the programs on-the-fly and it can select which operations will be executed on the hosting hardware. From an educational point of view Java Virtual Machine can be considered as a software emulation of a powerful hardware machine and as such a great tool to present the challenges and the solutions in the computer building process. In order to be able to write efficient and reliable Java programs, it is very important to understand the whole life cycle of a program, which encompasses writing Java programs, translating from Java to bytecode and executing prepared bytecode in Java Virtual Machine environment. In most courses that teach Java as a programming language only the first part of this cycle is presented to the students, while the details about the behind-the-scenes behaviour of translated program inside the JVM are omitted. Since we think that these details are very important for a comprehensive understanding of the Java world, we introduce them in some of our courses. In this paper we will describe some of these mechanisms and logic there and present tools that can be used both by teachers to support the educational process and later by trained professionals to write efficient programs.

The rest of this paper is organized as follows. In Sections. 2 and 3 we present some details on the organization of the Java class file and provide some information about the internal structure and behaviour of the Java Virtual Machine. In Section 4 we list selected tools, namely, `javap`, `hexdump`, `ClassEditor`, `Bytecode Visualizer` and `PyJVMGUI`, that could be used to demonstrate previously presented mechanisms and concepts in

***Corresponding Author: Tomaž Dobravec:** Faculty of Computer and Information Science, University of Ljubljana, Slovenia; Email: tomaz.dobravec@fri.uni-lj.si

real (running) programs. In Section 5 we provide some thoughts on using the presented tools in the classroom and we conclude the paper with final remarks in Section 6.

2 Java class files

In a traditional C-oriented static computer world the program-manufacturing process consists of the following steps: write the program source modules, compile each source module into the object module and finally to link all the object modules into an executable program. When such a simplified program (that does not use shared library objects) is executed the operating system expects that it is complete with all of its parts being linked and all the cross-usages being resolved. The only task to be performed in the loading stage is to relocate the direct addresses according to the given program load address and to start the execution. In JVM world his process is a bit different. To support the logic of dynamic program execution the demand that only completely linked and prepared programs can be executed was relaxed. Instead, JVM treats a program as a set of independent classes that are connected together on-the-fly in runtime only if and when this is really necessary. This means that the program-manufacturing process in the JVM world consists only of writing and compiling modules (*i.e.* classes), while the linking part of the process is postponed every time the program executes. This gives JVM an important role and makes the linking process very vulnerable and time consuming. To facilitate this process, additional data, packed in the so called constant pool, was added to a class file. A constant pool contains information about methods, fields and other types that are used in the linking process. Since the linking is performed only by using the symbol names, the content of a constant pool is vital in the class's living cycle.

Comparing the format of Java class file with the Executable and Linkable Format (ELF) object file format (which is used here as a representative of a standard object format in the static world) one can find several similarities and also some differences. Both formats provide headers with important information regarding the further content at the beginning. In both formats the content is written in chunks (arrays of data in Java class and sections or segments in ELF), but the file organization is different. While in the ELF format the section table reveals the location of each chunk, in the Java class file the chunks are encoded in such a way that one has to read the whole content of the previous chunk to find the beginning of the next one. Besides the constant pool, Java class file also contains information about interfaces, fields, methods and attributes. These parts of a class file could be compared with a symbol table and with sections (like .bss, .data, .txt) in the ELF object file. When talking about the methods section in a class file it is worth to mention that this section does not contain the machine code for a specific computer (as it is the case with ELF object files) but rather it contains bytecode, which is a machine language for the Java Virtual Machine.

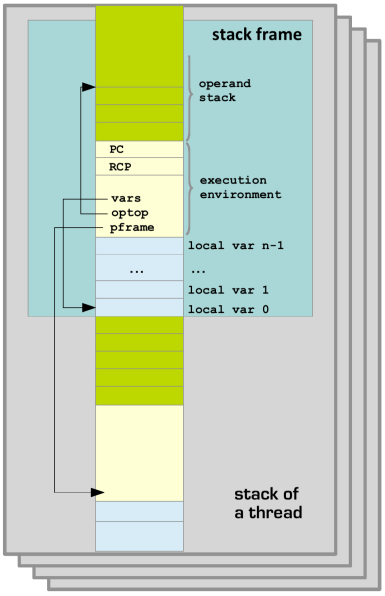

Figure 1: Schematic representation of the Java Virtual Machine stack organization

3 Inside the JVM

JVM is a stack-based virtual machine, which means that all the instructions are performed with the usage of the build-in stack – before the execution of an instruction the stack is used to store operands, after the execution it holds the value of the result. Besides the *stack*, which is for obvious reasons its most important part, JVM consists of three major components: the heap, the method area, and the registers [14]. The *heap* is a part of the virtual machine's memory where the space for object is allocated during the program execution. This part of the memory is controlled by the garbage collector which actively follows the references to allocated objects and occasionally clears-up all unused memory. The *method area* is a part of the memory, where the bytecode (compiled Java or other code) is stored. During the execution of the program, the JVM reads instruc-

tions from this area. A special thread-based counter, called the program counter, points to the location where the next instruction to be executed is stored. Although there are several objects of the same class active in a running program, the bytecode of their methods is written only once in the method area.

The JVM stack is a complex data structure used to store the information about the running program. To prevent misusage of this information and to ensure even distribution of memory usage, each thread gets its own independent stack. For each method that is executed within a thread, a frame is created on a top of the thread's stack. This frame is divided into three parts: execution environment, local variable array, and operand stack, as depicted in Figure 1.

To manipulate a running thread of a program, JVM uses only four *registers*, namely the `PC`, `frame`, `vars`, and `optop`, which point to the next instruction to be executed, to the execution environment of the current frame, to the array of local variables, and to the top of the operand stack, respectively. On the stack frame of a method being executed, there is a storage for local copies of these registers. The values of registers are stored when a sub-method is called (see Listings 1) and restored on return (see Listings 2).

Listings 1: Preparing the execution environment to support the call of a method `b()` from a method `a()`

1. create a `b_frame` on the top of the `a_frame`
2. save the values of the parameters passed to the method
   `b()` into the local variable array on `b_stack`
3. save `PC`, `optop`, `vars` to `a.PC`, `a.optop` and `a.vars`
4. set `b.pframe = frame`
5. `optop = b.optop`
6. `vars = b.vars`
7. `frame = b.frame`
8. `PC = address of the first instr. of the method b()`

Listings 2: Restoring the execution environment just before the return from the method `b()`

1. `b.local_var_0 = result`
2. `frame = b.pframe`
3. `vars = frame.vars`
4. `PC = frame.PC`
5. `optop = frame.optop++`

Note that the result of the execution of the method `b()` is stored in the `b()`'s first local variable, which resides on the bottom of the `b()` stack. Since the `b_frame` was placed on the top of the `a_frame`, the result is actually stored on the top of the `a()`'s operand stack and it can be simply `pop`ed to a local variable after the flow is returned to method `a()`.

Listings 3: A demo Java program

```
public class Sum {
  static       int multiplier = 42;
  static final int divider    = 21;

  int sumEven(int[] tab) {
    int s = 0;
    for (int i = 0; i < tab.length; i++)
      if (tab[i] \% 2 == 0)
        s += tab[i];
    return s;
  }

  public static void main(String[] args) {
    int[] numbers = {5,10,201,32769};
    Sum sum = new Sum();
    int c = sum.sumEven(numbers);
    c = c * multiplier / divider;

    System.out.println(c);
  }
}
```

The JVM logic of preparing and restoring the execution environment (especially the parameter- and result-passing part of it) very much resembles the mechanism that has been frequently and for a very long time used in the IA-32-like computers, where the `ESP` and `EBP` registers play the role of the frame pointers in a sense. But JVM didn't just adopt this logic, it has made a major step forward. Since the bytecode instructions are aware of the local variable array, they fetch their operands directly from there. For example, to push the integer value of the first local variable on the stack or to pop a float value from the stack and store it to the second local variable, the instructions `iload_0` and `fstore_1` are used, respectively. This simplification of the local variable usage not only makes the program more readable and concise but it also has a great impact on the execution efficiency.

4 The tools

In the following we present different tools that could be used to reflect, inspect, debug and/or manipulate Java bytecode. All the tools are publicly available and can freely be used in both the industrial and educational environment. To demonstrate the features of the program we will use a simple Java program (see Listings 3) that contains two static fields (`multiplier` and `divider`), a method `sumEven()` and a static method `main()`.

The first two of the presented programs, namely `hexdump` and `javap`, are simple command-line programs and are used mainly to browse the content of the class files, while the others (`Class Editor` [10], Bytecode Visualizer [3] and PyJVMGUI [6]) are programs with graphical user interface (GUI) that allow you to modify the class file and/or provide an interaction with a Java program during the execution of a bytecode.

4.1 The `hexdump` program

The `hexdump` program is used to present the pure binary content of a class file displayed as hexadecimal values. By combining the hexadecimal content with the ASCII character representation of a file (*e.g.* `hexdump -C Sum.class`), one can observe that a large part of a class file contains symbols and other textually represented data (constant pool table). The `hexdump` program can also be used to obtain some useful information about the class file. For example, since the 6th and 7th byte of a class file contain the major version of the `javac` compiler that was used to compile the source file, this information can be printed out by executing the following command: `hexdump -s 6 -n 2 Sum.class`.

```
$ hexdump -C Sum.class
00000000  ca fe ba be 00 00 00 33  00 2a 0a 00 09 00 1b 03  |.......3.*......|
00000010  00 00 80 01 07 00 1c 0a  00 03 00 1b 0a 00 03 00  |................|
00000020  1d 09 00 03 00 1e 09 00  1f 00 20 0a 00 21 00 22  |.......... ..!."|
00000030  07 00 23 01 00 0a 6d 75  6c 74 69 70 6c 69 65 72  |..#...multiplier|
00000040  01 00 01 49 01 00 07 64  69 76 69 64 65 72 01 00  |...I...divider..|
00000050  0d 43 6f 6e 73 74 61 6e  74 56 61 6c 75 65 03 00  |.ConstantValue..|
00000060  00 00 15 01 00 06 3c 69  6e 69 74 3e 01 00 03 28  |......<init>...(|
00000070  29 56 01 00 04 43 6f 64  65 01 00 0f 4c 69 6e 65  |)V...Code...Line|
00000080  4e 75 6d 62 65 72 54 61  62 6c 65 01 00 07 73 75  |NumberTable...su|
00000090  6d 45 76 65 6e 01 00 05  28 5b 49 29 49 01 00 0d  |mEven...([I)I...|
```

Figure 2: A part of the `Sum.class` file as printed by `hexdump`

An example of output produced by `hexdump` is presented in Figure 2. The colored parts of the output represent magic number (red), minor `javac` version (blue), major `javac` version (green), number of entries in the constant pool (yellow) and a part of the constant pool table (purple).

4.2 The `javap` program

Another quick but a little bit deeper inspection of the class file content can be performed by the `javap` program, which is a part of the standard Java Development Kit (JDK) distribution. By executing `javap -c -v ClassName`, a plethora of information about the class file, *i.e.*, the version of `javac` that was used to compile the class (major and minor version), MD5 checksum, the name of the source file, constant pool values and program bytecode listings are obtained. Figure 3 shows a selected part of a constant pool of the `Sum` class. The usage of constant pool can be seen in the bytecode that is also listed by `javap`.

```
                    Constant Pool
-----+-------------+----------------------
 idx | tag         | info
-----+-------------+----------------------
   1 | Class       | name=2
   2 | Utf8        | len=3; "Sum"
  ...
   5 | Utf8        | len=10; "multiplier"
   6 | Utf8        | len=1;  "I"
   7 | Utf8        | len=7;  "divider"
   8 | Utf8        | len=13; "Constant"
   9 | Integer     | bytes=21
  ...
  22 | Utf8        | len=7;  "sumEven"
  23 | Utf8        | len=5; bytes="([I)I"
  ...
  31 | Integer     | bytes=32769
  32 | Method      | class=1; nat=33
  33 | NAT         | name=22; desc=23
  ...
```

Figure 3: Selected parts of the constant pool for the `Sum` class

For example, the bytecode that is generated from the line 16 of Listings 3 (`int c = sum.sumEven(numbers);`) contains four instructions, namely `aload_2`, `aload_1`, `invokevirtual 32` and `istore_3`. The first two instructions load the reference values of the local variables 2 and 1 (*i.e.* `sum` and `numbers`) on the stack, the third instruction calls method 32 and the last instruction stores the result to the local variable number 3 (*i.e.* `c`). To decode the

meaning of the constant 32 (*i.e.* to find out which method is called by the third instruction), we could use the constant pool presented in Figure 3. The 32nd entry in the constant pool is a MethodReference that points to the 1st and the 33rd entry, which are the class name (*i.e.* `Sum`) and the NameAndType reference. The latter points again to the constant pool to the 22nd and 23rd entry, which contain the method's name and descriptor. Using the constant pool we found out that instruction `invokevirtual 32` will invoke the method `Sum.sumEven`, which receives an array of integers and returns an integer.

Another interesting piece of information that can be derived from the bytecode is that JVM uses three different instructions to store the values in a newly created array, where the instruction used depends on the value of the integer being stored in array. In particular, to store the values 10, 210 and 32769 to the `numbers` array, JVM uses instructions `bipush 10`, `sipush 210` and `ldc 31`, where 31 is an index to a constant pool entry, where the constant 32769 is stored. The difference between the first and the second instruction is that the first is encoded with two and the second with three bytes.

The listed bytecode and constant pool also reveals the difference between final and non-final static variables. The multiplication with a non-final static variable is compiled into bytecode as `getstatic 13; imul` (*i.e.*, load the value of an static field 13 (`multiplier`) to the stack and multiply with the other value on the stack), while the division with final static variable is compiled as `bipush 21; idiv`. It is obvious that the usage of final variables gives the compiler the opportunity to create more optimized code.

4.3 Class Editor

`Class Editor` is a program written by K. M. Tanmay that enables you to view or edit strings, attributes, methods and generate readable reports about the class files [10]. The main advantage of the `Class Editor` comparing to the `javap` program is in the interactivity and in ability not only to view but also to change the content of the class file. The interface of the program is simple but intuitive and therefore very easy to learn.

With its rich user interface program offers lots of possibilities to play with a class file, which include: searching for a string (in symbols, names, literals, ...), viewing the constant pool table as a tree structure (which gives a fine and transparent overview of the table), changing the content of the class file, and the like. Since the `Class Editor` understands the internal structure of the class file and it always tries to maintain the validity of the content, one could

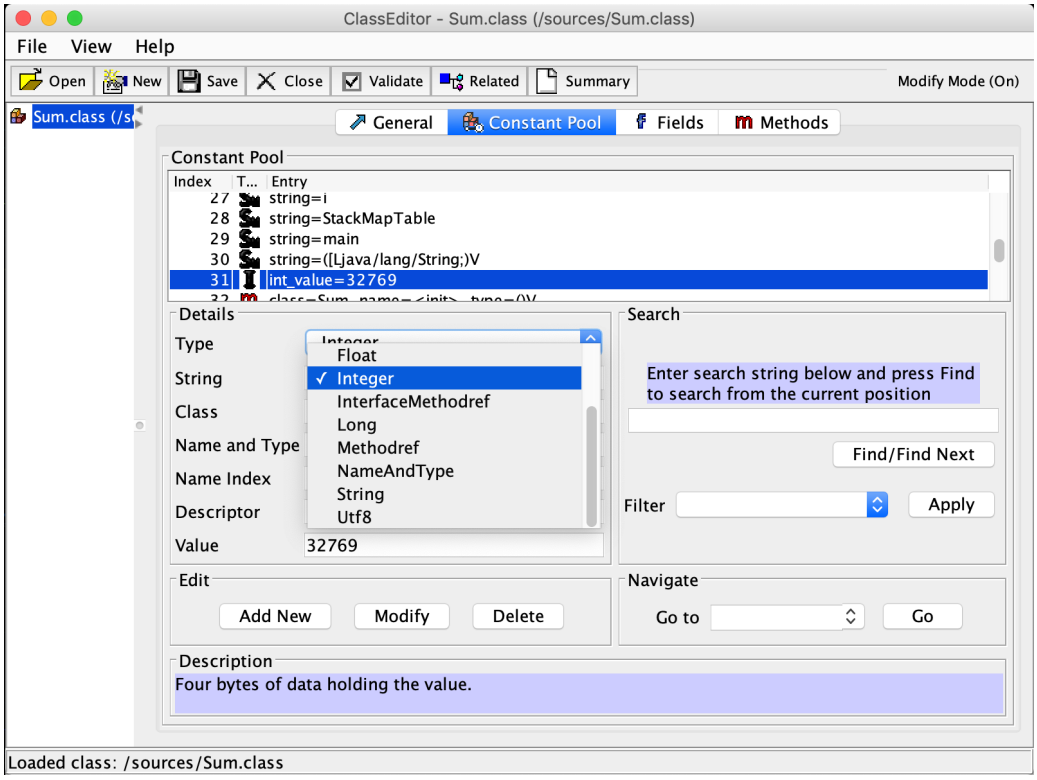

Figure 4: The Class Editor program in action with a part of the constant pool table of the `Sum.class` presented in the center

use this program to inspect the impact of selected changes to the content of the so generated file. For example, since the constants of types `int` and `long` differ in the number of slots used in the constant pool (the former needs only one slot, while the latest requires two), one can inspect the changes made in the constant pool after the change of a type of the constant as depicted in Figure 4.

4.4 Dr. Garbage's Bytecode Visualizer

Java Bytecode Visualizer (JBV) [3] is a tool used for visualizing and debugging Java bytecode. It is implemented as an Eclipse plugin which means that it can be used as additional view for the source files of the Java Eclipse project. JBV offers two additional windows (see Figure 5) in which it displays bytecode listings accompanied by labels of source code lines and an outline of a flowchart of the entire program. Using all the information provided by JBV, a user can debug a program and simultaneously compare source code with generated bytecode and observe the position of a program counter inside the flowchart.

```
int sumEven(int[] s) {
    /* L7 */
    0 iconst_0;
    1 istore_2;
    /* L8 */
    2 iconst_0;
    3 istore_3;
    4 goto 20;
    /* L9 */
    7 aload_1;
    8 iload_3;
    9 iaload;
    10 iconst_2;
    11 irem;
    12 ifne 9;
    /* L10 */
    15 iload_2;
    16 aload_1;
    17 iload_3;
    18 iaload;
    19 iadd;
    20 istore_2;
    /* L8 */
    21 iinc 3 1;
    24 iload_3;
    25 aload_1;
    26 arraylength;
    27 if_icmplt -20;
    /* L12 */
    30 iload_2;
    31 ireturn;
}
```

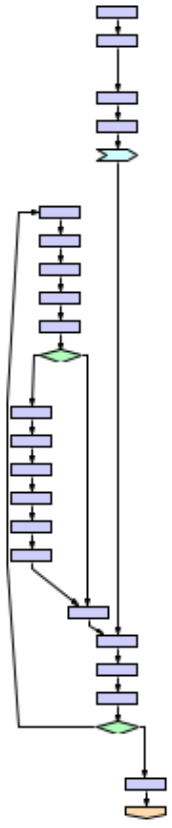

Figure 5: `Bytecode Visualizer` view to the `sumEven()` method

4.5 PyJVMGUI

Another very useful program used to demonstrate the dynamics of execution of Java programs is called PyJVMGUI [5, 6]. The program which is based on PyJVM framework [13] was designed by Matevž Fabjančič at Faculty of Computer and Information Science, University of Ljubljana, as an educational tool to be used at the System Software course. The main intent of the program is to show the three main components of the JVM that are used during the execution of the program. Namely, the main window of PyJVMGUI shows the following information: Java bytecode of the program (method) being executed, local variable array, operand stack and basic frame information. As an usage example of PyJVMGUI program let us observe the method `sumEven` of the `Sum` program. Figure 6 shows the content of PyJVMGUI window during the execution of this method. More precisely, in the left-side window, the bytecode of `sumEven` method is presented. The flow of the program is stopped in line 14 just after the instruction `irem` was fetched. This line of bytecode is a part of the `if (tab[i] % 2 == 0)` Java clause in which we check if the array element is even.

The `irem` takes the top and the next-to-top elements from the operand stack and calculates the remainder of dividing the former with the later. In our case the stack contains elements 2 and 32769 (see the Operand stack window in Figure 6), which were pushed there by previous instructions `iaload` (which pushed an array element) and `iconst_2` (which pushed number 2). The `if` clause being observed is a part of a `for` loop, which is driven by the counter `i`. The method `sumEven` has one formal parameter (`tab`) and two local variables (`s` and `i`). Since `sumEven` is not a static method, the first element of the local variable array represents a reference to the current object (`this`) followed by the formal parameters and the local variables (see the Locals & Args window in Figure 6). This makes the variable `i` the fourth element (*i.e.*, the element with the index 3) in this table. Current value of `i` is 3, which means that the program in our example is stopped in the last iteration of the `for` loop. Note that this observation is also compliant with the fact that the stack contains the number 32769, which is the last element of the `tab` array. After the execution of the `irem` the stack will contain the result of the operation (in this case 1, since the reminder of dividing 32769 by 2 is 1), which will be used by the following instruction (`ifne`) in line 15, which performs a 9 bytes

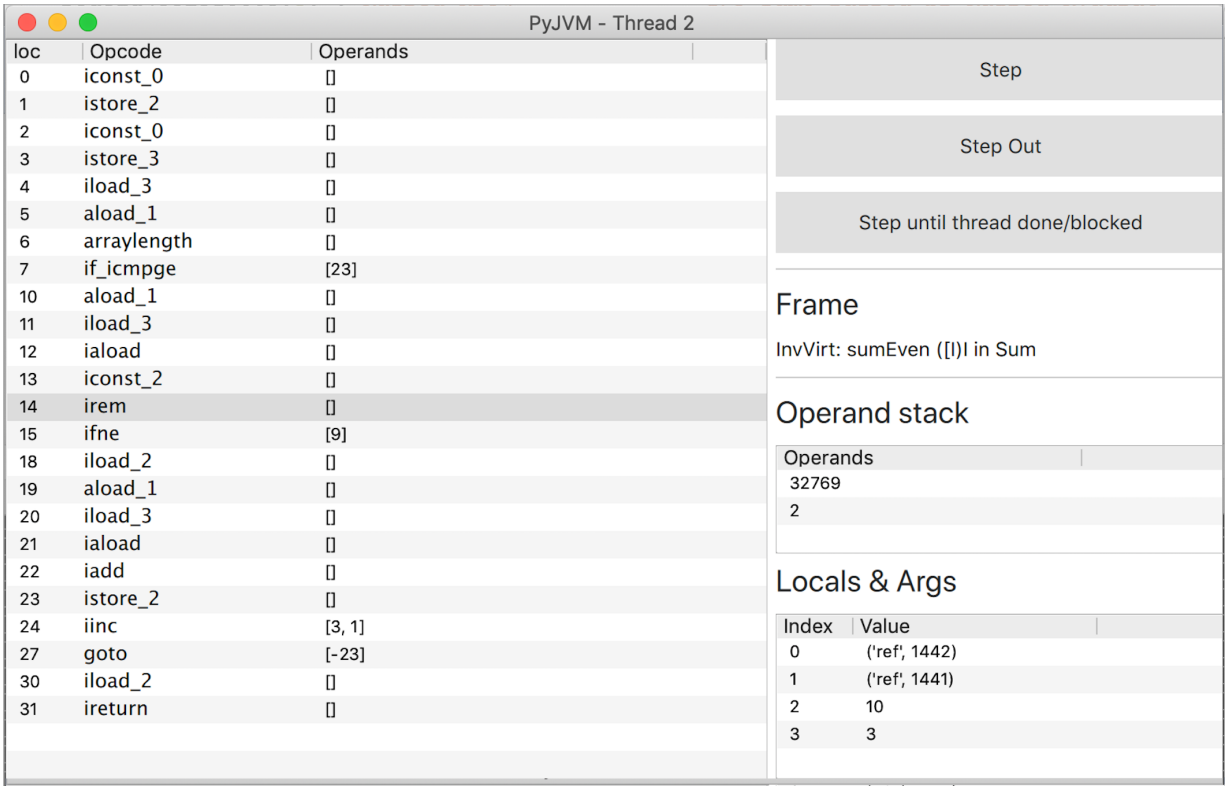

Figure 6: The main window of the `PyJVMGUI` program during execution of the `sumEven()` method

long jump (to line 27) if the first element on the stack is not zero. On the other hand, if the value is zero (*i.e.* if the division by 2 returned 0 as reminder, that is, if the number on stack is even), the instructions in line 18-24 will be executed. These instruction add the element value to the local variable with index 2. This short example shows how the information provided by PyJVMGUI can answer many question about the internal structure of the JVM. In our experience, the use of this program in the teaching process could greatly improve the understanding of the JVMs operation and virtual machines in general.

5 Java from a pedagogical point of view

For the last ten years, we have been teaching the backgrounds of the program's execution process as a part of the System Software course at the Faculty of Computer and Information Science in Ljubljana. Besides the extensive presentation of the static world (in which the programs to be executed are first compiled/assembled, then linked and finally loaded into the memory) we also present a more recent dynamic approach in which the linking phase is done just before the usage of the entity concerned. A very good example of this concept is used in the Java Virtual Machine, where the linking is performed on-the-fly by using the location values written in the Runtime Constant Pool. This JVM mechanism enables the usage of the programs in which not all the functionality is already present and it might be added later. To make the learning process more interesting and instructive, we have been using some dedicated tools [8, 11] for several years. These tools were developed especially for the needs of the System Software course and they help to present the behaviour of static world prograxms in the educational process. Our experiences show that the usage of these tools is a great encouragement for students that not only makes the learning process faster and more interesting but it also inspires and motivates students to dig into the lowest levels of computer architecture. Based on good experiences with the static world behaviour we wanted to provide a similar environment in the dynamic JVM world. Unfortunately, we found out that there are very few tools that could be used to present the dynamic execution of the bytecode in the virtual machine. The programs, that enable most of the desired functionality, are the `javap` and `Bytecode Visualizer`. They both present the bytecode of a given program and some accompanying parameters (like constant pool, method ta-

ble and the like). Even though the latest also enables dynamic monitoring of a program being executed (in debug mode), it does not show the internal JVM structure with the linking mechanisms. To bridge this gap, we have developed our own program (`PyJVMGUI`) that shows the behaviour of the JVM while executing the bytecode of a given program (*e.g.* the usage of methods stack, operand stack, local variable array, and JVM internal registers). Since this program was developed recently, we have not used it in the classroom yet, so we can not report the results on its usage. But according to our experiences with the SicSim [11] tool and based on the opinion of selected students that have already tested and evaluated the program, we can reasonably expect that the usage of this program in the educational process will encourage students to delve deeper into a very interesting world of the JVM and thus to gain and expand the knowledge of the computer logic at the lowest level.

6 Conclusions

In this paper, we present some behind-the-scenes information about the Java Virtual Machine with accompanying dynamic mechanisms and concepts, with the ultimate goal of impressing the reader with the usefulness of JVM as an educational tool. In the first part we present the structure of the class file and compare this structure with the ELF format from the static world. Next we present the internal functioning of the Java virtual Machine during the bytecode execution. Both, the content of the class file and the internal logic of the JVM, are very important in understanding the dynamic program execution process. In the second part of the paper we present several tools that we use in the educational process. All these tools are of great help to educators in bringing students closer to the inner world of Java execution and also of the computer operation at the lowest level.

References

[1] TIOBE Software BV. Tiobe index (june 2019). www.tiobe.com/tiobe-index, 2019

[2] Pierre Carbonnelle. Pypl popularity of programming language. pypl.github.io/PYPL.html, 2019

[3] Dr. Garbage Community. Dr. garbage tools. http://drgarbagetools.sourceforge.net, 2019

[4] Tomaž Dobravec. Java virtual machine educational tools. In William Steingartner, editor, *Proceedings of the Informatics 2019*, Piscataway, IEEE. cop,. Poprad, Slovakia, November 2019, 82–86

[5] Matevž Fabjančič. A GUI for the PyJVM. https://pypi.org/project/pyjvmgui, 2018

[6] Matevž Fabjančič. Simulating Java bytecode execution (in Slovene language). diploma thesis (mentor: T. Dobravec), Faculty of Computer and Information Science, University of Ljubljana, 2018

[7] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java™ Language Specification, Java SE 8 Edition*. Oracle America, Inc., California, USA, 2014

[8] Klemen Kloboves, Jurij Mihelič, Patricio Bulić, and Tomaž Dobravec. FPGA-based SIC/XE processor and supporting toolchain. *International journal of engineering education*, 2017, A(6), 1927–1939

[9] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java™ Virtual Machine Specification, Java SE 8 Edition*. Oracle America, Inc., California, USA, 2014

[10] Tanmay K. M. Java class file editor. http://classeditor.sourceforge.net, 2004

[11] Jurij Mihelič and Tomaž Dobravec. SicSim : a simulator of the educational SIC/XE computer for a system-software course. *Computer applications in engineering education*, 2015, 23(1), 137–146

[12] Stephen O'Grady. The redmonk programming language rankings: January 2019. https://redmonk.com/sogrady/2019/03/20/language-rankings-1-19, 2019

[13] Andrew Romanenco. PyJVM - Java 7 virtual machine implemented in pure python. https://github.com/andrewromanenco/pyjvm, 2014

[14] Bill Venners. The lean, mean, virtual machine. *JavaWorld*, 1996