# MASTERING JAVA BYTECODE AT THE CORE OF THE JVM

BY ANTON ARHIPOV, JREBEL PRODUCT LEAD

take a byte out of this code...

REBELLABS

# INRODUCTION

## WHAT YOU SHOULD KNOW

Whether you are a Java developer or architect, CxO or simply the user of a modern smart phone, Java bytecode is in your face, quietly supporting the foundation of the Java Virtual Machine (JVM).

Directors, executives and non-technical folks can take a breather here: All they need to know is that while their development teams are building and preparing to deploy the next amazing version of their software, Java bytecode is silently pumping through the JVM platform.

**Put simply,** Java bytecode is the intermediate representation of Java code (i.e. class files) and it is executed inside the JVM - so why should you care about it? Well, because you cannot run your entire development ecosystem without Java bytecode telling it all what to do, especially how to treat and define the code that Java developers are writing.

**From a technical POV**, Java bytecode is the code set used by the Java Virtual Machine that is JIT-compiled into native code at runtime. Without Java bytecode behind the scenes, the JVM would not be able to compile and mirror the non-bytecode Java code developers write to add new features, fix bugs and produce beautiful apps.

Many IT professionals might not have had the time to goof around with assembler or machine code, so Java bytecode can seem like an obscure piece of low-level magic. But, as you know, sometimes things go really wrong and understanding what is happening at the very foundation of the JVM may be what stands between you and solving the problem at hand.

In this RebelLabs report you will learn how to read and write JVM bytecode directly, so as to better understand how the runtime works, and be able to disassemble key libraries that you depend on.

In addition to getting the skinny on Java bytecode, we interviewed bytecode specialists **Cédric Champeau** and **Jochen Theodorou** working on the Groovy [1] ecosystem at SpringSource, and tech lead **Andrey Breslav** working on Kotlin [2] , a newcomer to the JVM language party, from JetBrains.

We will cover the following topics:

- How to obtain the bytecode listings
- How to read the bytecode
- How the language constructs are mirrored by the compiler: local variables, method calls, conditional logic
- Introduction to ASM
- How bytecode works in other JVM languages like Groovy and Kotlin

**So, get ready for your journey to the center of the JVM, and don't forget your compiler ;-)**

[1] : Groovy programming language http://groovy.codehaus.org

[2] : Kotlin programming language http://kotlin.jetbrains.org

# PART I

## GENTLE INTRODUCTION TO JAVA BYTECODE

Java bytecode is the form of instructions that the JVM executes. A Java programmer, normally, does not need to be aware of how Java bytecode works. However, understanding the low-level details of the platform is what makes you a better programmer after all (and we all want that, right?)

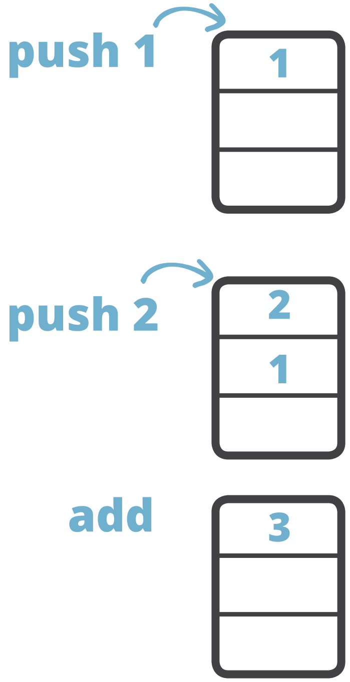
Understanding bytecode and what bytecode is likely to be generated by a Java compiler helps Java programmers in the same way that knowledge of assembly helps the C or C++ programmer [3] .

Understanding the bytecode, however, is essential to the areas of tooling and program analysis, where the applications can modify the bytecode to adjust the behavior according to the application's domain. Profilers, mocking frameworks, AOP -  to create these tools, developers must understand Java bytecode thoroughly.

[3] : Understanding bytecode makes you a better programmer
http://www.ibm.com/developerworks/ibm/library/it-haggar_bytecode

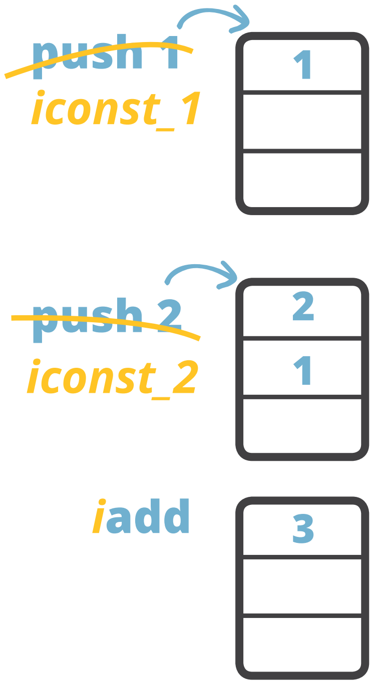
## **DOWN TO** THE BASICS

Lets start with a very basic example in order to understand how Java bytecode is executed. Consider a trivial expression, 1 + 2, which can be written down in reverse Polish notation as 1 2 +. Why is the reverse Polish notation any good here? It is easy to evaluate such expression by using a stack:



The result, 3, is on the top of the stack after the 'add' instruction executes.

The model of computation of Java bytecode is that of a stack-oriented programming language. The example above is expressed with Java bytecode instructions is identical, and the only difference is that the opcodes have some specific semantics attached:



The opcodes **iconst_1** and **iconst_2** put constants 1 and 2 to the stack. The instruction **iadd** performs addition operation on the two integers and leaves the result on the top of the stack.

## **GENERAL FACTS** ABOUT JAVA BYTECODE

As the name implies, Java bytecode consists of one-byte instructions, hence there are 256 possible opcodes. There are a little less real instructions than the set permits - approximately 200 opcodes are utilized, where some of the opcodes are reserved for debugger operation.

Instructions are composed from a type prefix and the operation name. For instance, 'i' prefix stands for 'integer' and therefore the iadd instruction indicates that the addition operation is performed for integers.

Depending on the nature of the instructions, we can group these into several broader groups:

1) **Stack manipulation instructions, including interaction with local variables.**
2) **Control flow instructions**
3) **Object manipulation, incl. methods invocation**
4) **Arithmetics and type conversion**

There are also a number of instructions of more specialized tasks such as synchronization and exception throwing.

## javap

To obtain the instruction listings of a compiled class file, we can apply the javap utility, the standard Java class file disassembler distributed with the JDK.

We will start with a class that will serve as an entry point for our example application, the moving average calculator.

```
public class Main {
    public static void main(String[] args){
        MovingAverage app = new MovingAverage();
    }
}
```

After the class file is compiled, to obtain the bytecode listing for the example above one needs to execute the following command: javap -c Main

The result is as follows:

```
Compiled from "Main.java"
public class algo.Main {
  public algo.Main();
        Code:
        0: aload_0
        1: invokespecial #1           // Method java/lang/Object."<init>":()V
        4: return

  public static void main(java.lang.String[]);
        Code:
        0: new           #2           // class algo/MovingAverage
        3: dup
        4: invokespecial #3           // Method algo/MovingAverage."<init>":()V
        7: astore_1
        8: return
}
```

As you can see there is a default constructor and a main method. You probably always knew that if you don't specify any constructor for a class there's still a default one, but maybe you didn't realize where it actually is. Well, here it is! We just proved that the default constructor actually exists in the compiled class, so it is java compiler who generates it.

The body of the constructor should be empty but there are a few instructions generated still. Why is that? Every constructor makes a call to super(), right? It doesn't happen automagically, and this is why some bytecode instructions are generated into the default constructor. Basically, this is the super() call;

The main method creates an instance of MovingAverage class and returns. We will review the class instantiation code in chapter 6.

You might have noticed that some of the instructions are referring to some numbered parameters with #1, #2, #3. This are the references to the pool of constants. How can we find out what the constants are and how can we see the constant pool in the listing? We can apply the **-verbose** argument to javap when disassembling the class:

```
$ javap -c -verbose HelloWorld
```

Here's some interesting parts that it prints:

```
Classfile /Users/anton/work-src/demox/out/production/demox/algo/Main.class
  Last modified Nov 20, 2012; size 446 bytes
  MD5 checksum ae15693cf1a16a702075e468b8aaba74
  Compiled from "Main.java"
public class algo.Main
  SourceFile: "Main.java"
  minor version: 0
  major version: 51
  flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
   #1 = Methodref          #5.#21         //  java/lang/Object."<init>":()V
   #2 = Class              #22            //  algo/MovingAverage
   #3 = Methodref          #2.#21         //  algo/MovingAverage."<init>":()V
   #4 = Class              #23            //  algo/Main
   #5 = Class              #24            //  java/lang/Object
```

Theres a bunch of technical information about the class file: when it was compiled, the MD5 checksum, which *.java file it was compiled from, which Java version it conforms to, etc.

We can also see the accessor flags there: ACC_PUBLIC and ACC_SUPER. The ACC_PUBLIC flag is kind of intuitive to understand: our class is public hence there is the accessor flag saying that it is public. But was is ACC_SUPER for? ACC_SUPER was introduced to correct a problem with the invocation of super methods with the invokespecial instruction. You can think of it as a bugfix to the Java 1.0 so that it could discover super class methods correctly. Starting from Java 1.1 the compiler always generates ACC_SUPER accessor flag to bytecode.

You can also find the denoted constant definitions in the constant pool:

```
 #1 = Methodref          #5.#21         //java/lang/Object."<init>":()V
```

The constant definitions are composable, meaning the constant might be composed from other constants referenced from the same table.

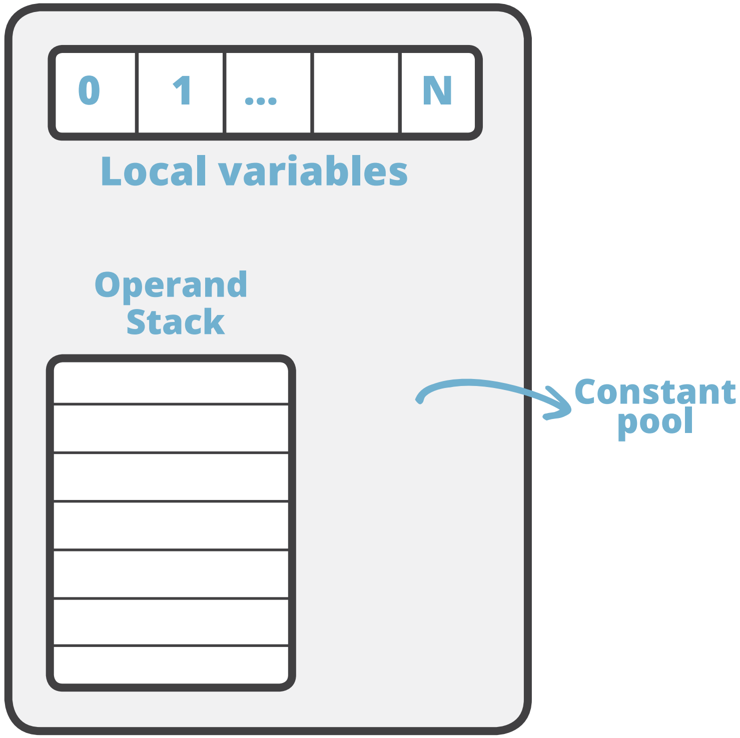
There are a few other things that reveal itself when using -verbose argument with javap. For instance there's more information printed about the methods:

```
public static void main(java.lang.String[]);
   flags: ACC_PUBLIC, ACC_STATIC
   Code:
     stack=2, locals=2, args_size=1
```

The accessor flags are also generated for methods, but we can also see how deep a stack is required for execution of the method, how many parameters it takes in, and how many local variable slots need to be reserved in the local variables table.

## The JVM as a stack machine

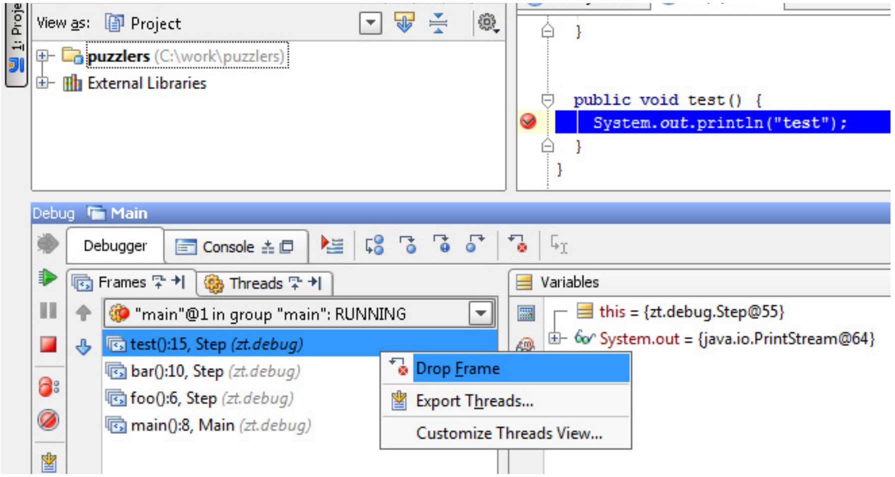
To understand the details of the bytecode, we need to have an idea of the model of execution of the bytecode. A JVM is a stack-based machine. Each thread has a JVM stack which stores frames. Every time a method is invoked a frame is created. A frame consists of an operand stack, an array of local variables, and a reference to the runtime constant pool of the class of the current method. We have seen all this in our initial example, the disassembled Main class.



The array of local variables, also called the local variable table, contains the parameters of the method and is also used to hold the values of the local variables. The size of the array of local variables is determined at compile time and is dependent on the number and size of local variables and formal method parameters.

The operand stack is a LIFO stack used to push and pop values. Its size is also determined at compile time. Certain opcode instructions push values onto the operand stack; others take operands from the stack, manipulate them, and push the result. The operand stack is also used to receive return values from methods.

In the debugger, we can drop frames one by one, however the state of the fields will not be rolled back.



```
    }

    public void test() {
        System.out.println("test");
    }
}
```

"main"@1 in group "main": RUNNING

test():15, Step (zt.debug)
bar():10, Step (zt.debug)
foo():6, Step (zt.debug)
main():8, Main (zt.debug)

this = {zt.debug.Step@55}
System.out = {java.io.PrintStream@64}

## What’s in the method body?

When looking at the bytecode listing from the HelloWorld example you might start to wonder, what are those numbers in front of every iinstruction? And why are the intervals between the numbers not equal?

```
0: new           #2       // class algo/MovingAverage
3: dup
4: invokespecial #3       // Method algo/MovingAverage."<init>":()V
7: astore_1
8: return
```

The reason: Some of the opcodes have parameters that take up space in the bytecode array. For instance, new occupies three slots in the array to operate: one for itself and two for the input parameters. Therefore, the next instruction - dup - is located at the index 3.

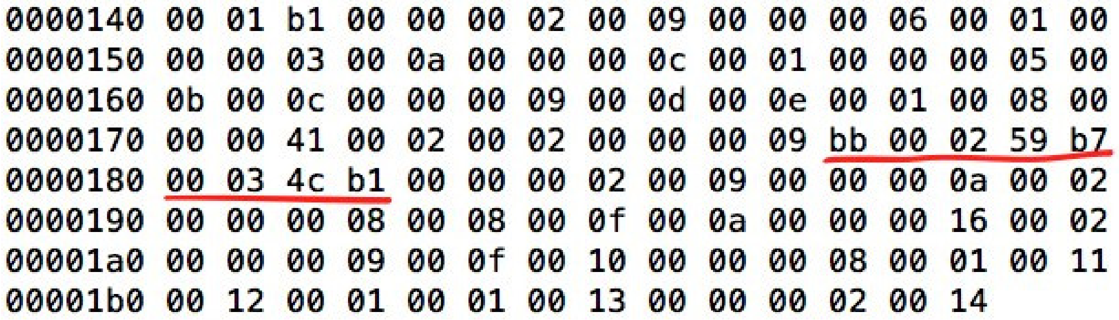
Here's what it looks like if we visualize the method body as an array:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| new | 00 | 02 | dup | invoke special | 00 | 03 | astore_1 | return |

Every instruction has its own HEX representation and if we use that we'll get the HEX string that represents the method body:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| ff | 00 | 02 | 59 | f7 | 00 | 03 | 4c | f1 |

By opening the class file in HEX editor we can find this string:



```
0000140 00 01 b1 00 00 00 02 00 09 00 00 00 06 00 01 00
0000150 00 00 03 00 0a 00 00 00 0c 00 01 00 00 00 05 00
0000160 0b 00 0c 00 00 00 09 00 0d 00 0e 00 01 00 08 00
0000170 00 00 41 00 02 00 02 00 00 00 09 bb 00 02 59 b7
0000180 00 03 4c b1 00 00 00 02 00 09 00 00 00 0a 00 02
0000190 00 00 00 08 00 08 00 0f 00 0a 00 00 00 16 00 02
00001a0 00 00 00 09 00 0f 00 10 00 00 00 08 00 01 00 11
00001b0 00 12 00 01 00 01 00 13 00 00 00 02 00 14
```

It is even possible to change the bytecode via HEX editor even though it is a bit fragile to do so. Besides there are some better ways of doing this, like using bytecode manipulation tools such as ASM or Javassist.

Not much to do with this knowledge at the moment, but now you know where these numbers come from.

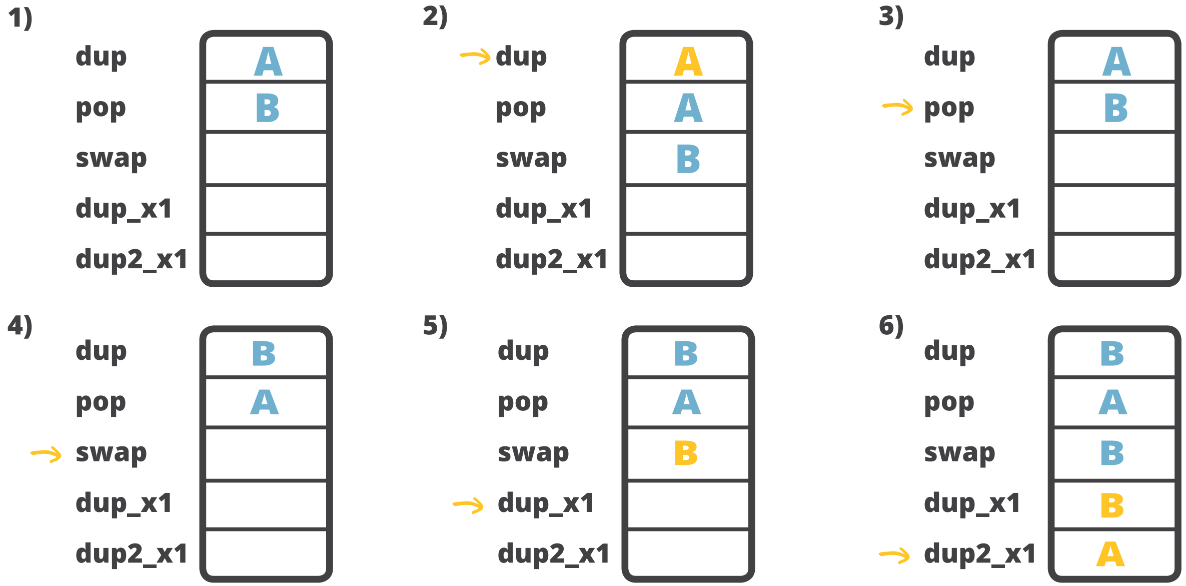
## What's in the method body?

There are a number of instructions that manipulate the stack in one way or another. We have already mentioned some basic instructions that work with the stack: push values to the stack or take values from the stack. But there's more; the swap instruction can swap two values on the top of the stack.

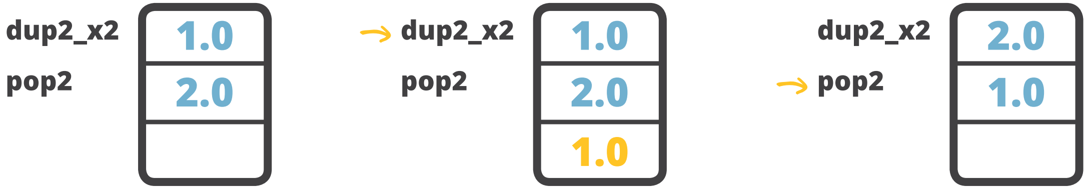
Here are some example instructions that juggle the values around the stack. Some basic instructions first: dup and pop. The dup instruction duplicates the value on top of the stack. The pop instruction removes the top value from the stack.

There are some more complex instructions: **swap, dup_x1 and dup2_x1,** for instance. The swap instruction, as the name implies, swaps two values on the top of the stack, e.g. A and B exchange positions (see example 4); **dup_x1** inserts a copy of the top value into the stack two values from the top (see example 5); **dup2_x1** duplicates two top values and inserts beneath the third (example 6).

**Examples:**



The **dup_x1** and **dup2_x1** instructions seem to be a bit esoteric - why would anyone need to apply such behavior - duplicating top values under the existing values in the stack? Here's a more practical example: how to swap 2 values of double type? The caveat is that double takes two slots in the stack, which means that if we have two double values on the stack they occupy four slots. To swap the two double values we would like to use the **swap** instruction but the problem is that it works only with one-word instructions, meaning it will not work with doubles, and swap2 instruction does not exist. The workaround is then to use **dup2_x2** instruction to duplicate the top double value below the bottom one, and then we can pop the top value using the **pop2** instruction. As a result, the two doubles will be swapped.



## Local variables

While the stack is used for execution, local variables are used to save the intermediate results and are in direct interaction with the stack.

**Let's now add some more code into our initial example:**

```
public static void main(String[] args) {
  MovingAverage ma = new MovingAverage();

  int num1 = 1;
  int num2 = 2;

  ma.submit(num1);
  ma.submit(num2);

  double avg = ma.getAvg();
}
```

We submit two numbers to the MovingAverage class and ask it to calculate the average of the current values. The bytecode obtained from this code is as follows:

```
Code:
     0: new           #2          // class algo/MovingAverage
     3: dup
     4: invokespecial #3          // Method algo/MovingAverage."<init>":()V
     7: astore_1

     8: iconst_1
     9: istore_2

     10: iconst_2
     11: istore_3

     12: aload_1
     13: iload_2
     14: i2d
     15: invokevirtual #4         // Method algo/MovingAverage.submit:(D)V

     18: aload_1
     19: iload_3
     20: i2d
     21: invokevirtual #4         // Method algo/MovingAverage.submit:(D)V

     24: aload_1
     25: invokevirtual #5         // Method algo/MovingAverage.getAvg:()D
     28: dstore     4
```

```
LocalVariableTable:
Start  Length  Slot  Name   Signature
       0       31      0  args    [Ljava/lang/String;
       8       23      1  ma      Lalgo/MovingAverage;
       10      21      2  num1    I
       12      19      3  num2    I
       30      1       4  avg     D
```

After creating the local variable of type MovingAverage the code stores the value in a local variable **ma**, with the **astore_1** instruction: 1 is the slot number of ma in the LocalVariableTable.

Next, instructions **iconst_1** and **iconst_2** are used to load constants 1 and 2 to the stack and store them in LocalVariableTable slots 2 and 3 respectively by the instructions **istore_2** and **istore_3**.

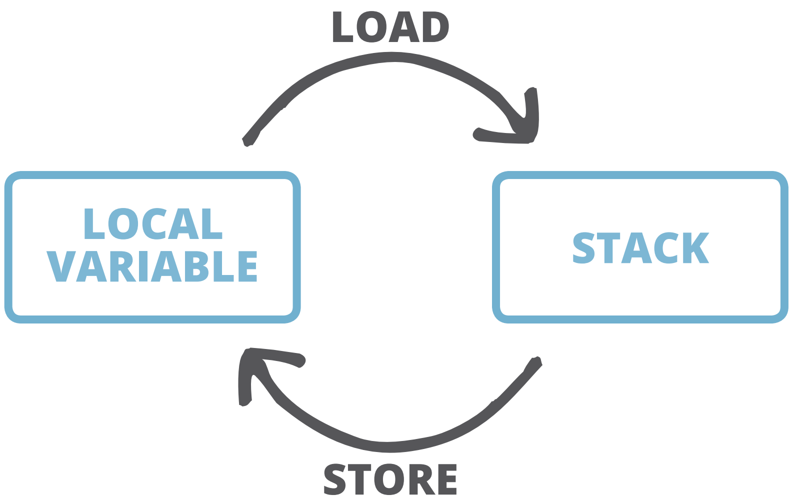
Note that the invocation of store-like instruction actually removes the value from the top of the stack. This is why in order to use the variable value again we have to load it back to the stack. For instance, in the listing above, before calling the submit method, we have to load the value of the parameter to the stack again:

```
12: aload_1
13: iload_2
14: i2d
15: invokevirtual #4 // Method algo/MovingAverage.submit:(D)V
```

After calling the getAvg() method the result of the execution locates on the top of the stack and to store it to the local variable again the dstore instruction is used since the target variable is of type double.

```
24: aload_1
25: invokevirtual #5 // Method algo/MovingAverage.getAvg:()D
28: dstore 4
```

One more interesting thing to notice about the LocalVariableTable is that the first slot is occupied with the parameter(s) of the method. In our current example it is the static method and there's no ***this*** reference assigned to the slot 0 in the table. However, for the non-static methods ***this*** will be assigned to slot 0.



The takeaway from this part is that whenever you want to assign something to a local variable, it means you want to **store** it by using a respective instruction, e.g. **astore_1**. The store instruction will always remove the value from the top of the stack. The corresponding **load** instruction will push the value from the local variables table to the stack, however the value is not removed from the local variable.

## Flow control

The flow control instructions are used to organize the flow of the execution depending on the conditions. If-Then-Else, ternary operator, various kinds of loops and even exception handling opcodes belong to the control flow group of Java bytecode. This is all about jumps and gotos now :)

We will now change our example so that it will handle an arbitrary number of numbers that can be submitted to the MovingAverage class:

```
MovingAverage ma = new MovingAverage();
for (int number : numbers) {
    ma.submit(number);
}
```

Assume that the numbers variable is a static field in the same class. The bytecode that corresponds to the loop that iterates over the numbers is as follows

```
0: new #2 // class algo/MovingAverage
3: dup
4: invokespecial #3 // Method algo/MovingAverage."<init>":()V
7: astore_1
8: getstatic #4 // Field numbers:[I
11: astore_2
12: aload_2
13: arraylength
14: istore_3
15: iconst_0
16: istore 4
18: iload 4
20: iload_3
21: if_icmpge 43
24: aload_2
25: iload 4
27: iaload
28: istore 5
30: aload_1
31: iload 5
```

```
33: i2d
34: invokevirtual #5       // Method algo/MovingAverage.submit:(D)V
37: iinc         4, 1
40: goto         18
43: return
 LocalVariableTable:
 Start  Length  Slot  Name   Signature
   30       7     5   number I
   12      31     2   arr$   [I
   15      28     3   len$   I
   18      25     4   i$     I
    0      49     0   args   [Ljava/lang/String;
    8      41     1   ma     Lalgo/MovingAverage;
   48    1        2   avg    D
```

The instructions at positions 8 through 16 are used to organize the loop control. You can see that there are three variables in the LocalVariableTable that aren't really mentioned in the source code: arr$, len$, i$ - those are the loop variables. The variable arr$ stores the reference value of the numbers field from which the length of the loop, len$, is derived using the arraylength instruction. Loop counter, i$ is incremented after each iteration using iinc instruction.

The first instructions of the loop body are used to perform the comparison of the loop counter to the array length:

```
18: iload 4
20: iload_3
21: if_icmpge 43
```

We load the values of i$ and len$ to the stack and call the if_icmpge to compare the values. The if_icmpge instruction meaning is that if the one value is greater or equal than the other value, in our case if i$ is greater or equal than len$, then the execution should proceed from the statement that is marked with 43. If the condition does not hold, then the loop proceeds with the next iteration.

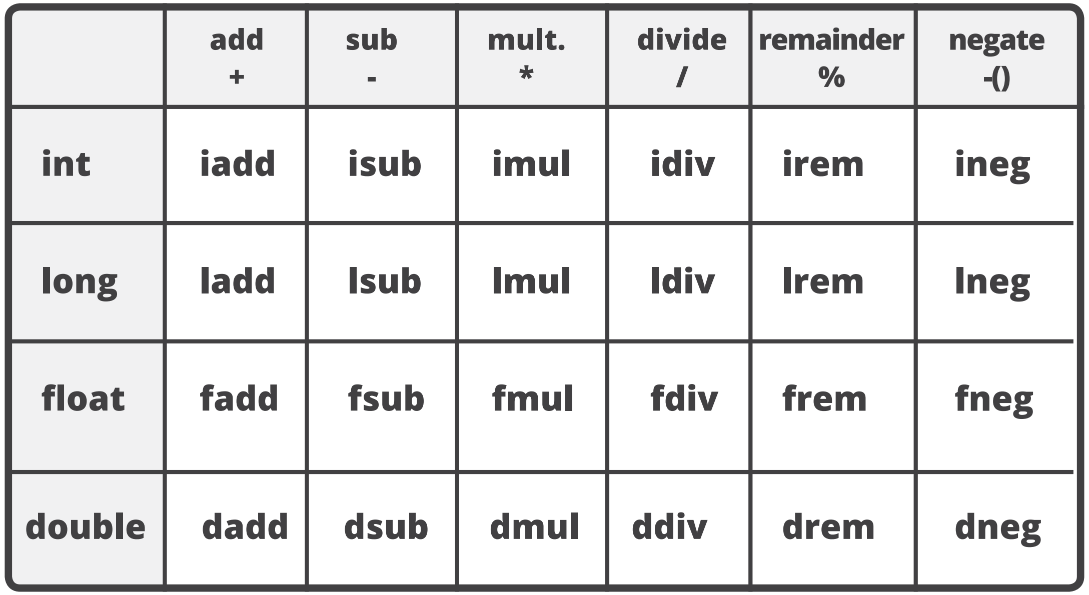
At the end of the loop it loop counter is incremented by 1 and the loop jumps back to the beginning to validate the loop condition again:

```
37: iinc          4, 1       // increment i$
40: goto       18         // jump back to the beginning of the loop
```

## Arithmetics & Conversion

As you have seen already, there's a number of instructions that perform all kind of arithmetics in Java bytecode. In fact, a large portion of the instruction set is denoted to the arithmetic. There are instructions of addition, subtraction, multiplication, division, negation for all kind of types - integers, longs, doubles, floats. Plus there's a lot of instructions that are used to convert between the types.

**Arithmetical opcodes and types**



| | add<br>+ | sub<br>- | mult.<br>* | divide<br>/ | remainder<br>% | negate<br>-() |
|---|---|---|---|---|---|---|
| int | iadd | isub | imul | idiv | irem | ineg |
| long | ladd | lsub | lmul | ldiv | lrem | lneg |
| float | fadd | fsub | fmul | fdiv | frem | fneg |
| double | dadd | dsub | dmul | ddiv | drem | dneg |

Type conversion happens for instance when we want to assign an integer value to a variable which type is long.

**Type conversion opcodes**

To

From

| | int | long | float | double | byte | char | short |
|---|---|---|---|---|---|---|---|
| int | - | i2l | i2f | i2d | i2b | i2c | i2s |
| long | l2i | - | l2f | l2d | - | - | - |
| float | f2i | f2l | - | f2d | - | - | - |
| double | d2i | d2l | d2f | - | - | - | - |

In our example where an integer value is passed as a parameter to submit() method which actually takes double, we can see that before actually calling the method the type conversion opcode is applied:

```
31: iload          5
 33: i2d
 34: invokevirtual #5      // Method algo/MovingAverage.submit:(D)V
```

It means we load a value of a local variable to the stack as an integer, and then apply i2d instruction to convert it into double in order to be able to pass it as a parameter.

The only instruction that doesn't require the value on the stack is the increment instruction, iinc, which operates on the value sitting in LocalVariableTable directly. All other operations are performed using the stack.

## new, <init> & <clinit>

There's a keyword new in Java but there's also a bytecode instruction called new. When we created an instance of MovingAverage class:

```
      MovingAverage ma = new MovingAverage();
```

 the compiler generated a sequence of opcodes that you can recognize as a pattern:

```
0: new #2 // class algo/MovingAverage
3: dup
4: invokespecial #3 // Method algo/MovingAverage."<init>":()V
```

When you see **new, dup** and **invokespecial** instructions together it must ring a bell - this is the class instance creation!

Why three instructions instead of one, you ask? The new instruction creates the object but it doesn't call the constructor, for that, the invokespecial instruction is called: it invokes the mysterious method called <init>, which is actually the constructor. The dup instruction is used to duplicate the value on the top of the stack. As the constructor call doesn't return a value, after calling the <init> method on the object the object will be initialized but the stack will be empty so we wouldn't be able to do anything with the object after it was initialized. This is why we need to duplicate the reference in advance so that after the constructor returns we can assign the object instance into a local variable or a field. Hence, the next instruction is usually one of the following:

**astore {N}** or **astore_{N}** - to assign to a local variable, where {N} is the position of the variable in local variables table.
**putfield** - to assign the value to an instance field
**putstatic** - to assign the value to a static field

While a call to <init> is a constructor invocation, there's another similar method, <clinit> which is invoked even earlier. This is the static initializer name of the class. The static initializer of the class isn't called directly, but triggered by one of the following instructions: new, getstatic, putstatic or invokestatic. That said, if you create a new instance of the class, access a static field or call a static method, the static initializer is triggered.

In fact, there is even more options to trigger the static initializer as described in the Chapter 5.5 of JVM specification [4]

[4] : JVM Specification http://docs.oracle.com/javase/specs/jvms/se7/html

## Method invocation and parameter passing

We have touched the method invocation topic slightly in the class instantiation part: the <init> method was invoked via invokespecial instruction which resulted in the constructor call. However, there are a few more instructions that are used for method invocation:

**invokestatic**, as the name implies, this is a call to a static method of the class. This is the fastest method invocation instruction there is.

**invokespecial** instruction is used to call the constructor, as we know. But it also is used to call private methods of the same class and accessible methods of the super class.

**invokevirtual** is used to call public, protected and package private methods if the target object of a concrete type.

**invokeinterface** is used when the method to be called belongs to an interface.

### *So what is the difference between invokevirtual and invokeinterface?*

Indeed a very good question. Why do we need both invokevirtual and invokeinterface, why not to use invokevirtual for everything? The interface methods are public methods after all! Well, this is all due to the optimization for method invocations. First, the method has to be resolved, and then we can call it. For instance, with invokestatic we know exactly which method to call: it is static, it belongs to only one class. With invokespecial we have a limited list of options - it is easier to choose the resolution strategy, meaning the runtime will find the required method faster.

With invokevirtual and invokeinterface the difference is not that obvious however. Let me offer a very simplistic explanation of the difference for these two instructions. Imagine that the class definition contains a table of method definitions and all the methods are numbered by position. Here's an example: class A, with methods method1 and method2 and a subclass B, which derives method1, overrides method2, and declares new method3. Note that method1 and method2 are at the same indexed position both in class A and class B.

```
class A
    1: method1
    2: method2

class B extends A
    1: method1
    2: method2
    3: method3
```

This means that if the runtime wants to call method2, it will always find it at position 2. Now, to explain the essential difference between invokevirtual and invokeinterface let's make class B to extend interface X which declares a new method:

```
class B extends A implements X
    1: method1
    2: method2
    3: method3
    4: methodX
```

The new method is at index 4 and it looks like it is not any different from method3 in this situation. However, what if theres another class, C, which also implements the interface but does not belong to the same hierarchy as A and B:

```
class C implements  X
    1: methodC
    2: methodX
```

The interface method is not at the same position as in class B any more and this is why runtime is more restricted in respect to invokinterface, meaning it can do less assumptions in method resolution process than with invokevirtual.

# PART II
# GETTING STARTED WITH ASM

**ObjectWeb ASM** is the *de-facto* standard for Java bytecode analysis and manipulation. ASM exposes the internal aggregate components of a given Java class through its visitor oriented API. The API itself is not very broad - with a limited set of classes you can achieve pretty much all you need. ASM can be used for modifying the binary bytecode, as well as generating new bytecode. For instance, ASM can be applied to implement a new programming language semantics (Groovy, Kotlin, Scala), compiling the high-level programming idioms into bytecode capable for execution in the JVM.

**ANDREY BRESLAV,** KOTLIN
We didn't even consider using anything else instead of ASM, because other projects at JetBrains use ASM successfully for a long time.

**JOCHEN THEODOROU,** GROOVY
My first touch with bytecode first hand was when I started helping in the Groovy project and by then we settled to ASM. ASM can do what is needed, is small and doesn't try to be too smart to get into your way. ASM tries to be memory and performance effective. For example you don't have to create huge piles of objects to create your bytecode. It was one of the first with support for invokedynamic btw. Of course it has its pro and con sides, but all in all I am happy with it, simply because I can get the job done using it.

**CÉDRIC CHAMPEAU,** GROOVY
I mostly know about ASM, just because it's the one used by Groovy :) However, knowing that it's backed by people like Rémi Forax, who is a major contributor in the JVM world is very important and guarantees that it follows the latest improvements.

To give you a very gentle introduction we will generate a “Hello World” example using the ASM library and add a loop to print the phrase an arbitrary number of times.

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println(“Hello, World!”);
    }
}
```

The most common scenario to generate bytecode that corresponds to the example source, is to create ClassWriter, visit the structure – fields, methods, etc, and after the job is done, write out the final bytes.

First, let’s construct the ClassWriter instance:

```
    ClassWriter cw = new ClassWriter(
        ClassWriter.COMPUTE_MAXS |
        ClassWriter.COMPUTE_FRAMES);
```

The ClassWriter instance can be instantiated with some constants that indicate the behavior that the instance should have. COMPUTE_MAXS tells ASM to automatically compute the maximum stack size and the maximum number of local variables of methods. COMPUTE_FRAMES flag makes ASM to automatically compute the stack map frames of methods from scratch.

The define a class we must invoke the visit() method of ClassWriter:

```
    cw.visit(
        Opcodes.V1_6,
        Opcodes.ACC_PUBLIC,
        "HelloWorld",
        null,
        "java/lang/Object",
        null);
```

Next, we have to generate the default constructor and the main method. If you skip generating the default constructor nothing bad will happen, but it is still polite to generate one.

```
  MethodVisitor constructor =
     cw.visitMethod(
           Opcodes.ACC_PUBLIC,
           "<init>",
           "()V",
           null,
           null);

  constructor.visitCode();


//super()
  constructor.visitVarInsn(Opcodes.ALOAD, 0);
  constructor.visitMethodInsn(Opcodes.INVOKESPECIAL,
     "java/lang/Object", "<init>", "()V");
  constructor.visitInsn(Opcodes.RETURN);

  constructor.visitMaxs(0, 0);
  constructor.visitEnd();
```

We first created the constructor using the visitMethod() method. Next, we indicate that we’re now about to start generating the body of the constructor by calling visitCode() method. At the end we call to visitMaxs() - this is to ask ASM to recompute the maximum stack size. As we indicated that ASM can do that for us automatically using COMPUTE_MAXS flag in ClassWriter’s constructor, we can pass in random arguments to visitMaxs() method. At last, we indicate that the generating bytecode for the method is complete with visitEnd() method.

Here’s what ASM code for main method looks like:

```
MethodVisitor mv = cw.visitMethod(
      Opcodes.ACC_PUBLIC + Opcodes.ACC_STATIC,
      "main", "([Ljava/lang/String;)V", null, null);

  mv.visitFieldInsn(Opcodes.GETSTATIC, "java/lang/System",
      "out", "Ljava/io/PrintStream;");
  mv.visitLdcInsn("Hello, World!");
  mv.visitMethodInsn(Opcodes.INVOKEVIRTUAL, "java/io/PrintStream",
      "println", "(Ljava/lang/String;)V");
  mv.visitInsn(Opcodes.RETURN);

  mv.visitMaxs(0, 0);
  mv.visitEnd();
```

By calling the visitMethod() again, we generated the new method definition with the name, modifiers and the signature. Again, visitCode(), visitMaxs() and visitEnd() methods are used the same way as in case with the constructor.

As you can see the code is full of constants, “flags” and “indicators” and the final code is not very fluently readably by human eyes. At the same time, to write such code one needs to keep in mind the bytecode execution plan to be able to produce correct version of bytecode. This is what makes writing such code rather a complicated task. This is where everyone has his own approach it writing code with ASM.

**ANDREY BRESLAV,** KOTLIN
Our approach is using Kotlin's ability to enhance existing Java APIs: we created some helper functions (many of them extension functions) that make ASM APIs look very much like a bytecode manipulation DSL.

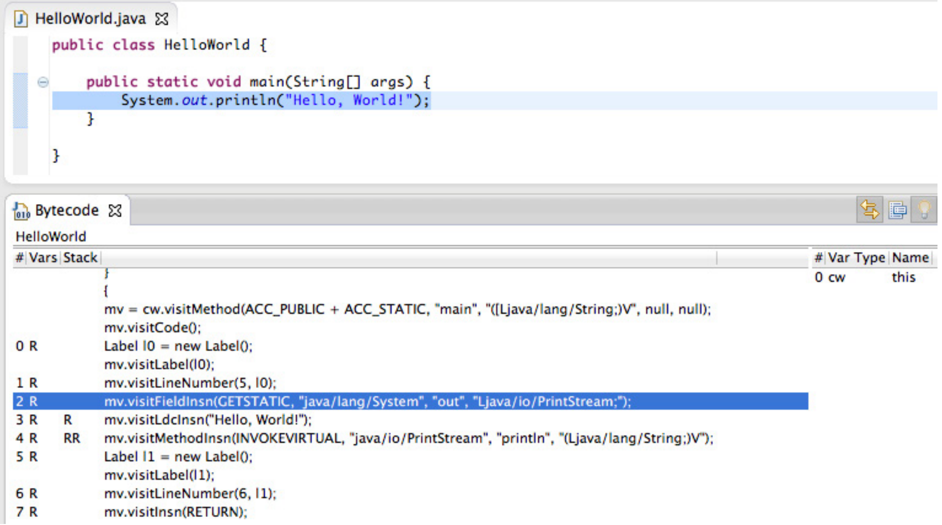
**JOCHEN THEODOROU,** GROOVY
I built some meta api into the compiler. For example it let's you do a swap, regardless of the involved types. It was not in the links above, but I assume you know, that double and long consume two slots, while anything else does only one. The swap instruction handles only the 1-slot version. So if you have to swap an int and a long, a long and an int or a long and a long, you get a different set of instructions. I also added a helper API for local variables, to avoid to have to manage the index.
If you want more nice looking code... Cedric wrote a Groovy DSL to generate bytecode. It is still the bytecode more or less, but less method around to make it less clear.

**CÉDRIC CHAMPEAU,** GROOVY
ASM is a nice low-level API, but I think we miss an up-to-date higher level API, for example for generating proxies and so on. In Groovy we want to limit the number of dependencies we add to the project, so it would be cool if ASM provided this out-of-the-box, but the general idea behind ASM is more to stick with a low level API.

## ASM and Tooling

The tools can be a great help for studying and working with bytecode. The best way to learn to use ASM is to write a Java source file that is equivalent to what you want to generate and then use the ASMifier mode of the Bytecode Outline plugin for Eclipse (or the ASMifier tool) to see the equivalent ASM code. If you want to implement a class transformer, write two Java source files (before and after transformation) and use the compare view of the plugin in ASMifier mode to compare the equivalent ASM code.

### Bytecode outline plugin view in Eclipse



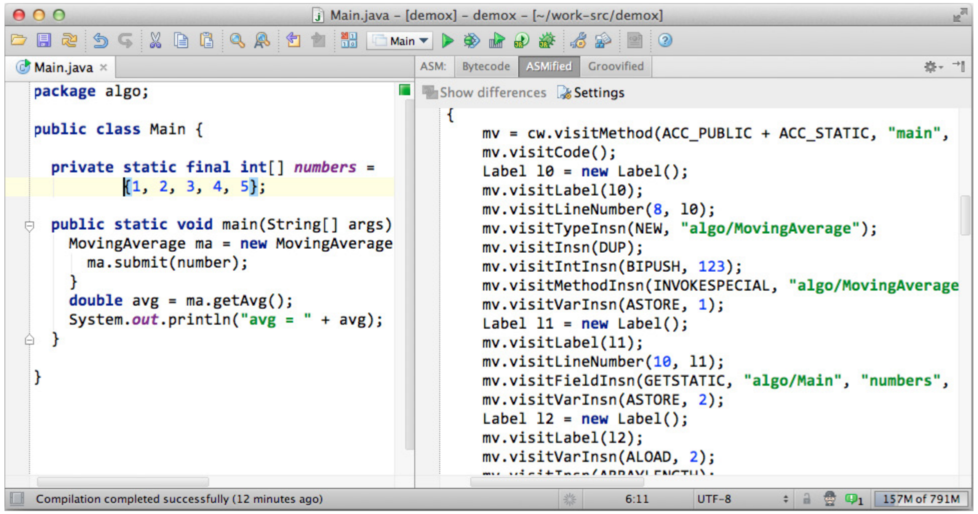
```
public class HelloWorld {

    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }

}
```

HelloWorld

```
# Vars Stack                                                                                  # Var Type Name
                 }                                                                            0 cw       this
                 {
                 mv = cw.visitMethod(ACC_PUBLIC + ACC_STATIC, "main", "([Ljava/lang/String;)V", null, null);
                 mv.visitCode();
0 R              Label l0 = new Label();
                 mv.visitLabel(l0);
1 R              mv.visitLineNumber(5, l0);
2 R              mv.visitFieldInsn(GETSTATIC, "java/lang/System", "out", "Ljava/io/PrintStream;");
3 R    R         mv.visitLdcInsn("Hello, World!");
4 R    RR        mv.visitMethodInsn(INVOKEVIRTUAL, "java/io/PrintStream", "println", "(Ljava/lang/String;)V");
5 R              Label l1 = new Label();
                 mv.visitLabel(l1);
6 R              mv.visitLineNumber(6, l1);
7 R              mv.visitInsn(RETURN);
```

For IntelliJ IDEA users there's the ASM bytecode outline plugin available in the plugins repository and it is quite easy to use too. Right click in the source and select Show Bytecode outline - this will open a view with the code generated by the ASMifier tool.

### ASM outline plugin in IntelliJ IDEA



```
package algo;

public class Main {

  private static final int[] numbers =
          {1, 2, 3, 4, 5};

  public static void main(String[] args)
    MovingAverage ma = new MovingAverage
      ma.submit(number);
    }
    double avg = ma.getAvg();
    System.out.println("avg = " + avg);
  }

}
```

```
{
    mv = cw.visitMethod(ACC_PUBLIC + ACC_STATIC, "main",
    mv.visitCode();
    Label l0 = new Label();
    mv.visitLabel(l0);
    mv.visitLineNumber(8, l0);
    mv.visitTypeInsn(NEW, "algo/MovingAverage");
    mv.visitInsn(DUP);
    mv.visitIntInsn(BIPUSH, 123);
    mv.visitMethodInsn(INVOKESPECIAL, "algo/MovingAverage
    mv.visitVarInsn(ASTORE, 1);
    Label l1 = new Label();
    mv.visitLabel(l1);
    mv.visitLineNumber(10, l1);
    mv.visitFieldInsn(GETSTATIC, "algo/Main", "numbers",
    mv.visitVarInsn(ASTORE, 2);
    Label l2 = new Label();
    mv.visitLabel(l2);
    mv.visitVarInsn(ALOAD, 2);
```

Compilation completed successfully (12 minutes ago)

You can also apply the ASMifier directly, without the IDE plugin, as it is a part of ASM libabray:

```
$java -classpath "asm.jar;asm-util.jar" \
       org.objectweb.asm.util.ASMifier \
       HelloWorld.class
```

**ANDREY BRESLAV,** KOTLIN

We use ASM bytecode outline for IntelliJ IDEA and our own similar plugin that displays bytecodes generated by our compiler.

**CÉDRIC CHAMPEAU,** GROOVY

Actually, I wrote the "bytecode viewer" plugin for IntelliJ IDEA, and I'm using it quite often :) On the Groovy side, I also use the AST browser view, which provides a bytecode view too, although it seriously needs improvements.

**JOCHEN THEODOROU,** GROOVY

My tools are mostly org.objectweb.asm.util.Textifier and org.objectweb.asm.util.CheckClassAdapter. Some time ago I also wrote a tool helping me to visualize the bytecode and the stack information. It allows me to go through the bytecode and see what happens on the stack. And while bytecode used to be a pita to read for me in the beginning, I have seen so much of it, that I don't even use that tool anymore, because I am usually faster just looking at the text produced by Textifier.

That is not supposed to tell you I am good at generating bytecode... no no.. I wouldn't be able to read it so good if I had not the questionable pleasure of looking at it countless times, because there again was a pop of an empty stack or something like that. It is more that the problems I have to look for tend to repeat themselves and I have a whit of what to look for even before I fire up Textifier

## Fun stories from bytecode experts

We asked Andrey, Jochen and Cédric to share some fun facts from their experiences with Java bytecode. While the words “bytecode” and “fun” might not stick very well together there are still cases to learn from and the guys warmly share the experiences:

**JOCHEN THEODOROU,** GROOVY

Hmm... bytecode and fun? What a strange combination of words in the same sentence ;)

Well.. one time maybe a little... I told you about the API I use to do a swap. In the beginning it was not working properly of course. That was partially due to me misunderstanding one for those DUP instructions, but mainly it was because I had a simple bug in my code in which I execute the 1-2 swap instead of the 2-1 swap (meaning swapping 1 and 2 slot operands). So I was looking at the code, totally confused, thinking this should work, looking at my code... then thinking I made it wrong with those dups and replacing the code with my new understanding...

All the while the code was not really all that wrong, only the swap cases where swapped. Anyway... after about a full day of getting a headache from too much looking at the bytecode I finally found my mistake and looked at the code to find it looks almost the same as before... and then it dawned on me, that it was only that simple mistake, that could have been corrected in a minute and which took me a full day. Not really funny, but there I laughed a bit at myself actually.

**CÉDRIC CHAMPEAU,** GROOVY

Actually, the funniest thing was when I wrote the "bytecode DSL" for Groovy, which allows you to write bytecode directly in the body of a method, using a DSL which is very close to what the ASM outline provides, and a nicer "groovy flavoured" DSL too. Although I started this project as a proof-of-concept and a personal experiment, I received a lot of feedback and interest about it.

Today I think it's a very simple way to have people test bytecode directly, for example for students. It makes writing bytecode a lot easier than using ASM directly. However, I also received a lot of complains, people saying I opened the Pandora box and that it would produce unreadable code in production :D (and I would definitely not recommend using it in production). Yet, it's been more than one year the project is out, and I haven't heard of anyone using it, so probably bytecode is really not that fun!

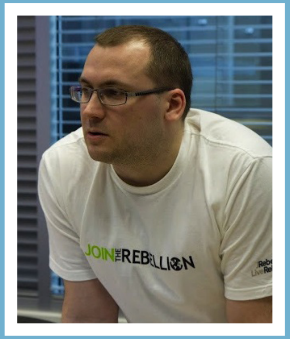
**ANDREY BRESLAV,** KOTLIN

Many fun things come in connection with Android: Dalvik is very picky about your bytecode conformance to the JVM spec. And HotSpot doesn't care a bit about many of these things. We were running smoothly on HotSpot for a long time, without knowing that we had so many things done wrong. Now we use Dalvik's verifier to check every class file we generate, to make sure nobody forgot to put ACC_SUPER on a class, proper offsets to a local variable table, and things like that.

We also came across a few interesting things in HotSpot, for example, if you call an absent method on an array object (like array.set()), you don't get a NoSuchMethodError, or anything like that. What you get (what we got on a HotSpot we had a year ago, anyway) is... a native crash. Segmentation fault, if I am not mistaken. Our theory is that the vtable for arrays if so optimized that it is not even there, and lookup crashes because of that.

# ABOUT THE AUTHOR

## ANTON ARHIPOV



**Anton Arhipov** is Software Engineer and JRebel Product Lead at ZeroTurnaround. Professional interests include programming laguages, middleware and tooling. Java enthusiast, vim fan, IntelliJ addict, loves tea and doesn't drink coffee. Anton is ZeroTurnaround representative at JCP for JSR342 (Java EE 7) and he is also a JetBrains Academy member. Anton has delivered talks at international Java conferences: Jfokus, JavaZone, EclipseCon, JavaOne, 33rd Degree, GeeCON and various JUG meet ups.

RebelLabs is the research & content division of ZeroTurnaround

Contact Us

Twitter: @RebelLabs
Web: http://zeroturnaround.com/rebellabs
Email: labs@zeroturnaround.com

**Estonia**
Ülikooli 2, 5th floor
Tartu, Estonia, 51003
Phone: +372 740 4533

**USA**
545 Boylston St., 4th flr.
Boston, MA, USA, 02116
Phone: 1(857)277-1199

**This report is brought to you by:**
Anton Arhipov, Erkki Lindpere, Ryan St. James & Oliver White