Applied Static Analysis

Java Bytecode

Software Technology Group
Department of Computer Science
Technische Universität Darmstadt
Dr. Michael Eichberg

Java Bytecode

In the following we are going to analyze Java Bytecode. For that, we need a basic understanding of it.

> [Java Bytecode is ...] A hardware- and operating system-independent binary format, known as the class file format [1].

Java Bytecode is interpreted by a Java Virtual Machine.

The Java Virtual Machine is a stack machine; i.e., all (except one) operations are preformed on the stack. For example, to add two values both values first have to be put on the stack. The result is then either further processed or stored in a local register.

Structure of the Java Virtual Machine

Types

Type **(Field Descriptor)**	***Computational Type / Category(~number of operands/registers used)***
Primitive Types:	
boolean `(Z)`, byte `(B)`, short `(S)`, int `(I)`, char `(C)`	int / cat. 1
long `(J)`	long / cat. 2
float `(F)`	float / cat. 1
double `(D)`	double / cat. 2
return address	return address / cat. 1
Reference Types:	
class `(A)`	reference value / cat. 1
array `(A)`	reference value / cat. 1
interface `(A)`	reference value / cat. 1

The JVM internally does not distinguish between boolean, byte, char, short and int values; they are all treated as int values. However, some special support for arrays may exist.

The instruction mnemonics loosely use the field descriptors to specify which type of values it handles. For example, the instruction to add to int values is actually called `iadd` and the one two add two float values is called `fadd`. However, the instructions which process long value actually use an initial `l`.

Structure of the Java Virtual Machine

Run-time Data Areas

- the **pc** (program counter) register contains the address of the instruction that is currently executed by a thread; each thread has its own pc
- each JVM thread has a **private stack** which holds local variables and partial results
- the **heap** is shared among all threads
- **frames** are allocated from a JVM thread’s private stack when a method is invoked; each frame has *its own array of local variables and operand stack*
- **local variables** are indexed
    - a single local variable can hold a value belonging to computational type category 1;
    - a pair of local variables can hold a value having computational type category 2
- the **operand stack** is empty at creation time; an entry can hold any value
- the **local variables** contain the parameters (including the implicit this parameter in local variable 0)

Hence, the minimum number of local variables of a method is determined by the number of parameters.

The maximum length of a method is 65536, which is a frequent issue with generated code.
A method can have only 65536 local variables; however, the maximum number of locals in the JDK is *only* 142. The maximum number of locals in OPAL is/was 1136. The maximum stack size of a single method is 65536, but the maximum observed stack size of any method in the JDK is *just* 42.

Structure of the Java Virtual Machine

Special Methods

- the name of instance initialization methods (Java constructors) is `<init>`
- the name of the class or interface initialization method (Java static initializer) is `<clinit>`

Exceptions

- are instance of the class `Throwable` or one of its subclasses; exceptions are thrown if:
  - an `athrow` instruction was executed
  - an abnormal execution condition occurred (e.g., division by zero)

At the JVM level no distinction between checked and unchecked exceptions is made.

Structure of the Java Virtual Machine

Instruction Set Categories

- Load and store instructions (e.g., aload_0, istore(x))
- Arithmetic instructions (e.g., iadd, iushr)
- (Primitive/Base) Type conversion instructions (e.g., i2d,l2d,l2i)
- Object/Array creation and manipulation (e.g., new, newarray, checkcast)
- (Generic) Operand stack management instructions (e.g., dup, dup2_x2, swap)
- Control transfer instructions (e.g., iflt, if_icmplt, goto)
- Method invocation instructions (e.g., invokespecial, invokestatic, invokevirtual)
- Return instructions (e.g., return, areturn)
- Throwing exceptions (athrow)
- Synchronization (monitorenter, monitorexit)

The control-transfer instructions `jsr` and `ret` are outdated and only found in very old Java code (Java 5 and earlier.)
Except of the load and store instructions, the only other instruction that manipulates a local variable is `iinc`.
The semantics of the generic stack management instructions depends on the computational type categories of the values on the stack.
To enable long jumps (offsets smaller than -128 or larger then +127) some of the control transfer-instructions can be modified using the *wide* operator.

Java Bytecode - Object Creation

In Java Bytecode, the creation of a new object:

```
Object o = new Object();
```

is a two step process:

```
new java/lang/Object;
dup; // <= typically
... // push constructor parameters on the stack (if any)
invokespecial java/lang/Object.<init>();
... // do something with the initialized object
```

Usages of an uninitialized object are strictly limited. It is, e.g., not possible to store it in fields of other objects or to pass it around.

Java Bytecode - Control Flow

```java
static int max(int i, int j) {
    if (i > j) return i;
    else       return j;
}
```

PC	Instruction	Remark	Stack (after execution)
0	`iload_0`	load the first parameter	`i` →
1	`iload_1`	load the second parameter	`i`, `j` →
2	`if_icmple` goto pc `+5`	jumps if `i` ≤ `j`	→
5	`iload_0`		`i` →
6	`ireturn`		→
7	`iload_1`		`j` →|
8	`ireturn`		→

In case of an instance method, the first parameter is the implicit self reference and is stored in local 0. Hence, `this` is typically loaded using `aload_0`.

Notice the compilation of the if statement. It is common that in the bytecode the if operator is the inverse one, because if the condition evaluates to true, we then perform the jump (to the else branch) while in the source code, we simply fall through in case the condition evaluates to true.

One of the most important requirements – which is checked by the bytecode verifier – is that the stack layout before the execution of a statement is always the same independent of the path that was taken. The stack layout is basically determined by the types of values on the stack. W.r.t. reference values the type is further annotated with the information whether the type is already properly initialized or not. E.g., you are not allowed to pass around an uninitialized object!

Java Bytecode - Infinite Loops

^It is possible to have methods which never return (normally):

```
public void run() {
    while (true) { try { doIt(); } catch (Throwable t) { log(t); } }
}
```

PC	Instruction
0	invokestatic ControlFlow.doIt()
3	goto 0
6 (catch exception)	astore_1
7	aload_1
8	invokestatic ControlFlow.log(java.lang.Throwable)
11	goto 0

Exception Handling

```
public delete(String s) {
try
/*1:*/ { new java.io.File(s).delete();
/*2:*/ }
catch (IOException e)
/*3:*/ { // handle IOException...
} catch (Exception e)
/*4:*/ { // handle Exception...
} finally
/*5:*/ { }
}
```

At runtime the first handler that can handle an exception will be invoked.
Exception handler table:

Start PC	End PC (exclusive)	Handler PC	Handled Exception
1	2	3	IOException
1	2	4	Exception
1	2	5	< ANY >

Please note, that the finally block is generally included twice. Once, for the case if no exception is thrown and once for the case when an exception is thrown.

Note that **this method has at least two exit points**: (5) - if some exception is thrown or if the execution returns normally.

The stack - when an exception handler is reached – will always just contain the thrown exception and nothing else. The thrown exception object will never be `null`.

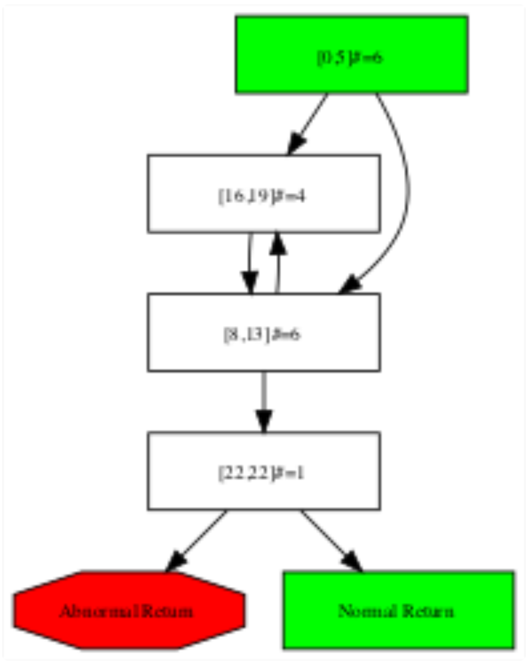
Irreducible CFGs - Example

The following code will never be generated by a Java/Scala/Groovy/Kotlin/… compiler.

PC	Instruction	Parameter
0	sipush	42
3	istore_0	
4	iload_0	
5	ifeq	16
8	iinc	reg=0, incBy=-1
11	iload_0	
12	iload_1	
13	if_icmpeq	22
16	iinc	reg=0, incBy=2
19	goto	8
22	return	

Irreducible CFGs are not common in Java bytecode; in particular not in bytecode generated by compilers. However, code obfuscators may make use of it to make decompilation (much) harder. Furthermore, explicitly engineered bytecode may also have irreducible CFGs.

The CFG for the above method is shown next:



A return instruction may throw an exception if synchronization is done in an improper way; basically, if we have unbalanced `monitorenter` / `monitorexit` statements.

Lambda Expressions

```
List<T> l = ...;
l.sort(
    (T a, T b) -> { return a.hashCode() - b.hashCode(); }
);
```

Lambda expression in Java source code are compiled to individual methods in the scope of the defining class, these special methods are invoked using `invokedynamic` instructions.

Handling of lambda expressions in a static analysis frameworks basically requires that the JVM's resolution mechanism, which includes the language's specific CallSite factory, is reimplemented in the static analysis frameworks. It is in particular necessary to generate *fake objects* to encapsulate the call and to make it possible to pass the reference to the lambda method around.

Java Bytecode - Invokedynamic

Let’s assume that the following lambda expression is used to implement a `Comparator<T>`:

```
(T a, T b) -> { return a.hashCode() - b.hashCode(); }`
```

This code is compiled to:

```
invokedynamic (
  Bootstrap_Method_Attribute[<index into the bootstrap methods table>],
  java.util.Comparator.compare() // required by the bytecode verifier
)
```

The bootstrap method attribute stores method handles along with the static arguments for each handle. The method handle will be resolved during resolution of a call site and then invoked with the specified arguments. The result of the resolution is a `CallSite` object which can eventually be used to invoke the lambda method when required.

Java (and also in most cases Scala) use `java.lang.invoke.LambdaMetafactory{ static CallSite metafactory(MethodHandles.Lookup caller, String invokedName, MethodType invokedType, MethodType samMethodType, MethodHandle implMethod, MethodType instantiatedMethodType) }` to perform method resolution. The first three parameters are stacked automatically.
The parameters are:

- samMethodType = `MethodType(int (java.lang.Object, java.lang.Object))`
- implMethod = `MethodHandle(invokeStatic(<DefiningClass> { int lambda$0 (java.lang.Object, java.lang.Object) }))`
- instantiatedMethodType = `MethodType(int (java.lang.Object, java.lang.Object))`

Java Bytecode - Peculiarities

- Reference types are represented using binary notation. In binary notation packages are separated using “/“: e.g., java/lang/Object.
- The JVM has no “negate” instruction. A negation in Java (!b) is compiled to an if instruction followed by a push of the corresponding value.
- The JVM has no direct support for shortcut-evaluation (&&, ||).
- The *catch block* is not immediately available; only the pc of the first instruction of the catch block is known.

Java Bytecode - Summary

- Has a very close relationship with Java source code.

Java Bytecode analysis is common place because Java compilers basically don’t perform optimizations and mapping back an analyses' result is often easily possible; in particular if debug information is embedded. The only optimizations that are regularly performed are constant propagation for final local variables and final fields (primitive values and Strings). Expressions are evaluated if all parameters are primitive constants no further optimizations are done.

- Java Bytecode is very compact and can efficiently be parsed.

On a modern notebook (Core i7 with 6 cores) ~19000 class files can be loaded and traversed in under one second.

- Having a stack and registers, makes data-flow analyses unnecessarily complex.

In particular the generic stack management operations makes it comparatively complex to determine the data-flow when compared to higher-level intermediate representations.

- The large instruction set complicates analyses because the same semantics may be expressed in multiple ways.

However, in many cases all known compilers will generate the code in the same way. This greatly facilitates the definition and identification of bug patterns.

References

1. The Java Virtual Machine Specification Java SE 8 Edition, Oracle America, Inc., 2014↩