Applied Static Analysis

Java Bytecode

Software Technology Group
Technische Universität Darmstadt
Dr. Michael Eichberg

Java Bytecode

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

Structure of the Java Virtual Machine

Types

Type (Field Descriptor)	*Computational Type*
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

Structure of the Java Virtual Machine

Run-time Data Areas

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

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

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

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*

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

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;

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

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

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 contains the parameters (including the implicit this parameter in local variable 0)

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)

Structure of the Java Virtual Machine

Instruction Set Categories

Structure of the Java Virtual Machine

Instruction Set Categories

- Load and store instructions (e.g., aload_0, istore(x))

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)

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)

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)

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, dup2_x2)

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, dup2_x2)
- Control transfer instructions (e.g., itlt, if_icmplt, goto)

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, dup2_x2)
- Control transfer instructions (e.g., itlt, if_icmplt, goto)
- Method invocation instructions (e.g., invokespecial, invokestatic, invokevirtual)

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, dup2_x2)
- Control transfer instructions (e.g., itlt, if_icmplt, goto)
- Method invocation instructions (e.g., invokespecial, invokestatic, invokevirtual)
- Return instructions (e.g., return, areturn)

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, dup2_x2)
- Control transfer instructions (e.g., itlt, if_icmplt, goto)
- Method invocation instructions (e.g., invokespecial, invokestatic, invokevirtual)
- Return instructions (e.g., return, areturn)
- Throwing exceptions (athrow)

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, dup2_x2)
- Control transfer instructions (e.g., itlt, if_icmplt, goto)
- Method invocation instructions (e.g., invokespecial, invokestatic, invokevirtual)
- Return instructions (e.g., return, areturn)
- Throwing exceptions (athrow)
- Synchronization (monitorenter, monitorexit)

Java Bytecode - Control Flow

```
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`		→

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
```

Java Bytecode - Exception Handling

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

}
```

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

Java Bytecode - Lambda Expressions

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

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
)
```

Java Bytecode - Peculiarities

Java Bytecode - Peculiarities

- Reference types are represented using binary notation. In binary notation packages are separated using “/“: e.g., 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.

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 (&&, ||).

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 is very compact and can efficiently be parsed.
- Having a stack and registers, makes data-flow analyses unnecessarily complex.
- The large instruction set complicates analyses because the same semantics may be expressed in multiple ways.