

SWEN430 - Compiler Engineering

Lecture 11 - Java Bytecode

Alex Potanin & David J. Pearce

*School of Engineering and Computer Science
Victoria University of Wellington*

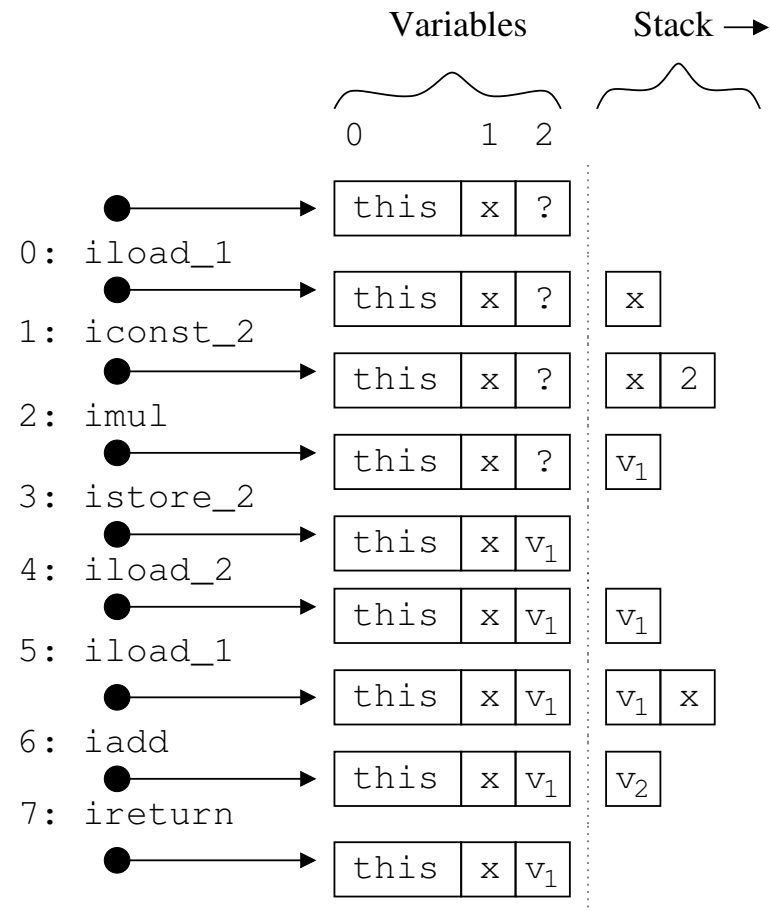
Java Bytecode

```
class Test {  
    public int f(int x) {  
        int y = x * 2;  
        return y + x;  
    }  
}
```

```
public int f(int);  
Code:  
Stack=2, Locals=3  
0:   iload_1  
1:   iconst_2  
2:   imul  
3:   istore_2  
4:   iload_2  
5:   iload_1  
6:   iadd  
7:   ireturn
```

- A stack-based language, similar to machine code
- Can decompile Java programs with `javap`
- For details of bytecode instructions, see *JVM Specification*

A Detailed Example



- Here, x represents value of parameter x on entry
- v_1 and v_2 represent intermediate values

Some Bytecode Instructions

<code>aload X</code>	Push reference onto stack from variable X
<code>astore X</code>	Pop reference off stack into variable X
<code>iconst X</code>	Push int constant <code>i</code> onto stack
<code>ladd</code>	Take top two (long) items off stack, perform long addition, and push (long) result back on stack
<code>imul</code>	Take top two items off stack, perform int multiplication, and push result back on stack
<code>areturn</code>	Return top item on stack as reference
goto X	Goto location X
<code>ifeq X</code>	Take one item off stack; if equal top zero goto location X

More Bytecode Instructions

<code>pop</code>	Pop top item off stack and discard
<code>dup</code>	Duplicate top item on stack
<code>getfield F</code>	Pop reference off stack and load field <code>F</code> from it onto stack
<code>putfield F</code>	Pop top two items of stack; write first to field <code>F</code> in object referred by second
<code>checkcast C</code>	Pop item from stack, check instance of <code>C</code> and push back
<code>iinc X, c</code>	Increment local variable <code>X</code> by constant <code>C</code>

JVM Types

- JVM variables have simplistic types, compared with Java:

`i` = integer, `l` = long, `f` = float, `d` = double, `a` = reference

<code>iload</code>	<code>lload</code>	<code>fload</code>	<code>dload</code>	<code>aload</code>
<code>istore</code>	<code>lstore</code>	<code>fstore</code>	<code>dstore</code>	<code>astore</code>
<code>iconst</code>	<code>lconst</code>	<code>fconst</code>	<code>dconst</code>	<code>aconst_null</code>
<code>ireturn</code>	<code>lreturn</code>	<code>freturn</code>	<code>dreturn</code>	<code>areturn</code>
<code>iadd</code>	<code>ladd</code>	<code>fadd</code>	<code>dadd</code>	
<code>imul</code>	<code>lmul</code>	<code>fmul</code>	<code>dmul</code>	
<code>idiv</code>	<code>ldiv</code>	<code>fdiv</code>	<code>ddiv</code>	
<code>ineg</code>	<code>lneg</code>	<code>fneg</code>	<code>dneg</code>	
<code>irem</code>	<code>lrem</code>	<code>frem</code>	<code>drem</code>	

- These all operate in essentially the same way, just for different types

Conditional Statements

```
class Test {  
  int abs(int x) {  
    if(x >= 0) {  
      return x;  
    } else {  
      return -x;  
    }  
  }  
}
```

```
int abs(int);  
Code:  
Stack=1, Locals=2  
0:   iload_1  
1:   iflt     6  
4:   iload_1  
5:   ireturn  
6:   iload_1  
7:   ineg  
8:   ireturn
```

- Control-flow implemented using *conditional branching*.

Looping Statements

```
public class Test {  
    int count(int end) {  
        int r = 0;  
        for(int i=0;i!=end;++i) {  
            r = r + i;  
        }  
        return r;  
    }  
}
```

```
int count(int);
```

Code:

```
Stack=2, Locals=4  
0:   iconst_0  
1:   istore_2  
2:   iconst_0  
3:   istore_3  
4:   iload_3  
5:   iload_1  
6:   if_icmpeq      19  
9:   iload_2  
10:  iload_3  
11:  iadd  
12:  istore_2  
13:  iinc      3, 1  
16:  goto      4  
19:  iload_2  
20:  ireturn
```

- Loops implemented using unconditional **backward branches**

JVM Type Conversions

```
long f(int x) {  
    return x; // implicit  
}
```

```
long f(int);
```

Code:

Stack=2, Locals=2

0: iload_1

1: i2l *// explicit*

2: lreturn

```
int f(int x, float y) {  
    if(x == y) { return x; }  
    return 0;  
}
```

```
int f(int, float);
```

Code:

Stack=2, Locals=3

0: iload_1

1: i2f *// explicit*

2: fload_2

3: fcmpl

4: ifne 9

7: ...

- Java permits **implicit conversions**
- JVM permits only **explicit conversions**

JVM Type Conversion Table

i2l	Convert int to long	
i2f	Convert int to float	(lossy)
i2d	Convert int to double	
l2i	Convert long to int	(lossy)
l2f	Convert long to float	(lossy)
l2d	Convert long to double	(lossy)
f2i	Convert float to int	(lossy)
f2l	Convert float to long	(lossy)
f2d	Convert float to double	

Long Types

```
public class Test {  
    long f(long x, int y) {  
        return x + y;  
    }  
}
```

```
long f(long, int);
```

Code:

Stack=4, Locals=4

0: lload_1

1: iload_3

2: i2l

3: ladd

4: lreturn

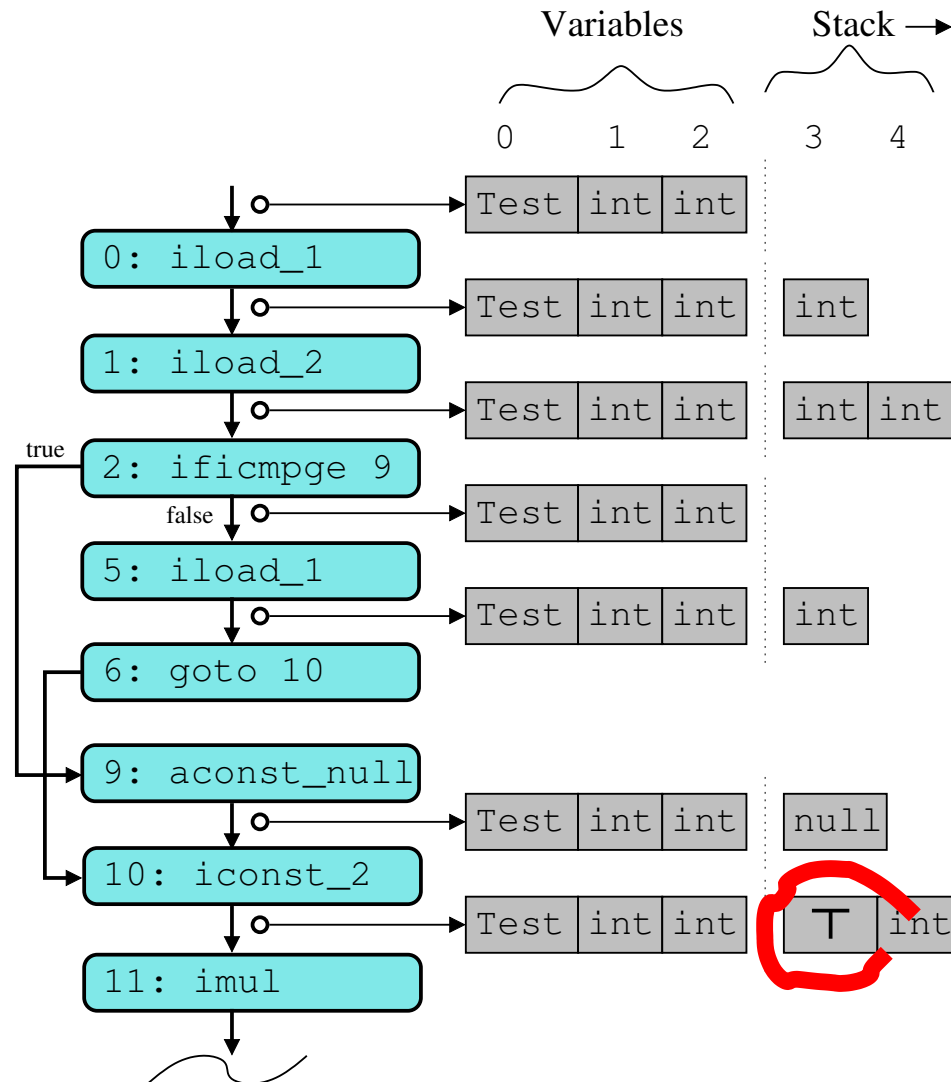
- “Long” types require **two variable slots**
 - i.e. **long** and **double** types
 - In example above, `x` occupies slots 1 + 2 and `y` occupies slot 3

Bytecode Verification

```
int f(int, int);  
Code:  
Stack=2, Locals=3  
0:   iload_1  
1:   iload_2  
2:   if_icmpge 9  
5:   iload_1  
6:   goto 10  
9:   aconst_null  
10:  iconst_2  
11:  imul  
12:  ireturn
```

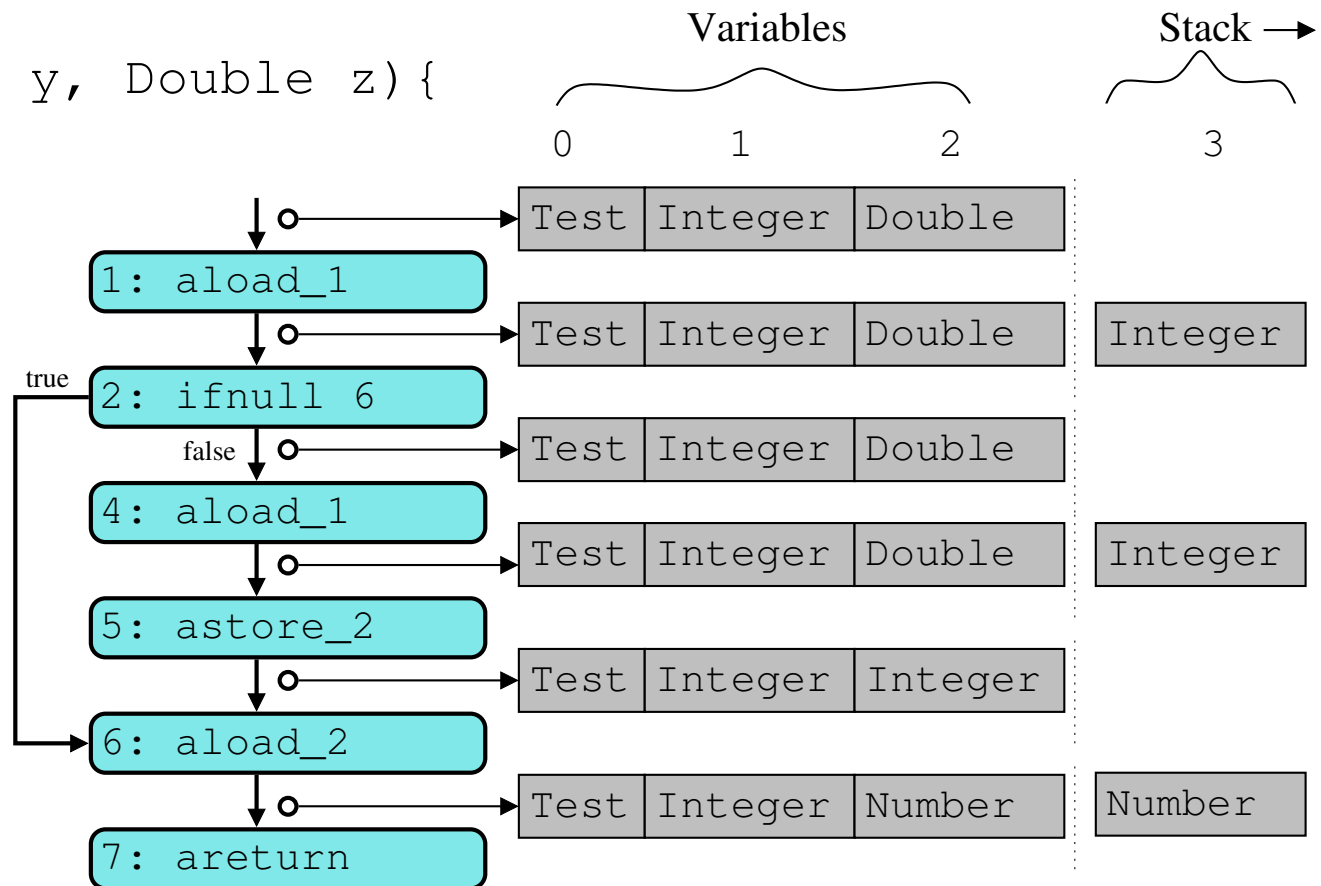
- Bytecode verification performed on every class loaded
- Algorithm used is a form of *data-flow analysis*

Bytecode Verification Example



Another Bytecode Verification Example

```
class Test {  
  Number f(Integer y, Double z) {  
    Number r = z;  
    if(y != null) {  
      r = y;  
    }  
    return r;  
  }  
}
```



- `Integer` \sqcup `Double` = `Number` — hence, method **verifies!**

Lattice of JVM Types

$$\overline{T_1 \leq T_1}$$

$$\overline{T_1 \leq \top}$$

$$\frac{C_1 \leq C_2 \quad C_2 \leq C_3}{C_1 \leq C_3}$$

$$\frac{\text{class } C_1 \text{ extends } C_2}{C_1 \leq C_2}$$

$$\overline{C_1 \leq \text{java.lang.Object}}$$

$$\overline{\text{null} \leq C_1}$$

- T_1 represents an arbitrary type; C_1, C_2 represent class references; \top is undefined type
- For simplicity, ignoring arrays, interfaces, etc
- This relation forms a *join semi-lattice* — i.e. \sqcup always exists, but not always \sqcap
- **Note:** e.g. `int ≤ long` does not hold here (although it does in normal Java)

References

- 1 “Java Bytecode Verification: Algorithms and Formalisations”, X. Leroy (an excellent read)
- 2 “The Java® Virtual Machine Specification, Java SE 7 Edition”, Tim Lindholm, Frank Yellin, Gilad Bracha and Alex Buckley.