SWEN430 - Compiler Engineering (2018)

Lecture 15 - Bytecode Verification

Lindsay Groves & David J. Pearce

School of Engineering and Computer Science
Victoria University of Wellington

Bytecode Verification

"Even though a compiler for the Java programming language must only produce class files that satisfy all the static and structural constraints in the previous sections, the Java Virtual Machine has no guarantee that any file it is asked to load was generated by that compiler or is properly formed. Applications such as web browsers do not download source code, which they then compile; these applications download already-compiled class files. The browser needs to determine whether the class file was produced by a trustworthy compiler or by an adversary attempting to exploit the Java Virtual Machine.

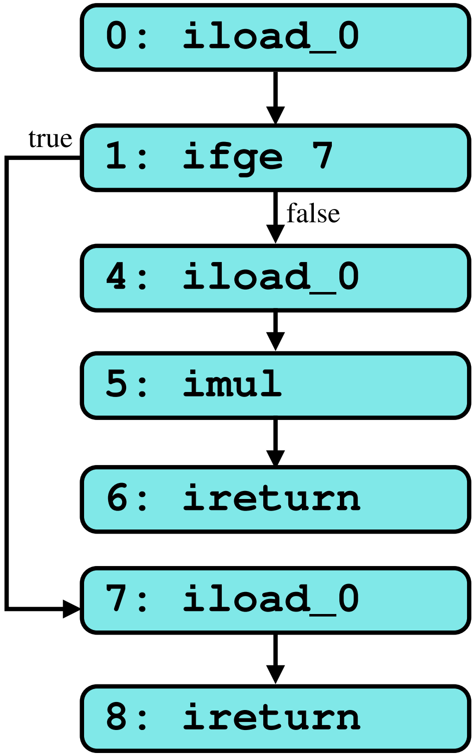
... Because of these potential problems, **the Java Virtual Machine needs to verify for itself that the desired constraints are satisfied by the class files** *it attempts to incorporate. A Java Virtual Machine implementation verifies that each class file satisfies the necessary constraints at linking time"*

– JVM Specification

Bytecode Verification

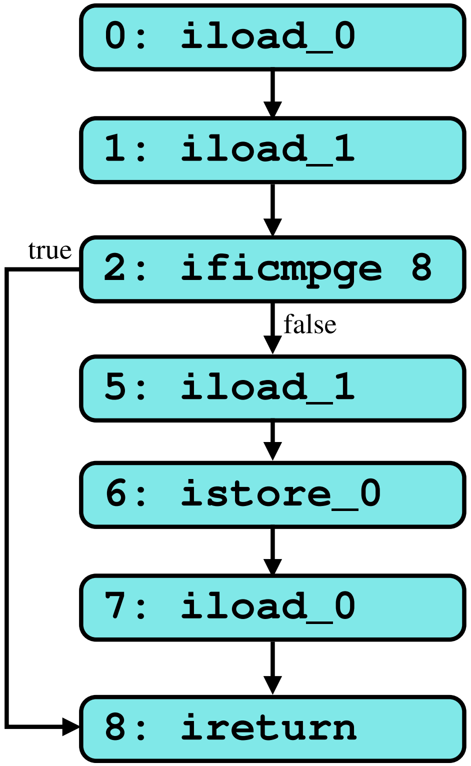
- Some of the checks performed during verification include:
  - Checking stack cannot **overflow** or **underflow**
  - Checking stack height is **statically determinable** at each location
  - Checking each variable or stack location is **defined before used**
  - Checking each variable or stack location has **appropriate type when used**
  - Checking branch targets are **within the given method**
  - Checking branch targets are on **bytecode boundaries**
  - Checking every method **terminated by return**

Example 1 — Stack Underflow



```
Exception in thread "main" java.lang.VerifyError:
(class: Test_1, method: abs signature: (I)I)
Unable to pop operand off an empty stack
```

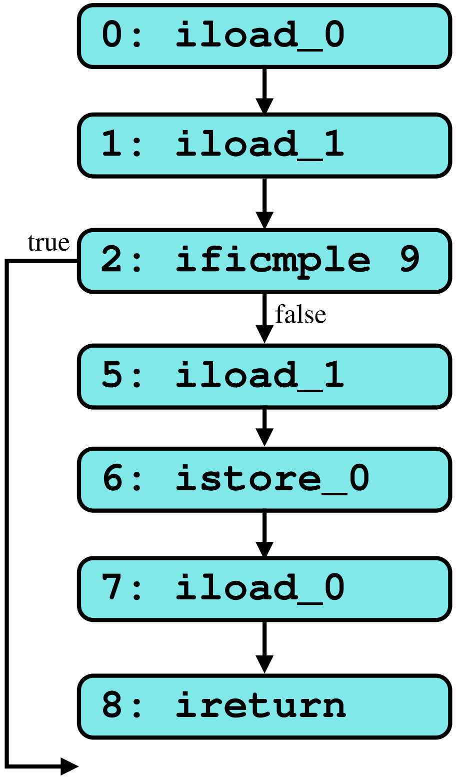
Example 2 — Stack Height



```
0: iload_0
1: iload_1
2: ificmpge 8
5: iload_1
6: istore_0
7: iload_0
8: ireturn
```

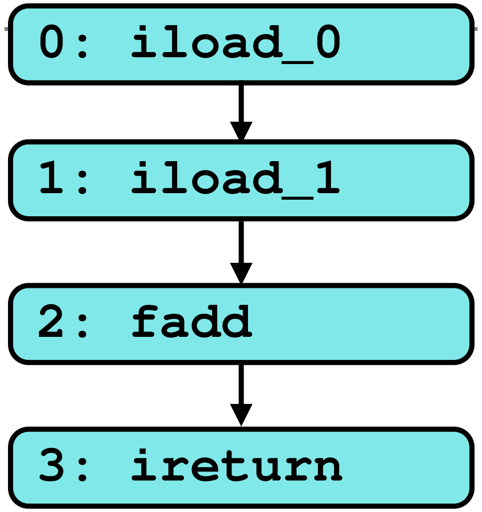
```
Exception in thread "main" java.lang.VerifyError:
(class: Test_2, method: max signature: (II)I)
Inconsistent stack height 1 != 0
```

Example 3 — Branch Destination



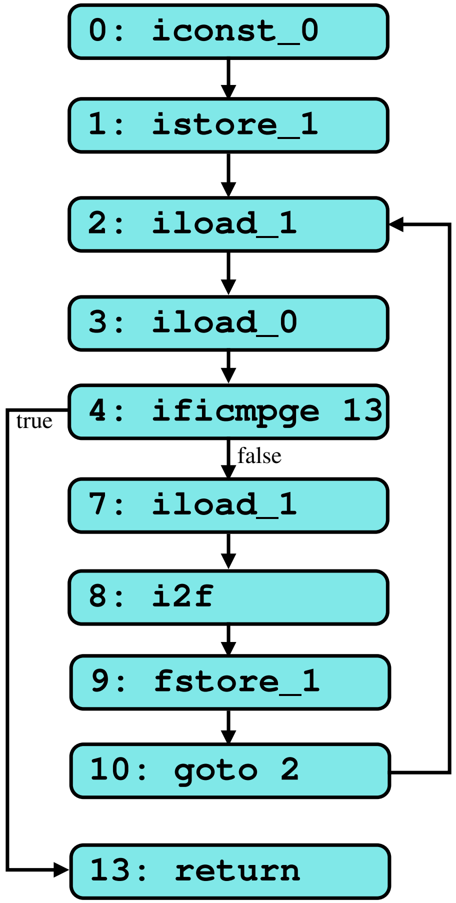
```
Exception in thread "main" java.lang.VerifyError:
(class: Test_3, method: min signature: (II)I)
Illegal target of jump or branch
```

Example 4 — Invalid Operand



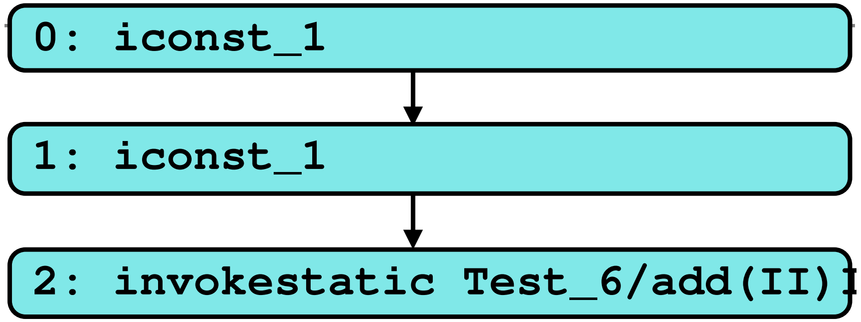
```
Exception in thread "main" java.lang.VerifyError:
(class: Test_4, method: add signature: (II)I)
Expecting to find float on stack
```

Example 5 — Type Around Loop



```
Exception in thread "main" java.lang.VerifyError:
(class: Test_5, method: f signature: (I)V)
Accessing value from uninitialized register 1
```

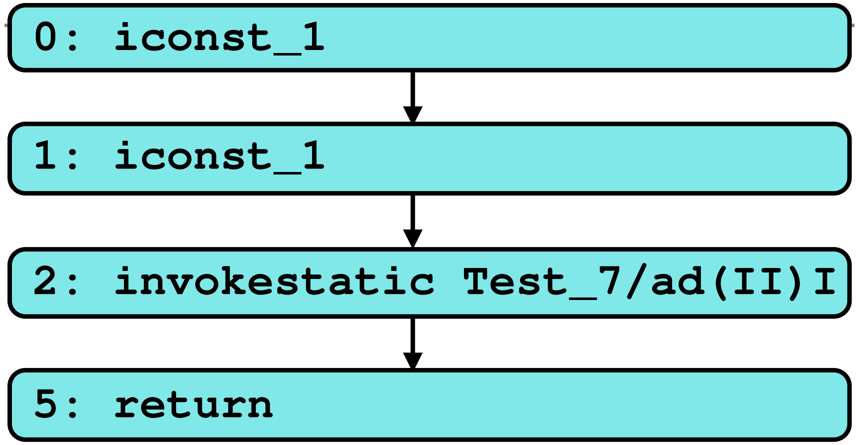
Example 6 — Missing Return



```
0: iconst_1
1: iconst_1
2: invokestatic Test_6/add(II)I
```

```
Exception in thread "main" java.lang.VerifyError:
(class: Test_6, method: main signature:
([Ljava/lang/String;)V)
Falling off the end of the code
```

Example 7 — Missing Method



```
0: iconst_1
1: iconst_1
2: invokestatic Test_7/ad(II)I
5: return
```

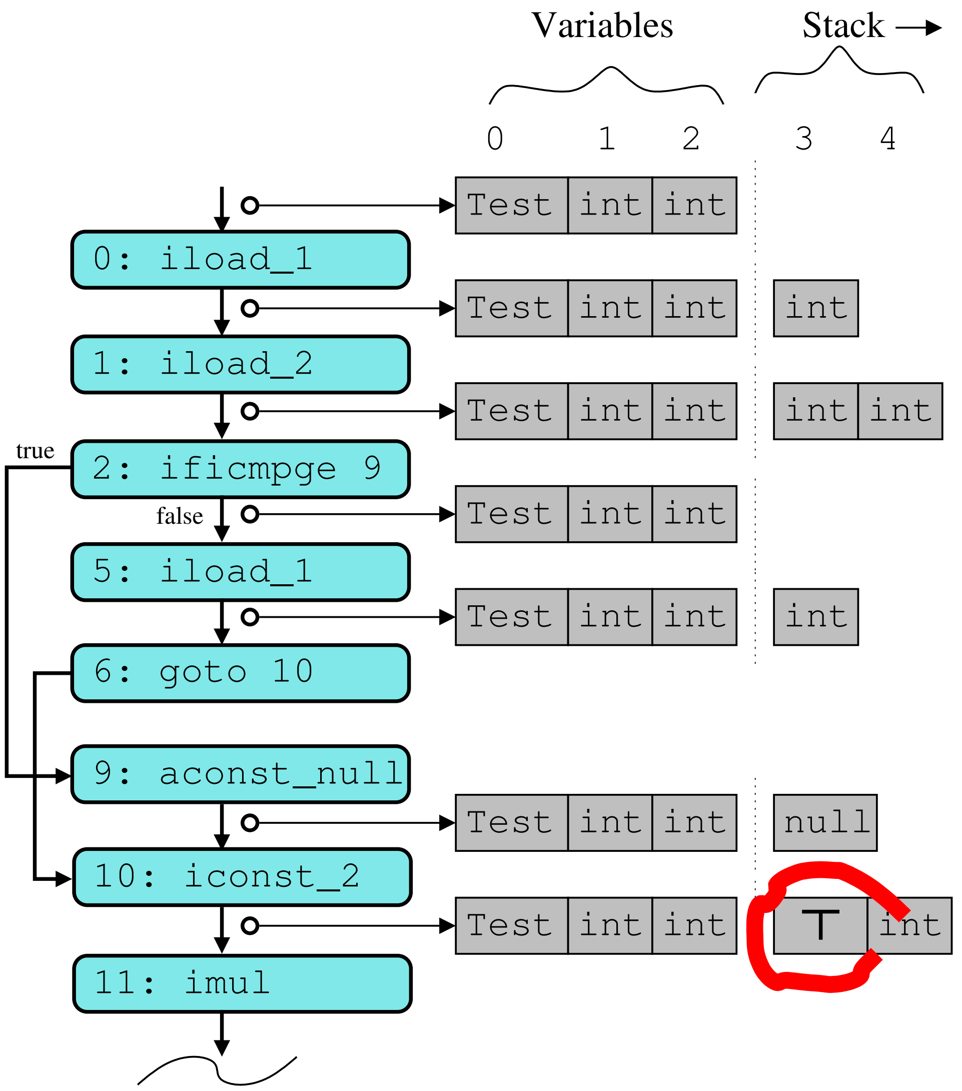
```
Exception in thread "main" java.lang.NoSuchMethodError:
Test_7.ad(II)I
        at Test_7.main(Test_7.j)
```

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*

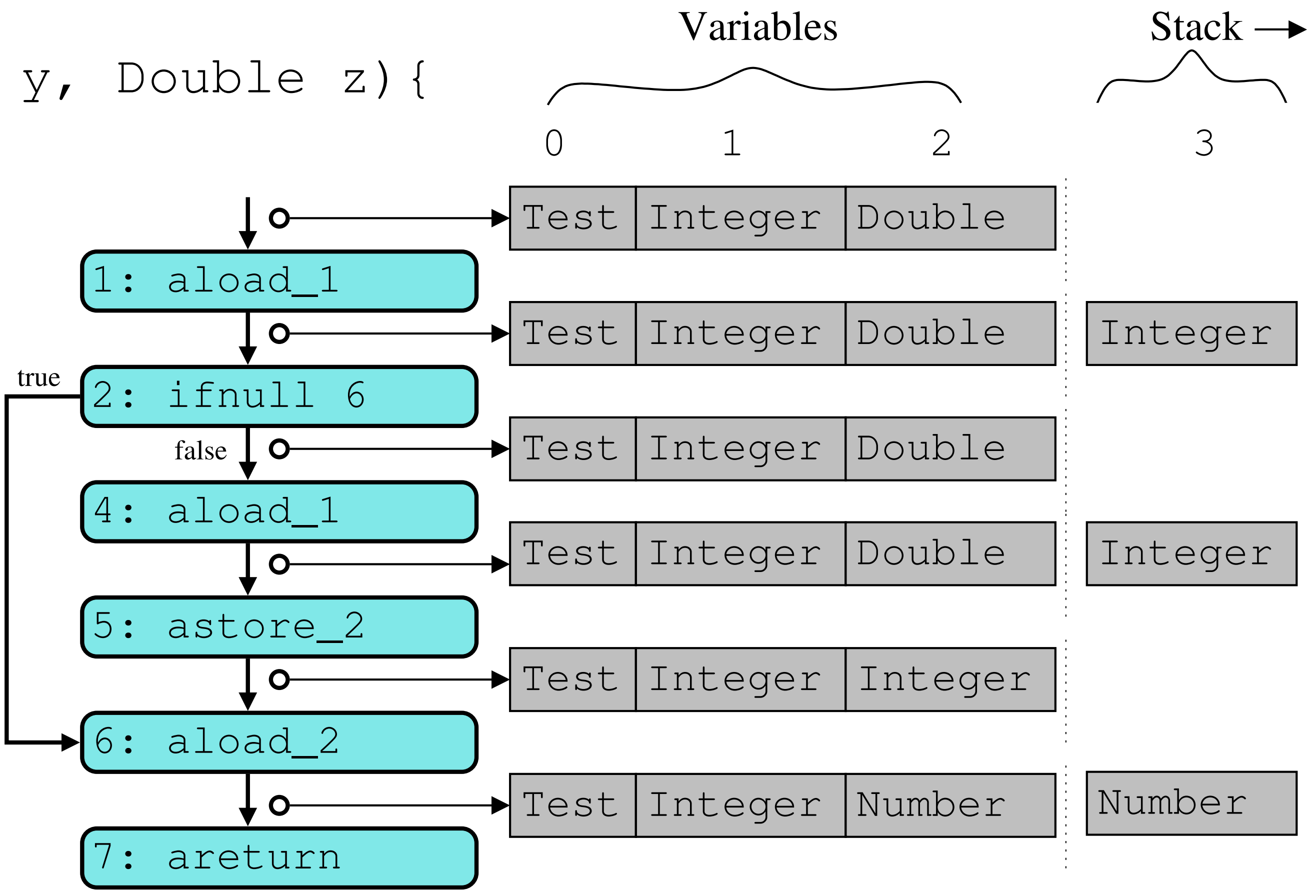
Bytecode Verification Example



	Variables			Stack →	
	0	1	2	3	4
	Test	int	int		
0: iload_1					
	Test	int	int	int	
1: iload_2					
	Test	int	int	int	int
2: ificmpge 9					
	Test	int	int		
5: iload_1					
	Test	int	int	int	
6: goto 10					
9: aconst_null					
	Test	int	int	null	
10: iconst_2					
	Test	int	int	⊤	int
11: imul					

Another Bytecode Verification Example

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



Variables

Stack →

0 1 2 3

Test | Integer | Double

1: aload_1

Test | Integer | Double | Integer

2: ifnull 6

true

false

Test | Integer | Double

4: aload_1

Test | Integer | Double | Integer

5: astore_2

Test | Integer | Integer

6: aload_2

Test | Integer | Number | Number

7: areturn

- Integer ⊔ Double = Number — hence, method **verifies!**

Lattice of JVM Types

$$\frac{}{\mathtt{T}_1 \leq \mathtt{T}_1} \qquad \frac{}{\mathtt{T}_1 \leq \top}$$

$$\frac{\mathtt{C}_1 \leq \mathtt{C}_2 \quad \mathtt{C}_2 \leq \mathtt{C}_3}{\mathtt{C}_1 \leq \mathtt{C}_3} \qquad \frac{\mathtt{class}\ \mathtt{C}_1\ \mathtt{extends}\ \mathtt{C}_2}{\mathtt{C}_1 \leq \mathtt{C}_2}$$

$$\frac{}{\mathtt{C}_1 \leq \mathtt{java.lang.Object}} \qquad \frac{}{\mathtt{null} \leq \mathtt{C}_1}$$

- $\mathtt{T}_1$ represents an arbitrary type; $\mathtt{C}_1, \mathtt{C}_2$ 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. $\mathtt{int} \leq \mathtt{long}$ does not hold here (although it does in normal Java)

References

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