SWEN430 - Compiler Engineering

Lecture 13 - Bytecode Generation II

Alex Potanin & David J. Pearce

School of Engineering and Computer Science
Victoria University of Wellington

Determining Maximum Stack Height

- Must determine **maximum stack height** of each method
- Java Compiler can calculate the **stack difference** for each bytecode:
- Examples:

Bytecode	Stack Difference	Bytecode	Stack Difference
`bipush`	+1	`pop`	-1
`iload X`	+1	`lload X`	+2
`iadd`	-1	`dadd`	-2
`iaload`	-1	`daload`	0
`ineg`	0	`d2f`	-1
`invokevirtual`	???		

- Then, it traverses bytecode sequence determining the max height
- How should we deal with branching?

Determining Maximum Stack Height (cont'd)

```
int f(java.lang.String[]);
Code:                              # diff # height
 0:   aload_1                      #  +1  #  1  #
 1:   ifnull  12                   #  -1  #  0  #
 4:   getstatic System.out         #  +1  #  1  #
 7:   ldc "Hello World"            #  +1  #  2  #
 9:   invokevirtual println:(Ljava/lang/String;)V
                                   #  -2  #  0  #
 12:  iconst_0                     #  +1  #  1  #
 13:  istore_2                     #  -1  #  0  #
 14:  iload_2                      #  +1  #  1  #
 15:  aload_1                      #  +1  #  2  #
 16:  arraylength                  #  0   #  2  #
 17:  iadd                         #  -1  #  1  #
 18:  istore_2                     #  -1  #  0  #
 19:  iload_2                      #  +1  #  1  #
 20:  ireturn                      #  -1  #  0  #
```

- Hence, the maximum stack height for this method is 2

Line Number Information

```
int f(int x, int y) {
  int r = x / y;
  return r;
}
```

```
int f(int, int);
   0:   iload_1
   1:   iload_2
   2:   idiv
   3:   istore_3
   4:   iload_3
   5:   ireturn
 LineNumberTable:
   line 3: 0
   line 4: 4
```

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Test.f(Test.java:3)
        at Test.main(Test.java:8)
```

- Can associate **line number information** with Bytecode
- `LineNumberTable` attribute is for this (see JVM Spec §4.7.8)
- Then can see **where** exceptions occur in original source file

Exception Handlers

```
String f(Integer i) {
 try {
  return i.toString();
 } catch(Exception e){
  return "";
}}
```

```
String f(Integer);
  Code:
   Stack=1, Locals=3
   0:   aload_1
   1:   invokevirtual Integer.toString()
   4:   areturn
   5:   astore_2
   6:   ldc     ""
   8:   areturn
  Exception table:
   from   to  target type
     0     4     5   Class Exception
```

- Exception handlers implemented as table rows:
  - » Range of bytecodes, destination and class
  - » Range cannot include handler itself

Exception Handlers Table

- Java Compiler generates exception handlers such that:
  - » Either exception handler ranges are **disjoint**, or one is a **subrange** of the other
  - » Exception handler code is never **within its own range**
  - » Entry for exception handler only via exception (not via e.g. `goto`)
- Surprisingly, these restrictions **not enforced** by bytecode verifier
  - » Because not considered a threat to integrity of JVM
  - » Still require that e.g. every nonexceptional path to handler has a single object on the operand stack, etc
  - » See JVM Specification, §3.10

Another Example

```
String f(Integer i) {
 try {
  return i.toString();
 } catch(NullPointerException e){
  return "null";
 } catch(Exception e){
  return "";
}}
```

```
String f(Integer);
  Code:
   Stack=1, Locals=3, Args_size=2
   0:   aload_1
   1:   invokevirtual Integer.toString:()
   4:   areturn
   5:   astore_2
   6:   ldc  "null"
   8:   areturn
   9:   astore_2
   10:  ldc     ""
   12:  areturn
  Exception table:
   from   to  target type
     0     4     5   Class NullPointerException

     0     4     9   Class Exception
```

- Multiple Exception handlers are triggered **in order of appearance**

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.

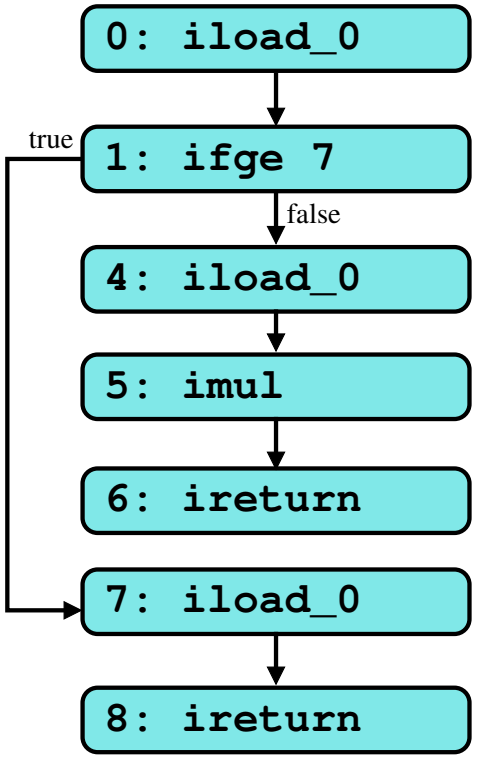
... Because of these potential problems, **the Java Virtual Machine needs to verify for itself that the desired constraints are satisfied by the class files** *it attempts to incorporate. A Java Virtual Machine implementation verifies that each class file satisfies the necessary constraints at linking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

– JVM Specification

Bytecode Verification

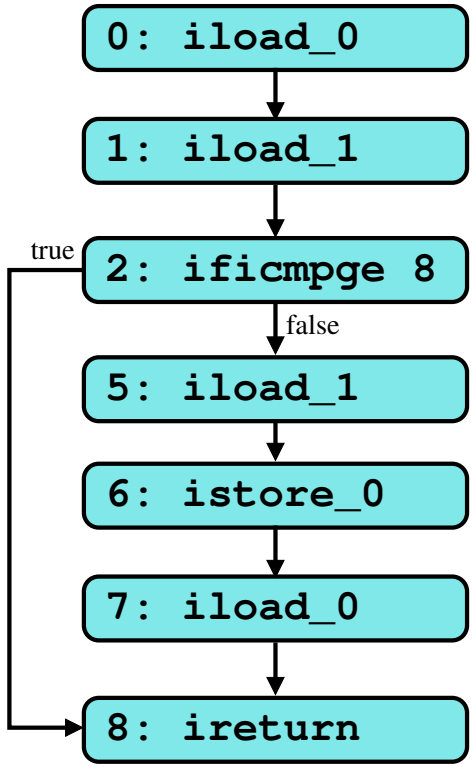
- Some of the checks performed during verification include:
  - » Checking stack cannot **overflow** or **underflow**
  - » Checking stack height is **statically determinable** at each location
  - » Checking each variable or stack location is **defined before used**
  - » Checking each variable or stack 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



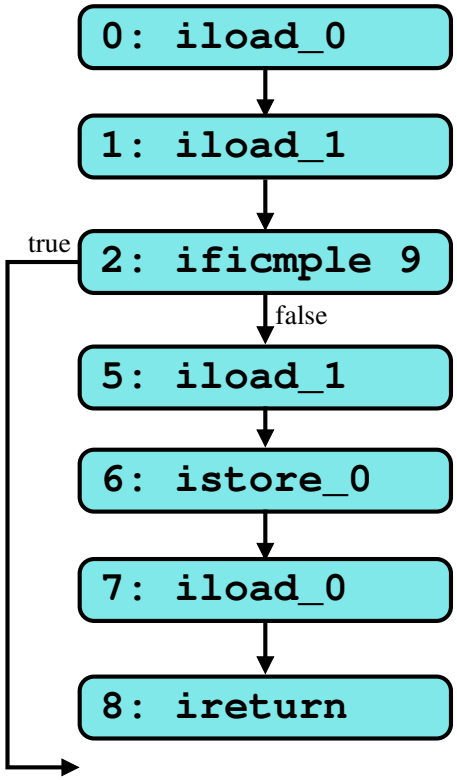

```
Exception in thread "main" java.lang.VerifyError:
(class: Test_1, method: abs signature: (I)I)
Unable to pop operand off an empty stack
```

Example 2 — Stack Height



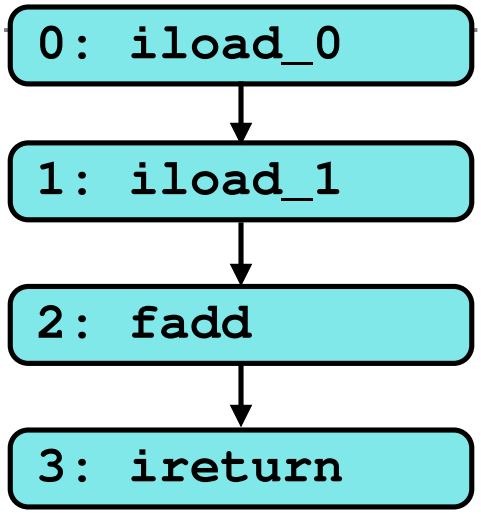

```
Exception in thread "main" java.lang.VerifyError:
(class: Test_2, method: max signature: (II)I)
Inconsistent stack height 1 != 0
```

Example 3 — Branch Destination



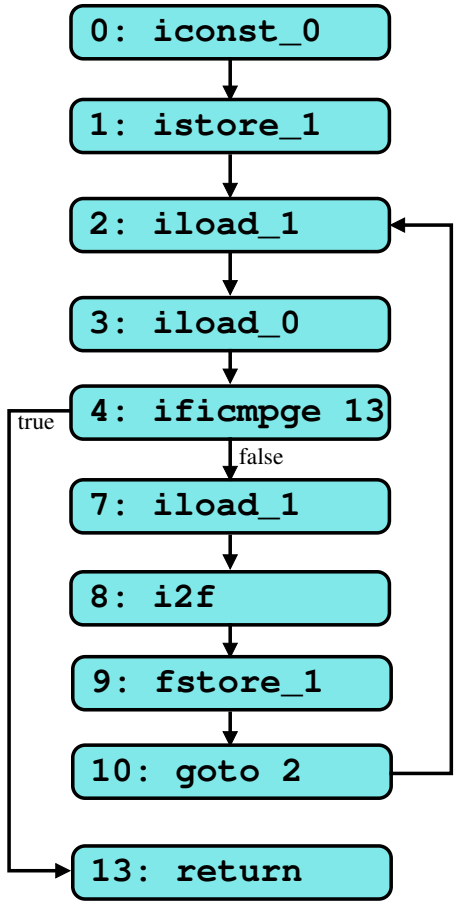

```
Exception in thread "main" java.lang.VerifyError:
(class: Test_3, method: min signature: (II)I)
Illegal target of jump or branch
```

Example 4 — Invalid Operand



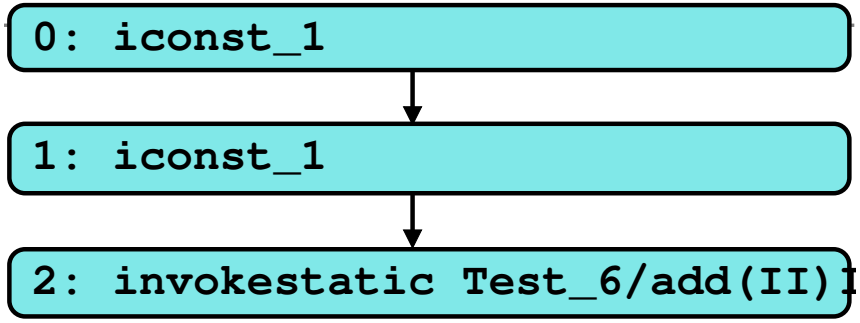

```
Exception in thread "main" java.lang.VerifyError:
(class: Test_4, method: add signature: (II)I)
Expecting to find float on stack
```

Example 5 — Type Around Loop



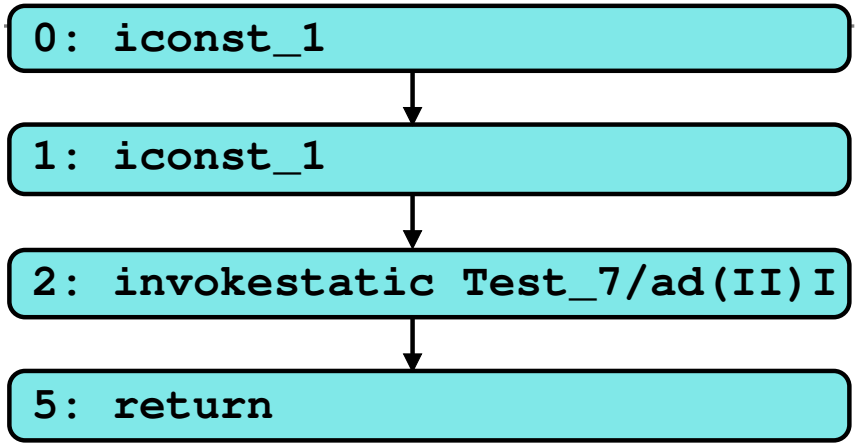

```
Exception in thread "main" java.lang.VerifyError:
(class: Test_5, method: f signature: (I)V)
Accessing value from uninitialized register 1
```

Example 6 — Missing Return




```
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




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