# SWEN430 - Compiler Engineering (2018)

## Lecture 14 - Bytecode Generation II

Lindsay Groves & 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**