Program Representations & Transformations

Outline

- Last Week's Reading
- Program Representations
  - Two program analysis applications
- The JVM
- Java Bytecode
- Lab Today: Ad-hoc testing

Undecidability of Program Properties

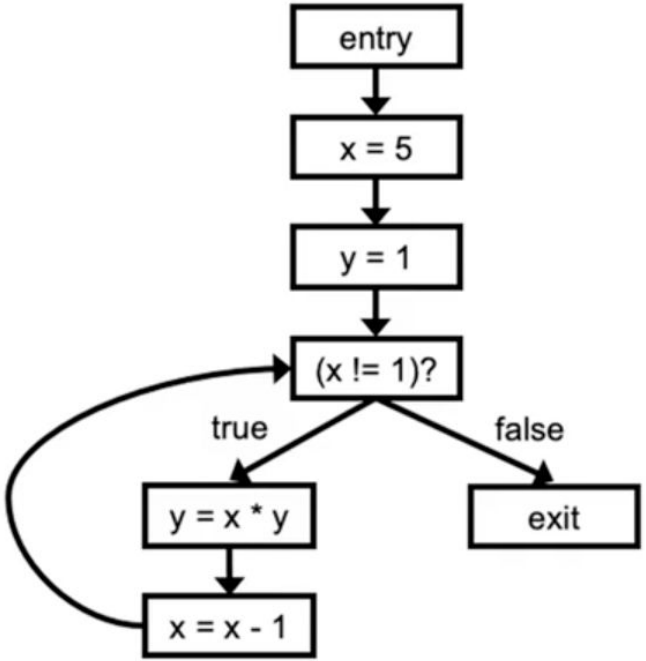
https://docs.google.com/document/d/19ceuvoFxnAqB4SM-GjMhtf2ptAZ6TMGd_GGNp4-Vl7c/edit?usp=sharing

Program Representations

Popular Program Representations

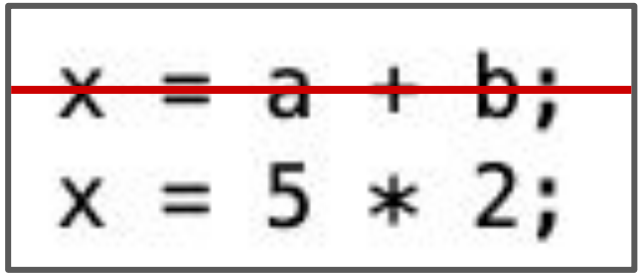
1. Text (Sequence of Tokens)
    a. Intermediate Representation (IR)
    b. bytecode
2. Trees
    a. AST
3. Graphs
    a. Control Flow Graph (CFG)
    b. Data Flow Graph
    c. Call Graph

Control Flow Graphs



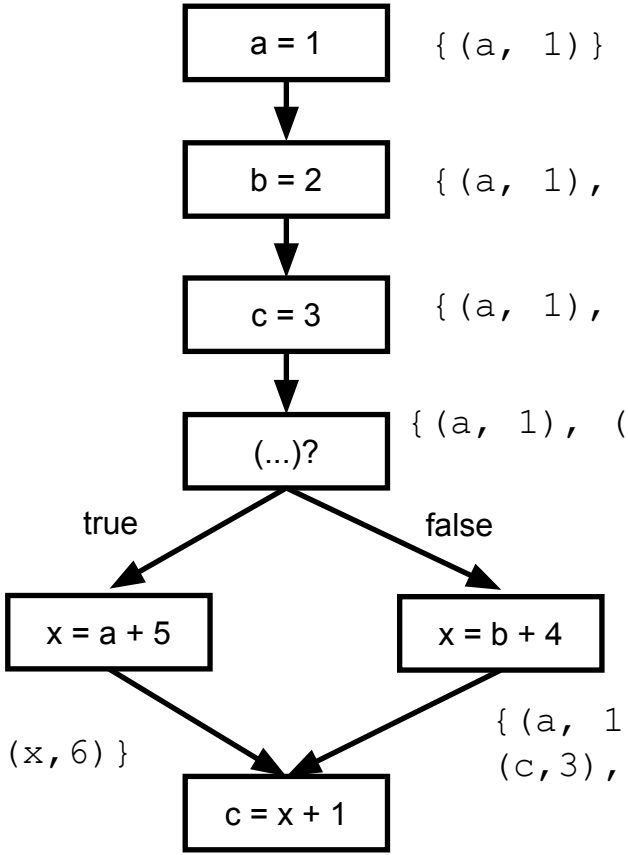

```
x = 5;
y = 1;
while (x != 1) {
  y = x * y;
  x = x - 1
}
```

Program Representations in Compiler Optimizations



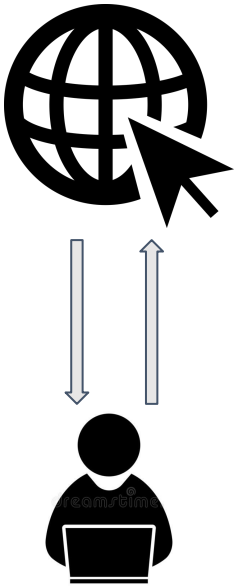
Control Flow Graphs

```
a = 1;
b = 2;
c = 3;
if (...) x = a + 5;
else x = b + 4;
c = x + 1;
```



{**(a, 1), (b,2), (c,7), (x,6)** }

Program Representations in Merge Resolution

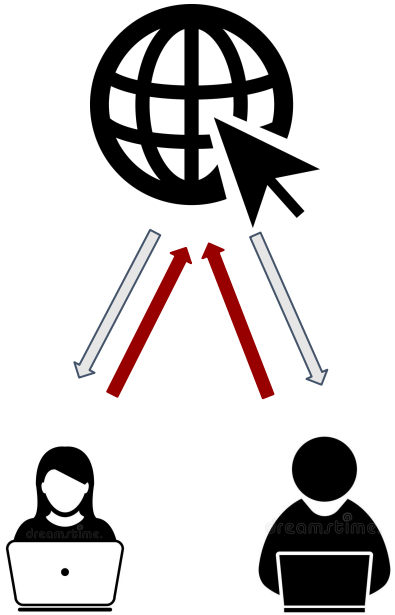


Dropbox

GitHub

Office Online

How do we integrate changes?



Dropbox

```
this.foo = function() {
<<<<<< ALICE
      Alice made some changes here
||||||| BASE
        Base contents here
=======
        Bob made some changes here
>>>>>>> BOB
};
```

Office Online

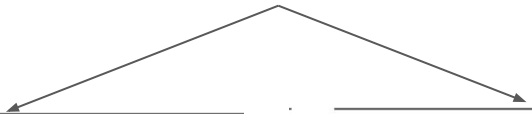
Text Based Merge Resolution

1) **Unstructured** merge - (`git merge`)

- Treats program as text. Declares a conflict if revisions modify the same textual location

Base Program STACK

```
1  import java.util.LinkedList;
2  public class Stack<T> implements Cloneable {
3    private LinkedList<T> items = new LinkedList<T>();
4    public void push(T item) {
5      items.addFirst(item);
6    }
7    public T pop() {
8      if(items.size() > 0) return items.removeFirst();
9      else return null;
10   }
11 }
```



```
1  import java.util.LinkedList;
2  public class Stack<T>
           implements Cloneable {
3    private LinkedList<T> items =
             new LinkedList<T>();
4    public void push(T item) {
5      items.addFirst(item);
6    }
7    public int size() {
8      return items.size();
9    }
10   public T pop() {
11     if(items.size() > 0) return
               items.removeFirst();
12     else return null;
13   }
14 }
```

```
1  import java.util.LinkedList;
2  public class Stack<T>
           implements Cloneable {
3    private LinkedList<T> items =
             new LinkedList<T>();
4    public void push(T item) {
5      items.addFirst(item);
6    }
7    public T top() {
8      return items.getFirst();
9    }
10   public T pop() {
11     if(items.size() > 0) return
               items.removeFirst();
12     else return null;
13   }
14 }
```

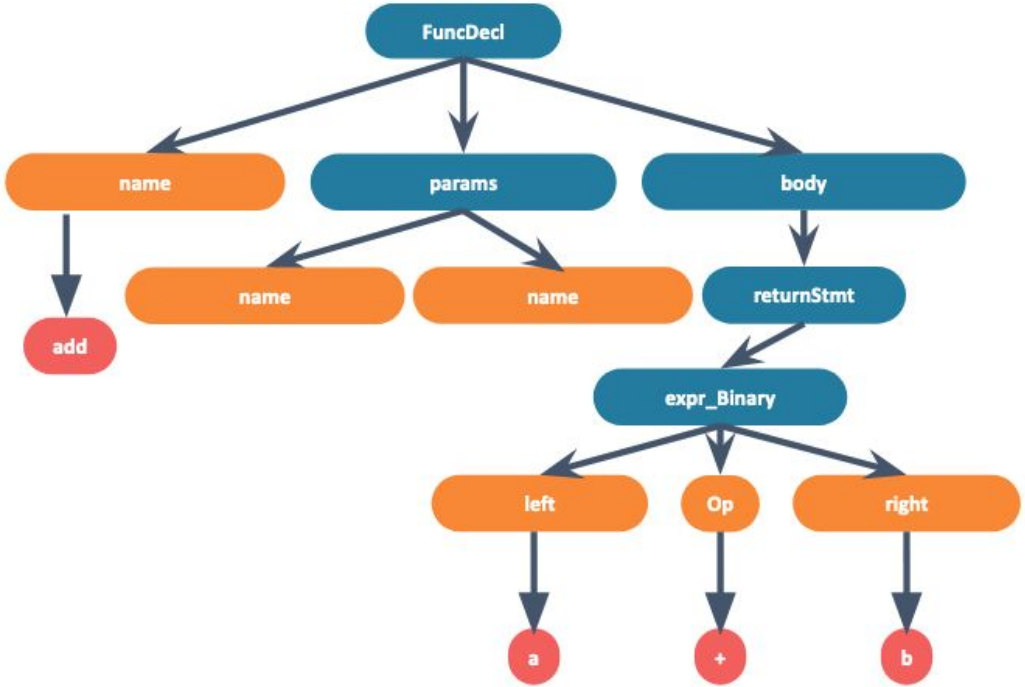
$merge_{unstructured}$(TOP, STACK, SIZE)

```
1  import java.util.LinkedList;
2  public class Stack<T> implements Cloneable {
3    private LinkedList<T> items = new LinkedList<T>();
4    public void push(T item) {
5      items.addFirst(item);
6    }
7  <<<<<<< Top/Stack.java
8    public T top() {
9      return items.getFirst();
10   }
11 =======
12   public int size() {
13     return items.size();
14   }
15 >>>>>>> Size/Stack.java
16   public T pop() {
17     if(items.size() > 0) return items.removeFirst();
18     else return null;
19   }
20 }
```

`git merge` works at the line level!

Abstract Syntax Trees

```
function add(a, b) {  return a + b; }
```

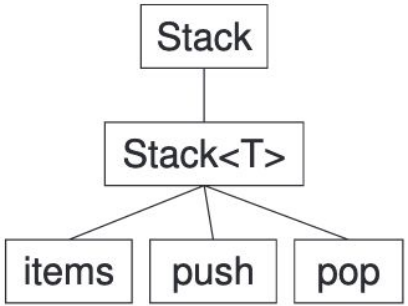


Approaches to Merge Programs

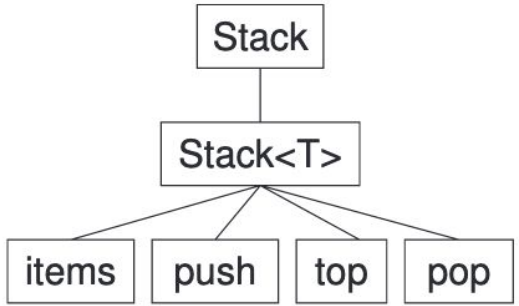
1) **Unstructured** merge - (`git merge`)

2) **(**Semi**) Structured** merge -

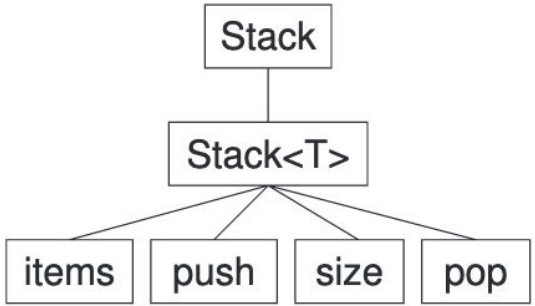
- Treats program as a graph (AST) and uses tree based merging algorithms based on the underlying language


Stack
Stack<T>
items
push
pop

BASE


Stack
Stack<T>
items
push
top
pop

A


Stack
Stack<T>
items
push
size
pop

B

The JVM and how Java code is executed

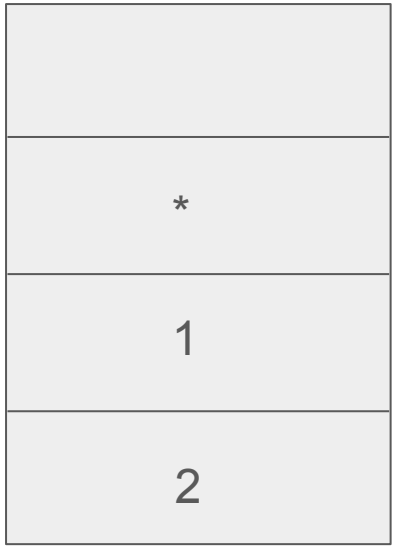
What representation does the JVM operate on?

The Java Virtual Machine

Stack Based Architecture

Most operands take values from the stack and return values back onto the stack

Ex: 2*1 (postfix calculator)



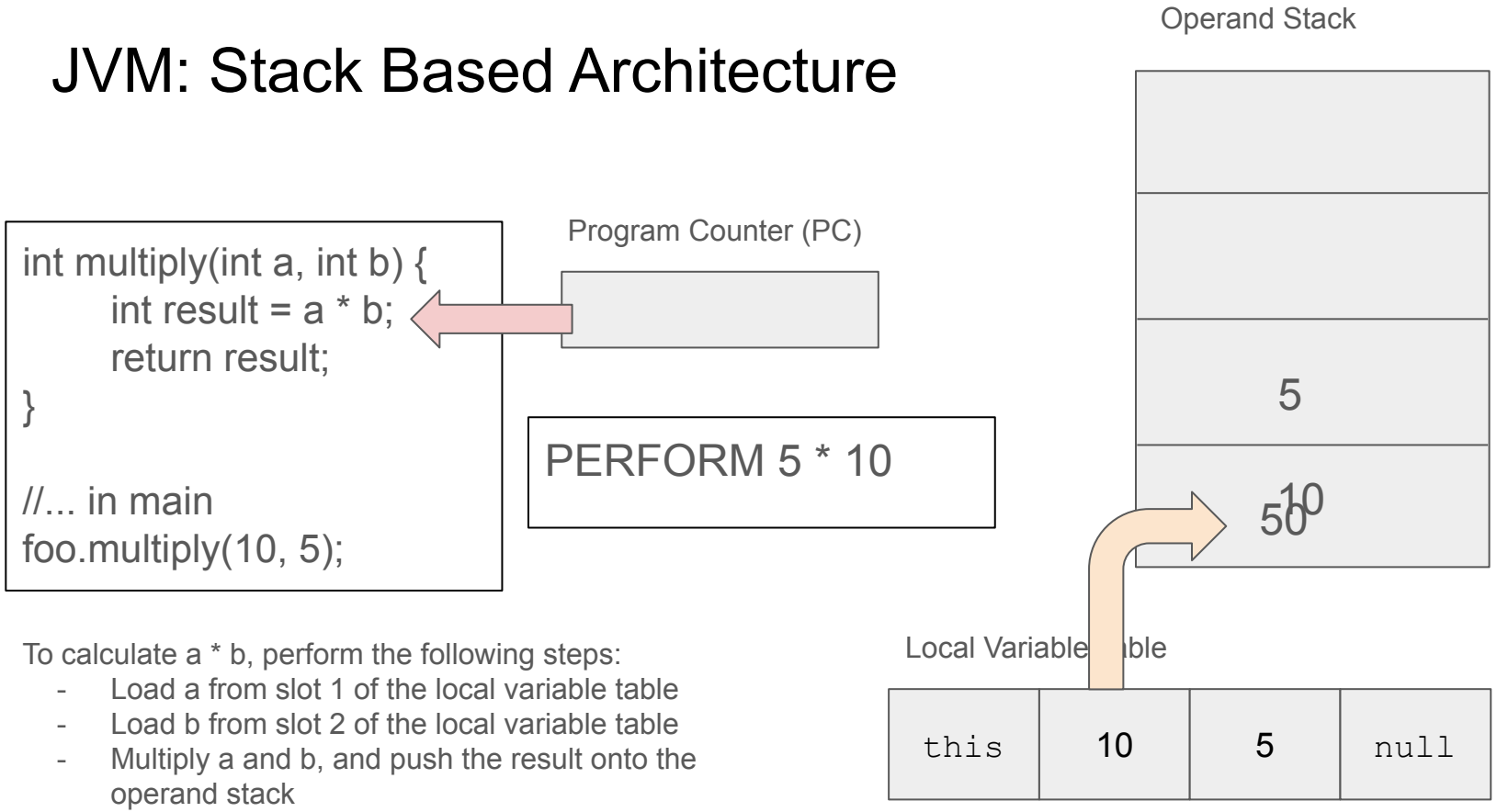
JVM: Stack Based Architecture

1. Operand stack
   a. used to store intermediate results of computations
   b. often called a "working stack"

2. Program Counter - stores the address of the next instruction
   a. increases continuously
   b. executes instructions one by one, and stores the data of the instruction execution process in the operand stack

3. Local Variable Table
   a. store variables that are in scope in the current method
   b. include method parameters and any local variables declared within the method.

JVM: Stack Based Architecture



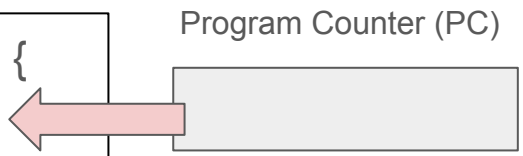
To calculate a * b, perform the following steps:

- Load a from slot 1 of the local variable table
- Load b from slot 2 of the local variable table
- Multiply a and b, and push the result onto the operand stack

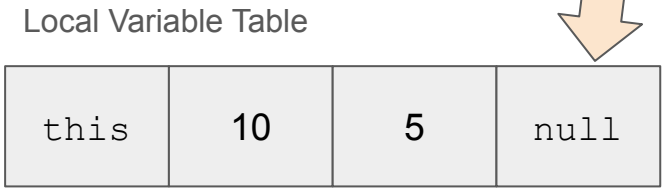
JVM: Stack Based Architecture

```
int multiply(int a, int b) {
        int result = a * b;
        return result;
}

//... in main
foo.multiply(10, 5);
```



Now, we need to store 50 in the result variable



JVM: Stack Based Architecture

We performed so many low level operations in one line!

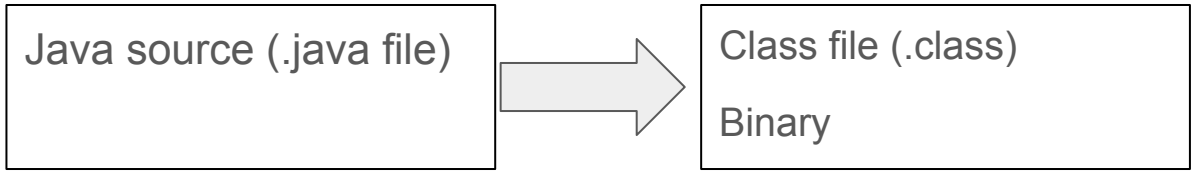
ABSTRACTION.

The programmer need not type all of the low level details. BUT the JVM functions at that level

ENTER COMPILATION - turning our high level human code into these low level operations

Compilation & Java Byte Code

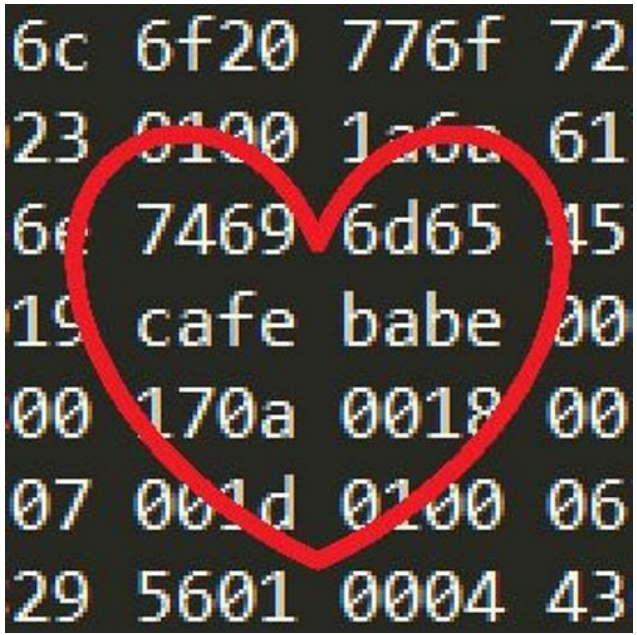
Compilation



Why does a .class file look like that?

The class file format is designed to be compact, efficient, and easy to parse by the JVM. Hex!

What's in a class?



- Magic Number
- Version Information
  - What javac version was used to compile it
  - What happens if you run java (different version) than the javac version you used?
- **Bytecode**
- Constant pool
  - Avoids need for duplication of things like Strings (would take more memory to store multiple times)
- List of all the method signatures

Java Bytecode

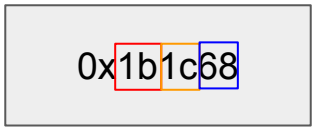
- low-level, platform-independent set of instructions that the Java Virtual Machine (JVM) executes

- **Intermediate Representation**

- How is it different from the source?
  - Optimized (constant folding)
  - Not human readable

Java Bytecode

- Java bytecode is stored in hex
- Each hex byte is two digits
  - 256 possible hex bytes
- Opcode (operation code)
- https://javaalmanac.io/bytecode/opcodes/
- https://en.wikipedia.org/wiki/List_of_Java_bytecode_instructions



1b = iload_1 (load the constant 1)
1c = iload_2 (load the constant 2)
68 = mul (multiply the previous two numbers)

Java Bytecode

Reading this in hex is not for humans!

Is there something between java source code and hex that is somewhat human readable? An *intermediate* representation.

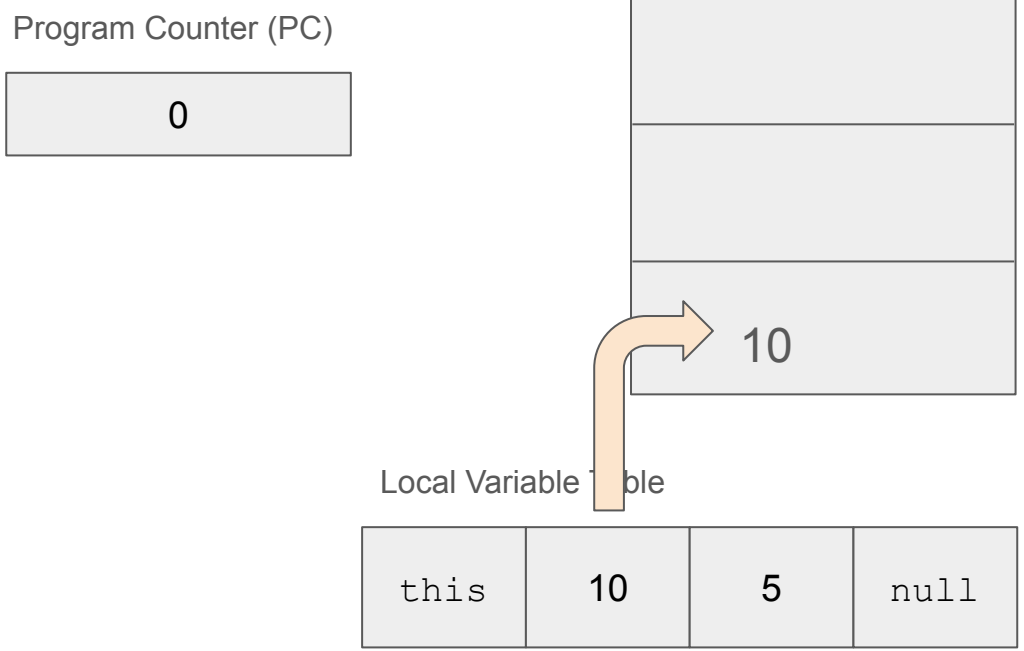
Multiply bytecode

```
int multiply(int a, int b) {
        int result = a * b;
        return result;
}
```

```
0: iload_1     // Load local variable 1 (a) onto the operand stack
1: iload_2     // Load local variable 2 (b) onto the operand stack
2: imul        // Multiply the two integers on top of the stack
3: istore_3    // Store the result (a * b) into local variable 3 (result)
4: iload_3     // Load the result from local variable 3 onto the operand stack
5: ireturn     // Return the integer result from the top of the stack
```

JVM: Stack Based Architecture

```
0: iload_1
1: iload_2
2: imul
3: istore_3
4: iload_3
5: ireturn
```



JVM: Stack Based Architecture

```
0: iload_1
1: iload_2
2: imul
3: istore_3
4: iload_3
5: ireturn
```

Program Counter (PC)

1

Operand Stack

5
10

Local Variable Table

this	10	5	null

JVM: Stack Based Architecture

```
0: iload_1
1: iload_2
2: imul
3: istore_3
4: iload_3
5: ireturn
```

Program Counter (PC)

1

Operand Stack

5
10

Local Variable Table

this	10	5	null

JVM: Stack Based Architecture

```
0: iload_1
1: iload_2
2: imul
3: istore_3
4: iload_3
5: ireturn
```

Program Counter (PC)

2

Operand Stack

50

Local Variable Table

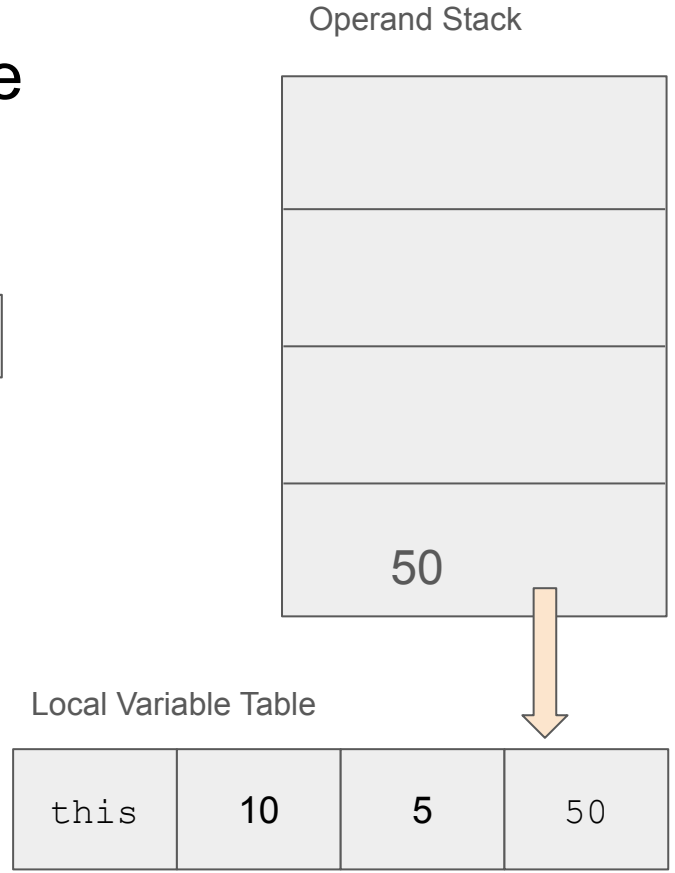
this	10	5	null

JVM: Stack Based Architecture

```
0: iload_1
1: iload_2
2: imul
3: istore_3
4: iload_3
5: ireturn
```

Program Counter (PC)

3

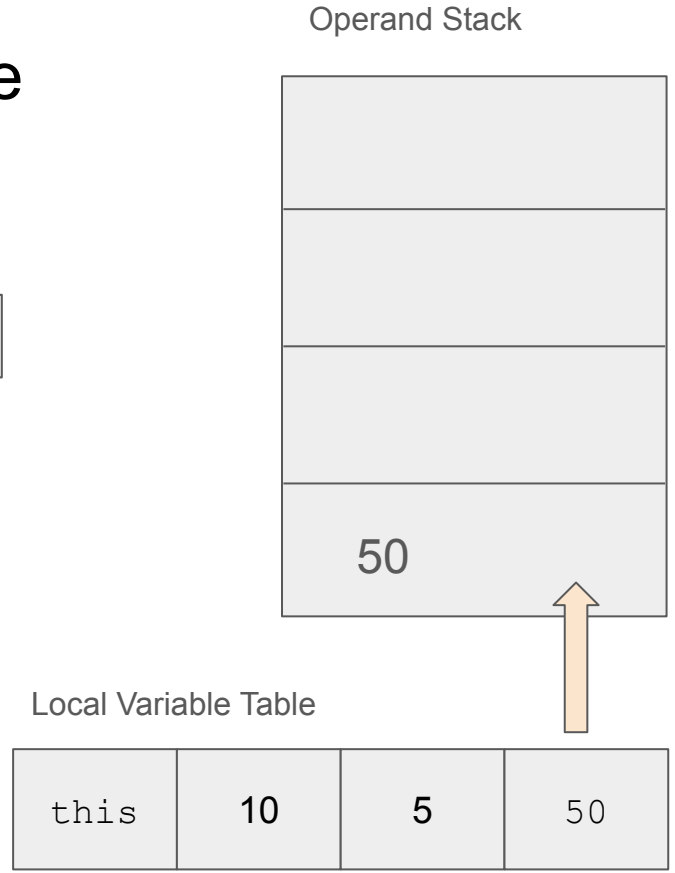


JVM: Stack Based Architecture

```
0: iload_1
1: iload_2
2: imul
3: istore_3
4: iload_3
5: ireturn
```

Program Counter (PC)

4



Multiplying doubles

```
double multiply(double a, double b) {
        double result = a * b;
        return result;
}
```

```
0: dload_1      // Load local variable 1 (a, double) onto the operand stack
1: dload_3      // Load local variable 3 (b, double) onto the operand stack
2: dmul         // Multiply the two doubles on top of the stack
3: dstore 5     // Store the result (a * b) into local variable 5 (result)
4: dload 5      // Load the result from local variable 5 onto the operand stack
5: dreturn      // Return the double result from the top of the stack
```

Example 2: Loops

```
public int sum(int n) {
        int total = 0;
        for (int i = 1; i <= n; i++) {
                total += i;
        }
        return total;
}
```

```
0: iconst_0
1: istore_2
2: iconst_1
3: istore_3
4: iload_3
5: iload_1
6: if_icmpgt          19
9: iload_2
10: iload_3
11: iadd
12: istore_2
13: iinc              3, 1
16: goto      4
19: iload_2
20: ireturn
```

```
0: iconst_0
1: istore_2
2: iconst_1
3: istore_3
4: iload_3
5: iload_1
6: if_icmpgt          19
9: iload_2
10: iload_3
11: iadd
12: istore_2
13: iinc              3, 1
16: goto      4
19: iload_2
20: ireturn
```

Program Counter (PC)

0

```
public int sum(int n) {
        int total = 0;
        for (int i = 1; i <= n; i++) {
                total += i;
        }
        return total;
}

sum(5)
```

Operand Stack

Local Variable Table

this	5	null	null

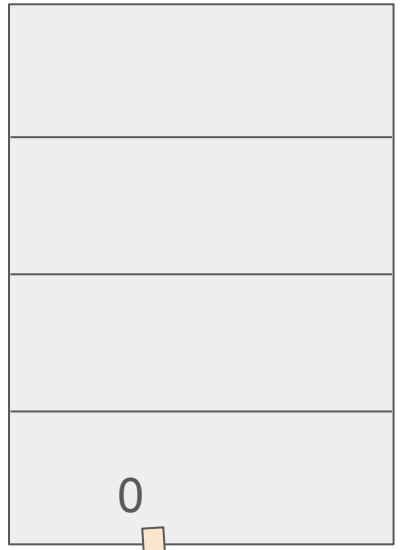
```
0: iconst_0
1: istore_2
2: iconst_1
3: istore_3
4: iload_3
5: iload_1
6: if_icmpgt        19
9: iload_2
10: iload_3
11: iadd
12: istore_2
13: iinc            3, 1
16: goto      4
19: iload_2
20: ireturn
```

Program Counter (PC)

1

```
public int sum(int n) {
        int total = 0;
        for (int i = 1; i <= n; i++) {
                total += i;
        }
        return total;
}
```

Operand Stack



Local Variable Table

```
0: iconst_0
1: istore_2
2: iconst_1
3: istore_3
4: iload_3
5: iload_1
6: if_icmpgt          19
9: iload_2
10: iload_3
11: iadd
12: istore_2
13: iinc              3, 1
16: goto      4
19: iload_2
20: ireturn
```

Program Counter (PC)

2

```
public int sum(int n) {
        int total = 0;
        for (int i = 1; i <= n; i++) {
                total += i;
        }
        return total;
}
```

Operand Stack

Local Variable Table

this	5	0	null

```
0: iconst_0
1: istore_2
2: iconst_1
3: istore_3
4: iload_3
5: iload_1
6: if_icmpgt          19
9: iload_2
10: iload_3
11: iadd
12: istore_2
13: iinc              3, 1
16: goto      4
19: iload_2
20: ireturn
```

Program Counter (PC)

3

```
public int sum(int n) {
        int total = 0;
        for (int i = 1; i <= n; i++) {
                total += i;
        }
        return total;
}
```

Operand Stack

Local Variable Table

```
0: iconst_0
1: istore_2
2: iconst_1
3: istore_3
4: iload_3
5: iload_1
6: if_icmpgt        19
9: iload_2
10: iload_3
11: iadd
12: istore_2
13: iinc            3, 1
16: goto      4
19: iload_2
20: ireturn
```

Program Counter (PC)

4

```
public int sum(int n) {
        int total = 0;
        for (int i = 1; i <= n; i++) {
                total += i;
        }
        return total;
}
```

Operand Stack

Local Variable Table

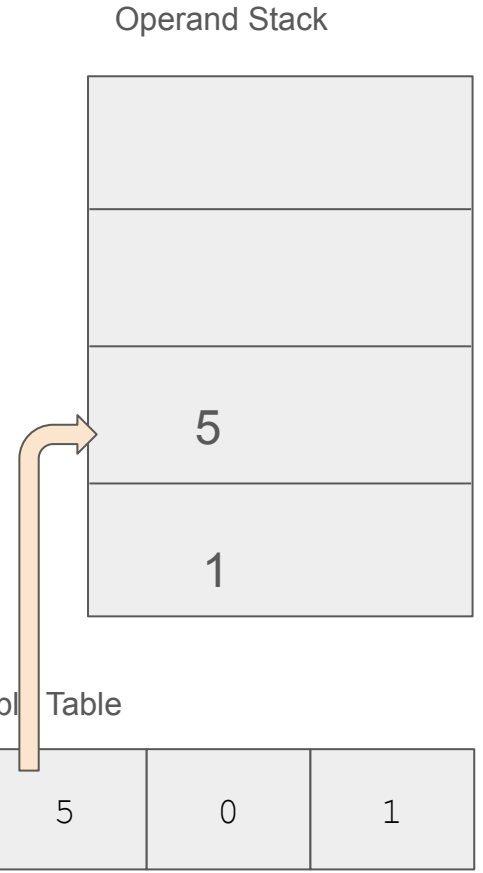
this	5	0	1

```
0: iconst_0
1: istore_2
2: iconst_1
3: istore_3
4: iload_3
5: iload_1
6: if_icmpgt        19
9: iload_2
10: iload_3
11: iadd
12: istore_2
13: iinc            3, 1
16: goto      4
19: iload_2
20: ireturn
```

Program Counter (PC)

5

```
public int sum(int n) {
        int total = 0;
        for (int i = 1; i <= n; i++) {
                total += i;
        }
        return total;
}
```

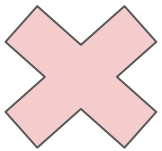


```
0: iconst_0
1: istore_2
2: iconst_1
3: istore_3
4: iload_3
5: iload_1
6: if_icmpgt      19
9: iload_2
10: iload_3
11: iadd
12: istore_2
13: iinc          3, 1
16: goto      4
19: iload_2
20: ireturn
```

Program Counter (PC)

6

```
public int sum(int n) {
        int total = 0;
        for (int i = 1; i <= n; i++) {
                total += i;
        }
        return total;
}
```



Operand Stack

Local Variable Table

this	5	0	1

```
0: iconst_0
1: istore_2
2: iconst_1
3: istore_3
4: iload_3
5: iload_1
6: if_icmpgt          19
9: iload_2
10: iload_3
11: iadd
12: istore_2
13: iinc              3, 1
16: goto      4
19: iload_2
20: ireturn
```

Program Counter (PC)


```
public int sum(int n) {
        int total = 0;
        for (int i = 1; i <= n; i++) {
                total += i;
        }
        return total;
}
```

Operand Stack

Local Variable Table

this	5	0	1

```
0: iconst_0
1: istore_2
2: iconst_1
3: istore_3
4: iload_3
5: iload_1
6: if_icmpgt        19
9: iload_2
10: iload_3
11: iadd
12: istore_2
13: iinc            3, 1
16: goto      4
19: iload_2
20: ireturn
```

Program Counter (PC)

10

```
public int sum(int n) {
        int total = 0;
        for (int i = 1; i <= n; i++) {
                total += i;
        }
        return total;
}
```

Operand Stack

Local Variable Table

this	5	0	1

```
0: iconst_0
1: istore_2
2: iconst_1
3: istore_3
4: iload_3
5: iload_1
6: if_icmpgt        19
9: iload_2
10: iload_3
11: iadd
12: istore_2
13: iinc            3, 1
16: goto      4
19: iload_2
20: ireturn
```

Program Counter (PC)

11

```
public int sum(int n) {
        int total = 0;
        for (int i = 1; i <= n; i++) {
                total += i;
        }
        return total;
}
```

Operand Stack

Local Variable Table

this	5	0	1

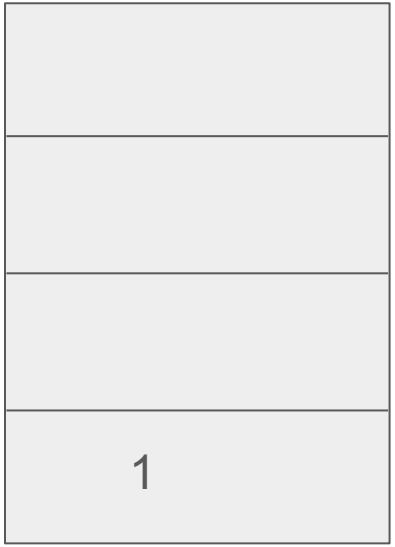
```
0: iconst_0
1: istore_2
2: iconst_1
3: istore_3
4: iload_3
5: iload_1
6: if_icmpgt        19
9: iload_2
10: iload_3
11: iadd
12: istore_2
13: iinc            3, 1
16: goto      4
19: iload_2
20: ireturn
```

Program Counter (PC)

12

```
public int sum(int n) {
        int total = 0;
        for (int i = 1; i <= n; i++) {
                total += i;
        }
        return total;
}
```

Operand Stack



Local Variable Table

this	5	1	1

```
0: iconst_0
1: istore_2
2: iconst_1
3: istore_3
4: iload_3
5: iload_1
6: if_icmpgt        19
9: iload_2
10: iload_3
11: iadd
12: istore_2
13: iinc            3, 1
16: goto      4
19: iload_2
20: ireturn
```

Program Counter (PC)

13

```
public int sum(int n) {
        int total = 0;
        for (int i = 1; i <= n; i++) {
                total += i;
        }
        return total;
}
```

Operand Stack

Local Variable Table

this	5	1	2

```
0: iconst_0
1: istore_2
2: iconst_1
3: istore_3
4: iload_3
5: iload_1
6: if_icmpgt      19
9: iload_2
10: iload_3
11: iadd
12: istore_2
13: iinc          3, 1
16: goto          4
19: iload_2
20: ireturn
```

Program Counter (PC)

16

```
public int sum(int n) {
        int total = 0;
        for (int i = 1; i <= n; i++) {
                total += i;
        }
        return total;
}
```

Operand Stack

Local Variable Table

this	5	1	2

```
0: iconst_0
1: istore_2
2: iconst_1
3: istore_3
4: iload_3
5: iload_1
6: if_icmpgt        19
9: iload_2
10: iload_3
11: iadd
12: istore_2
13: iinc            3, 1
16: goto      4
19: iload_2
20: ireturn
```

Program Counter (PC)

4

```
public int sum(int n) {
        int total = 0;
        for (int i = 1; i <= n; i++) {
                total += i;
        }
        return total;
}
```

Operand Stack

Local Variable Table

this	5	1	2

```
0: iconst_0
1: istore_2
2: iconst_1
3: istore_3
4: iload_3
5: iload_1
6: if_icmpgt      19
9: iload_2
10: iload_3
11: iadd
12: istore_2
13: iinc          3, 1
16: goto      4
19: iload_2
20: ireturn
```

Program Counter (PC)

6

```
public int sum(int n) {
        int total = 0;
        for (int i = 1; i <= n; i++) {
                total += i;
        }
        return total;
}
```

Operand Stack

Local Variable Table

this	5	4	5

```
0: iconst_0
1: istore_2
2: iconst_1
3: istore_3
4: iload_3
5: iload_1
6: if_icmpgt          19
9: iload_2
10: iload_3
11: iadd
12: istore_2
13: iinc              3, 1
16: goto      4
19: iload_2
20: ireturn
```

Program Counter (PC)

19

```
public int sum(int n) {
        int total = 0;
        for (int i = 1; i <= n; i++) {
                total += i;
        }
        return total;
}
```

Operand Stack

Local Variable Table

this	5	4	5

Method Calls

```java
public static void main(String[] args) {
    int a = 1;
    int b = 2;
    int c = calc(a, b);
}
static int calc(int a, int b) {
    return (int) Math.sqrt(Math.pow(a, 2) + Math.pow(b, 2));
}
```

```
public static void main(java.lang.String[]);
      0: iconst_1
      1: istore_1
      2: iconst_2
      3: istore_2
      4: iload_1
      5: iload_2
      6: invokestatic  #2    // calc:(II)I
      9: istore_3
      10: return
```

```
static int calc(int, int);
      0: iload_0
      1: i2d
      2: ldc2_w          #3     // double 2.0d
      5: invokestatic  #5       // java/lang/Math.pow:(DD)D
      8: iload_1
      9: i2d
      10: ldc2_w         #3     // double 2.0d
      13: invokestatic  #5      // java/lang/Math.pow:(DD)D
      16: dadd
      17: invokestatic  #6      // java/lang/Math.sqrt:(D)D
      20: d2i
      21: ireturn
```

Constant Pool

Table that stores various constants needed for execution

Includes:

- Literals
- Symbolic references: accessing an object in memory using its name (or identifier) rather than its direct memory address

`invokestatic #2` executes the static method at the second index of the constant pool

Number sign refers to the constant pool

Method Calls

```java
public static void main(String[] args) {
    int a = 1;
    int b = 2;
    int c = calc(a, b);
}
static int calc(int a, int b) {
    return (int) Math.sqrt(Math.pow(a, 2) + Math.pow(b, 2));
}
```

```
public static void main(java.lang.String[]);
      0: iconst_1
      1: istore_1
      2: iconst_2
      3: istore_2
      4: iload_1
      5: iload_2
      6: invokestatic  #2    // calc:(II)I
      9: istore_3
      10: return
```

```
static int calc(int, int);
      0:  iload_0
      1:  i2d
      2:  ldc2_w        #3      // double 2.0d
      5:  invokestatic  #5      // java/lang/Math.pow:(DD)D
      8:  iload_1
      9:  i2d
      10: ldc2_w        #3     // double 2.0d
      13: invokestatic  #5     // java/lang/Math.pow:(DD)D
      16: dadd
      17: invokestatic  #6     // java/lang/Math.sqrt:(D)D
      20: d2i
      21: ireturn
```

Non-static method calls

Method calls:

`invokevirtual` instead of **`invokestatic`**

Constructor calls (object creations):

`invokespecial`

Method Calls - nonstatic

```
class Example {
        public void greet() {
            System.out.println("hello");
        }
}

Example e = new Example();
example.greet();
```

```
0: new           #2          // Create a new Example object
3: dup                       // Duplicate the reference to the object on the operand stack
4: invokespecial  #3         // Call constructor (Example.<init>)
7: aload_1                   // Load the reference to the Example object (example)
8: invokevirtual  #4         // Call greet() on the instance of Example
11: return
```

Given this source, what is the bytecode?

```java
public class CountdownTimer {
    public static void main(String[] args) {
        CountdownTimer timer = new CountdownTimer();
        timer.startCountdown(10);
    }
    public void startCountdown(int start) {
        int count = start;
        while (count > 0) {
            System.out.println(count);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                System.err.println("Timer interrupted.");
            }
            count--;
        }
        System.out.println("Blast off!");
    }
}
```

```
public void startCountdown(int);

Code:

0: iload_1

1: istore_2

2: iload_2

3: ifle 22

6: getstatic #5 // Field java/lang/System.out:Ljava/io/PrintStream;

9: iload_2

10: invokevirtual #6 // Method java/io/PrintStream.println:(I)V

13: invokestatic #7 // Method java/lang/Thread.sleep:(J)V

16: iinc 2, -1

19: goto 2

22: getstatic #5 // Field java/lang/System.out:Ljava/io/PrintStream;

25: ldc #8 // String Blast off!

27: invokevirtual #6 // Method java/io/PrintStream.println:(Ljava/lang/String;)V

30: return
```

Given this source, what is the bytecode?

```java
public class CountdownTimer {
    public static void main(String[] args) {
        CountdownTimer timer = new CountdownTimer();
        timer.startCountdown(10);
    }
    public void startCountdown(int start) {
        int count = start;
        while (count > 0) {
            System.out.println(count);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                System.err.println("Timer interrupted.");
            }
            count--;
        }
        System.out.println("Blast off!");
    }
}
```

```
public CountdownTimer();
Code:
0: aload_0
1: invokespecial #1 // Method java/lang/Object."<init>":()V
4: return

public static void main(java.lang.String[]);
Code:
0: new #2 // class CountdownTimer
3: dup
4: invokespecial #3 // Method "<init>":()V
7: astore_1
8: aload_1
9: bipush 10
11: invokevirtual #4 // Method startCountdown:(I)V
14: return
```

`javap` command

- Tool provided with the JDK (Java Developer Kit)
- Runs on the class file (binary) to give you the information that is stored there in human readable format

```
javap -c ClassName
```

Method Descriptors

- Specify the method's return type and parameter types
- At a lower level than java source code
- Notably does not include the method name

- `()I` – Describes a method that takes no parameters and returns an `int`.

Method Descriptors

1. **Basic Types**:
   - `B` = `byte`
   - `C` = `char`
   - `D` = `double`
   - `F` = `float`
   - `I` = `int`
   - `J` = `long`
   - `L<classname>;` = an object of a class (e.g., `Ljava/lang/String;` for `String`)
   - `S` = `short`
   - `Z` = `boolean`
2. **Void Return Type**:
   - `V` is used for methods that return `void`.
3. **Array Types**:
   - For arrays, you use `[` followed by the descriptor for the element type (e.g., `[I` for an array of `int`).

Examples:

Method with one `String` parameter and `void` return type:

- `(Ljava/lang/String;)V` – A method that takes a `String` and returns `void`.

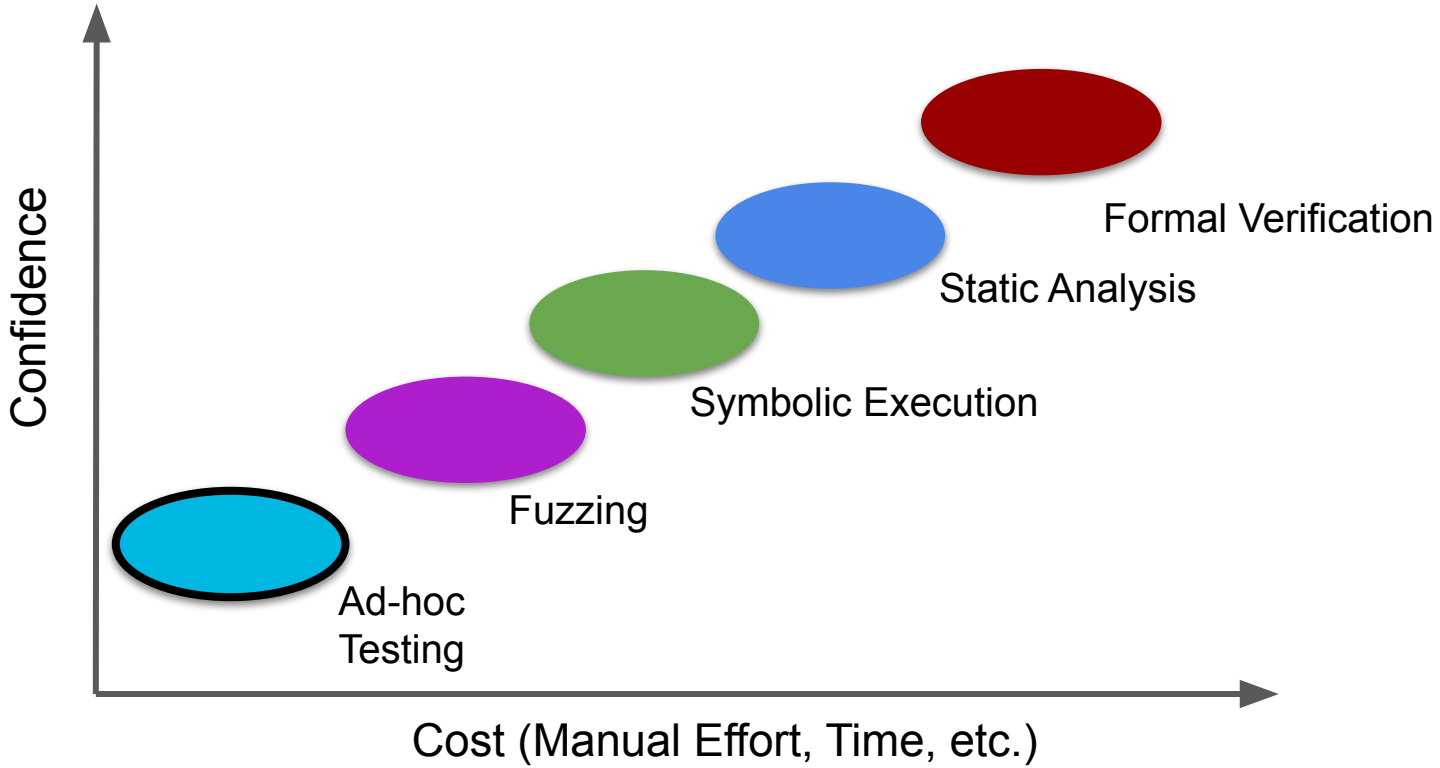
Method with two parameters: `int` and `double`, and returns `boolean`:

- `(ID)Z` – A method that takes an `int` and a `double`, and returns a `boolean`.

Method with an array of `int` and a `String` parameter, returning `void`:

- `([ILjava/lang/String;)V` – A method that takes an array of `int` and a `String`, and returns `void`.

Lab Today: Ad-hoc Testing



Lab Today

Jars and classpath

What is a jar file?

When I’m compiling a test, I need the jar file to be in the classpath

Due next Friday

Summary

- Program Representations are
- JVM operates on a stack
  - Program Counter
  - Operand Stack
  - Local Variable Table
- Java Bytecode
  - Load, store, ...
- Method Descriptors
- HW1 Released