Introduction to Program Analysis and Software Testing

CS/SWE 795, Fall 2017
Program Analysis for Software Testing

GEORGE MASON UNIVERSITY

Today

- Introductions + Logistics
- Motivation
- High level testing and test quality
- High level dynamic and static analysis
- Lab activity - Java bytecode engineering

Introductions

- Prof Jonathan Bell (me)
  - Office hour: ENGR 4422 Mon 3:30-4:30 pm or by appointment; can do Google Hangouts too.
  - Areas of research: Software Engineering, Program Analysis, Software Systems

Two hobbies: cycling, ice cream

Motivation

- Insert list of multi-million dollar bugs here
- Even in non safety-critical situations, bugs stink
  - Lose users
  - Lose money
  - Lose time fixing those bugs

How do we make more reliable software?

- Human processes designed to increase quality
- Manual processes to find bugs **(Testing**)
- Automated processes to find bugs **(Auomated Testing**)
- Semi-automated processes to prove the absence of bugs **(Verification**)

Topics in this class

- Automated techniques to create or improve existing tests
  - Measuring and improving quality of tests
  - Generating new tests
- Automated techniques to prove properties about programs

Class Structure

- Grading:
  - 30% HW (3 assignments, 1 week each)
  - 10% Reading Writeups
  - 10% Participation + In-class activities
  - 50% Project
- Web page: http://jonbell.net/swe-795-fall-17-program-analysis-for-software-testing/ (or just click “Teaching” on jonbell.net)

Class Structure

- On a normal day, we will discuss an analysis or testing topic, first with a brief summary setting the context of the topic, then talking about 1-2 research papers
- I will provide the structure for the discussion, but I expect **everyone** to read the papers and participate
- Before class, you will submit a (very short) writeup of each paper (less than half a page each, max)
- Each class will also include a lab activity, where you will apply some of the concepts we talked about that day in practice

Class Structure

- 3 homework assignments done individually, all this month
- Assignments are designed to point you in the right direction to do your project
- Term project: the majority of your grade

Term Project

- Most important part of your grade
- Proposal (end of month)
- Updates (every 2 weeks)
- Writeup (last day of class)
- Presentation (exam day)
- Projects will be intensive
- Individual or pair
- Unless there is a strong compelling reason otherwise, everything is in Java

Term Project Tracks

- Research track:
  - Improve on state-of-the-art, create a prototype, ends with a project that could lead to academic publication
- Industry
  - Transfer the state-of-the-art to practice: enhance an existing open source tool, ends with pull request/patch to include your changes in the tool

Fair Warnings

- This class is likely very different from any class you have taken
    - Even if you've taken seminars
- A LOT of programming is expected of you
    - Including working with existing open source tools and technologies in ways you likely have never interacted with before
- BUT: you will end up learning a lot, and having built a project that will be a great discussion point applying to grad school, or topic of a paper if you are already in a PhD, or in an interview looking for a job

Repeated Warning

- You will be pushed in this class, and will program a lot
- Common question: "Am I qualified for this class?"
- Today we'll do a programming activity in class, and we'll talk about the first homework, due 9/5 (conveniently, the date to drop this class with no financial penalty)
  - If you are unable to do this assignment, it is not very likely you will succeed in this class
  - If you find this assignment challenging, but enjoyed doing it, then this is for you!
  - If you find this assignment easy, then let me know, I have more :)

Project Overview

- OK, so what is the project?
- We'll refine topics over the course of this month, but here are some samples:
- Industry track:
    - Enhance the Maven's Surefire testing framework support for rerunning failed tests to rerun each failed test in its own JVM, maintaining the architectural design of Surefire, and submit this as a new feature
- Research track:
    - Enhance an automated input generation tool like EvoSuite to collect and solve path constraints to systematically explore more input spaces

Confusing Registration Stuff

- If you are a CS student, register for CS 795
- If you are a SWE or IT student, register for SWE 795
- The classes are identical, you are sitting in both of them right now. The distinction is just for satisfying your program requirements.

Pre-class survey

- To gather some basic demographics about you and your interests
- Even if you are not planning to stay in the class, please fill this out
- Go to: http://b.socrative.com and put in CS795 as the class name

This lecture: Overview

- What do we test for?
- How do we write those tests?
- When are we done testing?

What do we test for

- General properties
  - Division by zero
  - Failed assertions
  - Uncaught exceptions
  - Memory bugs
  - Concurrency bugs
  - Termination bugs
- Functional properties
  - Mistakes in translating specs to code

Functional vs General Properties

- General properties are easy to check automatically
  - If you can run the program and witness one of these properties fail, you found a bug!
- Functional properties are difficult to automatically check
  - Need a specification!
- In practice, we write test cases that express functional properties and general properties
- Automated tools can help check general properties, not so much with functional properties

Other properties

- Liveness
    - E.g. system will eventually process all inputs
- Performance
    - E.g. system will process all inputs within a given time bound

How do we write tests?

- In Java land: JUnit

```
@Test
public void testMyFunction(){
    assertEquals(3,myFunction();
}
```

How do we run those tests?

- JUnit is primarily a framework for writing tests, and running individual tests
- JUnit itself doesn't have much support for running several tests at once, collecting the output, etc
- We use build tools, like maven, ant, and gradle to run tests and collect the output

```
$ mvn test
<snip>
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running CanaryTest
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.033 sec
Running FlakyTest
Tests run: 7, Failures: 2, Errors: 0, Skipped: 0, Time elapsed: 0.003 sec <<< FAILURE
```

How do we write tests?

- Specifications can be expressed using Java Modeling Language (JML, not often used in practice)
- Then, these specifications can be checked at runtime, or perhaps proven to always hold true

```
//@ requires 0 < amount && amount + balance < MAX_BALANCE;
//@ assignable balance;
//@ ensures balance == \old(balance) + amount;
public void credit(final int amount)
{
  this.balance += amount;
}
```

Automatically checking general properties

- Maybe we can automatically find **bugs** that are related to these general properties ("No input should exist that causes my program to crash")
- Or, maybe we can automatically prove that **no bugs** could exist regarding some property ("A null pointer exception is not possible")
- Two high level approaches:
    - Static Analysis
    - Dynamic Analysis

Dynamic Analysis

- Runs the code we are studying, and collects additional information about that execution
- Advantages:
  - Precise - NO false alarms (if we witness some input cause a violation, we know that violation can really occur, because we saw it)
- Disadvantages:
  - Generally hard to analyze a single component in isolation
  - Quality of analysis is tied to the quality of the inputs you used to drive it

Dynamic Analysis Examples

- Many used in practice:
  - Detect memory leaks (Valgrind)
  - Detect data races (Eraser)
  - Check array bounds (Purify)
  - Find likely invariants (Daikon)
- Other analyses we will discuss:
  - Dynamic symbollic execution
  - Control-flow, data flow integrity

Static Analysis

- Reason about a program without running it
- Advantages:
  - Does not require inputs
  - Can **prove** things about programs
  - Can be applied to code fragments, not just full apps
- Disadvantages:
  - Typically lots of false positives - to reason about things quickly, lots of assumptions are made that are "safe" but imprecise

Example Analysis: Detect Program Invariants

```
public int doStuff(int in)
{
  int z;
  z = 10;
  return z;
}
```

Invariant: At return statement, z = 10

Example Analysis: Detect Program Invariants

- Why does this matter?

```
public int doStuff(int in)
{
  int z;
  z = 10;
  return z;
}
```

```
public int stupidCaller(int x)
{
  if(doStuff(x) < 5)
    crashAndExplode();
  else
    doNiceThings();
}
```

Nice to know, crashAndExplode() won't be called

Example Analysis: Detect Program Invariants

- Of course, it's not that easy usually (assume magic is some complex function in some library somewhere we can call)

```
public int doStuff(int in)
{
  int z = 2;
  if(magic(in) > 0)
    z = 10;
  if (in > 5)
    z = z * 5;
  return z;
}
```

Can we discover potential invariants that might hold about the return value of this function?

Dynamically Detect Likely Program Invariants

- "Daikon" tool by Michael Ernst
- Idea: Observe execution of program, start off by assuming that any invariant is possible, then cross off the ones that are violated
- What remains *might* be an invariant

```
public int doStuff(int in)
{
  int z = 2;
  if(magic(in) > 0)
    z = 10;
  if (in > 5)
    z = z * 5;
  return z;
}
```

Executions

i	z
0	10
4	10
9	10

Possible Invariants

~~z=0~~

~~z=5~~

~~z=-1~~

z=10?

Statically Proving Invariants

- Well... let's look at the code
- Reasoning: Can do some simple constant propagation (magic always returns 1), then clearly reduce doStuff to always return 10

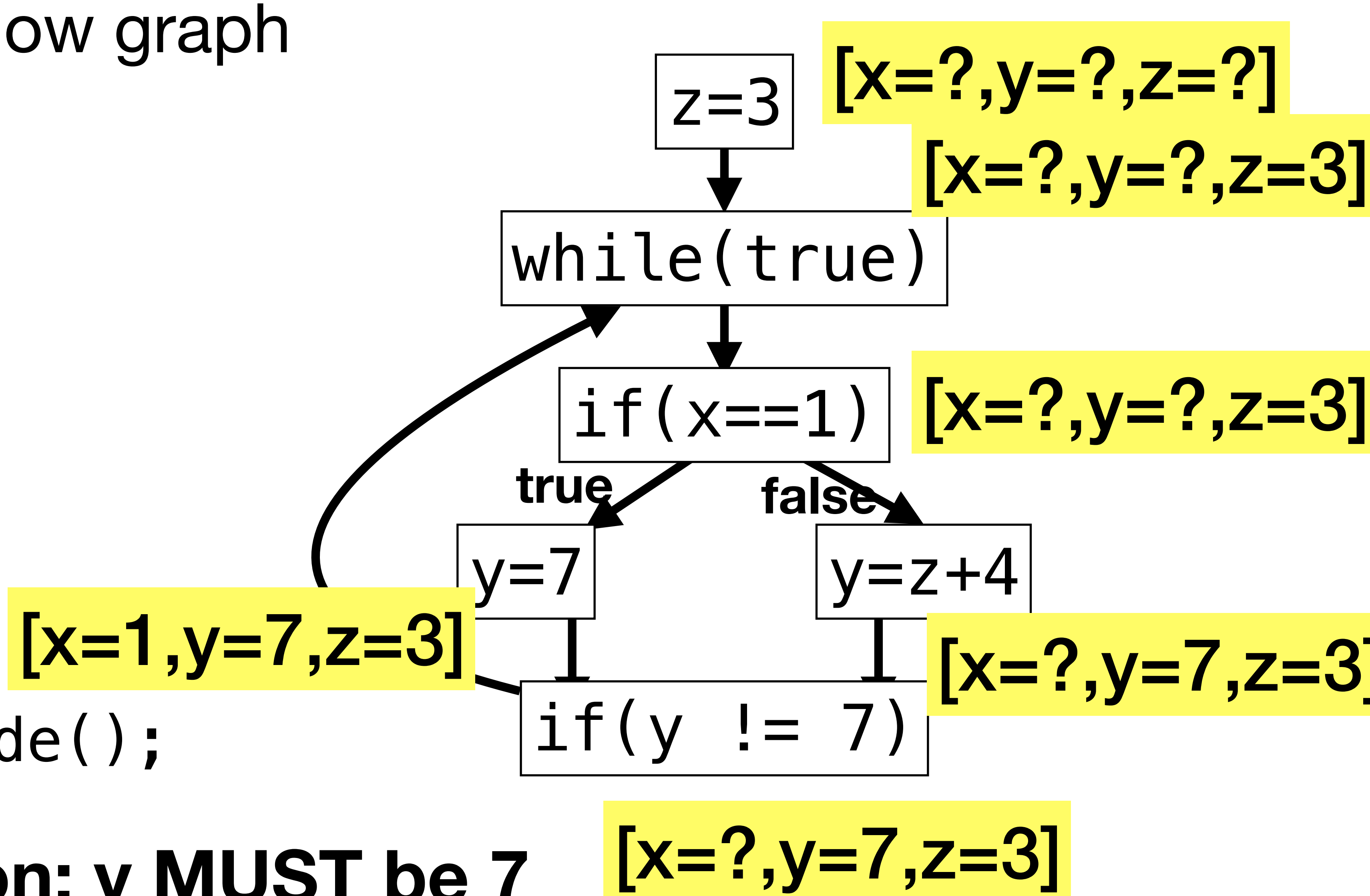
```
public int doStuff(int in)
{
  int z = 2;
  if(magic(in) > 0)
    z = 10;
  if (in > 5)
    z = z * 5;
  return z;
}
```

```
public int magic(int x)
  {
    return 1;
  }
```

Statically Proving Invariants

- What if want to PROVE that this invariant ALWAYS holds, but it's more complicated?
- Static analysis operates on some abstraction of the code, such as a control flow graph

```
void magic()
  {
    int z = 3;
    while(true){
      if(x==1)
        y = 7;
      else
        y = z + 4;
      if(y != 7)
        crashAndExplode();
    }
  }
```



Conclusion: y MUST be 7

Soundness and Completeness

- Completeness - Underapproximates information about the program
  - Static analysis is often NOT complete (dynamic analysis is)
  - Any violation of some property reported is guaranteed to be a true violation
- Soundness - Knowing that your conclusions hold for all cases
  - Guaranteed to identify all violations of some property
  - Static analysis generally IS sound
  - Dynamic analysis is NOT sound

Undecidability

- Can we have a static analysis that is both **sound** and **complete**?
- Not if you want it to terminate
- Rationale (halting problem):
  - "Will this program terminate?"
  - If I look at it really hard, I can't tell
  - So I have to run it
  - But if I run it, and it doesn't terminate, I won't know
  - Any program analysis might have this problem embedded in it, and hence be undecidable

So, which do we pick?

- If I want to make a compiler optimization, it needs to be **sound**, so maybe it will be static
- If I want to make a program verifier, it needs to be **sound**, so maybe it will be static too
- If I want to make a bug finder…
    - Static: Might get false alarms, drown a developer in warnings and they abandon the whole thing
    - Dynamic: Might miss bugs, not actually be very helpful

So, which do we pick?

- We’ll consider both static and dynamic analysis in this class
- However, mostly we will be talking about dynamic analysis
- Our goal is to try to find as many bugs as possible - incomplete is OK if the warnings we provide are good
- Cool study talking about why static analysis tools are often not used in practice: “Why don't software developers use static analysis tools to find bugs?” by Johnson et al, ICSE 2013

Are we done yet?

- OK, so we wrote some tests, used some analysis tools
- Are we done testing?
- Two high level approaches to measuring the “quality” of a test suite:
  - Coverage
  - Mutation Analysis

Code Coverage

- Some metric that quantifies the proportion of code tested by a given test suite
- Generally some percentage, 0-100%
- Many different kinds of criteria can be used for defining coverage:
  - Class: Which class files were used?
  - Method: Which methods were used?
  - Statement: Which lines were executed?
  - Branch: Which branches were taken?
  - Path: What paths through the program were taken?
  - More

Cheap to compute, Easy to satisfy

↓

Expensive to compute, hard to satisfy

Code Coverage

- Reminder: 100% statement coverage does not imply fully tested

```
public int magic(int in)
{
    int[] ar = new int[5];
    for(int i = 0; i < in; i++)
    {
        ar[i] = i*2;
    }
}
```

```
void testMagic()
{
    assert(magic(2));
}
```

Code Coverage

- How do people use coverage?
  - In high-assurance fields: might have minimum coverage required
- In general: ???
  - Require that new code be covered?
  - Require that new commits don't reduce coverage?
  - Some startups in this area, like https://coveralls.io

Mutation Analysis

- Intuition: A test suite is good if it can find all of the bugs
- But how do we know if we are finding all of the bugs?
- Idea: What if we know where all of the bugs are, because we put them there?
- Mutation analysis: Introduce “mutants” (which hopefully represent real bugs), then see if your test suite labels these as bugs

Mutation Analysis

- Core principle is that most bugs can be easily represented by a simple change
  - For example, replace a + with a -, or a < with a >=
- Run each mutant on each test until some test fails (“kills the mutant”)
- Calculate the mutation score (Number of “killed mutants”)/(Total number of mutants)

Mutation Analysis Example

```
public void credit(final int amount)
{
  this.balance += amount;
}
```

Original

```
public void credit(final int amount)
{
  this.balance -= amount;
}
```

Mutant

```
public void credit(final int amount)
{
  this.balance *= amount;
}
```

Mutant

Mutation Analysis Challenges

- What if the mutation causes the program to be equivalent to the original program?
- Then, no test will kill it
- Problems:
  - 1 - How do I know if I didn't kill a mutant because my tests are bad, or it's just an equivalent mutant
  - 2 - Ends up causing us to run LOTS of mutant-test pairs
- Proving that two programs are equivalent is undecidable in the general case

Equivalent Mutants

```java
public int myFunction()
{
    int i = 0;
    while(true){
        i++;
        if(i == 10)
            break;
    }
}
```

Original

```java
public int myFunction()
{
    int i = 0;
    while(true){
        i++;
        if(i >= 10)
            break;
    }
}
```

Mutant

Other topics with testing

- Maintenance
- What happens when I have too many tests?
  - Selection
  - Minimization
  - Prioritization
- Augmentation

Making it work: Instrumenting Code

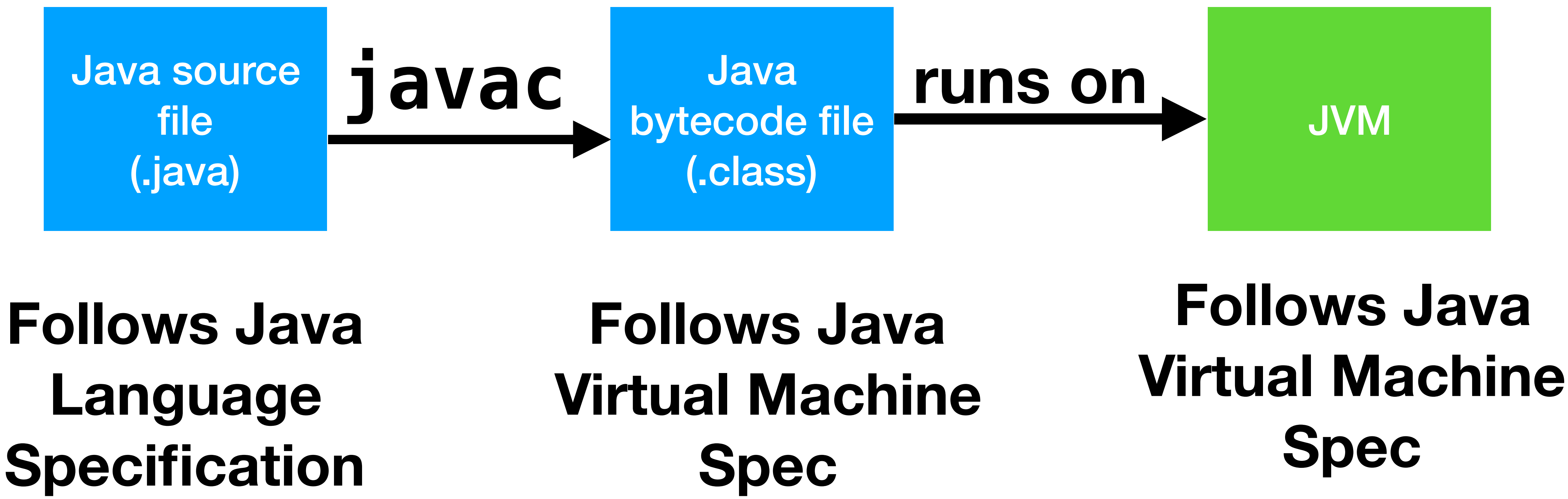
```
public int doStuff(int in)
{
1:   int z;
2:   z = 10;
3:   return z;
}
```

```
public int doStuff(int in)
{
     Logger.logLineCovered("example","doStuff",1);
1:   int z;
     Logger.logLineCovered("example","doStuff",2);
2:   z = 10;
     Logger.logLineCovered("example","doStuff",3);
3:   return z;
}
```

Making it work: Instrumenting Code

- Option 1: Modify the source code
- Option 2: Modify the binaries
- Option 3: Customize some runtime that the code runs in (e.g. special JVM, special hardware emulator)
- Of course, there are tools available for various languages to make this easier

Java Bytecode



Java vs Bytecode

```java
public int doStuff(int in)
{
    int z;
    z = 10;
    return z;
}
```

javac (compile)
javap (disassemble)

```
public int doStuff(int);
  descriptor: (I)I
  flags: ACC_PUBLIC
  Code:
    stack=1, locals=3, args_size=2
       0: bipush        10
       2: istore_2
       3: iload_2
       4: ireturn
    LineNumberTable:
      line 5: 0
      line 6: 3
```

Tip: Inspect compiled bytecode by using the javap utility, with the flags -private and -verbose

Java Bytecode

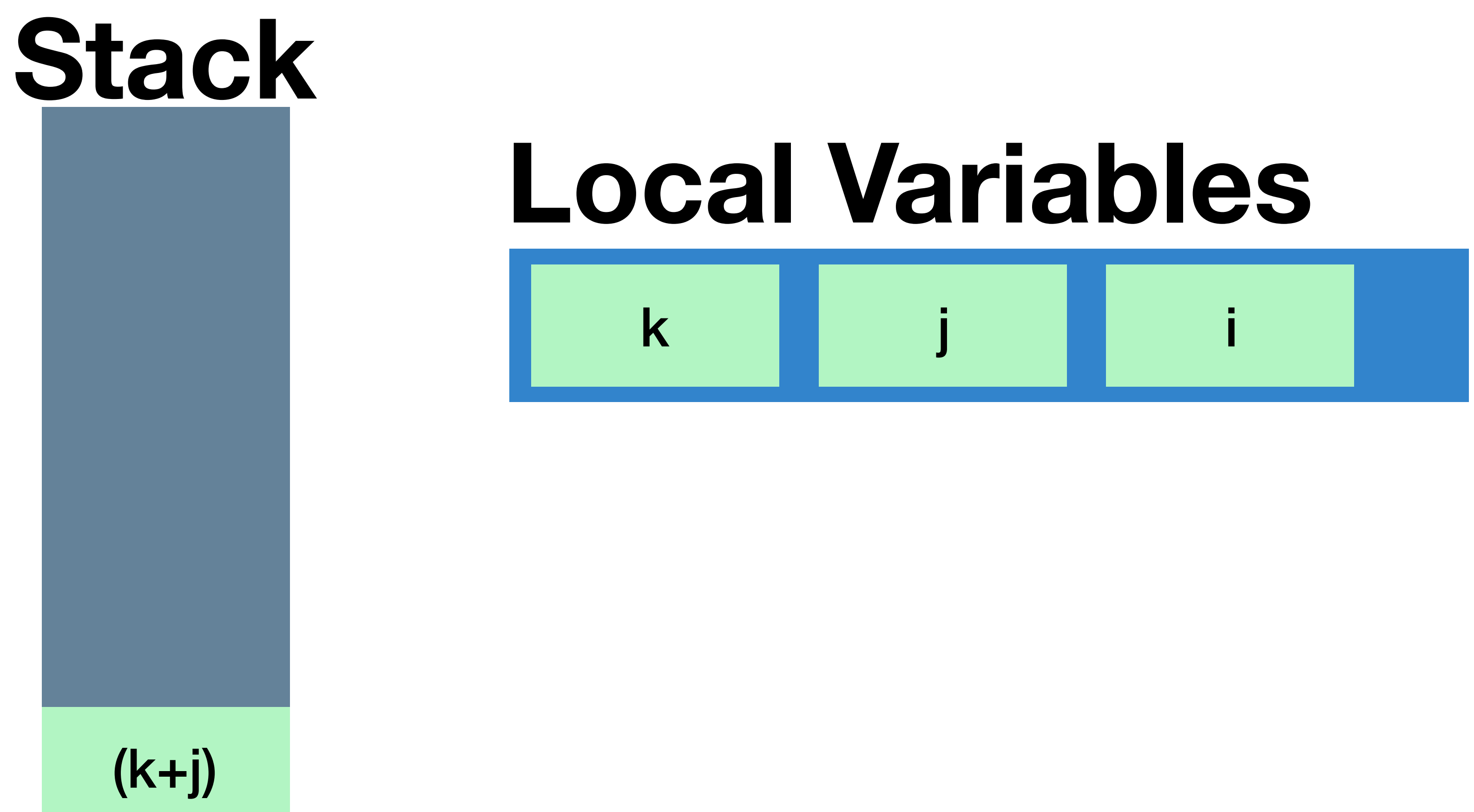
- Stack-based machine
- Example: `int i = j + k;`

```
ILOAD 1 //Or whatever index k is
ILOAD 2 //Or whatever index j is
IADD
ISTORE 3 //Or whatever index i is
```

Operand Stack

Current Instruction

```
ILOAD 1 //Or whatever index k is
ILOAD 2 //Or whatever index j is
IADD
ISTORE 3 //Or whatever index i is
```



Bytecode Operator Categories

- Arrays
- Local Variables
- Fields
- Method invocation
- Arithmetic
- Constant loading (e.g. put 0 on the stack)
- Stack manipulation (copy, pop, etc)
- Jumps
- Other miscellaneous
- https://en.wikipedia.org/wiki/Java_bytecode_instruction_listings
- https://docs.oracle.com/javase/specs/jvms/se8/html/index.html

Types in Bytecode

Java Type	Representation in bytecode
int	I
short	S
long	J
float	F
double	D
byte	B
boolean	Z
char	C
Array (e.g. int)	[I
foo.bar.Class	Lfoo/bar/Class;

ASM Bytecode Manipulation Library

- ASM is a lightweight and well-supported library for manipulating java byte code, hiding lots of the dirty work
- Provides two high level APIs: “visitor” and “tree”
- We will only discuss the visitor API
- Visitor API gives you the ability to emit some additional (or less) instructions, looking at one instruction at a time
- Two high level classes we need: ClassVisitor, MethodVisitor

ClassVisitor API

`visit(int version, int access, String name, String signature, String superName, String[] interfaces)`

`visitAnnotation(String desc, boolean visible)`
Visits an annotation of the class.

`visitAttribute(Attribute attr)`
Visits a non standard attribute of the class.

`visitEnd()`
Visits the end of the class.

`visitField(int access, String name, String desc, String signature, Object value)`

`visitInnerClass(String name, String outerName, String innerName, int access)`

`visitMethod(int access, String name, String desc, String signature, String[] exceptions)`

`visitOuterClass(String owner, String name, String desc)`
Visits the enclosing class of the class.

`visitSource(String source, String debug)`
Visits the source of the class.

`visitTypeAnnotation(int typeRef, TypePath typePath, String desc, boolean visible)`

MethodVisitor API

- Way more functions here, check out the full list in the documentation - http://asm.ow2.org/asm50/javadoc/user/org/objectweb/asm/MethodVisitor.html
- Some examples:

`visitCode()`
Starts the visit of the method's code, if any (i.e. non abstract method).

`visitInsn(int opcode)`
Visits a zero operand instruction.

`visitFieldInsn(int opcode, String owner, String name, String desc)`
Visits a field instruction.

Types in ASM

- ASM provides a handy utility class to convert between the different formats that are used to represent a type in java bytecode, a class called Type
- Formats supported are:
  - Type descriptor (object classes are in format Lclass/Name;), used for the type of a field, method parameter, etc
  - Internal Name - only for object classes, (in format class/Name), used for type instructions (new array), and to refer to the owner of a field, method, etc.

Reading and Writing .class files

```
ClassReader cr = new ClassReader(new FileInputStream(clazz));
ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_MAXS);
cr.accept(cw, 0);
byte[] instrumentedClass = cw.toByteArray();
```

Read a class file in (name clazz), write it back out to byte[]

```
ClassReader cr = new ClassReader(new FileInputStream(clazz));
ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_MAXS);
ClassVisitor cv = new MethodProfilingCV(cw);
cr.accept(cv, 0);
byte[] instrumentedClass = cw.toByteArray();
```

Read a class file in (name clazz), write it back out to byte[], instrumenting it along the way

Reading and Writing .class files

- Can do this either before anything runs (“offline”): instrument a whole bunch of files, then run those instrumented files, or:
- Can do this “online” as files are loaded by Java. This is done using a “javaagent” that has a “Premain” class, which intercepts classes as they are loaded

Lab: Bytecode Instrumentation

- We’ll go through all of this ASM stuff
- Two high level tasks:
    - 1 - Get started with ASM, play with ASMifier, inject some simple code (uses the “offline” instrument approach)
    - 2 - Trace what methods get called (uses the “online” instrument approach)