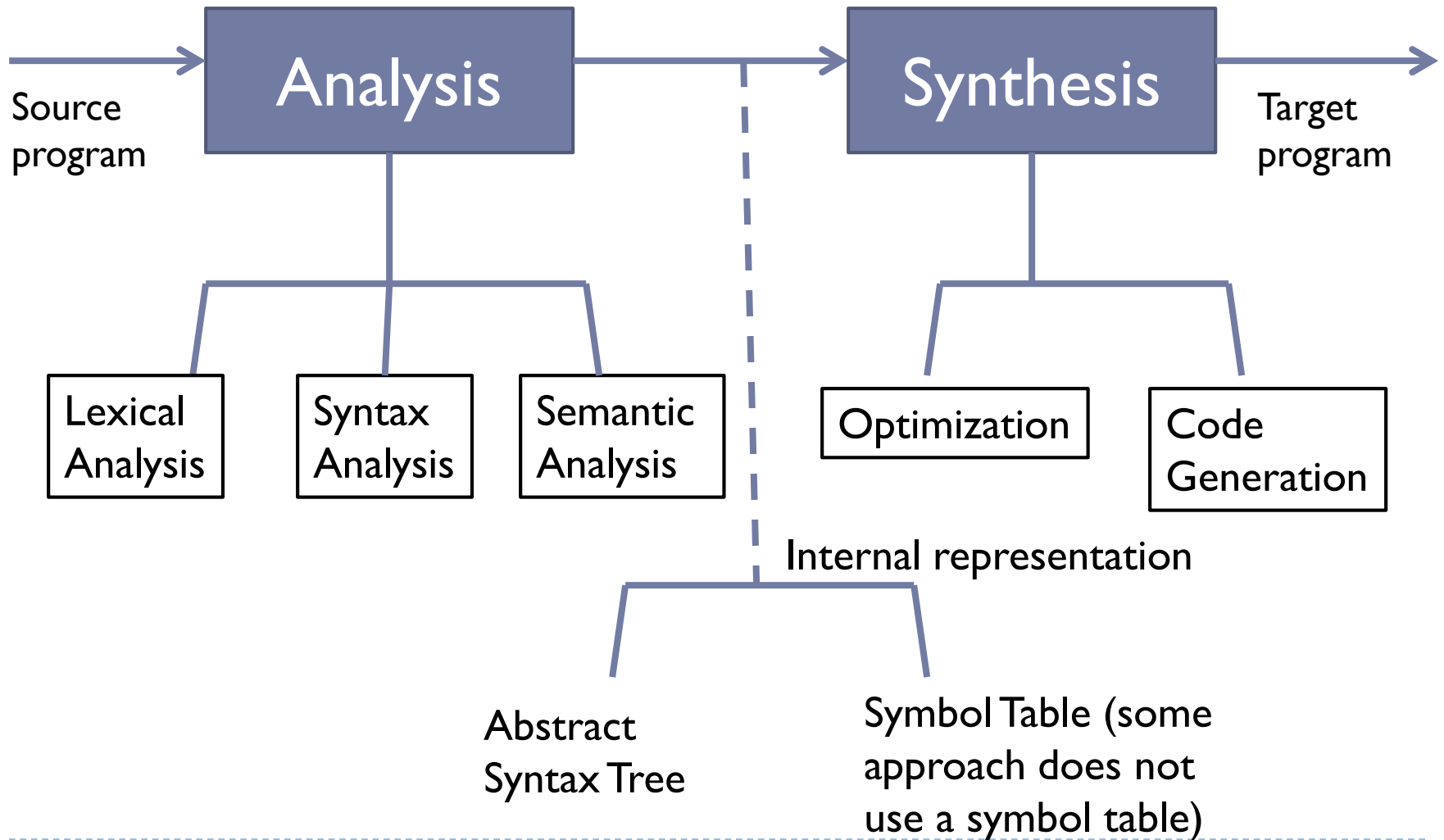


Compiler Techniques

5. Code Generation

Ta N. B. Duong

The Structure of a Compiler



Overview

- ▶ The code generator produces executable code from an abstract syntax tree representation of the program



- ▶ The code generator may also perform *optimisations* to make the generated code smaller or faster: see next chapter

Intermediate Representations

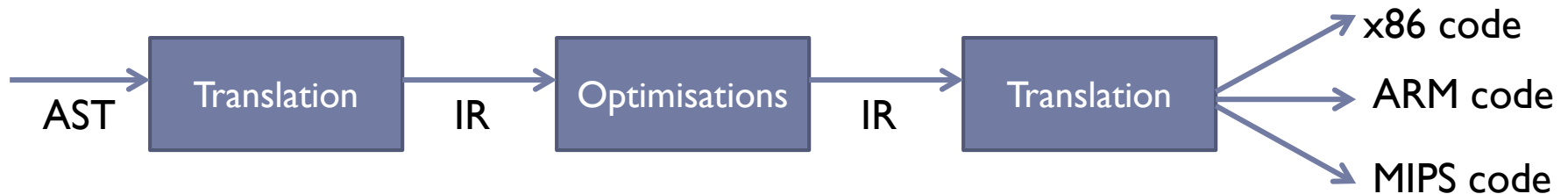
- ▶ The task of code generation is often split up into several different passes
- ▶ Each pass may perform a different optimisation or transformation
- ▶ The passes communicate using one or more *intermediate representations* (IRs):



- ▶ In a sense, the AST and the executable code are just the first and the last “intermediate” representation

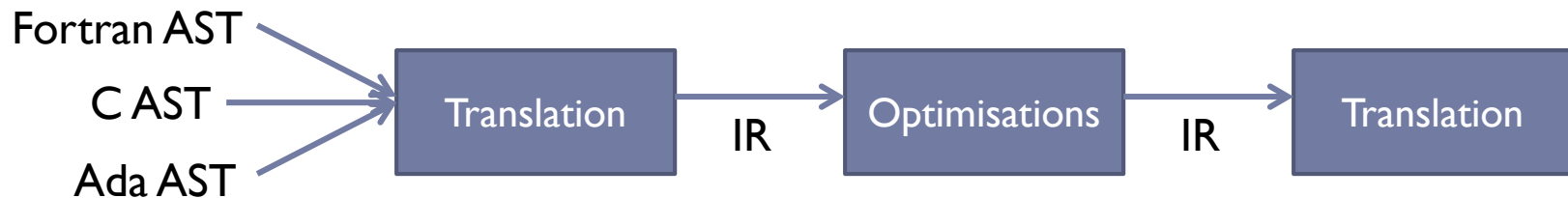
Why Intermediate Representations?

- ▶ Translating directly from the AST to executable code can sometimes be difficult, having an IR allows the compiler to break up the transformation into simpler components
- ▶ Some transformations require a certain intermediate representation to work
- ▶ Some compilers (such as gcc or llvm) support multiple different target platforms. Platform-independent optimisations are done on an IR, then the optimised IR is converted to native executable code:



Why Intermediate Representations? (2)

- ▶ Conversely, some compilers (again, like gcc) support multiple different input languages; the ASTs for these languages are all converted into the same intermediate representation, so optimisations can be shared:



Virtual Machines

- ▶ We can go one step further, and just have the compiler produce a platform-independent IR as its result (instead of platform-specific native code)
- ▶ Of course, we then also need an interpreter to run this IR on different platforms
- ▶ The advantage is that many different compilers can target this IR and do not have to generate code for different platforms
- ▶ Such IRs are known as *bytecodes*, and the interpreters running them as *virtual machines*
- ▶ Examples: Java Virtual Machine (Sun/Oracle), Common Language Runtime (Microsoft), Dalvik (Google)

Compiling to Bytecode vs. Executable Code

- ▶ **Advantages of compiling to bytecode:**
 - ▶ Only need to generate code for one platform: the virtual machine
 - ▶ Native instruction sets emphasise efficient execution over ease of compilation, so code generation is difficult
 - ▶ Virtual machine instruction sets and runtime environments are often more high-level than their native counterparts: e.g. both the JVM and the CLR offer automated garbage collection
 - ▶ Virtual machine optimisations benefit every compiler targeting it
 - ▶ Bytecode is often more compact than native code: particularly suitable for resource-constrained platforms (e.g. mobile apps on Android phones)
- ▶ **Advantages of compiling to native code:**
 - ▶ Running code on a virtual machine is, in general, slower than running native code

Just-in-Time Compilation

- ▶ Modern virtual machines are rarely pure interpreters
- ▶ Usually, they compile bytecode to native code on-the-fly; this is known as “just-in-time” compilation (a.k.a. JIT compilation)
- ▶ Trade-off: native code is faster, but compilation takes time
- ▶ Often, only very heavily used (“hot”) parts of the program are compiled and cached, the rest is interpreted
 - ▶ Once a method has been executed several times, it is compiled to native code - code that can be directly executed by the underlying computer
 - ▶ The compiled native code is cached so that it may be re-used in subsequent invocations of the method

Just-in-Time Compilation

- ▶ Modern JIT compilers also exploit runtime profiling for optimisation, hence they can perform more targeted optimisations than compilers targeting native code directly
- ▶ Consequently, JIT-based virtual machines are now competitive with native code for most application areas except heavy numeric computations

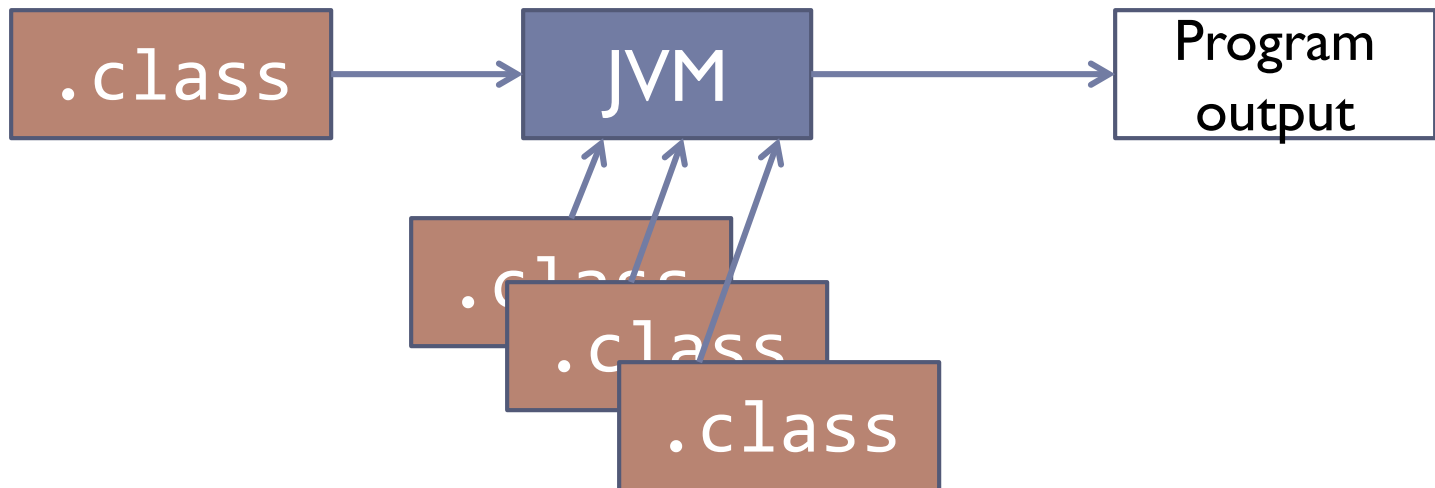
Outline of this chapter

- ▶ In this course we mostly consider compilation to bytecode, more specifically to JVM bytecode
 - ▶ We use the Soot framework, which provides several IRs
-
1. Overview of the JVM
 2. Overview of the Soot framework
 3. Code generation using Soot
 4. Code generation for physical architectures

1. Overview of the Java Virtual Machine (JVM)

The Big Picture

- ▶ The JVM interprets class files, each describing a single class
- ▶ Usually, class files are produced by a Java compiler, but there are other compilers that also target the JVM
- ▶ The JVM automatically loads any other classes referenced from the class files it is executing



Inside a Class File

A class file contains:

- ▶ Name of the class itself and its superclass and superinterfaces
- ▶ Descriptions of the class' members:
 - ▶ For fields: accessibility (**public**, **private**, **protected**), type, name
 - ▶ For methods: accessibility, return type, name, parameter types, bytecode for method body
- ▶ This information is encoded as binary data, so a class file is not human-readable
- ▶ Command line tool for displaying information in class file in readable form:

```
javap -v Test
```

Names in Class Files

- ▶ Names of classes, interfaces and fields are always given in fully-qualified form, including their package name
 - ▶ `java.lang.String` instead of just `String`
 - ▶ `java.lang.System.out` instead of `System.out`
- ▶ For methods, their name includes their full signature (i.e. types of method parameters, but not return type)
 - ▶ `java.lang.String.indexOf(int)`
 - ▶ `java.lang.String.indexOf(int, int)`
- ▶ Thus every class, interface, field and method in the program has a *unique* name
- ▶ Local variables and parameters do not have names in bytecode, they are represented by numerical indices (details later)

The JVM's loop

```
do {  
    fetch an opcode ;  
    if ( operands ) fetch operands ;  
    execute the action for the opcode ;  
  
} while ( there is more to do );
```

- ▶ A JVM instruction consists of:
 - ▶ An operation code – *opcode* – specifying the operation to be executed
 - ▶ Zero or more operands supplying arguments or data

Bytecode

- ▶ All code to be executed by the JVM has to be expressed as JVM bytecode
- ▶ JVM bytecode consists of a sequence of instructions, each performing one single operation, such as adding two numbers, or reading the value of a variable
- ▶ Unlike Java expressions, JVM instructions cannot be nested, so if one instruction needs to use a value computed by another instruction, it has to be explicitly stored into and read from a temporary location
- ▶ Native platforms such as x86 use *registers* for this purpose, but the JVM instead uses an *operand stack*: instructions take their operands from the stack, and push their result back onto it

Example of Stack-based Computing

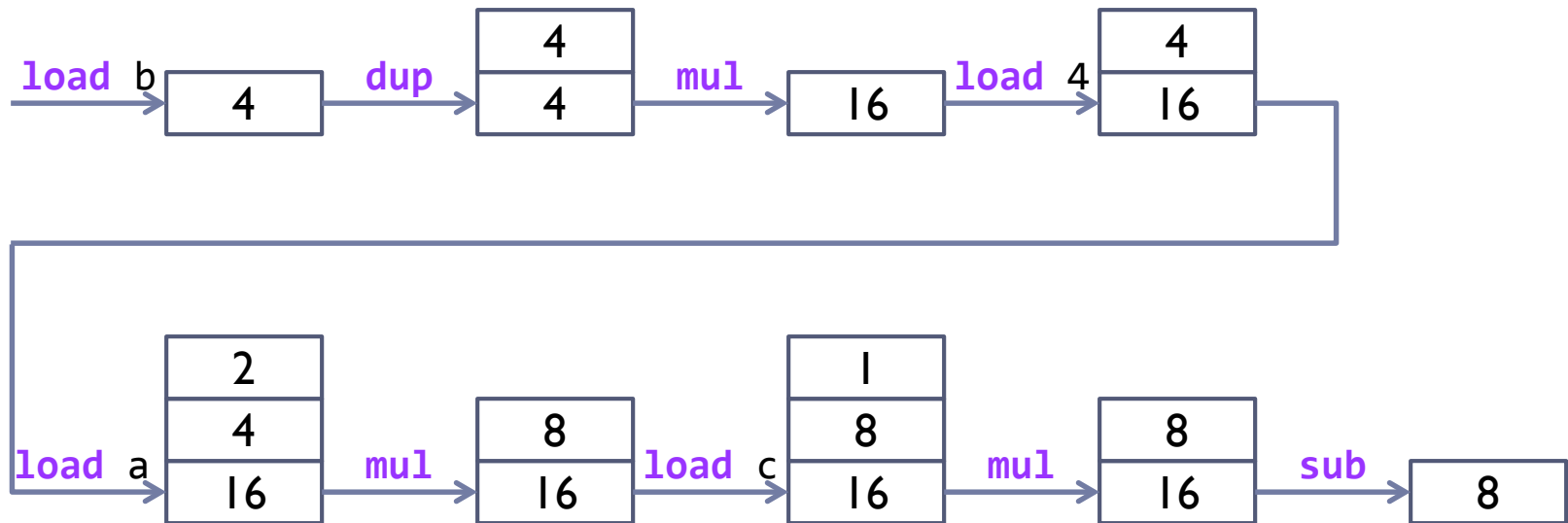
- ▶ An expression like $b*b - 4*a*c$ is computed as follows:

load b	load value of b onto stack
dup	duplicate value on top of stack
mul	pop two values, multiply, push result
load 4	load constant value 4 onto stack
load a	load value of a onto stack
mul	pop two values, multiply, push result
load c	load value of c onto stack
mul	pop two values, multiply, push result
sub	pop two values, subtract, push result

(**Note:** this is not actual JVM bytecode yet!)

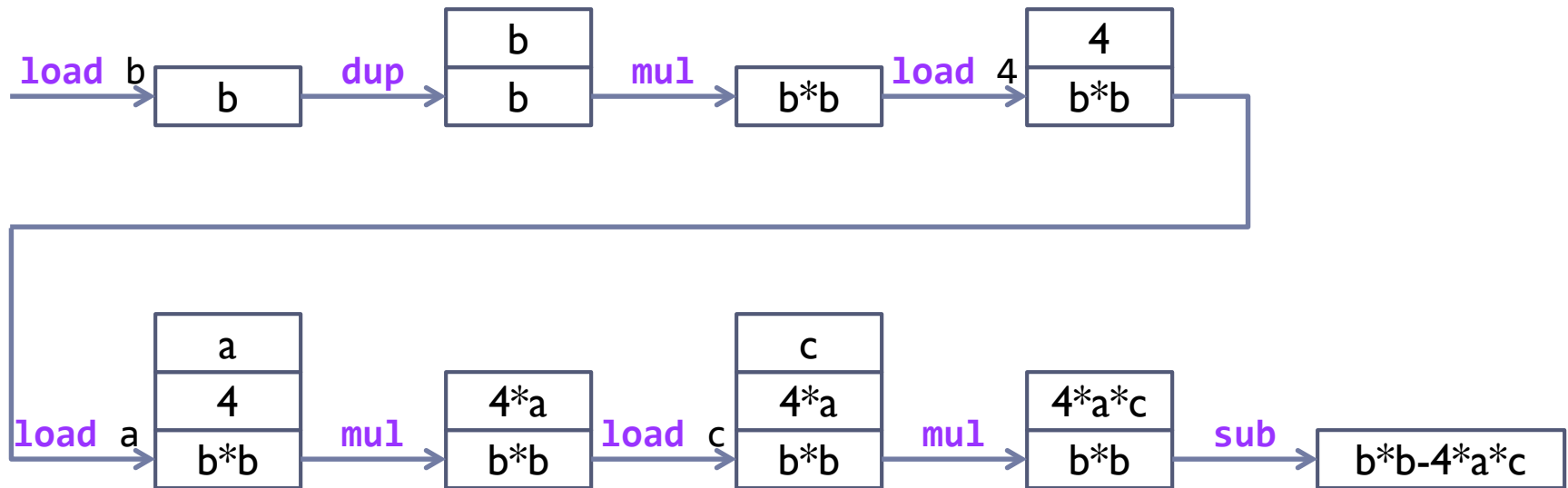
Example of Stack-based Computing (2)

- Assume a is 2, b is 4, and c is 1; then the computation works out like this:



Example of Stack-based Computing (3)

- And in the general case:



Runtime Storage Organisation

- ▶ At runtime, the JVM provides memory areas for:
 1. The *PC Register*: each JVM program has its own program counter, which contains the address of the instruction being executed
 2. The *Method Area*: shared among all JVM programs, used to store per-class structures such as constant pools, data and code for fields/methods, etc.
 3. The *Constant Pool*: used to store string constants and numerical constants appearing in the program
 4. The *Stack*: used to store the values of parameters and local variables, as well as intermediate results (operand stack)
 5. The *Heap*: used to store objects and arrays

Runtime Storage Organisation

- ▶ The JVM enforces a static typing discipline: every memory location has a type, which never changes; every value stored in the location has to be of the same type
- ▶ In particular, a reference to an object cannot be converted to an integer or vice versa (as in C): no pointer arithmetic!

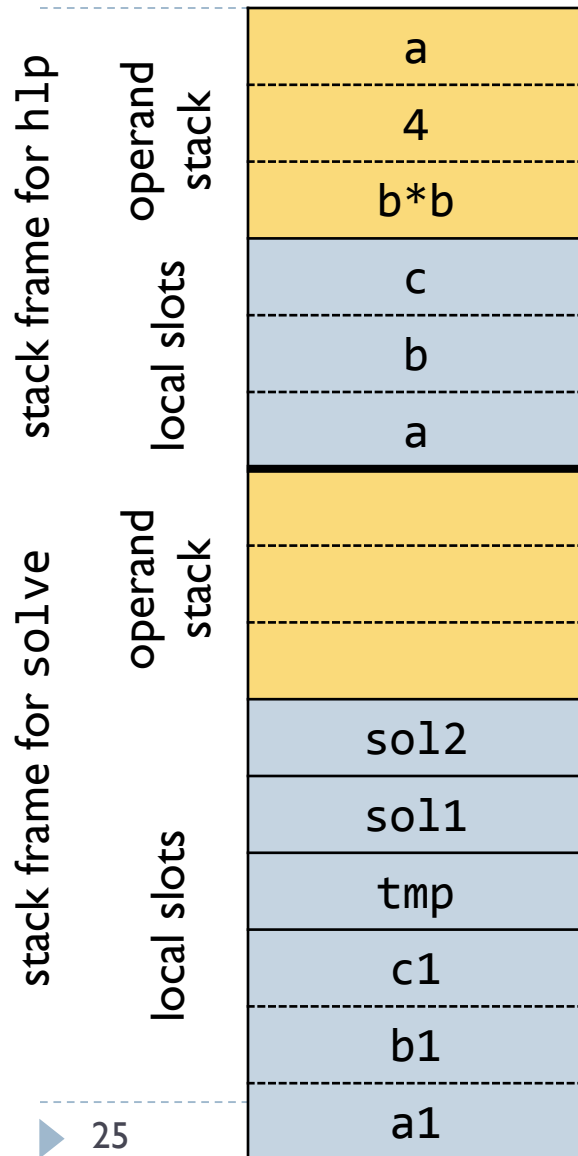
The Constant Pool

- ▶ The constant pool contains all numerical and string constants used in the program; in particular, it contains string constants for all class/method/field names referenced in the code
- ▶ Bytecode instructions that need to use constants contain indices into the constant pool where the actual value is found
- ▶ This saves space: different instructions that use the same constant can refer to the same constant in the constant pool
- ▶ For very commonly used small constants (-1, 0, 1, 2, 3, 4), the JVM offers specialised bytecode instructions, e.g. `iconst_0` pushes the constant 0 onto the stack; these constants do not need to be stored in the constant pool
- ▶ Each stack frame contains a reference to the constant pool for the class of the frame's method.

The Stack

- ▶ The stack is used to store local variables (including method parameters) and intermediate computation results
- ▶ It is subdivided into *frames*; whenever a method is invoked, a new frame is pushed onto the stack to store local variables and intermediate results for this method; when the method exits, its frame is popped off, exposing the frame of its caller beneath it
- ▶ Every frame in turn consists of two parts: a block of *local slots* for storing local variables, and an *operand stack* on top of it
- ▶ A bytecode method has to indicate how many local slots it needs, and how big its operand stack can get; thus, the size of a frame for a method is always fixed

Example Stack



Example methods:

```
double hlp(double a, double b, double c) {  
    return Math.sqrt(b*b-4*a*c);  
}
```

```
void solve(double a1, double b1, double c1)  
{  
    double tmp = hlp(a1, b1, c1);  
    double sol1 = (-b1 + tmp)/2*a1;  
    double sol2 = (-b1 - tmp)/2*a1;  
    System.out.println(sol1 + ", " + sol2);  
}
```

The stack on the left depicts the situation where solve has called hlp, and hlp is just computing its result as shown before

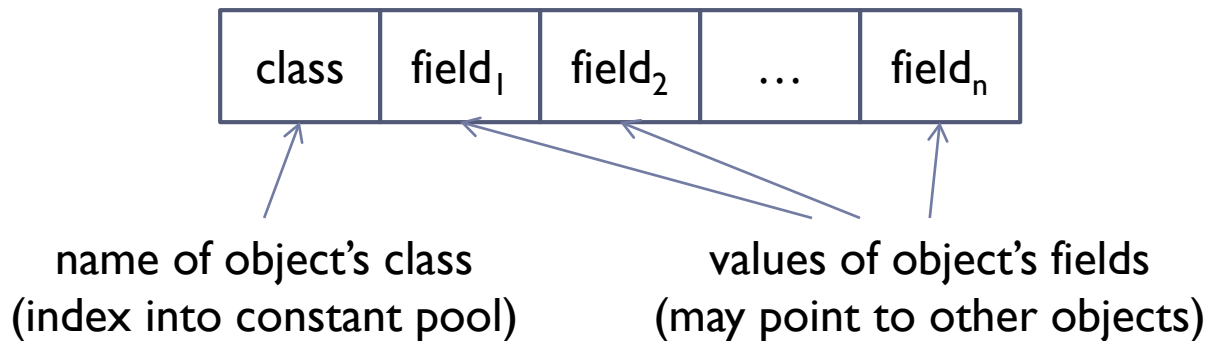
For instance based (non-static) methods, slot 0 holds **this**, i.e. object's self reference (not shown in diagram)

Stack Types

- ▶ Every element on the stack (both locals and intermediate results) has to be of one of the following types:
 1. **int**: 32-bit integer
 2. **long** : 64-bit integer
 3. **float** : 32-bit floating point
 4. **double** : 64-bit floating point
 5. **address** : pointer to object or array on heap
- ▶ **byte**, **char**, **short** are stored as **int**; values of type **boolean** are stored as integers 0 and 1

The Heap

- ▶ The heap is used to store arrays and objects
- ▶ The JVM specification does not mandate a particular layout for the heap, but usually objects are stored like this:



- ▶ Objects are represented on the stack as references into the heap, which is shared among all JVM threads
- ▶ Objects on the heap are only removed by the garbage collector

Bytecode Instruction Set

- ▶ The JVM defines 256 instructions, encoded as numbers from 0 to 255 (fits into a single byte, hence the name bytecode)
- ▶ All the JVM instructions also have mnemonic names
- ▶ Most instructions take operands from the stack and leave their result on top of it
- ▶ Additionally, some instructions take extra operands that are encoded together with the instruction in the bytecode; such instructions are then longer than one byte
- ▶ Almost all JVM instructions only operate on a single type of data; there are usually several variant instructions performing the same operation on different data types

Instruction Categories

► There are eight categories of instructions:

1. Load and Store Instructions
2. Arithmetic Instructions
3. Type Conversion Instructions
4. Object Creation and Manipulation Instructions
5. Operand Stack Management Instructions
6. Control Transfer Instructions
7. Method Invocation and Return Instructions
8. Other Instructions

► We will discuss typical examples from each category

Load and Store Instructions

- ▶ Instructions for loading constants onto the stack:

<code>iconst_0, ..., iconst_5</code>	push int constant 0, ..., 5
<code>iconst_m1</code>	push int constant -1
<code>lconst_i, fconst_i, dconst_i</code>	same for long, float, double
<code>aconst_null</code>	push constant null
<code>ldc</code>	push constant from constant pool

- ▶ Instructions for loading local variables onto the stack:

<code>iload_0, ..., iload_3, iload i</code>	push local int variable 0, ..., 3, <i>i</i> onto stack
<code>lload, fload, dload, aload</code>	same for long, float, double, address

- ▶ Instructions for storing the value on top of the stack into a local variable are similar: `istore, lstore, ...`

Arithmetic Instructions

- ▶ These instructions correspond to Java's arithmetic and logical operators, but each instruction only works on operands of one type
- ▶ Operands are popped off the stack, result pushed back onto the stack

iadd, ladd, fadd, dadd	addition (int , long , float , double)
isub, lsub, fsub, dsub	subtraction (int , long , float , double)
tdiv, trem, tneg	division, modulo, negation (4 variants each)
ishl, ishr, iushr	<<, >>, >>> on int (lshl etc. for long)
ior, iand, ixor	, &, ^ on int (lor etc. for long)

Arithmetic Instructions (2)

- ▶ There are also instructions `lcmp`, `fcmp`, and `dcmp` for comparing values of type **long**, **float**, and **double**
- ▶ They each pop operands y and x off the stack, then compare them; if $x > y$, push 1; if $x < y$, push -1; otherwise push 0
- ▶ There is *no* such instruction for **int**: comparisons between integers always take the form of conditional jumps (explained below)
- ▶ Conversely, conditional jumps cannot directly compare **long**, **float** or **double** values, but have to use `lcmp`, `fcmp` or `dcmp` and then compare their (integer) result against 0

Type Conversion Instructions

- ▶ Arithmetic instructions require both of their operands to have the same type
- ▶ If they are of different types, one of them has to be converted using a type conversion instruction first
- ▶ Widening conversions: `i2l` (“**int** to **long**”), `i2f`, `i2d`, `l2f`, `l2d`, `f2d`
- ▶ Narrowing conversions: `l2i`, `f2i`, `f2l`, `d2i`, `d2l`, `d2f`
- ▶ Integers can also be converted to **byte**, **short**, or **char**: `i2b`, `i2s`, `i2c`; these instructions do not change the type of the top stack element (remember that stack elements cannot be of types smaller than **int**!), but truncate its value
- ▶ Note that there are no conversions from/to **address**

Object Creation and Manipulation

- ▶ Create new class instance: `new "java.lang.String"`
 - ▶ Class name stored in constant pool
 - ▶ This only allocates memory; it does not invoke the constructor!
- ▶ Similar instruction `newarray` for allocating arrays
- ▶ Read instance field value: `getfield "A.f"`
 - ▶ The topmost stack value specifies the object whose field `f` is read
- ▶ Instruction `putfield` writes to field; instructions `getstatic` and `putstatic` access static fields
- ▶ Special instruction `arraylength` to read length of an array
- ▶ Instructions `iaload`, `iastore` access elements of **int** array; similar for other types
- ▶ Instructions **instanceof**, **checkcast** do dynamic type checks

Operand Stack Management Instructions

- ▶ dup: duplicate top stack element
- ▶ pop: pop off top stack element
- ▶ Many other, less commonly used instructions
- ▶ These are the only type-generic instructions: one instruction to handle all possible types

Control Transfer Instructions

- ▶ Unconditional jump to instruction at address t : goto t
- ▶ Conditional jumps pop off y and x ; if some condition is true, they jump to t , otherwise continue with next instruction:

if_icmpeq t	x and y must be ints ; jump to t if $x == y$; if_icmpne/lt/le/ge/gt jump if $!=, <, <=, >=, >$
if_acmpeq t	x and y must be addresses ; jump to t if $x == y$; if_acmpne jumps if $x != y$

- ▶ Instructions ifeq, ifne, iflt, ifle, ifgt, ifge are similar: they only pop off a single (**int**) operand x and compare it to 0
- ▶ ifnull, ifnonnull pop off an **address** operand and compare it to null
- ▶ We do not discuss tableswitch and lookupswitch

Method Invocation and Return Instructions

- ▶ Arguments are pushed onto stack (receiver object first), then the following instructions are used to invoke the method:
 - ▶ `invokevirtual`: for normal, non-static methods
 - ▶ `invokeinterface`: for methods declared in interfaces
 - ▶ `invokespecial`: for **super** calls
 - ▶ `invokestatic`: for static methods (no receiver object)
- ▶ The operand of the `invoke` instruction is the signature of the method to invoke; in particular, it indicates how many arguments the method expects
- ▶ When a method exits, it uses `ireturn` (`lreturn`, `freturn`, `dreturn`, `areturn`) to return the value on top of the stack, or `return` to exit without returning a value
- ▶ The return value is left on top of the caller's operand stack

Other Instructions

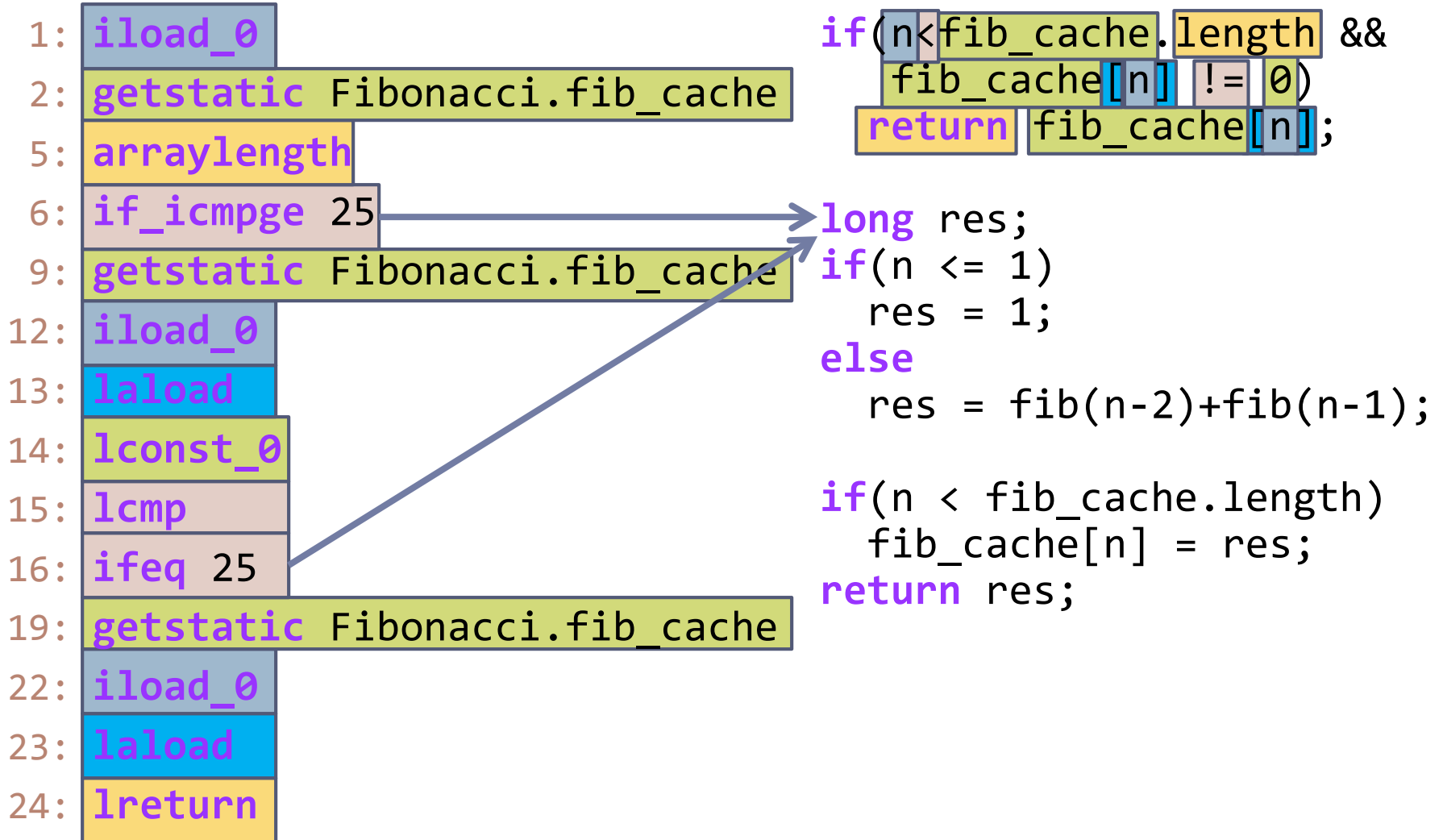
- ▶ Exception throwing: `athrow`
- ▶ Synchronisation: `monitorenter`, `monitorexit`
- ▶ `nop`: do nothing (surprisingly useful during compilation!)

Example: Fibonacci.java

- ▶ Consider the following Java program for computing Fibonacci numbers:

```
public class Fibonacci {  
    private static long fib_cache[] = new long[256];  
  
    public static long fib(int n) {  
        if(n < fib_cache.length && fib_cache[n] != 0)  
            return fib_cache[n];  
  
        long res;  
        if(n <= 1)  
            res = 1;  
        else  
            res = fib(n-2) + fib(n-1);  
  
        if(n < fib_cache.length)  
            fib_cache[n] = res;  
        return res;  
    }  
}
```

Bytecode for fib method

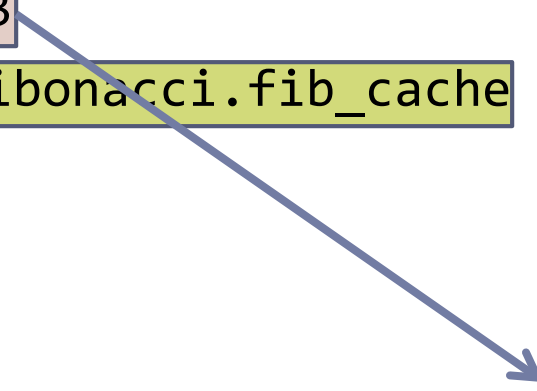


Bytecode for fib method (2)

25:	<code>iload_0</code>	<code>if(n < fib_cache.length &&</code>
26:	<code>iconst_1</code>	<code>fib_cache[n] != 0)</code>
27:	<code>if_icmpgt 35</code>	<code>return fib_cache[n];</code>
30:	<code>lconst_1</code>	
31:	<code>lstore_1</code>	<code>long res;</code>
32:	<code>goto 49</code>	<code>if(n <= 1)</code>
35:	<code>iload_0</code>	<code>res = 1;</code>
36:	<code>iconst_2</code>	<code>else</code>
37:	<code>isub</code>	<code>res = fib(n-2) + fib(n-1);</code>
38:	<code>invokestatic Fibonacci.fib(int)</code>	<code>if(n < fib_cache.length)</code>
41:	<code>iload_0</code>	<code>fib_cache[n] = res;</code>
42:	<code>iconst_1</code>	<code>return res;</code>
43:	<code>isub</code>	
44:	<code>invokestatic Fibonacci.fib(int)</code>	

Bytecode for fib method (3)

47: ladd	if (n<fib_cache.length &&
48: lstore_1	fib_cache[n] != 0)
49: iload_0	return fib_cache[n];
50: getstatic Fibonacci.fib_cache	long res;
53: arraylength	if (n <= 1)
54: if_icmpge 63	res = 1;
57: getstatic Fibonacci.fib_cache	else
60: iload_0	res = fib(n-2)+fib(n-1);
61: lload_1	if (n < fib_cache.length)
62: lstore	fib_cache[n] = res;
63: lload_1	return res;
64: lreturn	



2. Overview of Soot

Soot

- ▶ Soot <http://www.sable.mcgill.ca/soot> is an optimisation framework for JVM bytecode
- ▶ It defines several higher-level intermediate representations for bytecode that are easier to work with, transform and optimise than JVM bytecode
- ▶ In this chapter, we will use Soot as a library for generating bytecode; the next chapter will take a brief look at using Soot for optimisation
- ▶ It is also possible to generate bytecode directly from an AST, but using Soot's intermediate representations instead is more convenient

Soot's Intermediate Representations

- ▶ Soot defines four intermediate representations for bytecode:
 1. **Baf**: similar to bytecode; same instructions, but abstracts away from constant pool (constants are embedded in the code) and type prefixes (e.g. there is only one add instruction)
 2. **Jimple**: most important IR; instructions similar to bytecode, but uses explicitly named temporary variables instead of operand stack; only 15 instructions (vs 256 bytecode instructions)
 3. **Shimple**: variant of Jimple where each local variable is assigned at most once (this is known as Static Single Assignment or *SSA form*); better for some optimisations
 4. **Grimp**: high-level representation with nested expressions (so no need for operand stack); control flow still encoded as jumps
- ▶ Soot provides translators between these four IRs and JVM bytecode; can also read in class files directly

Used in
this course



- ▶ Soot provides a Java API for reading in classes from external files, or constructing them from scratch using any of the IRs
full API listing available online: www.sable.mcgill.ca/soot/doc
- ▶ The most fundamental data structure is class `soot.Scene`; it has a single instance, available through method `Scene.v()`
- ▶ The scene keeps track of all the classes that have been loaded from bytecode or created by Soot (automatically)
- ▶ It also stores analysis information about the entire program (as opposed to individual classes or methods)
- ▶ Method `Scene.loadClassAndSupport` is used to load a Java class from the standard library
- ▶ Usually, we want to load `java.lang.Object` first thing:
`Scene.v().loadClassAndSupport("java.lang.Object");`

Representing classes

- ▶ Each class or interface is represented as a SootClass object
- ▶ Can be constructed from scratch:

```
SootClass klass=new SootClass("tst.Test",Modifier.PUBLIC)
```

This creates a new public class with qualified name tst.Test

- ▶ A previously loaded/constructed class can be obtained from the scene by its fully qualified name:

```
SootClass object =  
    Scene.v().getSootClass("java.lang.Object")
```

- ▶ After creating a new SootClass, we need to set its super class:

```
klass.setSuperclass(object)
```

- ▶ We can also use SootClass.addInterface to add interfaces that the class implements
- ▶ Soot (like JVM bytecode) considers interfaces as classes with a special modifier **interface**

Modifiers

- ▶ Modifiers for classes/interfaces/fields/methods are represented as integers which are interpreted as the bitwise or of flags defined in class `java.lang.reflect.Modifier`
- ▶ List of flags (not all are applicable everywhere):

- `Modifier.ABSTRACT`
- `Modifier.FINAL`
- `Modifier.NATIVE`
- `Modifier.PRIVATE`
- `Modifier.PROTECTED`
- `Modifier.PUBLIC`
- `Modifier.STATIC`
- `Modifier.SYNCHRONIZED`
- `Modifier.TRANSIENT`
- `Modifier.VOLATILE`

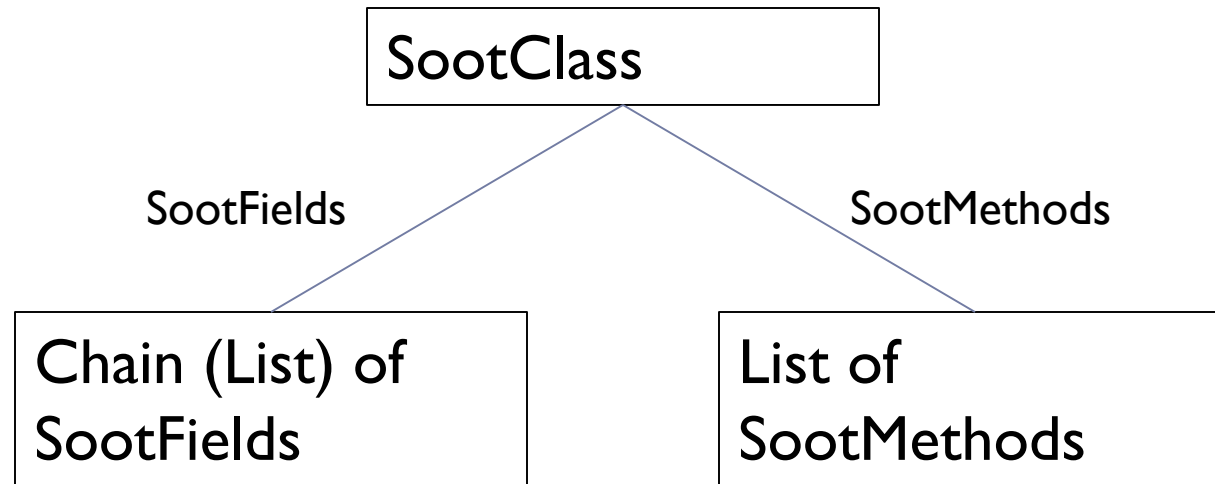
Some modifiers are not available at the Java source level:

- `Modifier.ANNOTATION`
- `Modifier.BRIDGE`
- `Modifier.ENUM`
- `Modifier.INTERFACE`
- `Modifier.STRICT`
- `Modifier.SYNTHETIC`
- `Modifier.VARARGS`

Representing types, fields and methods

- ▶ Types are represented by objects of type `soot.Type`:
 - ▶ Singleton classes (class with single instance) for primitive types, e.g. `IntType` represents `int`, an instance can be generated using `IntType.v()`
 - ▶ Class `RefType` represents class/interface types, can be obtained through `SootClass.getType()`
 - ▶ Class `ArrayType` represents array types
- ▶ Fields are represented by class `soot.SootField`:
`SootField(String name, Type type, int modifiers)`
- ▶ Methods are represented by class `soot.SootMethod`:
`SootMethod(String name, List<Type> paramTypes, Type returnType, int modifiers, List<SootClass> thrownExceptions)`

SootClass



- Fields and Methods are added to a class with `SootClass.addField` and `SootClass.addMethod` respectively

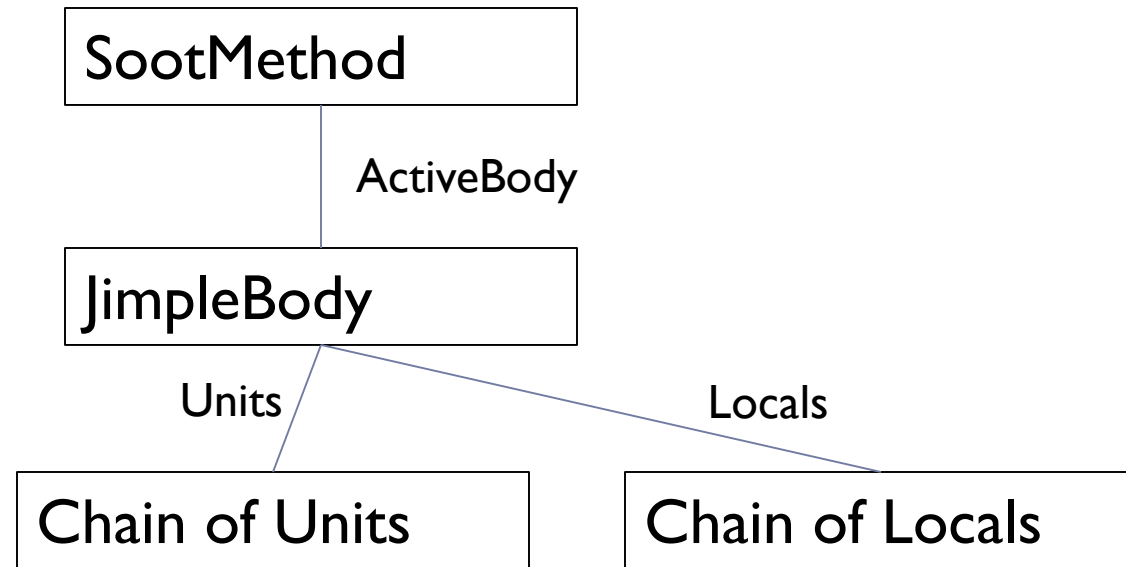
Method Bodies

- ▶ Every SootMethod has a body; there are different types of bodies, one for each intermediate representation
- ▶ We will be working with class JimpleBody, which represents method bodies implemented in Jimple
- ▶ Each body keeps track of three things:
 1. A list of local variables (class soot.Local)
 2. A list of statements, called “units” in Soot (class soot.Unit)
 3. A list of exception handlers, called “traps” in Soot (class soot.Trap)
- ▶ These are represented using Soot’s own implementation of Lists (class soot.Chain), which mostly has the same methods as java.util.List

Local variables

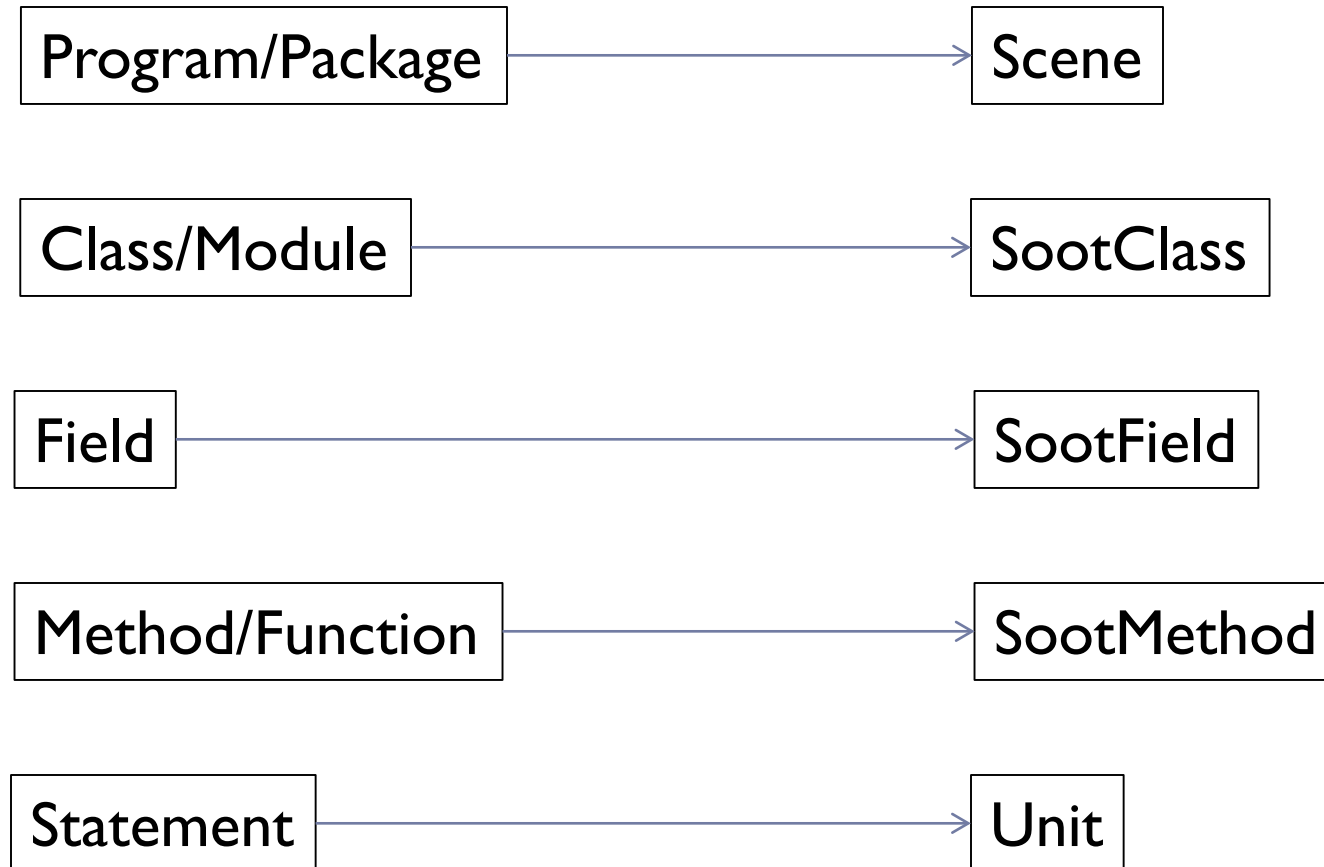
- ▶ Local variables are represented by class `soot.Local`
- ▶ New locals are created by factory method
`Jimple.newLocal(String name, Type type)`
- ▶ Add them to method body like this:
`body.getLocals().add(var)`
- ▶ Parameters are represented in the same way, but they need to be initialised in a special way
- ▶ Local variables in Jimple can be of any type including **byte**, **short**, and **char**; Soot will map them to the appropriate stack types when translating to bytecode

SootMethod



- ▶ The Body is set using `SootMethod.setActiveBody`
- ▶ The Units and Locals are added to the respective Chains
 - ▶ e.g. `body.getUnits().add(stmt)`
 - ▶ e.g. `body.getLocals().add(var)`

Soot's High-level Data Abstractions



Jimple Overview

- ▶ Jimple is a typical example of what is known as a *3-address code* intermediate representation
- ▶ Most instructions work on three operands, which all denote local variables (“addresses”): two operands are inputs, one operand is the output where the result is stored

- ▶ Typical example:

$$x = y + z$$

y and z are inputs, x is output

- ▶ Crucially, y and z cannot themselves be complex expressions; temporary variables have to be used to compute nested expressions
- ▶ Of course, some instructions have less than three operands
- ▶ In bytecode, most operands are implicit; in Jimple, all operands are explicit

Jimple Statements

► There are 15 statement types in Jimple:

1. **Assignments**
2. **Identity statements**
3. **If statements**
4. **Goto statements**
5. **Invocation statements**
6. **Return statement**
7. **Return void statement**
8. **Nop statement**
9. Tableswitch statement
10. Lookupswitch statement
11. Throw statement
12. Monitor enter statement
13. Monitor exit statement
14. Breakpoint statement
15. Ret statement

We will not use
types 9-15

Soot can output a
textual (readable)
form of Jimple

Jimple Operands

- ▶ Operands can be:
 - ▶ Local variables (interface `soot.Local`)
 - ▶ Field/array element reference (interfaces `soot.jimple.FieldRef` and `soot.jimple.ArrayRef`)
 - ▶ Constants (class `soot.jimple.Constant` and subtypes)
 - ▶ Expressions (interface `soot.jimple.Expr` and subtypes)
- ▶ There are many different kinds of expressions
- ▶ Expressions cannot be nested: the operands of an expression can be locals or constants, but cannot be other expressions
- ▶ All statements and expressions must be constructed using factory methods in class `Jimple`

Jimple Expressions

- ▶ The usual arithmetic and logical expressions are represented by subclasses of `BinopExpr` and `UnopExpr`; as with bytecode, there is a special `LengthExpr` for determining array length
- ▶ `CastExpr` and `InstanceOfExpr` represent cast expressions and **instanceof** checks
- ▶ For each expression type, there is a factory method in class `Jimple` (use method `Jimple.v()` to obtain the singleton instance of class `Jimple`)
 - ▶ e.g. `Jimple.newAddExpr` to create an add expression
- ▶ Field references are constructed using method in class `Scene`:
`Scene.makeFieldRef(SootClass declaringClass,
String name, Type type,
boolean isStatic)`

Assignments and Identity Statements

- ▶ An assignment (`soot.jimple.AssignStmt`) has a left operand and a right operand (accessor methods `getLeftOp` and `getRightOp`)
- ▶ Assignments are constructed by factory method
`Jimple.newAssignStmt(Value left, Value right)`
Here, `left` should be a local or a reference to a field
- ▶ Identity statements (`soot.jimple.IdentityStmt`) are special assignments that copy a parameter into a local; there should be one identity statement for every parameter
`Jimple.newIdentityStmt(Local l, ParameterRef p)`
- ▶ `ParameterRef` objects are constructed by
`Jimple.newParameterRef(Type type, int index)`
Here, `index` is the number of the parameter, `type` is its type

If statements and goto statements

- ▶ An if statement (`soot.jimple.IfStmt`) contains a condition and a target (accessor methods `getCondition` and `getTarget`)
- ▶ If statements are constructed by factory method

```
Jimple.newIfStmt(CmpExpr cond, Unit target)
```
- ▶ The target is the statement where execution continues if the condition evaluates to true (otherwise it continues at next statement)
- ▶ A goto statement (`soot.jimple.GotoStmt`) is similar to an if statement, except that it has no condition: execution always continues at the target

Method Calls

- ▶ Methods are identified by method references (`soot.SootMethodRef`), which can be constructed using method `Scene.makeMethodRef`
- ▶ Instance method calls are represented by class `soot.jimple.InstanceInvokeExpr`; they consist of a method reference, a base value, and a list of arguments
- ▶ Static method calls (`StaticInvokeExpr`) are similar, but have no base value
- ▶ Method calls to non-void methods usually appear on the right hand side of assignment statements
- ▶ Calls to void methods must be wrapped into an invocation statement (`soot.jimple.InvokeStmt`)

Return statements

- ▶ There are two types of return statements:
`soot.jimple.ReturnStmt` and
`soot.jimple.ReturnVoidStmt`
- ▶ The former statement returns a value; its factory method is
`Jimple.newReturnStmt(Value value)`
- ▶ The latter statement does not return a value; its factory method is
`Jimple.newReturnVoidStmt()`
- ▶ In Jimple (as in JVM bytecode), every control flow path must end either with a throw statement or with a return statement; “falling off” the end is not allowed, even in a **void** method

Other statements

- ▶ The nop statement is the same as the JVM nop statement: it does not do anything
- ▶ Its factory method is
`Jimple.newNopStmt()`
- ▶ The other statement types will not be discussed, they are not important for the practical assignment

Reference material on Soot

- ▶ List of tutorials: <http://www.sable.mcgill.ca/soot/tutorial/>
- ▶ In particular:
 - ▶ Soot survivor's guide: <http://www.brics.dk/SootGuide/>
 - ▶ “Creating a Class File from Scratch”
 - ▶ “On the Soot menagerie -- Fundamental Soot Objects”
 - ▶ Blog posts on Soot: <http://www.bodden.de/tag/soot-tutorial/>
- ▶ Note, though, that many of these materials are more about implementing compiler optimisations using Soot

Factory Design Pattern

- ▶ Soot and Jimple make heavy use of the **factory design pattern**
- ▶ In Soot, the standard name of `v()` is used for the (value) factory method, where the value of the instance is important (`v` stands for value)
 - ▶ An example of the use of a value factory method is to implement a constant, e.g. `IntConstant.v(23)` generates the instance representing the integer constant 23
- ▶ The **singleton pattern** is a special case of the value factory pattern that takes no arguments – the same object is guaranteed to be returned each time
 - ▶ An example is a primitive type, e.g. `IntType.v()` generates the object representing an integer type