Compiler Techniques

Lecture 8: Code Generation - JVM

Tianwei Zhang

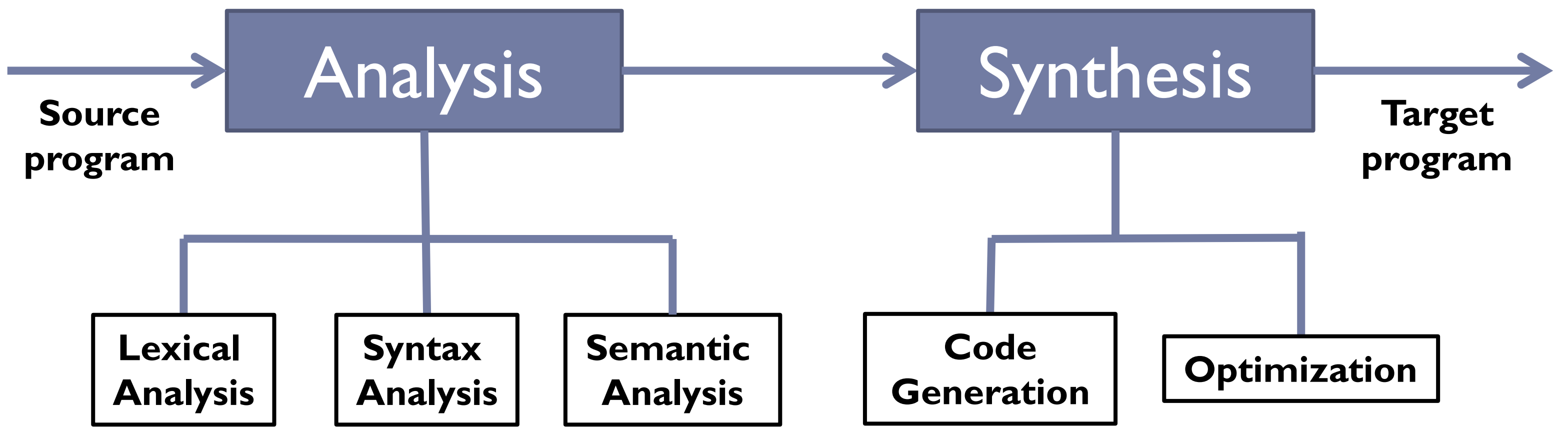
Outline

- **Overview of Backend Synthesis**
- **Java Virtual Machine**
  - **Overview and memory layout**
  - **Bytecode instruction set**

Outline

- **Overview of Backend Synthesis**
- Java Virtual Machine
  - Overview and memory layout
  - Bytecode instruction set

Architectural Overview of a Compiler



- Front-end analysis (completed)
  - Lexical Analysis
  - Syntax Analysis
  - Semantic Analysis
- Back-end synthesis (upcoming)
  - Code generation (lectures 8 - 9)
  - Optimization (lectures 10 - 13)

Main Function of Code Generator

- Code generator: produces executable code from an abstract syntax tree representation of a program

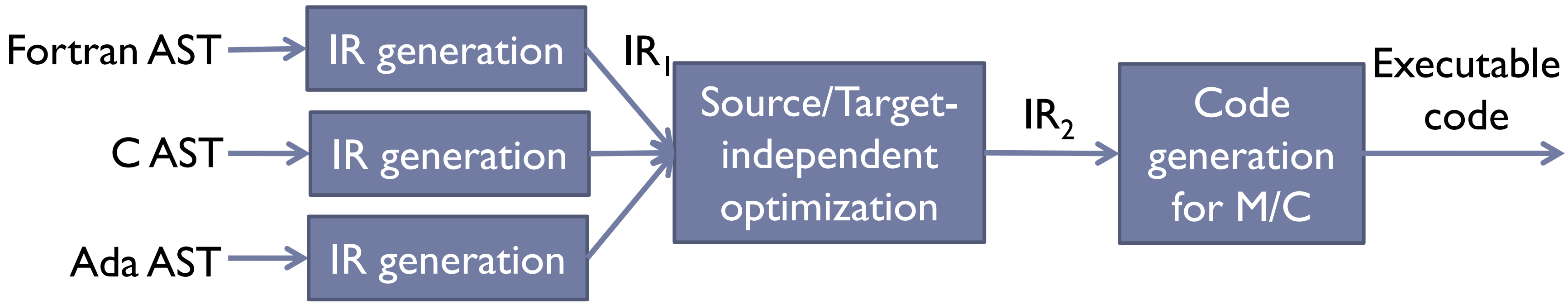
- The executable code should be:
  - equivalent to the source program
  - able to run on a machine
  - smaller or faster (*optimisations* may be performed)

Multi-pass Process

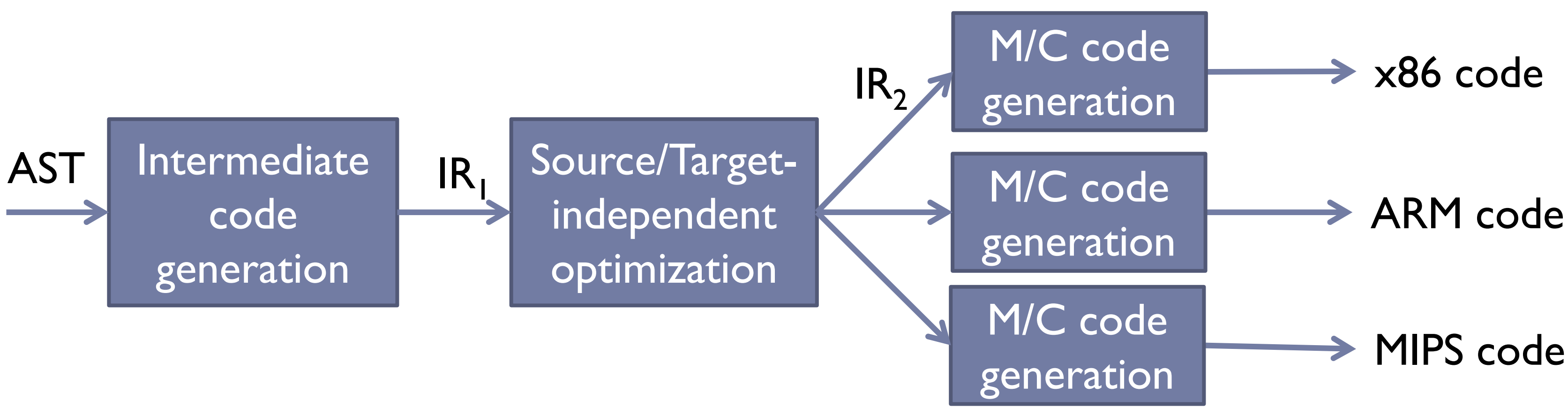
- Code generation is split up into several different passes, and each one produces a certain intermediate representation (IR)
  - Convert a complicated task into several easy tasks
  - Each pass performs a different optimisation or transformation

Multiple Languages and Platforms

- Some compilers (e.g., gcc) support multiple input languages.

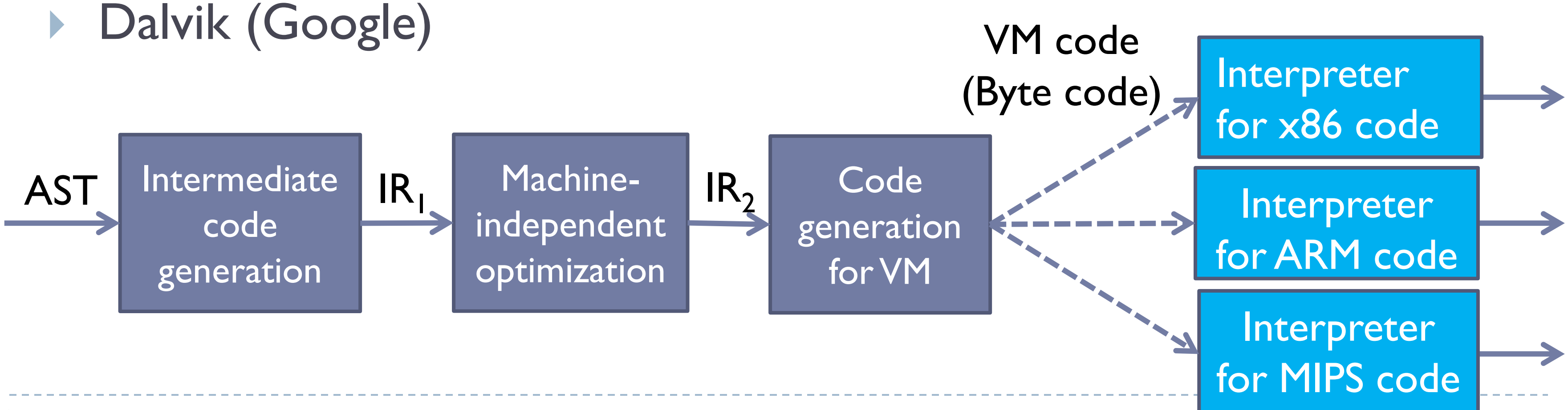


- Some compilers (e.g., gcc, llvm) support multiple platforms.



Virtual Machine Code (Bytecode)

- Compiler translates source code to virtual machine code
  - Independent of platforms
- Interpreter (virtual machine) runs Bytecodes on a physical machine
  - Platform-specific
- Examples:
  - Java Virtual Machine (Sun/Oracle)
  - Common Language Runtime (Microsoft)
  - Dalvik (Google)



Bytecode vs. Executable Code

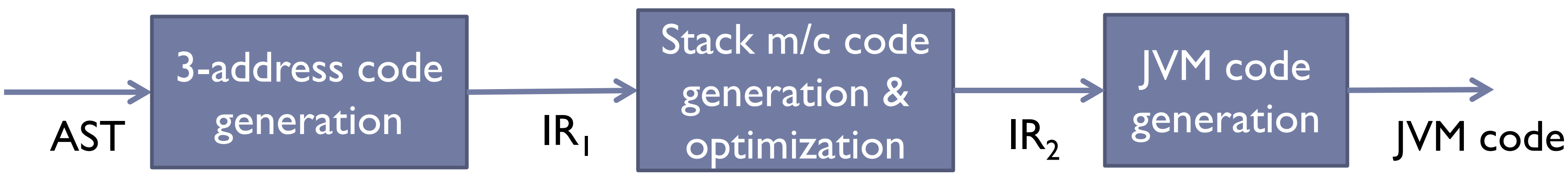
- Advantages of compiling to bytecode:
  - Convenient: only need to generate code for one platform
  - Easy: easier to perform code generation with bytecode.
    - Virtual machine instruction sets: high-level with more functionalities, e.g., automated garbage collection
    - Native instruction sets: emphasize efficient execution over ease of compilation
  - Compact: often more compact and suitable for resource-constrained platforms
- Advantages of compiling to native code:
  - Efficient: Running native code is in general faster than running virtual machine code

Just-in-Time (JIT) Compilation

- Interpreters compile bytecode to native code on-the-fly
  - Trade-off: native code is faster, but compilation takes time
- Optimizations
  - Compile heavily-used ("hot") parts of the program (e.g., methods being executed several times)
  - Interpret the rest parts.
  - Exploit runtime profiling to perform more targeted optimizations than compilers targeting native code directly
- JIT-based virtual machines are competitive with native code for most application areas except heavy numeric computations

Outline of the Following Lectures

- Scope:
  - Lecture 8: Compilation to bytecode, more specifically to JVM bytecode
  - Lecture 9: Soot framework to provide several IRs and optimizations



Outline

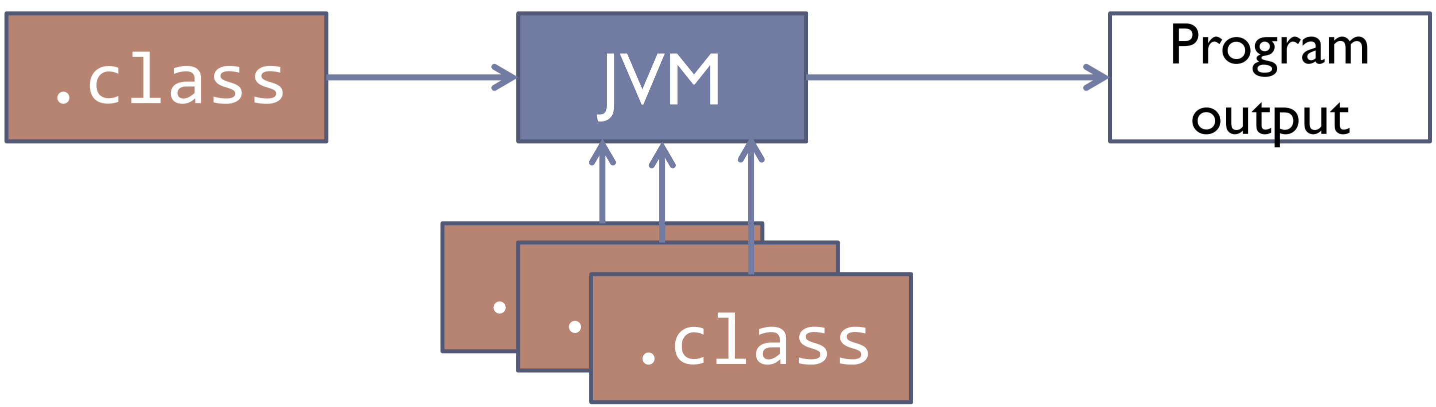
- Overview of Backend Synthesis
- **Java Virtual Machine**
  - Overview and memory layout
  - Bytecode instruction set

Outline

- Overview of Backend Synthesis
- **Java Virtual Machine**
  - **Overview and memory layout**
  - Bytecode instruction set

The Big Picture

- A Java compiler generates different class files.
- The JVM interprets class files
- The JVM automatically loads any other classes referenced from the class files it is executing
- Can be applied to other languages
  - Scala, Kotlin, …



Inside a Class File

A class file contains:

- Name of the class itself and its superclass and superinterfaces
- Descriptions of the class' members:
  - Fields: accessibility (**public**, **private**, **protected**), type, name
  - Methods: accessibility, return type, name, parameter types, bytecode for method body
- Binary format, not human-readable
  - Command line tool for displaying information in class file in readable form:

```
javap -v Test
```

Names in Class Files

- Classes, interfaces and fields: fully-qualified form with package name
  - `java.lang.String` instead of `String`
  - `java.lang.System.out` instead of `System.out`
- Methods: full signature (*i.e.*, types of method parameters, but not return type)
  - `java.lang.String.indexOf(int, int)`
- Local variables and parameters: represented by numerical indices

The JVM's loop

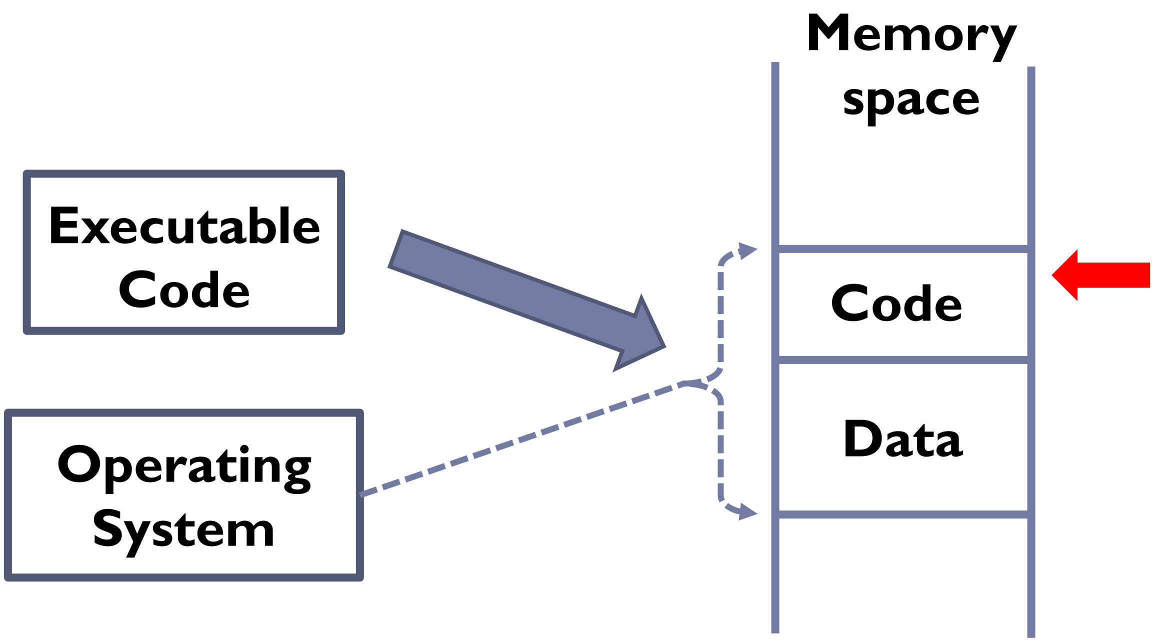
- A JVM instruction consists of:
  - An operation code (opcode): *the* operation to be executed
  - Zero or more operands supplying arguments or data

```
do {
        fetch an opcode ;
        if ( operands )
                fetch operands ;
        execute the action for the opcode ;

} while ( there is more to do );
```

Overview of Program Execution

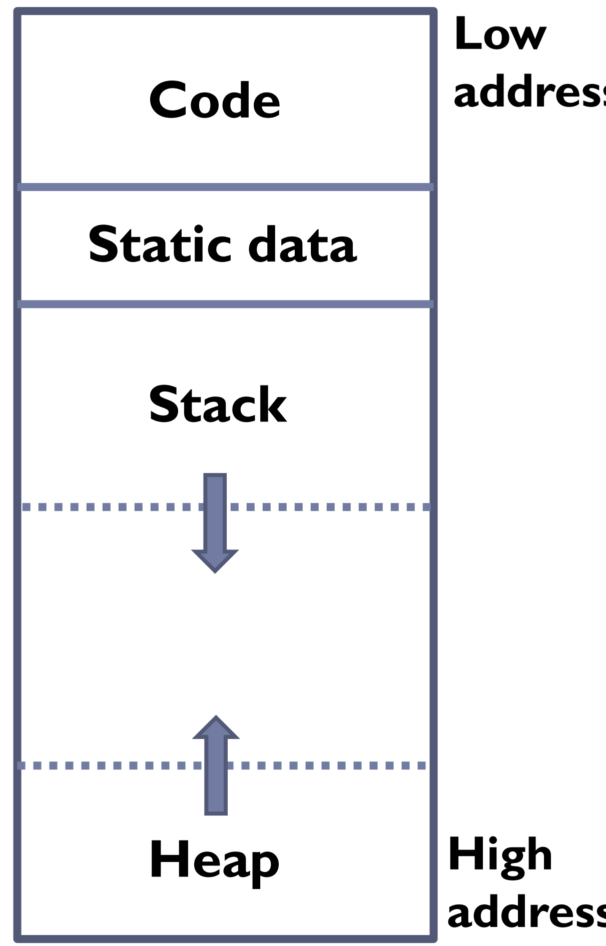
- When a program is invoked:
  - OS allocates memory space (may not be contiguous)
  - OS loads the code into part of the space
  - The execution jumps to the entry point, *i.e.*, first instruction, of main()



Memory Layout of a Compiled Program

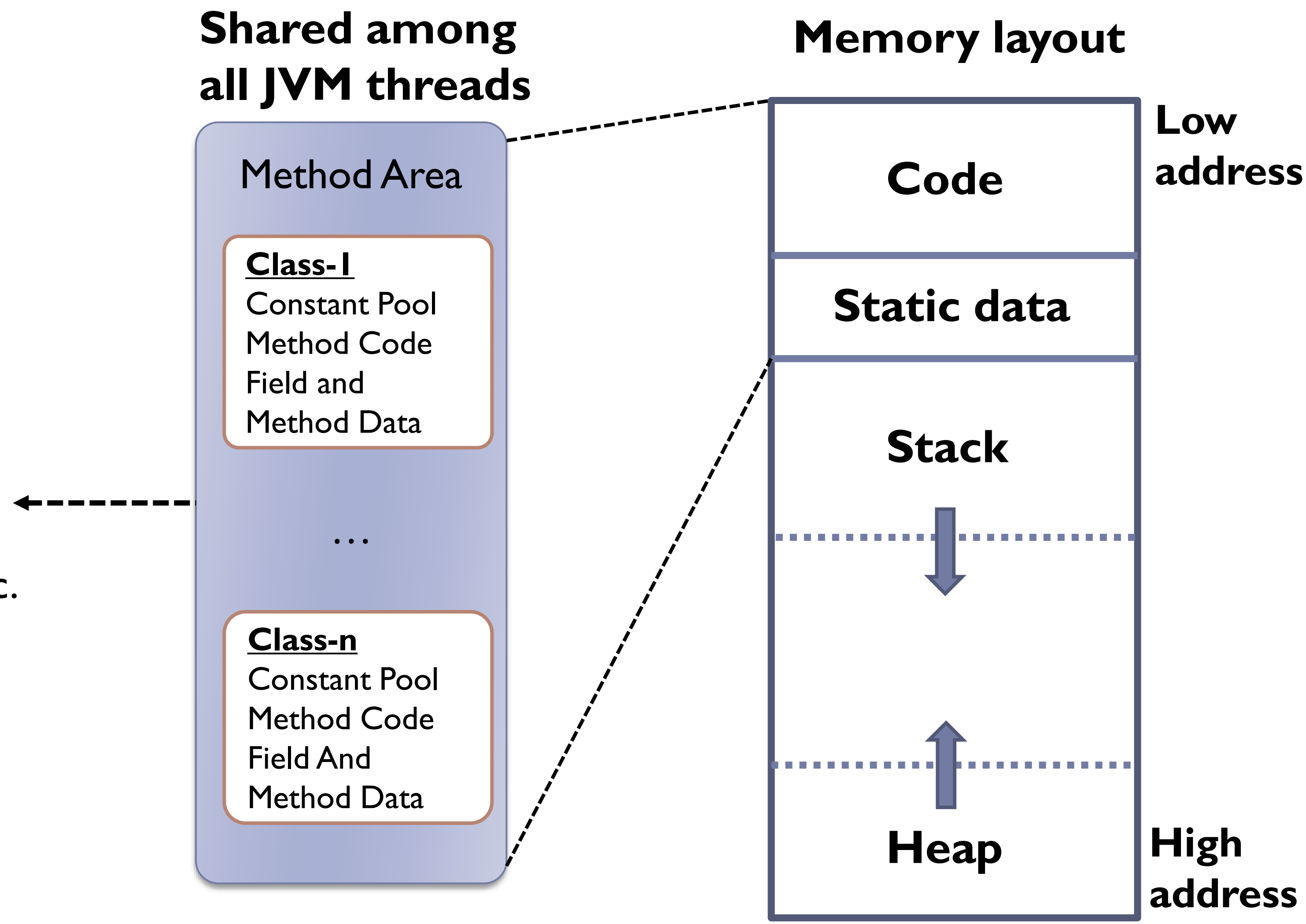
- Memory layout (for many languages)
  - Code area: fixed size and read only
  - Static data: statically allocated data
    - variables/constants
  - Stack: parameters and local variables of methods as they are invoked.
    - Each invocation of a method creates one **frame** which is pushed onto the stack
  - Heap: dynamically allocated data
    - class instances/data array
  - Stack and heap grow towards each other

Memory layout



Memory Layout of JVM

store per-class structures, e.g., constant pools, data and code for fields/methods, etc.

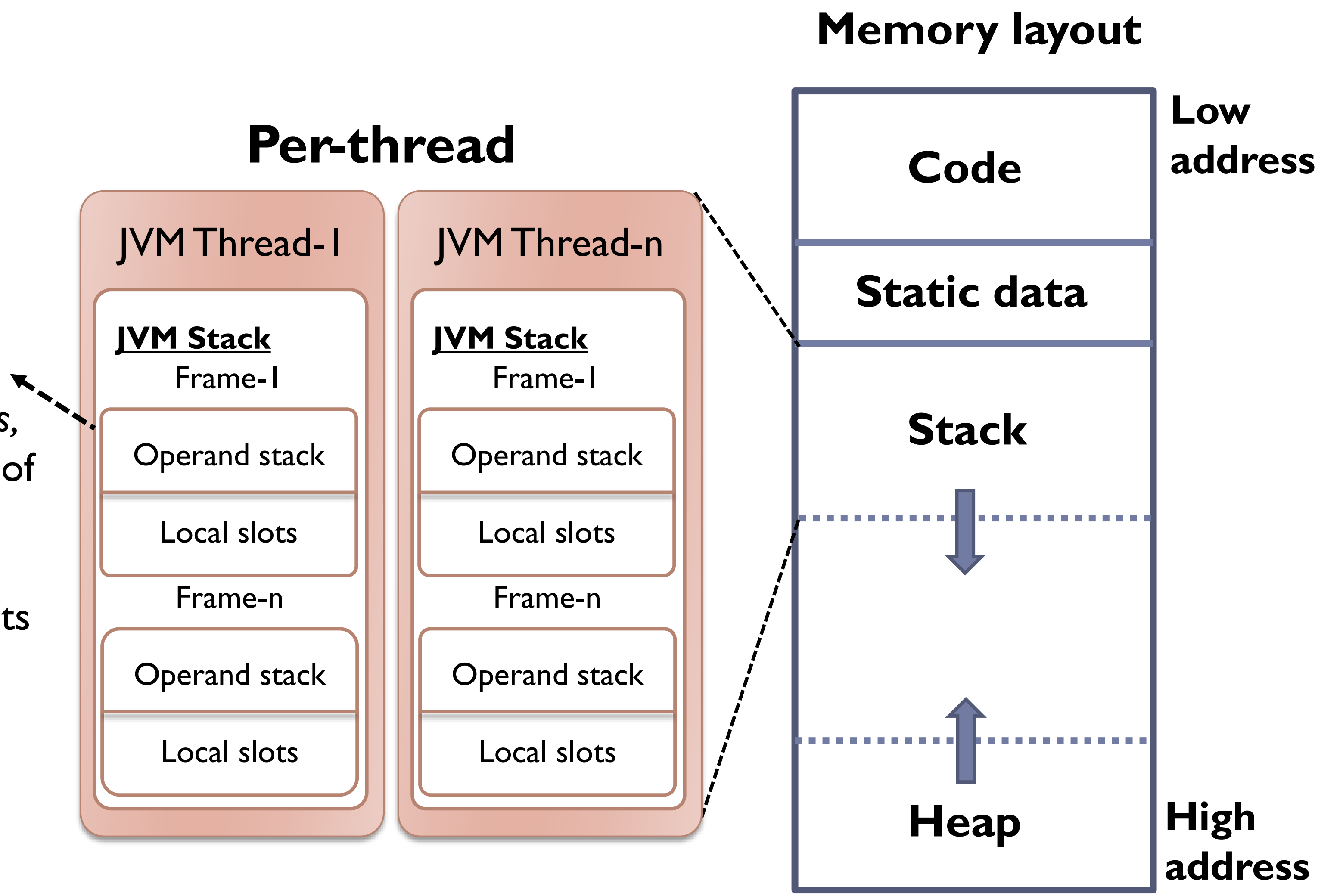


Constant Pool

- Contains all numerical and string constants
- Bytecode instructions that use constants contain indices into the constant pool where the actual value is found
  - Saving space: different instructions that use the same constant can refer to the same constant in the constant pool
- Very commonly used small constants (-1, 0, 1, 2, 3, 4) do not need to be stored in the constant pool.
  - JVM offers specialized bytecode instructions, e.g., `iconst_0` pushes the constant 0 onto the stack;
- Each stack frame contains a reference to the constant pool for the class of the frame's method.

Memory Layout of JVM

Stack:
divided into *frames*, storing the values of parameters, local variables, and intermediate results

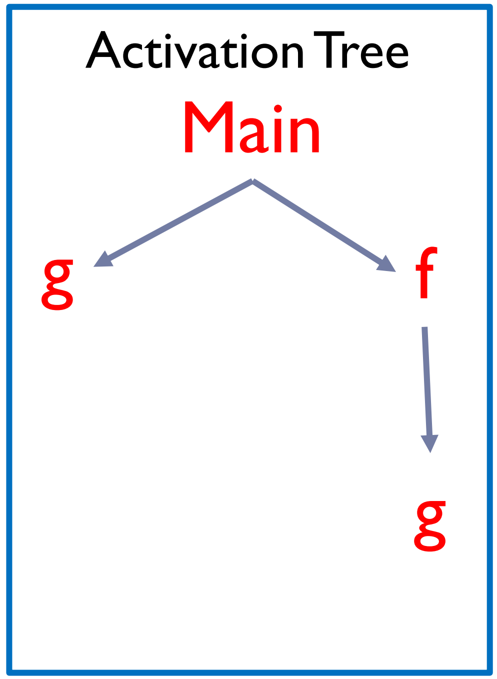


Stack

- Store local variables (including method parameters) and intermediate computation results
- A stack is subdivided into multiple frames:
  - A method is invoked: a new frame is pushed onto the stack to store local variables and intermediate results for this method;
  - A method exits: its frame is popped off, exposing the frame of its caller beneath it

Frame Examples

```
Main( ) {
  g( );
  f( );
}
f( ) {
  return g( );
}
g( ) {
  return 1;
}
```



Main's frame

g's frame

f's frame

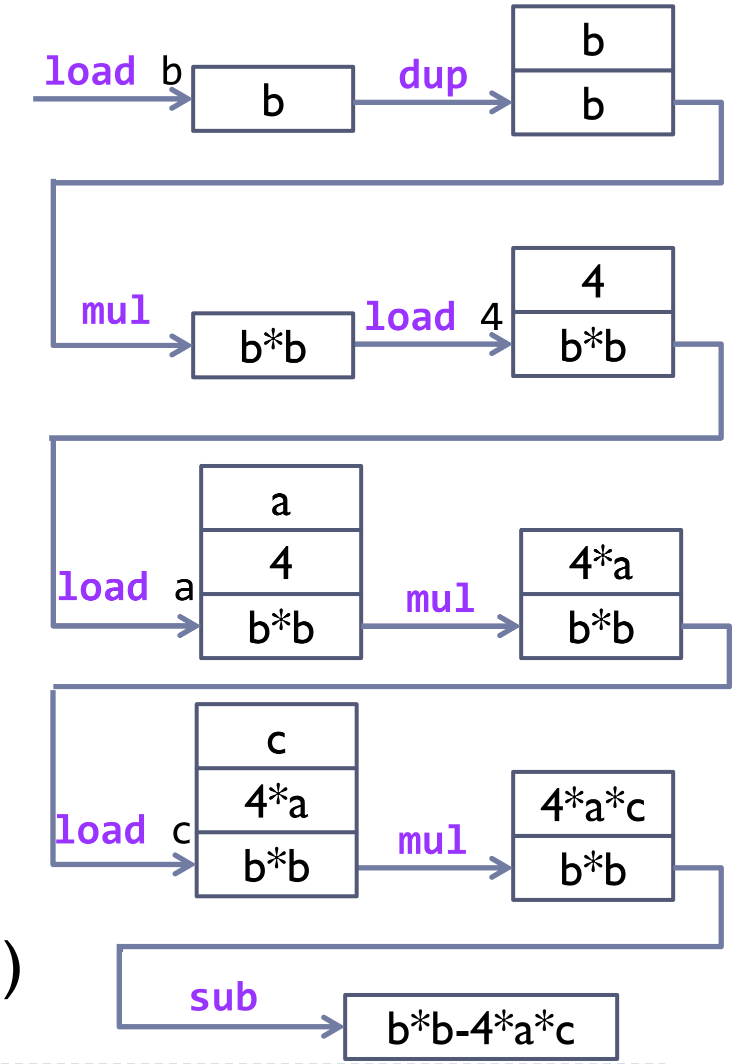
g's frame

Stack

- Store local variables (including method parameters) and intermediate computation results
- A stack is subdivided into multiple frames:
  - A method is invoked: a new frame is pushed onto the stack to store local variables and intermediate results for this method;
  - A method exits: its frame is popped off, exposing the frame of its caller beneath it
- Every frame consists of two parts
  - Local slots: store local variables
  - Operand stack: store operands

Operand Stack of Computing `b*b - 4*a*c`

`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

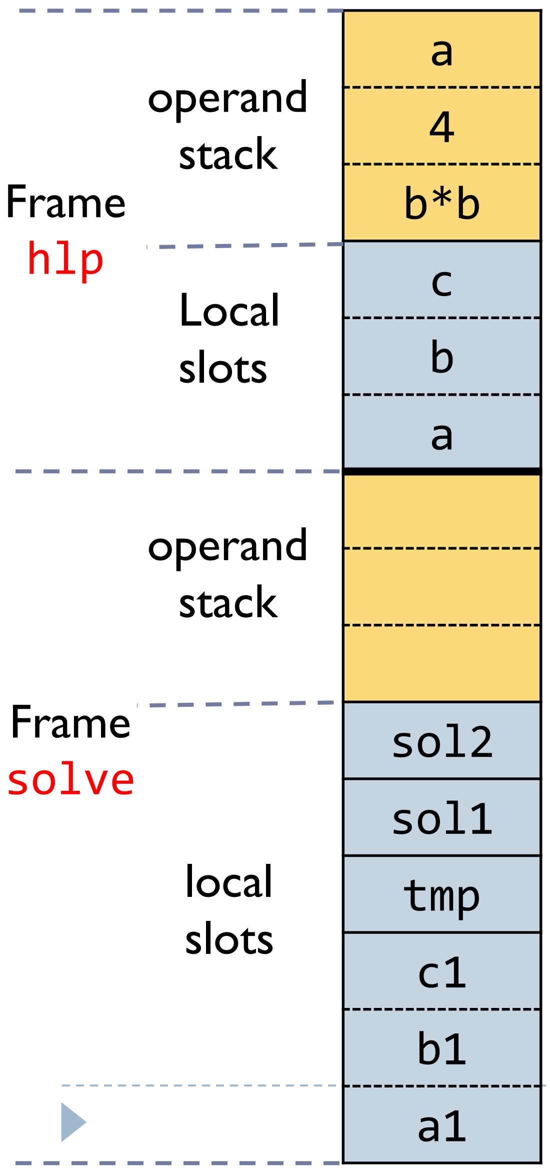


(**Note**: this is not actual JVM bytecode yet!)

Stack

- Store local variables (including method parameters) and intermediate computation results
- A stack is subdivided into multiple frames:
  - A method is invoked: a new frame is pushed onto the stack to store local variables and intermediate results for this method;
  - A method exits: its frame is popped off, exposing the frame of its caller beneath it
- Every frame consists of two parts
  - Local slots: store local variables
  - Operand stack: store operands
- A bytecode method has to indicate:
  - How many local slots it needs
  - How big its operand stack can get

Example Stack



Example methods:

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

static 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);
}
```

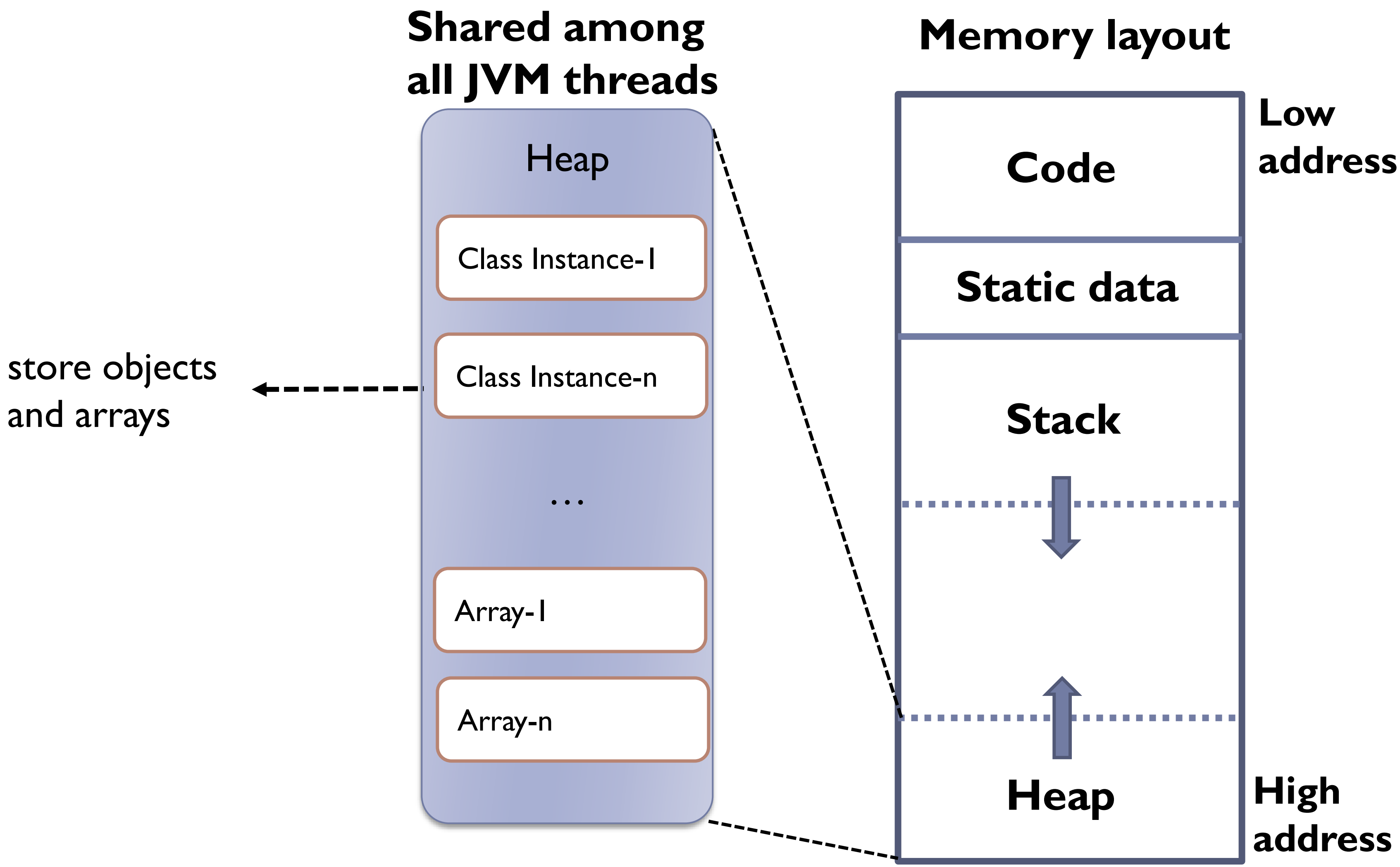
The stack on the left depicts the situation where `solve` has called `hlp`, and `hlp` is just computing its result as shown before

For instance-based (non-static) methods, slot 0 holds object's reference (not shown in diagram)

Stack Types

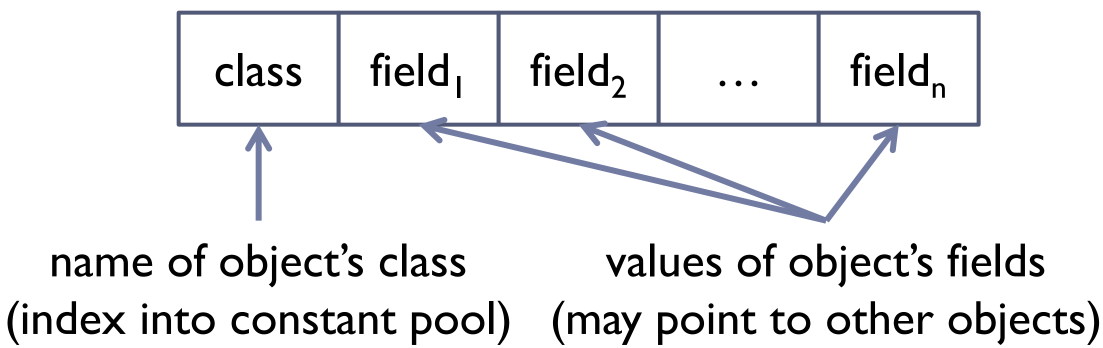
- Every element on the stack (both locals and intermediate results) has to be of one of the following types:
  1. `int`: 32-bit integer
  2. `long` : 64-bit integer
  3. `float` : 32-bit floating point
  4. `double` : 64-bit floating point
  5. `address` : pointer to object or array on heap
- `byte`,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

Memory Layout of JVM



Heap

- Store arrays and objects shared among all JVM threads
- JVM specification does not mandate a particular layout for the heap
- A common layout:



- Objects are represented on the stack as references into the heap
- Objects on the heap are only removed by the garbage collector

Outline

- Overview of Backend Synthesis

- **Java Virtual Machine**
  - Overview and memory layout
  - **Bytecode instruction set**

Bytecode Instruction Set

- 256 instructions (0 to 255)
- Most instructions take operands from the stack and leave their result on top of it
- Some instructions take extra operands encoded together with the instruction in the bytecode; such instructions are then longer than one byte
- Almost all 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

- 256 Instructions belonging to 8 categories:

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

1. Load and Store Instructions

- Loading constants onto the stack:

`iconst_0, …, iconst_5`	push **int** constant 0, …, 5
`iconst_m1`	push **int** constant -1
`aconst_null`	push constant **null**
`ldc`	push constant from constant pool

- Loading local variables onto the stack

`iload_0, …, iload_3,` `iload` *i*	push local **int** variable 0, …,3, *i* onto stack

  - Replace *i* with *l*, *f*, *d* for **long**, **float** and **double**
  - Replace load with store

2. Arithmetic Instructions

- Arithmetic and logical operators.
  - Each instruction only works on operands of one type
  - Operands are popped off the stack. Result are pushed back onto the stack

`iadd, isub`	addition (**int**), subtraction (**int**)
`idiv, irem, ineg`	division (**int**), modulo (**int**), negation (**int**)
`ishl, ishr, iushr`	`<<, >>, >>>` on **int**
`ior, iand, ixor`	`|, &, ^` on **int**

  - Replace *i* with *l*, *f*, *d* for **long**, **float** and **double**

2. Arithmetic Instructions

- Comparison instructions
  - pop operands y and x off the stack and compare them;
  - if x > y, push 1; if x < y, push -1; otherwise push 0

`lcmp`	Comparing values of type **long**

  - Replace *l* with *f*, *d* for **float** and **double.**
  - There is **no** such instruction for **int**; replaced by conditional jumps

3. Type Conversion Instructions

- Convert one type to another
  - Widening conversion
  - Narrow conversion
- Convert integers to **byte**, **short**, or **char**
  - Pop an integer from the stack, truncate its value, then sign-extended to an int, and push the results onto the stack
- No conversions from/to **address**

`i2l`, `i2f`, `i2d`, `l2f`, `l2d`, `f2d`	Widening conversion
`l2i`, `f2i`, `f2l`, `d2i`, `d2l`, `d2f`	Narrowing conversion
`i2b`, `i2s`, `i2c`	Convert integers to **byte**, **short**, or **char**

4. Object Creation and Manipulation Instructions

`new "java.lang.String"`	create new class instance. Class name is stored in constant pool. This only allocates memory without invoking the constructor
`getfield/putfield "A.f"`	access instance field value. The topmost stack value specifies the object A. Operand is the index to the constant pool for `f`
`getstatic/putstatic`	access static fields
`newarray`	allocate arrays
`arraylength`	read length of an array
`iaload/iastore`	access elements of **int** array; similar for other types
`instanceof/checkcast`	do dynamic type checks

5. Operand Stack Management Instructions

- Operate the stacks
  - These are the only type-generic instructions

dup	duplicate top stack element
pop	pop off top stack element

6. Control Transfer Instructions

- Unconditional jump

`goto t`	jump to *t*

- Conditional jumps:
  - pop off y and x; if some condition is true, then jump to *t*, otherwise continue with next instruction

`if_icmp{eq/ne/lt/le/ge/gt} t`	For **int**; jump to *t* if x == y, `!=, <, <=, >=, >`
`if_acmp{eq/ne} t`	For **address**; jump to *t* if x == y, !=

  - pop off a single operand x and compare it to 0 or null

`ifeq`, `ifne`, `iflt`, `ifle`, `ifgt`, `ifge` t	For **int**; jump to *t* if x == 0, `!=, <, <=, >=, >`
`ifnull`, `ifnonnull` t	For **address**; jump to *t* if x == null, !=

  - Conditional jumps do not apply to other types (**long**, **float** and **double**). Using comparison operations instead.

Replacement between Control Transfer and Arithmetic Instructions

- Comparison instructions are only for **long**, **float** and **double**.
  - **int**: replaced by conditional jump.

    e.g., check whether **int** x = y (`lcmp` for **long**)

```
   if_icmpeq t
   ......
t  push 0
```

- Conditional jump instructions are only for **int**
  - **long**, **float** and **double**: replaced by comparisons

    e.g., jump to t when **long** x = y (`if_icmpeq` for **int**)

```
   lcmp
   ifeq t
   ......
t
```

7. Method Invocation and Return Instructions

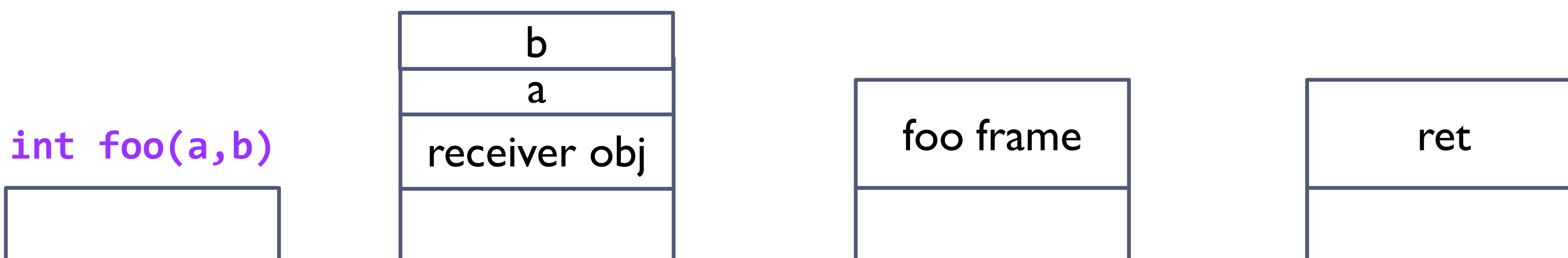
- Arguments are pushed onto stack (receiver object first for instance-based method)
- Invoke instruction (operand is the signature of the method)

`invokevirtual`	invoke normal, non-static methods
`invokeinterface`	invoke methods declared in interfaces
`invokespecial`	invoke super calls, instance initialization and private methods
`invokestatic`	invoke static methods (no receiver object)

- Method exits:

`ireturn`	return the value on top of the stack, or return to exit without a value

  - Replace *i* with *l*, *f*, *d* for **long**, **float** and **double**



8. Other Instructions

`athrow`	Exception throwing
`Monitorenter/monitorexit`	synchronization
`nop`	do nothing (surprisingly useful during compilation!)

Example: Fibonacci.java

```java
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;
    }
}
```

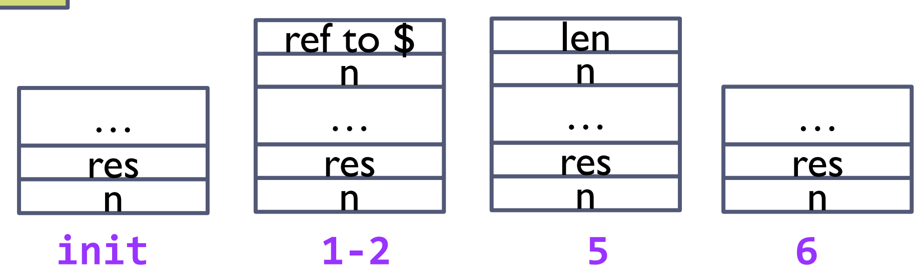
Bytecode for fib method

```
1: iload_0
2: getstatic Fibonacci.fib_cache
5: arraylength
6: if_icmpge 25
```

```
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;
```



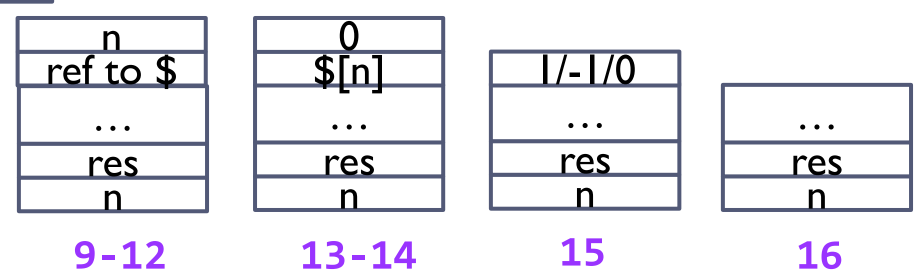
Bytecode for `fib` method

```
1:  iload_0
2:  getstatic Fibonacci.fib_cache
5:  arraylength
6:  if_icmpge 25
9:  getstatic Fibonacci.fib_cache
12: iload_0
13: laload
14: lconst_0
15: lcmp
16: ifeq 25
```

```
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;
```



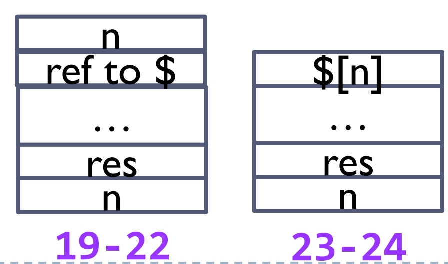
Bytecode for `fib` method

```
1:  iload_0
2:  getstatic Fibonacci.fib_cache
5:  arraylength
6:  if_icmpge 25
9:  getstatic Fibonacci.fib_cache
12: iload_0
13: laload
14: lconst_0
15: lcmp
16: ifeq 25
19: getstatic Fibonacci.fib_cache
22: iload_0
23: laload
24: lreturn
```

```
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;
```



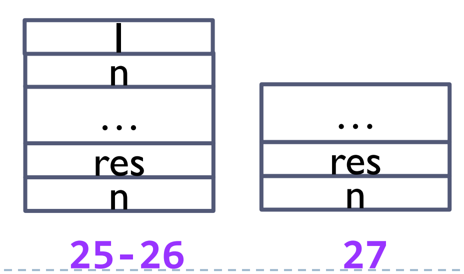
Bytecode for `fib` method

```
25: iload_0
26: iconst_1
27: if_icmpgt 35
```

```java
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;
```



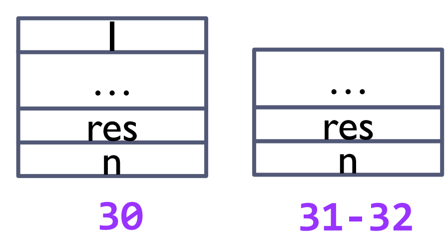
Bytecode for `fib` method

```
25: iload_0
26: iconst_1
27: if_icmpgt 35
30: lconst_1
31: lstore_1
32: goto 49
```

```
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

```
25: iload_0
26: iconst_1
27: if_icmpgt 35
30: lconst_1
31: lstore_1
32: goto 49
35: iload_0
36: iconst_2
37: isub
38: invokestatic Fibonacci.fib(int)
41: iload_0
42: iconst_1
43: isub
44: invokestatic Fibonacci.fib(int)
```

```java
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;
```

35-36	37	38	44
2			fib(n-1)
n	n-2	fib(n-2)	fib(n-2)
…	…	…	…
res	res	res	res
n	n	n	n

Bytecode for `fib` method

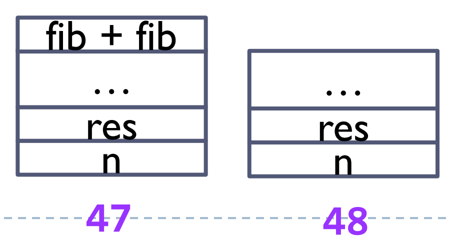
47: ladd

48: lstore_1

```
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;
```



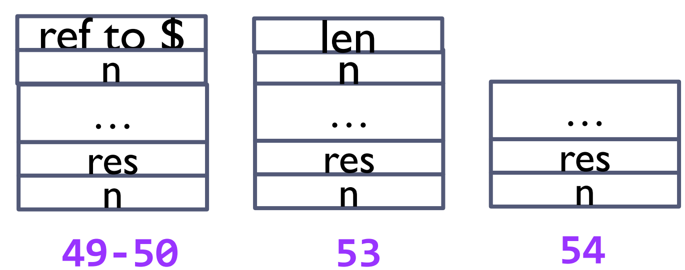
Bytecode for `fib` method

```
47: ladd
48: lstore_1
49: iload_0
50: getstatic Fibonacci.fib_cache
53: arraylength
54: if_icmpge 63
```

```
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

```
47: ladd
48: lstore_1
49: iload_0
50: getstatic Fibonacci.fib_cache
53: arraylength
54: if_icmpge 63
57: getstatic Fibonacci.fib_cache
60: iload_0
61: lload_1
62: lastore
63: lload_1
64: lreturn
```

```
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;
```