Under the Hood:
# The Java Virtual Machine

Lecture 24 – CS 2110 – Fall 2011

Compiling for Different Platforms

- Program written in some high-level language (C, Fortran, ML, ...)

- Compiled to intermediate form

- Optimized

- Code generated for various platforms (machine architecture + operating system)

- Consumers download code for their platform

Problem: Too Many Platforms!

- Operating systems

  – DOS, Win95, 98, NT, ME, 2K, XP, Vista, 7, ...

  – Unix, Linux, FreeBSD, Aix, Ubuntu, ...

  – VM/CMS, OS/2, Solaris, Mac OS X, ...

- Architectures

 – Pentium, PowerPC, Alpha, SPARC, MIPS, ...

Compiling for Different Platforms

High-Level Program

Compiler

Intermediate-Level Program

Code Gen 1   Code Gen 2   Code Gen 3

Machine Code 1   Machine Code 2   Machine Code 3

producer
consumers

4
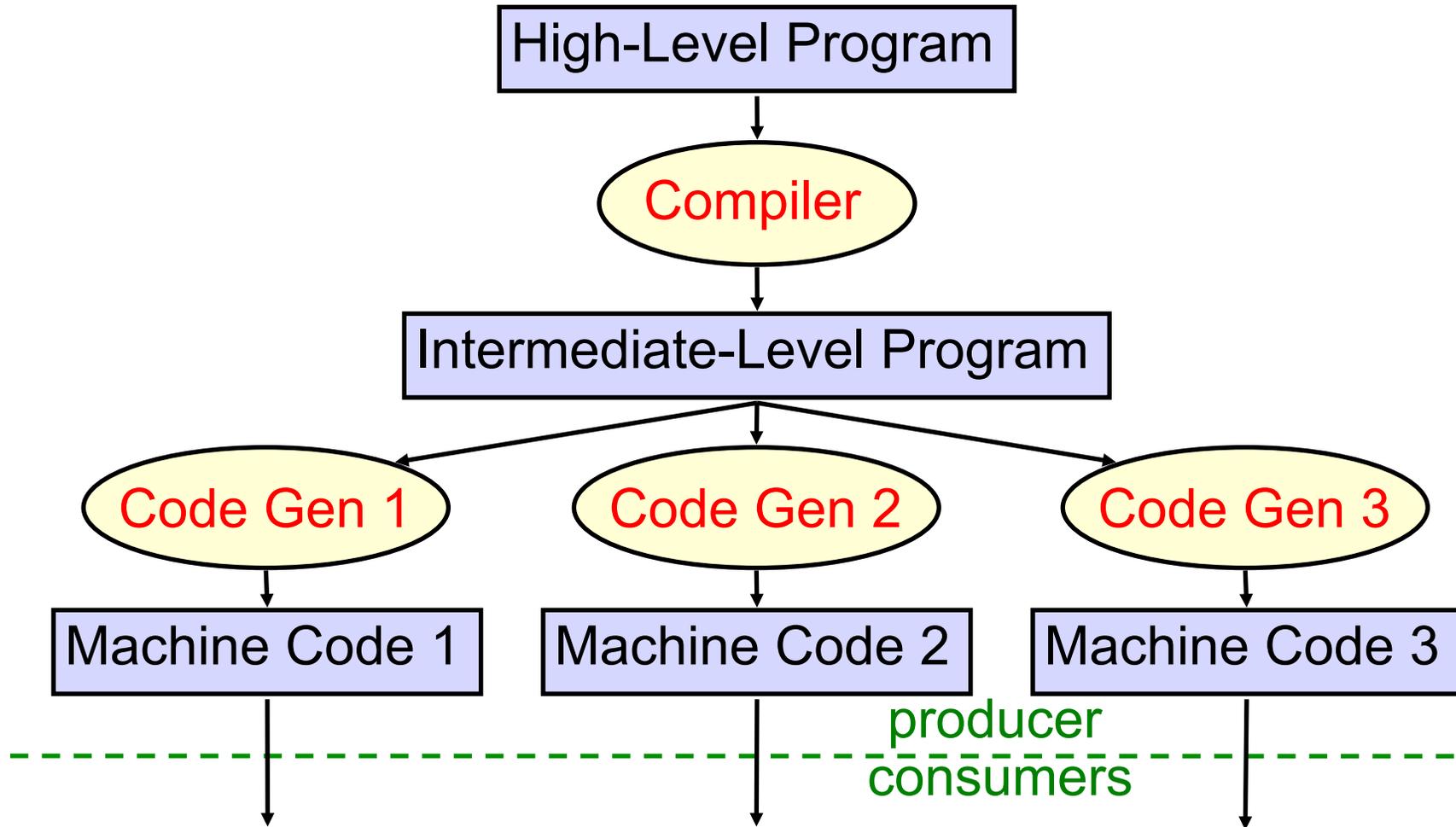
Dream: Platform Independence

- Compiler produces *one* low-level program for all platforms

- Executed on a *virtual machine* (VM)

- A different VM implementation needed for each platform, but installed once and for all

Platform Independence with Java

```
        ┌─────────────────────┐
        │    Java Program     │
        └─────────────────────┘
                   │
                   ▼
             ╭───────────╮
             │  Compiler │
             ╰───────────╯
                   │
                   ▼
        ┌─────────────────────┐
        │  Bytecode Program   │
        └─────────────────────┘
```

producer
- -
consumers

```
   ╭─────────╮     ╭─────────╮     ╭─────────╮
   │  JVM 1  │     │  JVM 2  │     │  JVM 3  │
   ╰─────────╯     ╰─────────╯     ╰─────────╯
```

Java Virtual Machines
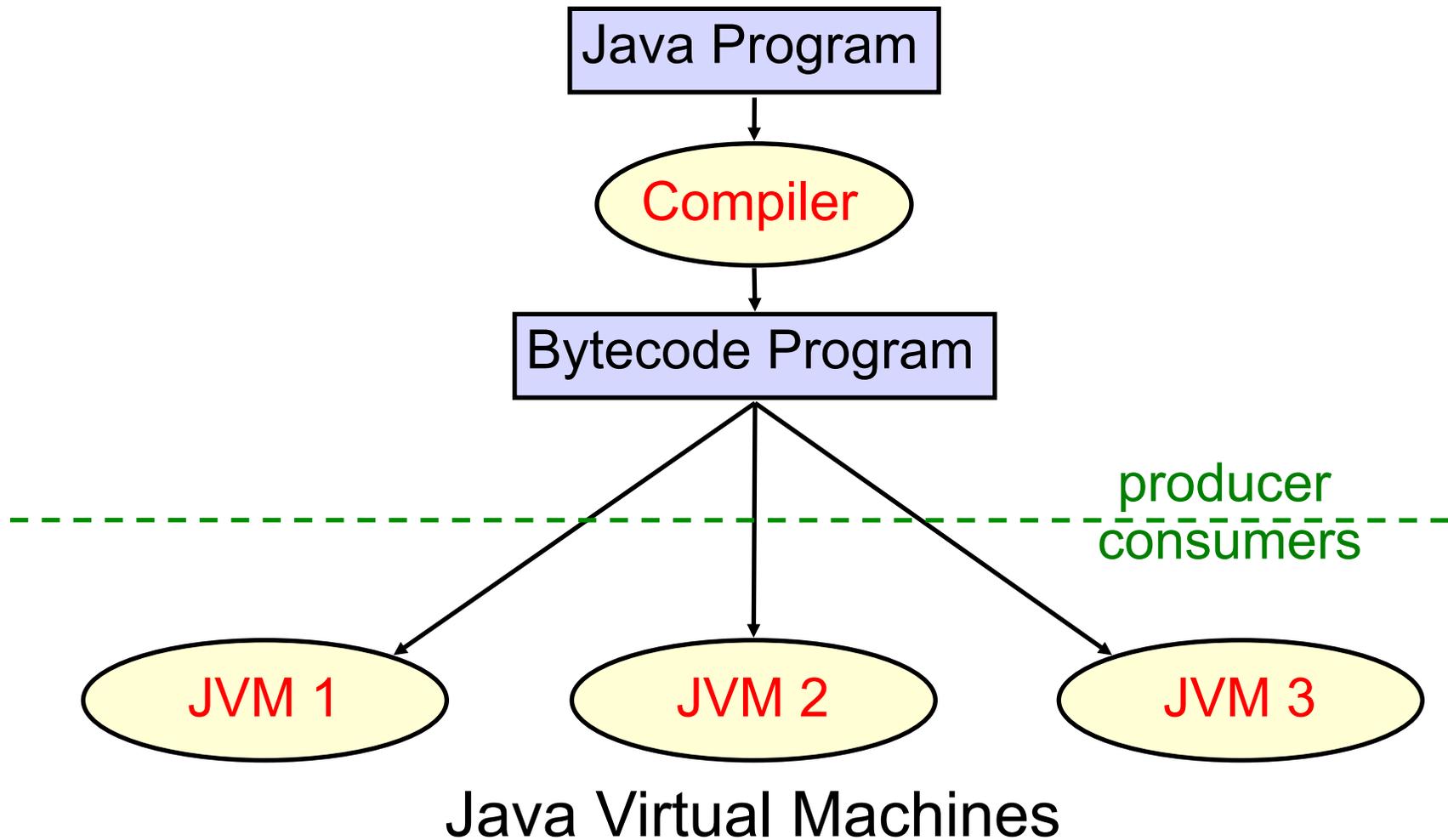
Platform Independence with Java
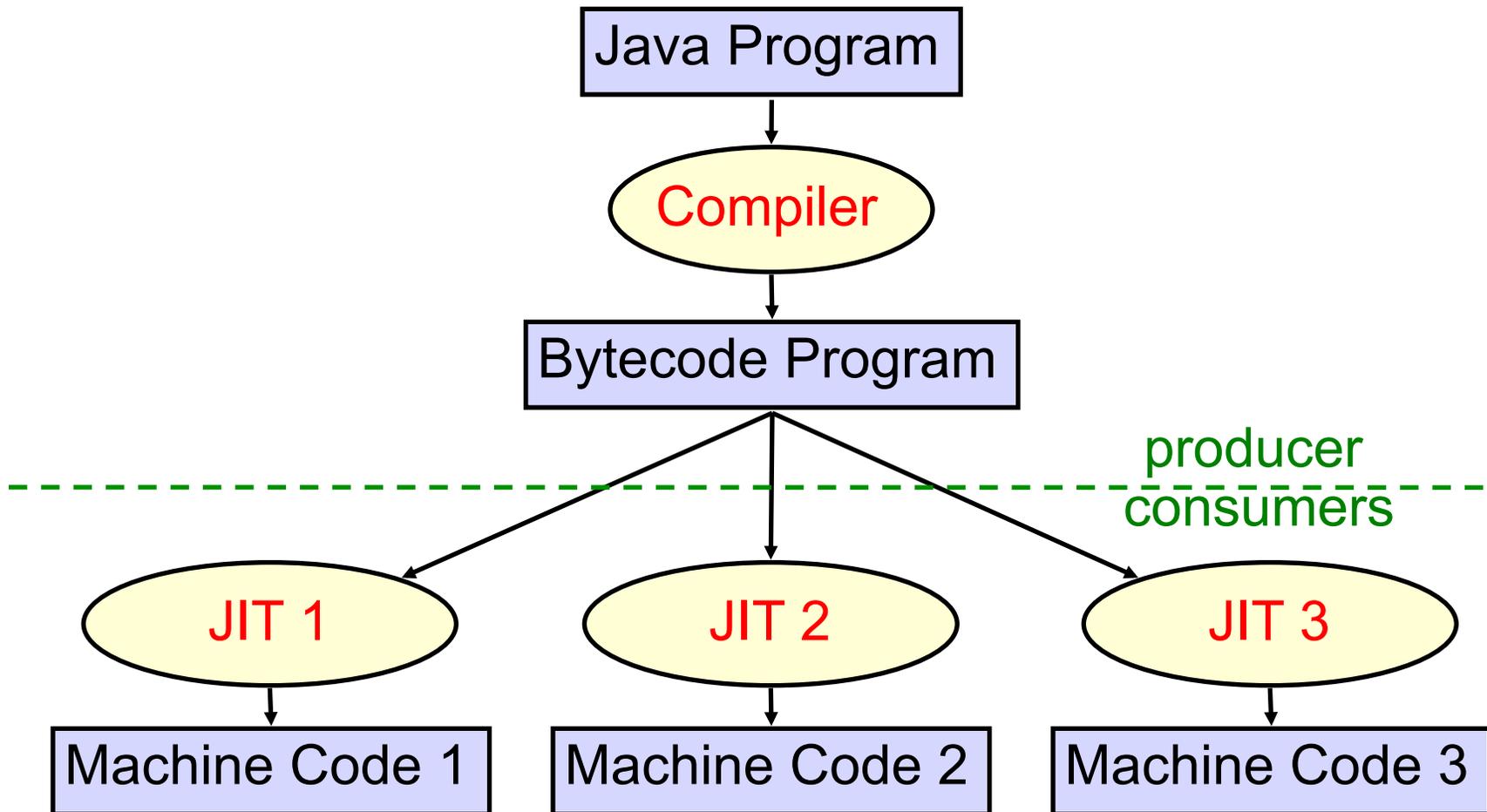
Just-in-Time Compilers

Java Bytecode

- Low-level compiled form of Java
- Platform-independent
- Compact
  - Suitable for mobile code, applets
- Easy to interpret
  - Java virtual machine (JVM) in your browser
  - Simple stack-based semantics
  - Support for objects

Class Files

InputFile.java

```
class Foo {
    ...
}

class Bar {
    ...
}

class Baz {
    ...
}
```

Compiler

Foo.class

```
...
```

Bar.class

```
...
```

Baz.class

```
...
```

9

What's in a Class File?

- Magic number, version info
- Constant pool
- Super class
- Access flags (public, private, ...)
- Interfaces
- Fields
  - Name and type
  - Access flags (public, private, static, ...)
- Methods
  - Name and signature (argument and return types)
  - Access flags (public, private, static, ...)
  - Bytecode
  - Exception tables
- Other stuff (source file, line number table, ...)

10

Class File Format

magic number	4 bytes	0xCAFEBABE
major version	2 bytes	0x0021
minor version	2 bytes	0x0000

- magic number identifies the file as a Java class file
- version numbers inform the JVM whether it is able to execute the code in the file

Constant Pool

CP length	2 bytes
CP entry 1	(variable)
CP entry 2	(variable)
...	...

- constant pool consists of up to $65536 = 2^{16}$ entries
- entries can be of various types, thus of variable length

Constant Pool Entries

Utf8 (unicode)	literal string (2 bytes length, characters)
Integer	Java int (4 bytes)
Float	Java float (4 bytes)
Long	Java long (8 bytes)
Double	Java double (8 bytes)
Class	class name
String	String constant -- index of a Utf8 entry
Fieldref	field reference -- name and type, class
Methodref	method reference -- name and type, class
InterfaceMethodref	interface method reference
NameAndType	Name and Type of a field or method

Constant Pool Entries

- Many constant pool entries refer to other constant pool entries

Fieldref
  index to a Class
    index to a Utf8 (name of class containing it)
  index to a NameAndType
    index to a Utf8 (name of field)
    index to a Utf8 (type descriptor)

- Simple text (Utf8) names used to identify classes, fields, methods -- simplifies linking

Example

```
class Foo {
    public static void main(String[] args) {
        System.out.println("Hello world");
    }
}
```

Q) How many entries in the constant pool?

Example

```
class Foo {
   public static void main(String[] args) {
      System.out.println("Hello world");
   }
}
```

Q) How many entries in the constant pool?

A) 33

```
1)CONSTANT_Methodref[10](class_index = 6, name_and_type_index = 20)
2)CONSTANT_Fieldref[9](class_index = 21, name_and_type_index = 22)
3)CONSTANT_String[8](string_index = 23)
4)CONSTANT_Methodref[10](class_index = 24, name_and_type_index = 25)
5)CONSTANT_Class[7](name_index = 26)
6)CONSTANT_Class[7](name_index = 27)
7)CONSTANT_Utf8[1]("<init>")
8)CONSTANT_Utf8[1]("()V")
9)CONSTANT_Utf8[1]("Code")
10)CONSTANT_Utf8[1]("LineNumberTable")
11)CONSTANT_Utf8[1]("LocalVariableTable")
12)CONSTANT_Utf8[1]("this")
13)CONSTANT_Utf8[1]("LFoo;")
14)CONSTANT_Utf8[1]("main")
15)CONSTANT_Utf8[1]("([Ljava/lang/String;)V")
16)CONSTANT_Utf8[1]("args")
17)CONSTANT_Utf8[1]("[Ljava/lang/String;")
18)CONSTANT_Utf8[1]("SourceFile")
19)CONSTANT_Utf8[1]("Foo.java")
20)CONSTANT_NameAndType[12](name_index = 7, signature_index = 8)
21)CONSTANT_Class[7](name_index = 28)
22)CONSTANT_NameAndType[12](name_index = 29, signature_index = 30)
23)CONSTANT_Utf8[1]("Hello world")
24)CONSTANT_Class[7](name_index = 31)
25)CONSTANT_NameAndType[12](name_index = 32, signature_index = 33)
26)CONSTANT_Utf8[1]("Foo")
27)CONSTANT_Utf8[1]("java/lang/Object")
28)CONSTANT_Utf8[1]("java/lang/System")
29)CONSTANT_Utf8[1]("out")
30)CONSTANT_Utf8[1]("Ljava/io/PrintStream;")
31)CONSTANT_Utf8[1]("java/io/PrintStream")
32)CONSTANT_Utf8[1]("println")
33)CONSTANT_Utf8[1]("(Ljava/lang/String;)V")
```

Field Table

count	2 bytes	length of table
Field Table 1	variable	index into CP
Field Table 2	variable	index into CP
...	...	...

- table of field table entries, one for each field defined in the class

Field Table Entry

access flags	2 bytes	e.g. public, static, ...
name index	2 bytes	index of a Utf8
descriptor index	2 bytes	index of a Utf8
attributes count	2 bytes	number of attributes
attribute 1	variable	e.g. constant value
attribute 2	variable	...
...	...	...

Method Table

count	2 bytes	length of table
Method Table 1	variable	index into CP
Method Table 2	variable	index into CP
...	...	...

- table of method table entries, one for each method defined in the class

Method Table Entry

access flags	2 bytes	e.g. public, static, ...
name index	2 bytes	index of a Utf8
descriptor index	2 bytes	index of a Utf8
attributes count	2 bytes	number of attributes
code attribute	variable	...
attribute 2	variable	...
...	...	...

Code Attribute of a Method

maxStack	2 bytes	max operand stack depth
maxLocals	2 bytes	number of local variables
codeLength	2 bytes	length of bytecode array
code	codeLength	the executable bytecode
excTableLength	2 bytes	number of exception handlers
exceptionTable	excTableLength	exception handler info
attributesCount	2 bytes	number of attributes
attributes	variable	e.g. LineNumberTable

Stack Frame of a Method

local variable array

this p_0 p_1 p_2

parameters   other locals

maxLocals

operand stack

maxStack
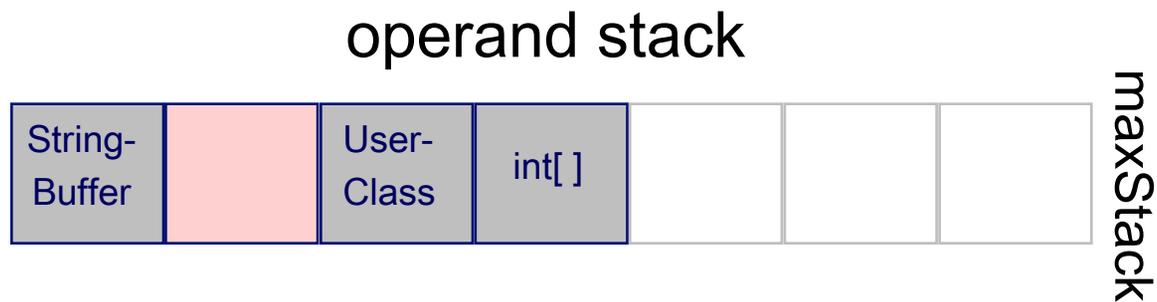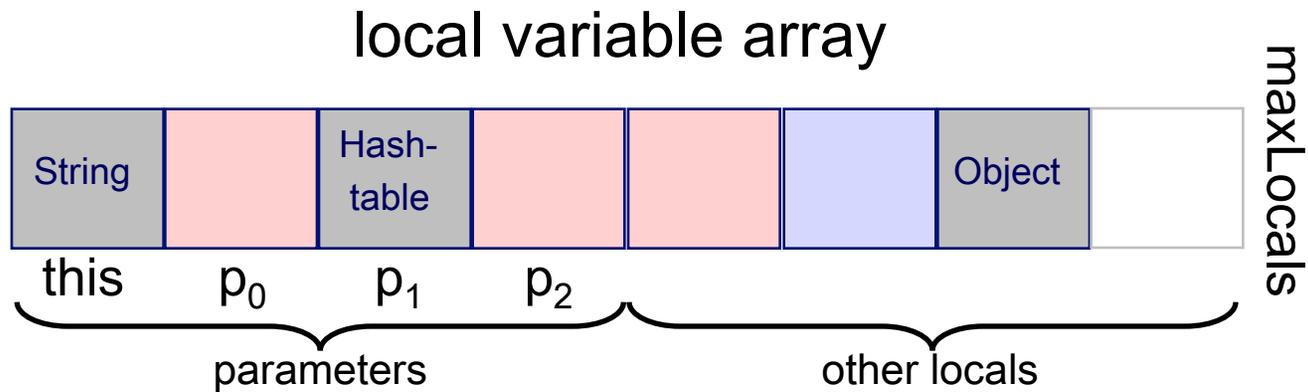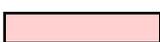
☐ = reference type
☐ = integer (boolean, byte, ...)
☐ = continuation
☐ = useless

23

Example Bytecode

```
if (b) x = y + 1;
    else x = z;
```

```
5:    iload_1    //load b
6:    ifeq 16    //if false, goto else
9:    iload_3    //load y
10:   iconst_1   //load 1
11:   iadd       //y+1
12:   istore_2   //save x
13:   goto 19    //skip else
16:   iload 4    //load z
18:   istore_2   //save x
19:   ...
```

} then clause

} else clause

Examples

```
class Foo {
   public static void main(String[] args) {
      System.out.println("Hello world");
   }
}
```

Q) How many methods?

Examples

```
class Foo {
   public static void main(String[] args) {
      System.out.println("Hello world");
   }
}
```

Q) How many methods?

A) 2

```
public static void main (String[] args)
       Code: maxStack=2 maxLocals=1 length=9
       exceptions=0
       attributes=2
          source lines=2
          local variables=1
             java/lang/String[] args startPC=0 length=9 index=0
----------------------------------------------------------------
0:    getstatic java/lang/System.out
3:    ldc "Hello world"
5:    invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
8:    return
================================================================
void <init> ()
       Code: maxStack=1 maxLocals=1 length=5
       exceptions=0
       attributes=2
          source lines=1
          local variables=1
             Foo this startPC=0 length=5 index=0
----------------------------------------------------------------
0:    aload_0
1:    invokespecial java/lang/Object.<init>()V
4:    return
```

Exception Table Entry

start	2 bytes	start of range handler is in effect
end	2 bytes	end of range handler is in effect
entry	2 bytes	entry point of exception handler
catchType	2 bytes	type of exception handled

- An exception handler is just a designated block of code
- When an exception is thrown, table is searched in order for a handler that can handle the exception

Class Loading

Java class loading is *lazy*

- A class is loaded and initialized when it (or a subclass) is first accessed
- Classname must match filename so class loader can find it
- Superclasses are loaded and initialized before subclasses
- Loading = reading in class file, verifying bytecode, integrating into the JVM

Class Initialization

- Prepare static fields with default values
  - 0 for primitive types
  - `null` for reference types

- Run static initializer `<clinit>`
  - performs programmer-defined initializations
  - only time `<clinit>` is ever run
  - only the JVM can call it

Class Initialization

```
class Staff {
  static Staff Dexter = new Staff();
  static Staff Basu = new Staff();
  static Staff Craig = new Staff();
  static Staff Sarah = new Staff();
  static Map<Staff,Job> h =
          new HashMap<Staff,Job>();
  static {
    h.put(Dexter, INSTRUCTOR);
    h.put(Basu, TA);
    h.put(Craig, TA);
    h.put(Sarah, TA);
  }
  ...
}
```

Compiled to `Staff.<clinit>`

Initialization Dependencies

```
class A {
  static int a = B.b + 1; //code in A.<clinit>
}


class B {
  static int b = 42; //code in B.<clinit>
}
```

Initialization of **A** will be suspended while
B is loaded and initialized

Initialization Dependencies

```
class A {
   static int a = B.b + 1; //code in A.<clinit>
}


class B {
   static int b = A.a + 1; //code in B.<clinit>
}
```

Q) Is this legal Java? If so, does it halt?

Initialization Dependencies

```
class A {
  static int a = B.b + 1; //code in A.<clinit>
}


class B {
  static int b = A.a + 1; //code in B.<clinit>
}
```

Q) Is this legal Java? If so, does it halt?

A) yes and yes

Initialization Dependencies

```
class A {
  static int a = B.b + 1; //code in A.<clinit>
}


class B {
  static int b = A.a + 1; //code in B.<clinit>
}
```

Q) So what are the values of `A.a` and `B.b`?

Initialization Dependencies

```
class A {
  static int a = B.b + 1; //code in A.<clinit>
}


class B {
  static int b = A.a + 1; //code in B.<clinit>
}
```

Q) So what are the values of `A.a` and `B.b`?

A) `A.a` = 1   `B.b` = 2

Initialization Dependencies

```
class A {
  static int a = B.b + 1; //code in A.<clinit>
}


class B {
  static int b = A.a + 1; //code in B.<clinit>
}
```

Q) So what are the values of **A.a** and **B.b**?

A) **A.a** = ~~1~~ 2   **B.b** = ~~2~~ 1

Object Initialization

- Object creation initiated by `new` (sometimes implicitly, e.g. by +)
- JVM allocates heap space for object – room for all instance (non-static) fields of the class, including inherited fields, dynamic type info
- Instance fields prepared with default values
  - `0` for primitive types
  - `null` for reference types

Object Initialization

- Call to object initializer `<init>(...)` explicit in the compiled code
  - `<init>` compiled from constructor
  - if none provided, use default `<init>()`
  - first operation of `<init>` must be a call to the corresponding `<init>` of superclass
  - either done explicitly by the programmer using `super(...)` or implicitly by the compiler

Object Initialization

```
class A {
  String name;
  A(String s) {
    name = s;
  }
}
```

```
<init>(java.lang.String)V
0: aload_0  //this
1: invokespecial java.lang.Object.<init>()V
4: aload_0  //this
5: aload_1  //parameter s
6: putfield A.name
9: return
```

Next Time

- Class file format

- Class loading and initialization

- Object initialization

- Method dispatch

- Exception handling

- Bytecode verification & stack inspection