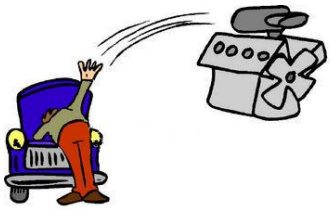


Under the Hood: The Java Virtual Machine

Lecture 23 – CS211 – Summer 2008

Announcements

- Please fill out course evaluation on-line at Summer Session website
  - Participation counts as a quiz grade!

Agenda

- Garbage collection
  - JVM memory architecture
  - Heap free-list
  - Mark and sweep collection
- Java bytecode
  - Platform independence
  - Anatomy of class files
  - Stack machine
  - Running bytecode
- Mobile code
  - Security and access restrictions

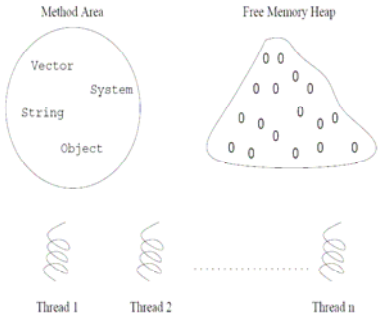
Garbage Collection: Motivation

- Memory management is error prone
  - Double free bugs
  - Memory leaks
- Can the computer do it for us?

⇒Garbage collection!

⇒Automatically free memory no longer needed

JVM Memory Architecture



Method Area

Type information (aka class definitions):

- Fully qualified name (e.g. java.lang.String)
- FQN of super class (e.g. java.lang.Object)
- Interface or class?
- Modifiers (public, abstract, final, …)
- Interfaces implemented
- Constant pool (per type)
  - (literal values & symbolic names)
- Static fields
- Method signatures & instructions (and more)

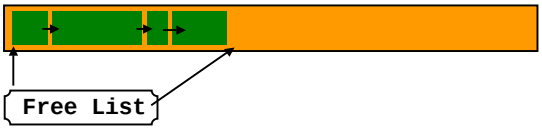
Memory Heap

Free space in heap

Object in heap

Heap = large contiguous block of memory

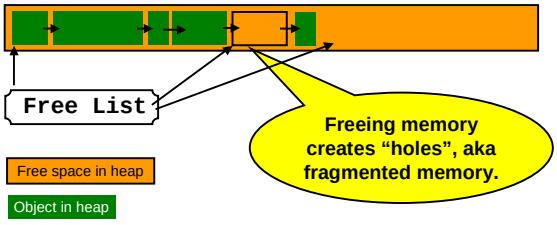
Memory Heap



Free space in heap

Object in heap

Memory Heap



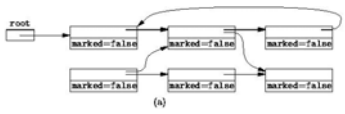
Free space in heap

Object in heap

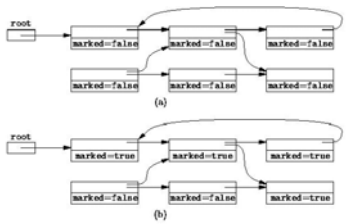
Mark & Sweep Collection

Idea:

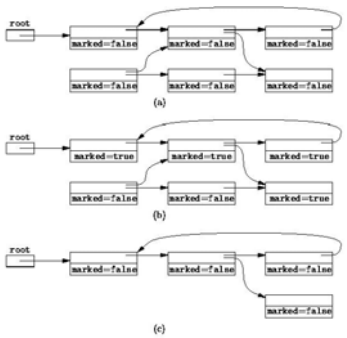
- Traverse graph of objects reachable from active memory (e.g. DFS)
- Mark visited objects as live.
- Sweep through heap and delete all non-live objects.
- Compact heap to get contiguous free space



Marking



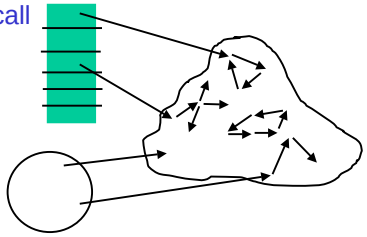
Marking



Marking & Sweeping

Roots of Active Memory?

- References in call stacks
  - For all threads
- Method area
  - E.g. static variables



Sweep & Compact

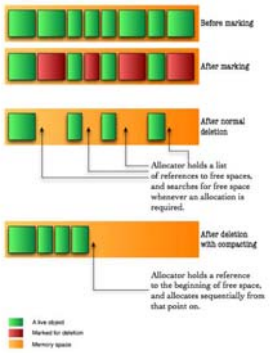


Image Source:
http://chaoticjava.com/posts/how-does-garbage-collection-work/

Comments on Garbage Collection

- Usually lazy!
- Mark & Sweep stops the world
  - More complicated algorithms address this
- Most objects short lived (*high infant mortality)*
  - Motivation for generational collection

Agenda

- Garbage collection
  - JVM memory architecture
  - Heap free-list
  - Mark and sweep collection
- Java bytecode
  - Platform independence
  - Anatomy of class files
  - Stack machine
  - Running bytecode
- Mobile code
  - Security and access restrictions


```
Compiled from "Route.java"
public class Route extends java.lang.Object implements java.lang.Comparable{

public int capacity();
  Code:
     0:   aload_0
     1:   getfield      #8; //Field connexOwners:Ljava/util/ArrayList;
     4:   invokevirtual  #10; //Method java/util/ArrayList.size:()I
     7:   ireturn

public int getDistance();
  Code:
     0:   aload_0
     1:   getfield      #5; //Field distance:I
     4:   ireturn


public CardColor getConnectionColor(int);
  Code:
     0:   aload_0
     1:   getfield      #9; //Field connexColors:Ljava/util/ArrayList;
     4:   iload_1
     5:   invokevirtual  #13; //Method java/util/ArrayList.get:(I)Ljava/lang/Object;
     8:   checkcast     #14; //class CardColor
    11:   areturn

public int compareTo(java.lang.Object);
  Code:
     0:   aload_1
     1:   checkcast     #31; //class Route
     4:   astore_2
     5:   aload_0
     6:   getfield      #3; //Field cityA:Ljava/lang/String;
     9:   aload_2
    10:   getfield      #3; //Field cityA:Ljava/lang/String;
    13:   invokevirtual  #32; //Method java/lang/String.equals:(Ljava/lang/Object;)Z
    16:   ifne          31
    19:   aload_0
    20:   getfield      #3; //Field cityA:Ljava/lang/String;
    23:   aload_2
    24:   getfield      #3; //Field cityA:Ljava/lang/String;
    27:   invokevirtual  #2; //Method java/lang/String.compareTo:(Ljava/lang/String;)I
    30:   ireturn
    31:   aload_0
    32:   getfield      #4; //Field cityB:Ljava/lang/String;
    35:   aload_2
    36:   getfield      #4; //Field cityB:Ljava/lang/String;
    39:   invokevirtual  #2; //Method java/lang/String.compareTo:(Ljava/lang/String;)I
    42:   ireturn
}
```

Java Bytecode

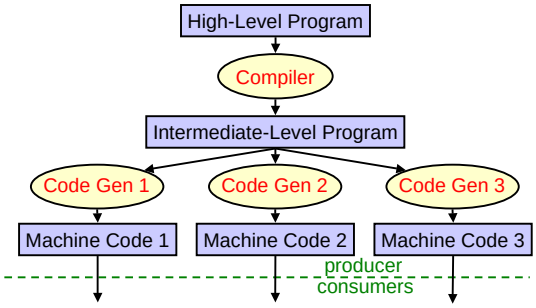
Compiling for Different Platforms

- 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, ...
  - Unix, Linux, Solaris, FreeBSD, AIX, Mac OS ...
  - VM/CMS, OS/2, ...
- Architectures
  - Pentium, PowerPC, Alpha, SPARC, MIPS, ...

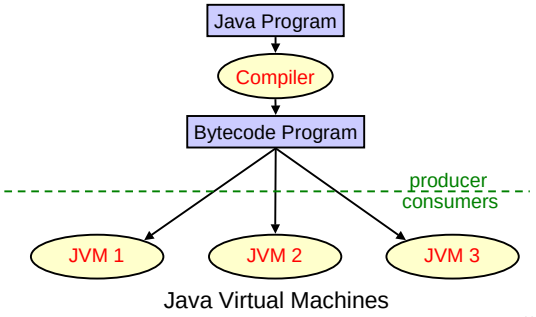
Compiling for Different Platforms



Dream: Platform Independence

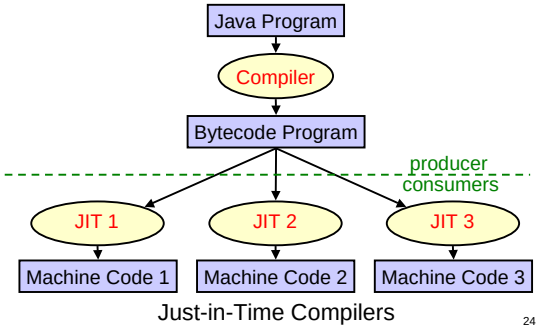
- Compiler produces *one* low-level program for all platforms
- Executed on a *virtual machine* (VM)
- A different VM implementation needed for each platform, but installed once and for all

Platform Independence with Java



Java Virtual Machines

Platform Independence with Java

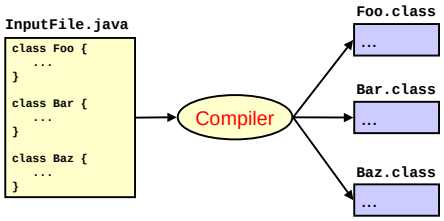


Just-in-Time Compilers

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



What is 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, ...)

Class File Plumbing

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?

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

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

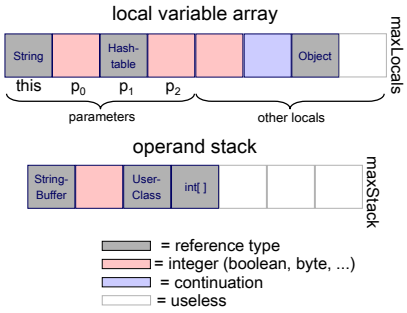
```
class Foo {
   public static void main(String[] args) {
      System.out.println("Hello world");
   }
}
```

Q) How many methods?

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
```

Stack Frame of a Method



JVM is a Stack Machine!

- Bytecode instructions manipulate top of stack.
- Arguments to instructions are implicit.

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 Dave = new Staff();
  static Staff Kelly = new Staff();
  static Map h = new HashMap();
  static {
    h.put(Dexter, INSTRUCTOR);
    h.put(Dave, INSTRUCTOR);
    h.put(Kelly, ADMINISTRATOR);
  }
  ...
}
```

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?

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**?

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
```

Instance Method Dispatch

`x.foo(...)`

- compiles to **`invokevirtual`**
- Every loaded class knows its superclass
  - name of superclass is in the constant pool
  - like a parent pointer in the class hierarchy
- bytecode evaluates arguments of **`x.foo(...)`**, pushes them on the stack
- Object **`x`** is always the first argument

Instance Method Dispatch

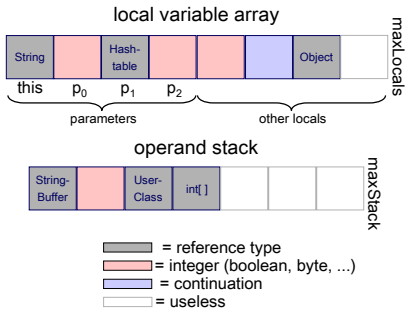
`invokevirtual foo (...)V`

- Name and type of **`foo(...)`** are arguments to **`invokevirtual`** (indices into constant pool)
- JVM retrieves them from constant pool
- Gets the dynamic (runtime) type of **`x`**
- Follows parent pointers until finds **`foo(...)V`** in one of those classes – gets bytecode from code attribute

Instance Method Dispatch

- Creates a new *stack frame* on runtime stack around arguments already there
- Allocates space in stack frame for locals and operand stack
- Prepares locals (int=0, ref=null), empty stack
- Starts executing bytecode of the method
- When returns, pops stack frame, resumes in calling method after the **`invokevirtual`** instruction

Stack Frame of a Method



Instance Method Dispatch

```
byte[] data;
void getData() {
   String x = "Hello world";
   byte[] data = x.getBytes();
}
```

```
Code(maxStack = 2, maxLocals = 2, codeLength = 12)
0: ldc "Hello world"
2: astore_1
3: aload_0 //object of which getData is a method
4: aload_1
5: invokevirtual java.lang.String.getBytes ()[B
8: putfield A.data [B
11: return
```


Exception Handling

- Each method has an *exception handler table* (possibly empty)
- Compiled from **`try/catch/finally`**
- An exception handler is just a designated block of code
- When an exception is thrown, JVM searches the exception table for an appropriate handler that is in effect
- **`finally`** clause is executed last

Exception Handling

- Finds an exception handler → empties stack, pushes exception object, executes handler
- No handler → pops runtime stack, returns exceptionally to calling routine
- **`finally`** clause is always executed, no matter what

Exception Table Entry

`startRange`	start of range handler is in effect
`endRange`	end of range handler is in effect
`handlerEntry`	entry point of exception handler
`catchType`	exception handled

- **`startRange`** → **`endRange`** give interval of instructions in which handler is in effect
- **`catchType`** is any subclass of **`Throwable`** (which is a superclass of **`Exception`**) -- any subclass of **`catchType`** can be handled by this handler

Example

```
Integer x = null;
Object y = new Object();

try {
    x = (Integer)y;
    System.out.println(x.intValue());
} catch (ClassCastException e) {
    System.out.println("y was not an Integer");
} catch (NullPointerException e) {
    System.out.println("y was null");
} finally {
    System.out.println("finally!");
}
```



```
0: aconst_null
1: astore_1
2: new java.lang.Object
5: dup
6: invokespecial java.lang.Object.<init> ()V
9: astore_2
10: aload_2
11: checkcast java.lang.Integer
14: astore_1
15: getstatic java.lang.System.out Ljava/io/PrintStream;
18: aload_1
19: invokevirtual java.lang.Integer.intValue ()I
22: invokevirtual java.io.PrintStream.println (I)V
25: getstatic java.lang.System.out Ljava/io/PrintStream;
28: ldc "finally!"
30: invokevirtual java.io.PrintStream.println (Ljava/lang/String;)V
33: goto #89
36: astore_3
37: getstatic java.lang.System.out Ljava/io/
40: ldc "y was not an Integer"
42: invokevirtual java.io.PrintStream.println
45: getstatic java.lang.System.out Ljava/io/
48: ldc "finally!"
50: invokevirtual java.io.PrintStream.println
53: goto #89
56: astore_3
57: getstatic java.lang.System.out Ljava/io/
60: ldc "y was null"
62: invokevirtual java.io.PrintStream.println
65: getstatic java.lang.System.out Ljava/io/
68: ldc "finally!"
70: invokevirtual java.io.PrintStream.println
73: goto #89
76: astore 4
78: getstatic java.lang.System.out Ljava/io/
81: ldc "finally!"
83: invokevirtual java.io.PrintStream.println (Ljava/lang/String;)V
86: aload 4
88: athrow
89: return
```

```
From To Handler Type
10   25   36   java.lang.ClassCastException
10   25   56   java.lang.NullPointerException
10   25   76   <Any exception>
36   45   76   <Any exception>
56   65   76   <Any exception>
76   78   76   <Any exception>
```

```
Integer x = null;
Object y = new Object();

try {
    x = (Integer)y;
    System.out.println(x.intValue());
} catch (ClassCastException e) {
    System.out.println("y was not an Integer");
} catch (NullPointerException e) {
    System.out.println("y was null");
} finally {
    System.out.println("finally!");
}
```



```
0: aconst_null
1: astore_1
2: new java.lang.Object
5: dup
6: invokespecial java.lang.Object.<init> ()V
9: astore_2
10: aload_2
11: checkcast java.lang.Integer
14: astore_1
15: getstatic java.lang.System.out Ljava/io/PrintStream;
18: aload_1
19: invokevirtual java.lang.Integer.intValue ()I
22: invokevirtual java.io.PrintStream.println (I)V
25: getstatic java.lang.System.out Ljava/io/PrintStream;
28: ldc "finally!"
30: invokevirtual java.io.PrintStream.println (Ljava/lang/String;)V
33: goto #89
36: astore_3
37: getstatic java.lang.System.out Ljava/io/
40: ldc "y was not an Integer"
42: invokevirtual java.io.PrintStream.println
45: getstatic java.lang.System.out Ljava/io/
48: ldc "finally!"
50: invokevirtual java.io.PrintStream.println
53: goto #89
56: astore_3
57: getstatic java.lang.System.out Ljava/io/
60: ldc "y was null"
62: invokevirtual java.io.PrintStream.println
65: getstatic java.lang.System.out Ljava/io/
68: ldc "finally!"
70: invokevirtual java.io.PrintStream.println
73: goto #89
76: astore 4
78: getstatic java.lang.System.out Ljava/io/
81: ldc "finally!"
83: invokevirtual java.io.PrintStream.println (Ljava/lang/String;)V
86: aload 4
88: athrow
89: return
```

```
From To Handler Type
10   25   36   java.lang.ClassCastException
10   25   56   java.lang.NullPointerException
10   25   76   <Any exception>
36   45   76   <Any exception>
56   65   76   <Any exception>
76   78   76   <Any exception>
```

```
Integer x = null;
Object y = new Object();

try {
    x = (Integer)y;
    System.out.println(x.intValue());
} catch (ClassCastException e) {
    System.out.println("y was not an Integer");
} catch (NullPointerException e) {
    System.out.println("y was null");
} finally {
    System.out.println("finally!");
}
```



```
0: aconst_null
1: astore_1
2: new java.lang.Object
5: dup
6: invokespecial java.lang.Object.<init> ()V
9: astore_2
10: aload_2
11: checkcast java.lang.Integer
14: astore_1
15: getstatic java.lang.System.out Ljava/io/PrintStream;
18: aload_1
19: invokevirtual java.lang.Integer.intValue ()I
22: invokevirtual java.io.PrintStream.println (I)V
25: getstatic java.lang.System.out Ljava/io/PrintStream;
28: ldc "finally!"
30: invokevirtual java.io.PrintStream.println (Ljava/lang/String;)V
33: goto #89
36: astore_3
37: getstatic java.lang.System.out Ljava/io/
40: ldc "y was not an Integer"
42: invokevirtual java.io.PrintStream.println
45: getstatic java.lang.System.out Ljava/io/
48: ldc "finally!"
50: invokevirtual java.io.PrintStream.println
53: goto #89
56: astore_3
57: getstatic java.lang.System.out Ljava/io/
60: ldc "y was null"
62: invokevirtual java.io.PrintStream.println
65: getstatic java.lang.System.out Ljava/io/
68: ldc "finally!"
70: invokevirtual java.io.PrintStream.println
73: goto #89
76: astore 4
78: getstatic java.lang.System.out Ljava/io/
81: ldc "finally!"
83: invokevirtual java.io.PrintStream.println (Ljava/lang/String;)V
86: aload 4
88: athrow
89: return
```

```
From To Handler Type
10   25   36   java.lang.ClassCastException
10   25   56   java.lang.NullPointerException
10   25   76   <Any exception>
36   45   76   <Any exception>
56   65   76   <Any exception>
76   78   76   <Any exception>
```

```
Integer x = null;
Object y = new Object();

try {
    x = (Integer)y;
    System.out.println(x.intValue());
} catch (ClassCastException e) {
    System.out.println("y was not an Integer");
} catch (NullPointerException e) {
    System.out.println("y was null");
} finally {
    System.out.println("finally!");
}
```

```
0: aconst_null
1: astore_1
2: new java.lang.Object
5: dup
6: invokespecial java.lang.Object.<init> ()V
9: astore_2
10: aload_2
11: checkcast java.lang.Integer
14: astore_1
15: getstatic java.lang.System.out Ljava/io/PrintStream;
18: aload_1
19: invokevirtual java.lang.Integer.intValue ()I
22: invokevirtual java.io.PrintStream.println (I)V
25: getstatic java.lang.System.out Ljava/io/PrintStream;
28: ldc "finally!"
30: invokevirtual java.io.PrintStream.println (Ljava/lang/String;)V
33: goto #89
36: astore_3
37: getstatic java.lang.System.out Ljava/io/
40: ldc "y was not an Integer"
42: invokevirtual java.io.PrintStream.println
45: getstatic java.lang.System.out Ljava/io/
48: ldc "finally!"
50: invokevirtual java.io.PrintStream.println
53: goto #89
56: astore_3
57: getstatic java.lang.System.out Ljava/io/
60: ldc "y was null"
62: invokevirtual java.io.PrintStream.println
65: getstatic java.lang.System.out Ljava/io/
68: ldc "finally!"
70: invokevirtual java.io.PrintStream.println
73: goto #89
76: astore 4
78: getstatic java.lang.System.out Ljava/io/
81: ldc "finally!"
83: invokevirtual java.io.PrintStream.println (Ljava/lang/String;)V
86: aload 4
88: athrow
89: return
```

From	To	Handler	Type
10	25	36	java.lang.ClassCastException
10	25	56	java.lang.NullPointerException
10	25	76	<Any exception>
36	45	76	<Any exception>
56	65	76	<Any exception>
76	78	76	<Any exception>

```
Integer x = null;
Object y = new Object();

try {
   x = (Integer)y;
   System.out.println(x.intValue());
} catch (ClassCastException e) {
   System.out.println("y was not an Integer");
} catch (NullPointerException e) {
   System.out.println("y was null");
} finally {
   System.out.println("finally!");
}
```



```
0: aconst_null
1: astore_1
2: new java.lang.Object
5: dup
6: invokespecial java.lang.Object.<init> ()V
9: astore_2
10: aload_2
11: checkcast java.lang.Integer
14: astore_1
15: getstatic java.lang.System.out Ljava/io/PrintStream;
18: aload_1
19: invokevirtual java.lang.Integer.intValue ()I
22: invokevirtual java.io.PrintStream.println (I)V
25: getstatic java.lang.System.out Ljava/io/PrintStream;
28: ldc "finally!"
30: invokevirtual java.io.PrintStream.println (Ljava/lang/String;)V
33: goto #89
36: astore_3
37: getstatic java.lang.System.out Ljava/io/
40: ldc "y was not an Integer"
42: invokevirtual java.io.PrintStream.println
45: getstatic java.lang.System.out Ljava/io/
48: ldc "finally!"
50: invokevirtual java.io.PrintStream.println
53: goto #89
56: astore_3
57: getstatic java.lang.System.out Ljava/io/
60: ldc "y was null"
62: invokevirtual java.io.PrintStream.println
65: getstatic java.lang.System.out Ljava/io/
68: ldc "finally!"
70: invokevirtual java.io.PrintStream.println
73: goto #89
76: astore 4
78: getstatic java.lang.System.out Ljava/io/
81: ldc "finally!"
83: invokevirtual java.io.PrintStream.println (Ljava/lang/String;)V
86: aload 4
88: athrow
89: return
```

From	To	Handler	Type
10	25	36	java.lang.ClassCastException
10	25	56	java.lang.NullPointerException
10	25	76	<Any exception>
36	45	76	<Any exception>
56	65	76	<Any exception>
76	78	76	<Any exception>

```
Integer x = null;
Object y = new Object();

try {
   x = (Integer)y;
   System.out.println(x.intValue());
} catch (ClassCastException e) {
   System.out.println("y was not an Integer");
} catch (NullPointerException e) {
   System.out.println("y was null");
} finally {
   System.out.println("finally!");
}
```



```
0: aconst_null
1: astore_1
2: new java.lang.Object
5: dup
6: invokespecial java.lang.Object.<init> ()V
9: astore_2
10: aload_2
11: checkcast java.lang.Integer
14: astore_1
15: getstatic java.lang.System.out Ljava/io/PrintStream;
18: aload_1
19: invokevirtual java.lang.Integer.intValue ()I
22: invokevirtual java.io.PrintStream.println (I)V
25: getstatic java.lang.System.out Ljava/io/PrintStream;
28: ldc "finally!"
30: invokevirtual java.io.PrintStream.println (Ljava/lang/String;)V
33: goto #89
36: astore_3
37: getstatic java.lang.System.out Ljava/io/
40: ldc "y was not an Integer"
42: invokevirtual java.io.PrintStream.println
45: getstatic java.lang.System.out Ljava/io/
48: ldc "finally!"
50: invokevirtual java.io.PrintStream.println
53: goto #89
56: astore_3
57: getstatic java.lang.System.out Ljava/io/
60: ldc "y was null"
62: invokevirtual java.io.PrintStream.println
65: getstatic java.lang.System.out Ljava/io/
68: ldc "finally!"
70: invokevirtual java.io.PrintStream.println
73: goto #89
76: astore 4
78: getstatic java.lang.System.out Ljava/io/
81: ldc "finally!"
83: invokevirtual java.io.PrintStream.println (Ljava/lang/String;)V
86: aload 4
88: athrow
89: return
```

From	To	Handler	Type
10	25	36	java.lang.ClassCastException
10	25	56	java.lang.NullPointerException
10	25	76	<Any exception>
36	45	76	<Any exception>
56	65	76	<Any exception>
76	78	76	<Any exception>

```
Integer x = null;
Object y = new Object();

try {
   x = (Integer)y;
   System.out.println(x.intValue());
} catch (ClassCastException e) {
   System.out.println("y was not an Integer");
} catch (NullPointerException e) {
   System.out.println("y was null");
} finally {
   System.out.println("finally!");
}
```



```
0: aconst_null
1: astore_1
2: new java.lang.Object
5: dup
6: invokespecial java.lang.Object.<init> ()V
9: astore_2
10: aload_2
11: checkcast java.lang.Integer
14: astore_1
15: getstatic java.lang.System.out Ljava/io/PrintStream;
18: aload_1
19: invokevirtual java.lang.Integer.intValue ()I
22: invokevirtual java.io.PrintStream.println (I)V
25: getstatic java.lang.System.out Ljava/io/PrintStream;
28: ldc "finally!"
30: invokevirtual java.io.PrintStream.println (Ljava/lang/String;)V
33: goto #89
36: astore_3
37: getstatic java.lang.System.out Ljava/io/
40: ldc "y was not an Integer"
42: invokevirtual java.io.PrintStream.println
45: getstatic java.lang.System.out Ljava/io/
48: ldc "finally!"
50: invokevirtual java.io.PrintStream.println
53: goto #89
56: astore_3
57: getstatic java.lang.System.out Ljava/io/
60: ldc "y was null"
62: invokevirtual java.io.PrintStream.println
65: getstatic java.lang.System.out Ljava/io/
68: ldc "finally!"
70: invokevirtual java.io.PrintStream.println
73: goto #89
76: astore 4
78: getstatic java.lang.System.out Ljava/io/
81: ldc "finally!"
83: invokevirtual java.io.PrintStream.println (Ljava/lang/String;)V
86: aload 4
88: athrow
89: return
```

From	To	Handler	Type
10	25	36	java.lang.ClassCastException
10	25	56	java.lang.NullPointerException
10	25	76	<Any exception>
36	45	76	<Any exception>
56	65	76	<Any exception>
76	78	76	<Any exception>

```
Integer x = null;
Object y = new Object();

try {
   x = (Integer)y;
   System.out.println(x.intValue());
} catch (ClassCastException e) {
   System.out.println("y was not an Integer");
} catch (NullPointerException e) {
   System.out.println("y was null");
} finally {
   System.out.println("finally!");
}
```



```
0: aconst_null
1: astore_1
2: new java.lang.Object
5: dup
6: invokespecial java.lang.Object.<init> ()V
9: astore_2
10: aload_2
11: checkcast java.lang.Integer
14: astore_1
15: getstatic java.lang.System.out Ljava/io/PrintStream;
18: aload_1
19: invokevirtual java.lang.Integer.intValue ()I
22: invokevirtual java.io.PrintStream.println (I)V
25: getstatic java.lang.System.out Ljava/io/PrintStream;
28: ldc "finally!"
30: invokevirtual java.io.PrintStream.println (Ljava/lang/String;)V
33: goto #89
36: astore_3
37: getstatic java.lang.System.out Ljava/io/
40: ldc "y was not an Integer"
42: invokevirtual java.io.PrintStream.println
45: getstatic java.lang.System.out Ljava/io/
48: ldc "finally!"
50: invokevirtual java.io.PrintStream.println
53: goto #89
56: astore_3
57: getstatic java.lang.System.out Ljava/io/
60: ldc "y was null"
62: invokevirtual java.io.PrintStream.println
65: getstatic java.lang.System.out Ljava/io/
68: ldc "finally!"
70: invokevirtual java.io.PrintStream.println
73: goto #89
76: astore 4
78: getstatic java.lang.System.out Ljava/io/
81: ldc "finally!"
83: invokevirtual java.io.PrintStream.println (Ljava/lang/String;)V
86: aload 4
88: athrow
89: return
```

From	To	Handler	Type
10	25	36	java.lang.ClassCastException
10	25	56	java.lang.NullPointerException
10	25	76	<Any exception>
36	45	76	<Any exception>
56	65	76	<Any exception>
76	78	76	<Any exception>

```
Integer x = null;
Object y = new Object();

try {
   x = (Integer)y;
   System.out.println(x.intValue());
} catch (ClassCastException e) {
   System.out.println("y was not an Integer");
} catch (NullPointerException e) {
   System.out.println("y was null");
} finally {
   System.out.println("finally!");
}
```

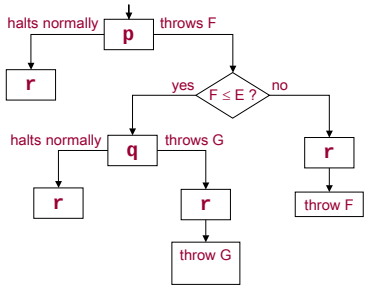

Try/Catch/Finally

`try {p} catch (E) {q} finally {r}`

- **r** is always executed, regardless of whether **p** and/or **q** halt normally or exceptionally
- If **p** throws an exception not caught by the catch clause, or if **q** throws an exception, that exception is *rethrown* upon normal termination of **r**

Try/Catch/Finally

```
try {p} catch (E) {q} finally {r}
```



Java Bytecode Security

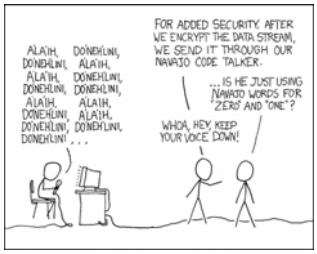


Image Source:
http://imgs.xkcd.com/comics/code_talkers.png

Java Security Model

- Bytecode verification
  - Type safety
  - Private/protected/package/final annotations
  - Basis for the entire security model
  - Prevents circumvention of higher-level checks
- Secure class loading
  - Guards against substitution of malicious code for standard system classes
- Stack inspection
  - Mediates access to critical resources

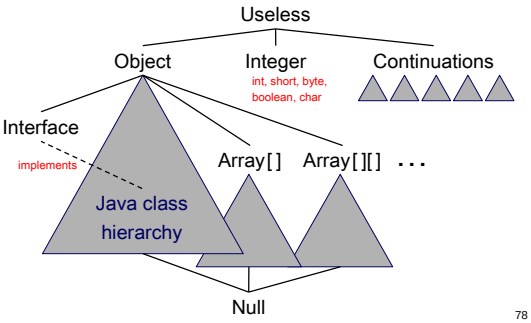
Bytecode Verification

- Performed at load time
- Enforces type safety
  - All operations are well-typed (e.g., may not confuse refs and ints)
  - Array bounds
  - Operand stack overflow, underflow
  - Consistent state over all dataflow paths
- Private/protected/package/final annotations

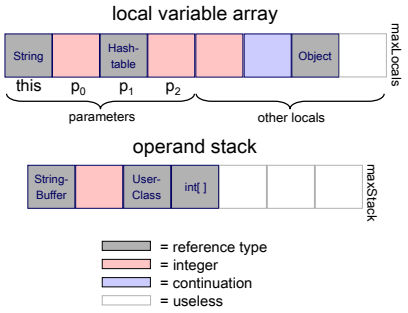
Bytecode Verification

- A form of *dataflow analysis* or *abstract interpretation* performed at load time
- Annotate the program with information about the execution state at each point
- Guarantees that values are used correctly

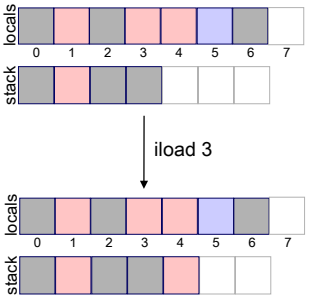
Types in the JVM



Typing of Java Bytecode



Example



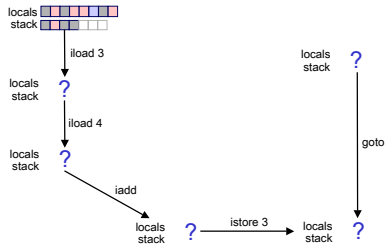
Preconditions for safe execution:

- local 3 is an integer
- stack is not full

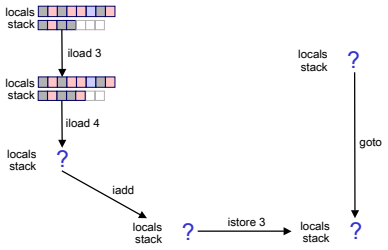
Effect:

- push integer in local 3 on stack

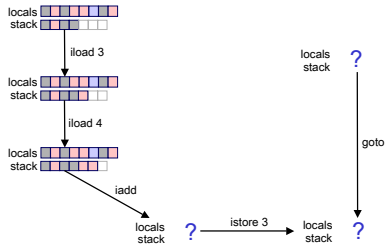
Example



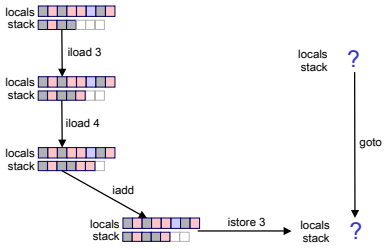
Example



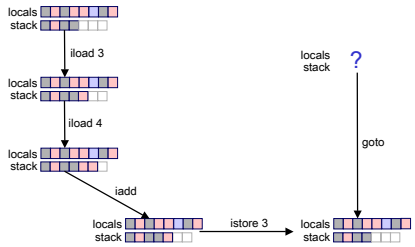
Example



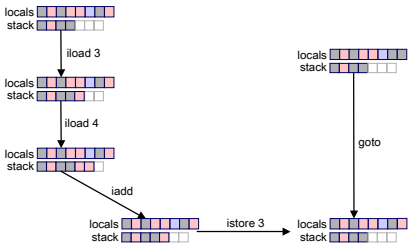
Example



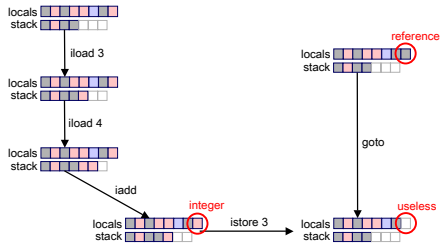
Example



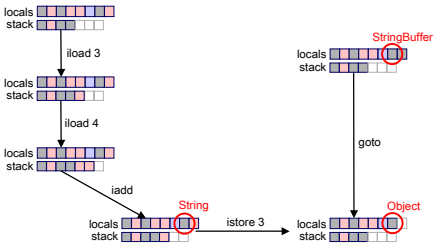
Example



Example



Example

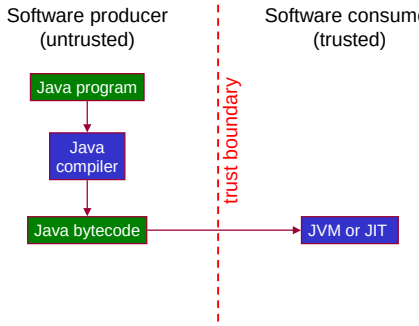


Agenda

- Garbage collection
  - JVM memory architecture
  - Heap free-list
  - Mark and sweep collection
- Java bytecode
  - Platform independence
  - Anatomy of class files
  - Stack machine
  - Running bytecode
- Mobile code
  - Security and access restrictions

Mobile Code

Mobile Code



Mobile Code

Problem: mobile code is not trustworthy!

- We often have *trusted* and *untrusted* code running together in the same virtual machine
  - e.g., applets downloaded off the net and running in our browser
- Do not want untrusted code to perform critical operations (file I/O, net I/O, class loading, security management,...)
- *How do we prevent this?*

Mobile Code

Early approach: *signed applets*

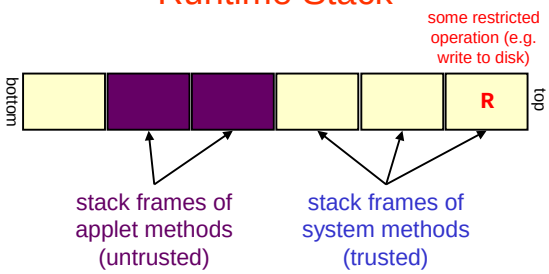
- Not so great
  - everything is either trusted or untrusted, nothing in between
  - a signature can only *verify* an already existing relationship of trust, it cannot *create* trust
- Would like to allow untrusted code to interact with trusted code
  - just monitor its activity somehow

Mobile Code

Q) Why not just let trusted (system) code do anything it wants, even in the presence of untrusted code?

A) Because untrusted code calls system code to do stuff (file I/O, etc.) – system code could be operating on behalf of untrusted code

Runtime Stack

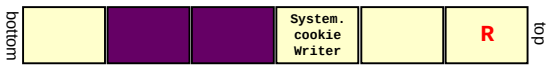


Runtime Stack

Maybe we want to disallow it

- the malicious applet may be trying to erase our disk
- it's calling system code to do that

Runtime Stack



Or, maybe we want to allow it
- it may just want to write a cookie
- it called **`System.cookieWriter`**
- **`System.cookieWriter`** knows it's ok

Runtime Stack

Maybe we want to allow it for another reason
- all running methods are trusted

Q) How do we tell the difference between these scenarios?

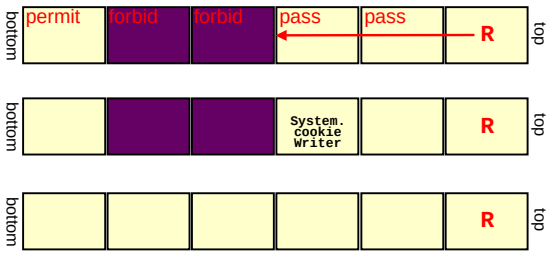
A) *Stack inspection!*

Stack Inspection

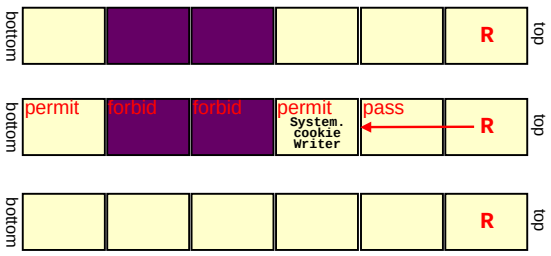
- An invocation of a trusted method, when calling another method, may either:
  - *permit* R on the stack above it
  - *forbid* R on the stack above it
  - *pass* permission from below (be transparent)
- An instantiation of an untrusted method must *forbid* R above it

Stack Inspection

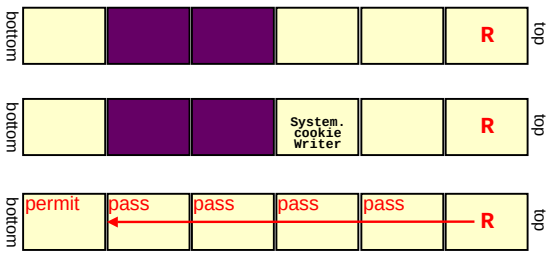
- When about to execute R, look down through the stack until we see either
  - a system method permitting R -- do it
  - a system method forbidding R -- don't do it
  - an untrusted method -- don't do it
- If we get all the way to the bottom, do it (IE, Sun JDK) or don't do it (Netscape)



Case A: R is not executed



Case B: R is executed



Case C: R is executed

Conclusion

Java and the Java Virtual Machine:
Lots of interesting ideas!