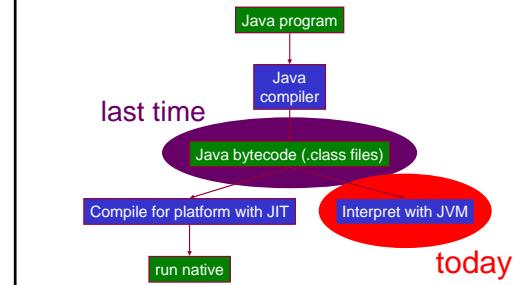


## Under the Hood: The Java Virtual Machine Part II



### A Whirlwind Tour

- Class loading and initialization
- Object initialization
- Method dispatch
- Exception handling
- Java security model
  - Bytecode verification
  - Stack inspection

### 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 Instructor {  
    static Instructor Dexter = new Instructor();  
    static Instructor Rich = new Instructor();  
    static Instructor Dave = new Instructor();  
    static Hashtable h = new Hashtable();  
    static {  
        h.put(Dexter,"Java");  
        h.put(Rich,"Data structures");  
        h.put(Dave,"GUI statics and dynamics");  
    }  
    ...  
}
```

Compiled to `Instructor.<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 = ~~2~~ 2      B.b = ~~1~~ 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  
1: invokespecial java.lang.Object.<init>()  
4: aload_0  
5: aload_1  
6: putfield A.name Ljava/lang/String;  
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

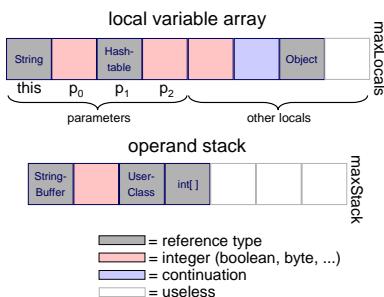
`invokevirtual foo ()V`

- 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, 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(max_stack = 2, max_locals = 2, code_length = 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

<code>startRange</code>	start of range handler is in effect
<code>endRange</code>	end of range handler is in effect
<code>handlerEntry</code>	entry point of exception handler
<code>catchType</code>	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: astore_1 null
1: astore_1
2: new java.lang.Object
3: astore_2
4: invokespecial java.lang.Object.<init> ()V
5: astore_2
10: ldc #2 [2]
11: checkcast java.lang.Integer
14: astore_2
15: ldc #3 [3]
16: invokevirtual java.lang.System.out Ljava/io/PrintStream;
17: aload_2
18: invokevirtual java.lang.Integer.intValue ()I
19: invokevirtual java.io.PrintStream.print I)V
20: getstatic java.lang.System.out Ljava/io/PrintStream;
21: ldc #4 [4]
22: invokevirtual java.io.PrintStream.println I)V
23: ldc #5 [5]
24: invokevirtual java.io.PrintStream.println Ljava/lang/String)V
25: ldc #6 [6]
26: invokevirtual java.io.PrintStream.println Ljava/lang/String)V
27: goto #89
36: astore_3
37: ldc #7 [7]
38: invokevirtual java.lang.System.out Ljava/io/PrintStream;
40: ldc #8 [8]
41: ldc #9 [9]
42: invokevirtual java.io.PrintStream.println Ljava/lang/String)V
43: ldc #10 [10]
44: invokevirtual java.lang.System.out Ljava/io/PrintStream;
45: ldc #11 [11]
46: ldc #12 [12]
47: invokevirtual java.io.PrintStream.println Ljava/lang/String)V
48: ldc #13 [13]
49: invokevirtual java.io.PrintStream.println Ljava/lang/String)V
50: ldc #14 [14]
51: ldc #15 [15]
52: invokevirtual java.io.PrintStream.println Ljava/lang/String)V
53: ldc #16 [16]
54: ldc #17 [17]
55: invokevirtual java.io.PrintStream.println Ljava/lang/String)V
56: ldc #18 [18]
57: ldc #19 [19]
58: invokevirtual java.io.PrintStream.println Ljava/lang/String)V
59: ldc #20 [20]
60: ldc #21 [21]
61: invokevirtual java.io.PrintStream.println Ljava/lang/String)V
62: ldc #22 [22]
63: ldc #23 [23]
64: invokevirtual java.io.PrintStream.println Ljava/lang/String)V
65: ldc #24 [24]
66: ldc #25 [25]
67: invokevirtual java.io.PrintStream.println Ljava/lang/String)V
68: ldc #26 [26]
69: ldc #27 [27]
70: invokevirtual java.io.PrintStream.println Ljava/lang/String)V
71: ldc #28 [28]
72: ldc #29 [29]
73: invokevirtual java.io.PrintStream.println Ljava/lang/String)V
74: ldc #30 [30]
75: ldc #31 [31]
76: invokevirtual java.io.PrintStream.println Ljava/lang/String)V
77: ldc #32 [32]
78: ldc #33 [33]
79: invokevirtual java.io.PrintStream.println Ljava/lang/String)V
80: ldc #34 [34]
81: ldc #35 [35]
82: ldc #36 [36]
83: ldc #37 [37]
84: ldc #38 [38]
85: ldc #39 [39]
86: ldc #40 [40]
87: ldc #41 [41]
88: ldc #42 [42]
89: return

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

```

```

0: astore_0
1: astore_1
2: new java.lang.Object
3: dup
4: invokespecial java.lang.Object.<init> ()V
9: astore_2
10: aload_0
11: aload_1
12: invokestatic java.lang.Integer
14: astore_3
15: invokevirtual java.io.PrintStream.println (Ljava/io/PrintStream;)V
16: astore_4
17: astore_5
18: astore_6
19: invokevirtual java.lang.Integer.intValue ()I
20: getstatic java.lang.System.out Ljava/io/PrintStream;
21: ldc ("finally!");
22: invokevirtual java.lang.System.out Ljava/io/PrintStream;
23: goto #39
26: astore_7
27: new java.lang.StringBuilder
28: dup
29: invokespecial java.lang.StringBuilder. ()V
30: ldc (" was not an Integer")
31: invokevirtual java.lang.StringBuilder.append (Ljava/lang/String;)Ljava/lang/StringBuilder;
32: invokevirtual java.lang.StringBuilder.toString ()Ljava/lang/String;)
33: goto #39
36: astore_8
37: new java.lang.StringBuilder
38: dup
39: ldc (" was not an Integer")
40: invokevirtual java.lang.StringBuilder.append (Ljava/lang/String;)Ljava/lang/StringBuilder;
41: ldc (" finally!")
42: invokevirtual java.io.PrintStream.print (Ljava/io/PrintStream;)V
43: astore_9
44: astore_10
45: astore_11
46: astore_12
47: astore_13
48: astore_14
49: astore_15
50: astore_16
51: astore_17
52: astore_18
53: astore_19
54: astore_20
55: astore_21
56: astore_22
57: astore_23
58: astore_24
59: astore_25
60: astore_26
61: ldc ("finally!")
62: invokevirtual java.io.PrintStream.println (Ljava/lang/String;)V
63: astore_27
64: astore_28
65: astore_29
66: astore_30
67: astore_31
68: astore_32
69: astore_33
70: astore_34
71: astore_35
72: astore_36
73: goto #89
75: astore_37
76: astore_38
77: astore_39
78: astore_40
79: astore_41
80: astore_42
81: ldc ("finally!")
82: invokevirtual java.io.PrintStream.println (Ljava/lang/String;)V
83: astore_43
84: astore_44
85: astore_45
86: astore_46
87: astore_47
88: astore_48
89: astore_49
90: return

```

```

0: account=null
1: astore_1
2: new java.lang.Object
3: dup
4: invokespecial java.lang.Object.<init>()V
9: astore_2
10: aload_2
11: instanceof java.lang.Integer
14: astore_1
15: getstatic java.lang.System.outLjava/io/PrintStream;
16: invokevirtual java.lang.System.outLjava/io/PrintStream;
17: ldc "finally"
18: astore_2
19: invokevirtual java.lang.Integer.intValue()I
20: invokevirtual java.io.PrintStream.println(I)V
21: getstatic java.lang.System.outLjava/io/PrintStream;
22: ldc "finally"
23: invokevirtual java.io.PrintStream.println();
24: astore_2
25: ldc "#$"
26: astore_1
27: getstatic java.lang.System.outLjava/io/PrintStream;
28: ldc "was not an Integer"
29: astore_2
30: invokevirtual java.io.PrintStream.println();
31: getstatic java.lang.System.outLjava/io/PrintStream;
32: ldc "#$"
33: astore_1
34: ldc "#$"
35: getstatic java.lang.System.outLjava/io/PrintStream;
36: ldc "by null"
37: astore_2
38: invokevirtual java.io.PrintStream.println();
39: ldc "#$"
40: astore_1
41: ldc "finally"
42: astore_2
43: invokevirtual java.io.PrintStream.println();
44: astore_1
45: ldc "#$"
46: astore_2
47: ldc "finally"
48: astore_1
49: ldc "#$"
50: astore_2
51: ldc "#$"
52: getstatic java.lang.System.outLjava/io/PrintStream;
53: ldc "#$"
54: astore_1
55: ldc "#$"
56: ldc "#$"
57: getstatic java.lang.System.outLjava/io/PrintStream;
58: ldc " by null"
59: astore_2
60: ldc "#$"
61: ldc "finally"
62: astore_1
63: ldc "#$"
64: astore_2
65: ldc "#$"
66: astore_1
67: ldc "#$"
68: astore_2
69: astore_1
70: return

```

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

From To	Handler	Type
0	25	36
1	25	36
2	new	java.lang.Object
5	dup	
6	ldc_w	java.lang.Object.<init> ()V
9	astore_1	
10	aload_1	
11	astore_1	java.lang.Integer
14	astore_1	
15	getstatic	java.lang.System.out Ljava/io/PrintStream;
16	invokevirtual	java.io.PrintStream.println (I)V
22	getstatic	java.lang.System.out Ljava/io/PrintStream;
25	invokevirtual	java.io.PrintStream.println (Ljava/lang/String;)V
28	ldc	"finally!"
30	invokevirtual	java.io.PrintStream.println (Ljava/lang/String;)V
32	ret	
36	astore_3	
37	getstatic	java.lang.System.out Ljava/io/PrintStream;
40	ldc	"y was not an Integer."
42	invokevirtual	java.io.PrintStream.println (Ljava/lang/String;)V
46	getstatic	java.lang.System.out Ljava/io/PrintStream;
48	ldc	"y = " + y
50	invokevirtual	java.io.PrintStream.println (Ljava/lang/String;)V
53	goto	#89
54	astore_3	
57	getstatic	java.lang.System.out Ljava/io/PrintStream;
60	ldc	"y was null"
62	invokevirtual	java.io.PrintStream.println (Ljava/lang/String;)V
65	getstatic	java.lang.System.out Ljava/io/PrintStream;
68	ldc	"finally!"
70	invokevirtual	java.io.PrintStream.println (Ljava/lang/String;)V
73	goto	#89
74	astore_3	
75	getstatic	java.lang.System.out Ljava/io/PrintStream;
78	ldc	"finally!"
81	invokevirtual	java.io.PrintStream.println (Ljava/lang/String;)V
84	iconst_4	
88	abrc	
89	return	

```

0: aconst null
1: astore_1
2: new java.lang.Object
3: dup
4: invokespecial java.lang.Object.<init> ()V
9: astore_2
10: aload_2
11: invokecast java.lang.Integer
14: astore_1
15: getstatic java.lang.System.out Ljava/io/PrintStream;
18: ldc #25
19: invokevirtual java.io.PrintStream.println(I)V
20: ldc #26
21: invokevirtual java.io.PrintStream.println(I)V
22: getstatic java.lang.System.out Ljava/io/PrintStream;
23: ldc #27
24: invokevirtual java.io.PrintStream.println(I)V
25: ldc #28
26: invokevirtual java.io.PrintStream.println(I)V
27: ldc #29
28: astore_3
29: aload_3
30: ldc #30
31: invokevirtual java.lang.System.out Ljava/io/PrintStream;
32: ldc #31
33: ldc #32
34: ldc #33
35: ldc #34
36: astore_3
37: getstatic java.lang.System.out Ljava/io/PrintStream;
38: ldc #35
39: ldc #36
40: ldc #37
41: ldc #38
42: invokevirtual java.io.PrintStream.println(I)V
43: getstatic java.lang.System.out Ljava/io/PrintStream;
44: ldc #39
45: ldc #40
46: ldc #41
47: ldc #42
48: astore_3
49: ldc #43
50: invokevirtual java.io.PrintStream.println(I)V
51: ldc #44
52: ldc #45
53: ldc #46
54: ldc #47
55: ldc #48
56: ldc #49
57: ldc #50
58: ldc #51
59: ldc #52
60: ldc #53
61: ldc #54
62: ldc #55
63: ldc #56
64: ldc #57
65: ldc #58
66: ldc #59
67: ldc #60
68: ldc #61
69: ldc #62
70: ldc #63
71: ldc #64
72: ldc #65
73: goto #89
74: astore_4
75: ldc #66
76: astore_5
77: ldc #67
78: astore_6
79: ldc #68
80: astore_7
81: ldc #69
82: ldc #70
83: ldc #71
84: ldc #72
85: ldc #73
86: ldc #74
87: ldc #75
88: ldc #76
89: astore
90: return

```

```

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

```

```

0: aconst null
1: astore_1
2: new java.lang.Object
3: dup
4: invokespecial java.lang.Object.<init> ()V
9: astore_2
10: aload_2
11: invokecast java.lang.Integer
14: astore_1
15: getstatic java.lang.System.out Ljava/io/PrintStream;
18: ldc #25
19: invokevirtual java.io.PrintStream.println(I)V
20: ldc #26
21: invokevirtual java.io.PrintStream.println(I)V
22: getstatic java.lang.System.out Ljava/io/PrintStream;
23: ldc #27
24: invokevirtual java.io.PrintStream.println(I)V
25: ldc #28
26: invokevirtual java.io.PrintStream.println(I)V
27: ldc #29
28: astore_3
29: aload_3
30: ldc #30
31: invokevirtual java.lang.System.out Ljava/io/PrintStream;
32: ldc #31
33: ldc #32
34: ldc #33
35: ldc #34
36: astore_3
37: getstatic java.lang.System.out Ljava/io/PrintStream;
38: ldc #35
39: ldc #36
40: ldc #37
41: ldc #38
42: invokevirtual java.io.PrintStream.println(I)V
43: getstatic java.lang.System.out Ljava/io/PrintStream;
44: ldc #39
45: ldc #40
46: ldc #41
47: ldc #42
48: astore_3
49: ldc #43
50: invokevirtual java.io.PrintStream.println(I)V
51: ldc #44
52: ldc #45
53: ldc #46
54: ldc #47
55: ldc #48
56: ldc #49
57: ldc #50
58: ldc #51
59: ldc #52
60: ldc #53
61: ldc #54
62: ldc #55
63: ldc #56
64: ldc #57
65: ldc #58
66: ldc #59
67: ldc #60
68: ldc #61
69: ldc #62
70: ldc #63
71: ldc #64
72: ldc #65
73: goto #89
74: astore_4
75: ldc #66
76: astore_5
77: ldc #67
78: astore_6
79: ldc #68
80: astore_7
81: ldc #69
82: ldc #70
83: ldc #71
84: ldc #72
85: ldc #73
86: ldc #74
87: ldc #75
88: ldc #76
89: return

```

```

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

```

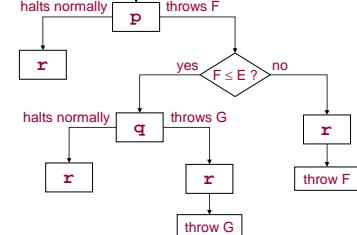
## Try/Catch/Finally

```
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 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

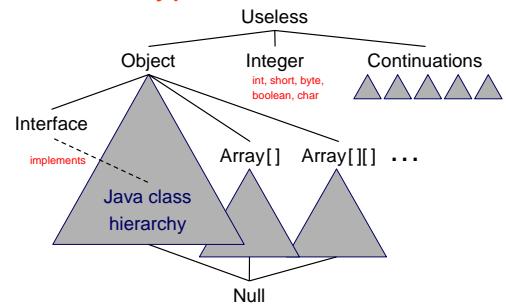
## Bytecode Verification

- 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

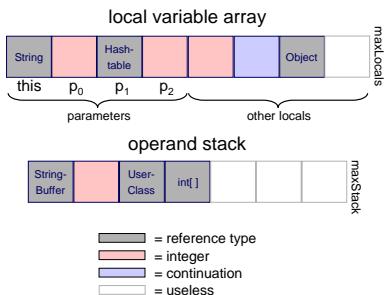
## Bytecode Verification

- 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

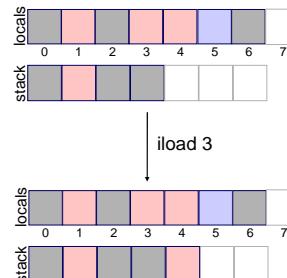
## Types in the JVM



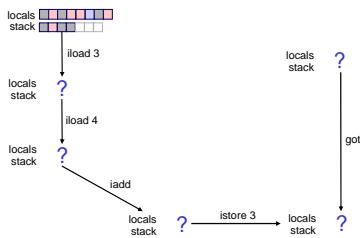
## Typing of Java Bytecode



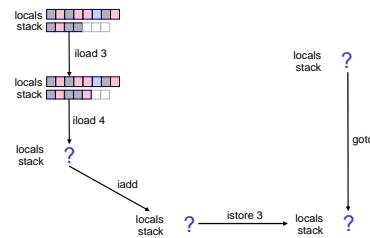
## Example



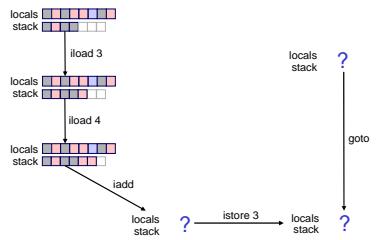
## Example



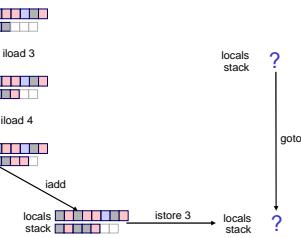
## Example



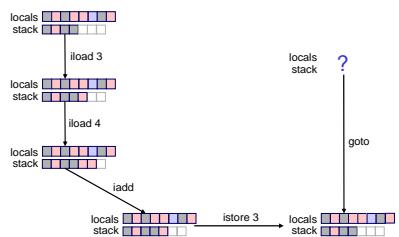
## Example



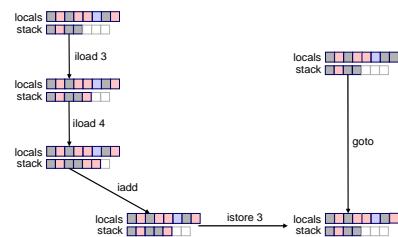
## Example



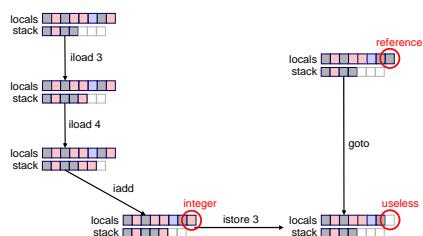
## Example



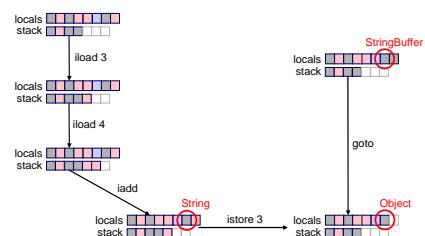
## Example

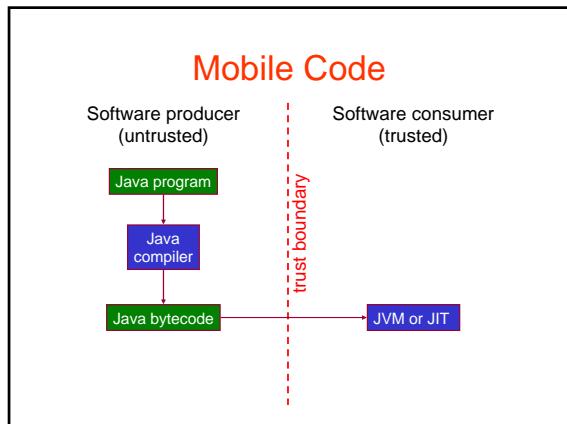


## Example



## Example





## 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

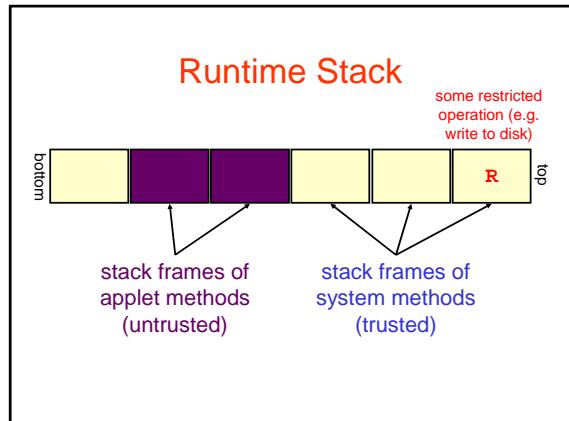
## Mobile Code

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

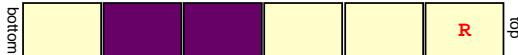
## Mobile Code

**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



Maybe we want to disallow it

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

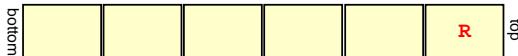
## Runtime Stack



Or, maybe we want to allow it

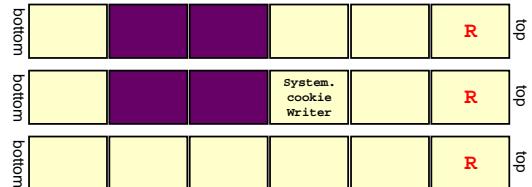
- it may just want to write a cookie
- it called `System.cookieWriter`
- `System.cookieWriter` knows it's ok

## 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

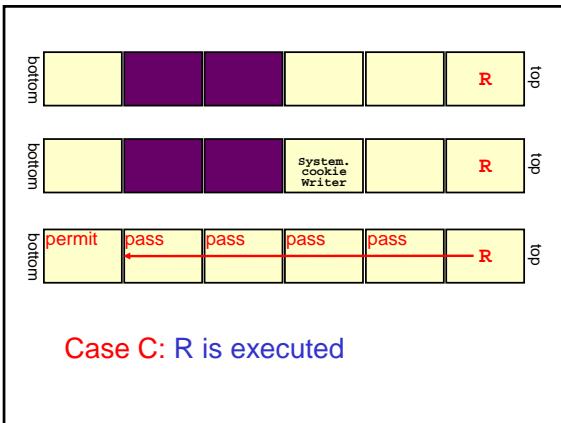
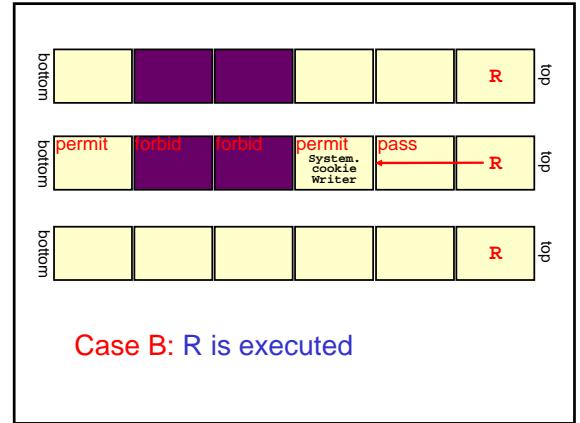
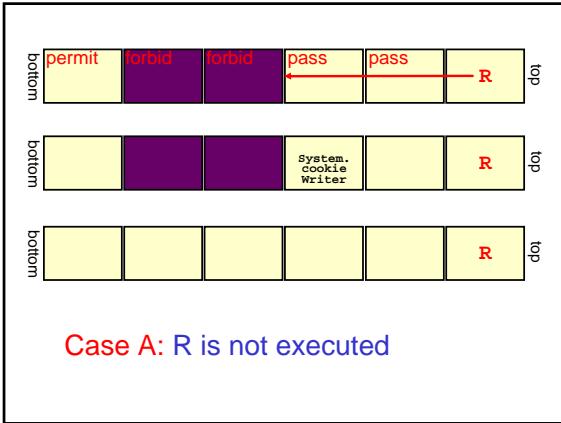


- An invocation of a trusted method, when calling another method, may either:
  - *permit* R on the stack above it
  - *forbid* R on the stack above it
  - *pass* permission from below (be transparent)
- An instantiation of an untrusted method must *forbid* R above it

## 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)



## Conclusion

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