CS/ENGRD 2110
Object-Oriented Programming and Data Structures

Spring 2011
Thorsten Joachims

Lecture 24:
Java Virtual Machine

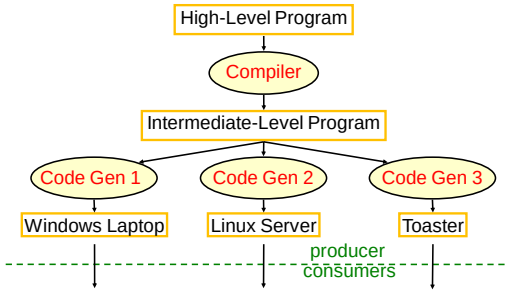
Compiling for Different Platforms

- 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, FreeBSD, Aix, ...
  - VM/CMS, OS/2, Solaris, Mac OS X, ...
- Processor Architectures
  - Pentium, PowerPC, Alpha, SPARC, MIPS, ...

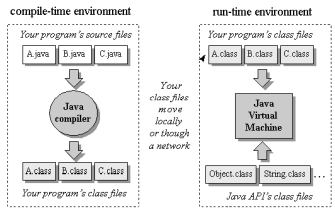
Compiling for Different Platforms



Dream: Platform Independence

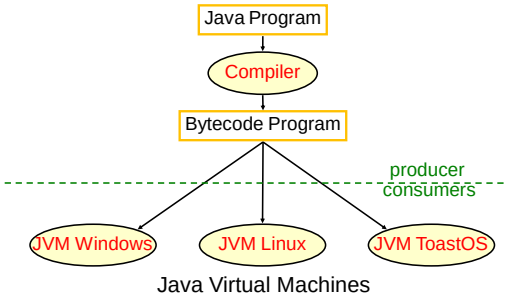
- Compiler produces one low-level program for all platforms (Bytecode)
  - Low-level compiled form of Java
  - Platform-independent
  - Compact
    - Suitable for mobile code, applets
  - Easy to interpret
    - Java virtual machine (JVM) in your browser
    - Simple stack-based semantics
    - Support for objects
- Executed on a virtual machine (VM)
- A different VM implementation needed for each platform, but installed once and for all

Java Platform



- Java Platform
  - the Java programming language
  - the Java Application Programming Interface
  - the Java class file format
  - the Java virtual machine

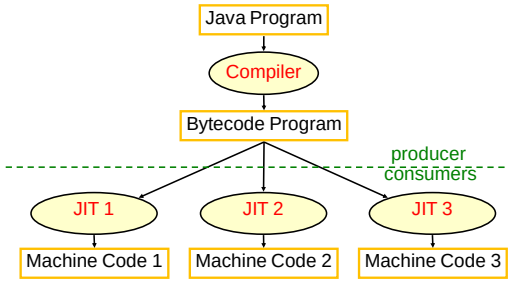
Platform Independence with Java



Java Virtual Machines

7

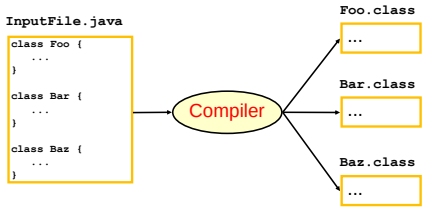
Platform Independence with Java



Just-in-Time Compilers

8

Class Files



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

Structure of Class File

```
ClassFile {
      u4 magic;            // 0xCAFEBABE ids class file
      u2 minor_version;    // Check if JVM can execute
      u2 major_version;    // this class file?
      u2 constant_pool_count;
      cp_info constant_pool[constant_pool_count-1];
      u2 access_flags;     // public, final, abstract
      u2 this_class;
      u2 super_class;
      u2 interfaces_count; // number of implemented
                           // interfaces
      u2 interfaces[interfaces_count];
      u2 fields_count;
      field_info fields[fields_count];
      u2 methods_count;
      method_info methods[methods_count];
      u2 attributes_count;
      attribute_info attributes[attributes_count];
 }
```

11

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

12

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

13

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

14

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

15

Example

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

Q) How many entries in the constant pool?

A) 33

16

```
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")
```

17

Field Table

count	2 bytes	length of table
Field Table 1	variable in size	See next slide
Field Table 2	variable in size	See next slide
…	…	…

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

18

Field Table Entry

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

19

Method Table

count	2 bytes	length of table
Method Table 1	variable	See next slide
Method Table 2	variable	See next slide
...	...	...

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

20

Method Table Entry

access flags	2 bytes	e.g. public, static, ...
name index	2 bytes	index into CP
descriptor index	2 bytes	index into CP (arguments)
attributes count	2 bytes	number of attributes
code attribute	variable	See next slide
attribute 2	variable	...
...	...	...

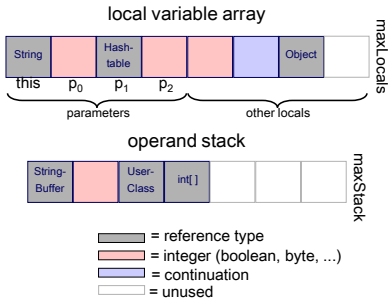
21

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	See later slide
attributesCount	2 bytes	number of attributes
attributes	variable	e.g. LineNumberTable

22

Stack Frame of a Method



23

Example Bytecode

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

```
5:   iload 1    //load b
6:   ifeq 16    //if 0, 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

24

Examples

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

Q) How many methods?

A) 2

25

```
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
```

26

Exception Table Entry

start	2 bytes	start of try/catch range handler is in effect for
end	2 bytes	end of try/catch range handler is in effect for
entry	2 bytes	Start point for exception handler code
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

27

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

28

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

29

Class Initialization

```
class Staff {
  static Staff Thorsten = new Staff();
  static Staff Robert = new Staff();
  static Staff Nikos = new Staff();
  static Map<Staff,String> h =
     new HashMap<Staff,String>();
  static {
    h.put(Thorsten, "INSTRUCTOR");
    h.put(Robert, "TA");
    h.put(Nikos, "TA");
  }
  ...
}
```

Compiled to **`Staff.<clinit>`**

30

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

31

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

32

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

33

Object Initialization

- Object creation initiated by **`new`** (sometimes implicitly, e.g. by + for strings)
- 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

34

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

35

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
```

36

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

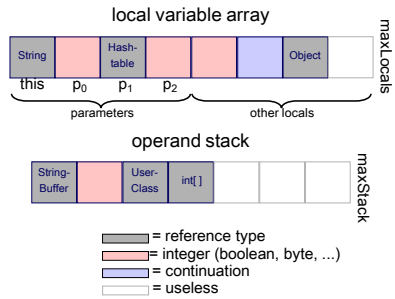
38

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

40

Stack Frame of a Method



41

Instance Method Dispatch

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

```
Code(maxStack = 2, maxLocals = 2, codeLength = 12)
0: ldc "Hello world" // load string
2: astore 1          // store in x
3: aload 0           // load this (used by putfield)
4: aload 1           // load x
5: invokevirtual java.lang.String.getBytes ()[B
8: putfield A.data [B // store result in A.data
11: return
```

42

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

43

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

44

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

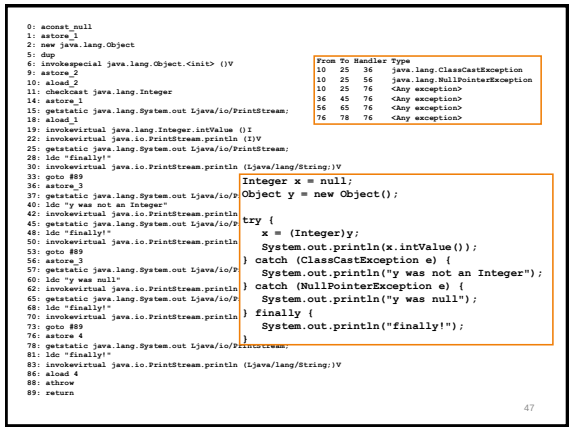
45

Example

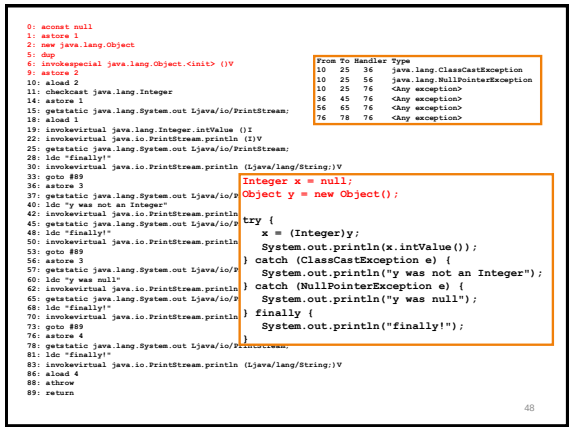
```
Integer x = null;
Object y = new Object();

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!");
}
```

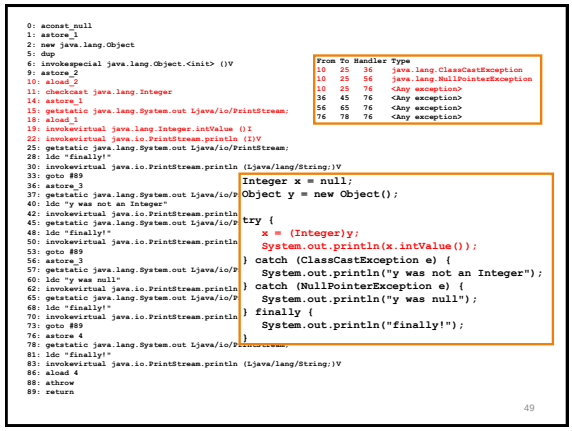
46



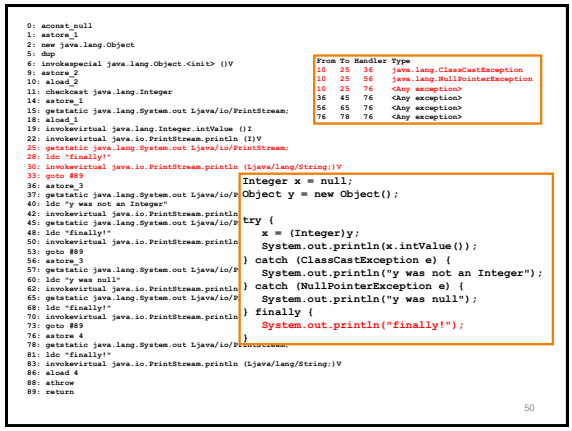
47



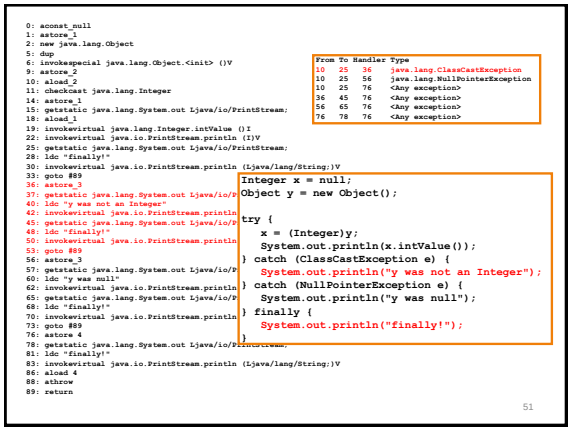
48

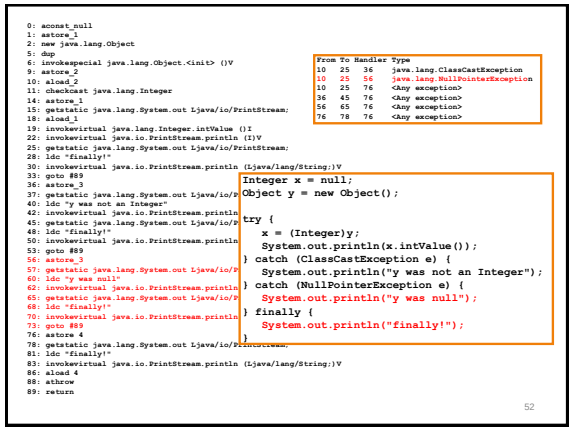


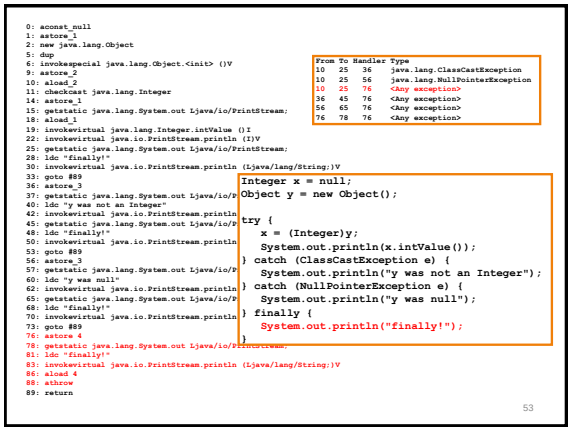
49

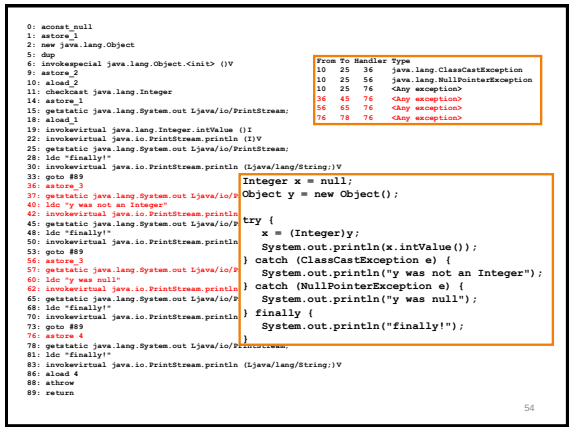


50







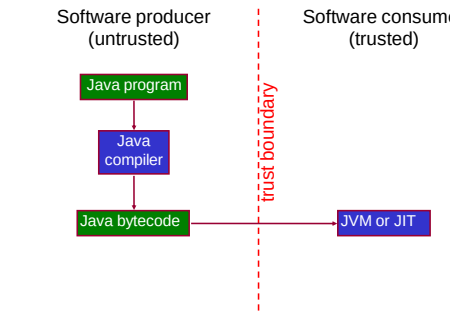


Java Security Model

- 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

57

Mobile Code



71

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?

72

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

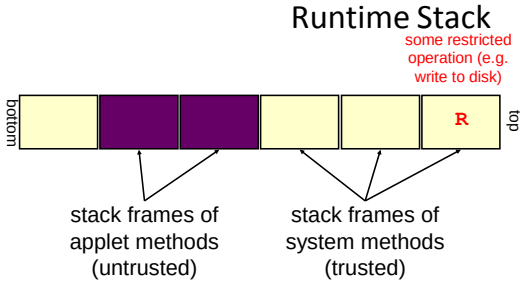
73

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

74

Runtime Stack



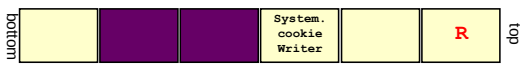
75

Runtime Stack

- Maybe we want to disallow it
  - the malicious applet may be trying to erase our disk
  - it's calling (trusted) system code to do that

76

Runtime Stack



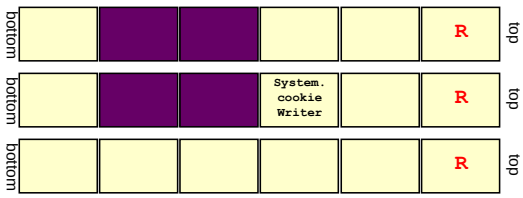
- Or, maybe we want to allow it
  - it may just want to write a cookie
  - it called System.cookieWriter
  - System.cookieWriter knows it's ok

77

Runtime Stack

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

78



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

A) *Stack inspection!*

79

Stack Inspection

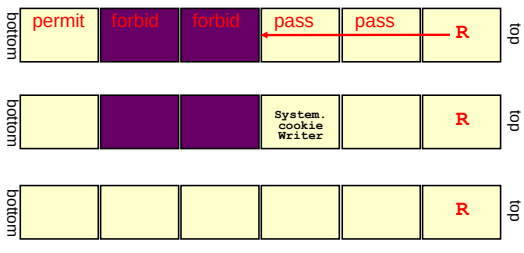
- An invocation of a trusted method, when calling another method, may either:
  - permit R on the stack above it
  - forbid R on the stack above it
  - pass permission from below (be transparent)
- An instantiation of an untrusted method must forbid R above it

80

Stack Inspection

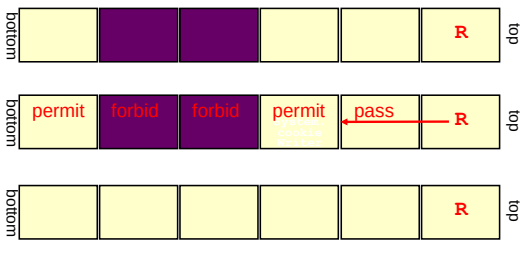
- When about to execute R, look down through the stack until we see either
  - a system method permitting R -- do it
  - a system method forbidding R -- don't do it
  - an untrusted method -- don't do it
- If we get all the way to the bottom, do it (IE, Sun JDK) or don't do it (Netscape)

81



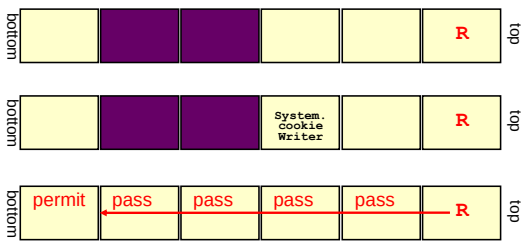
Case A: R is not executed

82



Case B: R is executed

83



Case C: R is executed

84

Conclusion

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

85