Today

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

Instance Method Dispatch

\texttt{x.foo(...)}

- \texttt{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 \texttt{x.foo(...)}, pushes them on the stack
- Object \texttt{x} is always the first argument

Instance Method Dispatch

\texttt{invokevirtual foo (...)V}

- Name and type of \texttt{foo(...)} are arguments to \texttt{invokevirtual} (indices into constant pool)
- JVM retrieves them from constant pool
- Gets the dynamic (runtime) type of \texttt{x}
- Follows parent pointers until finds \texttt{foo(...)}\texttt{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 \texttt{invokevirtual} instruction
Stack Frame of a Method

Instance Method Dispatch

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 Table Entry

• 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
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
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

- Object
- Integer
- Continuations
- Array
- Array

Java class hierarchy

- Interface
- implements

Types in the JVM

- Useless
- Null

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
Mobile Code

Software producer
(trusted)

Software consumer
(untrusted)

Java program
Java compiler
Java bytecode
JVM or JIT

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?

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

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

stack frames of applet methods (untrusted)

stack frames of system methods (trusted)

Maybe we want to disallow it
  – the malicious applet may be trying to erase our disk
  – it’s calling system code to do that
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

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!

• 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

• 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!