

CS560

LECTURE 10
March 10, 2005

Slides: From John Mitchell, Concepts in Programming Languages, 2003.

Announcements

- ◆ Term project is due.
 - Late days
- ◆ Homework 4
 - Key sent out tomorrow
- ◆ Final Exam Review

Class Tonight: Java Overview

- ◆ Language Overview
 - History and design goals
- ◆ Classes and Inheritance
 - Object features
 - Encapsulation
 - Inheritance
- ◆ Types and Subtyping
 - Primitive and ref types
 - Interfaces; arrays
 - Exception hierarchy
- ◆ Java virtual machine
 - Loader and initialization
 - Linker and verifier
 - Bytecode interpreter
- ◆ Method lookup
 - four different bytecodes

History

- ◆ James Gosling and others at Sun, 1990 - 95
- ◆ Oak language for "set-top box"
 - small networked device with television display
 - graphics
 - execution of simple programs
 - communication between local program and remote site
 - no "expert programmer" to deal with crash, etc.
- ◆ Internet application
 - simple language for writing programs that can be transmitted over network

Gates Saw Java as Real Threat

Publicly, Microsoft chief Bill Gates was nearly dismissive when he talked in 1996 about Sun Microsystems' Java programming language. But in internal company discussions, he wrote to staff members that Java and the threat the cross-platform technology posed to his company's Windows operating systems "scares the hell out of me."

[Wired News Report](#)
8:09 a.m. 22.Oct.98.PDT
(material from '98 trial)

Design Goals

- ◆ Portability
 - Internet-wide distribution: PC, Unix, Mac
- ◆ Reliability
 - Avoid program crashes and error messages
- ◆ Safety
 - Programmer may be malicious
- ◆ Simplicity and familiarity
 - Appeal to average programmer; less complex than C++
- ◆ Efficiency
 - Important but secondary

General design decisions

- ◆ Almost everything is an object
- ◆ All objects on heap, accessed through pointers
- ◆ No functions, no multiple inheritance, no go to, no operator overloading, no automatic coercions
- ◆ Bytecode interpreter on many platforms
- ◆ Run-time type and bounds checks
- ◆ Garbage collection
- ◆ Type safe
- ◆ Concurrency support

Java System

- ◆ The Java programming language
- ◆ Compiler and run-time system
 - Programmer compiles code
 - Compiled code transmitted on network
 - Receiver executes on interpreter (JVM)
 - Safety checks made before/during execution
- ◆ Library, including graphics, security, etc.
 - Large library made it easier for projects to adopt Java
 - Interoperability
 - Provision for "native" methods

Java Classes and Objects

- ◆ Syntax similar to C++
- ◆ Object
 - has fields and methods
 - is allocated on heap, not run-time stack
 - accessible through reference (only ptr assignment)
 - garbage collected
- ◆ Dynamic lookup
 - Similar in behavior to other languages
 - Static typing => more efficient than Smalltalk
 - Dynamic linking, interfaces => slower than C++

Point Class

```
class Point {  
    private int x;  
    protected void setX (int y) {x = y;}  
    public int getX() {return x;}  
    Point(int xval) {x = xval;}    // constructor  
};
```

- Visibility similar to C++, but not exactly

Object initialization

- ◆ Java guarantees constructor call for each object
 - Memory allocated
 - Constructor called to initialize memory
 - Some interesting issues related to inheritance
 - We'll discuss later ...
- ◆ Static fields of class initialized at class load time
 - Talk about class loading later

Garbage Collection and Finalize

- ◆ Objects are garbage collected
 - No explicit *free*
 - Avoid dangling pointers, resulting type errors
- ◆ Problem
 - What if object has opened file or holds lock?
- ◆ Solution
 - *finalize* method, called by the garbage collector
 - Before space is reclaimed, or when virtual machine exits
 - Space overflow is not really the right condition to trigger finalization when an object holds a lock...
 - Important convention: call `super.finalize`

Inheritance

- ◆ Similar to Smalltalk, C++
- ◆ Subclass inherits from superclass
 - Single inheritance only (but see interfaces)
- ◆ Some additional features
 - Conventions regarding *super* in constructor and *finalize* methods
 - Final classes and methods

Example subclass

```
class ColorPoint extends Point {  
    // Additional fields and methods  
    private Color c;  
    protected void setC (Color d) {c = d;}  
    public Color getC() {return c;}  
    // Define constructor  
    ColorPoint(int xval, Color cval) {  
        super(xval); // call Point constructor  
        c = cval; } // initialize ColorPoint field  
};
```

Constructors and Super

- ◆ Java guarantees constructor call for each object
- ◆ This must be preserved by inheritance
 - Subclass constructor must call super constructor
 - If first statement is not call to super, then call super() inserted automatically by compiler
 - If superclass does not have a constructor with no args, then this causes compiler error (yuck)
 - Exception to rule: if one constructor invokes another, then it is responsibility of second constructor to call super, e.g.,

is compiled without inserting call to super
- ◆ Different conventions for finalize and super
 - Compiler does not force call to super finalize

Final classes and methods

- ◆ Restrict inheritance
 - Final classes and methods cannot be redefined
- ◆ Example
 - java.lang.String
- ◆ Reasons for this feature
 - Important for security
 - Programmer controls behavior of all subclasses
 - Critical because subclasses produce subtypes
 - Compare to C++ virtual/non-virtual
 - Method is "virtual" until it becomes final



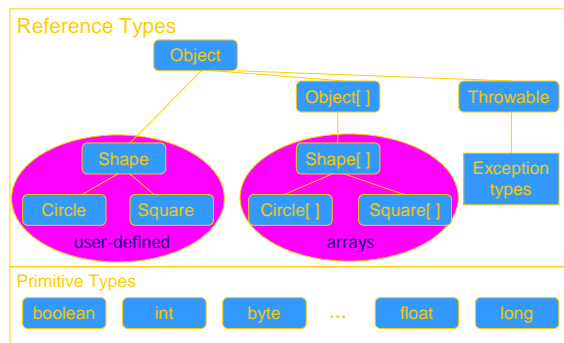
Class *Object*

- ◆ Every class extends another class
 - Superclass is *Object* if no other class named
- ◆ Methods of class *Object*
 - `getClass` – return the Class object representing class of the object
 - `toString` – returns string representation of object
 - `equals` – default object equality (not ptr equality)
 - `hashCode`
 - `Clone` – makes a duplicate of an object
 - `wait`, `notify`, `notifyAll` – used with concurrency
 - `finalize`

Java Types and Subtyping

- ◆ Primitive types – *not* objects
 - Integers, Booleans, etc
- ◆ Reference types
 - Classes, interfaces, arrays
 - No syntax distinguishing `Object *` from `Object`
- ◆ Type conversion
 - If `A <: B`, and `B x`, then can cast `x` to `A`
 - Casts checked at run-time, may raise exception

Classification of Java types



Class and Interface Subtyping

- ◆ Class subtyping similar to C++
 - Statically typed language
 - Subclass produces subtype
 - Single inheritance => subclasses form tree
- ◆ Interfaces
 - Completely abstract classes
 - no implementation
 - Java also has abstract classes (without full impl)
 - Multiple subtyping
 - Interface can have multiple subtypes

Example

```
interface Shape {
    public float center();
    public void rotate(float degrees);
}
interface Drawable {
    public void setColor(Color c);
    public void draw();
}
class Circle implements Shape, Drawable {
    // does not inherit any implementation
    // but must define Shape, Drawable methods
}
```

Properties of interfaces

◆ Flexibility

- Allows subtype graph instead of tree
- Avoids problems with multiple inheritance of implementations (remember C++ "diamond")

◆ Cost

- Offset in method lookup table not known at compile
- Different bytecodes for method lookup
 - one when class is known
 - one when only interface is known
 - search for location of method
 - cache for use next time this call is made (from this line)

Java class subtyping

◆ Signature Conformance

- Subclass method signatures must conform to those of superclass

◆ Three ways signature could vary

- Argument types
- Return type
- Exceptions

How much conformance is needed in principle?

◆ Java rule

- Arguments and returns must have identical types, may remove exceptions

Java Exceptions

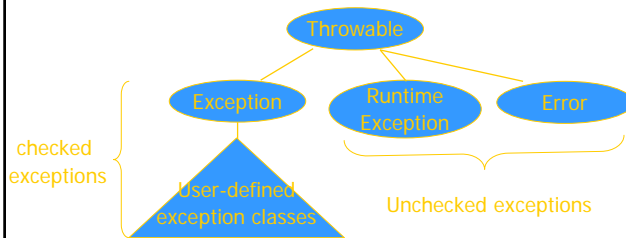
◆ Similar basic functionality to ML, C++

- Constructs to *throw* and *catch* exceptions
- Dynamic scoping of handler

◆ Some differences

- An exception is an object from an exception class
- Subtyping between exception classes
 - Use subtyping to match type of exception or pass it on ...
 - Similar functionality to ML pattern matching in handler
- Type of method includes exceptions it can throw
 - Actually, only subclasses of Exception (see next slide)

Exception Classes



- ◆ If a method may throw a checked exception, then this must be in the type of the method

Try/finally blocks

- ◆ Exceptions are caught in try blocks

```
try {
    statements
} catch (ex-type1 identifier1) {
    statements
} catch (ex-type2 identifier2) {
    statements
} finally {
    statements
}
```

- ◆ Implementation: finally compiled to jsr

Why define new exception types?

- ◆ Exception may contain data
 - Class `Throwable` includes a string field so that cause of exception can be described
 - Pass other data by declaring additional fields or methods
- ◆ Subtype hierarchy used to catch exceptions

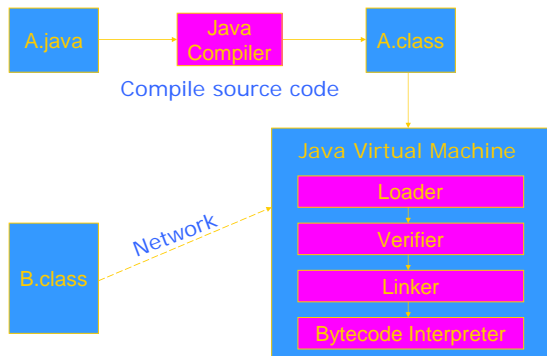
```
catch <exception-type> <identifier> { ... }
```

will catch any exception from any subtype of exception-type and bind object to identifier

Java Implementation

- ◆ Compiler and Virtual Machine
 - Compiler produces bytecode
 - Virtual machine loads classes on demand, verifies bytecode properties, interprets bytecode
- ◆ Why this design?
 - Bytecode interpreter/compilers used before
 - Pascal “pcode”; Smalltalk compilers use bytecode
 - Minimize machine-dependent part of implementation
 - Do optimization on bytecode when possible
 - Keep bytecode interpreter simple
 - For Java, this gives portability
 - Transmit bytecode across network

Java Virtual Machine Architecture



Class loader

- ◆ Runtime system loads classes as needed
 - When class is referenced, loader searches for file of compiled bytecode instructions
- ◆ Default loading mechanism can be replaced
 - Define alternate ClassLoader object
 - Extend the abstract ClassLoader class and implementation
 - ClassLoader does not implement abstract method loadClass, but has methods that can be used to implement loadClass
 - Can obtain bytecodes from alternate source
 - VM restricts applet communication to site that supplied applet

JVM Linker and Verifier

- ◆ Linker
 - Adds compiled class or interface to runtime system
 - Creates static fields and initializes them
 - Resolves names
 - Checks symbolic names and replaces with direct references
- ◆ Verifier
 - Check bytecode for class or interface before loaded
 - Throw VerifyError exception if error occurs

Example issue in class loading and linking:

Static members and initialization

```
class ... {  
    /* static variable with initial value */  
    static int x = initial_value  
    /* ---- static initialization block ---- */  
    static { /* code executed once, when loaded */ }  
}
```

- ◆ Initialization is important
 - Cannot initialize class fields until loaded
- ◆ Static block cannot raise an exception
 - Handler may not be installed at class loading time

Verifier

- ◆ Bytecode may not come from standard compiler
 - Evil hacker may write dangerous bytecode
- ◆ Verifier checks correctness of bytecode
 - Every instruction must have a valid operation code
 - Every branch instruction must branch to the start of some other instruction, not middle of instruction
 - Every method must have a structurally correct signature
 - Every instruction obeys the Java type discipline

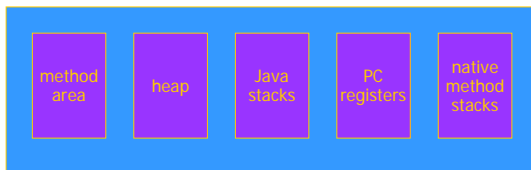
Last condition is fairly complicated

Bytecode interpreter

- ◆ Standard virtual machine interprets instructions
 - Perform run-time checks such as array bounds
 - Possible to compile bytecode class file to native code
- ◆ Java programs can call native methods
 - Typically functions written in C
- ◆ Multiple bytecodes for method lookup
 - invokevirtual - when class of object known
 - invokeinterface - when interface of object known
 - invokestatic - static methods
 - invokespecial - some special cases

JVM memory areas

- ◆ Java program has one or more threads
- ◆ Each thread has its own stack
- ◆ All threads share same heap



JVM uses stack machine

- ◆ Java


```
Class A extends Object {
    int i
    void f(int val) { i = val + 1; }
}
```
- ◆ Bytecode


```
Method void f(int)
    aload 0 ; object ref this
    iload 1 ; int val
    iconst 1
    iadd ; add val + 1
    putfield #4 <Field int i>
    return
```

↑
refers to const pool

JVM Activation Record

The diagram shows a vertical stack of memory cells. The top three cells are grouped by a bracket and labeled 'local variables'. The next three cells are grouped by a bracket and labeled 'operand stack'. The bottom cell is a blue box labeled 'data area' with a note: 'Return addr, exception info, Const pool res.'

Example

```
class ConstantPool {  
  
    public void Souter() {  
        int x = 0;  
        System.out.println("METHOD SOUTER");  
        x = Amie();  
    }  
  
    public int Amie() { return 10*10; }  
}
```

```
public void Souter();
```

Code:

```
Stack=2, Locals=2, Args_size=1  
0:  iconst_0  
1:  istore_1  
2:  getstatic   #2; //Field  
5:  ldc       #3;      //String METHOD SOUTER  
7:  invokevirtual #4; //Method  
   java/io/PrintStream.println:(Ljava/lang/String;)V  
10: aload_0  
11: invokevirtual #5; //Method Amie:()I  
14: istore_1  
15: return
```

Constant Pool Table

```
Compiled from "ConstantPool.java"  
class ConstantPool extends java.lang.Object  
  SourceFile: "ConstantPool.java"  
  minor version: 0  
  major version: 49  
  Constant pool:  
const #1 = Method   #7.#17;      // java/lang/Object.<init>:()V  
const #2 = Field    #18.#19;      //java/lang/System.out:Ljava/io/PrintStream;  
  
const #18 = class   #26;      // java/lang/System  
const #19 = NameAndType #27:#28; // out:Ljava/io/PrintStream;  
const #26 = Asciz   java/lang/System;  
const #27 = Asciz   out;  
const #28 = Asciz   Ljava/io/PrintStream;;  
const #29 = Asciz   java/io/PrintStream;  
const #30 = Asciz   println;  
const #31 = Asciz   (Ljava/lang/String;)V;
```

Constant Pool Table

```
Compiled from "ConstantPool.java"  
class ConstantPool extends java.lang.Object  
  SourceFile: "ConstantPool.java"  
  minor version: 0  
  major version: 49  
  Constant pool:  
const #5 = Method #6.#23; // ConstantPool.Amie:()I  
const #6 = class #24; // ConstantPool  
const #13 = Asciz Amie;  
const #14 = Asciz ()I;  
const #15 = Asciz SourceFile;  
const #16 = Asciz ConstantPool.java;  
const #23 = NameAndType #13:#14; // Amie:()I  
const #24 = Asciz ConstantPool;  
const #25 = Asciz java/lang/Object;  
const #26 = Asciz java/lang/System;  
const #27 = Asciz out;  
const #28 = Asciz Ljava/io/PrintStream;;  
const #29 = Asciz java/io/PrintStream;  
const #30 = Asciz println;  
const #31 = Asciz (Ljava/lang/String;)V;
```

Type Safety of JVM

◆ Run-time type checking

- All casts are checked to make sure type safe
- All array references are checked to make sure the array index is within the array bounds
- References are tested to make sure they are not null before they are dereferenced.

◆ Additional features

- Automatic garbage collection
- NO pointer arithmetic

If program accesses memory, the memory is allocated to the program and declared with correct type

Field and method access

◆ Instruction includes index into constant pool

- Constant pool stores symbolic names
- Store once, instead of each instruction, to save space

◆ First execution

- Use symbolic name to find field or method

◆ Second execution

- Use modified "quick" instruction to simplify search

invokeinterface <method-spec>

◆ Sample code

```
void add2(Incrementable x) { x.inc(); x.inc(); }
```

◆ Search for method

- find class of the object operand (operand on stack)
 - must implement the interface named in <method-spec>
- search the method table for this class
- find method with the given name and signature

◆ Call the method

- Usual function call with new activation record, etc.

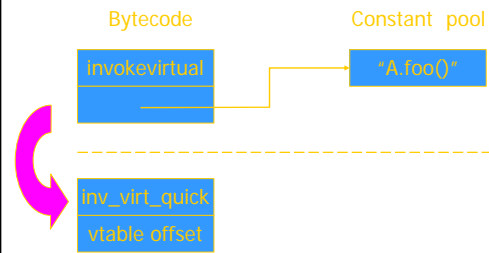
Why is search necessary?

```
interface Incrementable {  
    public void inc();  
}  
class IntCounter implements Incrementable {  
    public void add(int);  
    public void inc();  
    public int value();  
}  
class FloatCounter implements Incrementable {  
    public void inc();  
    public void add(float);  
    public float value();  
}
```

invokevirtual <method-spec>

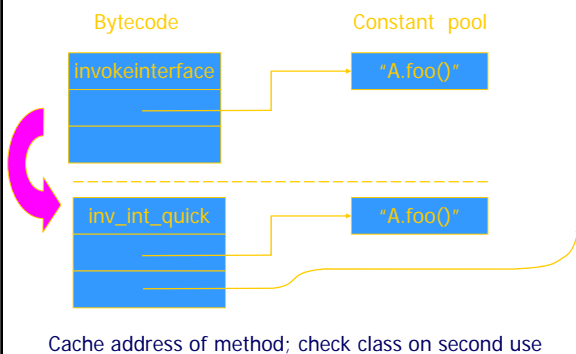
- ◆ Used when superclass of object is known
- ◆ Search for method
 - search the method table for this class
 - find method with the given name and signature
- ◆ Can we use static type for efficiency?
 - Each execution of an instruction will be to object from subclass of statically-known class
 - Constant offset into vtable
 - like C++, but dynamic linking makes search useful first time
 - See next slide

Bytecode rewriting: invokevirtual



- ◆ After search, rewrite bytecode to use fixed offset into the vtable. No search on second execution.

Bytecode rewriting: invokeinterface



Constant Pool Exercise