CMSC433, Spring 2007
Programming Language Technology and Paradigms

Java Review

Atif Memon

02/06/2007-02/13/2007

Java

- Descended from Simula67, SmallTalk, others
  - Superficially similar to C, C++
- Fully specified, compiles to *virtual machine*
  - machine-independent
- Secure
  - bytecode verification ("type-safe")
  - security manager

Object Orientation

- Combining data and behavior
  - objects, not developers, decide how to carry out operations
- Sharing via abstraction and inheritance
  - similar operations and structures are implemented once
- Emphasis on object-structure rather than procedure structure
  - … but procedure structure still useful

Example

```
public class Complex {
  private double r, i;

  public Complex(double r, double i) {
    this.r = r;  this.i = i;
  }
  public String toString() {
    return "(" + r + ", " + i + ")";
  }
  public Complex plus(Complex that) {
    return new Complex(r + that.r, i + that.i);
  }
}
```

Using Complex

```
public static void main(String[] args) {
     Complex a = new Complex(5.5,9.2);
     Complex b = new Complex(2.3,-5.1);
     Complex c;
     c = a.plus(b);
     System.out.println("a = " + a);
     System.out.println("b = " + b);
     System.out.println("c = " + c);
}
```

- instantiation
- **println** calls the **java.io.PrintStream.print(Object)** method
  - which calls the **String.valueOf** method
- The **String.valueOf** method is very simple**:**

```
public static String valueOf(Object obj)
{ return (obj == null) ? "null" : obj.toString();
}
```

5

The Class Hierarchy

- Classes by themselves are a powerful tool
  - Support *abstraction* and *encapsulation*
- Java also provides
  - *Inheritance* allows code reuse
    - Note: When you inherit from a class, you also "implement" the class's "interface" (will come back to interface)

6

Inheritance

- Each Java class *extends* or inherits code from exactly one superclass
- Permits reusing classes to define new objects
  - Can define the behavior of the new object in terms of the old one

7

Example

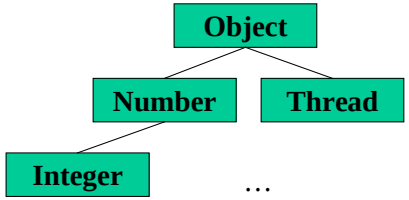
```
class Point {
  int getX() { ... }
  int getY() { ... }
}
class ColorPoint extends Point {
  int getColor() { ... }
}
```

- **ColorPoint** reuses **getX()** and **getY()** from **Point**
- **ColorPoint** "implements" the **Point** "interface"
  - They can be used anywhere a **Point** can be

8

Java Design

- Everything inherits from **Object***
  - Even arrays
  - Allows sharing, generics, and more



* Well, almost: there are primitive int, long, float, etc.

Interfaces

```
public interface MiniServlet extends Runnable {
        void setArg(String arg);
        void setOutputStream(OutputStream out);
}

class HelloWorld implements MiniServlet { ... }
class Print implements MiniServlet { ... }

MiniServlet s = new HelloWorld();
if (...) s = new Print();
s.setArg(...);
```

–*Interfaces* allow different classes to be treated the same

Interfaces

- An interface lists supported (public) methods
  - No constructors or implementations allowed
  - Can have final static variables
- A class can implement zero or more interfaces
- Given some interface **I**, declaring **I x** = ... means
  - **x** must refer to an instance of a class that implements **I**, or else **null**

Interface Inheritance

- Interfaces can *extend* other interfaces
  - Sometimes convenient form of reuse
- Given interfaces **I1** and **I2** where **I2** extends **I1**
  - If **C** implements **I2**, then **C** implements **I1**
- Because a class can implement multiple interfaces, interface extensions are often not needed

No Multiple Inheritance

- A class type can implement many interfaces
- But can only extend one superclass
- Not a big deal
  - Multiple inheritance rarely, if ever, necessary and often badly used
  - And it's complicated to implement well

13

Abstract Classes

- Sometimes want a class with some code, but with some methods unwritten
  - It can't be an interface because it has code
  - It can't be a regular class because it doesn't have all the code
    - You can't instantiate such a class
- Instead, we can mark such a class as *abstract*
  - And mark the unimplemented methods as *abstract*

14

Example from JDK

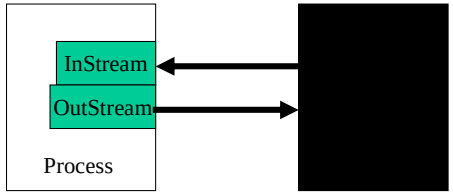
```
public abstract class OutputStream {
  public abstract void write(int b) ...;
  public void write(byte b[], int off, int len) ... {
      ... write(b[off + i]);...
  }
  ...
}
```

- Subclasses of OutputStream need not override the second version of write(...)
  - But they do need to override the first one as it is abstract

15

I/O streams

- Raw communication takes place using streams



- Java also provides readers and writers
  - character streams
- Applies to files, network connections, strings, etc.

16

I/O Classes

- **OutputStream** – byte stream going out
- **Writer** – character stream going out
- **InputStream** – byte stream coming in
- **Reader** – character stream coming in

17

Applications and I/O

- Java "external interface" is a public class
  - **public static void main(String [] args)**
- **args[0]** is first argument
  - unlike C/C++
- **System.out** and **System.err** are **PrintStreams**
  - **System.out.print(…)** prints a string
  - **System.out.println(…)** prints a string with a newline

18

Java Networking

- class Socket
  - Communication channel
- class ServerSocket
  - Server-side "listen" socket
  - Awaits and responds to connection requests

19

Example Client/Server

```
ServerSocket s = new ServerSocket(5001);
Socket conn = s.accept();
InputStream in = conn.getInputStream();
OutputStream out = conn.getOutputStream();
```

Server code


```
Socket conn = new Socket ("www.cs.umd.edu", 5001);
InputStream in = conn.getInputStream();
OutputStream out = conn.getOutputStream();
```

Client code

20

Example Client/Server

```
ServerSocket s = new ServerSocket(5001);
Socket conn = s.accept();
InputStream in = conn.getInputStream();
OutputStream out = conn.getOutputStream();
```

Server code

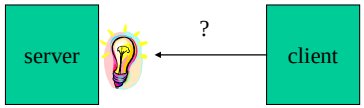

```
Socket conn = new Socket ("www.cs.umd.edu", 5001);
InputStream in = conn.getInputStream();
OutputStream out = conn.getOutputStream();
```

Client code

Example Client/Server

```
ServerSocket s = new ServerSocket(5001);
Socket conn = s.accept();
InputStream in = conn.getInputStream();
OutputStream out = conn.getOutputStream();
```

Server code



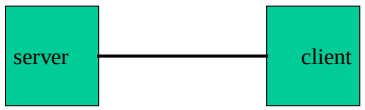

```
Socket conn = new Socket ("www.cs.umd.edu", 5001);
InputStream in = conn.getInputStream();
OutputStream out = conn.getOutputStream();
```

Client code

Example Client/Server

```
ServerSocket s = new ServerSocket(5001);
Socket conn = s.accept();
InputStream in = conn.getInputStream();
OutputStream out = conn.getOutputStream();
```

Server code



Note: The server can still accept other connection requests on port 5001

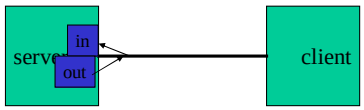
```
Socket conn = new Socket ("www.cs.umd.edu", 5001);
InputStream in = conn.getInputStream();
OutputStream out = conn.getOutputStream();
```

Client code

Example Client/Server

```
ServerSocket s = new ServerSocket(5001);
Socket conn = s.accept();
InputStream in = conn.getInputStream();
OutputStream out = conn.getOutputStream();
```

Server code



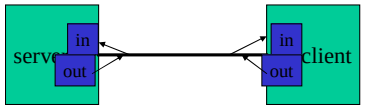

```
Socket conn = new Socket ("www.cs.umd.edu", 5001);
InputStream in = conn.getInputStream();
OutputStream out = conn.getOutputStream();
```

Client code

Example Client/Server

```
ServerSocket s = new ServerSocket(5001);
Socket conn = s.accept();
InputStream in = conn.getInputStream();
OutputStream out = conn.getOutputStream();
```

Server code




```
Socket conn = new Socket ("www.cs.umd.edu", 5001);
InputStream in = conn.getInputStream();
OutputStream out = conn.getOutputStream();
```

Client code

Possible Failures

- Server-side
  - ServerSocket port already in use
  - Client dies on accept
- Client-side
  - Server dead
  - No one listening on port
- In all cases IOException thrown
  - Must use appropriate throw/try/catch constructs

What is this code doing?

```
import java.lang.reflect.*;
public class ReflectionTest {
        public static void main(String args[])
                throws ClassNotFoundException,
                NoSuchMethodException,
                SecurityException,
                IllegalAccessException,
                InvocationTargetException
        {
                if( args.length != 2 ||
                   !(args[0].equals("toUpperCase") ||
                     args[0].equals("toLowerCase")) ) {
                        throw new IllegalArgumentException();
                }
                String command = args[0];
                Class str = Class.forName( "java.lang.String" );
                Method m = str.getMethod( command, null );
                Object result = m.invoke( args[1], null );
                System.out.println( result.toString() );
        }
```

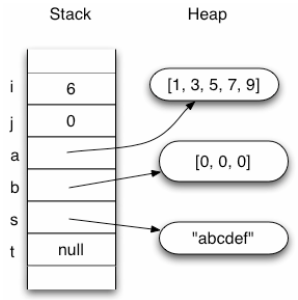

Class Objects

- For each class, there is an object of type **Class**
- Describes the class as a whole
  - used extensively in **Reflection** package
- **Class.forName("MyClass")**
  - returns class object for **MyClass**
  - will load **MyClass** if needed
- **Class.forName("MyClass").newInstance()**
  - creates a new instance of **MyClass**
- **MyClass.class** gives the **Class** object for **MyClass**

Objects and Variables

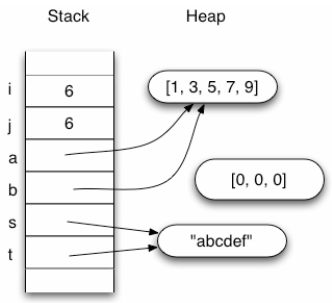
- Variables of a primitive type contain *values*
  - e.g., byte, char, int, ...
  - int i = 6;
  - Uninitialized values contain 0
  - Assignment copies values
- Variables of other types contain *references* to the heap
  - int[] a = new int[3];
  - Objects are allocated with new
  - Uninitialized object references are null
  - *Assignment copies references*, not the objects themselves

Example




```
int i = 6;
int j; // uninitialized
int [] a = {1, 3, 5, 7, 9};
int [] b = new int[3];
String s = "abcdef";
String t = null;
```


Example: Assignments




```
j = i;
b = a;
t = s;
```

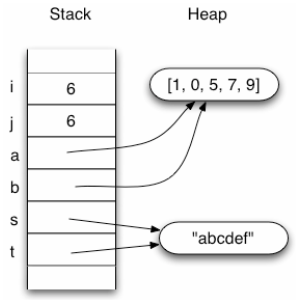

Garbage Collection

- What happens to array [0, 0, 0] in previous example?
  - It is no longer accessible
  - When Java performs *garbage collection* (GC) it will automatically reclaim the memory it uses
- Notice: No free() or delete in Java
  - Makes it much easier to write correct programs
  - Most of the time, very efficient

Mutability

- An object is mutable if its state can change
  - Example: Arrays are mutable
- An object is immutable if its state never changes
  - Once its been initialized
  - Example: Strings in Java are not mutable
    - There are no methods to change the state of a string

Example: Mutability




```
a[1] = 0;
// also changes b[1]
```

- Moral: Always be careful when you have *aliasing*
  - Multiple references to the same object

Method Invocation

- Syntax

  o.m(arg1, arg2, ..., argn);

  - Run the **m** method of object **o** with arguments **arg1...argn**
- Two ways to reuse method names:
  - Methods can be overridden
  - Methods can be overloaded

Overriding

- Define a method also defined by a superclass

```
class Parent {
  int cost;
  void add(int x) {
    cost += x;
  }
}
```

```
class Child extends Parent {
  void add(int x) {
    if (x > 0) cost += x;
  }
}
```

Overriding (cont'd)

- Method with same name and argument types in child class overrides method in parent class
- Arguments and result types must be identical
  - otherwise you are overloading the method
- Must raise the same or fewer exceptions
  - Why?

37

Declared vs. Actual Types

- The *actual type* of an object is its allocated type
  - **Integer o = new Integer(1);**
- A *declared type* is a type at which an object is being viewed
  - **Object o = new Integer(1);**
  - **void m(Object o) { ... }**
- Each object always has *one* actual type, but can have *many* declared types

38

Method Dispatch

- Consider again

  o.m(arg1, arg2, ..., argn);

- Only compiles if **o**'s declared type contains an appropriate **m** method
- Method corresponding to **o**'s actual type is what is invoked

39

Dynamic Dispatch Example

```
public class A {
   String f() { return "I'm an A! "; }
}

public class B extends A {
   String f() { return "I'm a B! "; }
   public static void main(String args[]) {
      A a = new B();
      B b = new B();
      System.out.println(a.f() + b.f());
   }
}
```

Prints I'm a B! I'm a B!

40

Self Reference

- **this** refers to the object the method is invoked on
  - Thus can access fields of this object as this.x or this.y
  - But more concise to omit
- **super** refers to the same object as **this**
  - But used to access methods/variables in superclass

41

Example of super

- Call a superclass method from a subclass

```
class Parent {
  int cost;
  void add(int x) {
    cost += x;
  }
}
```

```
class Child extends Parent {
  void add(int x) {
    if (x > 0) super.add(x);
  }
}
```

42

Overloading

- Methods with the same name, but different parameters (count or types) are overloaded
  - Invocation determined by name and types of params
  - Not return value or exceptions
- Resolved at **compile-time**, based on declared types
- Be careful: Easy to inadvertently overload instead of override!

43

Overloading Example

```
class Parent {
  int cost;
  void add(int x) {
    cost += x;
  }
  void add(Object s) throws NumberFormatException {
    cost += Integer.parseInt((String)s);
  }
}
class Child extends Parent {
  void add(String s) throws NumberFormatException {
    if (x > 0) cost += Integer.parseInt(s);
  }
}
```

```
Child c = new Child();
c.add((Object)"-1");
System.out.println(c.cost);
```

Prints -1

44

Static Fields and Methods

- *static* – stored "with the class"
  - Static fields allocated once, no matter how many objects created
  - Static methods are not specific to any class instance, so cannot refer to **this** or **super**
- Can reference class variables and methods through either class name or an object ref
  - Clearer to reference via the class name

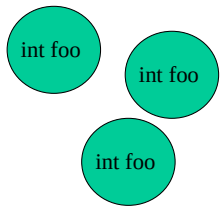
45

Static Field Example

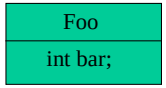
Class definition

```
Public class Foo {
  int foo;
  static int bar;
}
```

Objects of class Foo



Class implementation



46

Some Static Fields and Methods

- public **static** void main(String args[]) { … }
- public class Math {
  public final **static** PI = 3.14159…;
  }
- public class System {
  public **static** PrintStream out = …;
  }

47

Static Method Dispatch

- Let **B** be a subclass of **A**, and suppose we have

  A a = new B();

  *Declared type **A***
  *Actual type **B***

- Then
  - Class (static) methods invoked on **a** will get the methods for the *declared* type **A**
    - Invoking class methods via objects strongly discouraged
    - Instead, invoke through class
      - **A.m()** instead of **a.m()**

48

Static Method Dispatch Example

```
public class A {
   static String g() { return "This is A! "; }
}
public class B extends A {
   static String g() { return "This is B! "; }
   public static void main(String args[]) {
      A a = new B();
      B b = new B();
      System.out.println(a.g() + b.g());
   }
}
```

Prints This is A! This is B!

49

Better Use of Static Methods

```
public class A {
   static String g() { return "This is A! "; }
}
public class B extends A {
   static String g() { return "This is B! "; }
   public static void main(String args[]) {


      System.out.println(A.g() + B.g());
   }
}
```

Prints This is A! This is B!

50

Other Field Modifiers

- *final* – can't be changed
  - Must be initialized in declaration or in constructor
- *transient*, *volatile*
  - Will cover later
- *public*, *private*, *protected*, *package* (default)
  - Respectively, visible everywhere, only within this class, within same package or subclass, within same package

51

Other Method Modifiers

- *final* – this method cannot be overridden
  - Useful for security
  - Allows compiler to inline method
- *abstract* – no implementation provided
  - Class must be abstract
- *public* – visible outside this package
- *native*, *synchronized*
  - Will cover later

52

Poor man's polymorphism

- Every object is a subtype of **Object**
- Thus, a data structure **Set** that implements sets of **Objects**
  - can also hold **Strings**
  - or images
  - or … anything!
- The trick is getting them back out:
  - When given an **Object**, you have to downcast it

53

Subtyping

- Both inheritance and interfaces allow one class to be used where another is specified
  - This is really the same idea: subtyping
- We say that **A** is a *subtype* of **B** if
  - **A** extends **B** or a subtype of **B**, or
  - **A** implements **B** or a subtype of **B**

54

Downcasting

- **(Bar) foo**
  - Run-time exception if object reference by **foo** is not a subtype of **Bar**
  - Compile-time error if **Bar** is not a subtype of **foo** (i.e., it always throws an exception)
  - No effect at run-time; just treats the result as if it were of type **Bar**
- **o instanceof Foo** returns true iff **o** is a subtype of **Foo**

```
void m1(Object object) {
        if (object instanceof String) {
        // it is now safe to cast it
        String astr = (String) object; } }
```

55

Example

```
class DumbSet {
  public void insert(Object o) {..}
  public bool member(Object o) {..}
  public Object any() {..}
}

class MyProgram {
  public static void main(String[] args) {
    DumbSet set = new DumbSet();
    String s1 = "foo";
    String s2 = "bar";
    set.insert(s1);
    set.insert(s2);
    System.out.println(s1+"in set?"+set.member(s1));
    String s = (String)set.any(); // downcast
    System.out.println("got "+s);
  }
}
```

56

Java Classes and Objects

- Each object is an instance of a class
  - An array is an object
- Each class extends *one* superclass
  - **Object** if not specified
  - Class **Object** has no superclass

57

Objects Have Methods

- All objects, therefore, inherit them
  - Default implementations may not be the ones you want

public boolean **equals** (Object that) – *"conceptual" equality*
public String **toString**() – *returns printable representation*
public int **hashCode**() – *key for hash table*
public void **finalize**() – *called if object is garbage collected*

  - And others …

58

Equality

- Object .equals(Object) method
  - Structural ("conceptual") equality
- == operator (!= as well)
  - True if arguments reference the same object
  - **o == p** implies **o.equals(p)**

59

Overriding Equals

```
class Foo {
  public boolean equals(Foo f) { … }  // wrong!
}

class Foo {
  public boolean equals(Object o) {   // right!
    if (!(o instanceof Foo))
      return false;
    ...
  }
}
```

The first case creates an *overloaded* method, while the second *overrides* the parent (**Object**) method.

60

Overriding hashCode

- **hashCode()** is used for objects that may be stored in hash table
- Rule of thumb: If you override **equals()** or **hashCode()**, you should also override the other
  - **a.equals(b)** implies **a.hashCode() == b.hashCode()**
  - The default implementation of **equals()** in Object class simply checks if two object references x and y refer to the same object. i.e. It checks if x == y. This particular comparison is also known as "shallow comparison". However, the classes providing their own implementations of the equals method are supposed to perform a "deep comparison"; by actually comparing the relevant data members. Since Object class has no data members that define its state, it simply performs shallow comparison.

Preconditions

- Functions often have requirements on their inputs

```
// Return maximum element in A[i..j]
int findMax(int[] A, int i, int j) { ... }
```

  - **A** is non-empty
  - **i** and **j** must be non-negative
  - **i** and **j** must be less than **A.length**
  - **i** < **j** (maybe)
- These are called *preconditions* or *requires* clauses

Dealing with Errors

- What do you do if a precondition isn't met?
- What do you do if something unexpected happens?
  - Try to open a file that doesn't exist
  - Try to write to a full disk

Signaling Errors

- Style 1: Return invalid value

```
// Returns value key maps to, or null if no
// such key in map
Object get(Object key);
```

  - Disadvantages?

Signaling Errors (cont'd)

- Style 2: Return an invalid value and status

```
static int lock_rdev(mdk_rdev_t *rdev) {
  ...
  if (bdev == NULL)
    return -ENOMEM;
  ...
}

// Returns NULL if error and sets global
// variable errno
FILE *fopen(const char *path, const char *mode);
```

65

Problems with These Approaches

- What if all possible return values are valid?
- What if client forgets to check for error?
  - No compiler support
- What if client can't handle error?
  - Needs to be dealt with at a higher level

66

Exceptions in Java

- On an error condition, we *throw* an exception
- At some point up the call chain, the exception is *caught* and the error is handled
- Separates normal from error-handling code
- A form of non-local control-flow
  - Like goto, but structured

67

Throwing an Exception

- Create a new object of the class **Exception**, and **throw** it

```
if (i >= 0 && i < a.length )
  return a[i];
else throw new ArrayIndexOutOfBounds();
```

68

Method throws declarations

- A method declares the exceptions it might throw
  - public void openNext() throws UnknownHostException, EmptyStackException
    { … }
- Must declare any exception the method might throw
  - Unless it is caught in (masked by) the method
  - Includes exceptions thrown by called methods
  - Certain kinds of exceptions excluded

69

Exception Handling

- All exceptions eventually get caught
- First **catch** with supertype of the exception catches it
- **finally** is always executed

```
try { if (i == 0) return; myMethod(a[i]); }
catch (ArrayIndexOutOfBounds e) {
        System.out.println("a[] out of bounds"); }
catch (MyOwnException e) {
        System.out.println("Caught my error"); }
catch (Exception e) {
        System.out.println("Caught" + e.toString());  throw e;  }
finally { /* stuff to do regardless of whether an exception */
        /* was thrown or a return taken */  }
```

70

Masking Exceptions

- Handle exception and continue

```
while ((s = ...) != null) {
  try {
    FileInputStream f =
       new FileInputStream(s);
    ...
  }
  catch (FileNotFoundException e) {
     System.out.println(s + " not found");
  }
}
```

71

Reflecting Exceptions

- Pass exception up to higher level
  - Automatic support for throwing same exception
  - Sometimes useful to throw different exception

```
try {
  ... a[5] ...
}
catch (IndexOutOfBoundsException e) {
  throw new EmptyException("Arrays.min");
}
```

72

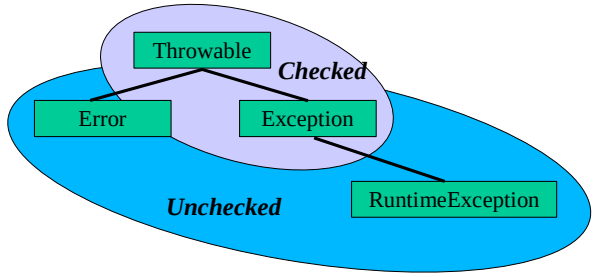
Exception Chaining

- Indicate the cause of a thrown exception
  - Specify the exception that caused this one
  - Shows cause chain in stack trace

```
try {
  ... a[0] ...
}
catch (IndexOutOfBoundsException e) {
  // e can be retrieved from getCause()
  throw new Exception("Arrays.min", e);
}
```

73

Exception Hierarchy



74

Unchecked Exceptions

- Subclasses of **RuntimeException** and **Error** are unchecked
  - Need not be listed in method specifications
- Currently used for things like
  - NullPointerException
  - IndexOutOfBoundsException
  - VirtualMachineError
- Is this a good design?

75