

CMSC433, Spring 2004 Programming Language Technology and Paradigms

Java Review

Jeff Foster
January 29, 2004

Administrivia

- Project 1 will be posted later today
 - Due February 11
- Reading: (today) Liskov ch. 1, 2, (Tue) ch. 4
 - Supplemental: Eckel ch. 1, 7, 8, 9

2

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

3

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
 - behavior more stable than implementation
 - ... but procedure structure still useful

4

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);
    }
}
```

5

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);
}
```

6

The Class Hierarchy

- Classes by themselves are a powerful tool
 - Support *abstraction* and *encapsulation*
- Java also provides two other abilities
 - *Interfaces* allow different classes to be treated the same
 - *Inheritance* allows code reuse
 - **Note:** When you inherit from a class, you also “implement” the class’s “interface”

7

Project 1: 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(...);
```

8

Interfaces

- An interface lists supported (public) methods
 - No constructors or implementations allowed
 - Can have *final* static variables
- A class can implement (be a subtype of) 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**

9

Interface Inheritance

- Interfaces can *extend* other interfaces
 - Sometimes convenient form of reuse (see project 1)
- Given interfaces **I1** and **I2** where **I2** extends **I1**
 - If **C** implements **I2**, then **C** implements **I1**
- Since a class can implement multiple interfaces, interface extensions are often not needed

10

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

11

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

12

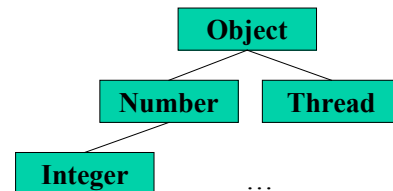
Subtyping

- 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**

13

Java Design

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



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

14

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

15

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*

16

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, since it's abstract
 - (Note: They may want to override anyhow for efficiency)

17

Example from Project 1

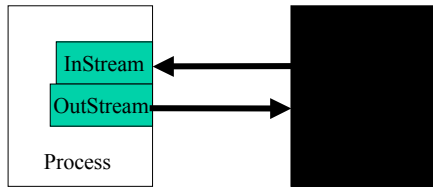
```
public abstract class ServletFilter extends
    OutputStream implements MiniServlet
{
    public abstract void write(int b) ...;
    public abstract void close() ...;
    public abstract void flush() ...;

    public abstract void setArg(String arg);
    public abstract void setServlet(MiniServlet s);
    public abstract void
        setOutputStream(OutputStream out);
}
```

18

I/O streams

- Raw communication takes place using streams



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

19

I/O Classes

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

20

Some OutputStreams and Writers

- Example classes
 - **ByteArrayOutputStream** – goes to byte []
 - **FileOutputStream** – goes to file
- **OutputStreamWriter**
 - wraps around **OutputStream** to get a **Writer**
 - takes characters, converts to bytes
 - can specify encoding used to convert
- Other wrappers
 - **PrintWriter** – supports **print**, **println**
 - **StringWriter**

21

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**
 - should be **PrintWriter**, but would break 1.0 code
 - **System.out.print(...)** prints a string
 - **System.out.println(...)** prints a string with a newline
- **System.in** is an **InputStream**
 - not quite so easy to use

22

Java Networking

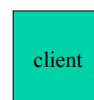
- class **Socket**
 - Communication channel
- class **ServerSocket**
 - Server-side “listen” socket
 - Awaits and responds to connection requests

23

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

24

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();
```

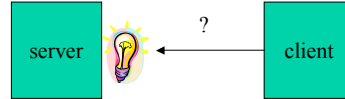
Client code

25

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();
```

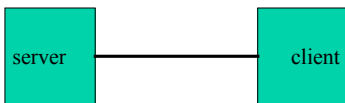
Client code

26

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();
```

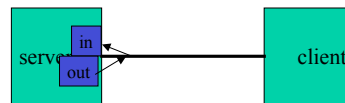
Client code

27

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();
```

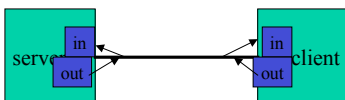
Client code

28

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

29

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

30

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**