CMSC433, Spring 2001 Programming Language Technology and Paradigms Basic Java, continued

Alan Sussman
February 12, 2002

Administrivia

- Project 1 due Wednesday, 6PM
  - threads issue
  - anyone not get email to reflector?
  - projects for commentary emailed after late deadline
- Second Java project posted soon

Last time - Java

- Instance variable/method modifiers
  - both: public, protected, package, private, static
  - method: abstract, final, native, synchronized
  - field: transient, volatile, final
- Overriding/Overloading
- Constructors
  - call another constructor or super() first
- Static methods/instance variables
  - associated with Class object
  - Class object used for reflection operations
- Interfaces are types

Types

- A type describes a set of values that can be:
  - held in a variable
  - returned in an expression
- Types include:
  - primitive types – boolean, char, short, int , …
  - Reference types:
    - Class types
    - Array types
    - Interface types

Class types

- Using the name of a class as a type means a reference to an instance of that class or a subclass is a permitted value
  - a subclass has *all* the fields of its superclass
  - a subclass has *all* the methods of its superclass
- **null** is also an allowed value

Array types

- If **S** is a subtype of **T**
  - **S[]** is a subtype of **T[]**
- **Object[]** is a supertype of all arrays of reference types
- Storing into an array generates a run-time check that the type stored is a subtype of the *declared* type of the array elements
- Performance penalty?
- Similar (and maybe worse) problems in C++

Example: Object[]

```
public class TestArrayTypes {
        public static void reverseArray(Object [] A) {
                for(int i=0, j=A.length-1; i<j; i++,j --) {
                        Object tmp = A[i];
                        A[i] = A[j];
                        A[j] = tmp;
                }
        }
        public static void main(String [] args) {
                reverseArray(args);
                for(int i=0; i < A.length; i++)
                        System.out.println(args[i]);
        }
}
```


Interface types

- Using the name of an interface as a type means
  - a reference to any instance of a class that implements the interface is a permitted value
  - **null** is also allowed
- Object referenced is guaranteed to support *all* the methods of the interface
  - invoking a method on an interface might be a bit less efficient

Object Obligations

- many operations have default implementations
  - which may not be the ones you want

```
public boolean equals (Object that) { … } // return this == that
public String toString() { … } // returns print representation
public int hashCode() { … } // key for accessing object
// important that a.equals(b) implies a.hashCode()==b.hashCode()
public void finalize() { … } // called before object garbage
// collected, default is {}
public Object clone() { … } // default is shallow bit-copy if class
// implements Cloneable, throw CloneNotSupportedException
// otherwise
```


Poor man's polymorphism

- Every object is an **Object**
- An **Object[]** can hold references to any objects
- E.g., for a data structure **Set** that holds a set of **Object**
  - can use it for a set of **String**
  - or a set of images
  - or a set of anything
- Java's container classes are all containers of **Object**
  - when you get a value out, have to downcast it

Interacting with External Environment

Applications and I/O

- Java external interface is a public class
- via **public static void main(String [] args)**
- **args[0]** is first argument
  - unlike C/C++
- **System.out** and **System.err** are **PrintStreams**'s
  - should be **PrintWriter**'s, 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

Input (JDK 1.1 and higher)

- Wrap **System.in** in an **InputStreamReader**
  - converts from bytes to characters
- Wrap the result in a **BufferedReader**
  - makes input operations efficient
  - supports **readLine()** interface
- **readLine()** returns a string
  - returns **null** if at EOF

Example Echo Application

```
import java.io.*;
public class Echo {
  public static void main(String [] args) {
        String s;
        BufferedReader in = new BufferedReader(
                new InputStreamReader(System.in));
        int i = 1;
        try {
                while((s = in.readLine()) != null)
                        System.out.println((i++) + ": " + s); }
        catch(IOException e) {
                System.out.println(e); }
  }
}
```


Java Programming Environments

Packages

- Classes grouped into packages
- Example: **java.awt.image**
  - avoids namespace clashes
- But no semantics to having a common prefix
  - e.g., between **java.awt** and **java.awt.image**
- Package names are an implicit or explicit part of a class name

Packages (cont.)

- Import makes a class or package name implicit
  - e.g., allows use of **ColorModel** instead of **java.awt.image.ColorModel** by:
    - **import java.awt.image.colorModel;**
  - to import all classes in a package, use *
    - e.g., **import java.awt.image.*;**
- Implicit at the beginning of every Java file
  - **import java.lang.*;**
- Import *never* required, just allows use of shorter names

Files – what goes where

- Each *public* class **C** must be in a file **C.java**
- If a class **C** is part of package **P**
  - **package P;** must be the first statement in **C.java**
  - which must be in a directory **P**
  - treats . in package name as subdirectories
- Reverse of domain name is reserved package name
  - **edu.umd.cs** is reserved for UMD CS department