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

Alan Sussman

October 2, 2001

Administrivia

- C++ project
  - private drivers posted soon, grades by end of week
  - comments on other students projects due 10/9
    - submit as project 11 (e.g. **submit 11 1.txt 2.txt**)
  - you will get the comments back for your project
- Java project
  - default compiler is still 1.2, use the one in /usr/opt/java130/bin for 1.3
  - try using jikes for compiling, in ~pugh/bin
    - faster than javac

Administrivia (cont.)

- Exam next Thursday, Oct. 11
  - practice exam on web page by tomorrow, talk about it on Tuesday
- Guest lecturer on Thursday – Bill Pugh
  - he can answer any Java question, and probably most questions about project, but see TA or Dr. Porter for detailed project questions on Thursday or Friday
- A warning
  - will be checking projects against each other for similarities (with a program)
    - and against similar projects from previous semesters

Last time - Java

- Overriding methods – dynamic dispatch
  - same name, same argument types
  - for normal methods, like C++ virtual functions
    - run-time type of object, static type of arguments
  - for static methods, based on static/declared type of reference
  - use **super** to get at methods/variables in superclass
- Overloading
  - same name, different argument types

Constructors

- Declaration syntax same as C++
  - no return type specified
  - method name same as class
- First statement can/should be **this(args)** or **super(args)**
  - if those are omitted, **super()** is called
  - must be very first statement, even before variable declarations
- *not* used for type conversions or assignments
- void constructor generated if no constructors given

Static class components

- They belong to the class
  - static variables allocated once, no matter how many objects created
  - static methods are not specific to any class instance, so can't refer to **this** or **super**
- Can reference class variables and methods through either class name or an object ref
  - poor style to reference via object references

Interfaces

- An interface is an object type – no associated code or instance variables
  - only describes methods supported by interface
- A class can *implement* (be a subtype of) many interfaces
- Interfaces may have final static variables
  - to define a set of constants (like **enum** in C++)

Interface example

```
public interface Comparable {
        public int compareTo(Object o)
}
public class Util {
        public static void sort(Comparable []) { … }
}
public class Choices implements Comparable {
        public int compareTo(Object o) {
                return … ;
        }
}
…
        Choices [] options = … ;
        Util.sort(options);
…
```


No multiple inheritance

- A class type can be a subtype of many other types (**implements**)
- But can only inherit method implementations from one superclass (**extends**)
- Not a big deal
  - multiple inheritance rarely, if ever, necessary and often poorly used
- And it's complicated to implement well

Garbage collection

- Objects that are no longer accessible can be garbage collected
- Method **void finalize()** called when an object is collected
  - best to avoid using it, since no way to tell when it will get called
- Garbage collection not a major performance bottleneck
  - **new/delete** in C++ can be expensive too

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**

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