Class, Strings, Packages

CS151

Chris Pollett

Sept. 21, 2005.

Outline

- Interfaces and Abstract Classes
- Strings
- Wrapper Classes
- Packages

Interfaces

- Interfaces are a special kind of class that only declares what features are to be supported by implementers.

  [*ClassModifiers*]interface *InterfaceName* [extends *Interface1*, *Interface2*, …]{

  InterfaceMemberDeclarations

  }

  For example,

  ```
  public interface Runnable {
      public void run();
  }
  ```

  Classes can implement an interface by overriding the methods declared in the interface.

Abstract Classes

- Classes can also declare methods and defer implementation to a subclass using the keyword abstract:

```
abstract class MyClass
{
    abstract void myMethod();
}
```

- An *abstract class* is a class with at least one abstract method.

Strings in Java

- A *string* is a sequence of characters. This notion in Java is encapsulated in the String class and StringBuffer classes.
- String are used for immutable sequences of characters; StringBuffers are used for mutable sequences.
- Strings are different from most usual classes in Java in that:
  - They can be created using String literals. String a="hello";
  - Operators + and += can be applied to them.

String Comparison

- s1== s2 checks equality of references for String’s, not if the two strings have the same characters.
- s1.equals(s2) checks the latter.
- Other useful methods are compareTo and equalsIgnoreCase.
- The String class supports a canonical representation of Strings.
- That is, it maintains an internal pool of unique String objects. To get this internal object, can do things like: String s2 = s1.intern();
- Comparing interned objects by reference always yields the same result as comparing by equality.

toString()

- Defining a toString() method for a class allows one to define a string representation of instances of a class.
- For example, for the Point class we might write: public String toString(){return “(”+x+ “, ”+y+“)”;}
- Once we’ve done this, the following would be legal:

```
Point p = new Point(10.0, 20.0);
System.out.println(“p” + p);
```

Converting between Character Arrays and Strings.

- Unlike C/C++, strings in Java are not character arrays like char[].
- However, Java does support converting in and out of such arrays:

```
char data[] = {‘f’, ‘o’, ‘o’};
String str = new String(data);
//Same as String str = “foo”;
also can do
String str=“bar”;
char data[] = str.toCharArray();
```

File I/O

- Java supports various kinds of Streams, Readers and Writers for doing I/O.
- For example, System.in is an InputStream object and System.out and System.err are PrintStream objects.
- One can also use readers to read from/write to a file:

```
try
{
    BufferedReader in=new BufferedReader(new
      FileReader(“infile.txt”);
    PrintWriter out=new PrintWriter(new
      BufferedWriter(
         new FileWriter(“outfile.txt”)));
      String line = in.readLine();
      out.println(line);
      out.flush(); out.close();
}
catch(IOException e){}
```

Working with Strings

- Often we need to split strings into smaller pieces known as tokens, that are separated by delimiters. For example:

```
String a="Michael:Owens:123 OakStreet:Chicago:IL:60606";
```

- We could use the methods indexOf() and substring() to do this.
- A better way is to use split() or to use a StringTokenizer.

```
StringTokenizer st = new StringTokenizer(a, ":");
while(st.hasMoreTokens()){
    System.out.println(st.nextToken());
}
```

Wrapper Classes

- For each primitive type in Java there is a class that allows one to wrap the primitive value in an object.
- For example, boolean--> Boolean, byte --> Byte, char-->Character, etc.
- Can create objects/get out using in a natural way:

```
Integer obj = new Integer(5);
int i = obj.intValue();
//for other types would use typeValue() to unbox.
//can also use valueOf to parse String version of type:
Double.valueOf("1E3").doubleValue();
```

- Java 5 supports autoboxing/unboxing of primitive types used in collection objects:

```
List myNums = new ArrayList();
myNums.add(55); // same as myNums.add(new Integer(55));
```

Packages

- A Java program consists of one or many classes.
- Java has two means for organizing large amounts of classes:
  - Files -- whichcan contain one public class and maybe several helper classes which are non public
  - Packages -- which comprise related classes, interfaces, or other packages.
- To specify a class belongs to a package we use the command:

  package *Name*;

  At the start of the file the class belongs to.
- A class can then be specified in two ways:
  - With a fully qualified name

    geometry.Point p = new geometry.Point()
  - By importing the class and using a simple class name:

    import geometry.Point;

    // now can use Point.

    //can also do import geometry.*;

More Packages

- The usual naming convention is to use reverse of internet domain names for packages: edu.sjsu.cs
- Packages should contain: closely related classes and classes that change together when a change is made.
- Packages should not contain: classes that are not reused together.
- When compiling the files for the package package1.package2, one would move to the directory, such that there are the .java files are in the package1/package2 subdirectory. Then we'd type:

  ```
  javac package1/package2/MyClass.java
  ```
- If we want to specify a different target directory we could use the -d option for javac.
- To run our class we then type:

  ```
  java package1.package2.Myclass.java
  ```
- In general, to make our package runnable from another directory, we'd have to add the path to our package in the CLASSPATH environment variable.