8355
Java Lab:
From the Very Beginning
- Part 3 of 3

Steve Ryder, JSR Systems: adapted from Object Oriented programming by:
Stephen Pipes IBM Hursley Park Labs, United Kingdom
SHARE Meeting

Intro to Java recap

- Classes are like user-defined types
- Objects are like variables of those types
- We send messages which invoke methods
- Classes have a class object

Agenda

- Inheritance and relationships
- Lab Exercises (see examples of polymorphic overloading, encapsulation, and inheritance)
- Make changes to programs to illustrate benefits of Inheritance and encapsulation.
- Abstraction and interfaces
- Polymorphism (if we have time)
- Overloading

The Circle Class

```
class Circle
{
   // Data encapsulated by the class
   private SimplePoint center;
   private int radius;

   // Methods that form external interface
   public double circumference() { ... }
   public double area() { ... }
   public SimplePoint getCenter() { return center; }
   public int getRadius() { return radius; }
}
```

The GraphicCircle class #1

```
class GraphicCircle
{
   // Data encapsulated by the class
   private SimplePoint center;
   private int radius;

   // Methods that form external interface
   public double circumference() { ... }
   public double area() { ... }
   public SimplePoint getCenter() { return center; }
   public int getRadius() { return radius; }

   public void draw(Graphics g) { ... }
 }
```

The GraphicCircle class #1

- It's a cut-and-paste job
- Error prone
- Two copies of the "Circle" code
- Harder to maintain

The GraphicCircle class #2

```
class GraphicCircle
{
   // Data encapsulated by the class
  private Circle circle;

   // Methods that form external interface
  public double circumference() { ... }
  public double area() { ... }
  public SimplePoint getCenter()
  { return circle.getCenter();
  }
  public int getRadius() { return circle.getRadius(); }

 public void draw(Graphics g) { ... }
}
```

The GraphicCircle class #2

- Uses existing code and logic for base circle
- But, needs lots of annoying wrapper methods!

The GraphicCircle class #3

```
class GraphicCircle extends Circle
{
   public void draw(Graphics g) { ... }
}
```

The GraphicCircle class #3

- GraphicCircle is defined as an extension of the Circle class
- We call it a subclass
- GraphicCircle has all of the functionality of Circle, plus its own additional methods (and data)
- We say that GraphicCircle inherits the functionality of Circle We call the act of extending a class inheritance

Inheritance

- GraphicCircle is a Circle You can use it anywhere a Circle is required
  - public aMethod(Circle c)
- You can treat it just like a Circle when you use it
  - graphicCircle.getRadius()
- Use inheritance when you have an "is a" relationship

Other key relationships

- "is a" -> inheritance
- "has a" -> data member (e.g. Circle has a SimplePoint, its center) - containment
- "uses" class A uses class B if:
  - a method of A sends a message to an object of class B
  - a method of A creates, receives or returns objects of class B
  - try to minimize the number of classes that use each other

Lab Exercises

Import from File System (JavaLabs/projects/JFTVB3):

This project was done several years ago (convert a COBOL extract and report job stream running on zOS to run on an NT box). I had convinced my client up front that it would cost him less than the cost of a COBOL compiler for me to just do it in Java. I actually developed on a Windows 98 box and it ran without a hitch on the NT box AND it ran faster on the NT box. The speed advantage was because COBOL did not have the ability to dynamically create a variable number of output files based on user input. The data files consisted of discrete sub-sets of data (report type and region). The COBOL application just sorted the whole file one time. Then each report (about 20 different executions of 3 different reports in the production environment) read the WHOLE file! I wrote a simple extract that simply split the file into 20 separate files (a simple trick of creating a new instance of JsrLineOut for each report-type, region combination). Then each report sorted one file (1/20th of the original sort so it was sorted in memory), and "printed" the report.

- Examine JsrUtil for examples of overloading (a form of polymorphism) and useful ways to do things in Java to mimic COBOL behavior.

- Examine JsrSysout for examples of class and instance variables and how they might be useful.

- Examine JsrLineIn and JsrLineOut as a useful example of encapsulating I/O logic (especially the exception handling).

- Examine IbfExtract and file IbfTest.txt as an example of allocating multiple output files depending on contents of input file. Run IbfExtract (Arguments IbfTest.txt). Examine console (benefits of JsrSysout). Navigate to Workspace in file system to see that other files were created in the workspace, but do not appear in the directory because they were not imported. Open logIbfExtract.txt

- Examine IbfReport (example of Super Class)

- Examine IbfReport1 (example of a sub-class)

- Run IbfReport1
  (Arguments cc72.txt PR5196COST 1.txt)

- Modify IbfReport to make explicit setter methods (setBranch and setMonth) that are abstract. What happens to IbfReport1?

- Modify IbfReport1 change setCC to setBranch and setMonth. This illustrates power of Abstract classes to force sub-classes to implement required methods.

- Run IbfReport1 again.

Questions?

Now for more detail on Abstraction and Interfaces

Abstraction and interfaces

We extend our super class to create more specific and useful sub classes

	Circle
Shape	Rectangle
	Hexagon
General super class	Specific sub class

```
public abstract class Shape
{
  public abstract double area();
  public abstract double circumference();
}
```

Abstraction and Interfaces

- **The Circle class can calculate the area and circumference of a circle only**

```
class Circle extends Shape
{
  protected double r; protected static final double
  PI=3.14159265358979323846;
  public Circle() { r = 1.0; }
  public Circle(double r) { this.r = r; }

  public double area() { return PI * r * r; }
  public double circumference() { return 2 * PI * r; }
  public double getRadius() { return r; }

}
```

Abstraction and Interfaces

- On the other hand, we want the Shape class (super-class) to encapsulate whatever features all our shapes have in common, such as area and circumference

- Since the Shape class is generic to all shapes, it cannot implement these features, so we use abstract methods as placeholders in our Shape class

Abstraction and Interfaces

- Abstract methods do not implement functionality; note the semicolon immediately after the method definition.
- Classes that contain abstract methods cannot be instantiated, since they contain no code
- A class may be declared abstract even if it has not abstract methods, thus preventing it from being instantiated.
- A subclass of an abstract class can be instantiated if it overrides each of the abstract methods of its superclass and provides an implementation (method body).

Abstraction and Interfaces

- Abstraction allows us to group types of classes and deal with them in a standard way
- We can group any classes that extend and implement the abstract Shape class

Abstraction and Interfaces

- **What is this code doing?**

```
Shape[] shapes = new Shape[2]; // create an array of Shape
shapes[0] = new Circle(2.0); // Circle is element 0
shapes[1] = new Rectangle(1.0, 2.0); // Rectangle is element 1

double total_area = 0;
for (int i = 0; i < shapes.length; i++) total_area += shapes[i].area(); //
compute the total area
```

Abstraction and Interfaces

- Defines a Shape array and populates with different types of Shape. We do not need to cast the Circle and Rectangle classes onto the Shape array, since Circle and Rectangle inherit from Shape

- Calculates the total area of all Shapes in the array by calling the abstract methods of the Shape class. The abstract Shape class provides pointers to the correct methods in the appropriate sub-classes

Abstraction and Interfaces

- We can draw our shapes by defining an abstract class called DrawableShape, which provides the abstract methods to draw the shape
- To draw a circle, we define DrawableCircle which provides the functionality to draw a circle, pointed to by the abstract DrawableShape class

Abstraction and Interfaces

- We need to calculate a circle before drawing it, so we extend our Circle class

- However, Java does not allow more than one super class, so we define DrawableShape as an interface

Drawable Shape	
	Drawable Circle
Circle	

Abstraction and Interfaces

Drawable Shape	
	Drawable Circle
Circle	

```
public interface DrawableShape
{
  public void setColor(Color c);
  public void setPosition(double x, double y);
  public void draw(DrawWindow dw);
}
```

Abstraction and Interfaces

- **Our DrawableCircle class will implement the DrawableShape interface**

```
public class DrawableCircle extends Circle implements DrawableShape
{
  private Color c;
  private double x, y;
  // Provide a constructor that sets the super-class…
  public DrawableCircle(double r) { super(r); }
  // Now we implement those methods defined in DrawableShape…
  public void setColor(Color c) { this.c = c; }
  public void setPosition(double x, double y) { this.x = x; this.y = y; }
  public void draw(DrawWindow dw) { dw.drawCircle(x, y, r, c); }
}
```

Abstraction and Interfaces

```
Shape[] shapes = new Shape[2]; // create an array of Shape
DrawableShape[] drawables = new DrawableShape[2];  // create an array of drawable shapes

// now we create some drawable shapes…
DrawableCircle dc = new DrawableCircle(1.1);
DrawableRectangle dr = new DrawableRectangle(…);

// since all drawable shapes extend Shape and implement DrawableShape, we can assign the
above drawable classes to both arrays…
shapes[0] = dc;
shapes[1] = dr;
drawables[0] = dc;
drawables[1] = dr;
double total_area = 0;
for (int i = 0; i < shapes.length; i++)
{
  total_area += shapes[i].area();  // compute the total area
drawables[i].setPosition(i*10.0, i*10.0);
drawables[i].draw(draw_window);
}
```

Abstraction and Interfaces

- Abstract classes may contain both abstract and implemented methods
- Interfaces contain only abstract methods. There can be no method code in an interface
- We may implement many interfaces in Java. This way, it is possible to implement a larger set of functionality into a single class.
- Constants may be defined in interfaces. These constants will be available to any class that implements the interface

Abstraction and Interfaces

- Interfaces can have super-interfaces in the same way that classes can have super-classes
- The one difference is that interfaces can inherit from more than one super-interface
- Should a class implement such an interface, it must implement the abstract methods defined in the interface and all its super-interfaces

Polymorphism

- Something that is polymorphic may appear in different guises. A polymorphic object may be cast appropriately to suit

Example: Thread cast to Object added to a Vector.

- Down-casting casts an object to one of its descendents

Example: Remove Thread from Object Vector.

Polymorphism

- We can retrieve our object from the Vector using the elementAt method. It will be returned as type Object since it was cast to type Object before we passed it to the Vector

- If we call our object's toString method, it will be downcast to its original type Thread. Why is this?

Overloading

- Method overloading is a useful form of polymorphism, which allows objects to behave in exactly the same way irrespective of the information passed it

- In order to overload methods, we define more than one method of the same name in a class, but with different types or numbers of args.

```
System.out.print method...
void print(Object arg) { … }
void print(String arg) { … }
void print(char[] arg) { … } …
```

Overloading

- If the print method is invoked with an argument of unknown type then the compiler will select the closest match

- If a Date object is passed to System.out.print, the void print(String arg) { … } method would be chosen (note: most objects implement a toString method).

Overloading

A simple addition class. We may pass two or three integer arguments in to the Addition object. The same operation will be performed in either case.

```
class Addition
{
   int calc(int a, int b)
   {
         return (a + b);
   }
   int calc(int a, int b, int c)
   {
       return (a + b + c);
   }
}
```

Overloading

- **Constructors may also be overloaded**

```
public class Shape
{
   public Shape(int xPos, int yPos) { … }
   public Shape(int xPos, int yPos, int width, int height) { … }
}
```

We may initialize this class in one of two ways.

```
new Shape(x,y); // provide x and y positions
new Shape(x,y,w,h);  // provide x, y positions & width, height data
new Shape(); // default constructor <--NO!
```

What do we know?

- Abstract classes provide a common interface to all sub-classes. They cannot be instantiated

- Abstract methods provide no implementation but are placeholders for methods in subclasses

- Classes may have only one super-class.
  Interfaces allow us to extend the functionality of a class by providing abstract methods.
  Interfaces may have many super-interfaces

- Polymorphism allows down-casting and overloading