Abstract, Interface, GUIs

Ch. 11 & 16

Abstract

- A class declared abstract cannot be instantiated (we can't create an object of its type).
- A method declared abstract MUST be implemented if a class subclasses from abstract class.
- Useful when we know certain methods should be implemented, and we know how other methods should be implemented.

```
public abstract class Shape {
   private boolean isSymmetric;
   public void setIsSymmetric(boolean s){
      isSymmetric = s;
   }
   public boolean getIsSymmetric(){
      return isSymmetric;
   }

   abstract double computeArea();
}
```

```
public class Rectangle extends Shape {

   private Point lowerLeft, upperRight;

   Rectangle(Point lowerLeft, Point upperRight) {
      this.lowerLeft = lowerLeft;
      this.upperRight = upperRight;
   }

   @Override
   public double computeArea() {
      // computes and returns area of rectangle
   }
}
```

```
public class Circle extends Shape {

   private double radius;
   private Point center;

   public Circle(Point center, double radius) {
      this.radius = radius;
      this.center = center;
   }

   @Override
   public double computeArea() {
      return (Math.PI * Math.pow(radius, 2));
   }
}
```

```
public class PolymorphismExample {
   public static void main(String[] args) {
      Shape[] shapes = new Shape[2];

      Circle circle = new Circle(new Point(0.0,0.0), 1.0);
      Rectangle rectangle = new Rectangle(
                                        new Point(0.0, 0.0),
                                        new Point(2.0, 2.0));
      shapes[0] = circle;
      shapes[1] = rectangle;

      for (Shape shape : shapes) {
         System.out.println("Shape is: " +
                            shape.getClass() +
                            " and area is: " +
                            shape.computeArea());
      }
   }
}
```

Interface

- An interface is not a class. We cannot instantiate interfaces.
- An interface specifies a set of methods that an implementing class **must** override and define.
- A class *implements* an interface. Meaning the class must implement (override) all methods defined by interface.

Interface

- Why use interface? It separates what a class should do vs. how it should do it.
- Why not use abstract? Because a class can implement multiple interfaces.
- Something to note: instance variables cannot be declared in an interface.

```
interface Person {
   void setAge(int age);
   void setGender(char gender);
   void setName(String name);
   int getAge();
   char getGender();
   String getName();
}
```

```
interface Student extends Person {
   void setCollege(String college);
   void setGpa(double gpa);
   void setYear(byte year);
   String getCollege();
   double getGpa();
   byte getYear();
}
```

```
interface Staff extends Person {
   void setSalary(double salary);
   void setStartDate(Date start);
   void setEndDate(Date end);
   void setSSN(String ssn);
   double getSalary();
   Date getStartDate();
   Date getEndDate();
   String getSSN();
}
```

```
class StudentEmployee implements Student, Staff {
   // methods required by Person
   public void setAge(int age){... }
   ...
   public String getName(){...}

   // methods required by Student
   public void setCollege(String college){... }
   ...
   public YearInSchool getYear(){...}

   // methods required by Staff
   public void setSalary(double salary){...}
   ...
   public String getSSN(){...}
}
```

GUI

- GUI stands for Graphical User Interface.
  - enables the user to interface with a program via the use of graphical components such as windows, buttons, text boxes, etc. as opposed to text-based interfaces like the traditional command line.

```
import javax.swing.*;
import java.awt.*;
class SampleAWT {
   public static void main(String[] args) {
      JFrame frame = new JFrame("Sample GUI Components");
      Container display = frame.getContentPane();
      display.setLayout(new FlowLayout());
      display.add(new JLabel("What programming languages
do you know?"));
      display.add(new JLabel("Check all that apply:"));
      display.add(new JCheckBox("Java",true));
      display.add(new JCheckBox("C",false));
      display.add(new JCheckBox("C++",false));
      display.add(new JButton("Submit"));
      frame.pack();
      frame.setVisible(true);
      System.out.println("The end of main().");
   }
}
```

The Event Thread

- Notice that although main() ends, the program is still running.
- Creating a JFrame, implicitly creates a separate thread of execution that enters an infinite loop, looking for events.
- This is called an event driven program.
- How do we get it to do something when an event occurs?

The Event Model

- Swing components are *event sources* - e.g. clicking a `JButton` creates an `ActionEvent`.
- To respond to an event you create an *event listener*, and tell the event *source* about the *event listener*.
- Event sources have methods like `addSomethingListener()`, where Something is a particular event type.

```
import javax.swing.*;
import java.awt.*;
class SampleAWT2 {
   public static void main(String[] args) {
      JFrame frame = new JFrame("Sample GUI Components");
      Container display = frame.getContentPane();
      // some lines deleted - same as before
      display.add(new JCheckBox("C++",false));
      JButton submit = new JButton("Submit");
      display.add(submit);
      submit.addActionListener(new Submit());
      frame.pack();
      frame.setVisible(true);
      System.out.println("The end of main().");
   }
}
```

```
import java.awt.event.*;

class Submit implements ActionListener {
   public void actionPerformed(ActionEvent e) {
      System.out.println("Submitted");
   }
}
```

```
import javax.swing.*;
import java.awt.*;
class SampleAWT2 {
   public static void main(String[] args) {
      JFrame frame = new JFrame("Sample GUI Components");
      Container display = frame.getContentPane();
      // some lines deleted - same as before
      display.add(new JCheckBox("C++",false));
      JButton submit;
      display.add(submit = new JButton("Submit"));
      submit.addActionListener(new Submit());
      frame.addWindowListener(new Exiter());
      frame.pack();
      frame.show();
      System.out.println("The end of main().");
  }
}
```

Listening to a `JButton`

- Create the button with new `JButton()`
- Get the `Container` for the `JFrame` using `getContentPane()`.
- Add the button to the content pane using `add()`.
- Create an `ActionEventListener` by
  - adding implements `ActionEventListener` to the class declaration and
  - defining an `actionPerformed()` method.
- Add the listener object to the list of listeners for the button by calling `button.addActionListener(listener)`

```
//TextInput.java
import java.awt.*;
import javax.swing.*;

class TextInput {
   public static void main(String[] args) {
      JFrame frame = new JFrame("TextInput");
      Container pane = frame.getContentPane();
      JTextField input = new JTextField(
                        "Edit this text then hit <return>");
      Echo listener = new Echo();
      input.addActionListener(listener);
      pane.add(input);
      frame.pack();
      frame.show();
   }
}
```

```
//Echo.java

import javax.swing.*;
import java.awt.event.*;

class Echo implements ActionListener {
   public void actionPerformed(ActionEvent e) {
      JTextField source = (JTextField)e.getSource();
      String text = source.getText();
      System.out.println(text);
   }
}
```

What does hitting <return> do here?

```
//TextInput.java
import java.awt.*;
import javax.swing.*;

class TextInput {
   public static void main(String[] args) {
      JFrame frame = new JFrame("TextInput");
      Container pane = frame.getContentPane();
      JTextField input = new
         JTextField("Edit this text then hit <return>");
      Echo listener = new Echo();
      input.addActionListener(listener);
      input.addActionListener(new Submit());
      pane.add(input);
      frame.pack();
      frame.show();
   }
}
```