More About Objects and Methods

Chapter 5
Reminders

• Project 3 released: due Sept 29 @ 10:30 pm
  - No Late Submissions
  - 2 weeks b/c of exam
  - turnin now enabled

• Follow the newsgroup for project information and questions – newsgroup postings from GTAs are official. Even if you don’t follow the newsgroup you are responsible for all updates/changes to any projects mentioned there.
Reminders 2

- Make sure to match given output exactly in your projects. **You will lose points if your output does not match exactly.** We give you sample inputs and outputs for this reason.

- Project submissions use your **lab section**, not your recitation section (this is clearly written on all project outlines)

```
  turnin -c cs180secXXXX -p project1 ...
```
Reminders 3 – Exam 1

• Exam will be returned in recitation next week
• Solution will be discussed in class today
• Scores posted by Monday on WebCT
The **null** Constant

- When the compiler requires an object reference to be initialized, set it to **null**.
  
  ```java
  String line = null;
  ```

- **null** is not an object, but is instead a constant that indicates that an object variable references no object.

- **==** and **!=** (rather than method `equals`) are used to determine if an object variable has the value **null**.
The null Constant, cont.

• An object reference initialized to null cannot be used to invoke methods in the object’s class
  – An attempt to do so results in a null pointer exception.
Integer, Double, and Other Wrapper Classes

• Sometimes a primitive value needs to be passed in as an argument, but the method definition creates an object as the corresponding formal parameter.

• Java’s wrapper classes convert a value of a primitive type to a corresponding class type.

```java
Integer n = new Integer(42);
```

– The instance variable of the object `n` has the value 42.
### Integer, Double, and Other Wrapper Classes, cont.

- **To retrieve the integer value**
  
  ```java
  int i = n.intValue();
  ```

- **type** | **class** | **method**
  - int | Integer | intValue
  - long | Long | longValue
  - float | Float | floatValue
  - double | Double | doubleValue
  - char | Character | charValue
Shorthand in Java 5.0

• Wrapping is done automatically in Java 5.0.
  
  ```java
  Integer n = 42;
  
  which is equivalent to
  
  Integer n = new Integer(42);
  ```

• Similarly
  
  ```java
  int i = n;
  
  is equivalent to
  
  int i = n.intValue;
  ```
Automatic Boxing and Unboxing

- Converting a value of a primitive type to an object of its corresponding wrapper class is called **boxing**.

  ```java
  Integer n = new Integer(42);
  ```

- Java 5.0 boxes automatically.

  ```java
  Integer n = 42;
  ```
Automatic Boxing and Unboxing, cont.

• Converting an object of a wrapper class to a value of the corresponding primitive type is called *unboxing*.
  
  ```java
  int i = n.intValue;
  ```

• Java 5.0 unboxes automatically.
  
  ```java
  int i = n;
  ```
Type Conversions

- Static methods in the wrapper classes can be used to convert a String to the corresponding number of type int, long, float, or double.

```java
String theString = "199.98";
double doubleSample =
    Double.parseDouble(theString);

or

    Double.parseDouble(theString.trim());
```

if the string has leading or trailing whitespace.
Type Conversions, cont.

• Methods for converting strings to the corresponding numbers
  
  `Integer.parseInt("42")`
  `Long.parseLong("42")`
  `Float.parseFloat("199.98")`
  `Double.parseDouble("199.98")`
Type Conversions, cont.

- Methods for converting numbers to the corresponding strings
  - `Integer.toString(42)`
  - `Long.toString(42)`
  - `Float.toString(199.98)`
  - `Double.toString(199.98)`
Overloading

• We’ve seen that different classes can have methods with the same names.
• Two or more methods in the same class class can be defined with the same name if the parameter list can be used to determine which method is being invoked.
  – This useful ability is called overloading.
Overloading, cont.

- The number of arguments and the types of the arguments determines which method `average` is invoked.
  - If there is no match, Java attempts simple type conversions of the kinds discussed earlier.
  - If there is still no match, an error message is produced.
Overloading and Automatic Type Conversion, cont.

• third example

```java
public static void oops (double n1, int n2);
...
public static void oops (int n1, double n2);
– This will compile, but the invocation
  sample.oops(5,10)
will produce an error message.
```
Overloading and the Return Type

• You cannot overload a method name by providing two definitions with headings that differ only in the return type.
Constructors

- When you create an object of a class, often you want certain initializing actions performed such as giving values to the instance variables.
- A *constructor* is a special method that performs initializations.
Defining Constructors

• New objects are created using

\[
\text{Class\_Name Object\_Name} = \text{new Class\_Name (Parameter(s))};
\]

• A constructor is called automatically when a new object is created.

  - \text{Class\_Name (Parameter(s)) calls the constructor and returns a reference.}

  - It performs any actions written into its definition including initializing the values of (usually all) instance variables.
Defining Constructors, cont.

• Each constructor has the same name as its class.
• A constructor does not have a return type, not even `void`.
• Constructors often are overloaded, each with a different number of parameters or different types of parameters.
• Typically, at least one constructor, the `default constructor`, has no parameters.
/**
 * Class for basic pet records: name, age, and weight.
 */
public class PetRecord
{
    private String name;
    private int age; // in years
    private double weight; // in pounds
    public void writeOutput()
    {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age + " years");
        System.out.println("Weight: " + weight + " pounds");
    }
    public PetRecord(String initialValue, int initialAge,
                     double initialWeight)
    {
        name = initialValue;
        if ((initialAge < 0) || (initialWeight < 0))
        {
            System.out.println("Error: Negative age or weight.");
            System.exit(0);
        }
        else
        {
            age = initialAge;
            weight = initialWeight;
        }
    }
    public void set(String newName)
    {
        name = newName; // age and weight are unchanged.
    }
    public PetRecord(int initialAge)
    {
        name = "No name yet.";
        weight = 0;
        if (initialAge < 0)
        {
            System.out.println("Error: Negative age.");
            System.exit(0);
        }
    }
    public void set(String newName, int newAge, double newWeight)
    {
        name = newName;
        if ((newAge < 0) || (newWeight < 0))
        {
            System.out.println("Error: Negative age or weight.");
            System.exit(0);
        }
    }
}

Display 5.20
PetRecord Class with Constructors
else
    age = initialAge;
}

public void set(int newAge)
{
    if (newAge < 0)
    {
        System.out.println("Error: Negative age.");
        System.exit(0);
    }
    else
        age = newAge;
    //name and weight are unchanged.
}

public PetRecord(double initialWeight)
{
    name = "No name yet";
    age = 0;
    if (initialWeight < 0)
    {
        System.out.println("Error: Negative weight.");
        System.exit(0);
    }
    else
        weight = initialWeight;
}

public void set(double newWeight)
{
    if (newWeight < 0)
    {
        System.out.println("Error: Negative weight.");
        System.exit(0);
    }
    else
        weight = newWeight; //name and age are unchanged.
}

public PetRecord()
{ /* Default constructor */
    name = "No name yet";
    age = 0;
    weight = 0;
}

public String getName()
{ /* Method */
    return name;
}

public int getAge()
{ /* Method */
    return age;
}

public double getWeight()
{ /* Method */
    return weight;
}

Display 5.20
PetRecord Class with Constructors

Chapter 5
Defining Constructors, cont.

• When a class definition does not have a constructor definition, Java creates a default constructor automatically.

• Once you define at least one constructor for the class, no additional constructor is created automatically.
Using Constructors

• A constructor can be called only when you create a new object.
  ```java
  newborn.PetRecord(“Fang”. 1, 150.0);
  // invalid
  ```

• After an object is created, a `set` method is needed to change the value(s) of one or more instance variables.
  ```java
  newBorn.set(“Fang”, 1, 150.0); // valid
  ```
Using Other Methods in a Constructor

• Other methods in the same class can be used in the definition of a constructor.
• Calls to one or more set methods are common.

```java
public Class_Name(parameter(s));
{
    set(...)
}
```
(optional) Graphics Supplement: Outline

• Adding Buttons
• Event-Driven Programming
• Programming Buttons
• Icons
• Changing Visibility
Adding Buttons

• A component in an applet that looks like a push button and can do something when it is clicked is referred to as a *button*.
• Buttons are added to applets the same way labels are added.
• But, unlike labels, actions can be associated with buttons.
Creating Buttons

• example

```java
JButton sunnyButton =
    new JButton("Sunny");
```

• (Until buttons are programmed, they do not perform any action besides depressing and returning to their undepressed state.)
import javax.swing.*;
import java.awt.*;

/**
 * Simple demonstration of putting buttons in an Applet.
 * These buttons do not do anything. That comes in a later version.
 */
public class PreliminaryButtonDemo extends JApplet
{

    public void init()
    {
        Container contentPane = getContentPane();
        contentPane.setBackground(Color.WHITE);
        contentPane.setLayout(new FlowLayout());

        JButton sunnyButton = new JButton("Sunny");
        contentPane.add(sunnyButton);

        JButton cloudyButton = new JButton("Cloudy");
        contentPane.add(cloudyButton);
    }
}
Event-Driven Programming

- Applets use events and listeners.
- An event is an object that represents some action such as clicking a mouse button.
- An object *fires* (or generates) an event.
- An object that can fire an event can have one or more *listener objects*, specified by the programmer.
Event-Driven Programming, cont.

• A listener object can have methods called event handlers, defined by the programmer, that specify what happens when events are sent to the listener.
  – “sent” means that some method is invoked automatically with the event object as an argument.
Event-Driven Programming, cont.

This event object is the result of a button click. The event object goes from the button to the listener.

This listener object performs some action, such as making text visible in the applet, when it receives the event object.

Display 5.27
Event Firing and an Event Listener
Event-Driven Programming, cont.

- Events determine the order in which things happen.
  - The “next” thing to happen is determined by the next event.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

/**
 * Simple demonstration of putting buttons in an Applet.
 * These buttons do something when clicked.
 */
public class ButtonDemo extends JApplet implements ActionListener
{
    public void init()
    {
        Container contentPane = getContentPane();
        contentPane.setBackground(Color.WHITE);
        contentPane.setLayout(new FlowLayout());

        JButton sunnyButton = new JButton("Sunny");
        contentPane.add(sunnyButton);
        sunnyButton.addActionListener(this);

        JButton cloudyButton = new JButton("Cloudy");
        contentPane.add(cloudyButton);
        cloudyButton.addActionListener(this);
    }

    public void actionPerformed(ActionEvent e)
    {
        Container contentPane = getContentPane();

        if (e.getActionCommand().equals("Sunny"))
            contentPane.setBackground(Color.BLUE);
        else if (e.getActionCommand().equals("Cloudy"))
            contentPane.setBackground(Color.GRAY);
        else
            System.out.println("Error in button interface.");
    }

Display 5.28
Adding Actions to the Buttons
Event-Driven Programming, cont.

Resulting Applet

Resulting Applet after Clicking Cloudy

Resulting Applet after Clicking Sunny

Display 5.28
Adding Actions to the Buttons
Programming Buttons

• The applet class definition needs to know two things:
  – for each button, which objects are listeners (called registering the listener)
  – the defined method(s) to be invoked when the event is sent to the listener
Programming Buttons, cont.

- registering the listener
  
  ```java
  sunnyButton.addActionListener(this);
  ```

  - The class `ButtonDemo` itself is the listener class.
Listeners

- Different kinds of components require different kinds of listener classes.
- Buttons generate *action events* which are handled by *action listeners*.
- An action listener is an object of type `ActionListener`.
  - `ActionListener` is not a class, but is instead an interface which we will discuss in Chapter 7.
Listeners, cont.

– Add the phrase `implements ActionListener` to the beginning of the class definition
– Define a method named `actionPerformed`.
– Often it is convenient to put the method `actionPerformed` in the applet it is intended to change.
– Because the applet itself is the action listener, the action event goes to the applet’s object, where it is passed automatically to the method `actionPerformed`.
Listeners, cont.

- Method `actionPerformed` typically needs to know the source of the action event.

```
e.getActionCommand()
```

returns the string written on the button which can be used to determine the source.

- An import statement is needed to define an action listener class.

```
import java.awt.event.*;
```
Applets and Constructors

- Applets normally have no constructors.
- Method `init` is used for any needed initialization.
Icons

• An *icon* typically is a small picture.
• Pictures can be produced in several formats for display on the screen (e.g. **GIF** and **JPEG**).
• These pictures can be used as the basis for an icon.
• A label or a button can include a string and/or an icon.
Icons, cont.

- Example
  
  ```java
  ImageIcon dukeIcon = 
  new ImageIcon("duke_waving.gif");
  niceLabel.setIcon(dukeIcon);
  ```
Icons, cont.

class IconDemo

```java
import javax.swing.*;

public class IconDemo extends JApplet
{
    public void init()
    {
        JLabel niceLabel = new JLabel("Java is fun!");
        ImageIcon dukeIcon = new ImageIcon("duke_waving.gif");
        niceLabel.setIcon(dukeIcon);
        getContentPane().add(niceLabel);
    }
}
```

Resulting Applet

![Applet Viewer: IconDemo...](image)

An Applet with an Icon Picture
• Buttons can have icons.

```java
ImageIcon smileyFaceIcon =
    new ImageIcon(“smiley.gif”);
sunnyButton.setIcon(smileyFaceIcon);
```
Icons, cont.

Resulting Applet

The code for this applet is in the file ButtonIconDemo.java in the source code on the Web.

Display 5.31
A Button with an Icon
Changing Visibility

• Labels, buttons, and other components have a method `setVisibility` to make the component visible or invisible.
  – If the argument is `true`, the component is visible.
  – If the argument is `false`, the component is invisible (and no longer present).
Summary

- You have learned more techniques for programming with classes and objects.
- You have learned about static methods and static variables.
- You have learned to define constructor methods.
- You have learned about packages and import statements.
Summary, cont.

• You have learned about the top-down design technique.
• You have learned techniques for testing methods (including the use of stub methods and driver programs).
• (optional) You have learned to add buttons and icons to applets.
• You have learned the basics of event driven programming.