ICOM 4015: Advanced Programming

Lecture 9

Reading: Chapter Nine: Interphases and Polymorphism
Chapter 9 – Interfaces and Polymorphism
Chapter Goals

• To be able to declare and use interface types
• To understand the concept of polymorphism
• To appreciate how interfaces can be used to decouple classes
• To learn how to implement helper classes as inner classes

G To implement event listeners in graphical applications
Using Interfaces for Algorithm Reuse

• Use *interface types* to make code more reusable

• In Chapter 6, we created a `DataSet` to find the average and maximum of a set of `numbers`

• What if we want to find the average and maximum of a set of `BankAccount` values?
public class DataSet // Modified for BankAccount objects {
    private double sum;
    private BankAccount maximum;
    private int count;
    ...
    public void add(BankAccount x) {
        sum = sum + x.getBalance();
        if (count == 0 || maximum.getBalance() < x.getBalance())
            maximum = x;
        count++;
    }

    public BankAccount getMaximum() {
        return maximum;
    }
}

Using Interfaces for Algorithm Reuse
Using Interfaces for Algorithm Reuse

Or suppose we wanted to find the coin with the highest value among a set of coins. We would need to modify the `DataSet` class again:

```java
public class DataSet // Modified for Coin objects
{
    private double sum;
    private Coin maximum;
    private int count;
    ...
    public void add(Coin x)
    {
        sum = sum + x.getValue();
        if (count == 0 || maximum.getValue() < x.getValue()) maximum = x;
        count++;
    }
```
Using Interfaces for Algorithm Reuse

```java
public Coin getMaxCoin() {
    return maximum;
}
```
Using Interfaces for Algorithm Reuse

• The algorithm for the data analysis service is the same in all cases; details of measurement differ

• Classes could agree on a method `getMeasure` that obtains the measure to be used in the analysis

• We can implement a single reusable `DataSet` class whose `add` method looks like this:

```java
sum = sum + x.getMeasure();
if (count == 0 || maximum.getMeasure() < x.getMeasure())
    maximum = x;
count++;
```
Using Interfaces for Algorithm Reuse

- What is the type of the variable $x$?
  - $x$ should refer to any class that has a `getMeasure` method

- In Java, an **interface type** is used to specify required operations:

  ```java
  public interface Measurable
  {
      double getMeasure();
  }
  ```

- Interface declaration lists all methods that the interface type requires
Syntax 9.1 Declaring an Interface

Syntax

```java
public interface InterfaceName
{
    method signatures
}
```

Example

```java
public interface Measurable
{
    double getMeasure();
}
```

The methods of an interface are automatically public. No implementation is provided.
Interfaces vs. Classes

An interface type is similar to a class, but there are several important differences:

- All methods in an interface type are abstract; they don’t have an implementation
- All methods in an interface type are automatically public
- An interface type does not have instance fields
Generic `DataSet` for Measurable Objects

```java
public class DataSet {
    private double sum;
    private Measurable maximum;
    private int count;
    ...
    public void add(Measurable x) {
        sum = sum + x.getMeasure();
        if (count == 0 || maximum.getMeasure() < x.getMeasure()) {
            maximum = x;
            count++;
        }
    }

    public Measurable getMaximum() {
        return maximum;
    }
}
```
Implementing an Interface Type

• Use `implements` reserved word to indicate that a class implements an interface type:

```java
public class BankAccount implements Measurable {
    public double getMeasure() {
        ...
        return balance;
    }
}
```

• A class can implement more than one interface type
  • `Class must declare all the methods that are required by all the interfaces it implements`
Implementing an Interface Type

• Another example:

```java
public class Coin implements Measurable {
    public double getMeasure()
    {
        return value;
    }
    ...
}
```
Code Reuse

- A service type such as DataSet specifies an interface for participating in the service
- Use interface types to make code more reusable

Figure 1
Attachments Conform to the Mixer’s Interface
**Syntax 9.2 Implementing an Interface**

**Syntax**

```java
public class ClassName implements InterfaceName, InterfaceName, ...
{
    instance variables
    methods
}
```

**Example**

```java
public class BankAccount implements Measurable
{
    ... public double getMeasure()
    {
        return balance;
    }
    ...
}
```

*List all interface types that this class implements.*

*This method provides the implementation for the method declared in the interface.*
Interfaces can reduce the coupling between classes

UML notation:

- Interfaces are tagged with a “stereotype” indicator «interface»
- A dotted arrow with a triangular tip denotes the “is-a” relationship between a class and an interface
- A dotted line with an open v-shaped arrow tip denotes the “uses” relationship or dependency

Note that **DataSet** is *decoupled* from **BankAccount** and **Coin**
This program tests the `DataSet` class.

```java
public class DataSetTester {
    public static void main(String[] args) {
        DataSet bankData = new DataSet();

        bankData.add(new BankAccount(0));
        bankData.add(new BankAccount(10000));
        bankData.add(new BankAccount(2000));

        System.out.println("Average balance: "+ bankData.getAverage());
        System.out.println("Expected: 4000");
        Measurable max = bankData.getMaximum();
        System.out.println("Highest balance: "+ max.getMeasure());
        System.out.println("Expected: 10000");

        DataSet coinData = new DataSet();
    }
}
```

Continued
coinData.add(new Coin(0.25, "quarter"));
coinData.add(new Coin(0.1, "dime"));
coinData.add(new Coin(0.05, "nickel"));

System.out.println("Average coin value: " + coinData.getAverage());
System.out.println("Expected: 0.133");
max = coinData.getMaximum();
System.out.println("Highest coin value: " + max.getMeasure());
System.out.println("Expected: 0.25");
}
Program Run:

Average balance: 4000.0
Expected: 4000
Highest balance: 10000.0
Expected: 10000
Average coin value: 0.13333333333333333
Expected: 0.133
Highest coin value: 0.25
Expected: 0.25
Suppose you want to use the `DataSet` class to find the `Country` object with the largest population. What condition must the `Country` class fulfill?

**Answer:** It must implement the `Measurable` interface, and its `getMeasure` method must return the population.
Self Check 9.2

Why can’t the `add` method of the `DataSet` class have a parameter of type `Object`?

**Answer:** The `Object` class doesn’t have a `getMeasure` method, and the `add` method invokes the `getMeasure` method.
Converting Between Class and Interface Types

• You can convert from a class type to an interface type, provided the class implements the interface

  BankAccount account = new BankAccount(10000);
  Measurable x = account; // OK

  Coin dime = new Coin(0.1, "dime");
  Measurable x = dime; // Also OK

• Cannot convert between unrelated types:

  Measurable x = new Rectangle(5, 10, 20, 30); // ERROR

Because Rectangle doesn’t implement Measurable
Variables of Class and Interface Types

Figure 3  Variables of Class and Interface Types
Casts

• Add **Coin** objects to **DataSet**:

```java
dataSet coinData = new DataSet();
coinData.add(new Coin(0.25, "quarter"));
coinData.add(new Coin(0.1, "dime"));
coinData.add(new Coin(0.05, "nickel"));
Measurable max = coinData.getMaximum(); // Get the largest coin
```

• What can you do with max? It’s not of type **Coin**:

```java
String name = max.getName(); // ERROR
```

• You need a cast to convert from an interface type to a class type

• You know it’s a **Coin**, but the compiler doesn’t. Apply a cast:

```java
Coin maxCoin = (Coin) max;
String name = maxCoin.getName();
```
Casts

• If you are wrong and `max` isn’t a coin, the program throws an exception and terminates

• Difference with casting numbers:
  • *When casting number types you agree to the information loss*
  • *When casting object types you agree to that risk of causing an exception*
Self Check 9.3

Can you use a cast \((\text{BankAccount}) \, x\) to convert a \texttt{Measurable} variable \(x\) to a \texttt{BankAccount} reference?

\textbf{Answer:} Only if \(x\) actually refers to a \texttt{BankAccount} object.
Self Check 9.4

If both BankAccount and Coin implement the Measurable interface, can a Coin reference be converted to a BankAccount reference?

Answer: No — a Coin reference can be converted to a Measurable reference, but if you attempt to cast that reference to a BankAccount, an exception occurs.
An interface variable holds a reference to object of a class that implements the interface:

```java
Measurable meas;
meas = new BankAccount(10000);
meas = new Coin(0.1, "dime");
```

Note that the object to which `meas` refers doesn’t have type `Measurable`; the type of the object is some class that implements the `Measurable` interface

You can call any of the interface methods:

```java
double m = meas.getMeasure();
```

Which method is called?
Interface Reference

**Figure 4** An Interface Reference Can Refer to an Object of Any Class that Implements the Interface
Polymorphism

- When the virtual machine calls an instance method, it locates the method of the implicit parameter's class — called *dynamic method lookup*

- If `meas` refers to a `BankAccount` object, then `meas.getMeasure()` calls the `BankAccount.getMeasure` method

- If `meas` refers to a `Coin` object, then method `Coin.getMeasure` is called

- Polymorphism (many shapes) denotes the ability to treat objects with differences in behavior in a uniform way
Animation 9.1: Polymorphism
Why is it impossible to construct a `Measurable` object?

**Answer:** `Measurable` is an interface. Interfaces have no fields and no method implementations.
Self Check 9.6

Why can you nevertheless declare a variable whose type is `Measurable`?

**Answer:** That variable never refers to a `Measurable` object. It refers to an object of some class — a class that implements the `Measurable` interface.
What does this code fragment print? Why is this an example of polymorphism?

```java
DataSet data = new DataSet();
data.add(new BankAccount(1000));
data.add(new Coin(0.1, "dime"));
System.out.println(data.getAverage());
```

**Answer:** The code fragment prints 500.05. Each call to `add` results in a call `x.getMeasure()`. In the first call, `x` is a `BankAccount`. In the second call, `x` is a `Coin`. A different `getMeasure` method is called in each case. The first call returns the account balance, the second one the coin value.
Using Interfaces for Callbacks

• Limitations of **Measurable** interface:
  
  • *Can add* **Measurable** interface only to classes under your control
  
  • *Can measure an object in only one way*
  
  • *E.g., cannot analyze a set of savings accounts both by bank balance and by interest rate*

• **Callback**: a mechanism for specifying code that is executed at a later time

• In previous **DataSet** implementation, responsibility of measuring lies with the added objects themselves
Using Interfaces for Callbacks

- Alternative: Hand the object to be measured to a method of an interface:

```java
public interface Measurer
{
    double measure(Object anObject);
}
```

- Object is the “lowest common denominator” of all classes
Using Interfaces for Callbacks

• The code that makes the call to the callback receives an object of class that implements this interface:

```java
public DataSet(Measurer aMeasurer)
{
    sum = 0;
    count = 0;
    maximum = null;
    measurer = aMeasurer; // Measurer instance variable
}
```

• The measurer instance variable carries out the measurements:

```java
public void add(Object x)
{
    sum = sum + measurer.measure(x);
    if (count == 0 || measurer.measure(maximum) < measurer.measure(x))
        maximum = x;
    count++;
}
```
Using Interfaces for Callbacks

• A specific callback is obtained by implementing the Measurer interface:

```java
public class RectangleMeasurer implements Measurer {
    public double measure(Object anObject) {
        Rectangle aRectangle = (Rectangle) anObject;
        double area = aRectangle.getWidth() * aRectangle.getHeight();
        return area;
    }
}
```

• Must cast from `Object` to `Rectangle`:

```java
Rectangle aRectangle = (Rectangle) anObject;
```
Using Interfaces for Callbacks

• Pass measurer to data set constructor:

```java
Measurer m = new RectangleMeasurer();
DataSet data = new DataSet(m);
data.add(new Rectangle(5, 10, 20, 30));
data.add(new Rectangle(10, 20, 30, 40));
...
```
Note that the Rectangle class is decoupled from the Measurer interface.
/**
   * Describes any class whose objects can measure other objects.
   */

public interface Measurer
{
    /**
     * Computes the measure of an object.
     * @param anObject the object to be measured
     * @return the measure
     */
    double measure(Object anObject);
}
import java.awt.Rectangle;

/**
   * Objects of this class measure rectangles by area.
   */

public class RectangleMeasurer implements Measurer
{
    public double measure(Object anObject)
    {
        Rectangle aRectangle = (Rectangle) anObject;
        double area = aRectangle.getWidth() * aRectangle.getHeight();
        return area;
    }
}
/**
 * Computes the average of a set of data values.
 */

public class DataSet
{
    private double sum;
    private Object maximum;
    private int count;
    private Measurer measurer;

    /**
     * Constructs an empty data set with a given measurer.
     * @param aMeasurer the measurer that is used to measure data values
     */
    public DataSet(Measurer aMeasurer)
    {
        sum = 0;
        count = 0;
        maximum = null;
        measurer = aMeasurer;
    }
}
/**
   * Adds a data value to the data set.
   * @param x a data value
   */
  public void add(Object x)
  {
      sum = sum + measurer.measure(x);
      if (count == 0 || measurer.measure(maximum) < measurer.measure(x))
          maximum = x;
      count++;
  }

  /**
   * Gets the average of the added data.
   * @return the average or 0 if no data has been added
   */
  public double getAverage()
  {
      if (count == 0) return 0;
      else return sum / count;
  }
/**
   * Gets the largest of the added data.
   * @return the maximum or 0 if no data has been added
   */

public Object getMaximum()
{
    return maximum;
}
}
import java.awt.Rectangle;

/**
 * This program demonstrates the use of a Measurer.
 */
public class DataSetTester2 {
    public static void main(String[] args) {
        Measurer m = new RectangleMeasurer();

        DataSet data = new DataSet(m);

        data.add(new Rectangle(5, 10, 20, 30));
        data.add(new Rectangle(10, 20, 30, 40));
        data.add(new Rectangle(20, 30, 5, 15));

        System.out.println("Average area: " + data.getAverage());
        System.out.println("Expected: 625");
    }
}
Rectangle max = (Rectangle) data.getMaximum();
System.out.println("Maximum area rectangle: " + max);
System.out.println("Expected: "
+ "java.awt.Rectangle[x=10,y=20,width=30,height=40]");
}
}

Program Run:
Average area: 625
Expected: 625
Maximum area rectangle: java.awt.Rectangle[x=10,y=20,width=30,height=40]
Expected: java.awt.Rectangle[x=10,y=20,width=30,height=40]
Self Check 9.8

Suppose you want to use the `DataSet` class of Section 9.1 to find the longest `String` from a set of inputs. Why can’t this work?

**Answer:** The `String` class doesn’t implement the `Measurable` interface.
Self Check 9.9

How can you use the Dataset class of this section to find the longest String from a set of inputs?

**Answer:** Implement a class `StringMeasurer` that implements the `Measurer` interface.
Self Check 9.10

Why does the `measure` method of the `Measurer` interface have one more parameter than the `getMeasure` method of the `Measurable` interface?

**Answer:** A measurer measures an object, whereas `getMeasure` measures “itself”, that is, the implicit parameter.
Inner Classes

- Trivial class can be declared inside a method:

```java
public class DataSetTester3
{
    public static void main(String[] args)
    {
        class RectangleMeasurer implements Measurer
        {
            ...
        }
        Measurer m = new RectangleMeasurer();
        DataSet data = new DataSet(m);
        ...
    }
}
```
Inner Classes

- If inner class is declared inside an enclosing class, but outside its methods, it is available to all methods of enclosing class:

```java
public class DataSetTester3
{
    class RectangleMeasurer implements Measurer
    {
        ...
    }

    public static void main(String[] args)
    {
        Measurer m = new RectangleMeasurer();
        DataSet data = new DataSet(m);
        ...
    }
}
```
Inner Classes

• Compiler turns an inner class into a regular class file:

    DataSetTester$1$RectangleMeasurer.class
import java.awt.Rectangle;

/**
   * This program demonstrates the use of an inner class.
   */

public class DataSetTester3
{
    public static void main(String[] args)
    {
        class RectangleMeasurer implements Measurer
        {
            public double measure(Object anObject)
            {
                Rectangle aRectangle = (Rectangle) anObject;
                double area
                = aRectangle.getWidth() * aRectangle.getHeight();
                return area;
            }
        }

        Measurer m = new RectangleMeasurer();

        DataSet data = new DataSet(m);
    }
}
data.add(new Rectangle(5, 10, 20, 30));
data.add(new Rectangle(10, 20, 30, 40));
data.add(new Rectangle(20, 30, 5, 15));

System.out.println("Average area: " + data.getAverage());
System.out.println("Expected: 625");

Rectangle max = (Rectangle) data.getMaximum();
System.out.println("Maximum area rectangle: " + max);
System.out.println("Expected: "+ "java.awt.Rectangle[x=10,y=20,width=30,height=40]" acompanado de un texto en castellano para ilustrar el uso de Rectangle.
Self Check 9.11

Why would you use an inner class instead of a regular class?

**Answer:** Inner classes are convenient for insignificant classes. Also, their methods can access variables and fields from the surrounding scope.
Self Check 9.12

How many class files are produced when you compile the `DataSetTester3` program?

**Answer:** Four: one for the outer class, one for the inner class, and two for the `DataSet` and `Measurer` classes.
Operating Systems

A Graphical Software Environment for the Linux Operating System
Mock Objects

• Want to test a class before the entire program has been completed
• A mock object provides the same services as another object, but in a simplified manner
• Example: a grade book application, GradingProgram, manages quiz scores using class GradeBook with methods:
  
  public void addScore(int studentId, double score)
  public double getAverageScore(int studentId)
  public void save(String filename)

• Want to test GradingProgram without having a fully functional GradeBook class
Mock Objects

• Declare an interface type with the same methods that the GradeBook class provides
  • *Convention: use the letter I as a prefix for the interface name:*
    ```java
    public interface IGradeBook {
        void addScore(int studentId, double score);
        double getAverageScore(int studentId);
        void save(String filename);
        ...
    }
    ```
  
• The GradingProgram class should *only* use this interface, never the GradeBook class which implements this interface
• Meanwhile, provide a simplified mock implementation, restricted to the case of one student and without saving functionality:

```java
public class MockGradeBook implements IGradeBook {
    private ArrayList<Double> scores;
    public void addScore(int studentId, double score) {
        // Ignore studentId
        scores.add(score);
    }
    double getAverageScore(int studentId) {
        double total = 0;
        for (double x : scores) {
            total = total + x;
        }
        return total / scores.size();
    }
    void save(String filename) {
        // Do nothing
    }
    // ...
}
```
Mock Objects

• Now construct an instance of `MockGradeBook` and use it immediately to test the `GradingProgram` class
• When you are ready to test the actual class, simply use a `GradeBook` instance instead
• Don’t erase the mock class — it will still come in handy for regression testing
Self Check 9.13

Why is it necessary that the real class and the mock class implement the same interface type?

**Answer:** You want to implement the `GradingProgram` class in terms of that interface so that it doesn’t have to change when you switch between the mock class and the actual class.
Self Check 9.14

Why is the technique of mock objects particularly effective when the `GradeBook` and `GradingProgram` class are developed by two programmers?

**Answer:** Because the developer of `GradingProgram` doesn’t have to wait for the `GradeBook` class to be complete.
Events, Event Sources, and Event Listeners

- User interface events include key presses, mouse moves, button clicks, and so on
- Most programs don’t want to be flooded by boring events
- A program can indicate that it only cares about certain specific events
Events, Event Sources, and Event Listeners

- **Event listener:**
  - Notified when event happens
  - Belongs to a class that is provided by the application programmer
  - Its methods describe the actions to be taken when an event occurs
  - A program indicates which events it needs to receive by installing event listener objects

- **Event source:**
  - User interface component that generates a particular event
  - Add an event listener object to the appropriate event source
  - When an event occurs, the event source notifies all event listeners
Events, Event Sources, and Event Listeners

• Example: A program that prints a message whenever a button is clicked:

```
~$ cd BigJava/ch09/button1
~/BigJava/ch09/button1$ javac ButtonViewer.java
~/BigJava/ch09/button1$ java ButtonViewer
I was clicked.
I was clicked.
I was clicked.
```

*Figure 6* Implementing an Action Listener
Events, Event Sources, and Event Listeners

• Use JButton components for buttons; attach an ActionListener to each button

• ActionListener interface:
  public interface ActionListener
  {
    void actionPerformed(ActionEvent event);
  }

• Need to supply a class whose actionPerformed method contains instructions to be executed when button is clicked

• event parameter contains details about the event, such as the time at which it occurred
Events, Event Sources, and Event Listeners

• Construct an object of the listener and add it to the button:

```java
ActionListener listener = new ClickListener();
button.addActionListener(listener);
```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

/**
 * An action listener that prints a message.
 */
public class ClickListener implements ActionListener {
    public void actionPerformed(ActionEvent event) {
        System.out.println("I was clicked.");
    }
}
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;

/**
   * This program demonstrates how to install an action listener.
   */
public class ButtonViewer {
    private static final int FRAME_WIDTH = 100;
    private static final int FRAME_HEIGHT = 60;

    public static void main(String[] args) {
        JFrame frame = new JFrame();
        JButton button = new JButton("Click me!");
        frame.add(button);
        ActionListener listener = new ClickListener();
        button.addActionListener(listener);
    }

Continued
frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setVisible(true);
Self Check 9.15

Which objects are the event source and the event listener in the ButtonViewer program?

**Answer:** The button object is the event source. The listener object is the event listener.
Self Check 9.16

Why is it legal to assign a `ClickListener` object to a variable of type `ActionListener`?

**Answer:** The `ClickListener` class implements the `ActionListener` interface.
Using Inner Classes for Listeners

• Implement simple listener classes as inner classes like this:

```java
JButton button = new JButton("...");
// This inner class is declared in the same method as the
// button variable
class MyListener implements ActionListener
{
    ...

};
ActionListener listener = new MyListener();
button.addActionListener(listener);
```

• This places the trivial listener class exactly where it is needed, without cluttering up the remainder of the project
Using Inner Classes for Listeners

• Methods of an inner class can access the variables from the enclosing scope
  • *Local variables that are accessed by an inner class method must be declared as final*

• **Example:** Add interest to a bank account whenever a button is clicked:
Using Inner Classes for Listeners

```java
JButton button = new JButton("Add Interest");
final BankAccount account =
    new BankAccount(INITIAL_BALANCE);
// This inner class is declared in the same method as
// the account and button variables.
class AddInterestListenable implements ActionListener
{
    public void actionPerformed(ActionEvent event)
    {
        // The listener method accesses the account
        // variable from the surrounding block
        double interest = account.getBalance()
            * INTEREST_RATE / 100;
        account.deposit(interest);
    }
}
ActionListener listener = new AddInterestListenable();
button.addActionListener(listener);
```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;

/**
   This program demonstrates how an action listener can access
   a variable from a surrounding block.
*/
public class InvestmentViewer1
{
    private static final int FRAME_WIDTH = 120;
    private static final int FRAME_HEIGHT = 60;

    private static final double INTEREST_RATE = 10;
    private static final double INITIAL_BALANCE = 1000;

    public static void main(String[] args)
    {
        JFrame frame = new JFrame();
    
Continued
// The button to trigger the calculation
JButton button = new JButton("Add Interest");
frame.add(button);

// The application adds interest to this bank account
final BankAccount account = new BankAccount(INITIAL_BALANCE);

class AddInterestListener implements ActionListener
{
    public void actionPerformed(ActionEvent event)
    {
        // The listener method accesses the account variable
        // from the surrounding block
        double interest = account.getBalance() * INTEREST_RATE / 100;
        account.deposit(interest);
        System.out.println("balance: " + account.getBalance());
    }
}

Continued
Program Run:

```
balance: 1100.0
balance: 1210.0
balance: 1331.0
balance: 1464.1
```
Self Check 9.17
Why would an inner class method want to access a variable from a surrounding scope?

**Answer:** Direct access is simpler than the alternative — passing the variable as a parameter to a constructor or method.
Self Check 9.18

Why would an inner class method want to access a variable from a surrounding If an inner class accesses a local variable from a surrounding scope, what special rule applies?

**Answer:** The local variable must be declared as `final`.
Building Applications with Buttons

• Example: Investment viewer program; whenever button is clicked, interest is added, and new balance is displayed:

![Screen capture of an application with a button labeled 'Add Interest' and a balance of 1100.0]

Figure 7  An Application with a Button
Building Applications with Buttons

• Construct an object of the JButton class:

```java
JButton button = new JButton("Add Interest");
```

• We need a user interface component that displays a message:

```java
JLabel label = new JLabel("balance: "
    + account.getBalance());
```

• Use a JPanel container to group multiple user interface components together:

```java
JPanel panel = new JPanel();
panel.add(button);
panel.add(label);
frame.add(panel);
```
Building Applications with Buttons

- Listener class adds interest and displays the new balance:

  ```java
  class AddInterestListener implements ActionListener {
      public void actionPerformed(ActionEvent event) {
          double interest = account.getBalance() * INTEREST_RATE / 100;
          account.deposit(interest);
          label.setText("balance=" + account.getBalance());
      }
  }
  ```

- Add `AddInterestListener` as inner class so it can have access to surrounding `final` variables (`account` and `label`).
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;

/**
 * This program displays the growth of an investment.
 */
public class InvestmentViewer2{

    private static final int FRAME_WIDTH = 400;
    private static final int FRAME_HEIGHT = 100;

    private static final double INTEREST_RATE = 10;
    private static final double INITIAL_BALANCE = 1000;

    public static void main(String[] args)
    {
        JFrame frame = new JFrame();
        
        Continued
// The button to trigger the calculation
JButton button = new JButton("Add Interest");

// The application adds interest to this bank account
final BankAccount account = new BankAccount(INITIAL_BALANCE);

// The label for displaying the results
final JLabel label = new JLabel("balance: " + account.getBalance());

// The panel that holds the user interface components
JPanel panel = new JPanel();
panel.add(button);
panel.add(label);
frame.add(panel);
class AddInterestListener implements ActionListener
{
    public void actionPerformed(ActionEvent event)
    {
        double interest = account.getBalance() * INTEREST_RATE / 100;
        account.deposit(interest);
        label.setText("balance: " + account.getBalance());
    }
}

ActionListener listener = new AddInterestListener();
button.addActionListener(listener);

frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setVisible(true);
Self Check 9.19

How do you place the "balance: ..." message to the left of the "Add Interest" button?

Answer: First add label to the panel, then add button.
Self Check 9.20

Why was it not necessary to declare the `button` variable as `final`?

**Answer:** The `actionPerformed` method does not access that variable.
Processing Timer Events

- `javax.swing.Timer` generates equally spaced timer events, sending events to installed action listeners
- Useful whenever you want to have an object updated in regular intervals
Processing Timer Events

• Declare a class that implements the `ActionListener` interface:

```java
class MyListener implements ActionListener {
    void actionPerformed(ActionEvent event) {
        // Listener action (executed at each timer event)
    }
}
```

• Add listener to timer and start timer:

```java
MyListener listener = new MyListener();
Timer t = new Timer(interval, listener);
t.start();
```
Displays a rectangle that can be moved

The `repaint` method causes a component to repaint itself. Call this method whenever you modify the shapes that the `paintComponent` method draws.

```java
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Rectangle;
import javax.swing.JComponent;

/**
 * This component displays a rectangle that can be moved.
 */
public class RectangleComponent extends JComponent {
    private static final int BOX_X = 100;
    private static final int BOX_Y = 100;
    private static final int BOX_WIDTH = 20;
    private static final int BOX_HEIGHT = 30;
}
```

Continued
private Rectangle box;

public RectangleComponent() {
    // The rectangle that the paintComponent method draws
    box = new Rectangle(BOX_X, BOX_Y, BOX_WIDTH, BOX_HEIGHT);
}

drawBox:
  
public void paintComponent(Graphics g) {
    Graphics2D g2 = (Graphics2D) g;
    
    g2.draw(box);
}

Continued
public void moveBy(int dx, int dy)
{
    box.translate(dx, dy);
    repaint();
}

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JFrame;
import javax.swing.Timer;

/**
 * This program moves the rectangle.
 */

public class RectangleMover {

    private static final int FRAME_WIDTH = 300;
    private static final int FRAME_HEIGHT = 400;

    public static void main(String[] args) {
        JFrame frame = new JFrame();

        frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
        frame.setTitle("An animated rectangle");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

    Continued

    Big Java by Cay Horstmann
    Copyright © 2009 by John Wiley & Sons. All rights reserved.
final RectangleComponent component = new RectangleComponent();
frame.add(component);

frame.setVisible(true);

class TimerListener implements ActionListener
{
    public void actionPerformed(ActionEvent event)
    {
        component.moveBy(1, 1);
    }
}

ActionListener listener = new TimerListener();

final int DELAY = 100; // Milliseconds between timer ticks
Timer t = new Timer(DELAY, listener);
t.start();
Self Check 9.21

Why does a timer require a listener object?

**Answer:** The timer needs to call some method whenever the time interval expires. It calls the `actionPerformed` method of the listener object.
Self Check 9.22

What would happen if you omitted the call to `repaint` in the `moveBy` method?

**Answer:** The moved rectangles won’t be painted, and the rectangle will appear to be stationary until the frame is repainted for an external reason.
Mouse Events

• Use a mouse listener to capture mouse events
• Implement the `MouseListener` interface:

```java
public interface MouseListener {
    void mousePressed(MouseEvent event);
    // Called when a mouse button has been pressed on a component
    void mouseReleased(MouseEvent event);
    // Called when a mouse button has been released on a component
    void mouseClicked(MouseEvent event);
    // Called when the mouse has been clicked on a component
    void mouseEntered(MouseEvent event);
    // Called when the mouse enters a component
    void mouseExited(MouseEvent event);
    // Called when the mouse exits a component
}
```
Mouse Events

- **mousePressed, mouseReleased**: Called when a mouse button is pressed or released

- **mouseClicked**: If button is pressed and released in quick succession, and mouse hasn’t moved

- **mouseEntered, mouseExited**: Mouse has entered or exited the component’s area
Mouse Events

• Add a mouse listener to a component by calling the `addMouseListener` method:

```java
public class MyMouseListener implements MouseListener {
    // Implements five methods
}
MouseListener listener = new MyMouseListener();
component.addMouseListener(listener);
```

• Sample program: enhance `RectangleComponent` — when user clicks on rectangle component, move the rectangle
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Rectangle;
import javax.swing.JComponent;

/**
 * This component displays a rectangle that can be moved.
 */
public class RectangleComponent extends JComponent {

    private static final int BOX_X = 100;
    private static final int BOX_Y = 100;
    private static final int BOX_WIDTH = 20;
    private static final int BOX_HEIGHT = 30;

    private Rectangle box;

    public RectangleComponent() {
        // The rectangle that the paintComponent method draws
        box = new Rectangle(BOX_X, BOX_Y, BOX_WIDTH, BOX_HEIGHT);
    }

    Continued
public void paintComponent(Graphics g)
{
    Graphics2D g2 = (Graphics2D) g;

    g2.draw(box);
}

/**
   Moves the rectangle to the given location.
   @param x the x-position of the new location
   @param y the y-position of the new location
*/
public void moveTo(int x, int y)
{
    box.setLocation(x, y);
    repaint();
}

Mouse Events

• Call `repaint` when you modify the shapes that `paintComponent` draws:

```java
box.setLocation(x, y);
repaint();
```
Mouse Events

- Mouse listener: if the mouse is pressed, listener moves the rectangle to the mouse location:

  ```java
  class MousePressListener implements MouseListener
  {
    public void mousePressed(MouseEvent event)
    {
      int x = event.getX();
      int y = event.getY();
      component.moveTo(x, y);
    }
    // Do-nothing methods
    public void mouseReleased(MouseEvent event) {}
    public void mouseClicked(MouseEvent event) {}
    public void mouseEntered(MouseEvent event) {}
    public void mouseExited(MouseEvent event) {}
  }
  ```

- All five methods of the interface must be implemented; unused methods can be empty
Figure 8
Clicking the Mouse Moves the Rectangle
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import javax.swing.JFrame;

/**
   * This program displays a RectangleComponent.
   */
public class RectangleComponentViewer {
    private static final int FRAME_WIDTH = 300;
    private static final int FRAME_HEIGHT = 400;

    public static void main(String[] args) {
      final RectangleComponent component = new RectangleComponent();
    }
}
// Add mouse press listener

class MousePressListener implements MouseListener
{
    public void mousePressed(MouseEvent event)
    {
        int x = event.getX();
        int y = event.getY();
        component.moveTo(x, y);
    }

    // Do-nothing methods
    public void mouseReleased(MouseEvent event) {}
    public void mouseClicked(MouseEvent event) {}
    public void mouseEntered(MouseEvent event) {}
    public void mouseExited(MouseEvent event) {}
}

MouseListener listener = new MousePressListener();
component.addMouseListener(listener);

Continued
JFrame frame = new JFrame();
frame.add(component);

frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setVisible(true);
Self Check 9.23

Why was the `moveBy` method in the `RectangleComponent` replaced with a `moveTo` method?

**Answer:** Because you know the current mouse position, not the amount by which the mouse has moved.
Self Check 9.24

Why must the `MouseListener` class supply five methods?

**Answer:** It implements the `MouseListener` interface, which has five methods.