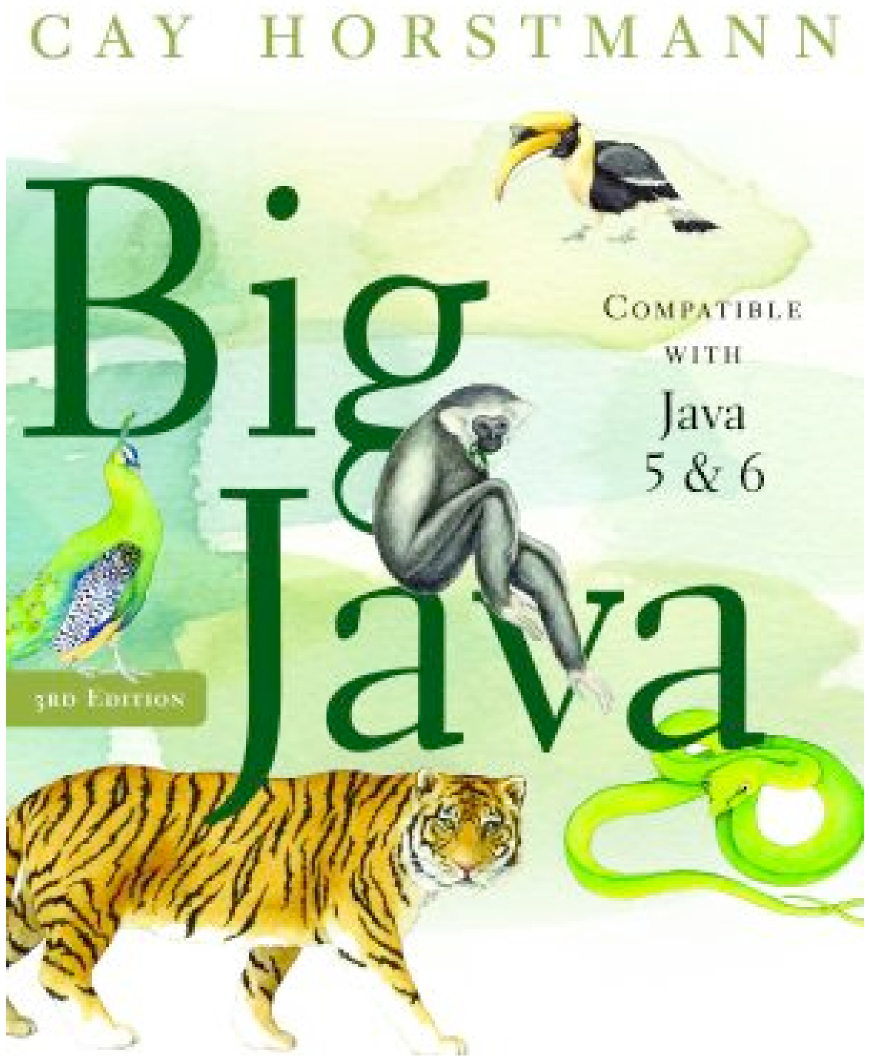


Chapter Nine: Interfaces and Polymorphism

Chapter Goals

- To learn about interfaces
- To be able to convert between class and interface references
- 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
- To understand how inner classes access variables from the surrounding scope
- To implement event listeners in graphical applications

Using Interfaces for Code 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 values (*numbers*)
- What if we want to find the average and maximum of a set of `BankAccount` values?

Continued

Using Interfaces for Code Reuse (cont.)

```
public class DataSet // Modified for BankAccount objects
{
   . . .
   public void add(BankAccount x)
   {
      sum = sum + x.getBalance();
      if (count == 0 || maximum.getBalance() <
         x.getBalance()) maximum = x;
      count++;
   }

   public BankAccount getMaximum()
   {
      return maximum;
   }

   private double sum;
   private BankAccount maximum;
   private int count;
}
```

Using Interfaces for Code 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:

```
public class DataSet // Modified for Coin objects
{
   . . .
   public void add(Coin x)
   {
      sum = sum + x.getValue();
      if (count == 0 || maximum.getValue() <
         x.getValue()) maximum = x;
      count++;
   }
```

Continued

Using Interfaces for Code Reuse

```
   public Coin getMaximum()
   {
      return maximum;
   }

   private double sum;
   private Coin maximum;
   private int count;
}
```

Using Interfaces for Code Reuse

- The mechanics of analyzing the data 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:

```
sum = sum + x.getMeasure();
if (count == 0 || maximum.getMeasure() <
    x.getMeasure())
    maximum = x;
    count++;
```

Continued

Using Interfaces for Code Reuse (cont.)

- 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

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

- Interface declaration lists all methods (and their signatures) that the interface type requires

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

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

   public Measurable getMaximum()
   {
      return maximum;
   }
```

Continued

Generic `DataSet` for Measurable Objects

```
   private double sum;
   private Measurable maximum;
   private int count;
}
```

Implementing an Interface Type

- Use `implements` keyword to indicate that a class implements an interface type

```
public class BankAccount implements Measurable
{
   public double getMeasure()
   {
      return balance;
   }
   // Additional methods and fields
}
```

- A class can implement more than one interface type
  - *Class must define all the methods that are required by all the interfaces it implements*

Continued

Implementing an Interface Type (cont.)

- Use `implements` keyword to indicate that a class implements an interface type

```
public class BankAccount implements Measurable
{
   public double getMeasure()
   {
      return balance;
   }
   // Additional methods and fields
}
```

- A class can implement more than one interface type
  - *Class must define all the methods that are required by all the interfaces it implements*

UML Diagram of `DataSet` and Related Classes

- 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-sh*
    relationship or dependency
- Note that `DataSet` is *decouple*

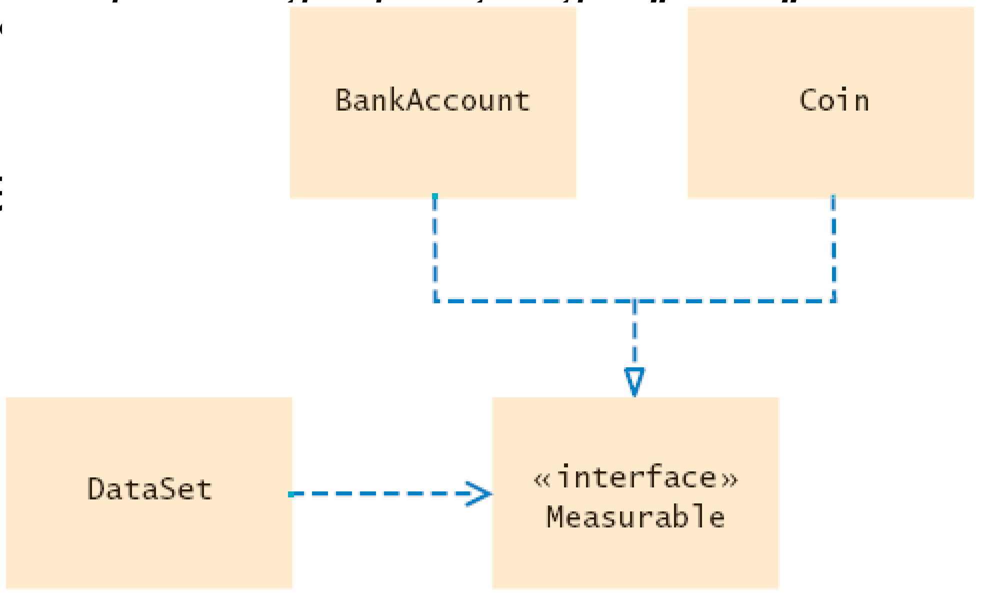


Figure 1 UML Diagram of the DataSet Class and the Classes That Implement the Measurable Interface

Syntax 9.1 Defining an Interface

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

Example:

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

Purpose:

To define an interface and its method signatures. The methods are automatically public.

Syntax 9.2 Implementing an Interface

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

Example:

```
public class BankAccount implements Measurable
{
   // Other BankAccount methods
   public double getMeasure()
   {
      // Method implementation
   }
}
```

Continued

Syntax 9.2 Implementing an Interface (cont.)

Purpose:

To define a new class that implements the methods of an interface.

ch09/measure1/DataSetTester.java

```java
/**
   This program tests the DataSet class.
*/
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");

```

ch09/measure1/DataSetTester.java (cont.)

```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"));

      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");
   }
}
```

ch09/measure1/DataSetTester.java (cont.)

Output:

```
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
```

Self Check 9.1

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`

Casts

- Add coin objects to `DataSet`

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

- What can you do with it? It's not of type `Coin`

  ```
  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:

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

Continued

Casts (cont.)

- If you are wrong and `max` isn't a coin, the compiler throws an exception
- 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 (`BankAccount`) `x` to convert a Measurable variable `x` to a `BankAccount` reference?

Answer: Only if `x` actually refers to a `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.

Polymorphism

- Interface variable holds reference to object of a class that implements the interface

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

- Note that the object to which x 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:

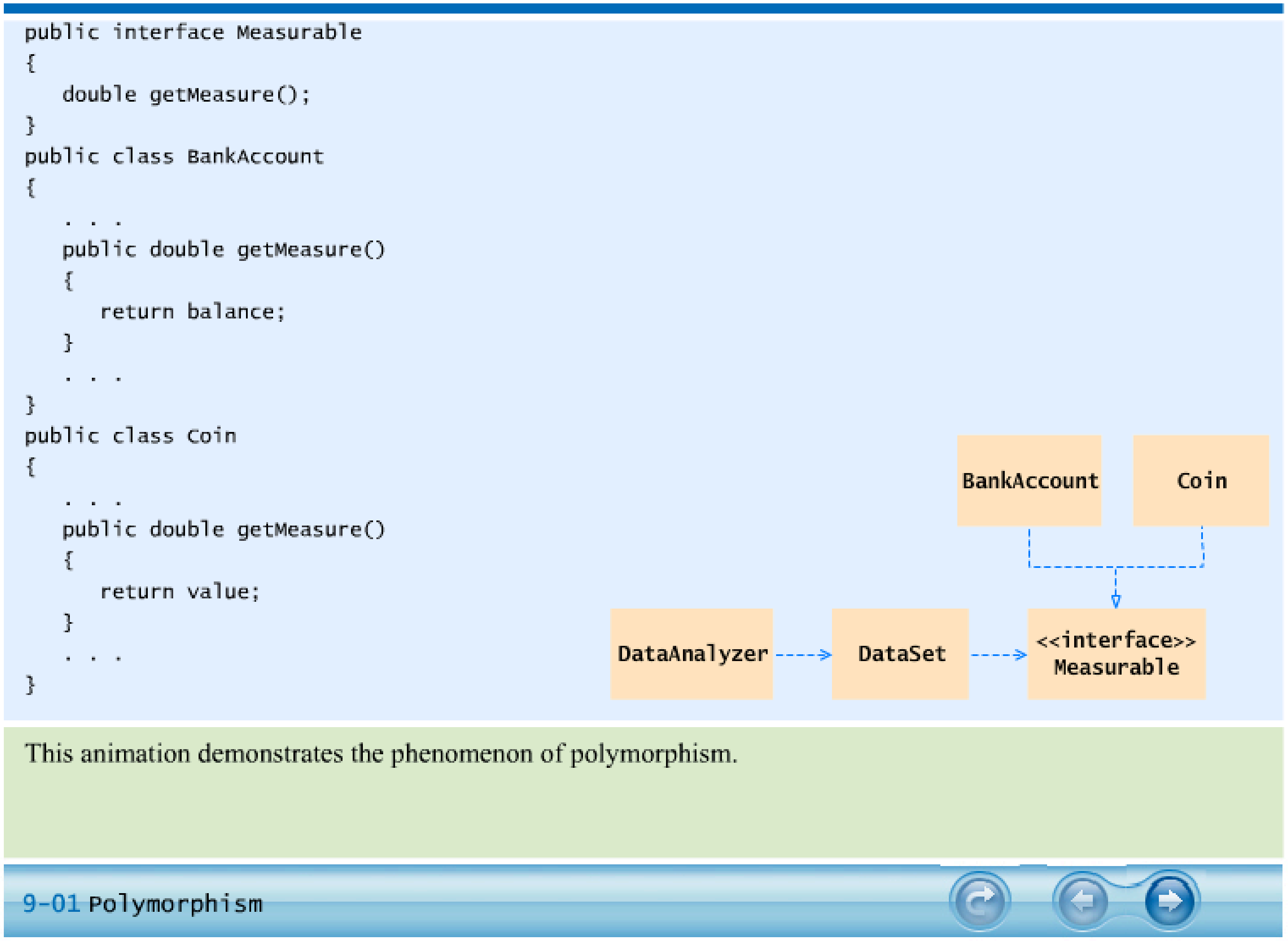
```
double m = x.getMeasure();
```

- Which method is called?

Polymorphism

- Depends on the actual object
- If x refers to a bank account, calls `BankAccount.getMeasure`
- If x refers to a coin, calls `Coin.getMeasure`
- Polymorphism (many shapes): Behavior can vary depending on the actual type of an object
- Called *late binding*: resolved at runtime
- Different from overloading; overloading is resolved by the compiler (*early binding*)

Animation 9.1 –



```
public interface Measurable
{
   double getMeasure();
}
public class BankAccount
{
   . . .
   public double getMeasure()
   {
      return balance;
   }
   . . .
}
public class Coin
{
   . . .
   public double getMeasure()
   {
      return value;
   }
   . . .
}
```

This animation demonstrates the phenomenon of polymorphism.

Self Check 9.5

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.

Self Check 9.7

What do overloading and polymorphism have in common? Where do they differ?

Answer: Both describe a situation where one method name can denote multiple methods. However, overloading is resolved early by the compiler, by looking at the types of the parameter variables. Polymorphism is resolved late, by looking at the type of the implicit parameter object just before making the call.

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 mechanism: allows a class to call back a specific method when it needs more information
- In previous `DataSet` implementation, responsibility of measuring lies with the added objects themselves

Continued

Using Interfaces for Callbacks (cont.)

- Alternative: Hand the object to be measured to a method:

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

- `Object` is the "lowest common denominator" of all classes

Using Interfaces for Callbacks

`add` method asks measurer (and not the added object) to do the measuring:

```
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

- You can define measurers to take on any kind of measurement

```
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`

```
Rectangle aRectangle = (Rectangle) anObject;
```

Continued

Using Interfaces for Callbacks (cont.)

- Pass measurer to data set constructor:

```
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)); . . .
```

UML Diagram of `Measurer` Interface and Related Classes

Note that the `Rectangle` class is decoupled from the `Measurer` interface

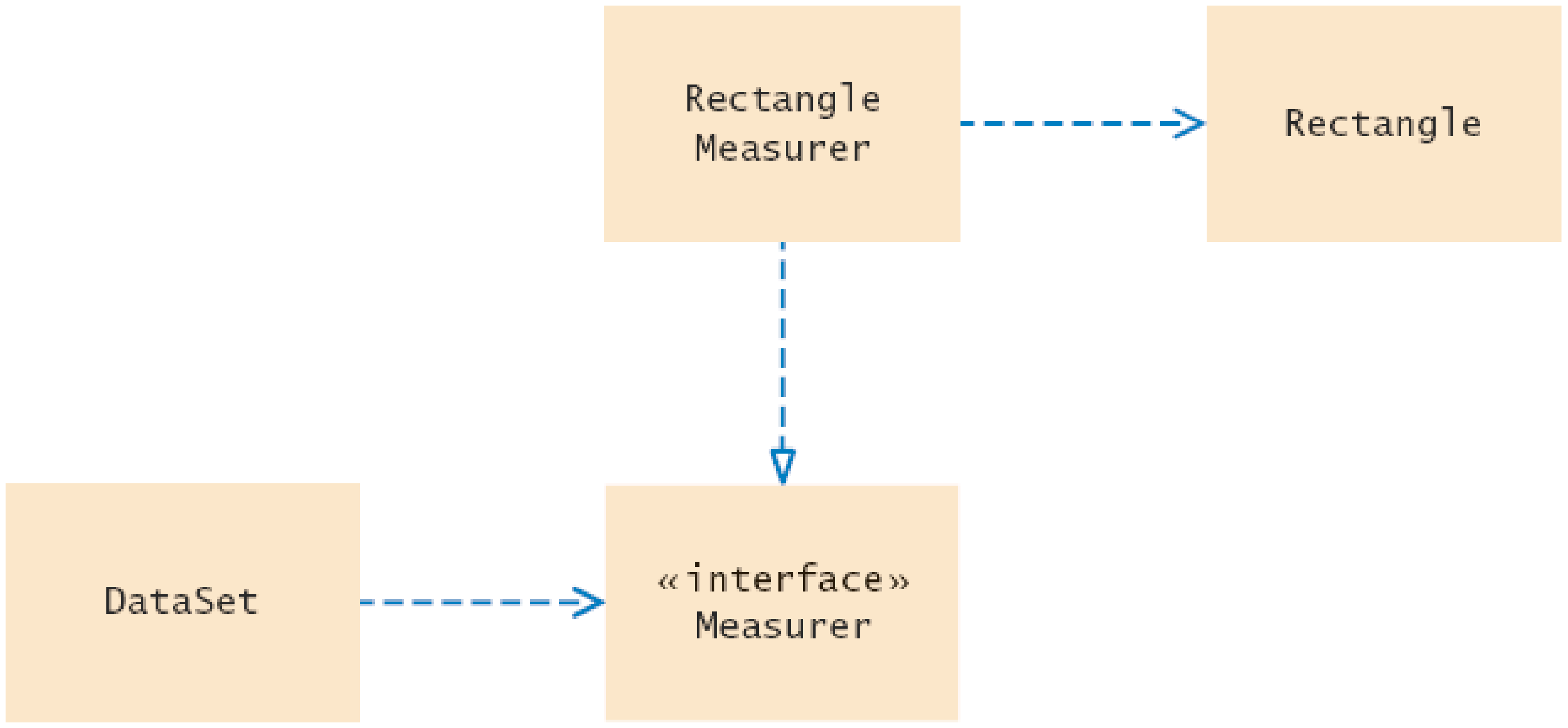


Figure 2 UML Diagram of the DataSet Class and the Measurer Interface

ch09/measure2/DataSet.java

```java
/**
   Computes the average of a set of data values.
*/
public class DataSet
{
   /**
      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
   */
```

Continued

ch09/measure2/DataSet.java (cont.)

```
   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;
   }

```

Continued

ch09/measure2/DataSet.java (cont.)

```
   /**
      Gets the largest of the added data.
      @return the maximum or 0 if no data has been added
   */
   public Object getMaximum()
   {
      return maximum;
   }

   private double sum;
   private Object maximum;
   private int count;
   private Measurer measurer;
}
```

ch09/measure2/DataSetTester2.java

```java
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");

```

Continued

ch09/measure2/DataSetTester2.java (cont.)

```
      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]");
   }
}
```

ch09/measure2/Measurer.java

```java
/**
   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);
}
```

ch09/measure2/RectangleMeasurer.java

```java
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;
   }
}
```

ch09/measure2/RectangleMeasurer.java (cont.)

Output:

```
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 9.10

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 defined inside a method

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

Continued

Inner Classes (cont.)

- If inner class is defined inside an enclosing class, but outside its methods, it is available to all methods of enclosing class
- Compiler turns an inner class into a regular class file:
  `DataSetTester$1$RectangleMeasurer.class`

Syntax 9.3 Inner Classes

Declared inside a method

```
class OuterClassName
{
   method signature
   {
      . . .
      class InnerClassName
      {
         // methods
         // fields
      }
      . . .
   }
   . . .
}
```

Declared inside the class

```
class OuterClassName
{
   // methods
   // fields
   accessSpecifier class
      InnerClassName
   {
      // methods
      // fields
   }
   . . .
}
```

Continued

Syntax 9.3 Inner Classes

Example:

```
public class Tester
{
   public static void main(String[] args)
   {
      class RectangleMeasurer implements Measurer
      {
         . . .
      }
      . . .
   }
}
```

Purpose:

To define an inner class whose scope is restricted to a single method or the methods of a single class.

ch09/measure3/DataSetTester3.java

```java
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;
         }
      }

```

Continued

ch09/measure3/DataSetTester3.java (cont.)

```
      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]");
   }
}
```

Self Check 9.11

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

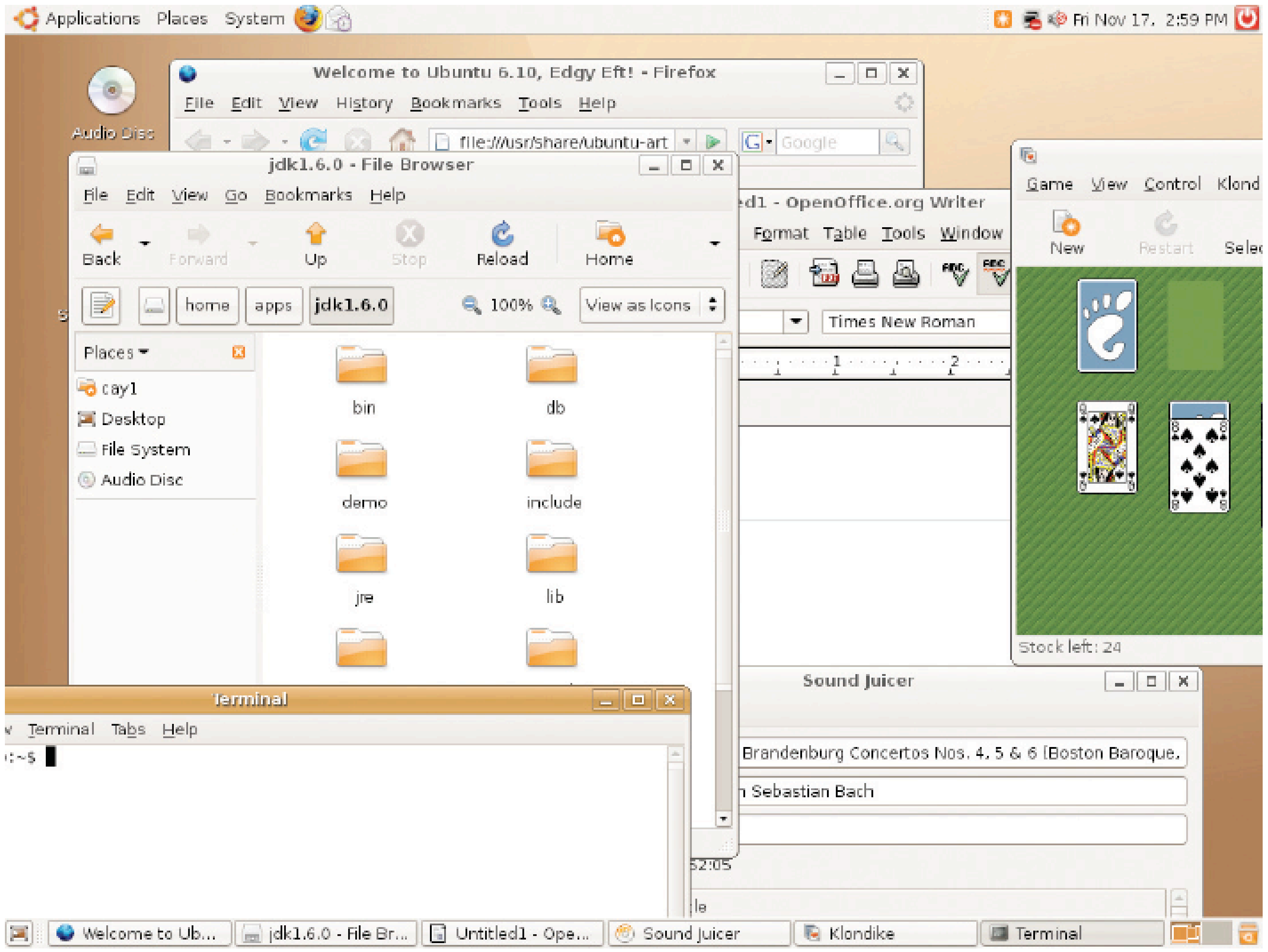
Answer: `Inner` classes are convenient for insignificant classes. Also, their methods can access variables and fields from the surrounding scope.

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

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
- 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:
  - *Event sources report on events*
  - *When an event occurs, the event source notifies all event listeners*

Events, Event Sources, and Event Listeners

- Example: 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

Continued

Events, Event Sources, and Event Listeners (cont.)

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

```
ActionListener listener = new ClickListener();
button.addActionListener(listener);
```

ch09/button1/ClickListener.java

```java
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.");
   }
}
```

ch09/button1/ButtonViewer.java

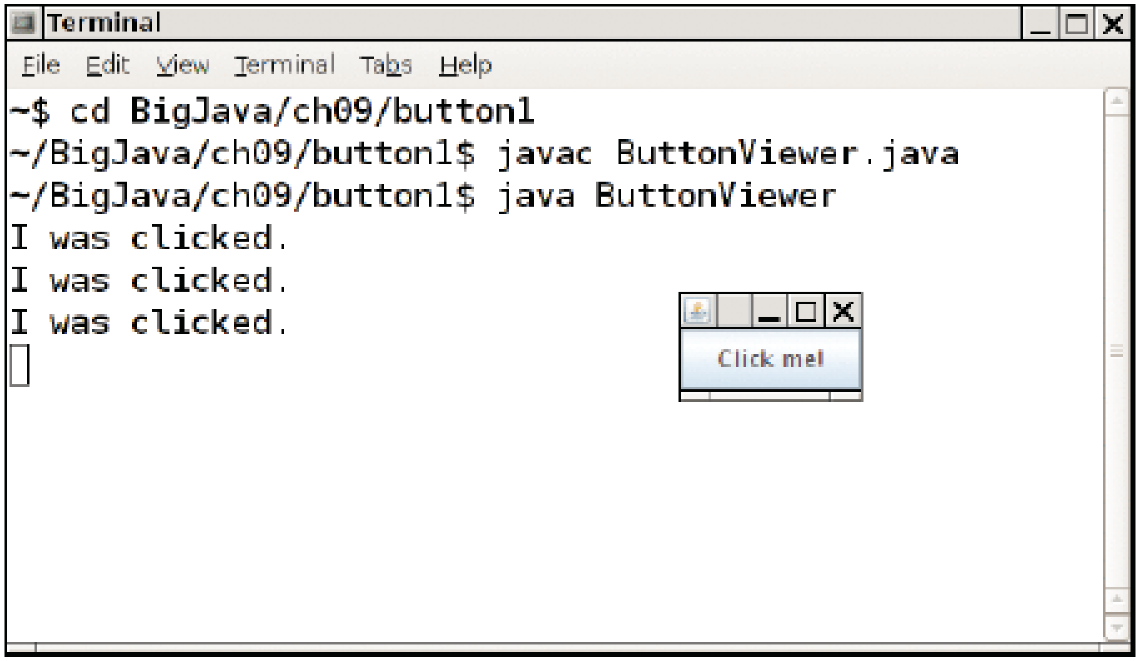
```java
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;

/**
   This program demonstrates how to install an action listener.
*/
public class ButtonViewer
{
   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);

      frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
```

Continued

ch09/button1/ButtonViewer.java (cont.)

```
    private static final int FRAME_WIDTH = 100;
    private static final int FRAME_HEIGHT = 60;
}
```

Output:



```
~$ 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 3 Implementing an Action Listener

Self Check 9.13

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 9.14

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:

```
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
- Methods of an inner class can access local variables from surrounding blocks and fields from surrounding classes

Using Inner Classes for Listeners

- 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:

```
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 AddInterestListener implements ActionListener
{
```

Continued

Using Inner Classes for Listeners (cont.)

```
      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 AddInterestListener();
   button.addActionListener(listener);
```

ch09/button2/InvestmentViewer1.java

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

      // The button to trigger the calculation
      JButton button = new JButton("Add Interest");
      frame.add(button);

```

Continued

ch09/button2/InvestmentViewer1.java (cont.)

```
      // 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());
         }
      }

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

```

Continued

ch09/button2/InvestmentViewer1.java (cont.)

```
      frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }

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

   private static final int FRAME_WIDTH = 120;
   private static final int FRAME_HEIGHT = 60;
}
```

Output:

```
balance: 1100.0 balance: 1210.0
balance: 1331.0 balance: 1464.1
```

Self Check 9.15

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 9.16

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

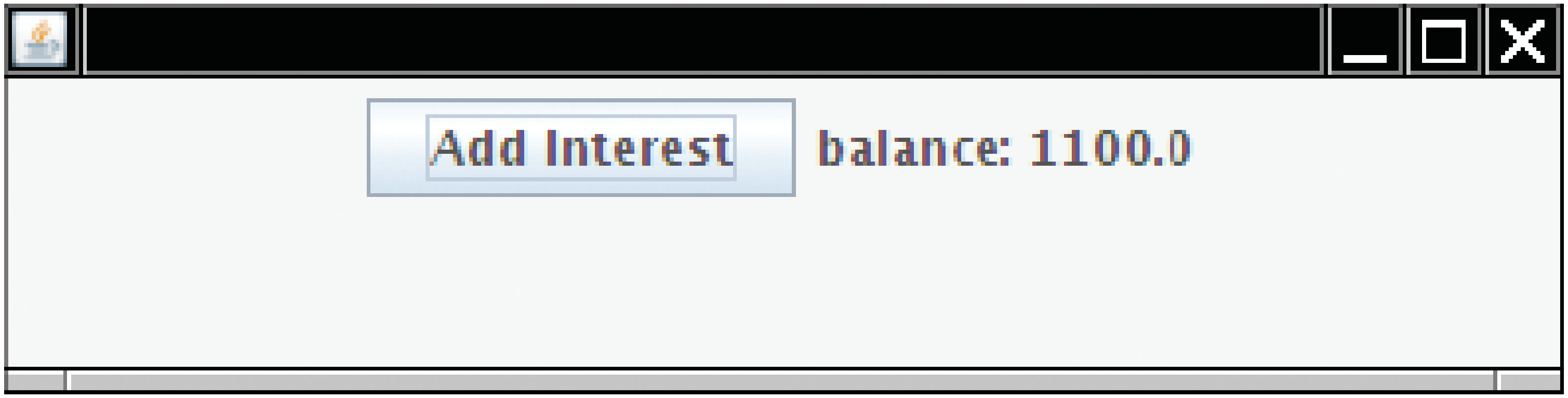


Figure 4 An Application with a Button

Continued

Building Applications with Buttons (cont.)

- Construct an object of the `JButton` class:
  ```
  JButton button = new JButton("Add Interest");
  ```
- We need a user interface component that displays a message:
  ```
  JLabel label = new JLabel("balance: " +
  account.getBalance());
  ```
- Use a `JPanel` container to group multiple user interface components together:
  ```
  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:

```
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`)

ch09/button3/InvestmentViewer2.java

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

      // The button to trigger the calculation
      JButton button = new JButton("Add Interest");
```

Continued

ch09/button3/InvestmentViewer2.java   (cont.)

```

      // 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;
```

Continued

ch09/button3/InvestmentViewer2.java (cont.)

```
               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);
      }

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

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

Self Check 9.17

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 9.18

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
- Useful whenever you want to have an object updated in regular intervals
- Sends events to action listener

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

Continued

Processing Timer Events (cont.)

- Define a class that implements the `ActionListener` interface

```
class MyListener implements ActionListener
{
   void actionPerformed(ActionEvent event)
   {
      // This action will be executed at each timer
         event
      Place listener action here
   }
}
```

- Add listener to timer

```
MyListener listener = new MyListener();
Timer t = new Timer(interval, listener);
t.start();
```

ch09/timer/RectangleComponent.java (cont.)

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

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

```

Continued

ch09/timer/RectangleComponent.java

```
        g2.draw(box);
    }

    /**
       Moves the rectangle by a given amount.
       @param x the amount to move in the x-direction
       @param y the amount to move in the y-direction
    */
    public void moveBy(int dx, int dy)
    {
       box.translate(dx, dy);
       repaint();
    }

    private Rectangle box;

    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;
}
```

ch09/timer/RectangleComponent.java

- 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

Continued

ch09/timer/RectangleMover.java

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

public class RectangleMover
{
   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);

      final RectangleComponent component = new RectangleComponent();
      frame.add(component);

      frame.setVisible(true);

```

Continued

ch09/timer/RectangleMover.java (cont.)

```
         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();
      }

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

Self Check 9.19

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 9.20

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:

```
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
```

Mouse Events

```
    void mouseExited(MouseEvent event);
    // Called when the mouse exits a component }
```

- `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:

```
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

ch09/mouse/RectangleComponent.java

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

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

```

Continued

ch09/mouse/RectangleComponent.java (cont.)

```java
      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();
   }

   private Rectangle box;

   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;
}
```

Mouse Events

- Call `repaint` when you modify the shapes that `paintComponent` draws `box.setLocation(x, y); repaint();`
- Mouse listener: if the mouse is pressed, `listener` moves the rectangle to the mouse location class `MousePressListener` implements

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

Continued

Mouse Events

-
  ```
  // 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

`RectangleComponentViewer` Program Output

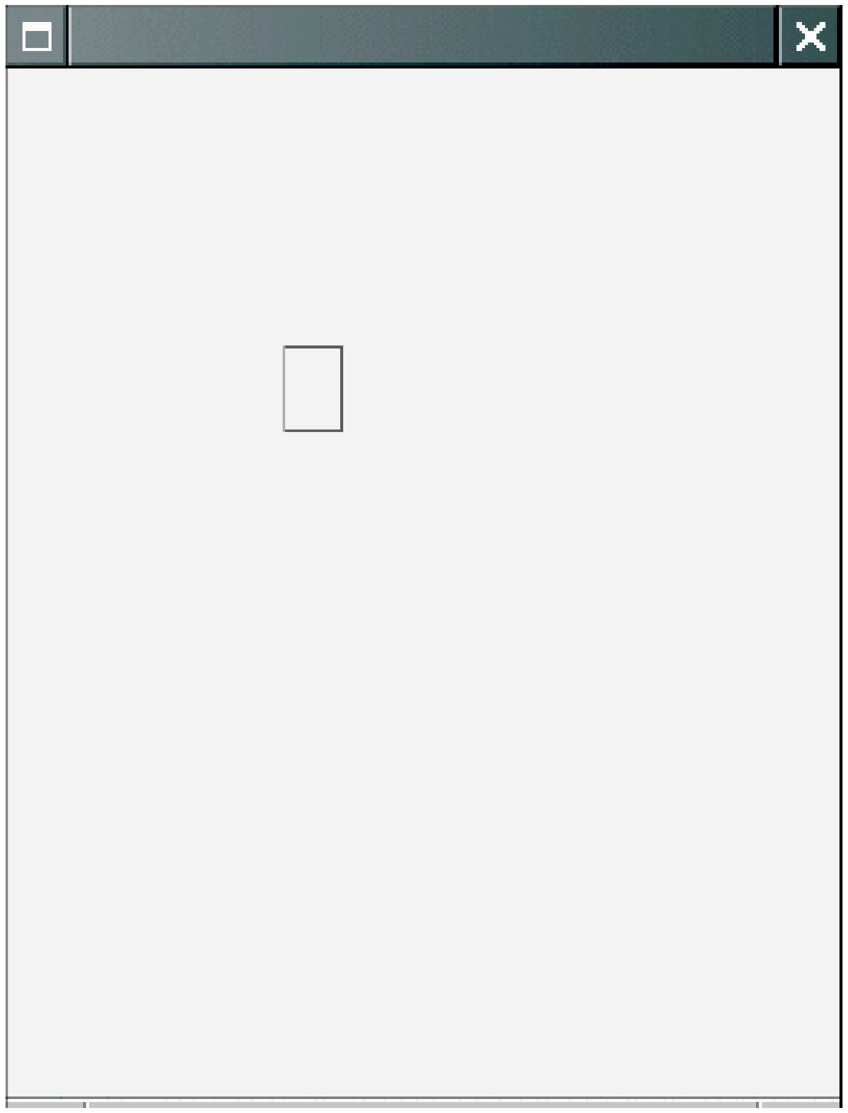


Figure 5 Clicking the Mouse Moves the Rectangle

ch09/mouse/RectangleComponentViewer.java

```java
import java.awt.event.MouseListener;
import java.awt.event.MouseEvent;
import javax.swing.JFrame;

/**
   This program displays a RectangleComponent.
*/
public class RectangleComponentViewer
{
   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);
         }
```

Continued

ch09/mouse/RectangleComponentViewer.java (cont.)

```java

            // 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);

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

         frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
         frame.setVisible(true);
      }

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

Self Check 9.21

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 9.22

Why must the `MousePressListener` class supply five methods?

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