Week 8 & 9

Interfaces & Event Handlers

(Chapter 10)

Chapter Goals

- To be able to declare and use interface types
- To appreciate how interfaces can be used to decouple classes
- To learn how to implement helper classes as inner classes
- To implement event listeners in graphical applications

Using Interfaces for Algorithm Reuse

- **Interface types** are used to express common operations.
- Interfaces make it possible to make a service available to a wide set.
- This restaurant is willing to serve anyone who conforms to the `Customer` interface with `eat` and `pay` methods.

Defining an Interface Type

- Example: a method to compute the average of an array of Objects
  - The algorithm for computing the average is the same in all cases
  - Details of measurement differ
- **Goal**: write one method that provides this service.
- We can't call `getBalance` in one case and `getArea` in another.
- **Solution**: all object who want this service must agree on a `getMeasure` method
  - `BankAccount's getMeasure` will return the balance
  - `Country's getMeasure` will return the area
- Now we implement a single average method that computes the sum:

```
sum = sum + obj.getMeasure();
```

Defining an Interface Type

- Problem: we need to declare a type for `obj`
- Need to invent a new type that describes any class whose objects can be measured.
- An interface type is used to specify required operations (like `getMeasure`):

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

- A Java interface type declares methods but does not provide their implementations.

Syntax 10.1 Declaring an Interface

Syntax

```
public interface InterfaceName
{
   method headers
}
```

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

The methods of an interface are automatically public.

No implementation is provided.

Defining an Interface Type

- An interface type is similar to a class.
- Differences between classes and interfaces:
  - An interface type *does not* have instance variables.
  - All methods in an interface type are `abstract` (or in Java 8, `static` or `default`)
    - They have a name, parameters, and a return type, but no implementation.
  - All methods in an interface type are automatically public.
  - An interface type has no constructor.
    - You cannot construct objects of an interface type.

Implementing an Interface Type

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

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

- `BankAccount` objects are instances of the `Measurable` type:

```
Measurable obj = new BankAccount(); // OK

Measurable obj = new Country(); // OK
```

Implementing an Interface Type

- A variable of type `Measurable` holds a reference to an object of some class that implements the `Measurable` interface.
- `Country` class can also implement the `Measurable` interface:

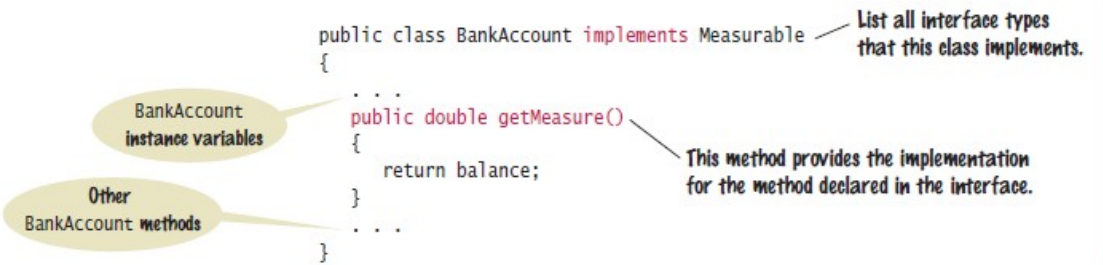
```
public class Country implements Measurable
{
   public double getMeasure()
   {
      return area;
   }
   . . .
}
```

- Use interface types to make code more reusable.

Syntax 10.2 Implementing an Interface

Syntax

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



Defining an Interface Type

- Implementing a reusable average method:

```
public static double average(Measurable[] objects)
{
  double sum = 0;
  for (Measurable obj : objects)
  {
    sum = sum + obj.getMeasure();
  }
  if (objects.length > 0)
  { return sum / objects.length; }
  else { return 0; }
}
```

- This method is can be used for objects of any class that conforms to the `Measurable` type.

© gregory horler/iStockphoto.

This stand-mixer provides the “rotation” service to any attachment that conforms to a common interface. Similarly, the average method at the end of this section works with any class that implements a common interface.

Implementing an Interface Type

- Put the `average` method in a class - say `Data`

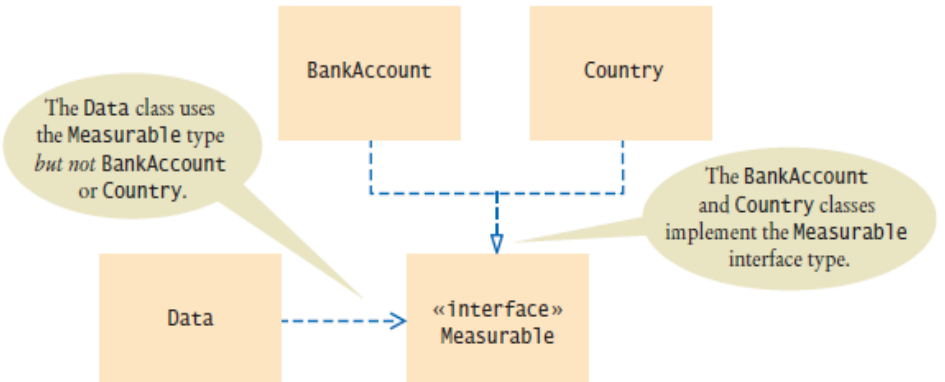
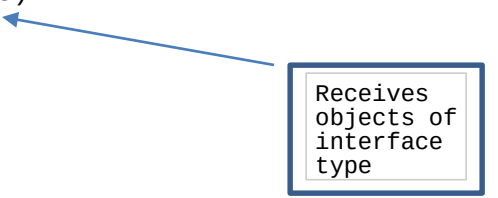


Figure 1 UML Diagram of the Data Class and the Classes that Implement the `Measurable` Interface

- `Data` class is decoupled from the `BankAccount` and `Country` classes.

section_1/Data.java

```
1  public class Data
2  {
3    /**
4       Computes the average of the measures of the given objects.
5       @param objects an array of Measurable objects
6       @return the average of the measures
7    */
8    public static double average(Measurable[] objects)
9    {
10     double sum = 0;
11     for (Measurable obj : objects)
12     {
13       sum = sum + obj.getMeasure();
14     }
15     if (objects.length > 0) { return sum / objects.length; }
16     else { return 0; }
17   }
18 }
```



section_1/MeasurableTester.java

```
1  /**
2     This program demonstrates the measurable BankAccount and Country classes.
3  */
4  public class MeasurableTester
5  {
6     public static void main(String[] args)
7     {
8        Measurable[] accounts = new Measurable[3];
9        accounts[0] = new BankAccount(0);
10       accounts[1] = new BankAccount(10000);
11       accounts[2] = new BankAccount(2000);
12
13       double averageBalance = Data.average(accounts);
14       System.out.println("Average balance: " + averageBalance);
15       System.out.println("Expected: 4000");
16
17       Measurable[] countries = new Measurable[3];
18       countries[0] = new Country("Uruguay", 176220);
19       countries[1] = new Country("Thailand", 513120);
20       countries[2] = new Country("Belgium", 30510);
21
22       double averageArea = Data.average(countries);
23       System.out.println("Average area: " + averageArea);
24       System.out.println("Expected: 239950");
25    }
26 }
```

Program Run:

```
Average balance: 4000
Expected: 4000
Average area: 239950
Expected: 239950
```

Class Example: Printable

```
//Printable.java
public interface Printable
{
 String printMe(); //public by default
}

//NewVehicle.java
public class NewVehicle implements Printable {
  private String vehicleName;
  public NewVehicle(String givenName) {this.vehicleName = givenName;}
  public String getName() {return this.vehicleName;}
  public void setName(String givenName) {this.vehicleName = givenName;}
  public String printMe() //must use public
  {
    return "Vehicle Name: "+vehicleName;
  }
}

//BankAccount.java
public class BankAccount implements Printable {
  private double acctBalance;
  public BankAccount(double givenBalance) {this.acctBalance = givenBalance;}
  public double getBalance() {return this.acctBalance;}
  public void setBalance(double givenBalance) {this.acctBalance = givenBalance;}
  public String printMe() //must use public
  {
    return "Account Balance: "+acctBalance;
  }
}
```

Class Example: Printable

```
//InterfaceTester.java
public class InterfaceTester {

   public static void main(String args[]) {

     NewVehicle myCar = new NewVehicle("Kia");
     System.out.println(myCar.getName());
     System.out.println(myCar);

     BankAccount myAcct = new BankAccount(123.45);
     System.out.println(myAcct.getBalance());
     System.out.println(myAcct);

     Printable[] TwoObjects = new Printable[2];
     TwoObjects[0] = new NewVehicle("Honda");
     TwoObjects[1] = new BankAccount(22.56);

     for (Printable pobj : TwoObjects)
     {
      System.out.println(pobj.printMe());
     }
   }
}
```

Comparing Interfaces and Inheritance

- Here is a different interface: `Named`

```
public interface Named
{
  String getName();
}
```

- A class can implement more than one interface:

```
public class Country implements Measurable, Named
```

- A class can only extend (inherit from) a single superclass.
- An interface specifies the behavior that an implementing class should supply (in Java 8, an interface can now supply a *default* implementation).
- A superclass provides some implementation that a subclass inherits.
- Develop interfaces when you have code that processes objects of different classes in a common way.

Converting From Classes to Interfaces

- You can convert from a class type to an interface type, provided the class implements the interface.
- A `Measurable` variable can refer to an object of the `BankAccount` class because `BankAccount` implements the `Measurable` interface:

```
BankAccount account = new BankAccount(1000);
Measurable meas = account; // OK
```

- A `Measurable` variable can refer to an object of the `Country` class because that class also implements the `Measurable` interface:

```
Country uruguay = new Country("Uruguay", 176220);
Measurable meas = uruguay; // Also OK
```

- A `Measurable` variable **cannot** refer to an object of the `Rectangle` class because `Rectangle` doesn't implement `Measurable`:

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

Variables of Class and Interface Types

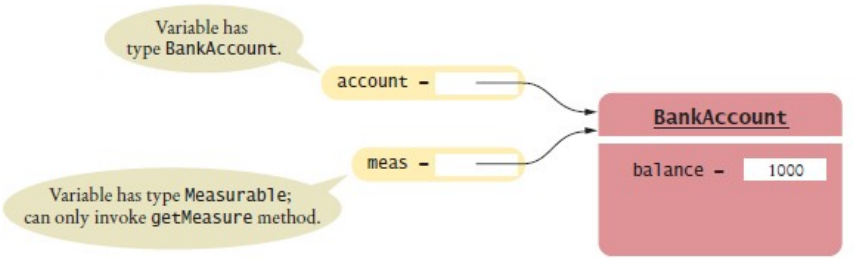


Figure 2 Two references to the same object

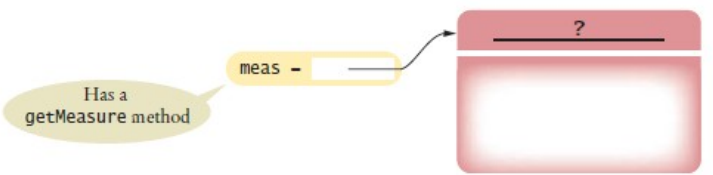


Figure 3 An Interface Reference Can Refer to an Object of Any Class that Implements the Interface

- Method calls on an interface reference are polymorphic. The appropriate method is determined at run time.

Casting from Interfaces to Classes

- Method to return the object with the largest measure:

```
public static Measurable larger(Measurable obj1, Measurable obj2)
{
  if (obj1.getMeasure() > obj2.getMeasure())
  {
    return obj1;
  }
  else
  {
    return obj2;
  }
}
```

- Returns the object with the larger measure, as a `Measurable` reference.

```
Country uruguay = new Country("Uruguay", 176220);
Country thailand = new Country("Thailand", 513120);
Measurable max = larger(uruguay, thailand);
```

Casting from Interfaces to Classes

- You know that `max` refers to a `Country` object, but the compiler does not.
- Solution: cast

```
Country maxCountry = (Country) max;
String name = maxCountry.getName();
```

- You need a cast to convert from an interface type to a class type.
- If you are wrong and `max` doesn't refer to a `Country` object, the program throws an exception at runtime.
- If a `Person` object is actually a `Superhero`, you need a cast before you can apply any `Superhero` methods.

The Comparable Interface

- `Comparable` interface is in the standard Java library.
- `Comparable` interface has a single method:

```
public interface Comparable
{
   int compareTo(Object otherObject);
}
```

- The call to the method:

```
a.compareTo(b)
```

- The `compareTo` method returns:
  - a negative number if a should come before b,
  - zero if a and b are the same
  - a positive number if b should come before a.
- Implement the `Comparable` interface so that objects of your class can be compared, for example, in a sort method.

The Comparable Interface

- `BankAccount` class' implementation of `Comparable`:

```
public class BankAccount implements Comparable
{
   . . .
   public int compareTo(Object otherObject)
   {
      BankAccount other = (BankAccount) otherObject;
      if (balance < other.balance) { return -1; }
      if (balance > other.balance) { return 1; }
      return 0;
   }
   . . .
}
```

- `compareTo` method has a parameter of reference type `Object`
- To get a `BankAccount` reference:

```
BankAccount other = (BankAccount) otherObject;
```

The Comparable Interface

- Because the `BankAccount` class implements the `Comparable` interface, you can sort an array of bank accounts with the `Arrays.sort` method:

```
BankAccount[] accounts = new BankAccount[3];
accounts[0] = new BankAccount(10000);
accounts[1] = new BankAccount(0);
accounts[2] = new BankAccount(2000);
Arrays.sort(accounts);
```

- Now the `accounts` array is sorted by increasing balance.
- The `compareTo` method checks whether another object is larger or smaller.

The Cloneable Interface

- `Cloneable` interface is in the standard Java library
- The call to the method:

```
protected Object clone()
```

- It creates a new object of the same type as the original object and *automatically copies the instance variables* from the original object to the cloned object.
- If an object contains a reference to another object, then the `clone` method *makes a copy of that object reference*, not a clone of that object.
- Such a copy is called a **shallow copy**

- Callers of `clone()` must catch `CloneNotSupportedException` exception

```
public class BankAccount implements Cloneable
{  . . .
   public Object clone() {
      try
      {
        return super.clone();
      }
      catch (CloneNotSupportedException e) {
        //Can't happen because we implement Cloneable but we still must catch it.
        return null;
      }
   }
}
```

The Cloneable Interface

- If an object contains a reference to another mutable object, then you must call `clone` for that reference

```
public class Customer implements Cloneable
{
   private String name;
   private BankAccount account;
   . . .
   public Object clone() {
      try {
        Customer cloned = (Customer) super.clone();
        cloned.account = (BankAccount) account.clone();
        return cloned;
      }
      catch(CloneNotSupportedException e)
      { // Can't happen because we implement Cloneable
        return null;
      }
   }
}
```

- In general, implementing the clone method requires these steps:
  - Make the class implement the `Cloneable` interface type.
  - In the clone method, call `super.clone()`.
  - Catch the `CloneNotSupportedException` if the superclass is `Object`.
  - Clone any mutable instance variables.

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 cars by both speed and price
- **Callback:** a mechanism for specifying code that is executed at a later time.
- Problem: the responsibility of measuring lies with the added objects themselves.
- Alternative: give the average method both the *data to be averaged* and *a method of measuring*.
- Create an interface:

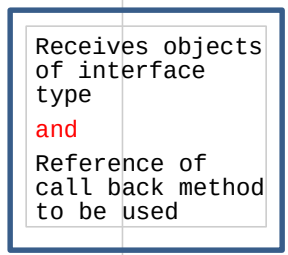
```
public interface Measurer
{
   double measure(Object anObject);
}
```

- All objects can be converted to `Object`.

Using Interfaces for Callbacks

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

```
public static double average(Object[] objects, Measurer meas)
{
  double sum = 0;
  for (Object obj : objects)
  {
    sum = sum + meas.measure(obj);
  }
  if (objects.length > 0) { return
  sum / objects.length; }  else {
  return 0; }
}
```



- The average method simply makes a callback to the measure method whenever it needs to measure any object.

Using Interfaces for Callbacks

- A specific callback is obtained by implementing the `Measurer` interface:

```
public class AreaMeasurer 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;
```

Using Interfaces for Callbacks

- To compute the average area of rectangles:
  - construct an object of the `AreaMeasurer` class and pass it to the average method:

```
Measurer areaMeas = new AreaMeasurer();
Rectangle[] rects = {
      new Rectangle(5, 10, 20, 30),
      new Rectangle(10, 20, 30, 40)
  };
  double averageArea = average(rects, areaMeas);
```

- The `average` method will ask the `AreaMeasurer` object to measure the rectangles.

Using Interfaces for Callbacks

- The `Data` class (which holds the `average` method) is decoupled from the class whose objects it processes (`Rectangle`).
- You provide a small "helper" class `AreaMeasurer`, to process rectangles.

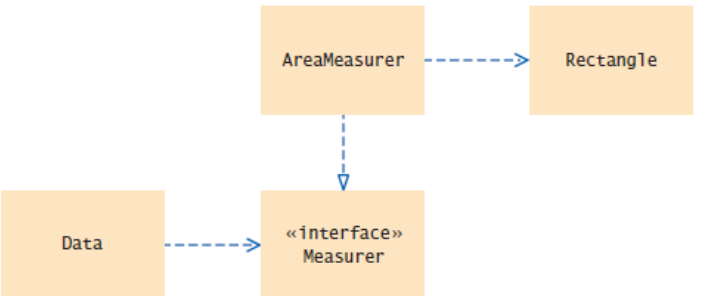


Figure 6 UML Diagram of the `Data` Class and the `Measurer` Interface

section_4/Measurer.java

```
1  /**
2     Describes any class whose objects can measure other objects.
3  */
4  public interface Measurer
5  {
6     /**
7        Computes the measure of an object.
8        @param anObject the object to be measured
9        @return the measure
10    */
11    double measure(Object anObject);
12 }
```

section_4/AreaMeasurer.java

```
1  import java.awt.Rectangle;
2
3  /**
4     Objects of this class measure rectangles by area.
5  */
6  public class AreaMeasurer implements Measurer
7  {
8     public double measure(Object anObject)
9     {
10       Rectangle aRectangle = (Rectangle) anObject;
11       double area = aRectangle.getWidth() * aRectangle.getHeight();
12       return area;
13    }
14 }
```

section_4/Data.java

```
1  public class Data
2  {
3     /**
4        Computes the average of the measures of the given objects.
5        @param objects an array of objects
6        @param meas the measurer for the objects
7        @return the average of the measures
8     */
9     public static double average(Object[] objects, Measurer meas)
10    {
11       double sum = 0;
12       for (Object obj : objects)
13       {
14          sum = sum + meas.measure(obj);
15       }
16       if (objects.length > 0) { return sum / objects.length; }
17       else { return 0; }
18    }
19 }
```

section_4/MeasurerTester.java

```
1  import java.awt.Rectangle;
2
3  /**
4     This program demonstrates the use of a Measurer.
5  */
6  public class MeasurerTester
7  {
8     public static void main(String[] args)
9     {
10       Measurer areaMeas = new AreaMeasurer();
11       Rectangle[] rects = new Rectangle[]
12          {
13             new Rectangle(5, 10, 20, 30),
14             new Rectangle(10, 20, 30, 40),
15             new Rectangle(20, 30, 5, 15)
16          };
17
18       double averageArea = Data.average(rects, areaMeas);
19       System.out.println("Average area: " + averageArea);
20       System.out.println("Expected: 625");
21    }
22 }
23
```

Program Run:

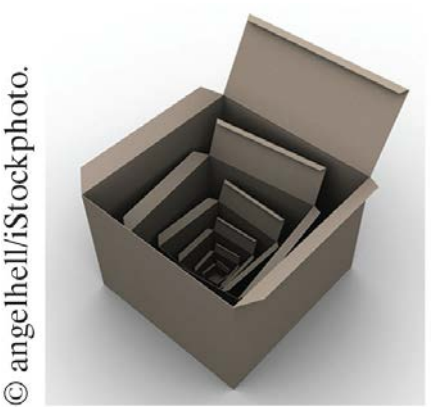
```
Average area: 625
Expected: 625
```

Inner Classes

- Trivial class can be declared inside a method:

```
public class MeasurerTester
{
   public static void main(String[] args)
   {
      class AreaMeasurer implements Measurer
      {
         . . .
      }
      . . .
      Measurer areaMeas = new AreaMeasurer();
      double averageArea = Data.average(rects, areaMeas);
      . . .
   }
}
```

- An inner class is a class that is declared inside another class.



Inner Classes

- You can declare inner class inside an enclosing class, but outside its methods.
- It is available to all methods of enclosing class:

```
public class MeasurerTester
{
   class AreaMeasurer implements Measurer
   {
      . . .
   }

   public static void main(String[] args)
   {
      Measurer areaMeas = new AreaMeasurer();
      double averageArea = Data.average(rects, areaMeas);
      . . .
   }
}
```

- Compiler turns an inner class into a regular class file with a strange name:

```
MeasurerTester$1AreaMeasurer.class
```

- Inner classes are commonly used for utility classes that should not be visible elsewhere in a program.

Anonymous Classes

- Typically, when something is only needed once.
- Can be defined within a method or within a class

Example: anonymous object

```
Country belgium = new Country("Belgium", 30510);
countries.add(belgium);
```

Can be declared now as

```
countries.add(new Country("Belgium", 30510));
```

- An inner class may create the object only once

```
public static void main(String[] args)
{
  //Construct an object of anonymous class
  Measurer m = new Measurer()
  {
    //Class declaration starts here
    public double measure(Object anObject)
    {
      Rectangle aRectangle = (Rectangle) anObject;
      return aRectangle.getWidth() * aRectangle.getHeight();
    }
  };

  double result = Data.average(rectanles, m);
  . . .
}
```

Event Handling

- In an event-driven user interface, the program receives an event whenever the user manipulates an input component.

© Serly Tryapitsyn/iStockphoto.

- User interface **events** include key presses, mouse moves, button clicks, and so on.
- Most programs don't want to be flooded by irrelevant events.
- A program must indicate which events it needs to receive.

Event Handling

Event listeners:

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

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 Handling

- **Example:** A program that prints a message whenever a button is clicked.
- Button listeners must belong to a class that implements the `ActionListener` interface:

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

- Your job is to supply a class whose `actionPerformed` method contains the instructions that you want executed whenever the button is clicked.

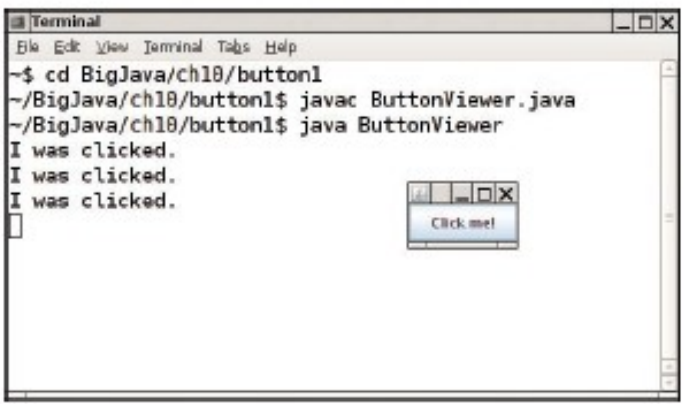


Figure 7 Implementing an Action Listener

section_7_1/ClickListener.java

```
1  import java.awt.event.ActionEvent;
2  import java.awt.event.ActionListener;
3
4  /**
5     An action listener that prints a message.
6  */
7  public class ClickListener implements ActionListener
8  {
9     public void actionPerformed(ActionEvent event)
10    {
11       System.out.println("I was clicked.");
12    }
13 }
```

```
1  import java.awt.event.ActionListener;
2  import javax.swing.JButton;
3  import javax.swing.JFrame;
4  /**
5     This program demonstrates how to install an action listener.
6  */
7  public class ButtonViewer
8  {
9     public static void main(String[] args)
10    {
11       JFrame frame = new JFrame();
12       JButton button = new JButton("Click me!");
13       frame.add(button);
14
15       ActionListener listener = new ClickListener();
16       button.addActionListener(listener);
17       . . .
18    }
19 }
```

Event Handling - Listening to Events

- `event` parameter of `actionPerformed` contains details about the event, such as the time at which it occurred.
- Construct an object of the listener and add it to the button:

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

- Whenever the button is clicked, it calls:

```
listener.actionPerformed(event);
```

- And the message is printed.
- Similar to a callback
- Use a `JButton` component for the button; attach an `ActionListener` to the button.

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);
```

- Advantages
  - 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 instance variables and methods of the surrounding class*

Using Inner Classes for Listeners

- Local variables that are accessed by an inner class method must be declared as `final` (or in Java 8, `effectively final` [not modified after initialized]).
- **Example:** add interest to a bank account whenever a button is clicked:

```
JButton button = new JButton("Add Interest");
frame.add(button);
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);
  }
}
ActionListener listener = new AddInterestListener();
button.addActionListener(listener);
```

section_7_2/InvestmentViewer1.java

```
1  import java.awt.event.ActionEvent;
2  import java.awt.event.ActionListener;
3  import javax.swing.JButton;
4  import javax.swing.JFrame;
5
6  /**
7     This program demonstrates how an action listener can access
8     a variable from a surrounding block.
9  */
10 public class InvestmentViewer1
11 {
12    private static final int FRAME_WIDTH =
13    120;  private static final int
14    FRAME_HEIGHT = 60;
15    private static final double INTEREST_RATE = 10;
16    private static final double INITIAL_BALANCE =
17    1000;
18    public static void main(String[] args)
19    {
20       JFrame frame = new JFrame();
21
22       // The button to trigger the calculation
23       JButton button = new JButton("Add
24       Interest");  frame.add(button);
25
26       // The application adds interest to this bank account
27       final BankAccount account = new
28       BankAccount(INITIAL_BALANCE);
29       class AddInterestListener implements ActionListener
30       {
31          public void actionPerformed(ActionEvent event)
32          {
33             // The listener method accesses the account variable
34             //from the surrounding block
35             double interest = account.getBalance() * INTEREST_RATE / 100;
```

Program Run:

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

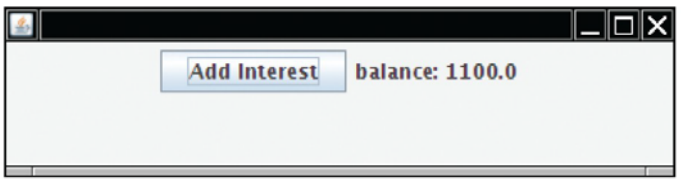
Event Handling

Whenever a button is pressed, the `actionPerformed` method is called on all listeners.

Specify button click actions through classes that implement the `ActionListener` interface.

Building Applications with Buttons

- **Example:** investment viewer program; whenever button is clicked, interest is added, and new balance is displayed:



- 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

- AddInterestListener 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 variables (prior to Java 8, `account` and `label` must be declared `final`).

section_8/InvestmentViewer2.java

```
1  import java.awt.event.ActionEvent;
2  import java.awt.event.ActionListener;
3  import javax.swing.JButton;
4  import javax.swing.JFrame;
5  import javax.swing.JLabel;
6  import javax.swing.JPanel;
7
8  /**
9     This program displays the growth of an investment.
```

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.
- Declare a class that implements the `ActionListener` interface:

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

- To create a timer, specify the frequency of the events and an object of a class that implements the `ActionListener` interface:

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

section_9/RectangleComponent.java

Displays a rectangle that moves

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

```
1 import java.awt.Graphics;
2 import java.awt.Graphics2D;
3 import java.awt.Rectangle;
4 import javax.swing.JComponent;
5
6 /**
7        This component displays a rectangle that can be moved.
8 */
9 public class RectangleComponent extends JComponent
```

section_9/RectangleFrame.java

```
1 import java.awt.event.ActionEvent;
2 import java.awt.event.ActionListener;
3 import javax.swing.JFrame;
4 import javax.swing.Timer;
5
6 /**
7    This frame contains a moving rectangle.
8 */
9 public class RectangleFrame extends JFrame
```

section_9/RectangleViewer.java

```
1  import javax.swing.JFrame;
2
3  /**
4     This program moves the rectangle.
5  */
6  public class RectangleViewer
7  {
8     public static void main(String[] args)
9     {
```

Mouse Events

- Use a mouse listener to capture mouse events.
- Implement the `MouseListener` interface which has five methods:

```
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

- 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 to the mouse location.

section_10/RectangleComponent2.java

First add a `moveRectangle` method to `RectangleComponent`:

```
1 import java.awt.Graphics;
2 import java.awt.Graphics2D;
3 import java.awt.Rectangle;
4 import javax.swing.JComponent;
5
6 /**
7        This component displays a rectangle that can be moved.
8 */
9 public class RectangleComponent2 extends JComponent
```

Mouse Events

- Call `repaint` to tell the component to repaint itself and show the rectangle in its new position.
- When the mouse is pressed,the mouse 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);
   }
   // 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.

RectangleViewer2 Program Run

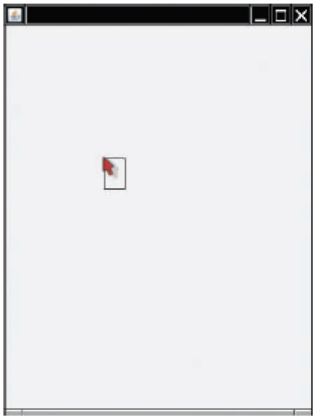


Figure 9 Clicking the Mouse Moves the Rectangle

section_10/RectangleFrame2.java

```
1 import java.awt.event.MouseListener;
2 import java.awt.event.MouseEvent;
3 import javax.swing.JFrame;
4
5 /**
6    This frame contains a moving rectangle.
7 */
8 public class RectangleFrame2 extends JFrame
9 {
```

section_10/RectangleViewer2.java

```
1  import javax.swing.JFrame;
2
3  /**
4     This program displays a rectangle that can be moved with the mouse.
5  */
6  public class RectangleViewer2
7  {
8     public static void main(String[] args)
9     {
```