Interfaces and Polymorphism

Advanced Programming

ICOM 4015

Lecture 10

Reading: Java Concepts Chapter 11

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

Continued…

Chapter Goals

- 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 for timer events

Using Interfaces for Code Reuse

- Use *interface types* to make code more reusable
- In Chap. 7, 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

```
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

Continued…

Using Interfaces for Code Reuse

```
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++;
   }

   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

- 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 Measureable 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;
   }

   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

- **Another example:**

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

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-shaped arrow tip denotes the "uses" relationship or dependency
- **Note that `DataSet` is *decoupled* from `BankAccount` and `Coin`**

Continued...

UML Diagram of Dataset and Related Classes

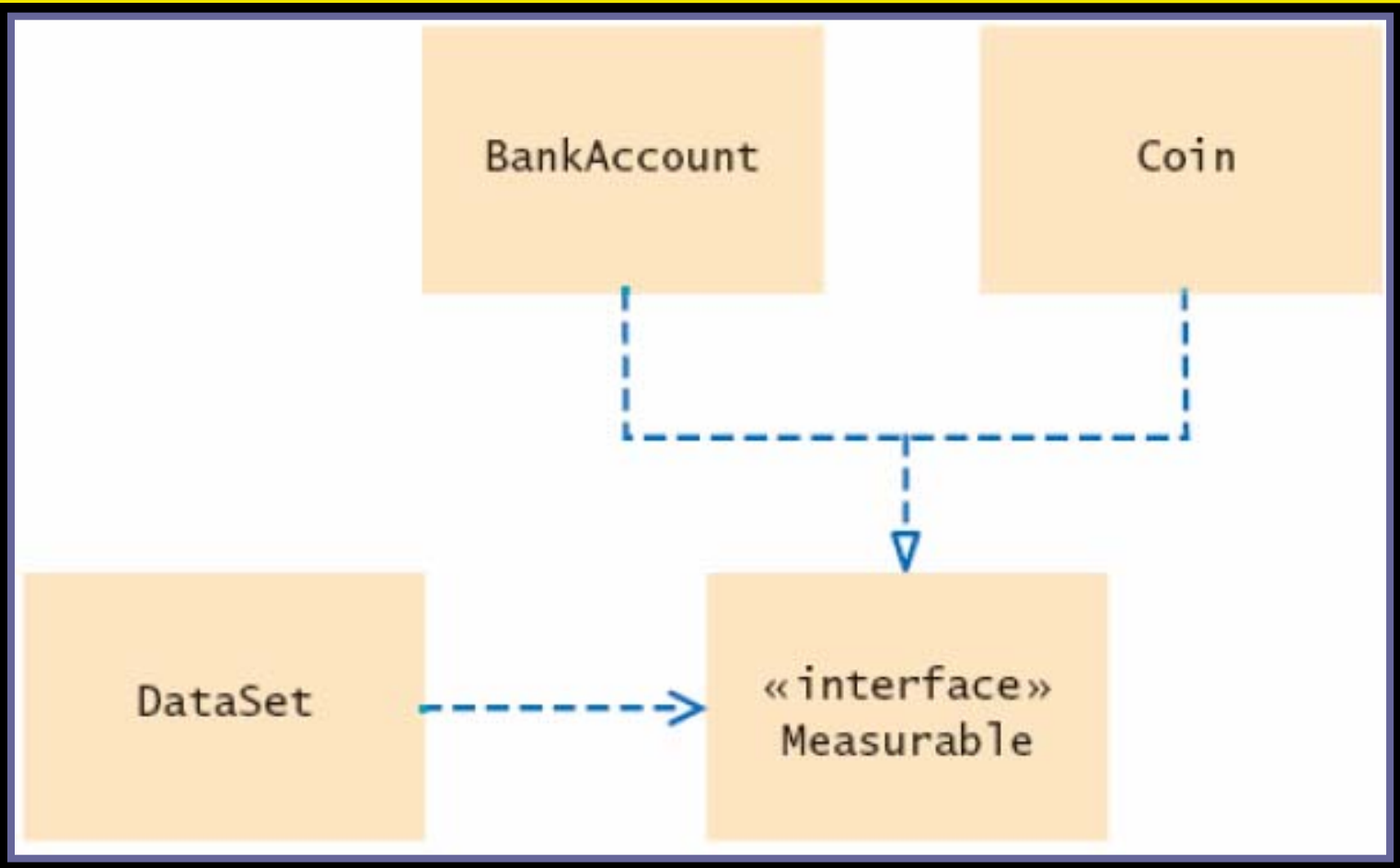

Figure 2:
UML Diagram of `Dataset` Class and the `Classes` that Implement the Measurable Interface

Syntax 11.1: 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 11.2: 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
   }
}
```

Purpose:

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

File DataSetTester.java

```
01: /**
02:    This program tests the DataSet class.
03: */
04: public class DataSetTester
05: {
06:    public static void main(String[] args)
07:    {
08:       DataSet bankData = new DataSet();
09:
10:       bankData.add(new BankAccount(0));
11:       bankData.add(new BankAccount(10000));
12:       bankData.add(new BankAccount(2000));
13:
14:       System.out.println("Average balance = "
15:             + bankData.getAverage());
16:       Measurable max = bankData.getMaximum();
17:       System.out.println("Highest balance = "
18:          + max.getMeasure());
```

Continued...

File DataSetTester.java

```
19:
20:       DataSet coinData = new DataSet();
21:
22:       coinData.add(new Coin(0.25, "quarter"));
23:       coinData.add(new Coin(0.1, "dime"));
24:       coinData.add(new Coin(0.05, "nickel"));
25:
26:       System.out.println("Average coin value = "
27:             + coinData.getAverage());
28:       max = coinData.getMaximum();
29:       System.out.println("Highest coin value = "
30:             + max.getMeasure());
31:    }
32: }
```


Continued…

File DataSetTester.java

Output:

```
Average balance = 4000.0
Highest balance = 10000.0
Average coin value = 0.13333333333333333
Highest coin value = 0.25
```

Self Check

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

Answers

1. It must implement the `Measurable` interface, and its `getMeasure` method must return the population
2. 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
```


Continued…

Converting Between Class and Interface Types

- 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
```


Continued…

Casts

- 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();
```

- If you are wrong and `max` isn't a coin, the JRE thows an exception

Casts

- **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

1. Can you use a cast (`BankAccount`) `x` to convert a `Measurable` variable `x` to a `BankAccount` reference?
2. If both `BankAccount` and `Coin` implement the `Measurable` interface, can a `Coin` reference be converted to a `BankAccount` reference?

Answers

1. Only if `x` actually refers to a `BankAccount` object.
2. 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

Continued…

Polymorphism

- 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

Continued…

Polymorphism

- **Called *late binding*: resolved at runtime**
- **Different from overloading; overloading is resolved by the compiler (*early binding*)**

Self Check

1. Why is it impossible to construct a `Measurable` object?
2. Why can you nevertheless declare a variable whose type is `Measurable`?
3. What do overloading and polymorphism have in common? Where do they differ?

Answers

1. `Measurable` is an interface. Interfaces have no fields and no method implementations.
2. That variable never refers to a `Measurable` object. It refers to an object of some class–a class that implements the `Measurable` interface.

Continued…

Answers

3. 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

Using Interfaces for Callbacks

- In previous `DataSet` implementation, responsibility of measuring lies with the added objects themselves
- 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;
   }
}
```

Using Interfaces for Callbacks

- Must cast from `Object` to `Rectangle`

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

- 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

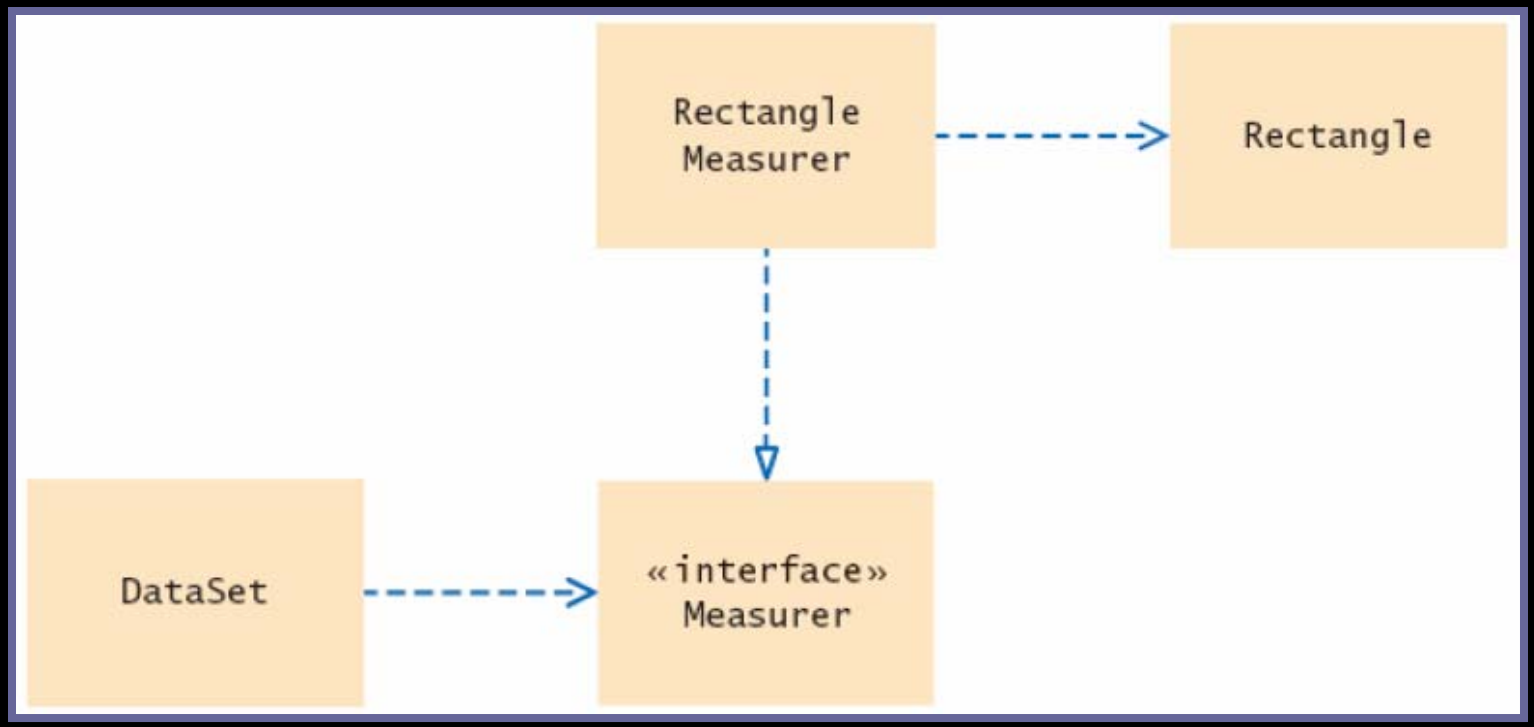

Figure 2:
UML Diagram of the `DataSet` Class and the `Measurer` Interface

File DataSet.java

```
01: /**
02:    Computes the average of a set of data values.
03: */
04: public class DataSet
05: {
06:    /**
07:       Constructs an empty data set with a given measurer.
08:       @param aMeasurer the measurer that is used to
             // measure data values
09:    */
10:    public DataSet(Measurer aMeasurer)
11:    {
12:       sum = 0;
13:       count = 0;
14:       maximum = null;
15:       measurer = aMeasurer;
16:    }
17:
```

Continued…

File DataSet.java

```
18:    /**
19:       Adds a data value to the data set.
20:       @param x a data value
21:    */
22:    public void add(Object x)
23:    {
24:       sum = sum + measurer.measure(x);
25:       if (count == 0
26:             || measurer.measure(maximum)
                      < measurer.measure(x))
27:          maximum = x;
28:       count++;
29:    }
30:
31:    /**
32:       Gets the average of the added data.
33:       @return the average or 0 if no data has been added
34:    */
```

Continued…

File DataSet.java

```
35:    public double getAverage()
36:    {
37:       if (count == 0) return 0;
38:       else return sum / count;
39:    }
40:
41:    /**
42:       Gets the largest of the added data.
43:       @return the maximum or 0 if no data has been added
44:    */
45:    public Object getMaximum()
46:    {
47:       return maximum;
48:    }
49:
```

Continued...

File DataSet.java

```
50:    private double sum;
51:    private Object maximum;
52:    private int count;
53:    private Measurer measurer;
54: }
```

File DataSetTester2.java

```
01: import java.awt.Rectangle;
02:
03: /**
04:    This program demonstrates the use of a Measurer.
05: */
06: public class DataSetTester2
07: {
08:    public static void main(String[] args)
09:    {
10:       Measurer m = new RectangleMeasurer();
11:
12:       DataSet data = new DataSet(m);
13:
14:       data.add(new Rectangle(5, 10, 20, 30));
15:       data.add(new Rectangle(10, 20, 30, 40));
16:       data.add(new Rectangle(20, 30, 5, 10));
17:
```

Continued…

File DataSetTester2.java

```
18:         System.out.println("Average area = " + data.getAverage());
19:         Rectangle max = (Rectangle) data.getMaximum();
20:         System.out.println("Maximum area rectangle = " + max);
21:     }
22: }
```

File Measurer.java

```
01: /**
02:    Describes any class whose objects can measure other objects.
03: */
04: public interface Measurer
05: {
06:    /**
07:       Computes the measure of an object.
08:       @param anObject the object to be measured
09:       @return the measure
10:    */
11:    double measure(Object anObject);
12: }
```

File RectangleMeasurer.java

```
01: import java.awt.Rectangle;
02:
03: /**
04:     Objects of this class measure rectangles by area.
05: */
06: public class RectangleMeasurer implements Measurer
07: {
08:    public double measure(Object anObject)
09:    {
10:       Rectangle aRectangle = (Rectangle) anObject;
11:       double area = aRectangle.getWidth()
                * aRectangle.getHeight();
12:       return area;
13:    }
14: }
15:
```


Continued…

File RectangleMeasurer.java

Output:

```
Average area = 616.6666666666666
Maximum area rectangle = java.awt.Rectangle[x=10,y=20,
   // width=30,height=40]
```

Self Check

1. Suppose you want to use the `DataSet` class of Section 11.1 to find the longest `String` from a set of inputs. Why can't this work?
2. How can you use the `DataSet` class of this section to find the longest `String` from a set of inputs?
3. Why does the `measure` method of the `Measurer` interface have one more parameter than the `getMeasure` method of the `Measurable` interface?

Answers

1. The `String` class doesn't implement the `Measurable` interface.
2. Implement a class `StringMeasurer` that implements the `Measurer` interface.
3. 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

- 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 11.3: 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 11.3: 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

File FileTester3.java

```
01: import java.awt.Rectangle;
02:
03: /**
04:    This program demonstrates the use of a Measurer.
05: */
06: public class DataSetTester3
07: {
08:    public static void main(String[] args)
09:    {
10:       class RectangleMeasurer implements Measurer
11:       {
12:          public double measure(Object anObject)
13:          {
14:             Rectangle aRectangle = (Rectangle) anObject;
15:             double area
16:                   = aRectangle.getWidth()
                            * aRectangle.getHeight();
17:             return area;
```

Continued…

File FileTester3.java

```
18:          }
19:       }
20:
21:       Measurer m = new RectangleMeasurer();
22:
23:       DataSet data = new DataSet(m);
24:
25:       data.add(new Rectangle(5, 10, 20, 30));
26:       data.add(new Rectangle(10, 20, 30, 40));
27:       data.add(new Rectangle(20, 30, 5, 10));
28:
29:       System.out.println("Average area = " + data.getAverage());
30:       Rectangle max = (Rectangle) data.getMaximum();
31:       System.out.println("Maximum area rectangle = " + max);
32:    }
33: }
```

Self Test

1. Why would you use an inner class instead of a regular class?
2. How many class files are produced when you compile the `DataSetTester3` program?

Answers

1. Inner classes are convenient for insignificant classes. Also, their methods can access variables and fields from the surrounding scope.
2. Four: one for the outer class, one for the inner class, and two for the `DataSet` and `Measurer` classes.

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

- **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
   }
}
```


Continued…

Processing Timer Events

- **Add listener to timer**

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

Example: Countdown

- Example: a timer that counts down to zero

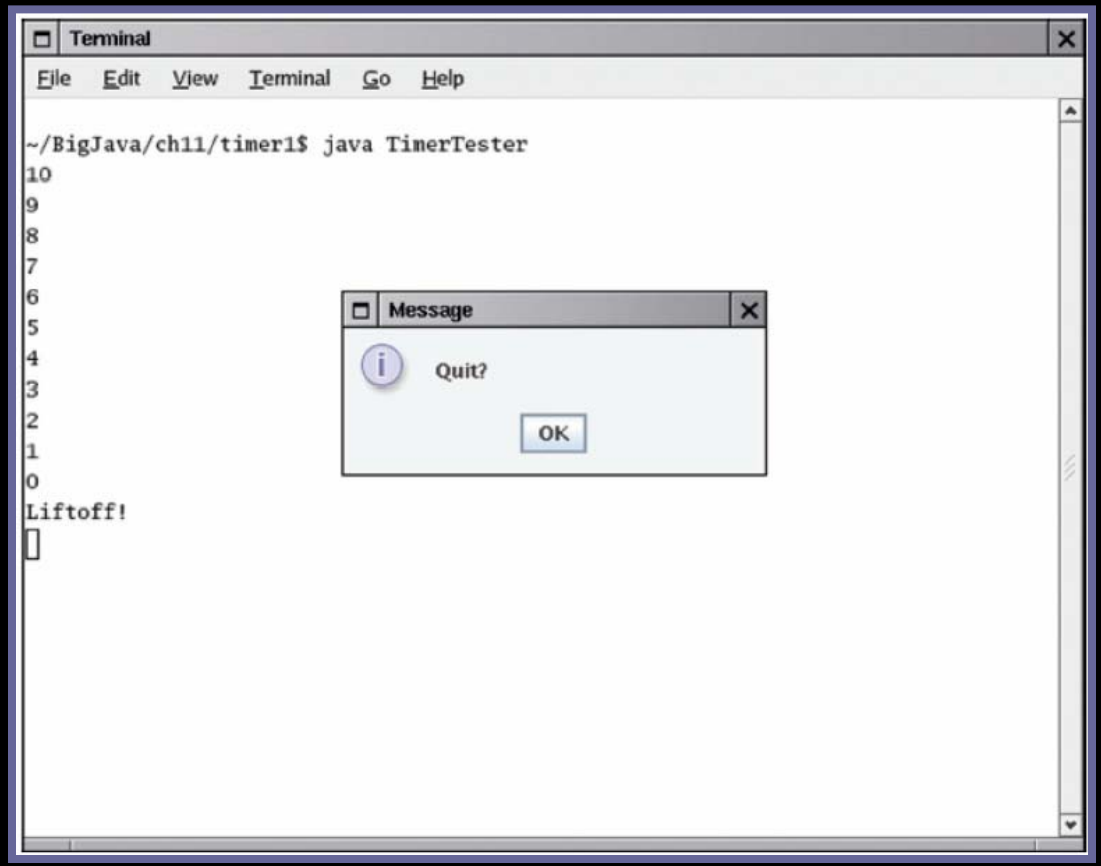

Figure 3:
Running the `TimeTester` Program

File TimeTester.java

```
01: import java.awt.event.ActionEvent;
02: import java.awt.event.ActionListener;
03: import javax.swing.JOptionPane;
04: import javax.swing.Timer;
05:
06: /**
07:    This program tests the Timer class.
08: */
09: public class TimerTester
10: {
11:    public static void main(String[] args)
12:    {
13:       class CountDown implements ActionListener
14:       {
15:          public CountDown(int initialCount)
16:          {
17:             count = initialCount;
18:          }
```

Continued…

File TimeTester.java

```
19:
20:          public void actionPerformed(ActionEvent event)
21:          {
22:             if (count >= 0)
23:                System.out.println(count);
24:             if (count == 0)
25:                System.out.println("Liftoff!");
26:             count--;
27:          }
28:
29:          private int count;
30:       }
31:
32:       CountDown listener = new CountDown(10);
33:
34:       final int DELAY = 1000; // Milliseconds between
             // timer ticks
```

Continued…

File TimeTester.java

```
35:         Timer t = new Timer(DELAY, listener);
36:         t.start();
37:
38:         JOptionPane.showMessageDialog(null, "Quit?");
39:         System.exit(0);
40:     }
41: }
```

Self Check

1. Why does a timer require a listener object?
2. How many times is the `actionPerformed` method called in the preceding program?

Answers

1. The timer needs to call some method whenever the time interval expires. It calls the `actionPerformed` method of the listener object.
2. It depends. The method is called once per second. The first eleven times, it prints a message. The remaining times, it exits silently. The timer is only terminated when the user quits the program.

Accessing Surrounding Variables

- Methods of inner classes can access variables that are defined in surrounding scope
- Useful when implementing event handlers
- Example: Animation
  Ten times per second, we will move a shape to a different position

Continued…

Accessing Surrounding Variables

```
class Mover implements ActionListener
{
   public void actionPerformed(ActionEvent event)
   {
      // Move the rectangle
   }
}

ActionListener listener = new Mover();
final int DELAY = 100;
// Milliseconds between timer ticks
Timer t = new Timer(DELAY, listener);
t.start();
```

Accessing Surrounding Variables

- The `actionPerformed` method can access variables from the surrounding scope, like this:

```
public static void main(String[] args)
{
   . . .
   final Rectangle box = new Rectangle(5, 10, 20, 30);

   class Mover implements ActionListener
   {
      public void actionPerformed(ActionEvent event)
      {
         // Move the rectangle
         box.translate(1, 1);
      }
   }
   . . .
}
```

Accessing Surrounding Variables

- Local variables that are accessed by an inner-class method must be declared as final
- Inner class can access fields of surrounding class that belong to the object that constructed the inner class object
- An inner class object created inside a static method can only access static surrounding fields

File TimeTester2.java

```
01: import java.awt.Rectangle;
02: import java.awt.event.ActionEvent;
03: import java.awt.event.ActionListener;
04: import javax.swing.JOptionPane;
05: import javax.swing.Timer;
06:
07: /**
08:    This program uses a timer to move a rectangle once per second.
09: */
10: public class TimerTester2
11: {
12:    public static void main(String[] args)
13:    {
14:       final Rectangle box = new Rectangle(5, 10, 20, 30);
15:
16:       class Mover implements ActionListener
17:       {
```

Continued…

File TimeTester2.java

```
18:          public void actionPerformed(ActionEvent event)
19:          {
20:             box.translate(1, 1);
21:             System.out.println(box);
22:          }
23:       }
24:
25:       ActionListener listener = new Mover();
26:
27:       final int DELAY = 100; // Milliseconds between timer ticks
28:       Timer t = new Timer(DELAY, listener);
29:       t.start();
30:
31:       JOptionPane.showMessageDialog(null, "Quit?");
32:       System.out.println("Last box position: " + box);
33:       System.exit(0);
34:    }
35: }
```

File TimeTester2.java

Output:

```
java.awt.Rectangle[x=6,y=11,width=20,height=30]
java.awt.Rectangle[x=7,y=12,width=20,height=30]
java.awt.Rectangle[x=8,y=13,width=20,height=30] . . .
java.awt.Rectangle[x=28,y=33,width=20,height=30]
java.awt.Rectangle[x=29,y=34,width=20,height=30]
Last box position: java.awt.Rectangle[x=29,y=34,width=20,height=30]
```

Self Check

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

Answers

1. Direct access is simpler than the alternative–passing the variable as a parameter to a constructor or method.
2. The local variable must be declared as final.

Operating Systems

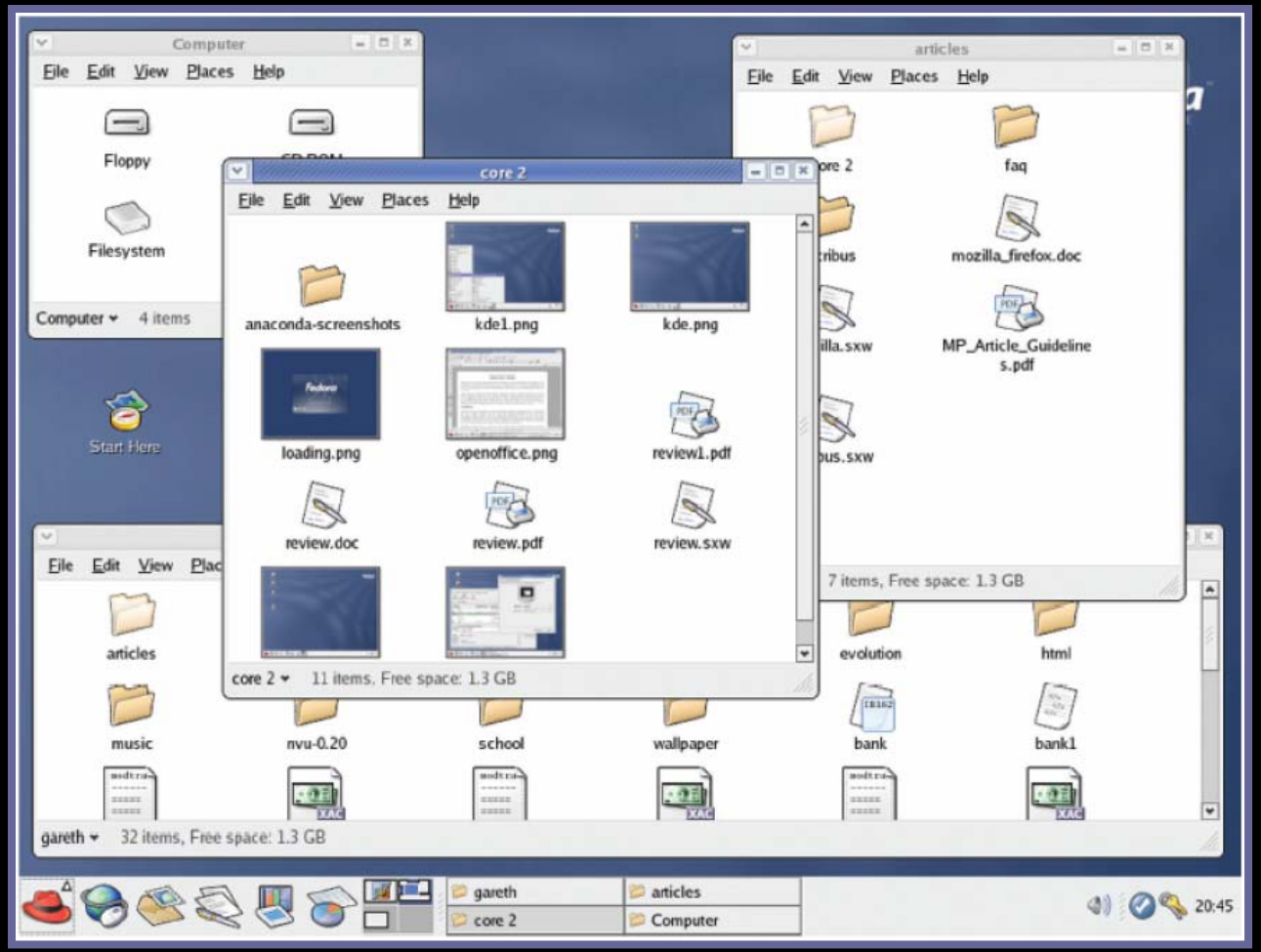

Figure 4:
A Graphical Software Environment for the Linux Operating System