



# D06

# PROGRAMMING with JAVA

## Ch11 – Interfaces & 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** for timer events

# Using Interfaces for Code Reuse

---



It is often possible to make code more general and more reusable by focusing on the essential operations that are carried out. **Interface types** are used to express these common operations

- In Chap. 7 (Iteration), we created the `DataSet` class to compute the average and maximum of a set of input values
- However, the `DataSet` class was suitable only for computing the average and maximum of a set of *numbers*. What if we want to find the average and maximum of a set of `BankAccount` values? Then, we would have to modify the class...
- 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 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

```
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, but details of measurement differ
- Classes could agree on a method `getMeasure` that obtains the measure to be used in the data analysis. For bank accounts, `getMeasure` returns the balance. For coins, `getMeasure` returns the coin value, and so on
- Then, 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`? Ideally, `x` should refer to any class that has a `getMeasure` method



In Java, an **interface type** is used to specify required operations. We will define an interface type that we call `Measurable`:

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

- The interface declaration lists all methods (and their signatures) that the interface type requires
- Note that the `Measurable` type is not a type in the standard library –it is a type that was created specifically for making the `DataSet` class more reusable

# 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**; that is, they have a name, parameters, and a return type, but they don't have an implementation
  - All methods in an interface type are automatically public
  - An interface type does not have instance fields
- Now we can use the interface type `Measurable` to declare the variables `x` and `maximum`:

# 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;
    }

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

# Implementing an Interface Type

- This `DataSet` class is usable for analyzing objects of any class that **implements** the `Measurable` interface. A class implements an interface type if it declares the interface in an `implements` clause. It should then implement the method or methods that the interface requires:

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

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

- A class can implement more than one interface type. The class must then define all the methods that are required by all the interfaces it implements
- Note that the class must declare the method as `public`, whereas the interface need not –all methods in an interface are public
- The `Measurable` interface expresses what all measurable objects have in common. This commonality makes the `DataSet` class reusable. Objects of the `DataSet` can be used to analyze collections of objects of any class that implements the `Measurable` interface

# 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

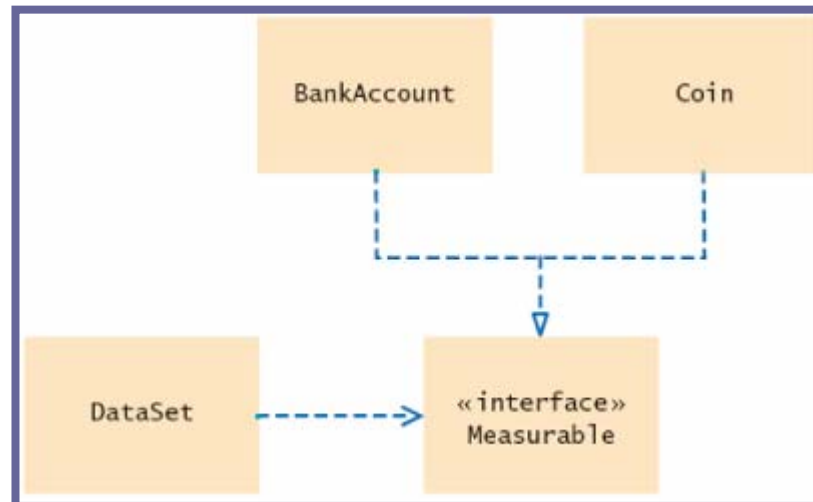


Figure 2:  
UML Diagram of DataSet  
Class and the Classes that  
Implement the Measurable  
Interface

# 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: }
```

## Output:

```
Average balance = 4000.0
Highest balance = 10000.0
Average coin value = 0.13333333333333333333
Highest coin value = 0.25
```

# Common Errors & Advanced Topics

---

- It is a common error to forget the `public` keyword when defining a method from an interface



Interfaces cannot have instance fields, but it is legal to specify constants. When defining a constant in an interface, you can (and should) omit the keywords `public static final`, because all fields in an interface are automatically `public static final`

# 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
```

- Thus, when you have an object variable of type `Measurable`, you don't actually know the exact type of the object to which `x` refers. All you know is that the object has a `getMeasure` method
- However, you cannot convert between unrelated types:

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

That assignment is an error, because the `Rectangle` class doesn't implement the `Measurable` interface

*Continued...*

# Converting Between Class and Interface Types

- Occasionally, it happens that you convert an object to an interface reference and you need to convert it back. This happens in the `getMaximum` method of the `DataSet` class. The `DataSet` stores the object with the largest measure, as a `Measurable` reference:

```
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
```

- Now what can you do with the `max` reference? You know it refers to a `Coin` object, but the compiler doesn't. For example, you cannot call the `getName` method:

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

That call is an error, because the `Measurable` type has no `getName` method

- However, as long as you are absolutely sure that `max` refers to a `Coin` object, you can use the **cast** notation to convert it back:

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

If you are wrong, your program will throw an exception and terminate

*Continued...*

# Converting Between Class and Interface Types

- There is one big difference between casting of number types and casting of class types:
  - When casting number types, you **lose information**, and you use the cast to tell the compiler that you agree to the information loss
  - When casting object types, on the other hand, you take a risk of causing an **exception**, and you tell the compiler that you agree to that risk



You can define variables whose type is an interface, for example:

```
Measurable x;
```

However, **you can never construct an interface**:

```
Measurable x = new Measurable(); // ERROR
```

Interfaces aren't classes. There are no objects whose types are interfaces. If an interface variable refers to an object, then the object must belong to some class –a class that implements the interface:

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

# Self Check

---

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

# Answers

---

3. Only if `x` actually refers to a `BankAccount` object.
4. 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

- When multiple classes implement the same interface, each class implements the methods of the interface in different ways. How is the correct method executed when the interface method is invoked?
- It is perfectly legal to have variables whose type is an interface, such as:

```
Measurable x;
```

- 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:

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

- What can you do with an interface variable, given that you don't know the class of the object that it references? You can invoke the methods of the interface:

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

- But... which `getMeasure` method is called?

*Continued...*

# Polymorphism

---

- How did the correct method get called if the caller didn't even know the exact class to which `x` belongs?
- The Java virtual machine makes a special effort to locate the correct method that belongs to the class of the actual object. That is, if `x` refers to a `BankAccount` object, then the `BankAccount.getMeasure` method is called. If `x` refers to a `Coin` object, then the `Coin.getMeasure` method is called



The principle that the actual type of the object determines the method to be called is called **polymorphism**. The same computation works for objects of many shapes, and adapts itself to the nature of the object. In Java, all instance methods are polymorphic

*Continued...*

# Polymorphism

- You have already seen another case in which the same method name can refer to different methods, namely when a method name is **overloaded**: that is, when a single class has several methods with the same name but different parameter types. Then, the compiler selects the appropriate method when compiling the program, simply by looking at the types of the parameters:

```
account = new BankAccount ( ) ; // Compiler selects BankAccount()  
account = new BankAccount ( 10000 ) ; // Compiler selects BankAccount(double)
```



There is an important difference between polymorphism and overloading. The compiler picks an overloaded method when translating the program, before the program ever runs (**early binding**). However, when selecting the appropriate `getMeasure` method in a call `x.getMeasure()`, the compiler does not make any decision when translating the method. The program has to run before anyone can know what is stored in `x`. Therefore, the virtual machine, and not the compiler, selects the appropriate method (**late binding**).

# Self Check

---

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

# Answers

---

5. `Measurable` is an interface. Interfaces have no fields and no method implementations.
6. That variable never refers to a `Measurable` object. It refers to an object of some class—a class that implements the `Measurable` interface.
7. 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.

# Inner Classes

- When you have a class that serves a very limited purpose, you can declare the class inside the method that needs it:

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

- Such a class is called an **inner class**. An inner class is any class that is defined inside another class. Since an inner class inside a method is not a publicly accessible feature, you don't need to document it as thoroughly

# Inner Classes

---

- You can also define an inner class inside an enclosing class, but outside of its methods. Then the inner class is available to all methods of the enclosing class
- When you compile the source files for a program that uses inner classes, you will find that the inner classes are stored in files with curious names

```
DataSetTester$1$RectangleMeasurer.class
```

The exact names aren't important. The point is that the compiler turns an inner class into a regular class file

# 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

# Self Test

---

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

# Answers

---

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

# Processing Timer Events

- Timer event handling uses interfaces in the same way that events triggered by buttons and menus in a graphical program do



The **Timer** class in the `javax.swing` package generates a sequence of events, spaced apart at even time intervals. This is useful whenever you want to have an object updated in regular intervals

- When a timer event occurs, the timer must call some method. The designers of the `Timer` class had no idea how you would want to use the `Timer`. Therefore, they simply chose an interface, called `ActionListener`, with a method that the timer can call:

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

A **timer** generates timer events at fixed intervals. An **event listener** is notified when a particular event occurs.

*Continued...*

# Processing Timer Events

- When you use a timer, you need to define a class that implements the `ActionListener` interface. Place whatever action you want to occur inside the `actionPerformed` method. Construct an object of that class. Pass it to the `Timer` constructor. Finally, start the timer:

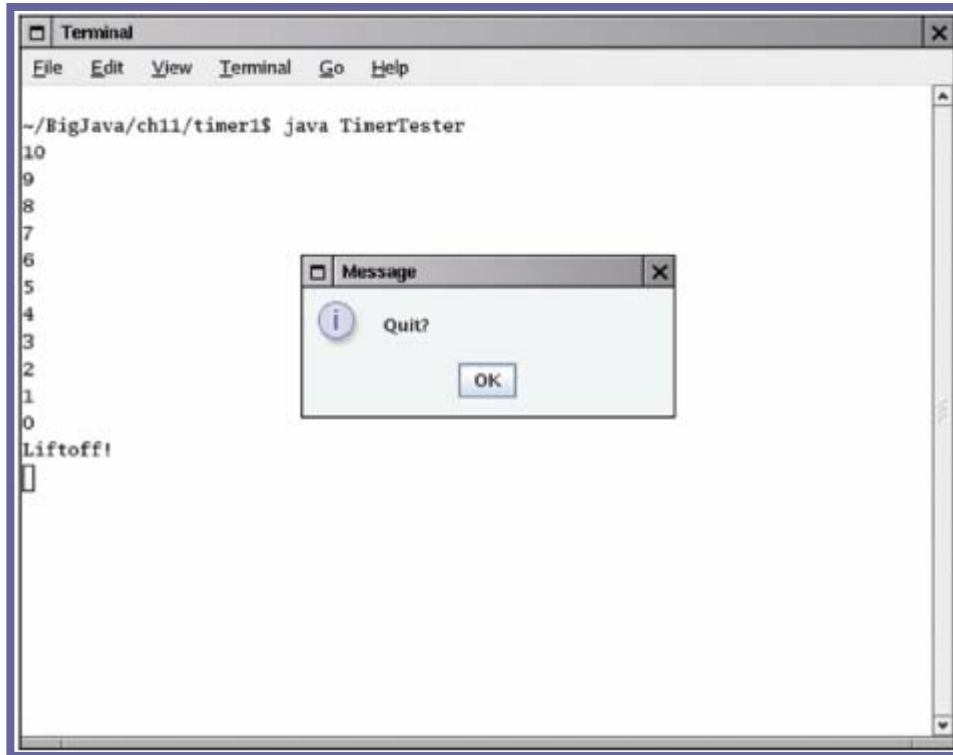
```
class MyListener implements ActionListener
{
    public void actionPerformed(ActionEvent event)
    {
        // This action will be executed at each timer event
        Place listener action here
    }
}
```

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

- Then the timer calls the `actionPerformed` method of the listener object every `interval` milliseconds
- When you implement an interface, you must define each method exactly as it is specified in the interface. Accidentally making small changes to the parameter or return types is a common error

# Example: Countdown

- Example: a timer that counts down to zero:



```
Terminal
File Edit View Terminal Go Help

~/BigJava/ch11/timer1$ java TinerTester
10
9
8
7
6
5
4
3
2
1
0
Liftoff!
█

Message
Quit?
OK
```

Figure 3:  
Running the TimeTester Program

Unlike a `for` loop, which would print all lines immediately, there is a one-second delay between decrements. To keep the program alive after setting up the timer, the program displays a message dialog box

# 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

---

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

# Answers

---

13. The timer needs to call some method whenever the time interval expires. It calls the `actionPerformed` method of the listener object.
14. 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

---

- An attractive feature of inner classes is that their methods can access variables that are defined in surrounding blocks. In this regard, method definitions of inner classes behave similarly to nested blocks
- Except for some technical restrictions, the methods of an inner class can access the variables from the enclosing scope. It allows the inner class to access variables without having to pass them as constructor or method parameters

# 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

---

- There is a technical wrinkle. An inner class can access surrounding **local variables** only if they are declared as **final**. That sounds like a restriction, but it is usually not an issue in practice (an object variable is `final` when the variable always refers to the same object: the state of the object can change, but the variable can't refer to a different object)
- An inner class can also access **fields** of the surrounding class, again with a restriction. The field must belong to the object that constructed the inner class object
- If the inner class object was created inside a **static** method, it 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: }
```

## 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

---

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

# Answers

---

15. Direct access is simpler than the alternative—passing the variable as a parameter to a constructor or method.
16. The local variable must be declared as final.

# Chapter Summary

---

- Use **interface types** to make code more reusable
- A Java interface type declares a set of methods and their signatures
- Unlike a class, an interface type provides no implementation
- Use the **implements** keyword to indicate that a class implements an interface type
- Interfaces can reduce the coupling between classes
- You can convert from a class type to an interface type, provided the class implements the interface
- You need a **cast** to convert from an interface type to a class type

*Continued*

# Chapter Summary

---

- **Polymorphism** denotes the principle that behavior can vary depending on the actual type of an object
- **Early binding** of methods occurs if the compiler selects a method from several possible candidates. **Late binding** occurs if the method selection takes place when the program runs
- An **inner class** is declared inside another class. Inner classes are commonly used for tactical classes that should not be visible elsewhere in a program
- A **timer** generates timer events at fixed intervals
- An **event listener** is notified when a particular event occurs
- Methods of an inner class can access variables from the surrounding scope
- Local variables that are accessed by an inner-class method must be declared as **final**