Chapter 9

Inheritance and Interfaces

Polymorphism for Classes and Interfaces

Outline

What is inheritance?
Rules for declaring subclasses
Simple examples of subclasses
Polymorphism
Abstract classes and polymorphism
The \texttt{Object} Class
Final classes
Interfaces
Multiple interfaces
Implementing the \texttt{Shape} interface
Turtle graphics class
Writing turtle graphics programs
Numerical applications of interfaces
9.1 Introduction

There are two hierarchies in object-oriented programming that help manage the complexity of large software systems.

The first is the object hierarchy considered in Chapter 4, that uses aggregation (composition) to construct more complex objects in terms of simpler ones, which in turn can be used to construct even more complex objects, and so on.

The second is the class hierarchy defined by inheritance. With inheritance we can define a subclass of a class (the superclass) that inherits all the functionality (methods) of the superclass. Some of these methods can be retained, or they can be overridden by providing new versions in the subclass, and new methods (functionality) can be added. This leads to a class inheritance hierarchy. The Object class is implicitly at the top of any inheritance hierarchy.

Java has only single-inheritance hierarchies for classes, which means that a subclass can inherit from only one superclass, but there are special important kinds of classes called interfaces, which we will also discuss, that permit a very useful form of multiple inheritance.

Abstract classes are also introduced. An abstract class declares one or more abstract methods but not their implementations and possibly other methods that do have implementations. It is up to the subclasses of an abstract class to provide implementations for the abstract methods. An Employee class hierarchy will be used as an example.

The related and important concept of polymorphism is also introduced. This permits all objects from classes in an inheritance hierarchy to be considered of the “same” type, which results in a uniform processing of objects in the hierarchy, without regard to the particular subclasses the objects really belong to, using the polymorphic methods of the class. The particular class that the object belongs to is not determined until run-time.

A polymorphic method is a method appearing in each subclass in an inheritance hierarchy with the same prototype. Each subclass can have its own version of this method. For an abstract class any abstract method is polymorphic. As an example we show how methods in an Employee hierarchy can be used to process employee salaries polymorphically.

A more general and often more important form of polymorphism is also possible using interfaces. An interface is like a special kind of abstract class that declares only method prototypes (no implementations). One can also have interface hierarchies where a subinterface inherits from a superinterface.

An important example is the Shape interface for defining geometrical shapes, drawing, and filling them (see Chapter 5). We will show how to make our own Shape objects.

Any class that implements the methods of an interface is said to implement the interface. Unlike classes interfaces also permit a useful form of multiple inheritance since it is possible for a class to implement several interfaces.

The difference compared to an inheritance hierarchy is that the set of classes that implement a particular interface do not need to be related in any other way. In particular they do not need to form a class inheritance hierarchy. Nevertheless, like classes in an inheritance hierarchy, they can be considered to be of the “same type”, namely the interface type. This leads to a more general form of polymorphism within this set of classes that implement the interface.
9.2 What is inheritance?

Inheritance defines a relationship between two classes. One is called the superclass and the other is called the subclass. Sometimes the superclass is called the parent class and the subclass is called the child class. This relationship sets up an inheritance hierarchy since a superclass can be a subclass of another class and so on until a class at the top of the hierarchy is reached that is the superclass of all lower classes. We say that the subclass inherits from the superclass. In Java the special Object class is at the top of all inheritance hierarchies. It is the ultimate superclass of all Java classes in the sense that an object from any class is a kind of Object.

We also express the superclass-subclass relationship in Java by saying that one class (the subclass) extends the other class (the superclass). We have already seen examples of inheritance since our classes in Chapter 5 all extend the JPanel class. Most of the complex functionality of displaying a window containing our graphics output was provided by JPanel and its superclasses and we did not need to understand how it works.

The importance of the inheritance hierarchy in the management of complex software is that each subclass can be constructed incrementally from its immediate superclass. This promotes code reuse since a subclass only specifies how its objects differ from those of the parent class. Thus, we take a given class and extend it to provide some additional functionality which can be specified in the subclass in three basic ways: (1) declare new data fields, (2) declare new methods, and (3) provide new versions of existing superclass methods (called overriding a superclass method).

Of course all the public methods of the superclass that were not overridden in the subclass are automatically available in the subclass.

**EXAMPLE 9.1** (Domestic animal hierarchy) We are familiar with many hierarchies such as family trees and classification systems in biology. Let us consider the part of the animal kingdom that contains domestic animals. At the top of this hierarchy we have a class called DomesticAnimal which specifies features common to all domestic animals. There are many subclasses. Two important ones are the Dog and Cat classes. These three classes are called abstract classes since they do not describe real animals. There are no DomesticAnimal, Dog, or Cat objects, only dogs or cats of particular breeds such as Terrier or Persian. Dogs and cats in each subclass have features which distinguish them from dogs and cats in other subclasses. For example, Cheshire cats are always grinning and they can make themselves invisible, unlike cats in other classes. A part of the DomesticAnimal hierarchy is shown in Figure 9.1 as a tree diagram.

**EXAMPLE 9.2** (Bank account inheritance hierarchy) The BankAccount class used in Chapter 9 has instance data fields for an account number, owner name, and balance. Suppose we want to consider bank accounts that have a joint owner. We can extend the BankAccount class to obtain a subclass called JointBankAccount. This class provides a new instance data field for the joint owner and a new method called getJointName to return the joint owner name. Also, the toString method needs to be overridden to include the new data field. All other methods from the BankAccount class, such as withdraw and deposit, are automatically available in the subclass and do not need to be overridden. Extending the existing class is a lot easier than writing a completely new class from scratch that has considerable overlap with the existing BankAccount class.

The bank account hierarchy diagram is shown in Figure 9.2 which uses a more compact way to
Inheritance and Interfaces

Figure 9.1: Part of the domestic animal inheritance hierarchy

Figure 9.2: Bank account inheritance hierarchy

represent tree diagrams than Figure 9.1. The diagram shows that JointBankAccount is a subclass of BankAccount which is a subclass of Object. Since the Object class is always at the top of any hierarchy we won’t normally show it on inheritance diagrams.

EXAMPLE 9.3 (Employee inheritance hierarchy) We can classify the various kinds of employees in a company using inheritance. At the top of the hierarchy is a class called Employee that represents everything common to all employees, such as name, employee number, and the date the employee was hired. This class is called the base class. In this example it is also a generic or abstract class. To obtain real employee classes we need to make subclasses for each kind of employee. For example, different kinds of real employees can be distinguished by the method for calculating their monthly salary: a Manager object has a fixed monthly salary and deductions, an HourlyWorker has hours worked, an hourly rate and deductions, a PartTimeWorker is the same but with no deductions, and a CommissionWorker has a fixed monthly salary and deductions, plus a commission that is a certain percentage of monthly sales. One way to design the class hierarchy is shown in Figure 9.3.

9.2.1 The “is-a” and “has-a” relationships

The aggregation and class inheritance hierarchies can be used to define two relationships between objects.

Inheritance is often called the “is-a” or “is-a-type-of” relationship. For example, in the bank account hierarchy a JointBankAccount object is a kind of BankAccount object. In the employee hierarchy a Manager object is a kind of Employee object. In the DomesticAnimal hierarchy a Terrier is a kind of Dog and a Dog is a kind of DomesticAnimal.
9.3 Rules for declaring subclasses

Aggregation is often called the “has-a” relationship. For example, in the *Circle* class from Chapter 4, page 122 which was used to illustrate aggregation, we say that a *Circle* object “has a” *Point* object, namely the center of the circle.

Deciding which of the two hierarchies to use in a given situation can be determined by stating relationships between classes and objects in terms of the “is-a” and ”has-a” relationships. For example, it does not make sense to say that a *Circle* object “is a” *Point* object.

### 9.3 Rules for declaring subclasses

In Java the keyword *extends* is used to denote the subclass relationship. To indicate that a class with name *SubclassName* is a subclass of *ClassName* we use a class declaration whose template is shown in Figure 9.4. The first line is like a normal class declaration except for the keyword *extends* which is followed by the name of the class to be extended (the superclass). As mentioned above, the subclass declaration only specifies how the subclass differs from the superclass according to the following basic rules (you can refer back to these rules as you read the Chapter):
1. A data field of the superclass is automatically a data field of a subclass. It is an error to declare it again in a subclass. Direct access to superclass data fields by a subclass follows the rules

   (a) A **public** data field can be directly accessed by any class, subclass or not.
   (b) A **private** data field can never be directly accessed by any other class.
   (c) A **protected** data field can be directly accessed by a subclass but not by any class outside the hierarchy. Protected means public for subclasses and private for other classes.

2. A subclass may declare new data fields.

3. Superclass constructors are never inherited so a subclass must provide its own constructors. In doing so, a subclass constructor may call a superclass constructor, as its first statement, using the syntax **super**(*actualArgList*) to construct the superclass part of an object.

4. A subclass may declare new methods.

5. A subclass may declare a new version of any public or protected superclass method to provide additional or new functionality not provided by the superclass method. This is called **method overriding**. In doing so, a subclass method can call the superclass version of the method using the syntax

   **super.methodName** (*actualArgList*).

6. Public or protected superclass methods that are not overridden by the subclass are automatically available to the subclass and are never declared again in the subclass.

9.4 **Simple examples of subclasses**

We now consider some simple examples to illustrate the rules.

9.4.1 **Graphics programs**

The simple graphics classes in Chapter 5 use inheritance. Recall that each graphics class had the structure

```java
public class MyGraphicsClass extends JPanel {
    public void paintComponent(Graphics g) {
        super.paintComponent(g)
        Graphics2D g2D = (Graphics2D) g;
        // additional statements to draw on the panel
    }
}
```
Here we are extending a class called JPanel which is part of the class hierarchy shown in Figure 9.5. Our class, indicated by MyGraphicsClass, is shown at the bottom of the hierarchy. The reason that our graphics classes were fairly small is that most of the work of supporting the complex graphical user interface is done by classes higher up in the hierarchy. All we do is extend JPanel and override its paintComponent method and use

\[
\text{super.paintComponent(g)}
\]

at the beginning of our method to let the superclass do its share of the work. In some of our graphics classes we also used the getWidth and getHeight methods to return the width and height of the drawing panel in pixels. If you look in the JPanel class documentation you won’t find these methods. However, you will find them higher up the hierarchy in the JComponent class so they are automatically inherited by our class. This is the power of inheritance. The GUI component hierarchy will be considered in more detail in Chapter 10.

9.4.2 Extending a circle calculator class

Consider the following version of the CircleCalculator class from Chapter 3, page 63.

```java
package chapter9.geometry;
/**
 * A simple class whose objects know how to calculate
 * the area of a circle given its radius.
 */
public class CircleCalculatorA
{
    protected double radius;
    private double area;
```
/**
 * Construct a circle.
 * @param r the radius of the circle
 */
public CircleCalculatorA(double r)
{
    radius = r;
    area = Math.PI * radius * radius;
}

/**
 * @return the radius of the circle
 */
public double getRadius()
{
    return radius;
}

/**
 * @return the area of the circle
 */
public double getArea()
{
    return area;
}
}

This class computes only the area of the circle. The radius data field declaration, which was private, is now protected so that subclasses can access it directly.

Suppose we also want to compute the circumference of the circle. Using inheritance we can create a subclass called CircleCalculatorB that extends CircleCalculatorA and also calculates the circumference. A constructor for this subclass will call the superclass constructor to compute the area, and then it will do its part and calculate the circumference. A new data field for the circumference and a corresponding inquiry method are needed.

### Class CircleCalculatorB

```java
package chapter9.geometry;
/**
 * A class whose objects know how to calculate
 * the area and circumference of a circle given its radius.
 */
public class CircleCalculatorB extends CircleCalculatorA
{
    private double circumference; // new instance data field

    /**
     * Construct a circle
     * @param r the radius of the circle
     */
```
9.4 Simple examples of subclasses

/*
 * @return the circumference of the circle
 */
public double getCircumference()
{
    return circumference;
}
}

The important statement in the constructor is

    super(r);

which calls the superclass constructor for the argument r. The effect is to construct the superclass part of a CircleCalculatorB object. This constructor call must be the first statement in the constructor. If the first statement in the constructor is not a super statement the default statement

    super();

is automatically inserted by the compiler. In our case the default statement is not appropriate since our superclass constructor has one argument, the radius of the circle.

In the superclass we could have left the radius data field private but then the subclass could not use it directly in the formula for the circumference: the statement

    circumference = 2.0 * Math.PI * radius;

would now be illegal since it tries to access a private data field of another class. However the subclass could still calculate the circumference but the statement

    circumference = 2.0 * Math.PI * getRadius();

would be required.

In the subclass we did not include the getRadius() and getArea() methods: all public and protected methods of a superclass are automatically available in all subclasses.

The following class can be used to test the two classes.

```java
package chapter9.geometry;
public class CircleCalculatorTester
{
    public void doTest()
    {
```
Inheritance and Interfaces

Figure 9.6: Inheritance in BlueJ is indicated by a solid arrow

```java
{  
  CircleCalculatorB circle = new CircleCalculatorB(3.0);  
  double radius = circle.getRadius();  
  double area = circle.getArea();  
  double circ = circle.getCircumference();  
  System.out.println("Radius: " + radius);  
  System.out.println("Area: " + area);  
  System.out.println("Circumference: " + circ);  
}

public static void main(String[] args)  
{  
  new CircleCalculatorTester().doTest();  
}
}

BlueJ project for the circle calculator classes

In BlueJ the inheritance relationship is indicated by a solid arrow from the subclass to the parent class as shown in Figure 9.6 for the classes CircleCalculatorA and CircleCalculatorB.

The object menu for a CircleCalculatorA object is shown in Figure 9.7(a) and the object menu for a CircleCalculatorB object is shown in Figure 9.6(b) as a two level menu. The first level shows the new method that has been added by CircleCalculatorB and the second level shows the two methods that have been inherited from CircleCalculatorA.

9.4.3 Extending the BankAccount class

We now want to make a subclass of the BankAccount class from Chapter 6 page 284 which is repeated here:
9.4 Simple examples of subclasses

Figure 9.7: (a) CircleCalculatorA object menu, (b) CircleCalculatorB object menu

Class BankAccount

```java
package chapter9.bank_account;

/**
 * A bank account object encapsulates the account number, owner name, and
 * current balance of a bank account.
 * This version checks for illegal method and constructor arguments.
 */
public class BankAccount
{
    private int number;
    private String name;
    private double balance;

    /**
     * Construct a bank account with given account number,
     * owner name and initial balance.
     * @param accountNumber the account number
     * @param ownerName the account owner name
     * @param initialBalance the initial account balance
     * @throws IllegalArgumentException if account number is negative,
     * owner name is null or empty, or if balance is negative.
     */
    public BankAccount(int accountNumber, String ownerName, double initialBalance)
    {
        if (accountNumber <= 0)
            throw new IllegalArgumentException("Account number must be positive");
        if (ownerName.equals("") || ownerName == null)
            throw new IllegalArgumentException("Owner name not defined");
        if (initialBalance < 0)
            throw new IllegalArgumentException("Balance must be non-negative");
        number = accountNumber;
        name = ownerName;
        balance = initialBalance;
    }

    /**
     * Deposit money in the account.
     * @param amount the deposit amount. If amount <= 0 the
     */
```
Inheritance and Interfaces

* account balance is unchanged.
* @throws IllegalArgumentException if deposit amount is negative
*
public void deposit(double amount)
{
    if (amount < 0)
        throw new IllegalArgumentException("Invalid amount for deposit");
    balance = balance + amount;
}

/**
 * Withdraw money from the account.
 * If account would be overdrawn the account balance is unchanged.
 * @param amount the amount to withdraw.
 * @throws IllegalArgumentException if withdraw amount is invalid
 */
public void withdraw(double amount)
{
    if (amount < 0 || amount > balance)
        throw new IllegalArgumentException("Invalid amount for withdraw");
    balance = balance - amount;
}

/**
 * Return the account number.
 * @return the account number.
 */
public int getNumber()
{
    return number;
}

/**
 * Return the owner name.
 * @return the owner name.
 */
public String getName()
{
    return name;
}

/**
 * Return the account balance.
 * @return the account balance.
 */
public double getBalance()
{
    return balance;
}

/**
 * string representation of this account.
9.4 Simple examples of subclasses

The new class will be called JointBankAccount. It will have a new data field for the name of the joint owner.

Most of the BankAccount methods can be used unchanged in the subclass. The only one we need to override is toString since the subclass version needs to display the new data field for the joint owner. We need one new method, getJointName, to return the name of the joint owner. The class will have the structure

```java
public class JointBankAccount extends BankAccount {
    // new data field for joint owner name goes here
    // constructors go here
    // new getJointName method goes here
    // overridden version of toString goes here.
}
```

For the new data field we can use the declaration

```java
private String jointName;
```

The three data fields in the BankAccount class are private so they cannot be directly modified by our subclass. However, they can be accessed using the inquiry methods.

Constructors are never inherited so we need to include a constructor that has four arguments, one for each of the four fields. It has the form

```java
public JointBankAccount(int accountNumber, String ownerName,
    String jointOwnerName, double initialBalance) {
    // initialize the four data fields here
}
```

We seem to run into a problem here. We would like to use the assignment statements

```java
this.number = accountNumber;
this.name = ownerName;
this.jointName = jointOwnerName;
this.balance = initialBalance;
```

in the body of the constructor but the three fields declared in the superclass are private, since they are managed entirely by the superclass. The way around this is to use super to call the superclass constructor and let it initialize the superclass part of a JointBankAccount object. This gives the constructor declaration
public JointBankAccount(int accountNumber, String ownerName, String jointOwnerName, double initialBalance) {
    super(accountNumber, ownerName, initialBalance); // superclass part
    this.jointName = jointOwnerName; // subclass part
}

The super statement is a call to the superclass constructor to initialize the superclass part (data fields) of a subclass object. Thus, there is no need for the subclass to directly access or change these data fields.

The overridden toString method can also use super.toString() to call the superclass version so that it can return its part of the string representation. Then the string expression

"JointBankAccount[" + super.toString() + ", " + jointName + "]";

can be used to provide a string representation of a subclass object. Here is the complete subclass declaration:

```java
class JointBankAccount {
    private String jointName;

    public JointBankAccount(int accountNumber, String ownerName, String jointOwnerName, double initialBalance) {
        super(accountNumber, ownerName, initialBalance);

        if (jointOwnerName.equals("") || jointOwnerName == null) {
            throw new IllegalArgumentException("Joint owner name not defined");
        }

        jointName = jointOwnerName;
    }
}
```
9.4 Simple examples of subclasses

/**
 * Return the joint owner name.
 * @return the joint owner name.
 */
public String getJointName()
{
    return jointName;
}

/**
 * string representation of this account.
 * @return string representation of this account.
 */
public String toString()
{
    return "JointBankAccount[" + super.toString() + ", " + jointName + "]";
}

BlueJ project for the bank account classes

To test the two bank account classes they can be placed in a BlueJ project called bankaccount as shown in Figure 9.8. The solid arrow indicates that the JointBankAccount class is a subclass of the BankAccount class.

In Figure 9.9 the object menu for each of the two classes is shown. The JointBankAccount object menu at the top level shows only the new getJointName method and the overridden toString method and the second-level menu shows the methods inherited from the BankAccount class and indicates that toString was overridden by the JointBankAccount class.
9.5 Polymorphism

We can now introduce polymorphism, one of the most important benefits of inheritance. There are two concepts: polymorphic types and polymorphic methods (the word polymorphism means “many forms”).

9.5.1 Polymorphic types

A polymorphic type is a hierarchy of classes defined by inheritance (later we will see that interfaces can also be used to define a polymorphic type). Each class is a subclass of the classes higher up in the hierarchy. Even though the various subclasses are different from each other we can think of them all as being of a similar type, namely the type of their top level superclass. Since Object is at the top of any hierarchy this means that every object is a type of Object.

This is sometimes called the “is a”, “is a kind of”, or the “is a type of” relationship. We will see that an important benefit of inheritance is that a superclass object reference can hold a reference to an object of any of its subclasses.

Polymorphic types in the BankAccount hierarchy

Consider the following statements:

```
BankAccount fred = new BankAccount(123, "Fred", 345.50);
JointBankAccount fredMary = new JointBankAccount(345, "Fred", "Mary", 450.65);
BankAccount ellenFrank = new JointBankAccount(456, "Ellen", "Frank", 3450.99);
```

The first two statements construct a BankAccount object and a JointBankAccount object and declare references to them. However, on the right side of the third statement a JointBankAccount object is created and its reference is then assigned to ellenFrank, which is a BankAccount reference (superclass reference). This permits us to treat the ellenFrank object as though it were a BankAccount object. The converse is not true and the following statement gives a compiler error:

```
JointBankAccount fred = new BankAccount(123, "Fred", 345.50);
```
Thus, we cannot assign a superclass reference to a subclass reference. This is understandable since we cannot say that a bank account object “is a” joint bank account object: it does not contain a joint owner name field.

To use the superclass `getName` method and the subclass `getJointName` method for `fredMary` it is only necessary to write statements such as

```java
String owner = fredMary.getName();
String jointOwner = fredMary.getJointName();
```

This follows since `fredMary` is declared as a `JointBankAccount` reference and the `getName` method is inherited from the superclass. However if we do the same with `ellenFrank`, namely

```java
String owner = ellenFrank.getName();
String jointOwner = ellenFrank.getJointName();
```

the first statement is fine but the compiler complains for the second statement that there is no `getJointName` method in the `BankAccount` class, which is true.

The `ellenFrank` object is being considered as a `BankAccount` object and has forgotten that it is really a `JointBankAccount` object. This is a form of object amnesia: when an object reference is assigned to a superclass reference the object forgets that it really belongs to the subclass. To overcome the object amnesia it is only necessary to write

```java
String jointOwner = ((JointBankAccount) ellenFrank).getJointName();
```

which uses a typecast. The extra parentheses are necessary here to indicate that the typecast should be applied to the `ellenFrank` object.

The following simple class illustrates the above ideas and can be used both inside and outside the BlueJ environment.

```java
package chapter9.bank_account;
import custom_classes.BankAccount;
import custom_classes.JointBankAccount;

/**<n
* A simple class to illustrate typecasting
* in the BankAccount hierarchy
*/
public class AccountTester
{
    public void doTest()
    {
        JointBankAccount fredMary = new JointBankAccount(123, "Fred", "Mary", 1000);
        BankAccount ellenFrank = new JointBankAccount(345, "Ellen", "Frank", 1000);

        String jointName1 = fredMary.getJointName();
        String jointName2 = ((JointBankAccount) ellenFrank).getJointName();
    }
}
```
Inheritance and Interfaces

```java
System.out.println("Joint name 1 is " + jointName1);
System.out.println("Joint name 2 is " + jointName2);
}

public static void main(String[] args)
{
    new AccountTester().doTest();
}
```

Here we construct two JointBankAccount objects but the second is assigned to a BankAccount reference so it forgets that it is really a JointBankAccount object.

Examples of polymorphism

**[EXAMPLE 9.4](Object class polymorphism)** The statements

```java
Object p = new Point(3,4);
Object c = new Circle((Point) p, 5);
```

show that a Point object "is a type of" Object and a Circle object "is a type of" Object. Both objects have forgotten their actual types so the statements

```java
System.out.println("x coordinate of p is " + p.getX());
System.out.println("Center of c is " + c.getCenter());
```

give compiler errors since the Object class does not have these methods. The statements

```java
System.out.println("x coordinate of p is " + ((Point) p).getX());
System.out.println("Center of c is " + ((Circle) c).getCenter());
```

that use a typecast are necessary to remind p that it is a Point and c that it is a Circle.

**[EXAMPLE 9.5](Graphics and Graphics2D)** In the paintComponent method (see Chapter 5) the statement

```java
Graphics2D g2D = (Graphics2D) g;
```

illustrates how a subclass reference can be extracted from a superclass reference. The Graphics class is an abstract class for basic graphics methods. Concrete subclasses are available for graphics output devices such as the screen or a printer. When Java 2D was introduced this class was extended to the abstract class Graphics2D which, as we have seen in Chapter 5, provides more graphics functionality. When the system calls the paintComponent method, whose argument is an object g from a subclass of Graphics, it actually provides a reference to a subclass instance of Graphics2D which implements all the abstract methods. Therefore the Graphics reference g can be typecast to a Graphics2D reference g2D which can be used in the paintComponent method to access the new Java 2D graphics methods. The original graphics methods can still be accessed using g instead of g2D.
Another form of polymorphism is the ability of an instance method in a class hierarchy to have many different forms, one for each subclass in the hierarchy. Such methods are called **polymorphic methods** and are made possible because we can override superclass methods in subclasses.

Do not confuse method overriding with method overloading, which has nothing to do with polymorphism. Method overloading simply refers to the concept that several methods in the same class can have the same name as long as they have distinguishable argument lists (signatures). In method overriding the methods have the same name and the same argument list but they are in different subclasses.

We will see that objects from the subclasses in an inheritance hierarchy that has polymorphic methods can be processed with these methods in a uniform manner, without regard to the particular subclass of the object, and the particular version of the method.

**Example 9.6** (Point2D superclass) In graphics programs we used statements such as

```java
Point2D.Double bottomRight = new Point2D.Double(300.0, 200.0);
```

Since `Point2D.Double` is a subclass of the `Point2D` class we can shorten this statement to

```java
Point2D bottomRight = new Point2D.Double(300.0, 200.0);
```

which just uses `Point2D` on the left side of the assignment. The same applies to the graphics classes such as `Line2D` and `Ellipse2D`.

**A polymorphic bank account transfer method**

Suppose we are processing bank transactions that transfer money from one account to another. We can write a transfer method to do this. Without inheritance we would need four separate methods with the prototypes

```java
public void transfer(BankAccount from, BankAccount to, double amount)
public void transfer(BankAccount from, JointBankAccount to, double amount)
public void transfer(JointBankAccount from, BankAccount to, double amount)
public void transfer(JointBankAccount from, JointBankAccount to, double amount)
```

that specify the source account (`from`), the destination account (`to`) and how much to transfer (`amount`), since there are four possibilities for the type of account. Since a `JointBankAccount` is also a `BankAccount` object we need only one method

```java
public void transfer(BankAccount from, BankAccount to, double amount)
{
    from.withdraw(amount);
    to.deposit(amount);
}
```

This is made possible by polymorphism. Because the `withdraw` and `deposit` methods are polymorphic within the bank account hierarchy, and because the `transfer` method arguments are declared to be of the base class type, we need only one form of the method.
For example, if `from` is a `JointBankAccount` object then the Java interpreter executes the statement

```java
from.withdraw(amount);
```

by first looking for a `withdraw` method in the `JointBankAccount` class. The method is not found so the interpreter goes up the hierarchy one level to the `BankAccount` class, finds the method there, and executes it.

### The polymorphic `toString` method

In the bank account hierarchy, each class has its own `toString` method: so this method has three forms, one in the `Object` class, one in the `BankAccount` class, and one in the `JointBankAccount` class. Therefore `toString` is a **polymorphic method**. The following simple class illustrates this idea.

```java
package chapter9.bank_account;
import custom_classes.BankAccount;
import custom_classes.JointBankAccount;

/**
 * A simple class to illustrate the polymorphic toString
 * method in the BankAccount hierarchy
 */
public class AccountTester2
{
    public void doTest()
    {
        BankAccount fred = new BankAccount(456, "Fred", 500);
        JointBankAccount fredMary = new JointBankAccount(123, "Fred", "Mary", 1000);
        BankAccount ellenFrank = new JointBankAccount(345, "Ellen", "Frank", 1000);

        System.out.println(fred);
        System.out.println(fredMary);
        System.out.println(ellenFrank);
    }

    public static void main(String[] args)
    {
        new AccountTester2().doTest();
    }
}
```

The output is

```
BankAccount[456, Fred, 500.0]
JointBankAccount[BankAccount[123, Fred, 1000.0], Mary]
JointBankAccount[BankAccount[345, Ellen, 1000.0], Frank]
```
9.6 Abstract classes and polymorphism

Even though the third account is assigned to the superclass BankAccount reference the run-time system knows that the account is really a JointBankAccount so the println method calls the toString method in this class.

Compile-time and run-time types

It is important to understand the difference between the compile-time type and the run-time type of an object. The compile-time type is the type given explicitly in the class. This may or may not also be the run-time type which is the actual type obtained using new and known to the run-time system (Java Virtual Machine).

For example in the above AccountTester2 class fred has BankAccount as both its compile-time and run-time type. Similarly, fredMary has JointBankAccount as its compile-time and run-time type. However, ellenFrank has compile-time type BankAccount but run-time type JointBankAccount. When println is executed at run-time it is always the run-time type that is used.

9.6 Abstract classes and polymorphism

We now introduce the concept of an abstract class and illustrate it with a simple employee inheritance hierarchy. An abstract class is a class that declares at least one method without providing a method body, i.e., no implementation is defined, only the method prototype. Each such method is called an abstract method. The class is specified using the abstract keyword.

When you specify an abstract method you are forcing each non-abstract subclass to provide an implementation (method body) for it having exactly the specified prototype. Thus, each abstract method is polymorphic.

9.6.1 An employee inheritance hierarchy

We now develop Java classes for the employee hierarchy given in Example 9.3 and Figure 9.3. The five classes can be described as follows:

Employee An abstract class that encapsulates the name of the employee. It has two abstract methods: grossSalary calculates and returns the gross monthly salary, and netSalary calculates and returns the net monthly salary (salary after deductions). It also has a getName method to return the name and a toString method. It will have a constructor with the prototype

```java
public Employee(String name)
```

Manager An employee with a gross monthly salary from which 10% is deducted to get the net monthly salary. It will implement the grossSalary and netSalary methods, provide a toString method, and have a constructor with the prototype

```java
public Manager(String name, double salary)
```
**HourlyWorker** An employee whose gross monthly salary is determined by the number of hours worked and the hourly wage. From this gross amount 5% is deducted to get the net monthly salary. It will implement the `grossSalary` and `netSalary` methods, provide a `toString` method, and have a constructor with the prototype

```java
public HourlyWorker(String name, double hoursWorked, double hourlyRate)
```

**PartTimeWorker** An employee like an hourly worker but with no deductions to get the net monthly salary. It will implement the `grossSalary` and `netSalary` methods, provide a `toString` method, and have a constructor with the prototype

```java
public PartTimeWorker(String name, double hoursWorked, double hourlyRate)
```

**CommissionWorker** An employee who receives a base monthly salary like a manager but a sales bonus is added to get the gross monthly salary. The bonus is a specified percentage of monthly sales. Thus, the gross monthly salary is

\[
\text{(base salary)} + \text{(monthly sales)} \times \left( \frac{\text{commission rate in percent}}{100.0} \right).
\]

From this 10% is deducted to get the net monthly salary. It will implement the `grossSalary` and `netSalary` methods, provide a `toString` method, and have a constructor with the prototype

```java
public CommissionWorker(String name, double baseSalary, double monthlySales, double commissionRate)
```

### 9.6.2 Employee and Manager classes

We give declarations for the first two classes here and the remaining three classes are left as an exercise (see Exercise 9.3).

#### Class Employee

```java
package chapter9.employee;

/**
 * An abstract class representing an employee.
 * The abstract grossSalary and netSalary methods are polymorphic.
 */
abstract public class Employee {
    private String name;

    /** Construct the name part of an employee.
     * @param name the name part of an employee
     */
    public Employee(String name) {
```
9.6 Abstract classes and polymorphism

```java
this.name = name;
}
/** Return the employee name.
 * @return the employee name
 */
public String getName()
{
    return name;
}
/** Return the gross salary of an employee.
 * @return gross salary of an employee
 */
abstract public double grossSalary();
/** Return the net salary of an employee.
 * @return net salary of an employee
 */
abstract public double netSalary();
}
Since this is an abstract class we cannot construct an Employee object so you may wonder why we have declared a constructor in the class. The reason is so that subclasses can use super to call the superclass Employee constructor to initialize the private data field for the employee name (see Manager class below).

Non-abstract subclasses of Employee must implement the two abstract methods. For Manager we have the class declaration

```java
package chapter9.employee;
/**
 * A class for employees that are managers.
 * The abstract grossSalary and netSalary methods are implemented
 * and the toString method is overridden.
 */
public class Manager extends Employee
{
    private double gross; // gross monthly salary
    private double net; // net monthly salary

    /** Construct a manager object with given name and salary
     * @param name the name of the manager
     * @param salary the gross salary of a manager
     */
    public Manager(String name, double salary)
    {
        super(name); // superclass is responsible for name
        gross = salary;
    
```
Inheritance and Interfaces

```java
net = 0.9 * gross;
}

/** Return the gross salary of a manager.
* @return the gross salary of a manager
*/
public double grossSalary()
{
    return gross;
}

/** Return the net salary of a manager.
* @return the net salary of a manager
*/
public double netSalary()
{
    return net;
}

/** Return the string representation of a manager.
* @return the string representation of a manager
*/
public String toString()
{
    return "Manager[" + "name = " + getName() + ", gross = " + grossSalary() + ", net = " + netSalary() + "]";
}
```

9.6.3 Polymorphism in the Employee hierarchy

There are three polymorphic methods in the Employee hierarchy: `grossSalary`, `netSalary`, and `toString`. Each of the four subclasses of Employee has its own version of these methods. We can illustrate polymorphism by writing a class that stores some objects in an array of Employee references and uses a loop to compute the total gross monthly salary of all employees, the total net monthly salary of all employees, and the total deductions. Here is a tester class containing a main method.

```java
package chapter9.employee;

public class EmployeeProcessor
{
    private Employee[] staff;
    private double totalGrossSalary;
    private double totalBenefits;
```
private double totalNetSalary;
/** Process an array of 5 employees and compute totals
* for gross salary, net salary, and benefits.
*/
public void doTest()
{
    staff = new Employee[5];
    staff[0] = new Manager("Fred", 800);
    staff[1] = new Manager("Ellen", 700);
    staff[2] = new HourlyWorker("John", 37, 13.50);
    staff[3] = new PartTimeWorker("Gord", 35, 12.75);
    staff[4] = new CommissionWorker("Mary", 400, 15000, 3.5);

    /* Compute the total gross salary, net salary and benefits for all
     * employees without knowing the kinds of employees when we write
     * the class.
     */
    totalGrossSalary = 0.0;
    totalNetSalary = 0.0;
    for (int i = 0; i < staff.length; i++)
    {
        totalGrossSalary = totalGrossSalary + staff[i].grossSalary();
        totalNetSalary = totalNetSalary + staff[i].netSalary();
        System.out.println(staff[i]);
    }

    totalBenefits = totalGrossSalary - totalNetSalary;
    System.out.println("Total gross salary: " + totalGrossSalary);
    System.out.println("Total benefits: " + totalBenefits);
    System.out.println("Total net salary: " + totalNetSalary);
}

public static void main(String[] args)
{
    new EmployeeProcessor().doTest();
}

The constructor first creates an array called staff of 5 references to base class Employee objects. They can refer to objects of any subclass so the next step is to construct five subclass objects and assign their references to the array elements. Finally, a simple loop, using the polymorphic grossSalary, netSalary and toString methods, displays the employee information and computes the total gross and net salaries of all employees. This is possible because these two methods were declared abstract in the Employee class so all the non-abstract subclasses are guaranteed to have implementations of them. Here is the output.

Manager[name = Fred, gross = 800.0, net = 720.0]
Manager[name = Ellen, gross = 700.0, net = 630.0]
HourlyWorker[name = John, gross = 499.5, net = 474.525]
PartTimeWorker[name = Gord, gross = 446.25, net = 446.25]
CommissionWorker[name = Mary, gross = 925.0, net = 832.5]
There are two important ideas here. The first is that it is not necessary to know anything about the kind of employee being processed with each loop iteration when the class is written. The system determines at run-time which kind of object is being used so the appropriate version of each polymorphic method is selected. The second is that if new kinds of employees are added to the hierarchy it is not necessary to make any modifications to the polymorphic loop that calculates the total gross and net salaries.

9.7 The Object class

The Object class is the ultimate parent of any Java class: an object of any Java class “is a type of” Object. This class declares several useful methods that are automatically inherited by any class. Here is a partial specification of the Object class

```java
public class Object {
  public Object() {...}
  public String toString() {...}
  public boolean equals(Object obj) {...}
  protected Object clone() {...}
  Class <? extends Object> getClass() {...}
  int hashCode() {...}
  // several other methods
}
```

We have already used the toString method and will discuss the others as needed.

9.7.1 Overriding Object class methods

Since the Object class is a superclass of all classes, any public or protected methods that it contains are automatically available to any class or can be overridden.

Overriding the toString method

For example, if we do not include a toString method in our classes we can still use it, since ultimately the Object class version will be called. This explains the output in Examples 4.15, 4.16 and 4.17 from Chapter 4. Without a toString method the doTest method in EmployeeProcessor would produce the output

```
Manager@310d42
Manager@5d87b2
HourlyWorker@77d134
PartTimeWorker@47e553
```
9.7 The Object class

CommissionWorker@20c10f
Total gross salary: 3370.75
Total benefits: 267.4749999999999
Total net salary: 3103.275

The strings produced are not very useful however; just the name of the class and a hexadecimal number. This is understandable since the Object class method does not know much about the Employee and Manager classes and other classes in the hierarchy. Therefore most classes override toString to provide a more meaningful string representation of an object.

Overriding the equals method

Similarly, the equals method provided in the Object class is not very useful since it just compares two references rather than the objects referenced. The Object class has no idea what kind of objects you are using and what your definition of equality is, so subclasses normally override it. For example, the String class in package java.lang has its own version of equals which we have used many times to compare two strings lexicographically.

To illustrate the equals method we can use the Point class from Chapter 4, page 119. Let us add an equals method to this class that compares two points using the definition that two Point objects are equal if both their x and y-coordinates are equal.

To test the equals method the new Point class has the structure

```java
public class Point {
    double x, y;
    public Point(double x, double y) { this.x = x; this.y = y; }
    public Point() { x = 0.0; y = 0.0; }
    public double getX() { return x; }
    public double getY() { return y; }
    public String toString() { return "Point[" + x + ", " + y + "]"; }

    public boolean equals(Object obj) { ... }
}
```

This class and the following PointEqualsTester class are in a package called chapter9.equals.

```java
package chapter9.equals;
/**
 * A tester class for equals method in Point class
 */
public class PointEqualsTester {
    public void doTest() {
```
Inheritance and Interfaces

```java
Point p = new Point(3,4);
Point q = new Point(3,4);
Point r = new Point(3,5);
if (p.equals(q))
    System.out.println("p and q are equal");
else
    System.out.println("p and q are not equal");

if (q.equals(r))
    System.out.println("q and r are equal");
else
    System.out.println("q and r are not equal");

public static void main(String[] args)
{
    new PointEqualsTester().test();
}
```

First compile the Point and PointTester classes without an equals method. This forces the Object version to be used. The output from the tester class is

```
p and q are not equal
q and r are not equal
```

which simply tells us that the three references p, q, and r are different. This is not very useful. We would like to have p.equals(q) return true to indicate that, even though the references p and q are different, the two objects are equal. Here are two versions of the equals method.

The first is

```java
public boolean equals(Point p)
{
    if (obj instanceof Point)
    {
        Point p = (Point) obj;
        return (x == p.x && y == p.y);
    }
    return super.equals(obj);
}
```

which uses the instanceof operator to check if obj has the correct type. Otherwise the type cast would throw a ClassCastException.

The second is

```java
public boolean equals(Object obj)
{
    if (obj == null) return false;
    if (! this.getClass().equals(obj.getClass())) return false;
    Point p = (Point) obj;
    return (x == p.x && y == p.y);
}
```
where we use the `getClass` method in the `Object` class to test if two objects have the same class. In our case the if statement compares the class of `this` object with the class of `obj`. If they are not the same then `false` is returned. Then the statement with the typecast will be executed only if `obj` really is a `Point` object. A value of `true` will be returned only if the `x` and `y` coordinates of the two points are the same.

With either version of `equals` the output of the tester class is

```
p and q are equal
q and r are not equal
```

which shows that the point objects are being compared, not their references.

The version using `instanceof` is the correct one if the class is declared to be final (see below). Otherwise the version using `getClass` should be used.

### 9.8 Final classes

A final class is one that cannot be extended so it can have no subclasses. The `final` keyword is used to indicate that a class is final. For example

```java
public final MyFinalClass
{
    // ...
}
```

is a final class so it would be a compiler error to try to write a class such as

```java
public final MySubClass extends MyFinalClass
{
    // ...
}
```

Final classes are usually more efficient. Also you gain more control over a class if its final since no one can override the methods in a final class. Many of the standard classes are final for efficiency and security. For example the `String` class is used everywhere and is declared to be final so it is impossible to override the `length` method and provide an incorrect value.

### 9.9 Interfaces

An `interface` is a kind of purely abstract class. It can contain only method prototypes and constants. No implementation of any of the methods can be provided. It is declared like a class using the keyword `interface` instead of the keyword `class`:

```java
public interface MyInterface
{
    // method prototypes go here, if any
}
```
Unlike abstract classes, interfaces cannot declare constructors since there are no objects to construct. Since all methods in an interface are abstract the `abstract` keyword is not needed. Also, every method in an interface must be public so the `public` keyword is also redundant but often included.

To make use of an interface we need to provide classes that “implement the interface”. Such classes must provide complete declarations (implementations) for all the methods declared by the interface. If `MyClass` is a class that implements an interface called `MyInterface` then the syntax of this class declaration is

```java
public class MyClass implements MyInterface {
    // data fields
    // constructors
    // methods not related to interface, if any
    // Implementations of the interface methods
}
```

Here the keyword `implements` is used instead of `extends`.

The important idea here is that we can say that an object of `MyClass` “is of type” `MyInterface`. This is possible since we can declare interface references and assign to them references to any object of any class that implements the interface. For example suppose `MyClass1` and `MyClass2` are classes that implement `MyInterface`. Then with the declarations

```java
MyInterface myObject1 = new MyClass1(...);
MyInterface myObject2 = new MyClass2(...);
```

`myObject1` “is a type of” `MyInterface` and so is `myObject2`. This means that polymorphism also applies to interfaces in the sense that the classes that implement an interface form a polymorphic type and the interface methods are polymorphic.

However, interfaces can be more flexible and general than class inheritance hierarchies since the classes that implement an interface do not need to be related in any other way. In particular, they do not have to belong to any class inheritance hierarchy.

It is also possible to have interface inheritance hierarchies. For example, we can extend `MyInterface` to obtain a subinterface called `MySubinterface` using

```java
public interface MySubinterface extends MyInterface {
    // optionally the MyInterface prototypes can be included here
    // new method prototypes are included here
}
```

If a class wants to implement `MySubinterface` it must implement all the methods in `MyInterface` as well as the new ones in `MySubinterface`. The method prototypes in `MyInterface` can also be repeated in `MySubinterface`.

Multiple inheritance is allowed for interfaces in the sense that a class can implement several interfaces but extend only one class. Therefore the general structure of a class declaration that extends another class and implements several interfaces is
public class MyClass extends MySuperclass
    implements MyInterface1, MyInterface2, ..., MyInterfaceN
{
    // MyClass data fields
    // MyClass constructors
    // MyClass methods not related to the interfaces, if any
    // Implementations of all interface methods
}

which indicates that MyClass extends MySuperclass and implements \( N \) interfaces. Two interfaces can have a method with the same name and argument types. Of course any class that implements both interfaces can only provide one implementation of the common method. If this implementation does not make sense for both interfaces then it is not possible to implement both interfaces with one class.

EXAMPLE 9.7 (Measurable interface) The interface

    public interface Measurable
    {
        public double area();
        public double perimeter();
    }

declares two abstract methods called area and perimeter that are supposed to represent the area and perimeter of a two-dimensional geometric object. Notice that the method prototypes are terminated by a semi-colon to indicate that there is no implementation. This example illustrates that an interface is a design specification. It specifies what it means for an object to be “measurable”: it means that the area and perimeter of the object can be calculated.

EXAMPLE 9.8 (Scalable interface) The interface

    public interface Scalable
    {
        public void scale(double s);
    }

declares one abstract method called scale that is supposed to scale a two-dimensional geometric object by the factor \( s \) in both directions. An object of any class that implements this interface is said to be a Scalable object.

EXAMPLE 9.9 (Extending the Scalable interface) The interface

    public interface Scalable2D extends Scalable
    {
        public void scale(double sx, double sy);
    }
extends the Scalable interface by providing the prototype for a more general scale method that can scale using different factors in each direction. This sets up an interface inheritance hierarchy. A class that implements the Scalable2D interface must implement both versions of the scale method. Objects of this class are Scalable2D objects and through inheritance they are also Scalable objects.

### 9.9.1 Implementing the Measurable interface

Let us illustrate interface polymorphism by writing geometric Circle and Rectangle classes that implement the Measurable interface in Example 9.7. A common error is to omit the ‘implements’ clause on the class declaration. The class will still compile but its objects will not be measurable.

#### A measurable circle class

The following simple Circle class (see Chapter 4, page 122 for a related class that uses a Point object for the circle center) implements the Measurable interface by providing implementations of the area and perimeter interfaces.

```java
package chapter9.interfaces;

/**
 * A class for measurable circles
 */
public class Circle implements Measurable {
    private double x, y; // coordinates of center
    private double radius;

    public Circle() {
        this(0,0,1);
    }

    public Circle(double xc, double yc, double r) {
        x = xc;
        y = yc;
        radius = r;
    }

    public double getX() {
        return x;
    }

    public double getY()
```
If c is a Circle object we can say that c “is of type” Measurable, or c is a Measurable object. In this case we could define a circle c1 using

Circle c1 = new Circle(0.0, 0.0, 1.0);

or we could define a circle c2 using

Measurable c2 = new Circle(0.0, 0.0, 1.0);

We use this statement in situations where we are only interested in the interface methods: with c1 we have access to all the methods of the Circle class, including those in the interface, but with c2 we have access, without a typecast, to only the area and perimeter methods. For example, the last of the statements

double a1 = c1.area();
double r1 = c1.getRadius();
double a2 = c2.area();
double r2 = c2.getRadius();

gives an error since c2, as a Measurable object, has forgotten that it is also a Circle object (object amnesia again). To fix this we need to typecast:

double r2 = ((Circle)c2).getRadius();

to remind c2 that it is also a Circle object.
A measurable rectangle class

The following simple Rectangle class implements the Measurable interface since it provides implementations of the area and perimeter interfaces.

```java
class Rectangle
{
    private double x, y;  // coordinates of lower left corner
    private double width, height;  // width and height of rectangle

    public Rectangle()
    {
        this(0,0,1,1);
    }

    public Rectangle(double x, double y, double w, double h)
    {
        this.x = x;
        this.y = y;
        width = w;
        height = h;
    }

    public double getX()
    {
        return x;
    }

    public double getY()
    {
        return y;
    }

    public double getWidth()
    {
        return width;
    }

    public double getHeight()
    {
        return height;
    }

    public String toString()
    {
        return this;
    }
}
```
Polymorphism with the `Measurable` interface

The following class shows how to use a polymorphic loop to compute the total area and perimeter of some measurable objects.

```java
class MeasurableTester {
    private Measurable[] a = new Measurable[3];

    public void test() {
        a[0] = new Circle(0,0,1);
        a[1] = new Circle(1,1,2);
        a[2] = new Rectangle(5,5,20,10);

        double areaSum = 0.0;
        double perimeterSum = 0.0;
        for (int k = 0; k < a.length; k++) {
            areaSum = areaSum + a[k].area();
            perimeterSum = perimeterSum + a[k].perimeter();
            System.out.println(a[k]);
            System.out.println("Perimeter = " + a[k].perimeter() + ", Area = " + a[k].area());
        }
        System.out.println("Total area is " + areaSum);
    }
}
```
Inheritance and Interfaces

```java
System.out.println("Total perimeter is " + perimeterSum);
}

public static void main(String[] args)
{
    new MeasurableTester().test();
}
```

The program declares an array of type Measurable and then assigns references to Circle and Rectangle objects to the array elements. The area and perimeter of these objects can then be calculated in a polymorphic loop since any Measurable object has perimeter and area methods. It is important that a is an array of Measurable type. The program output is

- Circle[x = 0.0, y = 0.0, radius = 1.0] Perimeter = 6.283185307179586, Area = 3.141592653589793
- Circle[x = 1.0, y = 1.0, radius = 2.0] Perimeter = 12.566370614359172, Area = 12.566370614359172
- Rectangle[x = 5.0, y = 5.0, width = 20.0, height = 10.0] Perimeter = 60.0, Area = 200.0
- Total area is 215.70796326794897
- Total perimeter is 78.84955592153875

### 9.9.2 Polymorphism with the Shape interface

The graphics hierarchy shown in Chapter 5, Figure 5.6, is a hierarchy in which classes such as Line2D and Rectangle2D implement the Shape interface. Therefore we can say that a Line2D object “is of type” Shape. The Shape interface simply specifies the methods that a class must implement in order to be called a Shape. Also, the RectangularShape class is an example of an abstract class that implements the Shape interface. Its purpose is to provide a base class for the classes that are specified using bounding rectangular boxes in their description. These are the Rectangle2D, RoundRectangle2D, Ellipse2D, and Arc2D classes. The polymorphic draw and fill methods in the Graphics2D class have prototypes

```java
public void draw(Shape s);
public void fill(Shape s);
```

so they can take as an argument an object of any class, such as Rectangle2D, that implements the Shape interface. Without the Shape polymorphism we would need separate draw and fill commands for each kind of shape: e.g., drawLine, drawRect, drawEllipse, and so on.

We can illustrate polymorphism in the set of classes implementing the Shape interface. Shape is an interface in package java.awt that declares 10 rather complicated methods that are needed to draw and fill shapes. Classes such as Line2D.Double and Rectangle2D.Double implement this interface so each object from one of these classes is a type of Shape. Previously we defined graphics objects using declarations such as

```java
Line2D.Double line = new Line2D.Double(0.0, 0.0, 200.0, 150.0);
Rectangle2D.Double rect = new Rectangle2D.Double(10.0, 10.0, 100.0, 150.0);
```
Since `line` and `rect` are each a type of `Shape`, instead of these declarations we could have used

```java
Shape line = new Line2D.Double(0.0, 0.0, 200.0, 150.0);
Shape rect = new Rectangle2D.Double(10.0, 10.0, 100.0, 150.0);
```

Doing it this way permits a polymorphic processing of graphical objects using the `draw` and `fill` methods.

If we have an array of type `Shape` and assign various objects to it from classes that implement the `Shape` interface then we can write a single polymorphic loop to process all the objects using `draw` and `fill`. For example, let us declare an array of `Shape` references:

```java
Shape[] shape = new Shape[5];
```

This is legal: even though there is no such thing as a `Shape` object we can declare an array of references to objects from classes that implement the `Shape` interface. Therefore, we can use statements such as

```java
shape[0] = new Line2D.Double(0.0, 0.0, 200.0, 150.0);
```

since a `Line2D.Double` object is a type of `Shape`. We can now draw all the shapes in one polymorphic loop such as

```java
for (int k = 0; k < shape.length; k++)
{
    g2D.draw(shape[k]);
}
```

without knowing the particular kinds of shape. Without polymorphism we would have to use a giant if statement inside the loop (if the object is of type T1 draw it this way, else if it is of type T2 draw it that way, else, ...). Here is a simple graphics program to illustrate this uniform processing of geometrical objects.

```
<table>
<thead>
<tr>
<th>Class ShapeTester</th>
</tr>
</thead>
</table>
```

```java
package chapter9.shapetest;
import custom_classes.GraphicsFrame;
/**
 * Illustrating polymorphism within the Shape hierarchy.
 */
import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

public class ShapeTester extends JPanel{
    Shape[] shape = new Shape[5]; // array to hold Shape objects

    public ShapeTester()
```
The output is shown in Figure 9.10.

9.10 Multiple interfaces

It is possible for a class to implement more than one interface. As an example we consider three interfaces that define methods useful in working with two-dimensional geometrical objects.
9.10 Multiple interfaces

9.10.1 Interface specifications

The first is the Measurable interface. It declares prototypes for methods that calculate the area and perimeter of a geometrical object. The interface declaration is

```java
package chapter9.multiple_interfaces;

/**
 * An interface for geometric objects that have
 * an area and perimeter
 */
public interface Measurable
{
    public double area();
    public double perimeter();
}
```

Similarly, we declare a Translatable interface which declares the prototype for a method that translates an object by a given distance in the $x$ and $y$ directions. The interface declaration is

```java
package chapter9.multiple_interfaces;

/**
 * An interface for geometric objects that can
 * be translated in a given direction
 */
public interface Translatable
{
    public void translate(double dx, double dy);
}
```

Finally we define a Scalable interface which declares the prototype for a method that scales an object by a given amount (zoom factor) in both directions. The interface declaration is

```java
package chapter9.multiple_interfaces;

/**
 * An interface for geometric objects that can
 * be scaled in a given direction
 */
public interface Scalable
{
    public void scale(double factor);
}
### Interface Scalable

```java
package chapter9.multiple_interfaces;
/**
 * An interface for geometric objects that can
 * be scaled (made larger or smaller)
 */
public interface Scalable {
    public void scale(double s);
}
```

### 9.10.2 Classes that implement the interfaces

We can now design classes of geometrical objects that implement one or more of these interfaces. As an example we modify our Circle and Rectangle classes from Section 9.9.1 so that they implement all three interfaces as follows

```java
package chapter9.multiple_interfaces;
/**
 * A class that implements multiple interfaces
 */
public class Circle implements Measurable, Translatable, Scalable {
    private double x, y;  // coordinates of center
    private double radius;

    public Circle(double xc, double yc, double r) {
        x = xc;
        y = yc;
        radius = r;
    }

    // implementations of getX, getY, getRadius, toString go here

    // Implement the three interfaces
    public double area() {
        return Math.PI * radius * radius;
    }

    public double perimeter() {
        return 2.0 * Math.PI * radius;
    }
}
```
public void translate(double dx, double dy)
{
    x = x + dx;
    y = y + dy;
}

public void scale(double s)
{
    radius = radius * s;
}

Class Rectangle

package chapter9.multiple_interfaces;
/**
 * A class that implements multiple interfaces
 */
public class Rectangle implements Measurable, Translatable, Scalable
{
    private double x, y;  // coordinates of lower left corner
    private double width, height;  // width and height of rectangle

    public Rectangle(double x, double y, double w, double h)
    {
        this.x = x;
        this.y = y;
        width = w;
        height = h;
    }

    // implementations of getX, getY, getWidth, getHeight, toString go here

    // Implement the interfaces
    public double area()
    {
        return width * height;
    }

    public double perimeter()
    {
        return 2.0 * (width + height);
    }

    public void translate(double dx, double dy)
    {
        x = x + dx;
        y = y + dy;
    }
9.10.3 Typecasts with multiple interfaces

Suppose we want to do multiple polymorphism: process objects using methods of more than one interface. We can use the Measurable, Translatable, and Scalable interfaces as an example. The program MeasurableTester processed Circle and Rectangle objects using only the Measurable interface, so we declared an array

```java
private Measurable[] a = new Measurable[3];
```

and used a single loop to calculate the area and perimeter of the objects. Similarly, if we just wanted to scale the objects we could declare an array

```java
private Scalable[] a = new Scalable[3];
```

Suppose that we want to both scale and translate objects in a single loop. We cannot declare a Scalable array since it would be an error to apply translate to a Scalable object and conversely we cannot declare a Translatable array since it would be an error to apply scale to a Translatable object.

Another solution is to use an array of type Object:

```java
private Object[] a = new Object[3];
```

and assign Circle and Rectangle object references to it. Then we can write one loop to process these objects. The only problem is that the Circle and Rectangle objects suffer amnesia when they are assigned to the Object array. Therefore, when we want to apply translate to an object we must remind the object, using a typecast, that it is Translatable. Similarly, when we want to apply scale we must use a typecast to Scalable. Here is a short program that illustrates this multiple polymorphism:

```java
package chapter9.multiple_interfaces;
/**
 * Illustrate multiple polymorphism of interfaces
 */
public class MultipleInterfaceTester {
    private Object[] a = new Object[3];
}
```
public void test()
{
    a[0] = new Circle(0,0,1);
    a[1] = new Circle(1,1,2);
    a[2] = new Rectangle(5,5,20,10);

    for (int k = 0; k < a.length; k++)
    {
        ((Translatable) a[k]).translate(1,1);
        ((Scalable) a[k]).scale(2);
        System.out.println(a[k]);
    }
}

public static void main(String[] args)
{
    new MultipleInterfaceTester().test();
}

In the first statement in the loop a[k] is typecast using

    (Translatable) a[k]

to make a Translatable object. Now we can apply the translate method to this object using

    ((Translatable) a[k]).translate(1,1);

The extra set of parentheses are necessary since “dot” has a higher precedence than the typecast. Similarly, a[k] is typecast to Scalable so that the scale method can be applied. The output is

Circle[x = 1.0, y = 1.0, radius = 2.0]
Circle[x = 2.0, y = 2.0, radius = 4.0]
Rectangle[x = 6.0, y = 6.0, width = 40.0, height = 20.0]

9.11 Implementing the Shape interface

In Chapter 5 we used several classes such as Rectangle2D and Ellipse2D that we could draw and fill using the polymorphic draw and fill methods that accept Shape arguments. We now show how to create our own classes that implement the Shape interface.

9.11.1 Shape interface methods

If you look at the Java documentation you will see that the Shape interface declares the following 10 methods:

    public boolean contains(Point2D p);
    public boolean contains(Rectangle2D r);
public boolean contains(double x, double y);
public boolean contains(double x, double y, double w, double h);
public Rectangle getBounds();
public Rectangle2D getBounds2D();
public PathIterator getPathIterator(AffineTransform at);
public PathIterator getPathIterator(AffineTransform at, double flatness);
public boolean intersects(Rectangle2D r);
public boolean intersects(double x, double y, double w, double h);

You probably have no idea what many of these methods mean, let alone how to implement them. The good news is that you don’t have to know. Since the GeneralPath class implements them we can design our own custom graphics classes that can be used as arguments to draw and fill. There are two ways to do this: using an adapter class that implements Shape or implementing the Shape interface directly.

### 9.11.2 Extending a ShapeAdapter class which implements Shape

Consider the following adapter class.

```java
package chapter9.shapes;
import java.awt.*;
import java.awt.geom.*;

/**
 * Extend this class to implement the Shape interface for defining geometrical objects.
 */
public class ShapeAdapter implements Shape
{
    /** The path used to define the Shape */
    protected GeneralPath path;

    /** Construct an empty path */
    public ShapeAdapter()
    {
        path = new GeneralPath();
    }

    /*
     * implementation of the Shape interface using the implementation provided by GeneralPath.
     */
    public boolean contains(Point2D p)
    { return path.contains(p); }
    public boolean contains(Rectangle2D r)
```
This is an example of an adapter class since it adapts a GeneralPath object by hiding its complexity and provides a simple representation of a specific geometric object such as a triangle or a polygon. Each method is implemented by simply returning what path’s version of each method produces.

Now if we have a graphics class such as MyGraphicsShape for drawing some geometric object we can write this class in the form

```java
public class MyGraphicsShape extends ShapeAdapter {
    // data fields, if any
    // constructors using the inherited path object to define the path
    // methods, if any
}
```

Since ShapeAdapter implements the Shape interface our MyGraphicsShape class will automatically implement it too.

Now we can use statements such a

```java
Shape s = new MyGraphicsShape(...);
...
g2D.draw(s);
```

to construct and draw an object of our class.

### 9.11.3 Triangle2D class that uses ShapeAdapter

As an example consider the following Triangle2D class that extends ShapeAdapter and therefore implements the Shape interface.
package chapter9.shapes;
import java.awt.geom.*;

/**
 * A Triangle2D object is represented as 3 Point objects for
 * the vertices and a path that can be used to draw or fill it.
 * This version of the class also implements the Shape interface
 * by extending the ShapeAdapter class
 */
public class Triangle2D extends ShapeAdapter{
    private Point2D.Double v1, v2, v3; // the triangle vertices

    /** Construct default triangle with vertices (0,0), (1,0), (0.5,1) */
    public Triangle2D()
    {
        this(new Point2D.Double(0,0), new Point2D.Double(1,0),
             new Point2D.Double(0.5,1));
    }

    /** Construct triangle with specified vertices.
     * @param x1 x coordinate of vertex 1
     * @param y1 y coordinate of vertex 1
     * @param x2 x coordinate of vertex 2
     * @param y2 y coordinate of vertex 2
     * @param x3 x coordinate of vertex 3
     * @param y3 y coordinate of vertex 3
     */
    public Triangle2D(double x1, double y1, double x2, double y2, double x3,
                      double y3)
    {
        this(new Point2D.Double(x1,y1), new Point2D.Double(x2,y2),
             new Point2D.Double(x3,y3));
    }

    /** Construct triangle with specified points as vertices.
     * @param p1 First vertex
     * @param p2 Second vertex
     * @param p3 Third vertex
     */
    public Triangle2D(Point2D.Double p1, Point2D.Double p2, Point2D.Double p3)
    {
        v1 = (Point2D.Double) p1.clone();
        v2 = (Point2D.Double) p2.clone();
        v3 = (Point2D.Double) p3.clone();
        path.moveTo((float) v1.x, (float) v1.y); // path inherited from ShapeAdapter
9.11 Implementing the Shape interface

```java
    path.lineTo((float) v2.x, (float) v2.y);
    path.lineTo((float) v3.x, (float) v3.y);
    path.closePath();
}
}
```

Here we have used the clone method in the Point2D.Double class to make copies of the points specified by the formal arguments so that each Triangle2D object will have its own copies of these points as instance data fields.

We can use this class in the following RandomTriangles class that draws some random triangles.

```
package chapter9.shapes;
import custom_classes.GraphicsFrame;
import java.awt.*;
import javax.swing.*;

/**
 * Use the Triangle2D class to draw some random triangles.
 * In this version a Triangle2D object is a Shape.
 */
public class RandomTriangles extends JPanel {

    private int numTriangles;

    /**
     * Construct object.
     * @param n the number of triangles to draw
     */
    public RandomTriangles(int n) {
        setNumTriangles(n);
        setBackground(Color.white); // set panel background color
    }

    public void setNumTriangles(int n) {
        numTriangles = n;
    }

    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        Graphics2D g2D = (Graphics2D) g;
        g2D.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                              RenderingHints.VALUE_ANTIALIAS_ON);

        double w = getWidth(); // panel width in pixels
```
Inheritance and Interfaces

double h = getHeight(); // panel height in pixels

for (int k = 1; k <= numTriangles; k++)
{
    // Generate random triangle that fits inside the panel
    Triangle2D t = new Triangle2D(
        w*Math.random(), h*Math.random(),
        w*Math.random(), h*Math.random(),
        w*Math.random(), h*Math.random() );

    // RGB values are in the range 0 to 255
    Color c = new Color( (int) (256*Math.random()),
        (int) (256*Math.random()), (int) (256*Math.random()) );

    // Fill triangle with random color and outline it in black
    g2D.setPaint(c);
    g2D.fill(t); // t is now a Shape
    g2D.setPaint(Color.black);
    g2D.draw(t);
}

/** draw some random triangles. */
public void draw()
{
    new GraphicsFrame("Random Triangles",
        new RandomTriangles(numTriangles), 301, 201);
}

public static void main(String[] args)
{
    int n;
    if (args.length == 1)
        n = Integer.parseInt(args[0]);
    else
        n = 10;
    RandomTriangles tri = new RandomTriangles(n);
    tri.draw();
}

Note that we do not use Triangle2D.Double (we are not providing both Float and Double versions).

9.11.4 Implementing Shape directly

Since a class can only extend one other class we run into a problem if MyGraphicsShape is already extending some class. In this case we can implement the Shape interface directly. For example,
9.12 Turtle graphics class

As a final example of implementing the Shape interface using ShapeAdapter we consider a turtle model for producing line drawings known as “turtle graphics” which originated as part of the Logo computer language.

We think of a turtle moving on a page. The turtle has a pen underneath which can be either up or down. If the pen is down then a line is drawn on the page as the turtle moves. This drawing model is similar to the one used by a pen plotter.

9.12.1 Specification of the class

The turtle can execute the following operations and motions that will correspond to instance methods in a Turtle2D class:

- There is a home operation that puts the turtle in a default “home” position which we choose to be the origin (0,0) with the turtle pointing north. When a turtle is constructed it is placed in the home position. We will make (0,0) correspond to the center of the drawing panel.

- It can move forward along a line a specified distance in the direction it is pointing. This direction is called the turtle heading.

- It can back up (move backward) by a specified distance along the line in the opposite direction from which it is pointing without changing its heading.

- It can rotate left in place through a specified angle to change its heading. This corresponds to a counterclockwise rotation if the angle is positive.

- It can rotate right in place through a specified angle to change its heading. This corresponds to a clockwise rotation if the angle is positive.

- It can retract the pen so that nothing is drawn by the forward and backward commands.

- It can put its pen down so that drawing takes place.

The Turtle2D class will use a GeneralPath object path, provided by ShapeAdapter, to construct the turtle’s path. The class specification is

```java
public class MyGraphicsClass extends AnotherClass implements Shape {
    GeneralPath path;
    // other data fields
    // constructors and methods not in Shape interface
    // implementation of the 10 shape methods go here using path
}
```
public class Turtle2D extends ShapeAdapter
{
    // data fields go here

    // Construct turtle given an approximate bounding rectangle.
    // The home position is at the center of this rectangle.
    public Turtle2D(double xTopLeft, double yTopLeft,
                    double width, double height) {...}

    // move the turtle home
    public void home() {...}

    // rotate turtle by given angle counterclockwise
    public void left(double angle) {...}

    // rotate turtle by given angle clockwise
    public void right(double angle) {...}

    // move turtle forward a given distance
    public void forward(double distance) {...}

    // move turtle backward a given distance
    public void backward(double distance) {...}

    // move the pen up
    public void penUp() {...}

    // move the pen down
    public void penDown() {...}
}

### 9.12.2 Implementation of the class

As data fields we need the turtle’s position \((x, y)\), the turtle’s heading angle, and a boolean variable `penUp` that is true if the pen is in the up position and false otherwise. We also need variables to describe the bounding rectangle of the path produced by the turtle.

```java
private double x, y;
private double heading; // in degrees
private boolean penUp;
private double xTopLeft, yTopLeft, width, height;
```

The `path` data field is inherited from `ShapeAdapter`.

To implement the `forward` method we need to use some trigonometry to express the coordinates of a point \((x_1, y_1)\) in terms of polar coordinates as

\[
\begin{align*}
x_1 & = r \cos \theta \\
y_1 & = r \sin \theta
\end{align*}
\]
where \( r \) is the distance from \((x_1, y_1)\) to the origin \((0,0)\) and \(\theta\) is the angle in radians measured from the \(x\)-axis as shown in Figure 9.11. Since the turtle is at position \((x, y)\), not \((0,0)\) we can translate the coordinates to obtain

\[
\begin{align*}
x_1 &= x + r \cos \theta \\
y_1 &= y + r \sin \theta
\end{align*}
\]

as the new coordinates of the turtle corresponding to \textit{forward}(r). Since \((x_1, y_1)\) becomes the new \((x, y)\) we can write

\[
\begin{align*}
x &= x + r \cos \theta \\
y &= y + r \sin \theta
\end{align*}
\]

Since we are dealing with a user space in Java 2D that has origin at the top left corner with the \(y\)-axis pointing downward, we need to change to a minus sign on the right side of the equation for \(y\). This gives the method implementation

```java
public void forward(double distance)
{
    double radians = Math.toRadians(heading);
    x = x + distance * Math.cos(radians);
    y = y - distance * Math.sin(radians);
    if (penUp)
        path.moveTo((float)x, (float)y);
    else
        path.lineTo((float)x, (float)y);
}
```

We can call this method to obtain \textit{backward}:

```java
public void backward(double distance)
{
    forward(-distance);
}
```
We can now implement `left` and `right`. Our first attempt at `left` would be

```java
public void left(double angle)
{
    heading = heading + angle;
}
```

However if `left` is called many times the angle can become quite large and the `sin` and `cos` functions will produce more round-off error. Therefore it is useful to always adjust the heading so that it is between 0 degrees and 360 degrees each time `left` is called. This gives the implementation

```java
public void left(double angle)
{
    heading = heading + angle
    while (heading > 360.0) heading = heading - 360.0;
    while (heading < 0.0) heading = heading + 360.0;
}
```

and similarly for `right`, where the angle is subtracted from the heading.

Here is the complete class declaration that also includes two methods called `setHeading` and `setDirection` for specifying an absolute rather than relative heading and direction.

---

**Class Turtle2D**

```java
package chapter9.shapes;

/**
 * A class that defines Turtle objects for drawing on a panel.
 * Each turtle object produces a Shape object which can be drawn
 * using the draw method of a Graphics2D graphics context.
 * <p>
 * The approximate bounding rectangle for the turtle’s path is
 * specified in the constructor but the final rectangle will generally
 * be different and can be obtained using the getBounds2D
 * method of the Shape interface.
 */
public class Turtle2D extends ShapeAdapter
{
    private double x, y; // coordinates of turtle
    private double heading; // turtle direction in degrees
    private boolean penUp; // turtle has a pen to draw with

    // Definition of the bounding rectangle
    private double xTopLeft, yTopLeft, width, height;

    /**
     * Construct a turtle that will draw a path.
     * The turtle state is defined by its current position and heading
     * in degrees, and by its pen state (up or down).
     */
```
### 9.12 Turtle graphics class

* The turtle begins at \((0,0)\) facing north with its pen down.

* **@param xTopLeft** top left x-coord of bounding rectangle
* **@param yTopLeft** top left y-coord of bounding rectangle
* **@param width** width of the bounding rectangle
* **@param height** height of the bounding rectangle

```java
public Turtle2D(double xTopLeft, double yTopLeft,
    double width, double height)
{
    penUp = false;
    this.xTopLeft = xTopLeft;
    this.yTopLeft = yTopLeft;
    this.width = width;
    this.height = height;
    home();
}
```

*/

/**
 * Move turtle to center, facing north
 */
public void home()
{
    x = xTopLeft + width / 2.0; // x coord of center
    y = yTopLeft + height / 2.0; // y coord of center
    heading = 90.0; // north default
    path.moveTo((float)x, (float)y); // set the position
}

/**
 * Set the turtle at a global position without changing its heading.
 * **@param x** x-coord of the position
 * **@param y** y-coord of the position
 */
public void setPosition(double x, double y)
{
    this.x = x;
    this.y = y;
    path.moveTo((float)x, (float)y);
}

/**
 * Set the turtle to a specified heading without changing its position.
 * **@param heading** angle in degrees \((0 = \text{east}, \; 90 = \text{north})\)
 */
public void setHeading(double heading)
{
    this.heading = heading;
}

/**
 * Rotate the turtle counterclockwise.
 */
* @param angle rotation angle in degrees
*/
public void left(double angle)
{
    heading = heading + angle;
    while (heading > 360.0) heading = heading - 360.0;
    while (heading < 0.0) heading = heading + 360.0;
}

/**
* Rotate the turtle clockwise.
* @param angle rotation angle in degrees
*/
public void right(double angle)
{
    heading = heading - angle;
    while (heading > 360.0) heading = heading - 360.0;
    while (heading < 0.0) heading = heading + 360.0;
}

/**
* Advance turtle a given distance along its heading.
* @param distance the distance to advance. A negative
* value would move the turtle backward.
*/
public void forward(double distance)
{
    double radians = Math.toRadians(heading);
    x = x + distance * Math.cos(radians);

    // negative sign in y since we have a user space with
    // origin at top left corner of panel and a y-axis
    // pointing downward.
    y = y - distance * Math.sin(radians);

    if (penUp)
        path.moveTo((float)x, (float)y);
    else
        path.lineTo((float)x, (float)y);
}

/**
* Advance turtle a given distance backward from its heading.
* @param distance the distance to move backward. A negative
* value would move the turtle forward.
*/
public void backward(double distance)
{
    forward(-distance);
}
9.12 Turtle graphics class

** Set the turtle’s pen in the up position.
** In this state forward and backward move turtle without drawing.
*/
public void penUp()
{
    penUp = true;
}

/**
 * Set the turtle’s pen in the down position.
 * In this state forward and backward move turtle with drawing.
 */
public void penDown()
{
    penUp = false;
}

9.12.3 Writing turtle graphics programs

We can use the GraphicsFrame class to write turtle graphics programs. Here is a simple template:

```java
import custom_classes.GraphicsFrame;
import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

public class MyTurtleGraph extends JPanel
{
    public void paintComponent(Graphics g)
    {
        super.paintComponent(g);
        Graphics2D g2D = (Graphics2D) g;

        // Construct a turtle for the entire drawing panel
        Turtle2D t = new Turtle2D(0,0, getWidth(), getHeight());

        // insert turtle graphics commands here to construct turtle path
        g2D.draw(t); // draw the turtle’s path
    }

    public static void main(String[] args)
    {
        new GraphicsFrame("MyTurtleGraph", new MyTurtleGraph(), 301, 301);
    }
```
EXAMPLE 9.10 (Drawing a box) Since the turtle starts at (0,0) pointing north, the statements

```java
for (int k = 1; k <= 2; k++)
{
    t.forward(10);
    t.right(90);
    t.forward(20);
    t.right(90);
}
```

define a rectangle with lower left corner at (0,0) and upper right corner at (20,10).

EXAMPLE 9.11 (Drawing a pentagon) The statements

```java
for (int k = 1; k <= 5; k++)
{
    t.forward(80);
    t.right(72);
}
```

show how easy it is to draw a pentagon with a vertex at (0,0).

Here is a class called PentagonSpinner that draws pentagons rotated about the lower left corner. Each time the pentagon is drawn the turtle rotates 36 degrees before drawing the next one.

Class PentagonSpinner

```java
package chapter9.shapes;
import custom_classes.GraphicsFrame;
import java.awt.*;
import javax.swing.*;
/**
 * Use A Turtle2D object to draw a pentagon and spin it.
 */
public class PentagonSpinner extends JPanel
{
    public void paintComponent(Graphics g)
    {
        super.paintComponent(g);
        Graphics2D g2D = (Graphics2D) g;
        g2D.setPaint(Color.blue);
        g2D.setStroke(new BasicStroke(2.0f));
        Turtle2D t = new Turtle2D(0,0, getWidth(), getHeight());
        for (int k = 1; k <= 10; k++)
```
9.12 Turtle graphics class

```java
{  
    drawPentagon(t);  
    t.left(36);  
}

g2D.draw(t);
}

public void drawPentagon(Turtle2D t)  
{  
    for (int k = 1; k <= 5; k++)  
    {  
        t.forward(80);  
        t.right(72);  
    }
}

public void draw()  
{  
    new GraphicsFrame("Spinning Pentagons", this, 301, 301);
}

public static void main(String[] args)  
{  
    new PentagonSpinner().draw();  
}
}
```

The output is shown in Figure 9.12. The turtle graphics model makes it easy to draw pictures like this.

![Image](image.png)

Figure 9.12: Output for PentagonSpinner program.
Recursive turtle graphics programs

Many interesting pictures can be drawn using recursive turtle graphics methods. Here is a method that draws the branches of a tree.

```java
public void tree(double length)
{
    if (length < 1) return;

    t.right(45);
    t.forward(length);
    tree(length / 1.7);
    t.backward(length);
    t.left(90);
    t.forward(length);
    tree(length / 1.7);
    t.backward(length);
    t.right(45);
}
```

The turtle is pointing north by default so it first rotates right by 45 degrees and draws the right branches, then it backs up and rotates left by 90 degrees and draws the left branches. Finally, it rotates right by 45 degrees so it is pointing north again. The recursion is controlled by the length of the branches. At each recursive step the branch length decreases and the recursion is stopped if the length becomes less than 1. Here is a class that uses this method.

```java
package chapter9.shapes;
import custom_classes.GraphicsFrame;
import java.awt.*;
import javax.swing.*;

/**
 * Use a Turtle2D object to draw a recursive tree.
 */
public class RecursiveTreeMaker extends JPanel
{
    Turtle2D t; // reference to the turtle

    public void paintComponent(Graphics g)
    {
        super.paintComponent(g);
        Graphics2D g2D = (Graphics2D) g;
        g2D.setColor(Color.blue);
        g2D.setStroke(new BasicStroke(2.0f));
        t = new Turtle2D(0,0, getWidth(), getHeight());
    }
}
```
9.13 Numerical applications of interfaces

```java
public void tree(double length)
{
    if (length < 1) return;

    t.right(45);  // position turtle for right side
    t.forward(length);  // draw right branches
    tree(length / 1.7);
    t.backward(length);

    t.left(90);  // position turtle for left side
    t.forward(length);  // draw left branches
    tree(length / 1.7);
    t.backward(length);

    t.right(45);  // position turtle for right side
}
```

```java
public void draw()
{
    new GraphicsFrame("Recursive Tree", this, 225, 225);
}
```

```java
public static void main(String[] args)
{
    new RecursiveTreeMaker().draw();
}
```

The output window is shown in Figure 9.13.

9.13 Numerical applications of interfaces

In numerical analysis it is quite common to work with many kinds of functions and it is necessary to write algorithms that have functions as their input. In Java this means that we need a way to use a function as a method argument. This can be done in an elegant fashion using interfaces. We illustrate this with two examples: (1) a class containing methods that display a table of values of a function and (2) a class containing methods to do function iteration.
9.13.1 Displaying a table of values of a function

Suppose we have a function \( f \) with values \( f(x) \) and we want to display a table of its values at the points \( x_{\text{start}}, x_{\text{start}} + 1, \ldots, x_{\text{end}} \). Assume that the \( x \) values are equally spaced so they have the form \( x_{\text{start}}, x_{\text{start}} + h, x_{\text{start}} + 2h, \ldots, x_{\text{end}} \), where \( h \) is the distance between successive \( x \) values. Given \( x_{\text{start}} \) and \( x_{\text{end}} \) the \( x \) values for the table can be expressed as \( x_k = x_{\text{start}} + kh, k = 0, \ldots, n \) and \( x_{\text{end}} = x_{\text{start}} + nh \), where \( n \) is the number of steps and \( n + 1 \) is the number of table values. A simple method for computing a table of values is

\[
\text{public void table(double xStart, double xEnd, double h)}
\{
    \text{int numSteps = (int) Math.round((xEnd - xStart) / h);}
    \text{for (int k = 0; k <= numSteps; k++)}
    \{
        \text{double x = xStart + k*h;}
        \text{System.out.println(x + " " + f(x));}
    \}
\}
\]

We also need a method to define the function \( f \). For example, if \( f(x) = e^{-x} \) then we can write the method

\[
\text{public double f(double x)}
\{
    \text{return Math.exp(-x);}
\}
\]

to return values of this function.
What happens if we want to produce a table for another function? We run into a problem since the function name \( f \) is built-in to our table method. We would have to change the body of the function \( f \) and re-compile the class containing it. Thus our table method is not reusable because the function is not supplied as an argument.

The solution is to write a function as an object from a class that implements a function interface. For example, we can use the following interface to represent a double-valued function of a double variable.

```java
package chapter9.functions;

public interface DoubleFunction {
    /**
     * Return value of a function \( f(x) \) at a given \( x \).
     * @param x value at which to evaluate the function
     * @return the value \( f(x) \) of the function
     */
    public double value(double x);
}
```

This interface represents all functions that have one `double` argument and return one `double` value. Any class that implements this interface must provide an implementation of the `value` method that gives the value of the function. An object of this class `is a` `DoubleFunction`. Now we can write the table method in a class called `TableMaker` using a `DoubleFunction` argument as

```java
package chapter9.functions;

public class TableMaker {
    private String fmt; // format code

    /**
     * Construct a table maker with formatting
     * @param fmt the format code for printing
     */
    public TableMaker(String fmt) {
        this.fmt = fmt;
    }

    /**
     * Display a table of the function \( f \).
     * @param f the function
     * @param xStart starting value of \( x \)
     */
    public void table(DoubleFunction f, double xStart) {
        // code to display the table
    }
}
```
Inheritance and Interfaces

```java
public void table(DoubleFunction f, double xStart, double xEnd, double h)
{
    int numSteps = (int) Math.round((xEnd - xStart) / h);
    for (int k = 0; k <= numSteps; k++)
    {
        double x = xStart + k*h;
        System.out.printf(fmt, x, f.value(x));
    }
}
```

Here we have included a simple formatting code for each line of the table. For example

```
TableMaker maker = new TableMaker("%5.2f %10.5f\n");
```

The important idea here is that the first argument of the `table` method can be an object of any class that implements the `DoubleFunction` interface so this `table` method is reusable. We simply use `f.value(x)` in the method body to obtain the value of the function at `x`.

For example, the functions \( f(x) = e^{-x} \) and \( g(x) = \cos x \) can be represented as objects of the following two simple classes that implement the `DoubleFunction` interface.

### Class ExpMinusFunction

```java
package chapter9.functions;

/**
 * Class for the function exp(-x)
 */
public class ExpMinusFunction implements DoubleFunction
{
    public double value(double x)
    {
        return Math.exp(-x);
    }
}
```

### Class CosFunction

```java
package chapter9.functions;

/**
 * Class for the function cos(x)
 */
public class CosFunction implements DoubleFunction
{
    public double value(double x)
    {
```
The following tester class shows how to produce tables for these two functions.

```
package chapter9.functions;
public class TableMakerTester {
  public void doTest() {
    TableMaker maker = new TableMaker("%5.2f %10.5f\n");
    DoubleFunction exp = new ExpMinusFunction();
    DoubleFunction cos = new CosFunction();
    System.out.println("Table of exp(-x)\n");
    maker.table(exp, 0.0, 0.5, 0.1);
    System.out.println("Table of cos(x)\n");
    maker.table(cos, 0.0, 0.5, 0.1);
  }
  public static void main(String[] args) {
    TableMakerTester tester = new TableMakerTester();
    tester.doTest();
  }
}

The output is

```

<table>
<thead>
<tr>
<th>Table of exp(-x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00  1.00000</td>
</tr>
<tr>
<td>0.10  0.90484</td>
</tr>
<tr>
<td>0.20  0.81873</td>
</tr>
<tr>
<td>0.30  0.74082</td>
</tr>
<tr>
<td>0.40  0.67032</td>
</tr>
<tr>
<td>0.50  0.60653</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table of cos(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00  1.00000</td>
</tr>
<tr>
<td>0.10  0.99500</td>
</tr>
<tr>
<td>0.20  0.98007</td>
</tr>
<tr>
<td>0.30  0.95534</td>
</tr>
<tr>
<td>0.40  0.92106</td>
</tr>
<tr>
<td>0.50  0.87758</td>
</tr>
</tbody>
</table>

**9.13.2 Function iteration**

The iterates of a function $f(x)$ form a sequence $x_0, x_1, ..., x_n, ...$ defined by $x_k = f(x_{k-1})$. Here we assume that $x_0$ is given. We also require that all the values $x_k$ are defined (they are in the domain of definition of the function $f$). We are interested in the long-term behaviour of the sequence for
a given function $f$ and initial value $x_0$. There are three possibilities: (1) the sequence converges, (2) the sequence diverges, (3) the sequence neither converges nor diverges. The third case can be quite interesting.

For example, if you choose the function $f(x) = 0.5(x + 2/x)$ and iterate it starting with $x_0 = 1$ then the sequence converges to $\sqrt{2}$:

$$x_0 = 1, \; x_1 = 0.5(x_0 + 2/x_0) = 1.5, \; x_2 = 0.5(x_1 + 2/x_1) = 1.4166...,\ldots$$

If $f(x) = 2x$ then the sequence clearly diverges.

An interesting case illustrating the third possibility is $f(x) = 3.83(1 - x)$. Here the sequence of iterates neither converges nor diverges. Instead it eventually repeats in a periodic manner in groups of 3. For example the sequence eventually looks like

$$0.9574165975188731, \; 0.15614931568360532, \; 0.5046664874084134, \; 0.9574165975188731, \; 0.15614931568360532, \; 0.5046664874084134, \; 0.9574165975188731, \; 0.15614931568360532, \; 0.5046664874084134,\ldots$$

Here is a class containing two methods iterate and iterate2 that can be used to iterate a function. The iterate method computes and displays the sequence $x_0, x_1, \ldots, x_{n-1}$ and iterate2 lets you skip iterations $x_0, x_1, \ldots, x_{\text{skip}-1}$ before displaying the sequence $x_{\text{skip}}, x_{\text{skip}+1}, \ldots, x_{\text{skip}+n-1}$. Each method has a DoubleFunction argument to specify the function to iterate.

```java
package chapter9.functions;

/**
 * This class shows how to iterate a function $f(x)$
 * using the DoubleFunction interface.
 */

class FunctionIterator
{
    /**
     * Compute iterates of a function.
     * @param f the function to iterate
     * @param x0 the initial value of x
     * @param n number of iterations to display
     * The values $x(0),\ldots,x(n-1)$ are displayed
     */
    public void iterate(DoubleFunction f, double x0, int n)
    {
        double x = x0;
        System.out.println(x);
        for (int k = 1; k < n; k++)
        {
            x = f.value(x);
            System.out.println(x);
        }
    }
}
```
/**
 * Compute iterates of a function, skipping the first few.
 * @param f the function to iterate
 * @param x0 the initial value of x
 * @param skip the number of iteration to skip before display
 * @param n the number of iterations to display
 * The values x(0),...,x(skip-1) are skipped, then the values
 * x(skip), ..., x(skip+n-1) are displayed.
 */
public void iterate2(DoubleFunction f, double x0, int skip, int n)
{
    double x = x0;
    for (int k = 0; k < skip; k++) x = f.value(x);
    iterate(f, x, n);
}

Here is a tester class that shows how to iterate the functions 0.5(x + 2/x) and 3.83x(1 − x) using inner classes to define the functions.

```
package chapter9.functions;

public class FunctionIteratorTester {
    public void doTest() {
        FunctionIterator f = new FunctionIterator();
        DoubleFunction sqrt = new Sqrt();

        System.out.println("5 Iterates of sqrt");
        f.iterate(sqrt, 1.0, 5);

        // Test iterate2 with a period 3 function
        // skip 10000 iterates and display 10
        DoubleFunction period3 = new Period3();
        System.out.println("Skip 10000 iterates of 3.83*x*(1-x)"ırım);
        f.iterate2(period3, 0.1, 10000, 6);
    }

    private static class Sqrt implements DoubleFunction {
        public double value(double x) {
            return 0.5*(x + 2/x);
        }
    }
}
```
private static class Period3 implements DoubleFunction {
    public double value(double x) {
        return 3.83*x*(1-x);
    }
}

public static void main(String[] args) {
    new FunctionIteratorTester().doTest();
}

The output is

5 Iterates of sqrt
1.0
1.5
1.4166666666666665
1.4142156862745097
1.4142135623746899
Skip 10000 iterates of 3.83*x*(1-x)
0.9574165975188731
0.15614931568360532
0.5046664874084134
0.9574165975188731
0.15614931568360532
0.5046664874084134

Sometimes functions contain parameters. For example, we can find the square root of the number $a \geq 0$ by iterating the function $f_a(x) = 0.5(x + a/x)$ and we can generalize the period 3 example to $f_a(x) = ax(1 - x)$. The following example shows how to do the parametrized square root function.

```java
package chapter9.functions;
public class SquareRootIterator {
    public void doTest() {
        FunctionIterator f = new FunctionIterator();
        DoubleFunction sqrt = new Sqrt(3.0);
        f.iterate(sqrt, 1.0, 7);
    }

    private static class Sqrt implements DoubleFunction {
        private double a;
        public Sqrt(double a) {
            this.a = a;
        }
        public double value(double x) {
            return a*x*(1-x);
        }
    }
}
```

Class SquareRootIterator
```java
{  
    this.a = a;  
}

public double value(double x)
{
    return 0.5*(x + a/x);
}

public static void main(String[] args)
{
    new SquareRootIterator().doTest();
}
```

The output for $\sqrt{3}$ is

1.0  
2.0  
1.75  
1.7321428571428572  
1.7320508100147274  
1.7320508075688772  
1.7320508075688772

The trick is to include a constructor in the Sqrt class that sets the value of the private data field a defining the parameter. This parameter is set using

```
DoubleFunction sqrt = new Sqrt(3.0);
```

## 9.14 Review exercises

- **Review Exercise 9.1** Define the following terms and give examples of each.
  
<table>
<thead>
<tr>
<th>inheritance</th>
<th>polymorphism</th>
<th>polymorphic type</th>
</tr>
</thead>
<tbody>
<tr>
<td>polymorphic method</td>
<td>adapter class</td>
<td>abstract class</td>
</tr>
<tr>
<td>abstract method</td>
<td>interface</td>
<td>multiple inheritance</td>
</tr>
<tr>
<td>subclass</td>
<td>superclass</td>
<td>Object class</td>
</tr>
<tr>
<td>extends</td>
<td>this(...)</td>
<td>super</td>
</tr>
<tr>
<td>protected</td>
<td>toString</td>
<td>Shape interface</td>
</tr>
<tr>
<td>equals</td>
<td>base class</td>
<td>final class</td>
</tr>
</tbody>
</table>

- **Review Exercise 9.2** Write a tester class called Amnesia similar to CircleCalculatorTester on page 459 but with the following test method.
  
  ```java
  public void test()
  {
      CircleCalculatorA circle = new CircleCalculatorB(3.0);
      double circumference = circle.getCircumference();
      System.out.println("Circumference: "+circumference);
  }
  ```
Explain what happens? Replace the line defining the circumference by

```java
double circumference = ((CircleCalculatorB)circle).getCircumference();
```

and explain why this works.

### 9.15 Programming exercises

**Exercise 9.1 (Inheritance using TriangleCalculator)**

In Chapter 3 we considered a class called `CircleCalculator` and in Chapter 9 we wrote the class `CircleCalculatorA` which calculates only the area and then we extended this class to `CircleCalculatorB` which calculated the circumference.

Do something similar with the `TriangleCalculator` class from Chapter 3, page 64 as follows:

- Write a class called `TriangleCalculatorA` that computes only the third side length \( c \) and the three angles \( \alpha, \beta, \gamma \). Also include a `toString` method.
- Write a class called `TriangleCalculatorB` that extends `TriangleCalculatorA` by additionally calculating the perimeter and the area.
- Write a tester class called `TriangleCalculatorTester` that tests the two classes.

**Exercise 9.2 (Polymorphic bank account transfer method)**

Write a tester class for the polymorphic bank account transfer method (see page 469).

**Exercise 9.3 (Completing the Employee class hierarchy)**

Complete the remaining three classes in the `Employee` hierarchy and test the class hierarchy using `EmployeeProcessor`, page 474.

**Exercise 9.4 (Completing the Employee class hierarchy)**

Remove the `toString` methods from the four classes in the `Employee` hierarchy and put the following `toString` method into the `Employee` class.

```java
public String toString()
{
    return getClass().getName() + "\n[name = " + getName() + ", gross = " + grossSalary() + ", net = " + netSalary() + "]\n";
}
```

Compile the class and use `EmployeeProcessor`, page 474 to test it. Can you explain why this `toString` method works.

**Exercise 9.5 (Another way to design the Employee hierarchy)**

Sometimes there is more than one way to design a hierarchy. For example, in the `Employee` hierarchy we could say that a part time worker is an hourly worker (one without deductions). This gives the hierarchy shown in Figure 9.14. Modify the classes of the preceding exercise to use this hierarchy.
9.15 Programming exercises

![Employee inheritance hierarchy diagram]

Figure 9.14: Another possible Employee inheritance hierarchy

**Exercise 9.6 (A student employee hierarchy)**
Write a class called `Person` and three subclasses called `Student` (a subclass of `Person`), `Employee` (a subclass of `Person`) and `StudentEmployee` (a subclass of `Student`) as follows

(a) The `Person` class encapsulates three private data fields: a name of type `String`, a social insurance number of type `String` and a year of birth of type `int`. The class should include a constructor, get methods for the private data fields and a `toString` method.

(b) The `Student` class is a subclass of `Person`. A student is a person with a major (MATH, COSC, etc.) and a student number. Provide a constructor, the appropriate get methods and a `toString` method.

(c) The `Employee` class is a subclass of `Person`. An employee is a person with a monthly salary. Provide a constructor, the appropriate get methods and a `toString` method.

(d) The `StudentEmployee` class is a subclass of `Student`. A student employee is a student with a salary. Provide a constructor, the appropriate get method and a `toString` method.

**Exercise 9.7 (Equals method for bank account classes)**
Write the `equals` method for the `BankAccount` and `JointBankAccount` classes assuming that two accounts are equal if they have the same account number.

**Exercise 9.8 (Using the Translatable interface)**
Write a class called `TranslatableTester`, similar to `MeasurableTester`, that declares an array of type `Translatable`. Write the for-loop so that it translates each object by (1,1) and displays results.

**Exercise 9.9 (Using the Scalable interface)**
Write a class called `ScalableTester`, similar to `MeasurableTester`, that declares an array of type `Scalable`. Write the for-loop so that it scales each object by a factor of 2 and displays results.

**Exercise 9.10 (An Employable interface)**
Do the Employee hierarchy in Section 9.6 using an Employable interface instead of an inheritance hierarchy. This interface is defined by
Inheritance and Interfaces

```java
public interface Employable
{
    /**
     * Return the gross monthly salary.
     * @return the gross monthly salary
     */
    public double grossSalary();
    /**
     * Return the net monthly salary.
     * @return the net monthly salary
     */
    public double netSalary();
}
```

Now there is no abstract Employee class. Instead, each of the classes Manager, HourlyWorker, PartTimeWorker, and CommissionWorker will need to implement the Employable interface. Each class must now have a name field and a getName method since we no longer have the Employee class to manage the name for us.

Modify the tester class EmployeeProcessor to use an array of Employable references instead of an array of Employee references. We now have an array of references to objects from classes that implement the Employable interface instead of an array of references to objects that extend the abstract Employee class.

Discuss any advantages or disadvantages of these two approaches to polymorphism.

▶ Exercise 9.11 (An equals method for the Circle class)
Write an equals method for the Circle class from Chapter 4.4.5, page 122

▶ Exercise 9.12 (Colored circles)
Consider the following Point and Circle and Color classes:

```java
public class Point
{
    private double x, y;
    public Point() {...}
    public Point(double x, double y) {...}
    public double getX() {...}
    public double getY() {...}
    public String toString() {...}
}

public class Circle
{
    private Point center;
    private double radius;
    public Circle() {...}
    public Circle(double x, double y, double radius) {...}
    public Circle(Point center, double radius) {...}
    public Point getCenter() {...}
    public double getRadius() {...}
}
```
9.15 Programming exercises

```java
public String toString() {...}
}

public class Color
{
    private int red, green, blue;
    public Color() {...}
    public Color(int red, int green, int blue) {...}
    public int getRed() {...}
    public int getGreen() {...}
    public int getBlue() {...}
    public String toString() {...}
}
```

Write a class called ColoredCircle that extends Circle by adding a new data field for the circle color and has constructors

```java
public ColoredCircle() {...}
published ColoredCircle(double x, double y, double radius, Color color) {...}
publish ColoredCircle(Circle circle, Color color) {...}
publish ColoredCircle(Point center, double radius, Color color {...}
```

Assuming that Color and Circle are immutable classes (objects cannot be modified after construction) write the ColoredCircle class so that it is also immutable.

**Exercise 9.13 (A polygon Shape class)**
Write a Polygon2D class that extends the ShapeAdapter class (it implements the Shape interface). Rewrite the RandomPolygon program class (page 497) so that it uses the Polygon2D class.

**Exercise 9.14 (A pentagon Shape class)**
Write a Pentagon2D class that extends the ShapeAdapter class and defines regular pentagons (5 sided polygons with all sides equal and central angle between consecutive vertices of 72 degrees). Each pentagon should be defined in user space by the coordinates of its center and the radius of the circumscribed circle, rather than the coordinates of its five vertices. Include a constructor of the form

```java
public Pentagon2D(double xCenter, double yCenter, double radius,
    double angle)
```

where angle is the angle in degrees of one of the vertices. See Chapter 8 Exercise 8.8 for the definitions of the pentagonal angles and vertices. Write a test program using GraphicsFrame that draws some pentagons.

**Exercise 9.15 (A happy face Shape class)**
Write a HappyFace2D class based on the happy face programs developed in Chapter 5 that extends the ShapeAdapter class. Use the bounding box to define the size of the face. Include some flexibility in the constructors that lets the user define the face using either the bounding rectangle or the center and radius. Also let the user choose the colors. Write a test program using GraphicsFrame that draws some happy faces.
Exercise 9.16 (A house Shape class)
Write a House2D class that extends the ShapeAdapter class. A house has a frame, door, windows and chimney. Include some flexibility in the constructors that lets the user define the house using the bounding rectangle and choose the fill colors, for example. Write a test program using GraphicsFrame that draws some houses.

Exercise 9.17 (Polymorphism in the Shape hierarchy)
Write a class that uses the Shape objects Pentagon2D, HappyFace2D, House2D, and Polygon2D from the previous exercises to illustrate polymorphism. To do this define an array of Shape objects, store some of these objects in the array, and use the draw method in a loop to draw a picture.

Exercise 9.18 (A general polygon spinner program)
The PentagonSpinner program only spins pentagons and only 10 times using an angle of 36 degrees (10 times 36 is a full circle). Write a PolygonSpinner program based on this class that spins a regular polygon. The program should read the number of sides and the spinning angle as command line arguments.

Exercise 9.19 (Modifying RecursiveTreeMaker)
The RecursiveTreeMaker class uses fixed values of 45 and 90 degrees for the angles and 1.7 for the branch length reduction factor. Rewrite the class so that it reads these three values as command line arguments.

Exercise 9.20 (Function iteration)
Using a class similar to SquareRootIterator write a class called LogisticIterator that can be used to iterate the function \( f_a(x) = ax(1 - x) \). Experiment with the long-term behaviour of the iterations for various values of \( s \) in the range \( 0 \leq a \leq 4 \). We have already considered the case \( a = 3.83 \).

Exercise 9.21 (Newton’s method for root finding)
If \( f(x) \) is differentiable with derivative denoted by \( f'(x) \) then a root of \( f(x) = 0 \) can often be calculated using the iteration scheme
\[
x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}, \quad n = 0, 1, \ldots
\]
starting with a good initial guess \( x_0 \).

Write a class called RootFinder that contains a method called newton that calculates and displays the iterations. This method will need two DoubleFunction arguments, one for \( f(x) \) and another for the derivative \( f'(x) \). Other arguments can be used to specify the initial guess and the maximum number of iterations. Also specify a tolerance \( \varepsilon \) such that the iteration process is stopped when either the maximum number of iterations is reached or \( |x_{n+1} - x_n| \leq \varepsilon \).

Write a tester class that illustrates several examples.