CHAPTER 10

Object-Oriented Programming, Part 3: Inheritance, Polymorphism, and Interfaces

CHAPTER CONTENTS

Introduction

One of the most common ways to reuse a class is through inheritance. Inheritance helps us to organize related classes into **hierarchies**, or ordered levels of functionality. To set up a hierarchy, we begin by defining a class that contains methods and fields (instance variables and class variables) that are common to all classes in the hierarchy. Then we define new classes at the next lower level of the hierarchy, which inherit the behavior and fields of the original class. In the new classes, we define additional fields and more specific methods. The original class is called the **superclass**, and the new classes that inherit from the superclass are called **subclasses**. Some OOP developers call a superclass the **base class** and call a subclass the **derived class**.

As in life, a superclass (parent) can have multiple subclasses (children), and each subclass can be a superclass (parent) of other subclasses (children) and so on. Thus, a class can be both a subclass (child) and a superclass (parent). In contrast to life, however, Java subclasses inherit directly from only one superclass.

A subclass can add fields and methods, some of which may **override**, or hide, a field or method inherited from a superclass.

Let's look at an example. To represent a hierarchy of vehicle types, we define a *Vehicle* class as a superclass. We then define an *Automobile* class that inherits from *Vehicle*. We also define a *Truck* class, which also inherits from *Vehicle*. We further refine our classes by defining a *Pickup* class and a *TractorTrailer* class, both of which inherit from the *Truck* class. Figure 10.1 depicts our hierarchy using a UML (Unified Modeling Language) diagram. Arrows pointing from a subclass to a superclass indicate that the subclass refers to the superclass for some of its methods and fields. The boxes below the class name are available for specifying instance variables and methods for each class. For simplicity, we will leave those boxes blank. Later in the chapter, we will illustrate UML diagrams complete with fields and methods.

The Java class library contains many class hierarchies. At the root of all Java class hierarchies is the *Object* class, the superclass for all classes. Thus, all classes inherit from the *Object* class.

The most important advantage to inheritance is that in a hierarchy of classes, we write the common code only once. After the common code has

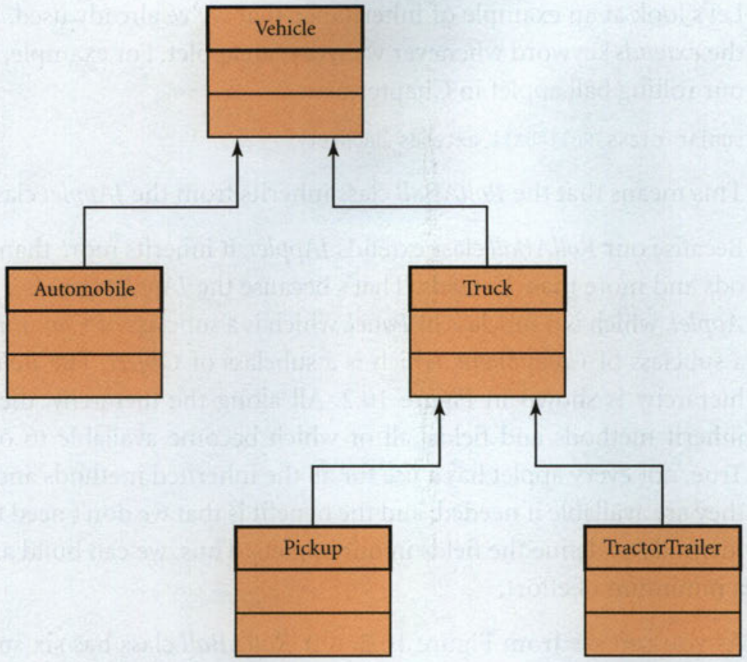


Figure 10.1
***Vehicle* Class Hierarchy**

been tested, we can reuse it with confidence by inheriting it into the subclasses. And when that common code needs revision, we need to revise the code in only one place.

10.1 Inheritance

The syntax for defining a subclass class that inherits from another class is to add an *extends* clause in the class header:

```
accessModifier class SubclassName extends SuperclassName
{
   // class definition
}
```

The *extends* keyword specifies that the subclass inherits members of the superclass. That means that the subclass begins with a set of predefined methods and fields inherited from its hierarchy of superclasses.

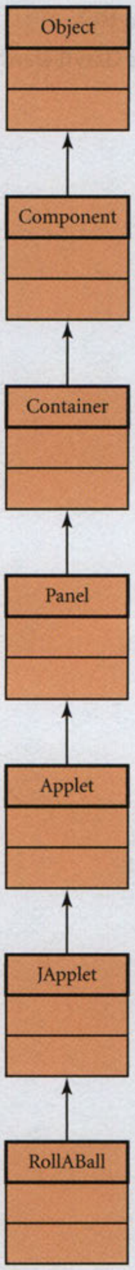


Figure 10.2
The *RollABall* Class Hierarchy

Let's look at an example of inheritance that we've already used. We've used the *extends* keyword whenever we wrote an applet. For example, we defined our rolling ball applet in Chapter 6 as:

```
public class RollABall extends JApplet
```

This means that the *RollABall* class inherits from the *JApplet* class.

Because our *RollABall* class extends *JApplet*, it inherits more than 275 methods and more than 15 fields. That's because the *JApplet* class is a subclass of *Applet*, which is a subclass of *Panel*, which is a subclass of *Container*, which is a subclass of *Component*, which is a subclass of *Object*. The *RollABall* class hierarchy is shown in Figure 10.2. All along the hierarchy, the subclasses inherit methods and fields, all of which become available to our applets. True, not every applet has a use for all the inherited methods and fields, but they are available if needed, and the benefit is that we don't need to write the methods or define the fields in our applets. Thus, we can build applets with a minimum of effort.

As you can see from Figure 10.2, our *RollABall* class has six superclasses. The class that a subclass refers to in the *extends* clause of the class definition is called its **direct superclass**. Thus, *JApplet* is the direct superclass of *RollABall*. Similarly, the class that *extends* the superclass is called the **direct subclass** of the superclass, so *RollABall* is a direct subclass of the *JApplet* class. A class can have multiple direct subclasses, but only one direct superclass.

10.2 Inheritance Design

We say that an "is a" relationship exists between a subclass and a superclass; that is, a subclass object "is a" superclass object. For example, we could define a student class hierarchy with a *Student* superclass and derive a *GraduateStudent* subclass. A graduate student "is a" student, but actually a special type of student. We could also define an employee class hierarchy with an *Employee* superclass and derive *Faculty* and *Staff* subclasses, because faculty and staff are both special types of employees.

To design classes for inheritance, our superclass should define fields and methods that will be common to all classes in the hierarchy. Each subclass will provide specialization by adding methods and fields. Where appropri-

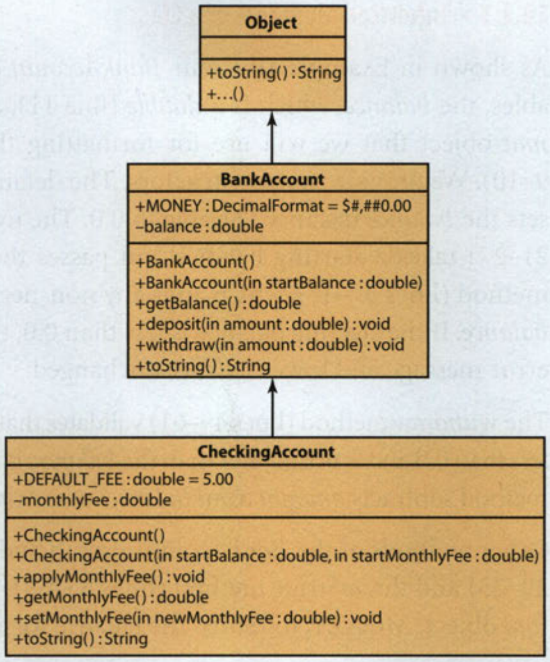

Object
+toString() : String
+...()
BankAccount
+MONEY : DecimalFormat = $#,##0.00
–balance : double
+BankAccount()
+BankAccount(in startBalance : double)
+getBalance() : double
+deposit(in amount : double) : void
+withdraw(in amount : double) : void
+toString() : String
CheckingAccount
+DEFAULT_FEE : double = 5.00
–monthlyFee : double
+CheckingAccount()
+CheckingAccount(in startBalance : double, in startMonthlyFee : double)
+applyMonthlyFee() : void
+getMonthlyFee() : double
+setMonthlyFee(in newMonthlyFee : double) : void
+toString() : String

Figure 10.3
The *BankAccount* Class Hierarchy

ate, subclasses can also provide new versions of inherited methods, which is called **overriding methods**.

SOFTWARE ENGINEERING TIP

The superclasses in a class hierarchy should contain fields and methods common to all subclasses. The subclasses should add specialized fields and methods.

Let's build a bank account class hierarchy. We start by defining a generic *BankAccount* superclass. The *BankAccount* class will contain the fields and methods that are common to all bank accounts. Then we will define a *CheckingAccount* class that inherits from the *BankAccount* class. The *CheckingAccount* class will add instance variables and methods that specifically support checking accounts. Our class hierarchy is shown in the UML diagram in Figure 10.3. In this diagram, we display the instance variables in the box immediately below the class name and the methods in the next lower box. A "+" preceding a class member indicates that the member is *public*, while a "–" indicates that the member is *private*. Each method's signature is given with each parameter and its type within parentheses and the return type following a colon. The *Object* class has more methods than we indicate on the UML diagram. However, the *toString* method is the only method of *Object* that we will deal with in this hierarchy, so we have omitted the other methods of *Object* on the diagram and indicate that other methods exist (+...()).

10.2.1 Inherited Members of a Class

As shown in Example 10.1, our *BankAccount* class has two instance variables, the *balance*, which is a *double* (line 11), and a constant *DecimalFormat* object that we will use for formatting the *balance* as money (lines 9–10). We provide two constructors. The default constructor (lines 13–19) sets the *balance* instance variable to 0.0. The overloaded constructor (lines 21–27) takes a starting balance and passes that parameter to the *deposit* method (lines 37–47), which adds any non-negative starting balance to the *balance.* If the starting balance is less than 0.0, the *deposit* method prints an error message and leaves *balance* unchanged.

The *withdraw* method (lines 49–61) validates that the *amount* parameter is not less than 0.0 and is not greater than the *balance.* If *amount* is valid, the *withdraw* method subtracts *amount* from *balance*; otherwise, it prints an error message.

Other methods of the *BankAccount* class include the *balance* accessor (lines 29–35) and the *toString* method (lines 63–69), which uses the *DecimalFormat* object, *MONEY*, to return the balance formatted as money.

```
 1 /**   BankAccount class, Version 1
 2 *     Anderson, Franceschi
 3 *     Represents a generic bank account
 4 */
 5 import java.text.DecimalFormat;
 6
 7 public class BankAccount
 8 {
 9   public final DecimalFormat MONEY
10                  = new DecimalFormat( "$#,##0.00" );
11   private double balance;
12
13   /** Default constructor
14   *   sets balance to 0.0
15   */
16   public BankAccount( )
17   {
18     balance = 0.0;
19   }
20
21   /** Overloaded constructor
```

```
*   @param startBalance  beginning balance
*/
public BankAccount( double startBalance )
{
  deposit( startBalance );
}

/** Accessor for balance
*   @return  current account balance
*/
public double getBalance( )
{
  return balance;
}

/** Deposit amount to account
*   @param amount  amount to deposit;
*                  amount must be >= 0.0
*/
public void deposit( double amount )
{
  if ( amount >= 0.0 )
    balance += amount;
  else
    System.err.println( "Deposit amount must be positive." );
}

/** withdraw amount from account
*    @param amount   amount to withdraw;
*                    amount must be >= 0.0
*                    amount must be <= balance
*/
public void withdraw( double amount )
{
  if (  amount >= 0.0 && amount <= balance )
    balance -= amount;
  else
    System.err.println( "Withdrawal amount must be positive "
                      + "and cannot be greater than balance" );
}

```

```
  /** toString
  *  @return  the balance formatted as money
  */
  public String toString( )
  {
    return ( "balance is " + MONEY.format( balance ) );
  }
}
```

EXAMPLE 10.1 *BankAccount* Class, Version 1

Now we can derive our *CheckingAccount* subclass. Example 10.2 shows Version 1 of our *CheckingAccount* class. For this initial version, we simply define the *CheckingAccount* class as extending *BankAccount* (line 5). The body of our class is empty for now, so we can demonstrate the fields and methods that a subclass inherits from its superclass.

```
1 /* CheckingAccount class, Version 1
2    Anderson, Franceschi
3 */
4
5 public class CheckingAccount extends BankAccount
6 { }
```

EXAMPLE 10.2 *CheckingAccount* Class, Version 1

When a class *extends* a superclass, all the *public* fields and methods of the superclass (excluding constructors) are inherited. That means that the *CheckingAccount* class inherits the *MONEY* instance variable and the *getBalance*, *deposit*, *withdraw*, and *toString* methods from the *BankAccount* class. An inherited field is directly accessible from the subclass, and an inherited method can be called by the other methods of the subclass. In addition, *public* inherited methods can be called by a client application using a subclass object reference.

Any fields and methods that are declared *private* are not inherited, and therefore are not directly accessible by the subclass. Nevertheless, the *private* fields and methods are still part of the subclass object. Remember that a *CheckingAccount* object "is a" *BankAccount* object, so a *CheckingAccount* object has a *balance* instance variable. However, the *balance* is declared to be *private* in the *BankAccount* class, so the *CheckingAccount* methods cannot directly access the *balance*. The *CheckingAccount* methods must call the accessor and mutator methods of the *BankAccount* class to access or change the value of *balance*.

Calling methods to retrieve and change values of an instance variable may seem a little tedious, but it enforces encapsulation. Allowing the *CheckingAccount* class to set the value of *balance* directly would complicate maintenance of the program. The *CheckingAccount* class would need to be responsible for maintaining a valid value for *balance*, which means that the *CheckingAccount* class would need to know all the validation rules for *balance* that the *BankAccount* class enforces. If these rules change, then the *CheckingAccount* class would also need to change. As long as the *BankAccount* class ensures the validity of *balance*, there is no reason for the *CheckingAccount* class to duplicate that code.

Java provides the ***protected*** access modifier so that fields and methods can be inherited by subclasses (like *public* fields and methods), while still being hidden from client classes (like *private* fields and methods). In addition, any class in the same package as the superclass can directly access a *protected* field, even if that class is not a subclass. Because more than one class can directly access a *protected* field, *protected* access compromises encapsulation and complicates maintenance of a program. For that reason, we prefer to use *private*, rather than *protected*, for our instance variables. We will discuss the difference between *private* and *protected* in greater detail later in the chapter.

Table 10.1 summarizes the fields and methods that are inherited by a subclass. We will add to this table as we explain more about inheritance.

Example 10.3 shows a client for the *CheckingAccount* class. In line 9, we instantiate an object of the *CheckingAccount* class. After instantiation, the *c1* object has two fields (*balance* and *MONEY*), and it has four methods (*getBalance*, *deposit*, *withdraw*, and *toString*).

We illustrate this by using the *c1* object reference to call the *deposit* method in line 12 and the *withdraw* method in line 15, and to call the *toString* method implicitly in lines 13 and 16. Figure 10.4 shows the output from this program.

TABLE 10.1 Inheritance Rules

Superclass Members	Inherited by Subclass?	Directly Accessible by Subclass?	Directly Accessible by Client of Subclass?
public fields	yes	yes, by using field name	yes
public methods	yes	yes, by calling method from other subclass methods	yes
protected fields	yes	yes, by using field name	no, must use accessors and mutators
protected methods	yes	yes, by calling method from subclass methods	no
private fields	no	no, must use accessors and mutators	no, must use accessors and mutators
private methods	no	no	no

```java
/* CheckingAccount Client, Version 1
   Anderson, Franceschi
*/

public class CheckingAccountClient
{
  public static void main( String [ ] args )
  {
    CheckingAccount c1 = new CheckingAccount( );
    System.out.println( "New checking account: " + c1 );

    c1.deposit( 350.75 );
    System.out.println( "\nAfter depositing $350.75: " + c1 );

    c1.withdraw( 200.25 );
    System.out.println( "\nAfter withdrawing $200.25: " + c1 );
  }
}
```

EXAMPLE 10.3 *CheckingAccount* Client, Version 1

Figure 10.4
Output from *CheckingAccountClient*, Version 1

```
New checking account: balance is $0.00

After depositing $350.75: balance is $350.75

After withdrawing $200.25: balance is $150.50
```

10.2.2 Subclass Constructors

Although constructors are *public*, they are not inherited by subclasses. However, to initialize the *private* instance variables of the superclass, a subclass constructor can call a superclass constructor either implicitly or explicitly.

When a class extends another class, the default constructor of the subclass automatically calls the default constructor of the superclass. This is called **implicit** invocation. Athough we did not code any constructors in our *CheckingAccount* class in Example 10.2, we were able to instantiate a *CheckingAccount* object (with a 0.0 *balance*) because the Java compiler provided a default constructor for the *CheckingAccount* class, which implicitly called the default constructor of the *BankAccount* class.

To **explicitly** call the constructor of the direct superclass, the subclass constructor uses the following syntax:

```
super( argument list );
```

Thus, if we want to instantiate a *CheckingAccount* object with a starting balance other than 0.0, we need to provide an overloaded constructor for the *CheckingAccount* class. That constructor will take the starting balance as a parameter and pass that starting balance to the overloaded constructor in the *BankAccount* class.

This call to the direct superclass constructor, if used, must be the first statement in the subclass constructor. Otherwise, the following compiler error is generated:

```
call to super must be first statement in constructor
```

COMMON ERROR TRAP

In a constructor, the call to the direct superclass constructor, if used, must be the first statement.

Example 10.4 shows Version 2 of the *BankAccount* class, which for simplicity, and to help us focus on constructors, has only a default and overloaded constructor and the *toString* method. To illustrate the order in which the constructors execute, we print a message in each constructor (lines 21 and 33), indicating that it has been called.

```
 1 /**   BankAccount class, Version 2
 2 *     Constructors and toString method only
 3 *     Anderson, Franceschi
 4 *     Represents a generic bank account
 5 */
 6
```

```
import java.text.DecimalFormat;

public class BankAccount
{
   public final DecimalFormat MONEY
                  = new DecimalFormat( "$#,##0.00" );
   private double balance;

   /** Default constructor
   *   sets balance to 0.0
   */
   public BankAccount( )
   {
     balance = 0.0;
     System.out.println( "In BankAccount default constructor" );
   }

   /** Overloaded constructor
   *   @param startBalance  beginning balance
   */
   public BankAccount( double startBalance )
   {
     if ( balance >= 0.0 )
         balance = startBalance;
     else
         balance = 0.0;
     System.out.println( "In BankAccount overloaded constructor" );
   }

   /** toString
   *  @return  the balance formatted as money
   */
   public String toString( )
   {
     return ( "balance is " + MONEY.format( balance ) );
   }
}
```

EXAMPLE 10.4 ***BankAccount*, Version 2**

Example 10.5 shows Version 2 of the *CheckingAccount* class, which has both a default constructor and an overloaded constructor. Again, we have inserted messages (lines 13–14 and 24–25) to indicate when a constructor is called.

```
 1 /* CheckingAccount class, Version 2
 2    Anderson, Franceschi
 3 */
 4
 5 public class CheckingAccount extends BankAccount
 6 {
 7    /** default constructor
 8    *   explicitly calls the BankAccount default constructor
 9    */
10    public CheckingAccount( )
11    {
12        super( ); // optional, call BankAccount constructor
13        System.out.println( "In CheckingAccount "
14                           + "default constructor" );
15    }
16
17    /** overloaded constructor
18    *   calls BankAccount overloaded constructor
19    *   @param  startBalance  starting balance
20    */
21    public CheckingAccount( double startBalance )
22    {
23        super( startBalance ); // call BankAccount constructor
24        System.out.println( "In CheckingAccount "
25                          + "overloaded constructor" );
26    }
27 }
```

EXAMPLE 10.5 ***CheckingAccount* Class, Version 2**

In the *CheckingAccount* default constructor, we explicitly call the default constructor of the *BankAccount* class (line 12). This statement is optional; without it, the *BankAccount* default constructor is still called implicitly.

In the *CheckingAccount* overloaded constructor, we pass the *startBalance* parameter to the *BankAccount* constructor (line 23) to initialize the

COMMON ERROR TRAP

An attempt by a subclass to directly access a *private* field or call a *private* method defined in a superclass will generate a compiler error. To set initial values for *private* variables, call the appropriate constructor of the direct superclass.

balance instance variable. Because *balance* has *private* access in the *BankAccount* class, our *CheckingAccount* class cannot access *balance* directly. If we attempted to initialize the *balance* directly using the following statement:

```
balance = startBalance;
```

the compiler would generate the following error:

```
balance has private access in BankAccount
```

SOFTWARE ENGINEERING TIP

Overloaded constructors in a subclass should explicitly call the direct superclass constructor to initialize the fields in its superclasses.

Example 10.6 shows Version 2 of our *CheckingAccount* client. On line 10, we instantiate a *CheckingAccount* object using the default constructor and print the balance by implicitly calling the *toString* method on line 11. Then on line 14, we instantiate a second *CheckingAccount* object with a starting balance of $100.00. Again we verify the result by printing the balance (line 15).

```
 1 /* CheckingAccount Client, Version 2
 2    Anderson, Franceschi
 3 */
 4
 5 public class CheckingAccountClient
 6 {
 7    public static void main( String [ ] args )
 8    {
 9       // use default constructor
10       CheckingAccount c1 = new CheckingAccount( );
11       System.out.println( "New checking account: " + c1 + "\n" );
12
13       // use overloaded constructor
14       CheckingAccount c2 = new CheckingAccount( 100.00 );
15       System.out.println( "New checking account: " + c2 );
16    }
17 }
```

EXAMPLE 10.6 *CheckingAccountClient*, Version 2

Figure 10.5 shows the output from this program. As you can see, when we construct the *c1* object, the *BankAccount* default constructor runs. When it finishes, we print a message indicating that the *CheckingAccount* default

```
In BankAccount default constructor
In CheckingAccount default constructor
New checking account: balance is $0.00

In BankAccount overloaded constructor
In CheckingAccount overloaded constructor
New checking account: balance is $100.00
```

Figure 10.5
Output from Example 10.6

TABLE 10.2 Inheritance Rules for Constructors

Superclass Members	Inherited by Subclass?	Directly Accessible by Subclass?	Directly Accessible by Client of Subclass Using a Subclass Reference?
constructors	no	yes, using `super( arg list )` in a subclass constructor	no

constructor is running. Similarly, when we construct the *c2* object, the *BankAccount* overloaded constructor runs, then we print a message from the *CheckingAccount* overloaded constructor.

Table 10.2 summarizes the inheritance rules for constructors.

10.2.3 Adding Specialization to the Subclass

At this point, our *CheckingAccount* class provides no more functionality than the *BankAccount* class. But our purpose for defining a *CheckingAccount* class was to provide support for a specialized type of bank account. To add specialization to our *CheckingAccount* subclass, we define new fields and methods. For example, we can define a *monthlyFee* instance variable, as well as an accessor and mutator method for the monthly fee and a method to charge the monthly fee to the account.

Example 10.7 shows Version 3 of the *CheckingAccount* class with the specialization added. This version *extends* the complete *BankAccount* class shown in Example 10.1. We added the *monthlyFee* instance variable on line 8, as well as a constant default value for the monthly fee (line 7). Our default constructor (lines 10–18) still calls the default constructor of the *BankAccount* class to initialize the *balance*, but it also initializes the *monthlyFee* to the default value.

Similarly, the overloaded constructor (lines 20–30) passes the *startBalance* parameter to the overloaded constructor of the *BankAccount* class and adds a *startMonthlyFee* parameter to accept an initial value for the *monthlyFee*, which it passes to the *setMonthlyFee* mutator method (lines 48–57).

The *applyMonthlyFee* method (lines 32–38), which charges the monthly fee to the checking account, calls the *withdraw* method inherited from the *BankAccount* class to access the *balance* instance variable, which is declared *private* in the *BankAccount* class.

```
 1 /* CheckingAccount class, Version 3
 2    Anderson, Franceschi
 3 */
 4
 5 public class CheckingAccount extends BankAccount
 6 {
 7    public final double DEFAULT_FEE = 5.00;
 8    private double monthlyFee;
 9
10    /** default constructor
11    *   explicitly calls the BankAccount default constructor
12    *   set monthlyFee to default value
13    */
14    public CheckingAccount( )
15    {
16       super( ); // optional
17       monthlyFee = DEFAULT_FEE;
18    }
19
20    /** overloaded constructor
21    *  calls BankAccount overloaded constructor
22    *  @param  startBalance  starting balance
```

```
    * @param  startMonthlyFee starting monthly fee
    */
    public CheckingAccount( double startBalance,
                            double startMonthlyFee )
    {
      super( startBalance ); // call BankAccount constructor
      setMonthlyFee( startMonthlyFee );
    }

    /** applyMonthlyFee method
    * charges the monthly fee to the account
    */
    public void applyMonthlyFee( )
    {
      withdraw( monthlyFee );
    }

    /** accessor method for monthlyFee
    *  @return  monthlyFee
    */
    public double getMonthlyFee( )
    {
      return monthlyFee;
    }

    /** mutator method for monthlyFee
    *  @param newMonthlyFee new value for monthlyFee
    */
    public void setMonthlyFee( double newMonthlyFee )
    {
      if ( newMonthlyFee >= 0.0 )
         monthlyFee = newMonthlyFee;
      else
         System.err.println( "Monthly fee cannot be negative" );
    }
}
```

EXAMPLE 10.7 ***CheckingAccount*, Version 3**

Example 10.8 shows Version 3 of our client program, which instantiates a *CheckingAccount* object and charges the monthly fee. The output is shown in Figure 10.6.

Figure 10.6
Output from *CheckingAccountClient*, Version 3

```
New checking account:
balance is $100.00; monthly fee is 7.5

After charging monthly fee:
balance is $92.50; monthly fee is 7.5
```

```
/* CheckingAccount Client, Version 3
   Anderson, Franceschi
*/

public class CheckingAccountClient
{
   public static void main( String [ ] args )
   {
     CheckingAccount c3 = new CheckingAccount( 100.00, 7.50 );
     System.out.println( "New checking account:\n"
                         + c3.toString( )
                         + "; monthly fee is "
                         + c3.getMonthlyFee( ) );

     c3.applyMonthlyFee( );  // charge the fee to the account
     System.out.println( "\nAfter charging monthly fee:\n"
                         + c3.toString( )
                         + "; monthly fee is "
                         + c3.getMonthlyFee( ) );
   }
}
```

EXAMPLE 10.8 *CheckingAccountClient*, Version 3

10.2.4 Overriding Inherited Methods

When the methods our subclass inherits do not fulfill the functions we need, we can **override** the inherited methods by providing new versions of those methods. We have seen this feature in our applets. Whenever we wrote an *init* or *paint* method, we were overriding the corresponding method inherited from the *JApplet* class.

To override an inherited method, we provide a new method with the same header as the inherited method; that is, the new method must have the same name, the same number and type of parameters, and the same return type. Overriding a method makes the inherited version of the method invisible to the client of the subclass. We say that the overridden method is hidden from the client. When the client calls the method using a subclass object reference, the subclass version of the method is invoked.

Methods in a subclass can still access the inherited version of the method by preceding the method call with the *super* keyword, as in the following syntax:

```
super.methodName( argument list )
```

In our *CheckingAccount* class, we inherited the *toString* method from the *BankAccount* class. But this method returns only the *balance*. In Example 10.8, we needed to call the *CheckingAccount* method *getMonthlyFee* to print the value of *monthlyFee*. Furthermore, as Figure 10.6 shows, the *balance* value is formatted and the *monthlyFee* value is not. Instead, the *toString* method in the *CheckingAccount* class should return formatted versions of both the *balance* and the *monthlyFee*. We can accomplish this by overriding the inherited *toString* method.

Example 10.9 shows Version 4 of the *CheckingAccount* class with the new *toString* method (lines 59–67). To format the *balance*, we call the *toString* method of the *BankAccount* class (line 65), then add the formatted value of *monthlyFee* to the *String* being returned. Notice that we didn't need to instantiate a new *DecimalFormat* object in order to format the *monthlyFee* instance variable. Because the *MONEY* object is declared to be *public* in the *BankAccount* class, we inherited the *MONEY* object, so we can simply call the *format* method using the *MONEY* object reference. An advantage to making the *MONEY* object *public* is that both the balance and the monthly fee will be printed using the same formatting rules. Another advantage is that if we want to change the formatting for printing the data, we need to make only one change: We redefine the value of the *MONEY* constant in the *BankAccount* class.

```
1  /* CheckingAccount class, Version 4
2     Anderson, Franceschi
3  */
4
5  public class CheckingAccount extends BankAccount
6  {
```

```
public final double DEFAULT_FEE = 5.00;
private double monthlyFee;

/** default constructor
*   explicitly calls the BankAccount default constructor
*   set monthlyFee to default value
*/
public CheckingAccount( )
{
   super( ); // call BankAccount constructor
   monthlyFee = DEFAULT_FEE;
}

/** overloaded constructor
*  calls BankAccount overloaded constructor
*  @param  startBalance  starting balance
*  @param  startMonthlyFee starting monthly fee
*/
public CheckingAccount( double startBalance,
                        double startMonthlyFee )
{
   super( startBalance ); // call BankAccount constructor
   setMonthlyFee( startMonthlyFee );
}

/** applyMonthlyFee method
* charges the monthly fee to the account
*/
public void applyMonthlyFee( )
{
   withdraw( monthlyFee );
}

 /** accessor method for monthlyFee
  *  @return  monthlyFee
  */
 public double getMonthlyFee( )
 {
   return monthlyFee;
 }

/** mutator method for monthlyFee
*  @param newMonthlyFee new value for monthlyFee
*/
public void setMonthlyFee( double newMonthlyFee )
```

```
    {
      if ( newMonthlyFee >= 0.0 )
        monthlyFee = newMonthlyFee;
      else
        System.err.println( "Monthly fee cannot be negative" );
    }

    /* toString method
    *  @return String containing formatted balance and monthlyFee
    *    invokes superclass toString to format balance
    */
    public String toString( )
    {
      return super.toString( )
        + "; monthly fee is " + MONEY.format( monthlyFee );
    }
}
```

EXAMPLE 10.9 ***CheckingAccount*** **Class, Version 4**

Example 10.10 shows Version 4 of the *CheckingAccountClient* class. In this class, we again instantiate a *CheckingAccount* object with an initial balance of $100.00 and a monthly fee of $7.50 (line 9), then implicitly invoke the *toString* method to print the data of the object (line 10). This time, we invoke the *toString* method of the *CheckingAccount* class, which returns both the *balance* and *monthlyFee* values, formatted as money, as shown in Figure 10.7.

```
 1 /* CheckingAccount Client, Version 4
 2    Anderson, Franceschi
 3 */
 4
 5 public class CheckingAccountClient
 6 {
 7   public static void main( String [ ] args )
 8   {
 9     CheckingAccount c4 = new CheckingAccount( 100.00, 7.50 );
10     System.out.println( "New checking account:\n" + c4 );
11   }
12 }
```

EXAMPLE 10.10 ***CheckingAccountClient*****, Version 4**

Figure 10.7
Output from *CheckingAccountClient*, Version 4

```
New checking account:
balance is $100.00; monthly fee is $7.50
```

COMMON ERROR TRAP

Do not confuse overriding a method with overloading a method. A subclass overriding a method provides a new version of that method, which hides the superclass version. A client calling the method using a subclass object reference will invoke the subclass version. A class overloading a method provides a new version of that method, which varies in the number and/or type of parameters. All *public* versions of overloaded methods are available to be called by the client of the class.

SOFTWARE ENGINEERING TIP

Methods that override inherited methods should explicitly call the direct superclass method whenever appropriate, in order to process inherited fields.

Table 10.3 summarizes the inheritance rules for inherited methods that have been overridden.

When you override a method, be sure that the method signature is identical to the inherited method. However, you can override an inherited method with a method that specifies a subclass object reference where the original method used a superclass reference. This is possible because a subclass object is a superclass object, so a subclass object reference can be substituted for any superclass object reference. If two methods of a class have the same name but different signatures (that is, if the number, order, or type of parameters is different), then the method is *overloaded*, not *overridden.*

For example, if in an applet we were to write the *init* method with the following header:

```
public void init( int a )
```

then our *init* method has a different signature from the *init* method we inherited from the *JApplet* class, which does not take any parameters. In this case, we are overloading the *init* method, not overriding it. In other words, we are providing an additional version of the *init* method. The inherited version is still visible and available to be called. When the applet starts executing, the browser or applet viewer would call the *init* method of the *JApplet* class, not our *init* method.

TABLE 10.3 Inheritance Rules for Overridden Methods

Superclass Members	Inherited by Subclass?	Directly Accessible by Subclass?	Directly Accessible by Client of Subclass Using a Subclass Reference?
public or *protected* inherited methods that have been overridden in the subclass	no	yes, using `super.methodName( arg list )`	no

TABLE 10.4 Overriding vs. Overloading Methods

	Method Names	Argument Lists	Return Types	Directly Accessible by Subclass Client Using a Subclass Object Reference?
Overriding a *public* Method	identical	identical	identical	only the subclass version can be called
Overloading a *public* Method	identical	different in number or type of parameters	identical	all versions of the overloaded method can be called

Skill Practice
with these end-of-chapter questions

10.10.1 Multiple Choice Exercises
Questions 1, 2, 4, 8, 9

10.10.3 Fill In the Code
Questions 21,22,23,24

10.10.5 Debugging Area
Questions 32, 34, 35

10.10.6 Write a Short Program
Questions 36,37,38,39,40,41,42,43,44,45,46

10.10.8 Technical Writing
Question 56

Table 10.4 illustrates the differences between overriding *public* methods and overloading *public* methods.

10.3 The *protected* Access Modifier

We have seen that the subclass does not inherit constructors or *private* members of the superclass. However, the superclass constructors are still available to be called from the subclass and the *private* fields of the superclass are implemented as fields of the subclass.

Although *private* fields preserve encapsulation, there is additional processing overhead involved with calling methods. Whenever a method is called, the JVM saves the return address and makes copies of the arguments. Then when a value-returning method completes, the JVM makes a copy of the return value available to the caller. The *protected* access modifier was designed to avoid this processing overhead and to facilitate coding by allowing the subclass to access any *protected* field without calling its accessor or mutator method.

Be aware, however, that *protected* fields and methods also can be accessed directly by other classes in the same package, even if the classes are not within the same inheritance hierarchy.

To classes outside the package, a *protected* member of a class has the same restrictions as a *private* member. In other words, a class outside the package in which the *protected* member is declared may not call any *protected* methods and must access any *protected* fields through *public* accessor or mutator methods.

The *protected* access modifier has tradeoffs. As we mentioned, any fields declared as *protected* can be accessed directly by subclasses. Doing so, however, compromises encapsulation because multiple classes can set the value of an instance variable defined in another class.

Thus, maintaining classes that define or use *protected* members becomes more difficult. For example, we need to verify that any class that has access to the *protected* instance variable either does not set the variable's value, or if the class does change the value, that the new value is valid. Because of this added maintenance complexity, we recommend that *protected* access be used only when high performance is essential.

SOFTWARE ENGINEERING TIP

Unless high performance is a critical requirement, avoid using the *protected* access modifier because doing so compromises encapsulation and complicates the maintenance of a program. Where possible, call superclass methods to change the values of *protected* instance variables.

We also recommend that subclass methods avoid directly setting the value of a *protected* instance variable. Instead, wherever possible, call superclass methods when values of *protected* variables need to be changed.

To illustrate how *protected* access can be used in class hierarchies, let's look closely at our *CheckingAccount* class. We have been calling the *withdraw* method inherited from the *BankAccount* class to apply the monthly fee. However, the *withdraw* method leaves the *balance* unchanged if the withdrawal amount is greater than the balance. Thus, if the account does not have sufficient funds, the monthly fee is not charged. We would like the *CheckingAccount* class to be able to charge the monthly fee to the account and let the balance become negative. When this happens, we will print a warning message that the account is overdrawn.

To accomplish this, we declare the *balance* instance variable to be *protected* instead of *private*. This allows us to directly access *balance* inside the *applyMonthlyFee* method of the *CheckingAccount* class, because *balance* is now inherited by *CheckingAccount*.

Example 10.11 shows the *BankAccount* class, Version 3. The only change, compared to Version 1 (Example 10.1), is that the *balance* instance variable is declared as *protected*, rather than *private* (line 11).

```
 1 /**   BankAccount class, Version 3
 2 *     Anderson, Franceschi
 3 *     Represents a generic bank account
 4 */
 5 import java.text.DecimalFormat;
 6
 7 public class BankAccount
 8 {
 9   public final DecimalFormat MONEY
10                    = new DecimalFormat( "$#,##0.00" );
11   protected double balance;
12
13   /** Default constructor
14   *   sets balance to 0.0
15   */
16   public BankAccount( )
17   {
18     balance = 0.0;
19   }
20
21   /** Overloaded constructor
22   *   @param startBalance  beginning balance
23   */
24   public BankAccount( double startBalance )
25   {
26     deposit( startBalance );
27   }
28
29   /** Accessor for balance
30   *   @return  current account balance
31   */
```

```java
  public double getBalance( )
  {
    return balance;
  }

  /** Deposit amount to account
  *   @param amount  amount to deposit;
  *                  amount must be >= 0.0
  */
  public void deposit( double amount )
  {
    if ( amount >= 0.0 )
      balance += amount;
    else
      System.err.println( "Deposit amount must be positive." );
  }

  /** withdraw amount from account
  *    @param amount   amount to withdraw;
  *                    amount must be >= 0.0
  *                    amount must be <= balance
  */
  public void withdraw( double amount )
  {
    if (  amount >= 0.0 && amount <= balance )
      balance -= amount;
    else
      System.err.println( "Withdrawal amount must be positive "
                          + "and cannot be greater than balance" );
  }

  /** toString
  *  @return  the balance formatted as money
  */
  public String toString( )
  {
    return ( "balance is " + MONEY.format( balance ) );
  }
}
```

EXAMPLE 10.11 ***BankAccount* Class, Version 3**

Example 10.12 shows Version 5 of the *CheckingAccount* class, which inherits from the *BankAccount* class in Example 10.11 that declares the *balance* as *protected.* The *CheckingAccount* class now inherits *balance,* and our *CheckingAccount* methods can access the *balance* variable directly. Nevertheless, in the default and overloaded constructors, we still call the superclass constructor to set the value of *balance* (lines 16 and 28). Otherwise, to avoid setting *balance* to an invalid initial value, we would need to know the validation rules for *balance* in *BankAccount* and unnecessarily duplicate that code.

Also, in the *toString* method (lines 61–69), we call the *toString* method of the *BankAccount* class. Again, we do this to be consistent with the superclass functionality and to avoid duplicating code in the *BankAccount* class.

In the *applyMonthlyFee* method (lines 32–40), however, we access *balance* directly. For this checking account, our bank will charge the monthly fee even if it results in a negative balance for the account, so we subtract *monthlyFee* from *balance*, which allows the balance to be negative, and if so, we print a warning. Notice that we change the value of *balance* directly instead of calling the *getBalance* and *withdraw* methods.

```
 1 /* CheckingAccount class, Version 5
 2    Anderson, Franceschi
 3 */
 4
 5 public class CheckingAccount extends BankAccount
 6 {
 7    public final double DEFAULT_FEE = 5.00;
 8    private double monthlyFee;
 9
10    /** default constructor
11    *   explicitly calls the BankAccount default constructor
12    *   set monthlyFee to default value
13    */
14    public CheckingAccount( )
15    {
16      super( );      // call BankAccount constructor
17      monthlyFee = DEFAULT_FEE;
18    }
19
20    /** overloaded constructor
21    *  calls BankAccount overloaded constructor
```

```
* @param  startBalance    starting balance
*  @param  startMonthlyFee starting monthly fee
*/
public CheckingAccount( double startBalance,
                        double startMonthlyFee )
{
  super( startBalance );  // call BankAccount constructor
  setMonthlyFee( startMonthlyFee );
}

/** applyMonthlyFee method
* charges the monthly fee to the account
*/
public void applyMonthlyFee( )
{
  balance -= monthlyFee;
  if ( balance < 0.0 )
    System.err.println( "Warning: account is overdrawn" );
}

/** accessor method for monthlyFee
*  @return  monthlyFee
*/
public double getMonthlyFee( )
{
  return monthlyFee;
}

/** mutator method for monthlyFee
*  @param newMonthlyFee new value for monthlyFee
*/
public void setMonthlyFee( double newMonthlyFee )
{
  if ( newMonthlyFee >= 0.0 )
     monthlyFee = newMonthlyFee;
  else
     System.err.println( "Monthly fee cannot be negative" );
}

/* toString method
*  @return String containing formatted balance and monthlyFee
*    invokes superclass toString to format balance
*/
```

```
  public String toString( )
  {
    return super.toString( )
           + "; monthly fee is " + MONEY.format( monthlyFee );
  }
}
```

EXAMPLE 10.12 ***CheckingAccount* Class, Version 5**

Example 10.13 shows Version 5 of the *CheckingAccountClient* class. In this class, we again instantiate a *CheckingAccount* object with an initial balance of $100.00 and a monthly fee of $7.50 (line 9). We then call *withdraw* (line 12), so that the resulting balance is less than the monthly fee. Next we call the *applyMonthlyFee* method (line 16). When we print the state of the object in line 17, the *balance* is negative. The output of Example 10.13 is shown in Figure 10.8.

```
 1 /* CheckingAccount Client, Version 5
 2     Anderson, Franceschi
 3 */
 4
 5 public class CheckingAccountClient
 6 {
 7   public static void main( String [ ] args )
 8   {
 9     CheckingAccount c5 = new CheckingAccount( 100.00, 7.50 );
10     System.out.println( "New checking account:\n" + c5 );
11
12     c5.withdraw( 95 );
13     System.out.println( "\nAfter withdrawing $95:\n" + c5 );
14
15     System.out.println( "\nApplying the monthly fee:" );
16     c5.applyMonthlyFee( );
17     System.out.println( "\nAfter charging monthly fee:\n" + c5 );
18   }
19 }
```

EXAMPLE 10.13 ***CheckingAccountClient* Class, Version 5**

Table 10.5 compiles all the inheritance rules we have discussed.

Figure 10.8
Output from *Checking-AccountClient*, Version 5

```
New checking account:
balance is $100.00; monthly fee is $7.50

After withdrawing $95:
balance is $5.00; monthly fee is $7.50

Applying the monthly fee:
Warning: account is overdrawn

After charging monthly fee:
balance is -$2.50; monthly fee is $7.50
```

TABLE 10.5 Inheritance Rules

Superclass Members	Inherited by Subclass?	Directly Accessible by Subclass?	Directly Accessible by Client of Subclass?
public fields	yes	yes, by using field name	yes
public methods	yes	yes, by calling method from other subclass methods	yes, by calling method using a subclass object reference
protected fields	yes	yes, by using field name	no, must use accessors and mutators
protected methods	yes	yes, by calling method from subclass methods	no
private fields	no	no, must use accessors and mutators	no, must use accessors and mutators
private methods	no	no	no
constructors	no	yes, using `super( arg list )` in a subclass constructor	no
public or *protected* inherited methods that have been overridden in the subclass	no	yes, using `super.methodName( arg list )`	no

Skill Practice
with these end-of-chapter questions

10.10.2 Reading and Understanding Code

Questions 12,13,14,15,16,17,18,19,20

CODE IN ACTION

On the CD-ROM included with this book, you will find a Flash movie with a step-by-step illustration of the use of inheritance in a program. Click on the link for Chapter 10 to start the movie.

10.4 Programming Activity 1: Using Inheritance

For this Programming Activity, you will create the *SavingsAccount* class, which inherits directly from the *BankAccount* class. The *SavingsAccount* class is similar to the *CheckingAccount* class in that both classes inherit from *BankAccount*. Figure 10.9 shows the resulting hierarchy.

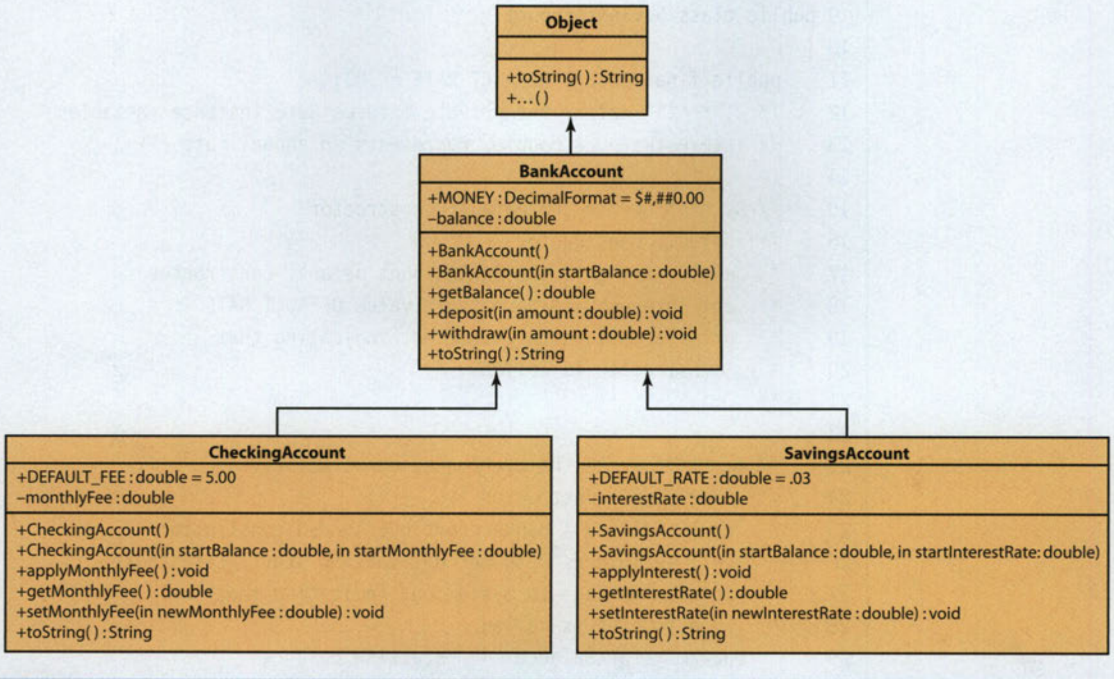

Object
+toString( ) : String
+...( )
BankAccount
+MONEY : DecimalFormat = $#,##0.00
–balance : double
+BankAccount( )
+BankAccount(in startBalance : double)
+getBalance( ) : double
+deposit(in amount : double) : void
+withdraw(in amount : double) : void
+toString( ) : String
CheckingAccount
+DEFAULT_FEE : double = 5.00
–monthlyFee : double
+CheckingAccount( )
+CheckingAccount(in startBalance : double, in startMonthlyFee : double)
+applyMonthlyFee( ) : void
+getMonthlyFee( ) : double
+setMonthlyFee(in newMonthlyFee : double) : void
+toString( ) : String
SavingsAccount
+DEFAULT_RATE : double = .03
–interestRate : double
+SavingsAccount( )
+SavingsAccount(in startBalance : double, in startInterestRate: double)
+applyInterest( ) : void
+getInterestRate( ) : double
+setInterestRate(in newInterestRate : double) : void
+toString( ) : String

Figure 10.9
Bank Account Hierarchy

The *SavingsAccount* class inherits from the version of the *BankAccount* class in which the *balance* is declared to be *private.* The *SavingsAccount* subclass adds an annual *interestRate* instance variable, as well as supporting methods to access, change, and apply the interest rate to the account balance.

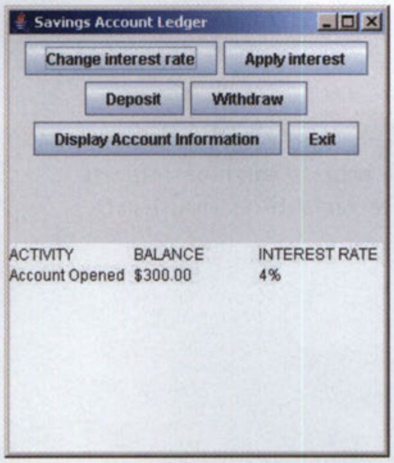
Instructions

Copy the source files in the Programming Activity 1 directory for this chapter to a directory on your computer. Load the *SavingsAccount.java* source file and search for five asterisks in a row (*****). This will position you to the six locations in the file where you will add code to complete the *SavingsAccount* class. The *SavingsAccount.java* file is shown in Example 10.14.

```
/* SavingsAccount class
   Anderson, Franceschi
*/

import java.text.NumberFormat;

// 1. ***** indicate that SavingsAccount inherits
//          from BankAccount
public class SavingsAccount
{
   public final double DEFAULT_RATE = .03;
   // 2. ****** define the private interestRate instance variable
   // interestRate, a double, represents an annual rate

   // 3. ***** write the default constructor
   /** default constructor
   *   explicitly call the BankAccount default constructor
   *   set interestRate to default value DEFAULT_RATE
   *   print a message to System.out indicating that
   *    constructor is called
   */

   // 4. ***** write the overloaded constructor
   /** overloaded constructor
   *   explicitly call BankAccount overloaded constructor
   *   call setInterestRate method, passing startInterestRate
   *   print a message to System.out indicating that
   *    constructor is called
   *  @param  startBalance      starting balance
   *  @param  startInterestRate starting interest rate
   */


   // 5. ****** write this method:
   /** applyInterest method, no parameters, void return value
   *  call the deposit method, passing a month's worth of interest
   *  remember that interestRate instance variable is annual rate
   */


   /** accessor method for interestRate
   *  @return  interestRate
   */
   public double getInterestRate( )
   {
     return interestRate;
   }

   /** mutator method for interestRate
   *  @param  newInterestRate new value for interestRate
   *          newInterestRate must be >= 0.0
   *            if not, print an error message
   */
   public void setInterestRate( double newInterestRate )
   {
     if ( newInterestRate >= 0.0 )
        interestRate = newInterestRate;
     else
        System.err.println( "Interest rate cannot be negative" );
   }

   // 6. ***** write this method
   /* toString method
   *  @return String containing formatted balance and interestRate
   *    invokes superclass toString to format balance
   *    formats interestRate as percent using a NumberFormat object
   *    To create a NumberFormat object for formatting percentages
   *    use the getPercentInstance method in the NumberFormat class,
   *    which has this API:
   *       static NumberFormat getPercentInstance( )
   */

}
```

EXAMPLE 10.14 ***SavingsAccount.java***

Figure 10.10
The Teller Window



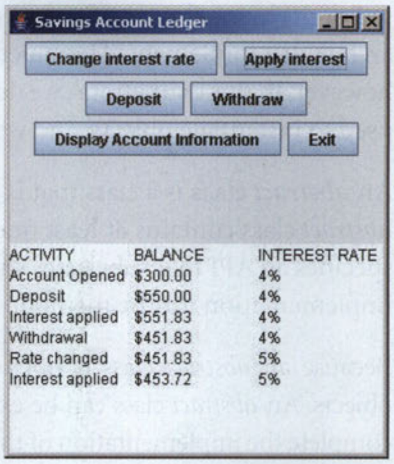
```
ACTIVITY        BALANCE     INTEREST RATE
Account Opened  $300.00     4%
```

When you have completed the six tasks, load, compile, and run the Teller application (*Teller.java*), which you will use to test your *SavingsAccount* class. When the Teller application begins, you will be prompted with a dialog box for a starting balance. If you press "Enter" or the "OK" button without entering a balance, the Teller application will use the default constructor to instantiate a *SavingsAccount* object. If you enter a starting balance, the Teller application will prompt you for an interest rate and will instantiate a *SavingsAccount* object using the overloaded constructor. Once the *SavingsAccount* object has been instantiated, the Teller application will open the window shown in Figure 10.10, which provides buttons you can use to call the *SavingsAccount* methods to test your code.

Below the buttons is a ledger that displays the current state of the savings account. As you click on the various buttons, the ledger will display the operation performed and the values of the balance and the interest rate when that operation is complete.

The operations performed by each button are already coded for you and are the following:

- *Change Interest Rate*—prompts for a new interest rate and calls your *setInterestRate* method
- *Apply Interest*—calls your *applyInterest* method
- *Deposit*—prompts for the deposit amount and calls the *deposit* method inherited from *BankAccount*
- *Withdraw*—prompts for the withdrawal amount and calls the *withdraw* method inherited from *BankAccount*



```
ACTIVITY          BALANCE    INTEREST RATE
Account Opened    $300.00    4%
Deposit           $550.00    4%
Interest applied  $551.83    4%
Withdrawal        $451.83    4%
Rate changed      $451.83    5%
Interest applied  $453.72    5%
```

Figure 10.11
Sample Teller Window After Performing Several Operations

- *Display Account Information*—calls your *toString* method and displays the result in a dialog box
- *Exit*—exits the program

Figure 10.11 shows the Teller window after several operations have been performed.

? DISCUSSION QUESTIONS

1. Explain why the Teller application can call the *withdraw* and *deposit* methods using a *SavingsAccount* object reference, even though we did not define these methods.
2. Explain why your *applyInterest* method in the *SavingsAccount* class needs to call the *deposit* method of the *BankAccount* class.

10.5 *Abstract* Classes and Methods

In our Bank Account hierarchy, we could instantiate *BankAccount* objects, *CheckingAccount* objects, and *SavingsAccount* objects. In some situations, however, we will design a class hierarchy where one or more classes at the top of the hierarchy specify patterns for methods that subclasses in the hierarchy must implement. The superclasses do not implement these methods, however. In these situations, we do not intend that these superclasses will be used to instantiate objects, and we define the superclasses as ***abstract.***

An ***abstract*** **class** is a class that is not completely implemented. Usually, an *abstract* class contains at least one ***abstract*** **method**, that is, a method that specifies an API that subclasses should implement, but does not provide an implementation for the method.

Because an *abstract* class is not complete, it cannot be used to instantiate objects. An *abstract* class can be extended, however, so that its subclasses can complete the implementation of the *abstract* methods and can be instantiated.

A class is declared to be *abstract* by including the *abstract* keyword in the class header, as shown in the following syntax:

```
accessModifier abstract class ClassName
```

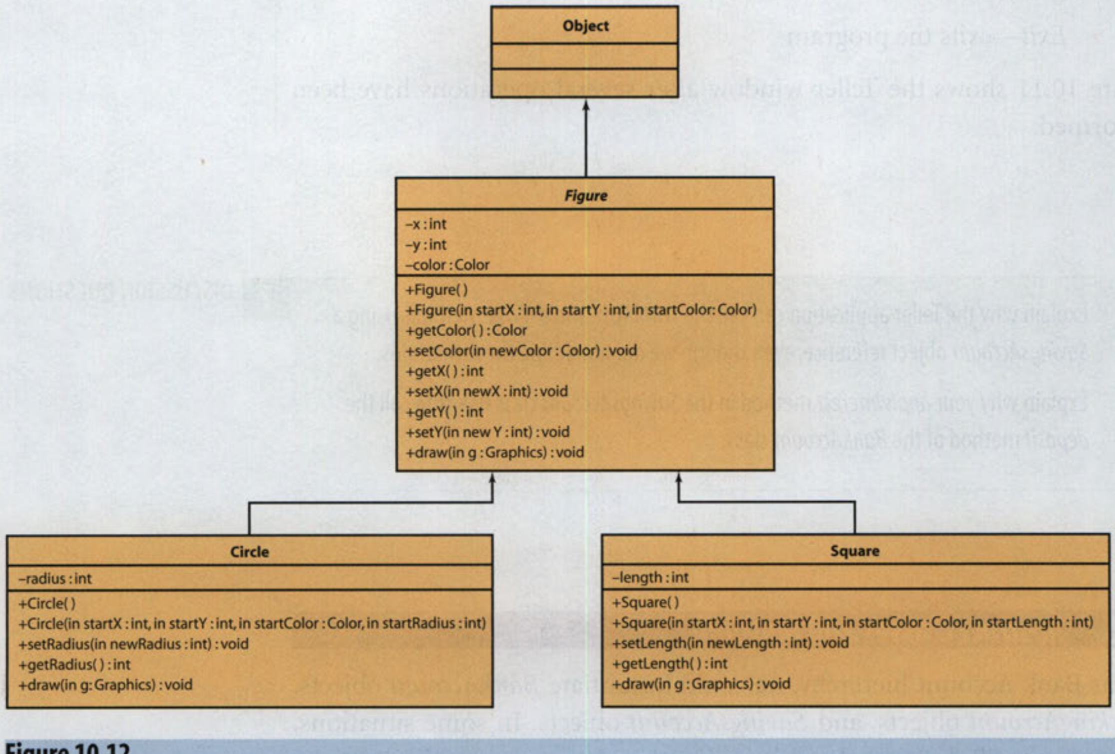

Object
Figure
–x : int
–y : int
–color : Color
+Figure( )
+Figure(in startX : int, in startY : int, in startColor: Color)
+getColor( ) : Color
+setColor(in newColor : Color) : void
+getX( ) : int
+setX(in newX : int) : void
+getY( ) : int
+setY(in new Y : int) : void
+draw(in g : Graphics) : void
Circle
–radius : int
+Circle( )
+Circle(in startX : int, in startY : int, in startColor : Color, in startRadius : int)
+setRadius(in newRadius : int) : void
+getRadius( ) : int
+draw(in g: Graphics) : void
Square
–length : int
+Square( )
+Square(in startX : int, in startY : int, in startColor : Color, in startLength : int)
+setLength(in newLength : int) : void
+getLength( ) : int
+draw(in g : Graphics) : void

Figure 10.12
The *Figure* Hierarchy

An *abstract* method is defined by including the *abstract* keyword in the method header and by using a semicolon to indicate that there is no code for the method, as shown in the following syntax:

```
accessModifier abstract returnType methodName( argument list );
```

Note that we do not include opening and closing curly braces for the method body—just a semicolon to indicate that the *abstract* method does not have a body.

For example, to draw figures, we can set up the hierarchy shown in Figure 10.12. The root superclass under *Object* is the *abstract Figure* class, and we derive two subclasses: *Circle* and *Square*. In the UML diagram, the name of the *Figure* class is set in italics to indicate that it is an *abstract* class.

All figures will have an (*x*, *y*) coordinate and a color, so the *Figure* class defines three fields: two *ints*, *x* and *y*, and a *Color* object named *color*.

We want all classes in the hierarchy to provide a *draw* method to render the figure; however, the *Figure* class has nothing but a point to draw, so its *draw* method has nothing to do. Thus, we will not provide an implementation of the *draw* method in the *Figure* class; instead, we will define the *draw* method as an *abstract* method.

Let's look at the code for the *Figure* hierarchy in detail. Example 10.15 shows the *abstract Figure* class. We define the class as *abstract* in the class header (line 7). The constructors (lines 13–22 and lines 24–37) instantiate the *x* and *y* values and the *Color* object that all figures will have in common. The *Figure* class also provides accessor and mutator methods for its instance variables. The *abstract draw* method (lines 88–91) provides the API for the *draw* method, but no implementation—just a semicolon. The *Circle* and *Square* subclasses of the *Figure* class will provide appropriate implementations of the *draw* method.

```
 1 /** abstract Figure superclass for drawing shapes
 2 *   Anderson, Franceschi
 3 */
 4 import java.awt.Graphics;
 5 import java.awt.Color;
 6
 7 public abstract class Figure
 8 {
 9   private int x;
10   private int y;
11   private Color color;
```

```

  /** default constructor
  *   sets x and y to 0
  *   sets color to black
  */
  public Figure( )
  {
    x = 0;
    y = 0;
    color = Color.BLACK;
  }

  /** overloaded constructor
  *   sets x to startX
  *   sets y to startY
  *   sets the color to startColor
  *   @param  startX      starting x pixel for figure
  *   @param  startY      starting y pixel for figure
  *   @param  startColor  figure color
  */
  public Figure( int startX, int startY, Color startColor )
  {
    x = startX;
    y = startY;
    color = startColor;
  }

  /** accessor method for color
  *   @return current figure color
  */
  public Color getColor( )
  {
    Color tempColor = color;
    return tempColor;
  }

  /** mutator method for color
  *   @param newColor  new color for figure
  */
  public void setColor( Color newColor )
  {
    color = newColor;
  }

  /** accessor method for x
```

```
     *    @return current x value
     */
    public int getX( )
    {
      return x;
    }

    /** mutator method for x
    *    @param newX  new value for x
    */
    public void setX( int newX )
    {
      x = newX;
    }

    /** accessor method for y
    *    @return current y value
    */
    public int getY( )
    {
      return y;
    }

    /** mutator method for y
    *    @param newY new y value
    */
    public void setY( int newY )
    {
      y = newY;
    }

    /** abstract draw method
    *    @param Graphics context for drawing figure
    */
    public abstract void draw( Graphics g );
}
```

EXAMPLE 10.15 The *abstract Figure* Class

When a subclass inherits from an *abstract* class, it can provide implementations for any, all, or none of the *abstract* methods. If the subclass does not completely implement all the *abstract* methods of the superclass, then the subclass must also be declared *abstract*. If, however, the subclass implements all the *abstract* methods in the superclass, and the subclass is not

COMMON ERROR TRAP

Do not include opening and closing curly braces in the definition of an *abstract* method. Including them would mean that the method is implemented, but does nothing. Instead, indicate an unimplemented method by using a semicolon.

COMMON ERROR TRAP

Attempting to instantiate an object of an *abstract* class will generate the following compiler error:

```
className is
abstract; cannot
be instantiated
```

where *className* is the name of the *abstract* class.

declared *abstract*, then the class is not *abstract* and we can instantiate objects of that subclass.

Example 10.16 shows the *Circle* class, which inherits from the *Figure* class and adds a *radius* instance variable. In the overloaded constructor, we pass the *startX*, *startY*, and *startColor* parameters to the constructor of the *Figure* class (line 34). On lines 54–63, the *Circle* class implements the *draw* method. We get the (*x*, *y*) coordinate and the color for the circle by calling the accessor methods of the *Figure* class because the *x*, *y*, and *color* instance variables are declared *private*.

```
 1 /* Circle class
 2 *  inherits from abstract Figure class
 3 *  Anderson, Franceschi
 4 */
 5
 6 import java.awt.Graphics;
 7 import java.awt.Color;
 8
 9 public class Circle extends Figure
10 {
11    private int radius;
12
13    /** default constructor
14    *   calls default constructor of Figure class
15    *   sets radius to 0
16    */
17    public Circle( )
18    {
19       super( );
20       radius = 0;
21    }
22
23    /** overloaded constructor
24    *   sends startX, startY, startColor to Figure constructor
25    *   sends startRadius to setRadius method
26    *   @param startX      starting x pixel
27    *   @param startY      starting y pixel
28    *   @param startColor  color for circle
29    *   @param startRadius radius of circle
30    */
31    public Circle( int startX, int startY, Color startColor,
32                   int startRadius )
33    {
34       super( startX, startY, startColor );
35       setRadius( startRadius );
36    }
37
38    /** mutator method for radius
39    *   @param newRadius  new value for radius
40    */
41    public void setRadius( int newRadius )
42    {
43       radius = newRadius;
44    }
45
46    /** accessor method for radius
47    *   @return radius
48    */
49    public int getRadius( )
50    {
51       return radius;
52    }
53
54    /** draw method
55    *  sets color and draws a circle
56    *  @param g  Graphics context for drawing the circle
57    */
58    public void draw( Graphics g )
59    {
60        g.setColor( getColor( ) );
61        g.fillOval ( getX( ), getY( ),
62                     radius * 2, radius * 2 );
63    }
64 }
```

EXAMPLE 10.16 The *Circle* Class

Similarly, Example 10.17 shows the *Square* class, which also inherits from the *Figure* class. The *Square* class adds a *length* instance variable and uses code similar to the *Circle* class to call the constructors of the *Figure* class (lines 19 and 34) and to implement its own version of the *draw* method (lines 54–63).

```
 1 /* Square class
 2 *  inherits from abstract Figure class
 3 *  Anderson, Franceschi
 4 */
```

```

import java.awt.Graphics;
import java.awt.Color;

public class Square extends Figure
{
   private int length;

   /** default constructor
   *   calls default constructor of Figure class
   *   sets length to 0
   */
   public Square( )
   {
     super( );
     length = 0;
   }

   /** overloaded constructor
   *   sends startX, startY, startColor to Figure constructor
   *   sets startLength to setLength method
   *   @param startX      starting x pixel
   *   @param startY      starting y pixel
   *   @param startColor  color for square
   *   @param startLength length of square
   */
   public Square( int startX, int startY, Color startColor,
                  int startLength )
   {
      super( startX, startY, startColor );
      setLength( startLength );
   }

   /** mutator method for length
   *   @param newLength  new value for length
   */
   public void setLength( int newLength )
   {
      length = newLength;
   }

   /** accessor method for length
   *   @return length
   */
   public int getLength( )
   {
      return length;
   }

   /** draw method
   *  sets color and draws a square
   *  @param g  Graphics context for drawing square
   */
   public void draw( Graphics g )
   {
      g.setColor( getColor( ) );
      g.fillRect( getX( ), getY( ),
                  length, length );
   }
}
```

EXAMPLE 10.17 The *Square* Class

REFERENCE POINT

For more information on *ArrayLists*, see Chapter 9.

Because we want to instantiate *Circle* and *Square* objects, we do not declare these classes *abstract* and they are forced to implement the *draw* method. Example 10.18 shows a client applet, *TrafficLight*, which paints a traffic light, shown in Figure 10.13. On lines 10 and 11, we declare two *ArrayLists*, one to hold *Circle* objects and one to hold *Square* objects. In the *init* method (lines 13–24), we instantiate both *ArrayLists* and add three *Square* objects to *squaresList* and three *Circle* objects to *circlesList*. Then in the *paint* method, we create the traffic light by calling the *draw* methods for all

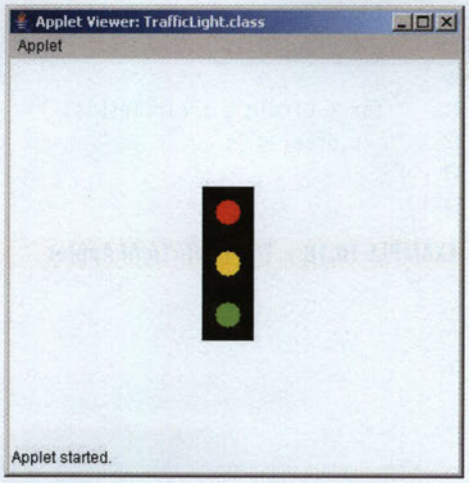


Figure 10.13
The *TrafficLight* Applet

the *Squares* in the *squaresList* (lines 28–29), then calling the *draw* method for all the *Circles* in the *circlesList* (lines 31–32).

```
 1 /* Figure Hierarchy Client
 2    Anderson, Franceschi
 3 */
 4 import javax.swing.JApplet;
 5 import java.awt.*;
 6 import java.util.ArrayList;
 7
 8 public class TrafficLight extends JApplet
 9 {
10   private ArrayList<Circle> circlesList;
11   private ArrayList<Square> squaresList;
12
13   public void init( )
14   {
15     squaresList = new ArrayList<Square>( );
16     squaresList.add( new Square( 150, 100, Color.BLACK, 40 ) );
17     squaresList.add( new Square( 150, 140, Color.BLACK, 40 ) );
18     squaresList.add( new Square( 150, 180, Color.BLACK, 40 ) );
19
20     circlesList = new ArrayList<Circle>( );
21     circlesList.add( new Circle( 160, 110, Color.RED, 10 ) );
22     circlesList.add( new Circle( 160, 150, Color.YELLOW, 10 ) );
23     circlesList.add( new Circle( 160, 190, Color.GREEN, 10 ) );
24   }
25
26   public void paint( Graphics g )
27   {
28     for ( Square s : squaresList )
29       s.draw( g );
30
31     for ( Circle c : circlesList )
32       c.draw( g );
33   }
34 }
```

EXAMPLE 10.18 The *TrafficLight* Applet

TABLE 10.6 Restrictions for Defining *abstract* Classes and Methods

abstract classes	• Classes must be declared *abstract* if the class contains any *abstract* methods.
	• *abstract* classes can be extended.
	• *abstract* classes cannot be used to instantiate objects.
abstract methods	• *abstract* methods can be declared only within an *abstract* class.
	• An *abstract* method must consist of a method header followed by a semicolon.
	• *abstract* methods cannot be called.
	• *abstract* methods cannot be declared as *private* or *static*.
	• A constructor cannot be declared *abstract*.

Java imposes a few restrictions on declaring and using *abstract* classes and methods. These rules are summarized in Table 10.6.

10.6 Polymorphism

An important concept in inheritance is that an object of a class is also an object of any of its superclasses. That concept is the basis for an important OOP feature, called **polymorphism**, which simplifies the processing of various objects in the same class hierarchy. The word *polymorphism,* which is derived from the word fragment *poly* and the word *morpho* in the Greek language, literally means "multiple forms."

Polymorphism allows us to use the same method call for any object in the hierarchy. We make the method call using an object reference of the superclass. At run time, the JVM determines to which class in the hierarchy the object actually belongs and calls the version of the method implemented for that class.

To use polymorphism in your application, the following conditions must be true:

- The classes are in the same hierarchy.
- The subclasses override the same method.

- A subclass object reference is assigned to a superclass object reference (that is, a subclass object is referenced by a superclass reference).
- The superclass object reference is used to call the method.

For example, we can take advantage of polymorphism in our traffic light applet by calling the *draw* method for either a *Circle* or *Square* object using a *Figure* object reference. Although we cannot instantiate an object from an *abstract* class, Java allows us to define object references of an *abstract* class.

Example 10.19 shows the rewritten traffic light applet. Instead of using separate *ArrayLists* for *Circle* and *Square* objects, we can declare and instantiate only one *ArrayList* of *Figure* references (lines 10 and 14). As each *Circle* and *Square* object is instantiated, we add its object reference to the *ArrayList* of *Figure* references (lines 15–22).

This greatly simplifies the *paint* method (lines 25–29), which steps through *figuresList*, calling the *draw* method for each element. For the method call, it doesn't matter whether the object reference in *figuresList* is a *Circle* or *Square* reference. We just call the *draw* method using that reference. At run time, the JVM determines whether the object is a *Circle* or a *Square* and calls the appropriate *draw* method for the object type. Notice that we have interwoven the adding of *Circles* and *Squares* to *figuresList*, rather than adding all *Squares*, then adding all *Circles*. Because the *ArrayList* is composed of *Figure* references, any element can be either a *Circle* or a *Square*—because a *Circle* and a *Square* are both *Figures*. The output of this applet is identical to that of Example 10.18, as shown in Figure 10.13.

```
 1 /* Figure hierarchy Client
 2    Anderson, Franceschi
 3 */
 4 import javax.swing.JApplet;
 5 import java.awt.*;
 6 import java.util.ArrayList;
 7
 8 public class TrafficLightPolymorphism extends JApplet
 9 {
10   private ArrayList<Figure> figuresList;
11
12   public void init( )
13   {
14     figuresList = new ArrayList<Figure>( );
15     figuresList.add( new Square( 150, 100, Color.BLACK, 40 ) );
```

```
    figuresList.add( new Circle( 160, 110, Color.RED, 10 ) );

    figuresList.add( new Square( 150, 140, Color.BLACK, 40 ) );
    figuresList.add( new Circle( 160, 150, Color.YELLOW, 10 ) );

    figuresList.add( new Square( 150, 180, Color.BLACK, 40 ) );
    figuresList.add( new Circle( 160, 190, Color.GREEN, 10 ) );
  }

  public void paint( Graphics g )
  {
    for ( Figure f : figuresList )
         f.draw( g );
  }
}
```

EXAMPLE 10.19 Traffic Light Using Polymorphism

Skill Practice
with these end-of-chapter questions

10.10.1 Multiple Choice Exercises

Questions 7,10,11

10.10.4 Identifying Errors in Code

Question 31

10.10.5 Debugging Area

Questions 33,34,35

10.7 Programming Activity 2: Using Polymorphism

In this Programming Activity, you will complete the implementation of the Tortoise and the Hare race. The Tortoise runs a slow and steady race, while the Hare runs in spurts with rests in between. Figure 10.14 shows a sample run of the race. In this figure, we show only one tortoise and one hare; however, using polymorphism we can easily run the race with any number and combination of tortoises and hares.

The class hierarchy for this Programming Activity is shown in Figure 10.15.

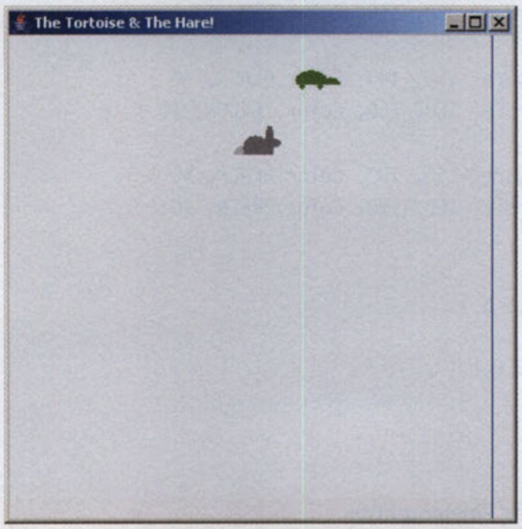


Figure 10.14
A Sample Run of the Tortoise and the Hare Race

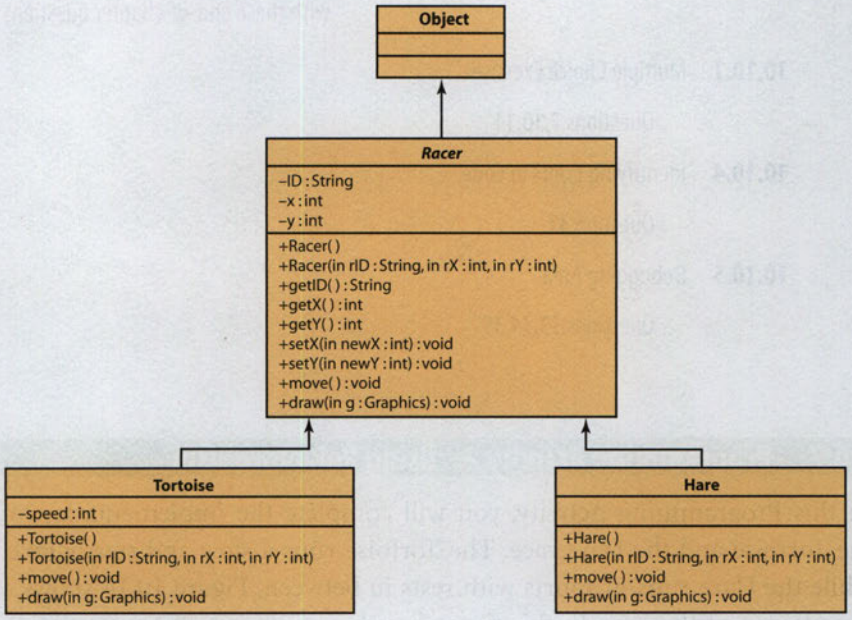

Object
Racer
-ID : String
-x : int
-y : int
+Racer( )
+Racer(in rID : String, in rX : int, in rY : int)
+getID( ) : String
+getX( ) : int
+getY( ) : int
+setX(in newX : int) : void
+setY(in newY : int) : void
+move( ) : void
+draw(in g : Graphics) : void
Tortoise
-speed : int
+Tortoise( )
+Tortoise(in rID : String, in rX : int, in rY : int)
+move( ) : void
+draw(in g: Graphics) : void
Hare
+Hare( )
+Hare(in rID : String, in rX : int, in rY : int)
+move( ) : void
+draw(in g: Graphics) : void

Figure 10.15
***Racer* Hierarchy**

The code for the *Racer* class, which is the superclass of the *Tortoise* and *Hare* classes, is shown in Example 10.20. The *Racer* class has three instance variables (lines 10–12): a *String ID*, which identifies the type of racer; and *x* and *y* positions, both of which are *ints*. The class has the usual constructors, as well as accessor and mutator methods for the *x* and *y* positions and ID. These instance variables and methods are common to all racers, so we put them in the *Racer* class. Individual racers, however, will differ in the way they move and in the way they are drawn. Thus, in line 8, we declare the *Racer* class to be *abstract*, and in lines 74–81, we define two *abstract* methods, *move* and *draw*. Classes that inherit from the *Racer* class will need to provide implementations of these two methods (or be declared *abstract* as well).

```
 1 /**  Racer class
 2 *    Abstract class intended for racer hierarchy
 3 *    Anderson, Franceschi
 4 */
 5
 6 import java.awt.Graphics;
 7
 8 public abstract class Racer
 9 {
10   private String ID;  // racer ID
11   private int x;      // x position
12   private int y;      // y position
13
14   /** default constructor
15   *    Sets ID to blank
16   */
17   public Racer( )
18   {
19     ID = "";
20   }
21
22   /** Constructor
23   *    @param rID   racer ID
24   *    @param rX    x position
25   *    @param rY    y position
```

```
*/
public Racer( String rID, int rX, int rY )
{
  ID = rID;
  x = rX;
  y = rY;
}

/** accessor for ID
*   @return  ID
*/
public String getID( )
{
  return ID;
}

/** accessor for x
*   @return  current x value
*/
public int getX( )
{
  return x;
}

/** accessor for y
*   @return  current y value
*/
public int getY( )
{
  return y;
}

/** mutator for x
*   @param  newX   new value for x
*/
public void setX( int newX )
{
  x = newX;
}

/** mutator for y
*   @param  newY   new value for y
*/
```

```
public void setY( int newY )
{
  y = newY;
}

/** abstract method for Racer's move
*/
public abstract void move( );

/** abstract method for drawing Racer
*  @param  g   Graphics context
*/
public abstract void draw( Graphics g );
}
```

EXAMPLE 10.20 The *abstract Racer* Class

The *Tortoise* and *Hare* classes inherit from the *Racer* class. Thus, their only job is to pass constructor arguments to the *Racer* class and implement the *draw* and *move* methods. For this Programming Activity, we have provided the *Tortoise* and *Hare* classes with the *draw* and *move* methods already written.

Your job is to add *Tortoise* and *Hare* objects to an *ArrayList* of *Racer* objects, as specified by the user. Then you will add code to run the race by stepping through the *ArrayList*, calling *move* and *draw* for each *Racer* object.

Instructions

Copy the source files in the Programming Activity 2 directory for this chapter to a directory on your computer.

1. Write the code to determine which racers will run the race. Load the *RacePoly.java* source file and search for five asterisks in a row (*****). This will position you inside the *prepareToRace* method.

   The dialog box to prompt the user for the racer type is already coded for you in the *getRacer* method. When your *switch* statement executes, the *getRacer* method has already been called and the *char* variable *input* contains the user's input. You do not need to call the *getRacer* method.

```
/** prepareToRace method
*   uses a dialog box to prompt user for racer types
*     and to start the race
*   racer types are 't' or 'T' for Tortoise,
*                   'h' or 'H' for Hare
*   's' or 'S' will start the race
*/
private void prepareToRace( )
{
   int yPos = FIRST_RACER;          // y position of first racer
   final int START_LINE = 40;       // x position of start of race
   final int RACER_SPACE = 50;      // spacing between racers
   char input;

   input = getRacer( ); // get input from user

   while ( input != 's' && input != 'S' )
   {
      /** 1. ***** Student writes this switch statement
      * input local char variable contains the racer type
      *     entered by the user
      * If input is 'T' or 't',
      *     add a Tortoise object to the ArrayList named racerList
      *            which is an instance variable of this class
      * The API of the Tortoise constructor is:
      *          Tortoise( String ID, int startX, int startY )
      *   a sample call to the constructor is
      *          new Tortoise( "Tortoise", START_LINE, yPos )
      *          where START_LINE is a constant local variable
      *            representing the starting x position for the race
      *          and yPos is a local variable representing
      *             the next racer's y position
      *
      *  If input is 'H' or 'h',
      *      add a Hare object to the ArrayList named racerList
      *  The API of the Hare constructor is:
      *          Hare( String ID, int startX, int startY )
      *      a sample call to the constructor is
      *          new Hare( "Hare", START_LINE, yPos )
      *          where START_LINE is a constant local variable
      *            representing the starting x position for the race
      *          and yPos is a local variable representing
      *             the next racer's y position
      *
      *  After adding a racer to the ArrayList racerList,
      *         increment yPos by the value of
      *         the constant local variable RACER_SPACE
      *
      *  if input is anything other than 'T', 't',
      *         'H' or 'h', pop up an error dialog box
      *         a sample method call for the output dialog box is:
      *           JOptionPane.showMessageDialog( this, "Message" );
      */
      // write your switch statement here

       /** end of student code, part 1 */

       repaint( );
       input = getRacer( );  // get input from user

     } // end while

   } // end prepareToRace
```

2. Next, write the code to run the race. In the *RacePoly.java* source file, search again for five asterisks in a row (*****). This will position you inside the *paint* method, where you will perform tasks 2 and 3. For task 2, you will write code to loop through the *ArrayList* of *Racers* and call the *move* and *draw* methods for each racer as they run the race. When you finish that task, search again for five asterisks in a row (*****), which will position you at the location of task 3. This task is similar to task 2, in that you need to loop through the *ArrayList* of *Racers.* However, in this task you will only call the *draw* method for each *Racer.* This code will be executed before the race begins as racers are added to the start line.

   The portion of the *paint* method where you will add your code is shown here.

```
/** paint method
*    @param g   Graphics context
*    draws the finish line;
*    moves and draws racers
*/
public void paint( Graphics g )
{
  super.paint( g );
```

```
      // draw the finish line
      finishX = getWidth( ) - 20;
      g.setColor( Color.blue );
      g.drawLine( finishX, 0, finishX, getHeight( ) );

      if ( raceIsOn )
      {
         /* 2. ***** student writes this code
         *  loop through instance variable ArrayList racerList,
         *    which contains Racer object references,
         *    calling move, then draw for each racer
         *  The API for move is:
         *      void move( )
         *  The API for draw is:
         *      void draw( Graphics g )
         *        where g is the Graphics context
         *        passed to the paint method
         */

         /** end of student code, part 2 */
       }
       else  // display racers before race begins
       {
         /* 3. ***** student writes this code
         *  loop through instance variable ArrayList racerList,
         *    which contains Racer object references,
         *    calling draw for each element. (Do not call move!)
         *  The API for draw is:
         *      void draw( Graphics g )
         *        where g is the Graphics context
         *        passed to this paint method
         */
         // student code goes here

         /** end of student code, part 3 */
      }
   }
```

When you have finished writing the code, compile the source code and run the *RacePoly* application. Try several runs of the race with a different number of racers and with a different combination of *Tortoises* and *Hares.* Figure 10.16 shows the race with four *Tortoises* and three *Hares.*

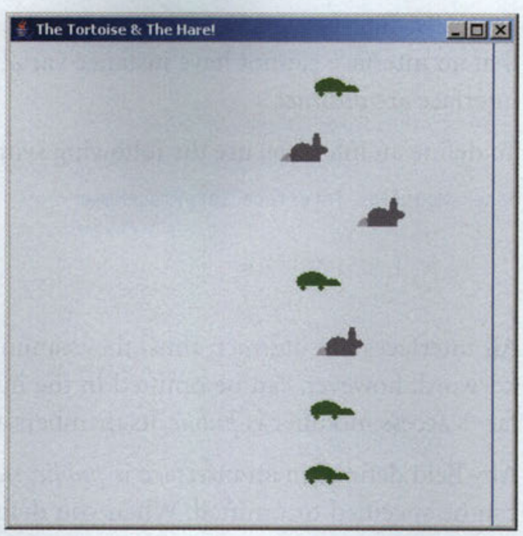


Figure 10.16
Another Run of the Race

? DISCUSSION QUESTIONS

1. Explain how polymorphism simplifies this application.
2. If you wanted to add another racer, for example, an aardvark, explain what code you would need to write and what existing code, if any, you would need to change.

10.8 Interfaces

In Java, a class can inherit directly from only one class; that is, a class can *extend* only one class. To allow a class to inherit behavior from multiple sources, Java provides the **interface**.

An interface typically specifies behavior that a class will *implement*. Interface members can be any of the following:

- classes
- constants
- *abstract* methods
- other interfaces

Typically, interfaces define only constants and *abstract* methods. Notice that an interface cannot have instance variables and that all methods in an interface are *abstract.*

To define an interface, use the following syntax:

```
accessModifier interface InterfaceName
{
   // body of interface
}
```

All interfaces are *abstract*; thus, they cannot be instantiated. The *abstract* keyword, however, can be omitted in the interface definition. If the interface's access modifier is *public,* its members are implicitly *public* as well.

Any field defined in an interface is *public, static,* and *final.* These keywords can be specified or omitted. When you define a field in the interface, you must also assign a value to the field at that time.

All methods within an interface must be *abstract,* so the method definition must consist of only a method header and a semicolon. Like the interface header, the *abstract* keyword can be omitted from the method definition.

To inherit from an interface, a class declares that it *implements* the interface in the class definition, using the following syntax:

```
accessModifier class ClassName extends SuperclassName
                                  implements Interface1, Interface2, ...
```

The *extends* clause is optional if the class inherits only from the *Object* class. A class can *implement* 0, 1, or more interfaces. If a class *implements* more than one interface, the interfaces are specified in a comma-separated list of interface names. When a class *implements* an interface, the class must provide an implementation for each method contained in the interface.

In the Programming Activity of the last section, we ran the Tortoise and the Hare race. Each racer inherited from the *abstract Racer* class, which specified two *abstract* methods: *draw* and *move.* Thus, both the *Tortoise* and *Hare* classes implemented those two methods.

Below are brief, summarized versions of the *Racer* and *Tortoise* classes from Programming Activity 2:

```
// Racer defines move and draw as abstract methods
public abstract class Racer
{
  public abstract void move( );
  public abstract void draw( Graphics g );

  // other fields and methods
}

// Tortoise inherits from Racer and implements move and draw
public class Tortoise extends Racer
{
  public void move( )
  {
     // implementation here
  }

  public void draw( Graphics g )
  {
     // implementation here
  }

  // other fields and methods
}
```

That class hierarchy worked well for running the race, but suppose we just wanted to draw a tortoise or a hare, or another animal, without racing them. In other words, we don't want to implement the *move* method, just the *draw* method.

In this case, we can define an *abstract* class, *Animal*, which contains only one *abstract* method: *draw*. To provide the optional capability of running a race, we define an interface, named *Moveable*, which includes the *abstract* method *move*. Then, if we just want to draw an animal, we define a class that *extends* the *Animal* class. If we want to make our animal a racer, we implement the *Moveable* interface.

Following are brief, summarized versions of the *Animal* class; the *Moveable* interface; a racing animal class, which we call *TortoiseRacer*; and a *TortoiseNonRacer* class that can be used in another application where animal objects are drawn, but do not race.

```
// Animal defines only one abstract method: draw
public abstract class Animal
{
  public abstract void draw( Graphics g );
  // other fields and methods
}

// Moveable defines move as an abstract method
public interface Moveable
{
  // some constants defined here
  public void move( );

}

// TortoiseRacer inherits from Animal and Moveable
//  and implements both move and draw
public class TortoiseRacer extends Animal implements Moveable
{
  public void draw( Graphics g )
  {
     // implementation here
  }

  public void move( )
  {
     // implementation here
  }

  // other fields and methods here
}

// non-racing Tortoise inherits only from Animal
// and implements draw method only
public class TortoiseNonRacer extends Animal
{
  public void draw( Graphics g )
  {
     // implementation here
  }

  // other fields and methods here
}
```

Example 10.21 shows the actual *Animal* class, which is similar to the *Racer* class shown in Example 10.20, except that its only *abstract* method is *draw* (lines 73–76).

```java
/**  Animal class
*    Anderson, Franceschi
*/

import java.awt.Graphics;

public abstract class Animal
{
  private int x;       // x position
  private int y;       // y position
  private String ID;   // animal ID

  /** default constructor
  *    Sets ID to empty String
  */
  public Animal( )
  {
    ID = "";
  }

  /** Constructor
  *    @param rID   Animal ID
  *    @param rX    x position
  *    @param rY    y position
  */
  public Animal( String rID, int rX, int rY )
  {
    ID = rID;
    x = rX;
    y = rY;
  }

  /** accessor for ID
  *   @return  ID
  */
  public String getID( )
  {
    return ID;
  }

  /** accessor for x
  *   @return  x coordinate
  */
  public int getX( )
  {
    return x;
  }

  /** accessor for y
  *   @return  y coordinate
  */
  public int getY( )
  {
    return y;
  }

  /** mutator for x
  *   @param  newX  new value for x position
  */
  public void setX( int newX )
  {
    x = newX;
  }

  /** mutator for y
  *   @param  newY  new value for y position
  */
  public void setY( int newY )
  {
    y = newY;
  }

  /** abstract method for drawing Animal
  *   @param   g    Graphics context
  */
  public abstract void draw( Graphics g );
}
```

EXAMPLE 10.21 The *Animal* Class

Example 10.22 shows the actual *Moveable* interface, which specifies the API for the *move* method (line 10). Although all methods in an interface are *abstract*, we do not need to specify the *abstract* keyword. Also, all members of a *public* interface are *public*, so we do not need to specify an access modifier either.

In lines 7–8, we define two constants, which will be used by the racer in the *move* method to determine how fast the racer will move. A hare would increment its *x* value by the *FAST* value (5), and the tortoise would increment its *x* value by the *SLOW* value (1).

```
/** Moveable interface
*   Anderson, Franceschi
*/

public interface Moveable
{
  int FAST = 5; // static constant
  int SLOW = 1; // static constant

  void move( ); // abstract method
}
```

EXAMPLE 10.22 The *Moveable* Interface

Example 10.23 shows the actual *TortoiseRacer* class, which is functionally equivalent to the *Tortoise* class in the Programming Activity. It inherits from the *Animal* class and *implements* the *Moveable* interface (line 10). It provides bodies for both the *draw* method (lines 29–53), inherited from *Animal*, and the *move* method (lines 55–63), inherited through the *Moveable* interface. Because implementing the *Moveable* interface is optional, we can also define classes that inherit from *Animal*, but do not need to provide a *move* method. In this way, we have designed for more optimal reuse of our classes.

```
 1 /**  TortoiseRacer class
 2 *    inherits from abstract Animal class
 3 *    implements Moveable interface
 4 *    Anderson, Franceschi
 5 */
 6
 7 import java.awt.Graphics;
 8 import java.awt.Color;
 9
10 public class TortoiseRacer extends Animal implements Moveable
11 {
12    /** Default Constructor: calls Animal default constructor
13    */
14    public TortoiseRacer( )
15    {
    super( );
  }

  /** Constructor
  *    @param rID  racer Id, passed to Animal constructor
  *    @param rX   x position, passed to Animal constructor
  *    @param rY   y position, passed to Animal constructor
  */
  public TortoiseRacer( String rID, int rX, int rY )
  {
    super( rID, rX, rY );
  }

  /** draw: draws the Tortoise at current (x, y) coordinate
  *       implements abstract method in Animal class
  *       @param g   Graphics context
  */
  public void draw( Graphics g )
  {
    int startX = getX( );
    int startY = getY( );

    g.setColor( new Color( 34, 139, 34 ) ); // dark green

    //body
    g.fillOval( startX, startY, 25, 15 );

    //head
    g.fillOval( startX + 20, startY + 5,  15, 10 );

    //flatten bottom
    g.clearRect( startX, startY + 11, 35, 4 );

    //feet
    g.setColor( new Color( 34, 139, 34 ) );  // brown
    g.fillOval( startX + 3, startY + 10,  5, 5 );
    g.fillOval( startX + 17, startY + 10, 5, 5 );
  }

   /** implements move method in Moveable interface
   *   move:  calculates the new x value for the racer
   *   Tortoise move characteristics: "slow & steady wins the race"
   *       increment x by SLOW (inherited from Moveable interface)
   */
```

```
    public void move( )
    {
        setX( getX( ) + SLOW );
    }
}
```

EXAMPLE 10.23 The *TortoiseRacer* Class

Example 10.24 shows a client applet that exercises the *TortoiseRacer* class. In this race, the tortoise, the only racer, always wins, as shown in Figure 10.17.

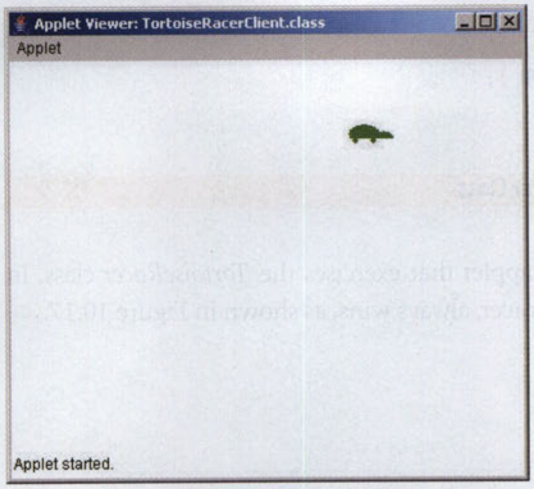
REFERENCE POINT

Interfaces are used extensively in GUI applications, as you will see in Chapter 12.

```
/** TortoiseRacer Client
*   Anderson, Franceschi
*/

import javax.swing.*;
import java.awt.*;

public class TortoiseRacerClient extends JApplet
{
  private TortoiseRacer t;

  public void init( )
  {
    t = new TortoiseRacer( "Tortoise", 50, 50 );
  }

  public void paint( Graphics g )
  {
    for ( int i = 0; i < getWidth( ); i++ )
    {
      t.move( );
      t.draw( g );

      Pause.wait( .03 );
      g.clearRect( 0, 0, getWidth( ), getHeight( ) );
    }
  }
}
```

EXAMPLE 10.24 A Client Applet for the *TortoiseRacer* Class

Figure 10.17
Output from Example 10.24



Skill Practice
with these end-of-chapter questions

10.10.1 Multiple Choice Exercises

Questions 3,5,6

10.10.3 Fill In the Code

Question 25

10.10.4 Identifying Errors in Code

Questions 26,27,28,29,30

10.10.8 Technical Writing

Question 57

CODE IN ACTION

On the CD-ROM included with this book, you will find a Flash movie with a step-by-step illustration of the use of *abstract* classes, polymorphism, and interfaces in a program. Click on the link for Chapter 10 to start the movie.

10.9 Chapter Summary

- Inheritance lets us organize related classes into ordered levels of functionality, called hierarchies. The advantage is that we write the common code only once and reuse it in multiple classes.
- A subclass inherits methods and fields of its superclass. A subclass can have only one direct superclass, but many subclasses can inherit from a common superclass.
- Inheritance implements the "is a" relationship between classes. Any object of a subclass is also an object of the superclass.
- All classes inherit from the *Object* class.
- To specify that a subclass inherits from a superclass, the subclass uses the *extends* keyword in the class definition, as in the following syntax:

  ```
  accessModifier class ClassName extends SuperclassName
  ```

- A subclass does not inherit constructors or *private* members of the superclass. However, the superclass constructors are still available to be called from the subclass, and the *private* fields of the superclass are implemented as fields of the subclass.
- To access private fields of the superclass, the subclass needs to use the accessor and mutator methods provided by the superclass.
- To call the constructor of the superclass, the subclass constructor uses the following syntax:

  ```
  super( argument list );
  ```

  If used, this statement must be the first statement in the subclass constructor.

- A subclass can override an inherited method by providing a new version of the method. The new method's API must be identical to the inherited method. To call the inherited version of the method, the subclass uses the *super* object reference using the following syntax:

  ```
  super.methodName( argument list )
  ```

- Any field declared using the *protected* access modifier is inherited by the subclass. As such, the subclass can directly access the field without calling its accessor or mutator method.

- An *abstract* class can be used to specify APIs for methods that subclasses should implement. An *abstract* class cannot be used to instantiate objects. A class is declared to be *abstract* by including the *abstract* keyword in the class header.

- An *abstract* class typically has one or more *abstract* methods. An *abstract* method specifies the API of the method, but does not provide an implementation. The API of an *abstract* method is followed by a semicolon.

- When a subclass inherits from an *abstract* class, it can provide implementations for any, all, or none of the *abstract* methods. If the subclass does not implement all the *abstract* methods of the superclass, then the subclass must also be declared as *abstract*. If, however, the subclass implements all the *abstract* methods in the superclass and is not declared *abstract*, then the class is not *abstract* and we can instantiate objects of that subclass.

- Polymorphism simplifies the processing of various objects in a hierarchy by allowing us to use the same method call for any object in the hierarchy. We assign an object reference of a subclass to a superclass reference, then make the method call using the superclass object reference. At run time, the JVM determines to which class in the hierarchy the object actually belongs and calls the appropriate form of the method for that class.

- Interfaces allow a class to inherit behavior from multiple sources. Interface members can be classes, constants, *abstract* methods, or other interfaces.

- To define an interface, use the following syntax:

```
accessModifier interface InterfaceName
{
   // body of interface
}
```

- To use an interface, a class header should include the *implements* keyword and the name of the interface, as in the following syntax:

```
accessModifier class ClassName implements InterfaceName
```

- To specify that a subclass inherits from a superclass and uses an interface, a class header should include the *implements* keyword and the name of the interface, as in the syntax that follows.

```
accessModifier class ClassName extends SuperclassName
                                  implements InterfaceName
```

10.10 Exercises, Problems, and Projects

10.10.1 Multiple Choice Exercises

1. The *extends* keyword applies to
   - ❑ a class inheriting from another class.
   - ❑ a variable.
   - ❑ a method.
   - ❑ an expression.
2. A Java class can inherit from two or more classes.
   - ❑ true
   - ❑ false
3. In Java, multiple inheritance is implemented using the concept of
   - ❑ an interface.
   - ❑ an *abstract* class.
   - ❑ a *private* class.
4. Which of the following is inherited by a subclass?
   - ❑ all instance variables and methods
   - ❑ *public* instance variables and methods only
   - ❑ *protected* instance variables and methods only
   - ❑ *protected* and *public* instance variables and methods
5. What Java keyword is used in a class header when a class is defined as inheriting from an interface?
   - ❑ *inherits*
   - ❑ *includes*

- ❑ *extends*
- ❑ *implements*

6. A Java class can implement one or more interfaces.
   - ❑ true
   - ❑ false
7. How do you instantiate an object from an *abstract* class?
   - ❑ With any constructor.
   - ❑ With the default constructor only.
   - ❑ You cannot instantiate an object from an *abstract* class.
8. When a class overrides a method, what Java keyword is used to call the method inherited from the superclass?
   - ❑ *inherited*
   - ❑ *super*
   - ❑ *class*
   - ❑ *methodName*
9. Where should the following statement be located in the body of a subclass constructor?

```
super( );
```

   - ❑ It should be the last statement.
   - ❑ It should be the first statement.
   - ❑ It can be anywhere.
10. If a class contains an *abstract* method, then
    - ❑ the class must be declared *abstract*.
    - ❑ the class is not *abstract*.
    - ❑ the class may or may not be *abstract*.
    - ❑ all of the above
11. What can you tell about the following method?

```
public void myMethod( )
{
}
```

- ❑ This method is *abstract.*
- ❑ This method is not *abstract.*

10.10.2 Reading and Understanding Code

For Questions 12 to 20, consider the following three classes:

```
public class A
{
 private int number;
 protected String name;
 public double price;

 public A( )
 {
   System.out.println( "A( ) called" );
 }

 private void foo1( )
 {
  System.out.println( "A version of foo1( ) called" );
 }

 protected int foo2( )
 {
  System.out.println( "A version of foo2( ) called" );
  return number;
 }

 public String foo3( )
 {
  System.out.println( "A version of foo3( ) called" );
  return "Hi";
 }
}

public class B extends A
{
 private char service;

 public B( )
 {
  super( );
  System.out.println( "B( ) called" );
 }
```

```
   public void foo1( )
   {
    System.out.println( "B version of foo1( ) called" );
   }

   protected int foo2( )
   {
    int n = super.foo2( );
    System.out.println( "B version of foo2( ) called" );
    return ( n + 5 );
   }

   public String foo3( )
   {
    String temp = super.foo3( );
    System.out.println( "B version of foo3( )" );
    return ( temp + " foo3" );
   }
 }

 public class C extends B
 {
   public C( )
   {
    super( );
    System.out.println( "C( ) called" );
   }

   public void foo1( )
   {
    System.out.println( "C version of foo1( ) called" );
   }
 }
```

12. Draw the UML diagram for the class hierarchy.
13. What fields and methods are inherited by which class?
14. What fields and methods are not inherited?
15. What is the output of the following code sequence?

```
B b1 = new B( );
```

16. What is the output of the following code sequence?

```
B b2 = new B( );
b2.foo1( );
```

17. What is the output of the following code sequence?

```
B b3 = new B( );
int n = b3.foo2( );
```

18. What is the output of the following code sequence?

```
// b4 is a B object reference
System.out.println( b4.foo3( ) );
```

19. What is the output of the following code sequence?

```
C c1 = new C( );
```

20. What is the output of the following code sequence?

```
// c2 is a C object reference
c2.foo1( );
```

10.10.3 Fill In the Code

For Questions 21 to 25, consider the following class *F* and the interface *I*:

```
public class F
{
 private String first;
 protected String name;

 public F( )
 { }

 public F( String f, String n )
 {
  first = f;
  name = n;
 }
 public String getFirst( )
 {
  return first;
 }
 public String getName( )
 {
  return name;
 }
 public String toString( )
 {
   return ( "first: " + first + "\tname: " + name );
 }
 public boolean equals( Object f )
 {
  if ( ! ( f instanceof F ) )
```

```
        return false;
      else
      {
        F objF = ( F ) f;
        return( first.equals( objF.first ) && name.equals( objF.name ) );
      }
    }

  public interface I
  {
   public static final String TYPE = "human";
   public abstract int age( );
  }
```

21. The *G* class inherits from the *F* class. Code the class header of the *G* class.

```
    // your code goes here
```

22. Inside the *G* class, which inherits from the *F* class, declare a *private* instance variable for the middle initial and code a constructor with three parameters, calling the constructor of the *F* class and assigning the third parameter, a *char*, to the new instance variable.

```
    // your code goes here
```

23. Inside the *G* class, which inherits from the *F* class, code the *toString* method, which returns a printable representation of a *G* object reference.

```
    // your code goes here
```

24. Inside the *G* class, which inherits from the *F* class, code the *equals* method, which compares two *G* objects and returns *true* if they have identical instance variables; *false* otherwise.

```
    // your code goes here
```

25. The *K* class inherits from the *F* class and the *I* interface; code the class header of the *K* class.

```
    // your code goes here
```

10.10.4 Identifying Errors in Code

For Questions 26 to 31, consider the following two classes, *C* and *D*, and interface *I*:

```
  public abstract class C
  {
   private void foo1( )
   {
    System.out.println( "Hello foo1" );
  }
  public abstract void foo2( );
  public abstract int foo3( );
 }

 public class D extends C
 {
  public void foo2( )
  {
   System.out.println( "Hello foo2( )" );
  }
  public int foo3( )
  {
   return 10;
  }
  private void foo4( )
  {
   System.out.println( "Hello D foo4( )" );
  }
 }

 public interface I
 {
  public static final double PI = 3.14;
 }
```

26. Where is the error in this code sequence?

```
C c1 = new C( );
```

27. Where is the error in this code sequence?

```
D d1 = new D( );
d1.foo1( );
```

28. Is there an error in this code sequence? Why or why not?

```
C c2;
c2 = new D( );
```

29. Where is the error in this new class?

```
public class E extends D
{
 public void foo4( )
 {
  super.foo4( );
  System.out.println( "Hello E foo4()" );
 }
}
```

30. Where is the error in this class?

```
public class J extends I
{
}
```

31. Where is the error in this class?

```
public class K
{
 public void foo( );
}
```

10.10.5 Debugging Area—Using Messages from the Java Compiler and Java JVM

32. You coded the following class:

```
public class N extends String, Integer
{
}
```

When you compile, you get the following message:

```
N.java:1: '{' expected
public class N extends String, Integer
                             ^
1 error
```

Explain what the problem is and how to fix it.

For Exercises 33 to 35, consider the following class:

```
public abstract class M
{
 private int n;
 protected double p;
 public abstract void foo1( );
}
```

33. You coded the following class:

```
public class P extends M
{
}
```

When you compile, you get the following message:

```
P.java:1: P is not abstract and does not override abstract method
foo1() in M
public class P extends M
       ^
1 error
```

34. You coded the following class:

```
public class P extends M
{
 public void foo1( )
 {
  System.out.println( "n is: " + n );
 }
}
```

When you compile, you get the following message:

```
P.java:5: n has private access in M
  System.out.println( "n is: " + n );
                                 ^
1 error
```

35. You coded the following classes:

```
public class P extends M
{
 public P( double newP )
 {
  p = newP;
 }
 public void foo1( )
 {
 }
}

public class Q extends P
{
 private int z;
 public Q( double newP, int newZ )
 {
  z = newZ;
  super( newP );  // line 7
 }
}
```

When you compile, you get the following message:

```
Q.java:5: constructor P in class P cannot be applied to given types
  {
  ^
  required: double
  found: no arguments
Q.java:7: call to super must be first statement in constructor
```

```
     super( newP );
         ^
   2 errors
```

10.10.6 Write a Short Program

36. Write an applet overriding the *init* and *paint* methods of the *JApplet* class with a simple output line to the screen, which will show in what order each method is called.

For Exercises 37 to 41, consider the following class:

```
public class Game
{
 private String description;

 public Game( String newDescription )
 {
  setDescription( newDescription );
 }

 public String getDescription( )
 {
  return description;
 }

 public void setDescription( String newDescription )
 {
  description = newDescription;
 }

 public String toString( )
 {
  return ( "description: " + description );
 }
}
```

37. Write a class encapsulating a PC-based game, which inherits from *Game*. A PC-based game has the following additional attributes: the minimum megabytes of RAM needed to play the game, the number of megabytes needed on the hard drive to install the game, and the minimum GHz performance of the CPU. Code the constructor and the *toString* method of the new class. You also need to include a client class to test your code.

38. Write a class encapsulating a board game, which inherits from *Game.* A board game has the following additional attributes: the number of players and whether the game can end in a tie. Code the constructor and the *toString* method of the new class. You also need to include a client class to test your code.
39. Write a class encapsulating a sports game, which inherits from *Game.* A sports game has the following additional attributes: whether the game is a team or individual game, and whether the game can end in a tie. Code the constructor and the *toString* method of the new class. You also need to include a client class to test your code.
40. Write a class encapsulating a trivia game, which inherits from *Game.* A trivia game has the following additional attributes: the ultimate money prize and the number of questions that must be answered to win the ultimate money. Code the constructor and the *toString* method of the new class. You also need to include a client class to test your code.
41. Write a class encapsulating a board game, which inherits from *Game.* A board game has the following additional attributes: the minimum number of players, the maximum number of players, and whether there is a time limit to finish the game. Code the constructor and the *toString* method of the new class. You also need to include a client class to test your code.

For Exercises 42 to 46, consider the following class:

```
public class Store
{
 public final double SALES_TAX_RATE = 0.06;
 private String name;

 public Store( String newName )
 {
  setName( newName );
 }

 public String getName( )
 {
  return name;
 }
```

```
    public void setName( String newName )
    {
      name = newName;
    }

    public String toString( )
    {
      return ( "name: " + name );
    }
}
```

42. Write a class encapsulating a web store, which inherits from *Store.* A web store has the following additional attributes: an Internet address and the programming language in which the website was written. Code the constructor and the *toString* method of the new class. You also need to include a client class to test your code.

43. Write a class encapsulating a music store, which inherits from *Store.* A music store has the following additional attributes: the number of titles it offers and its address. Code the constructor and the *toString* method of the new class. You also need to include a client class to test your code.

44. Write a class encapsulating a bike store, which inherits from *Store.* A bike store has the following additional attributes: the number of bicycle brands that it carries and whether it sponsors a bike club. Code the constructor and the *toString* method of the new class. You also need to include a client class to test your code.

45. Write a class encapsulating a grocery store, which inherits from *Store.* A grocery store has the following additional attributes: annual revenues and whether it is an independent store or part of a chain. Code the constructor and the *toString* method of the new class; also code a method returning the annual taxes paid by the store. You also need to include a client class to test your code.

46. Write a class encapsulating a restaurant, which inherits from *Store.* A restaurant has the following additional attributes: how many people are served every year and the average price per person. Code the constructor and the *toString* method of the new class; also code a method returning the average taxes per year. You also need to include a client class to test your code.

10.10.7 Programming Projects

47. Write a superclass encapsulating a rectangle. A rectangle has two attributes representing the width and the height of the rectangle. It has methods returning the perimeter and the area of the rectangle. This class has a subclass, encapsulating a parallelepiped, or box. A parallelepiped has a rectangle as its base, and another attribute, its length; it has two methods that calculate and return its area and volume. You also need to include a client class to test these two classes.

48. Write a superclass encapsulating a circle; this class has one attribute representing the radius of the circle. It has methods returning the perimeter and the area of the circle. This class has a subclass, encapsulating a cylinder. A cylinder has a circle as its base, and another attribute, its length; it has two methods, calculating and returning its area and volume. You also need to include a client class to test these two classes.

49. Write an *abstract* superclass encapsulating a shape: A shape has two *abstract* methods: one returning the perimeter of the shape, another returning the area of the shape. It also has a constant field named PI. This class has two non-*abstract* subclasses: one encapsulating a circle, and the other encapsulating a rectangle. A circle has one additional attribute, its radius. A rectangle has two additional attributes, its width and height. You also need to include a client class to test these two classes.

50. Write an *abstract* superclass encapsulating a vehicle: A vehicle has two attributes: its owner's name and its number of wheels. This class has two non-*abstract* subclasses: one encapsulating a bicycle, and the other encapsulating a motorized vehicle. A motorized vehicle has the following additional attributes: its engine volume displacement, in liters; and a method computing and returning a measure of horsepower—the number of liters times the number of wheels. You also need to include a client class to test these two classes.

51. Write an *abstract* superclass encapsulating some food; it has two attributes: its description and the number of calories per serving. It also has an *abstract* method taking a number of servings as a parameter and returning the number of calories. This class has two non-*abstract* subclasses: one encapsulating a liquid food (such as a drink,

for instance), and the other encapsulating a fruit. A liquid food has an additional attribute: its viscosity. A fruit has an additional attribute: its season. You also need to include a client class to test these two classes.

52. Write an *abstract* superclass encapsulating a college applicant: A college applicant has two attributes: the applicant's name and the college the applicant is applying to. This class has two non-*abstract* subclasses: one encapsulating an applicant for undergraduate school, and the other encapsulating an applicant for graduate school. An applicant for undergraduate school has two additional attributes: an SAT score and a GPA. An applicant for graduate school has one additional attribute: the college of origin. It also has a method that returns "from inside" if the college of origin is the same as the college applied to; otherwise, it returns "from outside." You also need to include a class to test these two classes.

53. Write an *abstract* superclass encapsulating a vacation: A vacation has two attributes: a budget and a destination. It has an *abstract* method returning by how much the vacation is over or under budget. This class has two non-*abstract* subclasses: one encapsulating an all-inclusive vacation, and the other encapsulating a vacation bought piece-meal. An all-inclusive vacation has three additional attributes: a brand (for instance ClubMed®); a rating, expressed as a number of stars; and a price. A piecemeal vacation has two additional attributes: a set of items (hotel, meal, airfare, . . .), and a set of corresponding costs. You also need to include a class to test these two classes.

54. Write an *abstract* superclass encapsulating a part, with two attributes: the part number, and a budget cost for it. This class has two non-*abstract* subclasses: one encapsulating a self-manufactured part, and the other encapsulating an outsourced part. A self-manufactured part has a cost and a drawing number; it also has a method returning whether it is over budget or under budget. An outsourced part has a set of suppliers, each with a price for the part. It also has a method to retrieve the lowest-cost supplier for a part and the corresponding cost. You also need to include a class to test these two classes.

55. Write an *abstract* superclass encapsulating a number; this class has one *abstract void* method: *square*. This class has two non-*abstract* subclasses: one encapsulating a rational number, and the other encapsulating a complex number. A rational number is represented by two integers, the numerator and the denominator of the rational number.

A complex number is represented by two real numbers, the real part and the complex part of the complex number. You also need to include a class to test these two classes.

10.10.8 Technical Writing

56. In a large organization, programmers develop a library of classes as they work on various projects. Discuss, in such an environment, how inheritance can be helpful in reusing code and therefore saving time.

57. Other programming languages allow multiple inheritance; that is, a class can inherit from several classes. In Java, a class can extend only one class, but can implement several interfaces. Discuss potential problems that can arise in other programming languages that allow inheritance from multiple classes.

10.10.9 Group Project (for a group of 1, 2, or 3 students)

58. Design and code a program, including the following classes, as well as a client applet class to test all the methods coded:

    An *abstract Shape* class, encapsulating a shape: a shape has one *abstract* method, *draw*, which takes one parameter, a *Graphics* object. *Shape* has three subclasses:

    - ❑ The *Line* class, encapsulating a line: A line can be represented by a starting (x, y) coordinate and an ending (x, y) coordinate. The *draw* method will draw a line between them.
    - ❑ The *Rectangle* class, encapsulating a rectangle: A rectangle can be represented by its (x, y) top-left corner of the rectangle, its width, and its height. The *draw* method will draw the corresponding rectangle.
    - ❑ The *Oval* class, encapsulating an oval: An oval can be represented by the (x, y) top-left corner coordinate of its bounding rectangle, its width, and its height. The *draw* method will draw the corresponding oval.

    Your applet class should prompt the user for the type of shape that the user wants to draw, prompt the user for the appropriate data, and then draw the figure in the applet window.