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.

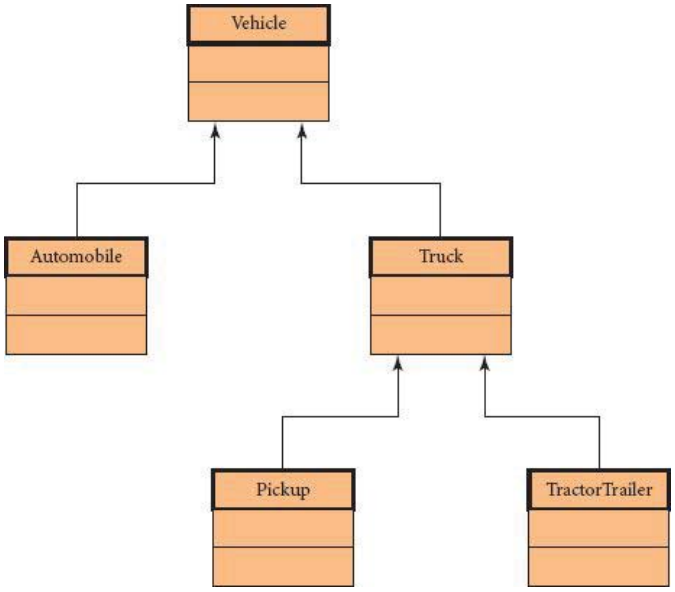
A subclass can add fields and methods, some of which may **override,** or hide, a field or method inherited from a superclass.

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 *Trac-torTrailer* 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 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.

Figure 10.1 *Vehicle* Class Hierarchy



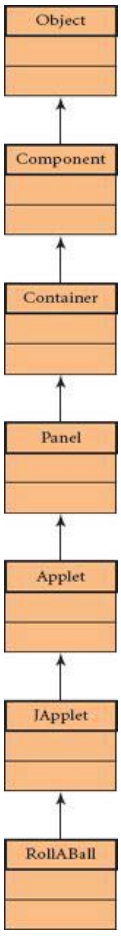
10.1 Inheritance

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.

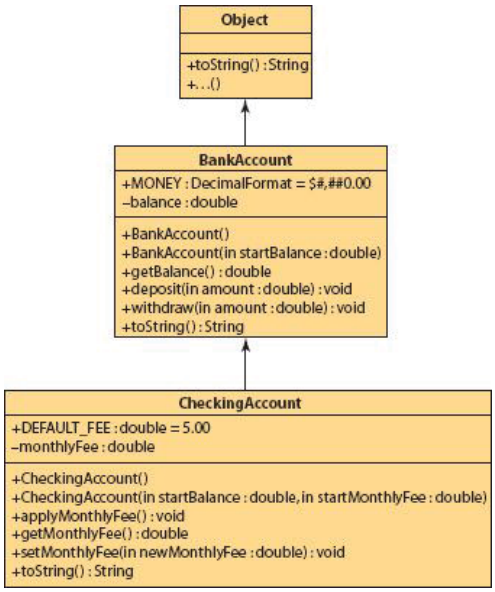
As you can see from Figure 10.2, our *RollABall* class has six superclasses. The class that a subclass refers to in the *extends* clause of the class definition is called its **direct superclass.** Thus, *JApplet* is the direct superclass of *RollABall.* Similarly, the class that *extends* the superclass 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 appropriate, subclasses can also provide new versions of inherited methods, which is called **overriding methods.**

Figure 10.3 The *BankAccount* Class Hierarchy



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

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   */
32   public double getBalance( )
33   {
34     return balance;
35   }
36
37   /** Deposit amount to account
38   *   @param amount  amount to deposit;
39   *                  amount must be >= 0.0
40   */
41   public void deposit( double amount )
42   {
43     if ( amount >= 0.0 )
44       balance += amount;
45     else
46       System.err.println( "Deposit amount must be positive." );
47   }
48
49   /** withdraw amount from account
50   *   @param amount   amount to withdraw;
51   *                   amount must be >= 0.0
52   *                   amount must be <= balance
53   */
54   public void withdraw( double amount )
55   {
56     if ( amount >= 0.0 && amount <= balance )
57       balance -= amount;
58     else
59       System.err.println( "Withdrawal amount must be positive "
60                           + "and cannot be greater than balance" );
61   }
62

63   /** toString
64   *   @return the balance formatted as money
65   */
66   public String toString( )
67   {
68     return ( "balance is " + MONEY.format( balance ));
```

```
69    }
70 }
```

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 *Checking-Account* 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 *BankAc-count* 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: *get-Balance, 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

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

```
 3  */
 4
 5  public class CheckingAccountClient
 6  {
 7    public static void main( String [ ] args )
 8    {
 9      CheckingAccount c1 = new CheckingAccount( );
10      System.out.println( "New checking account: " + c1 );
11
12      c1.deposit( 350.75 );
13      System.out.println( "\nAfter depositing $350.75: " + c1 );
14
15      c1.withdraw( 200.25 );
16      System.out.println( "\nAfter withdrawing $200.25: " + c1 );
17    }
18  }
```

EXAMPLE 10.3 *CheckingAccount* Client, Version 1

Figure 10.4 Output from *Checking-AccountClient*, 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

 7  import java.text.DecimalFormat;
 8
 9  public class BankAccount
10  {
11    public final DecimalFormat MONEY
12                  = new DecimalFormat( "$#,##0.00" );
```

```
13    privatedouble balance;
14
15    /** Default constructor
16    *   sets balance to 0.0
17    */
18    public BankAccount( )
19    {
20      balance = 0.0;
21      System.out.println( "In BankAccount default constructor" );
22    }
23
24    /** Overloaded constructor
25    *   @param startBalance beginning balance
26    */
27    public BankAccount( double startBalance )
28    {
29      if ( balance >= 0.0 )
30          balance = startBalance;
31       else
32          balance = 0.0;
33      System.out.println( "In BankAccount overloaded constructor" );
34    }
35
36    /** toString
37    *   @return the balance formatted as money
38    */
39    public String toString( )
40    {
41      return ( "balance is " + MONEY.format( balance ));
42    }
43  }
```

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

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.

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 *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
```

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).

SOFTWARE ENGINEERING TIP

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

```
 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 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.

Figure 10.5 Output from Example 10.6

```
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
```

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

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 *CheckingAc-count* 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

23    *  @param startMonthlyFee starting monthly fee
24    */
25    public CheckingAccount( double startBalance,
26                            double startMonthlyFee )
27    {
28       super( startBalance ); // call BankAccount constructor
29       setMonthlyFee( startMonthlyFee );
30    }
31
32    /** applyMonthlyFee method
33    * charges the monthly fee to the account
34    */
35    public void applyMonthlyFee( )
36    {
37       withdraw( monthlyFee );
38    }
39
40    /** accessor method for monthlyFee
41    *   @return monthlyFee
42    */
43    public double getMonthlyFee( )
44    {
45      return monthlyFee;
46    }
47
48    /** mutator method for monthlyFee
49    *   @param newMonthlyFee new value for monthlyFee
50    */
51    public void setMonthlyFee( double newMonthlyFee )
52    {
53       if ( monthlyFee >= 0.0 )
54           monthlyFee = newMonthlyFee;
55       else
56         System.err.println( "Monthly fee cannot be negative" );
57    }
58 }
```

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 *Checking-AccountClient*, 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
```

```
 1 /* CheckingAccount Client, Version 3
 2    Anderson, Franceschi
 3 */
 4
 5 public class CheckingAccountClient
 6 {
 7    public static void main( String [ ] args )
 8    {
 9      CheckingAccount c3 = new CheckingAccount( 100.00, 7.50 );
10      System.out.println( "New checking account:\n"
11                          + c3.toString( )
12                          + "; monthly fee is "
13                          + c3.getMonthlyFee( ) );
14
15      c3.applyMonthlyFee( ); // charge the fee to the account
16      System.out.println( "\nAfter charging monthly fee:\n"
17                          + c3.toString( )
18                          + "; monthly fee is "
19                          + c3.getMonthlyFee( ) );
20    }
21 }
```

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  {

 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
22  *   @param startBalance starting balance
23  *   @param startMonthlyFee starting monthly fee
24  */
25  public CheckingAccount( double startBalance,
26                          double startMonthlyFee )
27  {
28     super( startBalance ); // call BankAccount constructor
29     setMonthlyFee( startMonthlyFee );
30  }
31
32  /** applyMonthlyFee method
33  *   charges the monthly fee to the account
34  */
35  public void applyMonthlyFee( )
36  {
37     withdraw( monthlyFee );
38  }
39
40  /** accessor method for monthlyFee
41  *   @return monthlyFee
42  */
43  public double getMonthlyFee( )
44  {
45     return monthlyFee;
46  }
47
48  /** mutator method for monthlyFee
49  *   @param newMonthlyFee new value for monthlyFee
50  */
51  public void setMonthlyFee( double newMonthlyFee )

52  {
53    if ( monthlyFee >= 0.0 )
54         monthlyFee = newMonthlyFee;
55    else
56       System.err.println( "Monthly fee cannot be negative" );
57  }
58
59  /* toString method
60  *  @return String containing formatted balance and monthlyFee
61  *    invokes superclass toString to format balance
62  */
63  public String toString( )
64  {
65    return super.toString( )
66           + "; monthly fee is " + MONEY.format( monthlyFee );
67  }
68 }
```

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 /* Checking Account Client, Version 4
2    Anderson, Franceschi
3 */
4
5 public class CheckingAccountClient
6 {
7    public static void main( String [ ] args )
8    {
9      Checking Account 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 *Checking-AccountClient*, 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.

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.

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 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, 34

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   */

32   public double getBalance( )
33   {
34     return balance;
35   }
36
37   /** Deposit amount to account
38   *   @param amount  amount to deposit;
39   *                  amount must be >= 0.0
40   */
41   public void deposit( double amount )
42   {
43     if ( amount >= 0.0 )
44       balance += amount;
45     else
46       System.err.println( "Deposit amount must be positive." );
47   }
48
49   /** withdraw amount from account
50   *   @param amount    amount to withdraw;
51   *                    amount must be >= 0.0
52   *                    amount must be <= balance
53   */
54   public void withdraw( double amount )
55   {
56     if ( amount >= 0.0 && amount <= balance )
57       balance -= amount;
58     else
59       System.err.println( "Withdrawal amount must be positive "
60                           + "and cannot be greater than balance" );
61   }
62
63   /** toString
64   *   @return the balance formatted as money
65   */
66   public String toString( )
67   {
68     return ( "balance is " + MONEY.format( balance ) );
69   }
70 }
```

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 *getBalan*ce 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

22    *   @param  startBalance    starting balance
23    *   @param  startMonthlyFee starting monthly fee
24    */
25    public CheckingAccount( double startBalance,
26                            double startMonthlyFee )
27    {
28      super( startBalance );   // call BankAccount constructor
29      setMonthlyFee( startMonthlyFee );
30    }
31
32    /** applyMonthlyFee method
33    *   charges the monthly fee to the account
34    */
35    public void applyMonthlyFee( )
36    {
37      balance -= monthlyFee;
38       if ( balance < 0.0 )
39         System.err.println( "Warning: account is overdrawn" );
40    }
41
42    /** accessor method for monthlyFee
43    *   @return monthlyFee
44    */
45    public double getMonthlyFee( )
46    {
47      return monthlyFee;
48    }
49
50    /** mutator method for monthlyFee
51    *   @param newMonthlyFee new value for monthlyFee
52    */
53    public void setMonthlyFee( double newMonthlyFee )
54    {
55      if ( monthlyFee >= 0.0 )
56          monthlyFee = newMonthlyFee;
57      else
58         System.err.println( "Monthly fee cannot be negative" );
59    }
60
61    /* toString method
62    *  @return String containing formatted balance and monthlyFee
63    *  invokes superclass toString to format balance
64    */

65    public String toString( )
66    {
67      return super.toString( )
68              + "; monthly fee is " + MONEY.format( monthlyFee );
69    }
70 }
```

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 /* Checking Account Client, Version 5
 2     Anderson, Franceschi
```

```
 3 */
 4
 5 public class CheckingAccountClient
 6 {
 7    public static void main( String [ ] args )
 8    {
 9      Checking Account 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