Chapter 10 Inheritance

CHAPTER GOALS

- To learn about inheritance
- To understand how to inherit and override superclass methods
- To be able to invoke superclass constructors
- To learn about protected and package access control
- To understand the common superclass Object and how to override its toString and equals methods

G To use inheritance for customizing user interfaces

In this chapter, we discuss the important concept of inheritance. Specialized classes can be created that inherit behavior from more general classes. You will learn how to implement inheritance in Java, and how to make use of the Object class—the most general class in the inheritance hierarchy.

437

438

10.1 An Introduction to Inheritance

Inheritance is a mechanism for enhancing existing classes. If you need to implement a new class and a class representing a more general concept is already available, then the new class can inherit from the existing class. For example, suppose you need to define a class `SavingsAccount` to model an account that pays a fixed interest rate on deposits. You already have a class `BankAccount`, and a savings account is a special case of a bank account. In this case, it makes sense to use the language construct of inheritance. Here is the syntax for the class definition:

Inheritance is a mechanism for extending existing classes by adding methods and fields.

438

439

```
class SavingsAccount extends BankAccount
{
 new methods
 new instance fields
}
```

In the `SavingsAccount` class definition you specify only new methods and instance fields. The `SavingsAccount` class *automatically inherits* all methods and instance fields of the `BankAccount` class. For example, the `deposit` method automatically applies to savings accounts:

```
SavingsAccount collegeFund = new SavingsAccount(10);
    // Savings account with 10% interest
collegeFund.deposit(500);
    // OK to use BankAccount method with SavingsAccount object
```

We must introduce some more terminology here. The more general class that forms the basis for inheritance is called the *superclass*. In our example, `BankAccount` is the superclass and `SavingsAccount` is the subclass.

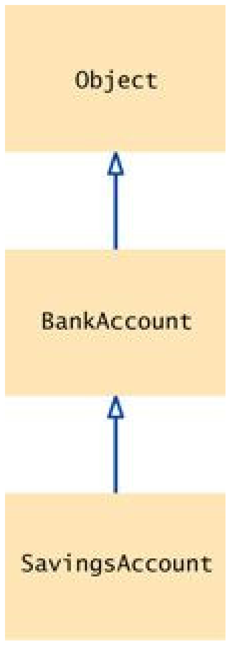
The more general class is called a superclass. The more specialized class that inherits from the superclass is called the subclass.

In Java, every class that does not specifically extend another class is a subclass of the class `Object`. For example, the `BankAccount` class extends the class `Object`. The `Object` class has a small number of methods that make sense for all objects, such as the `toString` method, which you can use to obtain a string that describes the state of an object.

Every class extends the Object class either directly or indirectly.

Figure 1 is a class diagram showing the relationship between the three classes `Object`, `BankAccount`, and `SavingsAccount`. In a class diagram, you denote inheritance by a solid arrow with a “hollow triangle” tip that points to the superclass.

Figure 1



An Inheritance Diagram

You may wonder at this point in what way inheritance differs from implementing an interface. An interface is not a class. It has *no state and no behavior*. It merely tells you which methods you should implement. A superclass has state and behavior, and the subclasses inherit them.

Inheriting from a class differs from implementing an interface: The subclass inherits behavior and state from the superclass.

One important reason for inheritance is *code reuse*. By inheriting an existing class, you do not have to replicate the effort that went into designing and perfecting that class. For example, when implementing the `SavingsAccount` class, you can rely on the `withdraw`, `deposit`, and `getBalance` methods of the `BankAccount` class without touching them.

One advantage of inheritance is code reuse.

Let's see how savings account objects are different from `BankAccount` objects. We will set an interest rate in the constructor, and we need a method to apply that interest periodically. That is, in addition to the three methods that can be applied to every account, there is an additional method `addInterest`. The new method and instance field must be defined in the subclass.

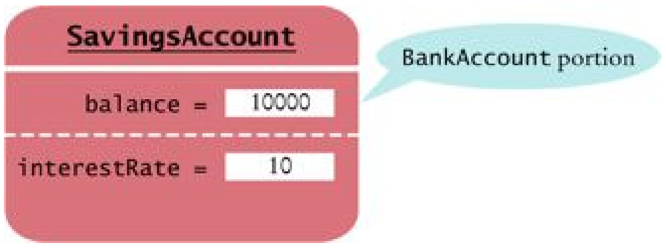
When defining a subclass, you specify added instance fields, added methods, and changed or overridden methods.

```
public class SavingsAccount extends BankAccount
{
   public SavingsAccount(double rate)
   {
  Constructor implementation
   }
   public void addInterest()
   {
    Method implementation
   }
   private double interestRate;
}
```

Figure 2 shows the layout of a `SavingsAccount` object. It inherits the `balance` instance field from the `BankAccount` superclass, and it gains one additional instance field: `interestRate`.

Next, you need to implement the new `addInterest` method. The method computes the interest due on the current balance and deposits that interest to the account.

Figure 2



Layout of a Subclass Object

SYNTAX 10.1 Inheritance

`class` *SubclassName* `extends` *SuperclassName*

```
    {
      methods
      instance fields
    }
```

Example:

public class SavingsAccount extends BankAccount

```
    {
       public SavingsAccount(double rate)
       {
          interestRate = rate;
       }
       public void addInterest()
       {
          double interest = getBalance() *
    interestRate / 100;
          deposit(interest);
       }
       private double interestRate;
    }
```

Purpose:

To define a new class that inherits from an existing class, and define the methods and instance fields that are added in the new class

```
public class SavingsAccount extends BankAccount
{
   public SavingsAccount(double rate)
   {
      interestRate = rate;
   }
   public void addInterest()
   {
      double interest = getBalance() * interestRate
/ 100;
      deposit(interest);
```

```
         }
         private double interestRate;
      }
```

You may wonder why the addInterest method calls the getBalance and deposit methods rather than directly updating the balance field of the superclass. This is a consequence of encapsulation. The balance field was defined as private in the BankAccount class. The addInterest method is defined in the SavingsAccount class. It does not have the right to access a private field of another class.

Note how the addInterest method calls the getBalance and deposit methods of the superclass without specifying an implicit parameter. This means that the calls apply to the same object, that is, the implicit parameter of the addInterest method. For example, if you call

```
collegeFund.addInterest();
```

then the following instructions are executed:

```
double interest = collegeFund. getBalance()
      * collegeFund. interestRate / 100;
collegeFund. deposit(interest);
```

In other words, the statements in the addInterest method are a shorthand for the following statements:

```
double interest = this. getBalance()
      * this.interestRate / 100;
this. deposit(interest);
```

(Recall that the this variable holds a reference to the implicit parameter.)

SELF CHECK

1. Which instance fields does an object of class SavingsAccount have?
2. Name four methods that you can apply to SavingsAccount objects.
3. If the class Manager extends the class Employee, which class is the superclass and which is the subclass?

COMMON ERROR 10.1: Confusing Super- and Subclasses

If you compare an object of type `SavingsAccount` with an object of type `BankAccount`, then you find that

- The keyword `extends` suggests that the `SavingsAccount` object is an extended version of a `BankAccount`.
- The `SavingsAccount` object is larger; it has an added instance field `interestRate`.
- The `SavingsAccount` object is more capable; it has an `addInterest` method.

It seems a superior object in every way. So why is `SavingsAccount` called the *subclass* and `BankAccount` the *superclass?*

The *super/sub* terminology comes from set theory. Look at the set of all bank accounts. Not all of them are `SavingsAccount` objects; some of them are other kinds of bank accounts. Therefore, the set of `SavingsAccount` objects is a *subset* of the set of all `BankAccount` objects, and the set of `BankAccount` objects is a *superset* of the set of `SavingsAccount` objects. The more specialized objects in the subset have a richer state and more capabilities.

10.2 Inheritance Hierarchies

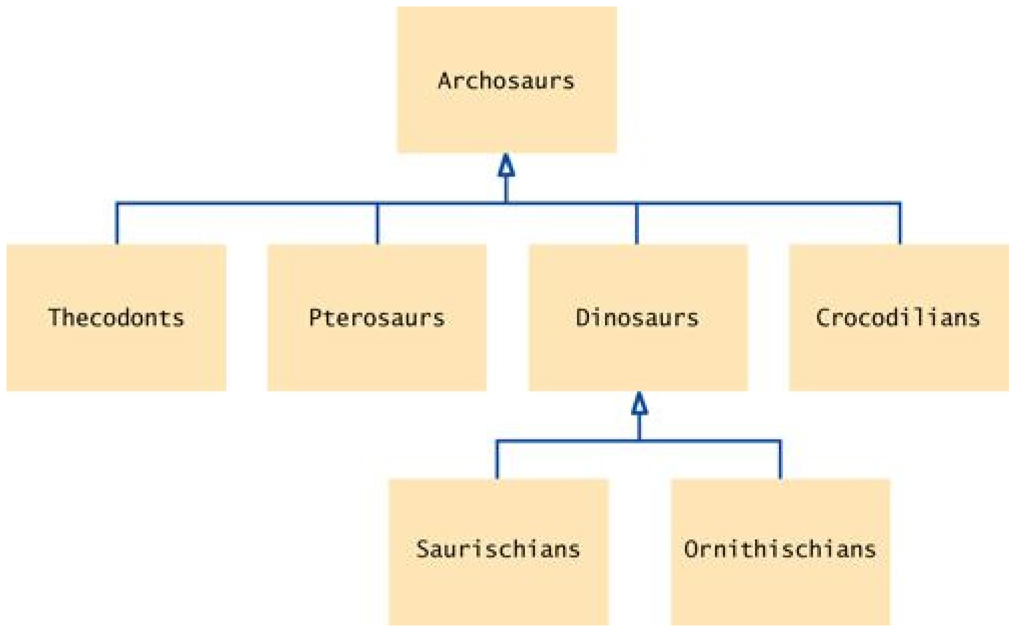
In the real world, you often categorize concepts into *hierarchies*. Hierarchies are frequently represented as trees, with the most general concepts at the root of the hierarchy and more specialized ones towards the branches. Figure 3 shows a typical example.

In Java it is equally common to group classes in complex *inheritance hierarchies*. The classes representing the most general concepts are near the root, more specialized classes towards the branches. For example, Figure 4 shows part of the hierarchy of Swing user interface components in Java.

Sets of classes can form complex inheritance hierarchies.

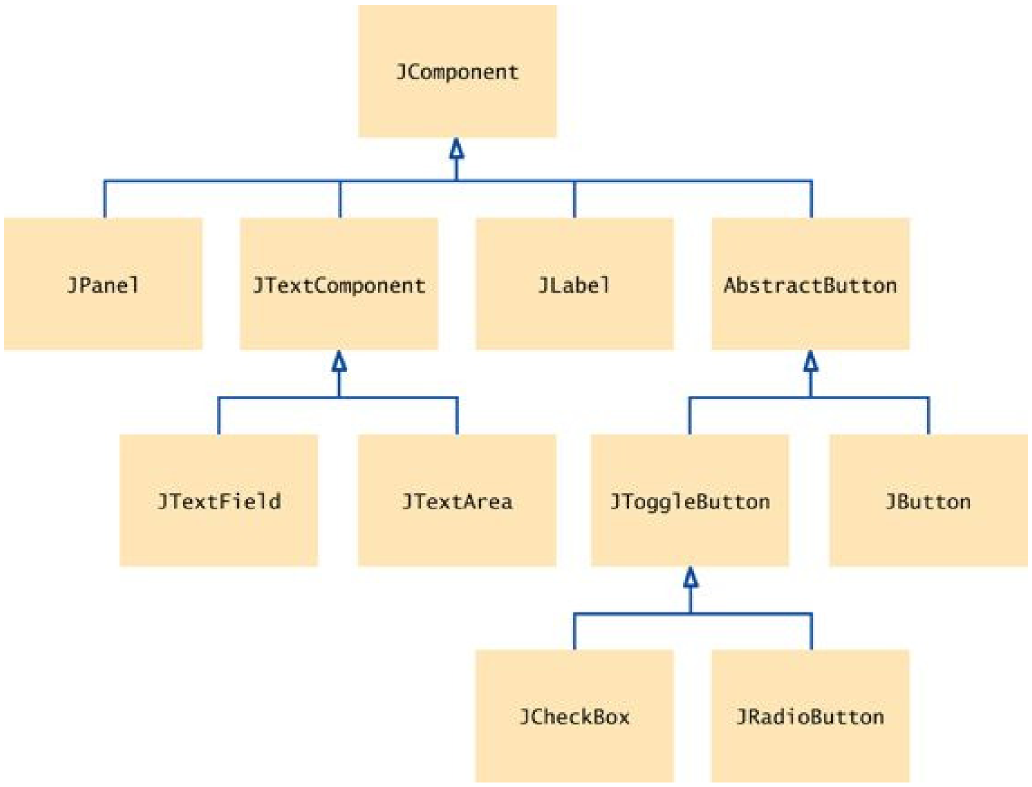
When designing a hierarchy of classes, you ask yourself which features and behaviors are common to all the classes that you are designing. Those common properties are collected in a superclass. For example, all user interface components have a width and height, and the `getWidth` and `getHeight` methods of the `JComponent` class return the component's dimensions. More specialized properties can be found in subclasses. For example, buttons can have text and icon labels. The class `AbstractButton`, but not the superclass `JComponent`, has methods to set and get the button text and icon, and instance fields to store them. The individual button classes (such as `JButton`, `JRadioButton`, and `JCheckBox`) inherit these properties. In fact, the `AbstractButton` class was created to express the commonality among these buttons.

Figure 3



A Part of the Hierarchy of Ancient Reptiles

Figure 4



A Part of the Hierarchy of Swing User Interface Components

We will use a simpler example of a hierarchy in our study of inheritance concepts. Consider a bank that offers its customers the following account types:

1. The checking account has no interest, gives you a small number of free transactions per month, and charges a transaction fee for each additional transaction.

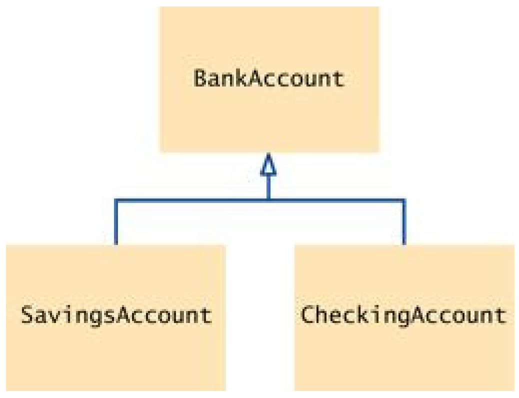
2. The savings account earns interest that compounds monthly. (In our implementation, the interest is compounded using the balance of the last day of the month, which is somewhat unrealistic. Typically, banks use either the average or the minimum daily balance. Exercise P10.1 asks you to implement this enhancement.)

Figure 5 shows the inheritance hierarchy. Exercise P10.2 asks you to add another class to this hierarchy.

Next, let us determine the behavior of these classes. All bank accounts support the `getBalance` method, which simply reports the current balance. They also support the `deposit` and `withdraw` methods, although the details of the implementation differ. For example, a checking account must keep track of the number of transactions to account for the transaction fees.

The checking account needs a method `deductFees` to deduct the monthly fees and to reset the transaction counter. The `deposit` and `withdraw` methods must be redefined to count the transactions.

Figure 5



Inheritance Hierarchy for Bank Account Classes

The savings account needs a method `addInterest` to add interest.

To summarize: The subclasses support all methods from the superclass, but their implementations may be modified to match the specialized purposes of the subclasses. In addition, subclasses are free to introduce additional methods.

SELF CHECK

4. What is the purpose of the `JTextComponent` class in Figure 4

5. Which instance field will we need to add to the `CheckingAccount` class?

10.3 Inheriting Instance Fields and Methods

When you form a subclass of a given class, you can specify additional instance fields and methods. In this section we will discuss this process in detail.

When defining the methods for a subclass, there are three possibilities.

1. You can *override* methods from the superclass. If you specify a method with the same *signature* (that is, the same name and the same parameter types), it overrides the method of the same name in the superclass. Whenever the method is applied to an object of the subclass type, the overriding method, and not the original method, is executed. For example, `CheckingAccount.deposit` overrides `BankAccount.deposit`.
2. You can *inherit* methods from the superclass. If you do not explicitly override a superclass method, you automatically inherit it. The superclass method can be applied to the subclass objects. For example, the `SavingsAccount` class inherits the `BankAccount.getBalance` method.
3. You can define new methods. If you define a method that did not exist in the superclass, then the new method can be applied only to subclass objects. For example, `SavingsAccount.addInterest` is a new method that does not exist in the superclass `BankAccount`.

The situation for instance fields is quite different. You can never override instance fields. For fields in a subclass, there are only two cases:

1. The subclass inherits all fields from the superclass. All instance fields from the superclass are automatically inherited. For example, all subclasses of the `BankAccount` class inherit the instance field `balance`.
2. Any new instance fields that you define in the subclass are present only in subclass objects. For example, the subclass `SavingsAccount` defines a new instance field `interestRate`.

What happens if you define a new field with the same name as a superclass field? For example, can you define another field named balance in the SavingsAccount class? This is legal but extremely undesirable. Each SavingsAccount object would have *two* instance fields of the same name. The two fields can hold different values, which is likely to lead to confusion—see Common Error 10.2.

We already implemented the BankAccount and SavingsAccount classes. Now we will implement the subclass CheckingAccount so that you can see in detail how methods and instance fields are inherited. Recall that the BankAccount class has three methods and one instance field:

```
public class BankAccount
{
   public double getBalance() { . . . }
   public void deposit(double amount) { . . . }
   public void withdraw(double amount) { . . . }
   private double balance;
}
```

The CheckingAccount class has an added method deductFees and an added instance field transactionCount, and it overrides the deposit and withdraw methods to increment the transaction count:

```
public class CheckingAccount extends BankAccount
{
   public void deposit(double amount) { . . . }
   public void withdraw(double amount) { . . . }
   public void deductFees() { . . . }
   private int transactionCount;
}
```

Each object of class CheckingAccount has two instance fields:

- balance (inherited from BankAccount)
- transactionCount (new to CheckingAccount)

446
447

You can apply four methods to CheckingAccount objects:

- getBalance() (inherited from BankAccount)
- deposit(double amount) (overrides BankAccount method)

- `withdraw(double amount)` (overrides `BankAccount` method)
- `deductFees()` (new to `CheckingAccount`)

Next, let us implement these methods. The `deposit` method increments the transaction count and deposits the money:

```
public class CheckingAccount extends BankAccount
{
   public void deposit(double amount)
   {
      transactionCount++;
      // Now add amount to balance
      . . .
   }
   . . .
}
```

Now we have a problem. We can't simply add `amount` to `balance`:

```
public class CheckingAccount extends BankAccount
{
   public void deposit(double amount)
   {
      transactionCount++;
      // Now add amount to balance
      balance = balance + amount; // Error
   }
   . . .
}
```

Although every `CheckingAccount` object has a `balance` instance field, that instance field is *private* to the superclass `BankAccount`. Subclass methods have no more access rights to the private data of the superclass than any other methods. If you want to modify a private superclass field, you must use a public method of the superclass.

A subclass has no access to private fields of its superclass.

How can we add the deposit amount to the balance, using the public interface of the `BankAccount` class? There is a perfectly good method for that purpose—namely, the `deposit` method of the `BankAccount` class. So we must invoke the `deposit` method on some object. On which object? The checking account into which the money is deposited—that is, the implicit parameter of the `deposit` method of the `CheckingAccount` class. To invoke another method on the implicit parameter, you don't specify the parameter but simply write the method name, like this:

```
public class CheckingAccount extends BankAccount
{
   public void deposit(double amount)
   {
      transactionCount++;
      // Now add amount to balance
      deposit(amount); // Not complete
   }
   . . .
}
```

But this won't quite work. The compiler interprets

```
deposit(amount);
```

as

```
this.deposit(amount);
```

The `this` parameter is of type `CheckingAccount`. There is a method called `deposit` in the `CheckingAccount` class. Therefore, that method will be called—but that is just the method we are currently writing! The method will call itself over and over, and the program will die in an infinite recursion (discussed in Chapter 13).

Use the super keyword to call a method of the superclass.

Instead, we must be specific that we want to invoke only the *superclass's* `deposit` method. There is a special keyword `super` for this purpose:

```
public class CheckingAccount extends BankAccount
{
   public void deposit(double amount)
   {
      transactionCount++;
```

```
         // Now add amount to balance
         super. deposit(amount);
      }
      . . .
   }
```

This version of the deposit method is correct. To deposit money into a checking account, update the transaction count and call the deposit method of the superclass.

The remaining methods are now straightforward.

```
   public class CheckingAccount extends BankAccount
   {
      . . .
      public void withdraw(double amount)
      {
         transactionCount++;
         // Now subtract amount from balance
         super. withdraw(amount);
      }
      public void deductFees()
      {
         if (transactionCount > FREE_TRANSACTIONS)
         {
            double fees = TRANSACTION_FEE
                  * (transactionCount -
   FREE_TRANSACTIONS);
            super. withdraw(fees);
         }
         transactionCount = 0;
      }
      . . .
      private static final int FREE_TRANSACTIONS = 3;
      private static final double TRANSACTION_FEE = 2.0;
   }
```

448

449

SYNTAX 10.2 Calling a Superclass Method

super. *methodName(parameters)*;

Example:

```
   public void deposit(double amount)
   {
      transactionCount++;
      super.deposit(amount);
   }
```

Purpose:

To call a method of the superclass instead of the method of the current class

SELF CHECK

6. Why does the `withdraw` method of the `CheckingAccount` class call `super.withdraw`?
7. Why does the `deductFees` method set the transaction count to zero?

COMMON ERROR 10.2: Shadowing Instance Fields

A subclass has no access to the private instance fields of the superclass. For example, the methods of the `CheckingAccount` class cannot access the `balance` field:

```
public class CheckingAccount extends BankAccount
{
   public void deposit(double amount)
   {
      transactionCount++;
      balance = balance + amount; // Error
   }
   . . .
}
```

It is a common beginner's error to “solve” this problem by adding *another* instance field with the same name.

```
public class CheckingAccount extends BankAccount
{
   public void deposit(double amount)
   {
      transactionCount++;
      balance = balance + amount;
   }
   . . .
```

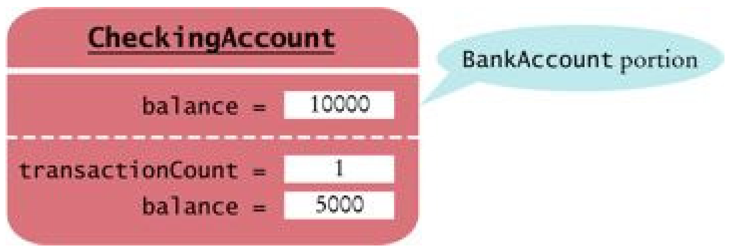
449

450

```
        private double balance; // Don't
    }
```

Sure, now the `deposit` method compiles, but it doesn't update the correct balance! Such a `CheckingAccount` object has two instance fields, both named `balance` (see Figure 6). The `getBalance` method of the superclass retrieves one of them, and the `deposit` method of the subclass updates the other.

Figure 6



Shadowing Instance Fields

COMMON ERROR 10.3: Failing to Invoke the Superclass Method

A common error in extending the functionality of a superclass method is to forget the `super.` qualifier. For example, to withdraw money from a checking account, update the transaction count and then withdraw the amount:

```
    public void withdraw(double amount)
    {
        transactionCount++;
        withdraw(amount);
        // Error—should be super.withdraw(amount)
    }
```

Here `withdraw(amount)` refers to the `withdraw` method applied to the implicit parameter of the method. The implicit parameter is of type `CheckingAccount`, and the `CheckingAccount` class has a `withdraw` method, so that method is called. Of course, that calls the current method all over again, which will call itself yet again, over and over, until the program runs out of memory. Instead, you must precisely identify which `withdraw` method you want to call.

Another common error is to forget to call the superclass method altogether. Then the functionality of the superclass mysteriously vanishes.

10.4 Subclass Construction

In this section, we discuss the implementation of constructors in subclasses. As an example, let us define a constructor to set the initial balance of a checking account.

We want to invoke the `BankAccount` constructor to set the balance to the initial balance. There is a special instruction to call the superclass constructor from a subclass constructor. You use the keyword `super`, followed by the construction parameters in parentheses:

```
public class CheckingAccount extends BankAccount
{
   public CheckingAccount(double initialBalance)
   {
      // Construct superclass
      super(initialBalance);
      // Initialize transaction count
      transactionCount = 0;
   }
   . . .
}
```

When the keyword `super` is followed by a parenthesis, it indicates a call to the superclass constructor. When used in this way, the constructor call must be *the first statement of the subclass constructor*. If `super` is followed by a period and a method name, on the other hand, it indicates a call to a superclass method, as you saw in the preceding section. Such a call can be made anywhere in any subclass method.

To call the superclass constructor, you use the super keyword in the first statement of the subclass constructor.

SYNTAX 10.3 Calling a Superclass Constructor

accessSpecifier ClassName(parameterType parameterName, . . .)

```
    {
       super(parameters);
       . . .
    }
```

Example:

public CheckingAccount(double initialBalance)

```
    {
       super(initialBalance);
       transactionCount = 0;
    }
```

Purpose:

To invoke the constructor of the superclass. Note that this statement must be the first statement of the subclass constructor.

The dual use of the `super` keyword is analogous to the dual use of the `this` keyword (see Advanced Topic 3.1).

If a subclass constructor does not call the superclass constructor, the superclass is constructed with its default constructor (that is, the constructor that has no parameters). However, if all constructors of the superclass require parameters, then the compiler reports an error.

For example, you can implement the `CheckingAccount` constructor without calling the superclass constructor. Then the `BankAccount` class is constructed with its default constructor, which sets the balance to zero. Of course, then the `CheckingAccount` constructor must explicitly deposit the initial balance.

Most commonly, however, subclass constructors have some parameters that they pass on to the superclass and others that they use to initialize subclass fields.

SELF CHECK

8. Why didn't the `SavingsAccount` constructor in Section 10.1 Call its Superclass Constructor?

9. When you invoke a superclass method with the `super` keyword, does the call have to be the first statement of the subclass method?

10.5 Converting Between Subclass and Superclass Types

It is often necessary to convert a subclass type to a superclass type. Occasionally, you need to carry out the conversion in the opposite direction. This section discusses the conversion rules.

Subclass references can be converted to superclass references.

The class `SavingsAccount` extends the class `BankAccount`. In other words, a `SavingsAccount` object is a special case of a `BankAccount` object. Therefore, a reference to a `SavingsAccount` object can be converted to a `BankAccount` reference.

```
SavingsAccount collegeFund = new SavingsAccount(10);
BankAccount anAccount = collegeFund;
```

Furthermore, all references can be converted to the type `Object`.

```
Object anObject = collegeFund;
```

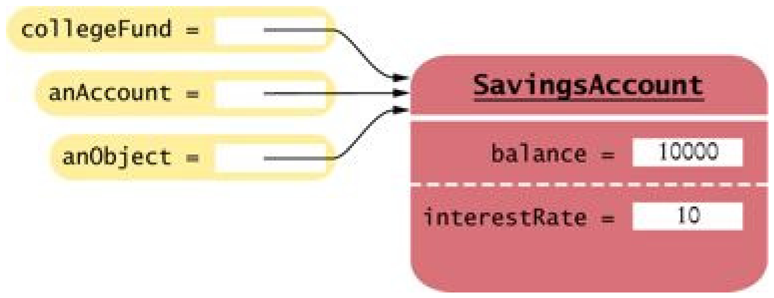
Now the three object references stored in `collegeFund`, `anAccount`, and `anObject` all refer to the same object of type `SavingsAccount` (see Figure 7).

However, the object reference `anAccount` knows less than the full story about the object to which it refers. Because `anAccount` is an object of type `BankAccount`, you can use the `deposit` and `withdraw` methods to change the balance of the savings account. You cannot use the `addInterest` method, though—it is not a method of the `BankAccount` superclass:

```
anAccount.deposit(1000); // OK
anAccount.addInterest();
```

// No—not a method of the class to which `anAccount` belongs

Figure 7



Variables of Different Types Refer to the Same Object

And, of course, the variable `anObject` knows even less. You can't even apply the `deposit` method to it—`deposit` is not a method of the `Object` class.

Conversion of references is different from a numerical conversion, such as a conversion from an integer to a floating-point number. If you convert an integer, say 4, into the `double` value 4.0, then the representation changes: the `double` value 4.0 uses a different sequence of bits than the `int` value 4. However, when you convert a `SavingsAccount` reference to a `BankAccount` reference, then the value of the reference stays the same—it is the memory location of the object. However, after conversion, less information is known about the object. We only know that it is a bank account. It might be a plain bank account, a savings account, or another kind of bank account.

Why would anyone *want* to know less about an object and store a reference in an object field of a superclass? This can happen if you want to *reuse code* that knows about the superclass but not the subclass. Here is a typical example. Consider a `transfer` method that transfers money from one account to another:

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

You can use this method to transfer money from one bank account to another:

```
BankAccount momsAccount = . . . ;
BankAccount harrysAccount = . . . ;
momsAccount.transfer(1000, harrysAccount);
```

You can *also* use the method to transfer money into a `CheckingAccount`:

```
CheckingAccount harrysChecking = . . . ;
momsAccount.transfer(1000, harrysChecking);
   // OK to pass a CheckingAccount reference to a method expecting a
BankAccount
```

The `transfer` method expects a reference to a `BankAccount`, and it gets a reference to the subclass `CheckingAccount`. Fortunately, rather than complaining about a type mismatch, the compiler simply copies the subclass reference `harrysChecking` to the superclass reference `other`. The `transfer` method 453 doesn't actually know that, in this case, `other` refers to a `CheckingAccount` 454 reference. It knows only that `other` is a `BankAccount`, and it doesn't need to know anything else. All it cares about is that the `other` object can carry out the `deposit` method.

Very occasionally, you need to carry out the opposite conversion, from a superclass reference to a subclass reference. For example, you may have a variable of type `Object`, and you know that it actually holds a `BankAccount` reference. In that case, you can use a cast to convert the type:

```
BankAccount anAccount = (BankAccount) anObject;
```

However, this cast is somewhat dangerous. If you are wrong, and `anObject` actually refers to an object of an unrelated type, then an exception is thrown.

To protect against bad casts, you can use the `instanceof` operator. It tests whether an object belongs to a particular type. For example,

```
anObject instanceof BankAccount
```

returns `true` if the type of `anObject` is convertible to `BankAccount`. This happens if `anObject` refers to an actual `BankAccount` or a subclass such as `SavingsAccount`. Using the `instanceof` operator, a safe cast can be programmed as follows:

```
if (anObject instanceof BankAccount)
{
   BankAccount anAccount = (BankAccount) anObject;
   . . .
}
```

The `instanceof` operator tests whether an object belongs to a particular type.

SYNTAX 10.4 The `instanceof` Operator

object `instanceof` *TypeName*

Example:

```
if (anObject instanceof BankAccount)
{
   BankAccount anAccount = (BankAccount) anObject;
   . . .
}
```

Purpose:

To return `true` if the *object* is an instance of *TypeName* (or one of its subtypes), and `false` otherwise

SELF CHECK

10. Why did the second parameter of the `transfer` method have to be of type `BankAccount` and not, for example, `SavingsAccount`?

11. Why can't we change the second parameter of the `transfer` method to the type `Object`?

454

455

10.6 Polymorphism

In Java, the type of a variable does not completely determine the type of the object to which it refers. For example, a variable of type `BankAccount` can hold a reference to an actual `BankAccount` object or a subclass object such as `SavingsAccount`. You already encountered this phenomenon in Chapter 9 with variables whose type was an interface. A variable whose type is `Measurable` holds a reference to an object of a class that implements the `Measurable` interface, perhaps a `Coin` object or an object of an entirely different class.

What happens when you invoke a method? For example,

```
BankAccount anAccount = new CheckingAccount();
anAccount.deposit(1000);
```

Which deposit method is called? The `anAccount` parameter has type `BankAccount`, so it would appear as if `BankAccount.deposit` is called. On the other hand, the `CheckingAccount` class provides its own `deposit` method that updates the transaction count. The `anAccount` field actually refers to an object of the subclass `CheckingAccount`, so it would be appropriate if the `CheckingAccount.deposit` method were called instead.

In Java, method calls *are always determined by the type of the actual object*, not the type of the object reference. That is, if the actual object has the type `CheckingAccount`, then the `CheckingAccount.deposit` method is called. It does not matter that the object reference is stored in a field of type `BankAccount`. As we discussed in Chapter 9, the ability to refer to objects of multiple types with varying behavior is called *polymorphism*.

If polymorphism is so powerful, why not store all account references in variables of type `Object`? This does not work because the compiler needs to check that only legal methods are invoked. The `Object` type does not define a `deposit` method—the `BankAccount` type (at least) is required to make a call to the `deposit` method.

Have another look at the `transfer` method to see polymorphism at work. Here is the implementation of the method:

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

Suppose you call

```
anAccount.transfer(1000, anotherAccount);
```

Two method calls are the result:

```
anAccount.withdraw(1000);
anotherAccount.deposit(1000);
```

Depending on the actual types of `anAccount` and `anotherAccount`, different versions of the `withdraw` and `deposit` methods are called.

If you look into the implementation of the `transfer` method, it may not be immediately obvious that the first method call

```
withdraw(amount);
```

depends on the type of an object. However, that call is a shortcut for

```
this.withdraw(amount);
```

The `this` parameter holds a reference to the implicit parameter, which can refer to a `BankAccount` or a subclass object.

The following program calls the polymorphic `withdraw` and `deposit` methods. You should manually calculate what the program should print for each account balance, and confirm that the correct methods have in fact been called.

ch10/accounts/AccountTester.java

```
/**
   This program tests the BankAccount class and
   its subclasses.
*/
public class AccountTester
{
   public static void main(String[] args)
   {
      SavingsAccount momsSavings
            = new SavingsAccount(0.5);

      CheckingAccount harrysChecking
            = new CheckingAccount(100);

      momsSavings.deposit(10000);

      momsSavings.transfer(2000,
harrysChecking);
      harrysChecking.withdraw(1500);
```

```
        harrysChecking.withdraw(80);

        momsSavings.transfer(1000,
harrysChecking);
        harrysChecking.withdraw(400);

        // Simulate end of month
        momsSavings.addInterest();
        harrysChecking.deductFees();

        System.out.println("Mom's savings
balance: "
        + momsSavings.getBalance());
        System.out.println("Expected: 7035");

        System.out.println("Harry's checking
balance: "
              + harrysChecking.getBalance());
        System.out.println("Expected: 1116");
    }
}
```

ch10/accounts/BankAccount.java

```
/**
   A bank account has a balance that can be changed by
   deposits and withdrawals.
*/
public class BankAccount
{
   /**
      Constructs a bank account with a zero balance.
   */
   public BankAccount()
   {
      balance = 0;
   }

   /**
      Constructs a bank account with a given balance.
      @param initialBalance the initial balance
   */
   public BankAccount(double initialBalance)
```

```
    {
        balance = initialBalance;
    }

    /**
        Deposits money into the bank account.
        @param amount the amount to deposit
    */
    public void deposit(double amount)
    {
        balance = balance + amount;
    }

    /**
        Withdraws money from the bank account.
        @param amount the amount to withdraw
    */
    public void withdraw(double amount)
    {
        balance = balance - amount;
    }

    /**
        Gets the current balance of the bank account.
        @return the current balance
    */
    public double getBalance()
    {
        return balance;
    }

    /**
        Transfers money from the bank account to another account.
        @param amount the amount to transfer
        @param other the other account
    */
    public void transfer(double amount, BankAccount other)
    {
        withdraw(amount);
        other.deposit(amount);
    }

    private double balance;
}
```

ch10/accounts/CheckingAccount.java

```
/**
    A checking account that charges transaction fees.
*/
public class CheckingAccount extends BankAccount
{
    /**
        Constructs a checking account with a given balance.
        @param initialBalance the initial balance
    */
    public CheckingAccount(double initialBalance)
    {
        // Construct superclass
        super(initialBalance);

        // Initialize transaction count
        transactionCount = 0;
    }

    public void deposit(double amount)
    {
        transactionCount++;
        // Now add amount to balance
        super.deposit(amount);
    }

    public void withdraw(double amount)
    {
        transactionCount++;
        // Now subtract amount from balance
        super.withdraw(amount);
    }

    /**
        Deducts the accumulated fees and resets the
        transaction count.
```

```
   */
   public void deductFees()
   {
      if (transactionCount > FREE_TRANSACTIONS)
      {
         double fees = TRANSACTION_FEE *
                (transactionCount -
FREE_TRANSACTIONS);
         super.withdraw(fees);
      }
      transactionCount = 0
   }

   private int transactionCount;

   private static final int FREE_TRANSACTIONS
= 3;
   private static final double TRANSACTION_FEE
= 2.0;
}
```

ch10/accounts/SavingsAccount.java

```
/**
   An account that earns interest at a fixed rate.
*/
public class SavingsAccount extends BankAccount
{
   /**
      Constructs a bank account with a given interest rate.
      @param rate the interest rate
   */
   public SavingsAccount(double rate)
   {
      interestRate = rate;
   }

   /**
      Adds the earned interest to the account balance.
   */
   public void addInterest()
   {
      double interest = getBalance() *
interestRate / 100;
```

```
        deposit(interest);
    }

    private double interestRate;
}
```

Output

```
Mom's savings balance: 7035.0
Expected: 7035
Harry's checking balance: 1116.0
Expected: 1116
```

SELF CHECK

12. If a is a variable of type `BankAccount` that holds a non-`null` reference, what do you know about the object to which a refers?

13. If a refers to a checking account, what is the effect of calling `a.transfer(1000, a)`?

ADVANCED TOPIC 10.1: Abstract Classes

When you extend an existing class, you have the choice whether or not to redefine the methods of the superclass. Sometimes, it is desirable to *force* programmers to redefine a method. That happens when there is no good default for the superclass, and only the subclass programmer can know how to implement the method properly.

Here is an example. Suppose the First National Bank of Java decides that every account type must have some monthly fees. Therefore, a `deductFees` method should be added to the `BankAccount` class:

```
public class BankAccount
{
   public void deductFees() { . . . }
   . . .
}
```

But what should this method do? Of course, we could have the method do nothing. But then a programmer implementing a new subclass might simply forget to implement the deductFees method, and the new account would inherit the do-nothing method of the superclass. There is a better way—declare the deductFees method as an *abstract method*:

```
public abstract void deductFees();
```

An abstract method has no implementation. This forces the implementors of subclasses to specify concrete implementations of this method. (Of course, some subclasses might decide to implement a do-nothing method, but then that is their choice—not a silently inherited default.)

An abstract method is a method whose implementation is not specified.

You cannot construct objects of classes with abstract methods. For example, once the BankAccount class has an abstract method, the compiler will flag an attempt to create a new BankAccount() as an error. Of course, if the CheckingAccount subclass overrides the deductFees method and supplies an implementation, then you can create CheckingAccount objects.

An abstract class is a class that cannot be instantiated.

A class for which you cannot create objects is called an *abstract class*. A class for which you can create objects is sometimes called a *concrete class*. In Java, you must declare all abstract classes with the keyword abstract:

```
public abstract class BankAccount
{
   public abstract void deductFees();
   . . .
}
```

A class that defines an abstract method, or that inherits an abstract method without overriding it, *must* be declared as abstract. You can also declare classes with no abstract methods as abstract. Doing so prevents programmers from creating instances of that class but allows them to create their own subclasses.

Note that you cannot construct an *object* of an abstract class, but you can still have an *object reference* whose type is an abstract class. Of course, the actual object to which it refers must be an instance of a concrete subclass:

```
BankAccount anAccount; // OK
anAccount = new BankAccount(); // Error—BankAccount is
abstract
anAccount = new SavingsAccount(); // OK
anAccount = null; // OK
```

The reason for using abstract classes is to force programmers to create subclasses. By specifying certain methods as abstract, you avoid the trouble of coming up with useless default methods that others might inherit by accident.

Abstract classes differ from interfaces in an important way—they can have instance fields, and they can have concrete methods and constructors.

ADVANCED TOPIC 10.2: Final Methods and Classes

In Advanced Topic 10.1 you saw how you can force other programmers to create subclasses of abstract classes and override abstract methods. Occasionally, you may want to do the opposite and *prevent* other programmers from creating subclasses or from overriding certain methods. In these situations, you use the `final` keyword. For example, the `String` class in the standard Java library has been declared as

```
public final class String { . . . }
```

That means that nobody can extend the `String` class.

The `String` class is meant to be *immutable*—string objects can't be modified by any of their methods. Since the Java language does not enforce this, the class designers did. Nobody can create subclasses of `String`; therefore, you know that all `String` references can be copied without the risk of mutation.

You can also declare individual methods as final:

```
public class SecureAccount extends BankAccount
{
   . . .
   public final boolean checkPassword(String
password)
   {
      . . .
```

```
        }
    }
```

This way, nobody can override the `checkPassword` method with another method that simply returns `true`.

10.7 Access Control

Java has four levels of controlling access to fields, methods, and classes:

- `public` access
- `private` access
- `protected` access (see Advanced Topic 10.3)
- package access (the default, when no access modifier is given)

You have already used the `private` and `public` modifiers extensively. Private features can be accessed only by the methods of their own class. Public features can be accessed by methods of all classes. We will discuss protected access in Advanced Topic 10.3—we will not need it in this book.

A field or method that is not declared as `public`, `private`, or `protected` can be accessed by all classes in the same package, which is usually not desirable.

If you do not supply an access control modifier, then the default is *package access*. That is, all methods of classes in the same package can access the feature. For example, if a class is declared as `public`, then all other classes in all packages can use it. But if a class is declared without an access modifier, then only the other classes in the same package can use it. Package access is a good default for classes, but it is extremely unfortunate for fields. Instance and static fields of classes should always be `private`. There are a few exceptions:

- Public constants (`public static final` fields) are useful and safe.
- Some objects, such as `System.out`, need to be accessible to all programs and therefore should be public.

- Very occasionally, several classes in a package must collaborate very closely. In that case, it may make sense to give some fields package access. But inner classes are usually a better solution—you have seen examples in Chapter 9.

It is a common error to *forget* the keyword `private`, thereby opening up a potential security hole. For example, at the time of this writing, the `Window` class in the `java.awt` package contained the following declaration:

```
public class Window extends Container
{
   String warningString;
   . . .
}
```

The programmer was careless and didn't make the field private. There actually was no good reason to grant package access to the `warningString` field—no other class accesses it. It is a security risk. Packages are not closed entities—any programmer can make a new class, add it to the `java.awt` package, and gain access to the `warningString` fields of all `Window` objects! (Actually, this possibility bothered the Java implementors so much that recent versions of the virtual machine refuse to load unknown classes whose package name starts with "`java.`". Your own packages, however, do not enjoy this protection.)

Package access for fields is rarely useful, and most fields are given package access by accident because the programmer simply forgot the `private` keyword.

Methods should generally be `public` or `private`. We recommend avoiding the use of package-visible methods.

Classes and interfaces can have public or package access. Classes that are generally useful should have public access. Classes that are used for implementation reasons should have package access. You can hide them even better by turning them into inner classes; you saw examples of inner classes in Chapter 9. There are a few examples of public inner classes, such as the `Ellipse2D.Double` class that you saw in Chapter 2 (Section 2.13). However, in general, inner classes should not be public.

SELF CHECK

14. What is a common reason for defining package-visible instance fields?

15. If a class with a public constructor has package access, who can construct objects of it?

COMMON ERROR 10.4: Accidental Package Access

It is very easy to forget the `private` modifier for instance fields.

```
public class BankAccount
{
   . . .
   double balance; // Package access really intended?
}
```

Most likely, this was just an oversight. The programmer probably never intended to grant access to this field to other classes in the same package. The compiler won't complain, of course. Much later, some other programmer may take advantage of the access privilege, either out of convenience or out of evil intent. This is a serious problem, and you must get into the habit of scanning your field declarations for missing `private` modifiers.

COMMON ERROR 10.5: Making Inherited Methods Less Accessible

If a superclass declares a method to be publicly accessible, you cannot override it to be more private. For example,

```
public class BankAccount
{
   public void withdraw(double amount) { . . . }
   . . .
```

463

464

```
}
public class CheckingAccount extends BankAccount
{
   private void withdraw(double amount) { . . . }
      // Error—subclass method cannot be more private
```

```
        . . .
    }
```

The compiler does not allow this, because the increased privacy would be an illusion. Anyone can still call the method through a superclass reference:

```
    BankAccount account = new CheckingAccount();
    account.withdraw(100000); // Calls
    CheckingAccount.withdraw
```

Because of polymorphism, the subclass method is called.

These errors are usually an oversight. If you forget the `public` modifier, your subclass method has package access, which is more restrictive. Simply restore the `public` modifier, and the error will go away.

ADVANCED TOPIC 10.3: Protected Access

We ran into a hurdle when trying to implement the `deposit` method of the `CheckingAccount` class. That method needed access to the `balance` instance field of the superclass. Our remedy was to use the appropriate method of the superclass to set the balance.

Java offers another solution to this problem. The superclass can declare an instance field as *protected*:

```
    public class BankAccount
    {
        . . .
        protected double balance;
    }
```

Protected data in an object can be accessed by the methods of the object's class and all its subclasses. For example, `CheckingAccount` inherits from `BankAccount`, so its methods can access the protected instance fields of the `BankAccount` class. Furthermore, protected data can be accessed by all methods of classes in the same package.

Protected features can be accessed by all subclasses and all classes in the same package.

Some programmers like the `protected` access feature because it seems to strike a balance between absolute protection (making all fields private) and no protection at all (making all fields public). However, experience has shown that protected fields are subject to the same kinds of problems as public fields. The designer of the superclass has no control over the authors of subclasses. Any of the subclass methods can corrupt the superclass data. Furthermore, classes with protected fields are hard to modify. Even if the author of the superclass would like to change the data implementation, the protected fields cannot be changed, because someone somewhere out there might have written a subclass whose code depends on them. 464

465 In Java, protected fields have another drawback—they are accessible not just by subclasses, but also by other classes in the same package.

It is best to leave all data private. If you want to grant access to the data to subclass methods only, consider making the *accessor* method protected.

10.8 Object: The Cosmic Superclass

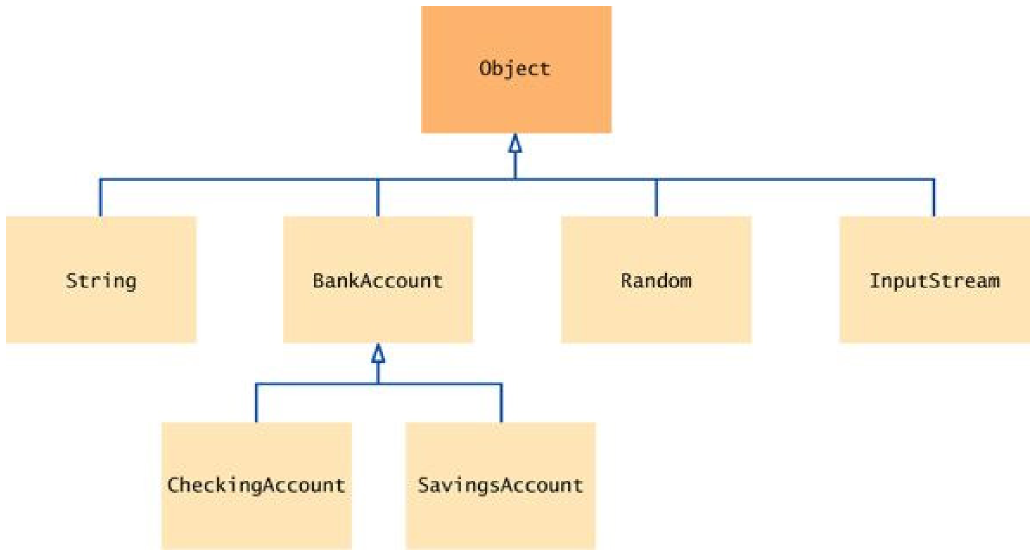
In Java, every class that is defined without an explicit `extends` clause automatically extends the class `Object`. That is, the class `Object` is the direct or indirect superclass of *every* class in Java (see Figure 8).

Of course, the methods of the `Object` class are very general. Here are the most useful ones:

Method	Purpose
`String toString()`	Returns a string representation of the object
`boolean equals(Object otherObject)`	Tests whether the object equals another object
`Object clone()`	Makes a full copy of an object

It is a good idea for you to override these methods in your classes.

Figure 8



The Object Class Is the Superclass of Every Java Class

10.8.1 Overriding the `toString` Method

The `toString` method returns a string representation for each object. It is used for debugging. For example,

```
Rectangle box = new Rectangle(5, 10, 20, 30);
String s = box.toString();
  // Sets s to
"java.awt.Rectangle[x=5,y=10,width=20,height=30]"
```

Define the `toString` method to yield a string that describes the object state.

In fact, this `toString` method is called whenever you concatenate a string with an object. Consider the concatenation

```
"box=" + box;
```

On one side of the + concatenation operator is a string, but on the other side is an object reference. The Java compiler automatically invokes the `toString` method to turn the object into a string. Then both strings are concatenated. In this case, the result is the string

```
"box=java.awt.Rectangle[x=5,y=10,width=20,height=30]"
```

The compiler can invoke the `toString` method, because it knows that *every* object has a `toString` method: Every class extends the `Object` class, and that class defines `toString`.

As you know, numbers are also converted to strings when they are concatenated with other strings. For example,

```
int age = 18;
String s = "Harry's age is " + age;
   // Sets s to "Harry's age is 18"
```

In this case, the `toString` method is not involved. Numbers are not objects, and there is no `toString` method for them. There is only a small set of primitive types, however, and the compiler knows how to convert them to strings.

Let's try the `toString` method for the `BankAccount` class:

```
BankAccount momsSavings = new BankAccount(5000);
String s = momsSavings.toString();
   // Sets s to something like "BankAccount@d24606bf"
```

That's disappointing—all that's printed is the name of the class, followed by the *hash code*, a seemingly random code. The hash code can be used to tell objects apart—different objects are likely to have different hash codes. (See Chapter 16 for the details.)

We don't care about the hash code. We want to know what is inside the object. But, of course, the `toString` method of the `Object` class does not know what is inside the `BankAccount` class. Therefore, we have to override the method and supply our own version in the `BankAccount` class. We'll follow the same format that the `toString` method of the `Rectangle` class uses: first print the name of the class, and then the values of the instance fields inside brackets.

```
public class BankAccount
   {
```

466

467

```
   . . .
   public String toString()
```

```
         {
            return "BankAccount[balance=" + balance + "]";
         }
      }
```

This works better:

```
      BankAccount momsSavings = new BankAccount(5000);
      String s = momsSavings.toString();
         // Sets s to "BankAccount[balance=5000]"
```

PRODUCTIVITY HINT 10.1: Supply `toString` in All Classes

If you have a class whose `toString()` method returns a string that describes the object state, then you can simply call `System.out.println(x)` whenever you need to inspect the current state of an object `x`. This works because the `println` method of the `PrintStream` class invokes `x.toString()` when it needs to print an object, which is extremely helpful if there is an error in your program and the objects don't behave the way you think they should. You can simply insert a few print statements and peek inside the object state during the program run. Some debuggers can even invoke the `toString` method on objects that you inspect.

Sure, it is a bit more trouble to write a `toString` method when you aren't sure your program ever needs one—after all, it might work correctly on the first try. Then again, many programs don't work on the first try. As soon as you find out that yours doesn't, consider adding those `toString` methods to help you debug the program.

ADVANCED TOPIC 10.4: Inheritance and the `toString` Method

You just saw how to write a `toString` method: Form a string consisting of the class name and the names and values of the instance fields. However, if you want your `toString` method to be usable by subclasses of your class, you need to work a bit harder. Instead of hardcoding the class name, you should call the `getClass` method to obtain a *class* object, an object of the `Class` class that describes classes and their properties. Then invoke the getName method to get the name of the class:

```
public String toString()
{
   return getClass().getName() + "[balance="
         + balance + "]";
}
```

Then the toString method prints the correct class name when you apply it to a subclass, say a SavingsAccount.

```
SavingsAccount momsSavings = . . . ;
System.out.println(momsSavings);
// Prints "SavingsAccount[balance=10000]"
```

Of course, in the subclass, you should override toString and add the values of the subclass instance fields. Note that you must call super.toString to get the superclass field values—the subclass can't access them directly.

```
public class SavingsAccount extends BankAccount
{
   public String toString()
   {
      return super.toString() +
            "[interestRate=" + interestRate + "]";
   }
}
```

Now a savings account is converted to a string such as SavingsAccount[balance=10000][interestRate=5]. The brackets show which fields belong to the superclass.

10.8.2 Overriding the `equals` Method

The equals method is called whenever you want to compare whether two objects have the same contents:

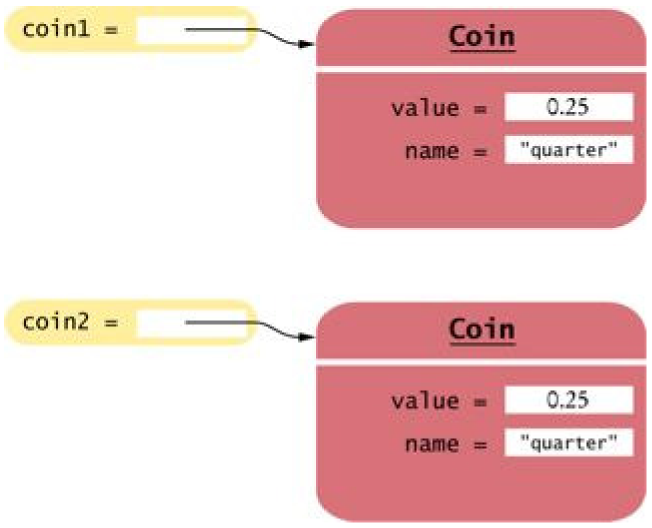
```
if (coin1.equals(coin2)) . . .
   // Contents are the same—see Figure 9
```

Define the equals method to test whether two objects have equal state.

This is different from the test with the == operator, which tests whether the two references are to the *same object*:

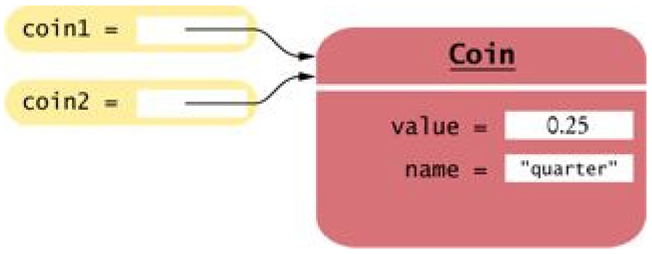
```
if (coin1 == coin2) . . .
   // Objects are the same—see Figure 10
```

Figure 9



Two References to Equal Objects

Figure 10



Two References to the Same Object

Let us implement the `equals` method for the `Coin` class. You need to override the `equals` method of the `Object` class:

```
public class Coin
{
   . . .
   public boolean equals(Object otherObject)
   {
      . . .
   }
   . . .
}
```

Now you have a slight problem. The `Object` class knows nothing about coins, so it defines the `otherObject` parameter of the `equals` method to have the type `Object`. When redefining the method, you are not allowed to change the object signature. Cast the parameter to the class `Coin`:

```
Coin other = (Coin) otherObject;
```

Then you can compare the two coins.

```
public boolean equals(Object otherObject)
{
   Coin other = (Coin) otherObject;
   return name.equals(other.name)
         && value == other.value;
}
```

Note that you must use `equals` to compare object fields, but use == to compare number fields.

When you override the `equals` method, you should also override the `hashCode` method so that equal objects have the same hash code—see Chapter 16 for details.

SELF CHECK

16. Should the call `x.equals(x)` always return `true`?

17. Can you implement `equals` in terms of `toString`? Should you?

469

COMMON ERROR 10.6: Defining the `equals` Method with the Wrong Parameter Type

Consider the following, seemingly simpler, version of the `equals` method for the `Coin` class:

```
public boolean equals(Coin other) // Don't do this!
{
   return name.equals(other.name) && value ==
other.value;
}
```

Here, the parameter of the `equals` method has the type `Coin`, not `Object`.

Unfortunately, this method *does not override* the `equals` method in the `Object` class. Instead, the `Coin` class now has two different `equals` methods:

```
boolean equals(Coin other) // Defined in the Coin class
boolean equals(Object otherObject) // Inherited from the
Object class
```

This is error-prone because the wrong `equals` method can be called. For example, consider these variable definitions:

```
Coin aCoin = new Coin(0.25, "quarter");
Object anObject = new Coin(0.25, "quarter");
```

The call `aCoin.equals (anObject)` calls the second `equals` method, which returns `false`.

The remedy is to ensure that you use the `Object` type for the explicit parameter of the `equals` method.

ADVANCED TOPIC 10.5: Inheritance and the `equals` Method

You just saw how to write an `equals` method: Cast the `otherObject` parameter to the type of your class, and then compare the fields of the implicit parameter and the other parameter.

But what if someone called `coin1.equals(x)` where `x` wasn't a `Coin` object? Then the bad cast would generate an exception, and the program would die. Therefore, you first want to test whether `otherObject` really is an instance of the `Coin` class. The easiest test would be with the `instanceof` operator. However, that test is not specific enough. It would be possible for `otherObject` to belong to some subclass of `Coin`. To rule out that possibility, you should test whether the two objects belong to the *same class*. If not, return `false`.

```
if (getClass() != otherObject.getClass()) return
false;
```

Moreover, the Java language specification [1] demands that the `equals` method return `false` when `otherObject` is `null`.

Here is an improved version of the `equals` method that takes these two points into account:

```
public boolean equals(Object otherObject)
{
   if (otherObject == null) return false;
   if (getClass() != otherObject.getClass())
      return false;
   Coin other = (Coin) otherObject;
   return name.equals(other.name) && value ==
other.value;
}
```

When you define `equals` in a subclass, you should first call `equals` in the superclass, like this:

```
public CollectibleCoin extends Coin
{
   . . .
   public boolean equals(Object otherObject)
   {
      if (!super.equals(otherObject)) return
false;
      CollectibleCoin other = (CollectibleCoin)
otherObject;
      return year == other.year;
   }
   private int year;
}
```

10.8.3 The `clone` Method

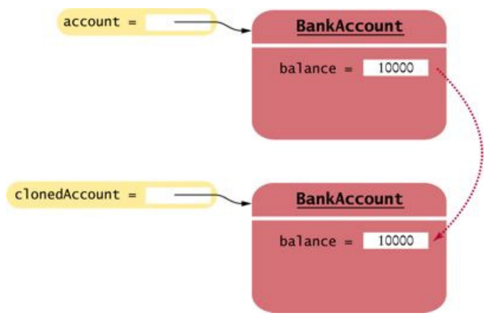
You know that copying an object reference simply gives you two references to the same object:

```
BankAccount account = new BankAccount(1000);
BankAccount account2 = account;
account2.deposit(500);
   // Now both account and account2 refer to a bank
account with a balance of 1500
```

What can you do if you actually want to make a copy of an object? That is the purpose of the `clone` method. The `clone` method must return a *new* object that has an identical state to the existing object (see Figure 11).

The `clone` method makes a new object with the same state as an existing object.

Figure 11



Cloning Objects

Implementing the `clone` method is quite a bit more difficult than implementing the `toString` or `equals` methods—see Advanced Topic 10.6 for details.

Let us suppose that someone has implemented the `clone` method for the `BankAccount` class. Here is how to call it:

```
BankAccount clonedAccount = (BankAccount)
account.clone();
```

The return type of the `clone` method is the class `Object`. When you call the method, you must use a cast to convince the compiler that `account.clone()` really has the same type as `clonedAccount`.

Common Error 10.7: Forgetting to Clone

In Java, object fields contain references to objects, not actual objects. This can be convenient for giving *two names to the same object*:

```
BankAccount harrysChecking = new BankAccount();
BankAccount slushFund = harrysChecking;
    // Use Harry's checking account for the slush fund
slushFund.deposit(80000)
    // A lot of money ends up in Harry's checking account
```

However, if you don't intend two references to refer to the same object, then this is a problem. In that case, you should use the `clone` method:

```
BankAccount slushFund = (BankAccount)
harrysChecking.clone();
```

Quality Tip 10.1: Clone Mutable Instance Fields in Accessor Methods

Consider the following class:

```
public class Customer
{
   public Customer(String aName)
   {
      name = aName;
      account = new BankAccount();
```

```
        }
        public String getName()
        {
           return name;
        }
        public BankAccount getAccount();
        {
           return account;
        }
        private String name;
        private BankAccount account;
     }
```

This class looks very boring and normal, but the `getAccount` method has a curious property. It *breaks encapsulation*, because anyone can modify the object state without going through the public interface:

```
     Customer harry = new Customer("Harry Handsome");
     BankAccount account = harry.getAccount();
        // Anyone can withdraw money!
     account.withdraw(100000);
```

Maybe that wasn't what the designers of the class had in mind? Maybe they wanted class users only to inspect the account? In such a situation, you should *clone* the object reference:

```
     public BankAccount getAccount();
     {
        return (BankAccount) account.clone();
     }
```

Do you also need to clone the `getName` method? No—that method returns a string, and strings are immutable. It is safe to give out a reference to an immutable object.

ADVANCED TOPIC 10.6: Implementing the `clone` Method

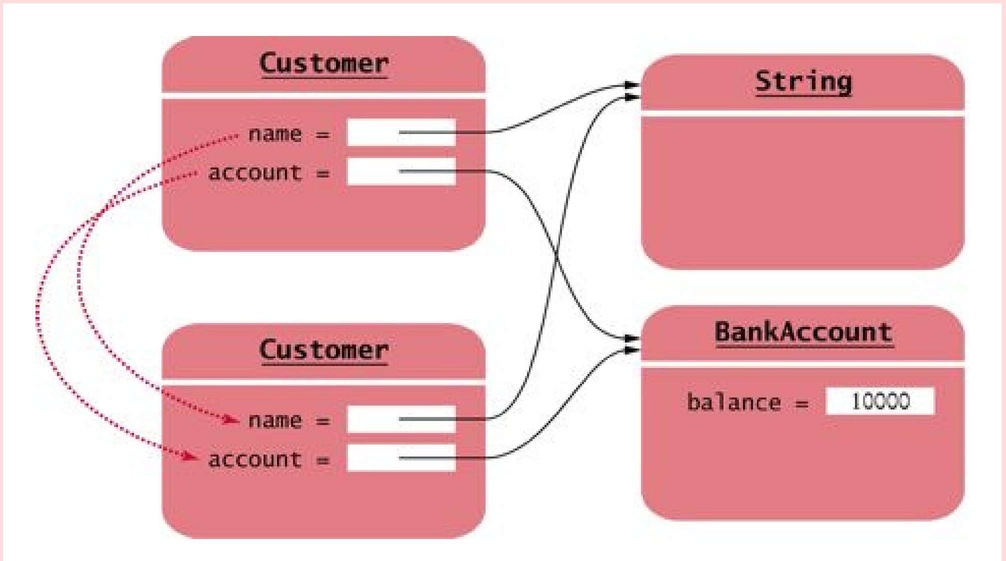
The `Object.clone` method is the starting point for the `clone` methods in your own classes. It creates a new object of the same type as the original object. It also automatically copies the instance fields from the original object to the cloned object. Here is a first attempt to implement the `clone` method for the `BankAccount` class:

```
public class BankAccount
{
   . . .
   public Object clone()
   {
      // Not complete
      Object clonedAccount = super.clone();
      return clonedAccount;
   }
}
```

However, this `Object.clone` method must be used with care. It only shifts the problem of cloning by one level; it does not completely solve it. Specifically, if an object contains a reference to another object, then the `Object.clone` method makes a copy of that object reference, not a clone of that object. The figure below shows how the `Object.clone` method works with a `Customer` object that has references to a `String` object and a `BankAccount` object. As you can see, the `Object.clone` method copies the references to the cloned `Customer` object and does not clone the objects to which they refer. Such a copy is called a *shallow copy*.

473

474



The `Object.clone` Method Makes a Shallow Copy

There is a reason why the `Object.clone` method does not systematically clone all sub-objects. In some situations, it is unnecessary. For example, if an object contains a reference to a string, there is no harm in copying the string reference, because Java string objects can never change their contents. The `Object.clone` method does the right thing if an object contains only numbers, Boolean values, and strings. But it must be used with caution when an object contains references to other objects.

For that reason, there are two safeguards built into the `Object.clone` method to ensure that it is not used accidentally. First, the method is declared `protected` (see Advanced Topic 10.3). This prevents you from accidentally calling `x.clone()` if the class to which `x` belongs hasn't redefined `clone` to be public.

As a second precaution, `Object.clone` checks that the object being cloned implements the `Cloneable` interface. If not, it throws an exception. The `Object.clone` method looks like this:

```
public class Object
{
   protected Object clone()
         throws CloneNotSupportedException
   {
      if (this instanceof Cloneable)
      {
         // Copy the instance fields
         . . .
      }
      else
         throw new CloneNotSupportedException();
   }
}
```

Unfortunately, all that safeguarding means that the legitimate callers of `Object.clone()` pay a price—they must catch that exception *even if their class implements* `Cloneable`.

```
public class BankAccount implements Cloneable
{
```

```
   . . .
   public Object clone()
```

```
      {
         try
         {
            return super.clone();
         }
         catch (CloneNotSupportedException e)
         {
            // Can’t happen because we implement Cloneabl e but
   we still must catch it.
            return null;
         }
      }
   }
```

If an object contains a reference to another mutable object, then you must call `clone` for that reference. For example, suppose the `Customer` class has an instance field of class `BankAccount`. You can implement `Customer.clone` as follows:

```
   public class Customer implements Cloneable
   {
      . . .
      public Object clone()
      {
         try
         {
            Customer cloned = (Customer)
   super.clone();
            cloned.account = (BankAccount)
   account.clone();
            return cloned;
         }
         catch(CloneNotSupportedException e)
         {
            // Can’t happen because we implement Cloneabl e
            return null;
         }
      }
      private String name;
      private BankAccount account;
   }
```

ADVANCED TOPIC 10.7: Enumerated Types Revisited

In Advanced Topic 5.3, we introduced the concept of an enumerated type: a type with a finite number of values. An example is

```
    public enum FilingStatus { SINGLE, MARRIED }
```

In Java, enumerated types are classes with special properties. They have a finite number of instances, namely the objects defined inside the braces. For example, there are exactly two objects of the `FilingStatus` class: `FilingStatus.SINGLE` and `FilingStatus.MARRIED`. Since `FilingStatus` has no public constructor, it is impossible to construct additional objects.

Enumeration classes extend the `Enum` class, from which they inherit `toString` and `clone` methods. The `toString` method returns a string that equals the object's name. For example, `FilingStatus.SINGLE.toString()` returns "`SINGLE`". The `clone` method returns the given object *without making a copy*. After all, it should not be possible to generate new objects of an enumeration class.

The `Enum` class inherits the `equals` method from its superclass, `Object`. Thus, two enumeration constants are only considered equal when they are identical.

You can add your own methods and constructors to an enumeration class, for example

```
    public enum CoinType
    {
        PENNY(0.01), NICKEL(0.05), DIME(0.1),
    QUARTER(0.25);
        CoinType(double aValue) { value = aValue; }
        public double getValue() { return value; }
        private double value;
    }
```

This `CoinType` class has exactly four instances: `CoinType.PENNY`, `CoinType.NICKEL`, `CoinType.DIME`, and `CoinType.QUARTER`. If you have one of these four `CoinType` objects, you can apply the `getValue` method to obtain the coin's value.

Note that there is a major philosophical difference between this `CoinType` class and the `Coin` class that we have discussed elsewhere in this chapter. A `Coin` object represents a particular coin. You can construct as many `Coin` objects as you like. Different `Coin` objects can be equal to another. We consider two `Coin` objects equal when their names and values match. However, `CoinType` describes a type of coins, not an individual coin. The four `CoinType` objects are distinct from each other.

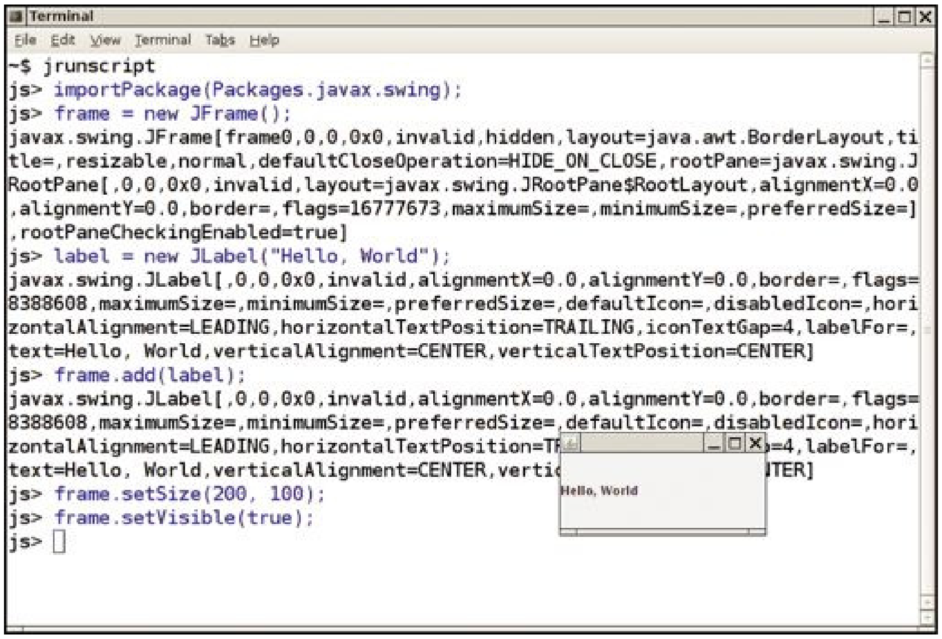
RANDOM FACT 10.1: Scripting Languages

Suppose you work for an office where you must help with the bookkeeping. Suppose that every sales person sends in a weekly spreadsheet with sales figures. One of your jobs is to copy and paste the individual figures into a master spreadsheet and then copy and paste the totals into a word processor document that gets e-mailed to several managers. This kind of repetitive work can be intensely boring. Can you automate it?

It would be a real challenge to write a Java program that can help you—you'd have to know how to read a spreadsheet file, how to format a word processor document, and how to send e-mail.

Fortunately, many office software packages include *scripting languages*. These are programming languages that are integrated with the software for the purpose of automating repetitive tasks. The best-known of these scripting languages is Visual Basic Script, which is a part of the Microsoft Office suite. The Macintosh operating system has a language called AppleScript for the same purpose.

In addition, scripting languages are available for many other purposes. JavaScript is used for web pages. (There is no relationship between Java and JavaScript—the name JavaScript was chosen for marketing reasons.) Tcl (short 476
for "tool control language" and pronounced "tickle") is an open source scripting 477
language that has been ported to many platforms and is often used for scripting software test procedures. Shell scripts are used for automating software configuration, backup procedures, and other system administration tasks.



```
~$ jrunscript
js> importPackage(Packages.javax.swing);
js> frame = new JFrame();
javax.swing.JFrame[frame0,0,0,0x0,invalid,hidden,layout=java.awt.BorderLayout,ti
tle=,resizable,normal,defaultCloseOperation=HIDE_ON_CLOSE,rootPane=javax.swing.J
RootPane[,0,0,0x0,invalid,layout=javax.swing.JRootPane$RootLayout,alignmentX=0.0
,alignmentY=0.0,border=,flags=16777673,maximumSize=,minimumSize=,preferredSize=]
,rootPaneCheckingEnabled=true]
js> label = new JLabel("Hello, World");
javax.swing.JLabel[,0,0,0x0,invalid,alignmentX=0.0,alignmentY=0.0,border=,flags=
8388608,maximumSize=,minimumSize=,preferredSize=,defaultIcon=,disabledIcon=,hori
zontalAlignment=LEADING,horizontalTextPosition=TRAILING,iconTextGap=4,labelFor=,
text=Hello, World,verticalAlignment=CENTER,verticalTextPosition=CENTER]
js> frame.add(label);
javax.swing.JLabel[,0,0,0x0,invalid,alignmentX=0.0,alignmentY=0.0,border=,flags=
8388608,maximumSize=,minimumSize=,preferredSize=,defaultIcon=,disabledIcon=,hori
zontalAlignment=LEADING,horizontalTextPosition=TR                =4,labelFor=,
text=Hello, World,verticalAlignment=CENTER,vertic                 TER]
js> frame.setSize(200, 100);
js> frame.setVisible(true);
js> 
```

Scripting Java Classes with JavaScript

Scripting languages have two features that makes them easier to use than full-fledged programming languages such as Java. First, they are *interpreted*. The interpreter program reads each line of program code and executes it immediately without compiling it first. That makes experimenting much more fun—you get immediate feedback. Also, scripting languages are usually *loosely typed*, meaning you don't have to declare the types of variables. Every variable can hold values of any type. For example, the Scripting Java Classes with JavaScript figure shows a scripting session with Rhino, a JavaScript implementation that allows you to manipulate Java objects. The script stores frame and label objects in variables that are declared without types. It then calls methods that are executed immediately, without compilation. The frame pops up as soon as the line with the `setVisible` command is entered. (If you use an earlier version of Java, you can achieve the same effect with the Rhino scripting engine. You can download Rhino from the Mozilla web site [2]). In recent years, authors of computer viruses have discovered how scripting languages simplify their lives. The famous "love bug" is a Visual Basic Script program that is sent as an attachment to an e-mail. The e-mail has an enticing subject line "I love you" and asks the recipient to click on an attachment masquerading as a love letter. In fact, the attachment is a script file that is executed when the user clicks on it. The script creates some damage on the recipient's computer and then, through the power of the scripting language, uses the Outlook e-mail client to mail itself to all addresses found in the address book. Try programming that in Java! By the way, the person suspected of authoring that virus was a student who had submitted a proposal to write a thesis researching how to write such programs. Perhaps not surprisingly, the proposal was rejected by the faculty.

Why do we still need Java if scripting is easy and fun? Scripts often have poor error checking and are difficult to adapt to new circumstances. Scripting languages lack many of the structuring and safety mechanisms (such as classes and type checking by the compiler) that are important for building robust and scalable programs.

10.9 Using Inheritance to Customize Frames

As you add more user interface components to a frame, the frame can get quite complex. Your programs will become easier to understand when you use inheritance for complex frames.

Define a `JFrame` subclass for a complex frame.

Design a subclass of `JFrame`. Store the components as instance fields. Initialize them in the constructor of your subclass. If the initialization code gets complex, simply add some helper methods.

Here, we carry out this process for the investment viewer program in Chapter 9.

```
public class InvestmentFrame extends JFrame
{
   public InvestmentFrame()
   {
      account = new BankAccount(INITIAL_BALANCE);
      // Use instance fields for components
      label = new JLabel("balance: " +
account.getBalance());
      // Use helper methods
      createButton();
```

```
         createPanel();
         setSize(FRAME_WIDTH, FRAME_HEIGHT);
      }
      private void createButton()
      {
         ActionListener listener = new
   AddInterestListener();
         button.addActionListener(listener);
         button = new JButton("Add Interest");
      }
      private void createPanel()
      {
         panel = new JPanel();
         panel.add(button);
         panel.add(label);
         add(panel);
      }
      private JButton button;
      private JLabel label;
      private JPanel panel;
      private BankAccount account;
   }
```

This approach differs from the programs in Chapter 9. In those programs, we simply configured the frame in the `main` method of a viewer class.

It is a bit more work to provide a separate class for the frame. However, the frame class makes it easier to organize the code that constructs the user-interface elements. We will use this approach for all examples in this chapter.

Of course, we still need a class with a `main` method:

```
public class InvestmentViewer2
{
   public static void main(String[] args)
   {
      JFrame frame = new InvestmentFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)
      frame.setVisible(true);
   }
}
```

SELF CHECK

18. How many Java source files are required by the investment viewer application when we use inheritance to define the frame class?

19. Why does the `InvestmentFrame` constructor call `setSize(FRAME_WIDTH, FRAME_HEIGHT)`, whereas the `main` method of the investment viewer class in Chapter 9 called `frame.setSize(FRAME_WIDTH, FRAME_HEIGHT)`?

ADVANCED TOPIC 10.8: Adding the `main` Method to the Frame Class

Have another look at the `InvestmentFrame` and `InvestmentViewer2` classes. Some programmers prefer to combine these two classes, by adding the `main` method to the frame class:

```
public class InvestmentFrame extends JFrame
{
   public static void main(String[] args)
   {
      JFrame frame = new InvestmentFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE
      frame.setVisible(true);
   }
   public InvestmentFrame()
   {
      account = new BankAccount(INITIAL_BALANCE);
      // Use instance fields for components
      label = new JLabel("balance: " +
account.getBalance());
```

479

480

```
      // Use helper methods
      createButton();
      createPanel();
      setSize(FRAME_WIDTH, FRAME_HEIGHT);
   }
   . . .
}
```

This is a convenient shortcut that you will find in many programs, but it does muddle the responsibilities between the frame class and the program. Therefore, we do not use this approach in this book.

10.10 Processing Text Input

A graphical application can receive text input by calling the `showInputDialog` method of the `JOptionPane` class, but popping up a separate dialog box for each input is not a natural user interface. Most graphical programs collect text input through *text fields* (see Figure 12). In this section, you will learn how to add text fields to a graphical application, and how to read what the user types into them.

The `JTextField` class provides a text field. When you construct a text field, you need to supply the width—the approximate number of characters that you expect the user to type.

```
final int FIELD_WIDTH = 10;
final JTextField rateField = new
JTextField(FIELD_WIDTH);
```

Users can type additional characters, but then a part of the contents of the field becomes invisible.

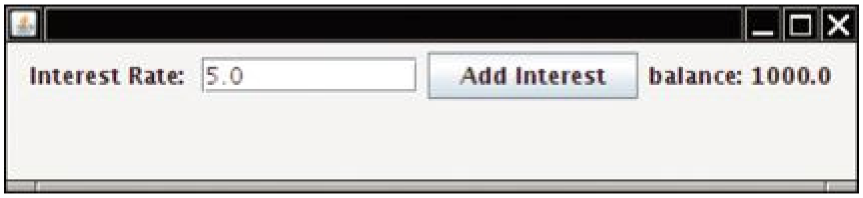
Use `JTextField` components to provide space for user input. Place a `JLabel` next to each text field.

You will want to label each text field so that the user knows what to type into it. Construct a `JLabel` object for each label:

```
JLabel rateLabel = new JLabel("Interest Rate: ");
```

You want to give the user an opportunity to enter all information into the text fields before processing it. Therefore, you should supply a button that the user can press to indicate that the input is ready for processing.

Figure 12



An Application with a Text Field

When that button is clicked, its `actionPerformed` method reads the user input
from the text fields, using the `getText` method of the `JTextField` class. The
`getText` method returns a `String` object. In our sample program, we turn the string into a number, using the `Double.parseDouble` method:

```
class AddInterestListener implements ActionListener
{
   public void actionPerformed(ActionEvent event)
   {
      double rate =
Double.parseDouble(rateField.getText());
      . . .
   }
}
```

The following application is a useful prototype for a graphical user-interface front end for arbitrary calculations. You can easily modify it for your own needs. Place other input components into the frame. Change the contents of the `actionPerformed` method to carry out other calculations. Display the result in a label.

ch10/textfield/InvestmentViewer3.java

```
import javax.swing.JFrame;

/**
   This program displays the growth of an investment.
*/
public class InvestmentViewer3
{
   public static void main(String[] args)
   {
        JFrame frame = new InvestmentFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_
        frame.setVisible(true);
    }
}
```

ch10/textfield/InvestmentFrame.java

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;

/**
    A frame that shows the growth of an investment with variable interest.
 */
public class InvestmentFrame extends JFrame
{
    public InvestmentFrame()
    {
        account = new BankAccount(INITIAL_BALANCE);

        // Use instance fields for components
        resultLabel = new JLabel(“balance: ” + account.getBalance());
```

```

        // Use helper methods
        createTextField();
        createButton();
        createPanel();

        setSize(FRAME_WIDTH, FRAME_HEIGHT);
    }

    private void createTextField()
    {
        rateLabel = new JLabel(“Interest Rate: ”);

```

```
      final int FIELD_WIDTH = 10;
      rateField = new JTextField(FIELD_WIDTH);
      rateField.setText("" + DEFAULT_RATE);
   }

   private void createButton()
   {
      button = new JButton("Add Interest");

      class AddInterestListener implements ActionListener
      {
         public void actionPerformed(ActionEvent event)
         {
            double rate = Double.parseDouble(
                  rateField.getText());
            double interest = account.getBalance()
                  * rate / 100;
            account.deposit(interest);
            resultLabel.setText(
                  "balance: " + account.getBalance());
         }
      }

      ActionListener listener = new AddInterestListener();
      button.addActionListener(listener);
   }

   private void createPanel()
   {
      panel = new JPanel();
      panel.add(rateLabel);
      panel.add(rateField);
      panel.add(button);
      panel.add(resultLabel);
      add(panel);
   }

   private JLabel rateLabel;
   private JTextField rateField;
   private JButton button;
```

```
    private JLabel resultLabel;
    private JPanel panel;
    private BankAccount account;

    private static final int FRAME_WIDTH = 500;
    private static final int FRAME_HEIGHT = 200;

    private static final double DEFAULT_RATE = 5;
    private static final double INITIAL_BALANCE = 1000;
}
```

SELF CHECK

20. What happens if you omit the first `JLabel` object?

21. If a text field holds an integer, what expression do you use to read its contents?

10.11 Text Areas

In Section 10.10, you saw how to construct text fields. A text field holds a single line of text. To display multiple lines of text, use the `JTextArea` class.

Use a `JTextArea` to show multiple lines of text.

When constructing a text area, you can specify the number of rows and columns:

```
final int ROWS = 10;
final int COLUMNS = 30;
JTextArea textArea = new JTextArea(ROWS, COLUMNS);
```

Use the `setText` method to set the text of a text field or text area. The `append` method adds text to the end of a text area. Use newline characters to separate lines, like this:

```
textArea.append(account.getBalance() + "\n");
```

If you want to use a text field or text area for display purposes only, call the `setEditable` method like this

```
textArea.setEditable(false);
```

Now the user can no longer edit the contents of the field, but your program can still call `setText` and `append` to change it.

As shown in Figure 4, the `JTextField` and `JTextArea` classes are subclasses of the class `JTextComponent`. The methods `setText` and `setEditable` are defined in the `JTextComponent` class and inherited by `JTextField` and `JTextArea`. However, the `append` method is defined in the `JTextArea` class.

To add scroll bars to a text area, use a `JScrollPane`, like this:

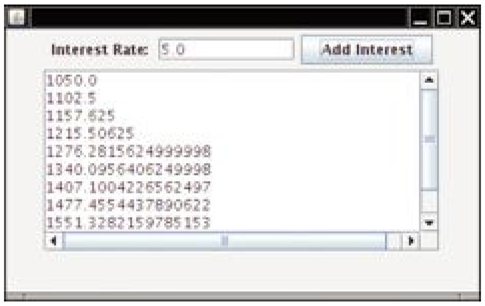
```
JTextArea textArea = new JTextArea(ROWS, COLUMNS);
JScrollPane scrollPane = new JScrollPane(textArea);
```

You can add scroll bars to any component with a `JScrollPane`.

Then add the scroll pane to the panel. Figure 13 shows the result.

The following sample program puts these concepts together. A user can enter numbers into the interest rate text field and then click on the "Add Interest" button). 483
The interest rate is applied, and the updated balance is appended to the text area. The text area has scroll bars and is not editable. 484

Figure 13



```
1050.0
1102.5
1157.625
1215.50625
1276.2815624999998
1340.0956406249998
1407.1004226562497
1477.4554437890622
1551.3282159785153
```

The Investment Application with a Text Area

This program is similar to the previous investment viewer program, but it keeps track of all the bank balances, not just the last one.

ch10/textarea/InvestmentFrame.java

```java
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;

/**
   A frame that shows the growth of an investment with variable interest.
*/
public class InvestmentFrame extends JFrame
{
   public InvestmentFrame()
   {
      account = new BankAccount(INITIAL_BALANCE);
      resultArea = new JTextArea(AREA_ROWS, AREA_COLUMNS);
      resultArea.setEditable(false);

      // Use helper methods
      createTextField();
      createButton();
      createPanel();

      setSize(FRAME_WIDTH, FRAME_HEIGHT);
   }

   private void createTextField()
   {
      rateLabel = new JLabel("Interest Rate: ");

      final int FIELD_WIDTH = 10;
      rateField = new JTextField(FIELD_WIDTH);
      rateField.setText("" + DEFAULT_RATE);
   }
```

```

    private void createButton()
    {
        button = new JButton(“Add Interest”);

        class AddInterestListener implements ActionListener
        {
            public void actionPerformed(ActionEvent event)
            {
                double rate = Double.parseDouble(
                     rateField.getText());
                double interest = account.getBalance()
                     * rate / 100;
                account.deposit(interest);
                resultArea.append(account.getBalance() + “\n”);
            }
        }

        ActionListener listener = new AddInterestListener();
        button.addActionListener(listener);
    }

    private void createPanel()
    {
        panel = new JPanel();
        panel.add(rateLabel);
        panel.add(rateField);
        panel.add(button);
        JScrollPane scrollPane = new JScrollPane(resultArea);
        panel.add(scrollPane);
        add(panel);
    }

    private JLabel rateLabel;
    private JTextField rateField;
    private JButton button;
    private JTextArea resultArea;
    private JPanel panel;
    private BankAccount account;

    private static final int FRAME_WIDTH = 400;
    private static final int FRAME_HEIGHT = 250;

    private static final int AREA_ROWS = 10;
    private static final int AREA_COLUMNS = 30;

    private static final double DEFAULT_RATE =
5;
    private static final double INITIAL_BALANCE
= 1000;
 }
```

Self Check

22. What is the difference between a text field and a text area?

23. Why did the `InvestmentFrame` program call `resultArea.setEditable(false)`?

24. How would you modify the `InvestmentFrame` program if you didn't want to use scroll bars?

How To 10.1: Implementing a Graphical User Interface (GUI)

A GUI program allows users to supply inputs and specify actions. The `InvestmentViewer3` program has only one input and one action. More sophisticated programs have more interesting user interactions, but the basic principles are the same.

Step 1 Enumerate the actions that your program needs to carry out.

For example, the investment viewer has a single action, to add interest. Other programs may have different actions, perhaps for making deposits, inserting coins, and so on.

Step 2 For each action, enumerate the inputs that you need.

For example, the investment viewer has a single input: the interest rate. Other programs may have different inputs, such as amounts of money, product quantities, and so on.

Step 3 For each action, enumerate the outputs that you need to show.

The investment viewer has a single output: the current balance. Other programs may show different quantities, messages, and so on.

Step 4 Supply the user interface components.

Right now, you need to use buttons for actions, text fields for inputs, and labels for outputs. In Chapter 18, you will see many more user-interface components that can be used for actions and inputs. In Chapter 3, you learned how to implement your own components to produce graphical output, such as charts or drawings.

Add the required buttons, text fields, and other components to a frame. In this chapter, you have seen how to lay out very simple user interfaces, by adding all components to a single panel and adding the panel to the frame. Chapter 18 shows you how you can achieve more complex layouts.

Step 5 Supply event handler classes.

For each button, you need to add an object of a listener class. The listener classes must implement the `ActionListener` interface. Supply a class for each action (or group of related actions), and put the instructions for the action in the `actionPerformed` method.

```
class Button1Listener implements ActionListener
{
   public void actionPerformed(ActionEvent event)
   {
      // button1 action goes here
      . . .
   }
}
```

486

487

Remember to declare any local variables accessed by the listener methods as `final`.

Step 6 Make listener objects and attach them to the event sources.

For action events, the event source is a button or other user-interface component, or a timer. You need to add a listener object to each event source, like this:

```
ActionListener listener1 = new Button1Listener();
button1.addActionListener(listener1);
```

COMMON ERROR 10.8: By Default, Components Have Zero Width and Height

The sample GUI programs of this chapter display results in a label or text area. Sometimes, you want to use a graphical component such as a chart. You add the chart component to the panel:

```
panel.add(textField);
panel.add(button);
panel.add(chartComponent);
```

However, the default size for a component is `0` by `0` pixels, and the chart component will not be visible. The remedy is to call the `setPreferredSize` method, like this:

```
chartComponent.setPreferredSize(new
Dimension(CHART_WIDTH, CHART_HEIGHT));
```

GUI components such as buttons and text fields know how to compute their preferred size, but you must set the preferred size of components on which you paint.

PRODUCTIVITY HINT 10.2: Code Reuse

Suppose you are given the task of writing another graphical user-interface program that reads input from a couple of text fields and displays the result of some calculations in a label or text area. You don't have to start from scratch. Instead, you can—and often should—*reuse* the outline of an existing program, such as the foregoing `InvestmentFrame` class.

To reuse program code, simply make a copy of a program file and give the copy a new name. For example, you may want to copy `InvestmentFrame.java` to a file `TaxReturnFrame.java`. Then remove the code that is clearly specific to the old problem, but leave the outline in place. That is, keep the panel, text field, event listener, and so on. Fill in the code for your new calculations. Finally, rename classes, buttons, frame titles, and so on.

Once you understand the principles behind event listeners, frames, and panels, there is no need to rethink them every time. Reusing the structure of a working program makes your work more efficient.

However, reuse by “copy and rename” is still a mechanical and somewhat error-prone approach. It is even better to package reusable program structures into a set of common classes. The inheritance mechanism lets you design classes for reuse without copy and paste.

CHAPTER SUMMARY

1. Inheritance is a mechanism for extending existing classes by adding methods and fields.
2. The more general class is called a superclass. The more specialized class that inherits from the superclass is called the subclass.
3. Every class extends the `Object` class either directly or indirectly.
4. Inheriting from a class differs from implementing an interface: The subclass inherits behavior and state from the superclass.
5. One advantage of inheritance is code reuse.
6. When defining a subclass, you specify added instance fields, added methods, and changed or overridden methods.
7. Sets of classes can form complex inheritance hierarchies.
8. A subclass has no access to private fields of its superclass.
9. Use the `super` keyword to call a method of the superclass.
10. To call the superclass constructor, you use the `super` keyword in the first statement of the subclass constructor.
11. Subclass references can be converted to superclass references.

12. The `instanceof` operator tests whether an object belongs to a particular type.
13. An abstract method is a method whose implementation is not specified.
14. An abstract class is a class that cannot be instantiated.
15. A field or method that is not declared as `public`, `private`, or `protected` can be accessed by all classes in the same package, which is usually not desirable.
16. Protected features can be accessed by all subclasses and all classes in the same package.
17. Define the `toString` method to yield a string that describes the object state.
18. Define the `equals` method to test whether two objects have equal state.
19. The `clone` method makes a new object with the same state as an existing object.
20. Define a `JFrame` subclass for a complex frame.
21. Use `JTextField` components to provide space for user input. Place a `JLabel` next to each text field.
22. Use a `JTextArea` to show multiple lines of text.
23. You can add scroll bars to any component with a `JScrollPane`.

FURTHER READING

1. James Gosling, Bill Joy, Guy Steele, and Gilad Bracha, *The Java Language Specification,* 3rd edition, Addison-Wesley, 2005.

2. http://www.mozilla.org/rhino The Rhino interpreter for the JavaScript language.

CLASSES, OBJECTS, AND METHODS INTRODUCED IN THIS CHAPTER

```
java.awt.Component
   setPreferredSize
java.awt.Dimension
java.lang.Cloneable
java.lang.CloneNotSupportedException
java.lang.Object
   clone
   toString
javax.swing.JTextArea
   append
javax.swing.JTextField
javax.swing.text.JTextComponent
   getText
   isEditable
   setEditable
   setText
```

REVIEW EXERCISES

★ **Exercise R10.1.** What is the balance of `b` after the following operations?

```
SavingsAccount b = new SavingsAccount(10);
b.deposit(5000);
b.withdraw(b.getBalance() / 2);
b.addInterest();
```

★ **Exercise R10.2.** Describe all constructors of the `SavingsAccount` class. List all methods that are inherited from the `BankAccount` class. List all methods that are added to the `SavingsAccount` class.

★★ **Exercise R10.3.** Can you convert a superclass reference into a subclass reference? A subclass reference into a superclass reference? If so, give examples. If not, explain why not.

★★ **Exercise R10.4.** Identify the superclass and the subclass in each of the following pairs of classes.

a. `Employee, Manager`

b. `Polygon, Triangle`

c. `GraduateStudent, Student`

d. `Person, Student`

e. `Employee, GraduateStudent`

f. `BankAccount, CheckingAccount`

g. `Vehicle, Car`

h. `Vehicle, Minivan`

i. `Car, Minivan`

j. `Truck, Vehicle`

★ **Exercise R10.5.** Suppose the class `Sub` extends the class `Sandwich`. Which of the following assignments are legal?

```
Sandwich x = new Sandwich();
Sub y = new Sub();
```

a. `x = y;`

b. `y = x;`

c. `y = new Sandwich();`

d. `x = new Sub();`

★ **Exercise R10.6.** Draw an inheritance diagram that shows the inheritance relationships between the classes:

- `Person`
- `Employee`
- `Student`
- `Instructor`

- `Classroom`
- `Object`

★★ **Exercise R10.7.** In an object-oriented traffic simulation system, we have the following classes:

- `Vehicle`
- `Car`
- `Truck`
- `Sedan`
- `Coupe`
- `PickupTruck`
- `SportUtilityVehicle`
- `Minivan`
- `Bicycle`
- `Motorcycle`

Draw an inheritance diagram that shows the relationships between these classes.

★★ **Exercise R10.8.** What inheritance relationships would you establish among the following classes?

- `Student`
- `Professor`
- `TeachingAssistant`
- `Employee`

- `Secretary`

- `DepartmentChair`
- `Janitor`
- `SeminarSpeaker`
- `Person`
- `Course`
- `Seminar`
- `Lecture`
- `ComputerLab`

★★★ **Exercise R10.9.** Which of these conditions returns `true`? Check the Java documentation for the inheritance patterns.

a. `Rectangle r = new Rectangle(5, 10, 20, 30);`

b. `if (r instanceof Rectangle) …`

c. `if (r instanceof Point) …`

d. `if (r instanceof Rectangle2D.Double) …`

e. `if (r instanceof RectangularShape) …`

f. `if (r instanceof Object) …`

g. `if (r instanceof Shape) …`

★★ **Exercise R10.10.** Explain the two meanings of the `super` keyword. Explain the two meanings of the `this` keyword. How are they related?

★★★ **Exercise R10.11.** (Tricky.) Consider the two calls

```
public class D extends B
{
   public void f()
   {
      this.g(); // 1
```

```
    }
    public void g()
    {
       super.g(); // 2
    }
    . . .
}
```

Which of them is an example of polymorphism?

★★★ **Exercise R10.12.** Consider this program:

```
public class AccountPrinter
{
   public static void main(String[] args)
   {
      SavingsAccount momsSavings
            = new SavingsAccount(0.5);
```

```
      CheckingAccount harrysChecking
            = new CheckingAccount(0);
      . . .
      endOfMonth(momsSavings);
      endOfMonth(harrysChecking);
      printBalance(momsSavings);
      printBalance(harrysChecking);
   }
   public static void endOfMonth(SavingsAccount
savings)
   {
      savings.addInterest();
   }
   public static void endOfMonth(CheckingAccount
checking)
   {
      checking.deductFees();
   }
   public static void printBalance(BankAccount
account)
   {
      System.out.println("The balance is $"
            + account.getBalance());
   }
}
```

Are the calls to the `endOfMonth` methods resolved by early binding or late binding? Inside the `printBalance` method, is the call to `getBalance` resolved by early binding or late binding?

★ **Exercise R10.13.** Explain the terms *shallow copy* and *deep copy*.

★ **Exercise R10.14.** What access attribute should instance fields have? What access attribute should static fields have? How about static final fields?

★ **Exercise R10.15.** What access attribute should instance methods have? Does the same hold for static methods?

★★ **Exercise R10.16.** The fields `System.in` and `System.out` are static public fields. Is it possible to overwrite them? If so, how?

★★ **Exercise R10.17.** Why are public fields dangerous? Are public static fields more dangerous than public instance fields?

★G **Exercise R10.18.** What is the difference between a label, a text field, and a text area?

★★G **Exercise R10.19.** Name a method that is defined in `JTextArea`, a method that `JTextArea` inherits from `JTextComponent`, and a method that `JTextArea` inherits from `JComponent`.

Additional review exercises are available in WileyPLUS.

PROGRAMMING EXERCISES

★ **Exercise P10.1.** Enhance the `addInterest` method of the `SavingsAccount` class to compute the interest on the *minimum* balance since the last call to `addInterest`. *Hint*: You need to modify the `withdraw` method as well, and you need to add an instance field to remember the minimum balance.

★★ **Exercise P10.2.** Add a `TimeDepositAccount` class to the bank account hierarchy. The time deposit account is just like a savings account, but you promise to leave the money in the account for a particular number of months, and there is a penalty for early withdrawal. Construct the account with the interest rate and the number of months to maturity. In the `addInterest` method, decrement the count of months. If the count is positive during a withdrawal, charge the withdrawal penalty.

★ **Exercise P10.3.** Implement a subclass `Square` that extends the `Rectangle` class. In the constructor, accept the *x*- and *y*-positions of the *center* and the side length of the square. Call the `setLocation` and `setSize` methods of the `Rectangle` class. Look up these methods in the documentation for the `Rectangle` class. Also supply a method `getArea` that computes and returns the area of the square. Write a sample program that asks for the center and side length, then prints out the square (using the `toString` method that you inherit from `Rectangle`) and the area of the square.

★ **Exercise P10.4.** Implement a superclass `Person`. Make two classes, `Student` and `Instructor`, that inherit from `Person`. A person has a name and a year of birth. A student has a major, and an instructor has a salary. Write the class definitions, the constructors, and the methods `toString` for all classes. Supply a test program that tests these classes and methods.

★★ **Exercise P10.5.** Make a class `Employee` with a name and salary. Make a class `Manager` inherit from `Employee`. Add an instance field, named `department`, of type `String`. Supply a method `toString` that prints the manager's name, department, and salary. Make a class `Executive` inherit from `Manager`. Supply appropriate `toString` methods for all classes. Supply a test program that tests these classes and methods.

★ **Exercise P10.6.** Write a superclass `Worker` and subclasses `HourlyWorker` and `Salaried-Worker`. Every worker has a name and a salary rate. Write a method `computePay(int hours)` that computes the weekly pay for every worker. An hourly worker gets paid the hourly wage for the actual number of hours worked, if `hours` is at most 40. If the hourly worker worked more than 40 hours, the excess is paid at time and a half. The salaried worker gets paid the hourly wage for 40 hours, no matter what the actual number of hours is. Supply a test program that uses polymorphism to test these classes and methods.

★★★ **Exercise P10.7.** Reorganize the bank account classes as follows. In the `BankAccount` class, introduce an abstract method `endOfMonth` with no implementation. Rename the `addInterest` and `deductFees` methods into `endOfMonth` in the subclasses. Which classes are now abstract and which are concrete? Write a static method `void test(BankAccount account)` that makes five transactions and then calls `endOfMonth`. Test it with instances of all concrete account classes.

★★★G **Exercise P10.8.** Implement an abstract class `Vehicle` and concrete subclasses `Car` and `Truck`. A vehicle has a position on the screen. Write methods `draw` that draw cars and trucks as follows:

Then write a method `randomVehicle` that randomly generates `Vehicle` references, with an equal probability for constructing cars and trucks, with random positions. Call it 10 times and draw all of them.

★G **Exercise P10.9.** Write a graphical application front end for a bank account class. Supply text fields and buttons for depositing and withdrawing money, and for displaying the current balance in a label.

★G **Exercise P10.10.** Write a graphical application front end for an `Earthquake` class. Supply a text field and button for entering the strength of the earthquake. Display the earthquake description in a label.

★G **Exercise P10.11.** Write a graphical application front end for a `DataSet` class. Supply text fields and buttons for adding floating-point values, and display the current minimum, maximum, and average in a label.

★G **Exercise P10.12.** Write an application with three labeled text fields, one each for the initial amount of a savings account, the annual interest rate, and the number of years. Add a button "Calculate" and a read-only text area to display the result, namely, the balance of the savings account after the end of each year.

★★G **Exercise P10.13.** In the application from Exercise P10.12, replace the text area with a bar chart that shows the balance after the end of each year.

★★★G **Exercise P10.14.** Write a program that contains a text field, a button "Add Value", and a component that draws a bar chart of the numbers that a user typed into the text field.

★★G **Exercise P10.15.** Write a program that prompts the user for an integer and then draws as many rectangles at random positions in a component as the user requested.

★G **Exercise P10.16.** Write a program that prompts the user to enter the *x*- and *y*-positions of the center and a radius. When the user clicks a "Draw" button, draw a circle with that center and radius in a component.

★★G **Exercise P10.17.** Write a program that allows the user to specify a circle by typing the radius in a text field and then clicking on the center. Note that you don't need a "Draw" button.

★★★G **Exercise P10.18.** Write a program that allows the user to specify a circle with two mouse presses, the first one on the center and the 494
second on a point on the periphery. *Hint*: In the mouse press handler, 495
you must keep track of whether you already received the center point in a previous mouse press.

★★★G **Exercise P10.19.** Write a program that draws a clock face with a time that the user enters in two text fields (one for the hours, one for the minutes).

Hint: You need to determine the angles of the hour hand and the minute hand. The angle of the minute hand is easy: The minute hand travels 360 degrees in 60 minutes. The angle of the hour hand is harder; it travels 360 degrees in 12×60 *minutes*.

★★G **Exercise P10.20.** Write a program that asks the user to enter an integer *n*, and then draws an *n*-by-*n* grid.

- Additional programming exercises are available in WileyPLUS.

PROGRAMMING PROJECTS

★★★ **Project 10.1.** Your task is to program robots with varying behaviors. The robots try to escape a maze, such as the following:

```
* *******
*     * *
* ***** *
* * *   *
* * *** *
*   *   *
*** * * *
*     * *
******* *
```

A robot has a position and a method `void move (Maze m)` that modifies the position. Provide a common superclass `Robot` whose `move` method does nothing. Provide subclasses `RandomRobot`, `RightHandRuleRobot`, and `MemoryRobot`. Each of these robots has a different strategy for escaping. The `RandomRobot` simply makes random moves. The `RightHandRuleRobot` moves around the maze so that it's right hand always touches a wall. The `MemoryRobot` remembers all positions that it has previously occupied and never goes back to a position that it knows to be a dead end.

★★★ **Project 10.2.** Implement the `toString`, `equals`, and `clone` methods for all subclasses of the `BankAccount` class, as well as the `Bank` class of Chapter 7. Write unit tests that verify that your methods work correctly. Be sure to test a `Bank` that holds objects from a mixture of account classes.

ANSWERS TO SELF-CHECK QUESTIONS

1. Two instance fields: `balance` and `interestRate`.
2. `deposit`, `withdraw`, `getBalance`, and `addInterest`.
3. `Manager` is the subclass; `Employee` is the superclass.
4. To express the common behavior of text fields and text components.

5. We need a counter that counts the number of withdrawals and deposits.

6. It needs to reduce the balance, and it cannot access the `balance` field directly.

7. So that the count can reflect the number of transactions for the following month.

8. It was content to use the default constructor of the superclass, which sets the balance to zero.

9. No—this is a requirement only for constructors. For example, the `Checking-Account.deposit` method first increments the transaction count, then calls the superclass method.

10. We want to use the method for all kinds of bank accounts. Had we used a parameter of type `SavingsAccount`, we couldn't have called the method with a `CheckingAccount` object.

11. We cannot invoke the `deposit` method on a variable of type `Object`.

12. The object is an instance of `BankAccount` or one of its subclasses.

13. The balance of `a` is unchanged, and the transaction count is incremented twice.

14. Accidentially forgetting the `private` modifer.

15. Any methods of classes in the same package.

16. It certainly should—unless, of course, `x` is `null`.

17. If `toString` returns a string that describes all instance fields, you can simply call `toString` on the implicit and explicit parameters, and compare the results. However, comparing the fields is more efficient than converting them into strings.

18. Three: `InvestmentFrameViewer`, `InvestmentFrame`, and `BankAccount`.

19. The `InvestmentFrame` constructor adds the panel to *itself*.

20. Then the text field is not labeled, and the user will not know its purpose.

21. `Integer.parseInt(textField.getText())`

22. A text field holds a single line of text; a text area holds multiple lines.

23. The text area is intended to display the program output. It does not collect user input.

24. Don't construct a `JScrollPane` and add the `resultArea` object directly to the frame.