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.
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 1 shows a typical example.

In Java it is equally common to group classes in *inheritance hierarchies*. The classes representing the most general concepts are near the root, more specialized classes towards the branches. For example, Figure 2 shows part of the hierarchy of Swing user-interface components in Java.

We must introduce some more terminology for expressing the relationship between the classes in an inheritance hierarchy. The more general class is called the *superclass*. The more specialized class that inherits from the superclass is called the *subclass*. In our example, `JPanel` is a subclass of `JComponent`.

Figure 2 uses the UML notation for inheritance. In a class diagram, you denote inheritance by a solid arrow with a “hollow triangle” tip that points to the superclass.

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 placed 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 variables to store them. The individual button classes (such
Chapter 10  Inheritance

as JButton, JRadioButton, and JCheckBox) inherit these properties. In fact, the AbstractButton class was created to express the commonality among these buttons.

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 3 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 overridden to count the transactions.

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**

1. What is the purpose of the JTextComponent class in Figure 2?

2. Why don’t we place the addInterest method in the BankAccount class?
In this section, we begin building the inheritance hierarchy of bank account classes. You will learn how to form a subclass from a given superclass. Let’s start with the SavingsAccount class. Here is the syntax for the class declaration:

```java
public class SavingsAccount extends BankAccount
{
    added instance variables
    new methods
}
```

In the SavingsAccount class declaration you specify only new methods and instance variables. The SavingsAccount class automatically inherits the methods of the BankAccount class. For example, the deposit method automatically applies to savings accounts:

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

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 variable must be declared in the subclass.

```java
public class SavingsAccount extends BankAccount
{
    private double interestRate;

    public SavingsAccount(double rate)
    {
        Constructor implementation
    }

    public void addInterest()
    {
        Method implementation
    }
}
```

A subclass object automatically has the instance variables declared in the superclass. For example, a SavingsAccount object has an instance variable balance that was declared in the BankAccount class.

Any new instance variables that you declare in the subclass are present only in subclass objects. For example, every SavingsAccount object has an instance variable interestRate. Figure 4 shows the layout of a SavingsAccount object.
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.

```java
public class SavingsAccount extends BankAccount {
    private double interestRate;

    public SavingsAccount(double rate) {
        interestRate = rate;
    }

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

The *addInterest* method calls the *getBalance* and *deposit* methods rather than directly updating the balance variable of the superclass. This is a consequence of *encapsulation*. The balance variable was declared as private in the *BankAccount* class. The *addInterest* method is declared in the *SavingsAccount* class. It does not have the right to access a private instance variable of another class.

Note how the *addInterest* method calls the inherited *getBalance* and *deposit* methods without specifying an implicit parameter. This means that the calls apply to the implicit parameter of the *addInterest* method.

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

```java
double interest = this.getBalance() * this.interestRate / 100;
this.deposit(interest);
```
This completes the implementation of the SavingsAccount class. You will find the complete source code below.

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

**ch10/accounts/SavingsAccount.java**

```java
/**
 * An account that earns interest at a fixed rate.
 */
public class SavingsAccount extends BankAccount {
    private double interestRate;

    /**
     * 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);
    }
}
```

**SELF CHECK**

3. Which instance variables does an object of class SavingsAccount have?
4. Name four methods that you can apply to SavingsAccount objects.
5. 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 reserved word extends suggests that the SavingsAccount object is an extended version of a BankAccount.
- The SavingsAccount object is larger; it has an added instance variable 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.

**Common Error 10.2**

Shadowing Instance Variables

A subclass has no access to the private instance variables of the superclass. For example, the methods of the SavingsAccount class cannot access the balance instance variable:

```java
public class SavingsAccount extends BankAccount {
    public void addInterest() {
        double interest = getBalance() * interestRate / 100;
        balance = balance + interest; // Error
    }
    ...
}
```

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

```java
public class SavingsAccount extends BankAccount {
    private double balance; // Don’t
    ...
    public void addInterest() {
        double interest = getBalance() * interestRate / 100;
        balance = balance + interest; // Compiles but doesn’t update the correct balance
    }
}
```

Sure, now the addInterest method compiles, but it doesn’t update the correct balance! Such a SavingsAccount object has two instance variables, both named balance (see Figure 5). The getBalance method of the superclass retrieves one of them, and the addInterest method of the subclass updates the other.

![Figure 5 Shadowing Instance Variables](image)
10.3 Overriding Methods

A subclass method **overrides** a superclass method if it has the same name and parameter types as a superclass method. When such a method is applied to a subclass object, the overriding method, and not the original method, is executed.

We turn to the CheckingAccount class for an example of overriding methods. Recall that the BankAccount class has three methods:

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

The CheckingAccount class declares these methods:

```java
public class CheckingAccount extends BankAccount {
    . . .
    public void deposit(double amount) { . . . }
    public void withdraw(double amount) { . . . }
    public void deductFees() { . . . }
}
```

The deposit and withdraw methods of the CheckingAccount class override the deposit and withdraw methods of the BankAccount class to handle transaction fees. However, the deductFees method does not override another method, and the getBalance method is not overridden.

Let’s implement the deposit method of the CheckingAccount class. It increments the transaction count and deposits the money:

```java
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:

```java
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 variable, that instance variable 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 instance variable, you must use a public method of the 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:

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

But this won’t quite work. The compiler interprets

```java
super.deposit(amount);
```

as

```java
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).

Instead, we must be specific that we want to invoke only the superclass’s deposit method. There is a special reserved word super for this purpose:

```java
public class CheckingAccount extends BankAccount
{
    public void deposit(double amount)
    {
        super.deposit(amount);
    }
    ...
}
```

### Syntax 10.2 Calling a Superclass Method

```java
Syntax super.methodName(parameters);
```

**Example**

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

*Calls the method of the superclass instead of the method of the current class.*

*If you omit super, this method calls itself.*

See page 378.
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 of the CheckingAccount class also invoke a superclass method.

```java
public class CheckingAccount extends BankAccount {
    private static final int FREE_TRANSACTIONS = 3;
    private static final double TRANSACTION_FEE = 2.0;

    private int transactionCount;
    . . .
    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;
    }
    . . .
}
```

**SELF CHECK**

6. Categorize the methods of the SavingsAccount class as inherited, new, and overridden.

7. Why does the withdraw method of the CheckingAccount class call super.withdraw?

8. Why does the deductFees method set the transaction count to zero?

**Common Error 10.3**

**Accidental Overloading**

Recall from Section 2.4 that two methods can have the same name, provided they have different method parameters. For example, the PrintStream class has methods called println with headers

```java
void println(int x)
```

and

```java
void println(String x)
```
These are different methods, each with its own implementation. The Java compiler considers them to be completely unrelated. We say that the `println` name is overloaded. This is different from overriding, where a subclass method provides an implementation of a method with the same method parameters.

If you mean to override a method but supply a different parameter type, then you accidentally introduce an overloaded method. For example,

```java
public class CheckingAccount extends BankAccount{
    ...
    public void deposit(int amount) // Error: should be double
    {
        ...
    }
}
```

The compiler will not complain. It thinks that you want to provide a `deposit` method just for `int` parameters, while inheriting another `deposit` method for `double` parameters.

When overriding a method, be sure to check that the parameter types match exactly.

---

**Common Error 10.4**

**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:

```java
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’s declare 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 reserved word `super`, followed by the construction parameters in parentheses:

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

When the reserved word `super` is immediately 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.

The dual use of the `super` reserved word is analogous to the dual use of the `this` reserved word (see Special Topic 3.1).

If a subclass constructor does not call the superclass constructor, the superclass must have a constructor without parameters. That constructor is used to initialize the superclass data. 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 `BankAccount()` 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 instance variables.

**Syntax 10.3 Calling a Superclass Constructor**

```java
 Syntax accessSpecifier ClassName(parameterType parameterName, . . .) {
    super(parameters);
    ...
}
```

**Example**

Invokes the constructor of the superclass.

Must be the first statement of the subclass constructor.

If not present, the superclass constructor with no parameters is called.
9. Why didn’t the SavingsAccount constructor in Section 10.2 call its superclass constructor?

10. When you invoke a superclass method with the `super` reserved word, does the call have to be the first statement of the subclass method?
Subclass references can be converted to superclass references.

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.

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.

```java
SavingsAccount collegeFund = new SavingsAccount(10);
BankAccount anAccount = collegeFund; // OK
```

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

```java
Object anObject = collegeFund; // OK
```

Now the three object references stored in collegeFund, anAccount, and anObject all refer to the same object of type SavingsAccount (see Figure 6).

However, the variables anAccount and anObject know less than the full story about the object references that they store. Because anAccount is a variable of type BankAccount, you can invoke the deposit and withdraw methods. You cannot use the addInterest method, though—it is not a method of the BankAccount class:

```java
anAccount.deposit(1000); // OK
anAccount.addInterest(); // No—not a method of the type of the anAccount variable
```

And, of course, the variable anObject knows even less. You can’t even invoke the deposit method on it—deposit is not a method of the Object class.

Why would anyone want to know less about an object reference and use a variable whose type is 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:

```java
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:

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

![Figure 6](https://example.com/figure6.png)

**Figure 6**
Variables of Different Types Can Refer to the Same Object
You can also use the method to transfer money into a `CheckingAccount`:

```java
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 a `CheckingAccount` object. That is perfectly legal. The `transfer` method doesn’t actually know that, in this case, the parameter variable `other` contains a reference to a `CheckingAccount` object. All it cares about is that the object can carry out the deposit method. This is assured because the `other` variable has the type `BankAccount`.

Very occasionally, you need to carry out the opposite conversion, from a superclass type to a subclass type. 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:

```java
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,

```java
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:

```java
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**

```
Syntax    object instanceof TypeName
```

**Example**

*If anObject is null, `instanceof` returns false.*

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

*Returns true if anObject can be cast to a BankAccount.*

The object may belong to a subclass of `BankAccount`.

You can invoke `BankAccount` methods on this variable.

Two references to the same object.
11. Why did the second parameter of the `transfer` method have to be of type `BankAccount` and not, for example, `SavingsAccount`?

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

10.6 Polymorphism and Inheritance

In Java, the type of a variable does not determine the type of the object to which it refers. For example, a variable of type `BankAccount` can hold a reference to a `BankAccount` object or to 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 on a variable of type `BankAccount`? For example,

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

Which deposit method is called? The `anAccount` variable 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 reference stored in the `anAccount` variable actually refers to an object of the subclass `CheckingAccount`, so it would be appropriate if the `CheckingAccount.deposit` method were called instead.

Java uses *dynamic method lookup* to determine which method to invoke. The method to be called is always determined by the type of the actual object, not the type of the variable. 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 variable of type `BankAccount`.

Have another look at the `transfer` method:

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

Suppose you call

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

Two method calls are the result:

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

Depending on the actual types of the objects whose references are stored in `anAccount` and `anotherAccount`, different versions of the `withdraw` and `deposit` methods are called. This is an example of *polymorphism*. As we discussed in Chapter 9, polymorphism is the ability to treat objects with differences in behavior in a uniform way.

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

```java
withdraw(amount);
```
depends on the type of an object. However, that call is a shortcut for

\[
\text{this.withdraw(amount)};
\]

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**

```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: ");
        System.out.println("Expected: 7035");
        System.out.println("Harry's checking balance: ");
        System.out.println("Expected: 1116");
    }
}
```

**Program Run**

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

**SELF CHECK**

13. 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?

14. If a refers to a checking account, what is the effect of calling a.transfer(1000, a)?
10.6 Polymorphism and Inheritance

**Special Topic 10.1**

Abstract Classes

Special Topic 10.1 introduces the concept of abstract classes and methods. An abstract method has no implementation. (All methods of an interface are automatically abstract.) You cannot construct objects of abstract classes, typically because the class has one or more abstract methods. However, abstract classes differ from interfaces in an important way—they can have instance variables, and they can have concrete methods and constructors.

**Special Topic 10.2**

Final Methods and Classes

Special Topic 10.2 discusses final methods and classes. A final method cannot be overridden in a subclass. A final class cannot be subclassed.

**Common Error 10.5**

Overriding Methods to Be Less Accessible

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

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

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 conflict with polymorphism. Suppose the AccountTester class has this method call:

```java
BankAccount account = new CheckingAccount();
account.withdraw(100000); // Should CheckingAccount.withdraw be called?
```

Polymorphism dictates that CheckingAccount.withdraw should be called, but that is a private method that should not be accessible in AccountTester. Therefore, the compiler reports an error if you override a public method and make it private or give it package access. The latter is a common 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.

Available online in WileyPLUS and at www.wiley.com/college/horstmann.
Special Topic 10.3

Protected Access

Special Topic 10.3 covers the protected access specifier. A protected instance variable or method can be accessed by all subclasses and by all classes in the same package.

How To 10.1

Developing an Inheritance Hierarchy

When you work with a set of classes, some of which are more general and others more specialized, you want to organize them into an inheritance hierarchy. This enables you to process objects of different classes in a uniform way.

To illustrate the design process, consider an application that presents a quiz and grades the user’s responses. A quiz consists of questions, and there are different kinds of questions:

- Fill-in-the-blank
- Choice (single or multiple)
- Numeric (where an approximate answer is ok; e.g., 1.33 when the actual answer is 4/3)
- Free response

Step 1

List the classes that are part of the hierarchy.

From the problem description, we can find these classes:

- FillInQuestion (fill in the blank)
- ChoiceQuestion (offers answer choices to the user)
- MultiChoiceQuestion (offers answer choices to the user; user can pick more than one)
- NumericQuestion
- FreeResponseQuestion

In addition, we introduce a common superclass Question to model the commonality among these classes.

Step 2

Organize the classes into an inheritance hierarchy.

Draw a UML diagram that shows super- and subclasses. Here is the diagram for our example.
Step 3  Determine the common responsibilities.

In Step 2, you will have identified a class at the root of the hierarchy. That class needs to have sufficient responsibilities to carry out the tasks at hand.

To find out what those tasks are, write pseudocode for processing the objects.

For each question
  Display the question to the user.
  Get the user response.
  Check whether the response is correct.

From the pseudocode, we obtain the following list of common responsibilities that every question must carry out:

  Display the question.
  Check the response.

Step 4  Decide which methods are overridden in subclasses.

For each subclass and each of the common responsibilities, decide whether the inherited behavior is appropriate or whether it needs to be overridden. Be sure to declare any methods that are inherited or overridden in the root of the hierarchy.

We place the responsibilities common to all questions into the Question superclass.

```java
public class Question
{

  /**
   * Displays this question.
   */
  public void display() {
      . . .
  }

  /**
   * Checks a given response for correctness.
   * @param response the response to check
   * @return true if the response was correct, false otherwise
   */
  public boolean checkAnswer(String response) {
      . . .
  }
}
```

The ChoiceQuestion class will need to override the display method to display all the choices. The NumericQuestion class will need to override the checkAnswer method, converting the response to a number and checking that it is approximately the same as the expected answer.

From now on, we will only consider the ChoiceQuestion in detail. For the other question types, see the programming exercises at the end of this chapter.

Step 5  Define the public interface of each subclass.

Typically, subclasses have responsibilities other than those of the superclass. List those, as well as the methods that need to be overridden. You also need to specify how the objects of the subclasses should be constructed.

With the ChoiceQuestion, we need a way of adding choices, like this:

```java
ChoiceQuestion question = new ChoiceQuestion(
    "In which country was the inventor of Java born?");
question.addChoice("Australia", false);
question.addChoice("Canada", true);
question.addChoice("Denmark", false);
question.addChoice("United States", false);
```
We then override the `display` method to display those choices in the form:

1: Australia  
2: Canada  
3: Denmark  
4: United States

Here are the methods that we just discovered for the `ChoiceQuestion` class:

```java
public class ChoiceQuestion extends Question {
    . . .
    /**
     * Adds an answer choice to this question.
     * @param choice the choice to add
     * @param correct true if this is the correct choice, false otherwise
     */
    public void addChoice(String choice, boolean correct) {
    }

    public void display() { . . . } // Overrides superclass method
}
```

**Step 6** Identify instance variables.

List the instance variables for each class. If you find a instance variable that is common to all classes, be sure to place it in the base of the hierarchy.

All questions have a question text and an answer. We store those values in the `Question` superclass.

```java
public class Question {
    private String text;
    private String answer;
    . . .
}
```

The `ChoiceQuestion` class needs to store the list of choices.

```java
public class ChoiceQuestion extends Question {
    private ArrayList<String> choices;
    . . .
}
```

We need to spend some thought on how question objects are constructed. We can supply the question text in the constructor. However, the answer for a choice question is only known when the correct choice is added, so we need a setter method for it:

```java
public class Question {
    . . .
    /**
     * Constructs a question with a given text and an empty answer.
     * @param questionText the text of this question
     */
    public Question(String questionText) { . . . }

    /**
     * Sets the answer for this question.
     * @param correctResponse the answer
     */
    public void setAnswer(String correctResponse) { . . . }
}
```
Step 7  Implement constructors and methods.

The methods of the Question class are very straightforward:

```java
public class Question {
    // ...
    public Question(String questionText) {
        text = questionText;
        answer = "";
    }
    public void setAnswer(String correctResponse) {
        answer = correctResponse;
    }
    public boolean checkAnswer(String response) {
        return response.equals(answer);
    }
    public void display() {
        System.out.println(text);
    }
}
```

The ChoiceQuestion constructor must call the superclass constructor to set the question text:

```java
public ChoiceQuestion(String questionText) {
    super(questionText);
    choices = new ArrayList<String>();
}
```

The `addChoice` method sets the answer when the correct choice is added.

```java
public void addChoice(String choice, boolean correct) {
    choices.add(choice);
    if (correct) {
        // Convert choices.size() to string
        String choiceString = "" + choices.size();
        setAnswer(choiceString);
    }
}
```

Finally, the `display` method of the ChoiceQuestion class displays the question text, then the choices. Note the call to the superclass method.

```java
public void display() {
    super.display();
    for (int i = 0; i < choices.size(); i++) {
        System.out.println(choiceNumber + ": " + choices.get(i));
    }
}
```
Step 8 Construct objects of different subclasses and process them.

In our sample program, we construct two questions and present them to the user.

```java
public class QuestionDemo
{
    public static void main(String[] args)
    {
        Question[] quiz = new Question[2];
        quiz[0] = new Question("Who was the inventor of Java?");
        quiz[0].setAnswer("James Gosling");

        ChoiceQuestion question = new ChoiceQuestion("In which country was the inventor of Java born?");
        question.addChoice("Australia", false);
        question.addChoice("Canada", true);
        question.addChoice("Denmark", false);
        question.addChoice("United States", false);
        quiz[1] = question;

        Scanner in = new Scanner(System.in);
        for (Question q : quiz)
        {
            q.display();
            System.out.print("Your answer: ");
            String response = in.nextLine();
            System.out.println(q.checkAnswer(response));
        }
    }
}
```

Program Run

Who was the inventor of Java?
Your answer: James Gosling
true
In which country was the inventor of Java born?
1: Australia
2: Canada
3: Denmark
4: United States
Your answer: 4
false

The complete program is contained in the ch10/questions directory of your source code.

Worked Example 10.1 Implementing an Employee Hierarchy for Payroll Processing

This Worked Example shows how to implement payroll processing that works for different kinds of employees.
10.7 **Object: The Cosmic Superclass**

In Java, every class that is declared 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 7).

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>String <code>toString()</code></td>
<td>Returns a string representation of the object</td>
</tr>
<tr>
<td>boolean <code>equals(Object otherObject)</code></td>
<td>Tests whether the object equals another object</td>
</tr>
<tr>
<td><code>Object clone()</code></td>
<td>Makes a full copy of an object</td>
</tr>
</tbody>
</table>

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

### 10.7.1 Overriding the `toString` Method

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

```java
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]"
```

![Figure 7](image-url) **Figure 7** The Object Class Is the Superclass of Every Java Class
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 provides a `toString` method.

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

```java
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:

```java
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 variables inside brackets.

```java
public class BankAccount
{
    ...;

    public String toString()
    {
        return "BankAccount[balance=\"" + balance + "]";
    }
}
```

This works better:

```java
BankAccount momsSavings = new BankAccount(5000);
String s = momsSavings.toString();
// Sets s to "BankAccount[balance=5000]"
```
10.7.2 Overriding the equals Method

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

```java
if (coin1.equals(coin2)) ... // Contents are the same—see Figure 8
```

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

```java
if (coin1 == coin2) ... // Objects are the same—see Figure 9
```

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

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

Now you have a slight problem. The Object class knows nothing about coins, so it declares the otherObject parameter of the equals method to have the type Object. When overriding the method, you are not allowed to change the parameter type. To overcome this problem, cast the parameter to the class Coin:

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

Then you can compare the two coins.

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

---

*Figure 8* Two References to Equal Objects

*Figure 9* Two References to the Same Object
Note that you must use equals to compare object references, but use == to compare numbers.

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.

10.7.3 The clone Method

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

```java
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 10).

Implementing the clone method is quite a bit more difficult than implementing the toString or equals methods—see Special 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:

```java
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.

### SELF CHECK

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

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

Inheritance and the toString Method

Special Topic 10.4 gives a recipe for implementing the toString method so that it can be easily extended in subclasses.

Common Error 10.6

Declaring the equals Method with the Wrong Parameter Type

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

```java
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:

```java
boolean equals(Coin other) // Declared 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 declarations:

```java
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.
**Special Topic 10.5**

**Inheritance and the equals Method**

Special Topic 10.5 analyzes the subtle problems that arise when the `equals` method is overridden in a subclass, and it gives you a recipe for minimizing these problems.

**Quality Tip 10.2**

**Clone Mutable Instance Variables in Accessor Methods**

Quality Tip 10.2 suggests that your accessor methods should not give out references to mutable instance variables, but that the instance variable values should first be cloned.

**Special Topic 10.6**

**Implementing the clone Method**

Special Topic 10.6 explains how to implement the `clone` method for your own classes.

**Special Topic 10.7**

**Enumeration Types Revisited**

Special Topic 10.7 revisits enumeration types and explains that they are all subclasses of the class `Enum`. The `Enum` class has suitable implementations of the `toString`, `equals`, and `clone` methods that are inherited by all enumeration types.

**Random Fact 10.1**

**Scripting Languages**

Random Fact 10.1 discusses scripting languages that are designed for rapid development, having a simple structure and fewer syntax rules, and often supporting a particular application (such as office software or a web browser).
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.

To do so, design a subclass of `JFrame`. Store the components as instance variables. 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.

```java
public class InvestmentFrame extends JFrame
{
    private JButton button;
    private JLabel label;
    private JPanel panel;
    private BankAccount account;

    public InvestmentFrame()
    {
        account = new BankAccount(INITIAL_BALANCE);

        // Use instance variables for components
        label = new JLabel("balance: " + account.getBalance());

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

        setSize(FRAME_WIDTH, FRAME_HEIGHT);
    }

    private void createButton()
    {
        button = new JButton("Add Interest");
        ActionListener listener = new AddInterestListener();
        button.addActionListener(listener);
    }

    private void createPanel()
    {
        panel = new JPanel();
        panel.add(button);
        panel.add(label);
        add(panel);
    }

    . . .
}
```

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.

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

```java
public class InvestmentViewer2
{
    . . .
}
```
Chapter 10  Inheritance

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

SELF CHECK

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

18. 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)?

Special 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 variables for components
        label = new JLabel("balance: " + account.getBalance());

        // 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.
Classes, Objects, and Methods Introduced in this Chapter

Summary of Learning Objectives

**Explain the notions of inheritance, superclasses, and subclasses.**
- Sets of classes can form complex inheritance hierarchies.

**Implement subclasses in Java.**
- Inheritance is a mechanism for extending existing classes by adding instance variables and methods.
- A subclass inherits the methods of its superclass.
- The instance variables declared in the superclass are present in subclass objects.
- A subclass has no access to private instance variables of its superclass.
- The more general class is called a superclass. The more specialized class that inherits from the superclass is called the subclass.
- Inheriting from a class differs from implementing an interface: The subclass inherits behavior from the superclass.

**Describe how a subclass can override methods from its superclass.**
- A subclass can inherit a superclass method or override it by providing another implementation.
- Use the `super` reserved word to call a method of the superclass.

**Describe how a subclass can construct its superclass.**
- To call the superclass constructor, you use the `super` reserved word in the first statement of the subclass constructor.

**Describe how to convert between class and superclass types.**
- Subclass references can be converted to superclass references.
- The `instanceof` operator tests whether an object belongs to a particular type.

**Describe dynamic method lookup and polymorphism.**
- When the virtual machine calls an instance method, it locates the method of the implicit parameter’s class. This is called dynamic method lookup.

**Provide appropriate overrides of the methods of the `Object` superclass.**
- Every class extends the `Object` class either directly or indirectly.
- In your classes, provide `toString` methods that describe each object’s state.
- When implementing the `equals` method, test whether two objects have equal state.
- The `clone` method makes a new object with the same state as an existing object.

**Use inheritance to customize frames.**
- Provide a `JFrame` subclass for a complex frame.

---

**Classes, Objects, and Methods Introduced in this Chapter**

- `java.lang.Cloneable`
- `java.lang.CloneNotSupportedException`
- `java.lang.Object`
- `clone`
- `toString`