Syntax 9.2 Constructor with Superclass Initializer

Syntax

```
public ClassName(parameterType parameterName, . . .)
{
   super(arguments);
   . . .
}
```

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

The superclass constructor is called first.

The constructor body can contain additional statements.

If you omit the superclass constructor call, the superclass constructor with no arguments is invoked.

9.4 Polymorphism

In this section, you will learn how to use inheritance for processing objects of different types in the same program.

Consider our first sample program. It presented two `Question` objects to the user. The second sample program presented two `ChoiceQuestion` objects. Can we write a program that shows a mixture of both question types?

With inheritance, this goal is very easy to realize. In order to present a question to the user, we need not know the exact type of the question. We just display the question and check whether the user supplied the correct answer. The `Question` superclass has methods for this purpose. Therefore, we can simply declare the parameter variable of the `presentQuestion` method to have the type `Question`:

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

As discussed in Section 9.1, we can substitute a subclass object whenever a superclass object is expected:

```
ChoiceQuestion second = new ChoiceQuestion();
. . .
presentQuestion(second); // OK to pass a ChoiceQuestion
```

A subclass reference can be used when a superclass reference is expected.

When the `presentQuestion` method executes, the object references stored in `second` and `q` refer to the same object of type `ChoiceQuestion` (see Figure 6).

However, the *variable* `q` knows less than the full story about the object to which it refers (see Figure 7).

Because `q` is a variable of type `Question`, you can call the `display` and `checkAnswer` methods. You cannot call the `addChoice` method, though—it is not a method of the `Question` superclass.

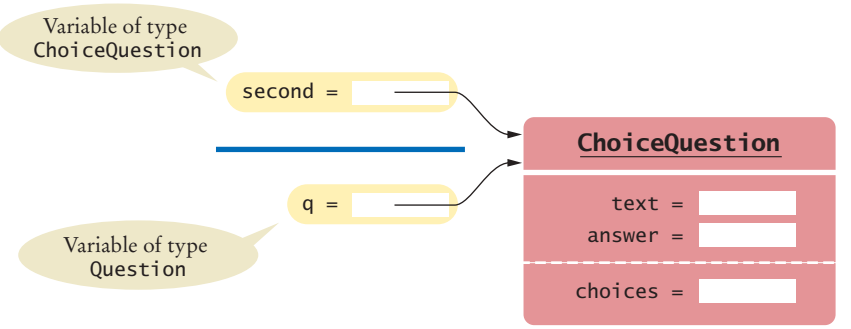


Figure 6 Variables of Different Types Referring to the Same Object

This is as it should be. After all, it happens that in this method call, `q` refers to a `ChoiceQuestion`. In another method call, `q` might refer to a plain `Question` or an entirely different subclass of `Question`.

Now let's have a closer look inside the `presentQuestion` method. It starts with the call

```
q.display(); // Does it call Question.display or ChoiceQuestion.display?
```

Which `display` method is called? If you look at the program output on page 433, you will see that the method called depends on the contents of the parameter variable `q`. In the first case, `q` refers to a `Question` object, so the `Question.display` method is called. But in the second case, `q` refers to a `ChoiceQuestion`, so the `ChoiceQuestion.display` method is called, showing the list of choices.

In Java, method calls *are always determined by the type of the actual object*, not the type of the variable containing the object reference. This is called **dynamic method lookup**.

Polymorphism ("having multiple shapes") allows us to manipulate objects that share a set of tasks, even though the tasks are executed in different ways.

Dynamic method lookup allows us to treat objects of different classes in a uniform way. This feature is called **polymorphism**. We ask multiple objects to carry out a task, and each object does so in its own way.

Polymorphism makes programs *easily extensible*. Suppose we want to have a new kind of question for calculations, where we are willing to accept an approximate answer. All we need to do is to declare a new class `NumericQuestion` that extends `Question`, with its own `checkAnswer` method. Then we can call the `presentQuestion` method with a mixture of plain questions, choice questions, and numeric questions. The `presentQuestion` method need not be changed at all! Thanks to dynamic method lookup, method calls to the `display` and `checkAnswer` methods automatically select the correct method of the newly declared classes.

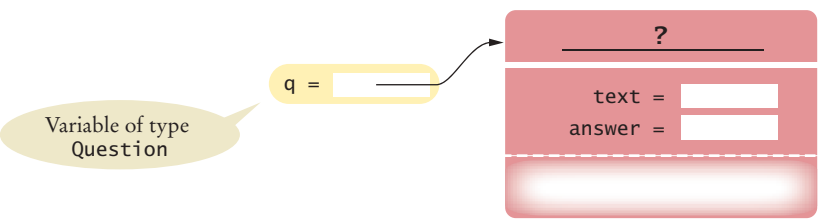


Figure 7 A `Question` Reference Can Refer to an Object of Any Subclass of `Question`

In the same way that vehicles can differ in their method of locomotion, polymorphic objects carry out tasks in different ways.

section_4/QuestionDemo3.java

```
1  import java.util.Scanner;
2
3  /**
4     This program shows a simple quiz with two question types.
5  */
6  public class QuestionDemo3
7  {
8     public static void main(String[] args)
9     {
10       Question first = new Question();
11       first.setText("Who was the inventor of Java?");
12       first.setAnswer("James Gosling");
13
14       ChoiceQuestion second = new ChoiceQuestion();
15       second.setText("In which country was the inventor of Java born?");
16       second.addChoice("Australia", false);
17       second.addChoice("Canada", true);
18       second.addChoice("Denmark", false);
19       second.addChoice("United States", false);
20
21       presentQuestion(first);
22       presentQuestion(second);
23    }
24
25    /**
26       Presents a question to the user and checks the response.
27       @param q the question
28    */
29    public static void presentQuestion(Question q)
30    {
31       q.display();
32       System.out.print("Your answer: ");
33       Scanner in = new Scanner(System.in);
34       String response = in.nextLine();
35       System.out.println(q.checkAnswer(response));
36    }
37 }
```

Program Run

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

16. Assuming `SavingsAccount` is a subclass of `BankAccount`, which of the following code fragments are valid in Java?
    a. `BankAccount account = new SavingsAccount();`
    b. `SavingsAccount account2 = new BankAccount();`
    c. `BankAccount account = null;`
    d. `SavingsAccount account2 = account;`
17. If `account` is a variable of type `BankAccount` that holds a non-`null` reference, what do you know about the object to which `account` refers?
18. Declare an array `quiz` that can hold a mixture of `Question` and `ChoiceQuestion` objects.
19. Consider the code fragment

```
ChoiceQuestion cq = . . .; // A non-null value
cq.display();
```

    Which actual method is being called?
20. Is the method call `Math.sqrt(2)` resolved through dynamic method lookup?

Practice It Now you can try these exercises at the end of the chapter: R9.6, P9.4, P9.20.

Special Topic 9.2

Dynamic Method Lookup and the Implicit Parameter

Suppose we add the `presentQuestion` method to the `Question` class itself:

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

Now consider the call

```
ChoiceQuestion cq = new ChoiceQuestion();
cq.setText("In which country was the inventor of Java born?");
. . .
cq.presentQuestion();
```

Which `display` and `checkAnswer` method will the `presentQuestion` method call? If you look inside the code of the `presentQuestion` method, you can see that these methods are executed on the implicit parameter.

```
public class Question
{
   public void presentQuestion()
   {
      this.display();
      System.out.print("Your answer: ");
      Scanner in = new Scanner(System.in);
      String response = in.nextLine();
      System.out.println(this.checkAnswer(response));
   }
}
```

The implicit parameter `this` in our call is a reference to an object of type `ChoiceQuestion`. Because of dynamic method lookup, the `ChoiceQuestion` versions of the `display` and `checkAnswer` methods are called automatically. This happens even though the `presentQuestion` method is declared in the `Question` class, which has *no knowledge* of the `ChoiceQuestion` class.

As you can see, polymorphism is a very powerful mechanism. The `Question` class supplies a `presentQuestion` method that specifies the common nature of presenting a question, namely to display it and check the response. How the displaying and checking are carried out is left to the subclasses.

Special Topic 9.3

Abstract Classes

When you extend an existing class, you have the choice whether or not to override the methods of the superclass. Sometimes, it is desirable to *force* programmers to override 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 `Account` class:

```
public class Account
{
   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 `Account` class has an abstract method, the compiler will flag an attempt to create a `new Account()` as an error.

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 reserved word `abstract`:

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

public class SavingsAccount extends Account // Not abstract
{
   . . .
   public void deductFees() // Provides an implementation
   {
      . . .
   }
}
```

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:

```
Account anAccount; // OK
anAccount = new Account(); // Error—Account 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.

Special Topic 9.4

Final Methods and Classes

In Special Topic 9.3 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` reserved word. 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. When you have a reference of type `String`, it must contain a `String` object, never an object of a subclass.

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`.

Special Topic 9.5

Protected Access

We ran into a hurdle when trying to implement the `display` method of the `ChoiceQuestion` class. That method wanted to access the instance variable `text` of the superclass. Our remedy was to use the appropriate method of the superclass to display the text.

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

```
public class Question
{
   protected String text;
   . . .
}
```

Protected data in an object can be accessed by the methods of the object's class and all its subclasses. For example, `ChoiceQuestion` inherits from `Question`, so its methods can access the protected instance variables of the `Question` superclass.

Some programmers like the `protected` access feature because it seems to strike a balance between absolute protection (making instance variables private) and no protection at all (making instance variables public). However, experience has shown that protected instance variables are subject to the same kinds of problems as public instance variables. 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 variables are hard to modify. Even if the author of the superclass would like to change the data implementation, the protected variables cannot be changed, because someone somewhere out there might have written a subclass whose code depends on them.

In Java, protected variables have another drawback—they are accessible not just by subclasses, but also by other classes in the same package (see Section 12.4 for information about packages).

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.

HOW TO 9.1

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.

As an example, we will consider a bank that offers customers the following account types:

- A savings account that earns interest. The interest compounds monthly and is computed on the minimum monthly balance.
- A checking account that has no interest, gives you three free withdrawals per month, and charges a $1 transaction fee for each additional withdrawal.

The program will manage a set of accounts of both types, and it should be structured so that other account types can be added without affecting the main processing loop. Supply a menu

```
D)eposit  W)ithdraw  M)onth end  Q)uit
```

For deposits and withdrawals, query the account number and amount. Print the balance of the account after each transaction.

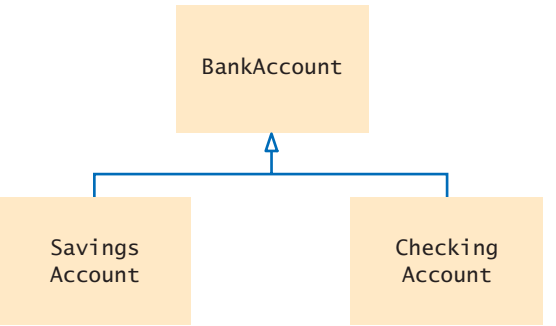
In the "Month end" command, accumulate interest or clear the transaction counter, depending on the type of the bank account. Then print the balance of all accounts.

Step 1 List the classes that are part of the hierarchy.

In our case, the problem description yields two classes: `SavingsAccount` and `CheckingAccount`. Of course, you could implement each of them separately. But that would not be a good idea because the classes would have to repeat common functionality, such as updating an account balance. We need another class that can be responsible for that common functionality. The problem statement does not explicitly mention such a class. Therefore, we need to discover it. Of course, in this case, the solution is simple. Savings accounts and checking accounts are special cases of a bank account. Therefore, we will introduce a common superclass `BankAccount`.

Step 2 Organize the classes into an inheritance hierarchy.

Draw an inheritance diagram that shows super- and subclasses. Here is one for our example:



Step 3 Determine the common responsibilities.

In Step 2, you will have identified a class at the base 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 user command
   If it is a deposit or withdrawal
      Deposit or withdraw the amount from the specified account.
      Print the balance.
   If it is month end processing
      For each account
         Call month end processing.
         Print the balance.
```

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

```
Deposit money.
Withdraw money.
Get the balance.
Carry out month end processing.
```

Step 4 Decide which methods are overridden in subclasses.

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

```
public class BankAccount
{
   . . .
```

```
   /**
      Makes a deposit into this account.
      @param amount the amount of the deposit
   */
   public void deposit(double amount) { . . . }

   /**
      Makes a withdrawal from this account, or charges a penalty if
      sufficient funds are not available.
      @param amount the amount of the withdrawal
   */
   public void withdraw(double amount) { . . . }

   /**
      Carries out the end of month processing that is appropriate
      for this account.
   */
   public void monthEnd() { . . . }

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

The `SavingsAccount` and `CheckingAccount` classes both override the `monthEnd` method. The `SavingsAccount` class must also override the `withdraw` method to track the minimum balance. The `CheckingAccount` class must update a transaction count in the `withdraw` method.

Step 5 Declare 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.

In this example, we need a way of setting the interest rate for the savings account. In addition, we need to specify constructors and overridden methods.

```
public class SavingsAccount extends BankAccount
{
   . . .
   /**
      Constructs a savings account with a zero balance.
   */
   public SavingsAccount() { . . . }

   /**
      Sets the interest rate for this account.
      @param rate the monthly interest rate in percent
   */
   public void setInterestRate(double rate) { . . . }

   // These methods override superclass methods
   public void withdraw(double amount) { . . . }
   public void monthEnd() { . . . }
}

public class CheckingAccount extends BankAccount
{
   . . .
   /**
```

```
      Constructs a checking account with a zero balance.
   */
   public CheckingAccount() { . . . }

   // These methods override superclass methods
   public void withdraw(double amount) { . . . }
   public void monthEnd() { . . . }
}
```

Step 6 Identify instance variables.

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

All accounts have a balance. We store that value in the `BankAccount` superclass:

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

The `SavingsAccount` class needs to store the interest rate. It also needs to store the minimum monthly balance, which must be updated by all withdrawals.

```
public class SavingsAccount extends BankAccount
{
   private double interestRate;
   private double minBalance;
   . . .
}
```

The `CheckingAccount` class needs to count the withdrawals, so that the charge can be applied after the free withdrawal limit is reached.

```
public class CheckingAccount extends BankAccount
{
   private int withdrawals;
   . . .
}
```

Step 7 Implement constructors and methods.

The methods of the `BankAccount` class update or return the balance.

```
public void deposit(double amount)
{
   balance = balance + amount;
}

public void withdraw(double amount)
{
   balance = balance - amount;
}

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

At the level of the `BankAccount` superclass, we can say nothing about end of month processing. We choose to make that method do nothing:

```
public void monthEnd()
{
}
```

In the `withdraw` method of the `SavingsAccount` class, the minimum balance is updated. Note the call to the superclass method:

```
public void withdraw(double amount)
{
   super.withdraw(amount);
   double balance = getBalance();
   if (balance < minBalance)
   {
      minBalance = balance;
   }
}
```

In the `monthEnd` method of the `SavingsAccount` class, the interest is deposited into the account. We must call the `deposit` method because we have no direct access to the `balance` instance variable. The minimum balance is reset for the next month.

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

The `withdraw` method of the `CheckingAccount` class needs to check the withdrawal count. If there have been too many withdrawals, a charge is applied. Again, note how the method invokes the superclass method:

```
public void withdraw(double amount)
{
   final int FREE_WITHDRAWALS = 3;
   final int WITHDRAWAL_FEE = 1;

   super.withdraw(amount);
   withdrawals++;
   if (withdrawals > FREE_WITHDRAWALS)
   {
      super.withdraw(WITHDRAWAL_FEE);
   }
}
```

End of month processing for a checking account simply resets the withdrawal count.

```
public void monthEnd()
{
   withdrawals = 0;
}
```

Step 8 Construct objects of different subclasses and process them.

In our sample program, we allocate 5 checking accounts and 5 savings accounts and store their addresses in an array of bank accounts. Then we accept user commands and execute deposits, withdrawals, and monthly processing.

```
BankAccount[] accounts = . . .;
. . .
Scanner in = new Scanner(System.in);
boolean done = false;
while (!done)
{
   System.out.print("D)eposit  W)ithdraw  M)onth end  Q)uit: ");
   String input = in.next();
   if (input.equals("D") || input.equals("W")) // Deposit or withdrawal
   {
```