3.3.2 Providing Constructors

A **constructor** has a simple job: to initialize the instance variables of an object.

Recall that we designed the `BankAccount` class to have two constructors. The first constructor simply sets the balance to zero:

```
public BankAccount()
{
   balance = 0;
}
```

The second constructor sets the balance to the value supplied as the construction argument:

```
public BankAccount(double initialBalance)
{
   balance = initialBalance;
}
```

To see how these constructors work, let us trace the statement

```
BankAccount harrysChecking = new BankAccount(1000);
```

one step at a time.

Here are the steps that are carried out when the statement executes (see Figure 5):

- Create a new object of type `BankAccount`. 1
- Call the second constructor (because an argument is supplied in the constructor call).
- Set the parameter variable `initialBalance` to 1000. 2
- Set the `balance` instance variable of the newly created object to `initialBalance`. 3
- Return an object reference, that is, the memory location of the object, as the value of the `new` expression.
- Store that object reference in the `harrysChecking` variable. 4

In general, when you implement constructors, be sure that each constructor initializes all instance variables, and that you make use of all parameter variables (see Common Error 3.2 on page 98).

A constructor is like a set of assembly instructions for an object.

For this purpose, we specify two constructors:

- `public BankAccount()`
- `public BankAccount(double initialBalance)`

They are used as follows:

```
BankAccount harrysChecking = new BankAccount();
BankAccount momsSavings = new BankAccount(5000);
```

The constructor name is always the same as the class name.

Don't worry about the fact that there are two constructors with the same name—*all* constructors of a class have the same name, that is, the name of the class. The compiler can tell them apart because they take different arguments. The first constructor takes no arguments at all. Such a constructor is called a **no-argument constructor**. The second constructor takes an argument of type `double`.

Just like a method, a constructor also has a body—a sequence of statements that is executed when a new object is constructed.

```
public BankAccount()
{
   constructor body—implementation filled in later
}
```

The statements in the constructor body will set the instance variables of the object that is being constructed—see Section 3.3.

When declaring a class, you place all constructor and method declarations inside, like this:

```
public class BankAccount
{
   private instance variables—filled in later

   // Constructors
   public BankAccount()
   {
      implementation—filled in later
   }

   public BankAccount(double initialBalance)
   {
      implementation—filled in later
   }

   // Methods
   public void deposit(double amount)
   {
      implementation—filled in later
   }

   public void withdraw(double amount)
   {
      implementation—filled in later
   }

   public double getBalance()
   {
      implementation—filled in later
   }
}
```

`Manager` class. Calling that method is a recursive call, which will never stop. Instead, you must tell the compiler to invoke the superclass method.

Whenever you call a superclass method from a subclass method with the same name, be sure to use the reserved word `super`.

Special Topic 9.1

Calling the Superclass Constructor

Consider the process of constructing a subclass object. A subclass constructor can only initialize the instance variables of the subclass. But the superclass instance variables also need to be initialized. Unless you specify otherwise, the superclass instance variables are initialized with the constructor of the superclass that has no arguments.

Unless specified otherwise, the subclass constructor calls the superclass constructor with no arguments.

In order to specify another constructor, you use the `super` reserved word, together with the arguments of the superclass constructor, as the *first statement* of the subclass constructor.

For example, suppose the `Question` superclass had a constructor for setting the question text. Here is how a subclass constructor could call that superclass constructor:

To call a superclass constructor, use the super reserved word in the first statement of the subclass constructor.

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

In our example program, we used the superclass constructor with no arguments. However, if all superclass constructors have arguments, you must use the `super` syntax and provide the arguments for a superclass constructor.

The constructor of a subclass can pass arguments to a superclass constructor, using the reserved word super.

When the reserved word `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.

Syntax 9.3 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.

Constructors

A constructor definition has the form

modifiers *ClassName*(*parameter*$_1$, *parameter*$_2$, . . .)
    [throws *ExceptionType*$_1$, *ExceptionType*$_2$, . . .]
{
   body
}

You invoke a constructor to allocate and construct a new object with a `new` expression

`new` *ClassName*(*parameterValue*$_1$, *parameterValue*$_2$, . . .)

A constructor can call the body of another constructor of the same class with the syntax

`this`(*parameterValue*$_1$, *parameterValue*$_2$, . . .)

For example,

```
public Employee()
{
   this("", 0);
}
```

It can call a constructor of its superclass with the syntax

`super`(*parameterValue*$_1$, *parameterValue*$_2$, . . .)

The call to `this` or `super` must be the first statement in the constructor.

Arrays are constructed with the syntax

`new` *ArrayType* `[ = {` *initializer*$_1$, *initializer*$_2$, . . . `}]`

For example,

```
new int[] = { 1, 4, 9, 16, 25 }
```

When an object is constructed, the following actions take place:

- All instance variables are initialized with `0`, `false`, or `null`.
- The initializers and initialization blocks are executed in the order in which they are declared.
- The body of the constructor is invoked.

When a class is loaded, the following actions take place:

- All static variables are initialized with `0`, `false`, or `null`.
- The initializers of static variables and static initialization blocks are executed in the order in which they are declared.

Statements

A *statement* is one of the following:

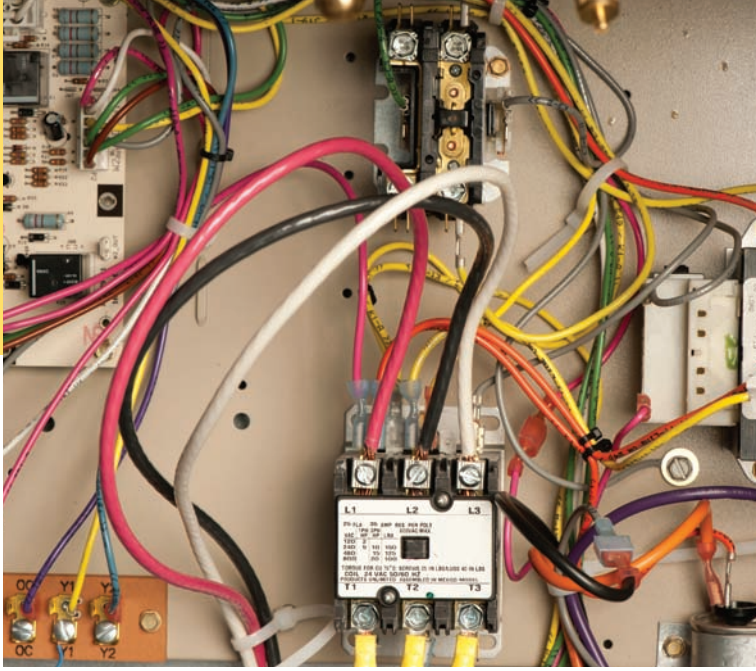
- An expression followed by a semicolon
- A branch or loop statement
- A `return` statement

K

L

CHAPTER 3

IMPLEMENTING CLASSES



CHAPTER GOALS

To become familiar with the process of implementing classes

To be able to implement and test simple methods

To understand the purpose and use of constructors

To understand how to access instance variables and local variables

To be able to write javadoc comments

To implement classes for drawing graphical shapes

CHAPTER CONTENTS

Special Topic 3.1

Calling One Constructor from Another

Consider the `BankAccount` class. It has two constructors: a no-argument constructor to initialize the balance with zero, and another constructor to supply an initial balance. Rather than explicitly setting the balance to zero, one constructor can call another constructor of the same class instead. There is a shorthand notation to achieve this result:

```
public class BankAccount
{
   public BankAccount(double initialBalance)
   {
      balance = initialBalance;
   }

   public BankAccount()
   {
      this(0);
   }
   . . .
}
```

The command `this(0);` means "Call another constructor of this class and supply the value `0`". Such a call to another constructor can occur only as the *first line in a constructor*.

This syntax is a minor convenience. We will not use it in this book. Actually, the use of the reserved word `this` is a little confusing. Normally, `this` denotes a reference to the implicit parameter, but if `this` is followed by parentheses, it denotes a call to another constructor of the same class.

3.8 Shape Classes

In this section, we continue the optional graphics track by discussing how to organize complex drawings in a more object-oriented fashion.

It is a good idea to make a class for any part of a drawing that can occur more than once.

When you produce a drawing that has multiple shapes, or parts made of multiple shapes, such as the car in Figure 8, it is a good idea to make a separate class for each part. The class should have a `draw` method that draws the shape, and a constructor to set the position of the shape. For example, here is the outline of the `Car` class:

```
public class Car
{
   public Car(int x, int y)
   {
      // Remember position
      . . .
   }

   public void draw(Graphics2D g2)
   {
      // Drawing instructions
      . . .
   }
}
```

You will find the complete class declaration at the end of this section. The `draw` method contains a rather long sequence of instructions for drawing the body, roof, and tires.

Each *feature* is either a declaration of the form

modifiers constructor|method|instance variable|class

or an initialization block

[static] { *body* }

See the section "Constructors" for more information about initialization blocks.

Potential *modifiers* include `public`, `private`, `protected`, `static`, and `final`.

An *instance variable* declaration has the form

Type variableName [= *initializer*];

A *constructor* has the form

ClassName(*parameter*$_1$, *parameter*$_2$, . . .)
    [throws *ExceptionType*$_1$, *ExceptionType*$_2$, . . .]
{
   body
}

A *method* has the form

Type methodName(*parameter*$_1$, *parameter*$_2$, . . .)
    [throws *ExceptionType*$_1$, *ExceptionType*$_2$, . . .]
{
   body
}

An *abstract method* has the form

abstract *Type methodName*(*parameter*$_1$, *parameter*$_2$, . . .);

Here is an example:

```
public class Point
{
   private double x;  // Instance variable
   private double y;

   public Point()  // Constructor with no arguments
   {
      x = 0; y = 0;
   }

   public Point(double xx, double yy)  // Constructor
   {
      x = xx; y = yy;
   }

   public double getX()  // Method
   {
      return x;
   }

   public double getY()  // Method
   {
      return y;
   }
}
```

A class can have both instance variables and `static` variables. Each object of the class has a separate copy of the instance variables. There is only a one per-class copy of the `static` variables.

In Java, you call a method when you want to apply an operation to an object. To figure out the exact specification of the method calls, imagine how a programmer would carry out the bank account operations. We'll assume that the variable `harrysChecking` contains a reference to an object of type `BankAccount`. We want to support method calls such as the following:

```
harrysChecking.deposit(2240.59);
harrysChecking.withdraw(500);
double currentBalance = harrysChecking.getBalance();
```

The first two methods are mutators. They modify the balance of the bank account and don't return a value. The third method is an accessor. It returns a value that you store in a variable or pass to a method.

From the sample calls, we decide the `BankAccount` class should declare three methods:

- `public void deposit(double amount)`
- `public void withdraw(double amount)`
- `public double getBalance()`

Recall from Chapter 2 that `double` denotes the double-precision floating-point type, and `void` indicates that a method does not return a value.

Here we only give the method *headers*. When you declare a method, you also need to provide the method **body**, which consists of statements that are executed when the method is called.

```
public void deposit(double amount)
{
   method body—implementation filled in later
}
```

We will supply the method bodies in Section 3.3.

Note that the methods have been declared as `public`, indicating that all other methods in a program can call them. Occasionally, it can be useful to have `private` methods. They can only be called from other methods of the same class.

Some people like to fill in the bodies so that they compile, like this:

```
public double getBalance()
{
   // TODO: fill in implementation
   return 0;
}
```

That is a good idea if you compose your specification in your development environment—you won't get warnings about incorrect code.

3.2.2 Specifying Constructors

Constructors set the initial data for objects.

As you know from Chapter 2, constructors are used to initialize objects. In Java, a **constructor** is very similar to a method, with two important differences:

- The name of the constructor is always the same as the name of the class (e.g., `BankAccount`).
- Constructors have no return type (not even `void`).

We want to be able to construct bank accounts that initially have a zero balance, as well as accounts that have a given initial balance.

Class java.util.GregorianCalendar

- **GregorianCalendar**()
  This constructs a calendar object that represents the current date and time.
- **GregorianCalendar**(int year, int month, int day)
  This constructs a calendar object that represents the start of the given date.
  Parameters: year, month, day The given date

Class java.util.HashMap<K, V>

- **HashMap**<K, V>()
  This constructs an empty hash map.

Class java.util.HashSet<E>

- **HashSet**<E>()
  This constructs an empty hash set.

Class java.util.InputMismatchException

This exception is thrown if the next available input item does not match the type of the requested item.

Interface java.util.Iterator<E>

- boolean **hasNext**()
  This method checks whether the iterator is past the end of the list.
  Returns: true if the iterator is not yet past the end of the list
- E **next**()
  This method moves the iterator over the next element in the linked list. This method throws an exception if the iterator is past the end of the list.
  Returns: The object that was just skipped over
- void **remove**()
  This method removes the element that was returned by the last call to next or previous. This method throws an exception if there was an add or remove operation after the last call to next or previous.

Class java.util.LinkedHashMap<K, V>

- **LinkedHashMap**<K, V>()
  This constructs an empty linked hash map. The iterator of a linked hash map visits the entries in the order in which they were added to the map.

Class java.util.LinkedList<E>

- void **addFirst**(E element)
- void **addLast**(E element)
  These methods add an element before the first or after the last element in this list.
  Parameters: element The element to be added

CHAPTER	 Common Errors	 How Tos and Worked Examples 
1 Introduction	Omitting Semicolons 14 Misspelling Words 16	Describing an Algorithm with Pseudocode 20 Writing an Algorithm for Tiling a Floor 22
2 Using Objects	Using Undeclared or Uninitialized Variables 42 Confusing Variable Declarations and Assignment Statements 42 Trying to Invoke a Constructor Like a Method 50	How Many Days Have You Been Alive?  Working with Pictures 
3 Implementing Classes	Declaring a Constructor as `void` 92 Ignoring Parameter Variables 98 Duplicating Instance Variables in Local Variables 108 Providing Unnecessary Instance Variables 108 Forgetting to Initialize Object References in a Constructor 109	Implementing a Class 98 Making a Simple Menu  Drawing Graphical Shapes 116
4 Fundamental Data Types	Unintended Integer Division 144 Unbalanced Parentheses 144	Carrying out Computations 151 Computing the Volume and Surface Area of a Pyramid  Computing Travel Time 
5 Decisions	A Semicolon After the `if` Condition 184 Using == to Compare Strings 192 The Dangling `else` Problem 204 Combining Multiple Relational Operators 216 Confusing && and || Conditions 216	Implementing an `if` Statement 193 Extracting the Middle 

 Available online at `www.wiley.com/college/horstmann`.

Programming Tips

Special Topics

Computing & Society

Table 1 Implementing Classes

Example	Comments
`public class BankAccount { . . . }`	This is the start of a class declaration. Instance variables, methods, and constructors are placed inside the braces.
`private double balance;`	This is an instance variable of type `double`. Instance variables should be declared as `private`.
`public double getBalance() { . . . }`	This is a method declaration. The body of the method must be placed inside the braces.
`. . . { return balance; }`	This is the body of the `getBalance` method. The `return` statement returns a value to the caller of the method.
`public void deposit(double amount) { . . . }`	This is a method with a parameter variable (`amount`). Because the method is declared as `void`, it has no return value.
`. . . { balance = balance + amount; }`	This is the body of the `deposit` method. It does not have a `return` statement.
`public BankAccount() { . . . }`	This is a constructor declaration. A constructor has the same name as the class and no return type.
`. . . { balance = 0; }`	This is the body of the constructor. A constructor should initialize the instance variables.

There is one method left, `getBalance`. Unlike the `deposit` and `withdraw` methods, which modify the instance variable of the object on which they are invoked, the `getBalance` method returns a value:

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

We have now completed the implementation of the `BankAccount` class—see the code listing below. There is only one step remaining: testing that the class works correctly. That is the topic of the next section.

section_3/BankAccount.java

```
1  /**
2     A bank account has a balance that can be changed by
3     deposits and withdrawals.
4  */
5  public class BankAccount
6  {
7     private double balance;
8
9     /**
10       Constructs a bank account with a zero balance.
11    */
12    public BankAccount()
13    {
14       balance = 0;
15    }
```

H

I

Common Error 3.2

Ignoring Parameter Variables

A surprisingly common beginner's error is to ignore parameter variables of methods or constructors. This usually happens when an assignment gives an example with specific values. For example, suppose you are asked to provide a class `Letter` with a recipient and a sender, and you are given a sample letter like this:

```
Dear John:

I am sorry we must part.
I wish you all the best.

Sincerely,

Mary
```

Now look at this incorrect attempt:

```
public class Letter
{
   private String recipient;
   private String sender;

   public Letter(String aRecipient, String aSender)
   {
      recipient = "John"; // Error—should use parameter variable
      sender = "Mary"; // Same error
   }
   . . .
}
```

The constructor ignores the names of the recipient and sender arguments that were provided to the constructor. If a user constructs a

```
new Letter("John", "Yoko")
```

the sender is still set to `"Mary"`, which is bound to be embarrassing.

The constructor should use the parameter variables, like this:

```
public Letter(String aRecipient, String aSender)
{
   recipient = aRecipient;
   sender = aSender;
}
```

HOW TO 3.1

Implementing a Class

This "How To" section tells you how you implement a class from a given specification.

Problem Statement Implement a class that models a self-service cash register. The customer scans the price tags and deposits money in the machine. The machine dispenses the change.

F

G

E

U

Step 1 Find out which methods you are asked to supply.

In a simulation, you won't have to provide every feature that occurs in the real world—there are too many. In the cash register example, we don't deal with sales tax or credit card payments. The assignment tells you *which aspects* of the self-service cash register your class should simulate. Make a list of them:

- Process the price of each purchased item.
- Receive payment.
- Calculate the amount of change due to the customer.

Step 2 Specify the public interface.

Turn the list in Step 1 into a set of methods, with specific types for the parameter variables and the return values. Many programmers find this step simpler if they write out method calls that are applied to a sample object, like this:

```
CashRegister register = new CashRegister();
register.recordPurchase(29.95);
register.recordPurchase(9.95);
register.receivePayment(50);
double change = register.giveChange();
```

Now we have a specific list of methods:

- `public void recordPurchase(double amount)`
- `public void receivePayment(double amount)`
- `public double giveChange()`

To complete the public interface, you need to specify the constructors. Ask yourself what information you need in order to construct an object of your class. Sometimes you will want two constructors: one that sets all instance variables to a default and one that sets them to user-supplied values.

In the case of the cash register example, we can get by with a single constructor that creates an empty register. A more realistic cash register might start out with some coins and bills so that we can give exact change, but that is well beyond the scope of our assignment.

Thus, we add a single constructor:

- `public CashRegister()`

Step 3 Document the public interface.

Here is the documentation, with comments, that describes the class and its methods:

```
/**
   A cash register totals up sales and computes change due.
*/
public class CashRegister
{
   /**
      Constructs a cash register with no money in it.
   */
   public CashRegister()
   {
   }

   /**
      Records the sale of an item.
      @param amount the price of the item
   */
   public void recordPurchase(double amount)
   {
```