

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`. ❶
- Call the second constructor (because an argument is supplied in the constructor call).
- Set the parameter variable `initialBalance` to 1000. ❷
- Set the balance instance variable of the newly created object to `initialBalance`. ❸
- 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. ❹

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.

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:

```
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.

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.

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

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

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

Syntax 9.3 Constructor with Superclass_INITIALIZER

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

The superclass constructor is called first.

The constructor body can contain additional statements.

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

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

Constructors

A constructor definition has the form

```
modifiers ClassName(parameter1, parameter2, . . .)
    [throws ExceptionType1, ExceptionType2, . . .]
{
    body
}
```

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

```
new ClassName(parameterValue1, parameterValue2, . . .)
```

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

```
this(parameterValue1, parameterValue2, . . .)
```

For example,

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

It can call a constructor of its superclass with the syntax

```
super(parameterValue1, parameterValue2, . . .)
```

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

Arrays are constructed with the syntax

```
new ArrayType [ = { initializer1, initializer2, . . . }]
```

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

I-14 Index

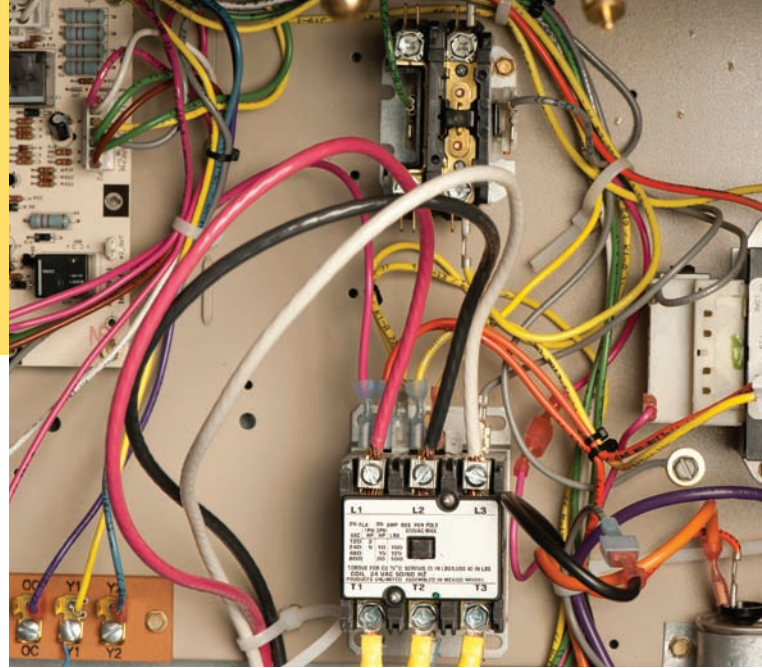
- java.util.logging.Level class, A-43
 - java.util.logging.Logger class
 - getGlobal method, 213
 - info method, 213
 - method summary, A-43. *See also specific methods.*
 - setLevel method, 213
 - java.util.Map<K, V> interface, A-39–40
 - java.util.NoSuchElementException, 537, A-40
 - java.util.PriorityQueue<E> class, A-40
 - java.util.Properties class, method summary, A-40
 - java.util.Queue class, A-40
 - java.util.QueueMap<E> interface, A-40
 - java.util.Random class
 - method summary, A-40–41. *See also specific methods.*
 - nextDouble method, 283
 - nextInt method, 283
 - java.util.Scanner class. *See also* java.io.File class.
 - hasNextDouble method, 219
 - hasNextInt method, 218–219
 - method summary, A-41. *See also specific methods.*
 - next method, 157
 - nextDouble method, 148
 - nextInt method, 147
 - java.util.Set<E> interface, A-41
 - java.util.TimeZone class, A-42
 - java.util.TreeMap<K, V> class, A-42
 - java.util.TreeSet<E> class
 - implementing sets, 681–682
 - interface references to, 685
 - method summary, A-42. *See also specific methods.*
 - javax.swing package, A-43–48
 - javax.swing.AbstractButton class, A-43
 - javax.swing.border package, A-48
 - javax.swing.border.EtchedBorder class, A-48
 - javax.swing.border.TitledBorder class, A-48
 - javax.swing.ButtonGroup class, 852–853, A-43
 - javax.swing.event package, A-48
 - javax.swing.event.ChangeEvent class, A-48
 - javax.swing.event.ChangeListener interface, A-48
 - javax.swing.ImageIcon class, A-44
 - javax.swing.JButton class, 486–489, A-44
 - javax.swing.JCheckBox class, A-44
 - javax.swing.JComboBox class, A-44
 - javax.swing.JComponent class, 63–65, A-44–45
 - javax.swing.JFileChooser class, 517–518, A-45
 - javax.swing.JFrame class
 - customizing frames, 844–845
 - method summary, A-45. *See also specific methods.*
 - setDefaultCloseOperation method, 61
 - javax.swing.JLabel class, 490–492, A-45
 - javax.swing.JMenu class, 863–868, A-45
 - javax.swing.JMenuBar class, A-46
 - javax.swing.JMenuItem class, 863–868, A-46
 - javax.swing.JOptionPane class
 - method summary, A-46. *See also specific methods.*
 - showInputDialog method, 162
 - showMessageDialog method, 163
 - javax.swing.JPanel class, 490–492, A-46
 - javax.swing.JRadioButton class, A-46
 - javax.swing.JScrollPane class, A-47
 - javax.swing.JSlider class, 870–874, A-47
 - javax.swing.JTextArea class, A-47
 - javax.swing.JTextField class, A-47
 - javax.swing.KeyStroke class, A-47
 - javax.swing.text package, A-49
 - javax.swing.text.JTextComponent class, 849–851, A-49
 - javax.swing.Timer class, 494–496, A-48
 - javax.xml.parsers package, A-49–50
 - javax.xml.parsers.DocumentBuilderFactory class, A-49–50
 - javax.xml.xpath package, A-50
 - javax.xml.xpath.XPath interface, A-50
 - javax.xml.xpath.
 - XPathExpressionException, A-50
 - javax.xml.xpath.XPathFactory class, A-50
 - JButton constructor, 490–492, A-44
 - JCheckBox constructor, 854, A-44
 - JComboBox constructor, A-44
 - JFileChooser constructor, A-45
 - JLabel constructor, A-45
 - JMenu constructor, 863–868, A-46
 - JMenuBar constructor, 863–868, A-46
 - JMenuItem constructor, 863–868, A-46
 - JRadioButton constructor, 852–853, A-46
 - JScrollPane constructor, A-47
 - JSlider constructor, A-47
 - JTextArea constructor, 849–851, A-47
 - JTextField constructor, 846–848, A-47
- K**
- Kahn, Bob, 454
 - Keyboard input, 147–148
 - keyPressed method, 504, A-16
 - keyReleased method, 500–501, A-16
 - keySet method, java.util.Map<K, V> interface, 687, A-39
 - keyTyped method, 501, A-16
 - “Knows-about” relationships. *See* Dependencies.
- L**
- Labels (user interface), 490
 - LargestInArray class, 329–330
 - Layout management
 - border layout, 842
 - complex layouts, 843–844
 - containers, 842
 - customizing frames with inheritance, 844–845
 - flow layout, 842
 - full code example, 844, 845
 - grid bag layout, 843
 - grid layout, 843
 - GUI builders, 862

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

3.1 INSTANCE VARIABLES AND ENCAPSULATION 82

Syntax 3.1: Instance Variable Declaration 83

3.2 SPECIFYING THE PUBLIC INTERFACE OF A CLASS 86

Syntax 3.2: Class Declaration 89

Common Error 3.1: Declaring a Constructor as void 92

Programming Tip 3.1: The javadoc Utility 92

3.3 PROVIDING THE CLASS IMPLEMENTATION 93

Common Error 3.2: Ignoring Parameter Variables 98

How To 3.1: Implementing a Class 98

Worked Example 3.1: Making a Simple Menu 

3.4 UNIT TESTING 102

Computing & Society 3.1: Electronic Voting Machines 104

3.5 PROBLEM SOLVING: TRACING OBJECTS 105

3.6 LOCAL VARIABLES 107

Common Error 3.3: Duplicating Instance Variables in Local Variables 108

Common Error 3.4: Providing Unnecessary Instance Variables 108

Common Error 3.5: Forgetting to Initialize Object References in a Constructor 109

3.7 THE THIS REFERENCE 109

Special Topic 3.1: Calling One Constructor from Another 112

3.8 SHAPE CLASSES 112

How To 3.2: Drawing Graphical Shapes 116

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

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

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

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(parameter1, parameter2, . . .)
    [throws ExceptionType1, ExceptionType2, . . .]
{
    body
}
```

A *method* has the form

```
Type methodName(parameter1, parameter2, . . .)
    [throws ExceptionType1, ExceptionType2, . . .]
{
    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 



Programming Tips



Special Topics



Computing & Society

Backup Copies	11		Computers Are Everywhere	5	
Choose Descriptive Variable Names	43	Testing Classes in an Interactive Environment	Computer Monopoly	60	
Learn By Trying	47				
Don't Memorize—Use Online Help	55				
The javadoc Utility	92	Calling One Constructor from Another	112	Electronic Voting Machines	104
Do Not Use Magic Numbers	139	Big Numbers	138	The Pentium Floating-Point Bug	146
Spaces in Expressions	145	Combining Assignment and Arithmetic	145		
Reading Exception Reports	162	Instance Methods and Static Methods	145		
		Using Dialog Boxes for Input and Output	162		
Brace Layout	184	The Conditional Operator	185	Denver's Luggage Handling System	195
Always Use Braces	184	The switch Statement	199		
Tabs	185	Block Scope	205	Artificial Intelligence	221
Avoid Duplication in Branches	186	Enumeration Types	206		
Hand-Tracing	203	Logging	212		
Make a Schedule and Make Time for Unexpected Problems	212	Short-Circuit Evaluation of Boolean Operators	217		
		De Morgan's Law	217		

- PrintWriter constructor
 - java.io.PrintStream class, A-20
 - java.io.PrintWriter class, A-20
 - Priority queues. *See also* Queues.
 - definition, 673
 - description, 693–694
 - full code example, 694
 - PriorityQueue constructor, A-40
 - Privacy issues, databases, 580
 - Private method implementation, 44
 - Probing sequence, 749
 - Problem solving. *See* Backtracking.
 - Problem statements, converting to pseudocode (How To), 151–154
 - Product.java class, 578–579
 - Program comparator, 606
 - Program development, examples.
 - See* “Hello, World” program;
 - Printing invoices.
 - Programming. *See also* Applets; Java language.
 - compilers, 6
 - definition, 2
 - getting started. *See* “Hello, World” program.
 - high-level languages, 6
 - machine code, 6
 - scheduling time for, 212
 - Prompting
 - for first match, with loops, 274
 - for input, 147–148
 - protected access feature, 442–443
 - Pseudocode
 - for algorithms, 19–20
 - writing (How To), 151–154
 - Pseudorandom numbers, 284
 - Public interfaces, classes
 - class declaration, 89
 - commenting, 89–92
 - constructors, specifying, 87–88
 - definition, 89
 - methods, specifying, 86–87
 - overview, 43–44
 - syntax, 89
 - uses for, 89
 - Public methods, interface types, 470
 - push method, java.util.Stack<E> class, 692
 - put method, java.util.Map<K, V> interface, 585, A-40
 - Puzzles, solving. *See* Backtracking.
 - Pyramids, computing volume and surface area (Worked Example), 154
- Q**
- Queen.java class, 618–619
 - QuestionDemo1.java class, 425
 - QuestionDemo2.java class, 432–433
 - QuestionDemo3.java class, 438–439
 - Question.java class, 424
 - Queues. *See also* Priority queues.
 - adding/removing elements, 693
 - definition, 673
 - description, 693
 - FIFO (first in, first out), 693
 - full code example, 694
 - getting head element, 693
 - of waiting customer (Worked Example), 702
 - Queues, implementing
 - as circular arrays, 738–740
 - efficiency, 739*t*
 - as linked lists, 737–738
 - sample code, 739–740
 - Quicksort, 646–647
 - Quotation marks (“”), string delimiters, 12
- R**
- Radiation therapy incidents, 359
 - Radio buttons
 - borders, 852–853
 - definition, 852
 - grouping, 852–853
 - Random access
 - code samples, 890–892
 - definition, 887
 - file pointers, 888
 - overview, 887–890
 - Random constructor, A-40
 - Random numbers
 - finding approximate solutions, 285–286
 - generating, 283–284
 - Monte Carlo method, 285–286
 - pseudorandom numbers, 284
 - RandomAccessFile constructor, A-21
 - read method, java.io.InputStream class, 884–886, 887, A-19
 - readChar method, java.
 - io.RandomAccessFile class, A-21
 - readDouble method, java.
 - io.RandomAccessFile class, 889–892
 - readDouble method, java.
 - io.RandomAccessFile class, A-21
 - Reading input. *See also* java.util.
 - Scanner class; Text files, reading and writing; Writing output.
 - into arrays, 328–330
 - binary. *See* Binary I/O.
 - characters, 520
 - characters from a string, 522
 - classifying characters, 520–521
 - from a console, 157
 - converting strings to numbers, 522–523
 - definition, 4
 - dialog boxes, 162–163
 - error handling, 545–549
 - full code example, 162
 - from a keyboard, 147–148
 - lines, 521–522
 - mixed input types, 523–524
 - from a prompt, 147–148
 - prompting for, 147–148
 - random access, 887–893
 - readers, 882–883
 - reading, 147–148
 - sequential access, 887
 - strings, 157
 - text. *See* Text I/O.
 - validating numbers, 523
 - white space, consuming, 519–520
 - words, 519–520
 - readInt method, java.
 - io.RandomAccessFile class, 889–892, A-21
 - readObject method
 - java.io.ObjectInputStream class, A-20
 - java.io.ObjectOutputStream class, 893–895
 - Rectangle constructor, A-15
 - RectangleComponent.java class, 64–65, 494–495
 - RectangleFrame.java class, 495–496
 - Rectangles
 - comparing, 190
 - drawing, 61–65
 - RectangleViewer.java class, 64–65, 496

Table 1 Implementing Classes	
Example	Comments
<code>public class BankAccount { . . . }</code>	This is the start of a class declaration. Instance variables, methods, and constructors are placed inside the braces.
<code>private double balance;</code>	This is an instance variable of type <code>double</code> . Instance variables should be declared as <code>private</code> .
<code>public double getBalance() { . . . }</code>	This is a method declaration. The body of the method must be placed inside the braces.
<code>. . . { return balance; }</code>	This is the body of the <code>getBalance</code> method. The return statement returns a value to the caller of the method.
<code>public void deposit(double amount) { . . . }</code>	This is a method with a parameter variable (<code>amount</code>). Because the method is declared as <code>void</code> , it has no return value.
<code>. . . { balance = balance + amount; }</code>	This is the body of the <code>deposit</code> method. It does not have a return statement.
<code>public BankAccount() { . . . }</code>	This is a constructor declaration. A constructor has the same name as the class and no return type.
<code>. . . { balance = 0; }</code>	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     }
```

grep program, 526
 Grid bag layout, 843
 Grid layout, 843
 GridLayout constructor, 843, A-15
 Grouping radio buttons, 852–853
 grow method, java.awt.Rectangle class, A-15
 GUI (Graphical User Interface), components. *See also* Layout management; *specific components*.
 default size, 845
 making selections. *See* Check boxes; Combo boxes; Radio buttons.
 GUI builders, 862
 Gutenberg project, 454

H

Halt checker, 606
 Halting problem, 606–607
 Hand tracing
 animation, 251
 arithmetic operations, 154–156
 description, 203–204
 loops, 249–253
 objects, 105–107
 Hardware, definition, 2
 “Has-a” relationships. *See* Aggregation.
 Hash codes
 collisions, 690–691
 description, 690–691, 741
 full code example, 691
 Hash functions, 690–691
 Hash tables, description, 741–742
 Hash tables, implementing
 adding/removing elements, 743–744
 buckets, 742
 collisions, 741
 efficiency, 745*t*
 finding elements, 743
 hash codes, 741
 iterating over hash tables, 744–749
 linear probing, 749
 open addressing, 742, 749
 probing sequence, 749
 sample code, 745–749
 separate chaining, 742

hashCode method, 681
 HashMap constructor, A-38
 HashSet constructor, A-38
 HashSetDemo.java class, 748–749
 HashSet.java class, 745–748
 hasNext method
 java.util.Iterator<E> interface, 677, 682, A-38
 java.util.Scanner class, A-41
 hasNextDouble method, java.util.Scanner class, 219, A-41
 hasNextInt method, java.util.Scanner class, 218–219, A-41
 hasNextLine method, java.util.Scanner class, A-41
 hasPrevious method, java.util.ListIterator<E> interface, 678, A-39
 HeapDemo.java class, 802–803
 Heaps. *See also* Binary search; Tree concepts.
 adding nodes, 794–795. *See also* Heapsort algorithm.
 definition, 793
 removing nodes, 796–797. *See also* Heapsort algorithm.
 sample code, 798–803
 sorting nodes. *See* Heapsort algorithm.
 storing in arrays, 798
 Heapsort algorithm, 804–808
 HeapSorter.java, 806–808
 Height of trees, 763
 “Hello, World” program
 analyzing, 12–14
 source code, 12
 writing, 8–11
 HelloPrinter.java class, 12
 Help, online, 55. *See also* Documentation.
 Hexadecimal numbers, A-73
 High-level languages, 6
 History of computers
 Altair 8800, 406
 Apple II, 406
 Babbage's Difference Engine, 652
 corporate monopolies, 60
 first kit computer, 406
 first programmer, 652
 hardware evolution, 5
 IBM, 60

microprocessors, 406
 Microsoft, 60
 personal computing, 406
 standardization, 680
 Turing machine, 606
 Univac Corporation, 60
 VisiCalc, 406

Hoff, Marcian E., 406
 Houston, Frank, 359
 HTML (Hypertext Markup Language)
 attributes, A-88–89
 entities, A-89–90
 overview, A-86
 tag summary, A-87–88
 Huffman trees, 767, 770

I

IBM, history of computers, 60
 IEEE floating-point numbers, A-72
 IETF (Internet Engineering Task Force), 680
 if statements. *See also* switch statements.
 animation, 196, 200
 dangling else problem, 204–205
 definition, 180
 duplicate code in branches, 186
 ending with a semicolon, 184–185
 flowchart for, 181
 implementing (How To), 193–195
 input validation, 218–220
 multiple alternatives, 196–199
 nesting, 200–203
 sample program, 182–183
 syntax, 182
 IllegalArgumentException, 534–535, A-23
 ImageIcon constructor, A-44
 Immutable classes, 383–384
 implements reserved word, 466–468
 Implicit parameters, 110
 import directive, 398–399, 401
 Importing
 classes from packages, 54
 packages, 401
 in method, java.lang.System class, 147
 Income tax computation, 200–203
 Income tax rate schedule, 200*t*
 Indefinite loops, 254

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.



- Exception handling (*continued*)
 - catch clause, 536–537, 542–543
 - catching exceptions, 536–537
 - checked exceptions, 537–539
 - definition, 534
 - designing exception types, 540–541
 - exception handlers, 536–537
 - finally clause, 539–540, 542–543
 - full code example, 539
 - internal errors, 537
 - quenching exceptions, 542
 - throw early, catch late, 542
 - throwing exceptions, 534–535, 543
 - throws clause, 538–539
 - try blocks, 536–537
 - try statement, 536–537, 542–543
 - unchecked exceptions, 537
- Exception reports, 162
- Exceptions
 - definition, 15
 - syntax, A-62
- Exclamation point, equal (!=), relational operator, 187*t*
- Exclamation point (!), *not* operator, 215
- Executable algorithms, 17
- execute method
 - `java.sql.PreparedStatement` class, A-32
 - `java.sql.Statement` class, A-34
- executeQuery method
 - `java.sql.PreparedStatement` class, A-32
 - `java.sql.Statement` class, A-34
- executeUpdate method
 - `java.sql.PreparedStatement` class, A-32
 - `java.sql.Statement` class, A-34
- exists method, `java.io.File` class, A-19
- exit method, `java.lang.System` class, A-28
- exp method, `java.lang.Math` class, 142*t*, A-25
- Explicit parameters, 110
- Expression trees, 767
- `ExpressionCalculator.java` class, 613–614
- Expressions, 139, 145, A-54
- `ExpressionTokenizer.java` class, 612–613
- Extending classes. *See* Inheritance.
- extends reserved word, 427
- F**
- Face, drawing, 69–70
- `FaceComponent.java` class, 69–70
- Fibonacci sequence, 598–603
- FIFO (first in, first out), 693
- Fifth-Generation Project, 221
- `File` class, 514
- `File` constructor, A-18
- File dialog boxes, 517–518
- File names, as string literals, 517
- File pointers, 888
- `FileInputStream` constructor, A-19
- `FileNotFoundException`, 515–516, 537
- `FileOutputStream` constructor, A-19
- Files
 - finding, with recursion, 596
 - overview, 10–11. *See also* Folders.
 - source, A-80
- `fill` method, `java.awt.Graphics2D` class, 68, A-14
- Filling arrays, 322
- Fills, drawing, 68
- Final classes, 442
- finally clause, 539–540, 542–543
- Financial calculations, data type for, 134
- First kit computer, 406
- First programmer, 652
- Fixed length arrays, 315
- Flags. *See* Boolean operators.
- Flags, drawing, 116–119
- `float` data type, 132*t*
- Floating-point numbers
 - assigning to integer variables, 134
 - comparing, 188
 - converting to integer, 142–143. *See also* cast operator.
 - description, A-72
 - mixing with integer, 139
 - precision, 133–134
- `floor` method, `java.lang.Math` class, 142*t*, A-25
- Flow layout, 842
- Flowcharting, loops, 263
- Flowcharts, 207–210. *See also* Storyboards.
- `FlowLayout` constructor, A-14
- Folders, 10. *See also* Files.
- `Font` constructor, A-14
- `FontFrame2.java` class, 865–868
- `FontFrame.java` class, 855–858
- `FontViewer2.java` class, 865
- `FontViewer.java` class, 855
- for loops, 254–259, 261
- for statement, coding guidelines, A-83
- Format flags, 524*t*
- `format` method
 - `java.lang.String` class, A-27
 - `java.text.DateFormat` class, A-34
- Format specifiers, 148–150, 525
- Format types, 525*t*
- Formatting output
 - format flags, 524*t*
 - format specifiers, 148–150, 525
 - format types, 525*t*
 - full code example, 525
 - overview, 524
- `forName` method, `java.lang.Class` class, A-22
- Four Queens problem, 616–620
- Frame windows, 61–62, 65
- Frames, customizing with inheritance, 844–845
- G**
- Garbage collection, 107
- Generic arrays, 834–835
- Generic classes
 - array lists, 348
 - array store exception, 829–830
 - binary search tree (Worked Example), 835
 - code samples, 823–824
 - constraining type parameters, 827–828
 - full code example, 821, 828
 - implementing, 821–825
 - inheritance, 829
 - linked lists, 676
 - naming type variables, 822
 - syntax, 822
 - type parameters, 820–821

Dongles, 253
 Dot (.), name syntax, 403–404
 Double constructor, A-23
 double data type
 definition, 132*t*
 for financial calculations, 134
 overflow, 133
 precision, 133
 Double-black problem, 789–792
 doubleValue method, java.lang.Double class, A-23
 Doubly-linked lists, 678, 730
 Dr. Java environment, 56
 draw method, java.awt.Graphics2D class, 64–65, A-14
 Drawing. *See also* Graphical applications; *specific shapes*.
 a car, 112–115
 circles, 66–67
 colors, 68
 on a component, 62–65
 ellipses, 66–67
 a face, 69–70
 fills, 68
 a flag (How To), 116–119
 graphical shapes (How To), 116–119
 lines, 67
 rectangles, 61–65
 shape classes, 112–116
 drawLine method, java.awt.Graphics class, A-14
 drawString method, java.awt.Graphics2D class, 67, A-14
 Dynamic method lookup, 438, A-58

E

E constant, java.lang.Math class, 135
 Earthquake descriptions
 full code example, 198
 Loma Prieta quake, 196
 Richter scale, 196*t*
 ECMA (European Computer Manufacturers Association), 680
 Editing pictures (Worked Example), 57
 Editors, definition, 8
 Eight Queens problem, 615–620
 EightQueens.java class, 619–620
 Electronic voting machines, 104

ElevatorSimulation2.java class, 219–220
 ElevatorSimulation.java, 182–183
 Ellipse2D.Double constructor, A-17
 Ellipses, drawing, 66–67
 else statements, dangling else problem, 204–205
 Empty string *vs* null reference, 191
 Empty strings, 156
 Empty trees, 765
 EmptyFrameViewer.java class, 62
 Encapsulation, instance variables, 84–86
 Encryption
 algorithms, 533
 Caesar cipher, 884–886
 Encryption keys, 884
 Enhanced for loop, 321–322, 349–350
 ENIAC (electronic numerical integrator and computer), 5
 Enumeration types, 206–207, A-56–57. *See also* Arrays.
 EOFException constructor, A-18
 “Equal exit cost” rule, 787–788
 Equal sign (=), assignment operator, 40–41
 Equal signs (==), relational operator
 comparing object references, 190
 comparing strings, 189, 192–193
 syntax, 187*t*
 testing for null, 191
 equals method
 comparing objects, 450–451
 inheritance, 454
 java.awt.Rectangle class, 190
 java.lang.Object class, A-26
 java.lang.String class, 188–190, A-27
 equalsIgnoreCase method, java.lang.String class, A-27
 Erasing type parameters, 831–833
 Error handling, input errors, 545–549. *See also* Exception handling.
 Error messages
 logging, 212–213
 reading exception reports, 162
 stack trace, 162
 trace messages, 212–213

Errors. *See also specific errors*.
 arrays, 318
 compile-time, 15
 dangling else problem, 204–205
 declaring constructors as void, 92
 declaring instance variables in
 local variables, 108
 “Division by zero,” 15
 exceptions, 15
 full code example, 15
 ignoring parameter values, 98
 logic, 15
 in loops, 247–249
 misspelling words, 16
 roundoff, 134, 188–189
 run-time, 15
 syntax, 15
 unbalanced parentheses, 144
 uninitialized object references, 109
 unintended integer division, 144
 unnecessary instance variables, 108–109
 Errors, diagnosing, encapsulation, 85
 ESA (European Space Agency), 544
 Escape sequences, 158
 EtchedBorder constructor, A-48
 European Computer Manufacturers Association (ECMA), 680
 evaluate method, javax.xml.xpath.XPath class, method summary, A-50
 Evaluator.java class, 611–612
 Event handling, user interface events, 485
 Event listeners
 calling listener methods, 490
 containers as, 493
 definition, 485
 event source, 485
 forgetting to attach, 493
 inner classes for, 487–489
 Event-controlled loops, 254
 Events
 button press, 491
 definition, 485
 timer, 494–496
 Exception handlers, 536–537
 Exception handling. *See also* Error handling.
 animation, 537

- Testing
 - black-box, 210
 - boundary cases, 210–211
 - classes, 56
 - code coverage, 210
 - for null, 190–191
 - objects. *See* Mock objects.
 - programs, 55–56
 - regression, 356–358
 - unit, 102–103
 - unit test frameworks, 407–408
 - white-box, 210
 - Text areas, 849–851, A-47
 - Text fields, 846–848, A-47
 - Text files, reading and writing, 514–516, 530–532. *See also* Reading input; Writing output.
 - Text I/O
 - code sample, 847–848, 850–851
 - multiple lines, 849–851
 - readers, 882–883
 - scroll bars, 850
 - single line, 846–848
 - text areas, 849–851
 - text fields, 846–848
 - writers, 882–883
 - Therac-25 incidents, 359
 - this reference, 109–111
 - Throw early, catch late, 542
 - throw statement, 534–535
 - Throwable constructor, A-29
 - Throwing exceptions, 534–535, 542, 543
 - throws clause, 538–539
 - Tilde (~), unary negation operator, A-74
 - Tiling a floor, algorithm for, 22–23
 - Timer constructor, A-48
 - Timer events, 494–496
 - TitledBorder constructor, A-48
 - toDegrees method, java.lang.Math class, 142*t*, A-26
 - Tokens, 609
 - toLowerCase method, java.lang.String class, A-28
 - toRadians method, java.lang.Math class, 142*t*, A-26
 - toString method
 - java.lang.Integer class, A-24
 - java.lang.Object class, A-27
 - java.util.Arrays class, A-36
 - toString method, java.util.Arrays class
 - inheritance, 453
 - inserting element separators, 323–324
 - overriding, 448–449
 - Total.java class, 515–516
 - Totals, computing with loops, 272
 - toUpperCase method, java.lang.String class, 43–44, A-28
 - Trace messages, 212–213
 - Tracing. *See* Hand tracing.
 - Tracing recursive methods, 590, 592–593, 599–601
 - Transistors, 3
 - translate method, java.awt.Rectangle class, 51, 55–56, A-15
 - Travel time computation (Worked Example), 156
 - Tree concepts. *See also* Binary search trees; Binary trees; Heaps; Red-black trees.
 - ancestors, 763
 - child nodes, 763
 - computing the size, 765
 - descendants, 763
 - empty trees, 765
 - full code example, 765
 - height, 763
 - implementing a tree, 764
 - interior nodes, 763
 - leaf nodes, 763
 - nodes, 763
 - overview, 762
 - parent nodes, 763
 - paths, 763
 - root nodes, 763
 - sibling nodes, 763
 - subtrees, 763
 - Tree iterators, 785
 - Tree traversal
 - breadth-first search, 783–785
 - depth-first search, 783–785
 - full code example, 785
 - inorder, 780–781
 - postorder, 781–782
 - preorder, 781–782
 - tree iterators, 785
 - visitor pattern, 782–783
 - TreeMap constructor, A-42
 - TreeSet constructor, A-42
 - Triangle numbers, 588–593
 - Triangle, java class, 590–591
 - TriangleTester, java class, 591
 - trim method, 522
 - Trimming white space, 522
 - Try blocks, 536–537
 - try statement, 536–537, 542–543
 - Turing, Alan, 606
 - Turing machine, 606–607
 - Two-dimensional arrays. *See* Arrays, two-dimensional.
 - Two's complement integers, A-71
 - Type parameters
 - erasing, 831–833
 - full code example, 833
 - in static context, 834
 - Type parameters, constraining
 - full code example, 831
 - generic classes, 827–828
 - generic methods, 827–828
 - wildcard types, 830–831
 - Type variables, naming in generic classes, 822
 - Types of data. *See* Data types.
- U**
- UML (Unified Modeling Language).
 - See also* CRC (class-responsibility-collaboration) cards.
 - attributes, 567
 - definition, 382
 - diagrams, A-76–78
 - methods, 567
 - in program design (How To), 566–567
 - relationship symbols, 566*t*, A-77
 - sample program, 572–573
 - The Unified Modeling Language User Guide*, 569
 - Unambiguous algorithms, 17
 - Unchecked exceptions, 537
 - Undeclared variables, 42
 - Underscore (_), in variable names, 39
 - Unfilled arrays, 318
 - Unicode
 - encoding, 519
 - international alphabets, 163
 - Latin and Latin-1 subsets, A-1*t*–3*t*

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)
    {
    }
}
```