Write method and constructor headers that describe the public interface of a class.
- In order to implement a class, you first need to know which methods are required.
- In a method header, you specify the return type, method name, and the types and
  names of the parameters.
- Constructors set the initial data for objects. The constructor name is always the
  same as the class name.

Write class documentation in javadoc format.
- Use documentation comments to describe the classes and public methods of your
  programs.
- Provide documentation comments for every class, every method, every parameter,
  and every return value.

Provide the private implementation of a class.
- The private implementation of a class consists of instance variables, and the bodies
  of constructors and methods.

Write tests that verify that a class works correctly.
- A unit test verifies that a class works correctly in isolation, outside a complete
  program.

Compare lifetime and initialization of instance, local, and parameter variables.
- Local variables are declared in the body of a method.
- When a method exits, its local variables are removed.
- Instance variables are initialized to a default value, but you must initialize local
  variables.

Recognize the use of the implicit parameter in method declarations.
- Use of an instance variable name in a method denotes the instance variable of the
  implicit parameter.
- The this reference denotes the implicit parameter.
- A method call without an implicit parameter is applied to the same object.

Implement classes that draw graphical shapes.
- It is a good idea to make a class for any part of a drawing that can occur more
  than once.
- To figure out how to draw a complex shape, make a sketch on graph paper.

Media Resources

- **Worked Example** Making a Simple Menu
- Lab Exercises
- **Animation** Lifetime of Variables
- Practice Quiz
- Code Completion Exercises

Review Exercises

- **R3.1** What is the interface of a class? How does it differ from the implementation of a
  class?
- **R3.2** What is encapsulation? Why is it useful?
- **R3.3** Instance variables are a part of the hidden implementation of a class, but they aren’t
  actually hidden from programmers who have the source code of the class. Explain to
  what extent the private reserved word provides information hiding.
- **R3.4** Consider a class `Grade` that represents a letter grade, such as A+ or B. Give two
  choices of instance variables that can be used for implementing the `Grade` class.
- **R3.5** Consider a class `Time` that represents a point in time, such as 9 A.M. or 3:30 P.M. Give
  two different sets of instance variables that can be used for implementing the `Time`
  class.
- **R3.6** Suppose the implementor of the `Time` class of Exercise R3.5 changes from one imple-
  mentation strategy to another, keeping the public interface unchanged. What do the
  programmers who use the `Time` class need to do?
- **R3.7** You can read the `values` instance variable of the `Counter` class with the `getValues`
  accessor method. Should there be a `setValues` mutator method to change it? Explain why
  or why not.
- **R3.8** a. Show that the `BankAccount(double initialBalance)` constructor is not strictly nec-
  essary. That is, if we removed that constructor from the public interface, how could a
  programmer still obtain `BankAccount` objects with an arbitrary balance?
  b. Conversely, could we keep only the `BankAccount(double initialBalance)` con-
     structor and remove the `BankAccount()` constructor?
- **R3.9** Why does the `BankAccount` class not have a reset method?
- **R3.10** What happens in our implementation of the `BankAccount` class when more money is
  withdrawn from the account than the current balance?
- **R3.11** What is the `this` reference? Why would you use it?
- **R3.12** What does the following method do? Give an example of how you can call the
  method.

```java
public class BankAccount {
    public void mystery(BankAccount that, double amount) {
        this.balance = this.balance - amount;
        that.balance = that.balance + amount;
    }
    ...// Other bank account methods
```
** R3.13 Suppose you want to implement a class `DepositAccount`. A time deposit account has a fixed interest rate that should be set in the constructor; together with the initial balance. Provide a method to get the current balance. Provide a method to add the earned interest to the account. This method should have no parameters because the earned interest rate is already known. It should have no return value because you already provided a method for obtaining the current balance. It is not possible to deposit additional funds into this account. Provide a withdraw method that removes the entire balance. Partial withdrawals are not allowed.

** R3.14 Consider the following implementation of a class `Square`:

```java
public class Square {
    private int sideLength;
    private int area; // Not a good idea
    public Square(int length) {
        sideLength = length;
    }
    public int getArea() {
        area = sideLength * sideLength;
        return area;
    }
}
```

Why is it not a good idea to introduce an instance variable for the area? Rewrite the class so that area is a local variable.

** R3.15 Consider the following implementation of a class `Square`:

```java
public class Square {
    private int sideLength;
    private int area;
    public Square(int initialLength) {
        sideLength = initialLength;
        area = sideLength * sideLength;
    }
    public int getArea() {
        return area;
    }
    public void grow() { sideLength = 1 + sideLength; }
}
```

What error does this class have? How would you fix it?

** R3.16 Provide a unit test class for the `Counter` class in Section 3.1.

** R3.17 Read Exercise P3.7, but do not implement the `Car` class yet. Write a tester class that tests a scenario in which gas is added to the car, the car is driven, more gas is added, and the car is driven again. Print the actual and expected amount of gas in the tank.

** G R3.18 Suppose you want to extend the car viewer program in Section 3.9 to show a suburb scene, with several cars and houses. Which classes do you need?

** G R3.19 Explain why the calls to the `getwidth` and `getheight` methods in the `CarComponent` class have no explicit parameter.

** G R3.20 How would you modify the `Car` class in order to show cars of varying sizes?

---

** Programming Exercises

** P3.1 Write a `BankAccountTester` class whose main method constructs a bank account, deposits $1,000, withdraws $500, withdraws another $400, and then prints the remaining balance. Also print the expected result.

** P3.2 Add a method

```java
public void addInterest(double rate)
```
to the `BankAccount` class that adds interest at the given rate. For example, after the statements

```java
BankAccount nonSavings = new BankAccount(1000);
nonSavings.addInterest(10); // 10% interest
```
the balance in nonSavings is $1,100. Also supply a `BankAccountTester` class that prints the actual and expected balance.

** P3.3 Write a class `SavingsAccount` that is similar to the `BankAccount` class, except that it has an added instance variable `interest`. Supply a constructor that sets both the initial balance and the interest rate. Supply a method `addInterest` (with no explicit parameter) that adds interest to the account. Write a `SavingsAccountTester` class that constructs a savings account with an initial balance of $1,000 and an interest rate of 10%. Then supply the `addInterest` method and print the resulting balance. Also compute the expected result by hand and print it.

** P3.4 Add a feature to the `CashRegister` class for computing sales tax. The tax rate should be supplied when constructing a `CashRegister` object. Add `recordPurchase` and `getTotalTax` methods. (Amounts added with `recordPurchase` are not taxable.) The `getTotalTax` method should correctly reflect the sales tax that is charged on taxable items.

** P3.5 After closing time, the store manager would like to know how much business was transacted during the day. Modify the `CashRegister` class to enable this functionality. Supply methods `getSalesTotal` and `getSalesCount` to get the total amount of all sales and the number of sales. Supply a method `reset` that resets any counters and totals so that the next day's sales start from zero.

** P3.6 Implement a class `Employee`. An employee has a name (a `String`), a base salary (a `double`), and a name (a `String`). Provide a constructor with two parameters

```java
public Employee(String name, double salary)
```
and methods

```java
public String getName() public double getSalary() public void raiseSalary(double amount)
```
These methods return the name and salary, and raise the employee's salary by a certain percentage. Sample usage:

```java
Employee harry = new Employee("Hacker, Harry", 50000);
harry.raiseSalary(10); // Harry gets a 10% raise
```

Supply an `EmployeeTester` class that tests all methods.

**P3.7** Implement a class `Car` with the following properties. A car has a certain fuel efficiency (measured in miles/gallon or liters/km—pick one) and a certain amount of fuel in the gas tank. The efficiency is specified in the constructor, and the initial fuel level is 0. Supply a method `drive` that simulates driving the car for a certain distance, reducing the amount of gasoline in the fuel tank. Also supply methods `getGasTank`, returning the current amount of gasoline in the fuel tank, and `addGas`, to add gasoline to the fuel tank. Sample usage:

```java
Car myHybrid = new Car(50); // 50 miles per gallon
myHybrid.addGas(20); // Add 20 gallons
myHybrid.drive(100); // Drive 100 miles
double gasLeft = myHybrid.getGasInTank(); // Get gas remaining in tank
```

You may assume that the `drive` method is never called with a distance that consumes more than the available gas. Supply a `CarTester` class that tests all methods.

**P3.8** Implement a class `Student`. For the purpose of this exercise, a student has a name and a total quiz score. Supply an appropriate constructor and methods `getName()`, `getTotalScore()`, and `getAverageScore()`. To compute the latter, you also need to store the number of quizzes that the student took.

Supply a `StudentTest` class that tests all methods.

**P3.9** Implement a class `Product`. A product has a name and a price, for example, new product("Toaster", 29.95). Supply methods `getName()`, `getPrice()`, and `reducePrice()`. Supply a program `ProductPriceTest` that makes two products, prints the name and price, reduces their prices by $5.00, and then prints the prices again.

**P3.10** Provide a class for authoring a simple letter. In the constructor, supply the names of the sender and the recipient:

```java
public Letter(String from, String to)
```

Supply a method to add a line of text to the body of the letter:

```java
public String getTest()
```

that returns the entire text of the letter. The text has the form:

Dear recipient name:

first line of the body

second line of the body

...

last line of the body

Sincerely,

blank line

sender name

Also supply a class `LetterPrinter` that prints this letter.

```java
Dear John:
I am sorry we must part.
I wish you all the best.
Sincerely,
Mary
```

Construct an object of the `letter` class and call `addLine` twice.

Hints: (1) Use the `concat` method to form a longer string from two shorter strings.
(2) The special string "\n" represents a new line. For example, the statement

```java
body = body.concat("Sincerely.").concat("\n");
```

adds a line containing the string "Sincerely," to the body.

**P3.11** Write a class `Bug` that models a bug moving along a horizontal line. The bug moves either to the right or left. Initially, the bug moves to the right, but it can turn to change its direction. In each move, its position changes by one unit in the current direction. Provide a constructor

```java
public Bug(int initialPosition)
```

and methods

```java
public void turn()
public void move()
public int getPosition()
```

Sample usage:

```java
Bug bugsy = new Bug(30);
bugsy.move(); // now the position is 11
bugsy.turn();
bugsy.move(); // now the position is 10
```

Your `BugTester` should construct a bug, make it move and turn a few times, and print the actual and expected position.

**P3.12** Implement a class `Path` that models a moth flying across a straight line. The moth has a position, the distance from a fixed origin. When the moth moves toward a point of light, its new position is halfway between its old position and the position of the light source. Supply a constructor

```java
public Moth(double initialPosition)
```

and methods

```java
public void moveLight(double lightPosition)
public double getPosition()
```

Your `MothTester` should construct a moth, move it toward a couple of light sources, and check that the moth's position is as expected.

**P3.13** Implement a class `RoachPopulation` that simulates the growth of a roach population. The constructor takes the size of the initial roach population. The breed method simulates a period in which the roaches breed, which doubles their population. The sprays method simulates spraying with insecticide, which reduces the population by
Chapter 3  Implementing Classes

10 percent. The getRooches method returns the current number of roaches. A program called RoachSimulation simulates a population that starts out with 10 roaches, breeds, sprays, and prints the roach count. Repeat three more times.

** P3.14 Implement a VotingMachine class that can be used for a simple election. Have methods to clear the machine state, to vote for a Democrat, to vote for a Republican, and to get the tallies for both parties. Extra credit if your program gives the nod to your favored party if the votes are tallied after 8 P.M. on the first Tuesday in November, but acts normally on all other dates. (Hint: Use the GregorianCalendar class—see Programming Project 2.1.)

**G P3.15 Draw a "bull's eye"—a set of concentric rings in alternating black and white colors. Hint: Fill a black circle, then fill a smaller white circle on top, and so on.

Your program should be composed of classes Bullseye, BullseyeComponent, and BullseyeViewer.

**G P3.16 Write a program that draws a picture of a house. It could be as simple as the accompanying figure, or if you like, make it more elaborate (3-D, skyscraper, marble columns in the entryway, whatever).

[Image of a house]

Implement a class House and supply a method draw(GraphicContext g2) that draws the house.

**G P3.17 Extend Exercise P3.16 by supplying a House constructor for specifying the position and size. Then populate your screen with a few houses of different sizes.

**G P3.18 Change the car viewer program in Section 3.9 to make the cars appear in different colors. Each car object should store its own color. Supply modified Car and CarComponent classes.

**G P3.19 Change the Car class so that the size of a car can be specified in the constructor. Change the CarComponent class to make one of the cars appear twice the size of the original example.

**G P3.20 Write a program to plot the string "HELLO", using only lines and circles. Do not call drawString, and do not use System.out. Make classes Letter0, Letter1, Letter2, and Letter3.

Programming Projects

**G P3.21 Write a program that displays the Olympic rings. Color the rings in the Olympic colors.

Provide a class OlympicRingViewer and a class OlympicRingComponent.

**G P3.22 Make a bar chart to plot the following data set. Label each bar. Make the bars horizontal for easier labeling.

<table>
<thead>
<tr>
<th>Bridge Name</th>
<th>Longest Span (ft)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Golden Gate</td>
<td>4,200</td>
</tr>
<tr>
<td>Brooklyn</td>
<td>1,395</td>
</tr>
<tr>
<td>Delaware Memorial</td>
<td>2,150</td>
</tr>
<tr>
<td>Mackinac</td>
<td>3,800</td>
</tr>
</tbody>
</table>

Provide a class BarChartViewer and a class BarChartComponent.

Project 3.1  In this project, you will enhance the BankAccount class and see how abstraction and encapsulation enable evolutionary changes to software.

Begin with a simple enhancement: charging a fee for every deposit and withdrawal. Supply a mechanism for setting the fee and modify the deposit and withdraw methods so that the fee is levied. Test your resulting class and check that the fee is computed correctly.

Now make a more complex change. The bank will allow a fixed number of free transactions (deposits or withdrawals) every month, and charge for transactions exceeding the free allotment. The charge is not levied immediately but at the end of the month.

Supply a new method deductMonthlyCharge to the BankAccount class that deducts the monthly charge and resets the transaction count. (Hint: Use Math.round transaction count, free transaction count) in your computation.

Produce a test program that verifies that the fees are calculated correctly over several months.

Project 3.2  In this project, you will explore an object-oriented alternative to the "Hello, World" program in Chapter 1.

Begin with a simple Greater class that has a single method, sayHello. That method should return a string, not print it. Use BlueJ to create two objects of this class and invoke their sayHello methods.
That is boring—of course, both objects return the same answer. Enhance the Greeter class so that each object produces a customized greeting. For example, the object constructed as new Greeter("Dave") should say "Hello, Dave". (Use the.concat method to combine strings to form a longer string, or peek ahead at Section 4.6 to see how you can use the + operator for the same purpose.)

Add a method sayGoodbye to the Greeter class. Finally, add a method refuel() to the Greeter class. It should return a string such as "I am sorry, Dave. I am afraid I can't do that."

Test your class in BlueJ. Make objects that greet the world and Dave, and invoke methods on them.

Answers to Self-Check Questions

1. public void reset()
   {
     value = 0;
   }

2. You can only access them by invoking the methods of the Clock class.

3. In one of the programmers who designed and implemented the Java library.

4. Other programmers who work on the personal finance application.

5. Baron's Checking, withdraw(Baron's Checking, getBalance())

6. The withdraw method has return type void. It doesn't return a value. Use the getBalance method to obtain the balance after the withdrawal.

7. Add an accountNumber parameter to the constructors, and add a getAccountNumber method. There is no need for a setAccountNumber method—the account number never changes after construction.

9. /**
   * This class models a tally counter.
   */
   public class Counter
   {
     private int value;

     /**
      * Gets the current value of this counter.
      * @return the current value
      */
     public int getValue()
     {
       return value;
     }

     /**
      * Advances the value of this counter by 1.
      */
     public void count()
     {
       value = value + 1;
     }

   }

10. /**
    * Constructs a new bank account with a given initial balance.
    * @param accountNumber the account number for this account
    * @param initialBalance the initial balance for this account
    */

11. The first sentence of the method description should describe the method—it is displayed in isolation in the summary table.

12. An instance variable
    private int accountNumber;
    needs to be added to the class.

13. Because the balance instance variable is accessed from the main method of BankTester.
    The compiler will report an error because it is not a method of the BankAccount class.

14. public int getWidth()
    {
      return width;
    }

15. There is more than one correct answer. One possible implementation is as follows:
    public void translate(int dx, int dy)
    {
      int newX = x + dx;
      int newY = y + dy;
      
16. One BankAccount object, no BankAccountTester object. The purpose of the BankAccountTester class is merely to hold the main method.

17. In those environments, you can issue interactive commands to construct BankAccount objects, invoke methods, and display their return values.

18. Variables of both categories belong to methods—they come alive when the method is called, and they die when the method exits. They differ in their initialization. Parameter variables are initialized with the call values; local variables must be explicitly initialized.

19. After computing the change due, payment and purchase were set to zero. If the method returned payment + purchase, it would always return zero.

20. One implicit parameter, called this, of type BankAccount, and one explicit parameter, called amount, of type double.

21. It is not a legal expression. this is of type BankAccount and the BankAccount class has no instance variable named amount.

22. No implicit parameter—the main method is not invoked on any object—and one explicit parameter, called args.

23. CarComponent

24. In the draw method of the Car class, call g2.fill(new Color(0, 255, 0));

25. Double all measurements in the draw method of the Car class.