Chapter 3

Writing Simple Classes

Using BeanShell and BlueJ

Outline

| Writing simple Java classes using BlueJ |
| Experimenting with and testing classes using BlueJ |
| Using BeanShell with Objects and Methods |
| Understanding the structure of simple classes |
| Writing class documentation for Javadoc |
| Understanding common syntax and logical errors |
| Understand basic object-oriented terminology |
3.1 Introduction

So far we have been writing simple sequences of statements that declare variables and assign the results of arithmetic expressions to them. We have used BeanShell to execute such sequences of statements.

In Java a program (application) consists of one or more classes that are used to construct objects which can interact with each another at execution time. Each class defines the functionality or behavior of its objects by defining a number or methods (functions) that an object can execute.

These abstract concepts can be quite confusing for the beginner. Fortunately, there is an integrated development environment (IDE) called BlueJ that lets us write classes and interactively construct objects and invoke methods on them. It is a leaning tool for understanding the three fundamental object oriented concepts of class, object, and method. In this sense BlueJ is unlike other development environments. Also, with BlueJ it is very easy to test our classes and when we are finished we can package our application in a form that can be executed outside the BlueJ environment.

3.2 CircleCalculator class using BlueJ

In this section we begin our study of object-oriented programming using BlueJ to write a simple class called CircleCalculator, using the formulas in Example 2.12 to define how to calculate the area and circumference of a circle.

In BlueJ we have the concept of a project. It is a directory (folder) that contains all the class files (java source files, byte code files) and other files associated with your project. You can create a project and type in the classes yourself or you can use the BlueJ projects supplied with this book that already contain them.

For this chapter we assume that the CircleCalculator class is found in a BlueJ project called book-projects/chapter3. The other classes in this Chapter, TriangleCalculator and QuadraticRootFinder, are also in this project.

3.2.1 Experimenting with the class

Before analyzing the source code for this class we can experiment with the class by launching BlueJ and opening the chapter3 project to get the display shown in Figure 3.1(a). Now perform the following steps to test the class:

1. The CircleCalculator rectangle represents the class. Before using the class for the first time its source code file must be compiled into an object code (bytecode) file. You will know when compilation is necessary because the class rectangle will be diagonally shaded. If this is the case you can select the compile button to compile it, or right click on the class rectangle and select compile.

2. Now right click on the CircleCalculator rectangle to bring up its menu.

3. Select the new CircleCalculator menu choice shown in Figure 3.1(b).
3.2 CircleCalculator class using BlueJ

Figure 3.1: (a) shows the chapter3 project with CircleCalculator highlighted, (b) shows how to right-click to get the constructor menu.

4. In the “Create Object” dialog box shown in Figure 3.2(a) you can give a name to the object or accept the default name. We have used the name circle1. Also you must provide a value for the radius and we have chosen 2. When you select OK an object of the class is constructed. The new object will appear on the object workbench as a red rounded rectangle (see Figure 3.2(b)). The name of the object and the name of the class are shown in this object box. Each object created from the class is called an instance of the class.

5. Right click on this object and select a method from the menu shown in Figure 3.2(b). If you select double getArea() then the getArea method will be executed and the result for the area of the circle with radius 2 will be as shown in a message box (see Figure 3.3).

6. Now repeat the previous step and execute the getCircumference and getRadius methods.

7. Go back to step 1 and create two more objects called circle2 for a radius of 3, and circle3 for a radius of 4. You can create as many objects (instances) as you want, each with a different name and radius. Figure 3.4 shows three objects on the object workbench.

8. If you double click on the yellow CircleCalculator rectangle an editor window appears showing the Java source code for the class.

This example elegantly illustrates the three fundamental object-oriented programming (OOP) concepts of class, object, and method.

The class, represented by the yellow rectangle with the class name in it, acts like a blueprint for creating objects. When you double click on it you see the Java code for the class. When you right click on it you can create an object or instance of the class by supplying a name and any arguments.

An object is represented by a red rounded rectangle showing the name of the object and the class that created it. When you right click on an object you can select one of the methods to invoke...
Figure 3.2: (a) shows the dialog box for entering the constructor argument (radius of the circle), and (b) shows how to invoke the `getArea` method by right clicking on the resulting object.

Figure 3.3: After choosing the `getArea` method the result for the area is displayed.

Figure 3.4: Three `CircleCalculator` objects for radii 2, 3, and 4.
on it. This is sometimes called “sending a message to the object”. This causes the Java code in the class to execute and any results returned by the method are displayed.

We also see that several objects can be constructed from a class. Each has a unique name and its own variables that define the object, such as the radius, area, and circumference.

### 3.2.2 CircleCalculator source code

Below we show the source code for the CircleCalculator class. For now we have omitted comments so that we can emphasize the structure of the class. Comments are very important for documenting a class and we will show how to include them later in the Chapter. The source code resides in a file called CircleCalculator.java and can be viewed in BlueJ by double clicking on the class rectangle.

```java
public class CircleCalculator {
    private double radius;
    private double area;
    private double circumference;

    public CircleCalculator(double r) {
        radius = r;
        area = Math.PI * radius * radius;
        circumference = 2.0 * Math.PI * radius;
    }

    public double getRadius() {
        return radius;
    }

    public double getArea() {
        return area;
    }

    public double getCircumference() {
        return circumference;
    }
}
```

### 3.2.3 Explanation of the source code

The source code consists of a class declaration containing three parts: (1) instance data fields, (2) constructor declarations, and (3) method declarations.
Class declaration

The following lines are called the class declaration.

```java
public class CircleCalculator {
    ...
}
```

It gives a name to the class and the class definition (class body) is contained within the opening and closing braces. A more general template for a class declaration is given in Figure 3.27.

Instance data fields

To define the class we need to specify the variables that uniquely define the state of an object of the class. This is done with the declarations

```java
private double radius;
private double area;
private double circumference;
```

The keyword private indicates that these variables will not be directly accessible outside the class. Each CircleCalculator object will have its own copies of these variables so they are called instance data fields or instance variables.

To see this right click on a CircleCalculator object and choose inspect on the menu. For example, if we right click on the objects called circle1 and circle2 in Figure 3.4 and select inspect the results are shown in Figure 3.5. This shows that each object has its own set of instance data fields. Therefore, an object is often called an instance of the class.

Constructor declaration

Next comes the constructor declaration
3.2 CircleCalculator class using BlueJ

```java
public CircleCalculator(double r)
{
    radius = r;
    area = Math.PI * radius * radius;
    circumference = 2.0 * Math.PI * radius;
}
```

A more general template for a constructor declaration is given in Figure 3.28.

A **constructor** is needed in order to construct an object of the class, so its main purpose is to give values to the instance data fields.

The first line gives the **constructor prototype**. It specifies the name of the constructor, which must always be the same as the name of the class and it specifies what arguments, if any, are necessary to create an object. In our case we only need to specify the radius of the circle as a value of type `double`.

Finally, within the matching braces we place the statements that should be executed when the constructor is used to create an object. These statements are called the **constructor body**.

In Figure 3.2(a) the constructor prototype is displayed and input boxes are available for specifying the object name and the constructor argument `r`. When you fill in these values and click OK the three assignment statements in the constructor are executed to define the object.

It is important to note that the types of the three variables must not be declared in the constructor body since they have already been declared in the instance data field section of the class.

**Method declarations**

At this stage we have an object on the workbench that is just waiting for something to do. We can tell an object what to do by **invoking** a method on it. These methods are sometimes called **instance methods**. The CircleCalculator class contains three such methods, getRadius, getArea, and getCircumference. For example, the getArea method declaration is

```java
public double getArea()
{
    return area;
}
```

and similarly for the other two methods. A more general template for a method declaration is given in Figure 3.29.

The first line of a method is called the **method prototype**. Our method is public to indicate that it is available outside the class.

Then comes the type of value returned by the method. In our case we are returning the area which has type `double`.

Next comes the name of the method followed by a pair of parentheses which would normally contain any arguments the method requires. The syntax here is the same as for constructor arguments. In our case no arguments are needed but the parentheses are still required.

Finally, within the matching braces we place the statements that should be executed when the method is invoked (called). These statements are called the **method body**.
When we right click on an object we see a menu of these methods. If we select `getArea` then the statements in its method body are executed. In our case we only need to return the value of one of the instance data fields and that is what the `return` statement does.

A common convention for a method that simply returns the value of an instance data field is to prefix its name with `get`. Such methods are often called *get methods* or *enquiry methods*.

The `CircleCalculator` class is a simple one and you should now understand how the actions performed within BlueJ correspond to the instance data fields, and the execution of code in the bodies of the constructors and methods defined in the class.

### 3.3 TriangleCalculator class using BlueJ

For our second class we will use the formulas for the side lengths and angles of a triangle to solve the following problem:

"Given the length of two sides of a triangle and the contained angle in degrees, compute the third side length, the other two angles, and the area and perimeter of the triangle."

If we assume that $a$ and $b$ are the two side lengths and $\gamma$ is the contained angle then the formulas are given in Figure 3.6.

#### 3.3.1 Experimenting with the class

We can try out this class by right clicking on the `TriangleCalculator` rectangle to get the “Create Object” dialog box shown in Figure 3.7. We have given our object the name `triangle` and have used the constructor arguments 1, 1, and 90 degrees for the two side lengths and the contained angle.

Right click on the object to get the method menu shown in Figure 3.8. There are nine methods, three return the side lengths, three return the angles, one returns the perimeter, one returns the area, and one checks how close the sum of the angles is to 180 degrees.
3.3 TriangleCalculator class using BlueJ

Figure 3.7: Dialog box for entering the arguments to construct a TriangleCalculator object.

Figure 3.8: The method menu for a TriangleCalculator object.
Figure 3.9: In (a) the value of the third side is shown, and in (b) the sum of the three angles is shown.

Figure 3.10: The result of choosing inspect from the object menu for a TriangleCalculator object.

The results for choosing the get method and the checkAngleSum method are shown in Figure 3.9. If we right click on the object again and choose inspect we see the dialog box shown in Figure 3.10. This shows that a TriangleCalculator object is defined by eight instance data fields: three sides, three angles, the area, and perimeter.

### 3.3.2 TriangleCalculator source code

Below we show the source code for the TriangleCalculator class with the comments omitted for now. The source code resides in a file called TriangleCalculator.java and can be viewed in BlueJ by double clicking on the class rectangle.

```java
public class TriangleCalculator {
    private double a, b, c;
    private double alpha;
    private double beta;
    private double gamma;
    private double perimeter, area;

    public TriangleCalculator(double sideA, double sideB, double g)
```
```java
3.3 TriangleCalculator class using BlueJ

```
public double getArea()
{
    return area;
}

public double checkAngleSum()
{
    return alpha + beta + gamma;
}

3.3.3 Explanation of the source code

The structure of this class is similar to CircleCalculator.

Class declaration

The class declaration is

    public class TriangleCalculator
    {
        ...
    }

Instance data fields

Inside the class declaration are the declarations for the eight instance data fields that were shown in the “Inspector” window in Figure 3.10:

    private double a, b, c;
    private double alpha;
    private double beta;
    private double gamma;
    private double perimeter, area;

Constructor declaration

The constructor declaration has the form

    public TriangleCalculator(double sideA, double sideB, double g)
    {
        ...
    }

The first line is the constructor prototype, and it indicates that three double arguments are required to construct an object. This constructor corresponds to the “Create Object” dialog box in Figure 3.7.
The constructor body contains the statements that are needed to calculate the third side length, the remaining two angles, the perimeter, and the area using the formulas in Figure 3.6. Nine variables are used here. Eight of them are just the variables that have already been declared as instance data fields. These variables are available anywhere within the class (inside constructor or method bodies).

However, there is also a variable \( s \) that is just an intermediate variable used only inside the constructor body to simply the calculations so its type must be declared:

```java
double s;
```

This is an example of a **local variable**.

### Method declarations

The nine methods of this class are shown on the object menu in Figure 3.8. Eight of these methods are just “get methods”. Each returns the value of one of the instance data fields.

An additional **enquiry method** with the declaration

```java
public double checkAngleSum()
{
    return alpha + beta + gamma;
}
```

is also included in the class. This method is useful when testing the class.

### 3.3.4 Testing TriangleCalculator

This class is not as simple to test as CircleCalculator since it involves some complicated mathematical formulas. If would be easy to make a mistake in the translating of these formulas into Java statements. For example, you could forget that the trigonometric functions require angles in radians instead of degrees, or you could have a plus sign instead of a minus sign, or you could have interchanged \( a \) and \( b \) somewhere. Therefore we need to be able to check our results. Here are some ways to do this using BlueJ.

- Use the checkAngleSum method. Any significant deviation from 180 degrees means there is some error in the formulas, either the translated ones or the original mathematical ones.

- Develop some test cases for which you know the answer independently. For example, in Figure 3.7, we choose side lengths of 1 and a contained angle of 90 degrees because we know that the third side length is \( \sqrt{2} \approx 1.4142 \), and the other two angles are 45 degrees. Another simple case is the 30, 60, 90 triangle with sides 1, \( \sqrt{3} \), 2. Other cases could be checked using a calculator.
A quadratic equation has the form \( ax^2 + bx + c = 0 \) with \( a \neq 0 \). The values of \( x \) which satisfy the equation are called the roots of the equation. The roots \( r_1 \) and \( r_2 \) are given in terms of \( a, b, \) and \( c \) by the well-known formulas

\[
 r_1 = \frac{1}{2a} \left( -b - \sqrt{b^2 - 4ac} \right), \quad r_2 = \frac{1}{2a} \left( -b + \sqrt{b^2 - 4ac} \right)
\]

We will assume that the roots are real and we do not check for the square root of a negative number. This requires conditional statements which are introduced in a later Chapter. For testing purposes it is also useful to know that the sum and product of the roots satisfy \( r_1 + r_2 = -b/a \) and \( r_1 r_2 = c/a \).

3.4 QuadraticRootFinder class using BlueJ

We can try out this class by right clicking in the QuadraticRootFinder rectangle and selecting the constructor to get the “Create Object” dialog box shown in Figure 3.11. We have entered values for the quadratic equation \( x^2 - 2x + 3/4 = 0 \) whose roots are \( r_1 = 1/2 \) and \( r_2 = 3/2 \). Right clicking on the object gives the method menu shown in Figure 3.12. There are eight methods associated with a QuadraticRootFinder object, five are get methods for returning the coefficients and the roots and three are set methods for modifying the coefficients.

Invoking the getRoot1 and getRoot2 methods gives the result boxes shown in Figure 3.13. Normally, to calculate roots for another quadratic equation we would have to go back and create another object, but this class has “set methods” which can be used to change one or more of the coefficients \( a, b, \) or \( c \) so it is not necessary to create a new object. If we want to change \( c \) to 1 and recalculate the roots we just invoke the setC method. This gives the dialog box in Figure 3.14 that prompts for a new value of the argument. Now we can invoke the getRoot1 and getRoot2 methods to see the new roots in Figure 3.15. Finally, if we right click on the object and choose inspect we get the “Inspector” window shown in Figure 3.16.
3.4 QuadraticRootFinder class using BlueJ

Figure 3.12: The method menu for a QuadraticRootFinder object.

Figure 3.13: The roots of the quadratic equation for $a = 1$, $b = -2$ and $c = 3/4$.

Figure 3.14: The dialog box for the setC method.

Figure 3.15: The roots of the quadratic equation for $a = 1$, $b = -2$, $c = 1$. 
3.4.2 QuadraticRootFinder source code

Below we show the source code for the QuadraticRootFinder class with the comments omitted for now. The source code resides in a file called QuadraticRootFinder.java and can be viewed in BlueJ by double clicking the class rectangle.

```java
public class QuadraticRootFinder {
    private double a, b, c;
    private double root1, root2;

    public QuadraticRootFinder(double a, double b, double c) {
        this.a = a;
        this.b = b;
        this.c = c;
        doCalculations();
    }

    private void doCalculations() {
        double d = Math.sqrt(b*b - 4*a*c);
        root1 = (-b - d) / (2.0 * a);
        root2 = (-b + d) / (2.0 * a);
    }

    public double getRoot1() {
        return root1;
    }

    public double getRoot2() {
        return root2;
    }
}
```

Figure 3.16: The result of choosing inspect from the object menu for a QuadraticRootFinder object corresponding to $a = 1, b = -2$ and $c = 1$. 
3.4 QuadraticRootFinder class using BlueJ

```java
public double getA()
{
    return a;
}

public double getB()
{
    return b;
}

public double getC()
{
    return c;
}

public void setA(double value)
{
    a = value;
    doCalculations();
}

public void setB(double value)
{
    b = value;
    doCalculations();
}

public void setC(double value)
{
    c = value;
    doCalculations();
}
```

3.4.3 Explanation of the source code

There are some new concepts in this class.

**Class declaration**

The class declaration is

```java
public class QuadraticRootFinder
{
    ...
}
```

**Instance data fields**

The instance data fields are
private double a, b, c;
private double root1, root2;

and they correspond to the “Inspector” window in Figure 3.16.

**Constructor declaration**

The constructor declaration is given by

```java
public QuadraticRootFinder(double a, double b, double c)
{
    this.a = a;
    this.b = b;
    this.c = c;
    doCalculations();
}
```

In the *CircleCalculator* and *TriangleCalculator* constructors we gave the constructor arguments names that were different from the instance data field names.

Here we give them the same names as the corresponding instance data field names. How do we distinguish between the instance data field names and the argument names? The Java designers thought of this and the answer is to use a special keyword called this as a prefix to indicate an instance data field. In the constructor body this.a refers to the instance data field variable a and a refers to the constructor argument whose value is supplied when we create an object by executing the statements in the constructor body.

**Method declarations**

In the constructor body we find something new, namely the statement

```
doCalculations();
```

When this statement is executed (called) the statements in the doCalculations method given by

```java
private void doCalculations()
{
    double d = Math.sqrt(b*b - 4*a*c);
    root1 = (-b - d) / (2.0 * a);
    root2 = (-b + d) / (2.0 * a);
}
```

are executed. They calculate the values of the two roots.

The method is declared private since it is really just a helper method not needed outside the class. This also means it doesn’t appear on the object menu in Figure 3.12.

Our get methods return a value using the return statement but the doCalculations method doesn’t return any value. It just calculates values for the two instance data fields for the two roots. Methods that don’t return a value indicate this using the keyword void for the return type.

Why do we introduce this method? Why not just use the constructor declaration
3.4 QuadraticRootFinder class using BlueJ

```java
public QuadraticRootFinder(double a, double b, double c)
{
    this.a = a;
    this.b = b;
    this.c = c;
    double d = Math.sqrt(b*b - 4*a*c);
    root1 = (-b - d) / (2.0 * a);
    root2 = (-b + d) / (2.0 * a);
}
```

The reason can be seen if you notice that the doCalculations method is being used in four different places in the class, once in the constructor, and once in each of the three "set" methods. We could have duplicated the three lines of root calculating code in four places but this is not normally good programming practice. Instead we use a technique called factoring that replaces repeated blocks of code with a method and uses (calls) the method in several places.

The remainder of the class consists of the five get methods for returning the three coefficients and the two roots, and three set methods. Each set method changes one of the instance data fields so it is often called a mutator method. This is useful since it means we can solve a new quadratic equation without constructing another object. We simply call the appropriate set methods to change one or more of the coefficients. For example, to change the coefficient \( a \) we have the method

```java
public void setA(double value)
{
    a = value;
    doCalculations();
}
```

The return type is void to indicate that no value is being returned and there is an argument whose value is used to change the value of the instance data field \( a \). Since this will change the roots it is necessary to recalculate them by calling the doCalculations method. Similar methods are included to change the values of \( b \) and \( c \).

### 3.4.4 Testing QuadraticRootFinder

To test this class first try some values of \( a, b, c \) that give known solutions. For example we have tried the example \( x^2 - 2x + 3/4 = (x - 3/2)(x - 1/2) = 0 \) and obtained the correct roots \( r_1 = 1/2 \) and \( r_2 = 3/2 \). Similarly we tried \( x^2 - 2x + 1 = (x - 1)^2 = 0 \) and obtained the double root \( r_1 = r_2 = 1 \).

What happens if you try \( x^2 + x + 1 = 0 \). In this case there are no real roots since \( b^2 - 4ac = -3 \). The object inspector gives the results shown in Figure 3.17. This shows that we get the answers NaN for both roots. In Java this stands for “not a number” meaning that the result is not a valid double number in this case because \( \sqrt{-3} \) is not a real number. In fact, we know that the roots are complex numbers in this case.

Another special case to try is \( a = 0, b = 1, c = 1 \). In this case the equation is not even a quadratic equation. Nevertheless we get the answers -Infinity and NaN for the two roots. From the formulas for the roots we see that the first root would be \(-2/0\) which is giving -Infinity and the second root would be \(0/0\) which is giving NaN.
Figure 3.17: The result of choosing inspect from the object menu for a QuadraticRootFinder object corresponding to \(a = 1, b = 1\) and \(c = 1\).

Another possible test is to check results using the formulas \(r_1 + r_2 = -b/a\) and \(r_1 r_2 = c/a\) for the sum and product of the roots.

### 3.5 Using BeanShell with objects

Within the BlueJ environment the creation of objects and the invoking of methods on them is done interactively using the mouse and a dialog box to create the object (for example, Figure 3.1(b) and Figure 3.2(a)) and then selecting the method we want to invoke on the object from the object method menu (for example, Figure 3.2(b)).

Outside the BlueJ environment it is necessary to write Java statements to do this. Table 3.1 shows some examples of the correspondence between BlueJ mouse actions and menu choices and the Java statements we use outside BlueJ.

#### 3.5.1 Constructor call expressions

Table 5.1 shows that to construct a CircleCalculator object called circle1 for a circle of radius 2 we use the statement

\[
\text{CircleCalculator circle1 = new CircleCalculator(2.0);}
\]

The left side of the statement indicates that circle1 will be the name of an object from the CircleCalculator class and the right side uses the keyword new to indicate that the constructor should be called to create a new object. The right side of this statement is called a constructor call expression.

We can create TriangleCalculator and QuadraticRootFinder object in a similar way. For example (see Figure 3.7 and Figure 3.11)

\[
\text{TriangleCalculator triangle = new TriangleCalculator(1,1,90);
QuadraticRootFinder rootFinder = new QuadraticRootFinder(1,-2,0.75);
}
3.5 Using *BeanShell* with objects

<table>
<thead>
<tr>
<th>BlueJ actions</th>
<th>Java statement in <em>BeanShell</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>Create a CircleCalculator object called circle1 with radius 2 (Figure 3.1(b) and Figure 3.2(a))</td>
<td>CircleCalculator circle1 = new CircleCalculator(2.0);</td>
</tr>
<tr>
<td>Invoke the getArea method on this object (Figure 3.2(b))</td>
<td>double result = circle1.getArea();</td>
</tr>
<tr>
<td>Seeing the result is automatic (Figure 3.3)</td>
<td>print(result);</td>
</tr>
<tr>
<td>Construct two more CircleCalculator objects (Figure 3.5)</td>
<td>CircleCalculator circle2 = new CircleCalculator(3.0); CircleCalculator circle3 = new CircleCalculator(4.0);</td>
</tr>
<tr>
<td>Choose inspect from object menu to see instance data fields (see Figure 3.5(a))</td>
<td>print(circle1.getRadius()); print(circle1.getArea()); print(circle1.getCircumference());</td>
</tr>
</tbody>
</table>

Table 3.1: BlueJ actions and their corresponding Java statements.

In each case the constructor defined in the class declaration is called to create the new object and the left side of the statement gives it a name.

### 3.5.2 Method call expressions

Now that we have some objects we can invoke methods on them. To do this we need to specify the method name, any required arguments, and the name of the object. For example (see Figure 3.2(a) and Figure 3.3) the statement

\[
\text{double result} = \text{circle1.getArea();}
\]

gets the area of the circle1 object. The right side of this statement is called a **method call expression** and is formed from the object name followed by a dot followed by the method name. We need an empty pair of parentheses here because this method has no arguments. Since we are using a get method, a value is returned and we can assign it to result, a double variable.

Similarly, we can invoke methods on TriangleCalculator and QuadraticRootFinder objects. For example (see Figure 3.8 and Figure 3.9)

\[
\text{double c} = \text{triangle.getC();}
\]
\[
\text{double sum} = \text{triangle.checkAngleSum();}
\]

return the third side and angle sum and assign them to variables, and (see Figure 3.13 to Figure 3.15)

\[
\text{double r1} = \text{rootFinder.getRoot1();}
\]
\[
\text{double r2} = \text{rootFinder.getRoot2();}
\]

return the two roots of a quadratic equation and assign them to variables.

The statement
invokes the setC method on the QuadraticRootFinder object named rootFinder. It is not an assignment statement since setC does not return a value (its return type is void). It is called an expression statement. We know that this method has one argument for the new value of the coefficient c in the quadratic equation so the effect of the method call expression is to changes the value of this coefficient to 1.0.

3.5.3 BeanShell examples

To use BeanShell to experiment with the objects of our three classes we need to tell it where to find the bytecode files that the BlueJ compiler has produced, for example CircleCalculator.class. These are located in the project directory. For example, assuming your project directory for this chapter is

c:/book-projects/chapter3

then you can type the following command into BeanShell

addClassPath("c:/book-projects/chapter3");

If you are running BeanShell using Windows then it is important to use forward slashes here instead of backslashes. This change to the classpath remains in effect until you exit BeanShell. The following examples show how BeanShell can be used to construct objects and invoke methods on them.

**Example 3.1** (CircleCalculator objects) The following statements use BeanShell to calculate the area and circumference of a circle using a CircleCalculator object.

```sh
bsh % addClassPath("c:/book-projects/chapter3");
bsh % CircleCalculator circle1 = new CircleCalculator(2.0);
bsh % double area1 = circle1.getArea();
bsh % print(area1);
12.566370614359172
bsh % double circum1 = circle1.getCircumference();
bsh % print(circum1);
12.566370614359172
```

This example is shown in Figure 3.18

**Example 3.2** (Three CircleCalculator objects) Continuing the previous example, the statements

```sh
bsh % CircleCalculator circle2 = new CircleCalculator(3.0);
bsh % CircleCalculator circle3 = new CircleCalculator(4.0);
bsh % double area2 = circle2.getArea();
bsh % double area3 = circle3.getArea();
bsh % double averageArea = (area1 + area2 + area3) / 3;
bsh % print(averageArea);
30.368728984701335
```
3.5 Using BeanShell with objects

Figure 3.18: Using BeanShell to construct a CircleCalculator object and invoke its methods.

construct two more objects, circle2 and circle3, and compute the average area of the three circles using the getArea method.

**Example 3.3** (TriangleCalculator objects) The statements

```bash
bsh % TriangleCalculator triangle = new TriangleCalculator(1,1,90);
bsh % double c = triangle.getC();
bsh % print(c);
1.414213562373095
bsh % double angleSum = triangle.checkAngleSum();
bsh % print(angleSum);
180.0
```

compute the length of the third side of the right-angled triangle and check the sum of the angles.

**Example 3.4** (QuadraticRootFinder objects) The statements

```bash
bsh % QuadraticRootFinder rootFinder = new QuadraticRootFinder(1,-2,0.75);
bsh % double r1 = rootFinder.getRoot1();
bsh % double r2 = rootFinder.getRoot2();
bsh % print(r1);
0.5
bsh % print(r2);
1.5
bsh % rootFinder.setC(1);
bsh % r1 = rootFinder.getRoot1();
bsh % r2 = rootFinder.getRoot2();
bsh % print(r1);
1.0
bsh % print(r2);
1.0
```
compute the roots \( r_1 = 1/2 \) and \( r_2 = 3/2 \) of the equation \( x^2 - 2x + 3/4 = 0 \). Then the coefficient \( c \) is changed to 1 and the new roots \( r_1 = r_2 = 1 \) are computed.

## 3.6 Writing and viewing Javadoc class documentation

The three classes in this chapter have been presented without any comments. It is essential that every class you write include comments to document the purpose of the class, its constructors and methods. There are three kinds of comments in Java:

### Single line comments

If you use `//` then these characters and all others following them on the same line are ignored by the Java compiler. For example in the `TriangleCalculator` class we can use

```java
private double gamma; // angle opposite side c
```

to indicate the purpose of the variable `gamma`.

### Multi-line comments

They begin with `/*` on one line and end on the same line or a following one with the characters `*/`. For example, the private method in `QuadraticRootFinder` can be documented as follows:

```java
/* This private method is used in the constructor and the
 * three set methods in order to update the roots in case
 * a coefficient is changed.
 */
private void doCalculations()
{
    double d = Math.sqrt(b*b - 4*a*c);
    root1 = (-b - d) / (2.0 * a);
    root2 = (-b + d) / (2.0 * a);
}
```

The extra asterisks on the two intermediate lines are not necessary but they are supplied by the BlueJ editor automatically so we will use them.

### Javadoc comments

They look like regular multi-line comments but they begin with `/**` so that the javadoc processor can identify them. Of course the Java compiler sees them as ordinary multi-line comments and ignores them. For example, here is a javadoc version for the `CircleCalculator` constructor.

```java
/** Constructor for an object with specified radius.
 * @param r the radius of the circle
 */
public CircleCalculator(double r)
{
```
3.6 Writing and viewing Javadoc class documentation

```java
radius = r;
area = Math.PI * radius * radius;
circumference = 2.0 * Math.PI * radius;
}
```

Within a javadoc comment HTML tags can be used and there are special tags beginning with the @ character. In this example we have used the @param tag to describe the constructor argument. The HTML and special tags are used to apply special formatting to the class documentation (interface). The resulting HTML document can be viewed by a browser.

The javadoc comments are also shown by BlueJ in the “Create Object” dialog boxes.

### 3.6.1 Javadoc rules

First we need some of the javadoc rules:

- Use a javadoc block comment immediately before the class declaration to give a description of the class. This comment can contain the special @author and @version tags.

- Use a javadoc comment immediately before each public constructor and method declaration. The first line (ended by first period) is special and is used in the summary part of the documentation. Any remaining lines give further information that is shown in the detail part of the documentation.

- Use a parameter line for each argument. It has the format

  ```
  @param name text.
  ```

  where name is the name of the argument and text describes the argument.

- Use a return line for each method that returns a value. It has the format

  ```
  @return text.
  ```

There are other javadoc tags that we won’t need yet. We can now give the complete javadoc versions of our three classes.

### 3.6.2 Javadoc version of CircleCalculator

```
package chapter3; // remove this line if you are not using packages
/**
 * The objects of this class know how to compute the area and
 * circumference of a circle, given its radius as a constructor
 * argument (parameter).
 */
public class CircleCalculator
```
/** Constructor for an object with specified radius.
 * @param r the radius of the circle
 */
public CircleCalculator(double r) {
    radius = r;
    area = Math.PI * radius * radius;
    circumference = 2.0 * Math.PI * radius;
}

/**
 * Return the radius of the circle.
 * @return circle radius
 */
public double getRadius() {
    return radius;
}

/**
 * Return the area of the circle.
 * @return circle area
 */
public double getArea() {
    return area;
}

/**
 * Return the circumference of the circle.
 * @return circle circumference
 */
public double getCircumference() {
    return circumference;
}

3.6.3 Javadoc version of TriangleCalculator

package chapter3; // remove this line if you are not using packages
/**
 * A TriangleCalculator represents a triangle by two side lengths and
 * the contained angle in degrees. From this information the third
 * side length and remaining two angles can be calculated. Then the
 * area and perimeter can be calculated. All values can be returned
 * using get methods.
 */

public class TriangleCalculator
{
    private double a, b, c; // triangle side lengths
    private double alpha; // angle opposite side a
    private double beta; // angle opposite side b
    private double gamma; // angle opposite side c

    private double perimeter, area;

    /**
     * Construct a triangle given two sides and contained angle.
     * @param sideA the first side length
     * @param sideB the second side length
     * @param g the contained angle in degrees
     */
    public TriangleCalculator(double sideA, double sideB, double g)
    {
        double s;
        a = sideA;
        b = sideB;
        c = Math.sqrt(a*a + b*b -2*a*b*Math.cos(Math.toRadians(g)));

        // Angle opposite side a, contained by sides b and c
        alpha = Math.acos((b*b + c*c - a*a) / (2*b*c));
        alpha = Math.toDegrees(alpha);

        // Angle opposite side b, contained by sides c and a
        beta = Math.acos((c*c + a*a - b*b) / (2*c*a));
        beta = Math.toDegrees(beta);

        gamma = g;

        // Calculate perimeter and use Heron’s formula for
        // the area in terms of the side lengths
        perimeter = a + b + c;
        s = perimeter / 2;
        area = Math.sqrt(s*(s-a)*(s-b)*(s-c));
    }

    /**
     * Return the length of side a.
     */
* @return the length of side a
*/
public double getA()
{
    return a;
}

/**
* Return the length of side b.
* @return the length of side b
*/
public double getB()
{
    return b;
}

/**
* Return the length of side c.
* @return the length of side c
*/
public double getC()
{
    return c;
}

/**
* Return the angle opposite side a.
* @return the angle opposite side a in degrees
*/
public double getAlpha()
{
    return alpha;
}

/**
* Return the angle opposite side b.
* @return the angle opposite side b in degrees
*/
public double getBeta()
{
    return beta;
}

/**
* Return the angle opposite side c.
* @return the angle opposite side c in degrees
*/
public double getGamma()
{
    return gamma;
}
3.6 Writing and viewing Javadoc class documentation

```java
/**
 * Return the perimeter of the triangle.
 * @return the perimeter of the triangle
 */
public double getPerimeter()
{
    return perimeter;
}

/**
 * Return the area of the triangle
 * @return the area of the triangle
 */
public double getArea()
{
    return area;
}

/**
 * Return the sum of the angles as a check that it is close
to 180 degrees.
 * @return the sum of the angles
 */
public double checkAngleSum()
{
    return alpha + beta + gamma;
}

3.6.4 Javadoc version of QuadraticRootFinder

package chapter3; // remove this line if you are not using packages

/**
 * An object of this class can calculate the real roots of the
 * quadratic equation ax^2 + bx + c = 0 given the coefficients a, b, and c.
 * The program does not check if there are real roots. Later when
 * we know how to make decisions (if statements) we can make a better
 * version of this class.
 */
public class QuadraticRootFinder
{
    // Instance data fields for coefficients and roots
    private double a, b, c;
    private double root1, root2;

    /**
     * Construct a quadratic equation root finder given the coefficients
     */
```
public QuadraticRootFinder(double a, double b, double c) {
    this.a = a;
    this.b = b;
    this.c = c;
    doCalculations();
}

/* This private method is used in the constructor and the
   three set methods in order to update the roots in case
   a coefficient is changed. */
private void doCalculations() {
    double d = Math.sqrt(b*b - 4*a*c);
    root1 = (-b - d) / (2.0 * a);
    root2 = (-b + d) / (2.0 * a);
}

/**
 * Return the first root.
 * @return the first real root or NaN if there are none
 */
public double getRoot1() {
    return root1;
}

/**
 * Return the second real root.
 * @return the second real root or NaN if there are none
 */
public double getRoot2() {
    return root2;
}

/**
 * Return the coefficient of x^2.
 * @return the coefficient of x^2
 */
public double getA() {
    return a;
}

/**
 * Return the coefficient of x.
 */
3.6 Writing and viewing Javadoc class documentation

```java
/**
 * @return the coefficient of x
 */
public double getB()
{
    return b;
}

/**
 * Return the constant coefficient.
 * @return the constant coefficient
 */
public double getC()
{
    return c;
}

/**
 * Change the value of the coefficient of x^2.
 * @param value the new value for the coefficient of x^2
 */
public void setA(double value)
{
    a = value;
    doCalculations();
}

/**
 * Change the value of the coefficient of x.
 * @param value the new value for the coefficient of x
 */
public void setB(double value)
{
    b = value;
    doCalculations();
}

/**
 * Change the value of the constant coefficient.
 * @param value the new value for the constant coefficient.
 */
public void setC(double value)
{
    c = value;
    doCalculations();
}
```

3.6.5 Viewing the documentation

With BlueJ it is very easy to generate and display the Java documentation. For example, to generate documentation for the CircleCalculator class double click on its rectangle to bring up the
Figure 3.19: An editor window.

editor window as shown in Figure 3.19. Now select the implementation button menu in the top right corner of the toolbar and select interface. The documentation will be generated in the editor window. Part of it is shown in Figure 3.20. You can use this button to toggle between the source code (implementation) and the interface (documentation).

3.6.6 Implementation and documentation views

The implementation and documentation give two different views of a class. The implementation gives the complete view of the source code including all comments, all constructor and method bodies. The documentation produced by javadoc is often called the public interface or specification of the class. It includes only the javadoc comments, the public class, constructor, and method prototypes (first lines) but not the method bodies, all nicely formatted as an HTML document. Private data fields and methods are not shown in the documentation.

These two views relate to how we use the class. As a programmer writing Java classes we are writing the complete source code (the implementation) but as someone that is simply using the class, as we did in our BlueJ experiments, it is only necessary to view the public interface. It gives all the information needed to use the class.

3.6.7 Project documentation

Normally a project contains more than one class. The project for this chapter contains three classes. It is possible to generate the Java documentation for all classes in a project simultaneously by
3.7 Syntax and logical errors

When writing and testing a Java class it is rare that your first attempt is without error so it is important to recognize errors and be able to fix them.

There are two kinds of errors that can occur: syntax errors and logical errors. The Java language is defined by a number of syntax or grammar rules that are used by the compiler to determine whether a Java statement is legal or not. If a statement is illegal we say that it contains one or more syntax errors. These errors are often called compile-time errors since they are found by the compiler when it attempts to compile your class to obtain the bytecode file. Forgetting the semi-colon at the end of a statement is a common example of a syntax error.

Logical errors are often called run-time errors since they occur when the Java interpreter is executing your class. A logical error may result in an abnormal termination of execution of a constructor or method in your class or it may simply produce erroneous results because you used a minus sign instead of a plus sign in some formula.

Finding logical errors can be difficult and can be accomplished only by thoroughly testing your classes. Therefore we will place a lot of emphasis on techniques for testing classes. BlueJ is an excellent environment for testing.
Class CircleCalculator

java.lang.Object
    |
    +--CircleCalculator

public class CircleCalculator
extends Object

The objects of this class know how to compute the area and circumference of a circle, given its radius as a constructor argument (parameter).

Constructor Summary

CircleCalculator(double r)
    Constructor for an object with specified radius.

Method Summary

double getArea()
    Return the area of the circle.

double getCircumference()
    Return the circumference of the circle.

double getRadius()
    Return the radius of the circle.

Methods inherited from class java.lang.Object
    clone, equals, finalize, getClass, hashCode, notify,
    notifyAll, toString, wait, wait, wait

Figure 3.21: A web page for the chapter3 project documentation
3.7 Syntax and logical errors

3.7.1 Some common syntax errors

Forgetting a semi-colon

For example, open the editor for the CircleCalculator class and remove the semi-colon from the end of the line

```
public double radius;
```

Now compile the class using the Compile button on the editor toolbar and you will get an error message at the bottom of the editor window as shown in Figure 3.22. Click on the question mark button to see the more detailed message shown in Figure 3.23.

Undeclared variables

In the TriangleCalculator class it would be easy to omit the line

```
double s;
```

in the constructor body. Then you would get the error message shown in Figure 3.24.
Declaring a variable more than once

In the TriangleCalculator constructor declaration the local variable s is declared for the first time using

```
double s;
```

Later it is used in the line

```
s = perimeter / 2;
```

Replace this line by

```
double s = perimeter / 2;
```

Compile the class and you will get the error message shown in Figure 3.25.
3.7 Syntax and logical errors

Misspelling the constructor name

For example, in the CircleCalculator class (page 63) suppose the constructor declaration is written as:

```java
public circleCalculator(double r)
{
    ...
}
```

We have used a lowercase `c` instead of an uppercase `C`. If we compile the class we get the syntax error shown in Figure 3.26. The error message indicates that the compiler is trying to consider `circleCalculator` as a method but it doesn’t have a return type. The compiler does not consider this as a constructor declaration because constructors must have the same name as the class.

Forgetting `new` in constructor call expressions

We can’t illustrate this error in BlueJ yet, but it is easy to make in some of our BeanShell examples. For example, try the following statements.

```shell
bsh % addClassPath("c:/book-projects/chapter3");
bsh % CircleCalculator circle = CircleCalculator(2.0);
// Error: Typed variable declaration : Command not found: CircleCalculator:
<at unknown location>
```

Here we forgot to use `new` on the right hand side of the statement. The error message is not so helpful here. Without `new` the Java interpreter is assuming that the right hand side is a method call (like `getArea()`) but there is no method with this name.

3.7.2 Some common logical errors

There are several very common logical errors that programmers make in Java.
Using an incorrect formula

As an example consider the TriangleCalculator formulas in Figure 3.6 that are implemented in the constructor (page 64). Any number of logical errors could be made here such as using a + sign instead of a – sign. Errors like this can be discovered by testing.

Redeclaring an instance variable

This is difficult error for beginners to detect. For example, in the CircleCalculator class (page 63) if we had replaced the assignment statements

```java
radius = r;
area = Math.PI * radius * radius;
circumference = 2.0 * Math.PI * radius;
```

in the constructor body by the declarations

```java
double radius = r;
double area = Math.PI * radius * radius;
double circumference = 2.0 * Math.PI * radius;
```

then the class would compile just fine; there are no syntax errors. However, when you construct an object and ask it for the radius, area, and circumference, you will get 0.0 as an answer for any radius. By declaring the variables in the constructor we have introduced three local variables that have nothing to do with the instance data fields of the same name so the assignment of values to them does not change the instance data field values.

For numeric instance data fields the compiler will automatically initialize their values to zero and they will remain zero since we didn’t initialize them in the constructor. The local variables disappear when the constructor body finishes execution. Therefore, when you invoke the getRadius, getArea, and getCircumference methods on an object, 0.0 is returned in each case. This declaration of a variable in a constructor or method having the same name as an instance variable is called shadowing and should be avoided.

Using a return type on a constructor

This is also a common error that is easy to make and is difficult for a beginner to find and understand.

Suppose the CircleCalculator constructor (page 63) is replaced by

```java
public void CircleCalculator(double r)
{
    ...
}
```

We have erroneously used void in the constructor prototype. This is not a syntax error. The compiler simply assumes that CircleCalculator is the name of a method, not the name of a constructor.
3.7 Syntax and logical errors

If you try this in BlueJ by right clicking on the class you will see `new CircleCalculator()` and you will not be asked to enter a radius to construct an object. When you construct an object and right click to get its menu of methods you will see `void CircleCalculator(r)` there, indicating that this is a method.

What has happened is that by using a return type we have in effect not used any constructor in our class. When the compiler notices this it automatically provides a so-called default do-nothing constructor having the form

```java
public CircleCalculator()
{
}
```

with no arguments and an empty body. So this is what you are using to create an object and again when you invoke the `getRadius`, `getArea`, and `getCircumference` methods on an object, 0.0 is returned in each case.

3.7.3 Invoking a method on a non-existent object

In our BeanShell examples we could have constructed our objects in two steps as we did sometimes for `int` and `double` variables: first declare them and later assign values to them. For example we could use the statements

```bash
bsh % addClassPath("c:/book-projects/chapter3");
bsh % CircleCalculator circle;
bsh % circle = new CircleCalculator(2.0);
bsh % double area = circle.getArea();
bsh % print(area);
12.566370614359172
bsh % circle = new CircleCalculator(3.0);
bsh % area = circle.getArea();
bsh % print(area);
28.27433882308138
bsh %
```

The first statement declares `circle` to be a variable of type `CircleCalculator` and the second statement constructs an object to assign to it.

However suppose the second statement was accidentally omitted. Then the third statement is trying to invoke the `getArea` method on a non-existent object. In BeanShell we would get

```bash
bsh % CircleCalculator circle;
bsh % double area = circle.getArea();
// Error: // Uncaught Exception: Typed variable declaration :
// Null Pointer in Method Invocation: <at unknown location>
Target exception: java.lang.NullPointerException
```

This cryptic error message tells us that we have a variable `circle` of type `CircleCalculator` but we have not constructed an object to assign to it. Later you will better understand this error message.
3.8 Summary of terminology

In this section we give simple definitions of the important terms introduced in this Chapter. Many terms are used to discuss and explain a language such as Java. It is important that you understand them and can give examples of each term.

For example, there are general language-independent terms such as variable that would be used in any computer language, and terms such as class, object, and method that would be used in any object-oriented language.

Other terms and definitions would be specific to Java. For example, the Java language is defined by a set of rules called the grammar or syntax of the language. Although these rules can be formally defined we will introduce them in simple informal way.

**Simple identifier**

A sequence of one or more letters, digits and underscores such that the first character is not a digit. Identifiers are used to give names to variables, classes, constructors, objects, and methods.

An almost universal convention is to begin the name of a class with an upper case letter. All other identifiers begin with a lower case letter. In either case capitalize the beginning letter of each interior word.

Identifiers are **case sensitive**.

*Example:* radius, numberOfStudents are variable names
*Example:* CircleCalculator is a class name
*Example:* doCalculations, checkAngleSum are method names
*Example:* circle1, triangle are object names.

**Type**

A specific kind of data such as the set of all integers or the set of all real numbers, or the set of all CircleCalculator objects.

*Example:* int, float, double are primitive types.
*Example:* CircleCalculator is an object type.

**Class (definition 1)**

A definition of a set of objects of a specific type and their behavior.

*Example:* CircleCalculator, TriangleCalculator

**Class (definition 2)**

A home for some functions not associated with any objects.

*Example:* Math is our only example so far.

**Object**

An entity that has identity (a name), state, and behavior.

*Example:* A CircleCalculator object.
### Instance

An object constructed from a class.

**Example:** A `CircleCalculator` object that has the name `circle1` is an instance of the `CircleCalculator` class.

### Method

A function or operation defined in a class that can be invoked on an object of the class. Such methods are often called **instance methods**. Later we will learn that there are also **static methods**. The instance methods of a class define the behavior of objects.

**Example:** `getArea`, `setA`, `checkAngleSum`, `doCalculations`

### Constructor

A special kind of method defined in a class that is used to create an object (instance of the class) having specified properties. An object must be created before any of its methods can be invoked (executed). A constructor must have the same name as its class.

**Example:** `QuadraticRootFinder`

### Variable Declaration

A statement having one of the forms

\[
\text{accessModifier} \ \text{typeName} \ \text{identifier}; \\
\text{accessModifier} \ \text{typeName} \ \text{identifier} = \ \text{expression};
\]

where, for now, `accessModifier` is either absent or `private`, `identifier` is the name of the variable, `typeName` is the variable type and `expression` is an expression that evaluates to a value that can be assigned to the variable.

**Example:** `double radius;`
**Example:** `double radius = 2.0;`
**Example:** `double area = Math.PI * radius * radius;`
**Example:** `CircleCalculator circle1 = new CircleCalculator(2.0);`
**Example:** `CircleCalculator circle1;`
**Example:** `int n=123, remainder, hundreds, tens, units;`
**Example:** `double area, circumference;`
**Example:** `double radius = 3.0, area;`

The final three examples show that multiple variables of the same type can be declared and optionally initialized in a single declaration.

### Constant Declaration

A constant has the form

\[
\text{typeName} \ \text{identifier};
\]

where `typeName` includes `static` along with `private`.
accessModifier static final typeName identifier = expression; The accessModifier can be public or private. The strange keyword static indicates that constants are associated with the class, not the objects of the class. The equally strange keyword final distinguishes a constant declaration from an initialized variable declaration.

It is conventional to name constants using upper case letters and the underscore to simulate a space.

**Example:** static final double CM_PER_INCH = 2.54;

### arithmetic expression

An expression involving variables and operators that evaluates to a numeric value.

**Example:** radius
**Example:** 2.0 * Math.PI * radius
**Example:** remainder % 10
**Example:** circle1.getArea() + circle2.getArea()

### assignment statement

A statement of the form

identifier = expression;

where identifier is the name of a variable that has already been declared and expression is an expression whose value is assigned to the variable.

**Example:** radius = 2.0;
**Example:** circle1 = new CircleCalculator(2.0);
**Example:** area = Math.PI * radius * radius;

### class declaration

A template for a simple class declaration is shown in Figure 3.27. The first box is replaced by the name of the class. The other boxes show that a class declaration has three parts, not all of which are required in every class. These parts are defined below.

**Example:**

```java
public class CircleCalculator
{
    ...
}
```

The part indicated by { ... } is called the **class body**.
3.8 Summary of terminology

public class ClassName {
    Data field declarations
    Constructor declarations
    Method declarations
}

Figure 3.27: A template for a simple Java class declaration.

modifiers ClassName (formalArgumentList) {
    local declarations and other statements
}

Figure 3.28: A template for a simple Java constructor declaration.

**instance data field**

An instance data field is a special variable declaration in the body of a class but outside any method or constructor. These variables are available anywhere in the class and are the only variable declarations that have a modifier such as `private`. Each object of the class has its own copies of the instance data fields.

**Example:** private double radius;
**Example:** private double root1, root2;

The second example shows that more than one variable can be declared in one declaration.

**constructor declaration**

A template for a constructor declaration is shown in Figure 3.28. The `modifiers` box can be replaced by `public` which is the only modifier we have discussed so far. Then we have the name of the class followed by a formal argument list in parentheses, if any. The `formalArgumentList` is a list of type-variable pairs separated by commas.

**Example:**

```java
public TriangleCalculator(double sideA, double sideB, double g) {
```
Figure 3.29: A template for a simple Java method declaration.

```java
double s;
a = sideA;
...
area = Math.sqrt(s*(s-a)*(s-b)*(s-c));
}
```

The part indicated by {...} is called the **constructor body**.

**constructor prototype**

The first line of a constructor declaration is the constructor prototype.

**Example:** public CircleCalculator(double r)

**Example:** public TriangleCalculator(double sideA, double sideB, double g)

For a constructor, only the prototype is part of the public interface (see Javadoc output).

**constructor call expression**

An expression of the form

```
new ClassName(actualArguments)
```

that constructs an object from the class whose name is *ClassName* and whose *actualArguments*, if any, is a list of expressions separated by commas that evaluate to a value of the type indicated in the *formalArgument* list of the constructor prototype.

**Example:** new CircleCalculator(3.0)

**Example:** new QuadraticRootFinder(1.0,1.0,90.0)

**method declaration**

A template for a method declaration is shown in Figure 3.29. The *modifiers* box can be replaced by public or private which are the only modifiers we have discussed so far. Then we have *returnType*, the name of the return type, followed by the name of the method,
followed by a formal argument list in parentheses, if any. The \texttt{formalArgumentList} is a list of type-variable pairs separated by commas. The difference between a constructor and a method is that a method always has a return type and a name that begins with a lowercase letter and a constructor has no return type and a name that begins with an uppercase letter.

\textbf{Example:}

\begin{verbatim}
public double getArea()
{
    return area;
}
\end{verbatim}

\textbf{Example:}

\begin{verbatim}
public void setC(double value)
{
    c = value;
    doCalculations();
}
\end{verbatim}

\textbf{Example:}

\begin{verbatim}
private void doCalculations()
{
    double d = Math.sqrt(b*b - 4*a*c);
    root1 = (-b - d) / (2.0 * a);
    root2 = (-b + d) / (2.0 * a);
}
\end{verbatim}

The part of a method declaration indicated by \{\ldots\} is called the \textbf{method body}.

\textbf{method prototype}

The first line of a method declaration is the method prototype.

\textbf{Example:} public double getArea()
\textbf{Example:} public void setC(double value)
\textbf{Example:} public void doCalculations()

For a method, only the prototype is part of the public interface (see Javadoc output).
method call expression

An expression of one of the forms

`ObjectName.methodName(actualArguments)`

The first form invokes a method called `methodName` on an object called `ObjectName`. The `actualArguments`, if any, is a list of expressions separated by commas that evaluate to a value of the type indicated in the formal argument list of the method prototype. The second form, without an object name, is used to invoke a method in the same class in which the method is defined.

Example: `circle1.getArea()`
Example: `rootFinder.setC(1.0)`
Example: `doCalculations()`

enquiry method

A method that returns information specific to an object of a class, without changing the object.

Example: In the `TriangleCalculator` class the `checkAngleSum` method returns the sum of the three angles.

mutator method

A method that changes the state of an object of a class, usually by modifying one or more instance data fields.

Example: `doCalculations` in the `QuadraticRootFinder` class calculates new values for the instance variables `root1` and `root2`.

get method

A special kind of enquiry method whose name is `getName` where `name` is the name of one of the instance data fields. Its purpose is to return the value of this field.

Example: `getArea` returns the value of instance variable `area` in `CircleCalculator`.

set method

A special kind of mutator method whose name is `setName` where `name` is the name of one of the instance data fields. Its purpose is to change the value of this field.

Example: In the `QuadraticRootFinder` class `setC` modifies the value of the instance variable `c`. 
3.8 Summary of terminology

**local variable**

A local variable is quite different from an instance variable. An instance variable is declared in a class outside any constructor or method and it can be used inside any constructor or method of the class.

A local variable is declared in the body of a constructor or method and cannot be used outside the constructor or method. It comes into existence each time the body is executed and disappears when the constructor or method finishes execution.

**Example:** double s; declares a local variable in the TriangleCalculator constructor.

**formal argument**

A formal argument in a method or constructor is a special local variable whose value is supplied when a method or constructor call expression is executed (called).

**Example:** In the setC method of the QuadraticCalculator class, with prototype void setC(double value), value is a local variable of type double.

**actual argument**

An actual argument is a variable or expression whose value is used as the value of the corresponding formal argument when a method or constructor call expression is executed (called).

**Example:** For the setC method of the QuadraticRootFinder class, with prototype void setC(double value), the expression rootFinder.setC(1.0) causes the value 1.0 to be assigned as the value of the formal argument value.

**return statement**

A return statement has two forms:

```
return expression;
return;
```

The first form is used to indicate that the value of expression is to be returned by the method. The second form is used in a method that has void return type. We haven't seen an example of this form yet. When a return statement is executed the method finishes execution immediately.

**Example:** return alpha + beta + gamma;

**single line comment**

A comment beginning with //. These characters and all following characters on the same line are part of the comment.

**multi-line comment**

A comment beginning with /* and ending with */ on the same or another line.
3.9 Review exercises

- **Review Exercise 3.1** Define the following terms and give examples of each.

```java
simple identifier                type                class
object                          instance            method
constructor                     variable declaration constant declaration
class declaration              instance data field constructor declaration
constructor prototype          constructor call expression method declaration
method prototype                method call expression enquiry method
“get” method                    mutator method      “set” method
local variable                  formal argument     actual argument
return statement                single line comment multi-line comment
javadoc comment                 @param              @return
addClassPath                   public interface    class specification
class implementation           syntax error        logical error
run-time error                  undeclared variable error duplicate definition error
redeclaration error
```

- **Review Exercise 3.2** Make a list of all the prototypes for the constructors used in this chapter.

- **Review Exercise 3.3** Make a list of all the constructor call expressions used in this chapter.

- **Review Exercise 3.4** Make a list of all the prototypes for the methods used in this chapter.

- **Review Exercise 3.5** Make a list of all the method call expressions used in this chapter.

3.10 Programming exercises

In each programming exercise you should include javadoc comments and indicate what data you have used to test your class.

- **Exercise 3.1 (Converting inches to feet and inches)**
  Write a class called InchesToFeetConverter whose constructor argument is an integer representing the height of a person in inches. This number is to be converted to feet and inches. For example for a height of 67 inches is 5 feet and 7 inches. Use the following class outline and fill in the details indicated by `{..}`.

```java
public class InchesToFeetConverter
{
    private int heightInInches;
```
3.10 Programming exercises

private int feetPart;
private int inchesPart;

public InchesToFeetConverter(int height) { ... }
public int getHeightInInches() { ... }
public int getFeetPart() { ... }
public int getInchesPart() { ... }

▶ Exercise 3.2 (Height in metric units)
Write a class called HeightConverter whose constructor has two integer arguments for the feet part and the inches part of a height. This height is to be converted into centimeters. Use constants for the conversion factors from centimeters to inches (2.54) and from feet to inches (12.0). For example, for a height of 5 feet 10 inches the height in centimeters is 177.8. Use the following class outline and fill in the details indicated by { .. }.

public class HeightConverter {
   // put your constants here
   private double heightCM;
   public HeightConverter(int feet, int inches) { ... }
   public double getHeightCM() { ... }
}

▶ Exercise 3.3 (Fahrenheit to Celsius temperature conversion)
Write a class called FToCConverter that can be used to convert a Fahrenheit temperature to a Celsius temperature. The constructor needs one argument for the given temperature in Fahrenheit. Do not include any “set methods” in your class.

▶ Exercise 3.4 (Celsius to Fahrenheit temperature conversion)
Write a class called CToFConverter that can be used to convert a Celsius temperature to a Fahrenheit temperature. The constructor needs one argument for the given temperature in Celsius. Do not include any “set” methods in your class.

▶ Exercise 3.5 (Heat loss and windchill calculator)
Write a class called WindChillCalculator with a constructor that has two double arguments. One us for the temperature in degrees Celsius and the other is the wind speed in kilometers per hour. The constructor should use the formulas in Example 2.22 and Example 2.23 to do the calculations. To use the heat loss formula in Example 2.23 you will have to convert the input wind speed from kilometers per hour to meters per second. As an example if the wind speed is 30 km/hr and the temperature is -15C then the windchill is approximately -33.8 and the heat loss is approximately 1487.2. Include set methods for the temperature and for the wind speed.
Exercise 3.6 (Making change)
Write a class called ChangeHelper that helps a cashier give change to a customer. The constructor has two inputs, (1) the amount due as a double value (e.g., 3.28 is 3 dollars and 28 cents) and the amount received as a double value (e.g., 5.00 is 5 dollars and 0 cents). Also assume that the amount received from the customer is equal to or greater than the amount due. The constructor should calculate the minimum number of dollars, quarters, dimes, nickels, and cents the customer should receive as change. Each of these values should be returned using a “get” method. HINT: First convert the two double numbers to total pennies (multiply by 100), round to the nearest integer, subtract and use / and % a few times to extract the numbers of each type of coin.

Exercise 3.7 (Calculating Easter)
The day and month on which Easter falls can be calculated using quotients and remainders with the following steps involving fifteen variables (a to p) starting with the value of y for the given year.

<table>
<thead>
<tr>
<th>Step</th>
<th>Dividend</th>
<th>Divisor</th>
<th>Quotient</th>
<th>Remainder</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>y</td>
<td>19</td>
<td>–</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>y</td>
<td>100</td>
<td>b</td>
<td>c</td>
</tr>
<tr>
<td>3</td>
<td>b</td>
<td>4</td>
<td>d</td>
<td>e</td>
</tr>
<tr>
<td>4</td>
<td>8b + 13</td>
<td>25</td>
<td>f</td>
<td>–</td>
</tr>
<tr>
<td>5</td>
<td>11(b − d − f) − 4</td>
<td>30</td>
<td>g</td>
<td>–</td>
</tr>
<tr>
<td>6</td>
<td>7a + g + 6</td>
<td>11</td>
<td>h</td>
<td>–</td>
</tr>
<tr>
<td>7</td>
<td>19a + (b − d − f) + 15 − h</td>
<td>29</td>
<td>–</td>
<td>i</td>
</tr>
<tr>
<td>8</td>
<td>c</td>
<td>4</td>
<td>j</td>
<td>k</td>
</tr>
<tr>
<td>9</td>
<td>(32 + 2e) + 2j − k − i</td>
<td>7</td>
<td>–</td>
<td>m</td>
</tr>
<tr>
<td>10</td>
<td>90 + (i + m)</td>
<td>25</td>
<td>n</td>
<td>–</td>
</tr>
<tr>
<td>11</td>
<td>19 + (i + m) + n</td>
<td>32</td>
<td>–</td>
<td>p</td>
</tr>
</tbody>
</table>

The value of n is the month number (3 for March, 4 for April) and the value of p is the day of the month. Write an EasterCalculator class whose constructor has one integer argument for the year. Provide “get methods” for the month number and the day number for Easter.

Exercise 3.8 (An interesting formula for the Fibonacci numbers)
The Fibonacci numbers (F₀, F₁, F₂, . . .) occur often in computer science. They are integers and the sequence beginning with F₀ can be calculated exactly using the recurrence relation Fₙ = Fₙ₋₁ + Fₙ₋₂ where F₀ = F₁ = 1. Later when we learn about loops you can use this formula to calculate them exactly.

However there is an interesting closed formula for the nth Fibonacci number:

\[ F_n = \frac{1}{\sqrt{5}} \left( \left( \frac{1 + \sqrt{5}}{2} \right)^n - \left( \frac{1 - \sqrt{5}}{2} \right)^n \right) \]

This is a strange formula since it appears that the results are not necessarily integers for all n = 0, 1, 2, . . . However, it can be shown that they are all integers (the \( \sqrt{5} \) factors all cancel out).

Write a class called FibonacciCalculator with a constructor taking one integer argument for the value of n. Provide a “get” method to return the value of \( F_n \) using double calculations with
the formula. For example, some exact values are $F_{10} = 55$, $F_{20} = 675$, $F_{30} = 832040$, and $F_{40} = 102334155$ but your program will only give the approximate result $F_{30} = 832040.0000000008$. Why are the answers not quite integers?

What is the largest $F_k$ that can be calculated exactly by truncating the floating point result?

**Exercise 3.9 (Calculating $e^x$)**

A series representing $e^x$ is given by

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots$$

in the sense that, for a given $x$, using more terms gives better approximations to $e^x$. Write a class called `ExpCalculator` with a constructor having one double argument for the value of $x$ and two “get” methods. One returns the accurate value of $e^x$ obtained using `Math.exp(x)` and the other returns the approximate value obtained using this series up to the term in $x^4$. Do not use the `Math.pow` method to compute the powers. Instead write the approximation in the form

$$e^x = 1 + x(1 + x(1/2 + x(1/6 + x(1/24))))$$

Use your class to discover the range of $x$ values for which the approximation agrees with the more exact value to at least 5 significant figures.

For example, when $x$ is 0.1 the accurate value is 1.1051709180756477 and the value from the series is 1.105170833333332, showing that for $x = 0.1$ the two results agree to at least 5 significant figures (in fact, 7 significant figures).