Chapter 4

Objects: Packaging Computation

Java is an *object-oriented language*. This means useful work is done through computational elements called *objects*. A variable is an instance of a type, such as an `int`, `double`, or `boolean`. These built-in types are called *primitive types* because they provide simply a value. An object, which also may be represented by a variable, is an instance of a programmer-defined type. A programmer-defined type can specify both value and behavior.

4.1 Classes

A programmer defines the characteristics of a type of object through a *class*. The term *class* as used here means *class of objects*. A class is a template or pattern that defines the structure and capabilities of an object. All objects that are members of the same class share many common characteristics. A class can define both *data* and *behavior*:

- data—objects can store primitive values (`int`s, `boolean`s, etc.), as well as references to other objects
- behavior–objects can specify operations that produce useful results to the user of the object

Initially we will define simple classes that allow us to create an object that can be used as a miniprogram. While a class can be defined within the DrJava interpreter, it is more convenient to use the editor (also known as the Definitions pane). Recall from geometry the formulas for the circumference and area of a circle given its radius. Figure 4.1 illustrates.

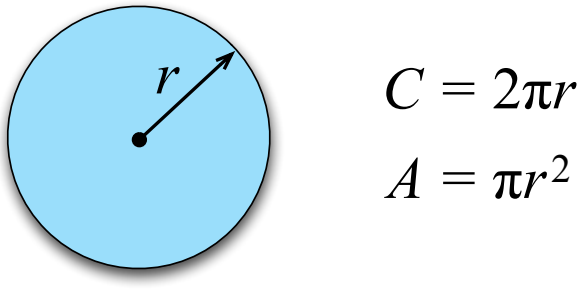

Figure 4.1: Formulas for the circumference and area of a circle given its radius: $C = 2\pi r$ and $A = \pi r^2$

Enter the code found in `CircleCalculator` (4.1) into the DrJava editor and save it as `CircleCalculator`. DrJava will then create a file named `CircleCalculator.java` that stores the source code you type in.

```
public class CircleCalculator {
    private final double PI = 3.14159;
    public double circumference(double radius) {
        return 2 * PI * radius;
    }
    public double area(double radius) {
        return PI * radius * radius;
    }
}
```

Listing 4.1: `CircleCalculator`—a class used to create `CircleCalculator` objects that can compute the circumference and area of circles

In this `CircleCalculator` class:

- Class declaration:

  ```
  public class CircleCalculator {
  ```

  The reserved word `class` signifies that we are defining a new class named `CircleCalculator`. The reserved word `public` when used before `class` indicates that the class is meant for widespread use. Most of the classes we consider will be public classes. The open curly brace (`{`) marks the beginning of the class body.

- Attribute declaration:

  ```
  private final double PI = 3.14159;
  ```

  This declaration means that every `CircleCalculator` object created will have a double-precision floating point constant named `PI`. The reserved word `private` indicates that this constant is only available to code within this class; it is not accessible to code outside of this class. It is a constant since it is declared `final`, and so it will always have the value of 3.14159. `PI` is created and initialized when the object itself is created.

- Method declaration:

  ```
  public double circumference(double radius) {
  ```

  The parentheses following the name `circumference` indicate that this is a *method declaration*. A *method* is a named collection of Java statements that can be executed as a miniprogram.

  – The `public` specifier means any code using `CircleCalculator` objects can invoke this method.

  – The `double` specifier before its name indicates the type that this method will return. A `return` statement must appear somewhere within this method's body, and the `return` statement must have an associated expression assignment-compatible with `double`.

  – This method's name is `circumference`. Clients that use objects of this class will *call*, or *invoke*, this method by its name.

  - Parentheses surround the method's *parameter list*, also known as its *argument list*. Clients are required to pass information to the method when its parameter list is not empty. Here, a client must provide a single `double` value when it calls this method. This parameter named `radius` is a variable that is used within the method body to participate in the computation.
  - The open curly brace marks the beginning of the *method body*.

  Unlike some other programming languages, Java does not allow a method to be defined outside the context of a class.

- Method body:

```
return 2 * PI * radius;
```

  A method's body consists of Java statements. Unlike in the DrJava interpreter, all Java statements must be terminated with a semicolon. This method contains only one statement which computes the product of three numbers:

  - `2`, a literal constant,
  - `PI`, the symbolic, programmer-defined constant mentioned above, and
  - `radius`, the parameter—the caller provides the value of `radius` as we will see shortly.

  The method returns the result of this expression to the caller.

- End of method body:

```
}
```

  The close curly brace after the `return` statement marks the end of the method's body. It matches the open curly brace that marks the beginning of the method's body.

- `area()` is a method similar to `circumference()`.

- End of class:

```
}
```

  The last close curly brace terminates the class definition. It matches the open curly brace that marks the beginning of the class definition. Curly braces with a Java class must be properly nested like parentheses within an arithmetic expression must be properly nested. Notice that the curly braces of the methods properly nest within the curly braces of the class definition.

In order to use this `CircleCalculator` class we must compile the source code into executable bytecode. DrJava has three ways to compile the code:

- select the Compile button in the toolbar,
- from the main menu, select *Tools*, then *Compile current document*, or
- press the Shift F5 key combination.

The Compiler Output pane should report

Compiler Output

```
Compilation completed.
```

If the compiler pane indicates errors, fix the typographical errors in the source code and try again. Once it compiles successfully, we can experiment with `CircleCalculator` objects in the DrJava interpreter:

Interactions

```
Welcome to DrJava.  Working directory is /Users/rick/java
> CircleCalculator c = new CircleCalculator();
> c
CircleCalculator@e8e5a7
> c.circumference(10.5)
65.97339
> c.area(10.5)
346.3602975
> x = c.circumference(10.5);
> x
65.97339
> c.PI
IllegalAccessException: Class
koala.dynamicjava.interpreter.EvaluationVisitor can not access a
  member of class CircleCalculator with modifiers "private final"
  at sun.reflect.Reflection.ensureMemberAccess(Reflection.java:65)
  at java.lang.reflect.Field.doSecurityCheck(Field.java:954)
  at java.lang.reflect.Field.getFieldAccessor(Field.java:895)
  at java.lang.reflect.Field.get(Field.java:357)
> c.PI = 2.5;
Error: This object cannot be modified
```

Let us analyze what transpired during this interactive session:

- First we created a `CircleCalculator` object, or instance, named `c`.

  ```
  > CircleCalculator c = new CircleCalculator();
  ```

  The declared type of variable `c` is `CircleCalculator`. The variable `c` here represents a reference to an *object*. The terms *object* and *instance* are interchangeable. The reserved word `new` is used to create an object from a specified class. Although it is a word instead of a symbol (like `+` or `*`), `new` is technically classified as an operator. The word following the the `new` operator must be a valid type name. The parentheses after `CircleCalculator` are required. This first line is a Java declaration and initialization statement that could appear in a real Java program.

  While `c` technically is a reference to an object, we often speak in less formal terms, calling `c` itself an object. The next item highlights why there truly is a distinction between an object and a reference to that object.

- Next, we evaluated the variable `c`.

  ```
  > c
  CircleCalculator@e8e5a7
  ```


Note the somewhat cryptic result. Later (§ 14.2) we will cover a technique to make our objects respond in a more attractive way. By default, when the interpreter evaluates an object of a class we create, it prints the object's class name, followed by the `@` symbol, and then a number in hexadecimal (base 16) format. Hexadecimal numbers use the digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f, You do not need to know the hexadecimal number system for our purposes; simply be aware that the Java runtime environment (JRE) maps the number to an address in the computer's memory. We can view this number as a unique ID or serial number for the object.[1]

The variable itself thus holds the address of an object, not the contents of the object itself. That is why we properly call `c` a reference to a `CircleCalculator` object. Again, often we will be less formal, calling `c` a `CircleCalculator` object.

What can we do with object `c`? `c` is a `CircleCalculator` object, so we can use `c` to calculate the circumference and area of circles.

- Once we have created the object `c` we use the `circumference()` method to compute the circumference of a circle with the radius 10.5.

```
> c.circumference(10.5)
65.97339
```

We say we are "calling" or "invoking" the `circumference()` method of the `CircleCalculator` class on behalf of object `c` and passing to it the radius 10.5. In order to call a method on behalf of an object, we write the name of the object (in this case `c`), followed by the dot operator (`.`), followed by the method's name, and then a set of parentheses:

object . *method* ( )

Within the Interactions environment we are a *client* of object `c` and a *caller* of its `circumference()` method.

- Next, we call the `area()` method to compute the area of a circle with the radius 10.5.

```
> c.area(10.5)
346.3602975
```

- We can assign the result of calling the method to a variable if we like:

```
double x = c.circumference(10.5);
```

- Finally, we attempt to use `c`'s `PI` constant. Since it is declared `private`, the interpreter generates an error.

```
> c.PI
IllegalAccessException: Class
koala.dynamicjava.interpreter.EvaluationVisitor can not access a
  member of class CircleCalculator with modifiers "private final"
  at sun.reflect.Reflection.ensureMemberAccess(Reflection.java:65)
  at java.lang.reflect.Field.doSecurityCheck(Field.java:954)
  at java.lang.reflect.Field.getFieldAccessor(Field.java:895)
  at java.lang.reflect.Field.get(Field.java:357)
```

Since we cannot look at `c.PI` we certainly cannot modify it:

[1] Why is hexadecimal numbering used? Hexadecimal numbers are commonly used for specifying memory addresses because the are easily converted to binary (base 2) numbers and they can be expressed in fewer digits than their decimal (base 10) counterparts.


```
> c.PI = 2.5;
Error: This object cannot be modified
```

As an experiment, see what happens if you change the declaration of `PI` to be `public` instead of `private`. Next, remove the `final` specifier and try the above interactive sequence again.

Now that we have a feel for how things work, let us review our `CircleCalculator` class definition. A typical class defines *attributes* and *operations*. `CircleCalculator` specifies one attribute, `PI`, and two operations, `circumference()` and `area()`. An attribute is also called a *field* or *instance variable*. In the case of `CircleCalculator`, our instance "variable" is really a constant because it is declared `final`. A more common name for an operation is *method*. A methods is distinguished from an instance variable by the parentheses that follow its name. These parentheses delimit the list of parameters that the method accepts in order to perform a computation. Both `circumference()` and `area()` each accept a single `double` value as a *parameter*, also known as an *argument*. Notice that the type of the parameter must be specified in the method's parameter list. The parameter has a name so that it can be used within the method's body.

A method may be written to accept as many parameters as needed, including none. To see how multiple parameters are used consider the task of computing the perimeter and area of a rectangle as shown in Figure 4.2. Whereas a

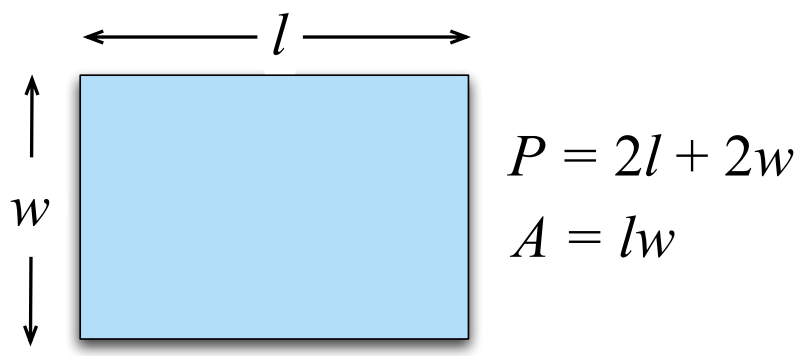

Figure 4.2: Formulas for the perimeter and area of a rectangle given its length and width: $P = 2l + 2w$ and $A = lw$.

a circle's perimeter and area can be computed from only one piece of information (its radius), two pieces of information are required to compute the perimeter and area of rectangles—length and width. If a method requires multiple pieces of information to do its job, multiple parameters are necessary. (Do not confuse the words *parameter* and *perimeter*!) Multiple parameters are separated by commas, as shown in `RectangleCalculator` (4.2):

```
public class RectangleCalculator {
    public double area(double length, double width) {
        return length * width;
    }
    public double perimeter(double length, double width) {
        return 2 * length + 2 * width;
    }
}
```

Listing 4.2: `RectangleCalculator`—a class used to create `RectangleCalculator` objects that can compute the areas and perimeters of rectangles

The types of each parameter must be specified separately as shown. Here, both parameters are of type `double`, but in general the parameter types may be different.

In the interpreter we are the clients of `CircleCalculator` objects and `RectangleCalculator` objects. As we will soon see, code that we or others may write can also use our `CircleCalculator` and `RectangleCalculator` classes to create and use `CircleCalculator` and `RectangleCalculator` objects. We refer to such code as *client code*.

The return type of a method must be specified in its definition. The value that a method may return can be of any legal Java type. If a variable can be declared to be a particular type, a method can be defined to return that type. In addition, a method can be declared to "return" type `void`. A `void` method does not return a value to the client. Based on what we have seen so far, `void` methods may appear to be useless, but in fact they are quite common and we will write a number of `void` methods. While `void` is a legal return type for methods, it is illegal to declare a variable to be type `void`.

Methods get information from clients via their parameter lists and return information to clients via their return statement(s). An empty parameter list (a pair of parentheses with nothing inside) indicates that the method can perform its task with no input from the client. If a client tries to pass information to a method defined to not accept any parameters, the compiler will issue an error. Likewise, any attempt by a client to pass the wrong kinds of parameters to a method will be thwarted by the compiler.

A method has exactly one definition, but it can be invoked many times (even none).

4.2 Comments

Good programmers annotate their code. They insert remarks that explain the purpose of a method or why they chose to write a section of code the way they did. These notes are meant for human readers, not the compiler. It is common in industry for programs to be reviewed for correctness by other programmers or technical managers. Well-chosen identifiers (§ 2.3) and comments can aid this process, especially when the comments disagree with the code! Also, in practice, teams of programmers develop software. A different programmer may be required to finish or fix a part of the program written by someone else. Well-written comments can help others understand new code quicker and increase their productivity modifying old or unfinished code. While it may seem difficult to believe, even the same programmer working on her own code months later can have a difficult time remembering what various parts do. Comments can help greatly.

Java supports three types of comments:

- single line comments
- block comments
- documentation comments

We'll describe the first two kinds of comments here; the documentation comments, which we will not cover here, are used to annotate the source code in such a way so that a special tool, `javadoc`, can create external documentation that can be read with any web browser.

Any text contained within comments is ignored by the compiler and the DrJava interpreter. Comments do not become part of the compiled bytecode, so providing generous comments does not affect the size or efficiency of the finished program in any way.

The first type of comment is useful for writing a single line remark:


```
>    //  Compute the circumference
> double circumference = 2 * PI * radius;
```

Here, the comment explains what the statement that follows it is supposed to do. The comment begins with the double forward slash symbols (`//`) and continues until the end of that line. The compiler and interpreter will ignore the `//` symbols and the contents of the rest of the line. This type of comment is also useful for appending a short comment to the end of a statement:

```
> count = 0;  //  Reset counter for new test
```

Here, an executable statement and the comment appear on the same line. The compiler or interpreter will read the assignment statement here, but it will ignore the comment. This is similar to the preceding example, but uses one line of source code instead of two.

The second type of comment is begun with the symbols `/*` and is in effect until the `*/` symbols are encountered. The `/* . . . */` symbols delimit the comment like parentheses delimit a parenthetical expression. Unlike parentheses, however, these block comments cannot be nested within other block comments. The block comment is handy for multi-line comments:

```
> /*  Now we are ready to compute the
    circumference of the circle..  */
> double circumference = 2 * PI * radius;
```

What should be commented? Avoid making a remark about the obvious; for example:

```
result = 0;  //  Assign the value zero to the variable named result
```

The effect of this statement is clear to anyone with even minimal Java programming experience. Thus, the audience of the comments should be taken into account. Generally, “routine” activities require no remarks. Even though the *effect* of the above statement is clear, its *purpose* may need a comment. For example:

```
result = 0;  //  Ensures 'result' has a well-defined minimum value
```

This remark may be crucial for readers to completely understand how a particular part of a program works. In general, programmers are not prone to providing too many comments. When in doubt, add a remark. The extra time it takes to write good comments is well worth the effort.

4.3 Local Variables

`TimeConverter` (4.3) conveniently packages into a class the seconds to hours:minutes:seconds conversion from Chapter 2. Its `convert()` method returns a string separating the hours, minutes, and seconds:

```
public class TimeConverter {
    public String convert(int seconds) {
        //  First compute hours, minutes, and seconds
        int hours = seconds/3600;
        seconds = seconds % 3600;
        int minutes = seconds/60;
```



```
        seconds = seconds % 60;
        //  Next, construct and return the string with the results
        return hours + " hr, " + minutes + " min, " + seconds + " sec";
    }
}
```

Listing 4.3: `TimeConverter`—converts seconds to hours, minutes, and seconds

In `TimeConverter` (4.3) the `convert()` method works as follows:

- The first part duplicates the work we did in an earlier interactive session, albeit with longer variable names:

  ```
  int hours = seconds/3600;
  seconds = seconds % 3600;
  int minutes = seconds/60;
  seconds = seconds % 60;
  ```

  The number of hours, minutes, and seconds are calculated.

- The second part simply assembles a string containing the results and returns the string. The string concatenation operator (+) works nicely to build our custom string combining known labels (*hr*, *min*, and *sec*) with to-be-determined values to put with these labels.

As shown in this Interactions session, the method works as desired:

Interactions

```
Welcome to DrJava.  Working directory is /Users/rick/java
> t = new TimeConverter();
> t.convert(10000)
"2 hr, 46 min, 40 sec"
> t.convert(10025)
"2 hr, 47 min, 5 sec"
> t.convert(0)
"0 hr, 0 min, 0 sec"
> t.convert(1)
"0 hr, 0 min, 1 sec"
> t.convert(60)
"0 hr, 1 min, 0 sec"
```

The variable `seconds` is a parameter to the method. The variables `hours` and `minutes` are declared and used *within* the `convert()` method. This means these particular variables can only be used within the `convert()` method. We say they are *local* to `convert()`, and they are called *local variables*. If `TimeConverter` had any other methods, these local variables would not be available to the other methods. Local variables declared within other methods would not be available to `convert()`. In fact, another method could declare its own local variable named `hours`, and its `hours` variable would be different from the `hours` variable within `convert()`. Local variables are useful because they can be used without fear of disturbing code in other methods. Compare a local variable to an instance variable like `PI` from `CircleCalculator` (4.1). `PI` is declared outside of both the `circumference()` and `area()` methods, and it can be used freely by both methods. The variable `hours` in `TimeConverter` (4.3) is declared inside the `convert()` method, not outside, and so it can only be used by statements within `convert()`.

Unlike instance variables, local variables cannot be declared to be `private` or `public`. The `private` specifier means available only to code within the class. Local variables are even more restricted, as other methods within the class cannot access them.

4.4 Method Parameters

A parameter to a method is a special local variable of that method. Whereas other local variables are assigned values within the method itself, parameters are assigned from outside the method during the method invocation. Consider the following interactive session using `CircleCalculator` (4.1):

Interactions

```
Welcome to DrJava.  Working directory is /Users/rick/java
> c = new CircleCalculator();
> c.circumference(10)
62.8318
> x = 5
5
> c.circumference(x)
31.4159
```

In the first use of `circumference()`, the value 10 is assigned to the parameter `radius`, and the code within `circumference()` is then executed. In the second use of `circumference()`, the value of the variable `x` is assigned to `radius`, and then the code within `circumference()` is executed. In these examples we more precisely refer to 10 and `x` as *actual parameters*. These are the pieces of information *actually* supplied by the client code to the method during the method's invocation. Inside the definition of `circumference()` within `CircleCalculator`, `radius` is known as the *formal parameter*. A formal parameter is the name used during the *formal* definition of its method.

Note in our interactive session above that the variable `x` that is used as an actual parameter to the call of `circumference()`:

```
c.circumference(x)
```

has a different name than the actual parameter, `radius`. They are clearly two different variables. We could chosen the names to be the same, as in:

Interactions

```
Welcome to DrJava.  Working directory is /Users/rick/java
> c = new CircleCalculator();
> radius = 5
5
> c.circumference(radius)
31.4159
```

but even in this case where the actual parameter has the same name as the formal parameter, we have two separate variables! Since the formal parameter `radius` is a special local variable, it has meaning only within the definition of `circumference()`. It cannot interfere with any variables anywhere else. The formal parameter exists in the computer's memory only while the method is being called. By contrast, the actual parameter exists before the call and after the call.

A formal parameter then is a placeholder so the code within the method can access a piece of data passed in by the client code. Like local variables, formal parameters are isolated from all other variables within the code or interactive session.

4.5 Summary

- A class defines the structure of an object. It is like a plan or blueprint that defines objects.
- Class names follow the identifier naming rules.
- Objects provide a service to clients.
- A client is the user of an object.
- Objects must be created to provide a service to clients.
- The `new` operator creates an object from its class description.
- An object is an active agent that provides a service to clients.
- Objects can possess both value and behavior.
- Object values are called attributes.
- Object behaviors are specified via methods.
- The dot (`.`) operator associates a method call with a particular object.
- Like a mathematical function, a method can accept information via parameters and compute a result.
- The types of a method's parameters must be specified within its parentheses.
- A method's result is indicated by a `return` statement.
- `public` class elements are accessible anywhere; `private` class elements are only accessible within the class itself.
- Comments come in three varieties: single line, multiple line, and documentation comments.
- Judicious comments improve the readability of source code.
- Variables declared within a method are called local variables.
- Local variables are isolated from all other variables within the code or interactive session.
- Parameters are special local variables used to communicate information into methods during their execution.
- A parameter used in the method definition is called a formal parameter.
- A parameter used during a method invocation is called an actual parameter.
- A method has exactly one definition, but it can be invoked zero or more times.

4.6 Exercises

1. What is a Java class?
2. What is an object? How is it related to a class?
3. How is object data specified?
4. How is object behavior specified?
5. How is a `public` class different from a non-`public` class?
6. What is a method?
7. What symbols delimit a class's body?
8. What do the `public` and `private` specifiers indicate within a class body?
9. What goes in the parentheses of a method's declaration?
10. What symbols delimit a method's body?
11. Within DrJava, how is a class compiled into bytecode?
12. What is client code?
13. What is a field?
14. What is an instance variable?
15. What is an attribute?
16. What is an operation?
17. How does client code create an instance of a class? Provide a simple example.
18. What happens when you try to evaluate an object directly within the interpreter? Loosely speaking, what the result mean?
19. What happens when clients attempt to evaluate the `private` data of an object?
20. Devise a class named `EllipseCalculator` that provides methods to compute the area and approximate perimeter of an ellipse given its major and minor radii (do a web search to see what is meant by these terms and to find appropriate formulas if you are uncertain). (Hint: to find the square root of a value in Java use `Math.sqrt()`, as in
$$\sqrt{x} = \texttt{Math.sqrt}(x)$$
21. What are the three kinds of comments supported by Java?
22. What is the purpose of comments?
23. What does the compiler do with the contents of comments?
24. Experiment putting comments inside of other comments. Under what circumstances is this possible?
25. How is a local variable different from an instance variable?
26. Rank the following kinds of variables from least restrictive most restrictive: `public` instance variable, local variable, `private` instance variable.

27. What advantage does a local variable have over an instance variable?
28. What is the difference between a formal parameter and an actual parameter?
29. What are the formal parameters of the `area` method of `RectangleCalculator` (4.2)?
30. How many different definitions may a method have?
31. How many times may a method be called?

