Chapter 3: All About Methods

All About Methods

We're going to need several methods and constructors for this class. I'm going to work from the simpler methods to the more complex. Along the way, I'll also show you how one method in a class can use another. By the time we finish the first version of our Temperature class, we'll have three methods and two constructors.

The simpler methods in this class are *accessor methods*. They access data in our objects and give it to our clients. (We call a class or method that's using our class its *client*. We'll write our client class in the next chapter.)

We said our Temperature class would provide Celsius and Kelvin temperatures when our client provides the Fahrenheit value, so let's write the methods to access those attributes first. We're going to call these methods *get methods*. Java programmers often write get methods to get data from a class for their clients. Here's my version of the two methods:

```
/**
* getDegreesCelsius retrieves the Celsius temperature value
*
* @return a double value containing the Celsius temperature
*/
public double getDegreesCelsius() {
    return degreesCelsius;
}

/**
* getDegreesKelvin retrieves the Kelvin temperature value
*
* @return a double value containing the Kelvin temperature
*/
public double getDegreesKelvin() {
    return degreesKelvin;
}
```

I've preceded each method with a Javadoc comment that will document its purpose. These two methods, getDegreesCelsius() and getDegreesKelvin(), each return a double value to the client. The second word in each of their first lines tells us that. Neither of them needs to receive any information, so they have empty parameter lists. They each have one statement (return) that returns the value from an instance variable to the method's caller, the client. We'll see how to use this information in our main() method, which will be our client for these methods.

Okay, the get methods will provide information to our client. How do we get the Fahrenheit temperature in the first place? We'll use a *set method* for that, since we're going to be setting the value in one or more attributes. It's also called a *mutator method* in more formal terms.

Java programmers use set methods to let clients change data in an object. I began the get method names with "get," so I'll use "set" to start the name of my set method:

```
/**
* The setDegreesFahrenheit method sets the Fahrenheit temperature
*
* @param degrees The Fahrenheit value to store
*/
public void setDegreesFahrenheit(double degrees) {
    degreesFahrenheit = degrees; // set Fahrenheit value
}
```

This method sets our internal Fahrenheit value based on the number our client gives us in the parameter degrees.

Can you think of anything we've overlooked? What's missing?

Hide answer

We've set our Fahrenheit value, but now it's out of sync with our Celsius and Kelvin values. We need to fix the program so that whenever someone changes the Fahrenheit value, the program recalculates the Celsius and Kelvin values.

This method shows one reason we don't give clients direct access to an object's data. We don't want them to change something that might cause problems or give bad results. We want to always be in control of the information in our objects.

In this method, we'll make sure that all three of our temperatures are always in sync. To do that, we'll need to calculate the other temperatures and set them in this method. And that means we need to discuss how to do calculations in Java.

Using Expressions

Java does its calculations in program code called *expressions*. An expression is a combination of symbols that has just one value. Expressions use *operators* to indicate what calculations to do and *operands* to provide information for the calculations.

Java's arithmetic operators are +, -, * (multiplication), / (division), and % (modulus, which I'll explain in the next lesson). Operands can be numeric literals, variables, constants, or other expressions. But an expression always has a single value.

I know this can be confusing, so let me break it down for you.

Name	What It Means	Examples	Value
	Performs calculations in Java and gives a single value as a result	value1 + value2	Sum of value1 and value2
Expression		a * (b + c)	Product of a and the sum of b and c
Operator	Tells Java which calculation to do and then returns a result	+ - * / %	
Operand	Tells Java which value to use in that part of the calculation	4 9.0 value1 a	

Here are the two statements we'll use to do our conversions. Take a look at them, and then I'll explain their parts.

```
degreesCelsius = (degreesFahrenheit - 32.0) * 5.0 / 9.0; // set Celsius
degreesKelvin = degreesCelsius + 273.15; // set Kelvin value
```

These two statements convert the Fahrenheit temperature to Celsius and Kelvin according to the formulas in the program description and store the results in the appropriate variable.

Making a Statement

Note

Statements in Java are roughly equivalent to sentences in a natural language. A *statement* forms a complete unit of execution. There are three basic statement types in Java: expression statements like the ones above, declaration statements that declare new names, and control flow statements like branches and loops that we'll learn about in the next two lessons.

These two statements are assignment statements because they assign values to variables. Every assignment statement must have a variable on the left side of its assignment operator to receive the assigned value. In Java, the assignment operator is the equal sign (=).

On the right side of the assignment operator, there's an expression that calculates a value. After the calculation is done, the statement stores the calculated value into the variable on the left side.

Expressions can be as complex as we need them to be, and Java will evaluate everything on the right side of an assignment before storing the result into the variable on the left.

Java evaluates expressions according to a few simple rules that many programming languages follow. You need to understand them in order to write expressions correctly.

Java's Expression Evaluation Rules

- Java evaluates operations inside parentheses before operations outside them.

- If there are no parentheses, Java evaluates multiplication (*) and division (/) before addition (+) and subtraction (–). Programmers say that multiplication and division have a higher priority of operation than addition and subtraction do. Java always executes operations in priority order.

- When there are multiple operations with the same priority, Java evaluates them from left to right.

In any arithmetic operation, if either of the operands is a floating-point number, the result will be a floating-point number. If both operands are integers, the result will be an integer. In other words, Java doesn't round off numbers for you.

So the first expression above does these steps:

1. Subtract 32.0 from the value in degreesFahrenheit. I used parentheses to subtract before multiplying and dividing.

2. Multiply the result of the subtraction by 5.0.

3. Divide the result of the multiplication by 9.0.

The result is the equivalent Celsius temperature, which the assignment operator stores into degreesCelsius.

The second assignment statement adds 273.15 to the Celsius value to get the Kelvin equivalent and stores it in degreesKelvin.

To make sure that all three values stay in sync, let's add the two statements above to our setDegreesFahrenheit() method so that whenever the Fahrenheit value changes, we recalculate the Celsius and Kelvin values at the same time. That makes our set method look like this:

```
/**
* The setDegreesFahrenheit method sets the Fahrenheit temperature
*
* @param degrees The Fahrenheit value to store
*/
public void setDegreesFahrenheit(double degrees) {
    degreesFahrenheit = degrees; // set Fahrenheit value
    degreesCelsius = (degreesFahrenheit - 32.0) * 5.0 / 9.0; // set Celsius
    degreesKelvin = degreesCelsius + 273.15; // set Kelvin value
}
```

We've converted the Fahrenheit temperature to Celsius and Kelvin, just as we said we would. But we still need two things:

- A constructor for our class to initialize objects (As I mentioned in Lessons 3 and 4, initializing something means giving it a value to begin with, even if that value changes later.)

- A main()method so our program can control the sequence of events in our process

Before we look at our main()method, let's discuss about constructors again.

Constructors and Literals

It's always a good idea to have a constructor. That way we can make sure that our instance variables get the correct initial value, and anyone reading the program can see exactly what we intended.

In this project, though, we're going to build two constructors for our class. The first one will be for someone who already has a Fahrenheit temperature and wants to create a Temperature object with that value. So this constructor will have one parameter—a double value to receive the Fahrenheit value. We'll want our constructor to store that value in degreesFahrenheit and to keep things in sync by calculating and storing the equivalent Kelvin and Celsius values.

Hang on a second! Didn't we just do that? We did, and we called it setDegreesFahrenheit(). We want our constructor to do exactly the same thing.

We could copy and paste the statements from the method into the constructor, but that would mean we'd have duplicate code, which most programmers consider poor programming. The longer your code is, the more difficult it is for others to analyze, and the more likely it is that an error will sneak in.

We can avoid the duplicate code if our constructor calls setDegreesFahrenheit() and uses it to do the work. That makes our constructor very simple—it'll just pass the Fahrenheit value to the method, and the method will do the work. Here's the constructor that does that:

```
/**
* This constructor for Temperature sets the Fahrenheit
* value to the value from degrees, then calculates
* equivalent Celsius and Kelvin values using setDegreesFahrenheit()
*
* @param degrees degrees Fahrenheit
*/
public Temperature(double degrees) {
    setDegreesFahrenheit(degrees);
}
```

Last, I want to build a default constructor in case someone wants to create a Temperature object without having a Fahrenheit value to give us. So there won't be a parameter for this constructor. We still need to initialize our object and have all three values in sync, so we'll have to pick a value to use. It can be any valid Fahrenheit temperature, so I'm going to pick zero.

I could let this constructor call setDegreesFahrenheit() just as the other one did, but I want to illustrate one other Java feature since you'll see it in many classes with multiple constructors. I'll show you the constructor, and then I'll explain what it does.

```
/**
* Default constructor for Temperature sets the default
* value of the Temperature object to 0.0 degrees
* Fahrenheit with Celsius and Kelvin equivalents
*/
public Temperature() {
    this(0.0);
}
```

Just like the first constructor, it's a one-liner. In this case, though, instead of calling a method by name, it uses the keyword this.

Java uses the keyword this to refer to the object that contains the keyword. When it appears with parentheses, it tells Java to call a constructor. Since there's a number in the parentheses, it's calling the constructor that has a double parameter. So this constructor calls the other constructor and gives it an argument of 0.0. The other constructor in turn calls setDegreesFahrenheit() with that same argument. And we only have to write the code that sets and converts the temperature one time!

That wraps up our Temperature class. Just to make sure we're on the same page, here's my complete class so you can compare it to yours. I've followed Java traditions and put my instance variables at the top of the class followed by the constructors, with all the other methods after that.

```
/**
* Temperature stores a temperature in Fahrenheit, Celsius,
* and Kelvin scales.
*
* @author Merrill Hall
* @version 1.0
*/
public class Temperature {
    private double degreesFahrenheit; // Fahrenheit temperature
    private double degreesCelsius; // Celsius temperature
    private double degreesKelvin; // Kelvin temperature

    /**
     * Default constructor for Temperature sets the default
     * value of the Temperature object to 0.0 degrees
     * Fahrenheit with Celsius and Kelvin equivalents
     */
    public Temperature() {
        this(0.0);
    }

    /**
     * This constructor for Temperature sets the Fahrenheit
     * value to the value from degrees, then calculates
     * equivalent Celsius and Kelvin values using the method
     * setDegreesFahrenheit()
     *
     * @param degrees degrees Fahrenheit
     */
    public Temperature(double degrees) {
        setDegreesFahrenheit(degrees);
    }

    /**
     * The setDegreesFahrenheit method sets the Fahrenheit temperature
     * and its Celsius and Kelvin equivalents
     *
     * @param degrees The Fahrenheit value to store
     */
    public void setDegreesFahrenheit(double degrees) {
        degreesFahrenheit = degrees; // set Fahrenheit value
        degreesCelsius = (degreesFahrenheit - 32.0) * 5.0 / 9.0; // set Celsius
```

```
    degreesKelvin = degreesCelsius + 273.15; // set Kelvin value
}

/**
 * getDegreesCelsius retrieves the Celsius temperature value
 *
 * @return a double value containing the Celsius temperature
 */
public double getDegreesCelsius() {
    return degreesCelsius;
}

/**
 * getDegreesKelvin retrieves the Kelvin temperature value
 *
 * @return a double value containing the Kelvin temperature
 */
    public double getDegreesKelvin() {
        return degreesKelvin;
    }
}
```

When you're ready to tackle the last topic, go to the next chapter.