Chapter 9

Immutable Objects

9.1 What Does Immutability Mean?

Many people read the statement that "Java `Strings` are immutable", and then write code like this:

```
String metal;
metal = "lead";
metal = "gold";
```

and ask "Didn't that just change the string?" No, that code does not change lead into gold.

Let's take a look at a memory diagram after the first assignment. `metal` is a *reference* to an area of memory (called the **heap**) where the string `"lead"` has been allocated.

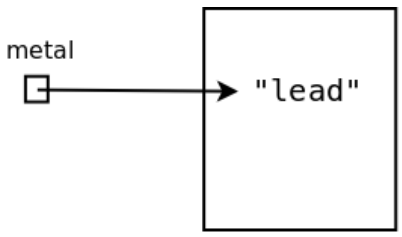

Figure 9.1: Reference to a String on the heap

After the second assignment, the *reference* has changed to refer to another portion of the heap that contains the string `"gold"`. The string `"lead"` is still on the heap, unmodified. It's just that nobody is referring to it any longer.

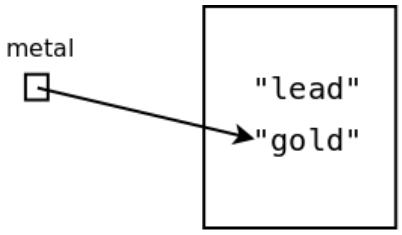

Figure 9.2: Reference to a second String on the heap

What would "changing the string" look like? In some languages, a string is treated as if it were an array of characters, and you could write code like this:

```
metal = "gold";
metal[0] = 's'; // change first letter to 's'
```

Java won't let you do that because the string itself, which is out there on the heap, cannot be modified.

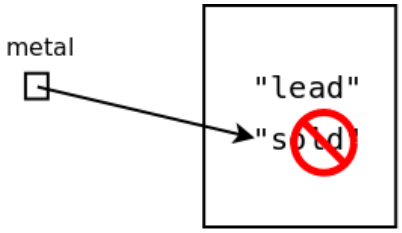

Figure 9.3: Non-Java modified string (illegal in Java)

But there's nothing to stop you from creating a brand new `String` on the heap and changing the value of `metal` to refer to that new `String`, as shown in Figure 9.4:

```
metal = "s" + metal.substring(1, 4);
```

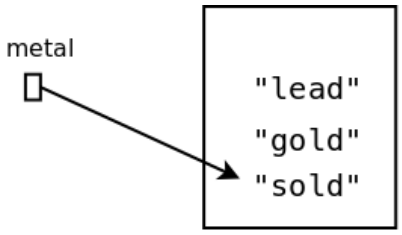

Figure 9.4: Reference to a newly created String on the heap

9.2 Methods and the `Scanner` Class

The *Think Java* book's subtitle is "How to Think Like a Computer Scientist," and in that spirit, places more emphasis on the concepts of computer science—developing and analyzing algorithms. In order to do this, the book deliberately avoids going into the minutiae of the Java language. At this point, however, we have to get into a detail of the `Scanner` class that comes up when you are generalizing and modularizing your code.

Whenever you generalize a method for doing input, do *not* create the `Scanner` inside the method. Instead, you must pass a `Scanner` object as one of the parameters.

Consider the following program that asks a user for two prices and calculates the percentage change:

```
import java.util.Scanner;

public class RepeatedCode {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    double originalPrice;
    do {
      System.out.print("Enter original price: ");
      originalPrice = input.nextDouble();
      if (originalPrice <= 0) {
        System.out.println("Price must be greater than zero.");
      }
    } while (originalPrice <= 0);

    double newPrice;
    do {
      System.out.print("Enter new price: ");
      newPrice = input.nextDouble();
      if (newPrice <= 0) {
        System.out.println("Price must be greater than zero.");
      }
    } while (newPrice <= 0);

    double pctChange = 100.0 *
      (newPrice - originalPrice) / originalPrice;
    System.out.printf("Price change: %.1f%%\n", pctChange);
  }
}
```

The code for the two inputs is identical except for the prompt (and the variable name). Here is a way to generalize that code by creating a method to get the input and having the input prompt as its parameter:

```
import java.util.Scanner;

public class GeneralInput1 {

  public static double getPrice(String prompt) {
    Scanner input = new Scanner(System.in);
    double price;
    do {
      System.out.print(prompt);
      price = input.nextDouble();
      if (price <= 0) {
        System.out.println("Price must be greater than zero.");
      }
    } while (price <= 0);
    return price;
  }


  public static void main(String[] args) {
    double originalPrice = getPrice("Enter original price: ");
    double newPrice = getPrice("Enter new price: ");
    double pctChange = 100.0 *
      (newPrice - originalPrice) / originalPrice;
    System.out.printf("Price change: %.1f%%\n", pctChange);
  }
}
```

This program works, but there's a trap hidden in it. Every time the code calls `getPrice`, a new `Scanner` is created, opening a new connection to the keyboard. This is inefficient, but not fatal—until someone tells you that you should always close an I/O device and you add the code `input.close();` between the end of the loop and the `return` statement.

Once the `input` is closed, its connection to the keyboard is broken *and cannot be re-opened.* Now, when you run the program, you get this result:

```
Enter original price: 3.50
Enter new price: Exception in thread "main"
  java.util.NoSuchElementException
        at java.base/java.util.Scanner.throwFor(Scanner.java:937)
        at java.base/java.util.Scanner.next(Scanner.java:1594)
        at java.base/java.util.Scanner.nextDouble(Scanner.java:2564)
        at BadClose.getPrice(BadClose.java:10)
        at BadClose.main(BadClose.java:22)
```

In order to avoid the minor problem of creating multiple connections to the keyboard and the major problem of inadvertently closing the connection before you want to, your code must create the `Scanner` once, and once only. Here is code that solves the problem. It creates the `Scanner` once in `main`, closes it only once the end of `main`, and passes it as an additional argument to the `getPrice` method:

```
import java.util.Scanner;

public class GeneralInput2 {

  public static double getPrice(Scanner in, String prompt) {
    double price;
    do {
      System.out.print(prompt);
      price = in.nextDouble();
      if (price <= 0) {
        System.out.println("Price must be greater than zero.");
      }
    } while (price <= 0);
    return price;
  }


  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    double originalPrice = getPrice(input, "Enter original price: ");
    double newPrice = getPrice(input, "Enter new price: ");
    double pctChange = 100.0 *
      (newPrice - originalPrice) / originalPrice;
    System.out.printf("Price change: %.1f%%\n", pctChange);
    input.close();
  }
}
```

9.3 The Software Development Cycle

Up to this point, you have been going through a two-step cycle when writing programs:

1. Write code
2. Test code

You go through this cycle until the testing shows no errors (or at least errors that you have anticipated), and your program is complete. Generalizing the code is part of this write-and-test cycle.

This is only a part of something called the *software development cycle*, which has many different definitions. Some have as few as five steps, others as many as ten.

Here is a seven-step definition of the software development cycle, adapted from https://en.wikipedia.org/wiki/Systems_development_life_cycle:

1. **Preliminary Analysis**: What is the problem you (or the customer) are trying to solve? What does the customer want the program to do?
2. **System Analysis and Design**: Determine what the program needs to do. This step requires you to talk to the end users of the application.
3. **System Design**: Take the result of the preceding step and develop the data structures and methods needed to accomplish the tasks the application does.
4. **Development**: This is the stage where the code is written.
5. **Testing**: Check to see that the program works as per the requirements. This is best done by a group of people who did *not* write the code.
6. **Deployment**: The program is sent out to the end users.
7. **Maintenance**: Bug fixes and feature enhancements happen here.

In each of these stages, it's possible to go back to the preceding stage or further. For example, if a bug is found in the deployment phase, you might need to go back to development. If it turns out that, upon deployment, the program design doesn't adequately meet users' needs, you might need to go all the way back to the System Analysis and Design phase.

9.4 Exercises

Exercise 9.1 In this exercise, we will once again rewrite the "dew point" program. Prompt the user for the air temperature in degrees Celsius and the relative humidity as a percent, and calculate the dew point with this formula:

$$dewPoint = temperature - \frac{100 - relHumidity}{5}$$

Generalize the code to use a method to get the temperature. This method should accept only numbers in the range -90 to 60 degrees Celsius, giving an error message if the temperature is out of range. (The range given here includes the lowest and highest recorded temperatures on earth.)

Write a method to get the relative humidity. This method should accept only numbers in the range 0 to 100, giving an error message if the relative humidity is out of range.

Here is what output might look like when the user enters several out-of-range values:

```
Enter temperature in degrees C: 85
Value must be in range -90 to 60.
Enter temperature in degrees C: -100
Value must be in range -90 to 60.
Enter temperature in degrees C: 20
Enter relative humidity as a percent: -20
Value must be in range 0 to 100.
Enter relative humidity as a percent: 105
Value must be in range 0 to 100.
Enter relative humidity as a percent: 80
The dew point is 16.0 degrees C.
```

Can you generalize further to use only one method to get both values? Hint: That method would need a `Scanner`, a prompt string, a minimum, and a maximum value as its parameters.

Exercise 9.2 There are two ways to represent latitudes and longitudes. One is as a decimal degree: 35.47°, and the other is as degrees, minutes, and seconds: 35° 28′ 12″ (also called "DMS" format).

Write a command-line program named *Angle.java* that converts between these formats. The program will get its input from command line arguments.

- If there is only one argument, it is presumed to be an angle in decimal format, and your program will convert it to degree-minute-second format.

- If there are two arguments, they are presumed to be degrees and minutes, and your program will convert it to decimal format.
- If there are three arguments, they are presumed to be degrees, minutes, and seconds; and your program will convert it to decimal format.
- For any other number of arguments, print a usage message.

Here is an example of running the program several times:

```
> java Angle
Usage: Angle decimaldegrees|deg min sec
> java Angle 37.272
37° 16′ 19″
> java Angle -20 16
-20.267°
> java Angle 10 47 11
10.786°
> java Angle 66 10 47 11
Usage: Angle decimaldegrees|deg min sec
```

Hints:

- The degree symbol ° is `"\u00b0"`; the minutes symbol ′ is `"\u2032"`, and the seconds symbol ″ is `"\u2033"`.
- In order to handle negative angles correctly, you might find the `Math.signum` method useful. It takes a `double` as an argument and returns -1.0 if the argument is negative, 0.0 if the argument is zero, and 1.0 if the argument is positive.
- Generalize by writing a `toDMS` method that takes a decimal degree and returns an array of three integers. Write another method named `toDecimal` which takes three parameters (degrees, minutes, and seconds) and returns the decimal equivalent as a `double`.

Exercise 9.3 Write a command-line program that accepts two numbers and a string representing an arithmetic operation: `"plus"`, `"minus"`, `"times"`, and `"div"` in either upper or lower case and evaluates the resulting expression.

Examples of running the program (source in *Calc.java*) several times:

```
> java Calc
Usage: number plus|minus|times|div number
> java Calc 3 plus 4.5
7.5
> java Calc 7.5 div 2
3.75
> java Calc 6 MINUS 1.25
4.75
> java Calc 7 blah 5
Unknown operation blah
```

The reason this program uses words instead of the symbols `+`, `-`, `*`, and `/` is that some of these characters are interpreted by the MacOS, Windows, and Linux command lines as wild card characters or part of a file name.