Programming via Java Strings

Programs must often manipulate text, like words or sentences. Such text is composed of characters; a character might be are a letter, digit, punctuation mark, or space. Any particular sequence of characters is called a string.

Java includes the `String` class, each instance of which represents a string of characters. This class is one of the most heavily used in Java, and we study it in this chapter.

8.1. The `String` class

All Java classes require the use of **`new`** to create new instances of that class — with one exception. That exception is the `String` class, which is so useful that Java designers built special support for it into the language. To create a `String` object representing a sequence of characters, you need only to enclose that sequence inside a pair of double quotes.

```
String sentence = "Hello, world.";
```

In this code fragment, we create a variable named `sentence`, and we assign it to refer to the string `Hello, world.`

Sometimes you may want to include quotation marks inside a string. To do this, you can precede the quotation marks with a backslash.

```
String quote = "Frobozz said, \"Hello, sailor.\"";
```

If for some reason you wanted a backslash in a string, then you would precede it with a backslash.

There is another special exception with `String` objects, too: For other classes, you cannot apply operators like `*` or `>` to their objects; such operators can be applied only to primitive types like **`int`** and **`double`**. But you can apply the `+` operator using the `String` class; in this case, it combines values into a larger string. In the below example, we suppose that `name` is a `String` variable that has already been assigned to reference the user's name.

```
String greeting = "Hello, " + name + ".";
```

This fragment creates a new variable `greeting`. If `name` references the string `Dave`, `greeting` will be assigned to reference the string `Hello, Dave.` If `name` were `Dolly`,`greeting` would reference `Hello, Dolly.`

Strictly speaking, it isn't quite true that `String` is the only class that permits any operators to be applied to its objects: There are three operators that can be applied to objects of all classes: `==`, `!=`, and `=`. And later we'll see a special object type called an *array* that permits the bracket operator. But the `String` class is the only normal class that allows any *other* operators to be applied to its instances.

A program can add any type onto a `String` object, resulting in another `String` object. Suppose for example that `k` is an **`int`** variable holding the value 42.

```
String s = "k is " + k + ".";
```

The above fragment would assign `s` to be the string `k is 42.` That is, it places the two characters '`4`' and '`2`' into the string, since these two digits constitute the decimal expansion of the value referenced by `k`.

Sometimes this usage of `+` can lead to unusual behavior. What string do you suppose the following would compute?

```
String t = "k+1 is " + k+1 + ".";
```

We would hope that `t` would reference `k+1 is 43.` But in fact this fragment creates the string `k+1 is 421.` The Java compiler simply sees the usage of the `+` operator, and computes it left-to-right. So the computer first adds `k` to `k+1 is `, then it adds `1` onto that, and finally it adds a period. (The compiler doesn't pay attention to the spaces.)

8.2. String input and output

The `GraphicsProgram` class includes several methods that use the `String` class. Most accept a `String` as a parameter that is then displayed for the user to see.

`void` `print(String message)`

> Displays `message` to the user, leaving the cursor at the character following the last character of `message` (without advancing to the next line).

`void` `println(String message)`

> Displays `message` to the user and advances the cursor to the next line's beginning.

`double` `readDouble(String message)`

Displays `message` to the user, waits for the user to type a number, and then returns the number the user typed.

`int readInt(String message)`

Displays `message` to the user, waits for the user to type a integer, and then returns the integer the user typed.

`String readLine(String message)`

Displays `message` to the user, waits for the user to type a line of text, and then returns a `String` holding the characters typed by the user (omitting the final end-of-line character).

Note that `GraphicsWindow` also has `readInt` and `readDouble` methods that take no parameters, in addition to the above. Java allows multiple methods with the same name: When a program invokes a method for which multiple methods are named identically, the compiler will count the parameters and identify the parameter types, and use that to determine the appropriate method. (The compiler will not allow a class to define two methods with the same number of parameters, and with the corresponding parameter types all being identical.)

Figure 8.1 contains an example of a program using these methods. It is an enhanced version of the `MovingBall` program of Figure 6.3. In this version, the user first enters the ball's initial velocity. This version also informs the user once the ball has exited the screen.

Figure 8.1: The `CustomMovingBall` program.

```java
import acm.program.*;
import acm.graphics.*;
import java.awt.*;

public class CustomMovingBall extends GraphicsProgram {
    public void run() {
        double xVelocity = readDouble("Horizontal velocity? ");
        double yVelocity = readDouble("Vertical velocity? ");

        GOval ball = new GOval(25, 25, 50, 50);
        ball.setFilled(true);
        ball.setFillColor(new Color(255, 0, 0));
        add(ball);
14
15          println("Ball starts rolling.");
16          int frames = 0;
17          while(ball.getX() < getWidth() && ball.getX() > -ball.getWidth()
18                  && ball.getY() < getHeight() && ball.getY() > -
ball.getHeight()) {
19              pause(40);
20              frames++;
21              ball.move(xVelocity, yVelocity);
22          }
23          println("Ball exited window after " + frames + " frames.");
24      }
25  }
```

Notice how line 23 builds the string to be printed by adding an **`int`** variable's value onto the end of a string.

When executed, the program displays the following; the user's input is displayed in boldface. This dialogue will take place in a completely separate place from where the graphics are displayed. Exactly where the dialogue takes place depends on your computer's configuration, but the best bet is the window from where you started the program.

```
Horizontal velocity? 8
Vertical velocity? 6
Ball starts rolling.     output pauses while ball rolls across screen
Ball exited screen after 78 frames.
```

8.3. `String` methods

As a class defined in the Java libraries, the `String` class defines several methods that an individual `String` object can perform.

`int` `length()`

Returns the number of characters in this string.

`String substring(`**`int`**` begin)`

Returns a string containing the characters of this string beginning at index `begin` and going to this string's end.

`String substring(int begin, int end)`

> Returns a string containing the characters of this string beginning at index `begin` and going up to — but not including — index `end`.

It's important to understand that the index of the first character of the string is 0, the second character's index is 1, and so on. The final character of a string `str` is at index `str.length() - 1`; the index is not at `str.length()` because the counting starts from 0.

Suppose we want to print a particular string in reverse; for example, if the user types straw, we want the program to print warts. The following code segment accomplishes this.

```
String str = readLine("Type a string to reverse: ");
int toPrint = str.length();
while(toPrint > 0) {
    print(str.substring(toPrint - 1, toPrint));
    toPrint--;
}
```

This fragment uses a counter `toPrint` to track how many characters are left to print. As long as there are characters are left (as the **`while`** loop indicates), the program will display the substring consisting of the `toPrint`th character, and then decrement `toPrint`. Notice that, in invoking `substring` method, the program passes `toPrint - 1`as the beginning point; we subtract 1 because the characters are numbered starting at 0. For the ending point, we pass `toPrint`, since this is the first index we do *not*want.

The `substring` method's behavior of omitting the last character in the range named is not what you would expect, and beginners often have trouble with this. The designers chose this because it tends to simplify the underlying program. For example, the number of characters fetched by the `substring` method is`end - begin`.

8.4. Equality testing

Suppose we wanted to write a program to repeatedly ask the user for a password until the user types the correct password. Supposing the password is *friend*, you might be tempted to retrieve the password using the following program fragment.

```
String password = readLine("Password? ");
while(password != "friend") {   // Wrong!
```

```
    password = readLine("Wrong. Password? ");
}
println("You're in.");
```

This reads a password from the user. As long as the word the user types isn't *friend*, the program continues asking for another word.

The idea's good, and the program will compile and run, but when we test it, it won't work. It will keep saying that we were wrong, even when we type friend at the prompt.

The problem is that the test in the `while` loop to see if two objects are equal actually tests to see if the two objects are the same. As it turns out, `password` and the string created by enclosing friend in double-quotations marks, are two different strings, located at different points in the computer's memory, even though they happen to contain the same letters. They're identical, but they're not the same. It's the same principle that leads me to assert that two identical blue M&M candies laid side by side are nonetheless different: Even if they are indistinguishible, they're not the same piece of candy.

Thus, the `while` loop's condition will always turn out to be `true`: `password` will never equal `"friend"`, even if `password` contains the same letters.

So how can we compare two strings to see if they contain the same letters? Luckily, Java's `String` class includes some methods to help with this dilemma.

`boolean equals(String other)`

> Return `true` if this string contains the same sequence of characters as `other` does. For the purpose of equality testing, lower-case letters are treated as different from their corresponding capital letters. Thus if `s` is `macintosh`, the invocation `s.equals("MacIntosh")` returns `false`.

`boolean equalsIgnoreCase(String other)`

> Return `true` if this string contains the same sequence of characters as `other` does, treating lower-case letters as identical to their corresponding capital letters.

We can repair our program fragment by using the `equals` method to compare the two strings.

```
while(!password.equals("friend")) {    // ok
```

The `equals` method will look through the characters of `password` and see if they all match up with the corresponding characters of *`"friend"`*. If they do, it returns **`true`**, and this **`while`** condition (with its exclamation point for representing *not*) will have a value of **`false`**, so that the computer will proceed to the first statement following the **`while`** loop. If they don't match, the `equals` method returns **`false`**, so that the value of the condition is **`true`**, and so the computer will go through another iteration of the loop, asking the user to try again.

Source: http://www.toves.org/books/java/ch08-string/index.html