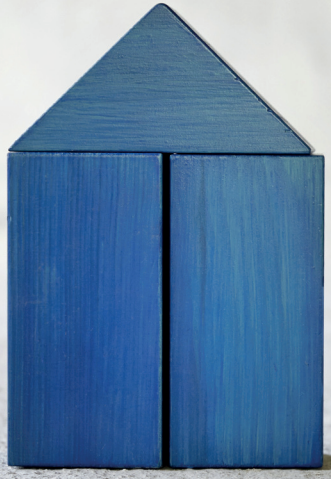
GLOBAL EDITION

Building Java™ Programs

A Back to Basics Approach

FOURTH EDITION

Stuart Reges • Marty Stepp



Pearson

Fourth Edition
Global Edition

Building Java Programs

A Back to Basics Approach

Stuart Reges
University of Washington

Marty Stepp
Stanford University

Pearson

Boston Columbus Indianapolis New York San Francisco Hoboken
Amsterdam Cape Town Dubai London Madrid Milan Munich Paris Montreal Toronto
Delhi Mexico City Sao Paulo Sydney Hong Kong Seoul Singapore Taipei Tokyo

This solution gives us a second version of the method:

```
public static String firstWord(String s) {
    int start = 0;
    int stop = 0;
    while (stop < s.length() && s.charAt(stop) != ' ') {
        stop++;
    }
    return s.substring(start, stop);
}
```

But remember that we assumed that the first word starts at position 0. That won't necessarily be the case. For example, if we pass a string that begins with several spaces, this method will return an empty string. We need to modify the code so that it skips any leading spaces. Accomplishing that goal requires another loop. As a first approximation, we can write the following code:

```
int start = 0;
while (s.charAt(start) == ' ') {
    start++;
}
```

This code works for most strings, but it fails in two important cases. The loop test assumes we will find a nonspace character. What if the string is composed entirely of spaces? In that case, we'll simply run off the end of the string, generating a `StringIndexOutOfBoundsException`. And what if the string is empty to begin with? We'll get an error immediately when we ask about `s.charAt(0)`, because there is no character at index `0`.

We could decide that these cases constitute errors. After all, how can you return the first word if there is no word? So, we could document a precondition that the string contains at least one nonspace character, and throw an exception if we find that it doesn't. Another approach is to return an empty string in these cases.

To deal with the possibility of the string being empty, we need to modify our loop to incorporate a test on the length of the string. If we add it at the end of our `while` loop test, we get the following code:

```
int start = 0;
while (s.charAt(start) == ' ' && start < s.length()) {
    start++;
}
```

But this code has the same flaw we saw before. It is supposed to prevent problems when `start` becomes equal to the length of the string, but when this situation occurs, a `StringIndexOutOfBoundsException` will be thrown before the computer reaches

the test on the length of the string. So these tests also have to be reversed to take advantage of short-circuited evaluation:

```
int start = 0;
while (start < s.length() && s.charAt(start) == ' ') {
    start++;
}
```

To combine these lines of code with our previous code, we have to change the initialization of `stop`. We no longer want to search from the front of the string. Instead, we need to initialize `stop` to be equal to `start`. Putting these pieces together, we get the following version of the method:

```
public static String firstWord(String s) {
    int start = 0;
    while (start < s.length() && s.charAt(start) == ' ') {
        start++;
    }
    int stop = start;
    while (stop < s.length() && s.charAt(stop) != ' ') {
        stop++;
    }
    return s.substring(start, stop);
}
```

This version works in all cases, skipping any leading spaces and returning an empty string if there is no word to return.

`boolean` Variables and Flags

All `if/else` statements are controlled by Boolean tests. The tests can be `boolean` variables or Boolean expressions. Consider, for example, the following code:

```
if (number > 0) {
    System.out.println("positive");
} else {
    System.out.println("not positive");
}
```

This code could be rewritten as follows:

```
boolean positive = (number > 0);
if (positive) {
    System.out.println("positive");
} else {
    System.out.println("not positive");
}
```

Using boolean variables adds to the readability of your programs because it allows you to give names to tests. Consider the kind of code you would generate for a dating program. You might have some integer variables that describe certain attributes of a person: looks, to store a rough estimate of physical beauty (on a scale of 1–10); IQ, to store intelligence quotient; income, to store gross annual income; and snothers, to track intimate friends ("snother" is short for "significant other"). Given these variables to specify a person's attributes, you can develop various tests of suitability. As you are writing the program, you can use boolean variables to give names to those tests, adding greatly to the readability of the code:

```
boolean cute = (looks >= 9);
boolean smart = (IQ > 125);
boolean rich = (income > 100000);
boolean available = (snothers == 0);
boolean awesome = cute && smart && rich && available;
```

You might find occasion to use a special kind of boolean variable called *a flag*. Typically we use flags within loops to record error conditions or to signal completion. Different flags test different conditions. As an analogy, consider a referee at a sports game who watches for a particular illegal action and throws a flag if it happens. You sometimes hear an announcer saying, "There is a flag down on the play."

Let's introduce a flag into the cumulative sum code we saw in the previous chapter:

```
double sum = 0.0;
for (int i = 1; i <= totalNumber; i++) {
    System.out.print("    #" + i + "? ");
    double next = console.nextDouble();
    sum += next;
}
System.out.println("sum = " + sum);
```

Suppose we want to know whether the sum ever goes negative at any point. Notice that this situation isn't the same as the situation in which the sum ends up being negative. Like a bank account balance, the sum might switch back and forth between positive and negative. As you make a series of deposits and withdrawals, the bank will keep track of whether you overdraw your account along the way. Using a boolean flag, we can modify the preceding loop to keep track of whether the sum ever goes negative and report the result after the loop:

```
double sum = 0.0;
boolean negative = false;
for (int i = 1; i <= totalNumber; i++) {
    System.out.print("    #" + i + "? ");
    double next = console.nextDouble();
```

```
    sum += next;
    if (sum < 0.0) {
        negative = true;
    }
}
System.out.println("sum = " + sum);
if (negative) {
    System.out.println("Sum went negative");
} else {
    System.out.println("Sum never went negative");
}
```

Boolean Zen

In 1974, Robert Pirsig started a cultural trend with his book *Zen and the Art of Motorcycle Maintenance: An Inquiry into Values.* A slew of later books copied the title with *Zen and the Art of X,* where *X* was Poker, Knitting, Writing, Foosball, Guitar, Public School Teaching, Making a Living, Falling in Love, Quilting, Stand-up Comedy, the SAT, Flower Arrangement, Fly Tying, Systems Analysis, Fatherhood, Screenwriting, Diabetes Maintenance, Intimacy, Helping, Street Fighting, Murder, and on and on. There was even a book called *Zen and the Art of Anything.*

We now join this cultural trend by discussing Zen and the art of type `boolean`. It seems to take a while for many novices to get used to Boolean expressions. Novices often write overly complex expressions involving `boolean` values because they don't grasp the simplicity that is possible when you "get" how the `boolean` type works.

For example, suppose that you are writing a game-playing program that involves two-digit numbers, each of which is composed of two different digits. In other words, the program will use numbers like 42 that are composed of two distinct digits, but not numbers like 6 (only one digit), 394 (more than two digits), or 22 (both digits are the same). You might find yourself wanting to test whether a given number is legal for use in the game. You can restrict yourself to two-digit numbers with a test like the following one:

```
n >= 10 && n <= 99
```

You also have to test to make sure that the two digits aren't the same. You can get the digits of a two-digit number with the expressions `n / 10` and `n % 10`. So you can expand the test to ensure that the digits aren't the same:

```
n >= 10 && n <= 99 && (n / 10 != n % 10)
```

This test is a good example of a situation in which you could use a method to capture a complex Boolean expression. Returning a `boolean` will allow you to call the method as many times as you want without having to copy this complex expression each time, and you can give a name to this computation to make the program more readable.

Suppose you want to call the method `isTwoUniqueDigits`. You want the method to take a value of type `int` and return `true` if the `int` is composed of two unique digits and `false` if it is not. So, the method would look like the following:

```
public static boolean isTwoUniqueDigits(int n) {
    ...
}
```

How would you write the body of this method? We've already written the test, so we just have to figure out how to incorporate it into the method. The method has a `boolean` return type, so you want it to return the value `true` when the test succeeds and the value `false` when it fails. You can write the method as follows:

```
public static boolean isTwoUniqueDigits(int n) {
    if (n >= 10 && n <= 99 && (n % 10 != n / 10)) {
        return true;
    } else {
        return false;
    }
}
```

This method works, but it is more verbose than it needs to be. The preceding code evaluates the test that we developed. That expression is of type `boolean`, which means that it evaluates to either `true` or `false`. The `if/else` statement tells the computer to return `true` if the expression evaluates to `true` and to return `false` if it evaluates to `false`. But why use this construct? If the method is going to return `true` when the expression evaluates to `true` and return `false` when it evaluates to `false`, you can just return the value of the expression directly:

```
public static boolean isTwoUniqueDigits(int n) {
    return (n >= 10 && n <= 99 && (n % 10 != n / 10));
}
```

Even the preceding version can be simplified, because the parentheses are not necessary (although they make it clearer exactly what the method will return). This code evaluates the test that we developed to determine whether a number is composed of two unique digits and returns the result (`true` when it does, `false` when it does not).

Consider an analogy to integer expressions. To someone who understands Boolean Zen, the `if/else` version of this method looks as odd as the following code:

```
if (x == 1) {
    return 1;
} else if (x == 2) {
    return 2;
} else if (x == 3) {
    return 3;
```

```
} else if (x == 4) {
    return 4;
} else if (x == 5) {
    return 5;
}
```

If you always want to return the value of `x`, you should just say:

```
return x;
```

A similar confusion can occur when students use `boolean` variables. In the last section we looked at a variation of the cumulative sum algorithm that used a `boolean` variable called `negative` to keep track of whether or not the sum ever goes negative. We then used an `if/else` statement to print a message reporting the result:

```
if (negative) {
    System.out.println("Sum went negative");
} else {
    System.out.println("Sum never went negative");
}
```

Some novices would write this code as follows:

```
if (negative == true) {
    System.out.println("Sum went negative");
} else {
    System.out.println("Sum never went negative");
}
```

The comparison is unnecessary because the `if/else` statement expects an expression of type `boolean` to appear inside the parentheses. A `boolean` variable is already of the appropriate type, so we don't need to test whether it equals `true`; it either *is* `true` or it isn't (in which case it is `false`). To someone who understands Boolean Zen, the preceding test seems as redundant as saying:

```
if ((negative == true) == true) {
    ...
}
```

Novices also often write tests like the following:

```
if (negative == false) {
    ...
}
```