# STRING METHODS AND EQUALITY TEST IN JAVA

## 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