

To test for equality, use the `equals(Object)` method inherited by every class and defined by `String` to return **true** if and only if the object passed in is a `String` containing the exact same data.

```
String greeting = "Hello World!";
if(greeting.equals("Hello World!")) { //true
    // ...
}

if(greeting.equals("hello world!")) { //false
    // ...
}
```

To order `String` objects, use the **`compareTo()`** method, which can be accessed wherever we use a `String` datatype. Let's take a look at an example:

```
String myString = "Hello World!";
...
if(myString.compareTo("Hello World!") == 0 )
{
    System.out.println("Match Found");
}
```

This snippet of code is comparing the `String` variable *myString* to "Hello World". The `compareTo` method returns a negative, zero, or positive number if the parameter is less than, equal to, or greater than the object on which it is called. If *myString* was to be different, even in the slightest manner we will get a value above or below 0 depending on the exact difference. The result is negative if this `String` object lexicographically precedes the argument string. The result is a positive integer if this `String` object lexicographically follows the argument string. Take a look at the Java API ^[26] for more details.

Splitting a String

Sometimes it is useful to split a string into separate strings, based on a *regular expression*. (For more information on regular expressions, see Regex.) The `String` class has a **`split()`** method, since Java 1.4, that will return a `String` array.

See the following example:

```
String person = "Brown, John:100 Yonge Street, Toronto:(416) 777-9999";
...
String [] personData = person.split( ":" );
...
String name      = personData[0];
String address   = personData[1];
String phone     = personData[2];
```

An other usefull application could be to 'split' the `String` text based on the 'new line' character, so you could process the text line by line.

Creating substrings

It may also be sometimes useful to create **substrings**, or strings using the order of letters from an existing string. This can be done in two methods.

The first method involves creating a substring out of the characters of a string from a given index to the end.

For example:

```
String str    = "coffee";  
String substr = str.substring(3);
```

In this example, `substr` would return `"fee"`. As previously discussed, the index of the first character in a string is 0. By counting from there, it is apparent that the character in index 3 is the second "f" in "coffee". This is known as the `beginIndex`. All characters from the `beginIndex` until the end of the string will be copied into the new substring.

The second method involves a user-defined `beginIndex` and `endIndex`. For example:

```
String str = "supporting";  
String substr = str.substring(3, 7);
```

The string returned by `substr` would be `"port"`. Please note that the `endIndex` is **not** inclusive. This means that the last character will be of the index `endIndex-1`. Therefore, in this example, every character from index 3 to index 6, inclusive, was copied into the substring.

Note: "Substring" is considered to be one word. This is why the method name does not seem to follow the common syntax of Java. It is easy to mistake the method `substr()` for `subStr()` (which does not exist and would return with a syntax error on compilation). Just remember that this style only applies to methods or other elements that are made up of more than one word.

Modifying String cases

The String Class also allows for the modification of cases. The two methods that make this possible are `toLowerCase()` and `toUpperCase()`. These methods are useful, for example, in the typical programming classroom assignment of evaluating whether or not a string is a palindrome.

```
String a = "WIKIBOOKS";  
String b = "wikipedia";
```

In this example, a call to `a.toLowerCase()` would return a result of `"wikibooks"`, and `b.toUpperCase()` would return `"WIKIPEDIA"`.

See also

- Java API: `java.lang.String` ^[27]
- Java API: `java.lang.StringBuffer` ^[28]
- Java API: `java.lang.StringBuilder` ^[29]
- Java Programming/API/`java.lang.StringBuffer`
- Java Programming/API/`java.lang.StringBuilder`

Arrays

Navigate **Language Fundamentals** topic:

Intro to Arrays

An **array** is similar to a table of data, keyed by number. In Java an array is an object like all other objects. Look at the following program:

```
import java.util.*;

public class ArrayExample {
    static Scanner input = new Scanner(System.in);

    public static void main(String[] args) {
        int numNames = getInt("Number of names?");
        String[] names = new String[numNames];
        for (int i = 0; i < names.length; i++) {
            names[i] = getString("Enter name #" + i);
        }
        for (int i = 0; i < names.length; ++i) {
            System.out.println(names[i]);
        }
    }

    public static int getInt(String prompt) {
        System.out.print(prompt + " ");
        int integer = input.nextInt();
        input.nextLine(); // Get rid of this line
                          // so that getString won't read it
        return integer;
    }

    public static String getString(String prompt) {
        System.out.print(prompt + " ");
        return input.nextLine();
    }
}
```

Copy the code and compile it. The program will ask you to enter some names then reprints the names in order. It demonstrates three major aspects of arrays: how to define an array, how to set data, and how to access it. The code `String[] names = new String[numNames];` tells Java to create an array of size *numNames* that will store Strings. To set data, use `names[x] = data` where *x* is the index to access. Note that all Java arrays start at 0 and go to (array size - 1). Thus, if you dimension an array to size 10, the highest index is 9.