Strings

Strings, which are widely used in Java programming, are a sequence of characters. In the Java programming language, strings are objects.

The Java platform provides the `String` class to create and manipulate strings.

Creating Strings

The most direct way to create a string is to write:

```
String greeting = "Hello world!";
```

In this case, "Hello world!" is a *string literal*—a series of characters in your code that is enclosed in double quotes. Whenever it encounters a string literal in your code, the compiler creates a `String` object with its value—in this case, `Hello world!`.

String Length

Methods used to obtain information about an object are known as *accessor methods*. One accessor method that you can use with strings is the `length()` method, which returns the number of characters contained in the string object. After the following two lines of code have been executed, `len` equals 17:

```
String palindrome = "Dot saw I was Tod";
int len = palindrome.length();
```

A *palindrome* is a word or sentence that is symmetric—it is spelled the same forward and backward, ignoring case and punctuation. Here is a short and inefficient program to reverse a palindrome string. It invokes the `String` method `charAt(i)`, which returns the i^{th} character in the string, counting from 0.

```
public class StringDemo {
    public static void main(String[] args) {
        String palindrome = "Dot saw I was Tod";
        int len = palindrome.length();
        char[] tempCharArray = new char[len];
        char[] charArray = new char[len];

        // put original string in an
        // array of chars
        for (int i = 0; i < len; i++) {
            tempCharArray[i] =
                palindrome.charAt(i);
        }

        // reverse array of chars
        for (int j = 0; j < len; j++) {
            charArray[j] =
                tempCharArray[len - 1 - j];
        }

        String reversePalindrome =
            new String(charArray);
        System.out.println(reversePalindrome);
    }
}
```

Running the program produces this output:

```
doT saw I was toD
```

To accomplish the string reversal, the program had to convert the string to an array of characters (first `for` loop), reverse the array into a second array (second `for` loop), and then convert back to a string. The `String` class includes a method, `getChars()`, to convert a string, or a portion of a string, into an array of characters so we could replace the first `for` loop in the program above with

```
palindrome.getChars(0, len, tempCharArray, 0);
```

Concatenating Strings

The `String` class includes a method for concatenating two strings:

```
string1.concat(string2);
```

This returns a new string that is string1 with string2 added to it at the end.

You can also use the `concat()` method with string literals, as in:

```
"My name is ".concat("Rumplestiltskin");
```

Strings are more commonly concatenated with the **+** operator, as in

```
"Hello," + " world" + "!"
```

which results in

```
"Hello, world!"
```

The **+** operator is widely used in `print` statements. For example:

```
String string1 = "saw I was ";
System.out.println("Dot " + string1 + "Tod");
```

which prints

```
Dot saw I was Tod
```

Such a concatenation can be a mixture of any objects. For each object that is not a `String`, its `toString()` method is called to convert it to a `String`.

Note: The Java programming language does not permit literal strings to span lines in source files, so you must use the `+` concatenation operator at the end of each line in a multi-line string. For example:

```
String quote =
    "Now is the time for all good " +
    "men to come to the aid of their country.";
```

Breaking strings between lines using the `+` concatenation operator is, once again, very common in `print` statements.

Manipulating Characters in a String

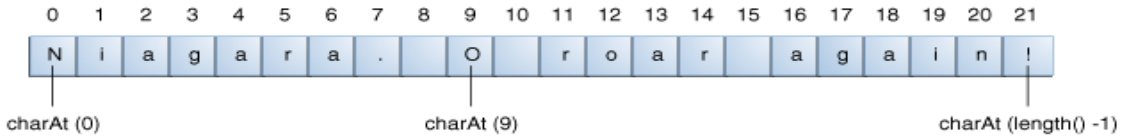
The `String` class has a number of methods for examining the contents of strings, finding characters or substrings within a string, changing case, and other tasks.

Getting Characters and Substrings by Index

You can get the character at a particular index within a string by invoking the `charAt()` accessor method. The index of the first character is 0, while the index of the last character is `length()-1`. For example, the following code gets the character at index 9 in a string:

```
String anotherPalindrome = "Niagara. O roar again!";
char aChar = anotherPalindrome.charAt(9);
```

Indices begin at 0, so the character at index 9 is 'O', as illustrated in the following figure:



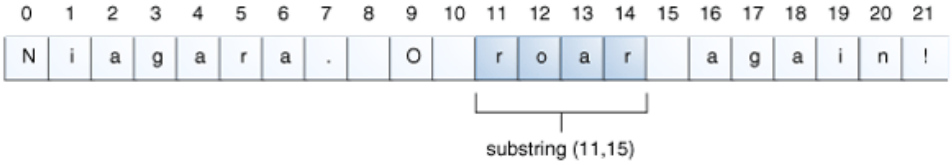
If you want to get more than one consecutive character from a string, you can use the `substring` method.
The `substring` method has two versions, as shown in the following table:

The `substring` Methods in the `String` Class

Method	Description
`String substring(int beginIndex, int endIndex)`	Returns a new string that is a substring of this string. The first integer argument specifies the index of the first character. The second integer argument is the index of the last character - 1.
`String substring(int beginIndex)`	Returns a new string that is a substring of this string. The integer argument specifies the index of the first character. Here, the returned substring extends to the end of the original string.

The following code gets from the Niagara palindrome the substring that extends from index 11 up to, but not including, index 15, which is the word "roar":

```
String anotherPalindrome = "Niagara. O roar again!";
String roar = anotherPalindrome.substring(11, 15);
```



Other Methods for Manipulating Strings

Here are several other `String` methods for manipulating strings:

Other Methods in the `String` Class for Manipulating Strings	
Method	**Description**
`String toLowerCase()` `String toUpperCase()`	Returns a copy of this string converted to lowercase or uppercase. If no conversions are necessary, these methods return the original string.

Searching for Characters and Substrings in a String

Here are some other `String` methods for finding characters or substrings within a string. The `String` class provides accessor methods that return the position within the string of a specific character or substring:`indexOf()` and `lastIndexOf()`. The `indexOf()` methods search forward from the beginning of the string, and the `lastIndexOf()` methods search backward from the end of the string. If a character or substring is not found, `indexOf()` and `lastIndexOf()` return -1.

The `String` class also provides a search method, `contains`, that returns true if the string contains a particular character sequence. Use this method when you only need to know that the string contains a character sequence, but the precise location isn't important.

The following table describes the various string search methods.

The Search Methods in the `String` Class	
Method	**Description**
`int indexOf(int ch)` `int lastIndexOf(int ch)`	Returns the index of the first (last) occurrence of the specified character.
`int indexOf(int ch, int fromIndex)` `int lastIndexOf(int ch, int fromIndex)`	Returns the index of the first (last) occurrence of the specified character, searching forward (backward) from the specified index.
`int indexOf(String str)` `int lastIndexOf(String str)`	Returns the index of the first (last) occurrence of the specified substring.
`int indexOf(String str, int fromIndex)` `int lastIndexOf(String str, int fromIndex)`	Returns the index of the first (last) occurrence of the specified substring, searching forward (backward) from the specified index.
`boolean contains(CharSequence s)`	Returns true if the string contains the specified character sequence.

Replacing Characters and Substrings into a String

The `String` class has very few methods for inserting characters or substrings into a string. In general, they are not needed: You can create a new string by concatenation of substrings you have *removed* from a string with the substring that you want to insert.

The `String` class does have four methods for *replacing* found characters or substrings, however. They are:

Methods in the `String` Class for Manipulating Strings

Method	Description
`String replace(char oldChar, char newChar)`	Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.
`String replace(CharSequence target, CharSequence replacement)`	Replaces each substring of this string that matches the literal target sequence with the specified literal replacement sequence.
`String replaceAll(String regex, String replacement)`	Replaces each substring of this string that matches the given regular expression with the given replacement.
`String replaceFirst(String regex, String replacement)`	Replaces the first substring of this string that matches the given regular expression with the given replacement.

An Example

The following class, `Filename`, illustrates the use of `lastIndexOf()` and `substring()` to isolate different parts of a file name.

Note: The methods in the following `Filename` class don't do any error checking and assume that their argument contains a full directory path and a filename with an extension. If these methods were production code, they would verify that their arguments were properly constructed.

```
public class Filename {
    private String fullPath;
    private char pathSeparator,
                 extensionSeparator;

    public Filename(String str, char sep, char ext) {
        fullPath = str;
        pathSeparator = sep;
        extensionSeparator = ext;
    }

    public String extension() {
        int dot = fullPath.lastIndexOf(extensionSeparator);
        return fullPath.substring(dot + 1);
    }

    // gets filename without extension
    public String filename() {
        int dot = fullPath.lastIndexOf(extensionSeparator);
        int sep = fullPath.lastIndexOf(pathSeparator);
        return fullPath.substring(sep + 1, dot);
    }

    public String path() {
        int sep = fullPath.lastIndexOf(pathSeparator);
        return fullPath.substring(0, sep);
    }
}
```

Here is a program, `FilenameDemo`, that constructs a `Filename` object and calls all of its methods:

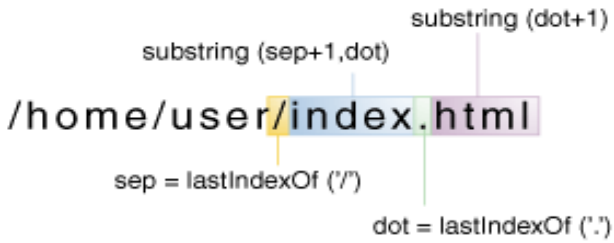
```
public class FilenameDemo {
    public static void main(String[] args) {
        final String FPATH = "/home/user/index.html";
        Filename myHomePage = new Filename(FPATH, '/', '.');
        System.out.println("Extension = " + myHomePage.extension());
        System.out.println("Filename = " + myHomePage.filename());
        System.out.println("Path = " + myHomePage.path());
    }
```

```
}
```

And here's the output from the program:

```
Extension = html
Filename = index
Path = /home/user
```

As shown in the following figure, our `extension` method uses `lastIndexOf` to locate the last occurrence of the period (.) in the file name. Then `substring` uses the return value of `lastIndexOf` to extract the file name extension — that is, the substring from the period to the end of the string. This code assumes that the file name has a period in it; if the file name does not have a period, `lastIndexOf` returns -1, and the substring method throws a `StringIndexOutOfBoundsException`.



Also, notice that the `extension` method uses `dot + 1` as the argument to `substring`. If the period character (.) is the last character of the string, `dot + 1` is equal to the length of the string, which is one larger than the largest index into the string (because indices start at 0). This is a legal argument to `substring` because that method accepts an index equal to, but not greater than, the length of the string and interprets it to mean "the end of the string."

Comparing Strings

`boolean equals(Object anObject)`	Returns `true` if and only if the argument is a `String` object that represents the same sequence of characters as this object.
`boolean equalsIgnoreCase(String anotherString)`	Returns `true` if and only if the argument is a `String` object that represents the same sequence of characters as this object, ignoring differences in case.