The *Java String* data type can contain a sequence (string) of characters, like pearls on a string. Strings are how you work with text in Java.

Strings in Java are represented internally using bytes, encoded as UTF-16. UTF-16 uses 2 bytes to represent a single character. UTF is a character encoding that can represent characters from a lot of different languages (alphabets).

**Creating a String**

Strings in Java are objects. Therefore you need to use the new operator to create a new Java String object. Here is a Java String instantiation (creation) example:

String myString = new String("Hello World");

The text inside the quotes is the text the String object will contain.

**Java String Literals**

Java has a shorter way of creating a new String:

String myString = "Hello World";

Instead of passing the text "Hello World" as a parameter to the String constructor, you can just write the text itself inside the double quote characters. This is called a String literal. The Java compiler will internally figure out how to create a new Java String representing the given text.

**Escape Characters**

Java Strings literals accepts a set of *escape characters* which are translated into special characters in the created String. These escape characters are:

|  |  |
| --- | --- |
| **Esc. Char** | **Description** |
| \\ | Translated into a single \ character in the String |
| \t | Translated into a single tab character in the string |
| \r | Translated into a single carriage return character in the string |
| \n | Translated into a single new line character in the string |

Here is an example of creating a Java String using escape characters:

String text = "\tThis text is one tab in.\r\n";

This String literal will result in a String that starts with a tab character and ends with a carriage return and a new line character.

**String Literals as Constants or Singletons**

If you use the same string (e.g. "Hello World") in other String variable declarations, the Java virtual machine may only create a single String instance in memory. The string literal thus becomes a de facto constant or singleton. The various different variables initialized to the same constant string will point to the same String instance in memory. Here is a Java String constant / singleton example:

String myString1 = "Hello World";

String myString2 = "Hello World";

In this case the Java virtual machine will make both myString1 and myString2 point to the same String object.

More precisely, objects representing Java String literals are obtained from a constant String pool which the Java virtual machine keeps internally. That means, that even classes from different projects compiled separately, but which are used in the same application may share constant String objects. The sharing happens at runtime. It is not a compile time feature.

If you want to be sure that two String variables point to separate String objects, use the new operator like this:

String myString1 = new String("Hello World");

String myString2 = new String("Hello World");

Even though the value (text) of the two Java Strings created is the same, the Java virtual machine will create two different objects in memory to represent them.

**Concatenating Strings**

Concatenating Strings means appending one string to another. Strings in Java are immutable meaning they cannot be changed once created. Therefore, when concatenating two Java String objects to each other, the result is actually put into a third String object.

Here is a Java String concatenation example:

String one = "Hello";

String two = "World";

String three = one + " " + two;

The content of the String referenced by the variable three will be Hello World; The two other Strings objects are untouched.

**String Concatenation Performance**

When concatenating Strings you have to watch out for possible performance problems. Concatenating two Strings in Java will be translated by the Java compiler to something like this:

String one = "Hello";

String two = " World";

String three = new StringBuilder(one).append(two).toString();

As you can see, a new StringBuilder is created, passing along the first String to its constructor, and the second String to its append() method, before finally calling the toString() method. This code actually creates two objects: A StringBuilder instance and a new String instance returned from the toString()method.

When executed by itself as a single statement, this extra object creation overhead is insignificant. When executed inside a loop, however, it is a different story.

Here is a loop containing the above type of String concatenation:

String[] strings = new String[]{"one", "two", "three", "four", "five" };

String result = null;

for(String string : strings) {

result = result + string;

}

This code will be compiled into something similar to this:

String[] strings = new String[]{"one", "two", "three", "four", "five" };

String result = null;

for(String string : strings) {

result = new StringBuilder(result).append(string).toString();

}

Now, for every iteration in this loop a new StringBuilder is created. Additionally, a String object is created by the toString() method. This results in a small object instantiation overhead per iteration: One StringBuilder object and one String object. This by itself is not the real performance killer though. But something else related to the creation of these objects is.

Every time the new StringBuilder(result) code is executed, the StringBuilder constructor copies all characters from the result String into the StringBuilder. The more iterations the loop has, the bigger the result String grows. The bigger the result String grows, the longer it takes to copy the characters from it into a new StringBuilder, and again copy the characters from the StringBuilder into the temporary String created by the toString() method. In other words, the more iterations the slower each iteration becomes.

The fastest way of concatenating Strings is to create a StringBuilder once, and reuse the same instance inside the loop. Here is how that looks:

String[] strings = new String[]{"one", "two", "three", "four", "five" };

StringBuilder temp = new StringBuilder();

for(String string : strings) {

temp.append(string);

}

String result = temp.toString();

This code avoids both the StringBuilder and String object instantiations inside the loop, and therefore also avoids the two times copying of the characters, first into the StringBuilder and then into a String again.

**String Length**

You can obtain the length of a String using the length() method. The length of a String is the number of characters the String contains - not the number of bytes used to represent the String. Here is an example:

String string = "Hello World";

int length = string.length();

**Substrings**

You can extract a part of a String. This is called a substring. You do so using the substring() method of the String class. Here is an example:

String string1 = "Hello World";

String substring = string1.substring(0,5);

After this code is executed the substring variable will contain the string Hello.

The substring() method takes two parameters. The first is the character index of the first character to be included in the substring. The second is the index of the character *after* the last character to be included in the substring. Remember that. The parameters mean "from - including, to - *excluding*". This can be a little confusing until you memorize it.

The first character in a String has index 0, the second character has index 1 etc. The last character in the string has has the index String.length() - 1.

**Searching in Strings With indexOf()**

You can search for substrings in Strings using the indexOf() method. Here is an example:

String string1 = "Hello World";

int index = string1.indexOf("World");

The index variable will contain the value 6 after this code is executed. The indexOf() method returns the index of where the first character in the first matching substring is found. In this case the W of the matched substring World was found at index 6.

If the substring is not found within the string, the indexOf() method returns -1;

There is a version of the indexOf() method that takes an index from which the search is to start. That way you can search through a string to find more than one occurrence of a substring. Here is an example:

String theString = "is this good or is this bad?";

String substring = "is";

int index = theString.indexOf(substring);

while(index != -1) {

System.out.println(index);

index = theString.indexOf(substring, index + 1);

}

This code searches through the string "is this good or is this bad?" for occurrences of the substring "is". It does so using the indexOf(substring, index) method. The index parameter tells what character index in the String to start the search from. In this example the search is to start 1 character after the index where the previous occurrence was found. This makes sure that you do not just keep finding the same occurrence.

The output printed from this code would be:

0

5

16

21

The substring "is" is found in four places. Two times in the words "is", and two times inside the word "this".

The Java String class also has a lastIndexOf() method which finds the last occurrence of a substring. Here is an example:

String theString = "is this good or is this bad?";

String substring = "is";

int index = theString.lastIndexOf(substring);

System.out.println(index);

The output printed from this code would be 21 which is the index of the last occurrence of the substring "is".

**Matching a String Against a Regular Expression With matches()**

The Java String matches() method takes a regular expression as parameter, and returns true if the regular expression matches the string, and false if not.

Here is a matches() example:

String text = "one two three two one";

boolean matches = text.matches(".\*two.\*");

**Comparing Strings**

Java Strings also have a set of methods used to compare Strings. These methods are:

* equals()
* equalsIgnoreCase()
* startsWith()
* endsWith()
* compareTo()

**equals()**

The equals() method tests if two Strings are exactly equal to each other. If they are, the equals() method returns true. If not, it returns false. Here is an example:

String one = "abc";

String two = "def";

String three = "abc";

String four = "ABC";

System.out.println( one.equals(two) );

System.out.println( one.equals(three) );

System.out.println( one.equals(four) );

The two strings one and three are equal, but one is not equal to two or to four. The case of the characters must match exactly too, so lowercase characters are not equal to uppercase characters.

The output printed from the code above would be:

false

true

false

**equalsIgnoreCase()**

The String class also has a method called equalsIgnoreCase() which compares two strings but ignores the case of the characters. Thus, uppercase characters are considered to be equal to their lowercase equivalents.

**startsWith() and endsWith()**

The startsWith() and endsWith() methods check if the String starts with a certain substring. Here are a few examples:

String one = "This is a good day to code";

System.out.println( one.startsWith("This") );

System.out.println( one.startsWith("This", 5) );

System.out.println( one.endsWith ("code") );

System.out.println( one.endsWith ("shower") );

This example creates a String and checks if it starts and ends with various substrings.

The first line (after the String declaration) checks if the String starts with the substring "This". Since it does, the startsWith() method returns true.

The second line checks if the String starts with the substring "This" when starting the comparison from the character with index 5. The result is false, since the character at index 5 is "i".

The third line checks if the String ends with the substring "code". Since it does, the endsWith() method returns true.

The fourth line checks if the String ends with the substring "shower". Since it does not, the endsWith()method returns false.

**compareTo()**

The compareTo() method compares the String to another String and returns an int telling whether this String is smaller, equal to or larger than the other String. If the String is earlier in sorting order than the other String, compareTo() returns a negative number. If the String is equal in sorting order to the other String, compareTo() returns 0. If the String is after the other String in sorting order, the compareTo() metod returns a positive number.

Here is an example:

String one = "abc";

String two = "def";

String three = "abd";

System.out.println( one.compareTo(two) );

System.out.println( one.compareTo(three) );

This example compares the one String to two other Strings. The output printed from this code would be:

-3

-1

The numbers are negative because the one String is earlier in sorting order than the two other Strings.

The compareTo() method actually belongs to the Comparable interface. This interface is described in more detail in my tutorial about [**Sorting**](http://tutorials.jenkov.com/java-collections/sorting.html).

You should be aware that the compareTo() method may not work correctly for Strings in different languages than English. To sort Strings correctly in a specific language, use a [**Collator**](http://tutorials.jenkov.com/java-internationalization/collator.html).

**Trimming Strings With trim()**

The Java String class contains a method called trim() which can trim a string object. By *trim* is meant to remove white space characters at the beginning and end of the string. White space characters include space, tab and new lines. Here is a Java String trim() example:

String text = " And he ran across the field ";

String trimmed = text.trim();

After executing this code the trimmed variable will point to a String instance with the value

"And he ran across the field"

The white space characters at the beginning and end of the String object have been removed. The white space character inside the String have not been touched. By *inside* is meant between the first and last non-white-space character.

The trim() method does not modify the String instance. Instead it returns a new Java String object which is equal to the String object it was created from, but with the white space in the beginning and end of the String removed.

The trim() method can be very useful to trim text typed into input fields by a user. For instance, the user may type in his or her name and accidentally put an extra space after the last word, or before the first word. The trim() method is an easy way to remove such extra white space characters.

**Replacing Characters in Strings With replace()**

The Java String class contains a method named replace() which can replace characters in a String. The replace() method does not actually replace characters in the existing String. Rather, it returns a new String instance which is equal to the String instance it was created from, but with the given characters replaced. Here is a Java String replace() example:

String source = "123abc";

String replaced = source.replace('a', '@');

After executing this code the replaced variable will point to a String with the text:

123@bc

The replace() method will replace all character matching the character passed as first parameter to the method, with the second character passed as parameter to the replace() method.

**replaceFirst()**

The Java String replaceFirst() method returns a new String with the first match of the regular expression passed as first parameter with the string value of the second parameter.

Here is a replaceFirst() example:

String text = "one two three two one";

String s = text.replaceFirst("two", "five");

This example will return the string "one five three two one".

**replaceAll()**

The Java String replaceAll() method returns a new String with all matches of the regular expression passed as first parameter with the string value of the second parameter.

Here is a replaceAll() example:

String text = "one two three two one";

String t = text.replaceAll("two", "five");

This example will return the string "one five three five one".

**Splitting Strings With split()**

The Java String class contains a split() method which can be used to split a String into an [**array**](http://tutorials.jenkov.com/java/arrays.html) of String objects. Here is a Java String split() example:

String source = "A man drove with a car.";

String[] occurrences = source.split("a");

After executing this Java code the occurrences array would contain the String instances:

"A m"

"n drove with "

" c"

"r."

The source String has been split on the a characters. The Strings returned do not contain the a characters. The a characters are considered delimiters to split the String by, and the delimiters are not returned in the resulting String array.

The parameter passed to the split() method is actually a [**Java regular expression**](http://tutorials.jenkov.com/java-regex/index.html). Regular expressions can be quite advanced. The regular expression above just matched all a characters. It even only matched lowercase a characters.

The String split() method exists in a version that takes a limit as a second parameter. Here is a Java String split() example using the limit parameter:

String source = "A man drove with a car.";

int limit = 2;

String[] occurrences = source.split("a", limit);

The limit parameter sets the maximum number of elements that can be in the returned array. If there are more matches of the regular expression in the String than the given limit, then the array will contain limit - 1 matches, and the last element will be the rest of the String from the last of the limit - 1matches. So, in the example above the returned array would contain these two Strings:

"A m"

"n drove with a car."

The first String is a match of the a regular expression. The second String is the rest of the String after the first match.

Running the example with a limit of 3 instead of 2 would result in these Strings being returned in the resulting String array:

"A m"

"n drove with "

" car."

Notice how the last String still contains the a character in the middle. That is because this String represents the rest of the String after the last match (the a after 'n drove with ').

Running the example above with a limit of 4 or higher would result in only the Split strings being returned, since there are only 4 matches of the regular expression a in the String.

**Converting Numbers to Strings With valueOf()**

The Java String class contains a set of overloaded static methods named valueOf() which can be used to convert a number to a String. Here are some simple Java String valueOf() examples:

String intStr = String.valueOf(10);

System.out.println("intStr = " + intStr);

String flStr = String.valueOf(9.99);

System.out.println("flStr = " + flStr);

The output printed from this code would be:

intStr = 10

flStr = 9.99

**Converting Objects to Strings**

The Object class contains a method named toString(). Since all Java classes extends (inherits from) the Object class, all objects have a toString() method. This method can be used to create a String representation of the given object. Here is a Java toString() example:

Integer integer = new Integer(123);

String intStr = integer.toString();

Note: For the toString() method to return a sane String representation of the given object, the class of the object must have overridden the toString() method. If not, the default toString() method (inherited from the Object class) will get called. The default toString() method does not provide that much useful information. Many built-in Java classes have a sensible toString() method already.

**Getting Characters and Bytes**

It is possible to get a character at a certain index in a String using the charAt() method. Here is an example:

String theString = "This is a good day to code";

System.out.println( theString.charAt(0) );

System.out.println( theString.charAt(3) );

This code will print out:

T

s

since these are the characters located at index 0 and 3 in the String.

You can also get the byte representation of the String method using the getBytes() method. Here are two examples:

String theString = "This is a good day to code";

byte[] bytes1 = theString.getBytes();

byte[] bytes2 = theString.getBytes(Charset.forName("UTF-8");

The first getBytes() call return a byte representation of the String using the default character set encoding on the machine. What the default character set is depends on the machine on which the code is executed. Therefore it is generally better to explicitly specify a character set to use to create the byte representation (as in the next line).

The second getBytes() call return a UTF-8 byte representation of the String.

**Converting to Uppercase and Lowercase**

You can convert Strings to uppercase and lowercase using the methods toUpperCase() and toLowerCase(). Here are two examples:

String theString = "This IS a mix of UPPERcase and lowerCASE";

String uppercase = theString.toUpperCase();

String lowercase = theString.toLowerCase();

**Additional Methods**

The String class has several other useful methods than the ones described in this tutorial. You can find them all in the String JavaDoc.