This document outlines a few methods available in the Java `String` class. Although the `String` class has some class (`static`) methods, the ones discussed here are instance (non-`static`) methods, and thus need to be called with an *instance* of the `String` class. For example, we cannot call the `String` class method `length()` by invoking "`String.length()`". We must first create an instance of the `String` class. This is shown in the code below. In line `1`, we create an instance of the `String` class with an identifier, `greeting`, and then in line `2` use the `greeting` object to call the `length()` method.

```
1 String greeting = "Hello World!";
2 int len = greeting.length();
```

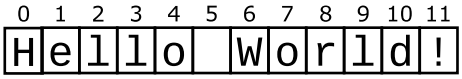
The `String` class methods below have been placed in an order that allows a step-by-step explanation of the concepts required to understand how strings work in most high-level (3rd generation) programming languages. If you are unfamiliar with how strings are commonly handled in computers, it is highly recommended that you read through this document in the order it is presented first. Once you have a good understanding of strings, more concise and complete documentation, such as the online Java documentation provided by Oracle, will likely be a more convenient reference. The URL for Oracle's Java 8 String class documentation at time of writing is available at the URL:

`https://docs.oracle.com/javase/8/docs/api/java/lang/String.html`

`char charAt(int index)`

This method returns the character (`char`) value at the specified index.

Consider the following diagram representing "`Hello World!`" stored as a `String`.



Character Indexes

For this diagram, each character in the string has been separated into its own box, and the index of the character in the string is written above each character. There are `12` characters in the string, and the indexes range from `0` through `11`. Because indexes are zero-based, the length of a string will always be one greater than the last index.

Using the `charAt` Method

The `charAt` method returns the character at the index that is given as a parameter to the method. As an example, for the `String` "`Hello World!`", `charAt(0)` will return the character "`H`", `charAt(1)` will return the character "`e`", and so on.

The Space Character

It is important to realize that index `5` in the string is not "empty" – it contains the "space" character, which is represented by the ASCII (and Unicode) value of `0x20` (which has a denary value of `32`). Thus, `charAt(5)` will return the space character.

`int length()`

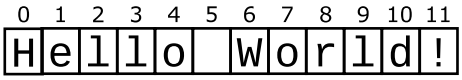
This method returns the number of characters in a `String`.

Using the `length` method

After executing the following code:

```
1 String greeting = "Hello World!";
2 int len = greeting.length();
```

The integer variable `len` will contain the value `12` because there are twelve characters in the string `greeting`. Note that every character, including the "space" character and punctuation is counted. Although the length is `12`, there is no character with index `12`. The indexes for the characters in the string run from `0` to `11`, as shown in the following diagram.



Using the `length` method to Iterate Through a String

If we wish to loop through all the characters in a string using a `for` loop, we would initialize the loop variable to `0` (the lowest index) and loop up to, but not including the length of the string. If the string identifier is "`greeting`", the loop condition will be "`i < greeting.length()`". Finally, we increment the loop counter by `1` with each iteration. Thus, to loop through each character in a string, our loop would look something like:

```
1 for( int i = 0; i < greeting.length(); i++ ) {
2     char c = greeting.charAt(i);
3     // more code
4 }
```

Using the `length` method to Iterate Through a String Backwards

There are a number of reasons we may wish to loop backwards over the string. To do so, we would initialize the loop variable to the index of the last character in the array. Notice that the value returned by the `length` is one greater than the index of the last element. We thus need to initialize the loop variable to "`greeting.length - 1`". Then we will want to include `0` as the final element, and thus the "`<`" operator must change to the "`>=`", giving us a loop condition of "`i >= 0`". Finally, we are counting down so will need to decrement our loop counter by `1` with each iteration. To loop through a string backwards, our loop will look something like:

```
1 for( int i = greeting.length() - 1; i >= 0; i-- ) {
2     char c = greeting.charAt(i);
3     // more code
4 }
```

Exercise 1:

Write a method named `printVertically` that takes a `String` parameter and uses the `String` class methods `charAt` and `length` to output the parameter to the console vertically. For example, the statement "`printVertically("Hello");`" would produce the following output:

```
H
e
l
l
o
```

`String substring(int beginIndex, int endIndex)`

This method makes a copy of a specified part of the `String`.

This method is overloaded; find a description of the single-parameter `substring` method below.

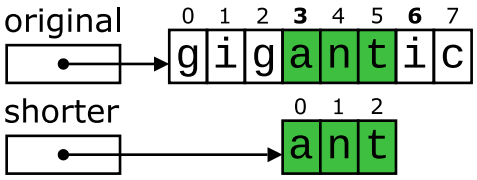
Using the `substring` method

This method is mostly intuitive, however, there is one thing you might find a little bit strange. That is the new string object contains the letter at the `beginIndex`, but does not include the letter at the `endIndex`.

Let us look at an example.

```
String original = "gigantic";
String shorter = original.substring(3, 6);
```

Remember that indexes for a string start at `0`, so if you count starting from zero, the character with index `3` is the letter "`a`". Then continue to count and you will find the character at index `6` is the second letter "`i`". Now the string is cut to include the letter at index `3` and exclude the letter at index `6`. Thus, a reference to the resulting string "`ant`" is stored in the variable named `shorter`. The diagram below shows this. Take special note of the indexes used to reference the start and end of the substring.



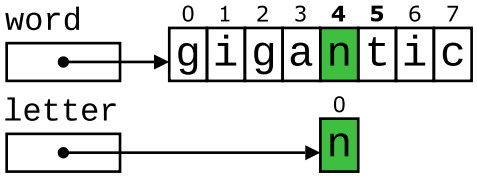
Note that the original string is unchanged.

Using the `substring` Method to Obtain a Single Character as a `String`

If you wish to get the character at a particular index returned as type `char`, you can use the `String` method `charAt`, but if you want a single character returned as type `String`, you can use the `substring` method by cutting from the index to the index plus one. Consider the following code.

```
String word = "gigantic";
int index = 4;
String letter = word.substring(index, index+1);
```

This code will set `letter` to a `String` containing the single character "n". The results of executing this code is represented in the diagram, below.



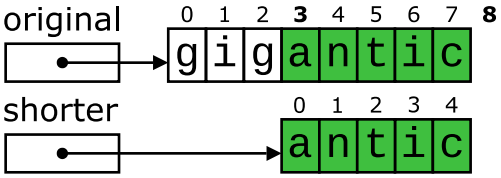
Using the `substring` Method to Obtain then Tail of a `String`

If we wished to cut to the end of the string, we can use the single-parameter substring method (described below); however, it is useful to understand how this would work with the this two-parameter version that we are discussing here.

Consider the following code.

```
String original = "gigantic";
String shorter = original.substring( 3, original.length() );
```

In this case, the call to "`original.length()`" will return a value of 8. Notice that the indexes of the string "`gigantic`" only run from 0 through 7. There is no index 8 in the `String`. This does not cause an error because the method `substring` does not include the final index when it copies the string. This is shown diagrammatically, below.



`String substring(int beginIndex)`

This method makes a copy of a part of the `String` starting at `beginIndex` and ending at the end of the original string.

This method is overloaded. The two-parameter `substring` method was described above, and should be well understood before reading about this version.

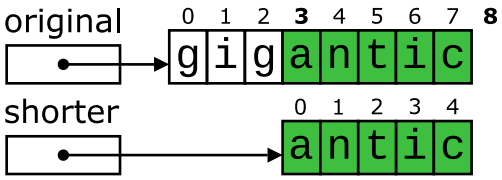
Using the `substring` Method to Obtain then Tail of a `String` (Revisited)

This method is equivalent to the two-parameter `substring` method, except that it will always copy the string until the end. Thus, given a string, `s`, the statements "`s.substring(i)`" and "`s.substring( i, s.length() )`" are equivalent.

For completeness, here is an example equivalent to the final example in the previous section.

```
String original = "gigantic";
String shorter = original.substring(3);
```

And the diagram from the previous section also applies here.



The code results in a reference to the string "`antic`" stored in the variable `shorter`. Again, the original string is left unchanged.

Exercise 2:

Write a method named `cutOffTail` that takes a `String` parameter called `s` and an `int` parameter called `n`, and uses only the `String` methods `length` and `substring` to return a `String` that contains the original string, except with the last `n` characters removed. Example test code and output are given below

Example Test Code

```
1 String s = "programming";
2 System.out.println( cutOffTail(s,4) );
3 System.out.println( cutOffTail(s,8) );
```

Example Test Code Output

```
program
pro
```

`boolean equals(Object s)`

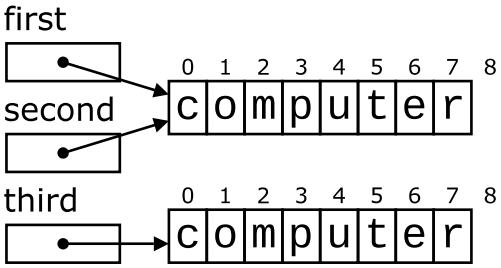
The `equals` method compares a `String` object to another object that is given as a parameter. The result is true if and only if the parameter is not `null` and is a `String` object that represents the same sequence of characters as the object.

Why the Equality Operator (==) is Generally Not Used on Objects

We will explain the meaning of the above description of the `equals` method more thoroughly below, but first it will help to understand why we usually cannot use the Java equality operator (==) to compare objects, including `String` objects. Consider some example code.

```
1 String first = "homework";
2 String second = "homework";
3 String third = new String("homework");
```

The code above creates a structure in memory that is represented in the diagram below.



Java compilers are intelligent enough that they will try to conserve memory space. Since the Java code sets both `first` and `second` to the same string literal value, "`homework`", it saves memory space to just create one copy of the string, and then save the *reference* to (location where to find) that string in both `first` and `second`. In the diagram, the references are shown as arrows. The arrows in both `first` and `second` point to the same string of characters.

The third line explicitly tells the compiler to create a new array of characters in memory containing the string of characters "`homework`". This is a separate string of characters from the ones referenced by `first` and `second`, and the reference saved in `third` points to the new set of characters.

For the code above, the following table contains two equality expressions are given and the value that the expression evaluates to.

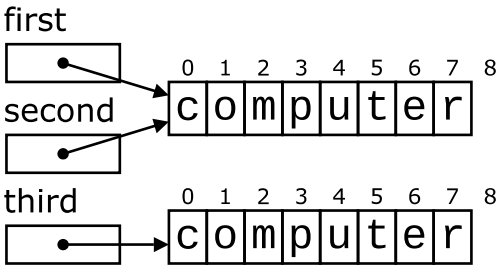
expression	evaluates to
`first == second`	`true`
`first == third`	`false`

Since the reference stored in `first` and `second` are the same value (the arrows point to the same string), the expression "`first == second`" will evaluate to `true`.

Since the reference stored in `third` is different from the reference stored in `first` (each arrow points to a different location in memory), the expression "`first == third`" evaluates to `false`. The equality operator only compares the reference values, it does not compare the characters, so even if each and every character in the string is the same, the references are not, so "`first == third`" will return `false`.

The code and diagram from above is repeated here for further discussion.

```
1 String first = "homework";
2 String second = "homework";
3 String third = new String("homework");
```



How to Compare Objects Using the `equals` Method

If one wishes to actually compare the values stored in an object – in this case a `String` object – one uses the `equals` method. Each class must define for itself what it means for two objects (instances of the class) to be "equal".

As it is defined in the `String` class, two objects of type `String` are considered equal if and only if the length of the two strings are the same, and each character at each and every index within the string is exactly the same. This includes each and every space character and punctuation character.

The `equals` method is case sensitive, meaning the character "`A`" is not equal to the character "`a`". For a case-insensitive comparison, there is a `String` class method `equalsIgnoreCase`.

The following table contains two different calls to the `equals` method, and the value that the expression evaluates to (assuming the strings are defined as in the code and diagram given above).

expression	evaluates to
`first.equals(second)`	`true`
`first.equals(third)`	`true`

Since the reference stored in `first` and `second` are the same value, obviously the call to method `equals` will return `true` because the characters in the string will be the same.

For the expression "`first.equals(third)`", the `equals` method will check each and every character in the two separate memory locations. Since the length of each string of characters is the same (equal to `8`), and the character at each and every location is the same, the method will return `true`.