Characters ← **Strings** → String Builders

Strings

Creating 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.

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

```java
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!*.

As with any other object, you can create `String` objects by using the `new` keyword and a constructor. The `String` class has thirteen constructors that allow you to provide the initial value of the string using different sources, such as an array of characters:

```java
char[] helloArray = { 'h', 'e', 'l', 'l', 'o', '.' };
String helloString = new String(helloArray);
System.out.println(helloString);
```

The last line of this code snippet displays `hello`.

> *Note: The `String` class is immutable, so that once it is created a `String` object cannot be changed. The `String` class has a number of methods, some of which will be discussed below, that appear to modify strings. Since strings are immutable, what*

> *these methods really do is create and return a new string that contains the result of the operation.*

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:

```java
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: Up until Java SE 15, 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:*

```java
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.

Starting with Java SE 15, you can write two-dimensional string literals:

```java
String html = """
              <html>
                  <body>
                      <p>Hello, world</p>
                  </body>
              </html>
              """;
```

Creating Format Strings

You have seen the use of the `printf()` and `format()` methods to print output with formatted numbers. The `String` class has an equivalent class method, `format()`, that returns a `String` object rather than a `PrintStream` object.

Using `String`'s static `format()` method allows you to create a formatted string that you can reuse, as opposed to a one-time print statement. For example, instead of

```java
System.out.printf("The value of the float " +
                  "variable is %f, while " +
                  "the value of the " + 
                  "integer variable is %d, " +
                  "and the string is %s", 
                  floatVar, intVar, stringVar); 
```

you can write

```java
String fs;
fs = String.format("The value of the float " +
                   "variable is %f, while " +
                   "the value of the " + 
                   "integer variable is %d, " +
                   " and the string is %s", 
                   floatVar, intVar, stringVar); 
System.out.println(fs); 
```

Converting Strings to Numbers

Frequently, a program ends up with numeric data in a string object—a value entered by the user, for example.

The `Number` subclasses that wrap primitive numeric types (`Byte`, `Integer`, `Double`, `Float`, `Long`, and `Short` each provide a class method named `valueOf()` that converts a string to an object of that type. Here is an example, `ValueOfDemo`, that gets two strings from the command line, converts them to numbers, and performs arithmetic operations on the values:

```java
public class ValueOfDemo {
    public static void main(String[] args) {

        // this program requires two 
        // arguments on the command line 
        if (args.length == 2) {
            // convert strings to numbers
            float a = (Float.valueOf(args[0])).floatValue();
            float b = (Float.valueOf(args[1])).floatValue();

            // do some arithmetic
            System.out.println("a + b = " +
                               (a + b));
            System.out.println("a - b = " +
                               (a - b));
            System.out.println("a * b = " +
                               (a * b));
            System.out.println("a / b = " +
                               (a / b));
            System.out.println("a % b = " +
                               (a % b));
        } else {
            System.out.println("This program " +
                "requires two command-line arguments.");
        }
    }
}
```

The following is the output from the program when you use `4.5` and `87.2` for the command-line arguments:

```
a + b = 91.7
a - b = -82.7
a * b = 392.4
a / b = 0.0516055
a % b = 4.5
```

> *Note: Each of the [`Number`] subclasses that wrap primitive numeric types also provides a `parseXXXX()` method. For example, [`parseFloat()`] can be used to convert strings to primitive numbers. Since a primitive type is returned instead of an object, the [`parseFloat()`] method is more direct than the [`valueOf()`] method. For example, in the `ValueOfDemo` program, we could use:*

```
float a = Float.parseFloat(args[0]);
float b = Float.parseFloat(args[1]);
```

Converting Numbers to Strings

Sometimes you need to convert a number to a string because you need to operate on the value in its string form. There are several easy ways to convert a number to a string:

```java
int i;
// Concatenate "i" with an empty string; conversion is handled for you.
String s1 = "" + i;
```

or

```java
// The valueOf class method.
String s2 = String.valueOf(i);
```

Each of the `Number` subclasses includes a class method, `toString()`, that will convert its primitive type to a string. For example:

```java
int i;
double d;
String s3 = Integer.toString(i);
String s4 = Double.toString(d);
```

The `ToStringDemo` example uses the `toString()` method to convert a number to a string. The program then uses some string methods to compute the number of digits before and after the decimal point:

```java
public class ToStringDemo {

    public static void main(String[] args) {
        double d = 858.48;
        String s = Double.toString(d);

        int dot = s.indexOf('.');

        System.out.println(dot + " digits " +
            "before decimal point.");
        System.out.println( (s.length() - dot - 1) +
            " digits after decimal point.");
    }
}
```

The output of this program is:

```
3 digits before decimal point.
2 digits after decimal point.
```

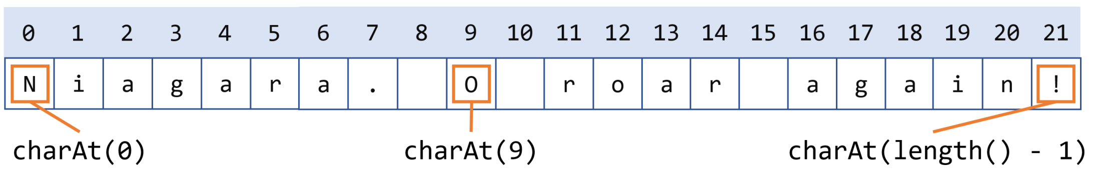
Getting Characters and Substrings by Index

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.

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:

```java
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:



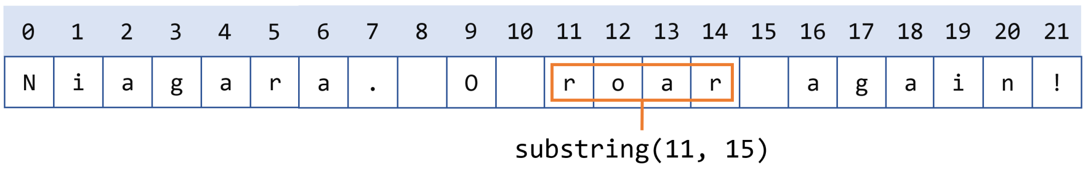
Char indexes in a string

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:

- `String substring(int beginIndex, int endIndex)`: Returns a new string that is a substring of this string. The substring begins at the specified `beginIndex` and extends to the character at index `endIndex - 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);
```



Extracting characters from a string with substring

Other Methods for Manipulating Strings

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

- `String[] split(String regex)` and `String[] split(String regex, int limit)`: Searches for a match as specified by the string argument (which contains a regular expression) and splits this string into an array of strings accordingly. The optional integer argument specifies the maximum size of the returned array. Regular expressions are covered in the section titled Regular Expressions.
- `CharSequence subSequence(int beginIndex, int endIndex)`: Returns a new character sequence constructed from `beginIndex` index up until `endIndex - 1`.
- `String trim()`: Returns a copy of this string with leading and trailing white space removed.
- `String toLowerCase()` and `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 is not important.

The search methods are the following:

- `int indexOf(int ch)` and `int lastIndexOf(int ch)`: Returns the index of the first (last) occurrence of the specified character.
- `int indexOf(int ch, int fromIndex)` and `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)` and `int lastIndexOf(String str)`: Returns the index of the first (last) occurrence of the specified substring.
- `int indexOf(String str, int fromIndex)` and `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.

> *Note: `CharSequence` is an interface that is implemented by the `String` class. Therefore, you can use a string as an argument for the `contains()` method.*

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:

- `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.

Regular expressions are discussed in the lesson titled Regular Expressions.

The String Class in Action

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 do not 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.*

```java
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 is 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 and Portions of Strings

The `String` class has a number of methods for comparing strings and portions of strings. The following table lists these methods.

- `boolean endsWith(String suffix)` and `boolean startsWith(String prefix)`: Returns `true` if this string ends with or begins with the substring specified as an argument to the method.
- `boolean startsWith(String prefix, int offset)`: Considers the string beginning at the index `offset`, and returns `true` if it begins with the substring specified as an argument.
- `int compareTo(String anotherString)`: Compares two strings lexicographically. Returns an integer indicating whether this string is greater than (result is > 0), equal to (result is = 0), or less than (result is < 0) the argument.
- `int compareToIgnoreCase(String str)`: Compares two strings lexicographically, ignoring differences in case. Returns an integer indicating whether this string is greater than (result is > 0), equal to (result is = 0), or less than (result is < 0) the argument.
- `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.
- `boolean regionMatches(int toffset, String other, int ooffset, int len)`: Tests whether the specified region of this string matches the specified region of the

String argument. Region is of length len and begins at the index toffset for this string and ooffset for the other string.

- boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len): Tests whether the specified region of this string matches the specified region of the String argument. Region is of length len and begins at the index toffset for this string and ooffset for the other string. The boolean argument indicates whether case should be ignored; if true, case is ignored when comparing characters.
- boolean matches(String regex): Tests whether this string matches the specified regular expression. Regular expressions are discussed in the lesson titled Regular Expressions.

The following program, RegionMatchesDemo, uses the regionMatches() method to search for a string within another string:

```
public class RegionMatchesDemo {
    public static void main(String[] args) {
        String searchMe = "Green Eggs and Ham";
        String findMe = "Eggs";
        int searchMeLength = searchMe.length();
        int findMeLength = findMe.length();
        boolean foundIt = false;
        for (int i = 0;
             i <= (searchMeLength - findMeLength);
             i++) {
           if (searchMe.regionMatches(i, findMe, 0, findMeLength)) {
              foundIt = true;
              System.out.println(searchMe.substring(i, i + findMeLength));
              break;
           }
        }
        if (!foundIt)
            System.out.println("No match found.");
    }
}
```

The output from this program is Eggs.

The program steps through the string referred to by searchMe() one character at a time. For each character, the program calls the regionMatches() method to determine whether the substring beginning with the current character matches the string the program is looking for.

Last update: *September 14, 2021*

Characters ← **Strings** → String Builders

Home > Tutorials > Numbers and Strings > Strings