**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](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html) 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!.

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:

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 ith 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](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html) 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.

**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

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

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 Between Numbers and Strings**

**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](https://docs.oracle.com/javase/8/docs/api/java/lang/Byte.html), [Integer](https://docs.oracle.com/javase/8/docs/api/java/lang/Integer.html), [Double](https://docs.oracle.com/javase/8/docs/api/java/lang/Double.html), [Float](https://docs.oracle.com/javase/8/docs/api/java/lang/Float.html), [Long](https://docs.oracle.com/javase/8/docs/api/java/lang/Long.html), and [Short](https://docs.oracle.com/javase/8/docs/api/java/lang/Short.html)) each provide a class method named valueOf that converts a string to an object of that type. Here is an example, [ValueOfDemo](https://docs.oracle.com/javase/tutorial/java/data/examples/ValueOfDemo.java) , that gets two strings from the command line, converts them to numbers, and performs arithmetic operations on the values:

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()) that 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:

int i;

// Concatenate "i" with an empty string; conversion is handled for you.

String s1 = "" + i;

or

// 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:

int i;

double d;

String s3 = Integer.toString(i);

String s4 = Double.toString(d);

The [ToStringDemo](https://docs.oracle.com/javase/tutorial/java/data/examples/ToStringDemo.java) 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:

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.

**anipulating 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 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);

**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[] split(String regex) 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 lesson 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() 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. |

**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:

|  |  |
| --- | --- |
| **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](https://docs.oracle.com/javase/tutorial/java/data/examples/Filename.java), 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](https://docs.oracle.com/javase/tutorial/java/data/examples/FilenameDemo.java), 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 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.

|  |  |
| --- | --- |
| **Methods for Comparing Strings** | |
| **Method** | **Description** |
| boolean endsWith(String suffix) 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.

**The StringBuilder Class**

[StringBuilder](https://docs.oracle.com/javase/8/docs/api/java/lang/StringBuilder.html) objects are like [String](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html) objects, except that they can be modified. Internally, these objects are treated like variable-length arrays that contain a sequence of characters. At any point, the length and content of the sequence can be changed through method invocations.

Strings should always be used unless string builders offer an advantage in terms of simpler code (see the sample program at the end of this section) or better performance. For example, if you need to concatenate a large number of strings, appending to a StringBuilder object is more efficient.

**Length and Capacity**

The StringBuilder class, like the String class, has a length() method that returns the length of the character sequence in the builder.

Unlike strings, every string builder also has a *capacity*, the number of character spaces that have been allocated. The capacity, which is returned by the capacity() method, is always greater than or equal to the length (usually greater than) and will automatically expand as necessary to accommodate additions to the string builder.

|  |  |
| --- | --- |
| **StringBuilder Constructors** | |
| **Constructor** | **Description** |
| StringBuilder() | Creates an empty string builder with a capacity of 16 (16 empty elements). |
| StringBuilder(CharSequence cs) | Constructs a string builder containing the same characters as the specified CharSequence, plus an extra 16 empty elements trailing the CharSequence. |
| StringBuilder(int initCapacity) | Creates an empty string builder with the specified initial capacity. |
| StringBuilder(String s) | Creates a string builder whose value is initialized by the specified string, plus an extra 16 empty elements trailing the string. |

For example, the following code

// creates empty builder, capacity 16

StringBuilder sb = new StringBuilder();

// adds 9 character string at beginning

sb.append("Greetings");

will produce a string builder with a length of 9 and a capacity of 16:

The StringBuilder class has some methods related to length and capacity that the String class does not have:

|  |  |
| --- | --- |
| **Length and Capacity Methods** | |
| **Method** | **Description** |
| void setLength(int newLength) | Sets the length of the character sequence. If newLength is less than length(), the last characters in the character sequence are truncated. If newLength is greater than length(), null characters are added at the end of the character sequence. |
| void ensureCapacity(int minCapacity) | Ensures that the capacity is at least equal to the specified minimum. |

A number of operations (for example, append(), insert(), or setLength()) can increase the length of the character sequence in the string builder so that the resultant length() would be greater than the current capacity(). When this happens, the capacity is automatically increased.

**StringBuilder Operations**

The principal operations on a StringBuilder that are not available in String are the append() and insert() methods, which are overloaded so as to accept data of any type. Each converts its argument to a string and then appends or inserts the characters of that string to the character sequence in the string builder. The append method always adds these characters at the end of the existing character sequence, while the insert method adds the characters at a specified point.

Here are a number of the methods of the StringBuilder class.

|  |  |
| --- | --- |
| **Various StringBuilder Methods** | |
| **Method** | **Description** |
| StringBuilder append(boolean b) StringBuilder append(char c) StringBuilder append(char[] str) StringBuilder append(char[] str, int offset, int len) StringBuilder append(double d) StringBuilder append(float f) StringBuilder append(int i) StringBuilder append(long lng) StringBuilder append(Object obj) StringBuilder append(String s) | Appends the argument to this string builder. The data is converted to a string before the append operation takes place. |
| StringBuilder delete(int start, int end) StringBuilder deleteCharAt(int index) | The first method deletes the subsequence from start to end-1 (inclusive) in the StringBuilder's char sequence. The second method deletes the character located at index. |
| StringBuilder insert(int offset, boolean b) StringBuilder insert(int offset, char c) StringBuilder insert(int offset, char[] str) StringBuilder insert(int index, char[] str, int offset, int len) StringBuilder insert(int offset, double d) StringBuilder insert(int offset, float f) StringBuilder insert(int offset, int i) StringBuilder insert(int offset, long lng) StringBuilder insert(int offset, Object obj) StringBuilder insert(int offset, String s) | Inserts the second argument into the string builder. The first integer argument indicates the index before which the data is to be inserted. The data is converted to a string before the insert operation takes place. |
| StringBuilder replace(int start, int end, String s) void setCharAt(int index, char c) | Replaces the specified character(s) in this string builder. |
| StringBuilder reverse() | Reverses the sequence of characters in this string builder. |
| String toString() | Returns a string that contains the character sequence in the builder. |

**Note:** You can use any String method on a StringBuilder object by first converting the string builder to a string with the toString() method of the StringBuilder class. Then convert the string back into a string builder using the StringBuilder(String str) constructor.

**An Example**

The StringDemo program that was listed in the section titled "Strings" is an example of a program that would be more efficient if a StringBuilder were used instead of a String.

StringDemo reversed a palindrome. Here, once again, is its listing:

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 converts the string to an array of characters (first for loop), reverses the array into a second array (second for loop), and then converts back to a string.

If you convert the palindrome string to a string builder, you can use the reverse() method in the StringBuilder class. It makes the code simpler and easier to read:

public class StringBuilderDemo {

public static void main(String[] args) {

String palindrome = "Dot saw I was Tod";

StringBuilder sb = new StringBuilder(palindrome);

sb.reverse(); // reverse it

System.out.println(sb);

}

}

Running this program produces the same output:

doT saw I was toD

Note that println() prints a string builder, as in:

System.out.println(sb);

because sb.toString() is called implicitly, as it is with any other object in a println() invocation.

**Note:** There is also a StringBuffer class that is *exactly* the same as the StringBuilder class, except that it is thread-safe by virtue of having its methods synchronized. Threads will be discussed in the lesson on concurrency.

**Summary of Characters and Strings**

Most of the time, if you are using a single character value, you will use the primitive char type. There are times, however, when you need to use a char as an object—for example, as a method argument where an object is expected. The Java programming language provides a *wrapper* class that "wraps" the char in a Character object for this purpose. An object of type Character contains a single field whose type is char. This [Character](https://docs.oracle.com/javase/8/docs/api/java/lang/Character.html) class also offers a number of useful class (i.e., static) methods for manipulating characters.

Strings are a sequence of characters and are widely used in Java programming. In the Java programming language, strings are objects. The [String](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html) class has over 60 methods and 13 constructors.

Most commonly, you create a string with a statement like

String s = "Hello world!";

rather than using one of the String constructors.

The String class has many methods to find and retrieve substrings; these can then be easily reassembled into new strings using the **+** concatenation operator.

The String class also includes a number of utility methods, among them split(), toLowerCase(), toUpperCase(), and valueOf(). The latter method is indispensable in converting user input strings to numbers. The Number subclasses also have methods for converting strings to numbers and vice versa.

In addition to the String class, there is also a [StringBuilder](https://docs.oracle.com/javase/8/docs/api/java/lang/StringBuilder.html) class. Working with StringBuilder objects can sometimes be more efficient than working with strings. The StringBuilder class offers a few methods that can be useful for strings, among them reverse(). In general, however, the String class has a wider variety of methods.

A string can be converted to a string builder using a StringBuilder constructor. A string builder can be converted to a string with the toString() method.