

To know the details of Character class click on the link: https://www.tutorialspoint.com/java/lang/java_lang_character.htm

To know the details of character class click on the link: <https://www.geeksforgeeks.org/string-class-in-java/>

Characters



Simple representation of single character in Java : `char ch = 'a';`
`char uniChar = '\u03A9';` // Unicode for uppercase Greek omega character

`char[] charArray = { 'a', 'b', 'c', 'd', 'e' };` // an array of chars

The **Java** also provides a **wrapper** class that "wraps" the char in a **Character object**. An object of type **Character** contains a single field, whose type is char. This **Character** class also offers a number of useful class (i.e., static) methods for manipulating characters.

`Character ch = new Character('a');` //create a Character object with the Character constructor

The Java compiler will also create a Character object for you under some circumstances. For example, if you pass a primitive char into a method that expects an object, the compiler automatically converts the char to a Character for you. This feature is called **autoboxing**—or **unboxing**, if the conversion goes the other way.

✓ **Note:** The Character class is **immutable**, so that once it is created, a Character object cannot be changed.

The following table lists some of the most useful methods in the Character class, but is not exhaustive.

Useful Methods in the Character Class	
Method	Description
<code>boolean isLetter(char ch)</code> <code>boolean isDigit(char ch)</code>	Determines whether the specified char value is a letter or a digit, respectively.
<code>boolean isWhitespace(char ch)</code>	Determines whether the specified char value is white space.
<code>boolean isUpperCase(char ch)</code> <code>boolean isLowerCase(char ch)</code>	Determines whether the specified char value is uppercase or lowercase, respectively.
<code>char toUpperCase(char ch)</code> <code>char toLowerCase(char ch)</code>	Returns the uppercase or lowercase form of the specified char value.
<code>toString(char ch)</code>	Returns a String object representing the specified character value — that is, a one-character string.

Escape Sequences

A character preceded by a backslash (\) is an *escape sequence* and has special meaning to the compiler.

The following table shows the Java escape sequences:

Escape Sequences	
Escape Sequence	Description
<code>\t</code>	Insert a tab in the text at this point.
<code>\b</code>	Insert a backspace in the text at this point.
<code>\n</code>	Insert a newline in the text at this point.

\r	Insert a carriage return in the text at this point.
\f	Insert a formfeed in the text at this point.
\'	Insert a single quote character in the text at this point.
\"	Insert a double quote character in the text at this point.
\\	Insert a backslash character in the text at this point.

When an escape sequence is encountered in a print statement, the compiler interprets it accordingly. For example, if you want to put quotes within quotes you must use the escape sequence, \", on the interior quotes. `System.out.println("She said \"Hello!\" to me.");` will print the sentence `She said "Hello!" to me.`

String Class

A Java String contains an **immutable** sequence of **Unicode characters**. A Java String is an object of the class **java.lang** which is used to create and manipulate strings.

```
public final class String extends Object  
implements Serializable, Comparable<String>, CharSequence
```

Java String is, however, **special** not like an ordinary class.

Instead of making everything an object Java retains primitive types, so as to improve the performance of the language.

- **Primitives** are stored in the **call stack**, which require less storage spaces and are cheaper to manipulate.
- On the other hand, **objects** are stored in the **program heap**, which **require complex memory management** and more storage spaces.

For performance reason, Java's String is designed to be **in between a primitive and a class**. The special features in String include:

- String is **immutable**. That is, its content **cannot be modified** once it is created. For example, the method `toUpperCase()` constructs and returns a new String instead of modifying the its existing content.
- The '+' operator, which performs addition on primitives (such as int and double), is **overloaded to operate on String objects**. '+' performs concatenation for two String operands. Java **does not support operator overloading**. The '+' operator is the **only** operator that is **internally overloaded to support string concatenation in Java**.
- A String can be constructed by either:
 1. directly assigning a **string literal** (double-quoted texts such as "Hello, world!") to a String reference - *just like a primitive*, or
 2. via the "new" operator and constructor, similar to any other classes. However, this is not commonly-used and is not recommended.

For example,

```
String str1 = "Java is Hot";           // Implicit construction via string literal
String str2 = new String("I'm cool");  // Explicit construction via new
```

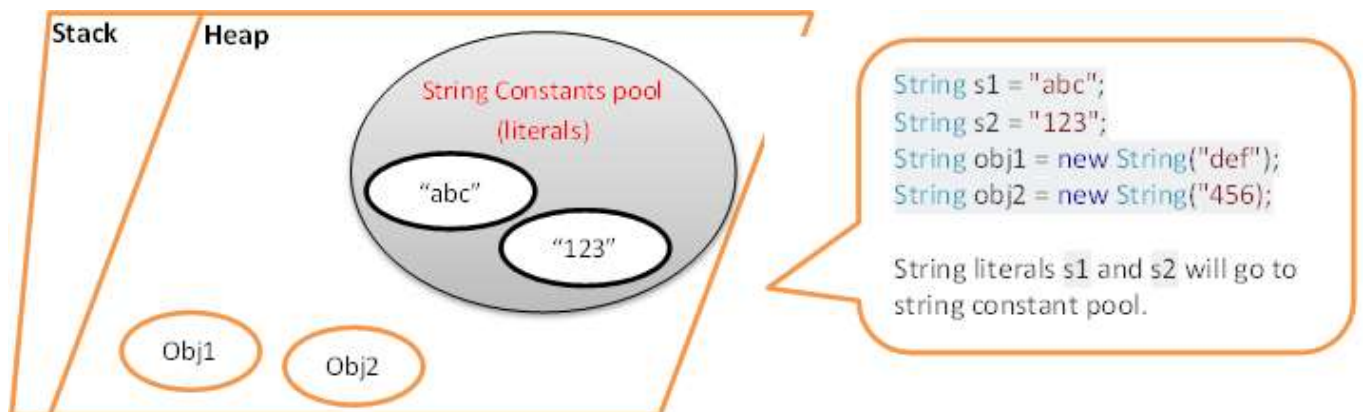
The String class has **eleven constructors** that allow you to provide the initial value of the string using different sources, such as an array of characters.

```
public class StringDemo{
    public static void main(String args[]){
        char[] helloArray = { 'h', 'e', 'l', 'l', 'o', '.' };
        System.out.println( helloString );
    }
}
```

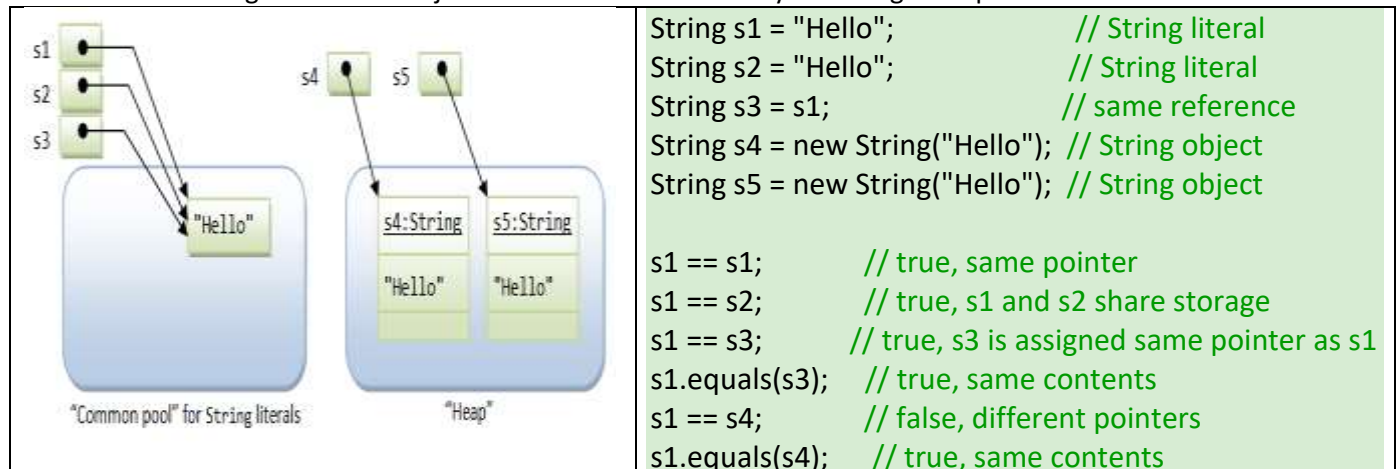
This would produce the following result: hello.

• String Literal vs. String Object

String literals are stored in a **common pool** (sometimes called **String Constants pool**) . This facilitates **sharing of storage** for strings with the same contents to conserve storage. String objects allocated via new operator are stored in the **heap**, and there is no sharing of storage for the same contents.



Allocation of string literals and objects can be demonstrated by following example.



	s4 == s5; // false, different pointers in heap s4.equals(s5); // true, same contents
--	---

- If two string literals have the same contents, they will share the same storage inside the **common pool**. This approach is adopted to **conserve storage for frequently-used strings**.
- String objects created via the **new** operator and constructor are kept in the **heap**.
- Each String object in the heap has its own storage just like any other object. **There is no sharing of storage in heap even if two String objects have the same contents.**

The method **equals()** of the String class is used to compare the contents of two Strings. However relational equality operator '==' is for comparing the **references** (or pointers) of two objects.

Important Notes:

String can be created by directly assigning a String literal which is shared in a common pool. It is uncommon and not recommended to **use the new operator** to construct a String object in the heap.

- **String is Immutable**

In C++ std::string class is **mutable**. Immutable strings are great when **memory is cheap**—this wasn't true when C++ was developed.

Since string literals with the same contents share storage in the common pool, Java's String is designed to be **immutable**. That is, once a String is constructed, **its contents cannot be modified**. Otherwise, the other String references sharing the same storage location will be affected by the change, which can be unpredictable and therefore is undesirable. Methods such as **toUpperCase()** might appear to modify the contents of a String object. **In fact, a completely new String object is created and returned to the caller.** The original String object will be reallocated, once there is no more references, and subsequently garbage-collected.

Because String is immutable, **it is not efficient to use String if you need to modify your string frequently** (that would create many new Strings occupying new storage areas). For example,

```
// inefficient codes
String str = "Hello";
for (int i = 1; i < 1000; ++i) {
    str = str + i;
}
```

If the contents of a String have to be modified frequently, use the **StringBuffer** or **StringBuilder** class instead.

- **Constructors**

Some Constructor and Description

String() : Initializes a newly created String object so that it represents an empty character sequence.

String(byte[] bytes) : Constructs a new String by decoding the specified array of bytes using the platform's default charset.

String(byte[] bytes, Charset charset) : Constructs a new String by decoding the specified array of bytes

Stringbuffer class:-

StringBuilder class:-

using the specified **charset**.

String(byte[] bytes, int offset, int length, **String** charsetName)

Constructs a new String by decoding the specified subarray of bytes using the specified charset.

The length of the new String is a function of the charset, and hence may not be equal to the length of the subarray.

String(byte[] bytes, **String** charsetName)

Constructs a new String by decoding the specified array of bytes using the specified **charset**.

String(char[] value) : Allocates a new String so that it represents the sequence of characters currently contained in the character array argument.

String(char[] value, int offset, int count)

Allocates a new String that contains characters from a subarray of the character array argument.

offset = the index of the first character of the subarray

count = the length of the subarray.

The contents of the subarray are copied; subsequent modification of the character array does not affect the newly created string.

String(**String** original)

Initializes a newly created String object so that it represents the same sequence of characters as the argument; in other words, the newly created string is a copy of the argument string.

String(**StringBuffer** buffer)

Allocates a new string that contains the sequence of characters currently contained in the string buffer argument.

String(**StringBuilder** builder)

Allocates a new string that contains the sequence of characters currently contained in the string builder argument.

Character set vs. **character encoding** - In these two terms, 'set' refers to the set of characters and their numbers (code points), and '**encoding**' refers to the representation of these code points. For example, **Unicode** is a character set, and **UTF-8** and **UTF-16** are different character encodings of Unicode.

StringBuffer & StringBuilder

JDK provides two classes to support **mutable strings**: **StringBuffer** and **StringBuilder** (in core package java.lang) . A StringBuffer or StringBuilder object is just like any ordinary object, which are stored in the heap and not shared, and therefore, can be modified without causing adverse side-effect to other objects.

- **StringBuilder** class was introduced in **JDK 1.5**. It is the same as **StringBuffer** class, except that **StringBuilder** is **not synchronized** for multi-thread operations. In other words, if multiple threads are accessing a **StringBuilder** instance at the same time, **its integrity cannot be guaranteed**. However, for

a single-thread program (most commonly), doing away with the overhead of synchronization makes the StringBuilder faster.

- StringBuilder is API-compatible with the StringBuffer class, i.e., having the **same set of constructors and methods**
- However, for single-thread program, StringBuilder, **without the synchronization overhead, is more efficient.**

Important methods of StringBuffer/StringBuilder

<pre>// Constructors StringBuffer() // an empty StringBuffer StringBuffer(int size) // specified initial size StringBuffer(String s) // specified initial content int length() // Length</pre>	<pre>// Methods for building up the content // type could be primitives, char[], String, // StringBuffer, etc StringBuffer append(type arg) StringBuffer insert(int offset, arg) ` The second argument is inserted into the contents of this sequence at the position indicated by offset. To insert after the second character, use the value 2. And to insert at the start, use zero. Example : // Initialize StringBuffer with this value. StringBuffer builder = new StringBuffer("abc"); // Insert this substring at position 2. builder.insert(2, "xyz"); System.out.println(builder); Result : abxyzc</pre>
<pre>// Methods for manipulating the content StringBuffer delete(int start, int end) StringBuffer deleteCharAt(int index) void setLength(int newSize) void setCharAt(int index, char newChar) StringBuffer replace(int start, int end, String s) StringBuffer reverse()</pre>	<pre>// Methods for extracting whole/part of the content char charAt(int index) String substring(int start) String substring(int start, int end) String toString()</pre>
<pre>// Methods for searching int indexOf(String searchKey) int indexOf(String searchKey, int fromIndex) int lastIndexOf(String searchKey) int lastIndexOf(String searchKey, int fromIndex)</pre>	<p>Note :</p> <ul style="list-style-type: none"> • Use a constructor to create a StringBuffer (instead of assigning to a String literal). • '+' operator does not apply to objects, including the StringBuffer. You need to use a proper method such as append() or insert() to manipulating a StringBuffer.

To create a string from parts, it is more efficient to use **StringBuffer (multi-thread)** or **StringBuilder (single-thread)** instead of via String concatenation. For example,

// Create a string of YYYY-MM-DD HH:MM:SS

```
int year = 2010, month = 10, day = 10;
int hour = 10, minute = 10, second = 10;
String dateStr = new StringBuilder()
    .append(year).append("-").append(month).append("-").append(day).append(" ")
    .append(hour).append(":").append(minute).append(":").append(second).toString();
System.out.println(dateStr);
```

// StringBuilder is more efficient than String concatenation

```
String anotherDataStr = year + "-" + month + "-" + day + " " + hour + ":" + minute + ":" + second;
System.out.println(anotherDataStr);
```

JDK compiler, in fact, uses both String and StringBuffer to handle string concatenation via the '+' operator. For examples,

String msg = "a" + "b" + "c"; will be compiled into the following codes for better efficiency:

```
String msg = new StringBuffer().append("a").append("b").append("c").toString();
```

Two objects are created during the process, an intermediate StringBuffer object and the returned String object.

Rule of Thumb: Strings are more efficient if they are not modified (because they are shared in the string common pool). However, if you have to modify the content of a string frequently (such as a status message), you should use the StringBuffer class (or the StringBuilder described below) instead.

Benchmarking String/StringBuffer/StringBuilder

The following program compare the times taken to reverse a long string via a String object and a StringBuffer.

// Reversing a long String via a String vs. a StringBuffer

```
public class StringsBenchMark {
    public static void main(String[] args) {
        long beginTime, elapsedTime;

        // Build a long string
        String str = "";    int size = 16536;    char ch = 'a';
        beginTime = System.nanoTime();    // Reference time in nanoseconds
        for (int count = 0; count < size; ++count) {
            str += ch;    ++ch;    if (ch > 'z') { ch = 'a';    }
        }
        elapsedTime = System.nanoTime() - beginTime;
        System.out.println("Elapsed Time is " + elapsedTime/1000 + " usec (Build String)");
    }
}
```

usec is microsec


```

// Reverse a String by building another String character-by-character in the reverse order
String strReverse = "";
beginTime = System.nanoTime();
for (int pos = str.length() - 1; pos >= 0 ; pos--) {
    strReverse += str.charAt(pos); // Concatenate
}
elapsedTime = System.nanoTime() - beginTime;
System.out.println("Elapsed Time is " + elapsedTime/1000 + " usec (Using String to reverse)");

// Reverse a String via an empty StringBuffer by appending characters in the reverse order
beginTime = System.nanoTime();
StringBuffer sBufferReverse = new StringBuffer(size);
for (int pos = str.length() - 1; pos >= 0 ; pos--) {
    sBufferReverse.append(str.charAt(pos)); // append
}
elapsedTime = System.nanoTime() - beginTime;
System.out.println("Elapsed Time is " + elapsedTime/1000 + " usec (Using StringBuffer to reverse)");

// Reverse a String by creating a StringBuffer with the given String and invoke its reverse()
beginTime = System.nanoTime();
StringBuffer sBufferReverseMethod = new StringBuffer(str);
sBufferReverseMethod.reverse(); // use reverse() method
elapsedTime = System.nanoTime() - beginTime;
System.out.println("Elapsed Time is " + elapsedTime/1000 + " usec (Using StringBuffer's reverse()
method)");

// Reverse a String via an empty StringBuilder by appending characters in the reverse order
beginTime = System.nanoTime();
StringBuilder sBuilderReverse = new StringBuilder(size);
for (int pos = str.length() - 1; pos >= 0 ; pos--) {
    sBuilderReverse.append(str.charAt(pos));
}
elapsedTime = System.nanoTime() - beginTime;
System.out.println("Elapsed Time is " + elapsedTime/1000 + " usec (Using StringBuilder to reverse)");

// Reverse a String by creating a StringBuilder with the given String and invoke its reverse()
beginTime = System.nanoTime();
StringBuffer sBuilderReverseMethod = new StringBuffer(str);
sBuilderReverseMethod.reverse();
elapsedTime = System.nanoTime() - beginTime;
System.out.println("Elapsed Time is " + elapsedTime/1000 + " usec (Using StringBuidler's reverse()
method)");
}

```



```
}
```

```
Elapsed Time is 332100 usec (Build String)
Elapsed Time is 346639 usec (Using String to reverse)
Elapsed Time is 2028 usec (Using StringBuffer to reverse)
Elapsed Time is 847 usec (Using StringBuffer's reverse() method)
Elapsed Time is 1092 usec (Using StringBuilder to reverse)
Elapsed Time is 836 usec (Using StringBuidler's reverse() method)
```

- **Observe StringBuilder is 2x faster than StringBuffer, and 300x faster than String.**
- The reverse() method is the fastest, which take about the same time for StringBuilder and StringBuffer.

Stringtokenizer class in Java: <https://www.geeksforgeeks.org/stringtokenizer-class-java-example-set-1-constructors/>

java.util.StringTokenizer (It's use is discouraged)

Very often, you need to break a line of texts into tokens delimited by white spaces. The java.util.StringTokenizer class supports this. For example, the following program reverses the words in a String.

// Reverse the words in a String using StringTokenizer

```
import java.util.StringTokenizer;
public class StringTokenizerTest {
    public static void main(String[] args) {
        String str = "Monday Tuesday Wednesday Thursday Friday Saturday Sunday";
        String strReverse;    StringBuilder sb = new StringBuilder();
        StringTokenizer st = new StringTokenizer(str);

        while (st.hasMoreTokens()) {
            sb.insert(0, st.nextToken());
            if (st.hasMoreTokens()) { sb.insert(0, " "); }
        }
        strReverse = sb.toString();    System.out.println(strReverse);
    }
}
```

Insert the token at the start and after each token insert space.

// Constructors

```
StringTokenizer(String s) // Constructs a StringTokenizer for the given string,
                          // using the default delimiter set of " \t\n\r\f"
                          // (i.e., blank, tab, newline, carriage-return, and form-feed).
                          // Delimiter characters themselves will not be treated as tokens.
StringTokenizer(String s, String delimiterSet) // Use characters in delimiterSet as delimiters.
```

// Methods

```
boolean hasNextToken() // Returns true if next token available
String nextToken() // Returns the next token
```

// Code Sample

```
StringTokenizer tokenizer = new StringTokenizer(aString);
while (tokenizer.hasNextToken()) {
    String token = tokenizer.nextToken();
    ....
}
```

The JDK documentation stated that **"StringTokenizer is a legacy class that is retained for compatibility reasons although its use is discouraged in new code."** It is recommended that anyone seeking this functionality use the **split()** method of String or the **java.util.regex** package instead."

For example, the following program uses the **split()** method of the String class to reverse the words of a String.

// Reverse the words in a String using split() method of the String class

```
public class StringSplitTest {
    public static void main(String[] args) {
        String str = "Monday Tuesday Wednesday Thursday Friday Saturday Sunday";
        String[] tokens = str.split("\\s");    // white space '\s' as delimiter
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < tokens.length; ++i) {
            sb.insert(0, tokens[i]);
            if (i < tokens.length - 1) { sb.insert(0, " "); }
        }
        String strReverse = sb.toString();
        System.out.println(strReverse);
    }
}
```

Commonly used methods of String Class

The class String includes methods for examining individual characters of the sequence, for comparing strings, for searching strings, for extracting substrings, and for creating a copy of a string with all characters translated to uppercase or to lowercase. Case mapping is based on the Unicode Standard version specified by the Character class.

Operation	Method Description and Use	
Length	<code>int length()</code> // returns the length of the String <code>// same as <i>thisString</i>.length == 0</code> <code>boolean isEmpty()</code>	<pre>public static void main(String args[]) { String palindrome = "Dot saw I was Tod"; int len = palindrome.length(); System.out.println("String Length is : " + len); } }</pre> <p>Result: String Length is : 17</p>

Comparison	<pre>// CANNOT use '==' or '!=' to compare two Strings in Java boolean equals(String another) boolean equalsIgnoreCase(String another) int compareTo(String another) // return 0 if both are same; // <0 if lexicographically less than another; or // >0</pre>	<pre>int compareToIgnoreCase(String another) boolean startsWith(String another) boolean startsWith(String another, int frIndex) // search begins at frIndex boolean endsWith(String another)</pre>
-------------------	---	--

compareTo : Compares two strings lexicographically. The comparison is based on the Unicode value of each character in the strings. The character sequence represented by this String object is compared lexicographically to the character sequence represented by the argument string.

public int compareTo(String anotherString)

Result : **negative** integer - if this String object lexicographically precedes the argument string.

positive integer - if this String object lexicographically follows the argument string.

zero - if the strings are equal

compareTo returns 0 exactly when the equals(Object) method would return true.

Definition of lexicographic ordering

- If two strings are different, then either they have different characters at some index that is a valid index for both strings, or their lengths are different, or both.
- If they have different characters at one or more index positions, let k be the smallest such index; then the string whose character at position k has the smaller value (using $<$ operator) lexicographically precedes the other string. In this case, compareTo returns the difference of the two character values at position k in the two string – that is, the value: `this.charAt(k)-anotherString.charAt(k)`
- If there is no index position at which they differ, then the shorter string lexicographically precedes the longer string. In this case, compareTo returns the difference of the lengths of the strings -- that is, the value: `this.length()-anotherString.length()`

startsWith

public boolean startsWith(String prefix, int toffset)

Tests if the substring of this string beginning at the specified index starts with the specified prefix.

Parameters: prefix - the prefix. toffset - where to begin looking in this string.

Returns:

true if the character sequence represented by the argument is a prefix of the substring of this object starting at index toffset; false otherwise.

The result is false if toffset is negative or greater than the length of this String object; otherwise the result is the same as the result of the expression

endsWith **public boolean endsWith(String suffix)** Tests if this string ends with the specified suffix.

Parameters: suffix - the suffix.

Returns:

true if the character sequence represented by the argument is a suffix of the character sequence represented by this object; false otherwise. Note that the result will be true if the argument is the empty string or is equal to this String object as determined by the `equals(Object)` method.

Searching & Indexing - 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.**

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.

Method	Description
<code>int indexOf(int ch)</code> <code>int lastIndexOf(int ch)</code>	Returns the index of the first (last) occurrence of the specified character.
<code>int indexOf(int ch, int fromIndex)</code> <code>int lastIndexOf(int ch, int fromIndex)</code>	Returns the index of the first (last) occurrence of the specified character, searching forward (backward) from the specified index.
<code>int indexOf(String str)</code> <code>int lastIndexOf(String str)</code>	Returns the index of the first (last) occurrence of the specified substring.
<code>int indexOf(String str, int fromIndex)</code> <code>int lastIndexOf(String str, int fromIndex)</code>	Returns the index of the first (last) occurrence of the specified substring, searching forward (backward) from the specified index.
<code>boolean contains(CharSequence s)</code>	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.

Special Note : (not in syllabus Unicode character)

Unicode Character Representations

The char data type (and therefore the value that a Character object encapsulates) are based on the **original Unicode specification**, which defined characters as **fixed-width 16-bit entities**. The Unicode Standard has since been changed to allow for characters whose representation requires **more than 16 bits**. The range of legal *code points* is now U+0000 to U+10FFFF, known as **Unicode scalar value**. (Refer to the *definition* of the U+n notation in the Unicode Standard.)

The set of characters from U+0000 to U+FFFF is sometimes referred to as the **Basic Multilingual Plane (BMP)**. Characters whose code points are greater than U+FFFF are called **supplementary characters**.

Architecture and terminology

Unicode defines a codespace of 1,114,112 [code points](#) in the range 0_{hex} to $10\text{FFFF}_{\text{hex}}$.^[5] Normally a Unicode code point is referred to by writing "U+" followed by its [hexadecimal](#) number. For code points in the [Basic Multilingual Plane](#) (BMP), four digits are used (e.g., U+0058 for the character LATIN CAPITAL LETTER X); for code points outside the BMP, five or six digits are used, as required (e.g., U+E0001 for the character LANGUAGE TAG and U+10FFFD for the character PRIVATE USE CHARACTER-10FFFD).^[6]

The Java platform uses the UTF-16 representation in char arrays and in the String and StringBuffer classes. In this representation, supplementary characters are represented as a pair of char values, the first from the *high-surrogates* range, (\uD800-\uDBFF), the second from the *low-surrogates* range (\uDC00-\uDFFF).

A char value, therefore, represents Basic Multilingual Plane (BMP) code points, including the surrogate code points, or code units of the UTF-16 encoding. An int value represents all Unicode code points, including supplementary code points. The **lower** (least significant) **21 bits** of int are used to represent Unicode code points and the **upper** (most significant) **11 bits** must be zero. Unless otherwise specified, the behavior with respect to supplementary characters and surrogate char values is as follows:

- The methods that only accept a char value cannot support supplementary characters. They treat char values from the surrogate ranges as undefined characters. For example, Character.isLetter('\uD840') returns false, even though this specific value if followed by any low-surrogate value in a string would represent a letter.
- The methods that accept an int value support all Unicode characters, including supplementary characters. For example, Character.isLetter(0x2F81A) returns true because the code point value represents a letter (a CJK ideograph).

In the Java SE API documentation, *Unicode code point* is used for character values in the range between U+0000 and U+10FFFF, and *Unicode code unit* is used for 16-bit char values that are code units of the UTF-16 encoding. For more information on Unicode terminology, refer to the [Unicode Glossary](#).

indexOf **public int indexOf(int ch)**

Returns the index within this string of the first occurrence of the specified character. If a character with value ch occurs in the character sequence represented by this String object, then the index (in Unicode code units) of the first such occurrence is returned. For values of ch in the range from 0 to 0xFFFF (inclusive), this is the smallest value *k* such that: `this.charAt(k) == ch` is true. For other values of ch, it is the smallest value *k* such that: `this.codePointAt(k) == ch`

May be ignored

is true. In either case, if no such character occurs in this string, then -1 is returned.

Parameters: ch - a character (Unicode code point).

indexOf **public int indexOf(int ch, int fromIndex)**

Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index.

For values of ch in the range from 0 to 0xFFFF (inclusive), this is the smallest value k such that:

`(this.charAt(k) == ch) && (k >= fromIndex)` is true.

For other values of ch, `(this.codePointAt(k) == ch) && (k >= fromIndex)` is true.

There is no restriction on the value of fromIndex. If it is negative, it has the same effect as if it were zero: this entire string may be searched. If it is greater than the length of this string, it has the same effect as if it were equal to the length of this string: -1 is returned.

All **indices** are specified in **char values** (Unicode code units).

indexOf **public int indexOf(String str)**

Returns the index within this string of the first occurrence of the specified substring.

The returned index is the smallest value k for which: `this.startsWith(str, k)`

If no such value of k exists, then -1 is returned.

Parameters: str - the substring to search for.

Returns: the index of the first occurrence of the specified substring, or -1 if there is no such occurrence.

indexOf **public int indexOf(String str, int fromIndex)**

Returns the index within this string of the first occurrence of the specified substring, starting at the specified index. The returned index is the smallest value k for which:

`k >= fromIndex && this.startsWith(str, k)` If no such value of k exists, then -1 is returned.

Extracting a char	// index from 0 to String's length - 1 char charAt(int index)	codePointAt(int index) Returns the character (Unicode code point) at the specified index.
Creating a new String or char[] from the original (Strings are immutable!)	String toLowerCase() String toUpperCase() // create a new String with oldChar replaced by newChar String replace(char oldChar, char newChar) // same as thisString + another String concat(String another) // create a char[] from this string	// create a new String removing white spaces from front and back String trim() char[] toCharArray() void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin) // copy into dst char[]

Static methods for converting primitives to String

static String ValueOf(type arg) // type can be primitives or char[]

Substring : `public String substring(int beginIndex)`

`public String substring(int beginIndex, int endIndex)`

It extends at the end of string

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`. Thus the length of the substring is `endIndex - beginIndex`.

Examples: `"hamburger".substring(4, 8)` returns `"urge"`
`"smiles".substring(1, 5)` returns `"mile"`

Parameters: `beginIndex` - the beginning index, **inclusive**. `endIndex` - the ending index, **exclusive**.

Throws: `IndexOutOfBoundsException` - if the `beginIndex` is negative, or `endIndex` is larger than the length of this `String` object, or `beginIndex` is larger than `endIndex`.

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:

Method	Description
<code>String replace(char oldChar, char newChar)</code>	Returns a new string resulting from replacing all occurrences of <code>oldChar</code> in this string with <code>newChar</code> .
<code>String replaceAll(String regex, String replacement)</code>	Replaces each substring of this string that matches the given regular expression with the given replacement.
<code>String replaceFirst(String regex, String replacement)</code>	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.

```
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() {
```

It extends at the end of the string


```

        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);
    }
    // Get the path name
    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
```

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`.

Creating Format Strings

Java provides `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);
```

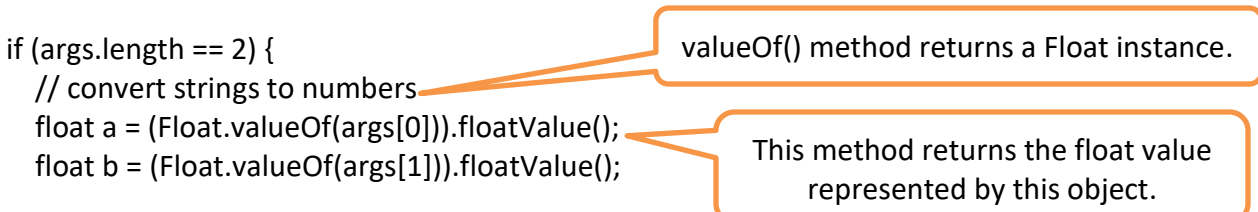
```
System.out.println(fs);
```

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, 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:

```
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));  
        } 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

```
int i;  
// Concatenate "i" with an empty string; conversion is handled for you.  
String s1 = "" + i;    or  
String s2 = String.valueOf(i);    // The valueOf class method.
```

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 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.

getChars : `public void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)`

Copies characters from this string into the destination character array.

The first character to be copied is at index `srcBegin`; the last character to be copied is at index `srcEnd-1` (thus the total number of characters to be copied is `srcEnd-srcBegin`). The characters are copied into the subarray of `dst` starting at index `dstBegin` and ending at index: `dstbegin + (srcEnd-srcBegin) - 1`

Parameters: `srcBegin` - index of the first character in the string to copy.
`srcEnd` - index after the last character in the string to copy.
`dst` - the destination array.
`dstBegin` - the start offset in the destination array.

Throws: `IndexOutOfBoundsException` - If any of the following is true:
`srcBegin` is negative or `srcBegin` is greater than `srcEnd` or
`srcEnd` is greater than the length of this string or `dstBegin` is negative or
`dstBegin+(srcEnd-srcBegin)` is larger than `dst.length`

replace : `public String replace(char oldChar, char newChar)`

Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.

If the character oldChar does not occur in the character sequence represented by this String object, then a **reference to this String object is returned**. Otherwise, a new String object is created that represents a character sequence identical to the character sequence represented by this String object, except that every occurrence of oldChar is replaced by an occurrence of newChar.

Examples:

"mesquite in your cellar".replace('e', 'o')	returns "mosquito in your collar"
"the war of baronets".replace('r', 'y')	returns "the way of bayonets"
"sparring with a purple porpoise".replace('p', 't')	returns "starring with a turtle tortoise"
"JonL".replace('q', 'x')	returns "JonL" (no change)

Parameters: oldChar - the old character. newChar - the new character.

Returns: a string derived from this string by replacing every occurrence of oldChar with newChar.

matches : `public boolean matches(String regex)`

Tells whether or not this string matches the given regular expression.

An invocation of this method of the form `str.matches(regex)` yields exactly the same result as the expression `Pattern.matches(regex, str)`

Returns: true if, and only if, this string matches the given regular expression

Throws: `PatternSyntaxException` - if the regular expression's syntax is invalid

split : `public String[] split(String regex, int limit)`

- The **java regular expression** to be used in evaluating how to split
- **Limit** - the maximum number of tokens the string will be split.

This method returns an **array** which corresponds to the result of splitting the String object given a **java regular expression** as a method parameter and the tokens are limited using the parameter limit.

The array returned by this method contains **each substring** of this string that is terminated by another substring that matches the given expression or is terminated by the end of the string.

If the expression does not match any part of the input then the **resulting array has just one element, namely this string.**

The **limit parameter** controls the number of times the pattern is applied and therefore affects the length of the resulting array. If the **limit n is greater than zero then the pattern will be applied at most $n - 1$** times, the array's length will be no greater than n , and the array's last entry will contain all input beyond the last matched delimiter.

If **n is non-positive** then the pattern will be applied **as many times as possible** and the array can have any length. If **n is zero** then the pattern will be applied as many times as possible, the array can have any length, and **trailing empty strings will be discarded**.

The string **"boo:and:foo"**, for example, yields the following results with these parameters:

Regex	Limit	Result
:	2	{ "boo", "and:foo" } split in maxi 2 tokens
:	5	{ "boo", "and", "foo" }
:	-2	{ "boo", "and", "foo" } split as many times as possible
o	5	{ "b", "", ":and:f", "", "" } (split between o gives "")
o	-2	{ "b", "", ":and:f", "", "" }
o	0	{ "b", "", ":and:f" }

An invocation of this method of the form `str.split(regex, n)` yields the same result as the expression `Pattern.compile(regex).split(str, n)`

Parameters: regex - the delimiting regular expression limit - the result threshold, as described above

split : **public String[] split(String regex)**

Splits this string around matches of the given regular expression.

This method works as if by invoking the two-argument split method with the given expression and a **limit argument of zero**. Trailing empty strings are therefore not included in the resulting array.

The string "boo:and:foo", for example, yields the following results with these expressions:

Regex	Result
:	{ "boo", "and", "foo" }
o	{ "b", "", ":and:f" }

The StringBuilder Class

StringBuilder objects are like String 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. 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.

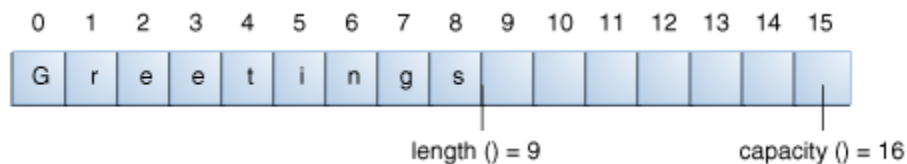
- Unlike strings, every string builder also has a *capacity*, the number of character spaces that have been allocated. The capacity, which is returned by the `capacity()` method, is always greater than or equal to the length (usually greater than) and will automatically expand as necessary to accommodate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

StringBuilder Constructors	
Constructor	Description
<code>StringBuilder()</code>	Creates an empty string builder with a capacity of 16 (16 empty elements).
<code>StringBuilder(CharSequence cs)</code>	Constructs a string builder containing the same characters as the specified <code>CharSequence</code> , plus an extra 16 empty elements trailing the <code>CharSequence</code> .
<code>StringBuilder(int initCapacity)</code>	Creates an empty string builder with the specified initial capacity.
<code>StringBuilder(String s)</code>	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

```
StringBuilder sb = new StringBuilder();    // creates empty builder, capacity 16
sb.append("Greetings");                  // adds 9 character string at beginning
```

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 which is not available in `String`.

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

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.

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.

An Example

The StringDemo program would be more efficient if a StringBuilder were used instead of a String. StringDemo **reversed a palindrome**.

```
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` 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);`

We can also do it using **StringBuilder** class.

```

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.