compact1, compact2, compact3

java.lang

Class String

java.lang.Object
    java.lang.String

All Implemented Interfaces:

Serializable, CharSequence, Comparable<String>

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

The `String` class represents character strings. All string literals in Java programs, such as `"abc"`, are implemented as instances of this class.

Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings. Because String objects are immutable they can be shared. For example:

```
String str = "abc";
```

is equivalent to:

```
char data[] = {'a', 'b', 'c'};
String str = new String(data);
```

Here are some more examples of how strings can be used:

```
System.out.println("abc");
String cde = "cde";
System.out.println("abc" + cde);
String c = "abc".substring(2,3);
String d = cde.substring(1, 2);
```

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.

The Java language provides special support for the string concatenation operator (+), and for conversion of other objects to strings. String concatenation is implemented through the `StringBuilder`(or `StringBuffer`) class and its `append` method. String conversions are implemented through the method `toString`, defined by `Object` and inherited by all classes in Java. For additional information on string concatenation and conversion, see Gosling, Joy, and Steele, *The Java Language Specification*.

Unless otherwise noted, passing a `null` argument to a constructor or method in this class will cause a `NullPointerException` to be thrown.

A `String` represents a string in the UTF-16 format in which *supplementary characters* are represented by *surrogate pairs* (see the section Unicode Character Representations in the `Character` class for more information). Index values refer to `char` code units, so a supplementary character uses two positions in a `String`.

The `String` class provides methods for dealing with Unicode code points (i.e., characters), in addition to those for dealing with Unicode code units (i.e., `char` values).

Since:

`JDK1.0`

See Also:

`Object.toString()`, `StringBuffer`, `StringBuilder`, `Charset`, `Serialized Form`

Field Summary

Fields

Modifier and Type	Field	Description
`static Comparator<String>`	`CASE_INSENSITIVE_ORDER`	A Comparator that orders `String` objects as by `compareToIgnoreCase`.

Constructor Summary

Constructors

Constructor	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 using the specified **`charset`**.
`String`(byte[] ascii, int hibyte)	**Deprecated.** This method does not properly convert bytes into characters. As of JDK 1.1, the preferred way to do this is via the `String` constructors that take a **`Charset`**, charset name, or that use the platform's default charset.
`String`(byte[] bytes, int offset, int length)	Constructs a new `String` by decoding the specified subarray of bytes using the platform's default charset.
`String`(byte[] bytes, int offset, int length, `Charset` charset)	Constructs a new `String` by decoding the specified subarray of bytes using the specified **`charset`**.
`String`(byte[] ascii, int hibyte, int offset, int count)	**Deprecated.**

	This method does not properly convert bytes into characters. As of JDK 1.1, the preferred way to do this is via the `String` constructors that take a **`Charset`**, charset name, or that use the platform's default 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.
`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.
String(int[] codePoints, int offset, int count)	Allocates a new String that contains characters from a subarray of the **Unicode code point** array argument.
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.

Method Summary

All Methods Static Methods Instance Methods Concrete Methods Deprecated Methods

Modifier and Type	Method	Description
char	**charAt**(int index)	Returns the char value at the specified index.
int	**codePointAt**(int index)	Returns the character (Unicode code point) at the specified index.
int	**codePointBefore**(int index)	Returns the character (Unicode code point) before the specified index.
int	**codePointCount**(int beginIndex, int endIndex)	Returns the number of Unicode code points in the specified text range of this String.
int	**compareTo**(**String** anotherString)	Compares two strings lexicographically.
int	**compareToIgnoreCase**(**String** str)	Compares two strings lexicographically, ignoring case differences.
String	**concat**(**String** str)	Concatenates the specified string to the end of this string.

boolean	**contains**(**CharSequence** s)	Returns true if and only if this string contains the specified sequence of char values.
boolean	**contentEquals**(**CharSequence** cs)	Compares this string to the specified CharSequence.
boolean	**contentEquals**(**StringBuffer** sb)	Compares this string to the specified StringBuffer.
static **String**	**copyValueOf**(char[] data)	Equivalent to **valueOf(char[])**.
static **String**	**copyValueOf**(char[] data, int offset, int count)	Equivalent to **valueOf(char[], int, int)**.
boolean	**endsWith**(**String** suffix)	Tests if this string ends with the specified suffix.
boolean	**equals**(**Object** anObject)	Compares this string to the specified object.
boolean	**equalsIgnoreCase**(**String** anotherString)	Compares this String to another String, ignoring case considerations.
static **String**	**format**(**Locale** l, **String** format, **Object**... args)	Returns a formatted string using the specified locale, format string, and arguments.
static **String**	**format**(**String** format, **Object**... args)	Returns a formatted string using the specified format string and arguments.
byte[]	**getBytes**()	Encodes this String into a

		sequence of bytes using the platform's default charset, storing the result into a new byte array.
byte[]	**getBytes**(**Charset** charset)	Encodes this `String` into a sequence of bytes using the given **charset**, storing the result into a new byte array.
void	**getBytes**(int srcBegin, int srcEnd, byte[] dst, int dstBegin)	**Deprecated.** This method does not properly convert characters into bytes. As of JDK 1.1, the preferred way to do this is via the **getBytes()** method, which uses the platform's default charset.
byte[]	**getBytes**(**String** charsetName)	Encodes this `String` into a sequence of bytes using the named charset, storing the result into a new byte array.
void	**getChars**(int srcBegin, int srcEnd, char[] dst, int dstBegin)	Copies characters from this string into the destination character array.
int	**hashCode**()	Returns a hash code for this string.
int	**indexOf**(int ch)	Returns the index within this string of the first occurrence of the specified character.

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.
int	**indexOf**(**String** str)	Returns the index within this string of the first occurrence of the specified substring.
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.
String	**intern**()	Returns a canonical representation for the string object.
boolean	**isEmpty**()	Returns `true` if, and only if, **`length()`** is `0`.
static **String**	**join**(**CharSequence** delimiter, **CharSequence**... elements)	Returns a new String composed of copies of the `CharSequence` `elements` joined together with a copy of the specified `delimiter`.
static **String**	**join**(**CharSequence** delimiter, **Iterable**<? extends **CharSequence**> elements)	Returns a new `String` composed of copies of the `CharSequence` `elements` joined together with a copy of the specified `delimiter`.

int	**lastIndexOf**(int ch)	Returns the index within this string of the last occurrence of the specified character.
int	**lastIndexOf**(int ch, int fromIndex)	Returns the index within this string of the last occurrence of the specified character, searching backward starting at the specified index.
int	**lastIndexOf**(**String** str)	Returns the index within this string of the last occurrence of the specified substring.
int	**lastIndexOf**(**String** str, int fromIndex)	Returns the index within this string of the last occurrence of the specified substring, searching backward starting at the specified index.
int	**length**()	Returns the length of this string.
boolean	**matches**(**String** regex)	Tells whether or not this string matches the given **regular expression**.
int	**offsetByCodePoints**(int index, int codePointOffset)	Returns the index within this `String` that is offset from the given `index` by `codePointOffset` code points.

boolean	**regionMatches**(boolean ignoreCase, int toffset, **String** other, int ooffset, int len)	Tests if two string regions are equal.
boolean	**regionMatches**(int toffset, **String** other, int ooffset, int len)	Tests if two string regions are equal.
String	**replace**(char oldChar, char newChar)	Returns a 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.
String[]	**split**(**String** regex)	Splits this string around matches of the given **regular expression**.
String[]	**split**(**String** regex, int limit)	Splits this string around matches of the given **regular expression**.

boolean	**startsWith**(**String** prefix)	Tests if this string starts with the specified prefix.
boolean	**startsWith**(**String** prefix, int toffset)	Tests if the substring of this string beginning at the specified index starts with the specified prefix.
CharSequence	**subSequence**(int beginIndex, int endIndex)	Returns a character sequence that is a subsequence of this sequence.
String	**substring**(int beginIndex)	Returns a string that is a substring of this string.
String	**substring**(int beginIndex, int endIndex)	Returns a string that is a substring of this string.
char[]	**toCharArray**()	Converts this string to a new character array.
String	**toLowerCase**()	Converts all of the characters in this `String` to lower case using the rules of the default locale.
String	**toLowerCase**(**Locale** locale)	Converts all of the characters in this `String` to lower case using the rules of the given `Locale`.
String	**toString**()	This object (which is already a string!) is itself returned.
String	**toUpperCase**()	Converts all of the characters in this `String` to upper case using the

		rules of the default locale.
String	**toUpperCase**(Locale locale)	Converts all of the characters in this String to upper case using the rules of the given Locale.
String	**trim**()	Returns a string whose value is this string, with any leading and trailing whitespace removed.
static String	**valueOf**(boolean b)	Returns the string representation of the boolean argument.
static String	**valueOf**(char c)	Returns the string representation of the char argument.
static String	**valueOf**(char[] data)	Returns the string representation of the char array argument.
static String	**valueOf**(char[] data, int offset, int count)	Returns the string representation of a specific subarray of the char array argument.
static String	**valueOf**(double d)	Returns the string representation of the double argument.
static String	**valueOf**(float f)	Returns the string representation of the float argument.
static String	**valueOf**(int i)	Returns the string representation of the int argument.

static String	valueOf(long l)	Returns the string representation of the long argument.
static String	valueOf(Object obj)	Returns the string representation of the Object argument.

Methods inherited from class java.lang.Object

clone, finalize, getClass, notify, notifyAll, wait, wait, wait

Methods inherited from interface java.lang.CharSequence

chars, codePoints

Field Detail

CASE_INSENSITIVE_ORDER

```
public static final Comparator<String> CASE_INSENSITIVE_ORDER
```

A Comparator that orders `String` objects as by `compareToIgnoreCase`. This comparator is serializable.

Note that this Comparator does *not* take locale into account, and will result in an unsatisfactory ordering for certain locales. The java.text package provides *Collators* to allow locale-sensitive ordering.

Since:

1.2

See Also:

Collator.compare(String, String)

Constructor Detail

String

```
public String()
```

Initializes a newly created `String` object so that it represents an empty character sequence. Note that use of this constructor is unnecessary since Strings are immutable.

String

```
public 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. Unless an explicit copy of `original` is needed, use of this constructor is unnecessary since Strings are immutable.

Parameters:

`original - A String`

String

```
public String(char[] value)
```

Allocates a new `String` so that it represents the sequence of characters currently contained in the character array argument. The contents of the character array are copied; subsequent modification of the character array does not affect the newly created string.

Parameters:

`value - The initial value of the string`

String

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

Allocates a new `String` that contains characters from a subarray of the character array argument. The `offset` argument is the index of the first character of the subarray and the `count` argument specifies 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.

Parameters:

`value - Array that is the source of characters`

`offset - The initial offset`

`count - The length`

Throws:

`IndexOutOfBoundsException - If the offset and count arguments index characters outside the bounds of the value array`

String

```
public String(int[] codePoints,
              int offset,
              int count)
```

Allocates a new `String` that contains characters from a subarray of the Unicode code point array argument. The `offset` argument is the index of the first code point of the subarray and the `count` argument specifies the length of the subarray. The contents of

the subarray are converted to chars; subsequent modification of the int array does not affect the newly created string.

Parameters:

codePoints - Array that is the source of Unicode code points

offset - The initial offset

count - The length

Throws:

IllegalArgumentException - If any invalid Unicode code point is found in codePoints

IndexOutOfBoundsException - If the offset and count arguments index characters outside the bounds of the codePoints array

Since:

1.5

String

```
@Deprecated
public String(byte[] ascii,
                          int hibyte,
                          int offset,
                          int count)
```

Deprecated. *This method does not properly convert bytes into characters. As of JDK 1.1, the preferred way to do this is via the `String` constructors that take a `Charset`, charset name, or that use the platform's default charset.*

Allocates a new String constructed from a subarray of an array of 8-bit integer values.

The offset argument is the index of the first byte of the subarray, and the count argument specifies the length of the subarray.

Each byte in the subarray is converted to a char as specified in the method above.

Parameters:

ascii - The bytes to be converted to characters

hibyte - The top 8 bits of each 16-bit Unicode code unit

offset - The initial offset

count - The length

Throws:

IndexOutOfBoundsException - If the offset or count argument is invalid

See Also:

String(byte[], int), String(byte[], int, int, java.lang.String), String(byte[], int, int, java.nio.charset.Charset), String(byte[], int, int), String(byte[], java.lang.String), String(byte[], java.nio.charset.Charset), String(byte[])

String

```
@Deprecated
public String(byte[] ascii,
                        int hibyte)
```

Deprecated. *This method does not properly convert bytes into characters. As of JDK 1.1, the preferred way to do this is via the `String` constructors that take a `Charset`, charset name, or that use the platform's default charset.*

Allocates a new `String` containing characters constructed from an array of 8-bit integer values. Each character *c*in the resulting string is constructed from the corresponding component *b* in the byte array such that:

```
c == (char)(((hibyte & 0xff) << 8)
                        | (b & 0xff))
```

Parameters:

`ascii` - The bytes to be converted to characters

`hibyte` - The top 8 bits of each 16-bit Unicode code unit

See Also:

`String(byte[], int, int, java.lang.String)`, `String(byte[], int, int, java.nio.charset.Charset)`, `String(byte[], int, int)`, `String(byte[], java.lang.String)`, `String(byte[], java.nio.charset.Charset)`, `String(byte[])`

String

```
public String(byte[] bytes,
              int offset,
              int length,
              String charsetName)
       throws UnsupportedEncodingException
```

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.

The behavior of this constructor when the given bytes are not valid in the given charset is unspecified. The `CharsetDecoder` class should be used when more control over the decoding process is required.

Parameters:

`bytes` - The bytes to be decoded into characters

`offset` - The index of the first byte to decode

`length` - The number of bytes to decode

`charsetName` - The name of a supported `charset`

Throws:

`UnsupportedEncodingException` - If the named charset is not supported

`IndexOutOfBoundsException` - If the offset and length arguments index characters outside the bounds of the bytes array

Since:

JDK1.1

String

```
public String(byte[] bytes,
              int offset,
              int length,
              Charset charset)
```

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.

This method always replaces malformed-input and unmappable-character sequences with this charset's default replacement string. The `CharsetDecoder` class should be used when more control over the decoding process is required.

Parameters:

`bytes` - The bytes to be decoded into characters

`offset` - The index of the first byte to decode

`length` - The number of bytes to decode

`charset` - The charset to be used to decode the bytes

Throws:

`IndexOutOfBoundsException` - If the offset and length arguments index characters outside the bounds of the bytes array

Since:

1.6

String

```
public String(byte[] bytes,
              String charsetName)
       throws UnsupportedEncodingException
```

Constructs a new `String` by decoding the specified array 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 byte array.

The behavior of this constructor when the given bytes are not valid in the given charset is unspecified. The `CharsetDecoder` class should be used when more control over the decoding process is required.

Parameters:

`bytes` - The bytes to be decoded into characters

`charsetName` - The name of a supported charset

Throws:

`UnsupportedEncodingException` - If the named charset is not supported

Since:

JDK1.1

String

```
public String(byte[] bytes,
              Charset charset)
```

Constructs a new `String` by decoding the specified array 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 byte array.

This method always replaces malformed-input and unmappable-character sequences with this charset's default replacement string. The `CharsetDecoder` class should be used when more control over the decoding process is required.

Parameters:

`bytes` - The bytes to be decoded into characters

`charset` - The charset to be used to decode the bytes

Since:

1.6

String

```
public String(byte[] bytes,
              int offset,
              int length)
```

Constructs a new `String` by decoding the specified subarray of bytes using the platform's default 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.

The behavior of this constructor when the given bytes are not valid in the default charset is unspecified. The `CharsetDecoder` class should be used when more control over the decoding process is required.

Parameters:

`bytes` - The bytes to be decoded into characters

`offset` - The index of the first byte to decode

`length` - The number of bytes to decode

Throws:

`IndexOutOfBoundsException` - If the offset and the length arguments index characters outside the bounds of the bytes array

Since:

JDK1.1

String

```
public String(byte[] bytes)
```

Constructs a new `String` by decoding the specified array of bytes using the platform's default charset. The length of the new `String` is a function of the charset, and hence may not be equal to the length of the byte array.

The behavior of this constructor when the given bytes are not valid in the default charset is unspecified. The `CharsetDecoder` class should be used when more control over the decoding process is required.

Parameters:

`bytes - The bytes to be decoded into characters`

Since:

`JDK1.1`

String

```
public String(StringBuffer buffer)
```

Allocates a new string that contains the sequence of characters currently contained in the string buffer argument. The contents of the string buffer are copied; subsequent modification of the string buffer does not affect the newly created string.

Parameters:

`buffer - A StringBuffer`

String

```
public String(StringBuilder builder)
```

Allocates a new string that contains the sequence of characters currently contained in the string builder argument. The contents of the string builder are copied; subsequent modification of the string builder does not affect the newly created string.

This constructor is provided to ease migration to `StringBuilder`. Obtaining a string from a string builder via the `toString` method is likely to run faster and is generally preferred.

Parameters:

`builder - A StringBuilder`

Since:

`1.5`

Method Detail

length

```
public int length()
```

Returns the length of this string. The length is equal to the number of Unicode code units in the string.

Specified by:

`length` in interface `CharSequence`

Returns:

the length of the sequence of characters represented by this object.

isEmpty

```
public boolean isEmpty()
```

Returns `true` if, and only if, `length()` is `0`.

Returns:

`true` if `length()` is `0`, otherwise `false`

Since:

1.6

charAt

```
public char charAt(int index)
```

Returns the `char` value at the specified index. An index ranges from `0` to `length() - 1`. The first `char` value of the sequence is at index `0`, the next at index `1`, and so on, as for array indexing.

If the `char` value specified by the index is a surrogate, the surrogate value is returned.

Specified by:

`charAt` in interface `CharSequence`

Parameters:

`index` - the index of the char value.

Returns:

the char value at the specified index of this string. The first char value is at index 0.

Throws:

`IndexOutOfBoundsException` - if the index argument is negative or not less than the length of this string.

codePointAt

```
public int codePointAt(int index)
```

Returns the character (Unicode code point) at the specified index. The index refers to `char` values (Unicode code units) and ranges from `0` to `length() - 1`.

If the `char` value specified at the given index is in the high-surrogate range, the following index is less than the length of this `String`, and the `char` value at the following index is in the low-surrogate range, then the supplementary code point corresponding to this surrogate pair is returned. Otherwise, the `char` value at the given index is returned.

Parameters:

index - the index to the char values

Returns:

the code point value of the character at the index

Throws:

IndexOutOfBoundsException - if the index argument is negative or not less than the length of this string.

Since:

1.5

codePointBefore

```
public int codePointBefore(int index)
```

Returns the character (Unicode code point) before the specified index. The index refers to char values (Unicode code units) and ranges from 1 to length.

If the char value at (index - 1) is in the low-surrogate range, (index - 2) is not negative, and the char value at (index - 2) is in the high-surrogate range, then the supplementary code point value of the surrogate pair is returned. If the char value at index - 1 is an unpaired low-surrogate or a high-surrogate, the surrogate value is returned.

Parameters:

index - the index following the code point that should be returned

Returns:

the Unicode code point value before the given index.

Throws:

IndexOutOfBoundsException - if the index argument is less than 1 or greater than the length of this string.

Since:

1.5

codePointCount

```
public int codePointCount(int beginIndex,
                          int endIndex)
```

Returns the number of Unicode code points in the specified text range of this String. The text range begins at the specified beginIndex and extends to the char at index endIndex - 1. Thus the length (in chars) of the text range is endIndex-beginIndex. Unpaired surrogates within the text range count as one code point each.

Parameters:

beginIndex - the index to the first char of the text range.

endIndex - the index after the last char of the text range.

Returns:

the number of Unicode code points in the specified text range

Throws:

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

Since:

1.5

offsetByCodePoints

```
public int offsetByCodePoints(int index,
                              int codePointOffset)
```

Returns the index within this String that is offset from the given index by codePointOffset code points. Unpaired surrogates within the text range given by index and codePointOffset count as one code point each.

Parameters:

index - the index to be offset

codePointOffset - the offset in code points

Returns:

the index within this String

Throws:

IndexOutOfBoundsException - if index is negative or larger then the length of this String, or if codePointOffset is positive and the substring starting with index has fewer than codePointOffset code points, or if codePointOffset is negative and the substring before index has fewer than the absolute value of codePointOffset code points.

Since:

1.5

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.
- srcBegin is greater than srcEnd
- srcEnd is greater than the length of this string
- dstBegin is negative
- dstBegin+(srcEnd-srcBegin) is larger than dst.length

getBytes

```
@Deprecated
public void getBytes(int srcBegin,
                                 int srcEnd,
                                 byte[] dst,
                                 int dstBegin)
```

Deprecated. *This method does not properly convert characters into bytes. As of JDK 1.1, the preferred way to do this is via the `getBytes()` method, which uses the platform's default charset.*

Copies characters from this string into the destination byte array. Each byte receives the 8 low-order bits of the corresponding character. The eight high-order bits of each character are not copied and do not participate in the transfer in any way.

The first character to be copied is at index `srcBegin`; the last character to be copied is at index `srcEnd-1`. The total number of characters to be copied is `srcEnd-srcBegin`. The characters, converted to bytes, 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
- srcBegin is greater than srcEnd
- srcEnd is greater than the length of this String
- dstBegin is negative
- dstBegin+(srcEnd-srcBegin) is larger than dst.length

getBytes

```
public byte[] getBytes(String charsetName)
                throws UnsupportedEncodingException
```

Encodes this String into a sequence of bytes using the named charset, storing the result into a new byte array.

The behavior of this method when this string cannot be encoded in the given charset is unspecified. The CharsetEncoder class should be used when more control over the encoding process is required.

Parameters:

charsetName - The name of a supported charset

Returns:

The resultant byte array

Throws:

UnsupportedEncodingException - If the named charset is not supported

Since:

JDK1.1

getBytes

```
public byte[] getBytes(Charset charset)
```

Encodes this String into a sequence of bytes using the given charset, storing the result into a new byte array.

This method always replaces malformed-input and unmappable-character sequences with this charset's default replacement byte array. The CharsetEncoder class should be used when more control over the encoding process is required.

Parameters:

charset - The Charset to be used to encode the String

Returns:

The resultant byte array

Since:

1.6

getBytes

```
public byte[] getBytes()
```

Encodes this String into a sequence of bytes using the platform's default charset, storing the result into a new byte array.

The behavior of this method when this string cannot be encoded in the default charset is unspecified. The CharsetEncoder class should be used when more control over the encoding process is required.

Returns:

The resultant byte array

Since:

JDK1.1

equals

```
public boolean equals(Object anObject)
```

Compares this string to the specified object. The result is `true` if and only if the argument is not `null` and is a `String` object that represents the same sequence of characters as this object.

Overrides:

`equals` in class `Object`

Parameters:

`anObject` - The object to compare this String against

Returns:

`true if the given object represents a String equivalent to this string, false otherwise`

See Also:

`compareTo(String)`, `equalsIgnoreCase(String)`

contentEquals

```
public boolean contentEquals(StringBuffer sb)
```

Compares this string to the specified `StringBuffer`. The result is `true` if and only if this `String` represents the same sequence of characters as the specified `StringBuffer`. This method synchronizes on the `StringBuffer`.

Parameters:

`sb` - The StringBuffer to compare this String against

Returns:

`true if this String represents the same sequence of characters as the specified StringBuffer, false otherwise`

Since:

1.4

contentEquals

```
public boolean contentEquals(CharSequence cs)
```

Compares this string to the specified `CharSequence`. The result is `true` if and only if this `String` represents the same sequence of char values as the specified sequence. Note that if the `CharSequence` is a `StringBuffer` then the method synchronizes on it.

Parameters:

`cs` - The sequence to compare this String against

Returns:

true if this String represents the same sequence of char values as the specified sequence, false otherwise

Since:

1.5

equalsIgnoreCase

```
public boolean equalsIgnoreCase(String anotherString)
```

Compares this String to another String, ignoring case considerations. Two strings are considered equal ignoring case if they are of the same length and corresponding characters in the two strings are equal ignoring case.

Two characters c1 and c2 are considered the same ignoring case if at least one of the following is true:

- The two characters are the same (as compared by the == operator)
- Applying the method Character.toUpperCase(char) to each character produces the same result
- Applying the method Character.toLowerCase(char) to each character produces the same result

Parameters:

anotherString - The String to compare this String against

Returns:

true if the argument is not null and it represents an equivalent String ignoring case; false otherwise

See Also:

equals(Object)

compareTo

```
public int compareTo(String anotherString)
```

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. The result is a negative integer if this String object lexicographically precedes the argument string. The result is a positive integer if this String object lexicographically follows the argument string. The result is zero if the strings are equal; compareTo returns 0 exactly when the equals(Object) method would return true.

This is the 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, as determined by using the < 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()
```

Specified by:

`compareTo` in interface `Comparable<String>`

Parameters:

`anotherString - the String to be compared.`

Returns:

`the value 0 if the argument string is equal to this string; a value less than 0 if this string is lexicographically less than the string argument; and a value greater than 0 if this string is lexicographically greater than the string argument.`

compareToIgnoreCase

```
public int compareToIgnoreCase(String str)
```

Compares two strings lexicographically, ignoring case differences. This method returns an integer whose sign is that of calling `compareTo` with normalized versions of the strings where case differences have been eliminated by calling `Character.toLowerCase(Character.toUpperCase(character))` on each character.

Note that this method does *not* take locale into account, and will result in an unsatisfactory ordering for certain locales. The java.text package provides *collators* to allow locale-sensitive ordering.

Parameters:

`str - the String to be compared.`

Returns:

`a negative integer, zero, or a positive integer as the specified String is greater than, equal to, or less than this String, ignoring case considerations.`

Since:

`1.2`

See Also:

`Collator.compare(String, String)`

regionMatches

```
public boolean regionMatches(int toffset,
                             String other,
                             int ooffset,
                             int len)
```

Tests if two string regions are equal.

A substring of this `String` object is compared to a substring of the argument other. The result is true if these substrings represent identical character sequences. The substring of this `String` object to be compared begins at index `toffset` and has length `len`. The substring of other to be compared begins at index `ooffset` and has length `len`. The result is `false` if and only if at least one of the following is true:

- `toffset` is negative.
- `ooffset` is negative.
- `toffset+len` is greater than the length of this `String` object.
- `ooffset+len` is greater than the length of the other argument.
- There is some nonnegative integer *k* less than `len` such that:
  `this.charAt(toffset + k) != other.charAt(ooffset + k)`

Parameters:

`toffset - the starting offset of the subregion in this string.`

`other - the string argument.`

`ooffset - the starting offset of the subregion in the string argument.`

`len - the number of characters to compare.`

Returns:

`true if the specified subregion of this string exactly matches the specified subregion of the string argument; false otherwise.`

regionMatches

```
public boolean regionMatches(boolean ignoreCase,
                             int toffset,
                             String other,
                             int ooffset,
                             int len)
```

Tests if two string regions are equal.

A substring of this `String` object is compared to a substring of the argument `other`. The result is `true` if these substrings represent character sequences that are the same, ignoring case if and only if `ignoreCase` is true. The substring of this `String` object to be compared begins at index `toffset` and has length `len`. The substring of `other` to be compared begins at index `ooffset` and has length `len`. The result is `false` if and only if at least one of the following is true:

- `toffset` is negative.
- `ooffset` is negative.
- `toffset+len` is greater than the length of this `String` object.
- `ooffset+len` is greater than the length of the other argument.
- `ignoreCase` is `false` and there is some nonnegative integer *k* less than `len` such that:

```
this.charAt(toffset+k) != other.charAt(ooffset+k)
```

- `ignoreCase` is `true` and there is some nonnegative integer *k* less than `len` such that:

```
Character.toLowerCase(this.charAt(toffset+k)) !=
    Character.toLowerCase(other.charAt(ooffset+k))
```

and:

```
Character.toUpperCase(this.charAt(toffset+k)) !=
        Character.toUpperCase(other.charAt(ooffset+k))
```

Parameters:

ignoreCase - if true, ignore case when comparing characters.

toffset - the starting offset of the subregion in this string.

other - the string argument.

ooffset - the starting offset of the subregion in the string argument.

len - the number of characters to compare.

Returns:

true if the specified subregion of this string matches the specified subregion of the string argument; false otherwise. Whether the matching is exact or case insensitive depends on the ignoreCase argument.

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

```
this.substring(toffset).startsWith(prefix)
```

startsWith

```
public boolean startsWith(String prefix)
```

Tests if this string starts with the specified prefix.

Parameters:

prefix - the prefix.

Returns:

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

Since:

`1. 0`

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.`

hashCode

```
public int hashCode()
```

Returns a hash code for this string. The hash code for a `String` object is computed as

```
s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]
```

using `int` arithmetic, where `s[i]` is the *i*th character of the string, `n` is the length of the string, and ^ indicates exponentiation. (The hash value of the empty string is zero.)

Overrides:

`hashCode in class Object`

Returns:

`a hash code value for this object.`

See Also:

`Object.equals(java.lang.Object), System.identityHashCode(java.lang.Object)`

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
```

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

Parameters:

ch - a character (Unicode code point).

Returns:

the index of the first occurrence of the character in the character sequence represented by this object, or -1 if the character does not occur.

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.

If a character with value ch occurs in the character sequence represented by this String object at an index no smaller than fromIndex, then the index 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)  && (k >= fromIndex)
```

is true. For other values of ch, it is the smallest value *k* such that:

```
(this.codePointAt(k) == ch)  && (k >= fromIndex)
```

is true. In either case, if no such character occurs in this string at or after position fromIndex, then -1 is returned.

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).

Parameters:

ch - a character (Unicode code point).

fromIndex - the index to start the search from.

Returns:

the index of the first occurrence of the character in the character sequence represented by this object that is greater than or equal to fromIndex, or -1 if the character does not occur.

lastIndexOf

```
public int lastIndexOf(int ch)
```

Returns the index within this string of the last occurrence of the specified character. For values of ch in the range from 0 to 0xFFFF (inclusive), the index (in Unicode code units) returned is the largest value *k* such that:

```
this.charAt(k) == ch
```

is true. For other values of ch, it is the largest value *k* such that:

```
this.codePointAt(k) == ch
```

is true. In either case, if no such character occurs in this string, then -1 is returned. The String is searched backwards starting at the last character.

Parameters:

ch - a character (Unicode code point).

Returns:

the index of the last occurrence of the character in the character sequence represented by this object, or -1 if the character does not occur.

lastIndexOf

```
public int lastIndexOf(int ch,
                       int fromIndex)
```

Returns the index within this string of the last occurrence of the specified character, searching backward starting at the specified index. For values of ch in the range from 0 to 0xFFFF (inclusive), the index returned is the largest value *k* such that:

```
(this.charAt(k) == ch)  && (k <= fromIndex)
```

is true. For other values of ch, it is the largest value *k* such that:

```
(this.codePointAt(k) == ch)  && (k <= fromIndex)
```

is true. In either case, if no such character occurs in this string at or before position fromIndex, then -1 is returned.

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

Parameters:

ch - a character (Unicode code point).

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

Returns:

the index of the last occurrence of the character in the character sequence represented by this object that is less than or equal to fromIndex, or -1 if the character does not occur before that point.

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.

Parameters:

str - the substring to search for.

fromIndex - the index from which to start the search.

Returns:

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

lastIndexOf

```
public int lastIndexOf(String str)
```

Returns the index within this string of the last occurrence of the specified substring. The last occurrence of the empty string "" is considered to occur at the index value `this.length()`.

The returned index is the largest 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 last occurrence of the specified substring, or -1 if there is no such occurrence.`

lastIndexOf

```
public int lastIndexOf(String str,
                       int fromIndex)
```

Returns the index within this string of the last occurrence of the specified substring, searching backward starting at the specified index.

The returned index is the largest value *k* for which:

```
k  <= fromIndex  && this.startsWith(str, k)
```

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

Parameters:

`str - the substring to search for.`

`fromIndex - the index to start the search from.`

Returns:

`the index of the last occurrence of the specified substring, searching backward from the specified index, or -1 if there is no such occurrence.`

substring

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

Returns a string that is a substring of this string. The substring begins with the character at the specified index and extends to the end of this string.

Examples:

```
"unhappy".substring(2) returns "happy"
"Harbison".substring(3) returns "bison"
"emptiness".substring(9) returns "" (an empty string)
```

Parameters:

beginIndex - the beginning index, inclusive.

Returns:

the specified substring.

Throws:

IndexOutOfBoundsException - if beginIndex is negative or larger than the length of this String object.

substring

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

Returns a 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.

Returns:

the specified substring.

Throws:

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

subSequence

```
public CharSequence subSequence(int beginIndex,
                                int endIndex)
```

Returns a character sequence that is a subsequence of this sequence.

An invocation of this method of the form

```
str.subSequence(begin, end)
```

behaves in exactly the same way as the invocation

```
str.substring(begin, end)
```

Specified by:

subSequence in interface CharSequence

API Note:

This method is defined so that the String class can implement the CharSequence interface.

Parameters:

beginIndex - the begin index, inclusive.

endIndex - the end index, exclusive.

Returns:

the specified subsequence.

Throws:

IndexOutOfBoundsException - if beginIndex or endIndex is negative, if endIndex is greater than length(), or if beginIndex is greater than endIndex

Since:

1.4

concat

```
public String concat(String str)
```

Concatenates the specified string to the end of this string.

If the length of the argument string is `0`, then this `String` object is returned. Otherwise, a `String` object is returned that represents a character sequence that is the concatenation of the character sequence represented by this `String` object and the character sequence represented by the argument string.

Examples:

```
"cares".concat("s") returns "caress"
"to".concat("get").concat("her") returns "together"
```

Parameters:

str - the String that is concatenated to the end of this String.

Returns:

a string that represents the concatenation of this object's characters followed by the string argument's characters.

replace

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

Returns a 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 `String` object is returned 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)
```

Parameters:

`regex - the regular expression to which this string is to be matched`

Returns:

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

Throws:

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

Since:

`1.4`

See Also:

`Pattern`

contains

```
public boolean contains(CharSequence s)
```

Returns true if and only if this string contains the specified sequence of char values.

Parameters:

`s - the sequence to search for`

Returns:

`true if this string contains s, false otherwise`

Since:

`1.5`

replaceFirst

```
public String replaceFirst(String regex,
                           String replacement)
```

Replaces the first substring of this string that matches the given regular expression with the given replacement.

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

```
Pattern.compile(regex).matcher(str).replaceFirst(repl)
```

Note that backslashes (\) and dollar signs ($) in the replacement string may cause the results to be different than if it were being treated as a literal replacement string; see `Matcher.replaceFirst(java.lang.String)`. Use `Matcher.quoteReplacement(java.lang.String)` to suppress the special meaning of these characters, if desired.

Parameters:

`regex - the regular expression to which this string is to be matched`

`replacement - the string to be substituted for the first match`

Returns:

`The resulting String`

Throws:

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

Since:

`1.4`

See Also:

`Pattern`

replaceAll

```
public String replaceAll(String regex,
                         String replacement)
```

Replaces each substring of this string that matches the given regular expression with the given replacement.

An invocation of this method of the form *str*.replaceAll(*regex*, *repl*) yields exactly the same result as the expression

```
Pattern.compile(regex).matcher(str).replaceAll(repl)
```

Note that backslashes (\) and dollar signs ($) in the replacement string may cause the results to be different than if it were being treated as a literal replacement string; see Matcher.replaceAll. Use Matcher.quoteReplacement(java.lang.String) to suppress the special meaning of these characters, if desired.

Parameters:

regex - the regular expression to which this string is to be matched

replacement - the string to be substituted for each match

Returns:

The resulting String

Throws:

PatternSyntaxException - if the regular expression's syntax is invalid

Since:

1.4

See Also:

Pattern

replace

```
public String replace(CharSequence target,
                      CharSequence replacement)
```

Replaces each substring of this string that matches the literal target sequence with the specified literal replacement sequence. The replacement proceeds from the beginning of the string to the end, for example, replacing "aa" with "b" in the string "aaa" will result in "ba" rather than "ab".

Parameters:

target - The sequence of char values to be replaced

replacement - The replacement sequence of char values

Returns:

The resulting string

Since:

1.5

split

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

Splits this string around matches of the given regular expression.

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. The substrings in the array are in the order in which they occur in this string. If the expression does not match any part of the input then the resulting array has just one element, namely this string.

When there is a positive-width match at the beginning of this string then an empty leading substring is included at the beginning of the resulting array. A zero-width match at the beginning however never produces such empty leading substring.

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" }
:	5	{ "boo", "and", "foo" }
:	-2	{ "boo", "and", "foo" }
o	5	{ "b", "", ":and:f", "", "" }
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`

Returns:

`the array of strings computed by splitting this string around matches of the given regular expression`

Throws:

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

Since:

`1.4`

See Also:

`Pattern`

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" }

Parameters:

`regex - the delimiting regular expression`

Returns:

`the array of strings computed by splitting this string around matches of the given regular expression`

Throws:

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

Since:

`1.4`

See Also:

`Pattern`

join

```
public static String join(CharSequence delimiter,
                          CharSequence... elements)
```

Returns a new String composed of copies of the `CharSequence elements` joined together with a copy of the specified `delimiter`.

For example,

```
String message = String.join("-", "Java", "is", "cool");
// message returned is: "Java-is-cool"
```

Note that if an element is null, then `"null"` is added.

Parameters:

`delimiter - the delimiter that separates each element`

`elements - the elements to join together.`

Returns:

`a new String that is composed of the elements separated by the delimiter`

Throws:

`NullPointerException - If delimiter or elements is null`

Since:

`1.8`

See Also:

StringJoiner

join

```
public static String join(CharSequence delimiter,
                          Iterable<? extends CharSequence> elements)
```

Returns a new String composed of copies of the CharSequence elements joined together with a copy of the specified delimiter.

For example,

```
List<String> strings = new LinkedList<>();
strings.add("Java");strings.add("is");
strings.add("cool");
String message = String.join(" ", strings);
//message returned is: "Java is cool"

Set<String> strings = new LinkedHashSet<>();
strings.add("Java"); strings.add("is");
strings.add("very"); strings.add("cool");
String message = String.join("-", strings);
//message returned is: "Java-is-very-cool"
```

Note that if an individual element is null, then "null" is added.

Parameters:

delimiter - a sequence of characters that is used to separate each of the elements in the resulting String

elements - an Iterable that will have its elements joined together.

Returns:

a new String that is composed from the elements argument

Throws:

NullPointerException - If delimiter or elements is null

Since:

1.8

See Also:

join(CharSequence,CharSequence...), StringJoiner

toLowerCase

```
public String toLowerCase(Locale locale)
```

Converts all of the characters in this String to lower case using the rules of the given Locale. Case mapping is based on the Unicode Standard version specified by the Character class. Since case mappings are not always 1:1 char mappings, the resulting String may be a different length than the original String.

Examples of lowercase mappings are in the following table:

Language Code of Locale	Upper Case	Lower Case	Description
tr (Turkish)	\u0130	\u0069	capital letter I with dot above -> small letter i
tr (Turkish)	\u0049	\u0131	capital letter I -> small letter dotless i
(all)	French Fries	french fries	lowercased all chars in String
(all)	ΙΧ ΘΥ Σ	ιχ θυ ς	lowercased all chars in String

Parameters:

`locale - use the case transformation rules for this locale`

Returns:

`the String, converted to lowercase.`

Since:

`1.1`

See Also:

`toLowerCase()`, `toUpperCase()`, `toUpperCase(Locale)`

toLowerCase

```
public String toLowerCase()
```

Converts all of the characters in this `String` to lower case using the rules of the default locale. This is equivalent to calling `toLowerCase(Locale.getDefault())`.

Note: This method is locale sensitive, and may produce unexpected results if used for strings that are intended to be interpreted locale independently. Examples are programming language identifiers, protocol keys, and HTML tags. For instance, `"TITLE".toLowerCase()` in a Turkish locale returns `"t\u0131tle"`, where '\u0131' is the LATIN SMALL LETTER DOTLESS I character. To obtain correct results for locale insensitive strings, use `toLowerCase(Locale.ROOT)`.

Returns:

`the String, converted to lowercase.`

See Also:

`toLowerCase(Locale)`

toUpperCase

```
public String toUpperCase(Locale locale)
```

Converts all of the characters in this `String` to upper case using the rules of the given `Locale`. Case mapping is based on the Unicode Standard version specified by the `Character` class. Since case mappings are not always 1:1 char mappings, the resulting `String` may be a different length than the original `String`.

Examples of locale-sensitive and 1:M case mappings are in the following table.

Language Code of Locale	Lower Case	Upper Case	Description
tr (Turkish)	\u0069	\u0130	small letter i -> capital letter I with dot above
tr (Turkish)	\u0131	\u0049	small letter dotless i -> capital letter I
(all)	\u00df	\u0053 \u0053	small letter sharp s -> two letters: SS
(all)	Fahrvergnügen	FAHRVERGNÜGEN	

Parameters:

`locale - use the case transformation rules for this locale`

Returns:

`the String, converted to uppercase.`

Since:

`1.1`

See Also:

`toUpperCase()`, `toLowerCase()`, `toLowerCase(Locale)`

toUpperCase

```
public String toUpperCase()
```

Converts all of the characters in this `String` to upper case using the rules of the default locale. This method is equivalent to `toUpperCase(Locale.getDefault())`.

Note: This method is locale sensitive, and may produce unexpected results if used for strings that are intended to be interpreted locale independently. Examples are programming language identifiers, protocol keys, and HTML tags. For instance, `"title".toUpperCase()` in a Turkish locale returns `"T\u0130TLE"`, where '\u0130' is the LATIN CAPITAL LETTER I WITH DOT ABOVE character. To obtain correct results for locale insensitive strings, use `toUpperCase(Locale.ROOT)`.

Returns:

`the String, converted to uppercase.`

See Also:

`toUpperCase(Locale)`

trim

```
public String trim()
```

Returns a string whose value is this string, with any leading and trailing whitespace removed.

If this `String` object represents an empty character sequence, or the first and last characters of character sequence represented by this `String` object both have codes

greater than `'\u0020'` (the space character), then a reference to this `String` object is returned.

Otherwise, if there is no character with a code greater than `'\u0020'` in the string, then a `String` object representing an empty string is returned.

Otherwise, let *k* be the index of the first character in the string whose code is greater than `'\u0020'`, and let *m* be the index of the last character in the string whose code is greater than `'\u0020'`. A `String` object is returned, representing the substring of this string that begins with the character at index *k* and ends with the character at index *m*-that is, the result of `this.substring(k, m + 1)`.

This method may be used to trim whitespace (as defined above) from the beginning and end of a string.

Returns:

`A string whose value is this string, with any leading and trailing white space removed, or this string if it has no leading or trailing white space.`

toString

```
public String toString()
```

This object (which is already a string!) is itself returned.

Specified by:

`toString` in interface `CharSequence`

Overrides:

`toString` in class `Object`

Returns:

`the string itself.`

toCharArray

```
public char[] toCharArray()
```

Converts this string to a new character array.

Returns:

`a newly allocated character array whose length is the length of this string and whose contents are initialized to contain the character sequence represented by this string.`

format

```
public static String format(String format,
                            Object... args)
```

Returns a formatted string using the specified format string and arguments.

The locale always used is the one returned by `Locale.getDefault()`.

Parameters:

format - A format string

args - Arguments referenced by the format specifiers in the format string. If there are more arguments than format specifiers, the extra arguments are ignored. The number of arguments is variable and may be zero. The maximum number of arguments is limited by the maximum dimension of a Java array as defined by *The Java™ Virtual Machine Specification*. The behaviour on a null argument depends on the conversion.

Returns:

A formatted string

Throws:

IllegalFormatException - If a format string contains an illegal syntax, a format specifier that is incompatible with the given arguments, insufficient arguments given the format string, or other illegal conditions. For specification of all possible formatting errors, see the Details section of the formatter class specification.

Since:

1.5

See Also:

Formatter

format

```
public static String format(Locale l,
                            String format,
                            Object... args)
```

Returns a formatted string using the specified locale, format string, and arguments.

Parameters:

l - The locale to apply during formatting. If l is null then no localization is applied.

format - A format string

args - Arguments referenced by the format specifiers in the format string. If there are more arguments than format specifiers, the extra arguments are ignored. The number of arguments is variable and may be zero. The maximum number of arguments is limited by the maximum dimension of a Java array as defined by *The Java™ Virtual Machine Specification*. The behaviour on a null argument depends on the conversion.

Returns:

A formatted string

Throws:

IllegalFormatException - If a format string contains an illegal syntax, a format specifier that is incompatible with the given arguments, insufficient arguments given the format string, or other illegal conditions. For specification of all possible formatting errors, see the Details section of the formatter class specification

Since:

1.5

See Also:

Formatter

valueOf

```
public static String valueOf(Object obj)
```

Returns the string representation of the `Object` argument.

Parameters:

`obj` - an `Object`.

Returns:

if the argument is null, then a string equal to "null"; otherwise, the value of obj.toString() is returned.

See Also:

Object.toString()

valueOf

```
public static String valueOf(char[] data)
```

Returns the string representation of the `char` array argument. The contents of the character array are copied; subsequent modification of the character array does not affect the returned string.

Parameters:

`data` - the character array.

Returns:

a String that contains the characters of the character array.

valueOf

```
public static String valueOf(char[] data,
                             int offset,
                             int count)
```

Returns the string representation of a specific subarray of the `char` array argument.

The `offset` argument is the index of the first character of the subarray. The `count` argument specifies the length of the subarray. The contents of the subarray are copied; subsequent modification of the character array does not affect the returned string.

Parameters:

`data` - the character array.

`offset` - initial offset of the subarray.

`count` - length of the subarray.

Returns:

a String that contains the characters of the specified subarray of the character array.

Throws:

IndexOutOfBoundsException - if offset is negative, or count is negative, or offset+count is larger than data.length.

copyValueOf

```
public static String copyValueOf(char[] data,
                                 int offset,
                                 int count)
```

Equivalent to valueOf(char[], int, int).

Parameters:

data - the character array.

offset - initial offset of the subarray.

count - length of the subarray.

Returns:

a String that contains the characters of the specified subarray of the character array.

Throws:

IndexOutOfBoundsException - if offset is negative, or count is negative, or offset+count is larger than data.length.

copyValueOf

```
public static String copyValueOf(char[] data)
```

Equivalent to valueOf(char[]).

Parameters:

data - the character array.

Returns:

a String that contains the characters of the character array.

valueOf

```
public static String valueOf(boolean b)
```

Returns the string representation of the boolean argument.

Parameters:

b - a boolean.

Returns:

if the argument is true, a string equal to "true" is returned; otherwise, a string equal to "false" is returned.

valueOf

```
public static String valueOf(char c)
```

Returns the string representation of the `char` argument.

Parameters:

`c - a char.`

Returns:

`a string of length 1 containing as its single character the argument c.`

valueOf

```
public static String valueOf(int i)
```

Returns the string representation of the `int` argument.

The representation is exactly the one returned by the `Integer.toString` method of one argument.

Parameters:

`i - an int.`

Returns:

`a string representation of the int argument.`

See Also:

`Integer.toString(int, int)`

valueOf

```
public static String valueOf(long l)
```

Returns the string representation of the `long` argument.

The representation is exactly the one returned by the `Long.toString` method of one argument.

Parameters:

`l - a long.`

Returns:

`a string representation of the long argument.`

See Also:

`Long.toString(long)`

valueOf

```
public static String valueOf(float f)
```

Returns the string representation of the `float` argument.

The representation is exactly the one returned by the `Float.toString` method of one argument.

Parameters:

`f - a float.`

Returns:

`a string representation of the float argument.`

See Also:

`Float.toString(float)`

valueOf

```
public static String valueOf(double d)
```

Returns the string representation of the `double` argument.

The representation is exactly the one returned by the `Double.toString` method of one argument.

Parameters:

`d - a double.`

Returns:

`a string representation of the double argument.`

See Also:

`Double.toString(double)`

intern

```
public String intern()
```

Returns a canonical representation for the string object.

A pool of strings, initially empty, is maintained privately by the class `String`.

When the intern method is invoked, if the pool already contains a string equal to this `String` object as determined by the `equals(Object)` method, then the string from the pool is returned. Otherwise, this `String` object is added to the pool and a reference to this `String` object is returned.

It follows that for any two strings `s` and `t`, `s.intern() == t.intern()` is `true` if and only if `s.equals(t)` is `true`.

All literal strings and string-valued constant expressions are interned. String literals are defined in section 3.10.5 of the *The Java™ Language Specification*.

Returns:

`a string that has the same contents as this string, but is guaranteed to be from a pool of unique strings.`

Submit a bug or feature

For further API reference and developer documentation, see Java SE Documentation. That documentation contains more detailed, developer-targeted descriptions, with conceptual overviews, definitions of terms, workarounds, and working code examples.