OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

Java™ Platform
Standard Ed. 8

PREV CLASS NEXT CLASS FRAMES NO FRAMES ALL CLASSES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

compact1, compact2, compact3

java.util

Class ArrayList<E>

```
java.lang.Object
    java.util.AbstractCollection<E>
        java.util.AbstractList<E>
            java.util.ArrayList<E>
```

All Implemented Interfaces:

`Serializable`, `Cloneable`, `Iterable<E>`, `Collection<E>`, `List<E>`, `RandomAccess`

Direct Known Subclasses:

`AttributeList`, `RoleList`, `RoleUnresolvedList`

```
public class ArrayList<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

Resizable-array implementation of the `List` interface. Implements all optional list operations, and permits all elements, including `null`. In addition to implementing the `List` interface, this class provides methods to manipulate the size of the array that is used internally to store the list. (This class is roughly equivalent to `Vector`, except that it is unsynchronized.)

The `size`, `isEmpty`, `get`, `set`, `iterator`, and `listIterator` operations run in constant time. The `add` operation runs in *amortized constant time*, that is, adding n elements requires O(n) time. All of the other operations run in linear time (roughly speaking). The constant factor is low compared to that for the `LinkedList` implementation.

Each `ArrayList` instance has a *capacity*. The capacity is the size of the array used to store the elements in the list. It is always at least as large as the list size. As elements are added to an ArrayList, its capacity grows automatically. The details of the growth policy are not specified beyond the fact that adding an element has constant amortized time cost.

An application can increase the capacity of an `ArrayList` instance before adding a large number of elements using the `ensureCapacity` operation. This may reduce the amount of incremental reallocation.

Note that this implementation is not synchronized. If multiple threads access an

`ArrayList` instance concurrently, and at least one of the threads modifies the list structurally, it *must* be synchronized externally. (A structural modification is any operation that adds or deletes one or more elements, or explicitly resizes the backing array; merely setting the value of an element is not a structural modification.) This is typically accomplished by synchronizing on some object that naturally encapsulates the list. If no such object exists, the list should be "wrapped" using the `Collections.synchronizedList` method. This is best done at creation time, to prevent accidental unsynchronized access to the list:

```
   List list = Collections.synchronizedList(new ArrayList(...));
```

The iterators returned by this class's `iterator` and `listIterator` methods are *fail-fast*: if the list is structurally modified at any time after the iterator is created, in any way except through the iterator's own `remove` or `add` methods, the iterator will throw a `ConcurrentModificationException`. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.

Note that the fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw `ConcurrentModificationException` on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness: *the fail-fast behavior of iterators should be used only to detect bugs.*

This class is a member of the Java Collections Framework.

Since:

`1.2`

See Also:

`Collection`, `List`, `LinkedList`, `Vector`, `Serialized Form`

Field Summary

Fields inherited from class java.util.AbstractList

`modCount`

Constructor Summary

Constructors

Constructor and Description

`ArrayList()`

Constructs an empty list with an initial capacity of ten.

`ArrayList(Collection<? extends E> c)`

Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.

`ArrayList(int initialCapacity)`

Constructs an empty list with the specified initial capacity.

Method Summary

All Methods **Instance Methods** **Concrete Methods**

Modifier and Type	Method and Description
`boolean`	`add(E e)` Appends the specified element to the end of this list.
`void`	`add(int index, E element)` Inserts the specified element at the specified position in this list.
`boolean`	`addAll(Collection<? extends E> c)` Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's Iterator.
`boolean`	`addAll(int index, Collection<? extends E> c)` Inserts all of the elements in the specified collection into this list, starting at the specified position.
`void`	`clear()` Removes all of the elements from this list.
`Object`	`clone()` Returns a shallow copy of this `ArrayList` instance.
`boolean`	`contains(Object o)` Returns `true` if this list contains the specified element.
`void`	`ensureCapacity(int minCapacity)` Increases the capacity of this `ArrayList` instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument.

`void`	`forEach(Consumer<? super E> action)` Performs the given action for each element of the `Iterable` until all elements have been processed or the action throws an exception.
`E`	`get(int index)` Returns the element at the specified position in this list.
`int`	`indexOf(Object o)` Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
`boolean`	`isEmpty()` Returns `true` if this list contains no elements.
`Iterator<E>`	`iterator()` Returns an iterator over the elements in this list in proper sequence.
`int`	`lastIndexOf(Object o)` Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
`ListIterator<E>`	`listIterator()` Returns a list iterator over the elements in this list (in proper sequence).
`ListIterator<E>`	`listIterator(int index)` Returns a list iterator over the elements in this list (in proper sequence), starting at the specified position in the list.
`E`	`remove(int index)` Removes the element at the specified position in this list.
`boolean`	`remove(Object o)` Removes the first occurrence of the specified element from this list, if it is present.
`boolean`	`removeAll(Collection<?> c)` Removes from this list all of its elements that are contained in the specified collection.
`boolean`	`removeIf(Predicate<? super E> filter)` Removes all of the elements of this collection that satisfy

	the given predicate.
`protected void`	**`removeRange`**`(int fromIndex, int toIndex)` Removes from this list all of the elements whose index is between `fromIndex`, inclusive, and `toIndex`, exclusive.
`void`	**`replaceAll`**`(`**`UnaryOperator`**`<`**`E`**`> operator)` Replaces each element of this list with the result of applying the operator to that element.
`boolean`	**`retainAll`**`(`**`Collection`**`<?> c)` Retains only the elements in this list that are contained in the specified collection.
`E`	**`set`**`(int index, `**`E`**` element)` Replaces the element at the specified position in this list with the specified element.
`int`	**`size`**`()` Returns the number of elements in this list.
`void`	**`sort`**`(`**`Comparator`**`<? super `**`E`**`> c)` Sorts this list according to the order induced by the specified **`Comparator`**.
`Spliterator``<`**`E`**`>`	**`spliterator`**`()` Creates a ***late-binding*** and *fail-fast* **`Spliterator`** over the elements in this list.
`List``<`**`E`**`>`	**`subList`**`(int fromIndex, int toIndex)` Returns a view of the portion of this list between the specified `fromIndex`, inclusive, and `toIndex`, exclusive.
`Object``[]`	**`toArray`**`()` Returns an array containing all of the elements in this list in proper sequence (from first to last element).
`<T> T[]`	**`toArray`**`(T[] a)` Returns an array containing all of the elements in this list in proper sequence (from first to last element); the runtime type of the returned array is that of the specified array.
`void`	**`trimToSize`**`()` Trims the capacity of this `ArrayList` instance to be the list's current size.

OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

Java™ Platform
Standard Ed. 8

PREV CLASS NEXT CLASS FRAMES NO FRAMES ALL CLASSES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

compact1, compact2, compact3

java.util

Class HashMap<K,V>

java.lang.Object
    java.util.AbstractMap<K,V>
        java.util.HashMap<K,V>

Type Parameters:

`K` - the type of keys maintained by this map

`V` - the type of mapped values

All Implemented Interfaces:

`Serializable`, `Cloneable`, `Map<K,V>`

Direct Known Subclasses:

`LinkedHashMap`, `PrinterStateReasons`

```
public class HashMap<K,V>
extends AbstractMap<K,V>
implements Map<K,V>, Cloneable, Serializable
```

Hash table based implementation of the `Map` interface. This implementation provides all of the optional map operations, and permits `null` values and the `null` key. (The `HashMap` class is roughly equivalent to `Hashtable`, except that it is unsynchronized and permits nulls.) This class makes no guarantees as to the order of the map; in particular, it does not guarantee that the order will remain constant over time.

This implementation provides constant-time performance for the basic operations (`get` and `put`), assuming the hash function disperses the elements properly among the buckets. Iteration over collection views requires time proportional to the "capacity" of the `HashMap` instance (the number of buckets) plus its size (the number of key-value mappings). Thus, it's very important not to set the initial capacity too high (or the load factor too low) if iteration performance is important.

An instance of `HashMap` has two parameters that affect its performance: *initial capacity* and *load factor*. The *capacity* is the number of buckets in the hash table, and the initial capacity is simply the capacity at the time the hash table is created. The *load factor* is a measure of how full the hash table is allowed to get before its capacity is automatically increased. When the number of entries in the hash table exceeds the product of the load factor and the current capacity, the hash table is *rehashed* (that is, internal data structures are rebuilt) so that the hash table has approximately twice the number of buckets.

As a general rule, the default load factor (.75) offers a good tradeoff between time and space costs. Higher values decrease the space overhead but increase the lookup cost (reflected in most of the operations of the `HashMap` class, including `get` and `put`). The expected number of entries in the map and its load factor should be taken into account when setting its initial capacity, so as to minimize the number of rehash operations. If the initial capacity is greater than the maximum number of entries divided by the load factor, no rehash operations will ever occur.

If many mappings are to be stored in a `HashMap` instance, creating it with a sufficiently large capacity will allow the mappings to be stored more efficiently than letting it perform automatic rehashing as needed to grow the table. Note that using many keys with the same `hashCode()` is a sure way to slow down performance of any hash table. To ameliorate impact, when keys are `Comparable`, this class may use comparison order among keys to help break ties.

Note that this implementation is not synchronized. If multiple threads access a hash map concurrently, and at least one of the threads modifies the map structurally, it *must* be synchronized externally. (A structural modification is any operation that adds or deletes one or more mappings; merely changing the value associated with a key that an instance already contains is not a structural modification.) This is typically accomplished by synchronizing on some object that naturally encapsulates the map. If no such object exists, the map should be "wrapped" using the `Collections.synchronizedMap` method. This is best done at creation time, to prevent accidental unsynchronized access to the map:

```
   Map m = Collections.synchronizedMap(new HashMap(...));
```

The iterators returned by all of this class's "collection view methods" are *fail-fast*: if the map is structurally modified at any time after the iterator is created, in any way except through the iterator's own `remove` method, the iterator will throw a `ConcurrentModificationException`. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.

Note that the fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw `ConcurrentModificationException` on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness: *the fail-fast behavior of iterators should be used only to detect bugs.*

This class is a member of the Java Collections Framework.

Since:

`1.2`

See Also:

`Object.hashCode()`, `Collection`, `Map`, `TreeMap`, `Hashtable`, `Serialized Form`

Nested Class Summary

Nested classes/interfaces inherited from class java.util.AbstractMap

`AbstractMap.SimpleEntry<K,V>`, `AbstractMap.SimpleImmutableEntry<K,V>`

Nested classes/interfaces inherited from interface java.util.Map

`Map.Entry<K,V>`

Constructor Summary

Constructors

Constructor and Description

`HashMap()`

Constructs an empty `HashMap` with the default initial capacity (16) and the default load factor (0.75).

`HashMap(int initialCapacity)`

Constructs an empty `HashMap` with the specified initial capacity and the default load factor (0.75).

`HashMap(int initialCapacity, float loadFactor)`

Constructs an empty `HashMap` with the specified initial capacity and load factor.

`HashMap(Map<? extends K,? extends V> m)`

Constructs a new `HashMap` with the same mappings as the specified `Map`.

Method Summary

All Methods   **Instance Methods**   **Concrete Methods**

Modifier and Type	Method and Description
`void`	`clear()` Removes all of the mappings from this map.
`Object`	`clone()` Returns a shallow copy of this `HashMap` instance: the keys and values themselves are not cloned.
`V`	`compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)` Attempts to compute a mapping for the specified key and its current mapped value (or `null` if there is no current mapping).
`V`	`computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction)` If the specified key is not already associated with a value (or is mapped to `null`), attempts to compute its value using the given mapping function and enters it into this map unless `null`.
`V`	`computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)` If the value for the specified key is present and non-null, attempts to compute a new mapping given the key and its current mapped value.
`boolean`	`containsKey(Object key)` Returns `true` if this map contains a mapping for the specified key.
`boolean`	`containsValue(Object value)` Returns `true` if this map maps one or more keys to the specified value.
`Set<Map.Entry<K,V>>`	`entrySet()` Returns a `Set` view of the mappings contained in this map.

`void`	**`forEach`**`(BiConsumer<? super K,? super V> action)` Performs the given action for each entry in this map until all entries have been processed or the action throws an exception.
`V`	**`get`**`(Object key)` Returns the value to which the specified key is mapped, or `null` if this map contains no mapping for the key.
`V`	**`getOrDefault`**`(Object key, V defaultValue)` Returns the value to which the specified key is mapped, or `defaultValue` if this map contains no mapping for the key.
`boolean`	**`isEmpty`**`()` Returns `true` if this map contains no key-value mappings.
`Set<K>`	**`keySet`**`()` Returns a `Set` view of the keys contained in this map.
`V`	**`merge`**`(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction)` If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value.
`V`	**`put`**`(K key, V value)` Associates the specified value with the specified key in this map.
`void`	**`putAll`**`(Map<? extends K,? extends V> m)` Copies all of the mappings from the specified map to this map.
`V`	**`putIfAbsent`**`(K key, V value)` If the specified key is not already associated with a value (or is mapped to `null`) associates it with the given value and returns `null`, else returns the current value.
`V`	**`remove`**`(Object key)` Removes the mapping for the specified key from this map if present.
`boolean`	**`remove`**`(Object key, Object value)` Removes the entry for the specified key only if it is currently mapped to the specified value.
`V`	**`replace`**`(K key, V value)` Replaces the entry for the specified key only if it is currently mapped to some value.
`boolean`	**`replace`**`(K key, V oldValue, V newValue)` Replaces the entry for the specified key only if currently mapped to the specified value.
`void`	**`replaceAll`**`(BiFunction<? super K,? super V,? extends V> function)` Replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception.

`int`	**`size`**`()` Returns the number of key-value mappings in this map.
`Collection<V>`	**`values`**`()` Returns a `Collection` view of the values contained in this map.

Methods inherited from class java.util.AbstractMap

`equals, hashCode, toString`

Methods inherited from class java.lang.Object

`finalize, getClass, notify, notifyAll, wait, wait, wait`

Methods inherited from interface java.util.Map

`equals, hashCode`

Constructor Detail

HashMap

```
public HashMap(int initialCapacity,
               float loadFactor)
```

Constructs an empty `HashMap` with the specified initial capacity and load factor.

Parameters:

`initialCapacity - the initial capacity`

`loadFactor - the load factor`

Throws:

`IllegalArgumentException - if the initial capacity is negative or the load factor is nonpositive`

HashMap

```
public HashMap(int initialCapacity)
```

Constructs an empty `HashMap` with the specified initial capacity and the default load factor (0.75).

Parameters:

`initialCapacity - the initial capacity.`

Throws:

`IllegalArgumentException - if the initial capacity is negative.`

HashMap

OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

Java™ Platform
Standard Ed. 8

PREV CLASS NEXT CLASS FRAMES NO FRAMES ALL CLASSES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

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

Constructor Summary

Constructors

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 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 and 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.