

1 Strings

1.1 Creating a String Instance

- `String str1 = "Hello";` // Using literal String
- `str2 = new String("World");` // Using 'new' keyword

1.2 String Methods

- `char charAt(int index)`
// Returns the character at a specified index. Throws `IndexOutOfBoundsException`.
- `boolean equalsIgnoreCase(String str)`
// Compares the string values (ignoring case) and returns boolean value.
- `boolean equals(Object obj)`
// Compares the string object and returns boolean value.
- `int compareTo(String str)`
// Compares two strings based upon Unicode value of each character.
 - return 0 if both strings are equal.
 - return positive value if calling string is lexicographically greater than the parameterized string.
 - return negative value if parameterized string is lexicographically greater than the calling string.
 - if used to compare a string where $length \geq 1$ to an empty string, `compareTo` returns string length.
- `int compareToIgnoreCase(String str)`
// Compares two strings based upon Unicode value of each character *ignoring case*.
- `boolean startsWith(String prefix, int offset)`
// Checks whether a substring (starting at offset index) has the supplied prefix.
- `boolean startsWith(String prefix)`
// Checks whether a string has the supplied prefix.
- `boolean endsWith(String suffix)`
// Checks whether a string has the supplied suffix.
- `int hashCode()`
// Returns the hash code of the string.
- `int indexOf(int ch)`
// Returns the index of the first occurrence of the character 'ch' in the string.

- `int indexOf(int ch, int fromIndex)`
// Same as `indexOf` but starts searching for 'ch' at the specified `fromIndex`.
- `int lastIndexOf(int ch)`
// Returns the index of the last occurrence of 'ch' in the string
- `int lastIndexOf(int ch, int fromIndex)`
// Same as above except beginning search from 'fromIndex'.
- `int indexOf(String str)`
// Returns the index (first letter) of the first occurrence of specified substring 'str'. If it doesn't exist, it returns `-1`.
- `int lastIndexOf(String str)`
// Returns the index (first letter) of the last occurrence of specified substring 'str'. If it doesn't exist, it returns `-1`.
- `String substring(int beginIndex)`
// Returns the substring starting at 'beginIndex' and ending at the end of the string.
- `String substring(int beginIndex, int endIndex)`
// Returns the substring starting at 'beginIndex' and ending at 'endIndex'.
- `String concat(String str)`
// Concatenates the specified string 'str' at the end of the calling string.
- `String replace(char oldChar, char newChar)`
// Returns a new string after where each instance of 'oldChar' is replaced by an instance of 'newChar'.
- `boolean contains(CharSequence s)`
// Checks if the calling string contains the specified sequence of char values. Throws `NullPointerException` if 's' is null.
- `String toUpperCase(Locale locale)`
// Converts the string to upper upper-case using the rules defined by the specified locale.
- `String toUpperCase()`
// Same as above and `locale = Locale.getDefault()`.
- `String intern()`
// Searches for the specified string in the memory pool and if found returns the reference to it. If the specified string is not found, the method allocates memory space to the specified string and assigns the reference to it. Java automatically interns string literals; this is useful when using 'new' keyword to make a string

instance

- ```
String str1 = ``New String``;
String str2 = ``New String``;
String str3 = new String(``New String``);
String str4 = new String(``New String``).intern();
System.out.println("Are str1 and str2 the same: " + (str1 == str2)); // Returns true
System.out.println("Are str1 and str3 the same: " + (str1 == str3)); // Returns false
System.out.println("Are str1 and str2 the same: " + (str1 == str4)); // Returns true
System.out.println("Are str1 and str3 the same: " + (str1 == str3.intern())); // Returns true
```
- `boolean isEmpty()`  
// Method returns true if the given string has 0 length.
  - `public static String join()`  
// Method joins the given strings using the specified delimiter and returns the concatenated Java String. CHECK AGAIN LATER.
  - `String replaceFirst(String regex, String replacement)`  
// Replaces the first occurrence of substring that fits the given regular expression 'regex' with 'replacement'.
  - `String replaceAll(String regex, String replacement)`  
// Replaces all occurrences of substrings that fit 'regex' with 'replacement'.
  - `String[] split(String regex, int limit)`  
// Returns an array of substrings delimited by the given regular expression. 'limit' is a result threshold.
  - `String[] split(String regex)`  
// Same as above, but without limit
  - `String toLowerCase(Locale locale)`  
// Converts all of the characters in this String to lower case using the rules of the given Locale. If 'locale' is not specified, method uses `Local.getDefault()`
  - `public static String format(Locale l, String format, Object... args)`  
// Returns a formatted string using the specified locale, format string, and arguments
  - `String trim()`

- `// Returns a copy of the string with the leading and trailing whitespace omitted.`
- `char toCharArray()`  
`// Converts this string to a new character array.`
- `public static String copyValueOf(char[] data)`  
`// Returns a String that represents the character sequence in the array specified.`
- `public static String copyValueOf(char[] data, int offset, int count)`  
`// Returns a String, starting at the initial 'offset' index and continuing for 'count' characters, that represents the character sequence in the array specified.`
- `public void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)`  
`// Copies characters from this string into the destination character array.`
- `public static String valueOf([argument])`  
`// Returns a string representation of argument, which can be boolean, char, int, long, float, double.`
- `boolean contentEquals(StringBuffer sb)`  
`// Argument can also be a CharSequence. Compares 'this' string to the specified StringBuffer (or CharSequence). The result is true if and only if this String represents the same sequence of characters as the specified StringBuffer (or CharSequence).`
- `boolean regionMatches(int srcoffset, String dest, int destoffset, int len)`  
`// Tests to see if two string regions are equal.`
- `boolean regionMatches(boolean ignoreCase, int srcoffset, String dest, int destoffset, int len)`  
`// Tests to see if two string regions are equal ignoring case.`
- `byte[] getBytes(String charsetName)`  
`// Encodes this String into a sequence of bytes using the named charset, storing the result into a new byte array. Can also take a Charset type argument.`
- `byte[] getBytes()`  
`// Encodes this String into a sequence of bytes using the platform's default charset, storing the result into a new byte array.`
- `int length()`  
`// Returns the length of 'this' string.`

- `boolean matches(String regex)`  
`// Tells whether or not this string matches the given regular expression.`
- `int codePointAt(int index)`  
`// Returns the character (Unicode code point) at the specified index.`

## 2 HashSet

### 2.1 Constructors

- `HashSet()`  
`// Constructs a new, empty set; the backing HashMap instance has default initial capacity 16 and load factor 0.75.`
- `HashSet(Collection<? extends E> c)`  
`// Constructs a new set containing the elements in the specified collection.`
- `HashSet(int initialCapacity)`  
`// Constructs a new, empty set; the backing HashMap instance has specified initial capacity and load factor 0.75.`
- `HashSet(int initialCapacity, float loadFactor)`  
`// Constructs a new, empty set; the backing HashMap instance has specified initial capacity and specified load factor.`

### 2.2 Creating a HashSet Instance

- `HashSet<T> set = new HashSet<T>();`  
`// Creates a new set of the specified type using default constructor.`
- `ArrayList<T> list = new ArrayList<T>();`  
`HashSet<T> set = new HashSet(list);`  
`// Creates a new set containing the elements of another collection.`

### 2.3 HashSet Methods

- `public boolean add(E e)`  
`// Adds the specified element to the set if it is not already present. If the set already contains the element, the set is unchanged and method returns false.`
- `public boolean remove(Object o)`  
`// Removes the specified element from the set if it is present. Returns true if the set contained the element and the element was true; else, method returns false.`
- `public void clear()`

- `// Removes all elements from 'this' set. The set will be empty after the call returns.`
- `public Object clone()`  
`// Returns a shallow copy of 'this' HashSet instance; the elements themselves are not cloned.`
- `public boolean contains(Object o)`  
`// Returns true if 'this' set contains the specified element.`
- `public boolean isEmpty()`  
`// Returns true if 'this' set contains no elements.`
- `public int size()`  
`// Returns the number of elements in this set (its cardinality).`
- `public Iterator<E> iterator()`  
`// Returns an iterator over the elements in 'this' set. The elements are returned in no particular order.`  

```
Iterator<T> itr = set.iterator();
While(itr.hasNext()){ System.out.println(itr.next());
}
```

## 3 HashMap

### 3.1 Constructors

- `HashMap()`  
`// Constructs an empty HashMap with default initial capacity 16 and default load factor 0.75.`
- `HashMap(int initialCapacity)`  
`// Constructs a new, empty HashMap with the specified initial capacity and load factor 0.75.`
- `HashMap(int initialCapacity, float loadFactor)`  
`// Constructs a new, empty HashMap with the specified initial capacity and specified load factor.`
- `HashMap(Map<? extends K, ? extends V> m)`  
`// Constructs a new HashMap with the same mappings as the specified Map.`

### 3.2 Creating a HashMap Instance

- `HashMap<Integer, String> hmap = new HashMap<Integer, String>();`

### 3.3 HashMap Methods

- `public int size()`  
`// Returns the number of key-value mappings in 'this' map.`

- public boolean **isEmpty()**  
// Returns true if 'this' map contains no key-value mappings.
- public 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. It could also be the case that the mapping of the given key is a null value - in which case, the `containsKey` operation must be used to distinguish.
- public boolean **containsKey**(Object key)  
// Returns true if this map contains a mapping for the specified key.
- public V **put**(K key, V value)  
// Associates the specified value with the specified key in 'this' map. If the map already contains a mapping for the key, the old value is replaced.
- public void **putAll**(Map<? extends K, ? extends V> m)  
// Copies all of the mappings from the specified map to this map. These mappings will replace any mappings that this map had for any of the keys currently in the specified map.
- public V **remove**(Object key)  
// Removes the mapping for the specified key from 'this' map if present.
- public void **clear**()  
// Removes all of the mappings from 'this' map.
- public boolean **containsValue**(Object value)  
// Returns true if 'this' map maps one or more keys to the specified value.
- public Set<K> **keySet**()  
// Returns a Set view of the keys contained in this map. The set is backed by the map, so changes to the map are reflected in the set, and vice-versa.
- public Collection<V> **values**()  
// Returns a Collection view of the values contained in 'this' map. Changes to the collection are reflected in the map, and vice-versa.
- public Set<Map.Entry<K, V> > **entrySet**()  
// Returns a Set view of the mappings contained in 'this' map. Changes to the map are reflected in the set, and vice-versa.
- public V **getOrDefault**(Object key, V defaultValue)  
// Returns the value to which the specified key is mapped, or de-

faultValue if 'this' map contains no mapping for the key.

- public V **putIfAbsent**(K key, V value)  
// If the specified key is not already associated with a value (or is mapped to null) method associates it with the given value and returns null; else method returns the current value.
- public boolean **remove**(Object key, Object value)  
// Removes the entry for the specified key only if it is currently mapped to the specified value.
- public boolean **replace**(K key, V oldValue, V newValue)  
// Replaces the entry for the specified key only if currently mapped to the specified value.
- public boolean **replace**(K key, V value)  
// Replaces the entry for the specified key only if it is currently mapped to some value.
- public 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), method attempts to compute its value using the given mapping function and enters it into 'this' map unless null.
- public V **computeIfPresent**(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction)  
// If the value for the specified key is present and non-null, method attempts to compute a new mapping given the key and its current mapped value. If the function returns null, mapping is removed. If the function throws an unchecked exception, mapping is unchanged.
- public 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). For example, to either create or append a String msg to a value mapping:  

```
map.compute(key, (k, v) -> (v == null) ?
msg : v.concat(msg))
```
- public V **merge**(K key, V value, BiFunction<? super K, ? 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.

Otherwise, replaces the associated value with the results of the given remapping function, or removes if the result is null. This method may be of use when combining multiple mapped values for a key. For example, to either create or append a String msg to a value mapping:

```
map.merge(key, msg, String::concat)
```

- public 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. Unless otherwise specified by the implementing class, actions are performed in the order of entry set iteration (if an iteration order is specified.) Exceptions thrown by the action are relayed to the caller.
- public 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. Exceptions thrown by the function are relayed to the caller.
- public Object **clone**()  
// Returns a shallow copy of this HashMap instance: the keys and values themselves are not cloned.

## 4 Hashtable

### 4.1 Constructors

- **Hashtable()**  
// Constructs an empty Hashtable with default initial capacity 11 and default load factor 0.75.
- **Hashtable(int initialCapacity)**  
// Constructs a new, empty Hashtable with the specified initial capacity and load factor 0.75.
- **Hashtable(int initialCapacity, float loadFactor)**  
// Constructs a new, empty Hashtable with the specified initial capacity and specified load factor.
- **Hashtable(Map<? extends K, ? extends V> t)**  
// Constructs a new Hashtable with the same mappings as the specified Map.

## 4.2 Creating a Hashtable Instance

- `Hashtable<String, String> hashtable = new Hashtable<String, String>();`

## 4.3 Hashtable Methods

// Also contains implementations of all methods defined in HashMap Methods.

- `public Enumeration<K> keys()`  
// Returns an enumeration of the values in this hashtable. Use the Enumeration methods on the returned object to fetch the elements sequentially.
- `public Enumeration<V> elements()`  
// Returns an enumeration of the values in this hashtable. Use the Enumeration methods on the returned object to fetch the elements sequentially.
- `public boolean contains(Object value)`  
// Tests if some key maps into the specified value in this hashtable. This operation is more expensive than the `containsKey` method. This is identical in functionality to `containsValue` (map interface).
- `protected void rehash()`  
// Increases the capacity of and internally reorganizes this hashtable, in order to accommodate and access its entries more efficiently. This method is called automatically when the number of keys in the hashtable exceeds this hashtable's capacity and load factor.
- `public String toString()`  
// Returns a string representation of this Hashtable object in the form of a set of entries, enclosed in braces and separated by the ASCII characters ", " (comma and space). Each entry is rendered as the key, an equals sign =, and the associated element, where the `toString` method is used to convert the key and element to strings.
- `public boolean equals(Object o)`  
// Compares the specified Object with this Map for equality, as per the definition in the Map interface.
- `public boolean hashCode()`  
// Returns the hash code value for this Map as per the definition in the Map interface.

## 5 Interface Enumeration<E>

// An object that implements the Enumeration interface generates a series of elements, one at a time. Successive calls to the `nextElement` method return successive elements of the series. `StringTokenizer` implements Enumeration. An example for printing all elements of a `Vector<E>` v:

- `for (Enumeration<E> e = v.elements(); e.hasMoreElements();) System.out.println(e.nextElement());`

### 5.1 Enumeration Methods

- `boolean hasMoreElements()`  
// Tests if this enumeration contains more elements.
- `E nextElement()`  
// Returns the next element of this enumeration if this enumeration object has at least one more element to provide.

## 6 Arrays

### 6.1 Arrays Methods

#### 6.1.1 Sort

- `public static void sort(int[] a)`  
// Sorts the specified array into ascending numerical order. Dual-Pivot Quicksort - O(nlogn).
  - `sort(int[] a, int fromIndex, int toIndex)`
  - // Sorts the specified range of the array into ascending order. The range to be sorted extends from the index `fromIndex`, inclusive, to the index `toIndex`, exclusive. If `fromIndex == toIndex`, the range to be sorted is empty.
- Other overloaded signatures have the same format, but the initial array can be of type:
- long
  - short
  - char
  - byte
  - float
  - double

- `public static void sort(Object[] a)`  
// Sorts the specified array of objects into ascending order, according to the natural ordering of its elements. All elements in the array must implement the `Comparable` interface. Furthermore,

all elements in the array must be mutually comparable (that is, `e1.compareTo(e2)` must not throw a `ClassCastException` for any elements `e1` and `e2` in the array).

- `public static void sort(Object[] a, int fromIndex, int toIndex)`  
// Same as above except the range to be sorted extends from index `fromIndex`, inclusive, to index `toIndex`, exclusive.
- `public static <T> void sort(T[] a, Comparator<? super T> cmp)`  
// Sorts the specified array of objects according to the order induced by the specified comparator. All elements in the array must be mutually comparable by the specified comparator.
- `public static <T> void sort(T[] a, int fromIndex, int toIndex, Comparator<? super T> cmp)`  
// Same as above except the range to be sorted extends from index `fromIndex`, inclusive, to index `toIndex`, exclusive.

#### 6.1.2 parallelSort

// The sorting algorithm is a parallel sort-merge that breaks the array into sub-arrays that are themselves sorted and then merged. When the sub-array length reaches a minimum granularity, the sub-array is sorted using the appropriate `Arrays.sort` method. If the length of the specified array is less than the minimum granularity, then it is sorted using the appropriate `Arrays.sort` method. The algorithm requires a working space no greater than the size of the original array. The `ForkJoin` common pool is used to execute any parallel tasks.

- `public static void parallelSort(byte[] a)`  
// Sorts the specified array into ascending numerical order.
  - `public static void parallelSort(byte[] a, int fromIndex, int toIndex)`  
// Sorts the specified range of the array into ascending numerical order. The range to be sorted extends from the index `fromIndex`, inclusive, to the index `toIndex`, exclusive. If `fromIndex == toIndex`, the range to be sorted is empty.
- Other overloaded signatures have the same format, but the initial array can be of type:
- long
  - short
  - char
  - int



- float
- double
- public static <T extends Comparable<? super T> void **parallelSort**(T[] a)  
// Sorts the specified array of objects into ascending order, according to the natural ordering of its elements. All elements in the array must implement the Comparable interface. Furthermore, all elements in the array must be mutually comparable (that is, e1.compareTo(e2) must not throw a ClassCastException for any elements e1 and e2 in the array). This sort is guaranteed *stable*.
- public static <T extends Comparable<? super T> void **parallelSort**(T[] a, int fromIndex, int toIndex)  
// Same as above but extends from index fromIndex (inclusive) to index toIndex (exclusive).
- public static <T> void **parallelSort**(T[] a, Comparator<? super T> cmp)  
// Sorts the specified array of objects according to the order induced by the specified comparator. All elements in the array must be mutually comparable by the specified comparator (that is, c.compare(e1, e2) must not throw a ClassCastException for any elements e1 and e2 in the array). This sort is guaranteed *stable*.
- public static <T> void **parallelSort**(T[] a, int fromIndex, int toIndex, Comparator<? super T> cmp)  
// Sorts the specified range of the specified array of objects according to the order induced by the specified comparator. Follows applicable rules above for appropriate parameters.

### 6.1.3 Parallel Prefix

- public static <T> void **parallelPrefix**(T[] array, BinaryOperator<T> op)  
// Cumulates, in parallel, each element of the given array in place, using the supplied function. For example if the array initially holds [2, 1, 0, 3] and the operation performs addition, then upon return the array holds [2, 3, 3, 6]. Parallel prefix computation is usually more efficient than sequential loops for large arrays.
- public static <T> void **parallelPrefix**(T[] array, int fromIndex, int toIndex, BinaryOperator<T> op)
- public static void **parallelPrefix**(long[] array, LongBinaryOperator op)

- public static void **parallelPrefix**(long[] array, int fromIndex, int toIndex, LongBinaryOperator op)
- public static void **parallelPrefix**(double[] array, DoubleBinaryOperator op)
- public static void **parallelPrefix**(double[] array, int fromIndex, int toIndex, DoubleBinaryOperator op)
- public static void **parallelPrefix**(int[] array, IntBinaryOperator op)
- public static void **parallelPrefix**(int[] array, int fromIndex, int toIndex, IntBinaryOperator op)

### 6.1.4 Binary Search

- public static int **binarySearch**(long[] a, long key)  
// Searches the specified array of longs for the specified value using the binary search algorithm. The array must be sorted (as by the sort(long[]) method) prior to making this call. If it is not sorted, the results are undefined. If the array contains multiple elements with the specified value, there is no guarantee which one will be found.
- public static int **binarySearch**(long[] a, int fromIndex, int toIndex, long key)  
// Same as above except between fromIndex (inclusive) to toIndex(exclusive).  
// The above two signatures are overloaded where the first and fourth parameters (the array and the key) can have the following data-types:
  - int
  - short
  - char
  - byte
  - double
  - float
  - Object
- public static <T> int **binarySearch**(T[] a, int fromIndex, int toIndex, T key, Comparator<? super T> c)  
// Searches the specified array for the specified object using the binary search algorithm. The array must be sorted into ascending order according to the specified comparator (as by the sort(T[], Comparator) method) prior to making this call. If it is not sorted, the results are undefined. If the array contains multiple elements

equal to the specified object, there is no guarantee which one will be found.

- public static <T> int **binarySearch**(T[] a, int fromIndex, int toIndex, T key, Comparator<? super T> c)  
// Searches a range of the specified array for the specified object using the binary search algorithm. The range must be sorted into ascending order according to the specified comparator (as by the sort(T[], int, int, Comparator) method) prior to making this call. If it is not sorted, the results are undefined. If the range contains multiple elements equal to the specified object, there is no guarantee which one will be found.

### 6.1.5 Equals

- public static boolean **equals**(long[] a, long[] a2)  
// Returns true if the two specified arrays of longs are equal to one another. Two arrays are considered equal if both arrays contain the same number of elements, and all corresponding pairs of elements in the two arrays are equal. In other words, two arrays are equal if they contain the same elements in the same order. Also, two array references are considered equal if both are null.  
// The above to signature is overloaded where both parameters can be arrays of type:
  - int
  - short
  - char
  - byte
  - boolean
  - double
  - float
  - Object

### 6.1.6 Fill

- public static void **fill**(long[] a, long val)  
// Assigns the specified long value to each element of the specified array of longs.
- public static void **fill**(long[] a, int fromIndex, int toIndex, long val)  
// Assigns the specified long value to each element of the specified range of the specified array of longs. The range to be filled extends from index fromIndex, inclusive, to index toIndex, exclusive. (If fromIndex==toIndex, the range to be filled is empty.)

// The above two signatures are overloaded where the array and the 'val' can have the following data-types:

- int
- short
- char
- byte
- boolean
- double
- float
- Object

### 6.1.7 copyOf

- public static <T> T[] **copyOf**(T[] original, int newLength)

// Copies the specified array, truncating or padding with nulls (if necessary) so the copy has the specified length. For all indices that are valid in both the original array and the copy, the two arrays will contain identical values. For any indices that are valid in the copy but not the original, the copy will contain null. Such indices will exist if and only if the specified length is greater than that of the original array. The resulting array is of exactly the same class as the original array.

- public static <T,U> T[] **copyOf**(U[] original, int newLength, Class<? extends T[]> newType)

// Copies the specified array, truncating or padding with nulls (if necessary) so the copy has the specified length. For all indices that are valid in both the original array and the copy, the two arrays will contain identical values. For any indices that are valid in the copy but not the original, the copy will contain null. Such indices will exist if and only if the specified length is greater than that of the original array. The resulting array is of the class newType. U - The class of objects in the original array; T - the class of objects in the returned array.

- public static byte[] **copyOf**(byte[] original, int newLength)

// Copies the specified array, truncating or padding with zeros (if necessary) so the copy has the specified length. For all indices that are valid in both the original array and the copy, the two arrays will contain identical values. For any indices that are valid in the copy but not the original, the copy will contain (byte)0. Such

indices will exist if and only if the specified length is greater than that of the original array.

// The above signature where the return type is primitive is overloaded where the array passed and returned can have the following data-types:

- int
- byte
- short
- char
- long
- boolean
- double
- float

### 6.1.8 copyOfRange

- public static <T> T[] **copyOfRange**(T[] original, int from, int to)

// Copies the specified range of the specified array into a new array. The initial index of the range (from) must lie between zero and original.length, inclusive. The value at original[from] is placed into the initial element of the copy (unless from == original.length or from == to). Values from subsequent elements in the original array are placed into subsequent elements in the copy. The final index of the range (to), which must be greater than or equal to from, may be greater than original.length, in which case null is placed in all elements of the copy whose index is greater than or equal to original.length - from. The length of the returned array will be to - from.

- public static <T, U> T[] **copyOfRange**(U[] original, int from, int to, Class<? extends T[]> newType)

// Same as above except that the resulting array is of the class newType.

- public static byte[] **copyOfRange**(byte[] original, int from, int to)

// The above signature where the return type is primitive is overloaded where the array passed and returned can have the following data-types:

- int
- byte
- short

- char
- long
- boolean
- double
- float

### 6.1.9 asList

- public static <T> List<T> **asList**(T... a)

// Returns a fixed-size list backed by the specified array. (Changes to the returned list "write through" to the array.) This method acts as bridge between array-based and collection-based APIs, in combination with Collection.toArray(). The returned list is serializable and implements RandomAccess.

A convenient way to create a fixed-size list initialized to contain several elements:

```
List<String> stooges = Arrays.asList("Larry",
"Moe", "Curly");
```

### 6.1.10 hashCode

- public static int **hashCode**(long[] a)

// Returns a hash code based on the contents of the specified array. For any two long arrays a and b such that Arrays.equals(a, b), it is also the case that Arrays.hashCode(a) == Arrays.hashCode(b). The value returned by this method is the same value that would be obtained by invoking the hashCode method on a List containing a sequence of Long instances representing the elements of a in the same order. If a is null, this method returns 0.

// The above signature always returns int, but the type of array passed can be:

- int
- byte
- short
- char
- long
- boolean
- double
- float
- Object

### 6.1.11 deepHashCode

- public static int **deepHashCode**(Object[] a)

// Returns a hash code based on the "deep contents" of the specified array. If the array contains other arrays as elements, the hash code is based on their contents and so on, ad infinitum. It is therefore unacceptable to invoke this method on an array that contains itself as an element, either directly or indirectly through one or more levels of arrays. The behavior of such an invocation is undefined.

#### 6.1.12 deepEquals

- public static boolean **deepEquals**(Object[] a1, Object[] a2)

// Returns true if the two specified arrays are deeply equal to one another. Unlike the equals(Object[],Object[]) method, this method is appropriate for use with nested arrays of arbitrary depth. Two array references are considered deeply equal if both are null, or if they refer to arrays that contain the same number of elements and all corresponding pairs of elements in the two arrays are deeply equal.

#### 6.1.13 toString

- public static String **toString**(long[] a)

// Returns a string representation of the contents of the specified array. The string representation consists of a list of the array's elements, enclosed in square brackets ("[]"). Adjacent elements are separated by the characters ", " (a comma followed by a space). Elements are converted to strings as by String.valueOf(long). Returns "null" if a is null.

// The above signature always returns String, but the type of array passed can be:

- int
- byte
- short
- char
- long
- boolean
- double
- float
- Object

#### 6.1.14 deepToString

- public static String **deepToString**(Object[] a)

// Returns a string representation of the "deep contents" of the specified array. If the array contains other arrays as elements, the string representation contains their contents and so on. This method is designed for converting multidimensional arrays to strings. To avoid infinite recursion, if the specified array contains itself as an element, or contains an indirect reference to itself through one or more levels of arrays, the self-reference is converted to the string "...". For example, an array containing only a reference to itself would be rendered as "[...]".

## 7 Vector

Roughly equivalent to an ArrayList, except Vectors are synchronized.

### 7.1 Constructors

- public **Vector**()  
// Constructs an empty vector so that its internal data array has size 10 and its standard capacity increment is zero.
- public **Vector**(int initialCapacity) // Constructs an empty vector with the specified initial capacity and with its capacity increment equal to zero.
- public **Vector**(int initialCapacity, int capacityIncrement)  
// Constructs an empty vector with the specified initial capacity and capacity increment.
- public **Vector**(Collection<? extends E> c)  
// Constructs a vector containing the elements of the specified collection, in the order they are returned by the collection's iterator.

### 7.2 Vector Methods

- public void **copyInto**(Object[] anArray)  
// Copies the components of this vector into the specified array. The item at index k in this vector is copied into component k of anArray.
- public void **trimToSize**()  
// Trims the capacity of this vector to be the vector's current size. If the capacity of this vector is larger than its current size, then the capacity is changed to equal the size by replacing its internal data array, kept in the field elementData, with a smaller one. An application can use this operation to minimize the storage of a vector.

- public void **ensureCapacity**(int minCapacity)  
// Increases the capacity of this vector, if necessary, to ensure that it can hold at least the number of components specified by the minimum capacity argument.
- public void **setSize**(int newSize)  
// Sets the size of this vector. If the new size is greater than the current size, new null items are added to the end of the vector. If the new size is less than the current size, all components at index newSize and greater are discarded.
- public int **capacity**()  
// Returns the current capacity of this vector.
- public int **size**()  
// Returns the number of components in this vector.
- public boolean **isEmpty**()  
// Tests if this vector has no components.
- public Enumeration<E> **elements**()  
// Returns an enumeration of the components of this vector. The returned Enumeration object will generate all items in this vector. The first item generated is the item at index 0, then the item at index 1, and so on.
- public boolean **contains**(Object o)  
// Returns true if this vector contains the specified element. More formally, returns true if and only if this vector contains at least one element e such that (o==null ? e==null : o.equals(e)).
- public int **indexOf**(Object o)  
// Returns the index of the first occurrence of the specified element in this vector, or -1 if this vector does not contain the element. More formally, returns the lowest index i such that (o==null ? get(i)==null : o.equals(get(i))), or -1 if there is no such index.
- public int **indexOf**(Object o, int index)  
// Returns the index of the first occurrence of the specified element in this vector, searching forwards from index, or returns -1 if the element is not found.
- public int **lastIndexOf**(Object o)  
// Returns the index of the last occurrence of the specified element in this vector, or -1 if this vector does not contain the element. More formally, returns the highest index i such that (o==null ? get(i)==null : o.equals(get(i))), or -1 if there is no such index.

- public int **lastIndexOf**(Object o, int index)  
// Returns the index of the last occurrence of the specified element in this vector, searching backwards from index, or returns -1 if the element is not found.
- public E **elementAt**(int index)  
// Returns the component at the specified index. Identical in function to List interface's get(int) method.
- public E **firstElement**()  
// Returns the first component (the item at index 0) of this vector.
- public E **lastElement**()  
// Returns the last component of the vector.
- public void **setElementAt**(E obj, int index)  
// Sets the component at the specified index of this vector to be the specified object. The previous component at that position is discarded.
- public void **removeElementAt**(int index)  
// Deletes the component at the specified index. Each component in this vector with an index greater or equal to the specified index is shifted downward to have an index one smaller than the value it had previously. The size of this vector is decreased by 1. Identical in function to List interface's remove(int) method, except that returns the old value at specified position.
- public void **insertElementAt**(E obj, int index)  
// Inserts the specified object as a component in this vector at the specified index. Each component in this vector with an index greater or equal to the specified index is shifted upward to have an index one greater than the value it had previously. Identical in function to List interface's add(int, E), except add reverses the order of parameters.
- public void **addElement**(E obj)  
// Adds the specified component to the end of this vector, increasing its size by one. The capacity of this vector is increased if its size becomes greater than its capacity. This method is identical in functionality to the add(E) method (which is part of the List interface).
- public boolean **removeElement**(Object obj)  
// Removes the first (lowest-indexed) occurrence of the argument from this vector. If the object is found in this vector, each component in the vector with an index greater or equal to the object's

index is shifted downward to have an index one smaller than the value it had previously. This method is identical in functionality to the remove(Object) method (which is part of the List interface).

- public Object **clone**()  
// Returns a clone of this vector. The copy will contain a reference to a clone of the internal data array, not a reference to the original internal data array of this Vector object.
- public Object[] **toArray**()  
// Returns an array containing all of the elements in this Vector in the correct order.
- public <T> T[] **toArray**(T[] a)  
// Returns an array containing all of the elements in this Vector in the correct order; the runtime type of the returned array is that of the specified array. If the Vector fits in the specified array, it is returned therein. Otherwise, a new array is allocated with the runtime type of the specified array and the size of this Vector.
- public E **get**(int index)  
// Returns the element at the specified position in this Vector.
- public E **set**(int index, E element)  
// Replaces the element at the specified position in this Vector with the specified element.
- public boolean **add**(E e)  
// Appends the specified element to the end of this Vector.
- public boolean **remove**(Object o)  
// Removes the first occurrence of the specified element in this Vector. If the Vector does not contain the element, it is unchanged. More formally, removes the element with the lowest index i such that (o==null ? get(i)==null : o.equals(get(i))) (if such an element exists).
- public void **add**(int index, E element)  
// Inserts the specified element at the specified position in this Vector. Shifts the element currently at that position (if any) and any subsequent elements to the right (adds one to their indices).
- public E **remove**(int index)  
// Removes the element at the specified position in this Vector. Shifts any subsequent elements to the left (subtracts one from their indices). Returns the element that was removed from the Vector.

- public void **clear**()  
// Removes all of the elements from this Vector. The Vector will be empty after this call returns (unless it throws an exception).
- public boolean **containsAll**(Collection<?> c)  
// Returns true if this Vector contains all of the elements in the specified Collection.
- public boolean **addAll**(Collection<? extends E> c)  
// Appends all of the elements in the specified Collection to the end of this Vector, in the order that they are returned by the specified Collection's Iterator. The behavior of this operation is undefined if the specified Collection is modified while the operation is in progress. (This implies that the behavior of this call is undefined if the specified Collection is this Vector, and this Vector is nonempty.)
- public boolean **removeAll**(Collection<?> c)  
// Removes from this Vector all of its elements that are contained in the specified Collection.
- public boolean **retainAll**(Collection<?> c)  
// Retains only the elements in this Vector that are contained in the specified Collection. In other words, removes from this Vector all of its elements that are not contained in the specified Collection.
- public boolean **addAll**(int index, Collection<? extends E> c)  
// Inserts all of the elements in the specified Collection into this Vector at the specified position. Shifts the element currently at that position (if any) and any subsequent elements to the right (increases their indices).
- public boolean **equals**(Object o)  
// Compares the specified Object with this Vector for equality. Returns true if and only if the specified Object is also a List, both Lists have the same size, and all corresponding pairs of elements in the two Lists are equal.
- public int **hashCode**()  
// Returns the hash code value for this Vector.
- public String **toString**()  
// Returns a string representation of this Vector, containing the String representation of each element.
- public List<E> **subList**(int fromIndex, int toIndex)



// Returns a view of the portion of this List between fromIndex, inclusive, and toIndex, exclusive. (If fromIndex and toIndex are equal, the returned List is empty.) The returned List is backed by this List, so changes in the returned List are reflected in this List, and vice-versa. The returned List supports all of the optional List operations supported by this List.

- **protected void removeRange**(int fromIndex, int toIndex)

// Removes from this list all of the elements whose index is between fromIndex, inclusive, and toIndex, exclusive. Shifts any succeeding elements to the left (reduces their index). This call shortens the list by (toIndex - fromIndex) elements. (If toIndex==fromIndex, this operation has no effect.)

- **public 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. The specified index indicates the first element that would be returned by an initial call to next. An initial call to previous would return the element with the specified index minus one.
- **public ListIterator<E> listIterator**()  
// Returns a list iterator over the elements in this list (in proper sequence).
- **public Iterator<E> iterator**()  
// Returns an iterator over the elements in this list in proper sequence.
- **public 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. Unless otherwise specified by the implementing class, actions are performed in the order of iteration (if an iteration order is specified). Exceptions thrown by the action are relayed to the caller.
- **public boolean removeIf**(Predicate<? super E> filter)  
// Removes all of the elements of this collection that satisfy the given predicate. Errors or runtime exceptions thrown during iteration or by the predicate are relayed to the caller.
- **public void replaceAll**(UnaryOperator<E> operator)  
// Replaces each element of this list with the result of applying the operator to that element. Errors or runtime exceptions thrown by

the operator are relayed to the caller.

- **public void sort**(Comparator<? super E> c)  
// Sorts this list according to the order induced by the specified Comparator. If the specified comparator is null then all elements in this list must implement the Comparable interface and the elements' natural ordering should be used.
- **public Splitter<E> splitter**()  
// Creates a late-binding and fail-fast Splitter over the elements in this list.

## 8 ArrayList

Roughly equivalent to a Vector, except an ArrayList is unsynchronized

### 8.1 Constructors

- **public ArrayList**()  
// Constructs an empty list with an initial capacity of ten.
- **public ArrayList**(int initialCapacity)  
// Constructs an empty list with the specified initial capacity.
- **public 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.

### 8.2 ArrayList Methods

- **public void trimToSize**()  
// Trims the capacity of this ArrayList instance to be the list's current size.
- **public 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.
- **public int size**()  
//Returns the number of elements in this list.
- **public boolean isEmpty**()  
// Returns true if this list contains no elements.
- **public boolean contains**(Object o)  
// Returns true if this list contains the specified element.
- **public 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.
- **public 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.

- **public Object clone**()  
// Returns a shallow copy of this ArrayList instance.
- **public Object[] toArray**()  
// Returns an array containing all of the elements in this list in proper sequence (from first to last element).
- **public <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.
- **public E get**(int index)  
// Returns the element at the specified position in this list.
- **public E set**(int index, E element)  
// Replaces the element at the specified position in this list with the specified element.
- **public boolean add**(E e)  
// Appends the specified element to the end of this list.
- **public void add**(int index, E element)  
// Inserts the specified element at the specified position in this list. Shifts the element currently at that position (if any) and any subsequent elements to the right (adds one to their indices).
- **public E remove**(int index)  
// Removes the element at the specified position in this list. Shifts any subsequent elements to the left (subtracts one from their indices).
- **public boolean remove**(Object o)  
// Removes the first occurrence of the specified element from this list, if it is present.
- **public void clear**()  
// Removes all of the elements from this list. The list will be empty after this call returns.
- **public 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. The behavior of this operation is undefined if the specified collection is modified while the operation is in progress. (This implies that the behavior of this call is undefined if the specified collection is this list, and this list is nonempty.)

- **public 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. Shifts the element currently at that position (if any) and any subsequent elements to the right (increases their indices).
- **protected void removeRange(int fromIndex, int toIndex)**  
// Removes from this list all of the elements whose index is between fromIndex, inclusive, and toIndex, exclusive. Shifts any succeeding elements to the left (reduces their index).
- **public boolean removeAll(Collection<?> c)**  
// Removes from this list all of its elements that are contained in the specified collection.
- **public boolean retainAll(Collection<?> c)**  
// Retains only the elements in this list that are contained in the specified collection.
- **public 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. The specified index indicates the first element that would be returned by an initial call to next. An initial call to previous would return the element with the specified index minus one.
- **public ListIterator<E> listIterator()**  
// Returns a list iterator over the elements in this list (in proper sequence).
- **public Iterator<E> iterator()**  
// Returns an iterator over the elements in this list in proper sequence.
- **public List<E> subList(int fromIndex, int toIndex)**  
// Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive. (If fromIndex and toIndex are equal, the returned list is empty.) The returned list is backed by this list, so non-structural changes in the returned list are reflected in this list, and vice-versa. The returned list supports all of the optional list operations.
- **public 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 excep-

tion.

- **public Spliterator<E> spliterator()**  
// Creates a late-binding and fail-fast Spliterator over the elements in this list.
- **public boolean removeIf(Predicate<? super E> filter)**  
// Removes all of the elements of this collection that satisfy the given predicate.
- **public void replaceAll(UnaryOperator<E> operator)**  
// Replaces each element of this list with the result of applying the operator to that element.
- **public void sort(Comparator<? super E> c)**  
// Sorts this list according to the order induced by the specified Comparator.