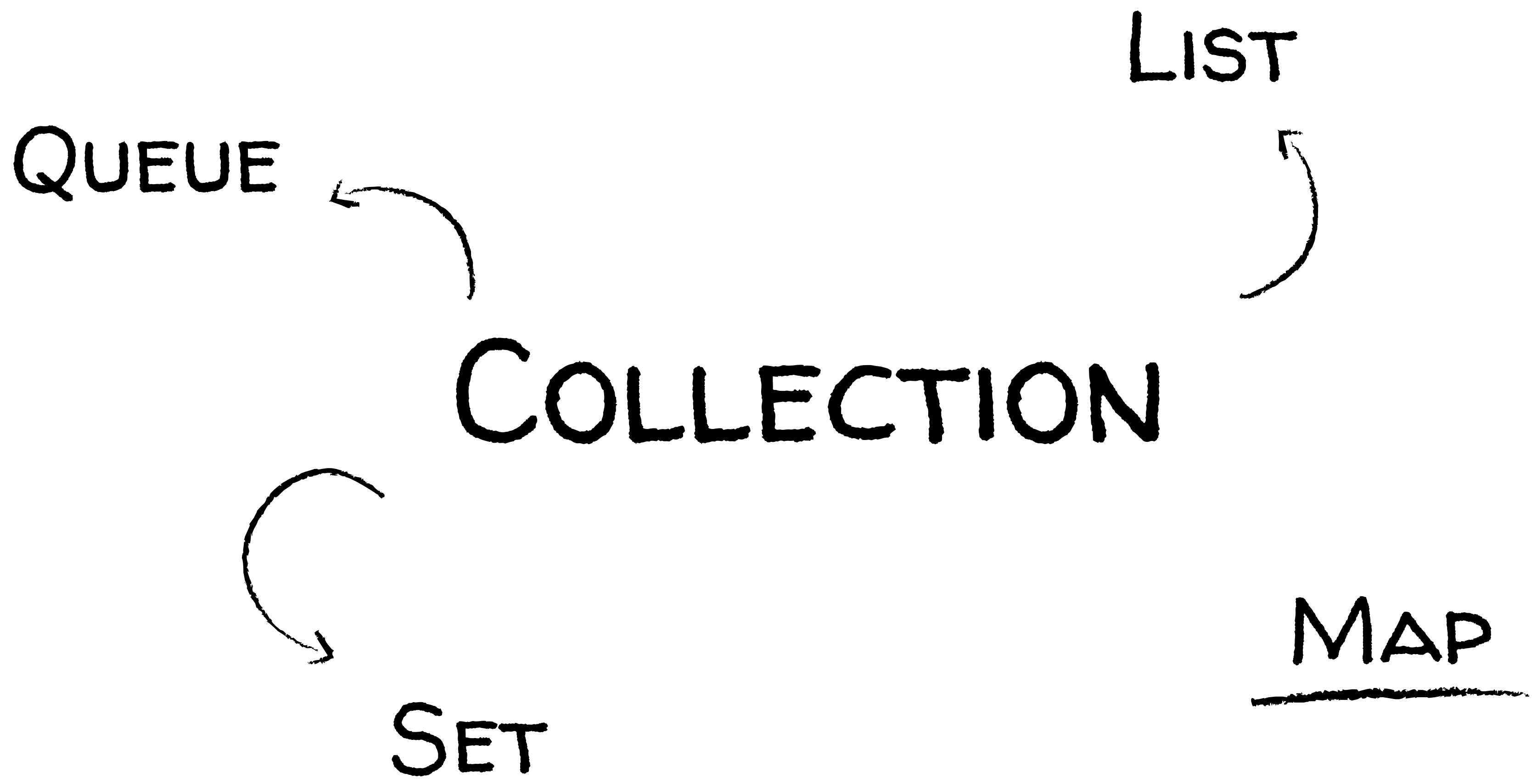

List
Queue
Collection
Map
Set

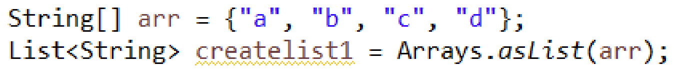
1.

* List

Allow duplicate elements
Can insert null elements
adding (and removing) is slower.

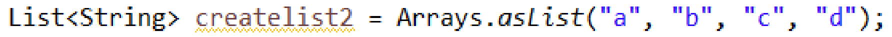
* List

An easy way to create a List is using the java.util.Arrays.asList method:



```java
String[] arr = {"a", "b", "c", "d"};
List<String> createlist1 = Arrays.asList(arr);
```

Or simply



```java
List<String> createlist2 = Arrays.asList("a", "b", "c", "d");
```

2.

SET

- doesn't allow duplicates. (return false;)

SET : HashSet

- doesn't guarantee the order or that the order will remain constant over time.
- adding and looking up elements is fast.
- accepts null values.

SET : TreeSet

- elements sorted and guarantees log(n) time cost
- doesn't add null values. (NullPointerException)

HaspSet

- boolean add(E e)

```
System.out.println(set.add("b")); // true
System.out.println(set.add("x")); // true
System.out.println(set.add("h")); // true
System.out.println(set.add("b")); // false
System.out.println(set.add(null)); // true
System.out.println(set.add(null)); // false
System.out.println(set); // [null, b, x, h]
```

- E remove()

```
// [20, 30, 30, 22]
deque.add(-1);      // true
// this will remove element at the first(head) postion
int retval = deque.remove();
System.out.println("Element removed is : " + retval);   // 20
// let us print all the elements available in deque
System.out.println(deque);  // [30, 30, 22, -1]
```

- E element()

```
// [30, 30, 22, -1]
System.out.println(deque.element());    // 30
deque.clear();
System.out.println(deque.element());    // java.util.NoSuchElementException
```

3.

Queue

- FIFO (first-in-first-out) way.
- LIFO (last-in-first-out)

FIFO : ArrayDeque

THROW AN EXCEPTION IF SOMETHING GOES WRONG:

- boolean add(E e)

```java
Queue<Integer> deque = new ArrayDeque<Integer>(5);
// use add() method to add elements in the deque
System.out.println("deque.add(20); "+ deque.add(20));    // true
System.out.println("deque.add(30); "+ deque.add(30));    // true
System.out.println("deque.add(30); "+ deque.add(30));    // true
System.out.println("deque.add(22); "+ deque.add(22));    // true
//System.out.println("deque.add(null); "+ deque.add(null)); // java.lang.NullPointerException
// let us print all the elements available in deque
System.out.println(deque);  // [20, 30, 30, 22]
```

- E remove()

```java
// [20, 30, 30, 22]
deque.add(-1);      // true
// this will remove element at the first(head) postion
int retval = deque.remove();
System.out.println("Element removed is : " + retval);   // 20
// let us print all the elements available in deque
System.out.println(deque);  // [30, 30, 22, -1]
```

- E element()

```java
// [30, 30, 22, -1]
System.out.println(deque.element());    // 30
deque.clear();
System.out.println(deque.element());    // java.util.NoSuchElementException
```

FIFO : ArrayDeque

RETURN NULL IF SOMETHING GOES WRONG:

- boolean offer(E e)

```
Queue<String> queue = new ArrayDeque<>();
System.out.println(queue.offer("a")); // true [a]
System.out.println(queue.offer("b")); // true [a, b]
System.out.println(queue.offer("b")); // true [a, b, b]
System.out.println(queue.offer(null)); // java.lang.NullPointerException
```

- E poll()

```
// [a, b, b]
System.out.println(queue.poll()); // a [b, b]
System.out.println(queue.poll()); // b [b]
System.out.println(queue.poll()); // b []
System.out.println(queue.poll()); // null []
```

- E peek()

```
// [a, b, b]
System.out.println(queue.peek()); // a
System.out.println(queue.peek()); // a
System.out.println(queue.peek()); // a
```

LIFO : ArrayDeque

- void push(E e)

```
ArrayDeque<String> stack = new ArrayDeque<>();
stack.push("a");    // [a]
stack.push("b");    // [b, a]
stack.push("c");    // [c, b, a]
stack.push(null);   // NullPointerException
```

- E pop()

```
System.out.println(stack.pop()); // c
System.out.println(stack.pop()); // b
System.out.println(stack.pop()); // a
System.out.println(stack.pop()); // NoSuchElementException
```

- E peek()

```
stack.push("a");    // [a]
stack.push("b");    // [b, a]
stack.push("c");    // [c, b, a]

System.out.println(stack.peek()); // c
System.out.println(stack.peek()); // c
System.out.println(stack.peek()); // c
```

4.

Map

- a map cannot contain duplicate keys. (Returns the old one)
- Map doesn't implement Collection

Map : HashMap

- doesn't guarantee the order
- accepts null values.

Map : TreeMap

- elements sorted and guarantees log(n) time cost
- doesn't add null values. (NullPointerException)

HashMap

- V put(K key, V value)

```
Map<String, Integer> map = new HashMap<>();
// Adding a key/value pair
System.out.println( map.put("oranges", 7) );    // null
System.out.println( map.put("apples", 5) );     // null
System.out.println( map.put("lemons", 2) );     // null
System.out.println( map.put("bananas", 7) );    // null
// Replacing the value of an existing key. Returns the old one
System.out.println( map.put("apples", 4) );     // 5
System.out.println( map.put(null, 8) );             // null
System.out.println( map.put("vegetable", null) );   // null
```

- V get(Object key)

```
// {null=8, oranges=7, bananas=7, vegetable=null, apples=4, lemons=2}
System.out.println( map.get("oranges") );   // 7
System.out.println( map.get("orangess") );  // null
System.out.println( map.get(null) );        // 8
```

- boolean containsKey(Object key)

```
// {null=8, oranges=7, bananas=7, vegetable=null, apples=4, lemons=2}
System.out.println( map.containsKey("apples") );    // true
System.out.println( map.containsKey("appless") );   // false
System.out.println( map.containsKey(null) );        // true
```

HashMap

- boolean containsValue(Object value)

```
// {null=8, oranges=7, bananas=7, vegetable=null, apples=4, lemons=2}
System.out.println( map.containsValue(5) );     // false
System.out.println( map.containsValue(7) );     // true
System.out.println( map.containsValue(null) );  // true
```

- V remove(Object key)

```
// {null=8, oranges=7, bananas=7, vegetable=null, apples=4, lemons=2}
// Removing the key/value pair and returning the value
System.out.println( map.remove("lemons") );     // 2
// Returns null if it can't find the key
System.out.println( map.remove("lemons") );     // null
System.out.println( map.remove(null) );         // 8
```

- Set<K> keySet() key)

```
Set<String> keys = map.keySet();
System.out.println(keys);
//[oranges, bananas, vegetable, apples]
```

- Collection<V> values()

```
Collection<Integer> values = map.values();
System.out.println(values);
//[7, 7, null, 4]
```

TreeMap

- V put(K key, V value)

```
Map<String, Integer> map = new TreeMap<>();
System.out.println( map.put("oranges", 7) );        // null
System.out.println( map.put("apples", 5) );         // null
System.out.println( map.put("lemons", 2) );         // null
System.out.println( map.put("bananas", 7) );        // null
System.out.println( map.put("bananas", 8) );        // 7
//System.out.println( map.put(null, 8) );           // NullPointerException
System.out.println( map.put("vegetable", null) );   // null
//{apples=5, bananas=8, lemons=2, oranges=7, vegetable=null}
```

- Set<K> keySet() key)

```
Set<String> keys = map.keySet();
System.out.println(keys);
//[apples, bananas, lemons, oranges, vegetable]
```

- Collection<V> values()

```
// {null=8, oranges=7, bananas=7, vegetable=null, apples=4, lemons=2}
System.out.println( map.containsKey("apples") );    // true
System.out.println( map.containsKey("appless") );   // false
System.out.println( map.containsKey(null) );        // true
```

TreeMap

- boolean containsValue(Object value)

```
// {null=8, oranges=7, bananas=7, vegetable=null, apples=4, lemons=2}
System.out.println( map.containsValue(5) );     // false
System.out.println( map.containsValue(7) );     // true
System.out.println( map.containsValue(null) );  // true
```

- V remove(Object key)

```
// {null=8, oranges=7, bananas=7, vegetable=null, apples=4, lemons=2}
// Removing the key/value pair and returning the value
System.out.println( map.remove("lemons") );     // 2
// Returns null if it can't find the key
System.out.println( map.remove("lemons") );     // null
System.out.println( map.remove(null) );         // 8
```

- Set<K> keySet() key)

```
Set<String> keys = map.keySet();
System.out.println(keys);
//[oranges, bananas, vegetable, apples]
```

- Collection<V> values()

```
Collection<Integer> values = map.values();
System.out.println(values);
//[5, 8, 2, 7, null]
```

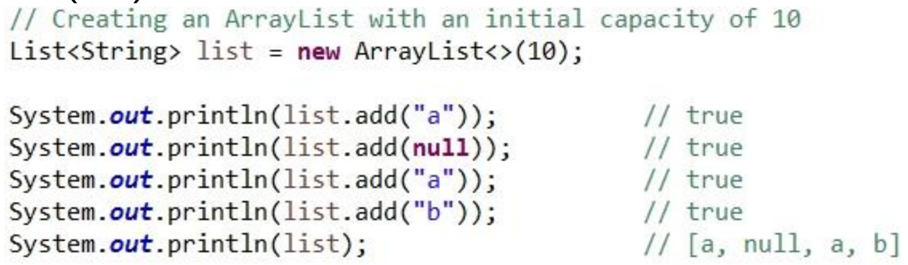
Key Points

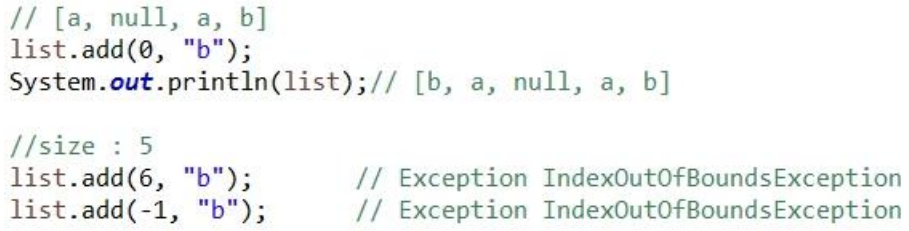
Collection	Interface	Implements Collection?	Allows duplicates?	Allows null values?	Ordered?
ArrayList	List	Yes	Yes	Yes	Yes (Insertion Order)
HashSet	List	Yes	No	Yes	No
TreeSet	List	Yes	No	No	Yes (Natural order or by Comparator)
ArrayDeque	Queue Deque	Yes	Yes	No	Yes (FIFO or LIFO)
HashMap	Map	No	Just for values	Yes	No
TreeMap	Map	No	Just for values	No	Yes (Natural order or by Comparator)

ArrayList

All Implemented Interfaces:
Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

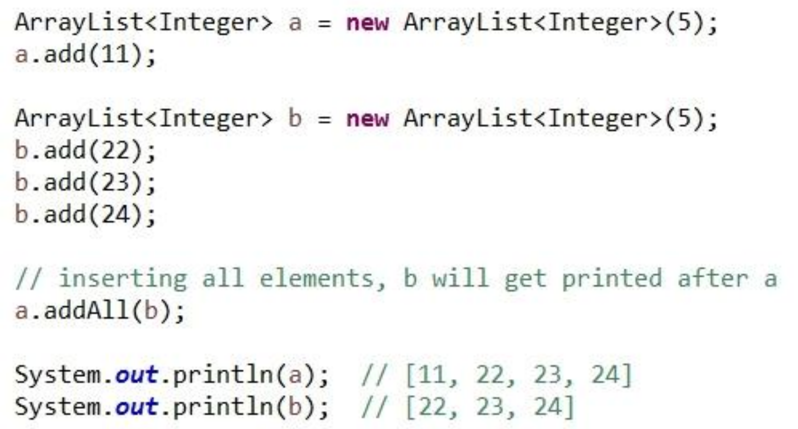
Modifier and Type	Method and Description

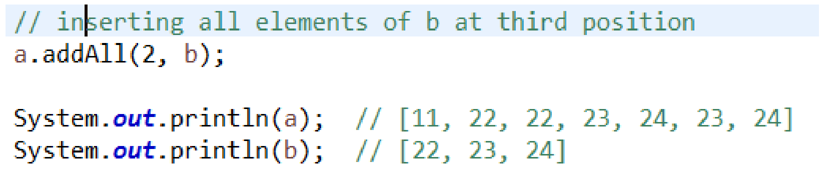
boolean	add(E e)
	
	`// Creating an ArrayList with an initial capacity of 10`
	`List<String> list = new ArrayList<>(10);`
	
	`System.out.println(list.add("a"));        // true`
	`System.out.println(list.add(null));       // true`
	`System.out.println(list.add("a"));        // true`
	`System.out.println(list.add("b"));        // true`
	`System.out.println(list);                 // [a, null, a, b]`

void	add(int index, E element)
	
	`// [a, null, a, b]`
	`list.add(0, "b");`
	`System.out.println(list);// [b, a, null, a, b]`
	
	`//size : 5`
	`list.add(6, "b");      // Exception IndexOutOfBoundsException`
	`list.add(-1, "b");     // Exception IndexOutOfBoundsException`

ArrayList

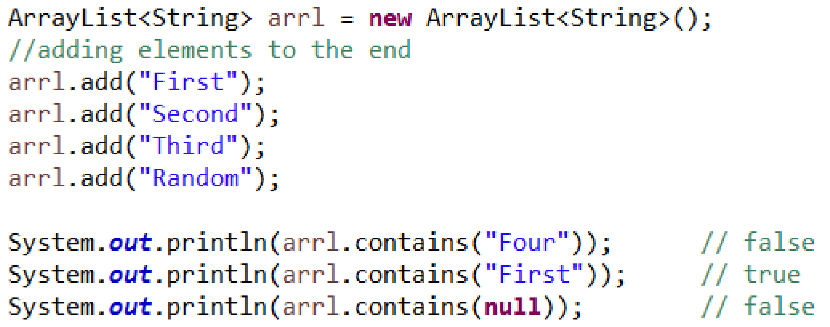
Modifier and Type	Method and Description

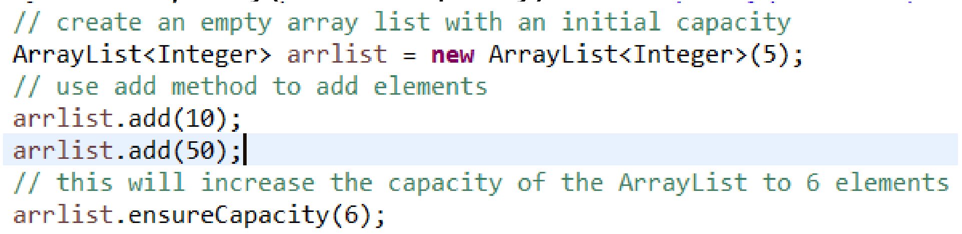
boolean	addAll(Collection<? extends E> c)
	
	`ArrayList<Integer> a = new ArrayList<Integer>(5);`
	`a.add(11);`
	
	`ArrayList<Integer> b = new ArrayList<Integer>(5);`
	`b.add(22);`
	`b.add(23);`
	`b.add(24);`
	
	`// inserting all elements, b will get printed after a`
	`a.addAll(b);`
	
	`System.out.println(a);  // [11, 22, 23, 24]`
	`System.out.println(b);  // [22, 23, 24]`

boolean	addAll(int index, Collection<? extends E> c)
	
	`// inserting all elements of b at third position`
	`a.addAll(2, b);`
	
	`System.out.println(a);  // [11, 22, 22, 23, 24, 23, 24]`
	`System.out.println(b);  // [22, 23, 24]`

ArrayList

Modifier and Type	Method and Description
void	clear()
boolean	clone()

boolean	contains(Object o)
	
	`ArrayList<String> arrl = new ArrayList<String>();`
	`//adding elements to the end`
	`arrl.add("First");`
	`arrl.add("Second");`
	`arrl.add("Third");`
	`arrl.add("Random");`
	
	`System.out.println(arrl.contains("Four"));      // false`
	`System.out.println(arrl.contains("First"));     // true`
	`System.out.println(arrl.contains(null));        // false`

void	ensureCapacity(int minCapacity)
	
	`// create an empty array list with an initial capacity`
	`ArrayList<Integer> arrlist = new ArrayList<Integer>(5);`
	`// use add method to add elements`
	`arrlist.add(10);`
	`arrlist.add(50);`
	`// this will increase the capacity of the ArrayList to 6 elements`
	`arrlist.ensureCapacity(6);`

ArrayList

Modifier and Type	Method and Description

void	forEach(Consumer<? super E> action)
	`List<String> items = new ArrayList<>();`
	`items.add("A");`
	`items.add("B");`
	`items.add("C");`
	`//lambda`
	`items.forEach(item->{`
	`    if("C".equals(item)){`
	`        System.out.println(item);              //Output : C`
	`    }`
	`});`
	`//lambda`
	`items.forEach(item->System.out.println(item));  //Output : A,B,C,D,E`
	`//method reference`
	`items.forEach(System.out::println);             //Output : A,B,C,D,E`

E	get(int index)

int	indexOf(Object o)
	`//[b, a, null, a, b]`
	`// Returning the index of the first match, -1 if not found`
	`System.out.println(list.indexOf("a"));          // 1`
	`System.out.println(list.indexOf("c"));          // -1`
	`System.out.println(list.indexOf(null));         // 2`

ArrayList

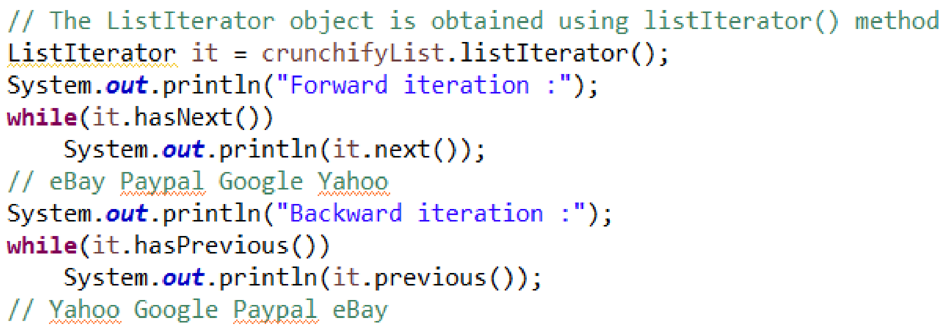
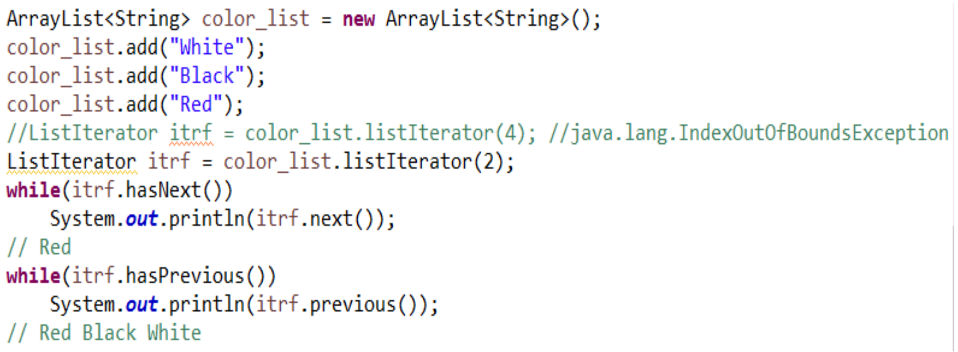
Modifier and Type	Method and Description

void	isEmpty()
	`// Creating an ArrayList with an initial capacity of 10`
	`List<String> listA = new ArrayList<>(10);`
	`System.out.println(listA.isEmpty());            // true`
	`listA.add(null);`
	`System.out.println(listA.isEmpty());            // false`

Iterator<E>	iterator()
	`// create list`
	`List<String> crunchifyList = new ArrayList<String>();`
	`// add 4 different values to list`
	`crunchifyList.add("eBay");`
	`crunchifyList.add("Paypal");`
	`crunchifyList.add("Google");`
	`crunchifyList.add("Yahoo");`
	`// iterate via "iterator loop"`
	`Iterator<String> crunchifyIterator = crunchifyList.iterator();`
	`while (crunchifyIterator.hasNext()) {`
	`    System.out.println(crunchifyIterator.next());`
	`}`

int	lastIndexOf(Object o)

ArrayList

Modifier and Type	Method and Description
ListIterator<E>	listIterator() 
ListIterator<E>	listIterator(int index) 

```java
// The ListIterator object is obtained using listIterator() method
ListIterator it = crunchifyList.listIterator();
System.out.println("Forward iteration :");
while(it.hasNext())
    System.out.println(it.next());
// eBay Paypal Google Yahoo
System.out.println("Backward iteration :");
while(it.hasPrevious())
    System.out.println(it.previous());
// Yahoo Google Paypal eBay
```

```java
ArrayList<String> color_list = new ArrayList<String>();
color_list.add("White");
color_list.add("Black");
color_list.add("Red");
//ListIterator itrf = color_list.listIterator(4); //java.lang.IndexOutOfBoundsException
ListIterator itrf = color_list.listIterator(2);
while(itrf.hasNext())
    System.out.println(itrf.next());
// Red
while(itrf.hasPrevious())
    System.out.println(itrf.previous());
// Red Black White
```

ArrayList

Modifier and Type	Method and Description

E	remove(int index)
	`// create an empty array list with an initial capacity`
	`ArrayList<String> arrList = new ArrayList<String>(5);`
	`// use add() method to add values in the list`
	`arrList.add("G");`
	`arrList.add("E");`
	`// Removes first occurrence of "E"`
	`System.out.println(arrList.remove(0));      // G`
	`System.out.println(arrList.remove(3));      // IndexOutOfBoundsException`

boolean	remove(Object o)
	`System.out.println(arrList.remove("e"));    // false`
	`System.out.println(arrList.remove("E"));    // true`
	`System.out.println(arrList.remove("A"));    // false`

boolean	removeAll(Collection<?> c)
	`System.out.println("First List :"+ color_list); // [White, Black, Red]`
	`System.out.println("Second List :"+ sample);    // [Green, Red, White]`
	`// remove all elements from second list if it exists in first list`
	`sample.removeAll(color_list);                   // [Green]`

boolean	removeIf(Predicate<? super E> filter)
	`//List of Colors : [White, Black, Red, White, Yellow, White]`
	`color_list.removeIf(t -> t.equals("White"));`
	`//Color list, after removing White colors : [Black, Red, Yellow]`

ArrayList

Modifier and Type	Method and Description

void	replaceAll(UnaryOperator<E> operator)
	`// color_list : [White, Black, Red, White, Yellow, White]`
	`UnaryOperator<String> unaryOpt = i -> "Whites";`
	`// Replace all colors with White color`
	`color_list.replaceAll(unaryOpt);`
	`System.out.println(color_list);`
	`// [Whites, Whites, Whites, Whites, Whites, Whites]`

boolean	retainAll(Collection<?> c)
	`// color_list : [White, Black, Red]`
	`// samples : [Green, Red, White]`
	`samples.retainAll(color_list);  // [Red, White]`