



# Collection **List**

Rawlabs Academy

# List

- The `List` interface provides a way to **store** and **ordered collection**.
- It is a **child interface** of `Collection`
- It is an ordered collection of object in which **duplicate values** can be stored.
- Since `List` preserves the insertion order, it allows **positional access** and **insertion of elements**.

# Methods

## List

```
List.java
☐ Inherited members (⌘F12) ☐ Anonymous Classes (⌘I) ☐ Lambdas (⌘L) 
List
  add(E): boolean ↑Collection
  add(int, E): void
  addAll(Collection<? extends E>): boolean ↑Collection
  addAll(int, Collection<? extends E>): boolean
  clear(): void ↑Collection
  contains(Object): boolean ↑Collection
  containsAll(Collection<?>): boolean ↑Collection
  copyOf(Collection<? extends E>): List<E>
  equals(Object): boolean ↑Collection
  get(int): E
  hashCode(): int ↑Collection
  indexOf(Object): int
  isEmpty(): boolean ↑Collection
  iterator(): Iterator<E> ↑Collection
  lastIndexOf(Object): int
  listIterator(): ListIterator<E>
  listIterator(int): ListIterator<E>
  of(): List<E>
  of(E): List<E>
  of(E, E): List<E>
  of(E, E, E): List<E>
  of(E, E, E, E): List<E>
  of(E, E, E, E, E): List<E>
  of(E, E, E, E, E, E): List<E>
  of(E, E, E, E, E, E, E): List<E>
  of(E, E, E, E, E, E, E, E): List<E>
  of(E, E, E, E, E, E, E, E, E): List<E>
  of(E, E, E, E, E, E, E, E, E, E): List<E>
  of(E...): List<E>
  remove(int): E
  remove(Object): boolean ↑Collection
  removeAll(Collection<?>): boolean ↑Collection
```

# List

## Hierarchy



# Abstract Collection

It is used to implement an unmodifiable collection, for which one need to only extend this

`AbstractCollection` class and implement only the iterator and the size methods.

```
AbstractCollection.java
☐ Inherited members (⌘F12) ☐ Anonymous Classes (⌘I) ☐ Lambdas (⌘L)
▼ (C) AbstractCollection
  (m) ? AbstractCollection()
  (m) ? add(E): boolean ↑Collection
  (m) ? addAll(Collection<? extends E>): boolean ↑Collection
  (m) ? clear(): void ↑Collection
  (m) ? contains(Object): boolean ↑Collection
  (m) ? containsAll(Collection<?>): boolean ↑Collection
  (m) ? finishToArray(T[], Iterator<?>): T[]
  (m) ? hugeCapacity(int): int
  (m) ? isEmpty(): boolean ↑Collection
  (m) ? iterator(): Iterator<E> ↑Collection
  (m) ? remove(Object): boolean ↑Collection
  (m) ? removeAll(Collection<?>): boolean ↑Collection
  (m) ? retainAll(Collection<?>): boolean ↑Collection
  (m) ? size(): int ↑Collection
  (m) ? toArray(): Object[] ↑Collection
  (m) ? toArray(T[]): T[] ↑Collection
  (m) ? toString(): String ↑Object
  (f) ? MAX_ARRAY_SIZE: int = Integer.MAX_VALUE - 8
```

# AbstractList

This class provides a skeletal implementation of the `List` interface to minimize the effort required to implement this interface backed by Random Access data store (such an array). For sequential access data (such as linked list), `AbstractSequentialList` should be used in preference to this class.

```
AbstractList.java
☐ Inherited members (⌘F12) ☐ Anonymous Classes (⌘I) ☐ Lambdas (⌘L) 
AbstractList
  (m) ? AbstractList()
  (m) ? add(E): boolean ↑AbstractCollection
  (m) ? add(int, E): void ↑List
  (m) ? addAll(int, Collection<? extends E>): boolean ↑List
  (m) ? clear(): void ↑AbstractCollection
  (m) ? equals(Object): boolean ↑Object
  (m) ? get(int): E ↑List
  (m) ? hashCode(): int ↑Object
  (m) ? indexOf(Object): int ↑List
  (m) ? iterator(): Iterator<E> ↑AbstractCollection
  (m) ? lastIndexOf(Object): int ↑List
  (m) ? listIterator(): ListIterator<E> ↑List
  (m) ? listIterator(int): ListIterator<E> ↑List
  (m) ? outOfBoundsMsg(int): String
  (m) ? rangeCheckForAdd(int): void
  (m) ? remove(int): E ↑List
  (m) ? removeRange(int, int): void
  (m) ? set(int, E): E ↑List
  (m) ? subList(int, int): List<E> ↑List
  (m) ? subListRangeCheck(int, int, int): void
  (f) ? modCount: int = 0
  > ? Itr
  > ? ListItr
```

# ArrayList

It provides us with **dynamic arrays** in java. Though, it may be slower than standard arrays but can be helpful in programs where **lots of manipulation** in the array needed.

40	55	63	17	22	68	89	97	89
0	1	2	3	4	5	6	7	8

<- Array Indices

Array Length = 9

First Index = 0

Last Index = 8

```
ArrayList.java
Inherited members (%F12) Anonymous Classes (%I) Lambdas (%L)
ArrayList
ArrayList()
ArrayList(Collection<? extends E>)
ArrayList(int)
add(E): boolean ↑AbstractList
add(E, Object[], int): void
add(int, E): void ↑AbstractList
addAll(Collection<? extends E>): boolean ↑AbstractCollection
addAll(int, Collection<? extends E>): boolean ↑AbstractList
batchRemove(Collection<?>, boolean, int, int): boolean
checkForComodification(int): void
checkInvariants(): void
clear(): void ↑AbstractList
clone(): Object ↑Object
contains(Object): boolean ↑AbstractCollection
elementAt(Object[], int): E
elementData(int): E
ensureCapacity(int): void
equals(Object): boolean ↑AbstractList
equalsArrayList(ArrayList<?>): boolean
equalsRange(List<?>, int, int): boolean
fastRemove(Object[], int): void
forEach(Consumer<? super E>): void ↑Iterable
get(int): E ↑AbstractList
grow(): Object[]
grow(int): Object[]
hashCode(): int ↑AbstractList
hashCodeRange(int, int): int
hugeCapacity(int): int
indexOf(Object): int ↑AbstractList
indexOfRange(Object, int, int): int
isClear(long[], int): boolean
isEmpty(): boolean ↑AbstractCollection
```

# ArrayList

## Work





# LinkedList

- **LinkedList** consists of nodes where each node contains data and a reference to the next node in the list
- Unlike an array, data is not stored in one contiguous block of memory and does not have a fixed size
- Instead, it consists of multiple blocks of memory at different addresses



```
LinkedList
m LinkedList()
m LinkedList(Collection<? extends E>)
m add(E): boolean ↑AbstractList
m add(int, E): void ↑AbstractSequentialList
m addAll(Collection<? extends E>): boolean ↑AbstractCollection
m addAll(int, Collection<? extends E>): boolean ↑AbstractSequentialList
m addFirst(E): void ↑Deque
m addLast(E): void ↑Deque
m checkElementIndex(int): void
m checkPositionIndex(int): void
m clear(): void ↑AbstractList
m clone(): Object ↑Object
m contains(Object): boolean ↑AbstractCollection
m descendingIterator(): Iterator<E> ↑Deque
m element(): E ↑Deque
m get(int): E ↑AbstractSequentialList
m getFirst(): E ↑Deque
m getLast(): E ↑Deque
m indexOf(Object): int ↑AbstractList
m isElementIndex(int): boolean
m isPositionIndex(int): boolean
m lastIndexOf(Object): int ↑AbstractList
m linkBefore(E, Node<E>): void
m linkFirst(E): void
m linkLast(E): void
m listIterator(int): ListIterator<E> ↑AbstractSequentialList
m node(int): Node<E>
m offer(E): boolean ↑Deque
m offerFirst(E): boolean ↑Deque
m offerLast(E): boolean ↑Deque
m outOfBoundsMsg(int): String
m peek(): E ↑Deque
```

# Representation **ArrayList** vs **LinkedList**



Array representation



# ArrayList Example

```
public class Main {  
    public static void main(String[] args) {  
        List<String> addresses = new ArrayList<>();  
        addresses.add("Milan");  
        addresses.add("London");  
        addresses.add("Guatemala");  
        addresses.add("London");  
  
        System.out.println(addresses.get(2));  
    }  
}
```

# LinkedList Example

```
public class Main {  
    public static void main(String[] args) {  
        List<String> addresses = new LinkedList<>();  
        addresses.add("Milan");  
        addresses.add("London");  
        addresses.add("Guatemala");  
        addresses.add("London");  
  
        System.out.println(addresses.get(2));  
    }  
}
```

# Immutable List Example

```
public class Main {  
    public static void main(String[] args) {  
        List<String> addresses = new LinkedList<>();  
        var immutableAddresses = Collections.unmodifiableList(addresses);  
        immutableAddresses.add("Texas");  
        addresses.add("Milan");  
        addresses.add("London");  
        addresses.add("Guatemala");  
        addresses.add("London");  
  
        System.out.println(addresses.get(2));  
    }  
}
```

# Stack

- The `Stack` class represents a **last-in-first-out** (LIFO) stack of objects
- It extends class `Vector` with 5 operations that allow a vector to be treated as a stack
- The usual **push** and **pop** operations are provided, as well as a method to peek at the top item on the stack, a method to test for whether the stack is empty, and a method to search the stack for an item and discover how far it is from the top
- A more complete and consistent set of LIFO stack operations is provided by the `Deque` **interface** and its implementations, which should be used in preference to this class.



# Stack Example

```
public class Main {  
    public static void main(String[] args) {  
        Stack<String> addresses = new Stack<>();  
        addresses.push("Milan");  
        addresses.push("London");  
        addresses.push("Guatemala");  
        addresses.push("London");  
  
        System.out.println(addresses.pop());  
    }  
}
```

# Task - **Array Merge**

Make a program to combine 2 arrays, then insert several objects into the array in the middle with the index entered.



# Task - **Play with Parking Area**

It is known that there is a parking lot that only contains 1 motorcycle in each row. Make a program to manage the parking lot so that it fills the farthest parking lot with the parking gate first and the motorbike closest to the parking gate can exit first.