Java List Interface

The List interface is part of the Java Collections Framework and represents an **ordered collection** of elements.
You **can access elements by their index**, add duplicates, and maintain the order of insertion. Since List is an interface, you cannot create a List object directly. Instead, you use a class that implements the List interface, such as:

ArrayList - like a resizable array with fast random access
LinkedList - like a train of cars you can easily attach or remove

Common List Methods

Method	Description
add()	Adds an element to the end of the list
get()	Returns the element at the specified position
set()	Replace the element at the specified position
remove()	Removes the element at the specified position
size()	Return the number of elements in the list

List vs. Array

Array	List
Fixed size	Dynamic size
Faster performance for raw data	More flexible and feature-rich
Not part of Collections Framework	Part of the Collections Framework

Java ArrayList

An ArrayList is like a resizable array.
It is part of the **java.util package** and implements the List interface.

The difference between a built-in array and an ArrayList in Java, is that the **size** of an array cannot be modified (if you want to add or remove elements to/from an array, you must create a new one). While **elements can be added and removed from an ArrayList whenever you want**.

- Create an ArrayList

To use an ArrayList, you must first import it from java.util:

Example

Create an **ArrayList object** called **cars** that will **store strings**:

```
import java.util.ArrayList; // import the ArrayList class

// Create an ArrayList object

ArrayList<String> cars = new ArrayList<String>();
```

Now you can **use methods** like add(), get(), set(), and remove() to manage your list of elements.

Add Elements

To add elements to an ArrayList, use the add() method:

Example

```
import java.util.ArrayList;

public class Main {
  public static void main(String[] args) {
    ArrayList<String> cars = new ArrayList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("Mazda");
    System.out.println(cars);   }}
```

Output

```
[Volvo, BMW, Ford, Mazda]
```

You can also **add an element at a specified position by referring to the index number**:

Example

```
import java.util.ArrayList;

public class Main {
  public static void main(String[] args) {
    ArrayList<String> cars = new ArrayList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");

    // Insert element at the beginning of the list (0)

    cars.add(0, "Mazda");
    System.out.println(cars);   }}
```

Output

[Mazda, Volvo, BMW, Ford]

An **ArrayList keeps elements in the same order you add them**, so the first item you add will be at index 0, the next at index 1, and so on.

Access an Element

To **access an element** in the ArrayList, use the **get()** method and refer to the index number:

Example

```
cars.get(0);  // Get the first element
```

Change an Element

To modify an element, use the set() method and refer to the index number:

Example

```
cars.set(0, "Opel");
```

Remove an Element

To remove an element, use the remove() method and refer to the index number:

Example

```
cars.remove(0);
```

To **remove all the elements** in the ArrayList, use the clear() method:

Example

```
cars.clear();
```

ArrayList Size

To find out how many elements an ArrayList has, use the size method:

Example

```
cars.size();
```

Loop Through an ArrayList

Loop through the elements of an ArrayList with a for loop, and use the size() method to specify how many times the loop should run:

Example

```
public class Main {
  public static void main(String[] args) {

    ArrayList<String> cars = new ArrayList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("Mazda");
    for (int i = 0; i < cars.size(); i++) {
      System.out.println(cars.get(i));   }  }}
```

You can also loop through an ArrayList with the **for-each** loop:

Syntax

```
for (dataType variableName : arrayOrCollection) {
  // Body of the loop
}
```

Example

```
public class Main {
  public static void main(String[] args) {
    ArrayList<String> cars = new ArrayList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("Mazda");
    for (String i : cars) {
      System.out.println(i);   } }}
```

Other Types

Elements in an **ArrayList** are objects. In the examples above, we created elements (objects) of type **"String"**. Remember that a **String in Java is an object** (not a primitive type). To use other types, such as int, you must specify an equivalent wrapper class: **Integer**. For other primitive types, use: **Boolean** for boolean, **Character** for char, **Double** for double, etc.

Example

Create an ArrayList to store numbers (add elements of type Integer):

```
import java.util.ArrayList;

public class Main {
  public static void main(String[] args) {
    ArrayList<Integer> myNumbers = new ArrayList<Integer>();
    myNumbers.add(10);
    myNumbers.add(15);
    myNumbers.add(20);
    myNumbers.add(25);
    for (int i : myNumbers) {
      System.out.println(i);  } }}
```

Sort an ArrayList

Another useful class in the java.util package is the **Collections class**, which includes the **sort() method** for sorting lists **alphabetically** or **numerically**:

Example Sort an ArrayList of Strings:

```
import java.util.ArrayList;
import java.util.Collections;  // Import the Collections class

public class Main {
  public static void main(String[] args) {
    ArrayList<String> cars = new ArrayList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("Mazda");

    Collections.sort(cars);  // Sort cars

    for (String i : cars) {
       System.out.println(i);  } }}
```

Example Sort an ArrayList of Integers:

```
import java.util.ArrayList;
import java.util.Collections;  // Import the Collections class

public class Main {
  public static void main(String[] args) {
    ArrayList<Integer> myNumbers = new ArrayList<Integer>();
    myNumbers.add(33);
    myNumbers.add(15);
    myNumbers.add(20);
    myNumbers.add(34);
    myNumbers.add(8);
    myNumbers.add(12);

    Collections.sort(myNumbers);  // Sort myNumbers

    for (int i : myNumbers) {
```

```
    System.out.println(i);   } }}
```

The var Keyword

From Java 10, you can use the **var keyword** to **declare an ArrayList variable without writing the type twice**. The compiler figures out the type from the value you assign.
This makes code shorter, **but many developers still use the full type for clarity**. Since var is valid Java, you may see it in other code, so it's good to know that it exists:

Example

```
// Without var
ArrayList<String> cars = new ArrayList<String>();

// With var
var cars = new ArrayList<String>();
```

Java Generics

Generics allow you to **write classes**, **interfaces**, and **methods that work with different data types**, without having to **specify the exact type in advance**.
This makes your code more flexible, reusable, and type-safe.

Why Use Generics?

- **Code Reusability:** Write one **class or method** that works with **different data types**.
- **Type Safety:** Catch type errors at compile time instead of runtime.
- **Cleaner Code:** No need for casting when retrieving objects.

Generic Class Example

You can create a class that works with different data types using generics:

```
class Box<T> {
  T value; // T is a placeholder for any data type

  void set(T value) {
    this.value = value;   }

  T get() {
```

```
    return value;   }

  public static void main(String[] args) {

    // Create a Box to hold a String

    Box<String> stringBox = new Box<>();

    stringBox.set("Hello");
    System.out.println("Value: " + stringBox.get());

    // Create a Box to hold an Integer
    Box<Integer> intBox = new Box<>();
    intBox.set(50);
    System.out.println("Value: " + intBox.get());   }}
```

Output

Value: Hello
Value: 50

T is a generic type parameter. It's like a **placeholder** for data type.

- When you **create a Box<String>**, **T becomes String**.
- When you **create a Box<Integer>**, **T becomes Integer**.

This way, the same class can be reused with different data types without rewriting the code.

Generic Method Example

You can also **create methods** that work with any data type using generics:

```
public class Main {

// Generic method: works with any type T

  public static <T> void printArray(T[] array) {
    for (T item : array) {
      System.out.println(item);    }   }

  public static void main(String[] args) {
```

```
    // Array of Strings
    String[] names = {"Jenny", "Liam"};

    // Array of Integers
    Integer[] numbers = {1, 2, 3};

    // Call the generic method with both arrays
    printArray(names);
    printArray(numbers);   }}
```

Output

```
Jenny
Liam
1
2
3
```

Example Explained

- <T> is a generic type parameter - it **means the method can work with any type**: String, Integer, Double, etc.
- The method printArray() takes an array of **type T** and prints every element.
- When you call the method, Java figures out what T should be based on the argument you pass on.

This is useful when you want to write one method that works with multiple types, instead of repeating code for each one.

Bounded Types

You can use the **extends keyword** to limit the types a generic class or method can accept.

For example, you can require that the type must be a **subclass of Number**:

```
class Stats<T extends Number> {
  T[] nums;

  // Constructor
  Stats(T[] nums) {
    this.nums = nums;   }
```

```
  // Calculate average
    double average() {
    double sum = 0;
    for (T num : nums) {
      sum += num.doubleValue();     }
    return sum / nums.length;   } }

public class Main {
  public static void main(String[] args) {

    // Use with Integer
    Integer[] intNums = {10, 20, 30, 40};
    Stats<Integer> intStats = new Stats<>(intNums);
    System.out.println("Integer average: " + intStats.average());

    // Use with Double
    Double[] doubleNums = {1.5, 2.5, 3.5};
    Stats<Double> doubleStats = new Stats<>(doubleNums);
    System.out.println("Double average: " + doubleStats.average());   }}
```

Output

```
Integer average: 25.0
Double average: 2.5
```

Even though int values are used in the first case, the .doubleValue() method converts them to double, so the result is shown with a decimal point.

Example Explained

- <T extends Number>: Restricts T to only work with numeric types like Integer, Double, or Float.
- .doubleValue(): Converts any number to a double for calculation.
- Works for any array of numbers if the elements are subclasses of Number.