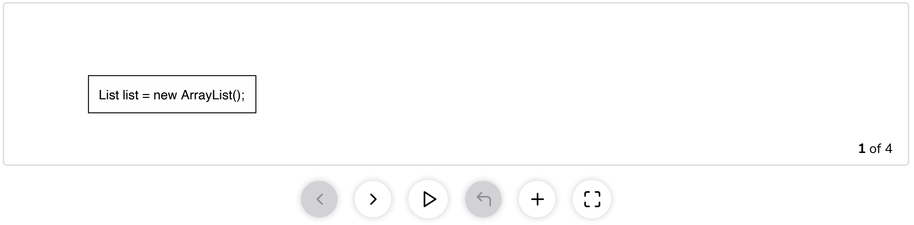
ArrayList is the most widely used implementation of the **List** interface. Some of the salient features of an **ArrayList** are:

1. Elements are stored in the order of insertion.
2. It allows the storage of duplicate elements.
3. ArrayList also supports null elements.

Internal implementation of ArrayList

An **ArrayList** stores data in a resizable array. Before Java 8, when an **ArrayList** was created, an array of default size ten was created internally. Now, when an **ArrayList** is created, an array of size zero is created. Only when the first element is inserted does the array size change to ten. This is called lazy initialization, and it saves a lot of memory.

Before adding an element in **ArrayList**, its capacity is checked. If the internal array is full, then a new array of size $\left(n + \frac{n}{2} + 1\right)$ is created (e.g., if the capacity is ten, then a new array of size 16 will be created). The elements from the old array will be copied to the new array. This increases the capacity of an **ArrayList**, which is a time-consuming process.



Time complexities for ArrayList operations

Let's see what the time complexities are for different operations in an **ArrayList**.

Adding an element

Since an array backs an **ArrayList**, the addition of an element takes $O(1)$ time in most of the cases. It will take more time if the **ArrayList** is full and needs to be resized. In that scenario, a new array will be created, and elements will be copied from the old array to the new array.

Removing an element

The remove operation has $O(n)$ complexity in the worst case and $O(1)$ in the best case. There are two overloaded versions of the `remove()` method in ArrayList:

1. The first one takes the index of the element that needs to be removed as input. The element can be found in $O(1)$ time using the index, but when the element is removed, the other elements need to be moved to the left. So, if the last element is removed the complexity will be $O(1)$ otherwise, $O(n)$.
2. In the second case, the `remove()` method takes the element that needs to be removed as input. The array is scanned from the beginning to find the first occurrence of that element, and then it is removed. This has a complexity of $O(n)$.

Fetching an element

Fetching an element from an array using an index is $O(1)$ constant-time operation. So, fetching an element from an **ArrayList** takes constant time.

Creating an ArrayList

There are three ways to create an **ArrayList**:

Using the no-arg constructor

The default constructor does not take any argument and creates a List of size zero. Below is the syntax to create **ArrayList** using the default constructor.

```
List list = new ArrayList();
```

Using the constructor that takes initial capacity

We can also provide an initial capacity while creating an **ArrayList**. The benefit is that if we know that our **ArrayList** will contain a minimum of 100 elements, then we can create the **ArrayList** with a size of 100. Thus, our **ArrayList** will not require frequent resizing.

```
List list = new ArrayList(50);
```

Using existing Collection

An **ArrayList** can also be created using an existing Collection. The newly created **ArrayList** will contain all the elements in the same order in the original collection.

```
List list = new ArrayList(oldList);
```