Arrays: It is a data structure that allows you to store sequence of values all of same time.

We can instantiate a new array using new Keyword. We have array declaration on the left side of = array creation expression on the right side.

Array Creation

```
int[] integerArray = new int[10];
```

The array initializer

```
int[] firstFivePositives = {1, 2, 3, 4, 5};
String[] names = {"Andy", "Bob", "Charlie", "David", "Eve"};
```

- You cannot change the size of an array once the array is instantiated.
- We can't add or delete elements. We can only assign values to one of the elements assigned in the array.
- In the example we use new keyword to create an array. And specified the length of the array as 10.

Array initializer as an anonymous array:

- Above is an example of anonymous array initializer which can be only used in declaration statement. Here the size of array is determined by the number of elements specified in {}
- We cannot declare an array in one line And initialize the array elements in another line without specifying the size of an array. However, we can Initialize an array as an anonymous array in a single line.

Array is a special class in Java which inherits from java.lang.Object. Java array type is very basic and it comes with limited built in functionality. However, Java provides a helper class named Java.Util.Arrays providing all the common functionalities and some static methods on arrays which can be used.

Array initialization to default values –

- For primitive types, this is zero for any kind of numeric primitive, like int, double or short.
- For booleans, the default value will be false.
- And for any class type, the elements will be initialized to null.

Reference types versus value types –

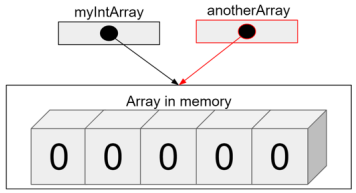
When you assign an object to a variable, the variable becomes reference to that object.

```
int[] myIntArray = new int[5];
int[] anotherArray = myIntArray;

System.out.println("myIntArray= " + Arrays.toString(myIntArray));
System.out.println("anotherArray= " + Arrays.toString(anotherArray));

anotherArray[0] = 1;

System.out.println("after change myIntArray= " + Arrays.toString(myIntArray));
System.out.println("after change anotherArray= " + Arrays.toString(anotherArray));
```



- Here we have instantiated an array called myIntArray using new keyword which is of length 5 elements.
- Here the variables myIntArray and anotherArray are referencing to the same array in memory.

Variable arguments. - We can replace the brackets after the String type in the main function with three periods. This is a special designation in Java that means Java will either take 0 or 1 or many strings as arguments to this method.

However, we need to remember 2 important concepts while using variable arguments.

- There can be only one variable argument in a method.
- The variable argument must be the last argument.

```arduino
Example.java:4: error: varargs parameter must be the last parameter
    public void printInfo(String message, int... numbers, String... names) {
                                                          ^
1 error
```

The above code throws an error because there is more than one variable argument as a parameter in the method.

```java
public void printInfo(String message, String[] names, int... numbers) {
    // Method body
}
```

We can resolve the error by having only one variable argument in the method as a parameter. And that parameter will be the last argument.

Two-dimensional and multi-dimensional arrays : Java supports 2 dimensional and three-dimensional arrays of varying dimensions using a concept called nested array.

Multidimensional arrays are implemented as arrays of arrays. We can specify each additional index using another set of square brackets.

- As shown in the below example, when we allocate memory for multi-dimensional array, we only need to specify the memory of first leftmost dimension. We can allocate the remaining dimensions separately.
- The only advantage of allocating the second dimensions Arrays individually is when we have different sizes of second dimension. However, the use of uneven or irregular multidimensional arrays may be not appropriate for many applications.

We can also initialize the two-dimensional array without specifying the size of nested arrays.

```
int[][] myDoubleArray;
int[] myDoubleArray[];
```

- These are the two different ways of initializing a 2-dimensional array. However, first method is preferred over the second to avoid confusion.

In this example, we have an outer array with three elements.

```
Object[] multiArray = new Object[3];
multiArray[0] = new Dog[3];
multiArray[1] = new Dog[3][];
multiArray[2] = new Dog[3][][];
```

The first element is itself a single-dimension array.

The second element is a two-dimensional array.

And lastly, the third element is a three-dimensional array.

Limitations.:

- One of the biggest limitations of arrays is that we won't be able to change the number of elements in an array after being instantiated.

Java has an entire library for Java containers which they called Collections. They take the arrays to the next level, they allow you to change the number of elements defined in an array. Two of the most important classes are.

- ArrayList
- LinkedList.

List: List is a special type in Java called interface., We can say that List interface describes a set of method signatures that all List classes are expected to have.

- It is an ordered collection, also known as sequence. The user of this interface has precise control over where in the list each element is inserted, and users can access elements by their integer index and search for the elements in the list. Unlike sets, lists typically allow duplicate elements.

ArrayList: Resizable array implementation of the List interface. This implements all optional list operations and permits all elements including null. In addition to implementing List interface, this class provides method to manipulate the size of array that is used internally to store the list.

Each arrayList instance has a capacity. The capacity is the size of the array used to store the elements in the list. It is always at least as large as the list size. As the elements are added to the array list, its capacity grows automatically.

```
ArrayList objectLists = new ArrayList();
objectLists.add(new GroceryItem( name: "butter"));
objectLists.add("grapes");
System.out.println(objectLists);
/*
[GroceryItem[name=butter, type=Dairy, count=1], grapes]
whenever we create a list we need to specify the generic type inorder the lists to accept only those.
Here since we did not specify the ArrayLists type hence it accepts everything,
either new GroceryItem object or even just a string list object we created in first scenario.
The ArrayLists is immutable we can add or delete the elements from our list as required.
        */
ArrayList<GroceryItem> groceryLists1 = new ArrayList<>();
groceryLists1.add(new GroceryItem( name: "butter"));
groceryLists1.add( index: 0, new GroceryItem( name: "Cheese"));
groceryLists1.add(new GroceryItem( name: "grapes",  type: "produce",  count: 10));
System.out.println(groceryLists1);
groceryLists1.remove( index: 1);
System.out.println(groceryLists1);
/*
[GroceryItem[name=Cheese, type=Dairy, count=1], GroceryItem[name=butter, type=Dairy, count=1], GroceryItem[name=grapes, type=
[GroceryItem[name=Cheese, type=Dairy, count=1], GroceryItem[name=grapes, type=produce, count=10]]
Angle brackets include record GroceryItem now, which will now only consider groceryItems,
ere we are telling the compiler to type check for the data it adds to the list
 */
```

```
Object[] groceryArray = new Object[3];
groceryArray[0] = new GroceryItem( name: "milk");
groceryArray[1] = new GroceryItem( name: "apples",  type: "Produce",  count: 5);
groceryArray[2] = "Oranges";
System.out.println(Arrays.toString(groceryArray));
/*
[GroceryItem[name=milk, type=Dairy, count=1], GroceryItem[name=apples, type=Produce, count=5], Oranges]
 */
GroceryItem[] groceryLists = new GroceryItem[3];
groceryLists[0] = new GroceryItem( name: "milk");
groceryLists[1] = new GroceryItem( name: "apples",  type: "Produce",  count: 5);
groceryLists[2] = new GroceryItem( name: "apples",  type: "Produce",  count: 5);
System.out.println(Arrays.toString(groceryLists));
/*
[GroceryItem[name=milk, type=Dairy, count=1], GroceryItem[name=apples, type=Produce, count=5],
GroceryItem[name=apples, type=Produce, count=5]]
Here we set the groceryLists to instance of new GroceryItem instead of Object.
So it doesn't accept string, it only accepts instances of grocery Items.
Here the length is fixed and cannot be changed after instatiation
 */
```

- The first screenshot is using array list which is resizable. The second screenshot uses integer array where we need to instantiate the size at the beginning of declaration. The array list also lets us specify the type of elements/Parameters in the list. See the example. Grocery items are specified in the array list.
- Also, in order to print the arrays, we need a helper class From Java (Ararys.toString()). However, We can directly print array list.

Common mistake while using array list - Whenever we used parameterized class of Arraylists, we need to make sure we have <> diamond operator at the end of instantiation.

Unmodifiable list: The List.of and List.copyOf - List created using these static factory methods are immutable. That is, they are unmodifiable, i.e., elements cannot be added, removed or replaced. Calling any muter method on this list will always cause **UnsupportedOperationException** through be thrown.

```
String[] items = {"apples", "bananas", "milk", "eggs"};
List<String> list = List.of(items);
System.out.println(list);
// causes below error for list.add("yogurt");
/*
List.Of() This method transformed array of strings to list of strings.
[apples, bananas, milk, eggs]
lists belong to class java.util.ImmutableCollections$ListN
Exception in thread "main" java.lang.UnsupportedOperationException
    */
```

```
ArrayList<String> a1 = new ArrayList<>(list);
a1.add("yogurt");
System.out.print(a1);
/*
[apples, bananas, milk, eggs, yogurt]
 */
```

Arrays versus arrayLists:

Feature	array	ArrayList
primitives types supported	Yes	**No**
indexed	Yes	Yes
ordered by index	Yes	Yes
duplicates allowed	Yes	Yes
nulls allowed	Yes, for non-primitive types	Yes
resizable	**No**	Yes
mutable	Yes	Yes
inherits from java.util.Object	Yes	Yes
implements List interface	No	**Yes**

An ArrayList can be instantiated by passing another list to it as we show below - We can use the List.of() factory method, which uses variable arguments, to create a pass-through immutable list.

Arrays can store primitive data types (such as int, car and double) whereas ArrayList can only store objects, not primitive types. For an example, if you want to store integers in array list, you will use Integer wrapper object instead of int primitives.

java Copy code

```java
ArrayList<Integer> numbers = new ArrayList<Integer>();
numbers.add(1); // Here, 1 is autoboxed into an Integer object
```

java Copy code

```java
int[] items = {1, 2, 3};
Integer[] boxedItems = Arrays.stream(items).boxed().toArray(Integer[]::new);
List<Integer> list = List.of(boxedItems);
```

	Accessing Array Element data	Accessing ArrayList Element data
	Example Array: `String[] arrays = {"first", "second", "third"};`	Example ArrayList: `ArrayList<String> arrayList = new ArrayList<>(List.of("first", "second", "third"));`
Index value of first element	`0`	`0`
Index value of last element	`arrays.length - 1`	`arrayList.size() - 1`
Retrieving number of elements:	`int elementCount = arrays.length;`	`int elementCount = arrayList.size();`
Setting (assigning an element)	`arrays[0] = "one";`	`arrayList.set(0, "one");`
Getting an element	`String element = arrays[0];`	`String element = arrayList.get(0);`

Array as an array list –

```
String[] originalArray = new String[] {"First", "Second", "Third"};
var originalList = Arrays.asList(originalArray);
```

- The Arrays.asList method returns an ArrayList, backed by an array.
- The code uses the Arrays.asList method, passing it the array, and assign it to a variable, using the var keyword, to a variable named originalList.
- Any change made to the list is a change to the array that backs it. This means an array list created by this method is not resizeable.

Using Arrays.asList	**Using List.of**
Returned List is NOT resizeable, but is mutable.	**Returned List is IMMUTABLE.**
`var newList = Arrays.asList("Sunday", "Monday", "Tuesday");`	`var listOne = List.of("Sunday", "Monday", "Tuesday");`
`String[] days = new String[] {"Sunday", "Monday", "Tuesday"};` `List<String> newList = Arrays.asList(days);`	`String[] days = new String[] {"Sunday", "Monday", "Tuesday"};` `List<String> listOne = List.of(days);`

In this example, newList is a List returned by Arrays.asList(), but when we try to add an element to it, we get an UnsupportedOperationException because the list's size cannot be changed. However, we can modify the first element of the list returned by Arrays.asList() using the set() method. The list is mutable, so we can change its elements.

In this example, listOne is an immutable list returned by List.of(). Trying to add an element or modifying an element using set() also results in an UnsupportedOperationException.

- // Create a new ArrayList and pass the elements from the immutable list
- List<String> mutableList = new ArrayList<>(listOne);

In this example, we first create an immutable list using List.of(). Then, we create a new ArrayList named mutableList and pass the elements of the immutable list to it using the constructor ArrayList<>(listOne). Now, mutableList is a mutable list backed by an ArrayList, and we can modify its elements using methods like set().

Arrays vs ArrayList vs LinkedList:

	Linked List		ArrayList	
Operation	**Worst Case**	**Best Case**	**Worst Case**	**Best Case**
add()	O(1)		O(1)*	
add(int index, E element)	O(n)	O(1)	O(n)	O(1)*
contains(E element)	O(n)	O(1)	O(n)	O(1)
get(int index)	O(n)	O(1)	O(1)	
indexOf(E Element)	O(n)	O(1)	O(n)	O(1)
remove(int index)	O(n)	O(1)	O(n)	O(1)
remove(E element)	O(n)	O(1)	O(n)	
set(int index, E element)	O(n)	O(1)	O(1)	

- O(1) - constant time - operation's cost (time) should be constant regardless of no. of elements.
- O(n) - linear time - operation's cost (time) will increase linearly with the number of elements n.
- O(1)* - constant amortized time - somewhere between O(1) and O(n), but closer to O(1) as efficiencies are gained.
- Use ArrayList when frequent random access to elements by index is required and the number of insertions or deletions in the middle is relatively low.
- Use LinkedList when frequent insertions or deletions in the middle of the list are required and random access to elements by index is less important.

Additionally, an array list is implemented on top of an array, but a linked list is a doubly linked list. Both implement all of list method but linked list also implements the queue and stack methods as well.

ArrayList:

- Elements are stored in continuous blocks of memory.
- Provides fast random access to elements using indexed based retrieval.
- Slower for adding and removing elements from the middle due to potential array resizing and element shifting.
- Requires less memory overhead per element. Since it only needs to store elements and an array. However, it waste a lot of memory if the initial capacity is set too high.
- Often used in scenarios where memory efficiency is a concern. It is usually better choice for a list, especially if the list is used predominantly for storing and reading data.

LinkedList:

- Internally implemented as double linked list. Each element is stored in a separate node and nodes are linked to each other.
- Provides fast insertion and deletion of elements in the middle of the list because it just requires updating the link between nodes.
- Slower access time for retrieving elements by index because it needs to travel the list from head to tail or from tail to head to reach the desired index.
- It requires more memory overhead per element because each element is stored in a separate node and each node has additional pointers to the previous and next nodes.
- Used in scenarios where the number of elements are unpredictable and changes frequently.

Linked lists can be used as a stack, a queue and a general list or as a double linked list.

- **Stack:** A stack follows last in first out (LIFO) principle, you can use linked list as a stag by performing stack operation in its elements like push, pop or peak.
- **Queue**: A queue follows first in First Out FIFO principle, You can use linked list as a queue by performing queue operations such as enqueue, dequeue, peak.
- **General list**: Unlinked links can be used as a general purpose list by adding, removing, getting and setting elements.

Method	Stack Operation	Queue Operation	Notes
`addFirst(E e)`	✓		Adds element to the beginning (stack push)
`addLast(E e)`		✓	Adds element to the end (queue enqueue)
`getFirst()`	✓		Retrieves first element (stack peek)
`getLast()`		✓	Retrieves last element (queue peek)
`pollFirst()`	✓		Removes and retrieves first element (stack pop)
`pollLast()`		✓	Removes and retrieves last element (queue dequeue)
`removeFirst()`	✓		Removes and returns first element (stack pop)
`removeLast()`		✓	Removes and returns last element (queue dequeue)
`offerFirst(E e)`	✓		Adds element to the beginning (stack push) and returns success status
`offerLast(E e)`		✓	Adds element to the end (queue enqueue) and returns success status

Stack:

- Push: Add an element to the top of the stack (addFirst() or offerFirst()).
- Pop: Remove and return the element from the top of the stack (pollFirst() or removeFirst()).
- Peek: Return the element from the top of the stack without removing it (getFirst()).

Queue:

- Enqueue: Add an element to the end of the queue (addLast() or offerLast()).
- Dequeue: Remove and return the element from the front of the queue (pollFirst() or removeFirst()).
- Peek: Return the element from the front of the queue without removing it (getFirst()).

General List:

- Add: Add an element at a specified position in the list (add(int index, E element)).
- Remove: Remove an element from the list based on its index (remove(int index)).
- Get: Retrieve an element from the list based on its index (get(int index)).
- Set: Update an element at a specified position in the list (set(int index, E element)).

ListIterator:

- We can use the traditional for loop and an index, to index into a list.
- We can use the enhanced for loop and a collection, to step through the elements, one at a time.

Java provides other means to traverse list: One is Iterator and the other is ListIterator. Iterator can be thought of as database cursor.

- When we create an instance of an iterator, we can call next method to get the next element in the list. We can use hasnext method to check if there are any elements remained to be processed.
- Difference between Iterator versus ListIterator is: iterator only moves forward and only supports the remove method whereas the list iterator can be used to go both forward (hasNext(), next()) and backward (hasPrevious(), previous()) and in addition to the remove method it also supports add and set methods.
- Linked list provides support for both forward and backward iteration through its elements, Using ListIterator, making it suitable for scenarios where bidirectional travel is required.