Arrays vs ArrayLists

This slide demonstrates that Arrays and ArrayLists have more in common than they don't.

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**

Instantiating without Values

Instantiating Arrays	Instantiating ArrayLists
`String[] array = new String[10];`	`ArrayList<String> arrayList = new ArrayList<>();`
An array of 10 elements is created, all with null references. The compiler will only permit Strings to be assigned to the elements.	An empty ArrayList is created. The compiler will check that only Strings are added to the ArrayList.

On this slide, I show the differences when creating a new instance of an array compared to a new instance of an ArrayList.

An array requires square brackets in the declaration.

On the right-hand side of the equals sign, square brackets are also required with a size specified inside.

An ArrayList should be declared with the type of element for the ArrayList in angle brackets.

Instantiating without Values

Instantiating Arrays	Instantiating ArrayLists
`String[] array = new String[10];`	`ArrayList<String> arrayList = new ArrayList<>();`
An array of 10 elements is created, all with null references. The compiler will only permit Strings to be assigned to the elements.	An empty ArrayList is created. The compiler will check that only Strings are added to the ArrayList.

You can use the diamond operator when creating a new instance in a declaration statement.

You should use a specific type rather than just the Object class because Java can then perform compile-time type checking.

Instantiating with Values

Instantiating Arrays

```
String[] array = new String[] {"first", "second", "third"};
```

An array of 3 elements is created, with

```
elements[0] = "first"

elements[1] = "second"

elements[2] = "third
```

Alternately, we can use this array initializer (anonymous array).

```
String[] array = {"first", "second", "third"};
```

Instantiating Lists and Array Lists

```
ArrayList<String> arrayList = new ArrayList<>(List.of("first",
        "second", "third"));
```

An ArrayList can be instantiated by passing another list to it as we show here.

We can use the List.of() factory method, which uses variable arguments, to create a pass through immutable list.

You can use an array initializer to populate array elements during array creation.

This feature lets you pass all the values in the array as a comma delimited list in curly braces.

Instantiating with Values

Instantiating Arrays

```
String[] array = new String[] {"first", "second", "third"};
```

An array of 3 elements is created, with

```
elements[0] = "first"

elements[1] = "second"

elements[2] = "third
```

Alternately, we can use this array initializer (anonymous array).

```
String[] array = {"first", "second", "third"};
```

Instantiating Lists and Array Lists

```
ArrayList<String> arrayList = new ArrayList<>(List.of("first",
        "second", "third"));
```

An ArrayList can be instantiated by passing another list to it as we show here.

We can use the List.of() factory method, which uses variable arguments, to create a pass through immutable list.

When you use an array initializer in a declaration statement, you can use what's called the anonymous version, as I show here.

You can use an ArrayList constructor that takes a collection or a list of values during ArrayList creation.

Instantiating with Values

Instantiating Arrays

```
String[] array = new String[] {"first", "second", "third"};
```

An array of 3 elements is created, with

```
elements[0] = "first"

elements[1] = "second"

elements[2] = "third
```

Alternately, we can use this array initializer (anonymous array).

```
String[] array = {"first", "second", "third"};
```

Instantiating Lists and Array Lists

```
ArrayList<String> arrayList = new ArrayList<>(List.of("first",
        "second", "third"));
```

An ArrayList can be instantiated by passing another list to it as we show here.

We can use the List.of() factory method, which uses variable arguments, to create a pass through immutable list.

The List.of method can be used to create such a list, with a variable argument list of elements.

Element information

	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);`

The number of elements is fixed when an array is created.

You can get the size of the array from the attribute length on the array instance.

Array elements are accessed with the use of square brackets and an index that ranges from 0 to one less than the number of elements.

Element information

	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);`

The number of elements in an ArrayList may vary and can be retrieved with a method on the instance, named size().

ArrayList elements are accessed with get and set methods, also using an index ranging from 0 to one less than the number of elements.

Getting a String representation for Single Dimension Arrays and ArrayLists

Array

Array Creation Code

```
String[] arrays = {"first", "second", "third"};
```

Printing Array Elements

```
System.out.println(Arrays.toString(arrays));
```

ArrayList

ArrayList Creation Code

```
ArrayList<String> arrayList = new ArrayList<>(List.of("first",
        "second", "third"));
```

Printing ArrayList elements

```
System.out.println(arrayList);
```

ArrayLists come with built-in support for printing out elements, including nested lists.

Arrays don't though, so you need to call Arrays.toString, passing the array as an argument.

This slide shows examples of single dimension arrays and ArrayLists.

Getting a String representation for Multi-Dimensional Arrays and ArrayLists

Array

Array Creation Code

```
String[][] array2d = {

        {"first", "second", "third"},

        {"fourth", "fifth"}

};
```

Printing Array Elements

```
System.out.println(Arrays.deepToString(array2d));
```

ArrayList

ArrayList Creation Code

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

Printing ArrayList elements

```
System.out.println(multiDList);
```

Here, I show examples of multi-dimensional arrays and ArrayLists, and how to print the elements in each.

A multi-dimensional ArrayList simply has a type, which in itself is an ArrayList.

Getting a String representation for Multi-Dimensional Arrays and ArrayLists

Array

Array Creation Code

```
String[][] array2d = {

        {"first", "second", "third"},

        {"fourth", "fifth"}

};
```

Printing Array Elements

```
System.out.println(Arrays.deepToString(array2d));
```

ArrayList

ArrayList Creation Code

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

Printing ArrayList elements

```
System.out.println(multiDList);
```

For a multi-dimensional array, you need to call the Arrays.deepToString method, passing the array as an argument.

For nested ArrayLists, we can still just pass the ArrayList instance directly, to System.out.println, as shown here.

Finding an element in an Array or ArrayList

Arrays methods for finding elements	ArrayList methods for finding elements

`int binarySearch(array, element)`
 	
** Array MUST BE SORTED
 	
Not guaranteed to return index of first element if there are duplicates
	`boolean contains(element)`
	
	`boolean containsAll(list of elements)`
	
	`int indexOf(element)`
	
	`int lastIndexOf(element)`

For arrays, you can use the binarySearch method to find a matching element, although this method requires that the array be sorted first.

In addition, if the array contains duplicate elements, the index returned from this search is not guaranteed to be the position of the first element.

For the ArrayList, we have several methods.

Finding an element in an Array or ArrayList

Arrays methods for finding elements	ArrayList methods for finding elements

`int binarySearch(array, element)`
 	
** Array MUST BE SORTED
 	
Not guaranteed to return index of first element if there are duplicates
	`boolean contains(element)`
	
	`boolean containsAll(list of elements)`
	
	`int indexOf(element)`
	
	`int lastIndexOf(element)`

You can use contains or containsAll, which simply returns a boolean if a match or matches were found.

In addition, like the String and StringBuilder, we have the methods, indexOf and lastIndexOf, which will return the index of the first or last match.

When a -1 is returned from these methods, no matching entry was found.

Sorting

Array	ArrayList

`String[] arrays = {"first", "second", "third"};`
 	
`Arrays.sort(arrays);`
	`ArrayList<String> arrayList = new ArrayList<>(List.of("first",`
	`"second", "third"));`
	
	`arrayList.sort(Comparator.naturalOrder());`
	`arrayList.sort(Comparator.reverseOrder());`

You can only sort arrays of elements that implement Comparable. We'll be discussing this in a future section. Character Sequence classes, like String and StringBuilder meet this requirement.	You can use the sort method with static factory methods to get Comparators.

Sorting seems like a simple concept when you think about sorting numbers and Strings.

We know there is a natural order for numbers and even for Strings.

Sorting

Array	ArrayList

`String[] arrays = {"first", "second", "third"};`
 	
`Arrays.sort(arrays);`
	`ArrayList<String> arrayList = new ArrayList<>(List.of("first",`
	`"second", "third"));`
	
	`arrayList.sort(Comparator.naturalOrder());`
	`arrayList.sort(Comparator.reverseOrder());`

You can only sort arrays of elements that implement Comparable. We'll be discussing this in a future section. Character Sequence classes, like String and StringBuilder meet this requirement.	You can use the sort method with static factory methods to get Comparators.

We can use the Arrays.sort method, for arrays with numeric primitive types and wrapper classes, as well as Strings and StringBuilders.

For the ArrayList, we can use the sort method again for numeric wrapper classes, Strings and StringBuilders.

Sorting

Array	ArrayList

`String[] arrays = {"first", "second", "third"};`
 	
`Arrays.sort(arrays);`
	`ArrayList<String> arrayList = new ArrayList<>(List.of("first",`
	`"second", "third"));`
	
	`arrayList.sort(Comparator.naturalOrder());`
	`arrayList.sort(Comparator.reverseOrder());`

You can only sort arrays of elements that implement Comparable. We'll be discussing this in a future section. Character Sequence classes, like String and StringBuilder meet this requirement.	You can use the sort method with static factory methods to get Comparators.

You pass a Comparator type argument to ArrayList's sort method that specifies how the sort should be performed.

You call static methods on the Comparator type to get a Comparator for either a natural order, or reverse order sort.

Array as an ArrayList

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

There are times when you'll want to switch between an Array and an ArrayList, and there is support for this on both the Arrays class and the ArrayList class.

The Arrays.asList method returns an ArrayList backed by an array.

Here, I show the creation of a three element array.

Then the code uses the Arrays.asList method, passing it the array, and assigning the result, a List of Strings, to a variable, originalList.

Array as an ArrayList

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

You can think of this conceptually, as putting an ArrayList wrapper of sorts around an existing array.

Any change made to the List is a change to the array that backs it.

This also means that an ArrayList created by this method is not resizable.

Creating Special Kinds of Lists

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);`

This slide demonstrates two ways to create a list. From elements or from an array of elements.

Both are static factory methods on types.

The first is the asList method on the Arrays class, and it returns a special instance of a List that is not resizable, but is mutable.

The second is the of method on the List interface, and it returns a special instance of a List that is immutable.

Creating Special Kinds of Lists

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);`

Both support variable arguments, so you can pass a set of arguments of one type, or you can pass an array.

I am showing examples of both here, first using variable arguments, and second, passing an array.

Creating Arrays from ArrayLists

```
ArrayList<String> stringLists = new ArrayList<>(List.of("Jan","Feb","Mar"));
String[] stringArray = stringLists.toArray(new String[0]);
```

This slide shows the most common method to create an array from an ArrayList using the method toArray.

This method takes one argument which should be an instance of a typed array.

This method returns an array of that same type.

If the length of the array you pass has more elements than the list, extra elements will be filled with the default values for that type.

If the length of the array you pass has less elements than the list, the method will still return an array, with the same number of elements in it, as the list.

In the example shown here, I pass a String array with zero as the size, but the array returned has three elements, which is the number of elements in the list.