NPTEL

noc24_cs43

Programming in Java

Live Interaction Session 6: 8th Mar 2024

Arrays

Java provides us with many types of containers, to store multiple values of the same type.

An array is a data structure, that allows you to store a sequence of values, all of the same type.

You can have arrays for any primitive type, like ints, doubles, booleans, or any of the 8 primitives we've learned about.

You can also have arrays for any class.

Arrays

Elements in an array are indexed, starting at 0.

If we have an array, storing five names, conceptually it looks as shown here.

Index	**0**	**1**	**2**	**3**	**4**
Stored values in an array with 5 elements	"Andy"	"Bob"	"Charlie"	"David"	"Eve"

The first element is at index 0, and is Andy.

The last element in this array is at index 4, and has the String value Eve.

Declaring an Array

Array Variable Declaration
`int[] integerArray;`
`String[] nameList;`
`String courseList[];`

Instantiating an Array

Array Creation	Object Creation
`int[] integerArray = new int[10];`	`StringBuilder sb = new StringBuilder();`

One way to instantiate the array, is with the new keyword, much as we've seen, with most of the classes we've used to date, with the exception of String.

On this slide, we have an array declaration on the left of the equals sign, and then an array creation expression on the right side.

For comparison, there is a typical array variable declaration, and a class instance, or object creation expression, using the StringBuilder class

Instantiating an Array

Array Creation	Object Creation
`int[] integerArray = new int[10];`	`StringBuilder sb = new StringBuilder();`

They look pretty similar, but there are two major differences.

Square brackets are required when using the new keyword, and a size is specified between them. So in this example, there will be 10 elements in the array.

An array instantiation doesn't have a set of parentheses, meaning we can't pass data to a constructor for an array.

In fact, using parentheses with an array instantiation, gives you a compiler error.

Invalid Array Creation – Compile Error because of ()
`int[] integerArray = new int[10]();`

An Array is NOT Resizable.

You can't change the size of an array, after the array is instantiated.

We can't add or delete elements, we can only assign values to one of the ten elements in this array, in this example.

The array initializer

An array initializer, makes the job of instantiating and initializing a small array, much easier.

The array initializer

```
int[] firstFivePositives = new int[]{1, 2, 3, 4, 5};
```

In this example, you can see we still use the new keyword, and have int, with the square brackets.

But here we specify the values, we want the array to be initialized to, in a comma delimited list, in curly braces.

Because these values are specified, the length of the array can be determined, so we don't specify the size in the square brackets.

And actually, Java provides an even simpler way to do this.

The array initializer as an anonymous array

Java allows us to drop new int[], from the expression, as we show here.

This is known as an anonymous array.

Here we're showing examples for both an int array, as well as a String array.

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

An anonymous array initializer, can only be used in a declaration statement.

What is an array, really?

An array is a special class in Java.

It's still a class.

The array, like all other classes, ultimately inherits from java.lang.Object.

Array initialization and default element values

When you don't use an array initializer statement, all array elements get initialized to the default value for that type.

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**.

The Enhanced For Loop, the For Each Loop

This loop was designed to walk through elements in an array, or some other type of collection.

It processes one element at a time, from the first element to the last.

Here I show you the syntax for the two types of for loop statements, side by side.

Enhanced For Loop	Basic For Loop

`for (declaration : collection) {`
`    // block of statements`
`}`
	`for (init; expression; increment) {`
	`    // block of statements`
	`}`

The enhanced for loop only has two components, vs. 3, defined in the parentheses after the for keyword.

The Enhanced For Loop, the For Each Loop

Enhanced For Loop	Basic For Loop

`for (declaration : collection) {`
`    // block of statements`
`}`
	`for (init; expression; increment) {`
	`    // block of statements`
	`}`

It's important to notice, that the separator character between components, is a **colon**, and not a semi-colon, for the Enhanced For Loop.

The **first** part is a **declaration expression**, which includes the type and a variable name. This is usually a local variable with the same type as an element in the array.

And the **second** component is the **array**, or some other collection variable.

java.util.Arrays

Java's array type is very basic, it comes with very little built-in functionality.

It has a single property or field, named length.

And it inherits java.util.Object's functionality.

Java provides a helper class named java.util.Arrays, providing common functionality, you'd want for many array operations.

These are static methods on Arrays, so are class methods, not instance methods.

Printing elements in an array using Arrays.toString()

The toString method on this helper class, prints out all the array elements, comma delimited, and contained in square brackets.

```
String arrayElementsInAString = Arrays.toString(newArray);
```

The output from this method is shown here, conceptually.

It prints the element at index 0 first, followed by a comma, then element at index 1 next, comma, and so on, until all elements are printed.

```
[ e[0], e[1], e[2], e[3], ... ]
```

Why use arrays?

We use arrays to manage many items of the same type.

Some common behavior for arrays would be sorting, initializing values, copying the array, and finding an element.

For an array, this behavior isn't on the array instance itself, but it's provided on a helper class, java.util.Arrays.

The Array Challenge

Create a program using arrays, that **sorts** a list of **integers**, in **descending order**. Descending order means from highest value to lowest.

In other words, if the array has the values 50, 25, 80, 5, and 15, your program should return an array, with the values in the descending order: 80, 50, 25, 15, and 5.

First, create an **array** of **randomly generated integers**.

Print the array before you sort it.

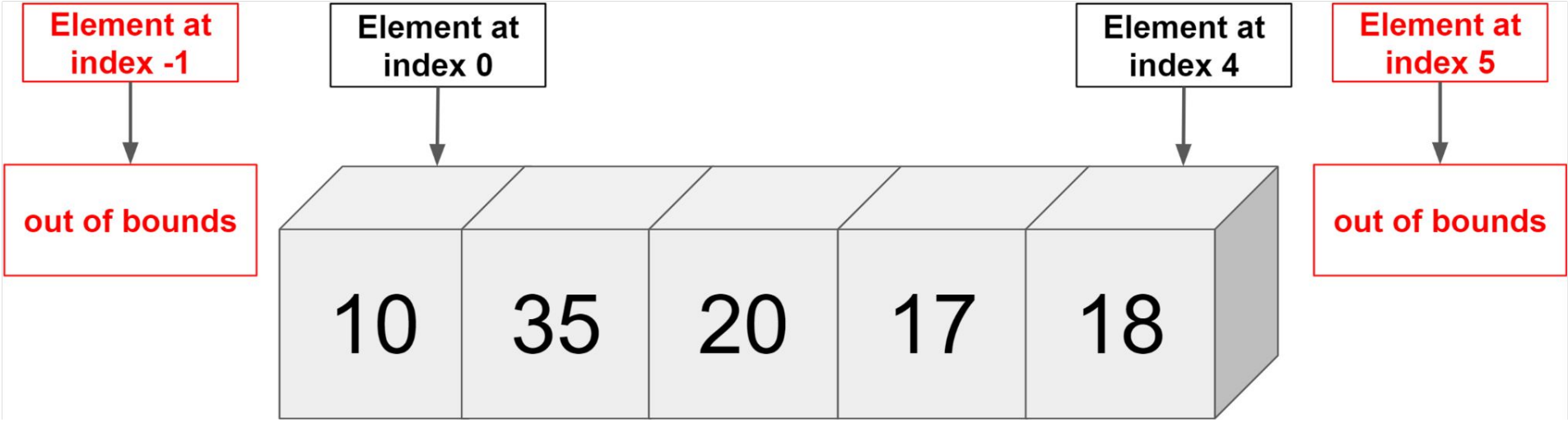
And print the array after you sort it.

You can choose to create a new sorted array, or just sort the current array.

First Common Error

```
int[] myArray = {10, 35, 20, 17, 18};
myArray[5] = 55;     // out of bounds
```

- Accessing index out of range will cause error in other words **ArrayIndexOutOfBoundsException**
- We have **5 elements and index range is 0 to 4**



Second Common Error

```
int[] myArray = {10, 35, 20, 17, 18};

for (int i = 1; i < myArray.length; i++) {
    System.out.println("value = " + myArray[i]);
}
```

OUTPUT:

value= 35
value= 20
value= 17
value= 18

Third Common Error

```
int[] myArray = {10, 35, 20, 17, 18};

for (int i = 0; i <= myArray.length; i++) {
    System.out.println("value = " + myArray[i]);
}
```

OUTPUT:

value = 10 // printed when i = 0 -> condition 0 <= 5 is true
value = 35 // printed when i = 1 -> condition 1 <= 5 is true
value = 20 // printed when i = 2 -> condition 2 <= 5 is true
value = 17 // printed when i = 3 -> condition 3 <= 5 is true
value = 18 // printed when i = 4 -> condition 4 <= 5 is true

ArrayIndexOutOfBoundsException when i = 5 since condition 5 <= 5 is true

Use Enhanced For Loop to avoid some of these errors

Use the enhanced for loop, if you're looping through elements from first to last, and want to process them one at a time, and you're not setting or assigning values to elements.

Enhanced For Loop (Preferred for this kind of processing)

```
int[] myArray = {10, 35, 20, 17, 18};

for (int myInt : myArray) {
    System.out.println("value = " + myInt);
}
```

Traditional For Loop

```
int[] myArray = {10, 35, 20, 17, 18};

for (int i = 0; i < myArray.length; i++) {
    System.out.println("value = " + myArray[i]);
}
```

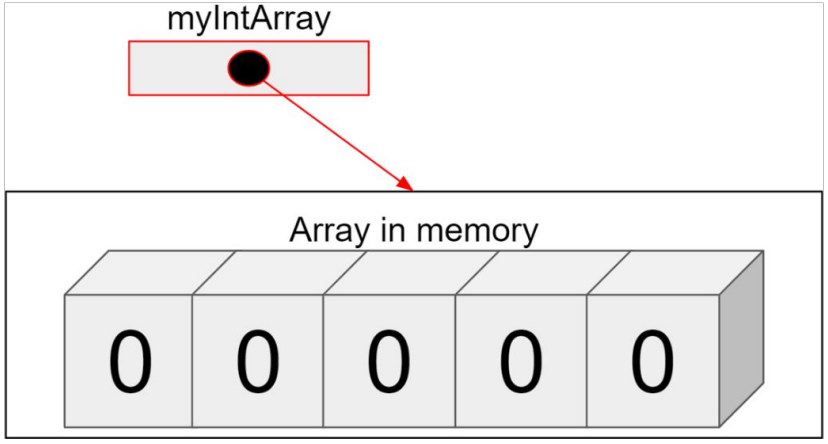
Reference Types vs. Value Types

```
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));
```



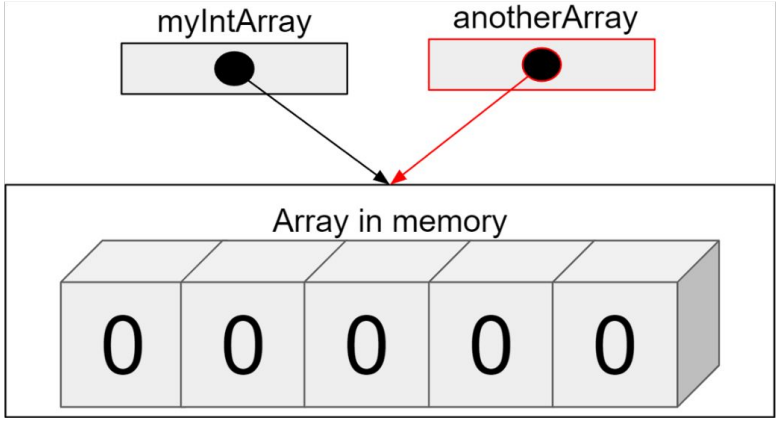
Reference Types vs. Value Types

```
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));
```



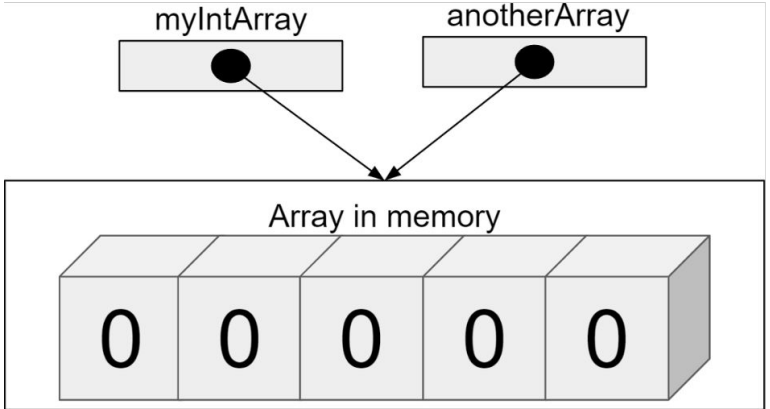
Reference Types vs. Value Types

```
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));
```



OUTPUT:

myIntArray= 0 0 0 0 0

anotherArray= 0 0 0 0 0

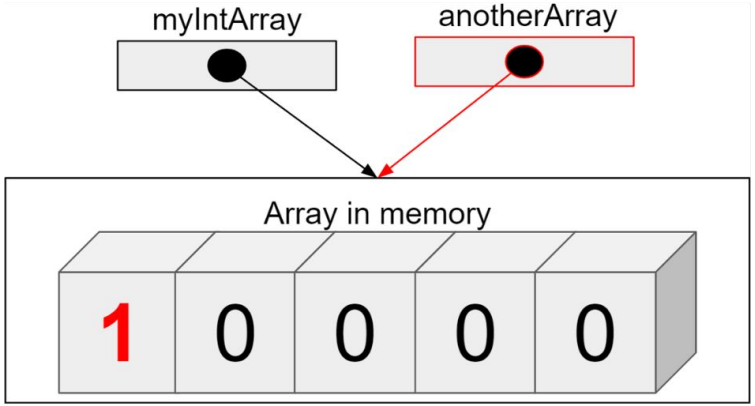
Reference Types vs. Value Types

```
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));
```



OUTPUT:

myIntArray= 0 0 0 0 0

anotherArray= 0 0 0 0 0

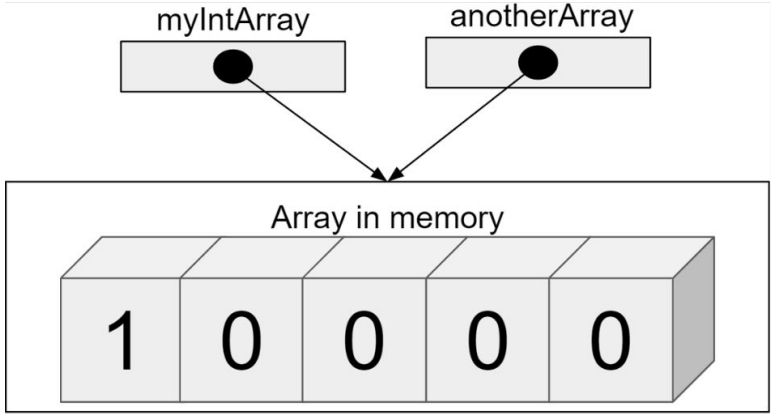
Reference Types vs. Value Types

```
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));
```



OUTPUT:

myIntArray= 0 0 0 0 0
anotherArray= 0 0 0 0 0
after change myIntArray= 1 0 0 0 0
after change anotherArray= 1 0 0 0 0

Arrays as method parameters

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

}
```

Notice here that the parameter to the main method, is an array of String.

This means we can pass an array of Strings to this method, when it's called.

Or, if we use this method as the entry point to our application, we can pass data on the command line to this method.

Up until now, I've only shown you this particular method signature.

Variable arguments (varargs)

But this signature can be written in a slightly different way.

We can replace the brackets after the String type, which we know tells us this method will take an array of String.

And we can instead replace that with three periods.

This is a special designation for Java, that means, Java will take zero, one, or many Strings, as arguments to this method, and create an array with which to process them, in the method.

```
public static void main(String... args) {

}
```

Variable arguments (varargs)

The array will be called args, and be of type String.

So what's the difference then?

The difference is minor within the method body, but significant to the code that calls the method.

```
public static void main(String... args) {

}
```

When can you use variable arguments (varargs) ?

There can be only one variable argument in a method.

The variable argument must be the last argument.

Minimum Element Challenge

Write a method called **readIntegers**, that reads a comma delimited list of numbers, entered by the user from the console, and then returns an **array**, containing those numbers that were entered.

Next, write a method called **findMin**, that takes the array as an argument, and returns the **minimum value** found in that **array**.

Minimum Element Challenge

In the **main method**

Call the method **readIntegers**, to get the array of integers from the user, and print these out, using a method found in java.util.Arrays.

Next, call the **findMin method**, passing the **array,** returned from the call to the **readIntegers** method.

Print the **minimum element** in the **array**, which should be returned from the **findMin** method.

A tip here. Assume that the user will only enter numbers - so you don't need to do any validation for the console input.

The Reverse Array Challenge

The challenge is to write a method called reverse, that takes an int array as a parameter.

In the main method, call the reverse method, and print the array both before and after the reverse method is called.

To reverse the array, you have to swap the elements, so that the first element is swapped with the last element, and so on.

So for example, if the array contains the numbers 1,2,3,4,5, then the reversed array should be, 5,4,3,2,1.

Java’s Nested Array

An array element can actually be an array. It's known as a nested array, or an array assigned to an outer array's element.

This is how Java supports two and three dimensional arrays, of varying dimensions.

Two-Dimensional Array

Array Initializer formatted over multiple lines

```
int[][] array = {
                        {1, 2, 3},
                        {11, 12, 13},
                        {21, 22, 23},
                        {31, 32, 33}
                        };
```

Array Initializer declared on one line

```
int[][] array = {{1, 2, 3}, {11, 12, 13}, {21, 22, 23}, {31, 32, 33}};
```

Two-Dimesional Array

A 2-dimensional array doesn't have to be a uniform matrix though.

This means the nested arrays can be different sizes, as we show with this next initialization statement.

```
int[][] array = {
                    {1, 2},
                    {11, 12, 13},
                    {21, 22, 23, 24, 25}
                    };
```

Here, we have an array with 3 elements.

Each element is an array of integers (a nested array).

Each nested array is a different length.

If you find that confusing, don't worry. It should all make sense shortly.

Two-dimensional Array

There are a lot of ways to declare a two-dimensional array.

We will cover the two most common ways here.

The most common, and in my opinion, clearest way, to declare a two-dimensional array, is to stack the square brackets as shown in the first example.

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

You can also split up the brackets as shown in the second example, and you'll likely come across this in Java code out in the wild.

Two Dimensional Array

When we declare multi-dimensional arrays, the declared type can itself be an array, and this is how Java supports two-dimensional arrays:

```
int[][] myArray = new int[3][];     // Declares and instantiates an array of 3 integer arrays,
//      whose sizes are not specified

Dog[][] myDogs = new Dog[3][];      // Declares and instantiates an array of 3 arrays,
//      which will have Dog elements, again, the sizes of the inner arrays aren't specified
```

Type and length of array	Possible Element Values (each element is an array and can be any length)
int[3][]	[5, 7, 9, 10] [3, 6] [11, 21, 31]
Dog[3][]	[pug, rottweiler] [germanShephard, poodle, cavapoo] [beagle, boxer, bulldog, yorkie]

Multi Dimensional Array

We can take that even further, the outer array can have references to any kind of array itself.

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.

Java Array vs Java List

An array is mutable, and we saw, that we could set or change values in the array, but we could not resize it.

Java gives us several classes that let us add and remove items, and resize a sequence of elements.

These classes are said to **implement** a List's behavior.

So what is a list?

So what is a List?

List is a special type in Java, called an Interface.

For now, I'll say a List Interface describes a set of method signatures, that all List classes are expected to have.

The ArrayList

The ArrayList is a class, that really maintains an array in memory, that's actually bigger than what we need, in most cases.

It keeps track of the capacity, which is the actual size of the array in memory.

But it also keeps track of the elements that've been assigned or set, which is the size of the ArrayList.

As elements are added to an ArrayList, its capacity may need to grow. This all happens automatically, behind the scenes.

This is why the ArrayList is resizeable.

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.

An array requires square brackets in the declaration.

In the new instance, square brackets are also required, with a size specified inside

An ArrayList should be declared, with the type of element in 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.

We 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
```

```
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
```

```
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.

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
```

```
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);`

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);
```

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)`

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.

The ArrayList Challenge

The challenge is to create an interactive console program.

And give the user a menu of options as shown here:

```
Available actions:

0 - to shutdown

1 - to add item(s) to list (comma delimited list)

2 - to remove any items (comma delimited list)

Enter a number for which action you want to do:
```

Using the Scanner class, solicit a menu item, 0, 1 or 2, and process the item accordingly.

The ArrayList Challenge

Your grocery list should be an ArrayList.

You should use methods on the ArrayList, to add items, remove items, check if an item is already in the list, and print a sorted list.

You should print the list, sorted alphabetically, after each operation.

You shouldn't allow duplicate items in the list.