Chapter 16. Arrays Basics

Topic Covered

1. Declaring, Constructing and Initializing an array.
2. Anonymous array declaration.
3. One dimentional array n two dimentional arrays.
4. NullPointerException and ArrayOutOfBoundsException.
5. Array element assignments for both Primitives and Objects(Objects, subclass objects and Interfaces).
6. Array of Interfaces.
7. Array reference assignments.

---

1. Arrays are objects in java that stores multiple variables of the same type. Arrays can hold either primitives or object references.

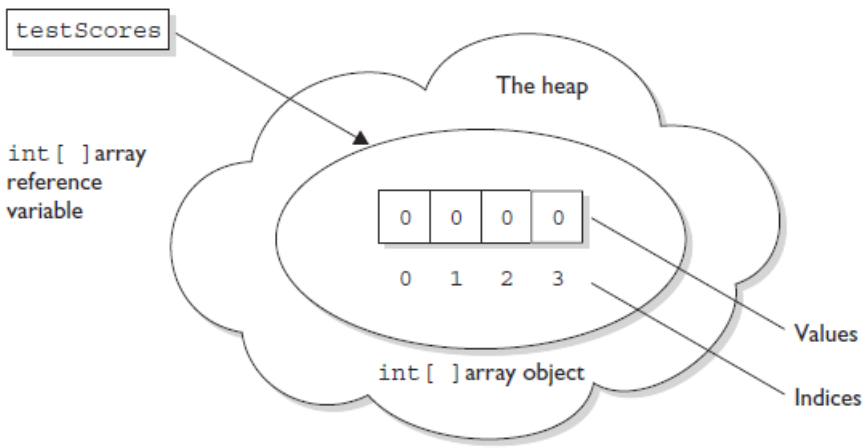
2. Constructing and initializing an array :

```
int [] testScores;//line one

testScores = new int[5];//line two
```

The above can also be written in one line as

```
int [] testScores = new int[5];
```

- Above left part called as declaring an array and right part is called constructing an array i.e. creating an Array object on the heap by doing a new on array type. While declaring an array wont make compiler to allot memory space. It only after constructing or calling new on array type memory is allocated.
- Above testScores is called identifier and int[] is array type i.e. its an array of type int. Arrays must be given a size at the time when they are constructed as JVM needs to allocate space for them. They cannot contract or expand dynamically as collections do. Hence their size is fixed.



3. Lets create array of objects.

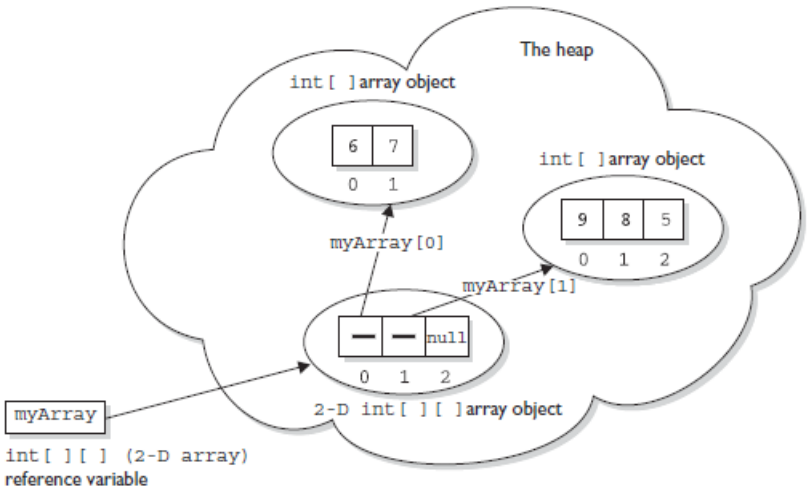
```
Thread[] threads = new Thread[5]; // no Thread objects created!
                                  // one Thread array created
```

Remember through above code thread constructor is not being called. It not creating a thread instance rather Thread array object, so there are no actual thread objects.

4. Constructing an multidimentional array :

```
int[][] myArray = new int[3][];
```

Above code shows how multidimensional array is defined. Multidimensional array is simply an array of arrays. In the above example 'myArray' is an array of arrays of type int. Size defined in the first bracket is enough for JVM to know the size needed to define for this array.



Picture demonstrates the result of the following code:

```
int[ ][ ] myArray = new int[3][ ];
myArray[0] = new int[2];
myArray[0][0] = 6;
myArray[0][1] = 7;
myArray[1] = new int[3];
myArray[1][0] = 9;
myArray[1][1] = 8;
myArray[1][2] = 5;
```

5. Initializing an Array :

- Initializing an array means putting things into it. These things in the arrays are called arrays elements. They are either primitives or object references. If those object references are not

pointing to any objects then accessing then through (.) operator throws **NullPointerException**.

- Array elements are accessed through index number which starts with zero (0) to (length of array – 1). Ex. for array of size 10 index number will run from 0 to 9.
- When someone tries to access an out-of-the-range array index then **ArrayIndexOutOfBoundsException** will be thrown.
- Ex:

```
public class SimpleArray {

public static void main(String[] args)
 {
        int [] intArr = new int[3];
        intArr[0] = 1;
        intArr[1] = 2;
        intArr[2] = 3;
        System.out.println(intArr[0]);
        System.out.println(intArr[1]);
        System.out.println(intArr[2]);
 }
}
```

```
Output:
1
2
3
```

- Array could also be initialized in a loop as shown below

```
public class SimpleArray {

 public static void main(String[] args)
 {

        int [] intArr = new int[3];
        //initialize array intArr
        for(int i=0;i<intArr.length;i++)
              intArr[i] = i+1;
        //print elements of intArr
        for(int j=0;j<intArr.length;j++)
              System.out.println(intArr[j]);
 }
}
```

```
Output:
1
2
3
```

Also for objects, we can loop as shown below.

```
Dog[] myDogs = new Dog[6]; // creates an array of 6 Dog references
for(int x = 0; x < myDogs.length; x++) {
myDogs[x] = new Dog(); // assign a new Dog to index position x
}
```

– When Array elements are initialized then they are set to their default values.

Example:

```
package OverloadingnArrays;

public class ArrayExceptionCls {

	public static void main(String[] args)
	{
		try
		{
			int [] arr = new int[6];
			arr[0] = 0;
			arr[1] = 1;
			arr[4] = 4;
			arr[5] = 5;
		//	arr[6] = 6;//ERROR

			System.out.println("Array Contains");
			for(int i = 0; i<6; i++)
			{

				System.out.println(arr[i]);
			}

		} catch (Exception e)
		{
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
	}
	Output:
	Array Contains
	0
	1
	0
	0
	4
	5
```

– Example for ArrayIndexOutOfBoundsException and NullP

– Example 1: ArrayIndexOutOfBoundsException

```
package OverloadingnArrays;

public class ArrayExceptionCls {

	public static void main(String[] args)
	{
		try
		{
			int [] arr = new int[6];
			arr[0] = 0;
			arr[1] = 1;
			arr[4] = 4;
			arr[5] = 5;
			arr[6] = 6;//ERROR

		} catch (Exception e)
		{
			// TODO Auto-generated catch block
```

```
                e.printStackTrace();
            }
        }
}
Output:
java.lang.ArrayIndexOutOfBoundsException: 6
        at
OverloadingnArrays.ArrayExceptionCls.main(ArrayExceptionCls.java:14)
```

– Also if one tries to access 'arr' array elements through for loop as shown below

```
for(int i = 0; i<=6; i++)
```

then also it will throw ArrayIndexOutOfBoundsException because of '=' in the for loop tries to access arr[6].

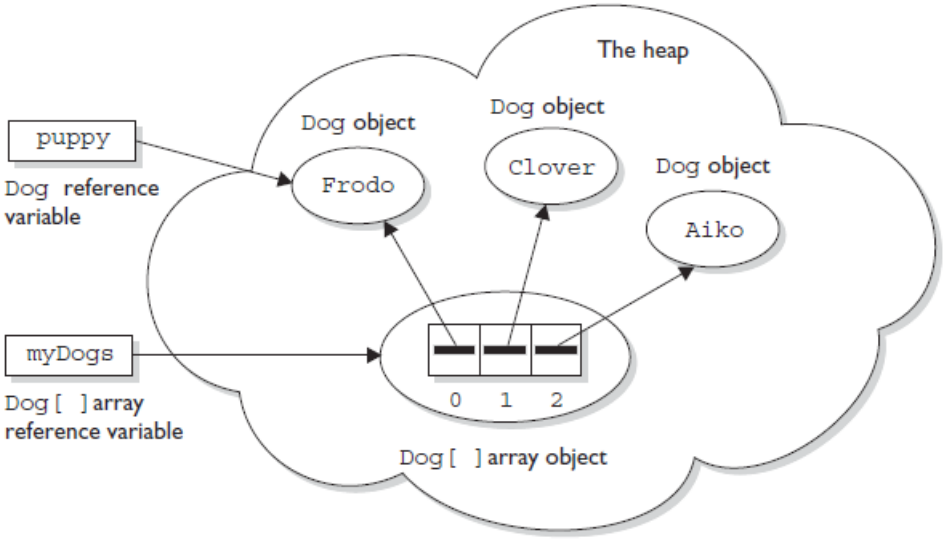
– Example 2 : NullPointerException

```
package OverloadingnArrays;

class A
{
        int a = 10;
}

public class ArrayNullPointerCls {

        public static void main(String[] args)
        {
                A a1 = new A();
                A a2 = new A();
                A a3 = new A();

                A [] objArr = new A[4];
                        //{a1, a2, a3};

                objArr[0] = a1;
                objArr[1] = a1;
                objArr[2] = a1;

                try {
                        for(int i=0; i<objArr.length;i++)
                        {
                                System.out.println(objArr[i].a);
                        }
                } catch (Exception e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
        }
}
Output:
10
10
10
java.lang.NullPointerException
        at
OverloadingnArrays.ArrayNullPointerCls.main(ArrayNullPointerCls.java:26)
```

6. Declaring, Constructing and Initializing an array in a single line

- `int [] intArr = {1,2,3};`
- It is very useful when Array elements are known well in advance.
- Above can be used for Array of objects too

```
Dog puppy = new Dog("Frodo");
Dog[] myDogs = {puppy, new Dog("Clover"), new Dog("Aiko")};
```



Picture demonstrates the result of the following code:

```
Dog puppy = new Dog ("Frodo");
Dog[ ] myDogs = {puppy, new Dog("Clover"), new Dog("Aiko")};
```

Four objects are created:
1 Dog object referenced by puppy and by myDogs[0]
1 Dog[ ] array referenced by myDogs
2 Dog object referenced by myDogs[1] and myDogs[2]

Example :

```
package OverloadingnArrays;

public class TwoDimensionalArray {

	public static void main(String[] args)
	{
		int arr1 [] = {1,3,5};
		int arr2 [] = {2,4,6};
```

```
            int arr [][] = new int [2][2];

            arr[0] = arr1;
            arr[1] = arr2;

            for(int i=0; i<2;i++)
            {
                  for(int j=0;j<2;j++)
                  {
                        System.out.println(arr[i][j]);
                  }
            }
      }
}
Output:
1
3
2
4
```

7. Anonymous Array Initialization :

```
int[] testScores;
testScores = new int[] {4,7,2};
```

Second shortcut for array creation is “anonymous array creation”. Here line one array has been declared and then in second line it has been constructed, initialized and then assigned to the identifier. It is very useful when array is passed as an argument to a method, as shown below.

Ex 1:

```
class ArrMethParam1
{
    void ArrMeth(int [] intArr)
    {
          for(int i = 0; i<intArr.length; i++)
                System.out.println(intArr[i]);
    }
}

public class ArrMethParam {
    public static void main(String[] args)
    {
          ArrMethParam1 amp = new ArrMethParam1();
          amp.ArrMeth(new int []{1,2,3,4});

    }
}
Output:
1
2
3
4
```

- Also remember one cannot specify a size when using anonymous array creation.

```
new Object[3] {null, new Object(), new Object()};
// not legal; size must not be specified
```

Ex.2

```
package OverloadingnArrays;

public class ArrayDeclarationCls
{
	public static void main(String[] args)
	{
		System.out.println("Standard Array initialization");
		int [] arr1 = new int[5];
		for(int i=0; i<arr1.length; i++)
		{
			arr1[i] = i+2;
		}

		for(int j = 0; j<arr1.length; j++)
		{
			System.out.println(arr1[j]);
		}

		System.out.println("Simplest Array Initialization");

		int [] arr2 = {2,4,6,8,10};

		for(int a=0; a<arr2.length; a++)
		{
			System.out.println(arr2[a]);
		}

		System.out.println("Anonymous Array Initialization");

		int [] arr3 = new int[]{1,3,5,7,9};

		for(int b=0; b<arr3.length; b++)
		{
			System.out.println(arr3[b]);
		}
	}
}
Output:
Standard Array initialization
2
3
4
5
6
Simplest Array Initialization
2
4
6
8
10
Anonymous Array Initialization
1
3
```

5
7
9

8. Legal Array Element Assignments :

- Arrays of Primitives: Primitive array can accept any value that could be promoted implicitly to the declared type of the array. For example, an int array can hold any value that could fit into a 32-bit int variable. Thus the following code is legal.

```
int[] weightList = new int[5];
byte b = 4;
char c = 'c';
short s = 7;
weightList[0] = b; // OK, byte is smaller than int
weightList[1] = c; // OK, char is smaller than int
weightList[2] = s; // OK, short is smaller than int
```

- Ex:

```
public class ArrayLegalAssign {

 public static void main(String[] args)
 {
        byte b = 10;
        short s = 20;
        char c = 30;
        int[] intArr = {b,s,c};

        for(int i=0; i<intArr.length; i++)
              System.out.println(intArr[i]);
 }
}
Output:
10
20
30
```

- Arrays of Object references: If declared array type is a class then one can put objects of any subclass of the declared class type. Suppose Ferrari is a subclass of Car then one can put objects of both Ferrari and Car in a array of type Car as follow.

```
class Car {}
class Subaru extends Car {}
class Ferrari extends Car {}
...
Car [] myCars = {new Subaru(), new Car(), new Ferrari()};
```

One should remember that array in array of type Car are nothing but reference variables to the Car Objects. Hence everything that can be assigned to the car reference could be element of array of car objects.

- Ex.

```
class Car
{
  void CarMeth()
```

```
   {
         System.out.println("inside car method");
   }
}

class Ferrari extends Car
{
   void CarMeth()
   {
         System.out.println("inside Ferrari method");
   }
}

public class ArrayLegalAssign {

   public static void main(String[] args)
   {
         byte b = 10;
         short s = 20;
         char c = 30;
         int[] intArr = {b,s,c};

         for(int i=0; i<intArr.length; i++)
               System.out.println(intArr[i]);

         Car car = new Car();
         Ferrari fer = new Ferrari();

         Car[] CarArr = {car,fer};

         for(int j=0; j<CarArr.length; j++)
         CarArr[j].CarMeth();
   }
}
Output:
10
20
30
inside car method
inside Ferrari method
```

- Similarly we can have array of interfaces and we can use all the utilities available for interfaces too. As shown in following example.

```
interface Sporty {
void beSporty();
}
class Ferrari extends Car implements Sporty {
public void beSporty() {
// implement cool sporty method in a Ferrari-specific way
}
}
class RacingFlats extends AthleticShoe implements Sporty {
public void beSporty() {
// implement cool sporty method in a RacingFlat-specific way
}
}
class GolfClub { }
class TestSportyThings {
public static void main (String [] args) {
Sporty[] sportyThings = new Sporty [3];
```

```
sportyThings[0] = new Ferrari(); // OK, Ferrari
// implements Sporty
sportyThings[1] = new RacingFlats(); // OK, RacingFlats
// implements Sporty
sportyThings[2] = new GolfClub(); // NOT ok..
// Not OK; GolfClub does not implement Sporty
// I don't care what anyone says
}
}
```

- The bottom line is this: Any object that passes the IS-A test for the declared array type can be assigned to an element of that array. Also An array declared as an interface type can hold those object references which implements that interface.
- Example :

```
package OverloadingnArrays;

interface ArrayInterface
{
	void meth();
}

class A1 implements ArrayInterface
{
	public void meth()
	{
		System.out.println("Inside A1 meth()");
	}
}

class A2 implements ArrayInterface
{
	public void meth()
	{
		System.out.println("Inside A2 meth()");
	}
}

class A3
{

}

class A4 extends A1
{

}

public class ArrayOfInterfaces
{
	public static void main(String[] args)
	{
		A1 a1 = new A1();
		A2 a2 = new A2();
		A3 a3 = new A3();
		A4 a4 = new A4();
		ArrayInterface ai;
		//ArrayInterface [] ai = {a1, a2, a3, a4};//ERROR: Type
		mismatch: cannot convert from A3 to ArrayInterface
```

```
            //ArrayInterface [] arrIn = {ai, a1, a2, a4};//ERROR: The local
            variable ai may not have been initialized
            ArrayInterface [] arrIn = {a1, a2, a4};
            for(int i=0;i<arrIn.length; i++)
            {
                  ai = arrIn[i];
                  ai.meth();
            }
      }
}
Output:
Inside A1 meth()
Inside A2 meth()
Inside A1 meth()
```

9. Array Reference Assignment :

- One dimentional arrays :
- For Primitives: Array reference means reference to the array object. Don't forget array itself is an object which could hold either primitives or object references of same type. For example, if we define one *int* array then this array reference could only be reassigned to an int array object(of any size) and it not to any other array which is not int.
- Here one should keep in mind that which array element assignment byte could be stored in int array but this rules dont hold true in case of array references. i.e. a int array reference could not be reassign to a byte array.
- Ex:

```
public class ArrayRefAssign {
 public static void main(String args[])
 {
       int[] intArr1 = {1,2,3};
       int[] intArr2 = {4,5,6,7};
       byte[] byteArr = {11,12,13};

       intArr1 = intArr2;
       System.out.println("intArr1 elements are");
       for(int i=0; i<intArr1.length; i++)
       System.out.println(intArr1[i]);

            //intArr1 = byteArr;//Type mismatch : cannot convert from
            byte[] to int[]
 }
 }
 Output:
 intArr1 elements are
 4
 5
 6
 7
```

- For Objects: For reference of arrays which hold objects the rules are not much restrictive. Here the only condition is if it passes IS-A relationship that reference of one object array could be reassign to another.

  Ex :

```
Car[] cars;
Honda[] cuteCars = new Honda[5];
cars = cuteCars; // OK because Honda is a type of Car
Beer[] beers = new Beer [99];
cars = beers; // NOT OK, Beer is not a type of Car
```

- Similarly for interfaces An array declared as an interface type can reference an array of any type that implements the interface. Also remember

  Ex:

```
Foldable[] foldingThings;
Box[] boxThings = new Box[3];
foldingThings = boxThings;
// OK, Box implements Foldable, so Box IS-A Foldable
```

- Example :

```
package ArrayBasics;

interface ArrayInterface
{
	void method();
}

class AA1 implements ArrayInterface
{
	public void method()
	{
		System.out.println("Inside AA1 method");
	}
}

class AA2 implements ArrayInterface
{
	public void method()
	{
		System.out.println("Inside AA2 method");
	}
}

class AA3
{
}

public class ArrayInterfaceRefAssignmnt {

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

```
            AA1 a11 = new AA1();
            AA1 a12 = new AA1();

            AA2 a21 = new AA2();
            AA2 a22 = new AA2();

            AA3 a31 = new AA3();
            AA3 a32 = new AA3();

            AA1 [] a1 = {a11, a12};
            AA2 [] a2 = {a21, a22};
            AA3 [] a3 = {a31, a32};

            ArrayInterface [] ai;

            ai = a1;//legal
            ai[0].method();

            ai = a2;//legal
            ai[0].method();

            //ai = a3;//illegal : cannot convert from AA3[] to
            ArrayInterface[]
      }
}
Output:
Inside AA1 method
Inside AA2 method
```

- For Multidimensional Arrays :

  Reference of different dimension of array cannot be assigned to each other. For example, a two-dimensional array of int arrays cannot be assigned to a regular *int* array reference.

```
int[] blots;
int[][] squeegees = new int[3][];
blots = squeegees; // NOT OK, squeegees is a
// two-d array of int arrays
int[] blocks = new int[6];
blots = blocks; // OK, blocks is an int array
```