TRAVERSING, COMPARING AND COPYING ARRAYS

Traversing arrays

Traversing an array is the process of visiting each element of an array.

Two common ways to traverse an array are:

- simple for loop / while loop
- for-each loop

Example array

```
int[] myArray = {1,2,3,4,5}
```

Example: for loop

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

Example: while loop

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

We should be careful about the indexes used. If the index value used is larger than the size of the array minus one, then an exception ArrayIndexOutOfBoundsException exception will be thrown. *Try changing the < symbol with <= in the above example.*

The for-each loop

The for-each loop provides a more convenient method of traversing an array if we do not need explicit access to each element's index value.

Above for-loop can be expressed as a for each loop as:

```
for(int num : myArray)
{
    System.out.println(num);
}
```

Even though we call this loop as for each loop, there is no each keyword.

for(int num : myArray) means that for each integer variable in myArray; and in each iteration one integer from array myArray will be assigned to the variable num.

Comparing arrays

There are several ways to compare arrays like:

- Element-by-element comparison
- Using the equality operator
- Using the equals method of the array
- Using the equals and deepEquals method of Arrays class

Consider two arrays:

```
int arr1[] = {1,2,3,4,5};
int arr2[] = {1,2,3,4,5} ;
```

Element by element comparison can be done inside a for loop:

```
boolean areEqual = true;
if(arr1.length != arr2.length)
  areEqual = false;
for(int i=0; i<arr1.length && areEqual; i++)
{
  if(arr1[i]!= arr2[i]) {
    areEqual = false;
  }
}
```

If we compare the two arrays **using the equality operator ==** as (arr1==arr2), we are actually comparing the array references and not array contents. So two different arrays with same values as our arr1 and arr2, will return false.

If we compare the two arrays **using the equals method of our array** as (arr1.equals(arr2)), we are again comparing the array references and not array contents. So two different arrays with same values as our arr1 and arr2, will return false here also.

Arrays class's static equals method will compare the values of an array. Arrays.equals **will work for one dimensional arrays.** Hence Arrays.equals(arr1,arr2) will return true in our case. Arrays.equals will not work for 2 dimensional arrays. 2 dimensional arrays are arrays of arrays i.e. an array with references to other arrays. Arrays.equals will compare these references and not the actual contained values in those arrays..

Arrays class's static deepEquals method performs a more in depth comparison of the array by going further and comparing the referenced object's values and not just references. Since 2 dimensional arrays are arrays of arrays, deepEquals will compare the values of the second level arrays referenced by the first level array and hence can be successfully used to **compare the values of a 2-dimensional array**. The**deepEquals** method requires an array of objects and will not work with array of ints. If we try Arrays.deepEquals(arr1, arr2), you will get a compile time error: The method deepEquals(Object[], Object[]) in the type Arrays is not applicable for the arguments (int[], int[]). To test deepEquals, we will create two 2-D arrays and then compare as:

```
int arr1[][] = {{1,2,3,4,5},{1,2,3,4,5}};
```

```
int arr2[][] = {{1,2,3,4,5},{1,2,3,4,5}};

System.out.println(Arrays.deepEquals(arr1, arr1));
```

This will return true.

Try comparing this 2-D array with equals:

```
System.out.println(Arrays.equals(arr1, arr2));
```

This will return false.

Copying arrays

Copy of objects can be shallow or deep.

In Shallow copy, only the reference values are copied. In a deep copy, the reference to the object is not copied, but a new copy of the object is created.

Various techniques used for Array copy are:

- element-by-element copy
- Using the System.arraycopy method
- Using the Arrays.copyOf method
- Using the Arrays.copyOfRange method
- Using the clone method

Element by element copy is copying the elements ourselves utilizing a loop:

```
for(int i = 0; i < arr1.length; i++) {

arr2[i] = arr1[i];

}
```

The **System class' arraycopy** method will attempt to copy all, or part, of one array to another. The arraycopy method does a shallow copy and only the contents of primary array are copied including any references. The beginning position in each array is specified, along with the number of elements to copy.

Syntax of arraycopy is:

arraycopy(Object src, int srcPosition, Object dest, int destPosision, int length)

Example:

System.arraycopy(arr1, 0, arr2, 0, 5);

This will copy the contents of arr1 from position 0 into array arr2 from position 0 to 5.

arraycopy does a copy from the location specified replacing the current contents.

If the operation goes beyond the size of the array, you will get java.lang.ArrayIndexOutOfBoundsException.

The **Arrays.copyOf method** creates a new array based on an existing array. The existing array is passed in as the first argument and the number of elements to copy is the second argument.
arr2 = Arrays.copyOf(arr1, 5);

The new array can be larger than the original array if we provide a larger number than the size of the original array as the second argument. In such cases, extra locations will be padded with defaults, for instance 0 for int.

The **Arrays.copyOfRange method** creates a new array from a range of elements in an existing array. The existing array is passed in as the first argument. Its second argument specifies the beginning index inclusive and the last argument specifies the ending index exclusive.
arr2 = Arrays.copyOfRange(arr1, 2, 5);

If we provide a larger number than the size of the original array as the second argument, extra locations will be padded with defaults, for instance 0 for int, similar to Arrays.copyOf.

Object class' clone method makes a shallow copy of the original object. As this would be fine for a primitive array, a copy of an array of objects through Object's clone will point to the same object references as only references are copied.
arr2 = arr1.clone();

Source : http://javajee.com/traversing-comparing-and-copying-arrays