Introduction to Arrays

There are often times when it would be convenient to be able to work with a series of variables that are very similar to make it easier to process them. One mechanism for accomplishing this is the Array. An Array is a list of variables that are all the same type. Arrays have rules regarding how they are created and used, but are mostly just another type of variable.

Let's look at an example of why an Array would be useful:

Creating a list of names with each variable having a different name in it without using arrays would look like this using a separate variable for each `String`:

```
String name1="Adam";
String name2="Becky";
String name3="Charlie";
String name4="Dennis";
```

And in order to print each `String` variable the code would look like this:

```
System.out.println(name1);
System.out.println(name2);
System.out.println(name3);
System.out.println(name4);
```

It should be easy to see that this system becomes very untenable if 10, 20, or 10,000 names are needed!

Arrays allow us to use a slightly more convenient system for both creating and using a list of variables that are all the same type.

First, we can use a convenient shorthand system for creating and filling the Array with all the names at once:
Let us examine this statement in detail:

```
String[] names={"Adam","Becky","Charlie","Dennis"};
```

Like any variable, first we must specify the type. In this case, we first indicate we are making a `String` variable, but then we use the `[]` characters, known as square braces to indicate it is an Array of Strings, so we have the type: `String[]`.

Just as with all other variables, the variable name comes next. Since this is not just one name but an Array where each element is a name, calling the variable `names` helps us remember it is an Array.

After the variable name we usually assign the variable an initial value. In this case we are using a JAVA convention to fill the Array with the values “Adam”, “Becky”, “Charlie”, and “Dennis” using the syntax of curly braces then a list of values. Since this is an Array of Strings each value must be a `String` so we must put each value in quotes.

If we were to draw a picture of the two ways we created our variables it would look like this:

Without using Arrays:

Variable Name	Value
name1	"Adam"
name2	"Becky"
name3	"Charlie"
name4	"Dennis"

Using Arrays:

Array Name	Index	Full Variable Name	Value
names	0	names[0]	"Adam"
names	1	names[1]	"Becky"
names	2	names[2]	"Charlie"
names	3	names[3]	"Dennis"

From this point on, an Array will be summarized by indicating the Variable Type and Name above the data summary with the index and values in the table. For the current example it looks like this:

`String[] names`

index	Value
0	"Adam"
1	"Becky"
2	"Charlie"
3	"Dennis"

How do I use an array in my program?

Since the variable "names" is actually a list of 4 String variables, we need to indicate which one we want when we wish to use it in a program. To output the first and last names in our list to the console we can write:

```
System.out.println(names[0]);
System.out.println(names[3]);
```

Which generates this console output:

```
Adam
Dennis
```

So, how do Arrays help us?

Once an Array is filled with data, we can now use other structures we are familiar with to help us process it. For example, printing the values in the Array can be accomplished using a loop to change the index of which Name we want to print. By changing the index variable `i` from `0` to `3`, we can output all the names:

```
for (int i=0; i<=3; i++) {
   System.out.println(names[i]);
}
```

```
Adam
Becky
Charlie
Dennis
```

Arrays also include a way to find out how many elements there are in the Array by writing `arrayName.length`. In our example, we can store the number of elements in the Array by assigning it to an integer. In this case, if we output this value to the console we will see the number 4 because there are four names in our list. Note that the index of the first element is always 0, therefore the index of the last element is one less than the length of the array!

```
int elements=names.length;
System.out.println(elements);
```

We can use the length of the array when designing our "for loop" to make it more robust. A for loop that is written using a set size can't easily be changed, but by calculating the array's length in the loop, the code will still work if we go back and add more elements to it.

```
int elements=names.length;
for (int i=0; i<=elements-1; i++) {
      System.out.println(names[i]);
}
```

A more compact way to write the loop looks like this:

Notice that we've also changed the code by using "less than" rather than "less than or equal to" and then subtracting one.

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

Examine the code below:

```
public class ArrayIntroduction2 {
  public static void main(String[] args) {
    String[] names={"Adam","Becky","Charlie","Dennis"};
    for (int i=0; i<names.length; i++) {
      System.out.println(names[i]);
    }
  }
}
```

Which generates the following console output:

```
Adam
Becky
Charlie
Dennis
```

Written in this manner, the program will work correctly regardless of how many elements we add to the array:

```
public class ArrayIntroduction2 {
  public static void main(String[] args) {
    String[] names={"Adam","Becky","Charlie","Dennis","Edith"};
    for (int i=0; i<names.length; i++) {
      System.out.println(names[i]);
    }
  }
}
```

Output:

```
Adam
Becky
Charlie
Dennis
Edith
```

Here are some examples of creating arrays of other primitive data types:

```
int[] arr1={1,2,3};
double[] arr2={.5,1,1.5};
boolean[] arr3={true,false,true};
```

`int[] arr1`

index	Value
0	1
1	2
2	3

`double[] arr2`

index	Value
0	.5
1	1.0
2	1.5

`boolean[] arr3`

index	Value
0	true
1	false
2	true

Declaring Arrays

So far all the arrays have been created using shortcut syntax with curly braces:

```
type[] varName={value1,…,valuen};
```

The more formal method of declaring an array uses this syntax:

```
type[] varName=new type[num];
```

So declaring a String array with 10 elements use this syntax:

```
String[] names=new String[10];
```

Just like before, the type can be **int**, **double**, **boolean**, String, or other type. The array has num elements in it, and the initial value of each element depends upon the type. The table to the right shows the initial value in an array of each type.

NOTE: Other primitive data types and objects also have default values (not shown).

Type	Initial Value
int	0
double	0.0
boolean	false
String	null

Generally speaking however, it is considered "best practice" to assume that an array declared using this syntax has null or unknown values and should therefore be "initialized" explicitly before being used.

At this point you may be wondering what the value NULL means. In JAVA **null** is a reserved word that means that the variable in question has "no value." Variables like String that are Objects in JAVA have no value until they are assigned one. Until that time, they are said to be **null**, or have a **null** value.

In the case of String variables, don't confuse null with empty. Look at the code and output below:

```
public class NullEmptyString {
  public static void main(String[] args) {
    String[] arr=new String[5];
    arr[2]="";
    for (int i=0; i<arr.length; i++) {
       System.out.println("i["+i+"]=" + arr[i]);
    }
  }
}
```

Output:

```
i[0]=null
i[1]=null
i[2]=
i[3]=null
i[4]=null
```

Notice that the element at index 2 is the only one that doesn't output the value null. Arrays with null elements can be very tricky because they can easily trigger "NULL POINTER EXCEPTIONS." A null pointer exception occurs when you try to use a method on an object such as a string that has a null value.

The line of code arr[i].equals("Hello") generates an error because arr[0] has the value null so the equals method cannot be used. For this reason, it is always a good idea to initialize the values of your arrays to a known value using a for loop like:

```
for (int i=0; i<arr.length; i++) {
  arr[i]="";
}
```

This guarantees that no element in the array has the value **null**.

```
public class NullEmptyString {
  public static void main(String[] args) {
    String[] arr=new String[5];
    arr[2]="Hello";
    for (int i=0; i<arr.length; i++) {
       if (arr[i].equals("Hello")) {
            arr[i]="Hello";
       }
       System.out.println("i["+i+"]=" + arr[i]);
    }
  }
}
```

```
Exception in thread "main" java.lang.NullPointerException
        at NullEmptyString.main(NullEmptyString.java:6)
```

Initializing Numeric Arrays

Numeric arrays using `int` and `double` have the value 0 in all elements after being declared. It is often useful to fill such an array with values. For example, having the first element in the array start with one and each element after go up by one.

```
public class InitializeNumericArray {
  public static void main(String[] args) {
    int[] arr=new int[5];
    for (int i=0; i<arr.length; i++) {
      arr[i]=i+1;
    }
    for (int i=0; i<arr.length; i++) {
      System.out.println("i["+i+"]="+arr[i]);
    }
  }
}
```

Output:

```
i[0]=1
i[1]=2
i[2]=3
i[3]=4
i[4]=5
```

Limits and Restrictions to Arrays:

When using arrays there are some limits and restrictions that must be observed.
First, an array must contain all elements that are the same type of variable:
These are all valid array declarations:

```
String[] arr={"One","2","Three","4"};
int[] arr={1,2,3,4};
double[] arr={1.0,2,3.5,4};
```

Each of these arrays can have all their values output to the console using the same code:

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

Below is output that would result from each:

Output with:

`arr={"One","2","Three","4"}`

```
One
2
Three
4
```

Output with:

`arr={1,2,3,4}`

```
1
2
3
4
```

Output with:

`arr={1.0,2,3.5,4}`

```
1.0
2.0
3.5
4.0
```

None of these are valid array declarations:

```
String[] arr1={"One",1,Two,"3"};
```

`arr1` is a `String` array so each of it's elements must be a `String`. In the declaration `arr1={"One",1,Two,"3"}`, the second and third elements don't have quotes around them so they aren't Strings.

```
int[] arr2={1,"2",3.5,4};
```

`arr2` is an `int` array so each of it's elements must be an `int`. In the declaration `arr2={1,"2",3.5,4}`, element 2 is a String, and element 3 is a double.

```
double[] arr3={1.0,"2.0","3",4};
```

`arr3` is a `double` array so each of it's elements must also be a `double`. In the declaration `arr3={1.0,"2.0","3",4}`, elements 2 and 3 are Strings. Element 4 is okay because it is converted automatically to 4.0.

Restriction: Arrays are fixed length

Once you have declared an area with a number of elements you cannot add new elements to it or remove elements from it, but you can replace an array with another array that has a different size. See the example below:

```
public class ReplaceArray {
  public static void main(String[] args) {
    String[] names={"Ann","Barbara","Chuck"};
    String[] names2={"Fred","Wilma","Pebbles","Barney","Betty","Bam-Bam"};
    System.out.println("BEFORE:");
    for (int i=0; i<names.length; i++) {
      System.out.println(names[i]);
    }
    names=names2;
    System.out.println("AFTER:");
    for (int i=0; i<names.length; i++) {
      System.out.println(names[i]);
    }
  }
}
```

This line of code "erases" the Array information stored in names. From this point on, names and names2 ***are*** the same array. Changing any part of one, changes both because there aren't actually two arrays anymore!

Output:

```
BEFORE:
Ann
Barbara
Chuck
AFTER:
Fred
Wilma
Pebbles
Barney
Betty
Bam-Bam
```

In the example above, the first array is replaced with a second array that has more elements than the original array.

Restriction: Arrays are "bounded"

The other important restriction when working with arrays is that you can only refer to array elements that exist. If an array has only four elements in it then trying to access the fifth element with generate an array index out of bounds error.

See the below for the most common example of how this occurs:

```
public class ArrayIndexOutOfBounds {
  public static void main(String[] args) {
    String[] names={"Ann","Betty","Chuck","Dale","Ethan"};
    for (int i=0; i<=names.length; i++) {
       System.out.println("i=" + i + ": names["+i+"]=" + names[i]);
    }
  }
}
```

Generates this console output:

```
i=0: names[0]=Ann
i=1: names[1]=Betty
i=2: names[2]=Chuck
i=3: names[3]=Dale
i=4: names[4]=Ethan
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5
       at ArrayIndexOutOfBounds.main(ArrayIndexOutOfBounds.java:5)
```

String names[]

index	Value
0	"Ann"
1	"Betty"
2	"Chuck"
3	"Dale"
4	"Ethan"

This code generates the `ArrayIndexOutOfBoundsException` because the index variable `i` increases by one until it is less than or equal to `names.length` which is 5. But the first element in the array has an index of 0 and the last an index of 4 so trying to access the element at array index 5 generates the exception.

The line of code that generates the error is:

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

It should read:

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