## [Java] Recursion when using Primitive data types and Objects [Beginner's Guide]

<https://leetcode.com/discuss/study-guide/2044134/javarecursion-when-using-primitive-data-types-and-objects-beginners-guide>

A very fundamental mistake which beginners(like me, lol) might find themselves in is while using the "variables" and how they work internally. So, if you are jumping straight into problems without this fundamental piece of knowledge, you might run into trouble.

Have a look into this code:

public void fun(int i) // -----> i is passed as '1'

{

if(i == 3)

return;

System.out.println(i);

fun(i + 1);

System.out.println(i);

}

The output will be:

1

2

2

1 (---> Retains it previous value)

This is general recursion which you are possibly aware of.

Although it's worth noticing that the variable i ' retains ' its previous value( i.e. =1) in the process.

Now,

public void fun(boolean[] arr, int i) //---------> Initial value of the elements of arr[] are: false, false

{

if(i == 3)

return;

arr[i] = true;

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

fun(arr, i + 1);

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

}

The output of the code will turn out to be:

true

true

true

true

**Take notice that the last value doesn't revert back to false as it did earlier.**

Now the reason behind this is: **Array are objects**

**Unlike the primitive data types which are passed by value, objects are passed as reference. This means that, whenever the recursion is called with primitive data types in the parameters, every recursion call will allocate a separate memory for its primitive data types.**

Suppose, a function is called two times recursively, then every recursive call will have access to their respective value of the primitive data types. Referring to the first example:

public void fun(int i)

{

if(i == 3)

return;

System.out.println(i);

fun(i + 1);

System.out.println(i);

}

Here first recursion call will have the value of i : 1

second recursion call will have value of i : 2

Hence when we **return** to recursive call *1*, we are able to access the value *1*.

But, in the second example:

public void fun(boolean[] arr, int i)

{

if(i == 3)

return;

arr[i] = true;

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

fun(arr, i + 1);

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

}

Here, the boolean[ ] arr in the parameter is just a pointer to 'arr' object.

So,

(After the value is set to true)

In recursive call 1: A pointer which points to an array where element is true

In recursive call 2: A pointer which points to an array where element is true

Hence, when we return to recursive call *1* all we have is "A pointer which points to an array where element is true".

This is what I meant to say:

**\*\*THE VALUES IN ARRAY ARE CHANGED FOR GOOD. THEY WONT REVERT BACK TO ITS PREVIOUS STATE SINCE EVERY RECURSIVE CALL WON'T CREATE A NEW MEMORY FOR THE ARRAY UNLIKE THE PRIMITIVE DATA TYPES. \*\***

Although things are different for **String**. Even though its an object it will have the **same behavior as that of the primitive data types** since they are **immutable**.

public void f(String str, int i)

{

if(i == 3)

return;

System.out.println(str);

f(str + "a", i + 1);

System.out.println(str);

}

Output:

str

stra

stra

str

**Clearly, the String is able to "revert" back. So, this is an exception among the objects.**

**Summary:**

(1) When using primitive data types in parameters, **while returning**, one will be able to access the values of the variables in the respective recursive calls

(2) When using objects(anything which is initialized using the "new" keyword), one will **not** be able to access the previous values. This is where you see lots of problems having the line of code such as:

visited[i][j] = false, or

list.remove(i), etc

at the end of the recursive call. These lines are explicitly meant to revert back to the previous state, **we call it backtracking**

(3) Even though String is an object, **it behaves as a primitive data type since String is immutable.**