## [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 (**[Only way to update original primitive variable is by returning value from the function](https://stacktraceguru.com/java/java-is-pass-by-value#update-primitive-value)**)**, 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, same for other wrapper class, e.g Integer**

**Example 1: Deep Copy and Backtracking**

<https://leetcode.com/problems/subsets/discuss/935783/java-pass-by-reference-or-value-in-recursion>

class Solution {

public List<List<Integer>> subsets(int[] nums) {

if (nums == null) return new ArrayList<>();

List<List<Integer>> result = new ArrayList<>();

List<Integer> numlist = new ArrayList<>();

dfs(nums, 0, numlist, result);

return result;

}

private void dfs(int[] nums, int index, List<Integer> numlist, List<List<Integer>> set) {

if (index == nums.length) {

// set.add(numlist); // ->>>>>>>>>> WRONG

set.add(new ArrayList<>(numlist)); // ->>>>>>>>> RIGHT

return;

}

// left:

numlist.add(nums[index]);

dfs(nums, index + 1, numlist, set);

// backtrack:

numlist.remove(numlist.size() - 1); // delete the last char in the sb and backtrack

dfs(nums, index + 1, numlist, set);

}

Q: I am very confused that, in the line I marked as WRONG and RIGHT, why is that when I use "set.add(numlist)" the result will be all empty which is [[],[],[],[],[],[]], but when I changed it to "set.add(new ArrayList<>(numlist))", the result is correct. Why do I have to create a new value copy of the numlist to be added to the result? I thought JAVA is always pass-by-reference, so every time it should be the value of numlist to be added to the result, and then remain unchanged, but why the the result arraylist changed if I didn't create a copy of the value?

I am very confused and thought it should be something related to reference/value.

A: Because when you do "numlist.add(nums[index]);", "set.add(numlist);" and " numlist.remove(numlist.size() - 1);", you are working on the same numlist object reference.. so, you need to take a copy of that list when you add it in the set, like "set.add(List.copyOf(numlist));"

**Example 2: Deep Copy**

<https://stackoverflow.com/questions/3171063/java-recursion-calling-it-with-objects-how-to-copy-the-objects>

The old value/reference things. Im getting ConcurrentModificationException for this adaptation of the Bron-Kerbosch.

public int[] bk(ArrayList<Integer> R, ArrayList<Integer> P, ArrayList<Integer> X) {

int count[] = new int[n];

int u=0, c = 0;

ArrayList<Integer> tempPX = new ArrayList<Integer>();

ArrayList<Integer> newP = P;

ArrayList<Integer> newX = X;

ArrayList<Integer> newR = R;

if (P.isEmpty() && X.isEmpty()) {

count[R.size()]++;

} else {

u = 0; c = 0; // find vertex with largest degree

tempPX.addAll(P); tempPX.addAll(X); // P â‹ƒ X

for (Integer v : tempPX) {

if (neighbours[v].size() > neighbours[u].size()) {

u = c;

}

c++;

}

P.removeAll(neighbours[u]); // P \ neighbours[u]

for (Integer v : newP) {

newR.add(v); // R â‹ƒ v

newP.retainAll(neighbours[v]); // P â‹‚ neighbours[v]

newX.retainAll(neighbours[v]); // X â‹‚ neighbours[v]

bk(newR, newP, newX);

P.remove(v); // removing object

X.add(v); // X â‹ƒ v

}

}

return count;

}

Q: The exception occurs at line for (Integer v : newP), and the recursive call in there. I need to P.removeAll(neighbours[u]); then loop over that resulting list, inside doing the things in the comments, AND PASS COPIES in the recursive call so it wont complain and work not pass the references and keep modifying the same object P/X/R. So how and WHEN do i copy them?? Those first lines.. I'm making copies of the references aren't i... (yes i know i "modify" newP then loop over the old P, they just point to the same object it seems)

A: As you've identified, you're copying the reference, not the list. You need to instantiate a new ArrayList object with the entries in the old list.

e.g.

```

List<Integer> newList = new ArrayList<Integer>(oldList);

```

So this explicitly creates a new object containing references to the same elements.

Note as an aside that I pass around a reference to the interface List rather than the implementation - generally good practise since you're not exposing implementation throughout the codebase.

In simple:

ArrayList newP = P;

This only creates a second reference to the same ArrayList. To copy the arraylist use

ArrayList newP = new ArrayList(P);

New code after implement deep copy logic:

public int[] bk(List<Integer> r, List<Integer> p, List<Integer> x) {

int count[] = new int[n];

int u = 1;

List<Integer> tempPX = new ArrayList<Integer>();

List<Integer> newR, newP, newX;

if (p.isEmpty() && x.isEmpty()) {

count[r.size()]++;

} else {

// find vertex with largest degree in P U X

tempPX.addAll(p);

tempPX.addAll(x);

for (Integer v : tempPX) {

if (neighbours[v].size() > neighbours[u].size()) {

u = v;

}

}

p.removeAll(neighbours[u]); // P \ neighbours[u]

newP = new ArrayList<Integer>(p);

for (Integer v : newP) {

r.add(v); // R U v

newR = new ArrayList<Integer>(r);

p.retainAll(neighbours[v]); // P /\ neighbours[v]

newP = new ArrayList<Integer>(p);

x.retainAll(neighbours[v]); // X /\ neighbours[v]

newX = new ArrayList<Integer>(x);

bk(newR, newP, newX);

p.remove(v); // removing object

x.add(v); // X U v

}

}

return count;

}

**Example 3: Java Recursive pass in object by reference?**

<https://stackoverflow.com/questions/7848312/recursive-pass-in-object-by-reference-java>

Q: So I have an algorithm that forces me to pass in an object as a parameter recursively and on different depths of the recursion it sets the values of the object. The problem is Java isn't allowing me to do this because they pass in by value or something else that is just messing me up. How can you make sure that the object passed in retains the value set?

A: You *never* pass an object as an argument in Java. You either pass a reference or a primitive value. The argument is always passed by value. Java doesn't have pass-by-reference at all.

If you're passing in a reference to an object, and you want to make sure the code you call doesn't change the data within that object, your options are:

* Create a copy and pass in a reference to that instead
* Make your type immutable in the first place.

EDIT: Just to make it clear, if you have:

Foo f = new Foo();

then the value of f is *not* a Foo object. It's a *reference* to a Foo object. If you now call:

doSomething(f);

then the value of f is copied as the original value of the parameter within doSomething. That behavior is basically the definition of pass-by-value. There's nothing doSomething can to do change the value of f - it will still refer to the same object afterwards. The data *within* that object may have changed, but it won't be a different object.

In a real-world analogy: if I give you a copy of my address, you can go and paint the front door, but you can't change where I live.