Java Object References and Memory

## Assigning a Variable an Object Instance

When you assign a **new** **object** to a variable, you are doing the following two things:

1. Instantiating a new object and storing it in memory.
2. Assigning the memory address of the object to the variable.

If you assign an **existing** **object** to a variable, you are only updating the memory address held by the variable to that of the existing object (i.e., step 2 in the list above).

1. Star sun = new Star();

2. Star pointMeToTheSun = sun;

In the code above, both `sun` and `pointMeToTheSun` hold the same memory address, that is, the memory address at which the `Star` object (created in line 1) is located.

Any method calls on this object, regardless of whether they are called using the `sun` or the `pointMeToTheSun` variable, will update the same object. This is because when you call a method on an object, you are modifying the data within the existing object. The reference to the object remains the same, but the contents of the object may change.

## Assigning a Variable a Primitive Data Type

When you assign a **primitive data type** to a variable in Java, the actual value is stored directly in the variable.

*\*An object is never truly assigned to a variable in the same sense as how a primitive data type is. Rather, a variable will only hold the address to an object, but never an actual object itself.*

int a = 2;  
int b = a;

In the above code, when `a` is assigned to `b`, the value 2 is being copied. There are now two separate instances of the value 2 at two different locations in memory.

## Working with Collections

When you put an object into a collection, the memory address of the object gets stored in the collection. That is, the collection holds a reference to the object, not a copy of it. The object's memory address does not change when it is put into a collection.

ArrayList<Star> stars = new ArrayList<Star>();  
stars.add(sun);  
stars.add(pointMeToTheSun);

In the above code, both elements in the `stars` ArrayList hold the same memory address. However, as Java does not have an option to directly obtain the memory address of a variable like you can in C++ using the `&` operator, below is an example in C++.

This code demonstrates that when an object is instantiated, its memory address assigned to a variable, then to another variable using the first variable name, and both variables are added to an array, both variables and array indexes still hold the same memory address:

int main() {  
  
 Star\* sun = new Star();  
 Star\* pointMeToTheSun = sun;  
  
 Star\* stars[] = {sun, pointMeToTheSun};  
  
 for(int i = 0; i < sizeof(stars) / sizeof(stars[0]); i++){  
 std::cout << stars[i] << std::endl;  
 }  
  
 delete sun;  
  
 return 0;  
}

The console output of this code is:

0x600000245240

0x600000245240

Process finished with exit code 0

*\*Note I did not use a std::vector or range-based for loop in this example, as these create copies of objects by default. Also, I explicitly used pointers rather than regular variables, as this is what is happening in languages like Java and C# under the hood.*

## Java Example Using Collections

All of the information discussed above is necessary to understand the following algorithm, which is used for parsing the command line arguments passed to a Java application:

private static HashMap<String, ArrayList<String>> parseArgs(String[] args) {  
 HashMap<String, ArrayList<String>> parsedArgs = new HashMap<>();  
 ArrayList<String> argValues = null;  
 for (String arg : args) {  
 if (arg.startsWith("-")) {  
 argValues = new ArrayList<>();  
 parsedArgs.put(arg, argValues);  
 continue;  
 }  
 if (argValues != null) {  
 argValues.add(arg);  
 }  
 }  
 return parsedArgs;  
}

When the `parseArgs` method receives a flag (as identified by a `-` prefix), it:

* Adds a reference (pointer) to this flag (string) as a new key in the `parsedArgs` HashMap, with the same name as the string content itself.
* Creates a new ArrayList object and updates the `argValues` variable to point to the memory location of this `ArrayList`.

Each non-flag argument is added to the `argValues` ArrayList, until another flag (or the end of the `args` array) is reached.

Every time that a new flag is reached:

* A new ArrayList is created and assigned to argValues (that is - a new ArrayList object is created, and the `argValues` variable is updated to hold the memory address this new ArrayList.
* This newly created ArrayList is `put` into the `parsedArgs` HashMap as a new key (that is, the key holds the memory address of the new ArrayList).

## Garbage Collection

Because the `parsedArgs` HashMap holds the address of (a reference to) the ArrayList of parameters associated with each string key, when the `argValues` variable is updated to point to a different memory location (by being assigned a new ArrayList object for a new flag), the previous ArrayList remains in memory and therefore remains accessible until there are no longer any references to it (or the program stops executing).

**An object will remain in memory until there are no longer any references to it, at which point it becomes eligible for garbage collection (in languages with automatic garbage collection like Java and C#)**.

If the `parsedArgs` HashMap was updated to point to another object without the previous object it pointed to being pointed to (referenced) by another variable (or index in a collection, as in this example), it would no longer be accessible, and would become eligible for garbage collection.