Allocation in Java

Allocation refers to the process by which a variable or value is placed into the computer's memory. *Where* and *how* something is initialized depends on its type, in particular whether it is a primitive or object type.

Scope and Variable Lifetimes

Scope refers to the region of code in which a particular variable – whether local, instance, or parameter – is accessible. Most languages follow the same notable scope rules: local variables cannot be used outside of their method/function; a variable declared in a nested block can't be used outside that block; etc.

Exercise 1. Identify the scopes of the variables `x` and `y` and `z` in the following fragment:

```
public static int doSomething(int x) {
    x = x / 2;
    int y = x;
    if (x > 0) {
        int z = y % x;
        y = z + 1;
    }
    return y;
}
```

Every variable in a program that has a value requires space in the computer's main memory (random access memory – RAM). We want to write programs that are efficient in their memory usage; RAM is a finite resource, and we want to preserve as much of it for other uses as possible. To do that, we tie a variable's scope to its **lifetime**: a variable only needs memory while it is "in scope" / "alive", and as soon as it leaves scope, its lifetime ends and can be destroyed.

Exercise 2. For the three variables in Exercise 1, argue why each can be destroyed once it is no longer in scope.

Automatic Lifetime and the Stack

So where does a variable "live" when it is in scope, and who is responsible for putting it there? In a region of memory called the **stack**, which is 100% managed by Java. When a variable enters scope, it is **pushed** onto the stack; when it leaves scope, it is **popped**. We can show that *every single time* a variable goes out of scope, *that exact variable* is *always* the top element of the stack.

Exercise 3. Push the variables in Exercise 1 onto a stack in the order they come into scope. Show that the variable at the top of the stack *always* gets popped before any variable below it needs to.

This is true even when we jump to other methods, use conditional statements, loops, etc. The next variable to die is always at the top of the stack. I Guarantee It.

Java does all this for us. When a variable in your code comes into scope, it is given space on top of the current stack – enough space to fit its size (if a primitive type), or 8 bytes (if an object type). When you use and manipulate that variable in code, its memory address on the stack is what is read and written to.

We know that primitive types store their actual instance where the variable is, so if `x` has a value of 10, then the 4-byte value for 10 is stored on the stack where `x` is allocated. But what about object types, which don't store instances, but *references* to instances? Where is the actual instance? **Not on the stack!**

Dynamic Lifetime and the Heap

Object type instances are not stored on the stack in Java; they are stored in a separate region of memory called the **heap**. Unlike the stack, the heap is unordered, and we cannot predict where instances will be placed when they are created on the heap. When we create a variable of an object type and initialize it with an instance, the picture we get is more complicated than with primitive types:

Exercise 4. In the following fragment, where is the variable `x` allocated, where is the `Scanner` **instance** allocated, and what does `x` store as its **value**?

```
public static void doSomething() {
    Scanner x = new Scanner(System.in);
    ...
}
```

To repeat: **all variables** are allocated on the stack, but their **instances** will be on the heap if they are an object type.

Since scope only applies to *variables* and not *instances*, it's unclear at the moment whether destroying `x` when it leaves scope in Excercise 4 will also destroy its `Scanner` instance. Let's examine arguments both ways; **when destroying an object type variable, should we also destroy its instance on the heap?**

Argument in favor:

Yes, when `x` is destroyed, **clearly** no one else is using that particular `Scanner` instance. Why leave it in memory? Get rid of it!

Huzzah! Heap instances should be destroyed when their variable is destroyed, no questions asked!

Argument against:

Consider the `...` in Exercise 4. What if we replaced it with:

```
public static void doSomething() {
    Scanner x = new Scanner(System.in);
    if (true) {
        Scanner y = x;
    }
    int z = x.nextInt();
}
```

The `if` statement creates a new variable `y` that refers to the same `Scanner` as `x`, since they are object types. `y` goes out of scope when the block ends... at that point, should we destroy the instance it refers to? How could we use `x` on the next line if we did that? Catastrophe!!!

It gets more complicated. What if our method *returns* a `Scanner`, the instance that `x` refers to? We can't destroy that instance when `x` dies because it needs to act as our return value. What if we pass `x` to another method? When that method's parameter goes out of scope, we can't destroy its instance because that's the

same instance that `x` needs, and `x` is still alive!!! There are **literally dozens** of scenarios in which a local object type variable going out of scope **should not** trigger the destruction of its instance.

That's kind of the point of references and object types. They are supremely useful for accomplishing **dynamic lifetimes**: values/instances that are not assumed to be dead the moment a variable goes out of scope, but rather "live on" beyond the scope of their variable, allowing them to be constructed in one scope and then shared to others, even after the initial scope is closed. There are other reasons to use heap allocation, but they don't always apply to Java.

Destroying Heap Instances

So we agree that objet type instances should not be destroyed when a variable goes out of scope. But *what if* this is seriously, honest-to-god, the ***last variable*** that is still referring to some object instance? There's no way for us to ever use this instance in the future once its last referrer is destroyed; we've lost the only map to that location and cannot recover it. Surely we can destroy that instance now?

Yes, we can. And Java will... but probably not right now. Java uses a process called a **garbage collector** to *periodically* find object instances on the heap that are no longer being referred to. This process is complicated, and you will learn more in your future coursework. But for now, have faith: we can't "leak" memory in Java, and when your object type variables go out of scope, their instances will eventually be reclaimed if they are no longer needed.

Exceptions to the Rules

There are, of course, exceptions.

1. **Fields/instance variable**. The phrase "all variables go on the stack" is really referring to **temporary variables**: locals and parameters, which are temporary and only exist in their immediate (and *obvious*) scopes. The fields/instance varaibles of a class have much less obvious scope/lifetimes. These variables do not go on the stack; they go "inside" the memory of the object they correspond to, whenever that object is actually constructed on the heap. For example, a `Student` instance has an `int id` field; when a `Student` instance is created on the heap, the memory for that instance contains 4 bytes for its `id` plus enough memory to fit all the other fields. Since `id` is a primitive type, its 4 bytes stores its integer instance. If other fields are object types, their 8 bytes will store references to their instances which will be in other locations on the heap.

2. **Static variables**. A `static` variable does not have a limited scope – it can be used (depending on access permissions) from anywhere in the program – so it has an **unlimited lifetime**. A static *variable* is not placed on the stack, instead in a special region called **static memory**. Its instance is either stored directly in the variable for primitive types, or stored on the heap for object types. It is only *the variable itself* which is treated differently.

3. **String literals**: a string literal is an actual string typed into your source code: `"Hello!"`. It is *not* the result of building or manipulating a string at runtime: `String s = scanner.nextLine()`. We don't need multiple instances of the same string literal, so Java will collect *all* the string literals in your program and **intern** them into a structure called a **string table**/**string pool**. This table is usually in static memory; it is never garbage collected.