Java OO Concepts Unit

Lesson 01.3: References, Storage, and Memory Management

Coding Bootcamp

SOFTWARE GUILD

Lesson 01.3: References, Storage, and Memory Management

Overview

In this portion of Lesson 01, we'll look at how and where data is stored in memory and how objects are referenced.

Constructors and the 'new' Keyword

We have already discussed constructors a bit and we saw that they are just special methods that are called when we create new objects of a type. When the new keyword and one of the constructors of a class are used together, the JVM creates a new object of that particular type on the **heap** and hands us back a **reference** to this object.

References and Storage

Variables of user-created types that we declare and use in our programs are called **references**. These variables point (or refer) to an object on the heap. They are similar to C pointers in some ways (for example, many references can point to the same object on the heap), but they do not have access to the underlying memory location. The memory for the reference variable itself is created on the **stack**.

Garbage Collection

As we have discussed, Java is a managed language. This means that the programmer is not responsible for allocating and releasing memory manually. In Java, we simply create objects and/or native data types as we need them in our programs and the JVM takes care of releasing all memory as appropriate. This process is called **garbage collection** and the component responsible for doing this is called the **garbage collector**.

Memory used by objects on the heap is eligible for garbage collection when there are no more references to the object. We can explicitly take a reference away from an object on the heap by setting the reference to **null**. Because the reference variables are stored on the heap, they will go away when their respective stack frames go away (i.e. when their enclosing method returns), which also removes the references from the object to which they point.

Pass by Value vs. Pass by Reference

Conceptually, there are two ways to pass parameters into a method — pass by value and pass by reference. Pass by value copies that value of the input parameter into another stack memory location so that the original variable and the parameter variable are completely disconnected. Pass by reference passes a pointer to the parameter's location, which means that any changes to the variable made inside the method also affect the original variable.

Java is a pass by value language.

This can be a source of some confusion for beginning Java developers because of the way references to objects work.

Let's start with native types. Native types work exactly as you would expect in a pass by value environment. If I pass an int into a method and the method changes the value of the parameter, it has no effect on the variable outside the method.

Now: user-defined types. If I pass a reference to an object into a method and the method changes a property on the object (for example, updates the last name field on a student object), **that change will be reflected in the object reference outside the method!** However, if I set the passed-in parameter to null inside the method, the object reference outside the method remains unaffected.

Why is this? This behavior makes sense if you remember that an object can have more than one reference pointing to it. When you pass a reference to an object into a method in Java, the reference is copied — now you have two things pointing to the same object, like having two remotes pointing to the same TV. As long as they both point to the same object, anything done by either to change properties on the object will be seen by both references. However, if you make one of the references point to another object (or null), it will not affect the other reference.