# Lesson 1 – Memory & Variable Types

## Variables

So far we have considered variables as being temporary storage for a given kind of data e.g. int x to store a whole numbers, double balance to store a decimal value, string text to store alphanumeric values, etc.

We have not taken into consideration that there are:

1. Two different Categories of Variables
2. Runtime and Memory deal with these 2 Categories Differently

## The Two Different Categories: Value Types and Reference Types

Value Types are what we most commonly associate with normal variables i.e. a collection of memory locations that store an actual value in them. All Primitive Data Types such as int, bool, short, long, byte, char, decimal (better known as structs) and Enumumerated Types (enums) are all Value Types.

Example: int x = 5;

5

X 🡪

When a Value Type is created, a single space in memory is allocated to store the value. When the runtime deals with a value type, it deals directly with its data.

This means that, if we create 2 value types and make them equal to each other, we have simply created 2 independent copies of the same thing; changes to one value type will not affect the other!

Example: int x = 5; int y = x;

5

X

Y

5

Y is a completely different set of memory locations from X, a different and independent variable therefore any changes to it will leave the value of X intact.

Example: int x = 5; int y = x; y = null;

5

X

~~5~~ null

Y

**So far, this seems pretty obvious but…..**

Objects, including String *(which we often mistake for a Primitive Data Type) are not Value Types!*

*These are* Reference Types*, hence, they behave differently from stucts and enums!*

Reference Types only store references to their data in memory, they do not store the actual data (or object itself). When creating a Reference Type, an object is created in memory but it is then handled through a separate reference – rather like a pointer.

Reference Types include all classes and interfaces created within a program as well as some in-built structures such as Object (the parent class for all objects created) and String.

Example: Object o = new Object ();

Object()

**o**

🡪

This means that, 2 Reference Types could actually be pointing to the same object! Therefore, operations on one Reference Type will affect the object referenced by the other variable.

Example: Object o = new Object (); Object o2 = o; o2 = null;

**o**

🡪

~~Object()~~

null

**o2**

🡪

***In this case, both o and o2 are equivalent to null as they were both pointing to the same object!***

## How Does Memory Deal with Value Types and Reference Types?

The Common Language Runtime allocates memory for objects in two places: the stack and the heap.

The stack is a simple first-in last-out memory structure, and is highly efficient.

The stack is always used to store the reference portion of reference-typed variables and parameters (i.e. the pointer part) and the actual value-typed variables and method parameters (including their data)

It is also responsible for keeping track of the program execution and what method is being executed.

When a method is invoked, the necessary frames and data are place on the top of the stack – this is the only accessible element (imagine stacking boxing on top of each other, you can only see what is in the top box). When the method (or current context) completes, the CLR just discards the topmost frame and all of its’ data so that the next element in the stack is accessible.

The heap can be pictured as a random jumble of objects. Its advantage is that it allows objects to be allocated or deallocated in a random order.

The heap is used to store the content of reference-type objects (i.e. the object itself) and anything within the reference-type object itself (i.e. any fields and methods within the object).

## How do Heaps and Stacks work together in a Program?

Both the heap and the stack are required during the runtime of a program, however, the heap requires the overhead of a memory manager and garbage collector to keep things in order. Luckily for us, .NET Framework handles these automatically for us but it was not always so!

Example:

public static void CreateNewObject()

{

Object o = new Object();

}

The above method creates a local variable, stored on the stack, which *references* the object and the object itself, which is stored on the heap:

o

Object()

***STACK*** ***HEAP***

Once CreateNewObject() has finished running, its local stack-allocated variable, o, will disappear from scope and be “popped” off the stack. This means that the Object in the heap does not have any references pointing to it which can access it and handle it. The Common Language Runtime’s garbage collector, will later on, automatically deallocate the object from the heap. The garbage collector will know to delete it, because the object has no valid reference.

This also means that data on the stack has a different life time then data on the heap!

Good Reads:

<http://www.albahari.com/valuevsreftypes.aspx>