Systemnahes Programmieren
Memory Management in Microsoft .Net

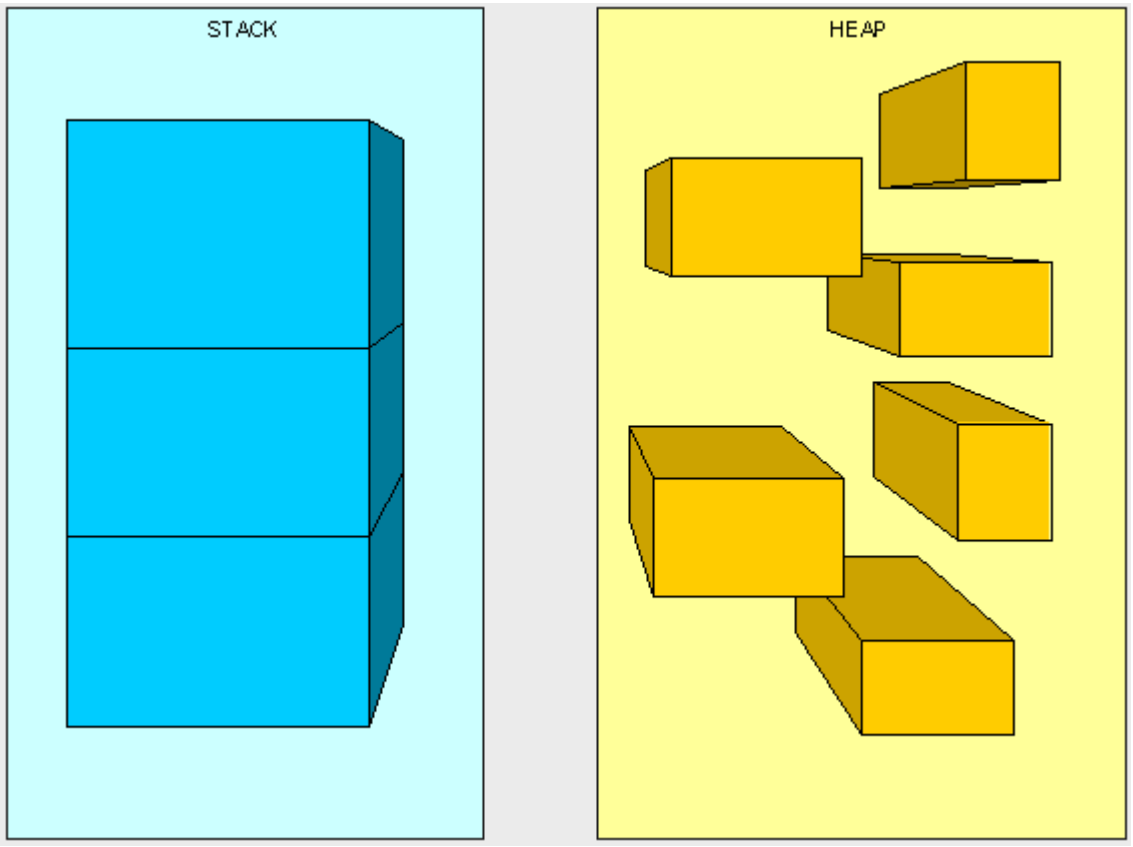
Even though with the .NET framework we don't have to actively worry about memory management and garbage collection (GC), we still have to keep memory management and GC in mind in order to optimize the performance of our applications. Also, having a basic understanding of how memory management works will help explain the behavior of the variables we work with in every program we write. In this article I'll cover the basics of the Stack and Heap, types of variables and why some variables work as they do.

There are two places the .NET framework stores items in memory as your code executes. If you haven't already met, let me introduce you to the Stack and the Heap. Both the stack and heap help us run our code. They reside in the operating memory on our machine and contain the pieces of information we need to make it all happen.

Stack vs. Heap: What's the difference?

The Stack is more or less responsible for keeping track of what's executing in our code (or what's been "called"). The Heap is more or less responsible for keeping track of our objects (our data, well... most of it - we'll get to that later.).

Think of the Stack as a series of boxes stacked one on top of the next. We keep track of what's going on in our application by stacking another box on top every time we call a method (called a Frame). We can only use what's in the top box on the stack. When we're done with the top box (the method is done executing) we throw it away and proceed to use the stuff in the previous box on the top of the stack. The Heap is similar except that its purpose is to hold information (not keep track of execution most of the time) so anything in our Heap can be accessed at any time. With the Heap, there are no constraints as to what can be accessed like in the stack. The Heap is like the heap of clean laundry on our bed that we have not taken the time to put away yet - we can grab what we need quickly. The Stack is like the stack of shoe boxes in the closet where we have to take off the top one to get to the one underneath it.



The picture above, while not really a true representation of what's happening in memory, helps us distinguish a Stack from a Heap.

The Stack is self-maintaining, meaning that it basically takes care of its own memory management. When the top box is no longer used, it's thrown out. The Heap, on the other hand, has to worry about Garbage collection (GC) - which deals with how to keep the Heap clean (no one wants dirty laundry laying around... it stinks!).

What goes on the Stack and Heap?

We have four main types of things we'll be putting in the Stack and Heap as our code is executing: Value Types, Reference Types, Pointers, and Instructions.

Value Types:

In C#, all the "things" declared with the following list of type declarations are Value types (because they are from System.ValueType):

- bool
- byte
- char
- decimal
- double
- enum
- float
- int
- long
- sbyte
- short
- struct
- uint

- ulong
- ushort

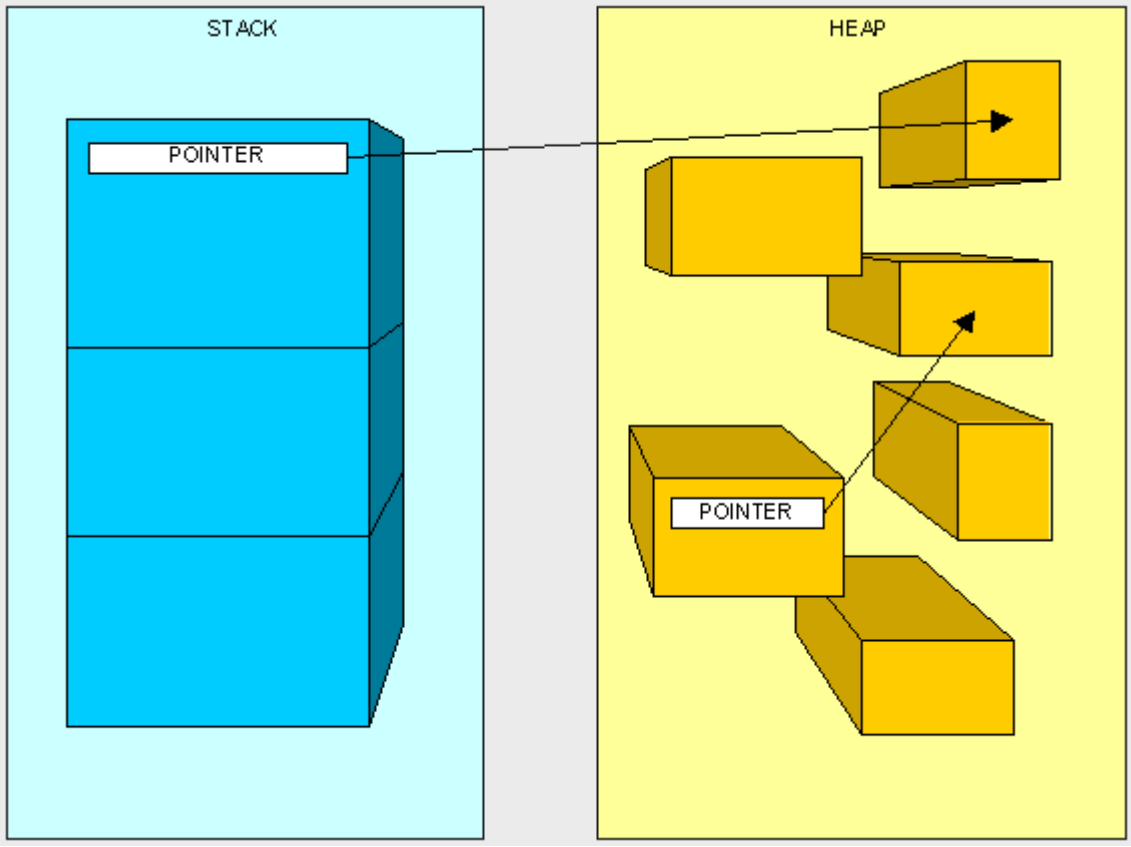
Reference Types:

All the "things" declared with the types in this list are Reference types (and inherit from System.Object... except, of course, for object which is the System.Object object):

- class
- interface
- delegate
- object
- string

Pointers:

The third type of "thing" to be put in our memory management scheme is a Reference to a Type. A Reference is often referred to as a Pointer. We don't explicitly use Pointers, they are managed by the Common Language Runtime (CLR). A Pointer (or Reference) is different than a Reference Type in that when we say something is a Reference Type is means we access it through a Pointer. A Pointer is a chunk of space in memory that points to another space in memory. A Pointer takes up space just like any other thing that we're putting in the Stack and Heap and its value is either a memory address or null.



Instructions:

You'll see how the "Instructions" work later in this article...

How is it decided what goes where? (Huh?)

Ok, one last thing and we'll get to the fun stuff.

Here are our two golden rules:

1. A Reference Type always goes on the Heap - easy enough, right?

2. Value Types and Pointers always go where they were declared. This is a little more complex and needs a bit more understanding of how the Stack works to figure out where "things" are declared.

The Stack, as we mentioned earlier, is responsible for keeping track of where each thread is during the execution of our code (or what's been called). You can think of it as a thread "state" and each thread has its own stack. When our code makes a call to execute a method the thread starts executing the instructions that have been JIT compiled and live on the method table, it also puts the method's parameters on the thread stack. Then, as we go through the code and run into variables within the method they are placed on top of the stack. This will be easiest to understand by example...

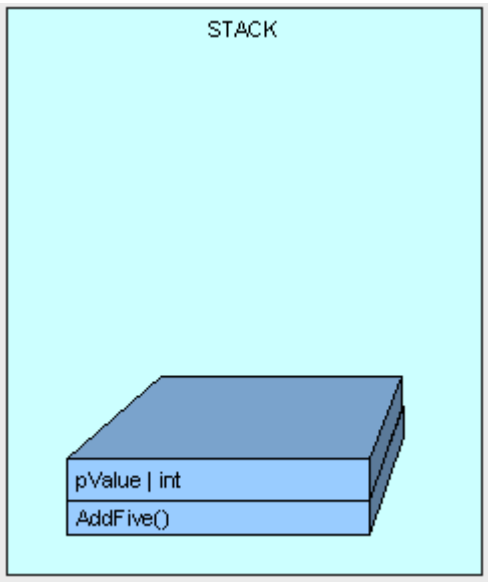
Take the following method.

```
public int AddFive(int pValue)
{
    int result;
    result = pValue + 5;
    return result;
}
```

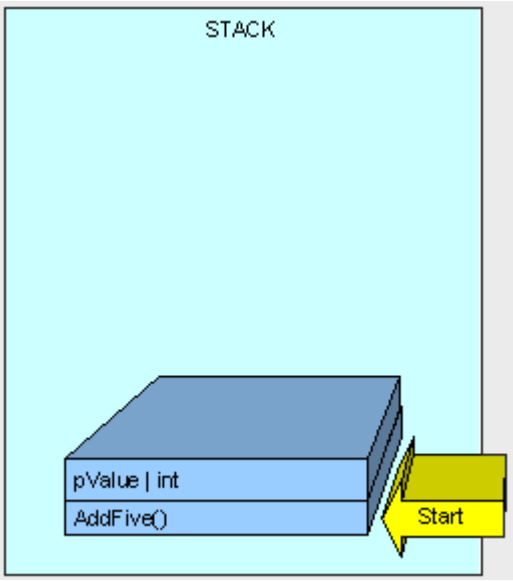
Here's what happens at the very top of the stack. Keep in mind that what we are looking at is placed on top of many other items already living in the stack:

Once we start executin ghte method, the method's parameters are placed on the stack (we'll talk more about passing parameters later).

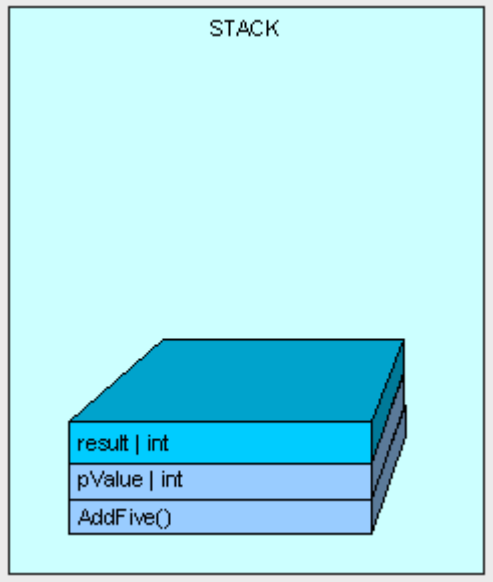
NOTE : the method does not live on the stack and is illustrated just for reference.



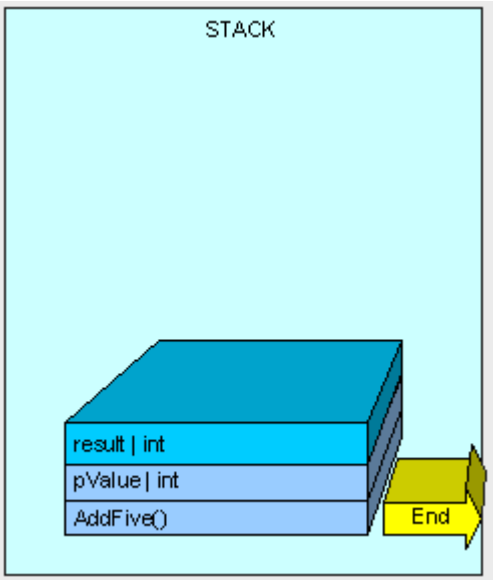
Next, control (the thread executing the method) is passed to the instructions to the AddFive() method which lives in our type's method table, a JIT compilation is performed if this is the first time we are hitting the method.



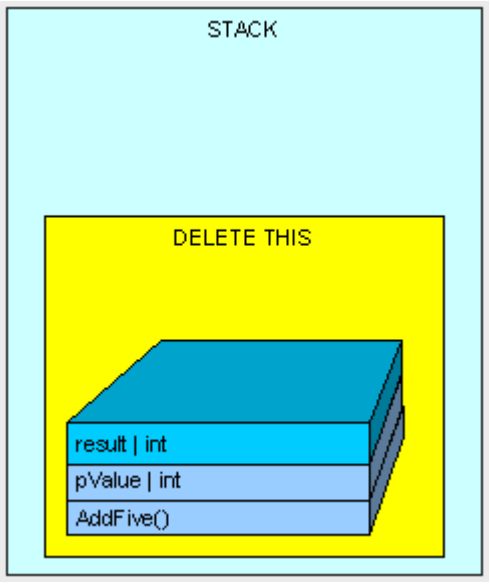
As the method executes, we need some memory for the "result" variable and it is allocated on the stack.



The method finishes execution and our result is returned.



And all memory allocated on the stack is cleaned up by moving a pointer to the available memory address where AddFive() started and we go down to the previous method on the stack (not seen here).



In this example, our "result" variable is placed on the stack. As a matter of fact, every time a Value Type is declared within the body of a method, it will be placed on the stack.

Now, Value Types are also sometimes placed on the Heap. Remember the rule, Value Types always go where they were declared? Well, if a Value Type is declared outside of a method, but inside a Reference Type it will be placed within the Reference Type on the Heap.

Here's another example.

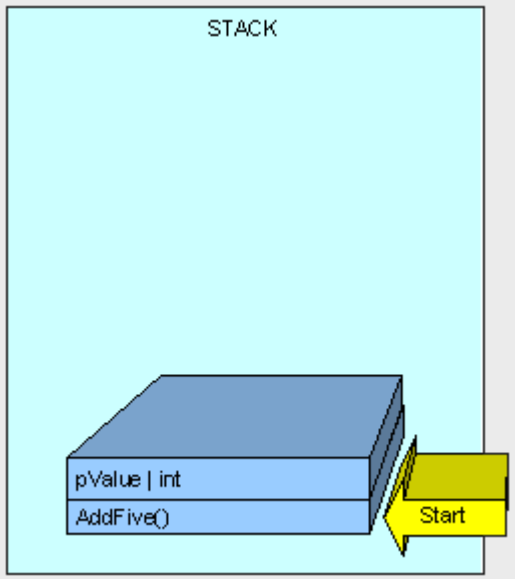
If we have the following MyInt class (which is a Reference Type because it is a class):

```
public class MyInt
{
  public int MyValue;
}
```

and the following method is executing:

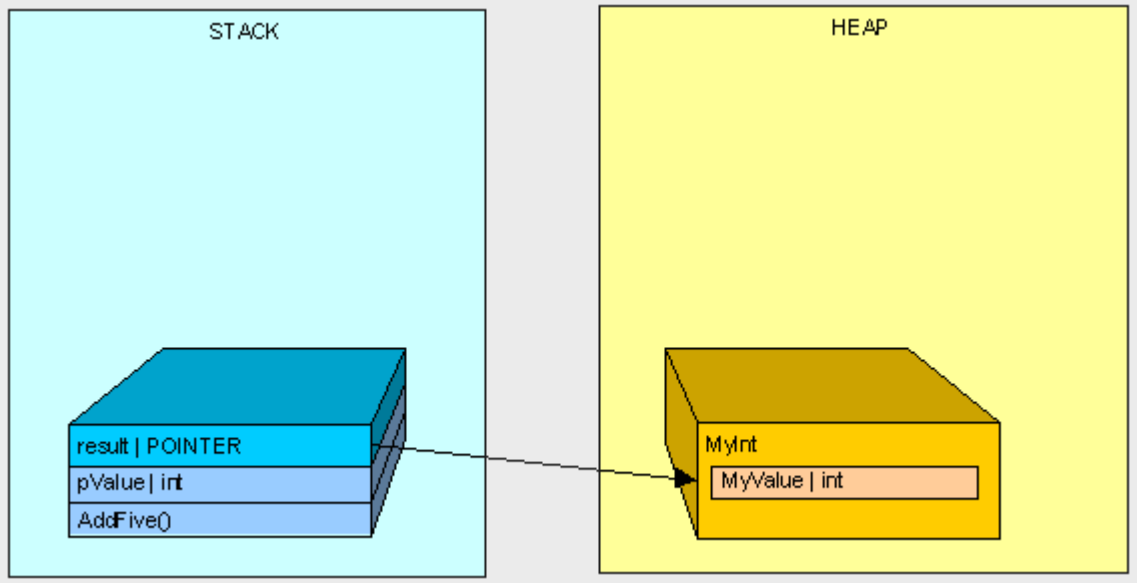
```
public MyInt AddFive(int pValue)
{
    MyInt result = new MyInt();
    result.MyValue = pValue + 5;
    return result;
}
```

Just as before, the thread starts executing the method and its parameters are placed on sthe thread's stack.

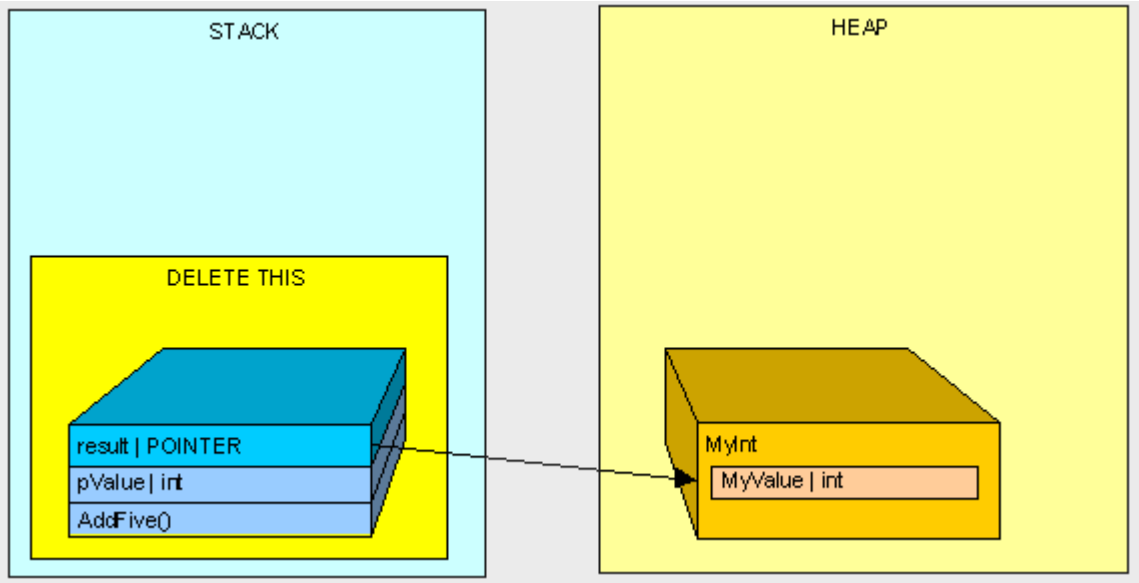


Now is when it gets interesting...

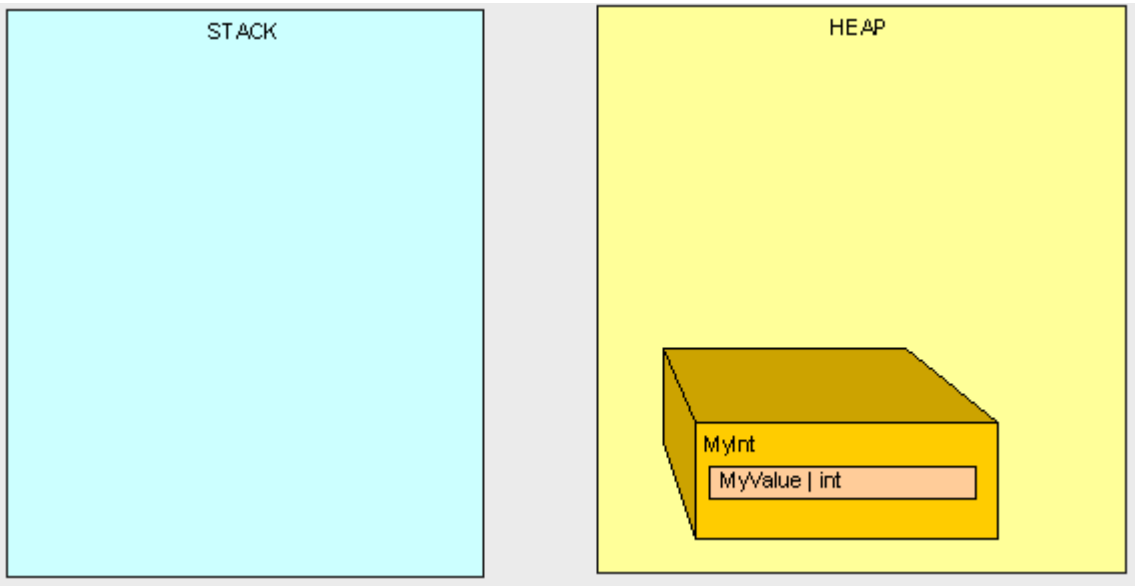
Because MyInt is a Reference Type, it is placed on the Heap and referenced by a Pointer on the Stack.



After AddFive() is finished executing (like in the first example), and we are cleaning up...



we're left with an orphaned MyInt in the heap (there is no longer anyone in the Stack standing around pointing to MyInt)!



This is where the Garbage Collection (GC) comes into play. Once our program reaches a certain memory threshold and we need more Heap space, our GC will kick off. The GC will stop all running threads (a FULL STOP), find all objects in the Heap that are not being accessed by the main program and delete them. The GC will then reorganize all the objects left in the Heap to make space and adjust all the Pointers to these objects in both the Stack and the Heap. As you can imagine, this can be quite expensive in terms of performance, so now you can see why it can be important to pay attention to what's in the Stack and Heap when trying to write high-performance code.

Ok... That great, but how does it really affect me?

Good question.

When we are using Reference Types, we're dealing with Pointers to the type, not the thing itself. When we're using Value Types, we're using the thing itself. Clear as mud, right?

Again, this is best described by example.

If we execute the following method:

```
public int ReturnValue()
{
    int x = new int();
    x = 3;
    int y = new int();
    y = x;
    y = 4;
    return x;
}
```

We'll get the value 3. Simple enough, right?

However, if we are using the MyInt class from before

```
public class MyInt
{
    public int MyValue;
}
```

and we are executing the following method:

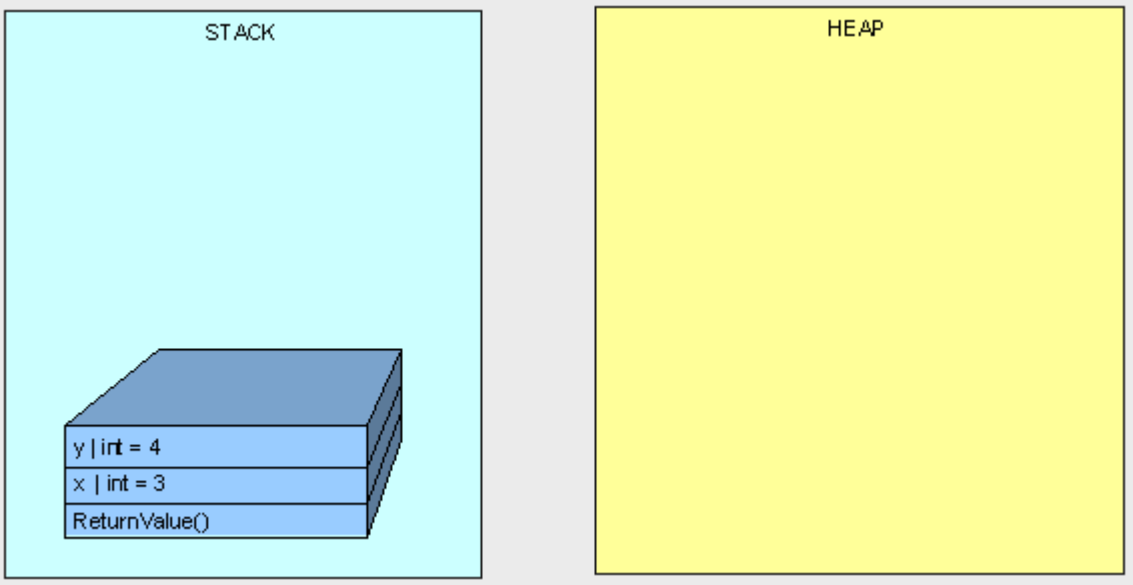
```
public int ReturnValue2()
{
    MyInt x = new MyInt();
    x.MyValue = 3;
    MyInt y = new MyInt();
    y = x;
    y.MyValue = 4;
    return x.MyValue;
}
```

What do we get?... 4!

Why?... How does x.MyValue get to be 4?... Take a look at what we're doing and see if it makes sense:

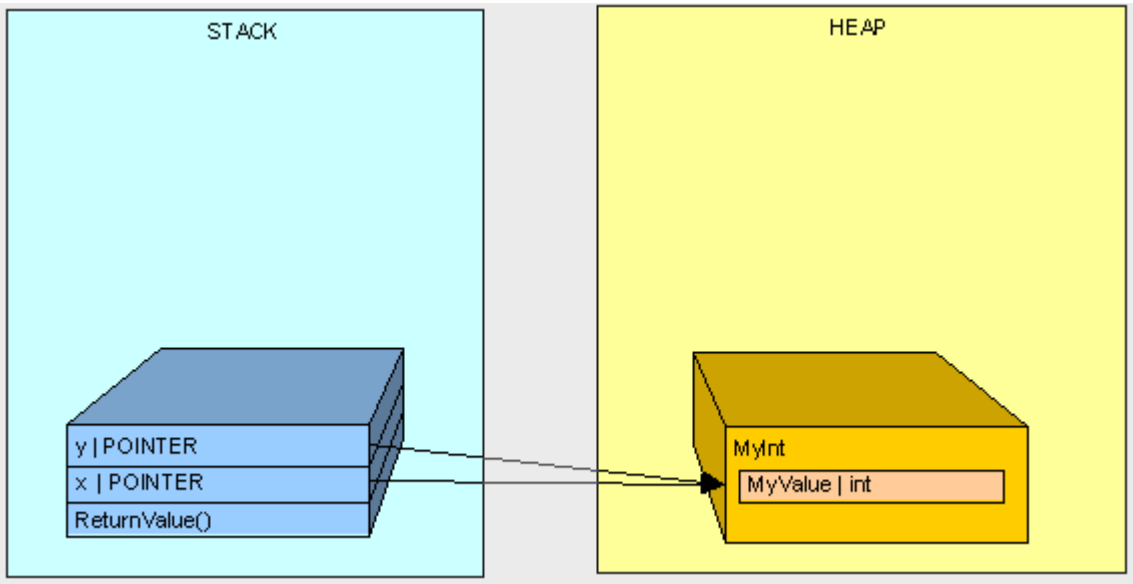
In the first example everything goes as planned:

```
public int ReturnValue()
{
    int x = 3;
    int y = x;
    y = 4;
    return x;
}
```



In the next example, we don't get "3" because both variables "x" and "y" point to the same object in the Heap.

```
public int ReturnValue2()
{
    MyInt x;
    x.MyValue = 3;
    MyInt y;
    y = x;
    y.MyValue = 4;
    return x.MyValue;
}
```



Hopefully this gives you a better understanding of a basic difference between Value Type and Reference Type variables in C# and a basic understanding of what a Pointer is and when it is used. In the next part of this series, we'll get further into memory management and specifically talk about method parameters.

(Quelle: http://www.c-sharpcorner.com/UploadFile/tkagarwal/MemoryManagementInNet11232005064832AM/MemoryManagementInNet.aspx)