Heap

So where does the Data Heap come into it? Well, the stack can store variables that are the primitive data types defined by .NET. These include the following types (see HTTP://TINYURL.COM/FRAMEWORKOVERVIEW):

- Byte
- Int32
- UInt32
- Double
- Decimal
- Structs
- SByte
- Int64
- UInt64
- Boolean
- IntPtr
- Int16
- UInt16
- Single
- Char
- UIntPtr

These are primitive data types, part of the **Common Type System (CTS)** natively understood by all NET language compilers, and collectively called **Value Types.** Any of these data types or struct definitions are usually stored on the stack.

On the other hand, instances of everything you have defined, including:

- classes
- interfaces
- delegates
- strings
- instances of “object”

...are all referred to as “reference types,” and are stored on the heap (the SOH or LOH, depending on their size).

When an instance of a reference type is created (usually involving the `new` keyword), only an **object reference** is stored on stack. The actual instance itself is created on the heap, and its address held on the stack.

Consider the following code:

```
void Method1()
{
      MyClass myObj=new MyClass();
      Console.WriteLine(myObj.Text);
}
```

Listing 1.2: Code example using a reference type.

In Listing 1.2, a new instance of the class `MyClass` is created within the Method1 call.

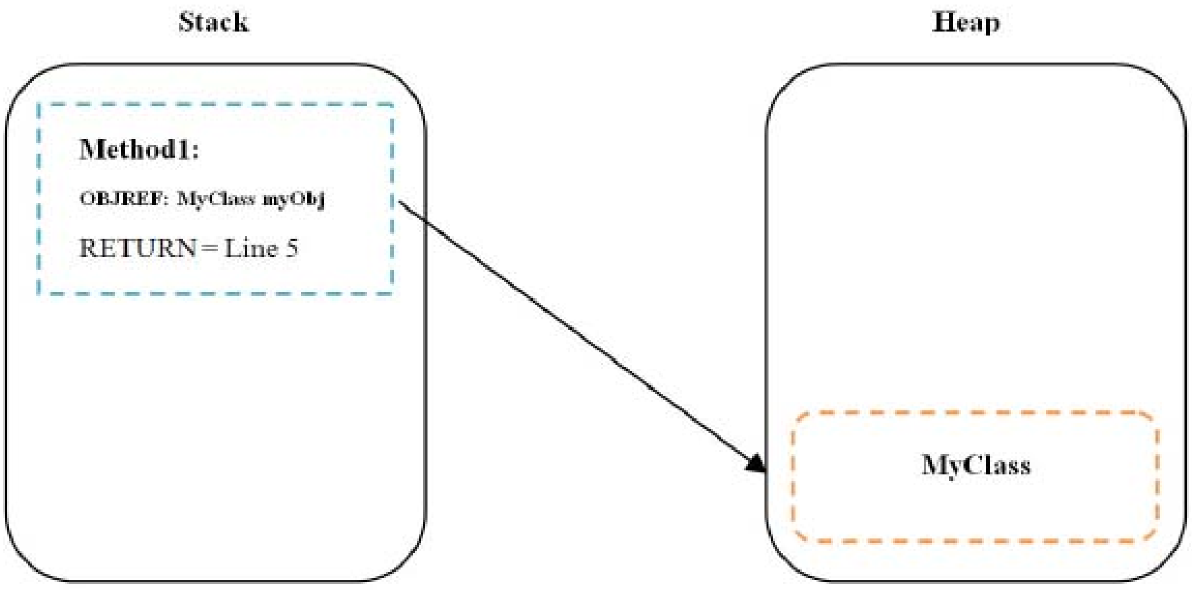

Figure 1.2: Object reference from stack to heap.

As we can see in Figure 1.2, to achieve this .NET has to create the object on the memory heap, determine its address on the heap (or object reference), and place that object reference within the stack frame for Method1. As long as Method1 is executing, the object allocated on the heap will have a reference held on the stack. When Method1 completes, the stack frame is removed (along with the object reference), leaving the object without a reference.

We will see later how this affects memory management and the garbage collector.

More on value and reference types

The way in which variable assignment works differs between reference and value types. Consider the code in Listing 1.3.

```
1  void ValueTest()
2  {
3    int v1=12;
4    int v2=22;
5    v2=v1;
6    Console.Writeline(v2);
7  }
```

Listing 1.3: Assignment of value types.

If a breakpoint was placed at Line 6, then the stack/heap would look as shown in Figure 1.3.

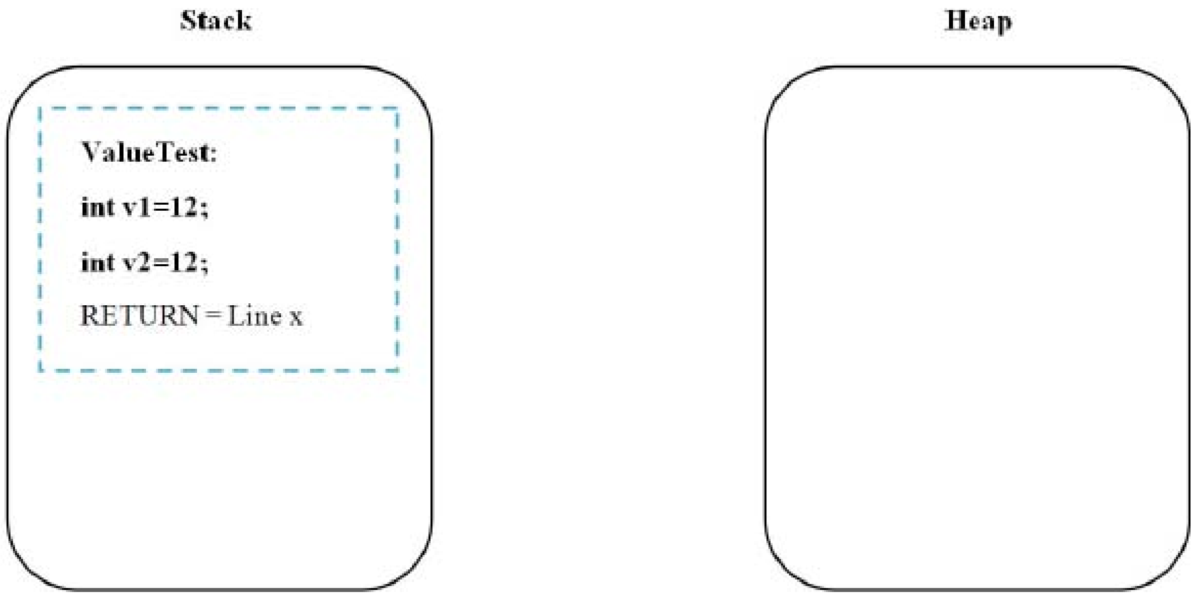

Figure 1.3: Stack example of value type assignment.

There are two separate integers on the stack, both with the same value.

Notice there are two stack variables, `v1` and `v2`, and all the assignment has done is assign the same value to both variables.

Let's look at a similar scenario, this time using a class I have defined, `MyClass`, which is (obviously) a reference type:

```
1  void RefTest()
2  {
3      MyClass v1=new MyClass(12);
4      MyClass v2=new MyClass(22);
5      v2=v1;
6      Console.Writeline(v2.Value);
7  }
```

Listing 1.4: Assignment with reference types.

Placing a break point on Line 5 in Listing 1.4 would see two `MyClass` instances allocated onto the heap.

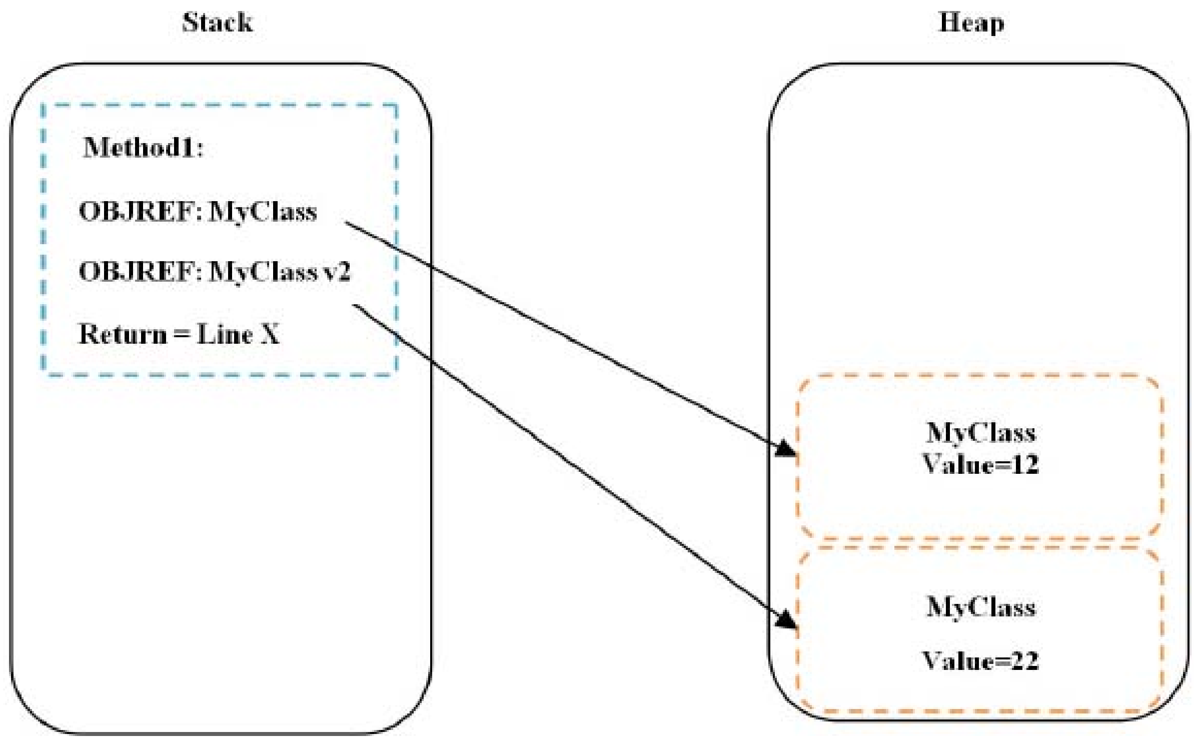

Figure 1.4: Variable assignment with reference types.

On the other hand, by letting execution continue, and allowing `v1` to be assigned to `v2` , the execution at Line 6 in Listing 1.4 would show a very different heap.

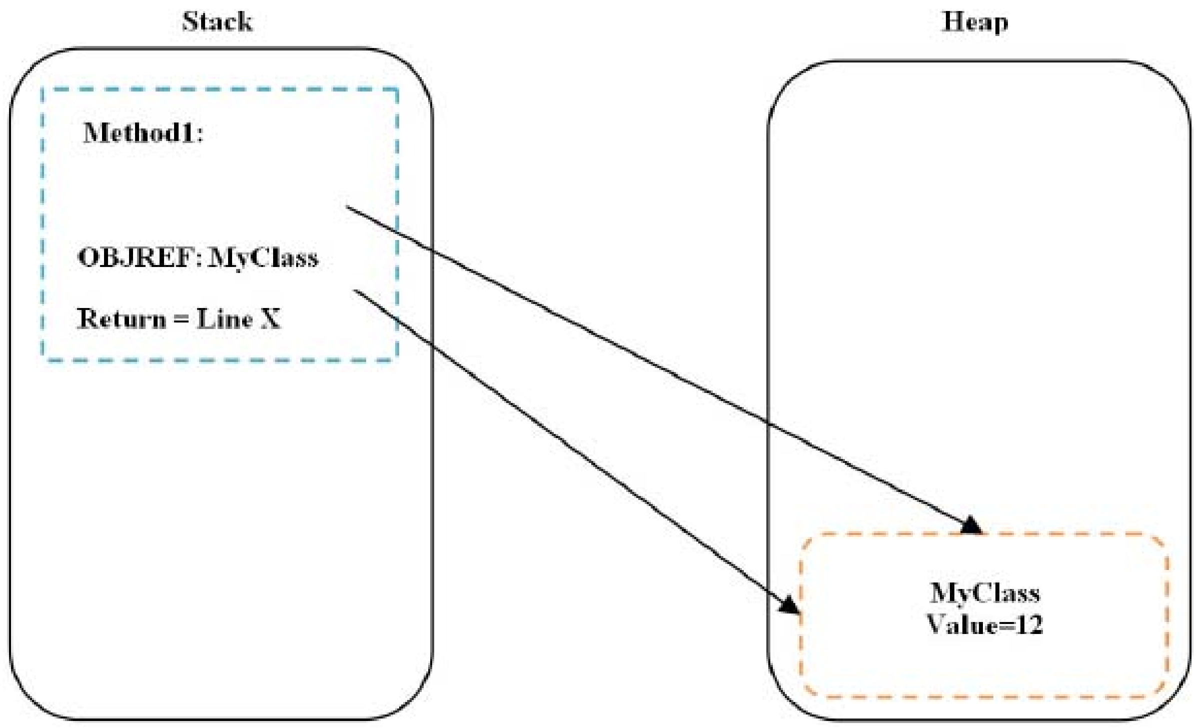

Figure 1.5: Variable assignment with reference types (2).

Notice how, in Figure 1.5, both object pointers are referencing only the one class instance after the assignment. Variable assignment with reference types makes the object pointers on the stack the same, and so they both point to the same object on the heap.

Passing parameters

When you pass a value type as a parameter, all you actually pass to the calling method is a copy of the variable. Any changes that are made to the passed variable within the method call are isolated to the method.

Hav ing copies of v alue ty pes on the stack isn't usually a problem, unless the v alue ty pe is large, as can be the case with structs. While structs are v alue ty pes and, as such, are also allocated onto the stack, they are also, by their nature, programmer-def inable structures, and so they can get pretty large. When this is the case, and particularly if they are passed as parameters between method calls, it can be a problem f or y our application. Hav ing multiple copies of the same struct created on the stack creates extra work in copy ing the struct each time. This might not seem like a big deal but, when magnif ied within a high iteration loop, it can cause a perf ormance issue.

One way around this problem is to pass specif ic v alue ty pes by ref erence. This is something y ou would do any way if y ou wanted to allow direct changes to the v alue of a passed v ariable inside a method call.

Consider the f ollowing code:

```
void Method1()
{
   int v1=22;
   Method2(v1);
   Console.WriteLine("Method1 = " + v1.ToString());
}
 void Method2(int v2)
{
   v2=12;
   Console.WriteLine("Method2 = " + v2.ToString());
}
```

Listing 1.5: Passing parameters by value.

Once Method1 completes we would see the f ollowing output:

```
Method 2 = 12
Method 1 = 22
```

Listing 1.6: Output from a parameter passed by value.

Because parameter `v1` was passed to Method2 by v alue, any changes to it within the call don't af f ect the original v ariable passed. That's why the f irst output line shows `v2` as being 12. The second output line demonstrates that the original v ariable remains unchanged.

Alternativ ely , by adding a `ref` instruction to both the method and the calling line, v ariables can be passed by ref erence, as in Listing 1.7.

```
void Method1()
{
   int v1=22;
   Method2(ref v1);
   Console.WriteLine("Method1 = " + v1.ToString());
}
void Method2(ref int v2)
{
   v2=12;
   Console.WriteLine("Method2 = " + v2.ToString());
}
```

Listing 1.7: Passing parameters by reference.

Once Method1 completes, we would see the output shown in Listing 1.8.

```
Method 2 = 12
Method 1 = 12
```

Listing 1.8: Output from a parameter passed by reference.

Notice both outputs display "12," demonstrating that the original passed value was altered.

Boxing and unboxing

Let's now talk about that topic you always get asked about in interviews, boxing and unboxing. It's actually really easy to grasp, and simply refers to the extra work required when your code causes a value type (e.g. `int` , `char` , etc.) to be allocated on the heap rather than the stack. As we saw earlier, allocating onto the heap requires more work, and is therefore less efficient.

The classic code example of boxing and unboxing looks something like this:

```
1 // Integer is created on the Stack
2 int stackVariable=12;
3 // Integer is created on the Heap = Boxing
4 object boxedObject= stackVariable
5 // Unboxing
6 int unBoxed=(int)boxedObject;
```

Listing 1.9: Classic boxing and unboxing example.

In Listing 1.9 an integer is declared and allocated on the stack because it's a value type (Line 2). It's then assigned to a new object variable (boxed) which is a reference type (Line 4), and so a new object is allocated on the heap for the integer. Finally, the integer is unboxed from the heap and assigned to an integer stack variable (Line 6).

The bit that confuses everyone is why you would ever do this – it makes no sense.

The answer to that is that you can cause boxing of value types to occur very easily without ever being aware of it.

```
1  int i=12;
2  ArrayList lst=new ArrayList();
3  // ArrayList Add method has the following signature
4  // int Add(object value)
5  lst.Add(i); // Boxing occurs automatically
6  int p=(int)lst[0]; // Unboxing occurs
```

Listing 1.10: Boxing a value type.

Listing 1.10 demonstrates how boxing and unboxing can sneakily occur, and I bet you've written similar code at some point. Adding an integer (value type) to the ArrayList will cause a boxing operation to occur because, to allow the array list to be used `for` all types (value and reference), the `Add` method takes an object as a parameter. So, in order to add the integer to the `ArrayList` , a new object has to be allocated onto the heap.

When the integer is accessed in Line 6, a new stack variable " `p` " is created, and its value set to the same value as the first integer in the `ArrayList.`

In short, a lot more work is going on than is necessary, and if you were doing this in a loop with thousands of integers, then performance would be significantly slower.