# Stack, heap, value types, reference types, boxing, and unboxing

## Introduction

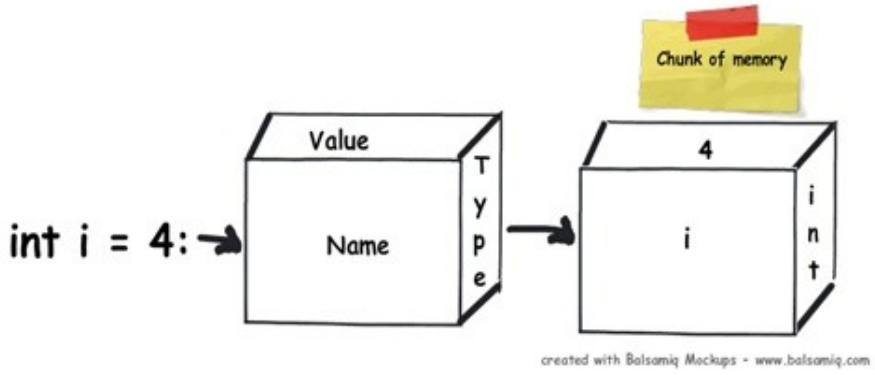
This article will explain six important concepts: stack, heap, value types, reference types, boxing, and unboxing. This article starts explaining what happens internally when you declare a variable and then it moves ahead to explain two important concepts: stack and heap.

The article then talks about reference types and value types and clarifies some of the important fundamentals around them.

## What goes inside when you declare a variable?

When you declare a variable in a .NET application, it allocates some chunk of memory in the RAM. This memory has three things: the name of the variable, the data type of the variable, and the value of the variable.

That was a simple explanation of what happens in the memory, but depending on the data type, your variable is allocated that type of memory. There are two types of memory allocation: **stack memory** and **heap memory**. In the coming sections, we will try to understand these two types of memory in more detail.




## Stack and heap

In order to understand stack and heap, let's understand what actually happens in the below code internally.

```
public void Method1()
{

    // Line 1
    int i=4;


    // Line 2
    int y=2;


    //Line 3
    class1 cls1 = new class1();
}
```

It's a three line code, let's understand line by line how things execute internally.

- Line 1: When this line is executed, the compiler allocates a small amount of memory in the stack. The stack is responsible for keeping track of the running memory needed in your application.

- Line 2: Now the execution moves to the next step. As the name says stack, it stacks this memory allocation on top of the first memory allocation. You can think about stack as a series of compartments or boxes put on top of each other.

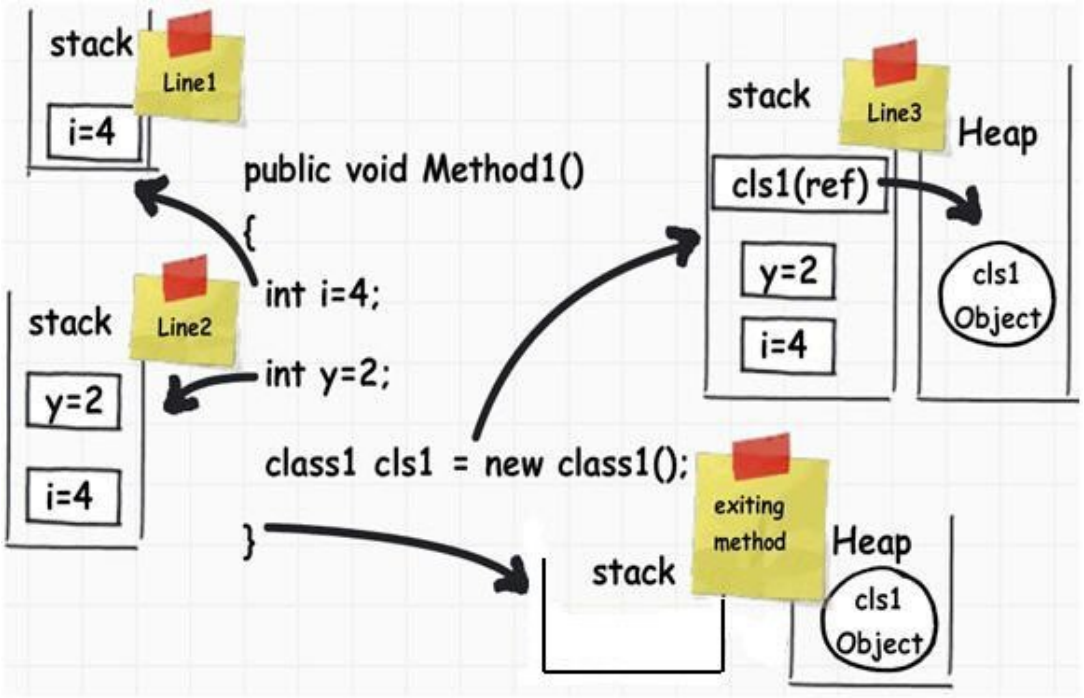
  Memory allocation and de-allocation is done using LIFO (Last In First Out) logic. **In other words memory is allocated and de-allocated at only one end of the memory, i.e., top of the stack.**

- Line 3: In line 3, we have created an object. When this line is executed it creates a pointer on the stack and the actual object is stored in a different type of memory location called 'Heap'. 'Heap' does not track running memory, it's just a pile of objects which can be reached at any moment of time. **Heap is used for dynamic memory allocation.**

  One more important point to note here is reference pointers are allocated on stack. The statement, `Class1 cls1;` does not allocate memory for an instance of `Class1`, it only allocates a stack variable `cls1` (and sets it to `null`). The time it hits the `new` keyword, it allocates on "heap".

  **Exiting the method (the fun):** Now finally the execution control starts exiting the method. When it passes the end control, it clears all the memory variables which are assigned on stack. In other words all variables which are related to `int` data type are de-allocated in 'LIFO' fashion from the stack.

  The **big** catch – It did not de-allocate the heap memory. This memory will be later de-allocated by the garbage collector.




Now many of our developer friends must be wondering why two types of memory, can't we just allocate everything on just one memory type and we are done?

If you look closely, **primitive data types are not complex, they hold single values like '`int i = 0`'. Object data types are complex, they reference other objects or other primitive data types**. In other words, they hold reference to other multiple values and each one of them must be stored in memory. Object types need dynamic memory while primitive ones needs static type memory. If the requirement is of dynamic memory, it's allocated on the heap or else it goes on a stack.

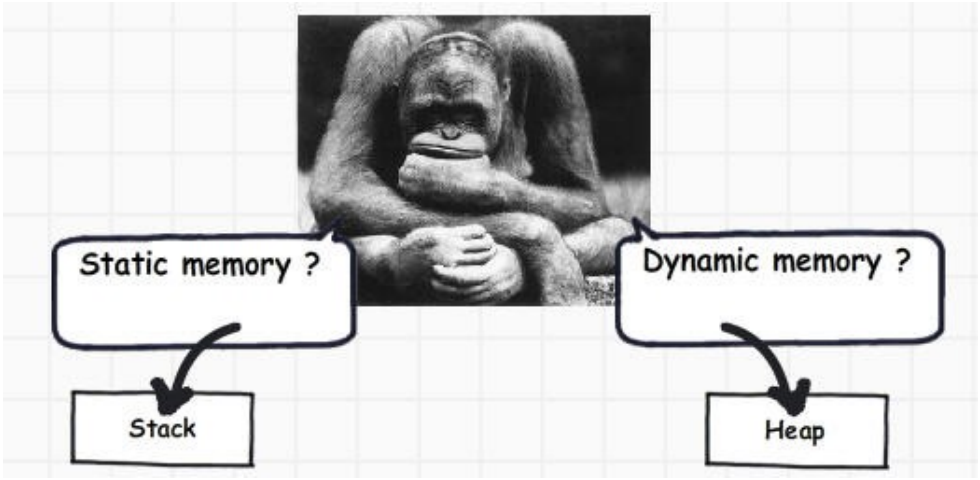


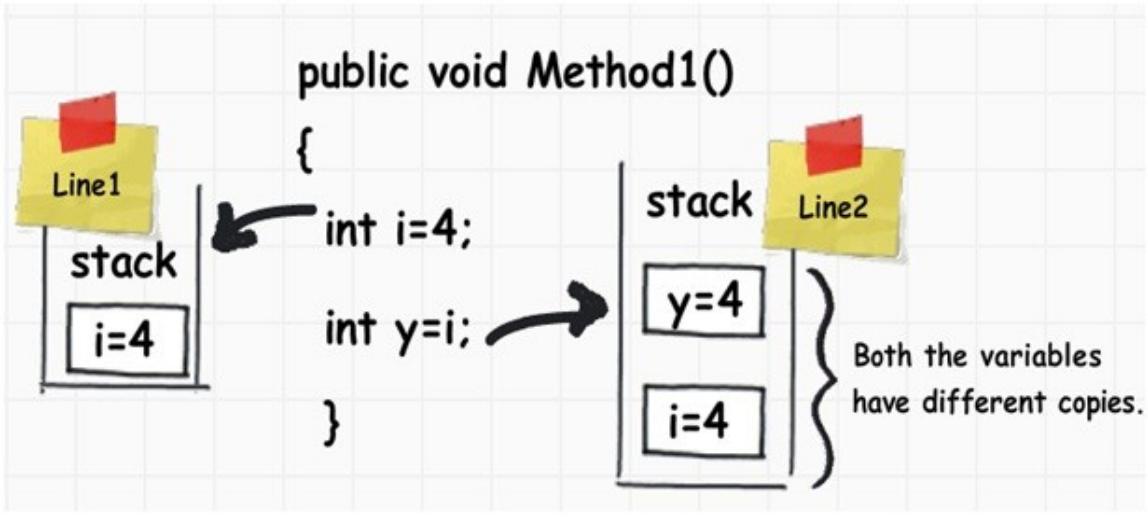
Image taken from http://michaelbungartz.wordpress.com/

# Value types and reference types

Now that we have understood the concept of Stack and Heap, it's time to understand the concept of **value types** and **reference types**. **Value types** are types which hold both data and memory on the same location. A **reference type** has a pointer which points to the memory location.

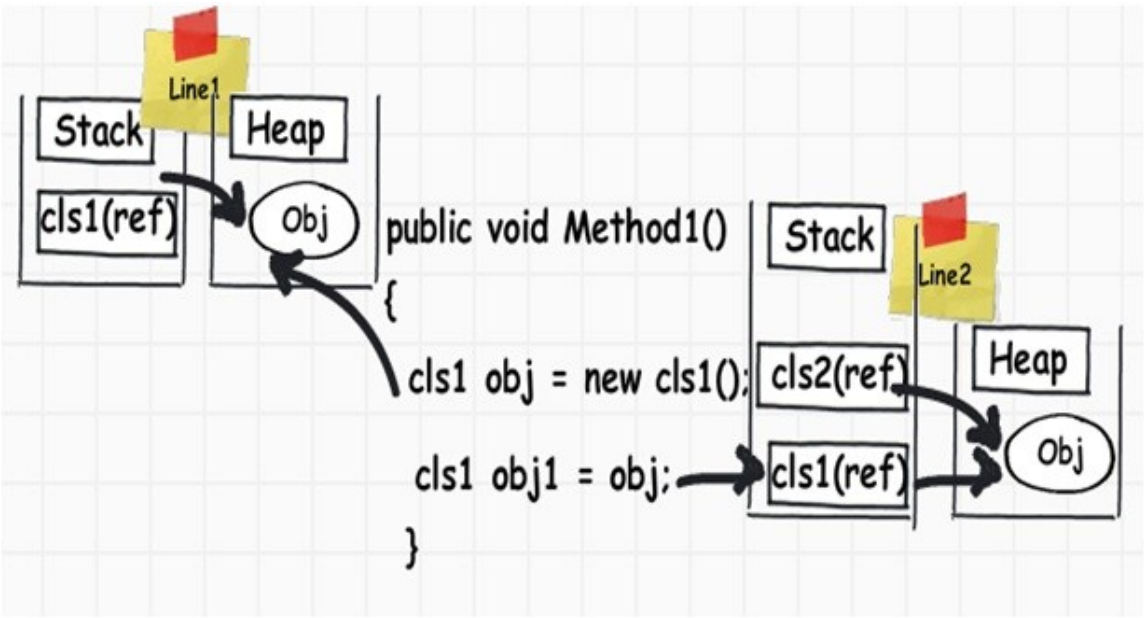
Below is a simple **integer data type** with name `i` whose value is assigned to another integer data type with name `j`. Both these memory values are allocated on the stack.

When we assign the `int` value to the other `int` value, it creates a completely different copy. In other words, if you change either of them, the other does not change. These kinds of data types are called as 'Value types'.



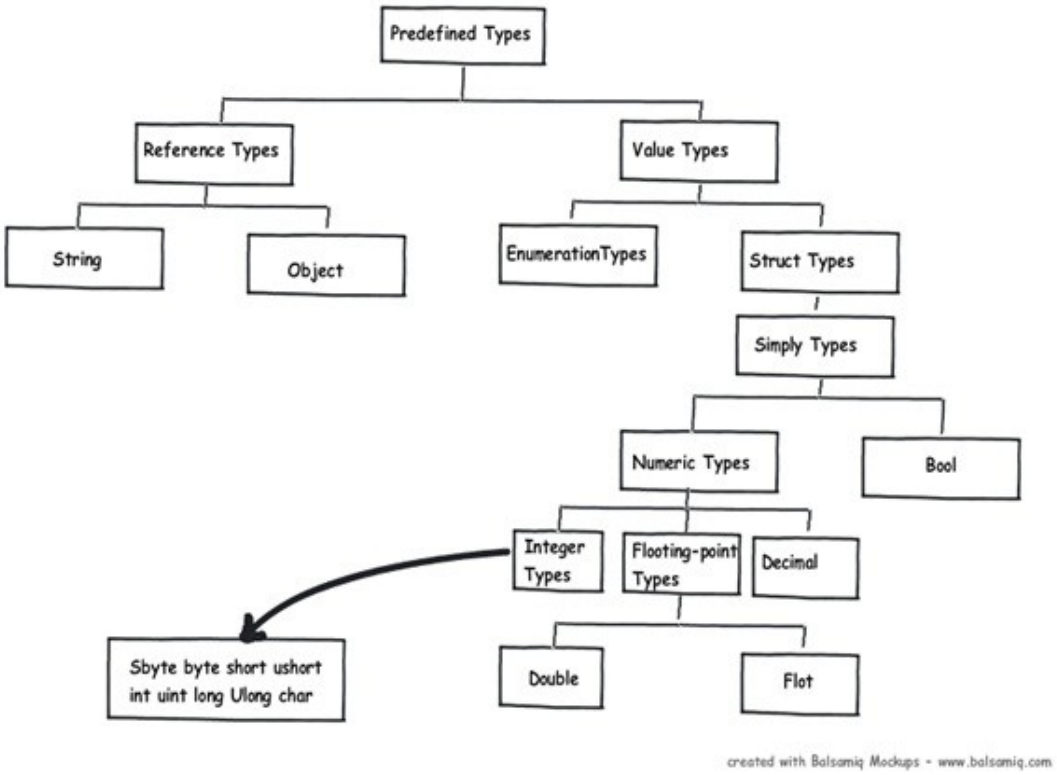

When we create an object and when we assign an object to another object, they both point to the same memory location as shown in the below code snippet. So when we assign `obj` to `obj1`, they both point to the same memory location.

In other words if we change one of them, the other object is also affected; this is termed as '**Reference types**'.




# So which data types are ref types and which are value types?

In .NET depending on the data type, the variable is either assigned on the stack or on the heap**.** '**String**' and '**Objects'** **are reference types**, and **any other .NET primitive data types are assigned on the stack**. The figure below explains the same in a more detail manner.



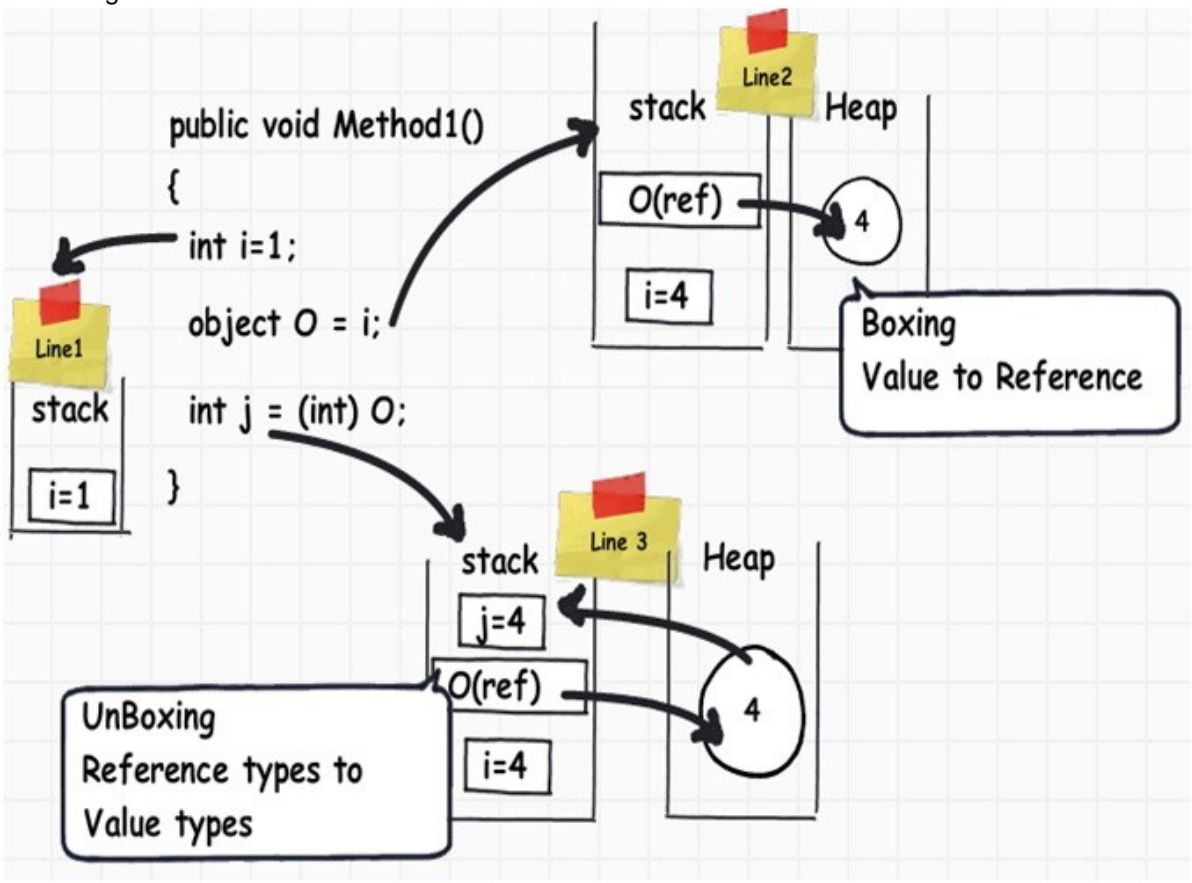
# Boxing and unboxing

Wow, you have given so much knowledge, so what's the use of it in actual programming? One of the biggest implications is to understand the performance hit which is incurred due to data moving from stack to heap and vice versa.

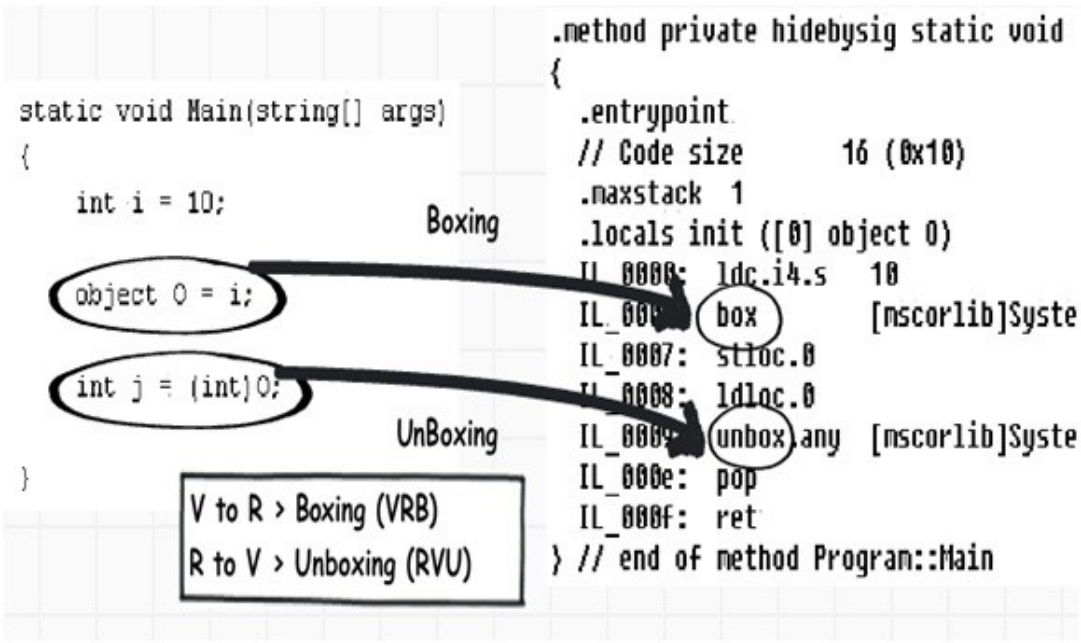
Consider the below code snippet. When we move a value type to reference type, data is moved from the stack to the heap. When we move a reference type to a value type, the data is moved from the heap to the stack.

This movement of data from the heap to stack and vice-versa creates a performance hit.

When the data moves from value types to reference types, it is termed 'Boxing' and the reverse is termed 'UnBoxing'.




If you compile the above code and see the same in ILDASM (http://msdn.microsoft.com/en-us/library/f7dy01k1%28v=vs.110%29.aspx), you can see in the IL code how 'boxing' and 'unboxing' looks. The figure below demonstrates the same.

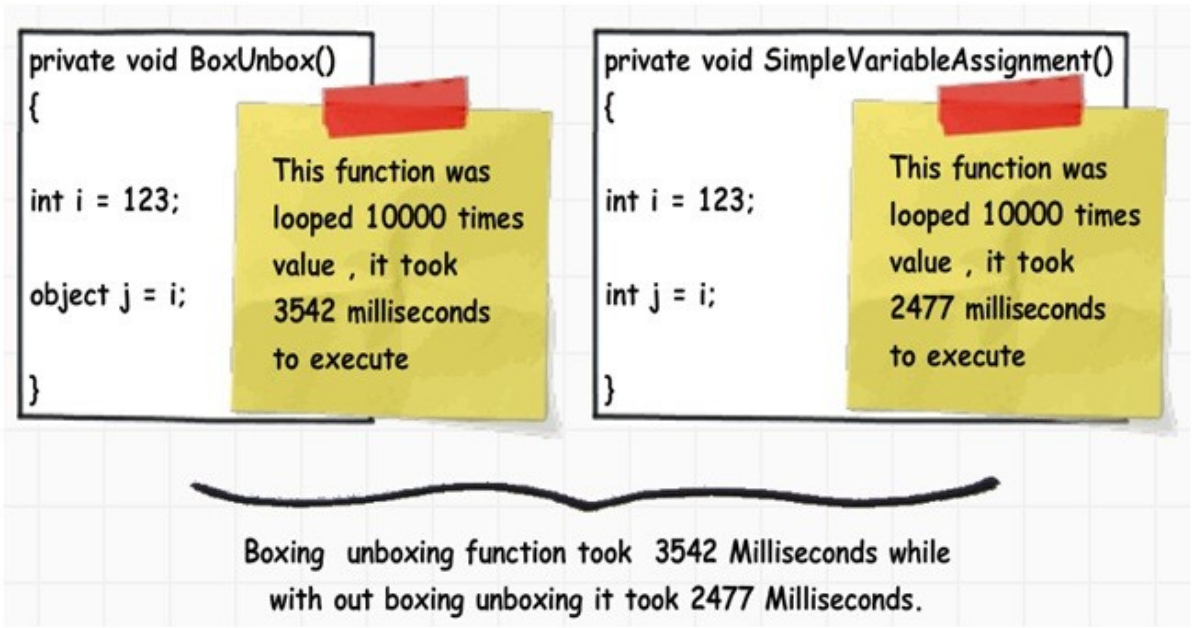



# Performance implication of boxing and unboxing

In order to see how the performance is impacted, we ran the below two functions 10,000 times. One function has boxing and the other function is simple. We used a stop watch object to monitor the time taken.

The boxing function was executed in 3542 ms while without boxing, the code was executed in 2477 ms. In other words try to avoid boxing and unboxing. In a project where you need boxing and unboxing, use it when it's absolutely necessary.

With this article, sample code is attached which demonstrates this performance implication.




http://www.codeproject.com/Articles/76153/Six-important-NET-concepts-Stack-heap-value-types