Stack and Heap Memory in .NET

Back to: C#.NET Tutorials For Beginners and Professionals

Stack and Heap Memory in .NET with Examples

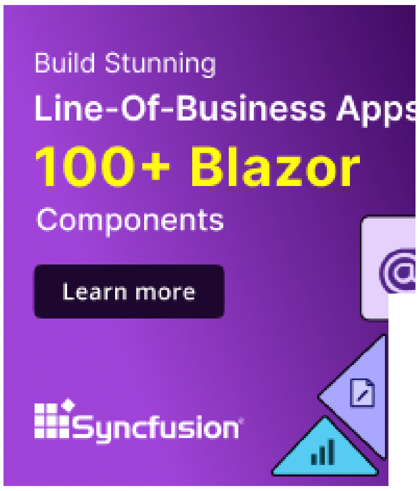
In this article, I will discuss **Stack and Heap Memory in .NET Applications** with Examples. Please read our previous article discussing the **Checked and Unchecked Keywords in C#** with Examples. As part of this article, first, we will discuss what happens internally when we declare a variable of value types and reference types. Then, we will move forward and learn two important concepts, i.e., stack and heap memory, and talk about value types and reference types.

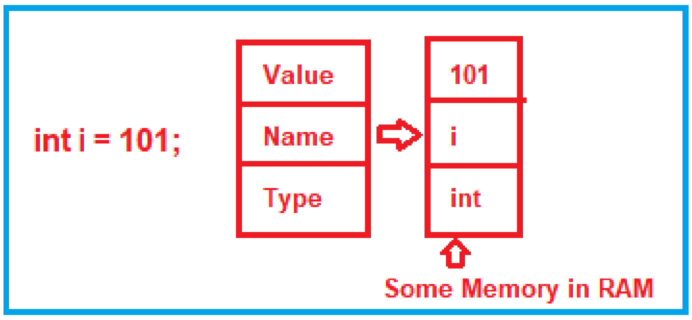
What Happens Internally When We Declare a Variable in a .NET Application?

When we declare a variable in a .NET application, it allocates some memory in the RAM. The memory that it allocates in RAM has three things are as follows:

1. **Name of the Variable,**
2. **The Data Type of the Variable, and**
3. **Value of the Variable.**

For a better understanding, please have a look at the following image. Here, we declare a variable of type int and assign a value 101.

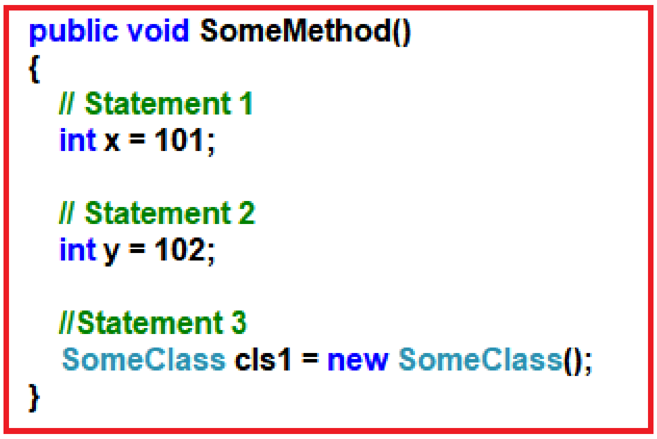




The above image shows a high-level overview of what is happening in the memory. But depending on the data type (i.e., depending on the value type and reference type), the memory may be allocated either in the stack or in the heap memory.

Understanding Stack and Heap Memory in C#:

There are two types of memory allocation for the variables we created in the .NET Application, i.e., Stack Memory and Heap Memory. Let us understand the Stack and Heap Memory with an Example. To understand Stack and Heap Memory, please have a look at the following code, and let's understand what actually happens in the below code internally.

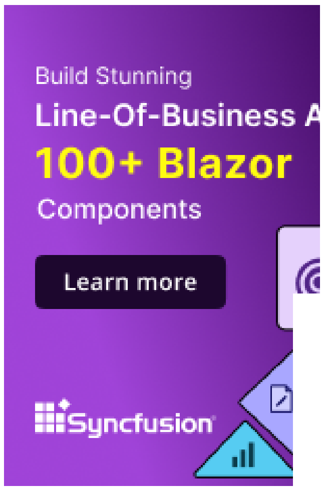


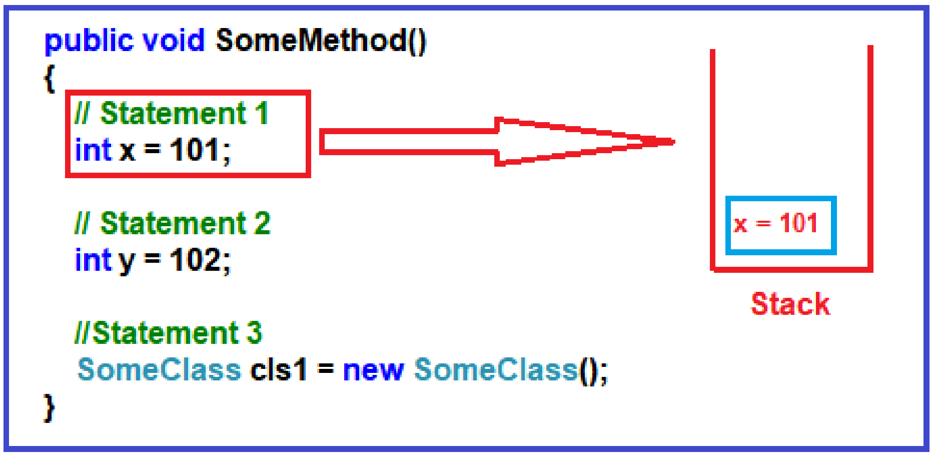
```
public void SomeMethod()
{
    // Statement 1
    int x = 101;

    // Statement 2
    int y = 102;

    //Statement 3
    SomeClass cls1 = new SomeClass();
}
```

As you can see in the above image, the SomeMethod has three statements. Let's understand statement by statement how things are executed internally.

Statement 1:

When the first statement is executed, the compiler allocates some memory in the stack. The stack memory is responsible for keeping track of the running memory needed in your application. For a better understanding, please have a look at the following image.

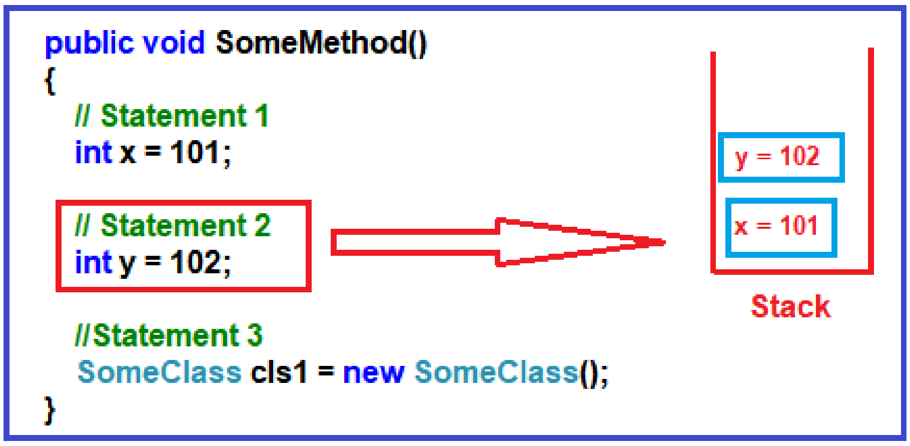




```
public void SomeMethod()
{
    // Statement 1
    int x = 101;

    // Statement 2
    int y = 102;

    //Statement 3
    SomeClass cls1 = new SomeClass();
}
```

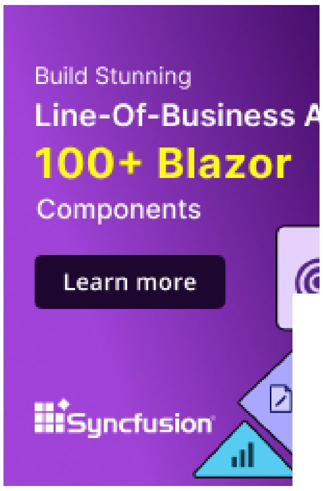
Statement 2:

When the second statement is executed, it stacks this memory allocation (memory allocation for variable y) on top of the first memory allocation (memory allocation for variable x). You can think about the stack as a series of plates or dishes put on top of each other. Please have a look at the following diagram for a better understanding.



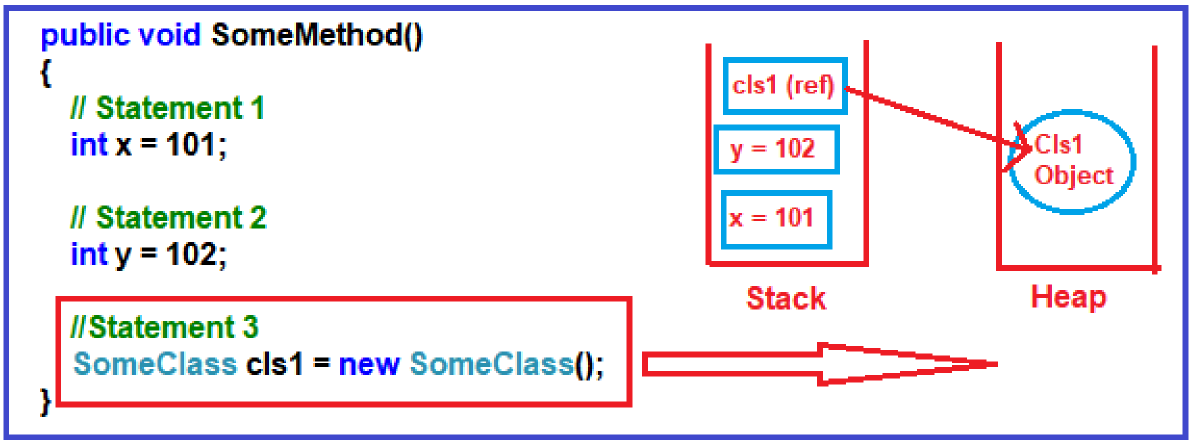
```
public void SomeMethod()
{
    // Statement 1
    int x = 101;

    // Statement 2
    int y = 102;

    //Statement 3
    SomeClass cls1 = new SomeClass();
}
```

The Stack Memory allocation and de-allocation in .NET uses the Last In, First Out Principle. In other words, we can say that the memory allocation and de-allocation are done only at one end of the memory, i.e., the top of the stack.



Statement3:

In the 3rd statement, we have created an object of SomeClass. When the 3rd statement is executed, it internally creates a pointer on the stack memory, and the actual object is stored in a different memory location called Heap memory. The Heap Memory location does not track running memory. Heap is used for dynamic memory allocation. For a better understanding, please have a look at the below image.

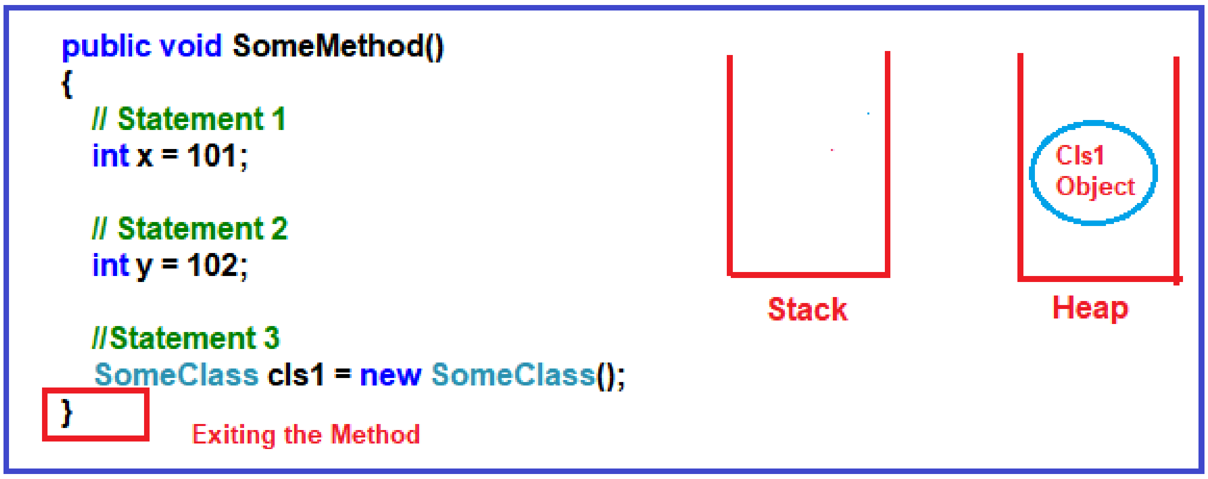


```
public void SomeMethod()
{
   // Statement 1
   int x = 101;

   // Statement 2
   int y = 102;

   //Statement 3
   SomeClass cls1 = new SomeClass();
}
```

Note: The reference pointers are allocated on the stack. The statement, **SomeClass cls1,** does not allocate any memory for an instance of **SomeClass.** It only allocates a variable with the name cls1 in the stack and sets its value to null. When it hits the new keyword, it allocates memory in the heap.

What Happens When the Method Completes Its Execution?

When the three statements are executed, the control will exit from the method. When it passes the end control, i.e., the end curly brace "}," it will clear all the memory variables created on the stack. It will de-allocate the memory from the stack in a 'LIFO' fashion. For a better understanding, please have a look at the below image.



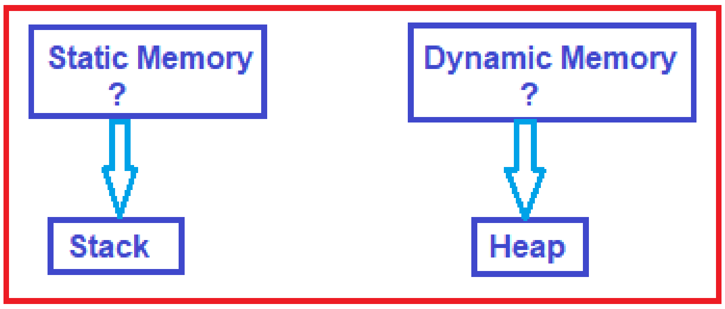
```
public void SomeMethod()
{
    // Statement 1
    int x = 101;

    // Statement 2
    int y = 102;

    //Statement 3
    SomeClass cls1 = new SomeClass();
}
```

It will not de-allocate the Heap memory. Later, the heap memory will be de-allocated by the garbage collector. Now, you may have one question in your mind: why two types of memory? Can't we allocate everything to just one memory type?

Why do we have two types of memory?

In C#, primitive data types, such as int, double, bool, etc., hold a single value. On the other hand, the reference data types or object data types are complex, i.e., an object data type or reference data type can have reference to other objects and other primitive data types.

So, the reference data type holds references to other multiple values, and each one of them must be stored in memory. Object types need dynamic memory, while primitive data types need static memory. Please have a look at the following image for a better understanding.

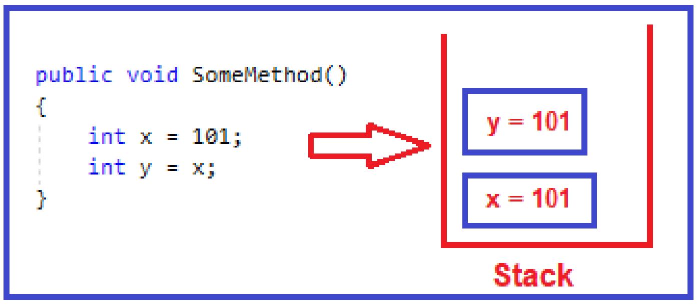


Value Types and Reference Types in C#.NET

As we understood the concept of Stack and Heap, Now, let us move forward and understand the concept value types and reference types in C#. The Value types are the types that hold both data and memory in the same location. On the other hand, a reference type is a type that has a pointer that points to the actual memory location.

Understanding Value Type in C#:

Let us understand value type with an example. Please have a look at the following image. As you can see in the image, first, we create an integer variable with the name x, and then we assign this x integer value to another integer variable named y. In this case, the memory allocation for these two variables will be done inside the stack memory.

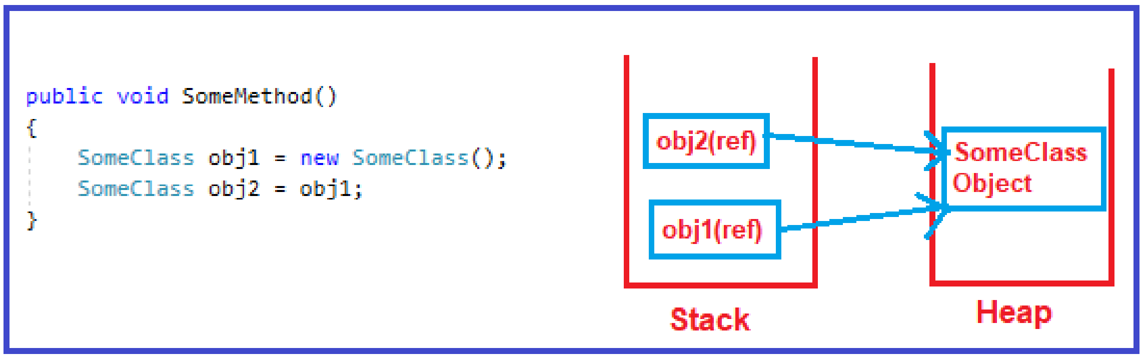


```
public void SomeMethod()
{
    int x = 101;
    int y = x;
}
```

In .NET, when we assign one integer variable value to another integer variable, it creates a completely different copy in the stack memory. That’s what you can see in the above image. So, if you change one variable value, the other variable will not be affected. In .NET, these data types are called Value types. So, bool, byte, char, decimal, double, enum, float, long, sbyte, int, short, ulong, struct, uint, ushort are examples of value types.

DOT NET TUTORIALS

Let us understand reference type with an example. Please have a look at the following image. Here, first, we create an object, i.e., obj1) and then assign this object to another object, i.e., obj2. In this case, both reference variables (obj1 and obj2) will point to the same memory location.



```
public void SomeMethod()
{
    SomeClass obj1 = new SomeClass();
    SomeClass obj2 = obj1;
}
```

In this case, when you change one of them, the other object is also affected. These kinds of data types are termed Reference types in .NET. So, class, interface, object, string, and delegate are examples of Reference Types.

How is the Heap Memory Freed Up?

The memory allocation on the stack is deallocated when the control moves out from the method, i.e., once the method completes its execution. On the other hand, the memory allocation, which is done on the heap, needs to be de-allocated by the garbage collector.

When an object stored on the heap is no longer used, that means the object does not have any reference pointing. Then, the object is eligible for garbage collection. The garbage collector will de-allocate this object from the heap at some point.

Stack Memory Key Points:

- **Allocation:** Stack memory is allocated for static memory allocation and local variables. It's managed by the CPU, making it faster and more efficient.
- **Usage:** When a method is called, a block of memory (a stack frame) is allocated on the stack for its local variables and parameters. When the method call returns, the block becomes unused and can be used for the next method call.

- **Lifespan:** Variables stored in the stack are only available during the lifetime of the method call.
- **Type of Data:** It stores value types in C#. These include primitive data types (like int, double, char), structs, and references to objects (the references themselves, not the objects).

Heap Memory Key Points:

- **Allocation:** Heap memory is used for dynamic memory allocation, which includes objects and complex data structures that require more flexibility and are managed by the garbage collector in .NET.
- **Usage:** Objects are allocated on the heap, and memory is managed at runtime. New objects are created using the new keyword, and the garbage collector automatically frees up heap memory when objects are no longer in use.
- **Lifespan:** Objects on the heap live from when they are created until they are no longer used and are garbage collected.
- **Type of Data:** It stores reference types like objects, arrays, and class instances.

Key Differences Between Stack and Head Memory in .NET:

- **Management:** Stack memory is automatically managed by the system, whereas heap memory is dynamically allocated and deallocated by the garbage collector.
- **Speed:** Stack memory is generally faster than heap memory because of its organization and the way it's managed.
- **Size:** The stack has size limits based on the thread, but the heap can dynamically grow as needed (limited by the system's available memory).
- **Access:** Stack memory access is more straightforward and faster, while heap memory requires more complex management.
- **Storage:** Value types are stored in stack memory, while reference types are stored in heap memory.

In the next article, I will discuss **Boxing and Unboxing in C#.NET** with Examples. In this article, I try to explain **Stack and Heap Memory in C#** with Examples. I hope you enjoy this Stack and Heap Memory in C# with Examples article. I would like to have your feedback. Please post your feedback, questions, or comments about this article.

Dot Net Tutorials

About the Author: Pranaya Rout

Pranaya Rout has published more than 3,000 articles in his 11-year career. Pranaya Rout has very good experience with Microsoft Technologies, Including C#, VB, ASP.NET MVC, ASP.NET Web API, EF, EF Core, ADO.NET, LINQ, SQL Server, MYSQL, Oracle, ASP.NET Core, Cloud Computing, Microservices, Design Patterns and still learning new technologies.

Previous Lesson
Checked and Unchecked Keywords in C#

Next Lesson
Boxing and Unboxing in C#

22 thoughts on “Stack and Heap Memory in .NET”

VAIBHAV
MAY 17, 2020 AT 10:31 PM

Nice explanation sir

Reply

DOT NET TUTORIALS
JUNE 24, 2022 AT 6:00 AM

Thank you for finding this article helpful.

Reply

ALDOS
MAY 18, 2020 AT 4:41 AM

much more understanding

Reply

RAHUL
JUNE 19, 2020 AT 5:14 PM

I love this site , each and everything is explain very well . All topic are covered by this site no need to go anywhere just read it carefully .

Thanks to make it

Reply

DOT NET TUTORIALS
JUNE 24, 2022 AT 6:01 AM

Thank you so much. Your feedback means a lot us.

Reply

NAGAMAHESH
JULY 15, 2021 AT 10:33 AM

Nice articles. I am very easy to understand these concepts. Boxing and Unboxing in the .NET concept are missing in the next article. please post this article.

Reply

DOT NET TUTORIALS
JUNE 24, 2022 AT 6:02 AM

Hi,
Thank you for your feedback. We will add the Boxing and Unboxing topics very soon.

Reply

SRINI
OCTOBER 16, 2021 AT 10:42 AM

Very helpful, very detailed for interview preparation

Reply

DOT NET TUTORIALS
JUNE 24, 2022 AT 6:03 AM

Thank you so much for giving your valuable feedback.

Reply

PNS
NOVEMBER 27, 2021 AT 1:17 PM

Nice Article sir...with a understandable examples

Reply

DOT NET TUTORIALS
JUNE 24, 2022 AT 6:01 AM

Your feedback motivates us to write such articles in depth.

Reply

SOMEONE
JUNE 29, 2022 AT 9:09 AM

Nice explanation sir, as a sign of gratitude, I can only click on ads so you can be even more enthusiastic about making more in-depth articles about .net, i can't wait the article about Boxing and Unboxing concept in .net, thank you very much

Reply

DOT NET TUTORIALS
JUNE 29, 2022 AT 2:14 PM

Your feedback is a lot of means to us. In the coming days, you will see Boxing and Unboxing articles along with all the missing articles. Currently, we are reworking on these tutorials.

Reply

BARNABAS.666
JULY 15, 2022 AT 6:59 PM

Hi, thanks for nice tutorial.

I would like to add some info about value types and stack.

Value types are only stored on the stack if they are local variables inside a method, or their parameters. This is what the stack is there for: storing local variables and parameters (and the return pointers of function calls).

If the value type was declared as a variable inside a method then it's stored on the stack.
If the value type was declared as a method parameter then it's stored on the stack.
If the value type was declared as a member of a class then it's stored on the heap, along with its parent.
If the value type was declared as a member of a struct then it's stored wherever that struct is stored.

Reply

DOT NET TUTORIALS
JULY 15, 2022 AT 7:25 PM

Hi BARNABAS.666,

Thanks for your valuable input. Peoples like you motivate us to write more and more content..

Reply

YOGESH
AUGUST 3, 2022 AT 4:28 PM

Please correct this statement –

When the three statements are executed, then the control will exist from the method

to

When the three statements are executed, then the control will exit from the method

there should be word – exit

Reply

DOT NET TUTORIALS
AUGUST 3, 2022 AT 6:51 PM

Hi Yogesh,

Thank you for identifying the typo error. We have corrected the same.

Reply

DEV
DECEMBER 26, 2022 AT 4:10 AM

very nicely explained , thanks alot

Reply

HAZEM
JANUARY 15, 2023 AT 6:41 PM

Thank You.

Reply

ARUN
APRIL 14, 2023 AT 12:38 PM

Nice explanation.

Reply

BHUWAN
MAY 4, 2023 AT 12:23 AM

well explained, very detailed, cover all the area, the best thing is each topic is explained with an example that make easy to understand the things quickly

Reply

王硕
FEBRUARY 26, 2025 AT 7:50 PM

Nice explanation

Reply

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment *

Name*

Email*

Website

Post Comment

Online Training

Most Popular C# Books