

6. C# Language Features

In this chapter we learn how to work with some of the more advanced C# language features. The C# language is a strongly typed language: this means that any attempt to pass a wrong kind of parameter as an argument, or to assign a value to a variable that is not implicitly convertible, will generate a compilation error. This avoids many errors that only happen at runtime in other languages.

6.1 C# Types

6.1.1 General

The types of the C# language are divided into two main categories: **reference types** and **value types**. Both value types and reference types may be *generic types*, which take one or more *type parameters*. Type parameters can designate both value types and reference types..

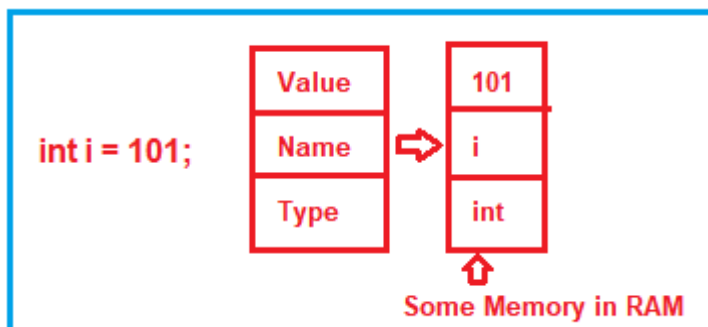
Value types differ from reference types in that variables of the value types directly contain their data, whereas variables of the reference types store *references* to their data, the latter being known as *objects*. With reference types, it is possible for two variables to reference the same object, and thus possible for operations on one variable to affect the object referenced by the other variable. With value types, the variables each have their own copy of the data, and it is not possible for operations on one to affect the other.

C#'s type system is unified such that a value of any type can be treated as an object. Every type in C# directly or indirectly derives from the object class type, and object is the ultimate base class of all types. Values of reference types are treated as objects simply by viewing the values as type object.

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 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.

6.1.1.1 Understanding Stack and Heap Memory in C#:

There are two types of memory allocation for the variables that we created in the .NET Application i.e. stack memory and heap memory. Let us understand the stack and heap memory with an example. In order to understand stack and heap, 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;

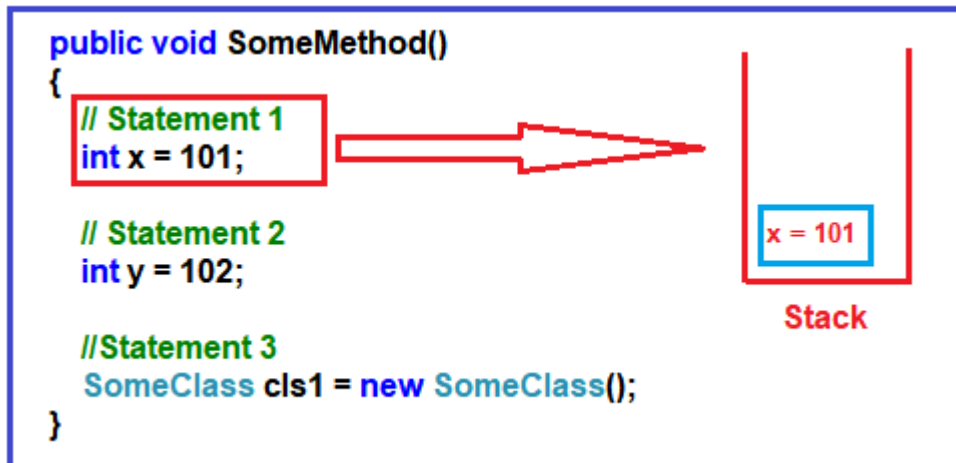
    // Statement 2
    int y = 102;

    //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.

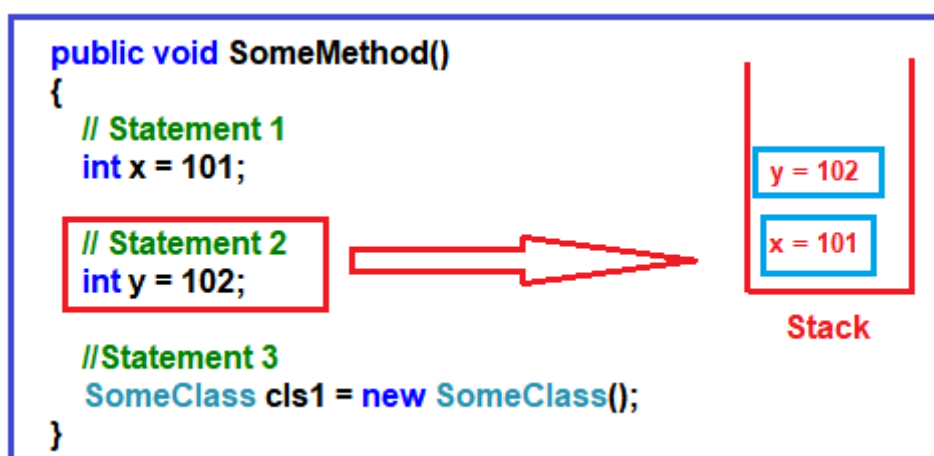
Statement1:

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 better understanding, please have a look at the following image.



Statement2:

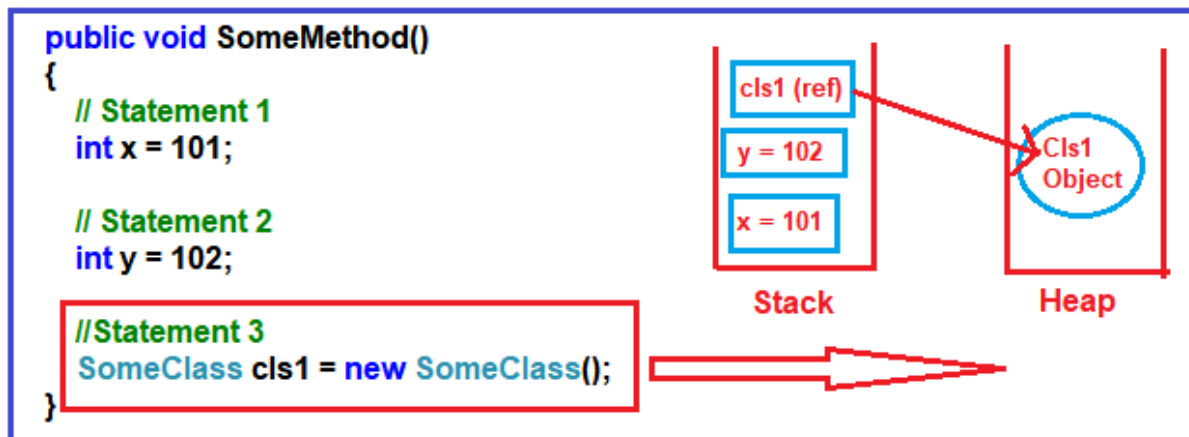
When the second statement is executed, it stacks this memory allocation (memory allocation for variable y) on top of the first memory allocation (memory allocation for variable x). You can think about the stack as a series of plates or dishes put on top of each other. Please have a look at the following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



The Stack Memory allocation and de-allocation in .NET are done using 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:

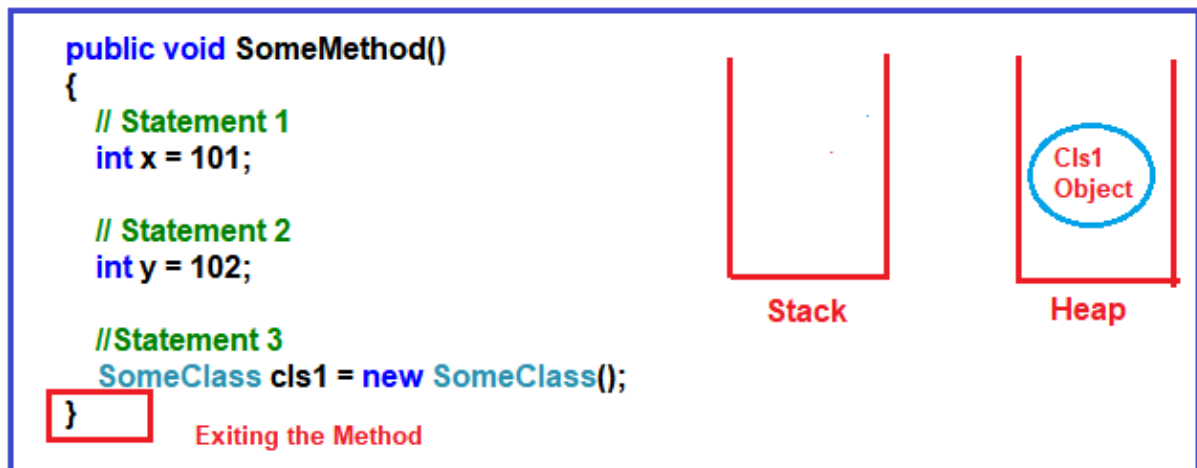
In the 3rd statement, we have created an object of SomeClass. When the 3rd statement is executed, it internally creates a pointer on the stack memory and the actual object is stored in a different memory location called Heap memory. The heap memory location does not track running memory. Heap is used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



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. The time 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, then 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 which are created on the stack. It will de-allocate the memory in a ‘LIFO’ fashion from the stack. For a better understanding please have a look at the below image.

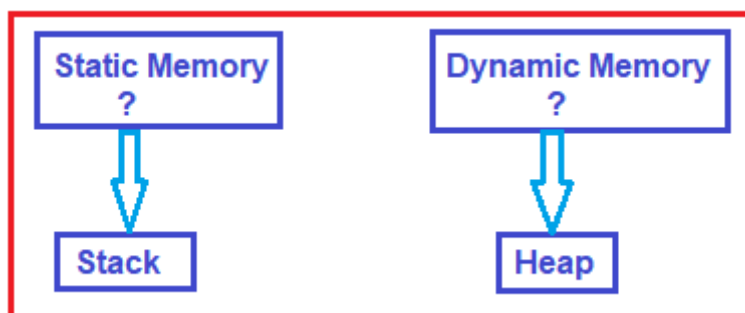


It will not de-allocate the heap memory. Later, the heap memory will be de-allocated by the garbage collector. Now you may have one question in your mind why two types of memory, can't we just allocate everything to just one memory type?

6.1.1.2 Why do we have two types of memory?

As we know, in C#, the primitive data types such as int, double, bool, etc. just 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 as well as other primitive data types.

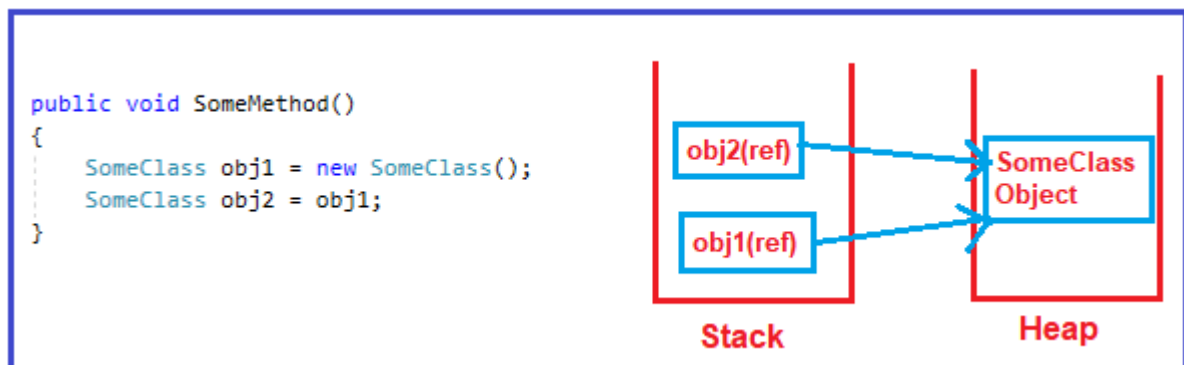
So, the reference data type holds references to other multiple values, and each one of them must be stored in memory. Object types need dynamic memory while primitive data types need static memory. Please have a look at the following image for a better understanding.



6.1.2 Reference Types

Let us understand reference types 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.



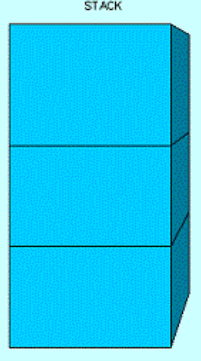
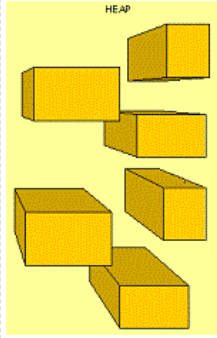
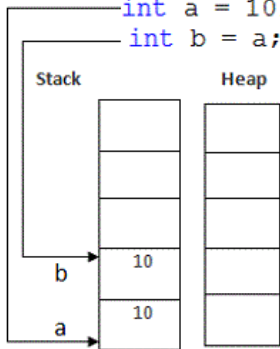
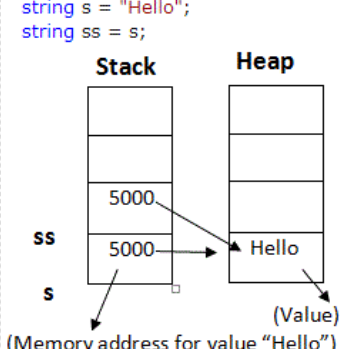
In this case, when you change one of them, the other object is also gets affected. These kinds of data types are termed as 'Reference types' in .NET. So, class, interface, object, string, and delegate are examples of Reference Types.

A reference type value is a reference to an *instance* of the type, the latter known as an object. The special value `null` is compatible with all reference types and indicates the absence of an instance.

6.1.3 Value Types

Unlike a variable of a reference type, a variable of a value type can contain the value `null` only if the value type is a nullable value type. For every non-nullable value type there is a corresponding nullable value type denoting the same set of values plus the value `null`.

Assignment to a variable of a value type creates a *copy* of the value being assigned. This differs from assignment to a variable of a reference type, which copies the reference but not the object identified by the reference.

Category	Stack Memory	Heap Memory
What is Stack & Heap?	<p>It is an array of memory.</p> <p>It is a LIFO (Last In First Out) data structure.</p> <p>In it data can be added to and deleted only from the top of it.</p>	<p>It is an area of memory where chunks are allocated to store certain kinds of data objects.</p> <p>In it data can be stored and removed in any order.</p>
How Memory is Manages?		
Practical Scenario	<pre>int a = 10; int b = a;</pre>  <p>Value of variable storing in stack</p>	<pre>string s = "Hello"; string ss = s;</pre>  <p>Value of variable storing in heap</p>

6.1.4 Stack

The Stack is more or less responsible for keeping track of what's executing in our code (or what's been "called").

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 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 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.

6.1.5 Heap

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.).

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 Heap 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

How is it decided what goes where?

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 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.

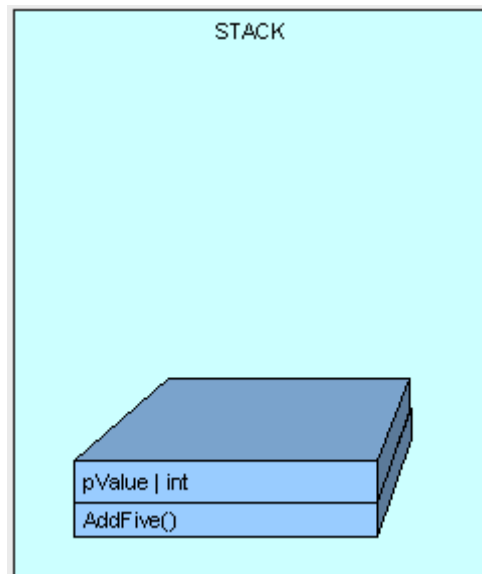
```
public int AddFive(int pValue)
{
    int result;
    result = pValue + 5;
    return result;
}
```

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:

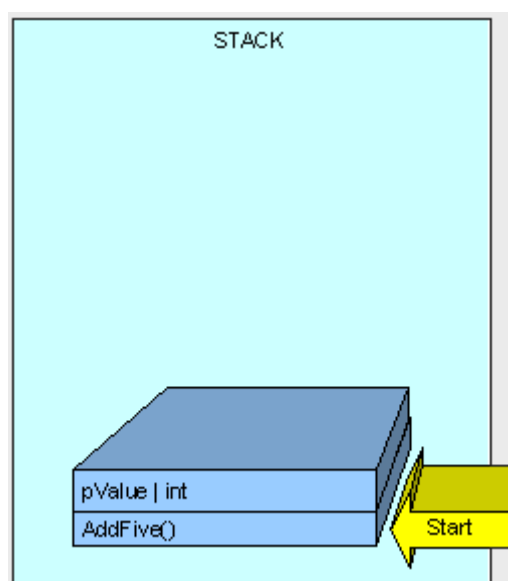
Once we start executing the method, the method's parameters are placed on the stack

NOTE

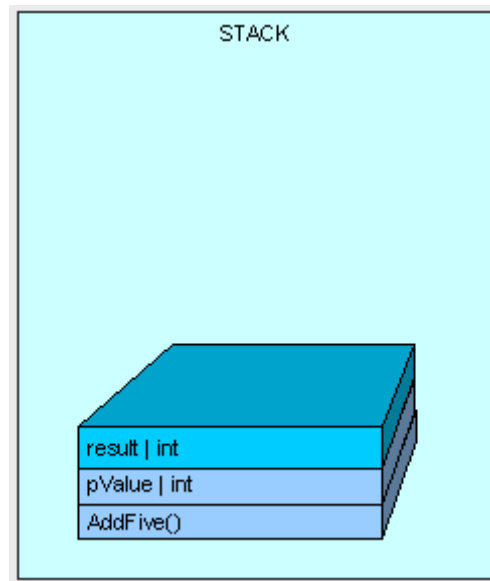
The method does not live on the stack and is illustrated just for reference.



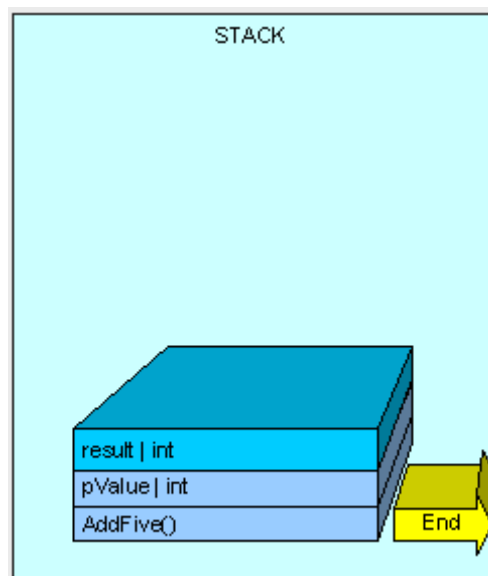
Next, control (the thread executing the method) is passed to the instructions to the AddFive() method which lives in our type's method table, a JIT compilation is performed if this is the first time we are hitting the method.



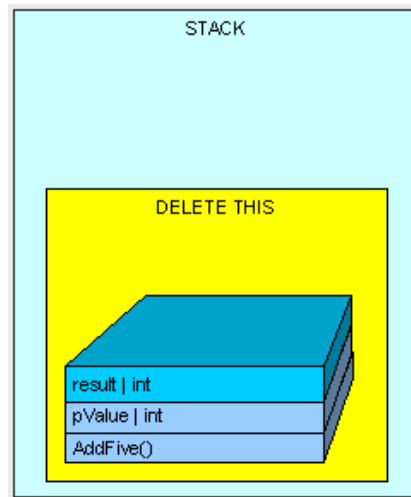
As the method executes, we need some memory for the "result" variable and it is allocated on the stack.



The method finishes execution and our result are 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.

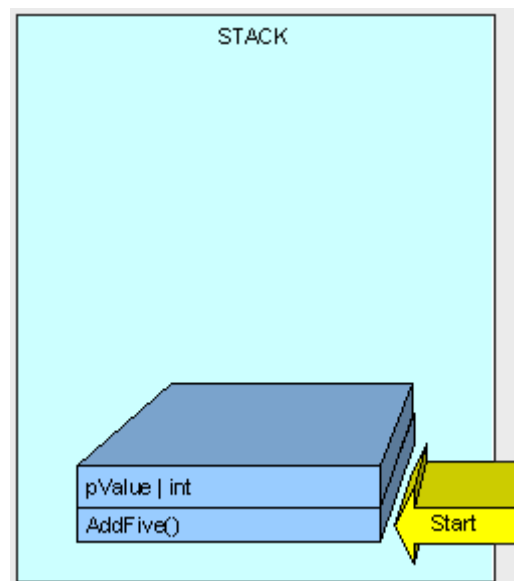
If we have the following MyInt class (which is a Reference Type because it is a class):

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

and the following method is executing:

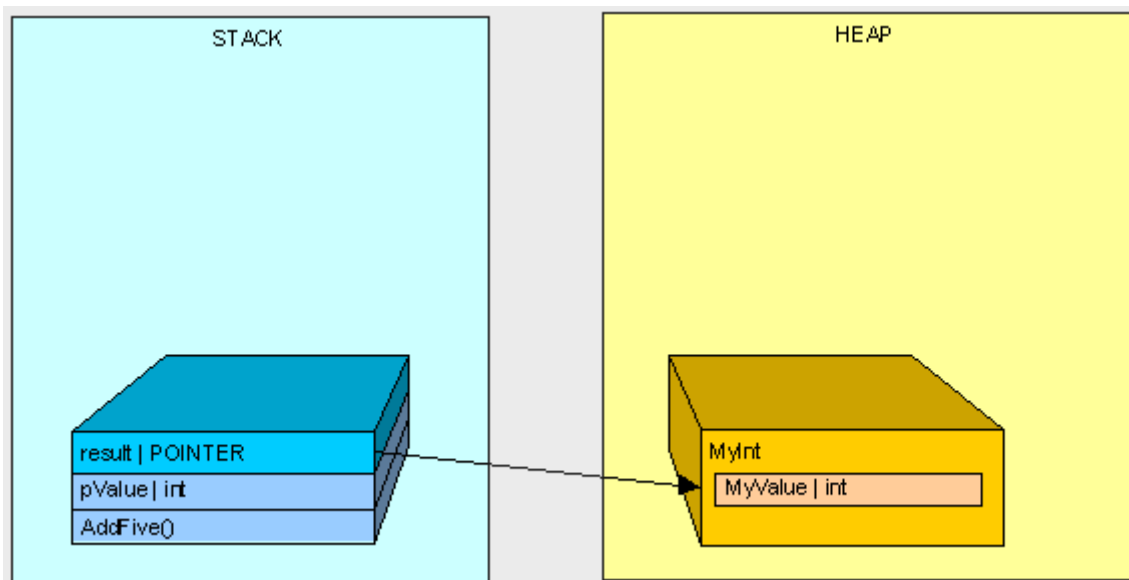
```
public MyInt AddFive(int pValue)
{
    MyInt result = new MyInt();
    result.MyValue = pValue + 5;
    return result;
}
```

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

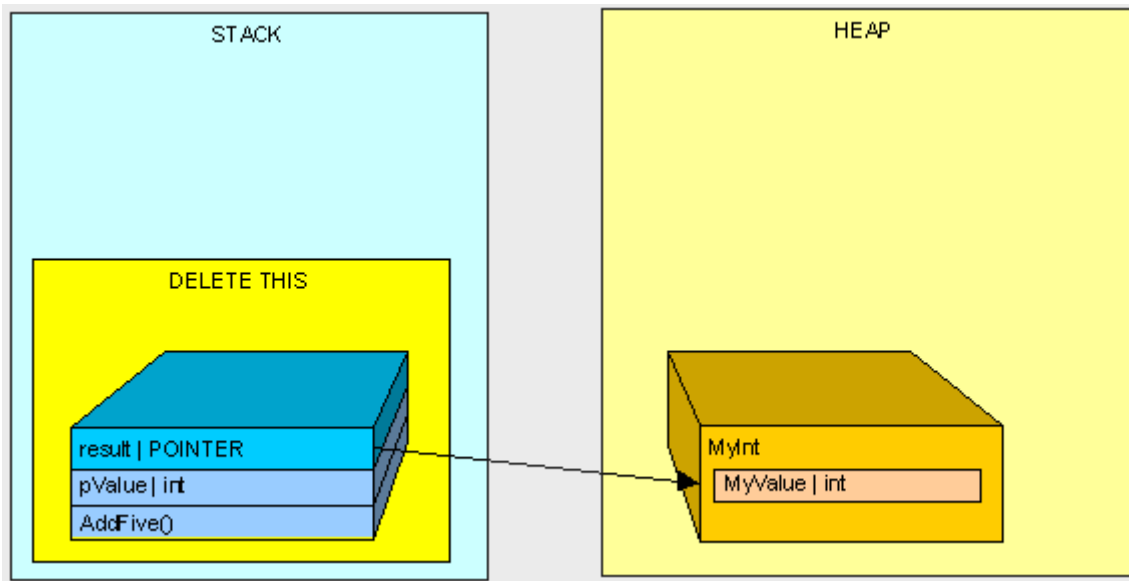


Now is when it gets interesting...

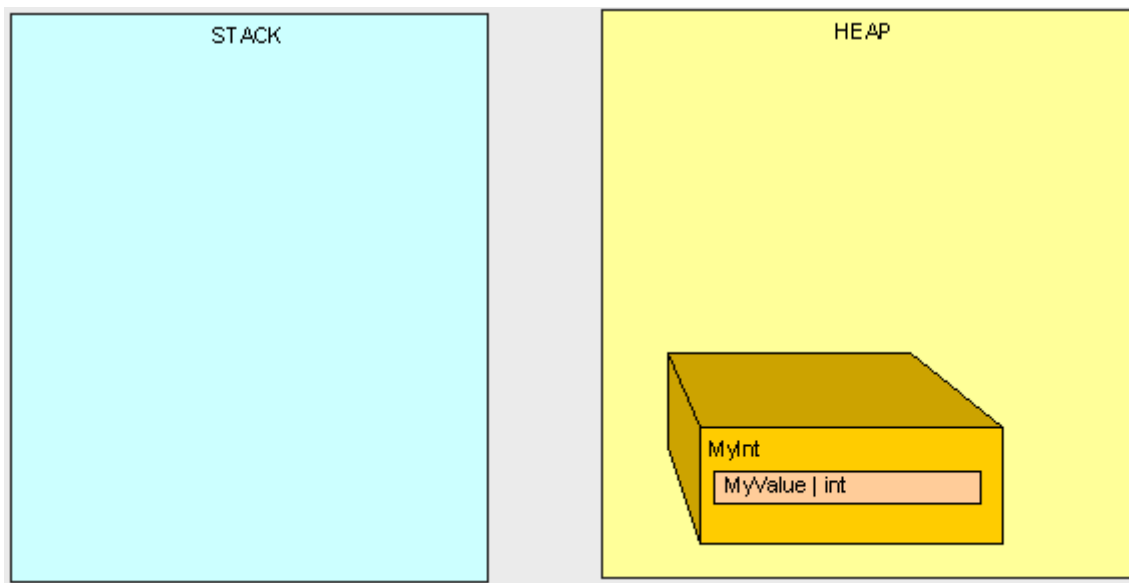
Because MyInt is a Reference Type, it is placed on the Heap and referenced by a Pointer on the Stack.



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.

How does it really affect me?

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.

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.

However, if we are using the MyInt class from before

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

and we are executing the following method:

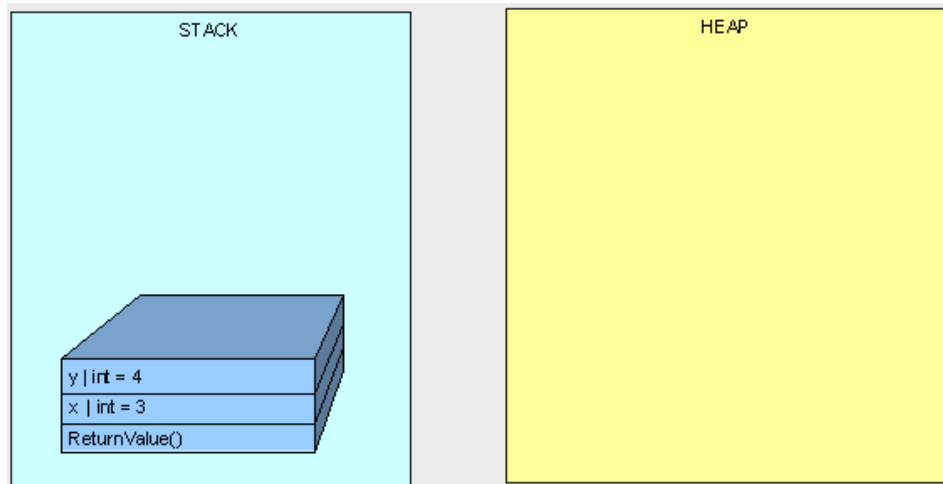
```
public int ReturnValue2()  
{  
    MyInt x = new MyInt();  
    x.MyValue = 3;  
    MyInt y = new MyInt();  
    y = x;  
    y.MyValue = 4;  
    return x.MyValue;  
}
```

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:

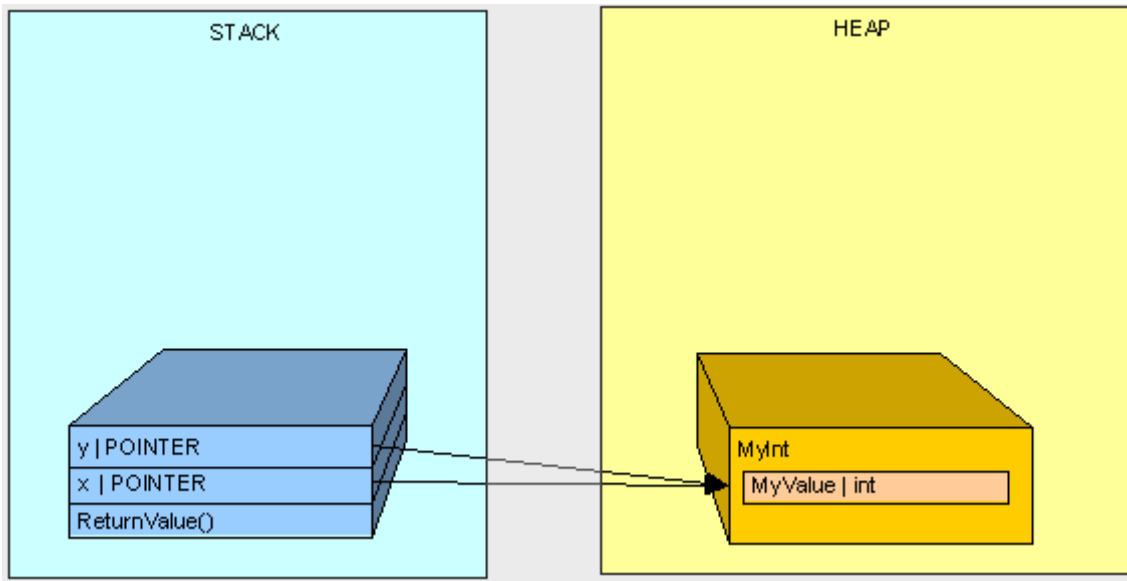
In the first example everything goes as planned:

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



In the next example, we don't get "3" because of 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.

6.1.6 Passing parameters

- Passing Value Types by Value

The following example demonstrates passing value-type parameters by value. The variable `n` is passed by value to the method `SquareIt`. Any changes that take place inside the method have no effect on the original value of the variable.

```
class PassingValByVal
{
    static void SquareIt(int x)
    {
        // The parameter x is passed by value.
        // Changes to x will not affect the original value of x.
        {
            x *= x;
            System.Console.WriteLine("The value inside the method: {0}", x);
        }
    }
    static void Main()
    {
        int n = 5;
        System.Console.WriteLine("The value before calling the method: {0}", n);

        SquareIt(n); // Passing the variable by value.
        System.Console.WriteLine("The value after calling the method: {0}", n);

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
The value before calling the method: 5
The value inside the method: 25
The value after calling the method: 5
*/
```

The variable `n` is a value type. It contains its data, the value 5. When `SquareIt` is invoked, the contents of `n` are copied into the parameter `x`, which is squared inside the method. In `Main`, however, the value of `n` is the same after calling the `SquareIt` method as it was before. The change that takes place inside the method only affects the local variable `x`.

- Passing Value Types by Reference

The following example is the same as the previous example, except that the argument is passed as a `ref` parameter. The value of the underlying argument, `n`, is changed when `x` is changed in the method.

```
class PassingValByRef
{
    static void SquareIt(ref int x)
    // The parameter x is passed by reference.
    // Changes to x will affect the original value of x.
    {
        x *= x;
        System.Console.WriteLine("The value inside the method: {0}", x);
    }
    static void Main()
    {
        int n = 5;
        System.Console.WriteLine("The value before calling the method: {0}", n);

        SquareIt(ref n); // Passing the variable by reference.
        System.Console.WriteLine("The value after calling the method: {0}", n);

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
    The value before calling the method: 5
    The value inside the method: 25
    The value after calling the method: 25
*/
```

In this example, it is not the value of `n` that is passed; rather, a reference to `n` is passed. The parameter `x` is not an `int`; it is a reference to an `int`, in this case, a reference to `n`. Therefore, when `x` is squared inside the method, what actually is squared is what `x` refers to, `n`.

- Passing Reference Types by Value

The following example demonstrates passing a reference-type parameter, `arr`, by value, to a method, `Change`. Because the parameter is a reference to `arr`, it is possible to change the values of the array elements. However, the attempt to

reassign the parameter to a different memory location only works inside the method and does not affect the original variable, `arr`.

```
class PassingRefByVal
{
    static void Change(int[] pArray)
    {
        pArray[0] = 888; // This change affects the original element.
        pArray = new int[5] {-3, -1, -2, -3, -4}; // This change is local.
        System.Console.WriteLine("Inside the method, the first element is: {0}", pArray[0]);
    }

    static void Main()
    {
        int[] arr = {1, 4, 5};
        System.Console.WriteLine("Inside Main, before calling the method, the first element is: {0}", arr [0]);

        Change(arr);
        System.Console.WriteLine("Inside Main, after calling the method, the first element is: {0}", arr [0]);
    }
}
/* Output:
    Inside Main, before calling the method, the first element is: 1
    Inside the method, the first element is: -3
    Inside Main, after calling the method, the first element is: 888
*/
```

In the preceding example, the array, `arr`, which is a reference type, is passed to the method without the `ref` parameter. In such a case, a copy of the reference, which points to `arr`, is passed to the method. The output shows that it is possible for the method to change the contents of an array element, in this case from 1 to 888.

However, allocating a new portion of memory by using the `new` operator inside the `Change` method makes the variable `pArray` reference a new array. Thus, any changes after that will not affect the original array, `arr`, which is created inside `Main`. In fact, two arrays are created in this example, one inside `Main` and one inside the `Change` method.

6.1.7 Structs

A structure type (or struct type) is a value type that can encapsulate data and related functionality. You use the `struct` keyword to define a structure type:

```
public struct Coords
{
    public Coords(double x, double y)
    {
        X = x;
        Y = y;
    }
}
```

```

public double X { get; }
public double Y { get; }

public override string ToString() => $"({X}, {Y})";
}

```

Structure types have *value semantics*. That is, a variable of a structure type contains an instance of the type. By default, variable values are copied on assignment, passing an argument to a method, and returning a method result. For structure-type variables, an instance of the type is copied. For more information, see [Value types](#).

Typically, you use structure types to design small data-centric types that provide little or no behavior. For example, .NET uses structure types to represent a number (both [integer](#) and [real](#)), a [Boolean value](#), a [Unicode character](#), a [time instance](#). If you're focused on the behavior of a type, consider defining a [class](#). Class types have *reference semantics*. That is, a variable of a class type contains a reference to an instance of the type, not the instance itself.

6.1.8 Enums

An *enumeration type* (or *enum type*) is a value type defined by a set of named constants of the underlying integral numeric type. To define an enumeration type, use the `enum` keyword and specify the names of *enum members*:

```

C# Copy

enum Color
{
    Red,
    Green,
    Blue
}

```

This example declares an enum type named `Color` with members `Red`, `Green`, and `Blue`.

By default, the associated constant values of enum members are of type `int`; they start with zero and increase by one following the definition text order. You can explicitly specify any other integral numeric type as an underlying type of an enumeration type. You can also explicitly specify the associated constant values, as the following example shows:

```

C# Copy

enum ErrorCode : ushort
{
    None = 0,
    Unknown = 1,
    ConnectionLost = 100,
    OutlierReading = 200
}

```

6.2 Property Accessibility

The `get` and `set` portions of a property or indexer are called *accessors*. By default these accessors have the same visibility or access level of the property or indexer to which they belong. However, it's sometimes useful to restrict access to one of these accessors.

You can create read-only properties, or give different accessibility to the set and get accessors. Suppose that your `Person` class should only enable changing the value of the `FirstName` property from other methods in that class. You could give the set accessor `private` accessibility instead of `public`:

```

public class Person
{
    public string FirstName { get; private set; }

    // Omitted for brevity.
}

```

Now, the `FirstName` property can be accessed from any code, but it can only be assigned from other code in the `Person` class.

You can add any restrictive access modifier to either the set or get accessors. Any access modifier you place on the individual accessor must be more limited than the access modifier on the property definition. The above is legal because the `FirstName` property is `public`, but the set accessor is `private`. You couldn't declare a `private` property with a `public` accessor. Property declarations can also be declared `protected`, `internal`, `protected internal`, or, even `private`.

It's also legal to place the more restrictive modifier on the `get` accessor. For example, you could have a `public` property, but restrict the `get` accessor to `private`. That scenario is rarely done in practice.

- Read-only

You can also restrict modifications to a property so that it can only be set in a constructor. You can modify the `Person` class so as follows:

```
public class Person
{
    public Person(string firstName) => FirstName = firstName;

    public string FirstName { get; }

    // Omitted for brevity.
}
```

6.3 Attributes

Attributes provide a way of associating information with code in a declarative way. They can also provide a reusable element that can be applied to a variety of targets.

Consider the `[Obsolete]` attribute. It can be applied to classes, structs, methods, constructors, and more. It *declares* that the element is obsolete. It's then up to the C# compiler to look for this attribute, and do some action in response.

In C#, attributes are classes that inherit from the `Attribute` base class. Any class that inherits from `Attribute` can be used as a sort of "tag" on other pieces of code. For instance, there is an attribute called `ObsoleteAttribute`. This is used to signal that code is obsolete and shouldn't be used anymore. You can place this attribute on a class, for instance, by using square brackets.

Note that while the class is called `ObsoleteAttribute`, it's only necessary to use `[Obsolete]` in the code. This is a convention that C# follows. You can use the full name `[ObsoleteAttribute]` if you choose.

When marking a class obsolete, it's a good idea to provide some information as to *why* it's obsolete, and/or *what* to use instead. Do this by passing a string parameter to the `Obsolete` attribute.

```
[Obsolete("ThisClass is obsolete. Use ThisClass2 instead.")]
public class ThisClass
{
}
```

The string is being passed as an argument to an `ObsoleteAttribute` constructor, just as if you were writing `var attr = new ObsoleteAttribute("some string").`

Attributes can be used on a number of "targets". The above examples show them on classes, but they can also be used on:

- Assembly
- Class
- Constructor
- Delegate
- Enum
- Event
- Field
- GenericParameter
- Interface
- Method
- Module
- Parameter
- Property
- ReturnValue
- Struct

6.4 Extension methods

Extension methods enable you to "add" methods to existing types without creating a new derived type, recompiling, or otherwise modifying the original type. Extension methods are static methods, but they're called as if they were instance methods on the extended type. For client code written in C#, there's no apparent difference between calling an extension method and the methods defined in a type.

Extension methods are defined as static methods but are called by using instance method syntax. Their first parameter specifies which type the method operates on. The parameter is preceded by the `this` modifier. Extension methods are only in scope when you explicitly import the namespace into your source code with a `using` directive.

The following example shows an extension method defined for the `System.String` class. It's defined inside a non-nested, non-generic static class:

```
namespace ExtensionMethods
{
    public static class MyExtensions
    {
        public static int WordCount(this string str)
        {
            return str.Split(new char[] { ' ', '.', '?' },
                             StringSplitOptions.RemoveEmptyEntries).Length;
        }
    }
}
```

The `WordCount` extension method can be brought into scope with this `using` directive, and it can be called from an application by using this syntax:

```
using ExtensionMethods;
```

```
string s = "Hello Extension Methods";
int i = s.WordCount();
```

You invoke the extension method in your code with instance method syntax. The intermediate language (IL) generated by the compiler translates your code into a call on the static method. The principle of encapsulation is not really being violated. Extension methods cannot access private variables in the type they are extending. Both the `MyExtensions` class and the `WordCount` method are static, and it can be accessed like all other static members. The `WordCount` method can be invoked like other static methods as follows:

```
string s = "Hello Extension Methods";
int i = MyExtensions.WordCount(s);
```

The preceding C# code:

- Declares and assigns a new `string` named `s` with a value of "Hello Extension Methods".
- Calls `MyExtensions.WordCount` given argument `s`

In general, you'll probably be calling extension methods far more often than implementing your own. Because extension methods are called by using instance method syntax, no special knowledge is required to use them from client code. To enable extension methods for a particular type, just add a `using` directive for the namespace in which the methods are defined.

6.5 Null-conditional operator

C# 6 introduces the "null-conditional operator" `?.` When used on an object on the right-hand side of an expression, the null-conditional operator returns the member value if the object is not `null` and `null` otherwise:

```
var employee = new Employee();
string lowerName = employee.FirstName?.ToLower();
```

The code in the previous example is the same as:

```
var employee = new Employee();
string lowerName;
if (employee.FirstName != null)
{
    lowerName = employee.FirstName.ToLower();
}
else
{
    lowerName = null;
}
```

If the operand *FirstName* is null then the *ToLower* method will not be invoked. The variable *lowerName* will get the value *null*.

If the operand *FirstName* is not null the *ToLower* method will be invoked. The variable *lowerName* will get the value returned by the *ToLower* method.

This operator can reduce the amount of if-tests that check for null to avoid a *NullReferenceException* and make the code more readable.

6.6 Nullable reference type

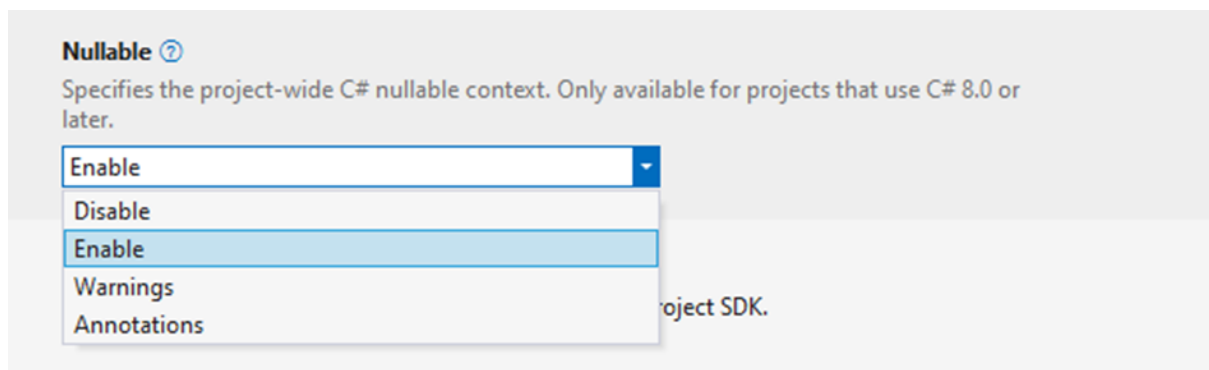
Prior to C# 8.0, all reference types were nullable. *Nullable reference types* refers to a group of features introduced in C# 8.0 that you can use to minimize the likelihood that your code causes the runtime to throw [System.NullReferenceException](#).

The rest of this article describes how those three feature areas work to produce warnings when your code may be dereferencing a `null` value. Dereferencing a variable means to access one of its members using the `.` (dot) operator, as shown in the following example:

```
string message = "Hello, World!";
int length = message.Length; // dereferencing "message"
```

When you dereference a variable whose value is `null`, the runtime throws a [System.NullReferenceException](#).

The nullable context can be set for a project using the [<Nullable> element](#) in your `.csproj` file.

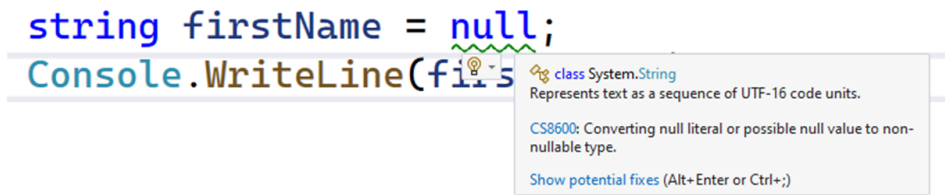


This element configures how the compiler interprets the nullability of types and what warnings are emitted. The following table shows the allowable values and summarizes the contexts they specify.

Context	Reference types	? suffix	! operator
disable	All are nullable	Can't be used	Has no effect
enable	Non-nullable unless declared with ?	Declares nullable type	Suppresses warnings for possible <code>null</code> assignment
warnings	All are nullable, but members are considered not <code>null</code> at opening brace of method	Produces a warning	Suppresses warnings for possible <code>null</code> assignment
annotations	Non-nullable unless declared with ?	Declares nullable type	Has no effect

- Enable: compiler produces a warning when assigning null to a non-nullable type (each reference type)

```
string firstName = null;
Console.WriteLine(firstName);
```



The screenshot shows a code editor with the following code:

```
string firstName = null;
Console.WriteLine(firstName);
```

A warning icon (lightbulb) is placed over the `null` value. A tooltip is displayed, showing the `class System.String` and the text "Represents text as a sequence of UTF-16 code units." Below this, it states "CS8600: Converting null literal or possible null value to non-nullable type." and provides a link to "Show potential fixes (Alt+Enter or Ctrl+;)"

- Nullable reference type: same syntax as nullable value type: the compiler will produce no warning because a `?` is used

```
string? firstName = null;
Console.WriteLine(firstName);
```

- Assigning null-reference type to a nullable reference type and vica-versa

```
string firstName = null;
Console.WriteLine(firstName);
string? lastName = null;
Console.WriteLine(lastName);

lastName= firstName;
firstName= lastName;
```

The fourth line will compile without any warnings, because `lastName` can take a string reference or null.

The sixth line however will give you a warning, because `firstName` can only accept a reference to a string and not null. Because `lastName` can be null, the compiler emits a warning.

- Static Code analysis

```
static string? CanReturnANull(bool returnNull)
    => returnNull ? null : "Hello";
```

```
static string CanNotReturnANull()
    => "World!";
```

```
string result1 = CanNotReturnANull();
Console.WriteLine(result1.Length);
```

```
result1 = CanReturnANull(true); // warning
result1 = CanReturnANull(false); // warning
```

With nullable reference types, the compiler will analyse your code for possible `NullReferenceExceptions` and issue warnings.

When you call `CanNotReturnANull` the compiler does not show a warning because it knows this function cannot return a null value:

However, when calling `CanReturnANull` the compiler does show a warning because this function can return a null value as indicated by the method's return type:

```
string? result2 = CanReturnANull(true);
Console.WriteLine(result2.Length);
```

In this case you need to assign the `CanReturnANull` result to a nullable reference: Using this result in some expression will again result in a compiler warning because it can cause an exception:

```
string? result2 = CanReturnANull(true);
if (result2 != null)
{
    Console.WriteLine(result2.Length);
}
```

You can remove the warning by adding a null check:

- Why nullable reference types?
 - Less Null checks (make the compiler do a lot of the work for you)
 - Let the compiler find potential null-references, instead of finding them at runtime, especially in production!
 - Make your code more expressive

6.7 The IS operator

The `is` operator checks if the result of an expression is compatible with a given type.

Beginning with C# 7.0, you can also use the `is` operator to match an expression against a pattern, as the following example shows:

```
static bool IsFirstFridayOfOctober(DateTime date) =>
    date is { Month: 10, Day: <=7, DayOfWeek: DayOfWeek.Friday };
```

In the preceding example, the `is` operator matches an expression against a [property pattern](#) (available in C# 8.0 and later) with nested [constant](#) and [relational](#) (available in C# 9.0 and later) patterns.

The `is` operator can be useful in the following scenarios:

- To check the run-time type of an expression, as the following example shows:

```
int i = 34;
object iBoxed = i;
int? jNullable = 42;
if (iBoxed is int a && jNullable is int b)
{
    Console.WriteLine(a + b); // output 76
}
```

The preceding example shows the use of a [declaration pattern](#).

- To check for null, as the following example shows:

```
if (input is null)
{
    return;
}
```

When you match an expression against `null`, the compiler guarantees that no user-overloaded `==` or `!=` operator is invoked. Beginning with C# 11, you can use `is null` on unconstrained generic types.

- Beginning with C# 9.0, you can use a [negation pattern](#) to do a non-null check, as the following example shows:

```
if (result is not null)
{
    Console.WriteLine(result.ToString());
}
```

- Beginning with C# 11, you can use list patterns to match elements of a list or array. The following code checks arrays for integer values in expected positions:

```
int[] empty = { };
int[] one = { 1 };
int[] odd = { 1, 3, 5 };
int[] even = { 2, 4, 6 };
int[] fib = { 1, 1, 2, 3, 5 };

Console.WriteLine(odd is [1, _, 2, ..]); // false
Console.WriteLine(fib is [1, _, 2, ..]); // true
Console.WriteLine(fib is [_, 1, 2, 3, ..]); // true
Console.WriteLine(fib is [.., 1, 2, 3, _]); // true
Console.WriteLine(even is [2, _, 6]); // true
Console.WriteLine(even is [2, .., 6]); // true
Console.WriteLine(odd is [.., 3, 5]); // true
Console.WriteLine(even is [.., 3, 5]); // false
Console.WriteLine(fib is [.., 3, 5]); // true
```

6.8 Readonly

In a [field declaration](#), `readonly` keyword indicates that assignment to the field can only occur as part of the declaration or in a constructor in the same class. A readonly field can be assigned and reassigned multiple times within the field declaration and constructor.

A `readonly` field can't be assigned after the constructor exits. This rule has different implications for value types and reference types:

- Because value types directly contain their data, a field that is a `readonly` value type is immutable.
- Because reference types contain a reference to their data, a field that is a `readonly` reference type must always refer to the same object. That object isn't immutable. The `readonly` modifier prevents the field from being replaced by a different instance of the reference type. However, the modifier doesn't prevent the instance data of the field from being modified through the read-only field.

In this example, the value of the field `year` can't be changed in the method `ChangeYear`, even though it's assigned a value in the class constructor:

```
class Age
{
    private readonly int _year;
    Age(int year)
    {
        _year = year;
    }
    void ChangeYear()
    {
        //_year = 1967; // Compile error if uncommented.
    }
}
```

You can assign a value to a `readonly` field only in the following contexts:

- When the variable is initialized in the declaration, for example:

```
public readonly int y = 5;
```
- In an instance constructor of the class that contains the instance field declaration.
- In the static constructor of the class that contains the static field declaration.

6.9 Anonymous types

Anonymous types provide a convenient way to encapsulate a set of read-only properties into a single object without having to explicitly define a type first. The type name is generated by the compiler and is not available at the source code level. The type of each property is inferred by the compiler.

You create anonymous types by using the [new](#) operator together with an object initializer. For more information about object initializers, see [Object and Collection Initializers](#).

The following example shows an anonymous type that is initialized with two properties named `Amount` and `Message`.

```
var v = new { Amount = 108, Message = "Hello" };

// Rest the mouse pointer over v.Amount and v.Message in the following
// statement to verify that their inferred types are int and string.
Console.WriteLine(v.Amount + v.Message);
```

Anonymous types typically are used in the [select](#) clause of a query expression to return a subset of the properties from each object in the source sequence. For more information about queries, see [LINQ in C#](#).

Anonymous types contain one or more public read-only properties. No other kinds of class members, such as methods or events, are valid. The expression that is used to initialize a property cannot be `null`, an anonymous function, or a pointer type.

The most common scenario is to initialize an anonymous type with properties from another type. In the following example, assume that a class exists that is named `Product`. Class `Product` includes `Color` and `Price` properties, together with other properties that you are not interested in. Variable `products` is a collection of `Product` objects. The anonymous type declaration starts with the `new` keyword. The declaration initializes a new type that uses only two properties from `Product`. Using anonymous types causes a smaller amount of data to be returned in the query.

If you don't specify member names in the anonymous type, the compiler gives the anonymous type members the same name as the property being used to initialize them. You provide a name for a property that's being initialized with an expression, as shown in the previous example. In the following example, the names of the properties of the anonymous type are `Color` and `Price`.

```
var productQuery =
    from prod in products
    select new { prod.Color, prod.Price };

foreach (var v in productQuery)
{
    Console.WriteLine("Color={0}, Price={1}", v.Color, v.Price);
}
```


Typically, when you use an anonymous type to initialize a variable, you declare the variable as an implicitly typed local variable by using `var`. The type name cannot be specified in the variable declaration because only the compiler has access to the underlying name of the anonymous type. For more information about `var`, see [Implicitly Typed Local Variables](#).

You can create an array of anonymously typed elements by combining an implicitly typed local variable and an implicitly typed array, as shown in the following example.

```
var anonArray = new[] { new { name = "apple", diam = 4 }, new { name = "grape", diam = 1 } };
```

Anonymous types are `class` types that derive directly from `object`, and that cannot be cast to any type except `object`. The compiler provides a name for each anonymous type, although your application cannot access it. From the perspective of the common language runtime, an anonymous type is no different from any other reference type.

If two or more anonymous object initializers in an assembly specify a sequence of properties that are in the same order and that have the same names and types, the compiler treats the objects as instances of the same type. They share the same compiler-generated type information.

6.10 Delegates and Events

Delegates provide a *late binding* mechanism in .NET. Late Binding means that you create an algorithm where the caller also supplies at least one method that implements part of the algorithm.

For example, consider sorting a list of stars in an astronomy application. You may choose to sort those stars by their distance from the earth, or the magnitude of the star, or their perceived brightness.

In all those cases, the `Sort()` method does essentially the same thing: arranges the items in the list based on some comparison. The code that compares two stars is different for each of the sort orderings.

These kinds of solutions have been used in software for half a century. The C# language delegate concept provides first class language support, and type safety around the concept.

As you'll see later, the C# code you write for algorithms like this is type safe. The compiler ensures that the types match for arguments and return types.

- **Define delegate types**

Let's start with the 'delegate' keyword, because that's primarily what you will use as you work with delegates. The code that the compiler generates when you use the `delegate` keyword will map to method calls that invoke members of the [Delegate](#) and [MulticastDelegate](#) classes.

You define a delegate type using syntax that is similar to defining a method signature. You just add the `delegate` keyword to the definition.

Let's continue to use the `List.Sort()` method as our example. The first step is to create a type for the comparison delegate:

```
// Define the delegate type:  
public delegate int Comparison<in T>(T left, T right);
```

The compiler generates a class, derived from `System.Delegate` that matches the signature used (in this case, a method that returns an integer, and has two arguments). The type of that delegate is `Comparison`. The `Comparison` delegate type is a generic type

Notice that the syntax may appear as though it is declaring a variable, but it is actually declaring a *type*. You can define delegate types inside classes, directly inside namespaces, or even in the global namespace.

The compiler also generates add and remove handlers for this new type so that clients of this class can add and remove methods from an instance's invocation list. The compiler will enforce that the signature of the method being added or removed matches the signature used when declaring the method.

- **Declare instances of delegates**

After defining the delegate, you can create an instance of that type. Like all variables in C#, you cannot declare delegate instances directly in a namespace, or in the global namespace.

```
// Declare an instance of that type:  
public Comparison<T> comparator;
```

The type of the variable is `Comparison<T>`, the delegate type defined earlier. The name of the variable is `comparator`.

That code snippet above declared a member variable inside a class. You can also declare delegate variables that are local variables, or arguments to methods.

- **Invoke delegates**

You invoke the methods that are in the invocation list of a delegate by calling that delegate. Inside the `Sort()` method, the code will call the comparison method to determine which order to place objects:

```
int result = comparator(left, right);
```

In the line above, the code *invokes* the method attached to the delegate. You treat the variable as a method name, and invoke it using normal method call syntax.

That line of code makes an unsafe assumption: There's no guarantee that a target has been added to the delegate. If no targets have been attached, the line above would cause a `NullReferenceException` to be thrown. The idioms used to address this problem are more complicated than a simple null-check, and are covered later on.

- **Assign, add, and remove invocation targets**

That's how a delegate type is defined, and how delegate instances are declared and invoked.

Developers that want to use the `List.Sort()` method need to define a method whose signature matches the delegate type definition, and assign it to the delegate used by the sort method. This assignment adds the method to the invocation list of that delegate object.

Suppose you wanted to sort a list of strings by their length. Your comparison function might be the following:

```
private static int CompareLength(string left, string right) =>
    left.Length.CompareTo(right.Length);
```

The method is declared as a private method. That's fine. You may not want this method to be part of your public interface. It can still be used as the comparison method when attached to a delegate. The calling code will have this method attached to the target list of the delegate object, and can access it through that delegate.

You create that relationship by passing that method to the `List.Sort()` method:

```
phrases.Sort(CompareLength);
```

Notice that the method name is used, without parentheses. Using the method as an argument tells the compiler to convert the method reference into a reference that can be used as a delegate invocation target, and attach that method as an invocation target.

You could also have been explicit by declaring a variable of type `Comparison<string>` and doing an assignment:

```
Comparison<string> comparer = CompareLength;
phrases.Sort(comparer);
```

In uses where the method being used as a delegate target is a small method, it's common to use [lambda expression](#) syntax to perform the assignment:

```
Comparison<string> comparer = (left, right) => left.Length.CompareTo(right.Length);
phrases.Sort(comparer);
```

Using lambda expressions for delegate targets is covered more in a later section.

The `Sort()` example typically attaches a single target method to the delegate. However, delegate objects do support invocation lists that have multiple target methods attached to a delegate object.

- **Delegate and MulticastDelegate classes**

The language support described above provides the features and support you'll typically need to work with delegates. These features are built on two classes in the .NET Core framework: [Delegate](#) and [MulticastDelegate](#).

The `System.Delegate` class and its single direct subclass, `System.MulticastDelegate`, provide the framework support for creating delegates, registering methods as delegate targets, and invoking all methods that are registered as a delegate target.

Interestingly, the `System.Delegate` and `System.MulticastDelegate` classes are not themselves delegate types. They do provide the basis for all specific delegate types. That same language design process mandated that you cannot declare a class that derives from `Delegate` or `MulticastDelegate`. The C# language rules prohibit it.

Instead, the C# compiler creates instances of a class derived from `MulticastDelegate` when you use the C# language keyword to declare delegate types.

This design has its roots in the first release of C# and .NET. One goal for the design team was to ensure that the language enforced type safety when using delegates. That meant ensuring that delegates were invoked with the right type and number of arguments. And, that any return type was correctly indicated at compile time. Delegates were part of the 1.0 .NET release, which was before generics.

The best way to enforce this type safety was for the compiler to create the concrete delegate classes that represented the method signature being used.

Even though you cannot create derived classes directly, you will use the methods defined on these classes. Let's go through the most common methods that you will use when you work with delegates.

The first, most important fact to remember is that every delegate you work with is derived from `MulticastDelegate`. A multicast delegate means that more than one method target can be invoked when invoking through a delegate. The original design considered making a distinction between delegates where only one target method could be attached and invoked, and delegates where multiple target methods could be attached and invoked. That distinction proved to be less useful in practice than originally thought. The two different classes were already created, and have been in the framework since its initial public release.

The methods that you will use the most with delegates are `Invoke()` and `BeginInvoke() / EndInvoke()`. `Invoke()` will invoke all the methods that have been attached to a particular delegate instance. As you saw above, you typically invoke delegates using the method call syntax on the delegate variable. As you'll see [later in this series](#), there are patterns that work directly with these methods.

Now that you've seen the language syntax and the classes that support delegates, let's examine how strongly typed delegates are used, created, and invoked.

6.11 Lambda's

You use a *lambda expression* to create an anonymous function. Use the lambda declaration operator `=>` to separate the lambda's parameter list from its body.

To create a lambda expression, you specify input parameters (if any) on the left side of the lambda operator and an expression or a statement block on the other side.

Specify zero input parameters with empty parentheses:

```
() => Console.WriteLine();
```

If a lambda expression has only one input parameter, parentheses are optional.

```
x => x * x * x;
```

Two or more input parameters are separated by commas.

```
(int x, string s) => s.Length > x;
```

Input parameters and return types of a lambda expression are strongly typed at compile time. When the compiler can infer the types of input parameters, you may omit type declarations. If you need to specify the type of input parameters, you must do that for each parameter (as you can see in the previous and next example)

```
int[] numbers = { 4, 7, 10 };
int product = numbers.Aggregate(1, (int interim, int next) => interim * next);
Console.WriteLine(product); // output: 280
```

A lambda expression can be of any of the following two forms:

- Expression lambda that has an expression as its body:

```
(input-parameters) => expression
```

```

public MainWindow()
{
    InitializeComponent();

    SubmitButton.Click += SubmitButton_Click;

    SubmitButton.Click += (sender, args) => MessageBox.Show("Click handled by lambda");
}
private void SubmitButton_Click(object sender, RoutedEventArgs e)
{
    MessageBox.Show("Click handled by classic method");
}

```

Inline parameters Lambda operator Method body

```

public delegate int CalculateDelegate(int a, int b);

public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();

        CalculateDelegate addDelegate = (x, y) => x + y;
        MessageBox.Show($"1 + 5 = {addDelegate(1, 5)}");
    }
}

```

There is no need to use the return keyword when the body only contains one line of code.

The compiler can infer that an int should be returned because the lambda is assigned to a *CalculateDelegate*.

- Statement lambda that has a statement block as its body:

```
(input-parameters) => { <sequence-of-statements> }
```

```

public delegate int CalculateDelegate(int a, int b);

public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();

        CalculateDelegate absoluteMaxDelegate = (x, y) =>
        {
            x = Math.Abs(x);
            y = Math.Abs(y);
            return Math.Max(x, y);
        };

        MessageBox.Show($"Absolute maximum of -5 and 1 = {absoluteMaxDelegate(-5, 1)}");
    }
}

```

With multiple lines of code in the body a return statement and curly brackets are needed.

- Action<T>

You can use the Built-in generic delegate Action<T> to pass a method as a parameter without explicitly declaring a custom delegate. It encapsulates a method that has a single parameter and does not return a value. T is the type of the parameter of the method that this delegate encapsulates.

The following example simplifies code by instantiating the [Action<T>](#) delegate instead of explicitly defining a new delegate and assigning a named method to it.

```

using System;
using System.Windows.Forms;

public class TestAction1
{
    public static void Main()
    {
        Action<string> messageTarget;

        if (Environment.GetCommandLineArgs().Length > 1)
            messageTarget = ShowWindowsMessage;
        else
            messageTarget = Console.WriteLine;

        messageTarget("Hello, World!");
    }

    private static void ShowWindowsMessage(string message)
    {
        MessageBox.Show(message);
    }
}

```

You can use Action<T1,T2,T3,...> for multiple parameters


```

using System;

public class TestAction3
{
    public static void Main()
    {
        string[] ordinals = {"First", "Second", "Third", "Fourth", "Fifth"};
        string[] copiedOrdinals = new string[ordinals.Length];
        Action<string[], string[], int> copyOperation = CopyStrings;
        copyOperation(ordinals, copiedOrdinals, 3);
        foreach (string ordinal in copiedOrdinals)
            Console.WriteLine(string.IsNullOrEmpty(ordinal) ? "<None>" : ordinal);
    }

    private static void CopyStrings(string[] source, string[] target, int startPos)
    {
        if (source.Length != target.Length)
            throw new IndexOutOfRangeException("The source and target arrays must have the same number of elements.");

        for (int ctr = startPos; ctr <= source.Length - 1; ctr++)
            target[ctr] = string.Copy(source[ctr]);
    }
}

```

of use Action for zero parameters

```

public delegate void Action();

```

You can also assign a lambda expression to an Action<T> delegate instance, as the following example illustrates.

```

using System;
using System.Windows.Forms;

public class TestLambdaExpression
{
    public static void Main()
    {
        Action<string> messageTarget;

        if (Environment.GetCommandLineArgs().Length > 1)
            messageTarget = s => ShowWindowsMessage(s);
        else
            messageTarget = s => Console.WriteLine(s);

        messageTarget("Hello, World!");
    }

    private static void ShowWindowsMessage(string message)
    {
        MessageBox.Show(message);
    }
}

```

- Func<T, TResult>

Encapsulates a method that has one parameters and returns a value of the type specified by the `TResult` parameter.

```
using System;
using System.IO;

public class TestDelegate
{
    public static void Main()
    {
        OutputTarget output = new OutputTarget();
        Func<bool> methodCall = output.SendToFile;
        if (methodCall())
            Console.WriteLine("Success!");
        else
            Console.WriteLine("File write operation failed.");
    }
}

public class OutputTarget
{
    public bool SendToFile()
    {
        try
        {
            string fn = Path.GetTempFileName();
            StreamWriter sw = new StreamWriter(fn);
            sw.WriteLine("Hello, World!");
            sw.Close();
            return true;
        }
        catch
        {
            return false;
        }
    }
}
```

6.12 Sources

[1] Microsoft, “C# Documentation”, Available:
<https://docs.microsoft.com/en-us/dotnet/csharp/>