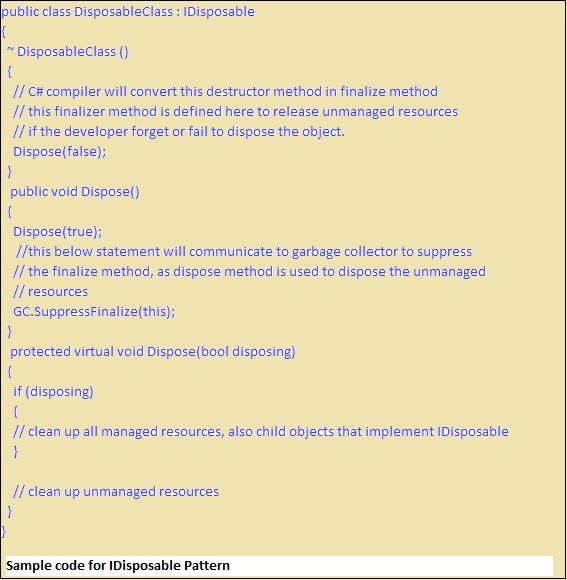
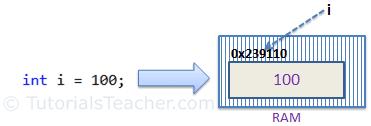
This article about the IDisposable pattern is the continuation of my previous article “[Object LifeTime in .NET Framework"](https://www.c-sharpcorner.com/UploadFile/b08196/object-lifetime-in-net-framework/).  
  
IDisposable is an interface that contains a single method, Dispose(), for releasing unmanaged resources, like files, streams, database connections and so on. This method is implemented explicitly in the code when we need to clean up a disposable object and to release unmanaged resources that this disposable object holds.  
  
The following is an implementation of IDisposable in a class:  
  


All the value types derive from System.ValueType, which in-turn, derives from System.Object. For example, consider integer variable int i = 100;

The system stores 100 in the memory space allocated for the variable i. The following image illustrates how 100 is stored at some hypothetical location in the memory (0x239110) for 'i':



The following data types are all of value type:

* bool
* byte
* char
* decimal
* double
* enum
* float
* int
* long
* sbyte
* short
* struct
* uint
* ulong
* ushort

Passing Value Type Variables

When you pass a value-type variable from one method to another, the system creates a separate copy of a variable in another method. If value got changed in the one method, it wouldn't affect the variable in another method.

Example: Passing Value Type Variables

Copy

static void ChangeValue(int x)

{

x = 200;

Console.WriteLine(x);

}

static void Main(string[] args)

{

int i = 100;

Console.WriteLine(i);

ChangeValue(i);

Console.WriteLine(i);

}

Output:

100  
200  
100

In the above example, variable i in the Main() method remains unchanged even after we pass it to the ChangeValue() method and change it's value there.

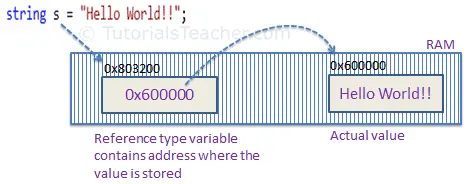
Reference Type

Unlike value types, a reference type doesn't store its value directly. Instead, it stores the address where the value is being stored. In other words, a reference type contains a pointer to another memory location that holds the data.

For example, consider the following string variable:

string s = "Hello World!!";

The following image shows how the system allocates the memory for the above string variable.



As you can see in the above image, the system selects a random location in memory (0x803200) for the variable s. The value of a variable s is 0x600000, which is the memory address of the actual data value. Thus, reference type stores the address of the location where the actual value is stored instead of the value itself.

The followings are reference type data types:

* String
* Arrays (even if their elements are value types)
* Class
* Delegate

### Passing Reference Type Variables

When you pass a reference type variable from one method to another, it doesn't create a new copy; instead, it passes the variable's address. So, If we change the value of a variable in a method, it will also be reflected in the calling method.

Example: Passing Reference Type Variable

Copy

static void ChangeReferenceType(Student std2)

{

std2.StudentName = "Steve";

}

static void Main(string[] args)

{

Student std1 = new Student();

std1.StudentName = "Bill";

ChangeReferenceType(std1);

Console.WriteLine(std1.StudentName);

}

[Try it](https://www.tutorialsteacher.com/codeeditor?cid=cs-WL8Xgk)

Output:

Steve

In the above example, we pass the Student object std1 to the ChangeReferenceType() method. Here, it actually pass the memory address of std1. Thus, when the ChangeReferenceType() method changes StudentName, it is actually changing StudentName of std1 object, because std1 and std2 are both pointing to the same address in memory.

String is a reference type, but it is immutable. It means once we assigned a value, it cannot be changed. If we change a string value, then the compiler creates a new string object in the memory and point a variable to the new memory location. So, passing a string value to a function will create a new variable in the memory, and any change in the value in the function will not be reflected in the original value, as shown below.

Example: Passing String

Copy

static void ChangeReferenceType(string name)

{

name = "Steve";

}

static void Main(string[] args)

{

string name = "Bill";

ChangeReferenceType(name);

Console.WriteLine(name);

}

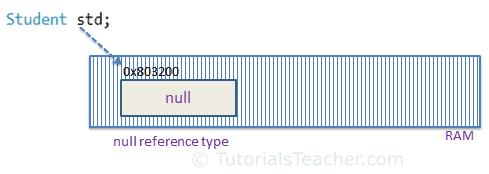
[Try it](https://www.tutorialsteacher.com/codeeditor?cid=cs-g2FG7h)

Output:

Bill

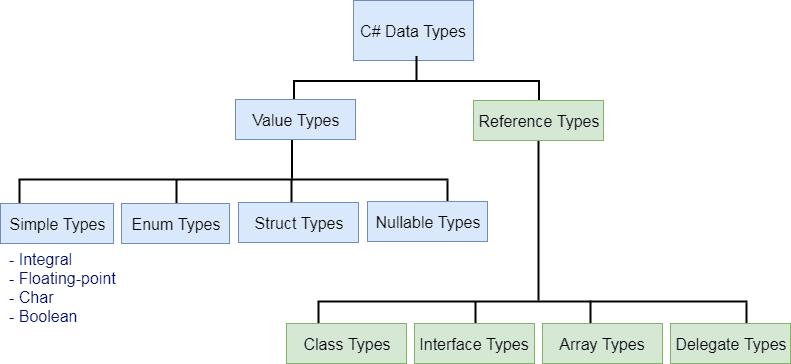
Null

The default value of a reference type variable is null when they are not initialized.Null means not refering to any object.



**A value type variable cannot be null because it holds value, not a memory address. C# 2.0 introduced nullable types, using which you can assign null to a value type variable or declare a value type variable without assigning a value to it.**

*C# is a strongly-typed language.*



C# - Nullable Types

As you know, a value type cannot be assigned a null value. For example, int i = null will give you a compile time error.

C# 2.0 introduced nullable types that allow you to assign null to value type variables. You can declare nullable types using Nullable<t>where T is a type.

Example: Nullable type

Copy

Nullable<int> i = null;

omething like the following structure.

Example: Nullable struct

Copy

[Serializable]

public struct Nullable<T> where T : struct

{

public bool HasValue { get; }

public T Value { get; }

// other implementation

}

A nullable of type *int* is the same as an ordinary *int* plus a flag that says whether the *int* has a value or not (is null or not). All the rest is compiler magic that treats "null" as a valid value.

Example: HasValue

Copy

static void Main(string[] args)

{

Nullable<int> i = null;

if (i.HasValue)

Console.WriteLine(i.Value); // or Console.WriteLine(i)

else

Console.WriteLine("Null");

}

Output:

Null

The HasValue returns **true** if the object has been assigned a value; if it has not been assigned any value or has been assigned a null value, it will return **false**.

Shorthand Syntax for Nullable Types

You can use the '?' operator to shorthand the syntax e.g. int?, long? instead of using Nullable<T>.

Example: Shorthand syntax for Nullable types

Copy

int? i = null;

double? D = null;

?? Operator

Use the '??' operator to assign a nullable type to a non-nullable type.

Example: ?? operator with Nullable Type

Copy

int? i = null;

int j = i ?? 0;

Console.WriteLine(j);

[Try it](https://www.tutorialsteacher.com/codeeditor?cid=cs-4m9elt)

Output:

0

In the above example, i is a nullable int and if you assign it to the non-nullable int j then it will throw a runtime exception if i is null. So to mitigate the risk of an exception, we have used the '??' operator to specify that if i is null then assign 0 to j.

Assignment Rules

In the following example, a nullable of int type is a field of the class, so it will not give any error.

Example: Nullable type as Class Field

Copy

class MyClass

{

public Nullable<int> i;

}

class Program

{

static void Main(string[] args)

{

MyClass mycls = new MyClass();

if(mycls.i == null)

Console.WriteLine("Null");

}

}

[Try it](https://www.tutorialsteacher.com/codeeditor?cid=cs-bYvkTO)

Output:

Null

Characteristics of Nullable Types

1. Nullable types can only be used with value types.
2. The Value property will throw an InvalidOperationException if value is null; otherwise it will return the value.
3. The HasValue property returns true if the variable contains a value, or false if it is null.
4. You can only use == and != operators with a nullable type. For other comparison use the Nullable static class.
5. Nested nullable types are not allowed. Nullable<Nullable<int>> i; will give a compile time error.