C#: Null Conditional and Null Coalescing operators

Developers often face null reference exceptions when they reference null object or forgets to make null checks. In real time software and web applications, null data are more common. Developers working on such software or applications have to make sure that a value (or variable) which is *expecting* a non-null value should in no case gets one. They write validation codes for the same, usually using conditional statements like *if.*

However when it comes to complex, real-time software, consisting of millions of data that range across several database tables- It becomes more time consuming, iterative and cumbersome to validate and make simple null checks and put *if*s everywhere in the code. Such checks sometimes even affect the readability and modularity of the code.

Let’s understand this with an example.

Consider a ‘Users’ table that records following properties for different users, as follows:



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Id | Name | Address | MobileNumber | MonthlyEarning |
| 1 | Peter Parker | SHIELD, New York | 9304023400 | 20000 |
| 2 | Steve Rogers | Avengers Tower, New York | 9720206940 | 50000 |
| 3 | Ash Ketchum | Pallet Town, Japan | 8920493040 | 12000 |
| 4 | Harry Potter | null | 7143250923 | 10000 |
| 5 | Chhota Bheem | Some random Indian village | 8913459200 | 5000 |

Now there is a requirement to view addresses of those users who has monthly earnings more than 5000. In that case, one would write following query in LINQ:



Suppose, instead of *FirstOrDefault,* mistakenly *First* would have used or worse even, the null check was missed here and directly ‘*user.Address’* was returned; the application would break down shouting out “Null Reference Exception: Object Reference Not Set to the instance of an object”.

Solution:

C# offers inbuilt operators to check and handle nulls. These operators, called Null Conditional operators, makes it very easy to handle and validate null values. So if we continue with above example, we can simply write:



Notice, here there are two operators:

1. ?. called **Null Conditional operator**

* It is called Null Conditional operator
* Before accessing member (using dot), it performs the null check.
* In above example, user?.Address evaluates ‘user’ property first and if it turns out to be **not** null, only then it will access Address property. If it turns out to be null, the access is aborted and null is returned.

1. ??

* It is called **Null Coalescing operator**
* Returns right-hand operand if left-hand operand is null and vice versa.
* In above example, left-hand operand is ‘user?.Address’ and right-hand operand is a string. If user?.Address turns out to be null, the string “User Not Found” will be returned, thereby ensuring that no null values are being transferred.

These operators indeed reduce a lot of explicit null checks using ifs. Moreover, they offer better readability to code and keeps the code potentially away from code smells which could arise due to mishandling nulls.

The null coalescing operator is pretty straightforward. It evaluates left-hand side expression and if tends to be null, it returns right-hand side expression. So, one must ensure that at least one of the operand evaluates to a non-null value. What do you think will return if I type: “return user?.Address ?? user?.Address” where the user is null?

Now let’s dive deeper in understanding how Null Conditional operators can be used in various scenarios:

1. Short Circuiting

* We can use ?. operator in the chain of member access where more than one property or member are of nullable types.
* For example:



* Have you thought, how much times you would have to write if statements to verify each nullable type?!

1. Index accessing

* Not only properties, you can also access indexes from arrays or list.
* For example,



* Would you have written a loop and ifs inside it, to check whether each list or array member is null or not and access properties?!

1. Method invocation

* Not only you can access property or index, but also invoke a method using this operator.
* For example, class user has a method ‘register’ which is invoked from another method using object of User class.



* As mentioned above example, a null object ‘a’ of User class was handled by null conditional operator before invoking Register method. Otherwise, one would have to use if statements to check object’s nullability. This scenario often occurs in delegates.
* If you have worked with delegates and events before, you must have known that null checks are often done before invoking delegates to ensure that delegate to be invoked still exists and has not been modified by any other thread.
* This can be done directly using ?. operator.
* For example, someDelegate?.Invoke() will first check whether delegate exists or not and if it does exist, the delegate will be invoked.

Conclusion:

The behavior of data in the natural world is unpredictable, dynamic and heavily random. There can be huge system breakdowns due to mishandling or non-handling of nulls in cases where nulls are not at all required. Till now, In such scenarios, generally nested layers of if statements are added inside iterative loops for checking and handling nulls. This was error prone exercise as failing to place an ‘if’ statement could cause the exception. The null conditional operator can be termed therefore as a better, readable and one line abstraction of such null checks, which if used with the null coalescing operator can handle nullable types and properties effectively.