

## 50. What are nullable reference types?

**Brief summary:** Nullable reference types is a feature introduced with C# 8, that enables explicit declaration of a reference type as nullable or not. The compiler will issue a warning when it recognizes the code in which a non-nullable object has a chance of being null, or when we use nullable reference types without null check, risking the `NullReferenceException`. This feature doesn't change the actual way of executing C# code; it only changes the generated warnings.

*"What are nullable reference types?"* At first glance, you may think this question is silly. After all, **all reference types are nullable** in C#, right? Well, yes, this is true and did not change with introducing the feature called "nullable reference types" with C# 8.

First, let me show you this code:

```
class House
{
    1 reference
    public string OwnerName { get; }
    1 reference
    public Address Address { get; }

    0 references
    public House(string ownerName, Address address)
    {
        OwnerName = ownerName;
        Address = address;
    }
}

3 references
class Address
{
    1 reference
    public string Street { get; }
    1 reference
    public string Number { get; }

    0 references
    public Address(string street, string number)
    {
        Street = street;
        Number = number;
    }
}
```

Let's say we work on an application that manages houses data. We use those two types - House and Address - all around it. Let's see a tiny fragment of this application, but keep in mind that the entire application can be huge.

```
string FormatHousesData(IEnumerable<House> houses)
{
    return string.Join("\n",
        houses.Select(
            house =>
                $"Owner is {house.OwnerName}, " +
                $"address is {house.Address.Number} " +
                $"{{house.Address.Street}}");
}
```

All right. We submit this code for code review and wait for the feedback. Soon after we see a new comment:

*"Looking good, but if house.Address is null, NullReferenceException will be thrown. You can see Visual Studio warning you about that with the green underline".*

It is true. After a short discussion with the reviewer, we make a decision - the owner of the house should never be a null string. Also, the Address can't be null, and both Street and Number can't be null either.

See what happened here: we made a **decision** that in this **particular** case, the **nullable** type that is string and Address, should actually **not be nullable**.

All right. We have work to do. If those properties should not be nullable, we must add some logic to the constructors:

```
public House(string ownerName, Address address)
{
    if(ownerName == null)
    {
        throw new ArgumentNullException(nameof(ownerName));
    }
    if (address == null)
    {
        throw new ArgumentNullException(nameof(address));
    }
    OwnerName = ownerName;
    Address = address;
}
```

```
public Address(string street, string number)
{
    if (street == null)
    {
        throw new ArgumentNullException(nameof(street));
    }
    if (number == null)
    {
        throw new ArgumentNullException(nameof(number));
    }
    Street = street;
    Number = number;
}
```

Great. Now it will simply be impossible to create a House in which the OwnerName is null, nor an Address with null Street or Number. We made them **practically not nullable**, even if as reference types **technically they are nullable**.

The problem is, Visual Studio (before it was updated to support C# 8) was not smart enough to know that we actually ensure that the Address is not null, and it would keep giving us the warning about possible NullReferenceException.

```
string FormatHousesData(IEnumerable<House> houses)
{
    return string.Join("\n",
        houses.Select(
            house =>
                $"Owner is {house.OwnerName}, " +
                $"address is {house.Address.Number} " +
                $"{{house.Address.Street}}"));
}
```

We could ignore those warnings, but it's not really a solution. Other developers, who are not aware that the constructor is enforcing values not to be null, may still be suspicious and will feel more comfortable with filling their code with endless checks for null values.

I guess you probably have seen such code quite often:

```
bool ValidateAddress(House house)
{
    if (house == null)
    {
        Console.WriteLine("house is null");
        return false;
    }
    if (house.Address == null)
    {
        Console.WriteLine("address is null");
        return false;
    }
    if (house.Address.Number == null)
    {
        Console.WriteLine("address/number is null");
        return false;
    }
    if (house.Address.Street == null)
    {
        Console.WriteLine("address/street is null");
        return false;
    }
    if (house.Address.Street == null)
    {
        Console.WriteLine("address/street is null");
        return false;
    }
    if (house.Address.Number.Length == 0)
    {
        Console.WriteLine("address/number is empty");
        return false;
    }
    if (house.Address.Street.Length == 0)
    {
        Console.WriteLine("address/street is empty");
        return false;
    }
    return true;
}
```

Such code sometimes takes a significant part of the entire codebase, making it bulky, hard to read, and unmaintainable.

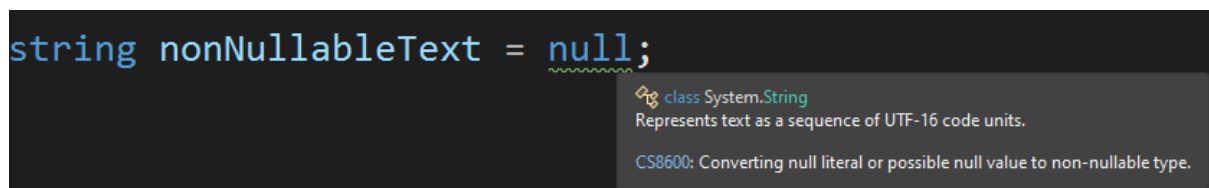
Wouldn't it be just simpler if we could **agree** that the Address and its components can't be null and that we promise to enforce it at the constructor level?

Well, the need for such an agreement was exactly the reason for introducing **nullable reference types**.

This feature gives us the ability to declare a reference type as nullable or not nullable. If a type is declared as **not nullable**, the compiler will give us warnings in any context in which there is a risk that the value may actually be null. If a type **is nullable**, the compiler will give us a warning where a `NullReferenceException` could happen.

With C# 8 and newer, the "old" way of declaring reference types will make them not nullable:

```
string nonNullableText = null;
```



As you can see, the compiler gives us a warning. We declared the variable as a non-nullable string, but we assigned null to it. This doesn't make much sense, hence the compiler warning. We can fix it by declaring the variable as nullable, the same way as we would declare nullable value types - by adding a question mark:

```
string? nullableText = null;
```

Now there is no warning. This variable is a nullable string, so assigning null to it is fine.

A very important note: **nullable reference types feature does not change anything in how the program is executed**. Even non-nullable values will still throw `NullReferenceExceptions` when null. **This feature only changes how compiler warnings are issued**.

All right. Now that we know the essence of nullable reference types, let's take another look at this type:

```

class Address
{
    4 references
    public string Street { get; }
    4 references
    public string Number { get; }

    0 references
    public Address(string street, string number)
    {
        Street = street;
        Number = number;
    }
}

```

There are no compiler warnings here because the constructor parameters are non-nullable strings. But let me change something:

```

class Address
{
    4 references
    public string Street { get; }
    4 references
    public string Number { get; }

    0 references
    public Address(string? street, string number)
    {
        Street = street;
        Number = number;
    }
}

```

I made the street parameter nullable, and now we see warnings. They are here because we assign a nullable parameter to a non-nullable property, which obviously doesn't make much sense, and it may make this seemingly non-nullable property null.

Let me show you one more case:

```

class Address
{
    3 references
    public string Street { get; }
    4 references
    public string Number { get; }

    0 references
    public Address(string number)
    {
        Number = number;
    }
}

```

Now I removed the assignment to the Street property completely. The warning at the constructor makes sense - the Street property should not be null, but it will be because it's not assigned anything.

And this actually answers quite a tricky question - what is the default value for non-nullable reference types? Well, ironically, it's null (because what else could it be?)

The great thing about the nullable reference types feature is that it has good support from Visual Studio and other modern IDEs. Let me show you some code:

```

int GetLength(string? nullableText)
{
    return nullableText.Length;
}

```

Here the warning is expected, because **nullableText** may be null. But let me add a null check:



```
int GetLength(string? nullableText)
{
    if(nullableText == null)
    {
        return 0;
    }
    return nullableText.Length;
}
```

The warning is gone because Visual Studio knows that in the last line of this method we can be sure that the parameter is not null, as this has already been checked.

Please notice that this IDE support is not infallible, and sometimes it can be tricked:

```
var array = new string[10];
Console.WriteLine(array[0].Length);
```

As you can see here I do something silly - I declare an array of non-nullable strings, yet by default, it is filled with nulls. No compiler warning appears, though, even if the "array[0].Length" will throw NullReferenceException.

It doesn't mean this feature is useless. Let me quote Jon Skeet on that: *"Being able to know when things might or might not be null, even when it's only to 90% confidence, is a lot better than 0% confidence."*

If you are curious what else Jon Skeet has to say about nullable reference types, check out his lecture about this feature from GOTO Copenhagen 2019 Conference: <https://www.youtube.com/watch?v=1tpyAQZFLZY>

All right. There are use cases when we actually know better than the compiler if something is or is not null. One of the outstanding examples is when some field is set with the SetUp method in unit tests:

```
class Calculator
{
    1 reference
    public int Add(int a, int b) => a + b;
}

[TestFixture]
0 references
public class CalculatorTests
{
    private Calculator _cut;

    [SetUp]
    0 references
    public void Setup()
    {
        _cut = new Calculator();
    }

    [Test]
    0 references
    public void Add5And10_ShallGive15()
    {
        Assert.AreEqual(15, _cut.Add(10, 5));
    }
}
```

If you don't know NUnit, let me give you a very quick introduction. The method with the **SetUp** attribute will be executed before each test, so it gives us the same guarantee as the constructor, that the **\_cut** (Class Under Test) field will not be null. Yet, the compiler warns us that the **\_cut** field may be null before exiting the constructor. C# compiler doesn't know how NUnit works, so it's not aware this field will never be null when the test is executed.

To get rid of the compiler warning, let's declare this field as nullable:

```

private Calculator? _cut;

[SetUp]
0 references
public void SetUp()
{
    _cut = new Calculator();
}

[Test]
0 references
public void Add5And10_ShallGive15()
{
    Assert.AreEqual(15, _cut.Add(10, 5));
}

```

Ugh... one warning disappeared, but another showed up. Now the compiler warns us that the `_cut` may be null when calling the `Add` method on it. Making the code compliant with the compiler's requirements about the nullable reference types is a bit like playing Whack-A-Mole. One warning disappears, but another pops out.

But in this case, **I know better**. I know the `SetUp` method will be executed first. I want to say "Quiet, compiler! I know it's not null!". And exactly for such situations, the **null-forgiving operator** was introduced:

```

_cut!.Add(10, 5));

```

As you can see, I can simply put "!" after a nullable reference type object which I know is not null to suppress the warning.

It can actually add it even if I know it's null, which is sometimes needed in unit tests. Let me show you an example.

First, let's take another look at the `House` class. Even if the `ownerName` and `address` parameters are non-nullable, I want to perform a null check here. This is a good practice - after all, one still can pass nulls there, because as we said, this

feature doesn't change how the code works - it only issues new warnings. Of course, the person that calls this constructor will see a warning when passing null as a parameter, but what if he or she will ignore it? After the House object is created, we want to be sure that we can really trust what was declared - that the OwnerName and Address properties are not null. That's why it's best to enforce it once and for all in the constructor.

```
class House
{
    2 references
    public string OwnerName { get; }
    8 references
    public Address Address { get; }

    1 reference
    public House(string ownerName, Address address)
    {
        if(ownerName == null)
        {
            throw new ArgumentNullException(nameof(ownerName));
        }
        if (address == null)
        {
            throw new ArgumentNullException(nameof(address));
        }
        OwnerName = ownerName;
        Address = address;
    }
}
```

This looks good. After this validation, we can be sure the OwnerName and Address will not be null anywhere in our code. We can forget about the neverending null-checks. The only thing left to do is to add unit tests for this constructor:

```
[TestFixture]
0 references
public class HouseTests
{
    [Test]
    0 references
    public void NullOwnerName_ShallThrowException()
    {
        Assert.Throws<ArgumentNullException>(() =>
            new House(null, new Address("Maple Street", "12B")));
    }
}
```

This test checks if an `ArgumentNullException` will be thrown if `OwnerName` is null. But even in this test scenario, the compiler gives me a warning. But I know what I'm doing. I want this null here, so I kindly ask the compiler to give me a break:

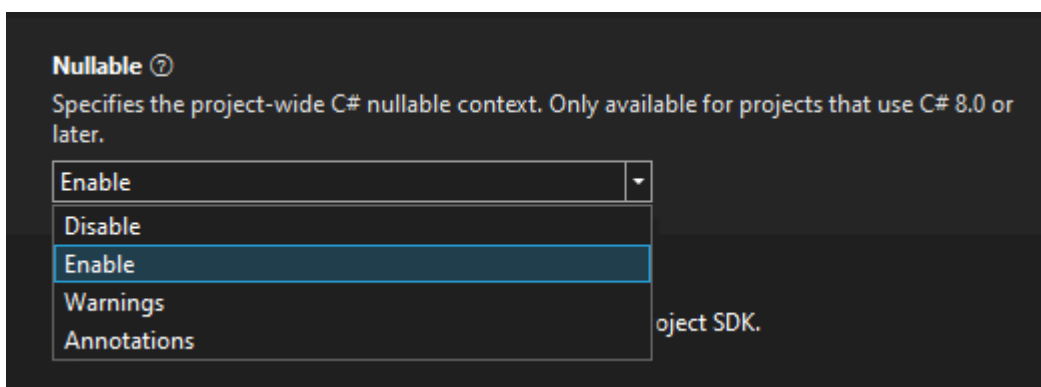
```
Assert.Throws<ArgumentNullException>(() =>
    new House(null!, new Address("Maple Street", "12B")));
```

Again, the null-forgiving operator proved to be useful.

Now we understand how the nullable reference types work. The question that remains is **when should we use them**.

Well, my advice is this: think about the types in your code, their fields, and properties, as well as local variables and parameters. Can they ever be null, or do you always ensure that they are not? If they can be null, make them nullable explicitly. This will clearly show what your intention was, and anyone working with your code will know that this thing may be null and needs to be checked for it.

Also, a word of caution about migration. Since the "old" type declarations are made non-nullable starting with C# 8, it may mean that after updating your .NET and C# version you will suddenly get an overwhelming number of warnings. Don't worry - they are a good thing and will help you migrate to this new feature. But if you really don't want to see them, you can disable this feature in project properties:



The migration process itself can be a bit tiring (remember the Whack-A-Mole metaphor?) but having nullable reference types can really save you a lot of pain, errors and null checks, making the code cleaner, more expressive, and easier to maintain.

Even if you don't decide to introduce this feature in an existing project due to complex migration, I highly recommend using it in new code. You can disable or

enable this feature per file or even a code fragment. This way, you can improve your code step by step, and not drown in an ocean of warnings.

```
#nullable disable
string text = null;
Console.WriteLine(text);
#nullable enable
```

As you can see there is no warning here, even if we assign null to a non-nullable string.

Let's summarize. Nullable reference types is a feature introduced with C# 8, that enables explicit declaration of a reference type as nullable or not. The compiler will issue a warning when it recognizes the code in which a non-nullable object has a chance of being null, or when we use nullable reference types without null check, risking the `NullReferenceException`. This feature doesn't change the actual way of executing C# code; it only changes the generated warnings.

#### **Bonus questions:**

- **"What is the default value of non-nullable reference types?"**  
*It is null.*
- **"What is the purpose of the null-forgiving operator?"**  
*It allows us to suppress a compiler warning related to nullability.*
- **"Is it possible to enable or disable compiler warnings related to nullable reference types on the file level? If so, how to do it?"**  
*It is possible. We can do it by using **#nullable enable** and **#nullable disable** preprocessor directives.*