# Nullable Types in C#

In C#, nullable types allow you to assign `null` to value types (such as `int`, `double`, `bool`, etc.), which are non-nullable by default. This feature is particularly useful for representing the absence of a value, typically in scenarios involving databases, user inputs, or any optional data.

## Declaring Nullable Types

To declare a nullable type, you use the `?` operator after the value type. For example:
```csharp
int? nullableInt = null;
double? nullableDouble = null;
bool? nullableBool = null;
```
Here, `nullableInt`, `nullableDouble`, and `nullableBool` can hold either a value of their respective types or `null`.

## Working with Nullable Types

### Checking for `null`
Before using a nullable type, it's essential to check if it contains a value:
```csharp
if (nullableInt.HasValue)
{
    Console.WriteLine($"Value: {nullableInt.Value}");
}
else
{
    Console.WriteLine("No value present");
}
```
Alternatively, you can use the `Value` property directly if you are sure it is not `null`:
```csharp
int nonNullableInt = nullableInt.Value;
```

### Using the `??` Operator
The `??` operator, known as the null-coalescing operator, provides a way to specify a default value if the nullable type is `null`:
```csharp
int defaultInt = nullableInt ?? 0;
```
Here, `defaultInt` will be assigned the value of `nullableInt` if it is not `null`; otherwise, it will be assigned `0`.

### Nullable Value Type Members
Nullable types also support several methods and properties:
- `HasValue`: Returns `true` if the nullable type contains a non-null value.
- `Value`: Gets the value if `HasValue` is `true`; otherwise, it throws an `InvalidOperationException`.
- `GetValueOrDefault()`: Returns the value if `HasValue` is `true`; otherwise, it returns the default value of the underlying type.
- `GetValueOrDefault(T defaultValue)`: Returns the value if `HasValue` is `true`; otherwise, it returns the specified default value.

### Example
Here's a more comprehensive example that demonstrates these concepts:
```csharp
int? nullableInt = 5;
int? nullableInt2 = null;

Console.WriteLine($"Has Value: {nullableInt.HasValue}"); // True
Console.WriteLine($"Value: {nullableInt.Value}");       // 5
Console.WriteLine($"Get Value Or Default: {nullableInt.GetValueOrDefault()}"); // 5
Console.WriteLine($"Get Value Or Default(10): {nullableInt.GetValueOrDefault(10)}"); // 5

Console.WriteLine($"Has Value: {nullableInt2.HasValue}"); // False
Console.WriteLine($"Get Value Or Default: {nullableInt2.GetValueOrDefault()}"); // 0
Console.WriteLine($"Get Value Or Default(10): {nullableInt2.GetValueOrDefault(10)}"); //
10

int defaultInt = nullableInt2 ?? 100;
Console.WriteLine($"Default Int: {defaultInt}"); // 100
```

## Nullable Reference Types (C# 8.0 and later)
Starting with C# 8.0, nullable reference types were introduced to enhance null safety. By default, reference types are non-nullable, and the compiler issues warnings if they are assigned `null` without proper checks.
To enable nullable reference types, you can add the following directive at the top of your file or in your project settings:
```csharp
#nullable enable
```
This change allows you to explicitly declare a reference type as nullable:
```csharp
string? nullableString = null;
```
You can disable nullable reference types for specific code regions using `#nullable disable`.

## Summary

Nullable types in C# provide a robust mechanism to handle the absence of values for both value types and reference types (starting with C# 8.0). They are essential for writing safer and more predictable code, especially when dealing with optional data.