In C#, the `nullable` keyword refers to a type that can either have a value or be `null`. It's denoted by appending `?` to the type.

For example:

- `string?` represents a nullable string type, meaning it can be either a string value or `null`.
- `int?` represents a nullable integer type, meaning it can be either an integer value or `null`.

The concept of nullable types was introduced to handle scenarios where a variable might not have a value assigned to it. Without nullable types, attempting to access a non-nullable variable that hasn't been assigned a value would result in a `NullReferenceException`.

Here's an example to illustrate the difference between nullable and non-nullable types:

```csharp
// Non-nullable string
string nonNullableString = null; // Compile-time error

// Nullable string
string? nullableString = null; // Allowed
```

In the first case, assigning `null` to a non-nullable `string` variable causes a compile-time error because the compiler knows that a non-nullable string can never be `null`.

In the second case, assigning `null` to a nullable `string?` variable is allowed because the type explicitly allows the value to be `null`.

Nullable types are particularly useful when working with databases, APIs, or any scenario where data might be missing or unavailable. They help prevent `NullReferenceExceptions` at runtime by allowing you to handle null values gracefully.

When working with nullable types, you need to perform null checks or use the null-conditional operator (`?.`) to safely access members or perform operations on the nullable value.

Overall, nullable types in C# provide a way to represent and handle the absence of a value in a type-safe manner, enabling better control over null values and reducing the risk of `NullReferenceExceptions`.

In C# 9 and later versions, the `init` keyword is used as an access modifier for property initializers. It allows you to initialize a property during object creation or constructor execution, but prevents the property from being modified after that initial assignment.

When you declare a property like this:

```csharp
public string? WebRootPath { get; init; }
```

It means that:

1. The property `WebRootPath` is a public property of type `string?` (nullable string).
2. The `get` accessor allows you to read the value of the property.
3. The `init` accessor allows you to set the value of the property only during object initialization or in the constructor. After that, the property value is immutable and cannot be changed.

Here's an example to illustrate how it works:

```csharp
public class MyClass
{
    public string? WebRootPath { get; init; }

    public MyClass(string? webRootPath)
    {
        WebRootPath = webRootPath; // This is allowed
    }

    public void SomeMethod()
    {
        WebRootPath = "/new/path"; // This will cause a compile-time error
    }
}
```

In the above example:

- The `WebRootPath` property can be initialized in the constructor.
- Attempting to modify the `WebRootPath` property outside of the constructor (e.g., in the `SomeMethod`) will result in a compile-time error.

The `init` accessor is useful when you want to ensure that a property can only be set during object initialization or construction, and remains immutable thereafter. This helps enforce immutability and can make your code more robust and easier to reason about.