Structs

A structure type (or struct type) is a value type that can encapsulate data and related functionality. For more information you can take a look at the documentation

```
public struct Point
{
    public int X;
    public int Y;

    public Point(int x, int y)
    {
        X = x;
        Y = y;
    }

    public void Display()
    {
        Console.WriteLine($"Point: ({X}, {Y})");
    }
}
```

Declare Point struct

```
public class Program
{
    public static void Main()
    {
        Point p1 = new Point(10, 20);
        Point p2 = p1; // p2 gets a copy of p1's values

        p1.Display(); // Output: Point: (10, 20)
        p2.Display(); // Output: Point: (10, 20)

        p1.X = 30;

        p1.Display(); // Output: Point: (30, 20)
        p2.Display(); // Output: Point: (10, 20)
    }
}
```

Use Struct

Characteristics of structs

Stack allocation: Structs are typically allocated on the stack rather than the heap. This allows for faster memory allocation and deallocation since stack operations are generally more efficient than heap operations.

Lightweight: Structs are usually smaller in size compared to classes.

Value semantics: Structs exhibit value semantics, meaning that they are compared and assigned based on their actual values rather than references.

Limited inheritance: structs cannot directly inherit from other structs or classes. However, they can implement interfaces to provide a level of abstraction and code reuse.

Default constructor: Structs automatically have a default parameterless (this is allowed for the latest versions of C# later than v10)

When to use a struct?

Consider using a struct if you need the following characteristics: (more information in the documentation)

When you need stack allocation: Structs are allocated on the stack, which can be beneficial in certain situations where you want to avoid heap allocation and the associated overhead.

When you want value semantics: Structs have value semantics, meaning they are copied by value when assigned or passed as parameters.

When you want to represent immutable data: Structs are inherently immutable if they only contain read-only fields or properties.

When performance is a critical concern: In some performance-sensitive scenarios, using structs can provide performance benefits due to their stack allocation and value semantics.

When it will not have to be boxed frequently.

How to achieve Inmutability

Benefits of Immutability in Practice:

- Simplified Debugging: Immutable objects make it easier to trace and debug code, as their state does not change unexpectedly.
- Enhanced Code Maintainability: Immutable objects encourage a functional programming style, leading to more modular and maintainable code.
- Improved Testing: Immutable objects simplify unit testing by eliminating the need to set up and modify complex object states.

Benefits of Inmutability

- Thread Safety: Immutable objects can be safely shared between multiple threads without the risk of concurrent modifications.
- Predictability: Immutability ensures that the state of an object remains consistent and avoids unexpected changes.
- Performance: Immutable objects can be cached and reused, eliminating the need for frequent object creation.

Records

```
public record Person(string Name, int Age);
```

Record declaration

C# 9 introduced a new feature called "records" that provides a concise and convenient way to define immutable data types. Records are similar to classes but come with built-in features for value-based equality, immutability, and value semantics.

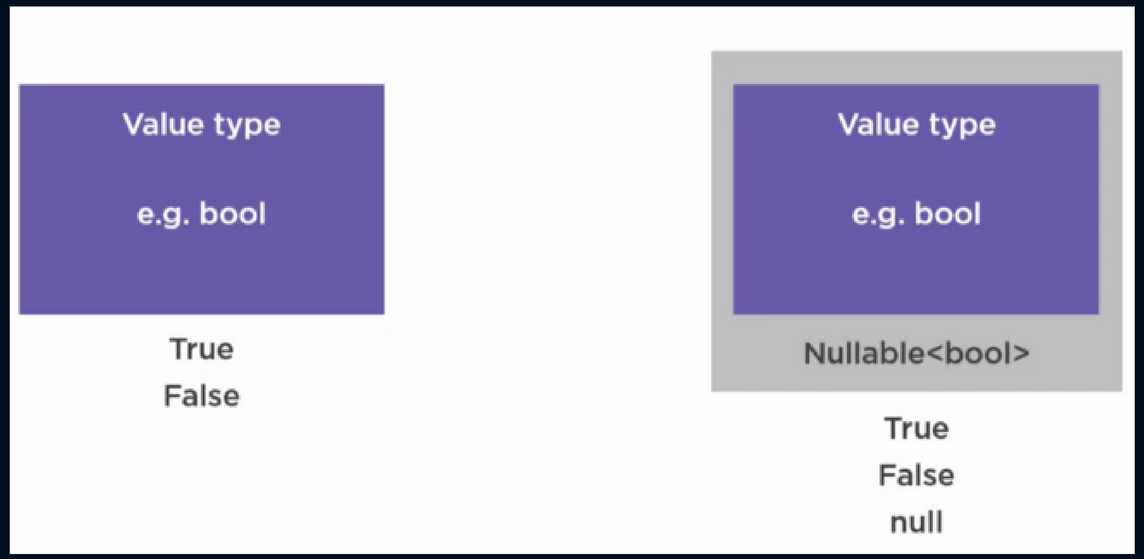
Some additional features, besides the already indicated, are: inheritance and deconstruction.
For more information check the documentation

```
public class Person : IEquatable<Person>
{
    public string Name { get; init; }
    public int Age { get; init; }

    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }

    public override bool Equals(object obj)
    {
        return Equals(obj as Person);
    }

    public bool Equals(Person other)
    {
        return other ≠ null &&
               Name == other.Name &&
               Age == other.Age;
    }

    public override int GetHashCode()
    {
        return HashCode.Combine(Name, Age);
    }

    public static bool operator ==(Person left, Person right)
    {
        return EqualityComparer<Person>.Default.Equals(left,
right);
    }

    public static bool operator ≠(Person left, Person right)
    {
        return !(left == right);
    }
}
```

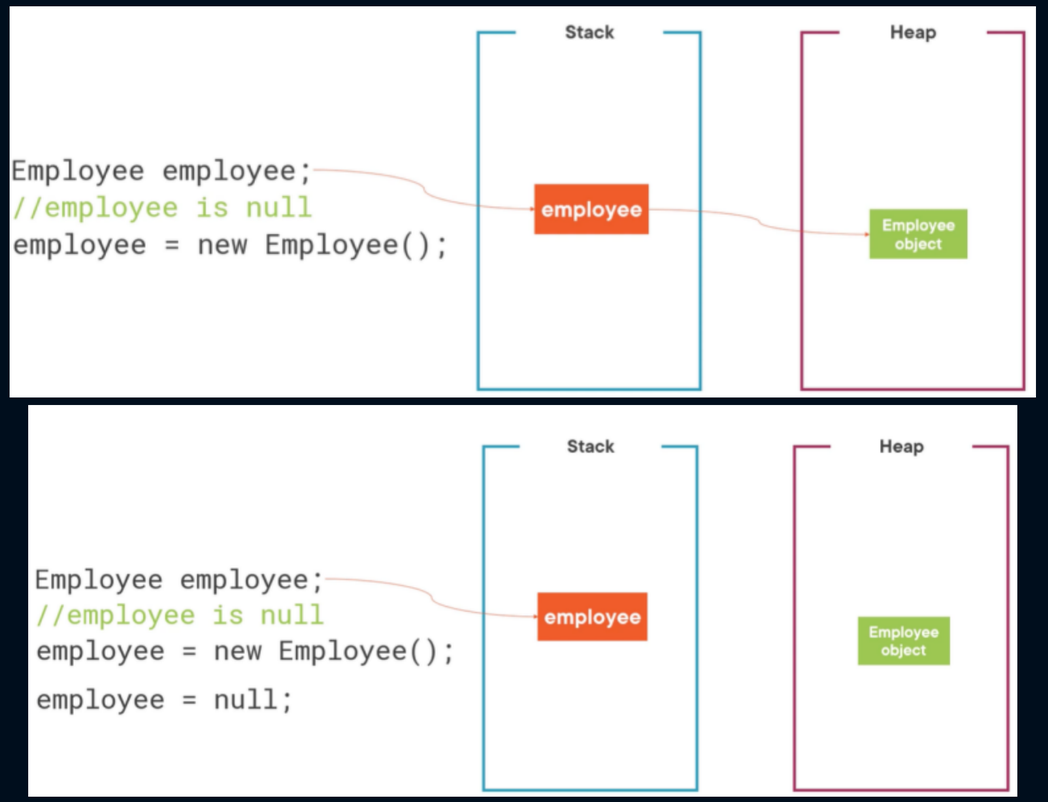
Behind the scenes implementation

What is a nullable type?

Instances of the System.Nullable struct. A nullable type can represent the correct range of values for its underlying value type, plus and additional null value.



How Nullable works



Using nullable type

Nullable methods

```
class PlayerCharacter
{
  public Nullable<int> DaysSinceLastLogin { get; set; }

  public PlayerCharacter()
  {
      DaysSinceLastLogin = null;
  }
}

// Using shorthand
class PlayerCharacter
{
  public int? DaysSinceLastLogin { get; set; }

  public PlayerCharacter()
  {
      DaysSinceLastLogin = null;
  }
}
```

Create a nullable property

```
string Name = null;

// Check if name is null
if(name is null){ ... }
if(string.IsNullOrEmpty(name)){ ... }
if(string.IsNullOrWhiteSpace(name)){ ... }
```

Null verification for strings

```
.HasValue // false if null, true if valid value
.Value // gets underlying value
.GetValueOrDefault() // underlying value or default
.GetValueOrDefault(default) // value or specified default
```

Nullabe methods

Null checking and forgiving operators

Conditional Operator
?:

It is used to evaluate a condition and execute one statement or another.

Null Coalescing Operator
??

It is used to assign a default value if a nullable type is null.

Null Conditional Operator
?. ?[

Helps to avoid multiple conditional operators

Null Forgiving Operator
!

Can be used to tell the compiler to ignore nullability warnings for a particular reference type.

Some examples

Check if a nullable type has a value

```
int? i = null;

if (i != null)
{
    Console.WriteLine("i is not null");
}
else
{
    Console.WriteLine("i is null");
}
```

Using **!=** operator

Which is the same as:

```
if (i.HasValue)
{
    Console.WriteLine("i is not null");
}
else
{
    Console.WriteLine("i is null");
}
```

Using HasValue method

Default value asignation

```
int j = i ?? 0;
int j = i.GetValueOrDefault(0);
int j = i.HasValue ? i.Value : 0;
```

Using null coalescing operator **??**

Using GetValueOrDefault

Using the conditional operator **?:**

Members of the Nullable struct

```
.HasValue // false if null, true if valid value
.Value // gets underlying value
.GetValueOrDefault() // underlying value or default
.GetValueOrDefault(default) // value or specified default
```

Data types default values

Use the default operator to produce the default value of a type, as the following example shows:

Type	Default value
Any reference type	`null`
Any built-in integral numeric type	0 (zero)
Any built-in floating-point numeric type	0 (zero)
bool	`false`
char	`'\0'` (U+0000)
enum	The value produced by the expression `(E)0`, where `E` is the enum identifier.
struct	The value produced by setting all value-type fields to their default values and all reference-type fields to `null`.
Any nullable value type	An instance for which the HasValue property is `false` and the Value property is undefined. That default value is also known as the *null* value of a nullable value type.

Default values

Default values of C# types

```
// this will use the default value
int b;
// or use the default keyword
int a = default;
```

Use the default keyword

Generics and nullable

```
Message? nullMessage = null;
Message nonNullMessage = new Message();

List<Message> m1 = new List<Message>();
m1.Add(nullMessage); // Warning: Possible null reference
m1.Add(nonNullMessage);

List<Message?> m2 = new List<Message?>();
m2.Add(nullMessage);
m2.Add(nonNullMessage);
```

Using nullable with generics

```
public static void LogNullable<T>(T value) where T : class?
{...}

LogNullable(nullMessage);
LogNullable(nonNullMessage);

LogNullable(nullDate);
// Error 'DateTime?' must be a reference type
```

Nullable constraint

```
public static void LogNonNullable<T>(T value) where T : class
{...}
LogNonNullable(nullMessage);
// Message? cannot be used as type parameter because it
doesn't match 'class' constraint

LogNonNullable(nonNullMessage);

LogNonNullable(nullDate);
// Error DateTime? must be a reference type in order to use
it as parameter.
```

Non nullable constraint