# Chapter 8: Null-conditional Operators

## Section 8.1: Null-Conditional Operator

The `?.` operator is syntactic sugar to avoid verbose null checks. It's also known as the Safe navigation operator.

Class used in the following example:

```
public class Person
{
    public int Age { get; set; }
    public string Name { get; set; }
    public Person Spouse { get; set; }
}
```

If an object is potentially null (such as a function that returns a reference type) the object must first be checked for null to prevent a possible `NullReferenceException`. Without the null-conditional operator, this would look like:

```
Person person = GetPerson();

int? age = null;
if (person != null)
    age = person.Age;
```

The same example using the null-conditional operator:

```
Person person = GetPerson();

var age = person?.Age;    // 'age' will be of type 'int?', even if 'person' is not null
```

**Chaining the Operator**

The null-conditional operator can be combined on the members and sub-members of an object.

```
// Will be null if either `person` or `person.Spouse` are null
int? spouseAge = person?.Spouse?.Age;
```

**Combining with the Null-Coalescing Operator**

The null-conditional operator can be combined with the null-coalescing operator to provide a default value:

```
// spouseDisplayName will be "N/A" if person, Spouse, or Name is null
var spouseDisplayName = person?.Spouse?.Name ?? "N/A";
```

## Section 8.2: The Null-Conditional Index

Similarly to the `?.` operator, the null-conditional index operator checks for null values when indexing into a collection that may be null.

```
string item = collection?[index];
```

is syntactic sugar for

```
string item = null;
```

```
if(collection != null)
{
    item = collection[index];
}
```

## Section 8.3: Avoiding NullReferenceExceptions

```
var person = new Person
{
    Address = null;
};

var city = person.Address.City; //throws a NullReferenceException
var nullableCity = person.Address?.City; //returns the value of null
```

This effect can be chained together:

```
var person = new Person
{
    Address = new Address
    {
        State = new State
        {
            Country = null
        }
    }
};

// this will always return a value of at least "null" to be stored instead
// of throwing a NullReferenceException
var countryName = person?.Address?.State?.Country?.Name;
```

## Section 8.4: Null-conditional Operator can be used with Extension Method

Extension Method can work on null references, but you can use `?.` to null-check anyway.

```
public class Person
{
    public string Name {get; set;}
}

public static class PersonExtensions
{
    public static int GetNameLength(this Person person)
    {
        return person == null ? -1 : person.Name.Length;
    }
}
```

Normally, the method will be triggered for `null` references, and return -1:

```
Person person = null;
int nameLength = person.GetNameLength(); // returns -1
```

Using `?.` the method will not be triggered for `null` references, and the type is `int?`:

```
Person person = null;
int? nameLength = person?.GetNameLength(); // nameLength is null.
```

This behavior is actually expected from the way in which the `?.` operator works: it will avoid making instance method calls for null instances, in order to avoid `NullReferenceExceptions`. However, the same logic applies to the extension method, despite the difference on how the method is declared.

For more information on why the extension method is called in the first example, please see the Extension methods - null checking documentation.