C# - Ultimate Guide - Beginner to Advanced | Master class

Section 18 – Extension Methods and Pattern Matching

Extension Methods

Extension method is a method injected (added) into an existing class (or struct or interface), without modifying the source code of that class (or struct or interface).

Existing Class

```
class ClassName
{
}
```

Static Class for Extension Method

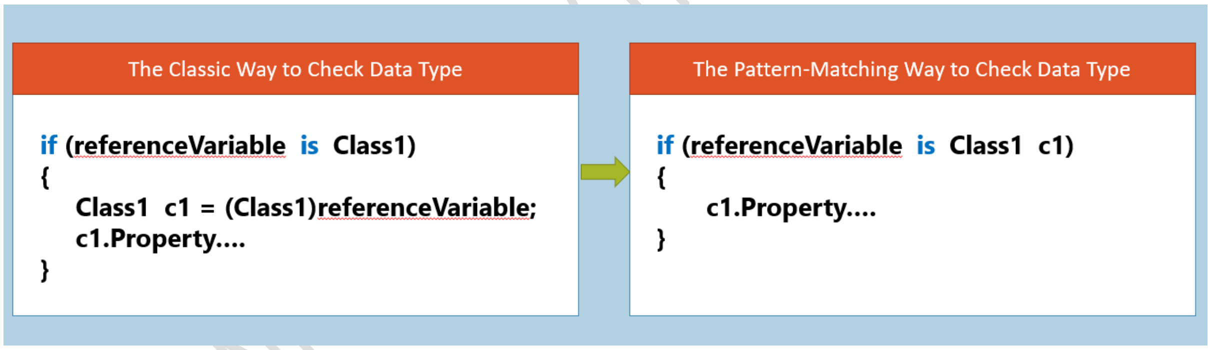
```
static class ClassName
{
  public static ReturnType MethodName(this ClassName ParameterName, …)
  {
    method body here
  }
}
```

- The developer of ClassLibrary, creates a class with a set of methods. The consumer of ClassLibrary, can add additional methods to the same class, without modifying the source code of the ClassLibrary.
- You can add additional methods to pre-defined classes / structures such as String, Int32, Console etc.
- You must create a static class with a static method; that it will be added as a non-static method to the specified class.
- This feature is introduced in C# 3.0.
- The first parameter of extension must be having "this" keyword; followed by the class name / structure name, to which you want to add the extension method. Eg: this ClassName parameter

- The parameter (with 'this' keyword) represents the current object, just like "this" keyword in the instance methods.
- Extension method can have any no. of additional parameters, where the "this" keyword parameter is must.
- Extension method does not support method overriding. That means, extension method's signature can't be same as any existing method.
- You can also add extension methods to sealed class.
- 'Extension Methods' concept can't be used to create fields, properties, or events.
- The static class of extension method can't be inner class.
- The namespace in which the static class of extension method is created, must be imported in order to call the extension method as non-static method.

Pattern Matching

It allows you to declare a variable, while checking the data type (class) of a reference variable, and automatically type-casts the reference variable into the specified data type (class).


The Classic Way to Check Data Type
The Pattern-Matching Way to Check Data Type

```
if (referenceVariable is Class1)
{
    Class1 c1 = (Class1)referenceVariable;
    c1.Property....
}
```

```
if (referenceVariable is Class1 c1)
{
    c1.Property....
}
```

Advantage: Simplified syntax to perform multiple checks of data types and type-casts.

Implicitly-Typed Variables

The variables that are declared with 'var' keyword are called as 'implicitly-typed variables' (a.k.a type-inference).

Implicitly-typed variables are declared without specifying the 'type' explicitly; so that the C# compiler automatically identifies the appropriate data type at compilation-time, based on the value assigned at the time of declaration.

Syntax: var variableName = value;

While declaration, the 'type' of implicitly-typed variables is fixed. It is not possible to change the type of that variable or assign "other type of values" into the implicitly typed variables, after declaration.

Implicitly Typed Variables can only be "local variables"; can't be used for method parameters, return type or fields.

Implicitly Typed Variables must be initialized along with declaration.

It is not possible to declare multiple implicitly typed variables in the same statement. Eg: var x = 10, y = 20; //error

It is not possible to assign "null" into implicitly typed variables (while declaration). Eg: var x = null; //error

Dynamically-Typed Variables

Dynamically Typed Variables are the variables that are declared with 'dynamic' keyword.

Declared without specifying the type explicitly.

There is no fixed type for the variable.

You can assign any type of value to these variables.

C# compiler skips "type-checking" at compilation time; instead, it resolves the data types of its values, at run time.

Syntax: dynamic variableName = value;

The "dynamic" type variables are converted as "object" type in most cases. **Eg:** dynamic dynamicVariable = 100; -> object dynamicaVariable = 100;

The Dynamically Typed Variable can change its data type, any no. of times, at run time.

Methods and other members of 'dynamically typed variables' will not be checked by the compiler at compilation time; will be checked by CLR at run time.

If the method or other member not available, it would not cause compile-time error; it raises run-time error, when the execution flow encountered that particular statement. **Eg:** dynamicVariable.NonExistingMethod( ); //run-time error (exception)

The Dynamically Typed Variables need not be initialized, while declaration.

The Dynamically Typed Variable doesn't have "Intellisense" in Visual Studio.

The "dynamic" keyword is allowed for local variables, method parameters, fields, properties, return types etc.

Inner Classes

"Inner Class" (a.k.a. Nested Class) is a class, which is created in another class (outer-class or containing-class).

Syntax:

```
class ClassName
{
  class InnerClassName
  {
    Members here
  }
}
```

Advantage: We can create all inter-related classes of a class, "inner classes".

Syntax to access inner classes: OuterClassName.InnerClassName

By default, inner class is "private"; so it is accessible within the same outer class. To make it available to outside of the outer class, you can use other access modifiers such as "protected", "private protected", "internal", "protected internal" or "public".

A nested class can be declared as a private (default), public, protected, internal, protected internal, or private protected.

Outer class can't access the members of inner class directly, without object.

Inner class can't access the members of outer class directly, without object.

You are allowed to create objects of inner class in outer class; and vice versa; but you can't do both; if you create objects vice-versa, it causes StackOverflowException.

You can create a child class for the inner class, outside the outer class.