String Interpolation:

WriteLine($"{planet} is actually named after {name}");

string name = "Sammy Jenkins";

double salary = 1000;

WriteLine($"{name}'s monthly salary is {salary:C2}");

WriteLine($"Man! This {name} is kind of a

{(salary >= 1000 ? "rich guy" : "poor guy")}");

*/\*Output:*

*Sammy Jenkins's monthly salary is $1000.00*

*Man! This Sammy Jenkins is kind of a rich guy*

*\*/*

Dictionary Initializers

*/\* The new and friendly syntax \*/*

            Dictionary<string, string=""> alien = new Dictionary<string, string="">()

            {

                ["Name"] = "Fizzy",

                ["Planet"] = "Kepler-452b"

            };

Auto-Property Initializers

  public class Employee

        {

            public string Name { get; set; }

            public decimal Salary { get; set; }

            public Employee()

            {

*/\* Initializing property through constructor \*/*

            Name = "Sammy Jenkins";

                Salary = 10000;

            }

        }

public class Employee

        {

*/\* Getter only property with inline initialization \*/*

            public string Name { get; } = "Sammy Jenkins"

*/\* Property with inline initialization \*/*

            public decimal Salary { get; set; } = 10000;

        }

nameof expression:

  private static void CallSomething()

        {

            int? x = null;

            if (x == null)

            {

                throw new Exception("x is null");

*/\* x is the type name. What if someone changes the*

*type name from x to i? The exception below would be inappropriate. \*/*

            }

        }

  private static void CallSomething()

        {

            int? number = null;

            if (number == null)

            {

                throw new Exception(nameof(number) + " is null");

            }

        }

Await in catch/finally Block:

catch (Exception exception)

            {

                try

                {

*/\* If the first request throws an exception,*

*this request will be executed.*

*Both are asynchronous request to a weather service\*/*

                    var result = await client.GetStringAsync

                    ("http://api.openweathermap.org/data/2.5/weather?q=NewYork,us");

                    WriteLine(result);

                }

                catch (Exception)

                {

                    throw;

                }

            }

Null Conditional Operator & Null Propagation

**Again, we have this new notion of null conditional operator where you can remove declaring a conditional branch to check to see if an instance of an object is null or not with this new ?. ?? null conditional operator syntax. The ?. is used to check if an instance is null or not, if it's not null then execute the code after ?. but if it is not, then execute code after ??. Check out the example below:**

*/\* old syntax of checking if an instance is null or not \*/*

            WriteLine(hero != null ? hero.SuperPower : "You aint a super hero.");

            ReadLine();

*/\* New null conditional operator \*/*

            WriteLine(hero?.SuperPower ?? "You aint a super hero.");

            ReadLine();

private static void Main(string[] args)

        {

            List<superhero> superHeroes = null;

            SuperHero hero = new SuperHero();

            if (hero.SuperPower != String.Empty)

            {

                superHeroes = new List<superhero>();

                superHeroes.Add(hero);

            }

            WriteLine(superHeroes?[0].SuperPower ?? "There is no such thing as super heros.");

            ReadLine();

        }

Expression Bodied Function & Property

*/\* Expression bodied function \*/*

        private static double AddNumbers(double x, double y) => x + y;

        public class Person

        {

            public string FirstName { get; } = "Fiyaz";

            public string LastName { get; } = "Hasan";

*/\* Expression bodied computed property \*/*

            public string FullName => FirstName + " " + LastName;

        }

Static Using with Extension Methods

class Program

    {

        private static void Main(string[] args)

        {

            Shape shape = new Shape();

            GenerateRandomSides(shape);

*/\* You can write WriteLine(ShapeUtility.IsPolygon(shape));.*

*But here I'm executing extension method on shape type,*

*that's why they are called extension methods*

*since there are just a extension of your type. duh! \*/*

            WriteLine(shape.IsPolygon());

            ReadLine();

        }

    }

    public class Shape

    {

        public int Sides { get; set; }

    }

    public static class ShapeUtility

    {

        public static bool IsPolygon(this Shape shape)

        {

            return shape.Sides >= 3;

        }

        public static void GenerateRandomSides(Shape shape)

        {

            Random random = new Random();

            shape.Sides = random.Next(1, 6);

        }

    }

Exception Filtering

Exception filtering is nothing but some condition attached to the catch block. Execution of the catch block depends on this condition. Let me give you a simple example.

In C# 5, we would have done something like the code given below to show users appropriate messages depending on the randomly generated http status codes.

Hide   Shrink    Copy Code

using System;

using static System.Console;

namespace NewInCSharp

{

    class Program

    {

        private static void Main(string[] args)

        {

            Random random = new Random();

            var randomExceptions = random.Next(400, 405);

            WriteLine("Generated exception: " + randomExceptions);

            Write("Exception type: ");

            try

            {

                throw new Exception(randomExceptions.ToString());

            }

            catch (Exception ex)

            {

                if(ex.Message.Equals("400"))

                    Write("Bad Request");

                else if (ex.Message.Equals("401"))

                    Write("Unauthorized");

                else if (ex.Message.Equals("402"))

                    Write("Payment Required");

                else if (ex.Message.Equals("403"))

                    Write("Forbidden");

                else if (ex.Message.Equals("404"))

                    Write("Not Found");

            }

            ReadLine();

        }

    }

}

Here, I'm randomly generating an exception code randomExceptions. Inside the catch block, I'm showing the appropriate error message respective to the exception code.

Now you can achieve the same thing using exception filtering but the syntax is a bit different. Rather than entering the catch block and checking to see which condition met our exception, we can now decide if we even want to enter the specific catch block. Here is the code:

Hide   Shrink    Copy Code

using System;

using static System.Console;

namespace NewInCSharp

{

    class Program

    {

        private static void Main(string[] args)

        {

            Random random = new Random();

            var randomExceptions = random.Next(400, 405);

            WriteLine("Generated exception: " + randomExceptions);

            Write("Exception type: ");

            try

            {

                throw new Exception(randomExceptions.ToString());

            }

            catch (Exception ex) when (ex.Message.Equals("400"))

            {

                Write("Bad Request");

                ReadLine();

            }

            catch (Exception ex) when (ex.Message.Equals("401"))

            {

                Write("Unauthorized");

                ReadLine();

            }

            catch (Exception ex) when (ex.Message.Equals("402"))

            {

                Write("Payment Required");

                ReadLine();

            }

            catch (Exception ex) when (ex.Message.Equals("403"))

            {

                Write("Forbidden");

                ReadLine();

            }

            catch (Exception ex) when (ex.Message.Equals("404"))

            {

                Write("Not Found");

                ReadLine();

            }

        }

    }

}

So, what's the main difference between these two. Well when you enter a catch block, the current execution state is lost. So, the actual cause of the exception is really hard to find. But in the exception filtering, we stay where we should be staying, i.e., current execution state. This means the stack stays unharmed.

So, exception filtering is good, right? Well there is a catch! Since in the exception filtering, entering a catch block depends on the filter applied on the catch, making a silly mistake can change the whole meaning of the exception. This may actually happen cause the filtering depends on a boolean result and this boolean result can be sent from any code block, making the exception have a different meaning. For example:

Hide   Shrink    Copy Code

using System;

using System.Net.Http;

using System.Threading.Tasks;

using static System.Console;

namespace NewInCSharp

{

    class Program

    {

       private static void Main(string[] args)

        {

            Task.Factory.StartNew(GetWeather);

            ReadLine();

        }

        private static async void GetWeather()

        {

            string customErrorMessage;

            HttpClient client = new HttpClient();

            try

            {

                HttpResponseMessage httpResponseMessage =

                await client.GetAsync

                ("http://api.openweathemap.org/data/2.5/weather?q=NewYorK");

                WriteLine(httpResponseMessage);

            }

            catch (Exception ex) when

            (DoASillyMistake(ex.Message, out customErrorMessage))

            {

                WriteLine(customErrorMessage);

            }

        }

        private static bool DoASillyMistake

        (string exceptionMessage, out string customErrorMessage)

        {

            if (exceptionMessage.Equals

            ("An error occurred while sending the request."))

            {

                customErrorMessage = "Unauthorized.";

                return true;

            }

            else

            {

                customErrorMessage = "Bad Gateway.";

                return true;

            }

        }

    }

}

This is a silly example I must say, but let's assume that if a request to a weather service fails, it's because of two reasons, one the service is not up and running [Bad Request] and the second is a user needs to be authorized before accessing the service [Unauthorized]. So, in my code, I know for sure that the HttpClient will throw an error message like below if it's a Bad Request.

Hide   Copy Code

"An error occurred while sending the request."

Again for any other error messages, let's assume it's an Unauthorized request. If you look carefully at the DoASillyMistake(string exceptionMessage, out string customErrorMessage) function, you will see I really did a silly mistake. I've shown the user that they are 'Unauthorized' to access the service while the message should be 'Bad Request' since the service url is not valid *[there is a letter is missing in the url; 'r' to complete the word weather]*. Now the funny and bit irritating part is that the user will now start looking for a registration process to get access to the service. But sadly, we all know that is not the case. The funnier part is even if he registered himself, from now he will get a 'Bad Request'. So, you must be careful when using exception filtering.

But even this side effect can sometimes come in handy. For example, you can attach a filter function which logs error messages and returns false so that the log contains an appropriate exception and also the execution cursor doesn't end up in the catch block. Here is a good example from that I found in this open source .NET git repo documentation, [New Language Features in C# 6](https://github.com/dotnet/roslyn/wiki/New-Language-Features-in-C%23-6).

Hide   Copy Code

private static bool Log(Exception e) { */\* log it \*/* ; return false; }

....

try { .... } catch (Exception e) when (Log(e)) {}