**C# Notes**

* dotnet new console

dotnet new console -o projectName

* **Introduction:**
* A **literal** value is a constant value that never changes. Ex: true, 019188, “hello”, …etc. We just need to display a literal instance of data type to output.
* Single quotes create a **character literal**. Recall that using double quotation marks creates a **string** data type.

Example:

1. “Hello World!” 🡪 string 🡪 use“”
2. ‘b’ 🡪 char 🡪 use ‘’

* The **\n** inside the WriteLine will only create a single new line.
* **Data types in C#:**

1. int, string, Char, float, double, bool, and decimal.
2. Implicitly typed local variables 🡪 **var** :

var message = "Hello World!";

message = 10.703m;

(2,11): error CS0029: Cannot implicitly convert type 'decimal' to 'string'

var in flutter can be initialized later but in C# it should be initialized when it is declared. Other programming languages use the var keyword differently than C#, variables are assigned a type by the **compiler** regardless of whether you use the actual data type name or allow the compiler to imply the data type. In other words, the type is locked in at **the time of declaration** and therefore will never be able to hold values of a different data type.

To create a **decimal** literal, append the letter m after the number. In this context, the m is called a *literal suffix*. The literal suffix tells the compiler you wish to work with a value of decimal type.

You can use either a lower-case **m** or upper-case **M** as the literal suffix for a decimal. In all even in **F** and **f**.

Float Type Precision

----------------------------

|  |  |  |
| --- | --- | --- |
| **C# type/keyword** | **Approximate range** | **Precision** |
| float | ±1.5 x 10−45 to ±3.4 x 1038 | ~6-9 digits |
| double | ±5.0 × 10−324 to ±1.7 × 10308 | ~15-17 digits |
| decimal | **±1.0 x 10-28 to ±7.9228 x 1028** | **28-29 digits** |

Suppose you needed to collect data from a user, like a phone number or postal code. Depending on the country/region where you live, that data may consist of numeric characters. However, since you rarely perform mathematical calculations on phone numbers and postal codes, you should prefer to use a **string** data type when working with them.

**Summary:**

* string for words, phrases, or any alphanumeric data for presentation, not calculation
* char for a single alphanumeric character
* int for a whole number
* decimal for a number with a fractional component
* bool for a true/false value

**Variable name rules:**

* Variable names can contain alphanumeric characters and the underscore character. Special characters like the hash symbol # (also known as the number symbol or pound symbol) or dollar symbol $ are not allowed.
* Variable names must begin with an alphabetical letter or an underscore, not a number.
* Variable names are case-sensitive, meaning that string Value; and string value; are two different variables.
* Variable names must **not** be a C# keyword. For example, you cannot use the following variable declarations: decimal decimal; or string string;.

**Variable name conventions:**

* Variable names should use **camel case**, which is a style of writing that uses a lower-case letter at the beginning of the first word and an upper-case letter at the beginning of each subsequent word. For example, string thisIsCamelCase;.
* Variable names should begin with an alphabetical letter. Developers use the underscore for a special purpose, so try to not use that for now.
* Variable names should be descriptive and meaningful in your application. You should choose a name for your variable that represents the kind of data it will hold (not the data type). For example: bool orderComplete;, NOT bool isComplete;.
* Variable names should be descriptive and meaningful in your app.
* Variable names should be one or more entire words appended together. Don't use contractions or abbreviations because the name of the variable (and therefore, its purpose) may be unclear to others who are reading your code.
* Variable names shouldn't include the data type of the variable, ex: string strValue;.

In the assign operator= The left-hand side of an assignment must be a variable, property or indexer. Ex:

**int thirdNumber = numbers[2];**   
In this example, 2 is the **index** of the third element in the numbers array, and the [] operator is the **indexer** used to access the element at that index.

**Combine strings using character escape sequences:**

* Use character escape sequences when you need to insert a special character into a literal string, like a tab \t, new line \n, or a double quotation mark \".
* Use an escape character for the backslash \\ when you need to use a backslash in all other scenarios.
* Use the @ directive to create a verbatim string literal that keeps all whitespace formatting and backslash characters in a string.
* Use the \u plus a four-character code to represent Unicode characters (UTF-16) in a string.
* Unicode characters may not print correctly depending on the application.

$ 🡪 interpolation

\ 🡪 escape special characters (( equal to )) @ 🡪 verbatim char

**Ex of string interpolation :** Console.WriteLine($@"C:\Output\{projectName}\Data");

* **A developer needs to capture a list of items from the user. The user will enter the keyboard combination ctrl + Esc to exit. Which is the best iteration statement for this purpose?**

1. foreach
2. do-while

**Correct! The do-while will allow the code to check each entry by the user until they enter the special keyboard combination to exit.**

**The do-while loop is the most suitable iteration statement for capturing input until a specific keyboard combination like Ctrl+Esc is entered.**

1. while

**Incorrect. It's possible to use the while statement, but the do-while will allow the user to enter a value before the code begins checking for the special keyboard combination.**

**Data types in C#:**

* **two kinds of types in C#: reference types and value types:**
  1. **Reference Types:**
* **Classes:**

Ex person that u can take object from it

* **Arrays:**

int[] numbers1 = { 1, 2, 3 };

* 1. **Value Types:**
* **Primitive Types:**

int num1 = 10;

* **Structs:**

struct Point

{

public int X { get; set; }

public int Y { get; set; }

}

* **Note:**

The code samples in this exercise are designed based on en-US culture settings, and use a period (.) as the decimal separator. Building and running the code with a culture setting that uses a different decimal separators (such as a comma ,) may give unexpected results or errors. To fix this issue, replace the period decimal separators in the code samples with your local decimal separator (such as ,). Alternatively, to run a program using the en-US culture setting, add the following code to the top of your program: using System.Globalization; and after any other using statements add CultureInfo.CurrentCulture = new CultureInfo("en-US");.

* **The** *out* **keyword in C# is used to pass arguments by reference to a method.**

void SwapNumbers(out int x, out int y)

{

int temp = x;

x = y;

y = temp;

}

int a = 10;

int b = 20;

SwapNumbers(out a, out b);

Console.WriteLine($"a = {a}, b = {b}"); // Output: a = 20, b = 10

* **Which technique should be used to change myInput, a string value "2.71828", into a decimal variable myInputDecimal?**

* + 1. decimal myInputDecimal = (decimal)(myInput);

**Incorrect. A string can't be cast into a decimal.**

* + 1. decimal myInputDecimal = myInput + 0;
    2. decimal.TryParse(myInput, out myInputDecimal);

**Correct. Using TryParse (or Parse()) is a valid technique.** **A string cannot be directly cast to a decimal type. This is because a string represents a sequence of characters, while a decimal represents a numerical value. Casting between incompatible types would lead to a compilation error.**

* **int.TryParse() does not round up the value.**

**(cast) truncate the value [in narrowing conversion only].**

**Convert.ToInt32() rounds up.**

* **Decimal** 🡪 **both**

**Float/ Double 🡪 1 of them is enough.**

* **Resize(): The Array.Resize() method adds or removes elements from the end of the array.  
  ------------**

**int[] numbers = { 1, 2, 3 };**

**Array.Resize(ref numbers, 5); // Increase the size to 5 elements**

**Console.WriteLine(numbers[3]); // Output: 0 (default value for int)**

* **.Add() cannot manipulate the size of an array, use a List<T>, Here's a table summarizing the key differences:**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Arrays** | **Lists** |
| Size | Fixed | Dynamic |
| Memory allocation | Contiguous block | Dynamic data structure |
| Performance | Generally faster for random access | Can be slower for random access, especially for large lists |
| Syntax | int[] numbers = new int[5]; | List<int> numbers = new List<int>(); |

* **The Array.Clear() method allows you to remove the contents of specific elements in your array and replace it with the array default value. For example, in a string array the element value cleared is replaced with null, when you clear a int array element the replacement is done with 0 (zero).**
* **Can you remove null elements from an array?**

If the Array.Resize() method doesn't remove empty elements from an array, is there another helper method that will do the job automatically? No. The best way to empty elements from an array would be to count the number of non-null elements by iterating through each item and increment a variable (a counter). Next, you would create a second array that is the size of the counter variable. Finally, you would loop through each element in the original array and copy non-null values into the new array. Using .copyTo();

* **In array use reverse directly. In string transform to array first :**

string value = "abc123";

char[] valueArray = value.**ToCharArray**();

Array.Reverse(valueArray);

string result = new string(valueArray);

Console.WriteLine(result);

* **To search for multiple symbols simultaneously, use .IndexOfAny(). You search with .IndexOfAny() to return the index of the** **first position of an array of char that occurs inside of another string.**
* **LastIndexOf() returns the last position of a character or string inside of another string.**
* **The Remove() method works similarly to the Substring() method. You supply a starting position and the length to remove those characters from the string.**
* **The Replace() method swaps all instances of a string with a new string.**
* When reading user entered values with the Console.ReadLine() method, it's best to enable a nullable type string using string? to avoid the code compiler generating a warning when you build the project.

string? readResult;

* **Understanding errors, Ex :**

error CS1002: ; expected. The ; expected suggests that you forgot to include a ; at the end of a statement. The Program.cs(53,18) portion of the message tells you the error location, on code line 53, at a position 18 characters in from the left.

* The expression **new S**tring**(' ', Console.BufferWidth)** creates a new string that consists entirely of spaces. The length of this string is determined by the value of **Console.BufferWidth**, which represents the width of the console window in character units. Console.WriteLine(new String(' ', Console.BufferWidth)); This will print a line of spaces that fills the entire width of the console, effectively clearing the screen.
* **Thread.Sleep(100);** Suspends the current thread for the specified number of milliseconds.
* The **Trim()** method in C# removes both leading and trailing white space characters from a string. White space characters include spaces, tabs, newlines, and carriage returns.
* It is important to remember that string is a reference type, but it is *immutable*. That means once it has been assigned a value, it can't be altered. In C#, when methods and operators are used to modify a string, the result that is returned is actually a new string object.
* **Positional** vs **named** parameters
* **Optional** parameters has a default value
* void MyMethod(int x, int y, string z)

{

// Method body

}

MyMethod(z: "Hello", x: 1, y: 2); **// Named parameters use them directly in the method call.**

* When combining named and positional arguments, you must use the correct order of parameters.
* int[,] myArray = new int[3, 4]; // Creates a 3x4 2D array

int numberOfRows = myArray.GetLength(0); // Gets the number of rows (3)

int numberOfColumns = myArray.GetLength(1); // Gets the number of columns (4)

* What data type is returned from the following statement: return 100 \* 0.5;

An int type

A decimal type

**A double type** (( not decimal because double is the default type in the implicit conversion by the compiler )))

* **Testing types:**

*Functional* and *Nonfunctional* testing. For example, functional and nonfunctional testing could be divided into the following subcategories:

* Functional testing - Unit testing - Integration testing - System testing - Acceptance testing
* Nonfunctional testing - Security testing - Performance testing - Usability testing - Compatibility testing
* errors that occur during the application **runtime** are referred to as **exceptions**. developers are responsible for handling exceptions by using **"try" and "catch"** statements in their code.
* **step over vs step into vs step out vs continue:**

A screenshot of a computer

Description automatically generated

1. Pause/Continue. This button can be used to pause execution when the code is running and Continue when code execution has been paused.
2. Step Over. This button can be used to execute the next method as a single command without inspecting or following its component steps.
3. Step Into. This button can be used to enter the next method or code line and observe line-by-line execution steps.
4. Step Out. When inside a method, this button can be used to return to the earlier execution context by completing all remaining lines of the current method as though they were a single command.
5. Restart. This button can be used to terminate the current program execution and start debugging again using the current configuration.
6. Stop. This button can be used to terminate the current program execution.
   * + **Breakpoints:**
7. Visual Studio Code enables setting breakpoints in the code editor or from the **Run** menu. Breakpoint code lines are marked with a red dot to the left of the line number.
8. Breakpoints can be removed or disabled using the same options used to set them. Bulk operations that affect all breakpoints are available on the **Run** menu.
9. Conditional breakpoints can be used to pause execution when a specified condition is met or when a 'hit count' is reached.
10. Logpoints can be used to log information to the terminal without pausing execution or inserting code.

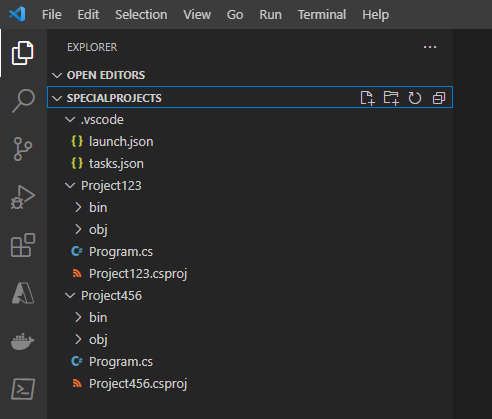
* **launch configurations :**

The launch.json file includes one or more launch configurations in the configurations list. The following attributes are **mandatory** for every launch configuration:

A screenshot of a computer program

Description automatically generated

* name: The reader-friendly name assigned to the launch configuration.
* type: The type of debugger to use for the launch configuration. A value of **codeclr** specifies the debugger type for .NET 5+ and .NET Core applications (including C# applications).
* request: The request type of the launch configuration. the values launch and attach are supported.
* **Remaining attributes:** <https://learn.microsoft.com/en-us/training/modules/implement-visual-studio-code-debugging-tools/6-examine-launch-configurations>
* The DEBUG CONSOLE panel ( == The default setting is internalConsole. The internalConsole setting corresponds to the DEBUG CONSOLE panel in the Panels area below the Visual Studio Code Editor.) doesn't support console input. For example, the DEBUG CONSOLE can't be used if the application includes a Console.ReadLine() statement. When you're working on a C# console application that reads user input, the console setting must be set to either integratedTerminal or externalTerminal. Console applications that write to the console, but don't read input from the console, can use any of the three console settings.

**Update the launch configuration to accommodate multiple applications:**Notice that the **name**, **preLaunchTask**, and **program** fields are all configured for application.

"version": "0.2.0",

"configurations": [

{

"name": "Launch Project123",

"type": "coreclr",

"request": "launch",

"preLaunchTask": "buildProject123",

"program": "${workspaceFolder}/Project123/bin/Debug/net7.0/Project123.dll",

"args": [],

"cwd": "${workspaceFolder}/Project123",

"console": "internalConsole",

"stopAtEntry": false

},

{

"name": "Launch Project456",

"type": "coreclr",

"request": "launch",

"preLaunchTask": "buildProject456",

"program": "${workspaceFolder}/Project456/bin/Debug/net7.0/Project456.dll",

"args": [],

"cwd": "${workspaceFolder}/Project456",

"console": "internalConsole",

"stopAtEntry": false

}

]

The **tasks.json** file contains the named tasks and the information required to complete the task:

"version": "2.0.0",

"tasks": [

{

"label": "buildProject123",

"command": "dotnet",

"type": "process",

"args": [

"build",

"${workspaceFolder}/Project123/Project123.csproj",

"/property:GenerateFullPaths=true",

"/consoleloggerparameters:NoSummary"

],

"problemMatcher": "$msCompile"

},

{

"label": "buildProject456",

"command": "dotnet",

"type": "process",

"args": [

"build",

"${workspaceFolder}/Project456/Project456.csproj",

"/property:GenerateFullPaths=true",

"/consoleloggerparameters:NoSummary"

],

"problemMatcher": "$msCompile"

}

]

With your updates to the launch.json and tasks.json files in place, the RUN AND DEBUG view displays launch options for debugging either the Project123 or Project456 application:

A screenshot of a computer program

Description automatically generated

* **Exception handling**

try

{

// try code block - code that may generate an exception

}

catch

{

// catch code block - code to handle an exception

}

finally

{

// finally code block - code to clean up resources. code block contains code that executes whether an exception occurs or not.

}

**Compiler-generated exceptions**

* ArrayTypeMismatchException: Thrown when an array can't store a given element because the actual type of the element is incompatible with the actual type of the array.
* DivideByZeroException: Thrown when an attempt is made to divide an integral value by zero. Only if two int :

Dividing a floating-point value by zero doesn't throw an exception; it results in positive infinity, negative infinity, or not a number (NaN), according to the rules of IEEE 754 arithmetic.

* FormatException: Thrown when the format of an argument is invalid.

int valueEntered;

string userValue = "two";

valueEntered = int.Parse(userValue); // FormatException occurs

* IndexOutOfRangeException: Thrown when an attempt is made to index an array when the index is less than zero or outside the bounds of the array.
* InvalidCastException: Thrown when an explicit conversion from a base type to an interface or to a derived type fails at runtime.
* NullReferenceException: Thrown when an attempt is made to reference an object whose value is null.

int[] values = null;

for (int i = 0; i <= 9; i++)

values[i] = i \* 2;

* OverflowException: Thrown when an arithmetic operation in a checked context overflows.

decimal x = 400;

byte i;

i = (byte)x; // OverflowException occurs

Console.WriteLine(i);

**Properties of the Exception class:**

* **Data**: The Data property holds arbitrary data in key-value pairs.
* **HelpLink**: The HelpLink property can be used to hold a URL (or URN) to a help file that provides extensive information about the cause of an exception.
* **HResult**: The HResult property can be used to access to a coded numerical value that's assigned to a specific exception.
* **InnerException**: The InnerException property can be used to create and preserve a series of exceptions during exception handling.
* **Message**: The Message property provides details about the cause of an exception.
* **Source**: The Source property can be used to access the name of the application or the object that causes the error.
* **StackTrace**: The StackTrace property contains a stack trace that can be used to determine where an error occurred.
* **TargetSite**: The TargetSite property can be used to get the method that throws the current exception.

**common exception types used when creating an exception:**

* ArgumentException or ArgumentNullException: Use these exception types when a method or constructor is called with an invalid argument value or null reference.
* InvalidOperationException: Use this exception type when the operating conditions of a method don't support the successful completion of a particular method call.
* NotSupportedException: Use this exception type when an operation or feature is not supported.
* IOException: Use this exception type when an input/output operation fails.
* FormatException: Use this exception type when the format of a string or data is incorrect.
* **The checked keyword in C# is used to enable overflow checking for arithmetic operations. without the checked keyword, the addition of x and y will result in integer overflow, and the value of z will be incorrect. With the checked keyword, the compiler will detect the overflow and throw an OverflowException.**
* **The is keyword in C# is used to check if an object is of a specific type at runtime:** expression is type

**With throw:**

void MethodA()

{ try

{

// Code that might throw an exception

}

catch (Exception ex)

{

Console.WriteLine("Error in MethodA: " + ex.Message);

throw; // Re-throw the exception

}}

void MethodB()

{ try{

MethodA();

}

catch (Exception ex)

{

Console.WriteLine("Error in MethodB: " + ex.Message);

}}

void Main()

{ try

{

MethodB();

}

catch (Exception ex){

Console.WriteLine("Error in Main: " + ex.Message);

}}

**Without throw :**

void MethodA()

{

try

{

// Code that might throw an exception

}

catch (Exception ex)

{

Console.WriteLine("Error in MethodA: " + ex.Message);

// No throw; statement

}

}

void MethodB()

{

try

{

MethodA();

}

catch (Exception ex)

{

Console.WriteLine("Error in MethodB: " + ex.Message);

}

}

void Main()

{

try

{

MethodB();

}

catch (Exception ex)

{

Console.WriteLine("Error in Main: " + ex.Message);

}

}

**Output with throw;:**

Error in MethodA: Error in MethodA

Error in MethodB: Error in MethodA

Error in Main: Error in MethodA

**Output without throw;:**

Error in MethodA: Error in MethodA

* if an exception occurs in MethodA, it will be caught and logged. However, the throw; statement re-throws the exception, allowing it to be caught by the catch block in MethodB and then by the catch block in Main. To throw an exception, you create an instance of an exception-derived class, configure its properties, and then use the throw keyword.

**Avoid:**

* Don't throw System.Exception, System.SystemException, System.NullReferenceException, or System.IndexOutOfRangeException intentionally from your own source code.
* Don't create exceptions that can be thrown in debug mode but not release mode. To identify runtime errors during the development phase, use Debug.Assert instead.

**Notes:**

* The Debug.Assert method is a tool for catching logic errors during development. By default, the Debug.Assert method works only in debug builds. You can use Debug.Assert in debug sessions to check for a condition that should never occur. The method takes two parameters: a Boolean condition to check, and an optional string message to display if the condition is false. Debug.Assert should not be used in place of throwing an exception, which is a way to handle exceptional situations during normal execution of your code. You should use Debug.Assert to catch errors that should never occur, and use exceptions to handle errors that could occur during normal execution of your program.
* The top-level statements include a Console.ReadLine() statement. The launch.json file will need to be updated before debugging.To use Console.ReadLine() in top-level statements, you need to update the launch.json file to set stopAtEntry to true.