**AR/VR Developer Roadmap**

**1. Foundation Stage**

**1.1 Programming Basics (C#) :**

**Definition**

C# is a simple & powerful object-oriented programming language developed by Microsoft. C# can be used to create various types of applications, such as web, windows, console applications, or other types of applications using Visual studio.

**First Program**

****

**Image 1.1.1: HelloWorld Program in C#**

**Explanation**

* using System; imports the System namespace which contains fundamental classes like Console.
* namespace HelloWorldApp groups related classes.
* class Program defines a class named Program.
* static void Main(string[] args) is the entry point of the program where execution starts.
* Console.WriteLine("Hello World!"); prints "Hello World!" to the console.
* Console.ReadKey(); waits for a key press to prevent the console window from closing immediately after printing the message.

**Keywords in C#**

C# has a set of reserved words known as **keywords** that have special meanings in the language and cannot be used as identifiers (such as variable names or method names). There are about **79 keywords** in C#, all written in lowercase.

**Common C# Keywords**

* **Value types:** bool, byte, char, decimal, double, float, int, long, sbyte, short, uint, ulong, ushort
* **Control flow:** if, else, switch, case, for, foreach, while, do, break, continue, goto, return, yield, throw, try, catch, finally
* **Modifiers and access:** public, private, protected, internal, static, readonly, sealed, abstract, virtual, override, const, unsafe
* **Others:** class, struct, interface, enum, event, delegate, null, true, false, sizeof, typeof, using, namespace, new, this, base

**Contextual Keywords**

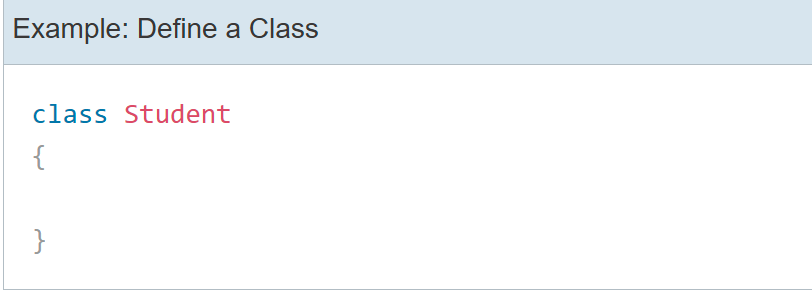
There are also **25 contextual keywords** that have special meanings only in certain contexts but can be used as identifiers otherwise. Examples include async, await, var, dynamic, partial, yield, when, etc.

These keywords are fundamental to C# programming and help define the language's syntax and structure. If needed, keywords can sometimes be used as identifiers by prefixing them with '@' (e.g., @class).

**Class in C#**

**Definition:** Class is a blueprint or template for creating objects. It defines the data (fields or properties) and behavior (methods) that the objects created from the class will have. A class acts as a logical structure and doesn't occupy memory by itself.

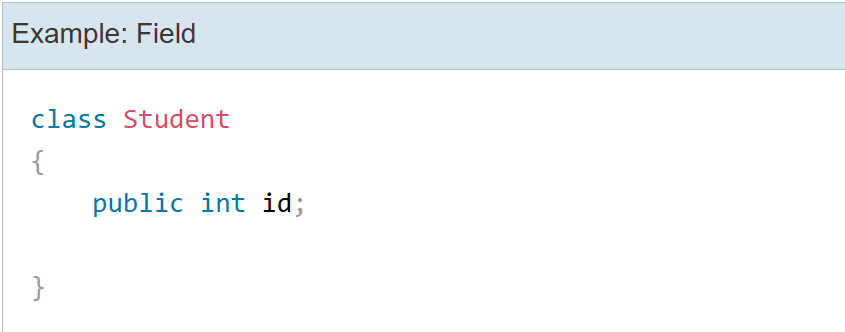
**Define a Class in c#:** In C#, a class can be defined by using the class keyword. Let's define a class named 'Student'.



**Image 1.1.1: defining Class in C#**

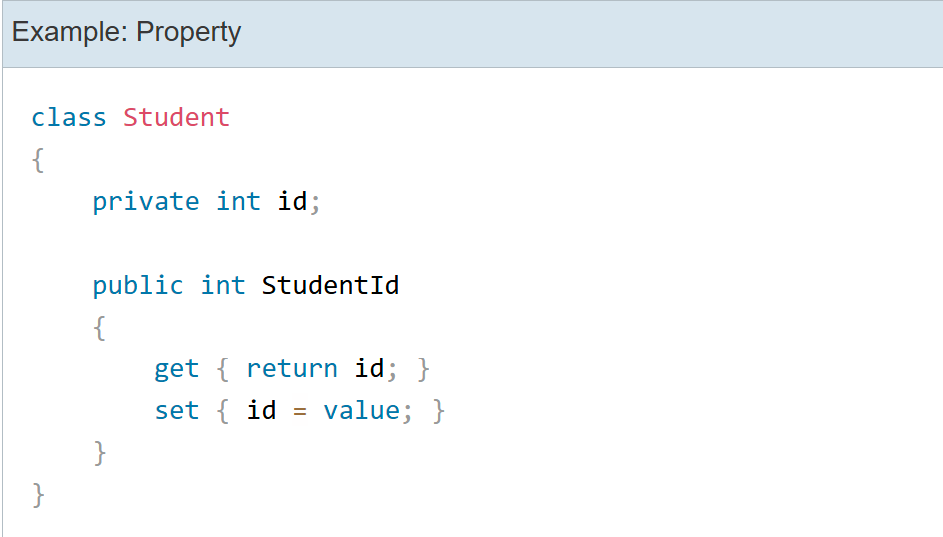
A class can contain one or more constructors, fields, methods, properties, delegates, and events. They are called class members. A class and its members can have access modifiers such as public, private, protected, and internal, to restrict access from other parts of the program.

**Field:** A class can have one or more fields. It is a class-level variable that holds a value. Generally, field members should have a private access modifier used with property.

****

**Image 1.1.2: defining Field in C#**

**Property:** In C#, a property is a member of a class that provides a flexible mechanism to read, write, or compute the value of a private field. Properties appear like public data members from outside the class but are implemented using special methods called **accessors** — specifically, a **get** accessor to retrieve a value and a **set** accessor to assign a value.

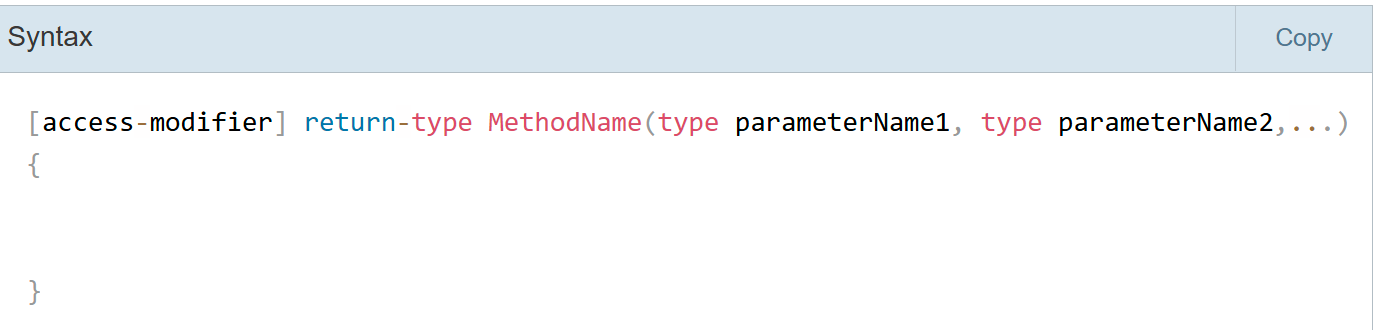
****

**Image 1.1.3: defining Property in C#**

In the above example, the id is a private field that cannot be accessed directly. It will only be accessed using the StudentId property. The get returns the value of the underlying field and set assigns the value to the underlying field id.

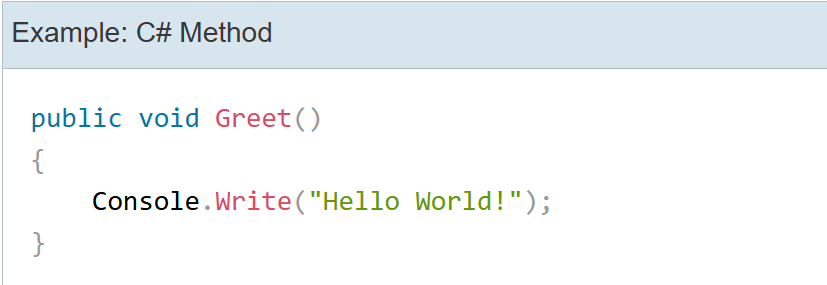
**Method:** A method can contain one or more statements to be executed as a single unit. A method may or may not return a value. A method can have one or more input parameters.

**Syntax:**

****

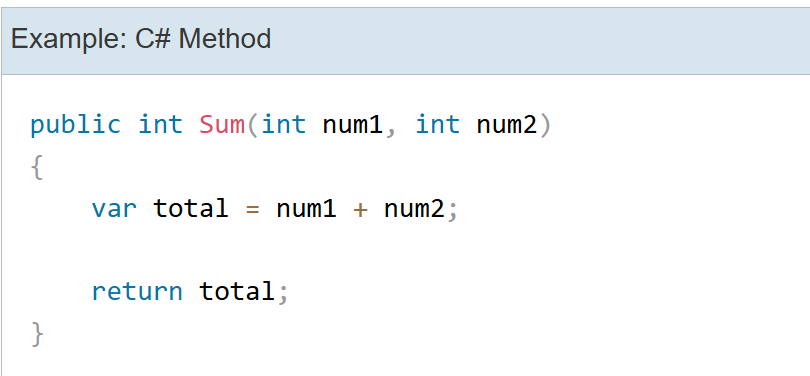
**Image 1.1.4: Syntax of Method in C#**

**Method without returning value:** The following method doesn't return anything and doesn't have any parameters. The return type is void.



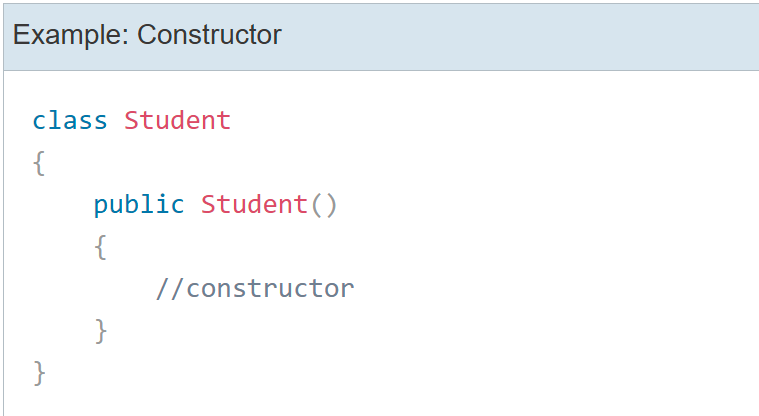
**Image 1.1.5: Method defined without returning any value in C#**

**Method with returning value:** The following defines the Sum method that returns the sum of two numbers.



**Image 1.1.6: Method defined with returning any value in C#**

**Constructor:** A constructor is a special type of method which will be called automatically when you create an Object of a class. A constructor is defined by using an access modifier and class name <access-modifier> <class-name>().



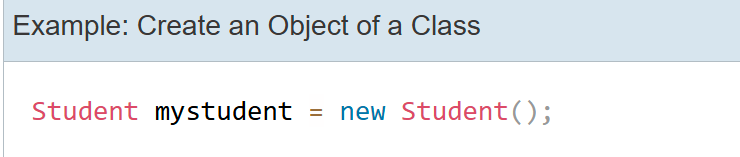
**Image 1.1.7 Example of constructor in C#**

**Note:**

* A constructor name must be the same as a class name.
* A constructor can be public, private, or protected.
* The constructor cannot return any value so cannot have a return type.
* A class can have multiple constructors with different parameters but can only have one parameterless constructor.
* If no constructor is defined, the C# compiler would create it internally.

**Object of a Class:** An **Object of class** in C# is a specific **instance** created based on the structure and behavior defined by that class. When an object is created, it represents a concrete entity with its own unique state and can perform behaviours (methods) defined by the class.

In C#, an object of a class can be created using the **new** keyword and assign that object to a variable of a class type. For example, the following creates an object of the **Student** class and assign it to a variable of the **Student** type.

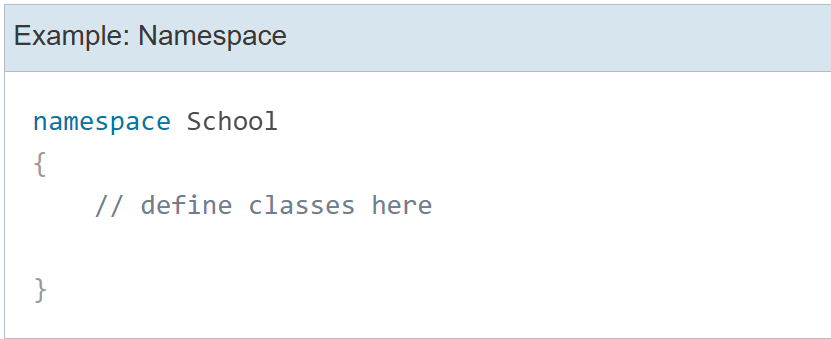


**Image 1.1.8: defining Object of a Class in C#**

**Namespace in C#**

**Definition:** A **namespace** is a **container that holds classes, structs, interfaces, enums, and other namespaces**. It is used to **organize code and avoid naming conflicts** in large programs.

In C#, a namespace can be defined using the namespace keyword.

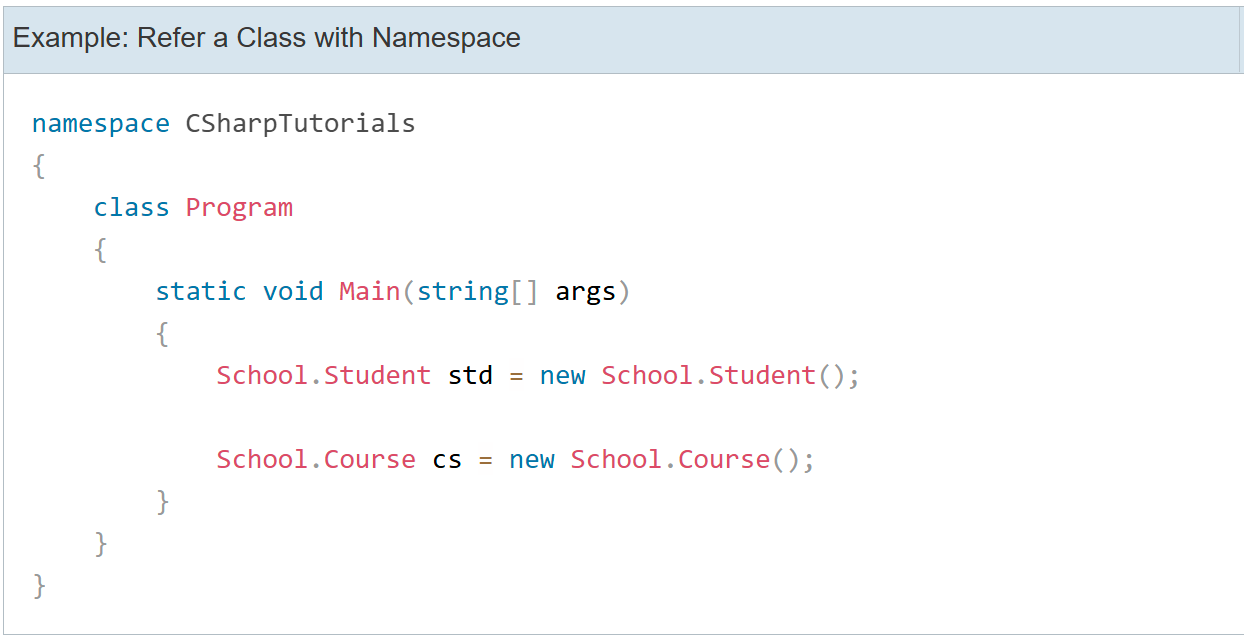


**Image 1.1.9 Syntax of namespace in C#**



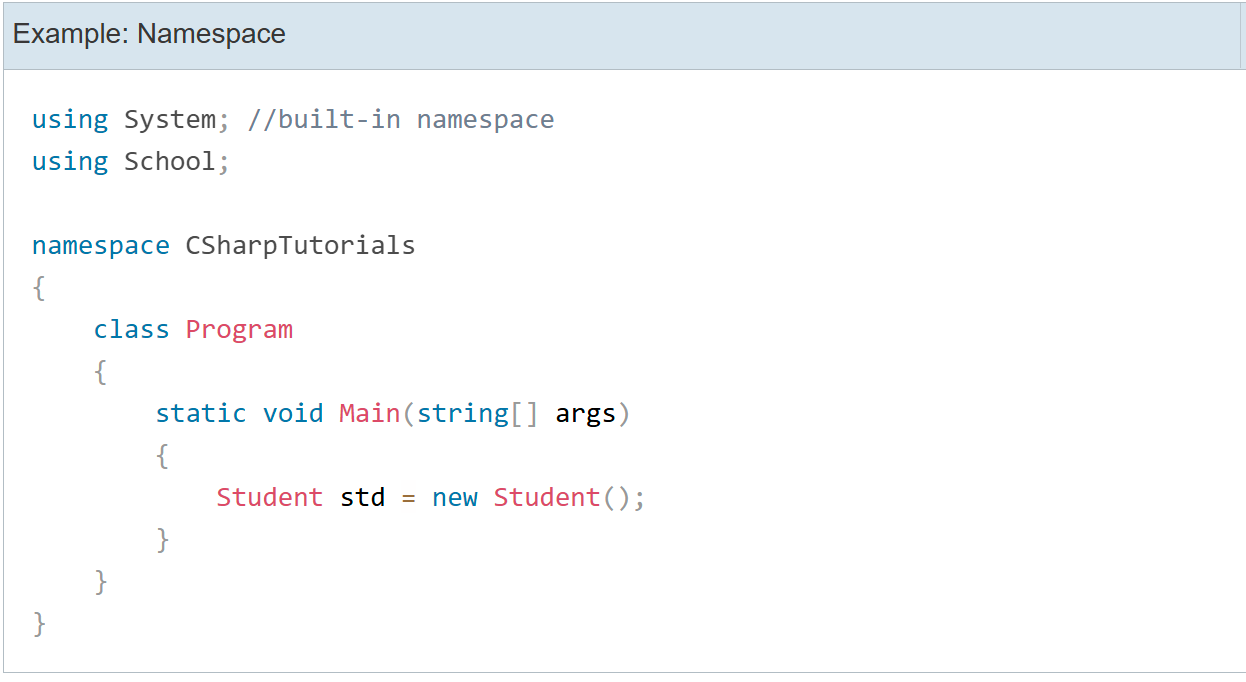
**Image 1.1.10 Example of Namespace in C#**

Classes under the same namespace can be referred to as namespace.classname syntax. For example, the Student class can be accessed as School.Student.

****

**Image 1.1.11 Example: Refer a class with namespace.**

To use classes under a namespace without the fully qualified name, import the namespace with the using keyword at the top of C# class file.

****

**Image 1.1.12 Example: Namespace with using keyword.**

**Variables in C#**

**Definition:** A variable is a container used to store data values during program execution. Each variable has a type, which determines the kind of data it can hold, such as numbers, text, or boolean values.

**Syntax:**

****

**Image 1.1.13 Example: Syntax of variable declaration.**

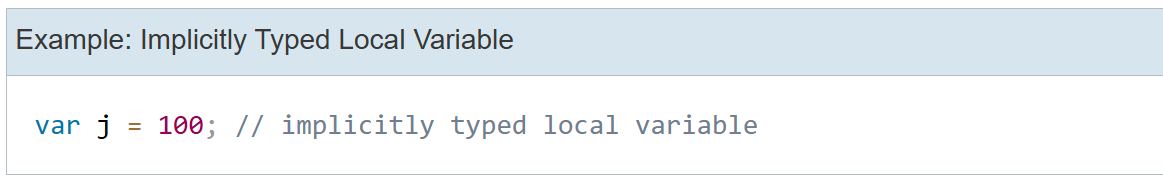
**The followings are naming conventions for declaring variables in C#:**

* Variable names must be unique.
* Variable names can contain letters, digits, and the underscore \_ only.
* Variable names must start with a letter.
* Variable names are case-sensitive,  num and Num are considered different names.
* Variable names cannot contain reserved keywords. Must prefix @ before keyword if want reserve keywords as identifiers.

**var in C#**

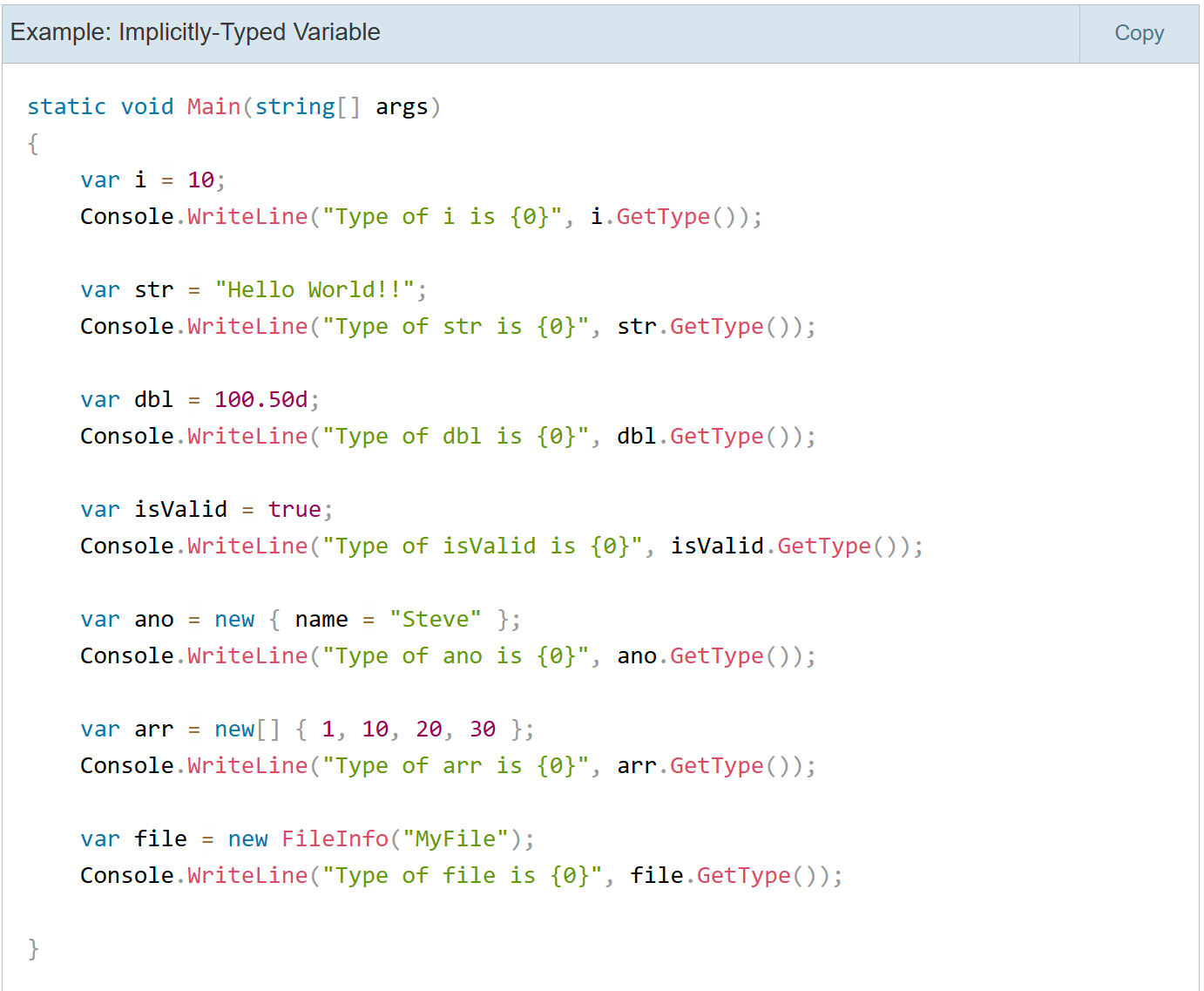
In C#, variables must be declared with the data type. These are called explicitly typed variables.

C# 3.0 introduced var keyword to declare method level variables without specifying a data type explicitly.

****

**Image 1.1.14 Example: declaration and initialization of var.**

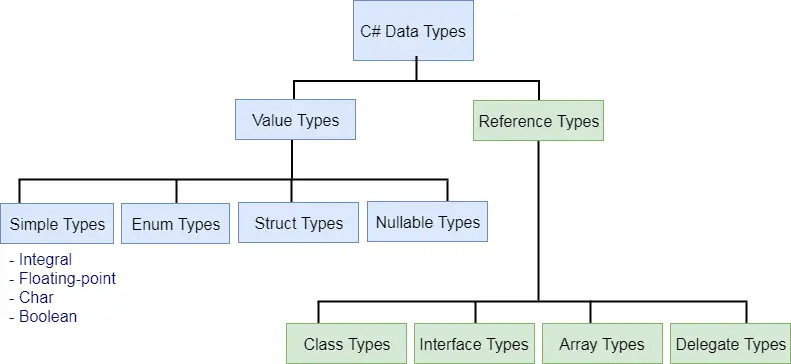
var can be used to declare any built-in data type or a user-defined type or an anonymous type variable. The following example shows C# compiler infers type based on the value:

****

**Image 1.1.15 Example: implicitly-typed variable**

**Data Types in C#**

**Definition:** A data types specifies the kind of data a variable can hold and determines the size and layout of the data in memory. C# supports a variety of built-in data types broadly categorized into a value types and reference types.

****

**Image 1.1.16 Types of data types**

**StringBuilder in C#**

In C#, **StringBuilder** is a class in the System.Text namespace designed for efficient manipulation of strings, especially when performing multiple modifications like appending, inserting, replacing, or removing text. Unlike regular strings in C# which are immutable (creating a new string with each modification), StringBuilder dynamically modifies the string content without creating new string objects, improving performance and reducing memory usage.

**Nullable Types in C#**

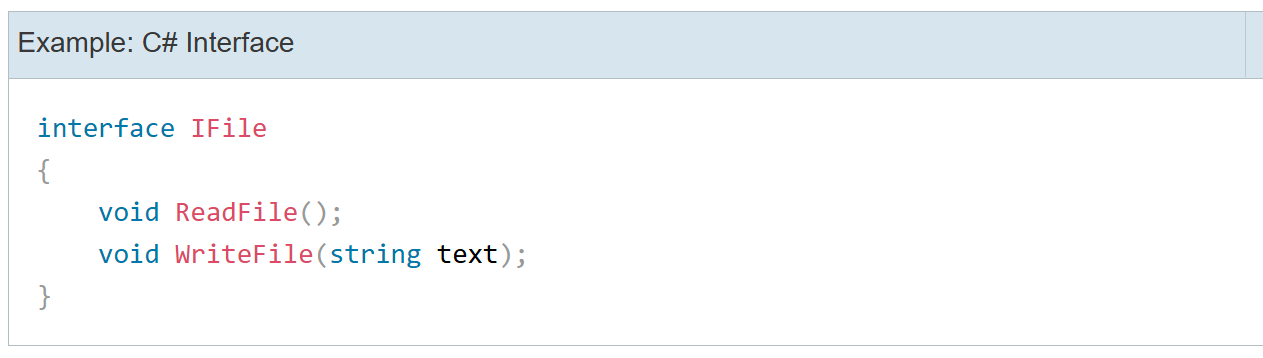
As you know, a value type cannot be assigned a null value. For example, int i = null will give you a compile time error.

C# 2.0 introduced nullable types that allow you to assign null to value type variables. You can declare nullable types using Nullable<t>where T is a type

**Interface in C#**

In C#, an interface is a contract or blueprint that defines a set of methods, properties, events, or indexers that a class or struct must implement. It specifies what a class must do, but not how to do it. Interfaces enforce a consistent API across different classes and enable multiple inheritance and polymorphism.

an interface can be defined using the interface keyword.

****

**Image 1.1.17 Declaration of Interface**

# if, else if, else Statements in C#

## C# if Statement: The if statement contains a boolean condition followed by a single or multi-line code block to be executed. At runtime, if a boolean condition evaluates to true, then the code block will be executed, otherwise not.

## Syntax:

## 

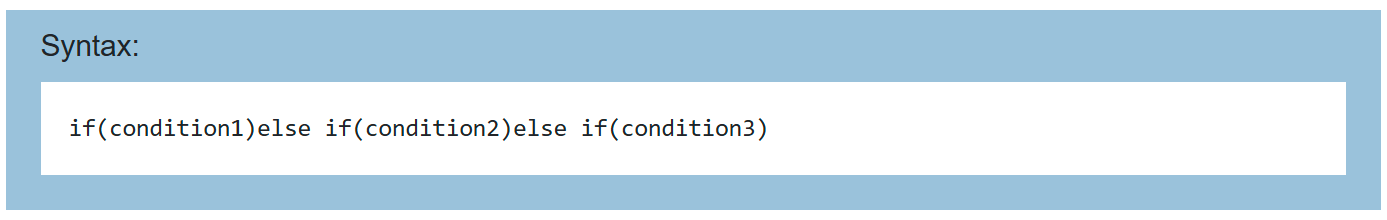
## Example:

## 

## Image 1.1.19 Example: if statement

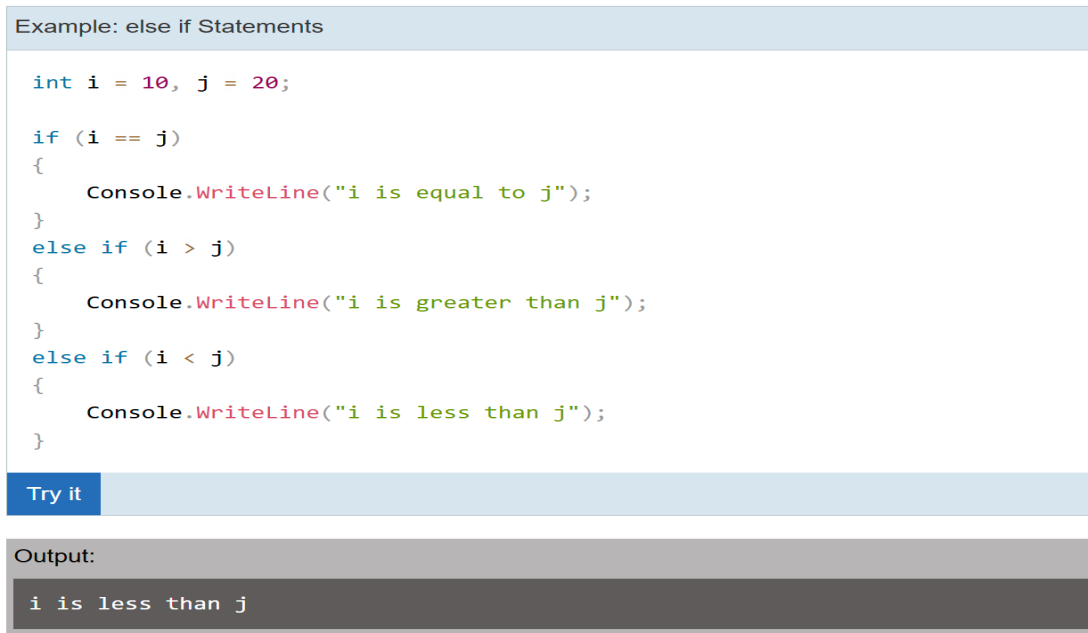
## else if Statement: Multiple else if statements can be used after an if statement. It will only be executed when the if condition evaluates to false. So, either if or one of the else if statements can be executed, but not both.

**Syntax:**



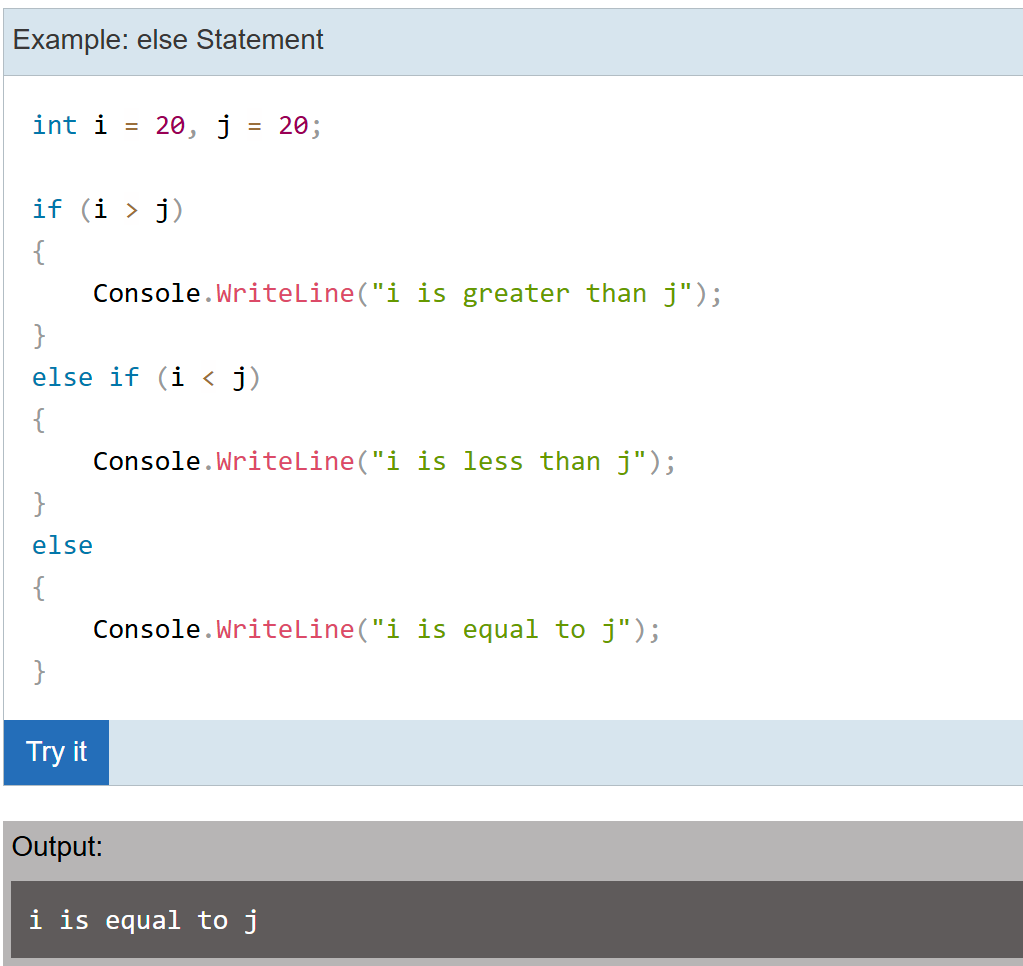
**Image 1.1.20 Syntax of else if Statement**

**Example:**

****

**Else Statement:** The else statement can come only after if or else if statement and can be used only once in the if-else statements. The else statement cannot contain any condition and will be executed when all the previous if and else if conditions evaluate to false.

**Example:**

****

**Image 1.1.22 Example of else statement**

# Ternary Operator in C#

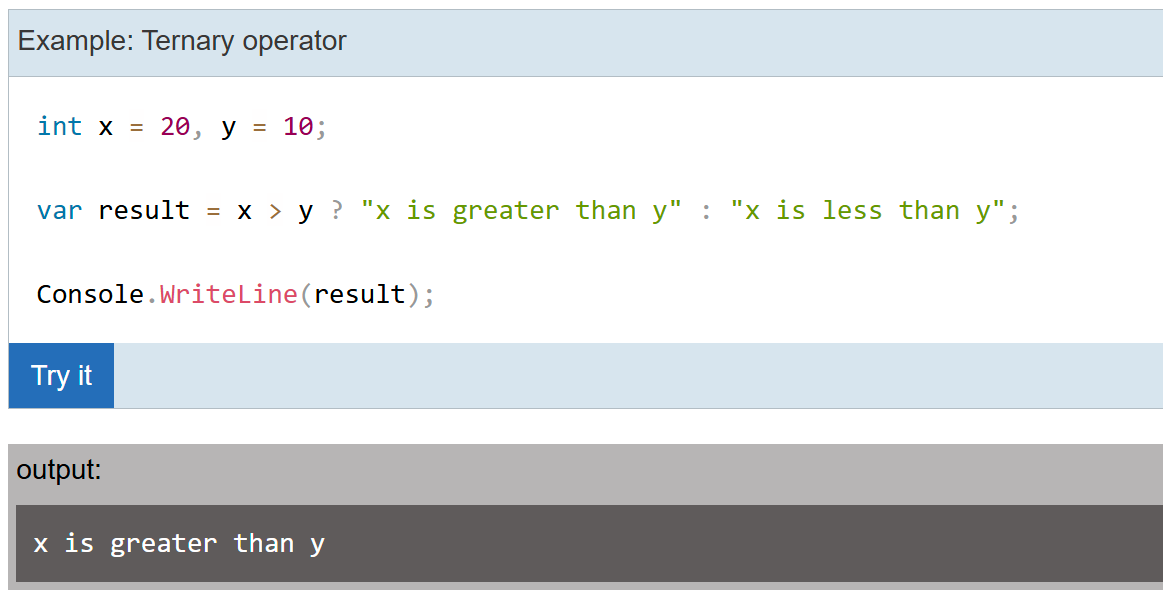
**Definition:** The Ternary Operator in C# is a concise way to write simple conditional statements. It is a shorthand for the if-else statement and is also known as the conditional operator. The ternary operator uses three operands and has the following syntax:

****

**Image 1.1.23 Syntax of Ternary Operator**

The ternary operator starts with a boolean condition. If this condition evaluates to true then it will execute the first statement after ?, otherwise the second statement after : will be executed.

**Example:**

****

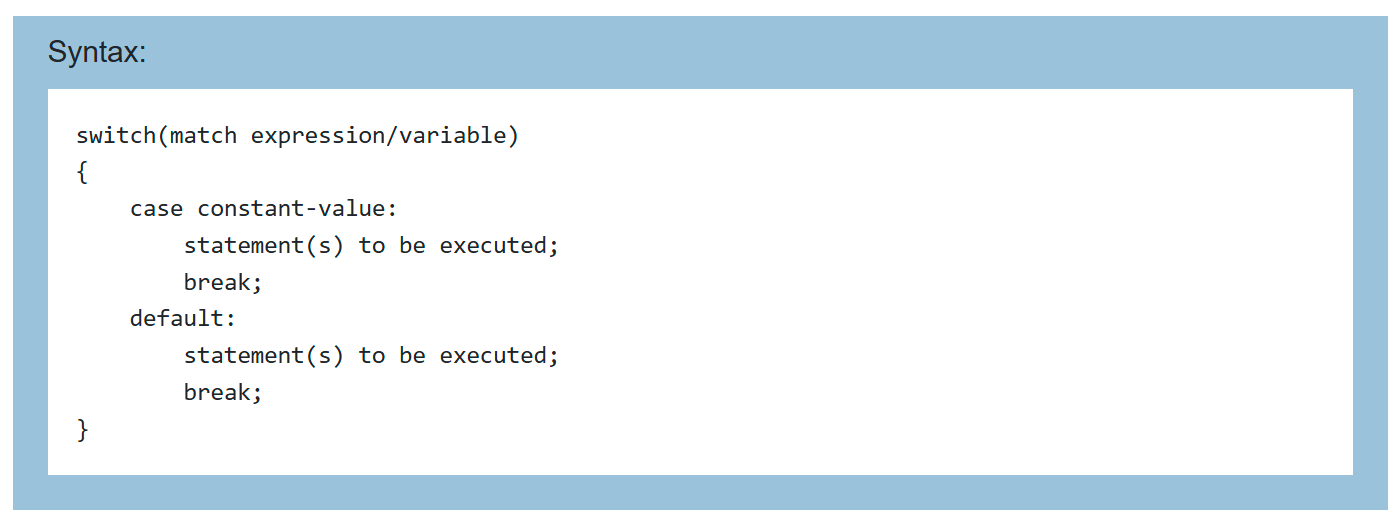
**Image 1.1.24 Example for Ternary Operator**

Above, a conditional expression x > y returns true, so the first statement after ? will be execute.

# Switch Statement in C#

**Definition:** The switch statement in C# is a control statement that allows selecting one of many code blocks to execute based on the value of an expression. It provides a cleaner alternative to multiple if-else statements when checking a variable against different constant values.

**Syntax:**



**Image 1.1.25 Syntax of switch statement**

The switch statement starts with the switch keyword that contains a match expression or a variable in the bracket switch(match expression). The result of this match expression or a variable will be tested against conditions specified as cases, inside the curly braces. A case must be specified with the unique constant value and ends with the colon :. Each case includes one or more statements to be executed. The case will be executed if a constant value and the value of a match expression/variable are equal. The switch statement can also contain an optional default label. The default label will be executed if no cases executed. The break, return, or goto keyword is used to exit the program control from a switch case.

**Example:**

****

Above, the switch(x) statement includes a variable x whose value will be matched with the value of each case value. The above switch statement contains three cases with constant values 5, 10, and 15. It also contains the default label, which will be executed if none of the case value match with the switch variable/expression. Each case starts after : and includes one statement to be executed. The value of x matches with the second case case 10:, so the output would be Value of x is 10.

# For Loop in C#

**Definition:** The **For loop** in C# is a control flow statement that allows you to execute a block of code repeatedly for a specified number of times. It is useful when the number of iterations is known beforehand.

**Syntax:**

****

**Image 1.1.27 Syntax of for loop**

The for loop contains the following three optional sections, separated by a semicolon:

**Initializer:** The initializer section is used to initialize a variable that will be local to a for loop and cannot be accessed outside loop. It can also be zero or more assignment statements, method call, increment, or decrement expression e.g., ++i or i++, and await expression.

**Condition**: The condition is a boolean expression that will return either true or false. If an expression evaluates to true, then it will execute the loop again; otherwise, the loop is exited.

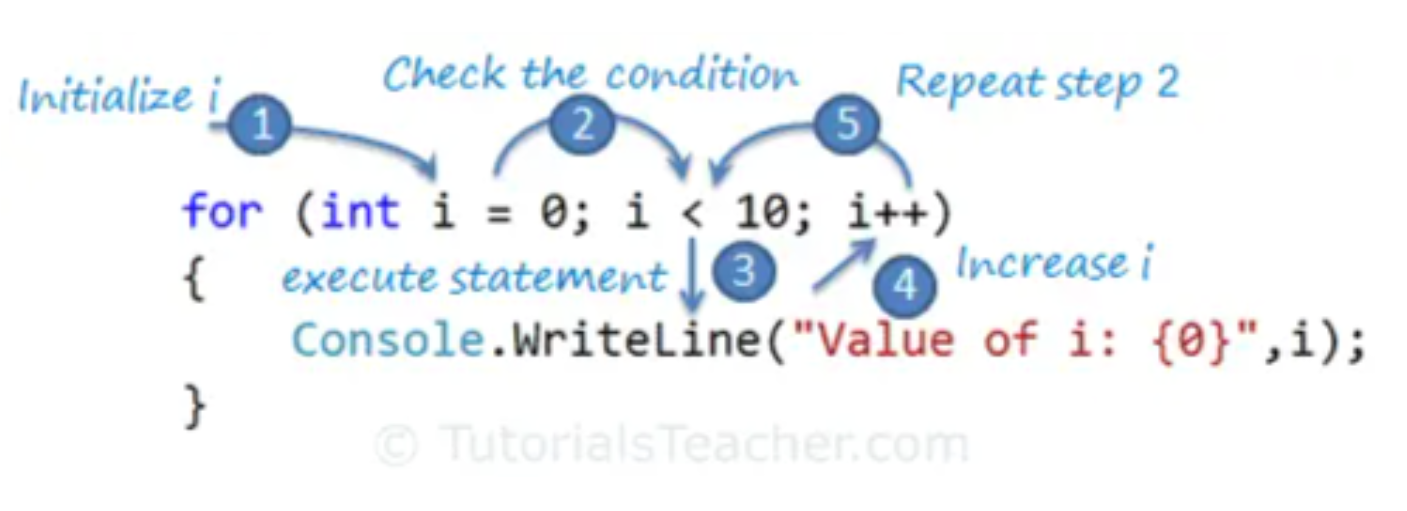
**Iterator**: The iterator defines the incremental or decremental of the loop variable.

**Example:**

In the example, int i = 0 is an initializer where we define an int variable i and initialize it with 0. The second section is the condition expression i < 10, if this condition returns true then it will execute a code block. After executing the code block, it will go to the third section, iterator. The i++ is an incremental statement that increases the value of a loop variable i by 1. Now, it will check the conditional expression again and repeat the same thing until conditional expression returns false.

****

**Image 1.1.28 Example of for loop**

****

**Image 1.1.29 Execution steps of for loop**

Practice object-oriented programming (classes, inheritance, events).

# While Loop in C#

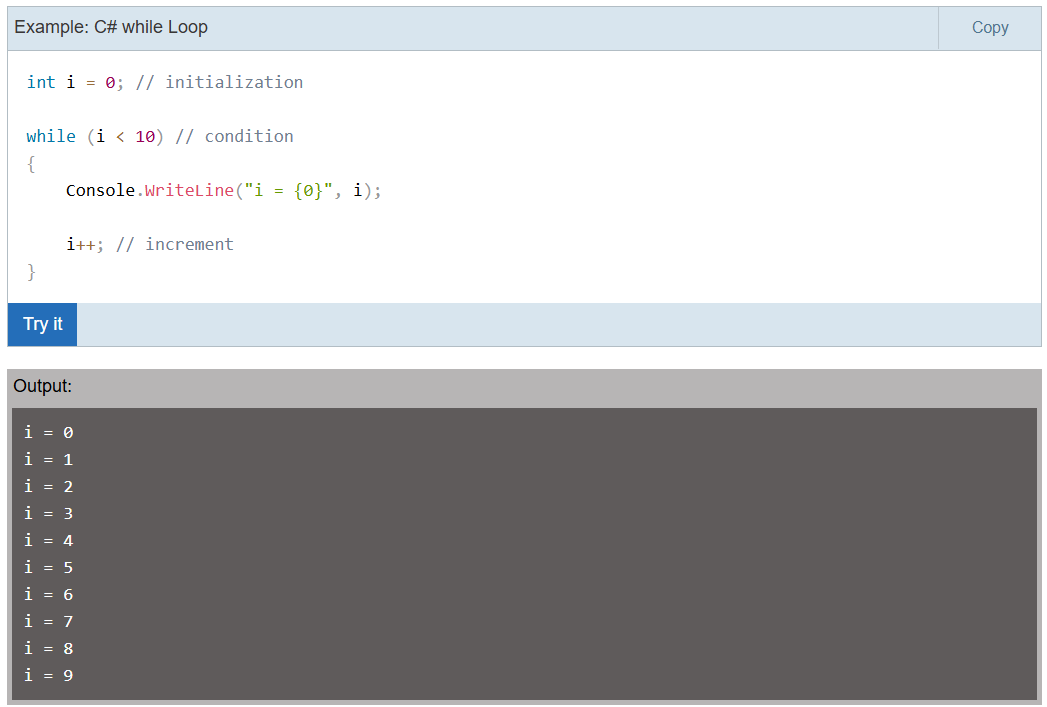
**Definition:** C# provides the while loop to repeatedly execute a block of code as long as the specified condition returns true.

**Syntax:**

****

The while loop starts with the while keyword, and it must include a boolean conditional expression inside brackets that returns either true or false. It executes the code block until the specified conditional expression returns false.

The for loop contains the initialization and increment/decrement parts. When using the while loop, initialization should be done before the loop starts, and increment or decrement steps should be inside the loop.

****

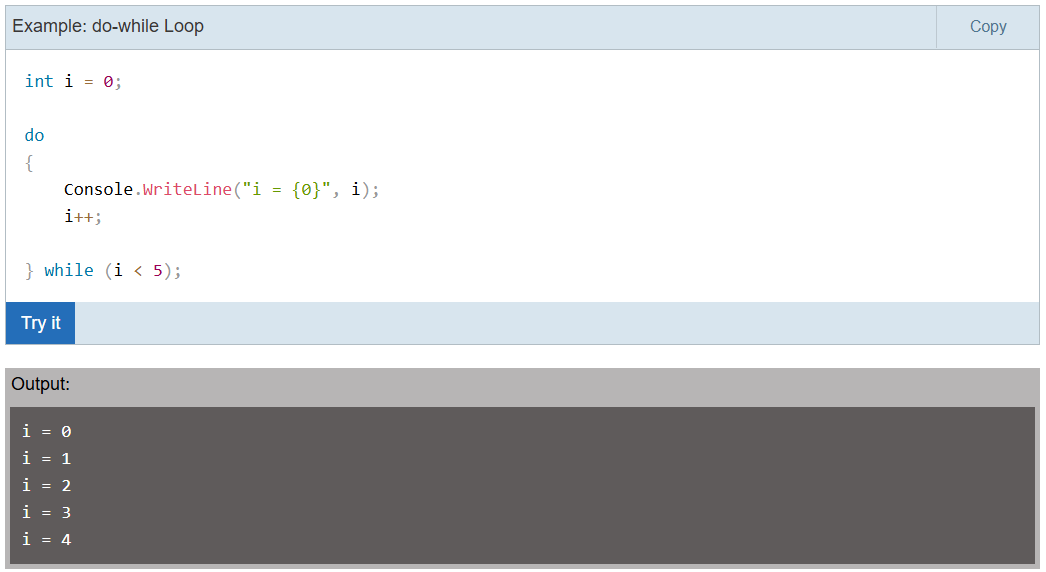
Above, a while loop includes an expression i < 10. Inside a while loop, the value of i increased to 1 using i++. The above while loop will be executed when the value of i equals to 10 and a condition i < 10 returns false.

# Do While Loop in C#

**Definition:** The do while loop is the same as while loop except that it executes the code block at least once.

The do-while loop starts with the do keyword followed by a code block and a boolean expression with the while keyword. The do while loop stops execution exits when a boolean condition evaluates to false. Because the while(condition) specified at the end of the block, it certainly executes the code block at least once.

**Example:**

****

**1.1.2 OOP’s (C#) :**

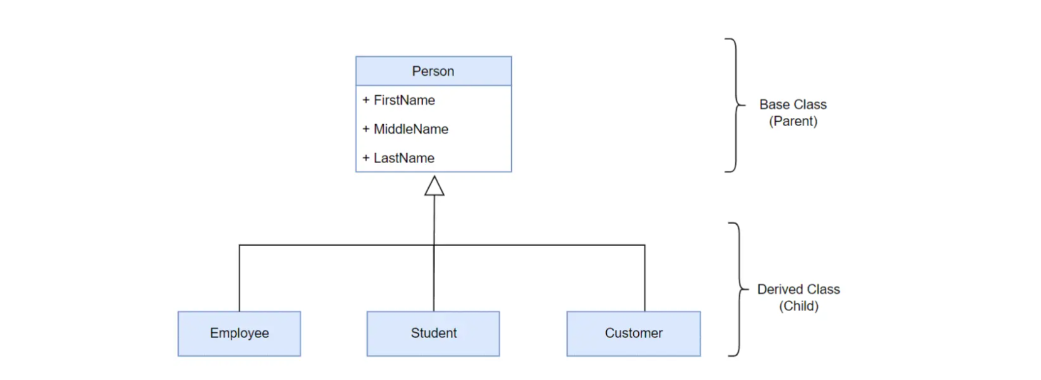
**Inheritance in C#**

In object-oriented programming, inheritance is another type of relationship between classes. Inheritance is a mechanism of reusing the functionalities of one class into another related class.

Inheritance is referred to as "is a" relationship. In the real world example, a customer is a person. In the same way, a student is a person and an employee is also a person. They all have some common things, for example, they all have a first name, middle name, and last name. So to translate this into object-oriented programming, we can create the Person class with first name, middle name, and last name properties and inherit the Customer, Student, and Employee classes from the Person class. That way we don't need to create the same properties in all classes and avoid the violation of the DRY (Do not Repeat Yourself) principle.

Note that the inheritance can only be used with related classes where they should have some common behaviors and perfectly substitutable. Follow the Liskov Substitution Principle in inheritance.

**Block Diagram:**

****

**Example:**

****

In the above example, the Person class is called the base class or the parent class, and the Employee class is called the derived class or the child class.

The Employee class inherits from the Person class and so it automatically acquires all the public members of the Person class. It means even if the Employee class does not include FirstName, LastName properties and GetFullName() method, an object of the Employee class will have all the properties and methods of the Person class along with its own members.

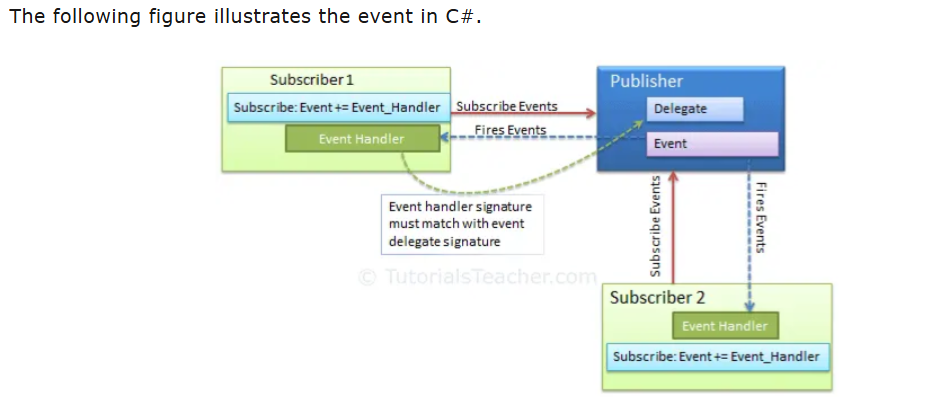
**Events in C#**

An event is a notification sent by an object to signal the occurrence of an action. Events in .NET follow the observer design pattern.

The class who raises events is called Publisher, and the class who receives the notification is called Subscriber. There can be multiple subscribers of a single event. Typically, a publisher raises an event when some action occurred. The subscribers, who are interested in getting a notification when an action occurred, should register with an event and handle it.

In C#, an event is an encapsulated delegate. It is dependent on the delegate. The delegate defines the signature for the event handler method of the subscriber class.

**Block Diagram:**

****

**1.2 Math & Graphics Basics:**

* + Linear algebra (vectors, matrices, transformations).
  + 3D coordinate systems, rotations (Euler vs Quaternion).
  1. **Design Tools**
  + Start with **Blender** for 3D modeling basics.
  + Explore **Canva** for UI mockups.

**2. Core AR/VR Development**

* **Game Engine**
  + Learn **Unity** (widely used for AR/VR).

**Introduction to Unity for AR/VR**

**2.1 What is Unity?**

Unity is a powerful and versatile cross-platform game engine used to create interactive 2D and 3D experiences such as games, simulations, and virtual/augmented reality applications. It offers a component-based architecture where game objects gain functionality through attachable components like visuals, physics, and scripts, primarily written in C#. The engine supports a wide range of platforms including mobile, desktop, consoles, VR, and AR devices.

****

**Image 2.1.1: Unity SDK logo.**

**2.2 Unity Editor Overview**

### 2.2.1 Scene View

### The Scene View is a visual workspace where developers construct and manipulate the game environment or virtual world. It displays the 3D or 2D representations of game objects, allowing users to move, rotate, and scale assets directly. This view provides essential spatial context for designing levels, setting up lighting and placing interactive elements.

### 2.2.2 Hierarchy Plane

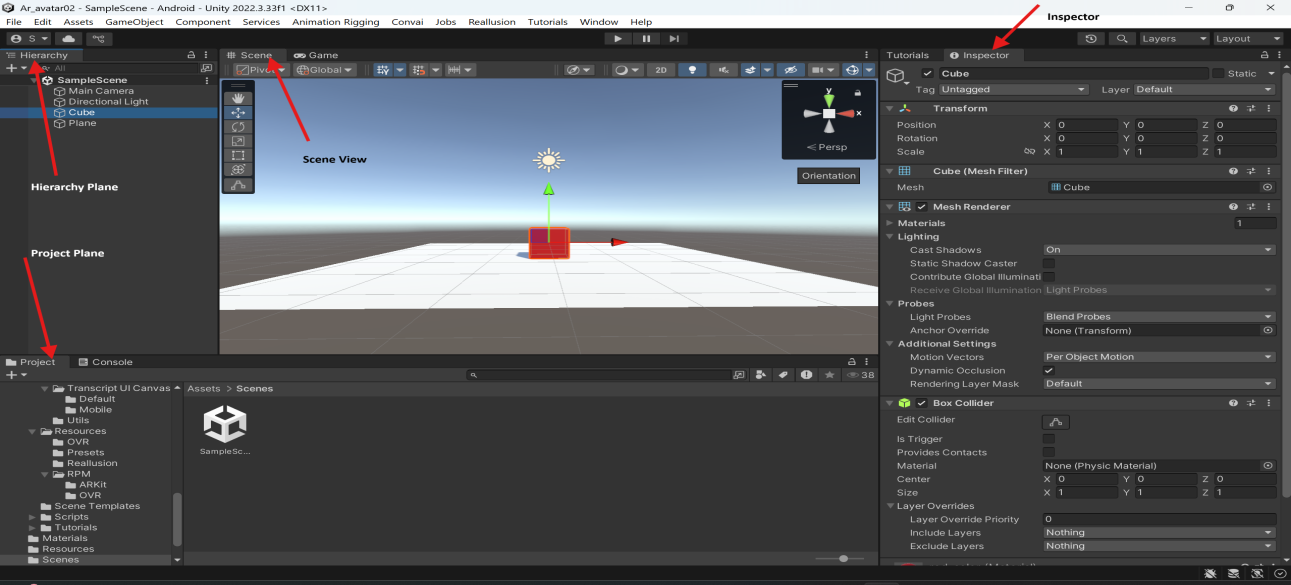
### The Hierarchy lists all the game objects present in the current scene in a tree structure. It organizes objects parented to one another, making it easier to select, group, and manage components of the scene. This panel reflects all assets visible and active in the Scene View.

### 2.2.3 Inspector Plane

### The Inspector displays detailed properties and components of the currently selected game object or asset. Developers use it to modify transform values (position, rotation, scale), add or remove components (scripts, colliders, renderers), and adjust properties such as material settings, behaviours and physics parameters.

### 2.2.4 Project Plane

### The Project Panel provides access to all assets, scripts, prefabs, textures, and resources included in the project. It acts as a file browser and manager, organizing files into folders for easy retrieval and drag-and-drop placement into the scene or hierarchy.

****

**Image 2.2.1: Unity** Editor Interface: Scene, Hierarchy, Project, and Inspector Panels

### 2.2.5 Game View

### The Game View shows the real-time output of the project, simulating what the player or user will see during gameplay or experience. It renders the scene as it will appear on the target device, enabling developers to test interactions, animations, and gameplay mechanics within the editor.

### 

**Image 2.2.2:** Game View Window in Unity Editor: Previewing Real-Time Scene Output

**2.3 Importing Assets in Unity Editor**

**Definition:**

Assets are the resources used to build a Unity project. These include **3D models, textures, audio, animations, prefabs, and scripts**. Importing assets means bringing external files or ready-made Unity content into your project so they can be used in scenes.

## 2.3.1 Importing 3D Models

* Unity supports common 3D model file formats like FBX, OBJ, and STL.
* To import, simply drag and drop the 3D model file into the **Project** panel or use **Assets > Import New Asset** from the menu.
* Upon import, Unity automatically processes the model and creates a usable asset.
* Models can then be dragged into the **Scene** or **Hierarchy** panel for placement.
* Unity provides options to configure the model's scale, mesh compression, and animation settings within the **Inspector**.

## 2.3.2 Importing Textures

* Textures such as PNG, JPEG, TIFF, or TGA files can be imported similarly by dragging them into the **Project** panel.
* Textures are used to add surface details to materials applied to 3D models or UI elements.
* Unity allows adjustments like wrap mode, filter mode, and compression quality in the **Inspector** after import.
* Multiple textures can be combined into materials inside Unity to define visual properties like colour, reflectivity, and transparency.

## 2.3.4 Importing Prefabs

* Prefabs are reusable game object templates that can include models, components, scripts, and more.
* Prefabs are created within Unity itself by dragging a configured game object from the **Hierarchy** into the **Project** panel.
* Prefabs can be imported if provided as Unity package files (.unity package), which are imported using **Assets > Import Package**.
* Using prefabs improves development efficiency by allowing duplication and easy updates to multiple instances across scenes.

## General Tips

* Always organize imported assets into descriptive folders within the **Project** panel for better management.
* Unity supports importing asset packages created in other engines or applications, expanding resources available.
* After import, assets can be optimized and customized through Unity’s powerful tools and scripting APIs.

## 2.4 Setting Up Build Settings in Unity (Android, iOS, XR)

Build settings in Unity configure the project for deployment on different platforms such as Android, iOS, and XR devices. Proper setup ensures that the application runs correctly and takes full advantage of platform-specific features.

## Accessing Build Settings

* Open Unity and go to **File > Build Settings**.
* This window shows a list of available platforms and settings for building the project.

## 2.4.1 Setting Up for Android

* Select **Android** from the platform list and click **Switch Platform** to target Android devices.
* Configure settings under **Player Settings**:
  + Set the package name (e.g., com.companyname.productname).
  + Define the minimum and target Android API levels.
  + Configure graphics APIs like OpenGL ES3 or Vulkan.
  + Adjust resolution and presentation settings for device screens.
* If using AR Core or Unity XR plugins, ensure they are installed via **Package Manager** and configured.
* Connect an Android device or emulator to test builds via USB debugging.

## 2.4.2 Setting Up for iOS

* Select **iOS** and switch platform.
* In **Player Settings**:
  + Set the bundle identifier (e.g., com.companyname.productname).
  + Specify the minimum iOS version supported.
  + Configure orientation, resolution, and other device-specific settings.
* Install necessary XR or ARKit packages if developing AR applications.
* After building, Unity generates an Xcode project that must be compiled and deployed using Xcode.

## 2.4.3 Setting Up for XR (Extended Reality)

* XR projects require enabling XR plug-in management via **Edit > Project Settings > XR Plug-in Management**.
* Install and activate relevant XR platforms (e.g., Oculus, Windows Mixed Reality, ARKit, ARCore).
* Configure XR-specific player settings like stereo rendering mode, tracking, and input handling.
* Ensure the build target platform matches the intended XR device (Android for Quest, iOS for ARKit, etc.).
* Test XR builds on target hardware with devices connected or through simulation.

### 

**Image 2.4.1:** Unity Build Settings: Preparing Project for Android Deployment

### ****2.5 Common Components in Unity****

Components in Unity are **building blocks** that define the behaviour and properties of GameObjects. Every GameObject must have at least a **Transform component**, and additional components can be added to give it functionality.

## 2.5.1 Transform

The **Transform** component is fundamental to every GameObject in Unity. It defines the GameObject’s position, rotation, and scale in the scene, effectively determining where and how it appears in the virtual world. The Transform component also manages the parent-child hierarchy (parenting), which controls how objects move and rotate relative to each other.

## 2.5.2 Colliders

**Colliders** define the shape of an object for physical collision detection. They do not render visually but provide boundaries that detect interactions with other colliders. Unity supports various collider types, such as BoxCollider, SphereCollider, CapsuleCollider (3D), and BoxCollider2D (2D), among others. Colliders are essential for physics-based interactions like detecting hits, triggers, or obstacles.

## 2.5.3 Rigidbodies

The **Rigidbody** component enables physics simulation on GameObjects. It allows objects to respond to forces such as gravity, collisions, and applied impulses dynamically rather than being static. Adding a Rigidbody pairs with Colliders to create realistic behaviors including falling, bouncing, and sliding.

## 2.5.4 Canvas

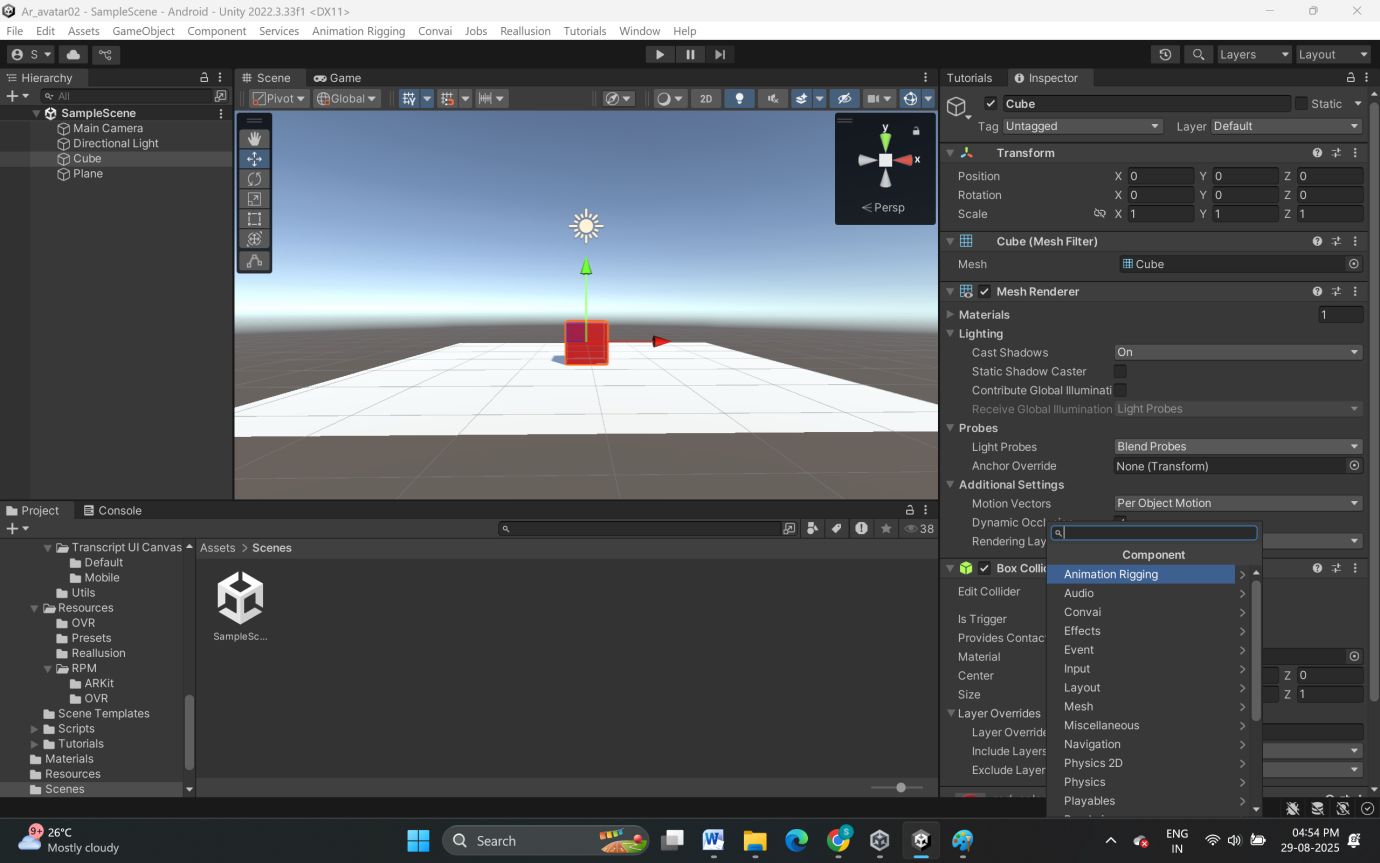
The **Canvas** component is the foundation for building user interfaces (UI) in Unity. It acts as a container for UI elements like buttons, text, and images, controlling how these elements are rendered and interact with the screen or world space. Canvas can be set to render in screen space (overlay or camera) or world space, enabling flexible UI designs.

## 2.5.5 Lighting

In Unity, lighting components control how light affects the scene and objects. Common lighting components include:

* **Directional Light:** Simulates sunlight with parallel light rays.
* **Point Light:** Emits light in all directions from a single point.
* **Spotlight:** Emits a cone-shaped light useful for focused illumination.

Lighting affects shadows, reflections, and overall ambiance, contributing significantly to the visual realism and mood of a scene.



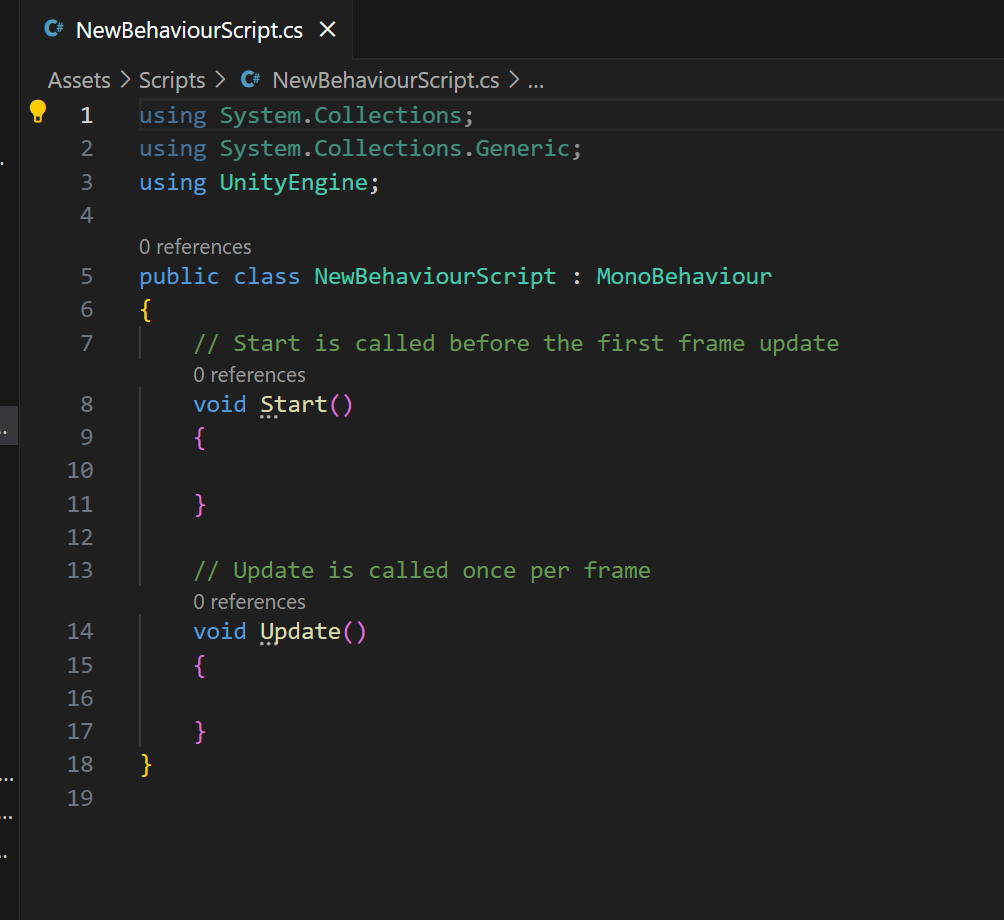
**Image 2.5.1:** Adding and Configuring Components in Unity’s Inspector Panel

**2.6 C# in Unity**

## 2.6.1 Definition and Role

In Unity, C# scripts are attached to GameObjects to create interactivity—moving objects, responding to input, detecting collisions, and managing game state. C# is chosen for its modern, object-oriented structure, wide .NET support, and ease of use for beginners and professionals.

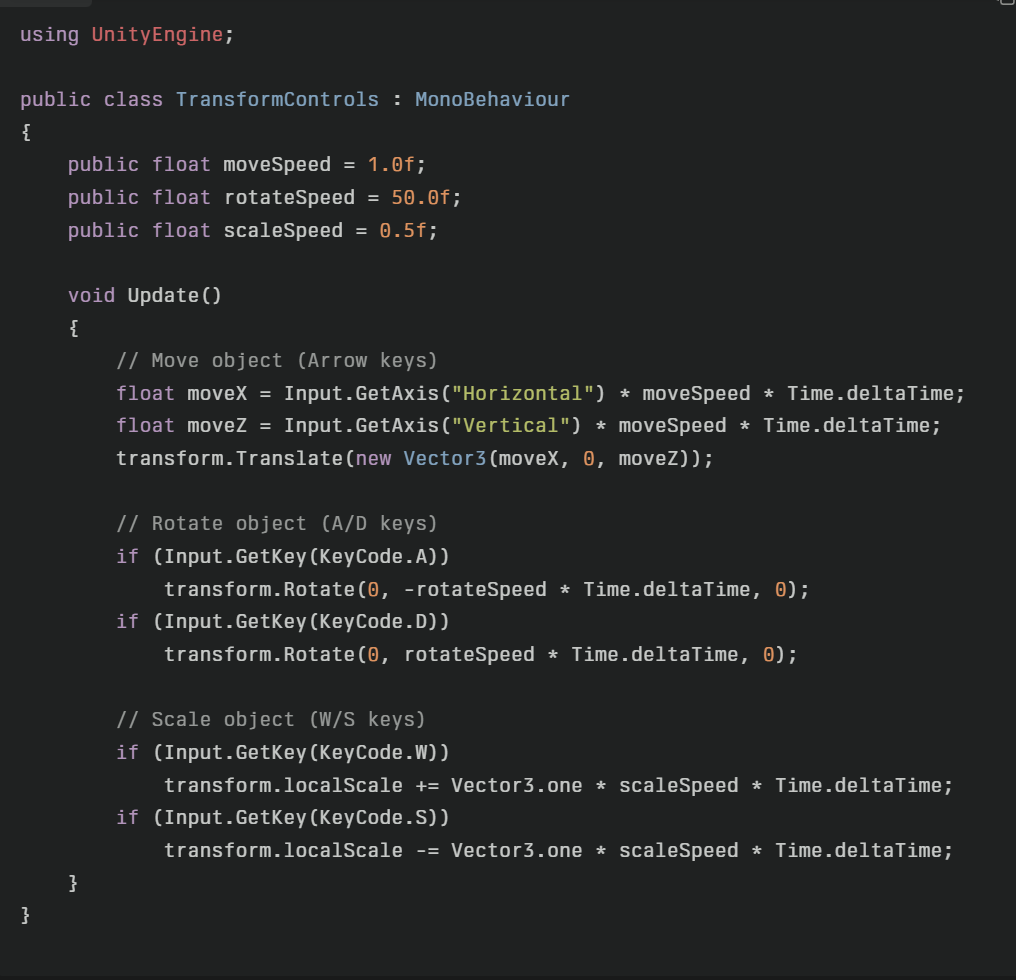
## 2.6.2 Default syntax of C# in Unity



**Image 2.6.1:** Syntax of C# script

* using System.Collections;
  + Imports the System.Collections namespace, enabling the use of collections such as lists and arrays in the script.
* using System.Collections.Generic;
  + Imports the System.Collections.Generic namespace to allow use of generic collections like List<T>, Dictionary<K,V>, etc.
* using UnityEngine;
  + Imports Unity’s core library, which contains basic game development classes such as MonoBehaviour, GameObject, and others.
* public class NewBehaviourScript : MonoBehaviour
  + Declares a new public class named NewBehaviourScript that inherits from MonoBehaviour. By inheriting MonoBehaviour, the class gains access to Unity’s event functions (e.g., Start, Update).
* { ... }
  + Encloses the body of the class where all fields and methods are defined.
* void Start()
  + Declares the Start method. This is a Unity event function called once before the first frame update (when the script instance is enabled or the scene starts).
* void Update()
  + Declares the Update method. This is a Unity event function called every frame while the script is active. It’s commonly used for checking input, updating movement, or running frame-by-frame logic.

## 2.6.3 c# script to Control a Transformation of GameObject



**Image 2.6.2:** Transformation Control C# script for GameObject