Basic Data Types

Type	Description	Example
int	Stores whole numbers from -2,147,483,648 to 2,147,483,647	`int myInt = 15;`
long	Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	`long myLong = 22L;`
float	Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits	`float myFloat = 12.86f;`
double	Stores fractional numbers. Sufficient for storing 15 decimal digits	`double myDouble = 8.66d;`
bool	Stores true or false values	`bool myBool = true;`
char	Stores a single character/letter	`char myChar = 'A';`
string	Stores a sequence of characters	`string myString = "Word"`

Basic Access Modifier

Modifier	Description	Show in Inspector
public	The code is accessible for all classes	Yes
private	The code is only accessible within the same class	No
protected	The code is accessible within the same class, or in a class that is inherited from that class	No
internal	The code is only accessible within its assembly, but not from another assembly	No

To show variables in Inspector that are not public, use [SerializeField]:

example:

```
[SerializeField]
private int showInt = 20;
```

Unity Types

Type	Description	Example
Vector2	A set of 2 float coordinates	`Vector2 myVector2 = new Vector2(1, 2);`
Vector3	A set of 3 float coordinates	`Vector3 myVector3 = new Vector2(1, 2, 3);`
GameObject	Game Objects are the base entity in Unity scenes. Each game object has a name, a transform for its position and rotation, and a list of components.	`GameObject exampleGameObject = null;`

More Data Structure

Type	Description	Example
Array	Used to store multiple values in a single variable with fixed size	`int[] myIntArray = new int[5];`
Stack<T>	Last-in First-out	`Stack<int> myIntStack = new Stack<int>();`
Queue<T>	First-in First-out	`Queue<int> myIntQueue = new Queue<int>();`
List<T>	Can store multiple items	`List<int> myIntList = new List<int>();`
Dictionary <TKey, TValue>	A collection that stores Key-Value pairs Value can be accessed by passing the Key	`Dictionary<int, string> myIntStringDictionary = new Dictionary<int, string>();`

Type	Add	Remove
Stack	myIntStack.Push(10);	myIntStack.Pop()
Push() adds to the top Pop() remove from the top		
Queue	myIntQueue.Enqueue(10);	myIntQueue.Dequeue();
Enqueue() adds to the rear Dequeue() removes from the front		
List	myIntList.Add(10); myIntList.Insert(2, 15);	myIntList.Remove(10); myIntList.RemoveAt(2);
Add() adds to the rear Insert(index, value) insert value to index Remove(value) remove the first value from the list RemoveAt(index) remove the item on the index		
Dictionary	myIntStringDictionary.Add(1, "One");	myIntStringDictionary.Remove(1);
Add(key, value) adds value to the rear Remove(key) remove the pair with key		

Basic Operators

Type	Description	Example (int a = 10, int b = 2)
+	Addition	a + b = 12
-	Subtraction	a - b = 8
*	Multiplication	a * b = 20
/	Division	a / b = 5
%	Modulus	a % b = 0
++	Increment	a ++ a = 11
--	Decrement	a -- a = 9
+=	Addition in place	a += 5 a = 15
-=	Subtraction in place	a -= 5 a = 5

Comparison Operators

Type	Description	Example	result
==	Returns true if both sides are equal	true == true	true
<	Returns true if the right side is greater than the left side	30 < 40	true
>	Returns true if the left side is greater than the right side	30 > 40	false
<=	Returns true if the right side is greater than or equal to the left side	30 <= 30	true
>=	Returns true if the left side is greater than or equal to the right side	25 >= 10	true
&&	If statement on both sides is true	true && false	false
||	If statement on either side is true	true || false	true
!	Returns true if statement is false	!false	true

Statements

Selection Statement
Decide which code to execute depending on the expression

Statement	Syntax	Description

If else	if(condition1)
	{
	  Do A;
	}
	else if(condition2)
	{
	  Do B;
	}
	else
	{
	  Do C;
	}
		If condition 1 is true, then do the A action.
		If not but condition 2 is true, then do the B action.
		If both are wrong, do the C action.

switch	switch(expression)
	{
	  case expression1:
	  Do A;
	  break;
	  case expression2:
	  Do B;
	  break;
	  default:
	  Do C;
	  break;
	}
		Depends on the expression.
		If it is expression1 then do the A action.
		If it is expression2 then do the B action.
		If none of the expression is correct then do the C action.
		
		An expression can be int, char, bool

Iteration Statement

Loop through data structures or perform a code multiple times

Statement	Syntax	Console Result

for	for(int i = 0; i < 3; i++)
	{
	    Debug.Log(i);
	}
		012

foreach	List<int> newList = new List<int>() {0, 1, 2};
	foreach(int i in newList)
	{
	    Debug.Log(i);
	}
		012

while	int i = 0;
	while(i < 3)
	{
	    Debug.Log(i);
	    i++;
	}
		012

do	int i = 0;
	do
	{
	    Debug.Log(i);
	    i++;
	} while(i < 3);
		012

Exception Handling Statement
Handle errors in code

Example	<pre>void Start()
	{
	try
	{
	CheckAge(17);
	}
	catch(UnderAgeException e)
	{
	Debug.Log("UnderAgeException: " + e.message);
	}
	finally
	{
	Debug.Log("Age is checked");
	}
	}
	
	public void CheckAge(int age)
	{
	if(age < 18)
	{
	throw(new UnderAgeException("Not old enough"));
	}
	else
	{
	Debug.Log("Old enough");
	}
	}
	public class UnderAgeException: Exception
	{
	public UnderAgeException(string message): base(message){
	}
	
	}</pre>

Console Result	UnderAgeException: Not old enough Age is checked
Explain	The function will run the CheckAge(int) function, however, was met with an UnderAgeException, which enters the catch part and prints out "UnderAgeException: Not old enough". After the try-catch was finished, the finally part will always be run, which prints out the “Age is checked”

Jump Statement

Control how the code exit and jump to other sections

Statement	Description	Example	Result

break	Breaks out of the current loop	for(int i = 0; i < 5; i++)
		{
		  Debug.Log(i);
		  if(i == 3)
		  break;
		}
			0123

continue	Starts a new iteration of the loop	for(int i = 0; i < 5; i++)
		{
		  if(i == 3)
		  continue;
		  Debug.Log(i);
		}
			0124

return	Terminates the execution of the function	public void CheckEven(int number)
		{
		  if(number%2 == 0)
		  {
		  Debug.Log(“Even”);
		  return;
		  }
		  Debug.Log(“Not even”);
		}
		Void Start()
		{
		  CheckEven(4);
		}
			Even

goto	Transfer control to a statement that is marked by a label	for(int i = 0; i < 5; i++)
		{
		  if(i == 3)
		  goto Leave;
		  Debug.Log(i);
		}
		Leave:
		  Debug.Log(“Left”);
			012Left

yield	Works with coroutine	IEnumerator TimePause(float sec)
		{
		  yield return new WaitForSeconds(sec);
		}	

Unity Functions

GameObject Lifetime

Function	Description
Awake()	• Awake is called when an enabled script instance is being loaded. • Awake is often used for GameObject **initialization**
Start()	• Start is called on the frame when a script is enabled just before any of the Update methods are called the first time. • Start will only be called once even if you disable and re-enable the GameObject again. • Start is often used for initialization. • With Awake they create a two-step **initialization** • Awake is used for initializing a GameObject's own references and variables • Start is used to create references to other objects and their components
Update()	• Update is called every frame if the MonoBehaviour is enabled. • Update is influenced by the frame rate, which is not good for physics that needs to be constant and stable. • However, Update is often used for **getting inputs** due to its often higher called rate than FixedUpdate
FixedUpdate()	• Frame-rate independent MonoBehaviour. • By default will be called every 0.02 seconds, which means 50 times per second. • FixedUpdate is often used for **physics** because of the stability
LateUpdate()	• LateUpdate is called every frame after Update if the Behaviour is enabled. • Since it is called later than Update, it is often used for **cameras that follow GameObjects** to make sure it executes after the GameObject position is decided
OnEnable()	• This function is called when the object becomes enabled and active. • OnEnable is called every time the GameObject is enabled compared to Start, which only triggers once
OnDisable()	• This function is called when the behavior becomes disabled • OnDisable is called every time the GameObject is disabled

Useful Stuff

Function	Description
GetComponent<T>()	• This is used for getting the component of a GameObect • T can be any type of component that can be found on a GameObject
transform.GetChild(int index)	• Returns a transform child by index in the Hierarchy
gameObject.Find("Name")	• Returns a transform child by name
transform.position = new Vector3(0,0,0)	• Change the GameObject position to (0,0,0)
transform.Translate(Vector3)	• Moves the object towards the Vector3 direction
Rigidbody.AddForce(Vector3 force)	• Add a force to the RigidBody
Rigidbody.Velocity = new Vector3(10,0,0)	• Give the RigidBody a velocity (10,0,0)
Time.deltaTime	• Returns the seconds passed since the last frame • Often used for a timer in Update • Every frame plus Time.deltaTime creates a timer
Instantiate(Object object, Vector3 position, Quaternion rotation)	• Spawns an object at position with rotation