VARIABLES

What: Named box that stores a value
When: You need to remember something (speed, health, name, score)

```
int score = 0;          // whole numbers
float speed = 5.5f;     // decimals
bool hasKey = false;    // true / false
string title = "Hello"; // text
var hasKey = false;     // compiler infered type*
```

💡 **Note:** *Use explicit types when learning or when the type isn't obvious.
Use var when the type is obvious or very long

⚠️ **Common Mistakes:**
• Forgetting f on float literals (ie. 5.5f)
• using vague names (x, thing, etc.)

Compound Operators

```
+= add and assign
-= subtract and assign
*= multiply and assign
/= divide and assign
++ add 1     // Pre or Post increment
-- minus 1   // Pre or Post increment
```

```
score += 5;   // score = score + 5
health -= 10; // health = health - 10
```

IF STATEMENTS & CONDITIONS

What: "If this is is true -> do this"
When: Game logic decisions (Health low, button pressed, dead/alive)

```
if (health <= 0)
{
    Die();
}
else if (health < 20)
{
    Debug.Log("Low Health!");
}
else
{
    Debug.Log("All Good.");
}
```

```
> < >= <= // math comparisons
== // equals
!= // not equal
&& // And
|| // Or
!  // Not
```

```
// Combining conditions

if (hasKey && doorLocked) {}    // AND
if (isDead || health <= 0) {}   // OR
if (!isGrounded) {}             // NOT
```

💡 **Note:** "Ask a yes / no question. if yes -> run this code block {}"

SWITCH

What: Checks one value against many possible options
When: Cleaner than many ' if ' statements when checking the same variable.

```
switch (weaponType)
{
    case "Sword":
        damage = 10;
        break;

    case "Bow":
        damage = 6;
        break;

    default:
        damage = 1;
        break;
}
```

💡 **Note:**
"Depending on what this variable is, run the matching case"

• A 'case fall through' is when more than one case points to the same block of code instead of duplicating logic. ie.

```
switch (food) {
    case Foods.Cake:
    case Foods.IceCream:
        Debug.log("Junk Food");
        break;

    case Foods.Carrots:
        Debug.log("Healthy");
        break;
}
```

⚠️ **Common Mistakes:**
• break; stops the switch and must follow every case.

FUNCTIONS (Methods)

What: A reusable block of code you can call by name
When: You repeat actions (damage, heal, spawn, play sound)

```
// Not returning data
void Jump()
{
    Debug.Log("Jump!");
}
```

```
// Returning Data
int CalculateDamage(int baseDamage, int bonus)
{
    return baseDamage + bonus;
}
```

```
//Calling them
// No data
Jump();
```

```
// Data
int damage = CalculateDamage(10,5);
enemyHealth -= damage;
```

💡 **Rules:**
• void returns nothing (runs a function but does not return a value)
• Any other type must return something.

LOOPS

What: Repeat something multiple times

for
• Is a counter with controlled repetition (until a rule says stop)

```
for (START ; CONDITION ; STEP)
    START → CHECK → RUN → STEP
        ↑              ↓
        └── repeat ──┘
```

```
// Create a counter from 0, if less than 5, add 1
// ' i ' is just a counter it could be anything.
for (int i = 0; i < 5; i++)
{
    Debug.Log(i);
}
```

foreach
• Run for every item in a collection (Automatic looping for all items)

```
foreach (Type item in collection)
{
    // runs once per item
}
```

```
//example - for each number, print it in console
List<int> numbers = new List<int>() { 1, 2, 3 };

foreach (int number in numbers)
{
    Debug.Log(number);
}
```

while
• Condition-based looping, loop while 'true' (unknown count)

```
while (ammo >0)
{
    Shoot();
    ammo--;
}
```

do while
• Code executes once and then checks condition at the end

```
do
{
    Debug.Log("Runs once no matter what");
}
while (false);
```

⚠️ **Common Mistakes:**
• Creating an infinite loop will crash the game, an example is a condition that is always true. ie.

```
for (int i = 0; true; i++)
{
    Debug.Log("Infinte Loop!")
}
```

Data Structures

What: Containers holding multiple values.
When: Inventory, enemies, spawn points, scores, waypoints

Array
• Size is decided once and cannot change.
• indexing starts at postion [0]
• unassigned value use default values (int = 0) or (string = null)
• Arrays can have empty slots

```
type[] name = new type[size];
```

Example.
```
int[] scores = new int [3];

scores[0] = 10;
scores[1] = 20;
```

• Instead of specifying the size of an array, a shortcut is:
```
int[] scores = {10, 20, 30};
```

• Get the size of an array using .Length
```
for (int i = 0; i < scores.Length; i++)
{
    Debug.Log(scores[i]);
}
```

💡 **Note:** Arrays are fast, but inflexible

Lists
• Can grow and shrink in size
• indexing also starts at position [0]
• Lists only store what you add (no empty slots)

```
List<string> items = new List<string>();
items.Add("Apple");
items.Add("Bread");

//Acces by index
Debug.Log(items[0]); // Apple
```

List Operations:
```
items.Count              // How many items
items.Add("Milk");       // Add item 'Milk'
items.Remove("Apple");   // Remove item 'Apple'
items.RemoveAt(0):       // Remove item @ index [0]
items.Clear();           // Clear ALL items
items.Contains("Bread");// Check if item 'Bread' exists
```

CLASSES

What: A 'blueprint' that groups data (variables) and behavior (functions) together. <u>Almost every script in Unity is a class.</u>
When: Players, enemies, items, weapons, managers, controllers etc.

```csharp
public class Player
{
    public int health = 100;

    public void TakeDamage(int damage)
    {
        health -= damage;
    }
}
```

Breakdown of the above:
Player → the type
health → data the player has
TakeDamage() → things the player can do

Unity Script Class

In unity if we want to attach a script to an scene object we need to add the MonoBehaviour to the class declaration which;
- Allows unity to attach the script to GameObjects
- Call start(), and Update()

```csharp
public class PlayerController : Monobehaviour
{
    void Start()
    {
        Debug.Log("Game Started");
    }

    void Update()
    {
        //Code runs every frame
    }
}
```

💡 **Note:** **.cs** script name must match the class name

CONSTRUCTORS

What: A special method that runs automatically when a class is created
Why: to set up initial values when an object is made.
- Has the same name as the class
- Has no return type (not even void)
- Runs <u>once</u>, automatically on creation

```csharp
public class Player
{
    public int health;

    public Player()
    {
        health = 100;
    }
}
```

Breakdown of the above:
Player() is the constructor
When a Player is created → health is set to 100

CONSTRUCTORS [CONTINUED]

Constructors with parameters

By adding external parameters to our constructors we can decide values when the object is create.

💡 **Note:** Think different health values for different enemy types

```csharp
public class Enemy
{
    public int health;
    public Enemy(int startHealth)
    {
        health = startHealth;
    }
}

// Creating the object
Enemy weakEnemy = new Enemy(50);
Enemy strongEnemy = new Enemy(200);


public class Item
{
    public string name;
    public int damage;

    public item(string name, int damage)
    {
        this.name = name;
        this.damage = damage;
    }
}

// creates the item in C# memory
item sword = new Item("Sword",100);
```

⚠️ No Constructors for : MonoBehaviour ⚠️

Unity's MonoBehaviour class does <u>NOT</u> use constructors, instead it uses the following;

```csharp
// Scene Loads
↓
// GameObjects are created
↓
Awake() // Before first frame is run
↓
Start() // Game Starts - first frame
↓
Update() // run each frame of the game
```

💡 **Note:** Awake() purpose is 'internal setup' anything that is needed during Start() is already instantiated and ready.

💡 **Note:** Constructors <u>DO</u> matter in Unity for plain C# classes (non-MonoBehaviour) ie. Data Classes, Inventory Items, Stats