v0.3

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. 5.5*f*)
- 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
```

COMMENTS

What: Sometimes it is handy to annotate within your code. Comments are ignored by the compiler and do not affect the code.
When:
- Explain intent or reasoning
- Clarify complex logic
- Temporarily disable code

```
// this is a comment

/* this
    is a multi
    line comment
*/
```

Common Mistakes:
- Do not clarify obvious code, it unessesarily bloats the code
- Do not leave outdated comments

CODE BLOCKS

What: Code blocks are the building blocks of scripts, they are sections of code wrapped in { } that run as a unit.

Why: They:

- Control when code runs
- Defines scope for variables
- Group related logic

```
for(int i = 0; i < 5; i++)
{
    //code block
}

public class Player
{
    //code block
}
```

SCOPE

What: Defines where a variable can be accessed and how long it exists.

Rules:

- A variable exists only inside the { } where it is declared
- Inner (child) blocks CAN access variables from outer (parent) blocks (FIG.1)
- Outer blocks CANNOT access variables declared in inner (child) blocks (FIG.2)

FIG.1

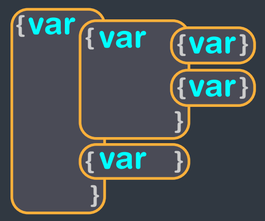
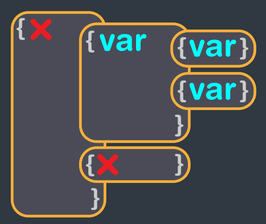


FIG.2




```
void CheckHealth()
{
    int playerHealth = 100;

    if(playerHealth < 150)
    {
        int damage = 25;
        Debug.Log(playerHealth); // ✅ Works
        Debug.Log(damage);       // ✅ Works
    }

    Debug.Log(playerHealth);     // ✅ Works
    Debug.Log(damage);           // ❌ ERROR
}
```

Note: Varriables live and die with their code block { }

CLASSES

What: A 'blueprint' that groups data (variables) and behavior (functions) together. Almost every script in Unity is a class.

When: Players, enemies, items, weapons, managers, controllers etc.

```
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()

```
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 once, automatically on creation

```
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

```
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 NOT use constructors, instead it uses the following;

```
// 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 DO matter in Unity for plain C# classes (non-MonoBehaviour) ie. Data Classes, Inventory Items, Stats

NAMING RULES

What: These are rules that you follow in naming your various symbols (functions, variables, properties, etc. They are VERY important. Following a consistent set of rules is the easiest way to improve the qulaity and clarity of your code.

Note: There are no 'best' naming rules, the most important thing is that you are consistent.

How: There are three primary case 'types':

PascalCase
camelCase
snake_case

Convention: - this is not 'the' way but a reccomended standard.

Constants

- UPPERCASE
- snake_case

```
public const int CONSTANT_FIELD = 56;
```

Properties

- PascalCase

```
public static MyCodeStyle Instance { get; private set; }
```

Events

- PascalCase

```
public event EventHandler OnSomethingHappened;
```

Fields

- camelCase

```
private float meberVariable;
```

Classes

- PascalCase

```
public class PlayerController
```

Function Names

- PascalCase

```
private void Awake()
{
Instance = this;
DoSomething(10f);
}
```

Function Params

- camelCase

```
private void MovePlayer(float moveSpeed, float deltaTime)
{
    float distanceMoved = moveSpeed * deltaTime;
}
```

Note:

- Spend time deciding ona proper name!
- Don't be afraid to rename things
- Don't use single letter names
- Don't use acronyms or abbreviations

STATIC

What: static means a variable or method belongs to the class itself, as is accessed through the class. non-static variables belong to the instance (object) and is called through the instance.

Why:
- There should only be one shared value ie. 'enemyCount'
- the data does not belong to a specific object
- you need global access without a reference to an object.
- static variables do NOT appear in the inspector

Note: Non-static is the default - static must be explicitly declared

Non-Static (per instance)

- Requires an object instance (ie. player)
- instance.variable (ie. player.playerName)

```
public class Player
{
    public string playerName;
}

Player p1 = new Player();
Player p2 = new Player();

p1.playerName = "Kronk"
p2.playerName = "Cuzco"
```

Note: Each player has their own name, specific to each instance of the player

Static (shared by class)

- No instance required, accessed through the class
- ClassName.variable (ie. GameManager.enemyCount)

```
public class GameManager
{
    public static int enemyCount;
}

GameManager.enemyCount++;
```

Note: There is only one enemyCount, shared by all.

Common Mistakes:
- Static methods CANNOT access non-static members. vice versa a non-static (instance) method CAN access static fields and static methods in the same class or other classes).

```
public class Player
{
    public static int totalPlayers;
    public string playerName;

    public void Register()
    {
        totalPlayers++;  // ✅ allowed
        Debug.Log(totalPlayers);
    }
}
```

```
public class Player
{
    public int health;

    public static void PrintHealth()
    {
        Debug.Log(health); // ❌ ERROR
    }
}
```

- Static methods have no instance
- health belongs to an instance
- Theres no specific object to read from

STATIC [CONTINUED]

Static Classes

A static class is like a 'tool box, not an object', and can only contain static members

- Cannot be instantiated (no new)
- All members MUST be static
- Cannot inherit from MonoBehaviour
- Used for utilities and shared logic

```
public static class MathUtils
{
    public static int Add(int a, int b)
    {
        return a + b;
    }
}
```

ACCESS MODIFIERS

What: Access modifiers control access between classes, not within a class.

Why:
- Protect data from unintended changes
- Define clear boundaries between scripts
- Make code safer and easier to maintain

public

- Accesible from anywhere
- Visible in the Inspector
- Most permissive

Use When: You want other scripts, functions, or the inspector to access it

```
public int health;
```

private

- Accesible only inside this class
- NOT visible in the inspector
- Default if no modifier is specified

Use When: The variable is internal logic and shouldn't be touched externally

```
private int damage;
int damage;          // this is already private(default)
```

protected

- Accesible in this class AND child classes
- NOT accesible from unrelated scripts

Use When: You expect other classes to inherit from this one.

```
protected int mana;
```

internal

- Accesible only within this project (essentially similar to public)

Use When: not entirely sure? maybe mod support?

```
internal int score;
```

[SerializeField]

In unity if we want a private variable to be saved by unity and visible & editiable in the inspector we need to add [serializeField] before the access modifier.

Note: Editable in Unity, protected in code

```
[SerializeField] private float speed;

private          // hidden from other scripts
[SerializeField] // visible in Inspector
```

ACCESS MODIFIERS [CONTINUED]

Example: Define a private int field for 'speed' and then 2 public functions to Get and Set that field.

```
public class Exercise : MonoBehaviour {

    private int speed;

    public int GetSpeed()
    {
        return speed;
    }


    public void SetSpeed(int speed)
    {
        this.speed = speed;
    }

}
```

CLEAN CODE GUIDELINES

What: Clean code generally is; easy to read, easy to understand and easy to modify. Some general principals are;

Group Related Logic / Data

What: Rather than defining separate fields, create a custom type that groups similar data together. ie.

❌
```
//Instead of 3 separte fields:
    private int strength;
    private int dexterity;
    private int wisdom;
```

✅
```
// Create a custom class
private PlayerStats PlayerStats;

public class PlayerStats
{
    private int strength;
    private int dexterity;
    private int wisdom;
}
```

Single Responsibility Principle

What: Functions(Methods) should generally try to perform one task. This avoids unessesarily long and complex functions that do many things, keeping the code easier to read and understand.

✅
```
public void DoSomething()
{
    // ...
}
```

❌
```
public void MovePlayerAndTestAttackAndEtc()
{
    // long function doing too many things
    // long function doing too many things
    // long function doing too many things
}
```

CLEAN CODE GUIDELINES [CONTINUED]

Don't Repeat Yourself (DRY)

What: If there is code you have to run multiple times, it should probably have a dedicated Function vs. copy pasting the code multiple times.

Avoid 'Magic Numbers'

What: 'Magic Numbers' are numbers that show up in the code that make it really tricky to understand what exactly the number represents. ie.

❌
```
// What does 5 represent?? Who knows
if (Distance(playerPosition, enemyPosition) < 5) {}
```

✅
```
// Whereas in this code we can define it
int attackDistance = 5;
if (Distance(playerPosition, enemyPosition) < attackDistance) {}
```

Avoid strings as identifiers

What: strings are very brittle and easy to misstype errors which are then difficult to debug. Whereas if you have a proper object reference, if you mistype the reference you get a very clear compiler error.

❌
```
//All look almost identical
"PLAYER10"
"PLAY3R10"  //3 for E
"PLAYER10"  //L for 1
"PLAYER1O"  //O for 0
```

✅
```
//Whereas an Object Reference
Player player10 = new Player();
Player10.DoSomething(); // 'Player10' does not exist
player10.DoSomething(); // 'player10' does not exist //(L)
player1O.DoSomething(); // 'player1O' does not exist //(O)
```

Refactoring

What: Improving the structure, readability or organization of code without changing what it does.This should happen organically as your code grows.

Why:
- Makes code easier to read and understand as it evolves
- Reduces bugs and duplication
- Makes future changes safer and faster