Orange Ninja Team Coding Standards

Version 1.0 — October 2025

> "The point of having style guidelines is to have a common vocabulary of coding so people can concentrate on what you are saying, rather than on how you are saying it."
> — *Google C++ Style Guide*

---

1. Purpose

This document defines the **coding standards** used by the *Orange Ninja* development team.
Our goals are:

- Consistency across all scripts
- Readable and maintainable code
- Fewer merge conflicts
- Easier collaboration and debugging

All Unity C# scripts (Player, Enemy, UI, Camera, etc.) should follow this style guide.

---

2. Naming Conventions

General Rules

- Use **descriptive names** that reflect purpose.
- Avoid single-letter names except for short-lived iterators (`i`, `j`, `k`).
- Use **English words** and avoid abbreviations unless well known (`UI`, `ID`).

Variables

- Use **camelCase** for variables.
- Prefix Boolean variables with **is**, **has**, or **can**.
- Use postfix like `_Count` or `_Speed` for measurements or counters.
- Use an underscore `_` prefix for private serialized fields visible in the Inspector.

Examples

```
[SerializeField] private float _moveSpeed;
int enemyCount;
float jumpForce;
bool isGrounded;
```

Constants

- Write constants in **ALL_CAPS** with underscores.

```
const float MAX_SPEED = 10f;
const int MAX_HEALTH = 100;
```

Classes, Structs, Enums

- Use **PascalCase**.

```
public class PlayerController { }
public struct PlayerStats { }
public enum PlayerState { Idle, Moving, Jumping }
```

Methods and Functions

- Use **PascalCase**.
- Function names start with a **verb** that describes the action.

```
void HandleInput() { }
void ApplyDamage(int amount) { }
bool IsAlive() { return health > 0; }
```

---

3. Commenting Style

File Header Comment

Every file must start with a header block:

```
/*****************************************
 * File: PlayerController.cs
 * Author: Orange Ninja Team
 * Description: Handles player input, jumping,
 * attacking, and damage for the Orange Ninja game.
 *****************************************/
```

Function Header

Use XML-style comments for methods (visible in IntelliSense):

```
/// <summary>
/// Handles all player inputs such as movement,
/// jumping, and attacking.
/// </summary>
void HandleInput() { ... }
```

Inline Comments

Use `//` for short explanations.

```
// Apply horizontal movement based on input
rb.linearVelocity = new Vector2(move * moveSpeed, rb.linearVelocity.y);
```

Only comment *why* something happens, not *what* obvious code does.

---

4. Indentation and Formatting

- **4 spaces** per indentation level (no tabs).
- **Braces** `{}` on the same line as the condition or method.
- **One blank line** between methods.

Example

```
void Update() {
    HandleInput();

    if ( !playerData.IsAlive() ) {
        Die();
    }
}
```

Conditional Formatting

```
if ( condition ) {
    // statements
}
else if ( otherCondition ) {
    // statements
}
else {
```

```
    // statements
}
```

Loop Formatting

```
for ( int i = 0; i < enemyCount; i++ ) {
    // iterate through enemies
}
```

---

5. Error Handling

- Use `Debug.Assert()` for logic validation.
- Use `Debug.LogError()` or `Debug.LogWarning()` for runtime errors.
- Never leave an empty `catch` block.

Example

```
void Start() {
    rb = GetComponent<Rigidbody2D>();
    Debug.Assert(rb != null, "Rigidbody2D missing on Player!");
}

void ApplyDamage( int amount ) {
    Debug.Assert(amount >= 0, "Damage must be non-negative!");
}
```

General Rules

- Prevent errors instead of catching them.
- Always check for `null` before using components.

---

6. Unity Lifecycle Usage

Method	Purpose
`Awake()`	Initialize references and singletons
`Start()`	Initialize gameplay variables
`Update()`	Handle per-frame logic
`FixedUpdate()`	Handle physics updates

Method	Purpose
`OnDrawGizmosSelected()`	Debug visuals (attack range, etc.)

Example

```
void Awake() {
    rb = GetComponent<Rigidbody2D>();
}

void Start() {
    playerData = new Player(100);
}

void FixedUpdate() {
    HandleMovement();
}
```

---

7. File and Folder Organization

Project Folder Structure

```
Assets/
 ├── Scripts/
 │    ├── Player/
 │    │    ├── Player.cs
 │    │    ├── PlayerController.cs
 │    │    └── FollowPlayer.cs
 │    ├── Enemy/
 │    │    └── EnemyController.cs
 │    └── UI/
 │    │    └── HealthBar.cs
 │    └── Levels /
 │    └── Audio /
 └── Tests/
      └── TeamLead1/
      └── TeamLead2/
      └── TeamLead3/
      └── TeamLead4/
            └── PlayerStressTest.cs
      └── TeamLead5/
```

Rules

- One public class per file.
- File name must match the class name exactly.
- Separate gameplay and testing code into different folders.

---

8. Debugging and Logging

- Use `Debug.Log()` for general messages.
- Use `Debug.LogWarning()` for potential issues.
- Use `Debug.LogError()` for critical problems.
- Remove or comment out logs before the final release.

Example

```
Debug.Log("Player Jump Successful");
Debug.LogWarning("Player out of bounds!");
Debug.LogError("Missing Rigidbody2D component!");
```

---

9. Example of Clean Code

```
/// <summary>
/// Applies damage to the player and checks for death.
/// </summary>
public void ApplyDamage(int amount) {
    playerData.TakeDamage(amount);
    Debug.Log("Player took " + amount + " damage.");

    if ( !playerData.IsAlive() ) {
        Die();
    }
}
```

---

10. Summary Table

Category	Rule
Variable Names	camelCase (`moveSpeed`, `isGrounded`)
Constants	ALL_CAPS with underscores (`MAX_HEALTH`)

Category	Rule
Classes & Methods	PascalCase (`PlayerController`, `HandleInput()`)
Indentation	4 spaces
Braces	Same line
Comments	Header + inline + XML summary
Error Handling	`Debug.Assert`, `Debug.LogError`
Unity Rules	Awake → Start → Update → FixedUpdate