Coding Standards

Standards are important to keep a codebase clean, readable, and consistent amongst a team of developers. It's common for different projects and development teams to have different preferences when it comes to how code is structured and formatted. However, the code within a given project should be consistent.

When working on a team project, your opinion is not considered. Follow the spec or best practices to the best of your ability. Some languages include tooling that will auto-format to a pre-defined set of rules… These tools are great, as they remove any opinions or arguments.

Sometimes, a team might like to be a bit more explicit.

The below guidelines can be helpful for code reviews, it provides us with an understanding of how to structure your code, and what things you should identify when reviewing another person's code within the team.

This is not an Exhaustive list, and some rules can be broken. This is meant as a guideline to help you develop good judgment.

Contents

Coding Standards - Naming Conventions

When naming classes, methods, or variables, they should follow some best practices.

Rule: Use Correct Casing

Your program should aim to be consistent with naming conventions and code stye. Sometimes you may work with 3rd party library that differs, and that's ok, but your own projects should aim for internal consistency.

Type	**PascalCase**	**camelCase**	**UPPER_CASE**
Class name	`public class GameStateManager`	N/A	N/A
Method Name	`Public int GetStatus()`	N/A	N/A
Method Parameters	N/A	`int SetHealth(int newHealth)`	N/A
Local variables	N/A	`bool isAlive;`	N/A
Constants	N/A	N/A	`const int NUM_ROWS = 5;`

Rule: Boolean variables should sound read as a yes/no questions

The name for Boolean variables and properties should be in the form of a yes/no questions.

Consider

```
bool isDog = true;
```

```
bool canMove = false;
```

```
bool hasWeapon = true;
```

Rule: Arrays / Collections should be named as plural, not singular

The name for an array, list of other collection should always be plural. This makes the singular and collection form easier to identify, and our code easier to understand.

Consider

```
string[] names = new string[] {"Bob", "Fred", "Ted" };

foreach(string name in names)
    Console.WriteLine(name);
```

Rule: Functions / Methods should be verbs

A function or method implies that something is going to happen, or be calculated. Functions should usually be named in the form of a verb, or verb + noun

Correct	Avoid

`int AddNumbers(int a, int b)`
`{`
`    // ...`
`}`
	`int Numbers(int a, int b)`
	`{`
	`    // ...`
	`}`

`void SetHealth(int newHealth)`
`{`
`    // ...`
`}`
	`void Health(int newHealth)`
	`{`
	
	`}`

`int[] CalculateFibonacci(int n)`
`{`
`    // ...`
`}`
	`int[] Fibonacci(int n)`
	`{`
	`    // ...`
	`}`

`Todo: insert better example`	

Rule: Use predefined data types

Avoid using System data types – use Predefined data types

Correct	Avoid
`int employeeId;`	`Int32 employeeId;`
`string employeeName;`	`String employeeName`
`bool isActive;`	`Boolean isActive`

Rule: Align curly braces vertically

Always align curly braces vertically

Correct	Avoid

`public class PlayerController`
`{`
`    public int GetStatus()`
`    {`
`        // ...`
	`public class PlayerController {`
	
	`    public int GetStatus()`
	`      {`
	`        // ...`

```
    }
}
```

```
    }
}
```

Rule: Specify protection level

Always explicitly specify protection level. Don’t rely on the default.

Correct

```
public class SceneObject
{
    private List<SceneObject> Children {get; set;}

    // ...
}
```

Avoid

```
public class SceneObject
{
    List<SceneObject> Children {get; set;}

    // ...
}
```

Rule: Structure of class variables, properties and methods

Order of content in classes.

- Public Variables and properties
- Protected/Private variables and Properties
- Constructors
- Public Methods
- Private Methods

Correct

```
public class SceneObject
{
    // 1. public variables and properties
    public Mat3 Transform get; set;
    public Mat3 GlobalTransform { get; set; }


    // 2. private variables and properties
    private List<SceneObject> Children {get; set;}
    private SceneObject Parent { get; set; }


    // 3. Constructors
    public SceneObject()
    {
        // ...
    }

    public SceneObject(SceneObject parent)
    {
        // ...
    }

    // 4. Public Methods
    public void Update()
    {
        // ...
    }

    public void Draw()
    {
        // ...
```

Avoid

```
public class SceneObject
{
    // 3. Constructors
    public SceneObject()
    {
        // ...
    }

    public SceneObject(SceneObject parent)
    {
        // ...
    }


    // 2. private variables and properties
    private List<SceneObject> Children {get;
set;}
    private SceneObject Parent { get; set; }


    // 1. public variables and properties
    public Mat3 Transform get; set;
    public Mat3 GlobalTransform { get; set; }


    // 5. Private Methods
    private DebugDraw()
    {
        // ...
    }
    }

    public void AddChild(SceneObject child)
    {
        // ...
    }

    // 5. Private Methods
    private DebugDraw()
    {
        // ...
    }
}
```

```
    // 4. Public Methods
    public void Update()
    {
        // ...
    }

    public void Draw()
    {
        // ...
    }

    public void AddChild(SceneObject child)
    {
        // ...
    }

}
```

Rule: Keep methods simple – Example 1:

Keep Load, Update and Draw methods as small and readable as posable. Break Code int smaller private methods to assist with readability.

Correct

```
void Draw()
{
    Raylib.BeginDrawing();

Raylib.ClearBackground(Color.DARKGRAY);

    DrawPlayer();
    DrawBullets();
    DrawAsteroids();

    Raylib.EndDrawing();
}

private void DrawPlayer()
{
    player.Draw();
}

private void DrawBullets()
{
    // draw all bullets
    for (int i = 0; i < bullets.Length;
i++)
    {
        if (bullets[i] != null)
        {
            bullets[i].Draw();
        }
    }
}

private void DrawAsteroids()
{
    // draw all asteroids
    for (int i = 0; i < asteroids.Length;
i++)
```

Avoid

```
void Draw()
{
    Raylib.BeginDrawing();
    Raylib.ClearBackground(Color.DARKGRAY);

    player.Draw();

    // draw all bullets
    for (int i = 0; i < bullets.Length; i++)
    {
        if (bullets[i] != null)
        {
            bullets[i].Draw();
        }
    }

    // draw all asteroids
    for (int i = 0; i < asteroids.Length; i++)
    {
        if (asteroids[i] != null)
        {
            asteroids[i].Draw();
        }
    }

    Raylib.EndDrawing();
}
```

```
    {
        if (asteroids[i] != null)
        {
            asteroids[i].Draw();
        }
    }
}
```

Rule: Keep methods Simple – Example 2:

Keep Load, Update and Draw methods as small and readable as posable. Break Code int smaller private methods to assist with readability, Ideally, each method has a single responsibility.

Correct

```
void Update()
{
    UpdateAsteroidSpawning();

    UpdatePlayer();
    UpdateBullets();
    UpdateAsteroids();

    CheckBulletAsteroidCollisions();
}

private void UpdateAsteroidSpawning()
{
    asteroidSpawnCooldown -=
Raylib.GetFrameTime();
    if (asteroidSpawnCooldown < 0.0f)
    {
        SpawnNewAsteroid();
        asteroidSpawnCooldown = 5.0f;
    }
}

private void UpdatePlayer()
{
    player.Update();
}

private void UpdateBullets()
{
    // update all bullets
    for (int i = 0; i < bullets.Length; i++)
    {
        if (bullets[i] != null)
        {
            bullets[i].Update();
        }
    }
}

private void UpdateAsteroids()
{
    // update all asteroids
    for (int i = 0; i < asteroids.Length; i++)
    {
        if (asteroids[i] != null)
        {
            asteroids[i].Update();
        }
    }
}
```

Avoid

```
void Update()
{

    // Update Asteroid Spawning
    asteroidSpawnCooldown -=
Raylib.GetFrameTime();
    if(asteroidSpawnCooldown < 0.0f)
    {
        SpawnNewAsteroid();
        asteroidSpawnCooldown = 5.0f
    }

    // Update Player
    player.Update();

    // update all bullets
    for(int i=0; i<bullets.Length; i++)
    {
        if( bullets[i] != null)
        {
            bullets[i].Update();
        }
    }

    // update all asteroids
    for (int i = 0; i < asteroids.Length;
i++)
    {
        if (asteroids[i] != null)
        {
            asteroids[i].Update();
        }
    }

    // check all bullets against all
asteroids
    foreach(var bullet in bullets)
    {
        foreach( var asteroid in asteroids)
        {

DoBulletAsteroidCollision(bullet,
asteroid);
        }
```

```
private void CheckBulletAsteroidCollisions()
{
    // check all bullets against all asteroids
    foreach (var bullet in bullets)
    {
        foreach (var asteroid in asteroids)
        {
            DoBulletAsteroidCollision(bullet,
asteroid);
        }
    }
}
```

```
    }
}
```

Rule: Constructors should explicitly invoke base class constructor

A Class should explicitly invoke base class constructors. Each class should be responsible for initialising its own member variables to the default. Do not implement derived classes member variable to default.

Correct

```
class Person
{
    public string name;

    public Person()
    {
        name = "";
    }

    public Person(string name)
    {
        this.name = name;
    }
}

class Doctor : Person
{
    public Doctor() : base()
    {

    }

    public Doctor(string name) : base(name)
    {
    }
}
```

Avoid

```
class Person
{
    public string name;

    public Person()
    {
        name = "";
    }

    public Person(string name)
    {
        this.name = name;
    }
}

class Doctor : Person
{
    public Doctor()
    {

    }

    public Doctor(string name)
    {
         this.name = name;
    }
}
```

Rule: Avoid Branching where posable

Branches are often introduced unnecessarily, consider the below examples.

Correct

```
bool IsAlive()
{
```

Avoid

```
bool IsAlive()
{
    if (health > 10)
    return health > 10;
}
```

```
        return true;

    return false;
}
```

Rule: Pure Functions should be declared static

Pure functions take some input and return some output based on that input. They are the simplest reusable building blocks of code in a program. A Pure function does not access member variables, global variables or other data other than from the provided parameters.

Pure functions should be declared static.

Correct

```
class MathHelper
{
    public static int AddNumbers(int a, int b)
    {
        return a + b;
    }
}

// somewhere else in code
int result = MathHelper.AddNumbers(10, 20);
```

Avoid

```
class MathHelper
{
    public int AddNumbers(int a, int b)
    {
        return a + b;
    }
}

// somewhere else in code
MathHelper helper = new MathHelper();
int result = Helper.AddNumbers(10, 20);
```

Rule: Prefer Property over Simple Getter and Setter methods

For variables with simple getters and setters, prefer to use C# Properties instead. The generated code is similar.

Correct

```
class Player
{
    public float Health {get; private
set;}

    // ...
}
```

Avoid

```
class Player
{
    private float health = 100;

    private void SetHealth(float
value)
    {
        health = value;
    }

    public float GetHealth()
    {
        return health;
    }

    // ...
}
```
