

# Coding Standards

## File Formatting

### *Indentation*

Each level of code should use four spaces of indentations. Either configure the tab key in your editor to equal four spaces or avoid using tabs altogether.

### *Libraries*

Library imports should be arranged alphabetically (Figure 1). The top-level library should appear first when importing multiple sub libraries (Figure 2).

```
using System.Collections;  
using System.Collections.Generic;  
using UnityEngine;
```

*Figure 1: Alphabetize libraries*

```
using System.Collections;  
using System.Collections.Generic;  
using UnityEngine;  
using UnityEngine.SceneManagement;  
using UnityEngine.UI;
```

*Figure 2: Top-level libraries appear first*

### *Placing Braces*

When initializing and calling functions, there should not be a space between the function name and parameter tuple. For conditional statements, there should be a space between the keyword (e.g. for, while) and the parentheses. See Figure 3 for example.

Function definitions should place the open and closing curly brace on separate lines. For conditional statements, the open curly brace should be one space apart from the closing parentheses; the closing curly brace should be on a separate line. See Figure 3 for example.

```
void Print(string name)  
{  
    if (name == "Lakerunner") {  
        Debug.Log(name);  
    }  
}
```

*Figure 3: Placement of braces*

### *Spacing*

Developers should follow the following spacing convention...

- Place one blank line after library imports.
- Place one blank line after the closing curly brace of function and class definitions.
- Place one blank line between private and public variable initializations.
- Place a space before and after operators (e.g. `'2 == x'`, `'4 / 2'`).

## Naming Conventions

### *Classes*

Classes should use the PascalCase naming scheme where the first letter of each word is capitalized. No special characters should be included in the class name. The name of the file should be the same as the name of the class it defines. Enum and struct types should also follow this naming convention.

### *Functions*

Functions should also follow the PascalCase naming scheme where the first letter of each word is capitalized. No special characters should be included in the class name.

### *Variables*

Variable names should follow the camelCase naming scheme where the first letter of each word is capitalized, except for the first word.

### *Files*

C# files should always be identical to the class defined within it. If the file contains multiple classes, then it should reflect the name of the most relevant class; this class should be defined within the file first.

Non-C# files should follow the PascalCase naming scheme where the first letter of each word is capitalized.

### *Directories*

Directories should follow the PascalCase naming scheme where the first letter of each word is capitalized. No special characters or numbers should be included.

## Documentation

### *Files*

File comments should be present at the top of every file, before any library imports or code. They should follow the format as shown in Figure 4.

```
/*  
 * Filename: FileName.cs  
 * Developer: Your Name  
 * Purpose: Brief explanation of this file.  
 */
```

*Figure 4: File comments*

### *Classes*

Class comments should be placed directly before the class definition. The first line should be a brief summary and follow the format as shown in Figure 5. Omit the member variables section when no member variables are present.

### *Functions*

Function comments should be placed directly before the function declaration as shown in Figure 6. Comments are not necessary for Unity function (e.g. Start(), Update()). Omit the parameters section if no parameters are present. Omit the returns section if the function returns void.

```
/*  
 * A brief description of the class.  
 *  
 * Member variables:  
 * x -- int to hold an integer value.  
 */  
public class ClassName  
{  
    int x;  
}
```

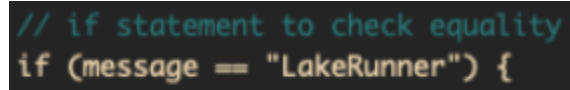
*Figure 5: Class comments*

```
/*  
 * A brief description of the function.  
 *  
 * Parameters:  
 * message -- string the holds the message.  
 *  
 * Returns:  
 * bool -- true if the message is "LakeRunner".  
 */  
bool CheckMessage(string message)  
{  
    if (message == "LakeRunner") {  
        return true;  
    }  
  
    return false;  
}
```

*Figure 6: Function comments*

### *Additional Comments*

Additional comments can be placed inline using the ‘//’ comment style as shown in Figure 7.

A screenshot of code with a dark background. The first line is a comment: `// if statement to check equality`. The second line is an if statement: `if (message == "LakeRunner") {`.

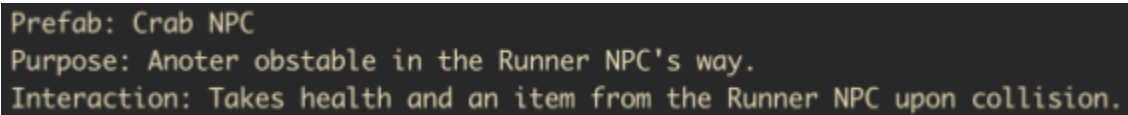
```
// if statement to check equality
if (message == "LakeRunner") {
```

*Figure 7: Inline comments*

### *Prefabs*

Prefabs should be documented with a “.txt” file in the same directory as the prefab it describes.

Text files should follow the format as shown in Figure 8.

A screenshot of a text file with a dark background. It contains three lines of text: 'Prefab: Crab NPC', 'Purpose: Anoter obstable in the Runner NPC's way.', and 'Interaction: Takes health and an item from the Runner NPC upon collision.'.

```
Prefab: Crab NPC
Purpose: Anoter obstable in the Runner NPC's way.
Interaction: Takes health and an item from the Runner NPC upon collision.
```

*Figure 8: Prefab text file content*

## **Other Formatting**

### *Files*

When formatting their code, developers should follow a maximum of about 100 characters per line.

### *Classes*

When formatting their classes, developers should follow the following rules...

- Public member variables appear before private member variables.
- Public custom functions should appear before private custom functions.
- Unity functions should appear before custom functions.

## **Error Handling**

### *Exceptions*

Exception cases should be used when there is a possibility of a major error occurring. Unity’s exception logging syntax should be used as shown in Figure 9.

```
try {  
    // area where an error may occur.  
}  
  
catch (Exception e) {  
    Debug.LogException(e, this);  
}
```

*Figure 9: Using try and catch for error handling*

### *Test Cases*

Stress tests should print a message indicating the conditions of the test at its completion. The message should be in the format “[Filename.cs -- FunctionName()] Stress test ended: description of condition.” An example is shown in Figure 10.

```
[UnityTest]  
public IEnumerator CustomStressTest()  
{  
    //perform stress test  
  
    if (fps < 30) {  
        Debug.Log("[StressTest.cs -- CustomStressTest()] Stress test ended: 30 FPS reached after "  
            + numItems + " items were spawned.");  
        break;  
    }  
}
```

*Figure 10: Stress test example*