Contents

1 Naming

This section goes over the naming standards for class names, method names, and variable names for Brain Stew. Names should be used to describe the method of the item being named. Abbreviations and acronyms that may not be recognized by other developers instantly should be avoided.

The following codes show how to name classes, methods, and variables:

```
/* Instances of names that do not follow the Brain Stew Coding Standards */

int mvmtSpeed; // variable names reveal intent. need to be more descriptive

class GameMng(); // class name uses unnecessary abbreviations

void goToMM(); // method name uses an uncommon abbreviation
```

```
/* Instances of names that follow the Brain Stew Coding Standards */

int movementSpeed; // More descriptive and searchable

class GameManager(); // More organized and readable

void goToMainMenu(); // Added more context to the method
```

1.1 Classes

Class names should follow the pascal case format. Pascal case format is the practice of writing phrases by separating each word with a capital letter. It should also be named for its purpose.

The following code provides examples of class names:

```
/* Class Names */

class GameManager(); // Uses PascalCase and contains the code for the game manager

class PlayerControl(); // Contains the code for the player's movement and actions

class EnemyControl(); // contains the code for the enemy's movement and actions
```

1.2 Methods

Method (function) names should follow the camel case format. Camel case format follows the same rules as pascal case, but the very first letter is in lowercase.

The following code provides examples of method names:

```
/* Method (Function) Names */

void movePlayer(); // Uses camelCase and describes the action of the method

void takeDamage(); // Uses camelCase and describes the action of the method

void attackEnemy(); // Uses camelCase and describes the action of the method
```

1.3 Variables

Variable names should follow the general naming rules described under section 1. Additional changes can be made depending on the variable's accessibility (i.e. public, public, protected, static, etc.).

1.3.1 Public Variables

Public variables should follow the camel case format.

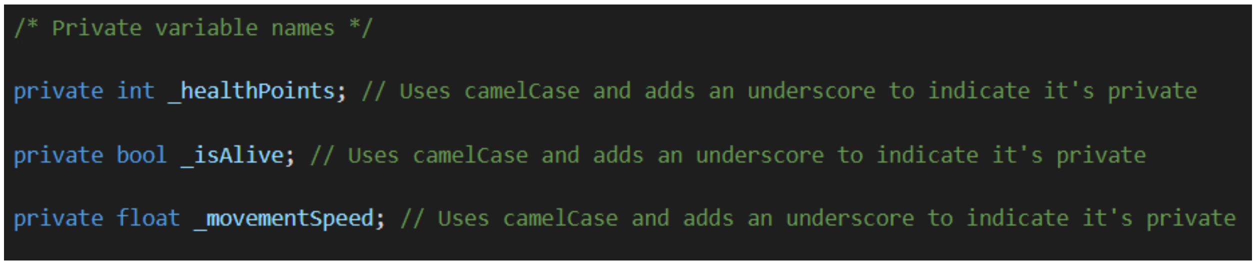
The following code provides examples of public variable names:

```
/* Public variable names */

public int healthPoints; // Describes the variable and uses camelCase

public bool isAlive; // Describes the variable and uses camelCase

public float movementSpeed; // Describes the variable and uses camelCase
```

1.3.2 Private Variables

Private variables should begin with an underscore and follow the camel case format for the rest of the variable name.

The following code provides examples of private variable names:

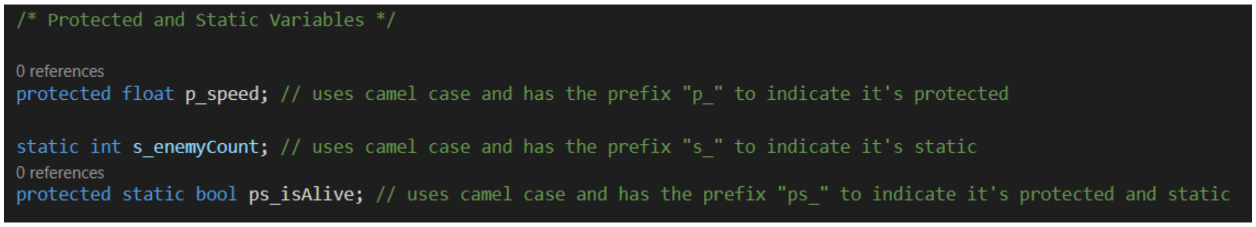


```
/* Private variable names */

private int _healthPoints; // Uses camelCase and adds an underscore to indicate it's private

private bool _isAlive; // Uses camelCase and adds an underscore to indicate it's private

private float _movementSpeed; // Uses camelCase and adds an underscore to indicate it's private
```

1.3.3 Protected and Static Variables

Protected variable names should begin with the prefix "p_", and the static variable names should begin with the prefix "s_" and use camel case format for the rest of the name of the variables. If a variable is both protected and static, the name should begin with the prefix "ps_".

The following code provides examples of protected and static variables:

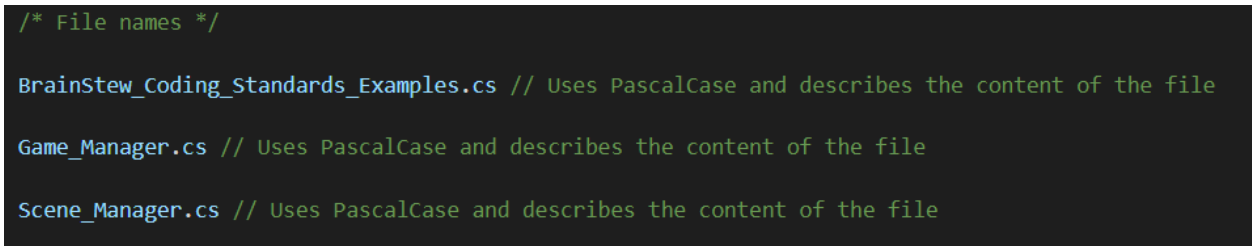


```
/* Protected and Static Variables */

0 references
protected float p_speed; // uses camel case and has the prefix "p_" to indicate it's protected

static int s_enemyCount; // uses camel case and has the prefix "s_" to indicate it's static
0 references
protected static bool ps_isAlive; // uses camel case and has the prefix "ps_" to indicate it's protected and static
```

1.4 File Names

File names (or scripts) should follow the pascal case format and separate each word with underscores while describing the content of the file.

The following are examples of properly named files:



```
/* File names */

BrainStew_Coding_Standards_Examples.cs // Uses PascalCase and describes the content of the file

Game_Manager.cs // Uses PascalCase and describes the content of the file

Scene_Manager.cs // Uses PascalCase and describes the content of the file
```

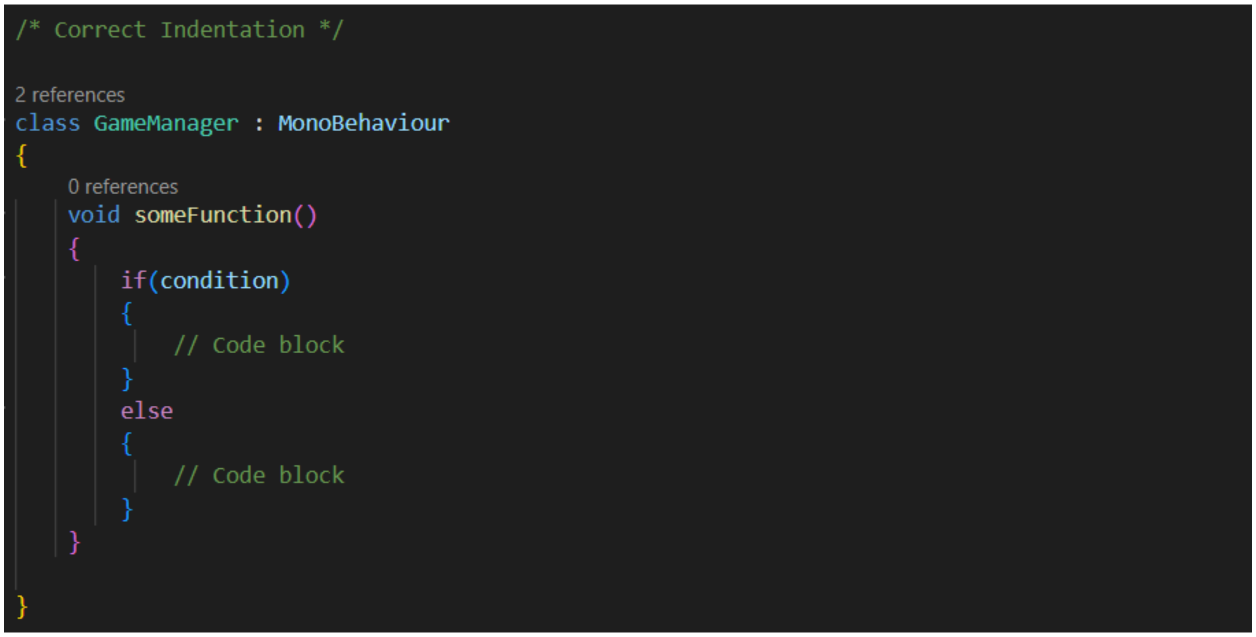
2 Formatting

This section details the organization of the file to improve consistency and readability of the contents of the file/script.

2.1 Indentation

In all cases, indentation should be 3 spaces (or 1 tab click). Any statements that use a curly brace with a start and an end like classes, methods, and conditional statements should follow the Allman style indentation style. The Allman style indentation places the opening curly braces on a new line.

The following code provides examples of appropriate indentation:



```
/* Correct Indentation */

2 references
class GameManager : MonoBehaviour
{
    0 references
    void someFunction()
    {
        if(condition)
        {
            // Code block
        }
        else
        {
            // Code block
        }
    }

}
```

2.2 Library Format

Library imports should be ordered alphabetically and in an ascending order with the base library on top, followed by its sub-library components as shown below:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using UnityEngine.SocialPlatforms.Impl;
using UnityEngine.UIElements.Experimental;
```

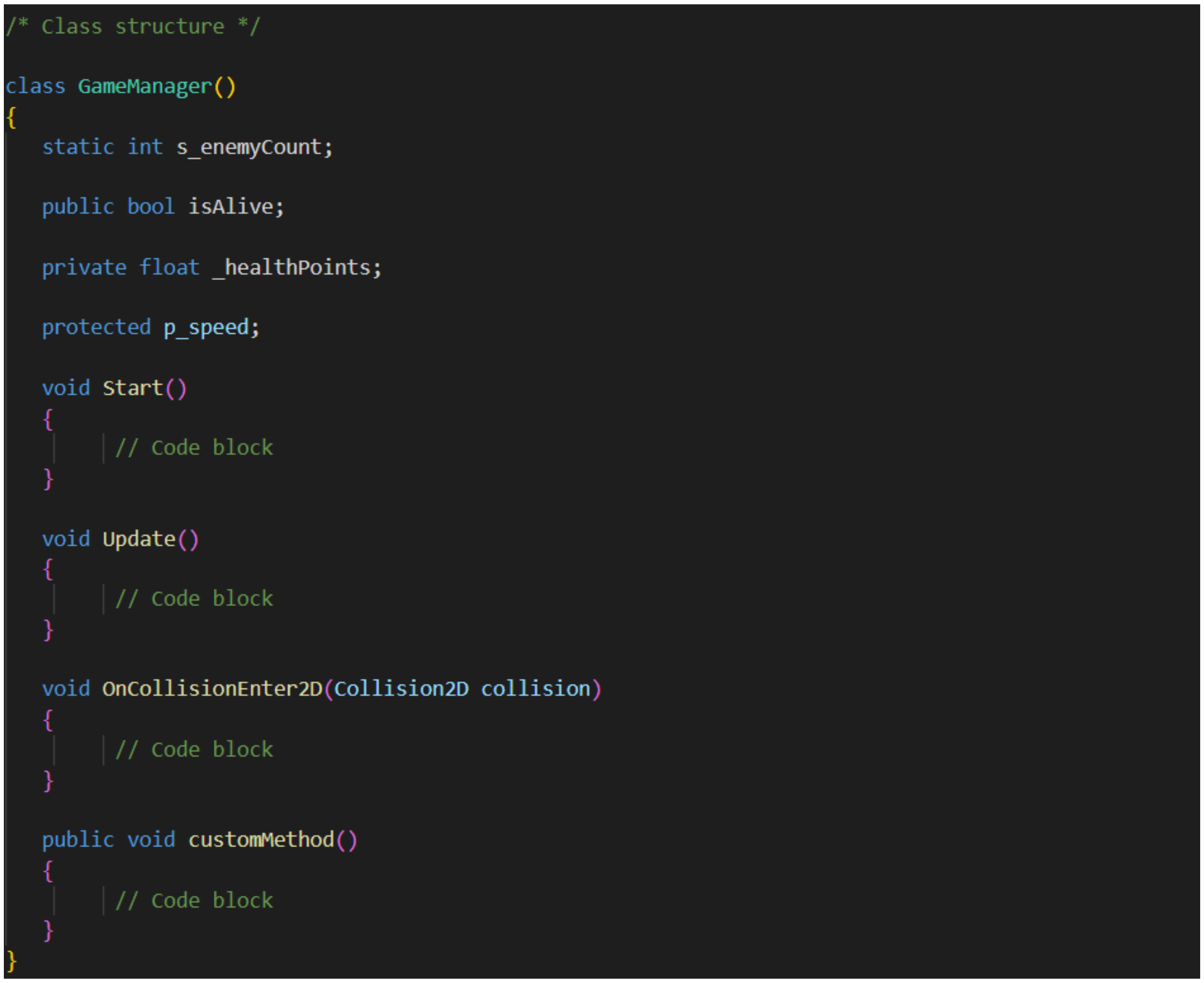
2.3 Class Format

Classes should follow the general instructions outlined in section 2.1. Furthermore, classes should follow a structure to improve readability, consistency, and searchability:

1. Class data members should be organized in this order:
   a. e.g. static → public → private → protected
2. The built-in Unity Start() method (if used).
3. The built-in Unity Update() method (if used).
4. Any other built-in Unity methods such as OnCollisionEnter2D().
5. Any additional written methods.

Each separate element of a class should be separated by a single empty line.

The following code is an example of a class structure:

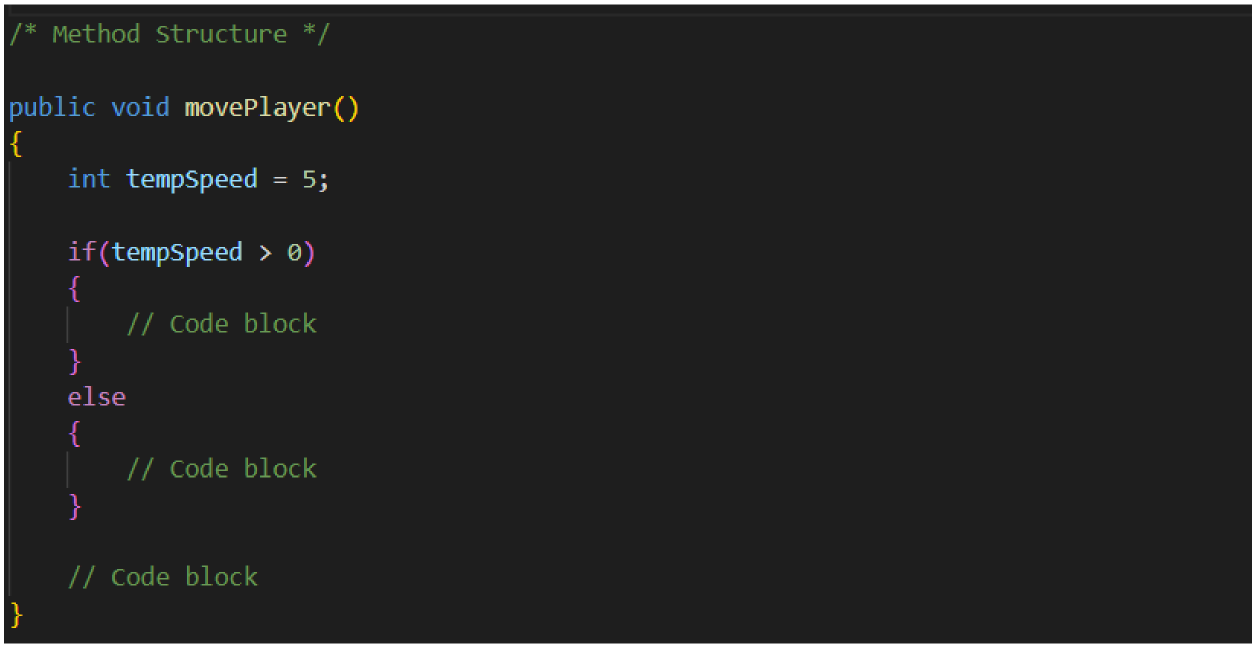


```
/* Class structure */

class GameManager()
{
   static int s_enemyCount;

   public bool isAlive;

   private float _healthPoints;

   protected p_speed;

   void Start()
   {
       // Code block
   }

   void Update()
   {
       // Code block
   }

   void OnCollisionEnter2D(Collision2D collision)
   {
       // Code block
   }

   public void customMethod()
   {
       // Code block
   }
}
```

2.4 Method Format

Methods should follow the general instructions outlined in section 2.1. Method parameters should be separated by a comma and a space. They should also be structured as classes outlined in section 2.3.

The following code is an example of a method structure:

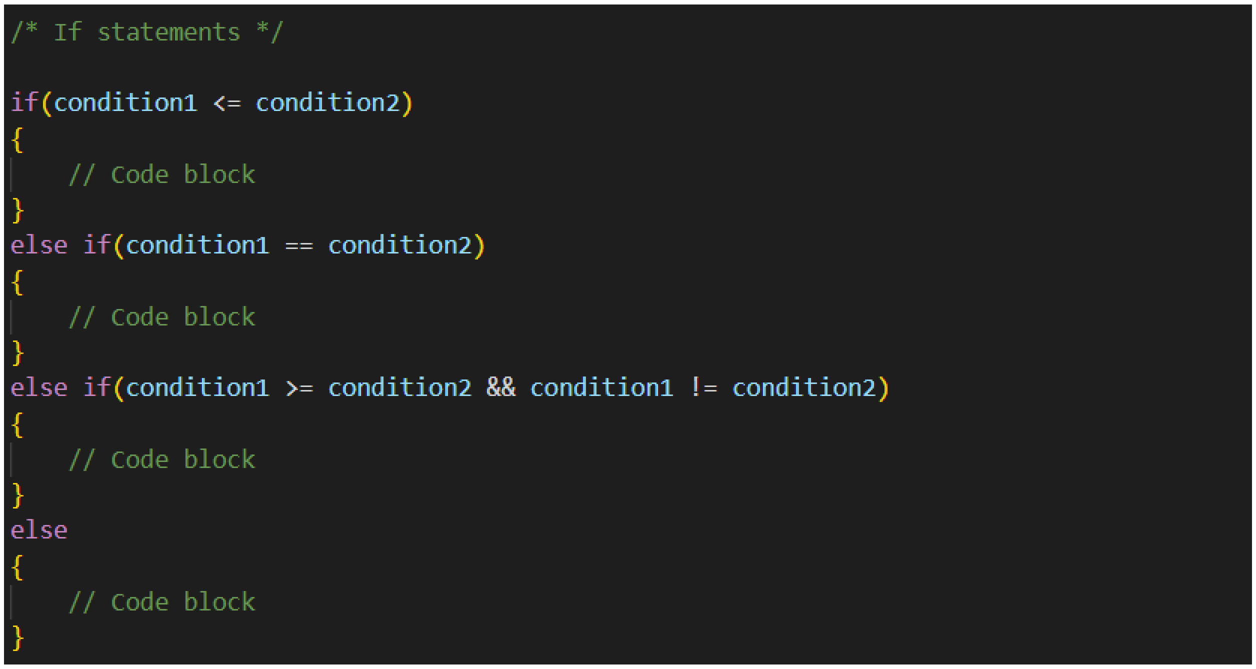


```
/* Method Structure */

public void movePlayer()
{
    int tempSpeed = 5;

    if(tempSpeed > 0)
    {
        // Code block
    }
    else
    {
        // Code block
    }

    // Code block
}
```

2.5 If Statement Structure

If statements should follow the general instructions outlined in section 2.1. They should be organized, readable, and easy to understand.

The following code provides some examples of if statements:

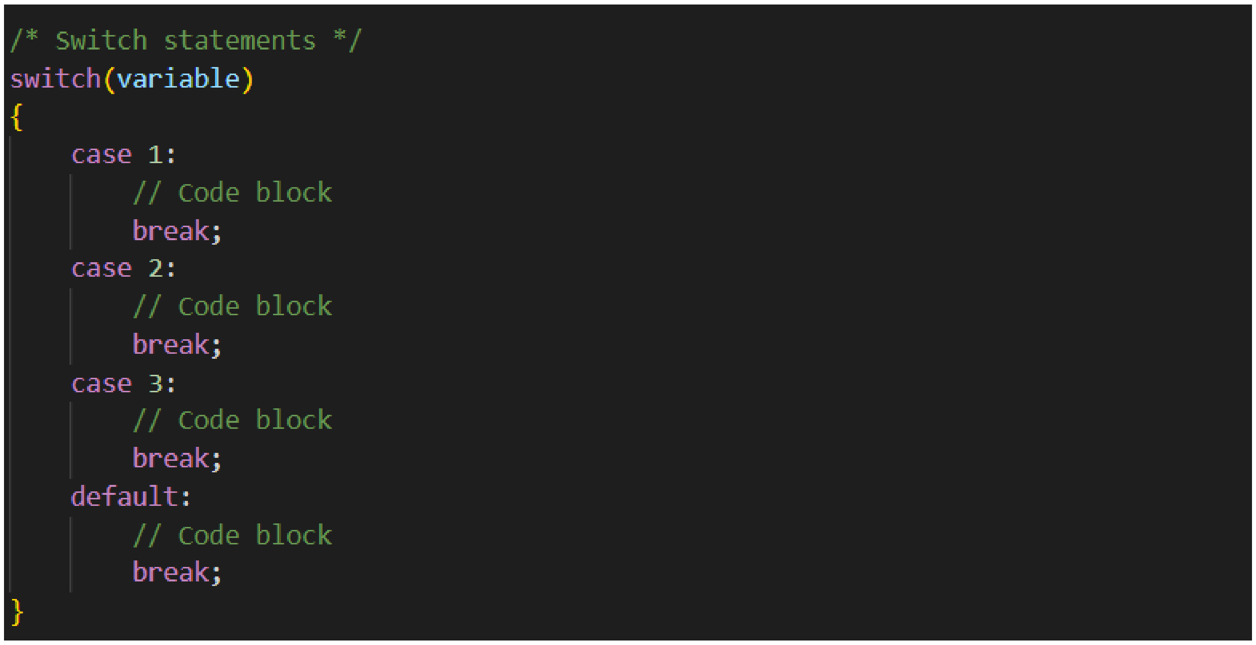


```
/* If statements */

if(condition1 <= condition2)
{
    // Code block
}
else if(condition1 == condition2)
{
    // Code block
}
else if(condition1 >= condition2 && condition1 != condition2)
{
    // Code block
}
else
{
    // Code block
}
```

2.6 Switch Statement Structure

Switch statements should follow the general instructions in section 2.1.

The following code provides an example for using switch statements:

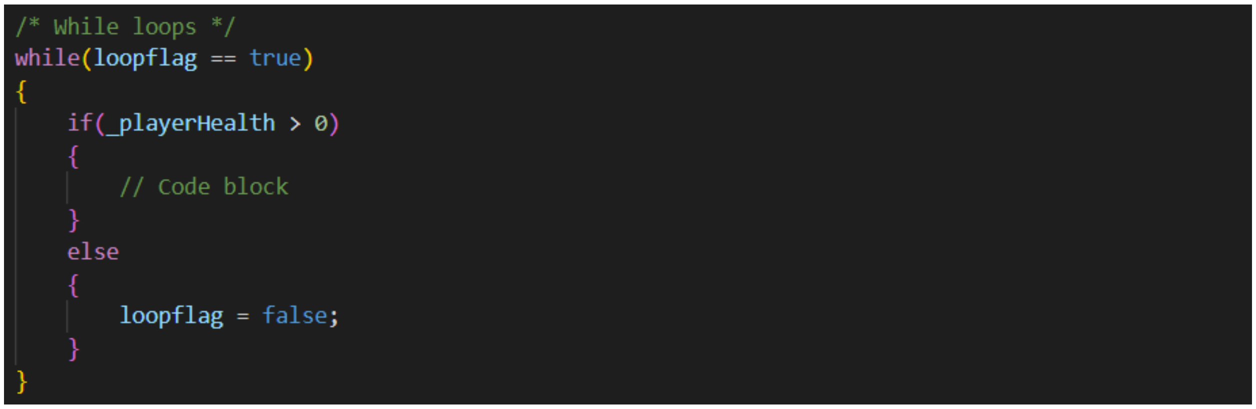


```
/* Switch statements */
switch(variable)
{
    case 1:
        // Code block
        break;
    case 2:
        // Code block
        break;
    case 3:
        // Code block
        break;
    default:
        // Code block
        break;
}
```

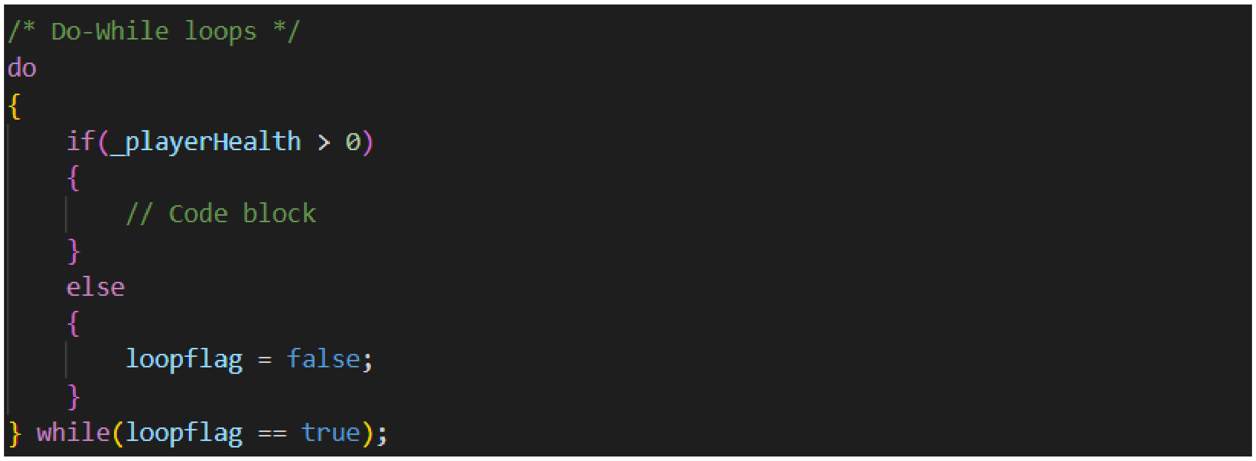
2.7 While and Do Loop Structure

While loops should follow the general instructions in section 2.1. For the conditions for the while or do-loop, instructions for if statement in section 2.5 should be followed.

The following code provides an example of while loop and do-while loop:



```
/* While loops */
while(loopflag == true)
{
    if(_playerHealth > 0)
    {
        // Code block
    }
    else
    {
        loopflag = false;
    }
}
```

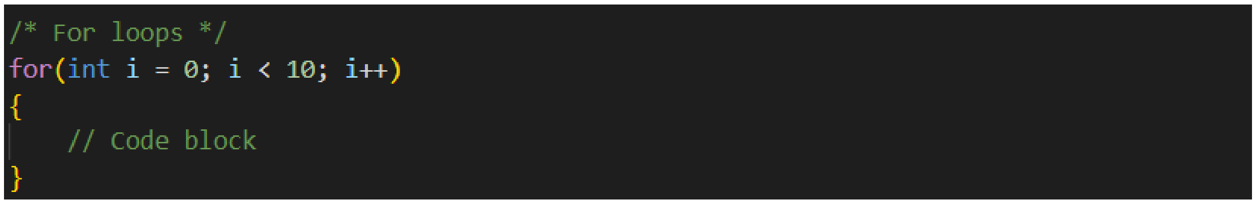


```
/* Do-While loops */
do
{
    if(_playerHealth > 0)
    {
        // Code block
    }
    else
    {
        loopflag = false;
    }
} while(loopflag == true);
```

2.8 For Loop Structure

For loops should follow the general instructions in section 2.1.

The following code provides an example of a for loop:



```
/* For loops */
for(int i = 0; i < 10; i++)
{
    // Code block
}
```

3 Documentation

Documentation is crucial to communicate ideas to other members of the project. When documenting, make sure to include *what* the code does and *why* you picked that approach. Any code that may not be immediately understood by another developer should be explained with comments.

This section will focus on how to document your progress using multi-line comments, inline comments, file headings, and method headings.

3.1 Multi-line Comments

Multi-line comments are used when comments exceed one line in length. Their purpose is to describe a class or a method in coding. If a comment is created on a new line, the syntax for the comment "/* Comment */" should be used.

The following is an example of a multi-line comment:

```
/* Multi-line comments */
/*
This is an example of a multi-line comment,
which can be used to describe the purpose of a class
or a method in more detail.
*/
```

3.2 Inline Comments

Inline comments are used for shorter, brief description of a block of code in the same line of the described code. The syntax for the inline comments "//" should be used. The comments should be close to lining up.

The following is an example of an inline comment:

```
// Inline comments
public bool isAlive;        // an example of an inline comment

private float _healthPoints; // an example of an inline comment

static int s_enemyCount; // an example of an inline comment
```

3.3 File Heading

At the beginning of each file or script there should be a brief description of what the file is for while following the multi-line comment format as outlined in section 3.1. It should include the name and the role of the responsible developer.

The following is an example of a file heading:

```
/* File Header Comments */
/*
Nihat K Polat
Team Lead 4: Project Manager
TechDown

This multi-line comment is an example of a file header comment,
which includes information about the author, team lead, and
project name.

Also, make sure the opening and closing slash and asterisk
are on separate lines.
*/

public class GameManager : MonoBehaviour
{
    // Class code
}
```

3.4 Method Heading

At the beginning of each method there should be a brief description of the method. As outlined in section 3.1, multi-line comments should be used for the description of the method.

The following is an example of a method heading:

```
/* Method Header Comments */
/*
This is an example of a method header comment,
which describes the purpose of the method, its parameters,
*/

public void movePlayer()
{
    // Method code
}
```