# Justification of Code

Max Morphy

**Application File Structure**

The complete application is contained in a folder called “Platformer”. Within this folder is a README.md file explaining what the application is and the “JS Platformer” folder which contains all the code for the game. The subfolders within the “JS Platformer” folder are: “assets”, “css” and “js”. The “JS Platformer” folder also holds the main html file: index.html.

The “assets” folder is where any images, sounds, fonts, etc… would be added to ensure the “JS Platformer” folder doesn’t become cluttered and difficult to navigate to find desired files.

The “css” folder contains the css file that dictates the style of the web page users interact with and all HTML objects, namely the canvas element.

The “depreciated” folder contains my earlier implementation of some of the classes, as the project progressed it became less relevant, but early on it was a helpful reference as I developed a more comprehensive solution, and more refined implementation of the various classes used throughout the game.

The “js” folder holds all the javascript code of the game, It was organised such that all the main files namely config.json, game.js and main.js are in the folder and all the other object classes are stored in the “classes” folder. This allows the files that are being updated constantly to remain easily accessible as additional object classes are added. Object classes, if implemented effectively, should be very much a “set and forget” proposition and should not need to be constantly accessed as other features of the application are developed.

**Analysis/ Justification of code / features**

Index.html

In my application the primary role of index.html is to initialise and link all the resources of the backend to the user. This includes linking to the css styles sheet, creating the canvas element which the entire game is based around, calling the main.js file which creates an instance of the game classic and from there the application essentially begins. An important note is that it is called main.js as a module so that import statements are allowed, without this the oop paradigm and the separation of classes into individual files would not work. It also contains an event listener that suppresses the right click menu which is not needed in the game and can interrupt the user if the user accidentally right clicks.

main.js

Although only 5 lines of code, this file creates an instance of the game class, which initialises the game

game.js

As mentioned earlier I have used oop, the game class responsible for coordinating all the game objects as well as managing variables and calling the methods that makeup the game.

The first thing in the game.js file is the importing of all the other classes that make up the game, so they can be used in the game. The constructor function of the game starts by creating a variable linking to the canvas html element and this canvas' context, this is critical in drawing graphics onto the screen.

Next, it loads in all the constant variables from the config.json file. This process uses various control structures to ensure the process is completed before other aspects of the game are initialised, as these variables are needed in the initialisation of other game objects and if they are undefined when the constructors of other classes are called this is an obvious problem. The loadData() function also has an internal controls structure that will throw various errors if particular aspects of the process are not performed correctly. Here it is important to consider the benefit of saving constant variables in a Json file *(discussed later) and the inherent dangers involved when compared to simply defining constant variables in the constructor of the game class. For example storing data in a json file then reading the data in from the json file becomes a potential point of failure, in the context of my application the transfer of data is neither large enough in scope or complexity to pose a likely or substantial risk to the smooth running of the application.

Next, the initialise() function initialises all the game-like aspects of the application. Some key items include:

- initialising the eventListeners, which is how the game can detect user input like keys being pressed, mouse movement, left click, resizing of the window and clicking off the application. Each of these eventListeners can be assigned functions/ code that will run in the event they are triggered e.g. calling the functions corresponding to pressed buttons, generating bullets, etc…
- Initialising the variables that are not reset when the player dies for example the player's money or the upgrades purchased.
- playPreFlight() is the code that sets up the actual playable game portion of the application, this includes:
    - Creating an instance of the player class
    - Creating the lists that will store all instances of platforms, bullets, enemies, coin, etc…
    - Variables that apply to each play of the game (variables that will reset upon the players death)
    - The final role of this function is to assign the variable this.gameState the value "play"
- The Game class also has shopPreFlight() and helpPreFlight() which are methods that initialise different gamState and alter the gameState variable as the user navigates to another section within the application.

- Finally the gameLoop() function is called, this is the actual game loop. This is organised into an update() and draw() method. Essentially, during every frame variables like player location, enemy location, score, etc… are calculated/ updated then the draw() method will present this information graphically to the player. The requestAnimationFrame(() => this.gameLoop()); code inside the gameLoop() method calls itself once it has run itself, thus creating the game loop.

The variable this.GameState() alters the draw() and update() method so that only the code that corresponds to that game state is run and relevant information presented to the user. It essentially keeps track of if the user is playing the game, browsing the shop, or reading the help section so that the methods/ code corresponding to the currently selected page is executed.

The general outline of every class I used is centred around an update() and draw() method similar to the game class. In practice the game's update() method is essentially just calling the update() methods of every object in the game. The game's update() class also manages the interaction between objects e.g. the collision between the player and enemies, the collision between enemies and bullets, and the collision between the enemies/player with the environment. The update() method also passes game variables to the object's update() methods so they can respond to player input, the location of other objects, etc… In the case of objects that are organised into lists like the bullets, enemies, environment entities, coins, etc.. the list.forEach() method was used to loop through the list and call a method of the item it is currently iterating through. An example of the implementation of this method is: this.list.forEach(item => item.update()) this was a very efficient and clean way to call the update method for each object in a list of objects. The game's update() method also calls the logic for spawning enemies.

The draw method basically draws everything onto the screen. An identical method to the update method was employed to deal with objects stored in lists. A critical control structure in the draw method is the if statements analysis the this.gameState variable and drawing the correct objects and text onto the screen.

One method used to gain input from the user was the implementation of buttons. Buttons were assigned a method of the game class that would be called when the button was clicked on by the user. I used the .blind() method built into js. Essentially, you can assign a class a method of another class. In the case of high school assessment that is a platformer, the obvious security concerns of using this feature of js can be safely overlooked, but one should note that use of this method in a software application where security is a primary consideration should be carefully reflected upon.

The click() method is called by the corresponding eventListener every time the user clicks, Through the use of various control structures taking into account gameState, if the player is alive and if any buttons are active the correct method of the game class or other objects are called in response to the aforementioned user input.

Some other method and variables within the game class of note are as follows:

- bulletCleanUp() loops through the list of bullets, then applies an algorithm to determine if bullets off screen and remove it from the list of bullets. This is important as the game will have to continually update each bullet, a process that isn't particularly computationally taxing in the case of a small number of bullets, as the number of bullets grows there is potential for the game to run slower as a result of performing unnecessary calculations
- spawnEnemies() uses an algorithm to calculate "difficulty" based on how many enemies the player has killed, as the difficulty increases the number of enemies the player has to fight off increases. Also, as the difficulty increases there is a probability that the player will face increasingly difficult enemies. This is communicated visually to the player as the enemies with more health will be a deeper shade of red.
- The game class has various functions that make use of the collide() method I developed in my rect class. *(more on this later)

config.json

The config.json file contains all the game variables that are loaded in upon the initialisation of the game. Through the use of a for loop that iterates through the data with reference to the key and corresponding data on the value, each key is made a variable of the game class with corresponding value of the key e.g. {key: value} -> this.key = value; this meant that additional variables did not have to be explicitly defined in the game class. This added a level of convenience to the process of adding classes whilst reducing clutter in the game class. By limiting the level of "hard coding", changes to variables can be made dynamically and globally throughout the application with manipulation of a single line in a json file or new variables added through the addition of a single line of a json file. It could be argued that there is a safety risk to loading variables from a json file straight into the game classes, without any validation or error checking however, as all the game variables can' be altered without physically altering json file (that is no data is being written into the file during the use of the application) the security risk and risk of improperly formatted data is negligible.

Vec2D.js

I created this class so I could treat the position, velocity, and acceleration of objects within the game as vectors with an x and y component corresponding to the 2 dimensions of the screen. The class has an add() method which adds the two vectors together (used to add vel to position and add acceleration to vel) simulating somewhat realistic physics. It also has a method getMagnitude() which gets the magnitude of the vector, some cool maths can be performed and the bullets travel in a uniform speed in the direction of where the user clicked.

rect.js

The rect class has a draw() method that is used to draw all of the rectangle objects on the screen. It also has a collide() method that takes another rect as a parameter and applies an algorithm to determine if the two rects collide, returning true in that case. This is a very important method that is used throughout the code, all objects in game, regardless of if they are a literal geometric rectangle or not they will have a rect property that serves as its hitbox so the .collide() method can be utilised.

physicsBody

The physicsBody class is a parent class I implemented from which an entity that I want to possess physics behaviour like movement, collisions, effect of gravity, etc… could inherit code from which would be common to all such objects, making use of the concept of inheritance, a key aspect of object oriented programming. Without this class I would end up re-writing code that is used in multiple classes multiple times. The most important method in this class is testCollisions() which is how the game handles collisions where it is important to subsequently change the position or velocity of the objects involved e.g. the player/ enemies colliding with the environment. The way it works is by creating a “temporary” instance of the rect class with position = this.pos + this.velocity; the .collide() method of the rect class can then be used to determine if the “temporary” rect collides with the rect passed in as a parameter of the method. Once it is determined if the “temporary” rect has collided, a comparison between the position of the object's current position and the position of the rect passed in as a parameter can be made. In this way it can be determined if the collision occurs on the top, bottom, right or left side and then subsequent code can be executed depending on the case.

player.js

The Player class is a child of the physicsBody class. Through a successful implementation of the principle of abstraction the player class only needs a method to update its velocity based on user input and a method to be run in the event the user jumps. By using inheritance I can call super.update() to handle all collisions of the player without writing any new code. Drawing of the player onto the screen is handled by the instance of the Rect class it has as a property.

enemy.js

Like the player class, the enemy class is a child of the physicsBody class. I added aI() method to its update method so the enemy chases the player, for the most part the code for the PhysicsBody class is reused, indicating a successful implementation of abstraction and inheritance.

ground.js

The ground class doesn't do much of note except for having a rect that draws at the bottom of the screen and a hitbox that the players and enemy can interact with.

platform.js
This is an extension of the PhysicsBody class, but I don't call the update method. So it is essentially a stationary hitbox. Once again I get to reuse already written code highlighting the benefit of the object oriented paradigm.

circle.js
The circle class has a position, radius and colour, an instance of the rect class that will act as a hitbox and a draw() method that will draw a circle on the screen. It is intended to be a parent class of circular objects like the bullet and coin class.

coin.js
A child of the circle class.

bullet.js
An extension of the circle class that takes the mouse position as a parameter and utilises the Vec2D classes' methods to calculate a velocity vector. The update() method just moves the bullet each frame.