The University of Melbourne
School of Computing and Information Systems
SWEN20003 Object Oriented Software Development

# ShadowMario

**Project 2, Semester 1, 2024**
Released: Friday, 19th April 2024 at 6:00pm AEDT
Project 2A Due: Monday, 29th April 2024 at 6:00pm AEDT
Project 2B Due: Friday, 17th May 2024 at 6:00pm AEDT

**Please read the complete specification before starting on the project, because there are important instructions through to the end!**

## Overview

In this project, you will create an arcade game called *ShadowMario* in the Java programming language, continuing from your work in Project 1. We will provide a full working solution for Project 1; you **may** use all or part of it, provided you add a comment explaining where you found the code at the top of each file that uses the sample code.

This is an **individual project**. You may discuss it with other students, but all of the implementation must be your **own work**. By submitting the project you declare that you understand the University's policy on academic integrity and are aware of consequences of any infringement, including the use of **artificial intelligence**.

You may use any platform and tools you wish to develop the game, but we recommend using IntelliJ IDEA for Java development as this is what we will support in class.

**Extensions & late submissions:** If you need an extension for the project, please complete the Extension form in the **Projects** module on Canvas. Make sure you explain your situation with some supporting documentation such as a medical certificate, academic adjustment plan, wedding invitation, etc. You will receive an email saying if the extension was approved or if we need more information.

If you submit late (**either** with or without an extension), please complete the Late form in the **Projects** module on Canvas. For both forms, you need to be logged in using your **university** account. Please **do not** email any of the teaching team regarding extensions or late submissions. All of this is explained again in more detail at the end of this specification.

There are two parts to this project, with different submission dates. The first task, **Project 2A**, requires that you produce a class design demonstrating how you plan to implement the game. This should be submitted in the form of a UML diagram showing all the classes you plan to implement, the relationships (e.g. inheritance and associations) between them, and their attributes, as well as their primary public methods. You **do not** need to show constructors, getters/setters, dependency, composition or aggregation relationships. If you so choose, you may show the relationship on a separate page to the class members in the interest of neatness, but you must use correct UML notation. Please submit as a **PDF file** only on Canvas.

The second task, **Project 2B**, is to complete the implementation of the game as described in the rest of this specification. You **do not** need to strictly follow your class design from Project 2A; you will likely find ways to improve the design as you implement it. Submission will be via GitLab and you must make **at least 5 commits** throughout your project.

## *Game Overview*

*"The aim is simple - move the **player** to jump over the **enemies** and collect the **coins**. To win each level, you need to reach the **end flag**. The second level features **flying platforms** that the player can jump on to, extra powers such as **invincibility** and **double score**. The third level includes the **enemy boss** that the player must defeat by shooting **fireballs**. Can you reach the end flags and beat the boss to win the game?"*

The game features three levels : *Level 1*, *Level 2* and *Level 3*. In Level 1, the player has to press the arrow keys to move left, right or jump. The player can collect coins by colliding with them - collecting one coin, increases the *score* by one. The player can avoid enemies by jumping over them. Unlike in Project 1, the enemies will be moving now in a fixed range (this is explained in detail later). If the player collides with an enemy, they will lose *health points*. To complete the level, the player needs to reach the end flag. If the player's health points reduce to zero, the game ends.

Level 2 features the same gameplay as above but the player now has to deal with additional features. Flying platforms are moving entities that the player can jump on to and move on. The player can gain two new powers by colliding with the invincibility entity and double score entity. Invincibility makes the player invincible to health loss from enemy collisions for a set period of time and double score grants the player, double the score from each coin collected for a set period of time. Once again, to complete the level, the player needs to reach the end flag.

Level 3 is the final level. It includes all of the above as well as an enemy boss at the end of the level. When the enemy boss and the player come into a fixed distance from each other, they can both shoot fireballs at each other that cause health points loss. To complete the level and finish the game, the player must beat the enemy boss and reach the end flag.

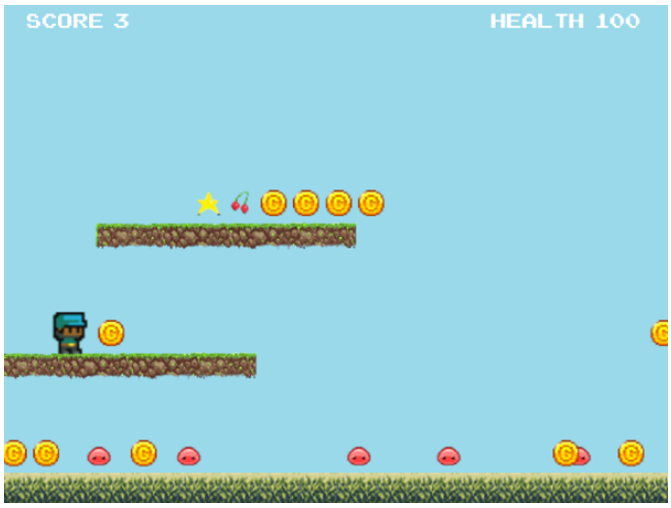
Note that the game does not need to be played progressively. You can choose which level to play from the start screen and also at the end of each level.

## *An Important Note*

Before you attempt the project or ask any questions about it on the discussion forum, it is crucial that you read through this entire document thoroughly and carefully. We've covered every detail below as best we can without making the document longer than it needs to be. Thus, if there is any detail about the game you feel was unclear, try referring back to this project spec first, as it can be easy to miss some things in a document of this size. And if your question is more to do on **how** a feature should be implemented, first ask yourself: *'How can I implement this in a way that*

*both* ***satisfies the description*** *given, and helps make the game* ***easy and fun to play****?'* More often than not, the answer you come up with will be the answer we would give you!


Figure 1: Start Screen Screenshot




(a) Completed Level 2 Screenshot


(b) Completed Level 3 Screenshot

Figure 2: Level Screenshots

Note : the actual positions of the entities in the levels we provide you may not be the same as in these screenshots.

# The Game Engine

The **B**asic **A**cademic **G**ame **E**ngine **L**ibrary (Bagel) is a game engine that you will use to develop your game. You can find the offline documentation for Bagel on Canvas under the Projects module.

### *Coordinates*

Every coordinate on the screen is described by an $(x, y)$ pair. $(0, 0)$ represents the top-left of the screen, and coordinates increase towards the bottom-right. Each of these coordinates is called a *pixel*. The Bagel `Point` class encapsulates this.

### *Frames*

Bagel will refresh the program's logic at the same refresh rate as your monitor. Each time, the screen will be cleared to a blank state and all of the graphics are drawn again. Each of these steps is called a **frame**. Every time a frame is to be rendered, the `update()` method in `ShadowMario` is called. It is in this method that you are expected to update the state of the game.

Your code will be marked on **120Hz screens**. The refresh rate is typically `120` times per second (Hz) but some devices might have a lower rate of 60Hz. In this case, when your game is running, it may look different to the demo videos as the constant values in this specification have been chosen for a refresh rate of 120Hz. For your convenience, when writing and testing your code, you **may** change these values to make your game playable (these changes are explained later). If you do change the values, **remember** to change them back to the original specification values before submitting, as your code will be **marked on 120Hz screens.**

# The Levels

Our game will have three levels, each with elements to implement that are described below.

### *Window and Background*

The background (`background.png`) should be rendered on the screen to completely fill up your window throughout the game (for the start screen and all the levels). The default window size should be `1024 * 768` pixels. The background has already been implemented for you in the skeleton package.

### *Start Screen*

Each level has the same start screen. The screen has a title message that reads `SHADOW MARIO` should be rendered in the font provided in `res` folder (`FSO8BITR.ttf`), in size `64`. The bottom left corner of this message should be located at `(220, 250)`.

Additionally, an instruction message consisting of 4 lines:

```
USE ARROW KEYS TO MOVE
ENTER LEVEL TO START - 1, 2, 3
```

should be rendered **below** the title message, in the font provided, in size `24`. The bottom left of the first line in the message should be coded as follows: the x-coordinate should be calculated such that the whole message looks centered horizontally, and the y-coordinate should be at `400` pixels.

There must be **adequate spacing** between the 2 lines to ensure readability (you can decide on the value of this spacing yourself, as long as it's not small enough that the text overlaps or too big that it doesn't fit within the screen). You can align the lines as you wish.

The player chooses which level to play by pressing the corresponding key (1, 2 or 3). Once the level is over, regardless of whether it was won or lost, the player will be re-directed back to the start screen.

### *Properties File*

The key values of the game are listed in two **properties files** which are given in the skeleton package. The message coordinates, image filenames and other values are given in the `app.properties` file. The message strings are given in the `message_en.properties` file. These files **shouldn't** be edited (unless you need to adjust values for any frame rate issues). All properties given in the files should be read-in and **not hard-coded**.

To read a value from one of these properties, a `Properties object` must be created. The `getProperty` method can be called on this object with the required value given as the parameter. For your reference, the skeleton package contains an example of how to read the background image filename, window width and window height values.

### *World File*

The entities will be defined in a **world file**, describing the type and their position in the window. The world files for each level are `level1.csv`, `level2.csv` and `level3.csv` correspondingly. A world file is a comma-separated value (CSV) file with rows in one of the following formats:

```
Type of entity, x-coordinate, y-coordinate
```

An example of a world file:

```
PLATFORM,3000,745
PLAYER,100,687
COIN,300,510
FLYING_PLATFORM,400,555
DOUBLE_SCORE,1700,690
ENEMY,400,695
INVINCIBLE_POWER,1800,505
END_FLAG,4100,670
```

The given `(x, y)` coordinates refer to the centre of each image and these coordinates should be used to draw each image. You must actually load it—copying and pasting the data, for example, is not allowed. Marking will be conducted on hidden **different** CSV files of the same format. **Note:** You can assume that there will always be at least one of each for all the entities. The total number of entites in each CSV may vary however.

### *End Screen*

Each level has the same end screen. The end screen has one message - either a win or loss message.

When the player has **reached** the end flag of a level, this is considered as a win. This differs only in Level 3, where the player must **both** reach the end flag and beat the enemy boss (its health must reduce to 0). For a win, the message has the 2 following lines:

```
CONGRATULATIONS, YOU WON!
PRESS SPACE TO CONTINUE
```

It should be rendered, in the font provided, in size `24`. The bottom left of the first line in the message should be coded as follows: the x-coordinate should should be calculated such that the whole message looks centered horizontally and the y-coordinate should be at `400` pixels.

If the player's health points reduces to **0 or below**, this is considered as a loss and the game ends. For a loss, the message has the 2 following lines:

```
GAME OVER, YOU LOST!
PRESS SPACE TO CONTINUE
```

It should be rendered, in the font provided, in size `24`. The bottom left of the first line in the message should be coded as follows: the x-coordinate should be calculated such that the whole message looks centered horizontally and the y-coordinate should be at `400` pixels.

When the player presses the **space key**, the start screen should be rendered again as described in the *Start Screen* section and the player can choose to play again. The player can terminate the game window at any point (by pressing the Escape key or by clicking the Exit button) - the window will simply close and no message will be shown.

**Hint:** The `drawString()` method in the Font class uses the given coordinates as the bottom left of the message. So to make the message look centered horizontally, you will need to calculate the coordinate using the `Window.getWidth() and Font.getWidth()` methods.

# The Game Entities

The following game entities have an associated image (or multiple!) and a starting location `(x, y)`. Remember that all images are drawn from the **centre** of the image using these coordinates.

## ***Player***

In our game, the player can move on screen in one of three directions (left, right and up) when the corresponding arrow key is pressed. However, for our ease of implementation, we assume the player can **only** move vertically and remains stationary in the horizontal direction (i.e. the other entities will be moving in relation to the player's arrow key pressed - *this is explained in detail later*).

(a) player_left.png

(b) player_right.png

Figure 3: The player's images

The player is represented by the two images shown above. Based on the direction the player is moving, the **corresponding** image should be rendered. The player will start the game facing right.

The player's jumping upwards motion will be considered in 3 stages, where the speed in the vertical direction will change:

- Player is currently on a platform & up arrow key is pressed => the vertical speed should be set to `-20` (i.e. the y-coordinate will be decreasing by `20 pixels per frame`).
- During the player's jumping motion => vertical speed should increase by `1` each frame.
- Player has finished jump & has reached platform again => vertical speed should be set to 0 and the player **should not** move below the platform.

**Hint:** Remember that y increases in the downward direction on screen.


Figure 4: Player's score

The player has an associated **score**. When the player collides with a coin, the player's score increases by `1` (the points value of the coin). The score is rendered in the top left corner of the screen in the format of `"SCORE k"` where `k` is the current score. The bottom left corner of this message should be located at `(35, 35)` and the font size should be 30.

When a player collides with an enemy (not including the enemy boss), the player's **health** decreases once by `0.05` (the damage points value of the enemy). The player starts each level with a health value of `1`. The health value is rendered in the top right corner of the screen in the format of `"HEALTH k"` where `k` is the current health, shown as integer




Figure 5: Player's health

percentage of the total health. The bottom left corner of this message should be located at `(750, 35)` and the font size should be 30.

If the player's health value becomes less than or equal to zero, the player moves vertically down off the screen and the game ends. This is done by setting the vertical speed to **2 pixels per frame**.

In Level 3, the player can inflict damage on the enemy boss by shooting fireballs by pressing the `S` key. This can only happen when the player is less than **500** pixels from the boss. This will be described in detail later. The rest of the player's behaviour in Level 2 and 3, is the same as in Level 1.

## Enemy

Figure 6: enemy.png

An enemy is an entity shown by `enemy.png`, that can move in the **horizontal** direction and appears in all 3 levels. It has a damage points value of `0.05`. The enemy has two movements - randomly in a set range and also in relation to the player's key press.

When the player's arrow keys are pressed, the enemy will move similar to Project 1. When the player's *right* arrow key is pressed or held down, the enemy will move to the *left* by **5 pixels per frame**. When the player's *left* arrow key is pressed or held down, the enemy will move to the *right* by the same speed.

The enemy's random movement is as follows. At creation, each enemy will choose to move either left or right **randomly**. Every frame, the enemy will move in this chosen direction by **1 pixel per frame**. It will continue this movement until it has reached a maximum displacement of `50` pixels from its initial starting position. The enemy will then reverse the direction and the same movement will occur in the reversed direction. (In other words, the enemy can move by 1 pixel per frame in a range of 50 pixels either side of its starting position). Note that this random movement will happen **concurrently** to the movement with relation to the player key presses, described above.

When the enemy collides with a player, it inflicts damage to the player's health as described earlier. Once an enemy has inflicted damage once, it cannot inflict damage again even if there are further collisions.

### Collision Detection

To detect collisions, a **range** is first calculated by adding the radius of the enemy image and the radius of the player image. Both values are given in the `app.properties` file. The **current distance** between the player and the enemy is determined by calculating the **Euclidean distance** between the two `(x, y)` coordinates. If the current distance is **less than or equal to** the range, this is considered as a collision.

### Enemy Boss

Figure 7: enemy_boss.png

The enemy boss is an entity shown by `enemy_boss.png`, that appears in Level 3. The enemy boss moves only in relation to the player's arrow key presses as described next. When the player's arrow keys are pressed, the enemy will move similar to Project 1. When the player's *right* arrow key is pressed or held down, the enemy will move to the *left* by **5 pixels per frame**. When the player's *left* arrow key is pressed or held down, the enemy will move to the *right* by the same speed. Unlike the normal enemy, the boss has no random movement.

The boss has an associated health score that has a value of `1` at the start. The health value is rendered in the top right corner of the screen in the format of `"HEALTH k"` where `k` is the current health, shown as integer percentage of the total health. The bottom left corner of this message should be located at `(750, 65)` and the font size should be 30. This message should be displayed in the colour red. **Hint:** The DrawOptions class in Bagel will help you do this.

Every 100 frames, the enemy boss will randomly inflict damage on the player by shooting a fireball if the player is at least `500` pixels from it. The randomness is decided as follows - every 100 frames, a random boolean is generated - if it is `true`, the enemy can fire and if `false`, it cannot fire. Similar to the player, if the enemy's health value becomes less than or equal to zero, the enemy dies and moves vertically down off the screen by **2 pixels per frame**.

### Fireball

Figure 8: fireball.png

The fireball is an entity that is shown by `fireball.png`, and appears in Level 2 and 3. It has a damage points value of `0.5`. Both the player and the enemy boss can shoot a fireball, as described in the previous sections.

Once shot, the fireball will start from the firing entity's current position and move horizontally in the direction of its target at a speed of **8 pixels per frame**. Collision detection (as described earlier) is used to determine if it hits a target or not. If the target is hit, the fireball will disappear from the screen. If the target is missed, the fireball will continue moving until it reaches the boundary of the window.

### Platform

Figure 9: platform.png (cropped to show on one page)

The platform is an entity shown by `platform.png`, that can move in the **horizontal** direction and appears in all 3 levels. When the player's arrow keys are pressed, the platform will move as described below.

When the player's *right* arrow key is pressed or held down, the platform will move to the *left* by **5 pixels per frame**. When the player's *left* arrow key is pressed or held down, the platform will move to the *right* by the same speed, only if the platform's current x-coordinate is **less than 3000**.

### Flying Platform

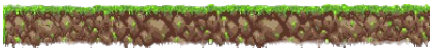


Figure 10: flying_platform.png

The flying platform is a special type of platform that appears in Level 2 and 3 and is shown by `flying_platform.png`. It has two movements - randomly in a set range and also in relation to the player's key press.

When the player's *right* arrow key is pressed or held down, the platform will move to the *left* by **5 pixels per frame**. When the player's *left* arrow key is pressed or held down, the platform will move to the *right* by the same speed. The random movement is the same as described in the `Enemy` section. It has a maximum displacement of **100** pixels.

When the player jumps up, it can land onto a flying platform. To determine this, the following three conditions need to be true. If these are true, the player's vertical speed is set to **zero** (placing them on the platform).

- The distance between the player's x-coordinate and the platform's x-coordinate is less than `200` (this value is called the half length and is given in the `app.properties` file).
- The distance between the player's y-coordinate and the platform's y-coordinate is less than or equal to `50` (this is called the half height).
- The distance between the player's y-coordinate and the platform's y-coordinate is greater than or equal to the (half height - 1).

(These three conditions are a simplified way of checking if the player is in the region right above the platform).

**Note** that from a higher flying platform, the player cannot jump down to a lower flying platform - the player will simply fall down to the normal platform.

### Coin

A coin is an entity shown by `coin.png`, that can move in **both** horizontal and vertical directions. It has a points value of `1`. When the player's arrow keys are pressed, the coin will move as described below.

Figure 11: coin.png

When the player's *right* arrow key is pressed or held down, the coin will move to the *left* by **5 pixels per frame**. When the player's *left* arrow key is pressed or held down, the coin will move to the *right* by the same speed.

When a coin collides with a player, the player's score increases by `1`. If the player's double score power is active, the score increases by double the points value. The collision detection is determined in the same way as described above in the *Enemy* section. Once a collision has happened, a coin will move upwards and disappear off screen. This is done by setting the vertical speed to **-10 pixels per frame**.

### Double Score Power

Figure 12: double_score.png

The double score power is an entity shown by `double_score.png` and can move in the horizontal direction. When the player's *right* arrow key is pressed or held down, it will move to the *left* by **5 pixels per frame**. When the player's *left* arrow key is pressed or held down, it will move to the *right* by the same speed.

If the player collides with it, the power becomes active for `500` frames. During this time, if the player collects a coin, the player receives double the points value. The collision detection is determined in the same way as described above in the *Enemy* section. Once a collision has happened, the power will move upwards and disappear off screen. This is done by setting the vertical speed to **-10 pixels per frame**.

### Invincible Power

The invincible power is an entity shown by `invincible_power.png` and can move in the horizontal direction. When the player's *right* arrow key is pressed or held down, it will move to the *left* by **5 pixels per frame**. When the player's *left* arrow key is pressed or held down, it will move to the *right* by the same speed.



Figure 13: invincible_power.png

If the player collides with it, the power becomes active for `500` frames. During this time, if the player collides with a normal enemy or gets hit by a fireball, the player doesn't receive any damage. (If the player collides with the same enemy outside of the invincibility period, the enemy **can** inflict damage). The collision detection is determined in the same way as described above in the *Enemy* section. Once a collision has happened, the power will move upwards and disappear off screen. This is done by setting the vertical speed to **-10 pixels per frame**.

### End Flag

Figure 14: endflag.png

The end flag is an entity shown by `endflag.png`, that can move in the **horizontal** direction. When the player's arrow keys are pressed, the flag will move as described below. When the player's *right* arrow key is pressed or held down, the flag will move to the *left* by **5 pixels per frame**. When the player's *left* arrow key is pressed or held down, the flag will move to the *right* by the same speed. The collision detection is checked in the same way as described in the *Enemy* section.

## Your Code

You must submit a class called `ShadowMario` that contains a `main` method that runs the game as prescribed above. You may choose to create as many additional classes as you see fit, keeping in mind the principles of object oriented design discussed so far in the subject. You will be assessed based on your code running correctly, as well as the effective use of Java concepts. As always in software engineering, appropriate comments and variables/method/class names are important.

## Implementation Checklist

To get you started, here is a checklist of the game features, with a suggested order for implementing them (in addition to the features in Project 1):

- Implement the new level start screen.
- Implement the Level end screen.
- Read the Level 2 CSV file.
- Implement the enemy's random movement.
- Implement the flying platforms.
- Implement the behaviour of the two powers.
- Read the Level 3 CSV file.
- Implement the enemy boss behaviour/logic.
- Implement the fireball's behaviour.

## Supplied Package and Getting Started

You will be given a package called `project-2-skeleton.zip` that contains the following: (1) Skeleton code for the `ShadowMario` and `IOUtils` classes to help you get started, stored in the `src` folder. (2) All graphics and fonts that you need to build the game, stored in the `res` folder. (3). The `pom.xml` file required for Maven. You should use this template exactly how you did for Project 1, that is:

1. Unzip it.
2. Move the **content** of the unzipped folder to the local copy of your [username]-project-2] repository.
3. Push to Gitlab.
4. Check that your push to Gitlab was successful and to the correct place.
5. Launch the template from IntelliJ and begin coding.
6. Commit and push your code regularly.

# Customisation (optional)

We want to encourage creativity with this project. We have tried to outline every aspect of the game design here, but if you wish, you may customise any part of the game, including the graphics, types of actors, behaviour of actors, etc (for example, an easy extension could be to introduce a new level with different entites or powers). You can also add entirely new features. For your customisation, you **may** use additional libraries (other than Bagel and the Java standard library).

However, to be eligible for full marks, you **must** implement all of the features in the above implementation checklist. Please submit the version **without** your customisation to [username]-project-2 repository, and save your customised version locally or push it to a new branch on your Project 2 repository.

For those of you with far too much time on your hands, we will hold a competition for the best game extension or modification, judged by the lecturers and tutors. The winning three will have their games shown at the final lecture, and there will be a prize for our favourite. Past modifications have included drastically increasing the scope of the game, adding jokes and adding polish to the game, and even introducing networked gameplay.

If you would like to enter the competition, please email the head tutor, Tharun Dharmawickrema at `dharmawickre@unimelb.edu.au` with your username, a short description of the modifications you came up with and your game (either a link to the other branch of your repository or a .zip file). You can email Tharun with your completed customised game anytime before **Week 12**. Note that customisation does **not** add bonus marks to your project, this is completely for fun. We can't wait to see what you come up with!

# Submission and Marking

## Project 2A

Please submit a **.pdf** file of your UML diagram for Project 2A via the Project 2A tab in the Assignments section on Canvas.

## Project 2B - Technical requirements

- The program must be written in the Java programming language.
- Comments and class names must be in English **only**.
- The program must not depend upon any libraries other than the Java standard library and the Bagel library (as well as Bagel's dependencies).
- The program must compile fully without errors.
- For full marks, **every** public attribute, method and class must have a short, descriptive Javadoc comment (which will be covered later in the semester).

Submission will take place through GitLab. You are to submit to your `<username>-project-2` repository. At the **bare minimum** you are expected to follow the structure below. You **can** create more files/directories in your repository if you want.

```
username-project-2
├── res
│   └── resources used for project 2
└── src
    ├── ShadowMario.java
    └── other Java files
```

On 17th May 2024 at 6:00pm, your latest commit will automatically be harvested from GitLab.

### Commits

You are free to push to your repository post-deadline, but only the latest commit on or before 17th May 2024 6:00pm will be marked. You **must** make at least 5 commits throughout the development of the project, and they must have meaningful messages (commit messages must match the code in the commit). If commits are anomalous (e.g. commit message does not match the code, commits with a large amount of code within two commits which are not far apart in time) you risk penalization.

Examples of **good, meaningful** commit messages:

- implemented movement logic
- fix the fireball's collision behaviour
- refactored code for cleaner design

Examples of **bad, unhelpful** commit messages:

- fesjakhbdjl
- yeah easy finished the flying platform
- fixed thingzZZZ

## Good Coding Style

Good coding style is a contentious issue; however, we will be marking your code based on the following criteria:

- You should *not* go back and comment your code after the fact. You should be commenting as you go.
- You should be taking care to ensure proper use of visibility modifiers. Unless you have a very good reason for it, all instance variables should be private. (Constants are allowed to be public or protected).
- Any constant should be defined as a final variable. Don't use magic numbers!
- Think about whether your code is written to be easily extensible via appropriate use of classes.

- Make sure each class makes sense as a cohesive whole. A class should have a single well-defined purpose, and should contain all the data it needs to fulfil this purpose.

### Extensions and late submissions

If you need an **extension** for the project, please complete Extension form in the **Projects** module on Canvas. Make sure you explain your situation with some supporting documentation such as a medical certificate, academic adjustment plan, wedding invitation, etc. You will receive an email saying if the extension was approved or if we need more information.

The project is due at **6:00pm sharp** on Monday 29th April 2024 (Project 2A) and on Friday 17th May 2024 (Project 2B). Any submissions received past this time (from 6:00pm onwards) will be considered late unless an extension has been granted. There will be no exceptions. There is a penalty of 1 mark for a late project, plus an additional 1 mark per 24 hours. If you submit **late** (*either* with or without an extension), please complete the Late form in the **Projects** module on Canvas. For both forms, you need to be logged in using your **university** account. Please **do not** email any of the teaching team regarding extensions or late submissions (as you will be redirected to the online forms).

### Marks

Project 2 is worth **20** marks out of the total 100 for the subject. You are **not required** to use any particular features of Java. For example, you may decide not to use any interfaces or generic classes. You will be marked based on the **effective and appropriate** use of the various object-oriented principles and tools you have learnt throughout the subject.

- Project 2A is worth **8 marks**.
  - Correct UML notation for methods: **2 marks**
  - Correct UML notation for attributes: **2 marks**
  - Correct UML notation for associations: **2 marks**
  - Good breakdown into classes: **1 mark**
  - Appropriate use of inheritance, interfaces and abstract classes/methods: **1 mark**
- Project 2B (feature implementation) is worth **8 marks**.
  - Correct implementation of start screen and level selection: **0.5 marks**
  - Correct implementation of player behaviour: **0.5 marks**
  - Correct implementation of platform and flying platform's behaviour: **1 mark**
  - Correct implementation of enemy behaviour (including image, movement and effects): **1 mark**
  - Correct implementation of enemy boss behaviour (including image, movement and effects): **1 mark**

  - Correct implementation of fireball behaviour (including image, movement and effects): **1 mark**
  - Correct implementation of each power's behaviour (including image, movement and effects): **2 marks**
  - Correct implementation of end flag behaviour: **0.5 marks**
  - Correct implementation of end screen: **0.5 marks**
- Coding Style is worth **4 marks**.
  - Delegation: breaking the code down into appropriate classes: **0.5 marks**
  - Use of methods: avoiding repeated code and overly complex methods: **0.5 marks**
  - Cohesion: classes are complete units that contain all their data: **0.5 marks**
  - Coupling: interactions between classes are not overly complex: **0.5 marks**
  - General code style: visibility modifiers, magic numbers, commenting etc.: **1 mark**
  - Use of Javadoc documentation: **1 mark**