Project 03: Super Scrolling Game

COSC 102 - Fall '23

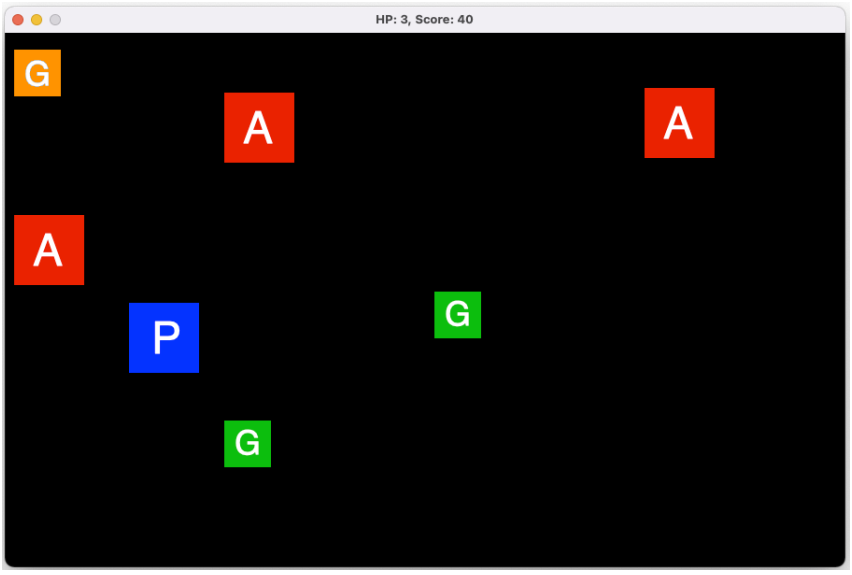
Goal: In your final project, you will work within a large code base, leveraging OOP, inheritance, polymorphism and `ArrayLists`. You will also showcase your creativity by designing your own visual theme and game mechanics.

1 Overview

Super Scrolling Game is a 2D side-scrolling game where players control an avatar and need to collect or avoid various entities scrolling at them, right-to-left. Provided to you is code that implements the game engine (or fundamental "framework") which you will build upon to recreate the Scrolling Game and, eventually a "creative game" utilizing your own graphical theme and game mechanics.

Past 102 students have done a variation of this project, but this code base has since been completely remade. The "Super" moniker of this new version has a double meaning: this rewritten code base gives students much more power to come up with creative mechanics and also has a greater emphasis on inheritance/polymorphism.

1.1 Rules of the Scrolling Game



The Scrolling Game is described below:

- The game launches in a separate window, first showing a splash screen. Pressing the `ENTER` key starts the game.
- The player controls the game using the arrow keys, which move their avatar around the game window.
- The player needs to avoid or collect entities that are generated on the right edge of the window and scroll left.
- Text drawn in the top bar of the window indicates the player's current *score* and *hit points* (HP). HP represents the number of obstacles the player can collide with before they lose the game. Players start with `3` HP.
- The entities drawn in the window are as follows:
  - a blue square containing a **P** represents the *player* (or, more specifically, their avatar in the game).
  - a red square containing an **A** represents an *avoid*. When collected, the player's HP is reduced by `1`.
  - a green square containing a **G** represents a *get*. When collected, the player's score is increased by `20`.
  - a square flashing multiple colors containing a **G** represents a *special get*. When collected, the player's score is increased by `20` and their HP increases by `1` . *Special gets* are generated less frequently than *gets*.
- The game is over when the player either reaches a score of `300` (win) or their HP is reduced to `0` (lose).

Other controls include the **`Esc`**, **`D`**, **`P`**, **`+`**, and **`-`** keys, which you will investigate later. When the game is over, players must click the "close" button on the window to exit the game.

1.2 Provided Code

Provided to you are **11 `.java`** files. A high level overview is provided below:

- **`GameEngine`:** handles the most fundamental responsibilities and properties for a Scrolling Game. You **cannot modify** this file, but *will* need to trace it and call its methods.
- **`ScrollingGame`:** handles the logic for the scrolling game detailed above and playable in the provided demo.
- **`Entity`:** implements basic functionality and attributes for any kind of entity drawn in the game window.
- **`Player`, `Avoid`, `Get`, `SpecialGet`:** maintains the specific, individual attributes and behaviors for the various types of entities that get drawn in the game window.
- **`Collectable`, `Scrollable`:** dictates the functionality for an entity that is considered collectable (modifies the players score/HP when collided with) or scrollable (moves automatically each time the game updates).
- **`Launcher`:** contains the `main` method used to launch the game.
- **`GameWindow`:** draws and animates the graphics in the game window. Also captures keyboard/mouse inputs. You **cannot modify** and **do not need to trace** this file (though feel free if you're extra curious!)

Also provided is a **`.jar`** file (more on this below) as well as numerous image files located in an `assets` folder. The `assets` folder is designed to hold the game's art resources to make it easier for you to send your game code.

2 Your Task

Your task for this project will consist of **three** parts, with each corresponding to a **milestone**:

1. **Milestone #1:** Trace the code base, responding to a series of short answer questions.
2. **Milestone #2:** Restore the functionality of the Scrolling Game, such that it matches the `DemoScrollingGame.jar` (you may optionally work with **one additional person** of your choosing).
3. **Milestone #3:** Create your own creative game, with its own theme (incorporating custom graphics) as well as at least two new additional game mechanics not seen in the Scrolling Game (must be completed **individually**).

2.1 Playing the Demo

Provided is a fully implemented version of the Scrolling Game, contained inside **`DemoScrollingGame.jar`**. A `.jar` is a collection of Java code and other assets (like images and audio), consolidated into an executable. **Before continuing**, spend time playing the **demo**, which you can run by simply double-clicking on the `.jar`.

If you're on a Mac, you may run into a security error – if so, follow the steps below:

1. Click on the *Apple icon* in the upper left corner of your screen (on the menu bar)
2. Click on *System Preferences*. In the window that opens, click on the *Privacy & Security* category
3. Towards the bottom of the window that opens, click the *Open* or *Open Anyways* button to open the `.jar` file

After playing the demo, you should be able to **answer the following questions** (if not, ask your TA/instructor!):

1. What does pressing the **`P`** key do?
2. What does pressing the **`Esc`** key do?
3. What does pressing the **`+`**, and **`-`** keys do?
4. What does pressing the **`D`** key do, and what does **debug mode** show us?

2.2 Milestone #1: Tracing the Codebase

This first step will acclimate you to the code base, giving you an understanding of the provided code's functionalities and relationships. Follow the instructions and respond to the prompts on the following pages.

(a) To begin, only open **GameEngine**, **ScrollingGame**, and **Launcher**. Look at Launcher's main. What type of variable is **game** and what type of Object does it instantiate?

game is a _____ variable pointing to a _____ Object

(b) Trace the call to ScrollingGame's constructor. Eventually, GameEngine's two argument constructor is called; what are these argument's values and what do they determine?

argument #1 = _____ argument #2 = _____

These arguments determine:

(c) After the instantiation, Launcher's main calls game.play(). Fill in the blanks below:

The play() method is implemented in _____.java

play()'s implementation is **four lines of code**. Ignore the fourth line for right now. Identify the methods called by play()'s first 3 lines, and where they are implemented:

#1: _____ is implemented in _____.java

#2: _____ is implemented in _____.java

#3: _____ is implemented in _____.java

(d) What is the difference between a *background* image and a *splash* image? (*trace their respective methods and read the comments!*)

(e) If you wanted to change the game to run at double its normal speed, what function in GameEngine would you call and what would you pass it as an argument?

set _____ (_____);

(f) Trace the `gameLoop()` method and read its comments. When running, the Scrolling Game spends most of its time in this function.

In the space below, describe what `ticksElapsed` represents:

(g) On each "tick" of `gameLoop()`, the game state is updated by calling `updateGame()` **UNLESS** one of two scenarios are true. Describe these two scenarios below:

Scenario #1:

Scenario #2:

(h) In `GameEngine`, review **`captureInput()`**. Why does this method create an `ArrayList`?

(i) How many **abstract methods** does `GameEngine` declare? How many of these abstract methods does `ScrollingGame` override? Does this make sense?

(j) Eventually, you will implement player movement via the arrow keys, but you cannot modify `GameEngine`. How will this be possible? *(hint: review `ScrollingGame`)*

(k) Next, open all the remaining `.java` files. Take a look at **`Entity.java`**.

Review `Entity`'s instance variables and comments. Below, describe what a **hit-box** is:

(l) `Entity` has **one** *overloaded* method.

Identify this method and describe the difference between its two versions below:

(m) Look at `Get.java`. Why does `Get` have a `getDamage()` method?

(n) Look at `Player.java`. What does the return values of `setHP` and `modifyHP` represent?

(o) Return to `GameEngine.java`.

`GameEngine` has **one** overloaded method which returns an `ArrayList` of `Entity` Objects. What does this returned `ArrayList` represent?

(p) `GameEngine`'s instance variable `displayList` is one of the most important components of this codebase. Describe what it stores below:

(q) The graphics drawn in the game window have a particular order of layering, *i.e.* there is a consistent logic as to which elements will be drawn on top of other elements.

Below, order the layering from bottom-most to top-most of the **five** components listed:

- first index of `displayList` — **#1 (bottom-most):** _____
- last index of `displayList` — **#2:** _____
- splash image — **#3:** _____
- background image — **#4:** _____
- background color — **#5 (top-most):** _____

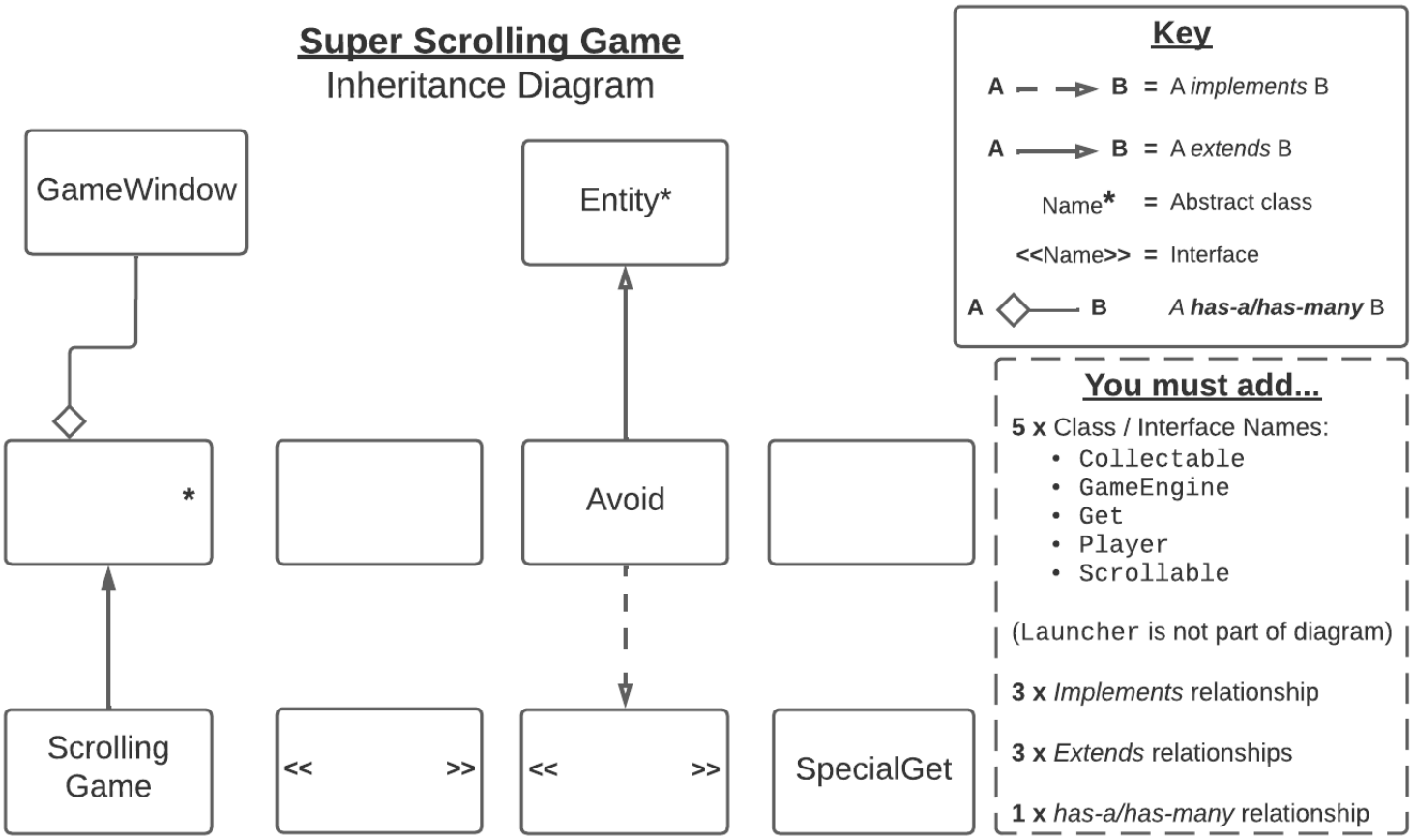
(r) You will eventually need to implement the logic to determine if two `Entities` are colliding with each other.

A clever way to do this is to check *all* the scenarios which would confirm the `Entities` **are <u>not</u>** colliding – if none of them are true, you know they *are* colliding.

Given two rectangles, **`r1`** and **`r2`**, you know they are **not colliding** if *any* of the following are `true` (fill in the blanks):

- **`r1`'s** *left-edge* is *to the right of* **`r2`'s** *right-edge*
- **`r1`'s** *top-edge* is *below* **`r2`'s** _____
- **`r1`'s** _____ is *to the left of* **`r2`'s** _____
- **`r1`'s** _____ is _____ **`r2`'s** _____

Finally, **complete the inheritance diagram** below. The list on the right side details what the diagram is missing.



***** You must check your diagram with a TA/Instructor before continuing! *****

Milestone #1 is due for all students **by the start of your next lab section** the week following this lab. You must show your packet with your responses to your respective lab instructor.

2.3 Milestone #2: Implementing the Scrolling Game

When you are ready to begin coding, your first task is to replicate the functionality of the Scrolling Game as seen in the `DemoScrollingGame.jar` by implementing the requisite functionality in **`ScrollingGame.java`**.

Specifically your Scrolling Game must:

- Start with the provided splash screen.
- Allow the Player to move their avatar same as the demo game.
- Allow players to pause and change the speed of the game.
- Generate `Avoid`s, `Get`s, and `SpecialGet`s at a rate similar to the demo game (not too challenging or too easy).
- End when the player reaches the win or lose state, with the appropriate messaging written to the title bar.

For Milestone #2, submit **only your 11 `.java` files** containing your implementation of the Scrolling Game to either your class' Moodle page. Before submitting, you will need to compress the 11 `.java` files into a single `.zip` file. The ability to do so is built in to both Windows and MacOS; Google it if you're unsure how!

Milestone #2 is due for all students on **Friday, November 10th** at **5:00PM**. You may optionally work with **one additional person** on Milestone #2, with the following criteria:

- You and your partner must have the same **lecture instructor**
- You and your partner must **both** submit the partner form on your lecture's Moodle page by **Thurs, Nov 2nd**
- Only **one person** from your pair should submit your Milestone #2 code. Please leave a comment on the submission Moodle page with your partner's name

2.4 Milestone #3: Creative Game

Once you have recreated the Scrolling Game, you will then implement your own creative game. You will do so in a **new class**, named **`YourLastNameGame.java`** (example: `LyboultGame.java`). This class will *extend* `ScrollingGame`.

Your creative game must:

- have a theme/story including a unique title, graphics (including splash and background images), and anything else that fits your concept.
- incorporate **at least two** additional game mechanics. These can include new rules, power-ups, levels, win/lose states, etc. For full credit, the features you add to your game should be creative and interesting.
- display a splash screen upon launching the game detailing the rules and controls.
- have a concrete win and lose conditions. Games that run forever are not appropriate – the player needs to be able to win in a reasonable time frame.
- be tuned to be fun to play. The speed and difficulty should make the game playable, meaning both winnable and challenging.

You are allowed to "borrow" images from the internet, so long as you cite their source in your submitted `ReadMe` file (more on this below). Any theme and graphics chosen for your game **must be appropriate for a general audience**. In other words, don't use anything you wouldn't want seen by your family or a prospective employer!

2.4.1 ReadMe

In addition to your code, you will also create and submit a `ReadMe` file. Your `ReadMe` will be a text document (meaning `.txt` file, **no *Word* or *Google Doc*** files please) containing the following:

- a description of your game's theme
- details of your custom features and game mechanics
- the specific win and lose conditions for the game
- sources for any "borrowed" assets (images, sound effects, etc)

2.4.2 Submission

You will again create a `.zip` file containing all of your "creative game" code, including `assets` sub-folder with all of the game's images, etc, and the `ReadMe` file.

Milestone #3 is due for all students on **Friday, December 1st** at **5:00PM**. Milestone #3 must be completed **independently**; you may not work with a partner.

Lastly, on **Thursday, December 7th**, from **11:30AM-12:30PM** in the COSC student lounge, our department tea will be dedicated to a Scrolling Game showcase where you can show off your game to fellow students in 101, 102, and beyond. Attendance isn't required, but we hope you will attend – there will be food and prizes (just for fun) as well!

3 Tips

Below are a few tricks to help you in your implementation:

- Work incrementally; set small goals, implement one feature at a time, save, compile, and test frequently, and use lots of helper functions!
- If you are stuck, look to your provided class/instance variables and methods (particularly in parent classes). All the provided code has descriptive comments – read them! Don't be afraid to refer back to the questions in Milestone #1 too!
- `PNG` and `GIF` images work best with the game. There are many free tools available via the web for things like making transparent backgrounds or animated `GIF`s (*Canva* has become very popular). There are also editors you can download and install on your Windows/MacOS computer (such a *GIMP*, *Paintbrush*, *Inkscape* and more).
- Related to above, any image for an *Entity* or background/splash screen will be stretched to fit the Entity's dimensions or the game window, respectively. That said, your graphics will look best if they are of the appropriate resolution and/or aspect ratio (*ie*, height-to-width ratio).
- Using too many debug print statements will adversely impact the speed and performance of the game – this may be ok for testing, but be sure to remove these from your submitted code. It may be more valuable to use the on-screen debug text instead via the **`setDebugText(...)`** method.