# COMP20300 40% Java Project:
# A JavaFX Implementation of Simon's Race

Last Updated: September 2022

## 1 Key Submission Details

**Deadline:** Dec 5th/6th 2022, 5pm (all deliverables uploaded to Brightspace).

**Late Submissions** Standard UCD policy on late submissions applies; see `https://www.ucd.ie/t4cms/latesub_po.pdf`. Your submission is deemed late if at least one deliverable is submitted late.

**Plagiarism:** The submission must be yours and yours alone. If you are unsure what is or is not plagiarism, the following is a none exhaustive list of example activities you cannot do:

- Copy the completed files of another student and submit them as your own
- Share copies, images or print outs of your code with another student (by e-mail, FB messenger, WhatsApp etc.)
- A group of students working on a single solution and then all submitting the same work or subset of the same work (regardless of whether variable, method, class names or ordering have been changed)
- Students collaborating at too detailed a level. For example, consulting each other after each line / block / segment of code and/or sharing the results.

For more details see:

- `https://csintranet.ucd.ie/sites/default/files/cs-plagiarism-policy_sept2020.pdf`, and
- `https://www.ucd.ie/governance/resources/policypage-plagiarismpolicy/`
- `https://www.ucd.ie/secca/studentconduct/`

Any submission suspected of plagiarism will be submitted to the School of Computer Science Plagiarism subcommittee for further investigation.

## 2 Game description

The game is called Simon's Race, it is a made up game, so there is no point Googling for implementations of it. Simon's Race is a board-based game which has a start line and a finish line, with a predefined number of cells in between. The goal is easy: the player to get from the start to the end first is the winner. However, there are a few rules that make this a little more complicated! The rules which determine if a move is valid are:

1. A 4-sided dice is rolled to determine how many squares a player can move: 1, 2, 3, or 4 squares, and another 4-sided dice determines in what direction, forward, forward, backward, miss a turn, i.e. there is a 50% chance of moving forwards, a 25% chance of moving backwards, and a 25% chance of missing your turn.

2. Players always move in the direction specified unless they are obstructed, in which case they can choose to move sideways or, if they prefer, end their turn.

3. Obstructions include: other players, the edge of the board, and obstacles in the game (fires, fences, pits, teleporting portals etc.).

To provide some context, consider the setup displayed in Figure 1. Here, there are 2 players, the red and the purple player. There are four obstructions: three bottomless pits, and one fire. Note that the fire consumes two squares (and thus you can have obstacles that occupy multiple squares). If the red player rolls 3 or more & forward, they will have to move to the right, to avoid the pit. The player that reaches the green finish area first, is the winner. However, note also that the area of the board circled in red in Figure 1 is a potential trap. If a player, happens to end up in this location, they could become stuck for a several turns if they roll the direction backwards at any point!

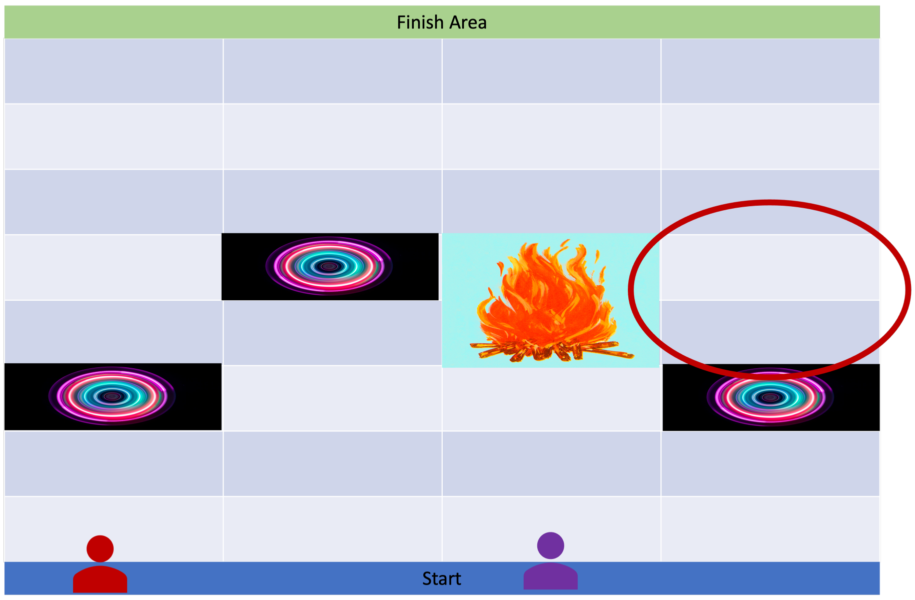

Figure 1: Example Simon's Race Board. Note that the area in red, is a potential trap, where a player could get stuck for 1 or more turns.

Note the following:

1. you need 2 or more players, but at most, one per lane (there are 4 lanes in Figure 1)
2. the player must select the square they wish to move to (you can decide how they do this)
3. you can have as long a board (i.e. the distance between start and end) and as many lanes as you like
4. the board should fit on the screen in at least one direction, i.e. either the full width (lanes) or full height of the board should be visible, the other direction can scroll if needed.
5. you can have as many obstacles as you like, however, as discussed in section 3 there are some minimum requirements

There is a lot of free design space for you, your game can look very different to the mockup shown in Figure 1: it is only a quick illustration. There is no legitimate reason for two projects to look the same, and as such two projects that are "too similar" may

## 3 Requirements

In this assignment you are making a multi-player game according the game brief presented in section 2, however, the following **MINIMUM** requirements should be observed:

1. All conditions noted in section 2 must be observed
2. The game **MUST** have a 2D JavaFX user interface (which can be quite simple: fancy animations are **NOT NEEDED!**)
3. When a player wins Simon's Race the game should end immediately and "announce" the winner
4. The players should be able to specify their name, this should be displayed in the interface

5. The interface should illustrate current value of the game dice
6. The game must well unit tested
7. You need to derive a scoring mechanism for the game, and have a persistent top 10 score board (e.g. in a flat file, serialised object, etc.); suggestion: number of games a player has won, how many squares a player has moved, etc.
8. all code should be placed within a well-structured Java package with accompanying JavaDoc that provides detailed information for all classes, methods, fields etc.
9. you must have at least 3 different types of obstacle, one of these should have “special rules” i.e. modify the game play in some manner, e.g. a button that when pressed (stepped on) rearranges the board, a tar pit that makes the player miss their next turn, a teleportation portal that swaps the position of both players, etc.

All these requirements must be fulfilled to achieve a B grade or higher. To increase your mark, you can add adaptions to the game that increase its complexity and allow you to go beyond the minimum requirements (above). **However, this should only be undertaken once all minimum requirements are met**. Example adaptions:

1. user-specified board sizes and layout (e.g. of obstacles)
2. a difficulty setting (more or less obstacles)
3. allow the player to undo moves (e.g. as a practice mode)
4. randomly generated boards (with user input to specify key aspects, e.g. number of lanes, distance between start and finish, number of obstacle types, difficulty settings, etc.)
5. record (persistently) the results of previous games (e.g. in a flat file, serialised object, etc.), i.e. the board state when the game ended, and be able to reload a previous final state for a player to inspect
6. < add your own ideas here >

## 4 Deliverables

| | |
|---|---|
| **Demo Video** | a (max of) 7.5 min video (with audio commentary) demonstrating your game, and key unit tests, you should also illustrate how the unit tests thoroughly test game in the video. You should use OBS to record your demo (see OBS tutorial on brightspace). |
| **Screenshot** | of your application showing an on-going game in anything but the start or end state |
| **Documentation** | a Java Doc for all classes as a .zip |
| **Code** | all .java files as a .zip archive |
| **README.txt** | a file explaining how to run the game, and tests for each class |
| **Challenge Video** | a (max of) 3 min video (with audio commentary) where you will perform a set a unique (to you) tasks. These will be released after the submission deadline. You need to record yourself making these changes to the code and running the result. These may include, but are not limited to: 1) change some part of the UI, reload the game to show these change, 2) initialise the board with a specified state, 3) modify the code in various ways to create specified situations then demo these situations, etc. You will have 24 hours to answer these questions. None of these questions will be difficult if you have a good understanding of your code. |

## 5 Tips and Grading Rubric

**Problem Definition** start the project by defining a problem definition, this is essentially a list of things your game needs to do. Work on paper to identify potential areas of functionality when considering classes to implement; your main classes are probably the nouns in your problem description e.g. board, piece, scoreboard, UI, etc.

**Interface** start with a command line interface: so that you can get the move logic correct. A functioning game is more important than a pretty interface!

**Start** with the "board" or environment data structure and implement code that makes a move / takes a turn (this, initially, should not include the UI!). Then add code to check a move / turn is valid. A simple game, well implemented is better than a complex game that doesn't work. Once you can check for valid moves, add functionality to check for the win / lose condition. Then build up to multiple turns, then start planning the GUI, etc. You should be using the model view controller design pattern for this project, and as such you can build most of the core functionality without a UI!

**Testing** should be undertaken as you implement key aspects of functionality. Don't leave it to the last minute. Key testing in this project should look at valid moves, changes in board / environment state, identifying the win / lose condition, etc. A poor project has less than 20 or so distinct test cases.

**Workload** it is expected that this project require 40-80 hours of work to complete.

**Grading** the game itself is "only" worth 50% make sure you allocate sufficient time for the other aspects of the project: video, testing, JavaDoc, etc.

**Model-View-Controller** this is the best design pattern to use for project.

**Scene Builder** makes the design of the UI much easier and enforces an MVC approach to the project.

| Criteria | A+, A, A- | B+, B, B- | C+, C, C- | D+, D, D- | $\leq$ E+ |
|---|---|---|---|---|---|
| **Game Implementation (50%)** | Class design is aligned with OO principles and evidences high cohesion. All minimum requirements met. One or more advanced features implemented. Game is functional, and evidences complexity. | Class design is commensurate with fundamental OO principles. All minimum requirements met. Game is functional, and evidences some complexity. | Class design is acceptable, but there would be better ways to implement the solution. OO principles are attempted. Most minimum requirements met. Game is functional, but simple. | Class design is acceptable, but there would be much better ways to implement the solution. Most minimum requirements met. Game mostly functional, but potentially a few features are unreliable or barely functioning. | Class design is poor. Minimum requirements not met. Game perhaps only partially functioning. |
| **Testing (25%)** | A very thorough set of unit tests are implemented resulting in a well tested game. | A thorough set of unit tests are implemented, yet some aspects of code coverage are limited. | A good attempt at unit testing all core functionality. Test cases capture both expected behaviour and some fringe cases. | Evidence of unit testing for most core functionality, but is not extensive. | Testing is superficial or (mostly) not present. |
| **JavaDoc (15%)** | A thorough JavaDoc is present for all classes, and captures all essential details. For key classes / methods, meaningful examples of how to use them are also present. | A JavaDoc is present for all classes, and captures all essential details. | A JavaDoc is present for all classes, and captures most essential details. | A JavaDoc is present for all classes, and captures essential detail, but lacks depth. | Superficial or not present. |
| **Video (10%)** | A well-conceived video demonstrating all key functionality. | A well-conceived video demonstrating key functionality. | A well-conceived video demonstrating a functioning game. | A demonstration video is provided that shows a functioning game. However, the video is poorly conceived and/or lacks depth. | A demonstration video may be provided, but is poorly conceived or does not clearly illustrate a functioning game. |