Assignment Submissions

This project will be submitted in two parts. Both parts of the assessment are equally important for the successful completion of your project, so it is essential that you understand the requirements of both parts ***before*** you start.

You may complete the tasks in your preferred IDE; however you **MUST** create a CLion project in order to submit. Your project folder must be identified by using your name and assignment number with the entire folder zipped into a single zip file for submission.

- ### Assignment 2 :: Risky Business Simulation (Part A: Project Plan & Prototype)

  Due Date: Sunday of Week 8

  Marks: This assignment will be marked out of 100 points.

  Weighting: 25% of your final mark for the unit.

  This assignment is the first part of a larger project, which you complete in Assignment 3. This task consists of your project documentation and an initial prototype.

  The documentation must include a flowchart, all UML class diagrams of your programme and a test plan for your final project.

  The prototype must include the application class (Main) and the following base classes, including individual header and definition files:

  - **Player:** stores data about the player such as their name, health, inventory, etc.
  - **Company:** stores data about the different companies (inheritance applies here)
  - **Risk:** stores data about "risks" the player may take during the game – please note that there is also an advanced Extra Functionality option for this class

  The Main class, for this part of the project, focuses on the testing of each of your classes – similar to the way you tested the different classes in the studios from Week 5 onwards and displaying the main user interface correctly.

  The purpose of this assignment is to get you comfortable with planning and developing a C++ programming project which you will complete for Assignment 3. The task is detailed later in this assignment specification, as are the specific marks allocation.

- ### Assignment 3 :: Risky Business Simulation (Part B: C++ Project Implementation)

  Due Date: Monday of Week 14

  Marks: This assignment will be marked out of 100 points.

  Weighting: 35% of your final mark for the unit.

  This assignment consists of your implementation of your project, as outlined in your Project Planning document (Assignment 2) including one Application file and associated custom Class files, including all header and cpp files, and any appropriate text files to ensure the programme compiles and runs.

  The purpose of this assignment is to get you comfortable with designing and implementing basic multi-class C++ programmes and following your project plan. The task is detailed later in this assignment specification, as are the specific marks allocation.

Submission Instructions: Read These Carefully

This project will be submitted in two parts:

- **Assignment 2 – Part A:** consists of your project plan documentation and prototype.

  The document must include an overview of your program structure as a flowchart, UML Class diagrams and test plans for the required scenarios. You can use Microsoft Visio to draw the flowchart and UML diagrams, or you can use any other software, provided that the diagrams are included in your submitted document (PowerPoint is also a useful tool).

  The documentation must be created and submitted as **a single Word or PDF document** and the prototype must be created in **CLion** as a single project. Both tasks must clearly identify both your **Name and Student ID** to facilitate ease of assessment and feedback.

  Your document **MUST** be named as follows:

  "***A2-YourFirstNameLastName***.docx" or "***A2-YourFirstNameLastName***.pdf".

  Your **CLion project folder** must be identified using your name as follows:

  "***A2-YourName-Prototype***"

  All files must be submitted via the Moodle assignment submission page **in a single ZIP file** that **MUST** be named "**A2-*YourFirstNameLastName***.zip".

  Explicit assessment criteria are provided, however please note you will also be assessed on the following broad criteria:

  ✓ Detail of a proposed project plan for the overall project.

  ✓ Creating accurate and complete flowchart and UML diagrams.

  ✓ Applying a solid Object-Oriented Design (OOD) for the overall project

  ✓ Using appropriate naming conventions, following the unit Programming Style Guide.

- **Assignment 3 – Part B:** consists of your implementation of your final game project.

  Your project must follow your project plan and must be submitted as a CLion project, including all header and code files, and any appropriate text files to ensure the programme compiles and runs.

  You may complete the tasks in your preferred IDE, however you **MUST** create a CLion project in order to submit. Your project folder must be identified using your name and assignment number: "***A3-YourName-Final***".

  The entire project folder must then be zipped up into one zip file for submission. The zip file **MUST** be named "**A3-*YourFirstNameLastName***.zip". This zip file must be submitted via the Moodle assignment submission page.

  Explicit assessment criteria are provided, however please note you will also be assessed on the following broad criteria:

  ✓ Meeting functional requirements as described in the assignment description

  ✓ Demonstrating a solid understanding of specific C++ concepts relating to the assignment tasks, including OO design, implementation and appropriate use of pointers, inheritance and polymorphism, a good programming practice techniques.

  ✓ Creating solutions that are as efficient and extensible as possible

  ✓ Reflecting on the appropriateness of your implemented design and meeting functional requirements as described in the assignment description

Please ask your tutor for clarification of any aspects of the assessments you do not understand as soon as possible because this project is worth 60% of your overall marks.

Major Project Overview: Risky Business Simulation

In this assignment, you are to design a **Risky Business Simulation** game, where you buy and sell shares in a number of different companies. You must use your corporate powers gained from the acquisition of companies to increase your wealth and assets in order to win the game!

For Part A of the assignment you will focus on documenting and prototyping the project with the three base classes required demonstrating your class design and basic testing to ensure the classes are ready for your final game implementation. In Part B you will focus on developing the game setup and player interactivity demonstrating the implementation of the game interface design and the more complex aspects of the player's interactions within the game environment.

Game Overview

Objective: You must manage a Share Market Portfolio by buying and selling the shares in one or more companies. To be the first player to own the target number of companies and have at least the minimum amount of cash to win the game.

Setup: You set the Difficulty level (1-3) and the number of players (1-4) with their names. The number of companies to own, the minimum amount of Cash in Hand required and the number of days you have left is set by the Difficulty Level.

Play: The game can be quite tricky, so be careful to take risks only as necessary and manage your assets as responsibly as possible.

The game is played over a number of days, in which each player has one turn. You may choose one of the following actions:

- ✓ [B]uy shares in a company - up to as many as you can afford
- ✓ [S]ell shares in a company - up to as many as you own
- ✓ [A]cquire a company by paying its cost in shares
- ✓ [P]ower uses a corporate power to gain an immediate bonus
- ✓ [R]isks may pay off (or not) so use with caution (advanced)
- ✓ [Q]uit ends the game for the player selecting it

The Share Market changes the price of shares every day so think carefully when you will buy and sell your shares.

The Required Game Interface

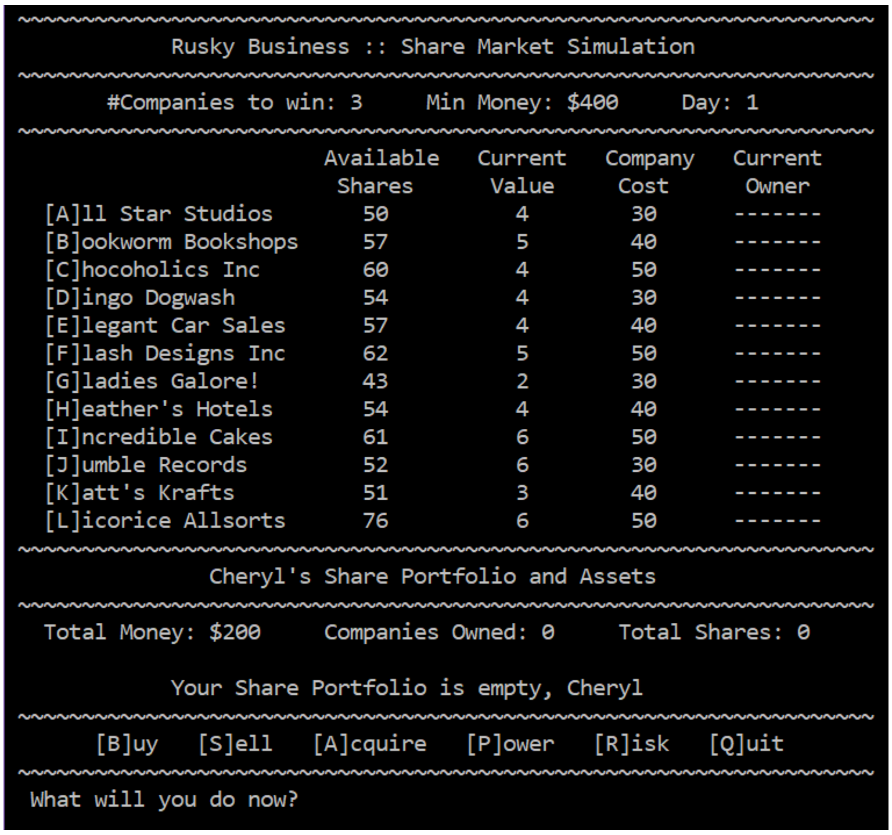
The main game interface must display all the information as shown here ➔

This includes the following:

- ✓ The game title.
- ✓ The required win/end conditions.
- ✓ Display the current values for all company shares, value and cost.
- ✓ The title of the current player and total assets and share portfolio.
- ✓ Followed by the options a player may take during their turn with an appropriate prompt message.
- ✓ This screen must be cleared and updated after a player's turn (so there is no scrolling).

Individual prompts and procedures are detailed in the relevant sections under Part B.

For ease of assessment you must use the required interface (shown here and partially in later screenshots).



```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
            Rusky Business :: Share Market Simulation
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
       #Companies to win: 3     Min Money: $400     Day: 1
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                        Available   Current   Company   Current
                         Shares      Value     Cost      Owner
  [A]ll Star Studios       50          4        30      -------
  [B]ookworm Bookshops     57          5        40      -------
  [C]hocoholics Inc        60          4        50      -------
  [D]ingo Dogwash          54          4        30      -------
  [E]legant Car Sales      57          4        40      -------
  [F]lash Designs Inc      62          5        50      -------
  [G]ladies Galore!        43          2        30      -------
  [H]eather's Hotels       54          4        40      -------
  [I]ncredible Cakes       61          6        50      -------
  [J]umble Records         52          6        30      -------
  [K]att's Krafts          51          3        40      -------
  [L]icorice Allsorts      76          6        50      -------
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
               Cheryl's Share Portfolio and Assets
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  Total Money: $200     Companies Owned: 0     Total Shares: 0

            Your Share Portfolio is empty, Cheryl
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
      [B]uy   [S]ell   [A]cquire   [P]ower   [R]isk   [Q]uit
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 What will you do now?
```

Player Interactions

The choices a player has include [B]uy shares, [S]ell Shares, [A]cquire a Company, Use a Corporate [P]ower, Take a [R]isk and [Q]uit. The letter [?] is the shortcut key the player can use to execute that command. You can read a full description of each command option in Part B of the brief.

On your turn, you may choose one of the options shown. All advanced features are described in detail under the Extra Functionality section and make up 20% of the total marks for Part B of the project.

If you choose not to implement any advanced features you will forfeit these 20 marks.

Game Data is Provided

Text data files are provided for the introductory game blurb, Companies and the Risks as well as the permitted re-usable functions. Please use this data in your prototype and final game. A detailed breakdown of these files is provided in Part B of this document.

Assignment 2: Preparing the Risky Business Simulation (Part A)

Project Documentation

Having a clear plan for your project before you begin coding is essential for finishing a successful project on time and with minimal stress. So part of this assignment is defining what will be in your final project and what you need to do to actually develop it.

> ***Important:*** *You must also read the requirements for Assignment 3 in order to be able to complete the documentation required for Assignment 2.*

The documentation you must submit will include the following:

✓ *Flowchart of the entire game structure*

In a nutshell, a flowchart is a visual representation of the game's algorithm. It's a useful tool that can help you see how your programme works. It makes it easier to focus on the flow of the game while developing your programme because you have already identified where all the game decisions and loops will be.

The symbols you'll most likely need are:

Symbol	Title: what is it used for
⬭	**Terminator:** Use at the beginning and end of a flow sequence. When you begin a sequence, add the name of the sequence in the symbol (eg: display map).
▭	**Process:** Use to describe the current step or action in the sequence (eg: get input, read data from text file, display data in varName, etc.).
◇	**Decision:** Use when the programme must choose something usually in the form of a question (eg: is Company owned? Has money left? etc.). Evaluates to true or false.
◫	**Pre-defined Process:** Use when you want to link to another flowchart. This is only necessary for processes that require a finer level of detail (eg: "buy shares" would appear in the main flowchart then you could create another smaller chart describing what happens when you buy shares). It's not necessary for things like, get user input or display title.

The level of detail required is "big picture" not "fine detail". We don't necessarily need to know all the fine details, as that is something we'll see in your code, but we need to be able to understand how you are thinking and processing the various elements within your game.

Remember to keep your symbols concise and easy to understand by someone else. This is meant to be read by others so now is not the time to use secret codes that only you understand!

> **Tip:** Use the **Process** box to show things happening in a sequence and the **Pre-defined Process** box to identify a function that encapsulates a specific task using an appropriate description.

Your flowchart should include:

- The game setup (everything that happens before the game starts)
- The player's turn (the sequence of events that happen during a turn)
- Processing player input (the commands your player can use)

…4

- Providing feedback to the player (in response to the player's interactions)
- The end game conditions (include all win and lose conditions)
- Include the functionality of all your game classes – see Assignment 3
- Any additional Features included, if any – see Assignment 3

Here is an example to get you started with an algorithm that you can convert into your flowchart:

- The Game Setup
  - Read game rules from text file and display
  - Add the player(s) – ask for the player's name, set default variables
  - You must include all the other things that will happen during initialisation, such as:
    - creating the companies, the risks and the game interface
    - initialising other game variables

As you can see, you only have to describe the actions the program will take, not the code. The idea here is to give you a starting point for when you start writing your code as you can use this as a checklist of the things you need to include.

✓ *UML Diagrams*

UML diagrams are designed to make structuring your program easier. How to create them will be covered in class, but the general structure is shown here – see Assignment 2 brief for more details about classes.

You will need UML diagrams for each of the classes you include in your game – at least a Player, Company, Risk and the Main (application) class.

ClassName
list of attributes (variables)
list of behaviours (functions)

✓ *Project Test Plan*

Part of any project development strategy is the inclusion of a project test plan. This document lays the foundations of how you plan to test the various elements within a project to ensure that it remains robust and does not crash when being used by others.

In this part of the assignment you are required to identify how you will test each individual class (as per your prototype) and the main game play elements (as per your final implementation). We will use this test plan when we are marking both your prototype and final project to ensure that you have done the required amount of testing.

We expect evidence of testing for the 3 required classes, and the following aspects of the game play – selecting options, and the win or lose end game conditions.

Create a table in your document, using the following example provided as a guide, filling each section with appropriate information that is specifically relevant to your implementation of the game.

How to create a test plan will be covered in the workshops with appropriate examples. You'll also have time during the workshops and studios to practice creating your own.

Class	**Function/Purpose**	**Sample Data Tested**	**Actual Result**
Class Name	What's its purpose? Outline its purpose in the game context.	How did you test it? Include a range of values you tested to ensure the class did not cause an error.	What was the result? Include both successful and failed test results.
add other classes here			
#Scenario	**Function/Puropse**	**Sample Data Tested**	**Actual Result**
Related to Scenario [?]	functionName() What's its purpose?	How did you test it? Include a range of values you tested to ensure the function did not cause an error.	What was the result? Include both successful and failed test results.
add other functions here			
You must include all your classes and all the related functions for the following game scenario criteria: 1) **selecting an option:** include each one and the test done to ensure correct functionality 2) **end game conditions:** include the player winning or losing the game with all possible outcomes			

Project Prototype: What Needs to Work

Your prototype is **NOT** the completed game. Its purpose is for you to demonstrate your initial development process of the custom Base Classes (Player, Company and Risk) and to provide appropriate evidence that each has been thoroughly tested before you begin final implementation.

The following are brief overviews of the 3 base classes you must include in your prototype, but please read the class descriptions under the Assignment 3 brief for more detailed information. Our expectation is that these classes are as close to fully formed as you can make them and should not require much additional development in Assignment 3, other than a few small tweaks as required.

You are permitted to use similar processes to the ones demonstrated during Weeks 6-8 for this part of the assignment, such as calling **testPlayer()**, **testCompany()** and **testRisk()** functions from the **main()** function.

✓ ***Player Class***

This class holds the player's details including their name, game mode, initial values for their money, companies owned, total shares, and power uses left attributes, and the details about the companies in their portfolio (similar to an inventory).

✓ ***Company Class***

This class holds the company's details including its name, initial values for its owner, cost, index, level (Bronze, Silver and Gold is used for inheritance in Part B), corporate power and share price attributes.

The company level is used for determining the values of the corporate power associated with each level of company. The Bronze, Silver and Gold Company classes inherit their base details from the Company class but will have their own specific functionality.

NOTE: Applying inheritance is taught in Weeks 8-9, so you only need to include the Bronze, Silver and Gold Company classes with their appropriate powers in your final implementation.

✓ ***Risk Class (Default or Advanced)***

The default version of this class randomly selects from adding either a small bonus money or shares to the player. There is a 2 in 6 chance that the player will lose money or shares instead.

In the advanced version of this class it holds the details of the different Risks the player may encounter including its name, effects, minimum and maximum values attributes.

Risks are randomly selected from the entire list and the results of the selected Risk is calculated via the appropriate function call in the main application based on the "effect" attribute. You will need to think carefully about this implementation in your final submission but for the prototype you only need to display the data information.

✓ ***Main (application)***

For each class you must demonstrate you can create a Class Object using both the default and an overloaded constructor with appropriate parameters and a have appropriate accessor and mutator functions for each attribute.

All classes must also include a **getDetails()** function that displays all the generated data of the class as per the testing procedures demonstrated during the studios from Week 5 onwards.

The data used for the companies and risks must be read from the provided text data files and used in the creation of your class objects.

The final thing that is required is displaying the game interface correctly (as shown on page 3). The variable values should be randomly generated using appropriate formulae, demonstrating that you can use appropriate formatting to display all the required information. This should also be encapsulated in one or more appropriate functions which you can easily transfer to your final project.

Assignment 2: Marking Criteria [100 marks in total]

NOTE: *Your submitted project must be correctly identified and submitted, otherwise it will receive an automatic* ***10% penalty*** *(applied after marking).*

Does the program compile and run? Yes or No

NOTE: *Your submitted program MUST compile and run. Submissions that do not compile will receive* ***ZERO****.*
Ensure you leave time to test your work thoroughly before you submit it.

Project Documentation [50]

- Flowchart Design **[17]**
  - Has used the correct flowchart symbols [1]
  - Has used appropriate descriptors in flowchart symbols [1]
  - Flowchart includes all game functionality (Parts A & B) [2]
  - Flowchart clearly shows correct sequence of the game (setup, gameplay, end game) [4]
  - Pre-defined processes include all the required game functionality [7]
  - Processes are performed in a logical order [2]
- UML Class Diagrams **[13]**
  - Correct structure used (Name, Attributes, Behaviours) [1]
  - Included the correct designations for public [+] and private [-] data [1]
  - All variables and functions have meaningful names [1]
  - Included constructor(s), destructor, and appropriate accessor and mutator functions in all custom classes [6]
  - Included a class diagram for the Player, Company, and Risk classes and the Main (application) [4]
- Project Test Plan **[20]**
  - Each class tests all constructors, accessor and mutator functions [12]
  - Appropriate testing procedures for selecting options and end game conditions [8]

Prototype Class Design [40]

- Overall Class Implementation **[12]**
  - All classes have a correctly structured header and definition files [3]
  - All classes include appropriate constructor, destructor, accessor, mutator functions [3]
  - All classes include all the required attributes as given in the class descriptions [3]
  - All classes include a **getDetails()** function to display all the class attributes appropriately [3]
- Individual Class Implementation **[12]**
  - The Player class initialises its attributes with random values based on the game mode [1]
  - The Player class includes appropriate accessor and mutator functions [2]
  - The Player class objects are stored in an appropriate collection variable [1]
  - The Risk class initialises its attributes with data read from the required data file [2]
  - The Risk class includes appropriate accessor and mutator functions [1]
  - The Rick class objects are stored in an appropriate collection variable [1]
  - The Company class initialises its attributes with data read from the required data file [2]
  - The Company class includes appropriate accessor and mutator functions [1]
  - The Company class objects are stored in an appropriate collection variable [1]
- Does each class have a **testClassName()** function that demonstrates the following: **[18]**
  - Create each class object using the default constructor and displays its details [4]
  - Create each class object using an overloaded constructor and displays its details [4]
  - The created class object can access each of its attributes and display it [4]
  - The created class object can modify all appropriate attributes and display the changes [4]
  - The class tests follow the tests described in the test plan [2]

Prototype Quality of Solution and Code [10]

- The code is well formatted and easy to read **[2]**
- Correct naming conventions have been applied **[4]**
  - Including all functions/methods and variables using meaningful names and applied camel case
- Appropriate commenting and code documentation **[4]**
  - Including file headers containing author's name, date, purpose, one line comment before each functions, non-trivial code has additional comments

…7

Assignment 3: Risky Business Simulation Implementation (Part B)

You are to implement the final version of the Risky Business Simulation game you started in Assignment 2 by completing your Visual Studio Project using your project prototype as presented in your previous submission.

Modifications to your prototype are permitted provided that you document the changes and justify why to did so. This should be included as part of your reflection document.

NOTE: *Please adjust the naming of your files as stated in the submission requirements.*

Project Class Design

You MUST implement your program using the following classes, as a minimum, but you may include more if it is appropriate for your game design:

- **Player class:** holds the player's details including their name, initial values for their difficulty level, money, companies owned, total shares, and corporate power uses left attributes, and the data relates to the individual companies in their portfolios, as defined below:
  - **name:** the player is asked for a name to use at the start of the game.
  - **mode:** is initialised using the selected difficulty level at the start of the game
  - **money:** is initialised using the formula: 5 x (10 x mode).
  - **total shares owned:** keeps a tally of all the shares bought by the player.
  - **total companies owned:** keeps a tally of new companies bought by the player – once owned the player never loses ownership even if they sell all their shares in that company.
  - **power uses left:** is initialised to the mode – once it's at zero this is disabled.
  - **company details:** you can store this data in one or more collection variables for easy access to the information when displaying the player's share portfolio.

  These attributes are modified by the shares and companies the player buys and sells during the game. You must include appropriate accessor and mutator functions to both validate sent data and to return data or do the appropriate updates as required.

- **Company Base class:** is the base class for all the companies used in the game. This class holds the company's details including its name, owner, cost (in shares), level (Bronze, Silver or Gold) and current share price (a random integer value). You may include other variables as suitable to your game design.
  - **name:** assigned during setup from the data in the `companies.txt` file provided
  - **owner:** is initialised to an empty string or something similar to "-------" or "Nobody".
  - **cost:** is initialised using the formula: 2 x (5 x level) – this is the number of shares in the company the player must own then pay to buy ownership of it.
  - **level:** is a numeric value equivalent to the level (3=Bronze, 4=Silver, 5=Gold) which is set when the companies are created and is used when calculating the price of the shares each day.
  - **shares:** is the current number of shares available in the company which is set when the companies are created and updated when a player buys or sell shares in the company
  - **share price:** is the daily value set for buying and selling shares – the minimum value is 1 and the maximum value is (level + 4).

  These values are initialised based on the type of company (Bronze, etc.) using its company level and powers (see inheriting classes below). You must include appropriate accessor and mutator functions to both validate sent data and to return data or do the appropriate updates as required.

- **Bronze, Silver and Gold Company classes:** inherit from the Company base class. Each of the companies is one of these types (Bronze, Silver, Gold) as each type has a different company level and Corporate Power.
  - **Bronze:** companies are level 3 and give extra cash when using its corporate power.
  - **Silver:** companies are level 4 and give bonus shares when using its corporate power.
  - **Gold:** companies are level 5 and give bonus assets (both extras cash and shares) when using its corporate power.

  Initial company data is provided in the `companies.txt` file provided. This data must be used to initialise the companies by type (1=Bronze, 2=Silver, 3=Gold) with a name and the activation key in [?] and the associated Corporate Power. For example:

  ```
  1;[A]ll Star Studios   ;+ money
  2;[B]ookworm Bookshops ;+ shares
  3;[C]hocoholics Inc    ;+ assets
  ```

The number of companies used in the game is also dictated by the difficulty level chosen at the start of the game – 12 in Easy mode, 15 in Tricky mode and 18 in Hard mode. So when initialising the companies before the game begins, you must only read in the data for the required number of companies.

- **Risk Class (Advanced):** has a range of different "events" that randomly selects one when the player takes the [R]isk action. Initial risk data is provided in the **`risks.txt`** file provided.

  ```
  Global Recovery: Everyone Collects $;money;-1;50
  Performance Bonus:;assets;5;10
  Company Share Bonus: ;shares;5;10
  Market Crash: Lose Shares ;shares;5;-10
  ```

  The data is broken down into a description, what it effects (money, shares or assets=both), and the range as a minimum and maximum value.

  - A **negative minimum** value always calculates money based on the (maximum value x the game mode) and is added to the player's current cash.
  - A **negative maximum** value always indicates the result is subtracted from the player's current cash or shares (as indicated by the effects).

  Generated Risk objects must be stored in a collection variable and shuffled to make random selection easier. However, the results of the selected "risk event" should be processed by the application and NOT in the Risk class itself.

  The class must only hold the required data members with appropriate accessor and mutator functions to both validate sent data and to return data or do the appropriate updates as required.

You may include other relevant attributes and behaviours to the above classes, as identified in your project document, that are required by your overall game design. Please ensure you read the following detailed descriptions of the functionality expected in your final project submission.

Project Functionality

- **Main (application)**: holds the main() function and appropriate other functions to control the overall flow of the game.

  Your main game interface will be displaying this data to the player in an uncluttered and easy-to-read manner, reflecting the current status of the game and current player data appropriately. See page 3 for a screenshot of the required interface.

  Your completed Risky Business Simulation game must demonstrate the following:

✓ The **Player** class must be able to do the following:

- assign a name which is requested at the start of the game and used in the feedback given
- assign the player's initial attributes to the required values (as stated in the brief)
- displays the player's portfolio during the game as required (similar to the getDetails() but in greater detail)
- check and updates the appropriate player attribute when companies are owned, shares are bought and sold, or a Corporate Power is used
- the player does not lose ownership of a company when they sell all their shares

✓ The **Company** base class and the inheriting **Bronze**, **Silver** and **Gold** classes must be able to do the following:

- assign the company's initial attributes of name, owner, cost, and level based on the company type (Bronze, etc.)
- randomise the total shares based on the company level with the minimum number of shares equal to the company cost plus a random value from 10-20 x the number of players
- randomise the share price based on the company level to fit within the range of 1-(level+4)
- apply appropriate inheritance and polymorphism techniques as demonstrated in class

✓ The **Risk (Default)** class must do the following:

- assign the risk's initial attributes of description, effects, minimum and maximum values as collections variables within the class
- randomly choose one of the 6 options, taking the 4 elements from their associated variables, to do the appropriate calculations before returning the results
- use appropriate accessor and mutator methods to return or set the required data (risk data does not change once it is initialised)

✓ The **Risk (Advanced)** class must do the following:

- please read the Extra Functionality section for more specific details for this class
- assign the risk's initial attributes of description, effects, minimum and maximum values
- use appropriate accessor and mutator methods to return or set the required data (risk data does not change once it is initialised)

✓ The **Game Application** (Main) must do the following:

- display the "how to play" information from the `RBintro.txt` file at the start of the game
- initialise the player, companies, and risks with appropriate attributes and behaviours
- display the ***required user interface*** providing relevant information to the player at all times
- allow the player to activate the permitted options using the indicated letter – accept either UPPER and lowercase input and process accordingly
- display the current player's portfolio at the start of each player's turn
- terminate the game (player wins) when a player met the required end game conditions
- terminate the game (player loses) when the number of days left reaches either 0 or the maximum (depending on how you decide to track this data)
- provide all players' details at the end of the game (only the totals for each player required)
- a player should be able to QUIT the game at any time without ending the game for other players (unless they are the last or only player)

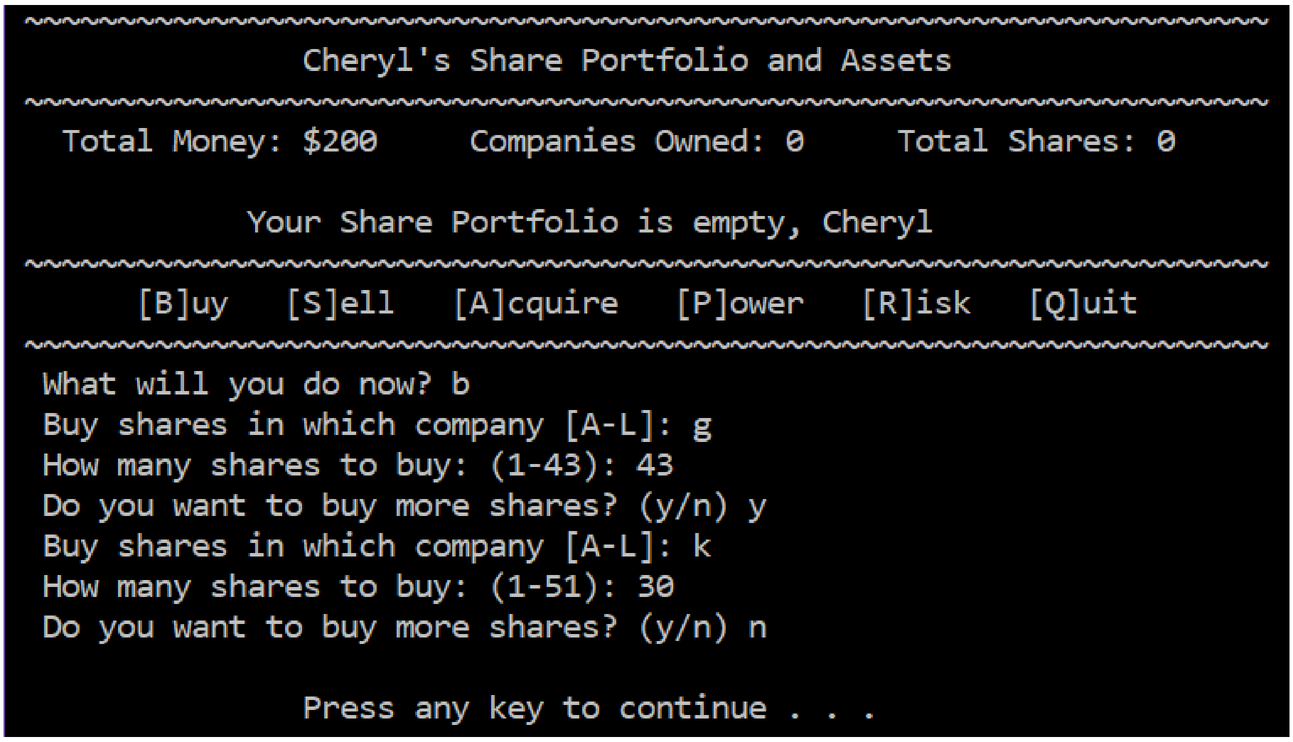
✓ ***The Required Game Interface***

For ease of assessment you must use the required interface (shown in full on page 3 and in part in later screenshots). The screenshot shows the randomised data as required by the game parameters so your screens may look a little different. The main gam screen should also clear whenever the player ends their turn, so players do not have to scroll through the display.

✓ ***Game Interactions***

The actual Player (Tutor) must be able to do the following actions. The screenshots show you the expected procedures and/or outcomes of the associated action.

- **[B]uy Shares:** If any listed company has shares and you have money, you may buy shares in one or more companies. When shares are bought the company's shares are decreased and your money is decreased, while your total shares and shares in that specific company are increased.

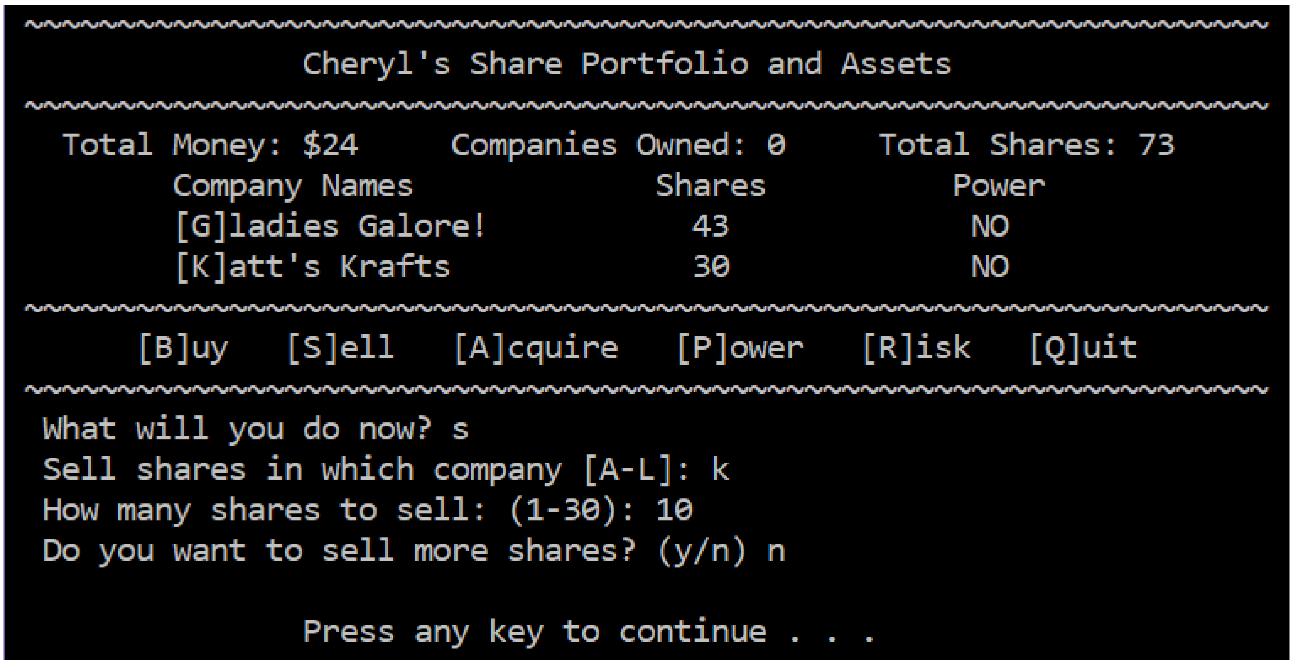


```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                   Cheryl's Share Portfolio and Assets
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  Total Money: $200     Companies Owned: 0     Total Shares: 0

               Your Share Portfolio is empty, Cheryl
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
       [B]uy   [S]ell   [A]cquire   [P]ower   [R]isk   [Q]uit
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 What will you do now? b
 Buy shares in which company [A-L]: g
 How many shares to buy: (1-43): 43
 Do you want to buy more shares? (y/n) y
 Buy shares in which company [A-L]: k
 How many shares to buy: (1-51): 30
 Do you want to buy more shares? (y/n) n

                   Press any key to continue . . .
```

- **[S]ell Shares:** If you have any shares in any number of companies, you may sell shares in one or more companies. When shares are sold the company's shares are increased and your money is increased, while your total shares and shares in that specific company are decreased.

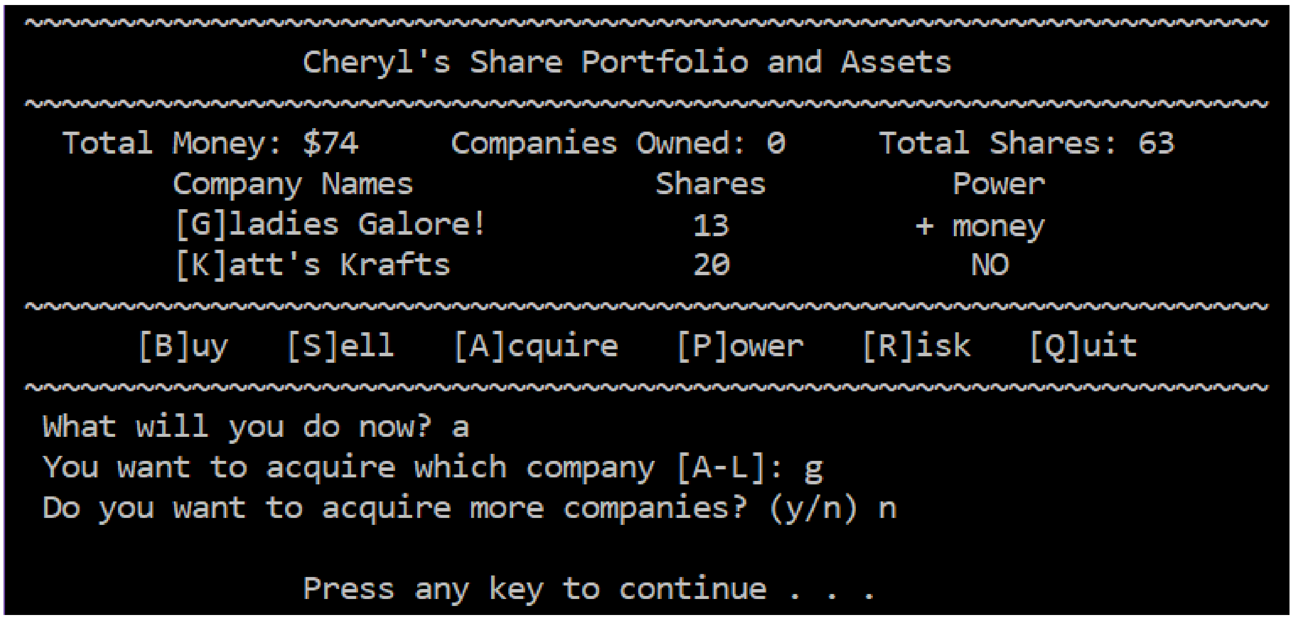


```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
              Cheryl's Share Portfolio and Assets
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  Total Money: $24     Companies Owned: 0     Total Shares: 73
        Company Names             Shares          Power
        [G]ladies Galore!           43             NO
        [K]att's Krafts             30             NO
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
      [B]uy   [S]ell   [A]cquire   [P]ower   [R]isk   [Q]uit
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 What will you do now? s
 Sell shares in which company [A-L]: k
 How many shares to sell: (1-30): 10
 Do you want to sell more shares? (y/n) n

              Press any key to continue . . .
```

- **[A]cquire a Company:** You do not have to own all the shares in a company in order to own it. However, you do need to have shares ≥ the company cost. Once you own a company you can never lose it, even if you don't own any shares in it.

  Your shares in the acquired company are decreased, but your portfolio should now include and list the Corporate Power that is associated with the company you just acquired. You also gain a bonus Corporate Power use when acquiring a company.



```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
              Cheryl's Share Portfolio and Assets
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  Total Money: $74     Companies Owned: 0     Total Shares: 63
        Company Names             Shares          Power
        [G]ladies Galore!           13           + money
        [K]att's Krafts             20             NO
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
      [B]uy   [S]ell   [A]cquire   [P]ower   [R]isk   [Q]uit
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 What will you do now? a
 You want to acquire which company [A-L]: g
 Do you want to acquire more companies? (y/n) n

              Press any key to continue . . .
```

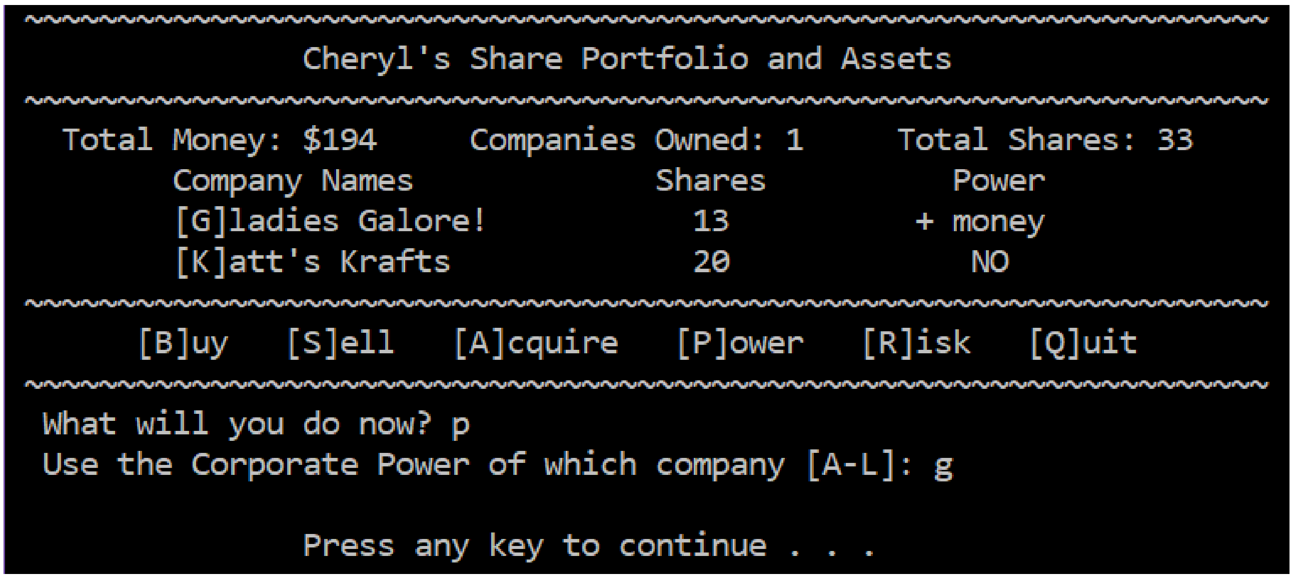
- **Use a Corporate [P]ower:** Allows you to use a Corporate Power associated with a company you own. However, you have a limited number you can use, set by the difficulty level (initial values are set as easy=6, tricky=5 and hard=4), so use them wisely.

  When using a Corporate power you must select a company you own for that company's Power to be activated, and your number of uses is reduced by 1.

  Each type of company has a different power as follows:

  - A randomised multiplier is applied to all powers: rand() % mode + 2
  - **+ money:** adds (10 * (mode * multiplier)) to your money

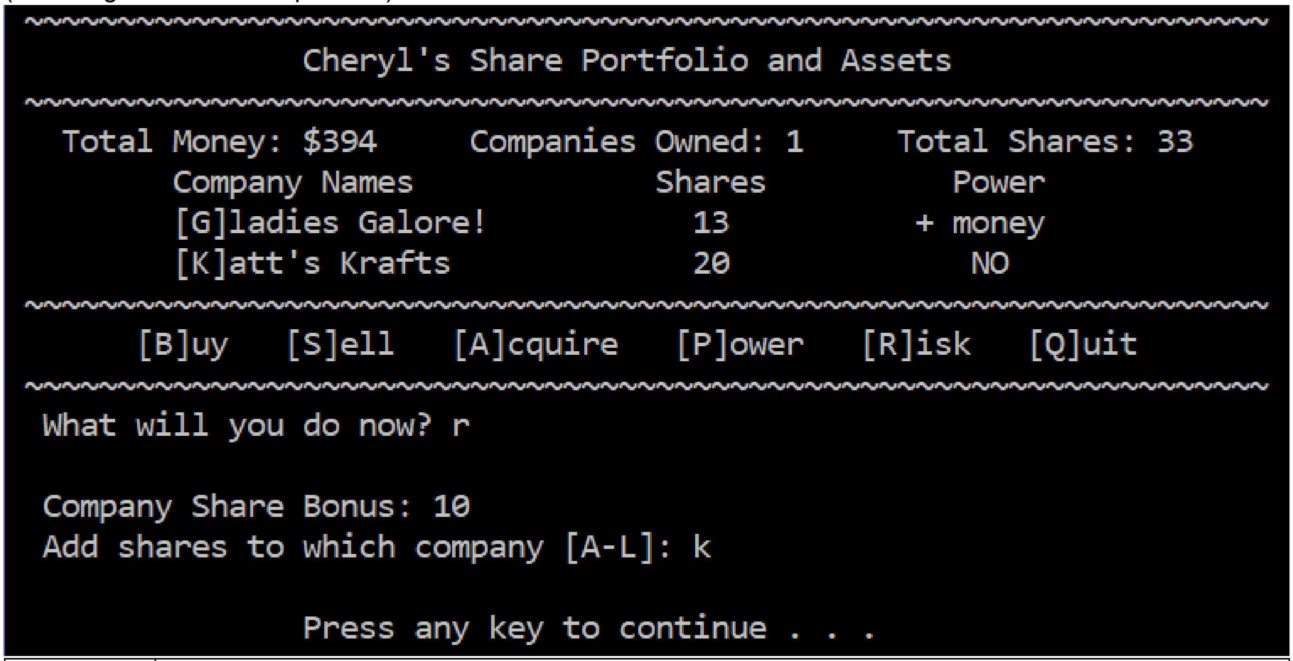
- **+ shares:** adds (mode * multiplier) to your selected company's shares
- **+ assets:** add the above to both money and shares using same formulae
- Finally -1 from your powers uses left



```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
              Cheryl's Share Portfolio and Assets
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  Total Money: $194     Companies Owned: 1     Total Shares: 33
        Company Names              Shares          Power
        [G]ladies Galore!            13          + money
        [K]att's Krafts              20             NO
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
      [B]uy   [S]ell   [A]cquire   [P]ower   [R]isk   [Q]uit
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 What will you do now? p
 Use the Corporate Power of which company [A-L]: g

              Press any key to continue . . .
```

- **Take a [R]isk (Advanced):** Allows you to gamble on getting a good result from one of 20 different events (25% negative and 75% positive).



```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
              Cheryl's Share Portfolio and Assets
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  Total Money: $394     Companies Owned: 1     Total Shares: 33
        Company Names              Shares          Power
        [G]ladies Galore!            13          + money
        [K]att's Krafts              20             NO
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
      [B]uy   [S]ell   [A]cquire   [P]ower   [R]isk   [Q]uit
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 What will you do now? r

 Company Share Bonus: 10
 Add shares to which company [A-L]: k

              Press any key to continue . . .
```

Effect	**Calculations**
money	- (min < 0): maxValue * gameMode for total money to add to player's money. + (max > 0): gameMode * (random value from min to max) to add to player's money. - (max > 0): gameMode * (random value from min to max) to minus from player's money.
shares	Player can select which company to +/- shares (see above). + (max > 0): (random value from min to max) to add to player's shares. - (max > 0): (random value from min to max) to minus from player's shares.
assets	Money and Share calculations are as described above. The outcome of both calculations is displayed to the player.

- **[Q]uit:** The player should also be able to quit the game at any time (during or after a game). When there are 2 or more players, only the quitting player is removed from the game, with any company shares and ownership being returned to the appropriate company before the player is removed. Other players can continue to play until there is a winner or all players have quit.

✓ ***Processing Player Input***

- The letters in [?] are the shortcut keys that must be used to execute the desired command.
- All player input must be processed, using basic validation to ensure that the programme does not crash if an invalid value is entered.
- Appropriate function calls are used to minimise clutter in the main game loop – other than validation checks all processing of player actions and game processes must be done in appropriately named functions (eg: displayMarket(), buyShares(), etc.)
- Appropriate feedback is provided to the player giving correct prompt and/or error messages to enhance the player experience and reduce player frustration.
- An appropriate game loop is established to run the simulation in a single function until the game is won. Recursion is NOT permitted.

Note: using a menu system like the one in Assignment 1 is NOT permitted.

Extra Functionality

The marking criteria indicates that you should make some individual additions to this project in order to achieve the final 20% of your mark.

Following is a list of additional features you can include, with the maximum number of marks [x] you can earn for each one. You may implement one or more features from the list, but you will only score up to the maximum of 20% of the total assignment marks or 20 marks.

You should aim to add some of these additional creative elements to the gameplay, to enhance the presentation of your project.

If you choose not to implement any advanced features you will forfeit these 20 marks.

- **[3]** The default number of players is one. Adding this feature allows for multiple players from 1 to 4 to play in the simulation. You must include adding all players to the game as individual Player Objects and store them in a collection variable at the start of the game.

  In the main game loop all players must have a turn during a single day – all players can buy or sell shares, etc. using the same share prices. When all players have had a turn, the share market changes and the number of days remaining is updated.

- **[3]** The default level of difficulty is Easy mode. Adding this feature allows the player to select a difficulty level (Easy, Tricky and Hard) which modifies the game parameters as follows:

Difficulty Level	Game Mode Value	Maximum Companies	Companies Owned to win	Minimum Money to win	Maximum Days to Play	Corporate Power Uses
Easy	4	1-12	3	500	40	6
Tricky	5	1-15	4	600	50	5
Hard	6	all	5	700	60	4

- **[4]** Saving Game Progress: Adding this feature allows the game to be saved and restored at the player's request. You must add a Sa[V]e option for players to select this action.

  All the relevant game data must be stored in a text file including the current state of the game and all individual player data. When reading the data in it must be assigned to the appropriate game variables and Player Objects.

- **[6]** Company Takeovers: Adding this feature forces a player to own all the shares in a company before they can [A]cquire it. You must add a [M]erge option that works as follows:
  - Ask the player which company they want to merge.
  - Ask the player which opponent player they want to merge with – must check that both players have shares in the selected company.
  - The player initiating the merge must pay the opponent the current value of the shares – check the player can afford the merger.
  - If all checks are good the complete the transactions – the player gains the shares, and the opponent gains the money.

- **[10]** Advanced Risks: Adding this feature allows for a greater range of "risks" the player may experience when taking this action. In this version the processing of a random "risk event" is controlled from the application (not in the class).

- You must store each event, as read from the risksAdvanced.txt file, as individual Risk objects in a collection variable and shuffle them to randomise the order every game. This is the only time the data is modified.
- You must randomly select one Risk object from the collection variable when processing a [R]isk action.
- You may only access the Risk data through appropriate accessor functions as you calculate the results and display the outcome to the player from the main application.
- All appropriate updates to the player's data must also be correctly applied.

Processing risks using this version requires careful thought for executing things in the correct order to ensure you streamline your code.

You certainly do not have to implement all of the above to earn marks for extra functionality. Just remember the maximum number of marks you can earn are given in [x] and this is based on how well you implement the given task.

Project Development Reflection Document

You must also provide a 500-1000 word written reflection of your object-oriented design and project development process and how well you believe it was implemented.

- Discuss why you designed it the way you did.
  - Why did you create your classes that way?
  - What aspects of the game design were easier or harder to do. Why?
  - How does this reflect an OO design or approach?
- Discuss how well you were able to code it.
  - Highlight any issues you found once you tried to implement your design.
  - What elements of the game were easier or harder to code. Why?
  - How did you resolve these issues?
- If you were to do this project again:
  - Discuss how you might change your design to make your solution easier to implement, more efficient, or improve for better for code reuse.
  - Describe what things you have learned during the development of your project.

Assignment 3: Marking Criteria [up to 100 marks in total]

NOTE: *Your submitted project must be correctly identified and submitted, otherwise it will receive an automatic* ***10% penalty*** *(applied after marking).*

Does the program compile and run? Yes or No

NOTE: *Your submitted program MUST compile and run. Submissions that do not compile will receive* ***ZERO****.*
Ensure you leave time to test your work thoroughly before you submit it.

Derived Class Design [15]

- Company and Derived Classes (Bronze, Silver and Gold Companies)
  - All classes have an appropriate header files [3]
  - Base class modified to suit requirements for inheritance and polymorphism [2]
  - Required data members and member functions of each class use meaningful names [1]
  - Appropriate application of virtual functionality in each inheriting class [3]
  - The appropriate implementation of pointers so that polymorphic functions work correctly [4]
  - Contains only aspects that relate to the specific inheriting class (has no data members or member functions that are not directly related to that class) [2]

Game Functionality [35]

- Displays the "how to play" information at the start as read from the text file. [1]
- Initialises the player and game variables appropriately [4]
- Creates the companies and risks using data read from text files [4]
- Implementation of the required User Interface display [4]
- Successful implementation of the Player class functionality [5]
- Successful implementation of the Company class functionality [5]
- Successful implementation of the Risk class functionality [5]
- Successful implementation of action processes and feedback displayed to the player [5]
- Appropriate end game conditions triggered (both win and lose) [2]

Evidence of Functionality Testing [10]

- All the available options tested with invalid options handled appropriately [4]
- Game sequence works as described in the brief with both win and lose results included [4]
- All possible end game scenarios are included and are selected when required [2]

Extra Functionality [20]

- Implement multiple players and modifying the main game loop appropriately [3]
- Implement difficulty levels which may be selected by the player and modifying all required game parameters to suit the selected level [3]
- Allow the game to be saved and restored at the player's request [4]
- Adding a [M]erge option to allow players to do Company Takeovers [6]
- Implementing the advanced version of the Risk events [10]

Quality of Solution and Code [10]

- Does the program perform the functionality in an efficient and extensible manner? [4]
  - Code Architecture – algorithms, data types, control structures and use of libraries [3]
  - Coding Style – Clear logic, clarity of variable names, and readability [1]
- Has a well-designed program been implemented? [2]
  - Applied good programming principles and practices [1]
  - Demonstrates the use of logical procedures [1]
- Has the Programming Style Guide been followed appropriately? [4]
  - Appropriate commenting and code documentation [3]
  - Correct formatting of code within *.h and *.cpp files [1]

Project Development Reflection Document [10]

- Discussion of motivations for the program design [3]
- Discussion of how well the design was to implement [3]
- Discussion of what you would do differently if they were to start it again [4]