Programming Tasks

These questions require you to load the **Skeleton Program** and to make programming changes to it.

Note that any alternative or additional code changes that you deemed appropriate to make must also be evidenced – ensuring that it is clear where in the Skeleton Program those changes have been made.

Task 1 — Marks: 2

This question refers to the **Dastan** class.

Introduce new functionality at the point at which both players are instantiated that allows players to have custom names set by the users. Ensure that players cannot both have the same name. This code will replace the two lines in the constructor that currently creates the players with a single call to a new private method, **CreateCustomPlayers**.

What you need to do

Task 1

Create a new method **CreateCustomPlayers** in the **Dastan** class. Allow the user to enter custom names for each player. Include checks in your code to ensure that two players cannot have the same custom name.

Allow the first player to enter any name they like, then repeatedly ask the user for the second player name until they are both different.

Task 2

Test that the change you have made work:

- run the skeleton program
- enter “Tom” as the first player name and then enter “Tom” as the second player name, when reprompted, enter “Tom” again and then at the next prompt, enter “Victoria”.
- show the game using one of the custom names to address the player in the main game menu.

Evidence that you need to provide:

- PROGRAM SOURCE CODE showing creation of a new **CreateCustomPlayers** method in the **Dastan** class
- SCREEN CAPTURE(S) showing the required test

Task 2 **Marks:** 4

This question refers to the **CreateMoveOptionOffer**, **CreateMoveOption** and **CreateMoveOptions** methods and creation of a new method **CreateFarisMoveOption** in the **Dastan** class.

Develop a new move option called a “Faris” (Knight). The Faris move option moves similarly to a knight in chess – either two squares forward / backwards and one square left / right or oppositely two squares left / right and one square forward / backwards. You should demonstrate the use of the Direction parameter.

What you need to do

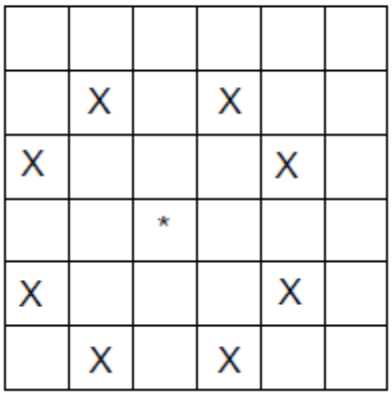
Task 1

i) Add new functionality into the **CreateMoveOptionOffer** & **CreateMoveOption** methods to perform a Faris move.

ii) Modify the **CreateMoveOptions** method to add the faris after the ryott for both players.

iii) Create a new method **CreateFarisMoveOption** which adds moves using the pattern below, to the **NewMoveOption** object.

Faris



Task 2

Test that the change you have made work:

- run the skeleton program
- play two turns, showing both players making legal Faris moves.

Evidence that you need to provide:

- PROGRAM SOURCE CODE showing changes made to the **CreateNewOptionOffer**, **CreateMoveOption** and **CreateMoveOptions** methods
- PROGRAM SOURCE CODE showing a new method **CreateFarisMoveOption**
- SCREEN CAPTURE(S) showing the required test

Task 3 **Marks:** 4

Develop a new move option called a “Sarukh” (Rocket). The Sarukh move option moves forward in a rocket shape. You should demonstrate the use of the Direction parameter.

What you need to do

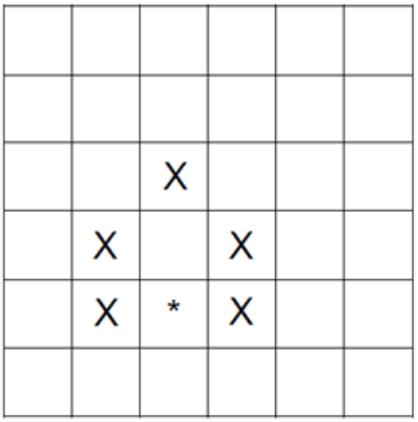
Task 1

i) Add new functionality into the **CreateMoveOptionOffer**, **CreateMoveOption** and **CreateMoveOptions** methods to perform a Sarukh move.

ii) Modify the **CreateMoveOptions** method to add the sarukh after the ryott for both players.

iii) Create a new method **CreateSarukhMoveOption** which adds moves using the pattern below, to the new **MoveOption** object. The pattern is shown from the viewpoint of player 2. For player 1, the layout is inverted.

Sarukh



Task 2

Test that the change you have made work:

- run the skeleton program
- play two turns, showing both players making legal Sarukh moves.

Evidence that you need to provide:

- PROGRAM SOURCE CODE showing changes made to the **CreateMoveOptionOffer**, **CreateMoveOption** and **CreateMoveOptions** methods
- PROGRAM SOURCE CODE showing a new method **CreateSarukhMoveOption**
- SCREEN CAPTURE(S) showing the required test

Task 4 **Marks:** 5

This question refers to the **PlayGame** method of the **Dastan** class and creation of a new method **AwardLuckyStar** in the **Dastan** class, **GetLuckyStarAwarded**, **SetLuckyStarAwarded** and **SetLuckyStarUsed** together with two new attributes **LuckyStarUsed** and **LuckyStarAwarded** in the **Player** class.

Create a "Lucky Star" award which can be applied to either player once per game. The "Lucky Star" has a 25% chance of being awarded to a player on their turn. On receipt of the "Lucky Star", the player has the option of ANY move from their move queue rather than just being able to select from the first 3 items. The "Lucky Star" award removes the move cost for the move the player selects for that turn.

Note: If the player makes an invalid move then they "lose" their lucky star and get no value from it, also the player should not be able to "take the offer" if a lucky star is awarded.

What you need to do

Task 1

i) Create a new method in the **Dastan** class called **AwardLuckyStar**. This method should have a 25% chance of returning true.

ii) Add a new private attribute to the **Player** class called **LuckyStarAwarded**. Include accessor and mutator (getter/setter) methods for this attribute.

Task 2

Update the **PlayGame** method in the **Dastan** class to call the new **AwardLuckyStar** method. If the player hasn't already been awarded a lucky star, print out a message saying "You have been awarded a lucky star, you can select any move from your queue for free this turn." Adjust the input range to allow any move option in the queue to be selected. Ensure that there is no score adjustment for playing a move and update the value of the attribute to ensure that they cannot receive another Lucky Star.

Task 3

Test that the change you have made work:

- run the skeleton program
- play the game to show a player being awarded a Lucky Star
- play a move option from position 4 or 5 in the move option queue.
- show the updated board and correctly modified score.

Evidence that you need to provide:

- PROGRAM SOURCE CODE showing changes made to the **PlayGame** method of the **Dastan** class, creation of a new method **AwardLuckyStar** in the **Dastan** class.
- PROGRAM SOURCE CODE showing changes made to the **Player** class and creation of the new methods **GetLuckyStarAwarded**, **SetLuckyStarAwarded** together with one new attribute **LuckyStarAwarded**.
- SCREEN CAPTURE(S) showing the required test

Task 5 **Marks:** 5

This question refers to the **PlayGame** and method of the **Dastan** class and the creation of a new method **GetJustQueue** in the **Player** class.

Introduce a new option 8 to the main game playing menu. On selecting this option, a player can look at their opponent's queue to spy what move options their opponent might be considering next. Spying on an opponent's queue, however, carries a cost of 5 points from the player's score. After spying on an opponent's queue, the player's turn should continue as normal.

What you need to do

Task 1

Create a new method in the **Player** class called **GetJustQueue** which uses the **GetQueueAsString** method to return a string version of just the player's queue.

Task 2

Modify the **PlayGame** method to introduce new functionality which adds a new option 8 to the main game playing menu. If the user selects this option display the move option queue for the opposing player (**Hint:** You can check the current player using the **SameAs** method and then pick the other player). Subtract 5 from the current player score and display the game state again allowing the player to continue their turn as normal.

Task 3

Test that the change you have made work:

- run the skeleton program
- show player one selecting option 8 from the main game menu.
- show the opponent queue being displayed clearly on the screen and the player one score reducing by 5 points.

Evidence that you need to provide:

- PROGRAM SOURCE CODE showing changes made to the **PlayGame** method and of the **Dastan** class.
- PROGRAM SOURCE CODE showing new method **GetJustQueue** in the **Player** class.
- SCREEN CAPTURE(S) showing the required test

Task 6 **Marks:** 6

This question refers to the **PlayGame** method together with the modification of **GetSquareReference**, **UseMoveOptionOffer** methods and creation of a new method **GetValidInt** in the **Dastan** class.

Currently the game has a number of areas where it does not handle erroneous user input. Introduce error handling into the **PlayGame**, **GetSquareReference** and **UserMoveOptionOffer** methods to prevent unhandled exceptions from occurring if the user inputs data in an incorrect data type. Allow the user to re-enter their input, until it is valid.

Note: There is no need to check that the square contains a player piece or that the move is valid, the player should still have a wasted turn if the move is invalid, the purpose of this is to stop the program from crashing.

What you need to do

Task 1

Create a new private method called **GetValidInt** in the **Dastan** class which checks if the user input is a valid integer. If the input is invalid, allow the user to keep trying again without penalty.

Task 2

Modify the **GetSquareReference** method to use the new **GetValidInt** method to test for erroneous user input. Add an error message if they enter an invalid square.

Task 3

Modify the **UseMoveOptionOffer** method to use the new **GetValidInt** method to test for erroneous user input and test to confirm that the user input is within the correct range.

Task 4

Test that the change you have made work:

- run the skeleton program
- from the main game playing menu, enter “help” as your choice and show a suitable error message. Then choose move 1.
- Enter a square of 19 and show the error message. Then choose square 22.
- select option 9 to take the offer move and choose position 8. Show the error message.

Evidence that you need to provide:

- PROGRAM SOURCE CODE showing changes made to the **GetSquareReference** method
- PROGRAM SOURCE CODE showing changes made to the **PlayGame** method
- PROGRAM SOURCE CODE showing changes made to the **UseMoveOptionOffer** method
- PROGRAM SOURCE CODE showing the creation of new method **GetValidInt** method
- SCREEN CAPTURE(S) showing the required test.

Task 7 **Marks:** 5

This question refers to the **PlayGame** and **UseMoveOptionOffer** methods in the **Dastan** class and the creation of a new attribute **ChoiceOptionsLeft** along with accessor and mutator (getter/setter) methods **DecreaseChoiceOptionsLeft** and **GetChoiceOptionsLeft** in the **Player** class.

Currently a player can repeatedly select option 9 from the main game playing menu, filling their queue with new move options. Introduce a limit so that a player can only "accept the offer" from the Move Option menu 3 times in a game. Each time a player accepts the offer, advise them of how many selections they have left and remove the menu for that player once they have used it 3 times.

What you need to do

Task 1

Modify the **Player** class to introduce a new private attribute called **ChoiceOptionsLeft.**

i) Initialize **ChoiceOptionsLeft** to 3.

ii) Create a new accessor method called **GetChoiceOptionsLeft** which returns the value of the attribute **ChoiceOptionsLeft**.

iii) Create a new mutator method called **DecreaseChoiceOptionsLeft** which decrements the **ChoiceOptionsLeft** attribute and prints out how many options you have left.

Task 2

Modify the **PlayGame** method to test the number of options choices the player has left so that they can only use 3 during the game.

i) Modify the **PlayGame** method so that if the player has used up all their option choices, option 9 will no longer be available to the player.

ii) Modify the **UseMoveOptionOffer** method so that when a move option is selected by the player, the number of options available to them decreases by one.

Task 3

Test that the change you have made work:

- run the skeleton program
- select 4 sequential option moves from the move option list adding them to positions 1 to 4 in the player one queue.
- show the removal of option 9 from the main game playing menu and show that it does nothing if the player attempts to select option 9.

Evidence that you need to provide:

- PROGRAM SOURCE CODE showing changes made to the **PlayGame** method
- PROGRAM SOURCE CODE showing changes made to the **UseMoveOptionOffer** method in the **Dastan** class
- PROGRAM SOURCE CODE showing changes made to the **Player** class
- SCREEN CAPTURE(S) showing the required test

Task 8 **Marks:** 5

This question refers to the **PlayGame** method of the **Dastan** class and creation of new methods **ResetQueueBack** in the **MoveOptionQueue** class and **ResetQueueBackAfterUndo** in the **Player** class.

Introduce a new option that allows a player to undo their last move, undoing any score gained or lost in that move and returning the game to its previous state. Undoing a move costs a player 5 points. After undoing a move, a player can then make an alternative move.

What you need to do

Task 1

Add the functionality to reset the queue if a move is undone.

i) Create a new method in the **MoveOptionQueue** class called **ResetQueueBack**. This method should move the last element of the queue back to the original position in the queue. The method should take one parameter, **Position** which is the place to which the last element of the queue will be restored.

ii) Create a new method in the player class called **ResetQueueBackAfterUndo**. This method should call the newly created **ResetQueueBack** method on the **Queue** attribute in the **Player** class. The method should take one parameter, **Position** which is the Choice that the player made from the menu.

Task 2

Modify the **PlayGame** method to introduce the new functionality.

i) If a move is legal, store the player score prior to the move.
ii) After displaying the board as a result of the move, give the player the option to undo it.
iii) If they choose to undo then: return the player score to the stored pre-move score, subtract 5 points and restore the board and player's queue back to their pre-move states.

Task 3

Test that the change you have made work:

- run the skeleton program
- show player one attempt a "chowkidar" move and then undo the move and play a "ryott"
- show the game board after the undo and the score set correctly and that player one can choose a new move

Evidence that you need to provide:

- PROGRAM SOURCE CODE showing changes made to the **PlayGame** method of the **Dastan** class
- PROGRAM SOURCE CODE showing the creation of new methods **ResetQueueBack** in the **MoveOptionQueue** class
- PROGRAM SOURCE CODE showing the creation of the new method **ResetQueueBackAfterUndo** in the **Player** class.
- SCREEN CAPTURE(S) showing the required test

Task 9 **Marks:** 8

This question refers to the **PlayGame** method together with the modification of **CreateMoveOptionOffer** and **CreateMoveOption** methods and creation of three new methods **CreateRaaketMoveOption, ChoiceIsRaaket** and **CalculateRaaketMove** in the **Dastan** class. It also refers to a new attribute **RaaketUsed** in the **Player** class along with Create two new methods, **GetRaaketStatus** and **SetRaaketUsed** which operate as the accessor and mutator (getter/setter) methods for the new created **RaaketUsed** attribute.

Develop a new "Raaket" move option (Rocket). The Raaket can only be fired once in a game per player and is fired instead of a piece moving. A Raaket can be fired by any piece. The Raaket fires in a straight line forwards from the player destroying any opponent piece(s) in its way except a Kotla which is strong enough to withstand an attack and protect any piece inside it. The Raaket is only made available to a player through the **MoveOptionOffer**.

What you need to do

Task 1

Add new functionality into the **CreateMoveOptionOffer**, and **CreateMoveOption** methods and create a new private **CreateRaaketMoveOption** method to perform a Raaket move

i) Modify the **CreateMoveOptionOffer** method to offer the new "raaket" move first.

ii) Create the new private **CreateRaaketMoveOption** method to allow the player to select which piece fires the raaket.
Note: The move should not actually move the piece anywhere, i.e. 0 rows and columns.

iii) Modify the **CreateMoveOption** method to handle Raaket.

Task 2

Modify the Player class to allow the user to use their Raaket only once.

i) Add a new **RaaketUsed** attribute in the **Player** class which is initialized to False.

ii) Create two new methods, **GetRaaketStatus** and **SetRaaketUsed** which operate as the accessor and mutator (getter/setter) methods for the new created **RaaketUsed** attribute.

iii) Modify the **CheckPlayerMove** method so that it doesn't check the **FinishSquareReference** for a raaket move.

iv) Create a method **ChoiceIsRaaket** method which takes a parameter and checks if the move option chosen is a raaket move whereupon it returns True.

(TASK CONTINUES ON THE NEXT PAGE)

Task 3

Modify the **PlayGame** method to test to see if the player has selected a Raaket move from the **MoveOptionOffer** menu and if it has already been used. If the selected firing piece is valid, the Raaket should destroy any opponent pieces in a straight line from the firing piece, except a Kotla. The firing player should collect any points from multiple pieces destroyed by the Raaket.

i) Modify **PlayGame** to call the new method **ChoiceIsRaaket** and only asks for the start square if it is.

ii) Create a new private method in the Player class called **CalculateRaaketMove** which will calculate the points for a Raaket move and destroy the pieces that are hit (unless they are in a kotla).

iii) Modify **PlayGame** to so that is calls the new method **CalculateRaaketMove** to get the points for the Raaket move and destroys the relevant pieces. It should also call the **SetRaaketUsed** method for the current player.

Task 4

Test that the change you have made work:

- run the skeleton program
- select a chowkidar move for player one (option 2) and choose square 22 as the "from" and square 33 as the "to" to diagonally move one piece in front of another player one piece in the kotla column.
- select 9 from menu for player two to accept the offer. Choose 1 to put it in position 1 and then choose option 1 to select the raaket move. Choose the piece on square 53 to fire the Raaket and show the updated board with both player one pieces removed from the board by the Raaket fired by player two but not the mirza which is safely inside the kotla.
- show the correct adjustment of player two's score.

Evidence that you need to provide:

- PROGRAM SOURCE CODE showing changes made to the **PlayGame** method
- PROGRAM SOURCE CODE showing changes made to the **CreateMoveOptionOffer** and **CreateMoveOption** methods
- PROGRAM SOURCE CODE showing the creation of new **CreateRaaketMoveOption, ChoiceIsRaaket** and **CalculateRaaketMove** methods
- PROGRAM SOURCE CODE showing changes made to the **Player** class
- SCREEN CAPTURE(S) showing the required test

Task 10 **Marks:** 4

This question refers to the **PlayGame** method in the **Dastan** class.

Introduce a new option 7 to the main game playing menu. On selecting this option, a player can select one of their own pieces to destroy and replace with a second Kotla. A new Kotla can only be placed in the square that the piece was sacrificed. A player can only replace one of their own pieces. Replacing a piece with a Kotla should use up a player turn and they should not score any points for that turn.

What you need to do

Task 1

Modify the **PlayGame** method in the **Dastan** class to introduce a new option 7 into the main game playing menu. Allow the player to select a piece which they would like to replace with a new Kotla. Use validation to ensure that the user can only select one of their pieces and it cannot be the Kotla. On confirmation, replace the piece with a second Kotla assigned to the correct team.

Task 2

Test that the change you have made work:

- run the skeleton program
- select option 7 for player one from the main game menu
- show the user selecting 52 as an invalid square for the new Kotla.
- show the Kotla being placed correctly square 22, a valid square and assigned to player one.

Evidence that you need to provide:

- PROGRAM SOURCE CODE showing changes made to the **PlayGame** method
- SCREEN CAPTURE(S) showing the required test

Task 11 **Marks:** 9

This question refers to the **PlayGame** method together with a new method called **ModifyQueueOptions** in the **Dastan** class, additional new methods **ReverseQueue**, **SwapFirstAndLast** and **MoveItemToFront** in the **MoveOptionQueue** class together with new methods **GetQueueAsString**, **SwapQueue**, **GetMoveOptionQueue**, **ReversePlayerQueue**, **SwapFirstAndLast** and **MoveItemToFront** in the **Player** class.

Introduce a new option 6 to the main game playing menu. On selecting this option, a player can choose sub options for making changes to their move queue using the following menu:

Option
a) Reverse the current player queue
b) Swap the current player queue with the opponent queue
c) Swap the first and last elements in the current player queue
d) Move one of the move options to the front of the current player queue
e) Nothing (make normal move)

Note: Options (a) – (d) cost 3 points, but they can choose (e) for free.
Note: This does not count as the player's turn and they should still be able to play a move.

What you need to do

Task 1

Modify the **Dastan** class to introduce the new menu option.

i) Modity the **PlayGame** method to add option 6 to the move options menu.

ii) Create a new private method in the **Dastan** class called **ModifyQueueOptions** which gives the player the above menu. Include validation to ensure that the user can only enter one of the option choices from the menu.

iii) Adjust the score by 3 if options (a) – (d) are chosen but not if option (e) is.

Task 2

Modify the **MoveOptionQueue** class to add the required methods.

i) Create new method **ReverseQueue** to allow the current player's queue to be reversed.

ii) Create new method **SwapFirstAndLast** to swap the first and last elements of the current player's queue.

iii) Create new method **MoveItemtoFront** to move the item from the chosen position to the start of the queue for the current player. There is no need to validate the input for the position to move the option from.

Task 3

Modify the **Player** class to create the required methods.

i) Create new methods **ReverseQueue**, **SwapFirstAndLast**, **MoveItemtoFront** in the **Player** class to expose the new **MoveQueueOptions** choices/methods to the **Dastan** class.

ii) Create new method **ReplaceQueue** to allow the current player's queue to be replaced with the queue passed in as a parameter. Note that is should return the current queue.

Task 4

Test that the change you have made work:

- run the skeleton program
- show player one selecting option 6 from the main game menu.
- show the player selecting each one of the queue options in turn and the updated queue on the screen as a result of the change

Evidence that you need to provide:

- PROGRAM SOURCE CODE showing changes made to the **PlayGame** method
- PROGRAM SOURCE CODE for the new **ModifyQueueOptions** method in the **Dastan** class
- PROGRAM SOURCE CODE showing changes made to the **MoveOptionsQueue** class
- PROGRAM SOURCE CODE showing changes made to the **Player** class
- SCREEN CAPTURE(S) showing the required test

Task 12 **Marks:** 7

This question refers to the creation of a new protected attribute **NoOfPieces**, modification of the existing **PlayGame** method and creation of two new methods **CheckReincarnation** and **CountNormalPieces** in the **Dastan** class.

Introduce a new feature whereby if a player manages to get one of their players to the opponent's back row, they are given the opponent to gain a new player piece and place it on any unoccupied space on their own back row. Note that the player cannot reincarnate pieces that are not dead so they should not be able to have more pieces than they started with.

What you need to do

Task 1

Create a new private method in the **Dastan** class called **CountNormalPieces** that will return the number of pieces that the current player has excluding the mirza.

Task 2

i) Modify the constructor of the **Dastan** class to store the number of pieces passed in as a new protected attribute called **NoOfPieces**.

ii) Modify the **PlayGame** method of the **Dastan** class to call a new private method **CheckReincarnation** after the move is legal.

Task 3

Create a new private method **CheckReincarnation** in the **Dastan** class. This should take one parameter which is the **FinishSquareReference** for the current player's move. If the player's move ended on the opponents back row (e.g. row 6 for player one) and the player has fewer pieces than they started with then allow them to reincarnate a piece on their back row in an empty square. You need to validate that the square is empty and allow them to reselect if it is not.

Task 4

Test that the change you have made work:

- add the following four lines of code to the START of the private method **CreatePieces** in the Dastan class:

```
NoOfPieces = 2
self._Board[self.__GetIndexOfSquare(51)].SetPiece(Piece("piece", self._Players[0], 1, "!"))
self._Board[self.__GetIndexOfSquare(21)].SetPiece(Piece("piece", self._Players[1], 1, '"'))
self._Board[self.__GetIndexOfSquare(54)].SetPiece(Piece("piece", self._Players[1], 1, '"'))
```

- run the skeleton program
- select a ryott move for player one, enter a start square of 51 and an end square of 61
- show player one attempting to reincarnate a piece in column 3 and being given an error message saying that the square must be empty
- show player one attempting to reincarnate a piece in column 4 and the board being updated appropriately
- Select a ryott move for player two, enter a start square of 21 and an end square of 11
- show player two not receiving a reincarnation message

Evidence that you need to provide:

- PROGRAM SOURCE CODE showing changes made to the Dastan class
- SCREEN CAPTURE(S) showing the required test

Task 13 **Marks:** 8

This question refers to the **PlayGame** method together with modification of the **CreateBoard** method in the **Dastan** class. Additionally it involves the creation of a new method **GetCampedTwoTurns** in the **Square** class and the creation of a new **AdvantageCastle** class which inherits from **Square**.

Create a new type of game square the Advantage Castle (similar to the kotla) which is placed in the middle of the playing board (or slightly closer to player two if there are an even number of rows). Either player can occupy the Advantage Castle with any of their pieces. If a player can occupy the Advantage Castle for 2 turns by both players (entering the castle is considered a player's first turn), then their next move choice will have zero cost. This gives a player zero cost move, but risks sitting in the middle of the playing board to get it.

What you need to do

Task 1

Create a new Class **AdvantageCastle** which should inherit from the **Square** class.

i) Add a new protected attribute **CampedTurns** and initialise it to 0.

ii) Override the **SetPiece** and **RemovePiece** methods from the **Square** class. **SetPiece** should adjust the **AdvantageCastle** symbol to an upper case "A" if player one owns the Advantage Castle and a lower case "a" if player two owns the castle (you may assume that the player with a **Direction** of 1 is the player at the top - player one). When a player piece leaves the castle, ownership of the square should be set to null and the symbol set to a lower case "x".

iii) Override the new method **GetCampedTwoTurns** from the **Square** class. Each time the castle is captured by a new player **CampedTurns** should be reset back to zero. The **GetCampedTwoTurns** method shouldcheck the number of turns using the **CampedTurns** attribute and return true if it is >= 2.

iv) Create a new method **CheckCamp** that checks if the same player is still in the square and increments the **CampedTurns** attribute if they are.

Task 2

Modify the **CreateBoard** method in the **Dastan** class to place an Advantage Castle on the square closest to the middle of the board with a lower case "x" symbol when the board is first created.

Note that in the case where there are an even number of rows, the castle should be slightly closer to player two, also if there are an even number of columns then it should be slightly closer to the left. In the case of the starting board this will place it on square 43 but it should work for any size board.

The initial Advantage Castle does not belong to either player.

Task 3

Modify the **PlayGame** method so that if a move is legal the game should test to see if the Advantage Castle has been camped in for two full turns and if so, give the selected move to the player at zero cost.

Task 4

Test that the change you have made work:

- run the skeleton program
- use a Cuirassier move option 3 to move a player one piece into the Advantage Castle (from 23 to 43)..
- play the game until both players have had two turns – leaving the player one piece in the Advantage Castle without attacking it using player two.
- after both players have had two turns, show a move option by player one which incurs zero cost.

Evidence that you need to provide:

- PROGRAM SOURCE CODE showing changes made to the **PlayGame** method
- PROGRAM SOURCE CODE showing changes made to the **CreateBoard** method
- PROGRAM SOURCE CODE showing the new **GetCampedTwoTurns** virtual method in the **Square** class
- PROGRAM SOURCE CODE showing the new **AdvantageCastle** class
- SCREEN CAPTURE(S) showing the required test

Task 14 **Marks:** 10

This question refers to the **PlayGame** method together with creation of a new private **WeatherEventOccurs** method in the **Dastan** class. Additionally it involves the creation of a new class **WeatherEvent** with the methods **CountDownComplete**, **SetWeatherLocation** and **GetWeatherLocation**.

The Weather Event has a 50% chance of appearing in any turn and can appear in any unoccupied space on the board. On appearance on the board, both players are given a timer warning that the Weather Event will destroy EVERY piece on the same column as the Weather Event in two turns time. After two turns by each player, the Weather Event strikes and any piece from either player still on that column is destroyed including the Kotla.

What you need to do

Task 1

Create a new class **WeatherEvent** which should include new methods **CountDownComplete**, **SetWeatherLocation** and **GetWeatherLocation**. On instantiation, the Weather Event should store a countdown to count the number of game turns before the event occurs. **CountDownComplete** should test to see if the countdown has expired. The **SetWeatherLocation** and **GetWeatherLocation** methods should get and set the location of the Weather Event on the board.

Task 2

Create a new method called **WeatherEventOccurs** in the **Dastan** class which has a 50% chance of creating a Weather Event square into a random empty square on the board. When a Weather Event has occurred, let the player know.

Task 3

Modify the **PlayGame** method in the **Dastan** class to test to see if a Weather Event has occurred and if so if the Weather Event countdown has expired. If it has, use the Weather Event location to remove any piece (from either player) from the same column as the Weather Event including the Kotla. No points are awarded for this event.

Task 4

Test that the change you have made work:

- run the skeleton program
- when a weather event occurs, move player pieces to be on the same column as the weather event over the next two turns.
- show the board during the countdown to the Weather Event and after the countdown has expired, showing the pieces from both players removed from the board.

Evidence that you need to provide:

- PROGRAM SOURCE CODE showing changes made to the **PlayGame** method
- PROGRAM SOURCE CODE showing the new **WeatherEventOccurs** method
- PROGRAM SOURCE CODE showing the new **WeatherEvent** class
- SCREEN CAPTURE(S) showing the required test

Task 15 **Marks:** 15

This question refers to the **PlayGame** method together with modification of the **CheckSquareIsValid** and **CreatePieces** methods and creation of three new private methods, **CheckBarrierIsValid**, **PlaceBarrier** and **CheckManhattanDistance** in the **Dastan** class. Additionally it involves the creation of new public method **ContainsBarrier** in the **Square** class and the creation of a new **Barrier** class which inherits from **Square**.

Create a new game piece called a barrier. On creation of the board each player can choose where they would like to place their barrier on the board. The barrier is 3 squares wide. This cannot be outside of the board or in a position occupied by a normal piece or an opponent's barrier. The barrier piece cannot be moved, occupied or jumped over by either player.

What you need to do

Task 1

i) Create a new class **Barrier** which should inherit from the **Square** class. A Barrier should be assigned an owner and given the symbol of a capital "B" if it belongs to player one and a lowercase "b" if it belongs to player two.

ii) Create a new public method **ContainsBarrier** in the **Square** class which returns true if a Barrier has been placed in that square

Task 2

i) Modify the **CheckSquareIsValid** method to check if the square being tested contains a Barrier so that a piece cannot occupy it or attempt to move it.

ii) Create a new method **CheckBarrierIsValid** in the **Dastan** class which checks that the location of a Barrier being placed by a player fits within the bounds of the board and only covers empty squares.

iii) Create a new method called **PlaceBarrier** in the **Dastan** class which places a three-square wide barrier onto the board. The barrier will always be horizontal and the player should enter the center square when being asked where to place the barrier.

Task 3

i) Create a new method called **CheckManhattanDistance** in the **Dastan** class which checks both paths from a starting square reference to a finishing square reference by traversing along the starting row then down the finishing column and also down the starting column and along the finishing row. This is used to check if a selected move can traverse around a barrier rather than over the top of it.

ii) Modify **PlayGame** to call **CheckManhattanDistance** which should replace the call to **CheckPlayerMove** used to set the value of the variable **MoveLegal**.

Note: this should be used for all moves even if they are too short to potentially jump a barrier as they may be able to go round. For a single or double move either horizontally or vertically, only one path should be considered, only for diagonal moves should you consider horizontal and then vertical or vertical and then horizontal.

Task 4

Test that the change you have made work:

- run the skeleton program
- enter a position of 34 for the player one barrier
- enter a position of 42 for the player two barrier
- for player 1: choose 9, then 1, then 1, then 24, then 46
- for player 2: choose 3, then 53, then 31
- for player 1: choose 2, then 25, then 45
- for player 2: choose 1, then 52, then 42, then 51

Evidence that you need to provide:

- PROGRAM SOURCE CODE showing changes made to the **PlayGame** method
- PROGRAM SOURCE CODE showing changes made to the **CheckSquareIsValid** and **CreatePieces** methods in the **Dastan** class
- PROGRAM SOURCE CODE for the new private **CheckBarrierIsValid, PlaceBarrier and CheckManhattanDistance** methods in the **Dastan** class
- PROGRAM SOURCE CODE showing changes made to the **Square** class
- PROGRAM SOURCE CODE showing the new **Barrier** class
- SCREEN CAPTURE(S) showing the required test

