

**Programming Tasks (Mark Scheme) Part 2**

# Task 11 (11 marks)

## Coding

* Adding option 6 to the menu which brings up a list of options to modify the queue. **[1 mark]**
* Displaying a suitably formed menu as per the question/example code. [1 mark]
* Validating that the user entered an option from the menu correctly before proceeding. [1 mark]
* Option 1 correctly reverses the player’s queue in a method inside MoveOptionQueue. **[1 mark]**
* Option 2 correctly swaps queue with the opponent without breaking encapsulation by exposing additional attributes that shouldn’t be exposed. **[1 mark]**
* Option 3 correctly swaps the first and last elements of your queue. **[1 mark]**
* Option 4 correctly moves an element to the front of the queue. **[1 mark]**
* Option 5 quits and doesn’t cost any points but the other options all cost 3 points. **[1 mark]**

### Example Solution

Changes to PlayGame:

while (!GameOver)

{

DisplayState();

bool SquareIsValid = false;

int Choice;

do

{

//CHANGE

Console.Write("Choose move option to use from queue (1 to 3) or 9 to take the offer or 6 to modify your queue options: "); //Q11

Choice = Convert.ToInt32(Console.ReadLine());

if (Choice == 6)

{

ModifyQueueOptions();

DisplayState();

}

else if (Choice == 9)

{

UseMoveOptionOffer();

DisplayState();

}

//END CHANGE

}

while (Choice < 1 || Choice > 3);

Code for ModifyQueueOptions:

//CHANGE

private void ModifyQueueOptions() //Q11

{

bool Valid = false;

int UserChoice = 0;

while (!Valid)

{

Console.WriteLine();

Console.WriteLine("Select from the Queue menu choices below.");

Console.WriteLine("1: Reverse your Queue \t\t\t\t\t\t3 points");

Console.WriteLine("2: Swap Queues with your Opponent \t\t\t\t3 points");

Console.WriteLine("3: Swap the first and last items in your queue \t\t\t3 points");

Console.WriteLine("4: Move a MoveOption of your choice to the front of your Queue \t3 points");

Console.WriteLine("5: Nothing, make a normal move \t\t\t\t\t0 points");

Console.WriteLine();

UserChoice = Convert.ToInt32(Console.ReadLine());

if (UserChoice > 0 && UserChoice < 6)

{

Valid = true;

}

}

switch (UserChoice)

{

case 1:

CurrentPlayer.ReversePlayerQueue();

CurrentPlayer.ChangeScore(-3);

break;

case 2:

MoveOptionQueue TempPlayer1Queue = Players[0].GetMoveOptionQueue();

MoveOptionQueue TempPlayer2Queue = Players[1].GetMoveOptionQueue();

Players[0].ReplaceQueue(TempPlayer2Queue);

Players[1].ReplaceQueue(TempPlayer1Queue);

Console.ReadLine();

CurrentPlayer.ChangeScore(-3);

break;

case 3:

CurrentPlayer.SwapFirstAndLast();

CurrentPlayer.ChangeScore(-3);

break;

case 4:

CurrentPlayer.MoveItemToFront();

CurrentPlayer.ChangeScore(-3);

break;

case 5:

Console.WriteLine("No change made - returning to main menu");

break;

}

}

//END CHANGE

Changes to MoveOptionQueue:

//CHANGE

public void ReverseQueue() //Q11

{

Queue.Reverse();

}

public void SwapFirstAndLast() //Q11

{

MoveOption FirstItem = Queue[0];

MoveOption LastItem = Queue[Queue.Count - 1];

Queue.RemoveAt(0);

Queue.Insert(0, LastItem);

Queue.RemoveAt(Queue.Count - 1);

Queue.Add(FirstItem);

}

public void MoveItemToFront(int Position) //Q11

{

MoveOption Temp = Queue[Position];

Queue.RemoveAt(Position);

Queue.Insert(0, Temp);

}

//END CHANGE

Changes to Player:

//CHANGE

public void ReplaceQueue(MoveOptionQueue NewQueue)

{

Queue = new MoveOptionQueue();

Queue = NewQueue;

}

public MoveOptionQueue GetMoveOptionQueue()

{

return Queue;

}

public void ReversePlayerQueue() //Q11

{

Queue.ReverseQueue();

}

public void SwapFirstAndLast() //Q11

{

Queue.SwapFirstAndLast();

}

public void MoveItemToFront() //Q11

{

int SelectedOption = 0;

do

{

Console.WriteLine("Select a MoveOption to move to the front of your queue. Select an option 1 to 5:" + Environment.NewLine);

Console.WriteLine(Queue.GetQueueAsString());

SelectedOption = Convert.ToInt32(Console.ReadLine());

}

while (SelectedOption < 1 || SelectedOption > 5);

Queue.MoveItemToFront(SelectedOption - 1);

}

//END CHANGE

## Testing:

* Showing at least one of options 1-4 working. [**1 mark]**
* Showing the remaining three options working. [1 mark]
* Showing option 5 and the scoring working correctly. [1 mark]

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

# Task 12 (7 marks)

## Coding

* Creating and storing the number of pieces correctly in the new protected attribute NoOfPieces. **[1 mark]**
* Adding a call to CheckReincarnation in the correct place. **[1 mark]**
* Creating CountNormalPieces to correctly return the number of pieces excluding the Mirza for the currently player. **[1 mark]**
* Correctly detecting when a piece reaches the opponent's back row. **[1 mark]**
* Having a condition to only allow reincarnation if the player has fewer pieces than they started with. **[1 mark]**
* Correctly handling the reincarnation on the player’s own back row and checking that the square is empty. **[1 mark]**

### Example Solution

Changes to Dastan constructor:

protected Random RGen = new Random();

//CHANGE

protected int NoOfPieces; //Q12

//END CHANGE

//CHANGE

public Dastan(int R, int C, int NumberOfPieces) //Q12

{

NoOfPieces = NumberOfPieces; //Q12

//END CHANGE

Players.Add(new Player("Player One", 1));

Players.Add(new Player("Player Two", -1));

Changes to PlayGame:

bool MoveLegal = CurrentPlayer.CheckPlayerMove(Choice, StartSquareReference, FinishSquareReference);

if (MoveLegal)

{

//CHANGE

CheckReincarnation(FinishSquareReference); //Q12

//END CHANGE

int PointsForPieceCapture = CalculatePieceCapturePoints(FinishSquareReference);

CurrentPlayer.ChangeScore(-(Choice + (2 \* (Choice - 1))));

CurrentPlayer.UpdateQueueAfterMove(Choice);

UpdateBoard(StartSquareReference, FinishSquareReference);

UpdatePlayerScore(PointsForPieceCapture);

Console.WriteLine("New score: " + CurrentPlayer.GetScore() + Environment.NewLine);

}

Code for CheckReincarnation:

//CHANGE

private void CheckReincarnation(int SquareReference) //Q12

{

int Row = SquareReference / 10;

if (CurrentPlayer.SameAs(Players[0]))

{

if (Row == NoOfRows && CountNormalPieces() < NoOfPieces)

{

Console.WriteLine("Congratulations, you have earned a reincarnation!");

bool Valid = false;

while (!Valid)

{

Console.WriteLine("Which column would you like your piece to be reincarnated on?");

int ReincarnationCol = Convert.ToInt32(Console.ReadLine());

if (Board[GetIndexOfSquare(10 + ReincarnationCol)].GetPieceInSquare() != null)

{

Console.WriteLine("The square must be empty.");

}

else

{

Board[GetIndexOfSquare(10 + ReincarnationCol)].SetPiece  
(new Piece("piece", Players[0], 1, "!"));

Valid = true;

}

}

}

}

else

{

if (Row == 1 && CountNormalPieces() < NoOfPieces)

{

Console.WriteLine("Congratulations, you have earned a reincarnation!");

bool Valid = false;

while (!Valid)

{

Console.WriteLine("Which column would you like your piece to be reincarnated on?");

int ReincarnationCol = Convert.ToInt32(Console.ReadLine());

if (Board[GetIndexOfSquare(6 \* 10 + ReincarnationCol)].GetPieceInSquare() != null)

{

Console.WriteLine("The square must be empty.");

}

else

{

Board[GetIndexOfSquare(6 \* 10 + ReincarnationCol)].SetPiece  
(new Piece("piece", Players[1], 1, "\""));

Valid = true;

}

}

}

}

}

//END CHANGE

Code for CountNormalPieces:

//CHANGE

private int CountNormalPieces() //Q12

{

int Pieces = 0;

foreach (Square S in Board)

{

Piece PieceInSquare = S.GetPieceInSquare();

if (PieceInSquare != null)

{

if (PieceInSquare.GetBelongsTo().SameAs(CurrentPlayer) && PieceInSquare.GetTypeOfPiece() == "piece")

{

Pieces += 1;

}

}

}

return Pieces;

}

//END CHANGE

## Testing:

* Correctly showing the moves as requested in the tests, specifically including the choice of the blocked square to reincarnate on and then the correct one. [**1 mark]**

Text

Description automatically generated

Text

Description automatically generated

# Task 13 (8 marks)

## Coding

* Putting the Ta’ziz in the correct place regardless of board size. **[1 mark]**
* Having a mechanism that correctly counts the number of turns that the Ta’ziz has been camped for. **[1 mark]**
* Resetting the CampedTurns attribute if the square becomes empty or changes owner. **[1 mark]**
* Allowing the player to make a move that costs zero points when they have held the Ta’ziz for two turns. **[1 mark]**
* Showing the correct ‘A’ and ‘a’ symbols when the Ta’ziz is occupied by overriding SetPiece. **[1 mark]**
* Correctly resetting the symbol for the Ta’ziz to ‘x’ when a player leaves by overriding RemovePiece. **[1 mark]**

### Example Solution

Changes to CreateBoard:

for (int Row = 1; Row <= NoOfRows; Row++)

{

for (int Column = 1; Column <= NoOfColumns; Column++)

{

if (Row == 1 && Column == NoOfColumns / 2)

{

S = new Kotla(Players[0], "K");

}

//CHANGE

else if (Row == NoOfRows / 2 && Column == NoOfColumns / 2) //Q13

{

S = new Taziz("x");

}

//END CHANGE

Changes to PlayGame:

if (MoveLegal)

{

int PointsForPieceCapture = CalculatePieceCapturePoints(FinishSquareReference);

//CHANGE

CurrentPlayer.UpdateQueueAfterMove(Choice); //Q13

UpdateBoard(StartSquareReference, FinishSquareReference);

if (!Board[GetIndexOfSquare(Convert.ToInt32((NoOfRows / 2).ToString() + (NoOfColumns / 2).ToString()))].GetCampedTwoTurns()) //Q13

{

CurrentPlayer.ChangeScore(-(Choice + (2 \* (Choice - 1))));

}

else

{

Console.WriteLine("You have camped for two whole turns - well done. This move is for free!");

}

//END CHANGE

UpdatePlayerScore(PointsForPieceCapture);

Console.WriteLine("New score: " + CurrentPlayer.GetScore() + Environment.NewLine);

}

Changes to Square

//CHANGE

public virtual bool GetCampedTwoTurns() //Q13

{

return false;

}

//CHANGE

Code for Taziz:

//CHANGE

class Taziz : Square //Q13

{

private int CampedTurns = 0;

public Taziz(string S) : base()

{

Symbol = S;

}

public override void SetPiece(Piece P)

{

base.SetPiece(P);

BelongsTo = P.GetBelongsTo();

if (P.GetBelongsTo().GetName() == "Player Two")

{

Symbol = "a";

CampedTurns = 0;

}

else

{

Symbol = "A";

CampedTurns = 0;

}

}

public override Piece RemovePiece()

{

Symbol = "x";

CampedTurns = 0;

Piece PieceToReturn = PieceInSquare;

PieceInSquare = null;

return PieceToReturn;

}

public override bool GetCampedTwoTurns()

{

if (CampedTurns == 4)

{

CampedTurns = 0;

return true;

}

else

{

if (PieceInSquare != null)

{

CampedTurns++;

}

return false;

}

}

}

//END CHANGE

## Testing:

* Show the Ta’ziz being occupied and changing from x to A. [**1 mark]**
* Show player one getting a free move. [1 mark]

Text

Description automatically generated

*…..(steps while player 1 is camping not shown)*

Text

Description automatically generated

# Task 15 (15 marks)

## Coding

* Creating a Barrier class that accepts the parameters Player and Symbol and sets the attributes correctly.   
  **[1 mark]**
* Creating ContainsBarrier that returns true for B or b and false otherwise. **[1 mark]**
* Modifying CheckSquareIsValid to return false if the end square contains a barrier. **[1 mark]**
* Creating CheckBarrierIsValid which checks that all the squares for the barrier are in bounds. **[1 mark]**
* Creating CheckBarrierIsValid which checks that all the squares for the barrier are empty. **[1 mark]**
* Creating PlaceBarrier that successfully creates a barrier 3 squares wide on the board. **[1 mark]**
* Modifying CreatePieces with suitable input messages that calls CheckBarrierIsValid and makes two calls to PlaceBarrier, one for each player. **[1 mark]**
* Creating CheckManhattanDistance and modifying PlayGame to call that as well as calling CheckPlayerMove for the line starting MoveLegal=. **[1 mark]**
* Inside CheckManhattanDistance, swapping the start column/row and end column/row where necessary to reduce the number of checks which need to be made. **[1 mark]**
* Inside CheckManhattanDistance, iterating along the row and column and vice-versa even if this doesn’t work and has only been attempted. **[1 mark]**
* Inside CheckManhattanDistance, correctly iterating along the row and column and vice-versa for both players in all combinations of up, down, left and right (with and without vertical/horizontal movement) for both players in all possible move orientations. **[1 mark]**

### Example Solution

Code for Barrier:

//CHANGE

class Barrier : Square //Q15

{

public Barrier(Player P, string S) : base()

{

BelongsTo = P;

Symbol = S;

}

}

//END CHANGE

Changes to Square:

//CHANGE

public virtual bool ContainsBarrier() //Q15

{

if (Symbol == "B" || Symbol == "b")

{

return true;

}

else

{

return false;

}

}

//END CHANGE

Changes to CheckSquareIsValid (Dastan class):

private bool CheckSquareIsValid(int SquareReference, bool StartSquare)

{

if (!CheckSquareInBounds(SquareReference))

{

return false;

}

//CHANGE

if (Board[GetIndexOfSquare(SquareReference)].ContainsBarrier()) //Q15

{

return false;

}

//END CHANGE

Code for CheckBarrierIsValid (Dastan class):

//CHANGE

private bool CheckBarrierIsValid(int SquareReference) //Q15

{

for (int i = SquareReference - 1; i <= SquareReference + 1; i++)

{

if (!CheckSquareInBounds(i))

{

return false;

}

Piece PieceInSquare = Board[GetIndexOfSquare(i)].GetPieceInSquare();

if (PieceInSquare != null || Board[GetIndexOfSquare(i)].ContainsKotla() || Board[GetIndexOfSquare(i)].ContainsBarrier())

{

return false;

}

}

return true;

}

//END CHANGE

Code for PlaceBarrier (Dastan class):

//CHANGE

private void PlaceBarrier(Player P, int BarrierMiddleReference, string S) //Q15

{

int BarrierStartReference = BarrierMiddleReference - 1;

int BarrierEndReference = BarrierMiddleReference + 1;

for (int i = BarrierStartReference; i <= BarrierEndReference; i++)

{

Barrier B = new Barrier(P, S);

Board[GetIndexOfSquare(i)] = B;

}

}

//END CHANGE

Changes to CreatePieces (Dastan class):

CurrentPiece = new Piece("mirza", Players[1], 5, "2");

Board[GetIndexOfSquare(NoOfRows \* 10 + (NoOfColumns / 2 + 1))].SetPiece(CurrentPiece);

//CHANGE

for (int i = 0; i < 2; i++) //Q15

{

bool Valid = false;

while (!Valid)

{

DisplayBoard();

Console.WriteLine("Placing barrier piece for " + Players[i].GetName());

Console.WriteLine("A barrier is 3 squares wide. You cannot jump it, move it or occupy it. \nSelect the location for the middle piece of the barrier: ");

int BarrierLocationReference = Convert.ToInt32(Console.ReadLine());

int StartRow = BarrierLocationReference / 10;

int StartColumn = BarrierLocationReference % 10;

bool ValidLocation = CheckBarrierIsValid(BarrierLocationReference);

if (ValidLocation)

{

if (i == 0)

{

PlaceBarrier(Players[i], BarrierLocationReference, "B");

Valid = true;

}

else

{

PlaceBarrier(Players[i], BarrierLocationReference, "b");

Valid = true;

}

}

else

{

Console.WriteLine("That is not a valid location, please try again.");

}

}

}

//END CHANGE

Changes to PlayGame (Dastan class):

while (!SquareIsValid)

{

FinishSquareReference = GetSquareReference("to move to");

SquareIsValid = CheckSquareIsValid(FinishSquareReference, false);

}

//CHANGE

bool MoveLegal = CurrentPlayer.CheckPlayerMove(Choice, StartSquareReference, FinishSquareReference) && CheckManhattanDistance(StartSquareReference, FinishSquareReference); //Q15

if (MoveLegal)

{

int PointsForPieceCapture = CalculatePieceCapturePoints(FinishSquareReference);

CurrentPlayer.ChangeScore(-(Choice + (2 \* (Choice - 1))));

CurrentPlayer.UpdateQueueAfterMove(Choice);

UpdateBoard(StartSquareReference, FinishSquareReference);

UpdatePlayerScore(PointsForPieceCapture);

Console.WriteLine("New score: " + CurrentPlayer.GetScore() + Environment.NewLine);

}

else

{

Console.WriteLine(Environment.NewLine + "That is not a valid move" + Environment.NewLine);

}

//END CHANGE

Code for CheckManhattanDistance (Dastan class):

//CHANGE

public bool CheckManhattanDistance(int StartSquareReference, int FinishSquareReference) //Q15

{

int StartRow = StartSquareReference / 10;

int StartColumn = StartSquareReference % 10;

int FinishRow = FinishSquareReference / 10;

int FinishColumn = FinishSquareReference % 10;

bool RouteAVerticalClear = true;

bool RouteAHorizontalClear = true;

bool RouteBVerticalClear = true;

bool RouteBHorizontalClear = true;

if (StartRow > FinishRow) //If required, swap the start and finish points around so we only have to check one way

{

int Temp = StartRow;

StartRow = FinishRow;

FinishRow = Temp;

}

if (StartColumn > FinishColumn)

{

int Temp = StartColumn;

StartColumn = FinishColumn;

FinishColumn = Temp;

}

for (int i = StartRow; i <= FinishRow; i++)

{

int TargetSquareReference = int.Parse(i.ToString() + StartColumn.ToString());

if (Board[GetIndexOfSquare(TargetSquareReference)].ContainsBarrier())

{

RouteAVerticalClear = false;

}

TargetSquareReference = int.Parse(i.ToString() + FinishColumn.ToString());

if (Board[GetIndexOfSquare(TargetSquareReference)].ContainsBarrier())

{

RouteBVerticalClear = false;

}

}

for (int i = StartColumn; i <= FinishColumn; i++)

{

int TargetSquareReference = int.Parse(StartRow.ToString() + i.ToString());

if (Board[GetIndexOfSquare(TargetSquareReference)].ContainsBarrier())

{

RouteBHorizontalClear = false;

}

TargetSquareReference = int.Parse(FinishRow.ToString() + i.ToString());

if (Board[GetIndexOfSquare(TargetSquareReference)].ContainsBarrier())

{

RouteAHorizontalClear = false;

}

}

return (RouteAVerticalClear && RouteAHorizontalClear) || (RouteBVerticalClear && RouteBHorizontalClear);

}

//END CHANGE

## Testing:

* Moving the piece correctly when only one route is valid. **[1 mark]**
* Not moving the piece for a cuirassier move when there is a barrier in the way. **[1 mark]**
* Not moving the piece when the end square is a barrier. **[1 mark]**
* Not moving the piece when there is a barrier in the way on both routes and the travel directions are both inverted (right to left and bottom to top). **[1 mark]**

Text

Description automatically generated

A screenshot of a computer

Description automatically generated with medium confidence

Text

Description automatically generated