Game Mechanics:

→ The main data structure used in this version of the game is a table, where each of the cells represents a square on the Player's/CPU boards. Each cell contains a number which represents the state of the square. A legend outlining the interpretation of the values is given below:

Value in a cell:	Interpretation
0 :	Empty
1:	Occupied by a Ship
2:	HIT
3:	MISS

→ We also defined an enumeration type, SelectionMode to understand the click behavior on the form Its members and their attributes are given as follows:

PRE_GAME: The Game has not begun yet.
VERTICAL_HIGHLIGHT: The orientation of the ship is vertical, cells are highlighted vertically. Used when player is placing the ships.
HORIZONTAL_HIGHLIGHT: The orientation of the ship is horizontal, cells are highlighted horizontally. Used when player is placing ships.
PLAYER_MOVE: Its Player's turn to move. Computer is waiting for a move from the player.
GAME_WIN: Player won the Game
GAME_LOSE: Player lost the Game

We made various static variable declarations to define size of ships and other variables that are used extensively through the game. They are as follows:

static SelectionMode mode: Used to easily access the enumeration type SelectionMode

static MousePosition position: Used to access position of the mouse on the form using the class MousePosition

static final int[] shipSizes : Used to store the different ship sizes. In case you wish to change a value, all you have to do is update the information in this array and it will approve the changes without breaking the code.

static final int placement: Stores the placement of the ship with respect to the size of the ship

Game winner is determined using the squareRemainingPlayer/CPU. These variables hold the total number of squares that are holding a ship. This can be also thought as the sum of all the ship sizes. When a player/computer runs out of squares they lose the game

static Random random: Used to generate random numbers for the computer moves.

static int[][] possibleMoves: A 2D array that stores all the possible moves that AI can make.

static int[] moveIndex: An array that stores all the index of the current/future move location for AI.

static ArrayList<PlayerRecord> stats: An array list that stores the stats of the current player.

static String currentPlayerName: Stores the name of the current Player.

Module Interface Specification

Class: BattleshipGUI

startNewGame ( )
Clears boards and resets parameters to pregame state.

placeShip (length: IN int; row: IN int; column: IN int; isHorizontal: IN boolean)
If the ship location is valid, sets the state of squares on the player's board to occupied, then calls computerPlaceShip.

computerPlaceShip ( )
Algorithmically determines a valid location, then sets the state of the squares to occupied.

playerMove (row: IN int; column: IN int)
Changes the state of the square at the given row and column on the computer's board to either a hit or miss. Checks for win condition, then calls computerMove.

computerMove ( )
Algorithmically determines the indices of a square on the player's board, then changes its state to either hit or miss. Checks for win condition.

recordStats (name: IN string; spread: IN int)
Updates player statistics. Spread is given as the difference between player and computer squares remaining, and is used to calculate scores.

saveStats ( )
Saves player statistics to file.

loadStats ( )
Loads player statistics from file.

saveGameState ( )
Saves board states and the AI memory variables to file. The game can then be loaded later.

resumeGameState ( )
Loads a previously saved game from file.

initComponents()
Initiates the Graphical User Interface (GUI) of the game; setting the board and menu.

hvToggleActionPerfermed()
Initiates when the "H/V" button is pressed, which allows the player to toggle the orientation of the ship between Horizontal and Vertical when placing the ships in the beginning.

startGameButtonActionPerformed()
Initiates when the "Start" button is pressed, which can get the user's name, store their stats, or allow the user to place their ships.

newGameMenuItemActionPerformed()

Initiates when “New Game” is pressed, which resets all stats, the game board and starts a new game.

playerStatsMenuItemActionPerformed()
Initiates when “Player Stats” button is pressed, which displays the stats of a player by their choosing.

saveGameMenuItemActionPerformed()
Initiates when “Save Game” is pressed, which saves the current state of the game and remembers current position of ships and previous moves.

loadGameMenuItemActionPerformed()
Initiates when “Load Game” is pressed, which opens a previously saved game and displays the position of the past hits and misses.

clearAllStatsButtonActionPerformed()
Initiates when “Clear all Stats” is pressed, which removes all previous saved statistics of all Players.

clearPlayerStatsButtonActionPerformed()
Initiates when “Clear Player Stats” is pressed, which removes the selected Player’s stats.

Class: PlayerRecord

PlayerRecord (name: IN string)

Constructs a record for the given name with default values.

addMatch (score: IN int)

Updates wins/losses, average score, and best score.

Class: HoverMouseAdapter

mouseMoved (event: IN MouseEvent)

Gets the new location of the mouse cursor every time it moves, and repaints the table.

Class: ComputerBoardClickListener

mousePressed(event: IN MouseEvent)

Calls playerMove using the square clicked. Does nothing during ship placement or post-game.

Class: PlayerBoardClickListener

mousePressed(event: IN MouseEvent)

Calls placeShip using the square clicked. Does nothing after ship placement.

Class: BoardRenderer

This class controls everything related to displaying the player and computer game boards. Implementation- and language-specific.

Implementation

Class: BattleshipGUI

Uses: None

Variables:

length: **int**
Length of the Ship

row: **int**
Row number on Player's/Computer's Board. Ranges from 0 to 9

column: **int**
Column number on Player's/Computer's Board. Ranges from 0 to 9

isHorizontal: **boolean**
Checks whether the ship is in horizontal orientation or not

canPlace: **boolean**
Checks whether a ship can be placed on the selected position or not

isHorizontalInt: **int**
Variable for storing the orientation of the ship, i.e., 1 = Horizontal and 2 = Vertical

goodGuess: **boolean**
Makes sure that the guess is legal.

squaresRemainingCPU: **int**
Stores the number of ships on board for AI

squaresRemainingPlayer: **int**
Stores the number of ships on board for Player

Access Program:

PlaceShip (**int** length, **int** row, **int** column, **boolean** isHorizontal): **NULL**

Place a Ship, for Player, at desired location after performing necessary check to see that the specified location is NOT OUT OF BOUND and/or OCCUPIED by another ship.

```
define boolean  canPlace  is equal to true

if (isHorizontal is equal to true)
   for all  int i less than (column + length)
         if (value at specified row and column, i, in the Player's Table is equal to 1)
              set canPlace equal to false

Exception: if the index of the specified row and column are OUT OF BOUND set canPlace
equal to false

   if (canPlace is equal to true)
      for all  int i less than (column + length)
           set value of the specified row and column ,i, in the Player's Table equal to 1)
       print "Ship Placed"   //Inform User about the Ship Status
      decrement possible number of ship to be placed by factor of 1

    else
        print "Can't place here."   //Inform User about the Error Status

else
     for all  int i less than (row + length)
         if (value at row, i,  and the specified column in the Player's Table is equal to 1)
              set canPlace equal to false

Exception: if the index of the specified row and column are OUT OF BOUND set canPlace
equal to false

if (canPlace is equal to true)
    for all  int i less than (row + length)
         set value of row, i,  and the specified column in the Player's Table equal to 1)
     print "Ship Placed"   //Inform User about the Ship Status
    decrement possible number of ship to be placed by factor of 1

 else
      print "Can't place here."   //Inform User about the Error Status

if(Possible placement locations is equal to ZERO)
   wait for the PLAYER to Attack the computer's board  //Player's Turn
```

Pre-Condition: Let the number of possible ship placement location on Player's Board be **n** before the method execution

Post-Condition: Let the number of possible ship placement location on Player's Board should be **0** after the method execution

ComputerPlaceShip (int length): NULL

Place a Ship, for Computer or AI, at randomly generated location after performing necessary check to see that the generated location is NOT OUT OF BOUND and/or OCCUPIED by another ship.

```
define generator, generator1, and generator2 , random number generators, for row, column, and
orientation, vertical or horizontal

generate 3 integers for row [range 10], column [range 10], and isHorizontalInt [range 2]
define boolean  isHorizontal

if(isHorizontalInt is equal to 1)
   then isHorizontal is equal to true

else
      isHorizontal is equal to false

define boolean  canPlace  is equal to true

if (isHorizontal is equal to true)
    for all  int i less than (column + length)
          if (value at generated row and column, i, in the Computer's Table is equal to 1)
             set canPlace equal to false

Exception: if the index of the generated row and column are OUT OF BOUND set canPlace
equal to false


    if (canPlace is equal to true)
        for all  int i less than (column + length)
              set value of the generated row and column ,i, in the Computer's Table equal to 1)
        print "Ship Placed"   //Inform User about the Ship Status
        decrement possible number of ships to be placed locations by factor of 1

    else
         call the method ComputerPlaceShip(length)

else
     for all  int i less than (row + length)
           if (value at row, i,  and the generated column in the Computer's Table is equal to 1)
               set canPlace equal to false

Exception: if the index of the generated row and column are OUT OF BOUND set canPlace
equal to false


     if (canPlace is equal to true)
         for all  int i less than (row + length)
              set value of row, i,  and the generated column in the Computer's Table equal to 1)
```

Pre-Condition: Let the number of ships to be placed on Computer's Board be **n** before the method execution

Post-Condition: Let the number of ships to be placed on Computer's Board should be **0** after the method execution

ComputerMove (): NULL

Generates attacks for Computer by first using randomly generated locations. Once it detects a hit it explores the neighboring area by searching all four possible locations around the acquired hit.

```
/* moveIndex corresponds to search direction in possibleMoves
A legend for interpreting the moveIndex is as follows:

moveIndex: Search Direction
-1: Search Random Square
0: Search UP
1: Search DOWN
2: Search LEFT
3: Search RIGHT
possibleMoves[moveIndex] = NULL, iff DIRECTION NOT POSSIBLE
*/

Define row for storing row index
Define col for storing column index

if( moveIndex is equal to -1)
    //Pick Random numbers to attack on the PLAYER'S Board
    generate random  integers for row and col, between the range  of [0,9]
    while(value on the Player's board at generated row and col is greater than 1)
            set row equal to another randomly generated integer between [0,9]
            set col equal to another randomly generated integer between [0,9]

     //If attack miss, do not change anything
     if(value on the Player's board at generated row and col is equal to 0)
        set the value at that position equal to 3

     //Make random hit
     else if(value on the Player's board at generated row and col is equal to 1)
            decrement squaresRemainingPlayer by the factor of 1
            set the value at that position equal to 2

            //Search UP
            if( (generated row MINUS 1)is greater than or equal to 0)
               if(value on the Player's board at generated (row MINUS 1) and col is less than  2)
                  set possibleMoves indexed at ZERO equal  to new integer array containing generated
                      (row MINUS 1) and col
            //Search DOWN
            if( (generated row PLUS 1)is strictly less than  10)
               if(value on the Player's board at generated (row PLUS 1) and col is less than  2)
                  set possibleMoves indexed at ONE equal  to new integer array containing generated
                      (row PLUS 1) and col

            //Search LEFT
            if( (generated col MINUS 1)is greater than or equal to 0)
               if(value on the Player's board at generated row and (col MINUS 1)  is less than  2)
                  set possibleMoves indexed at TWO equal  to new integer array containing generated
                      row and (col MINUS 1)

            //Search RIGHT
            if( (generated col PLUS 1)is strictly less than  10)
               if(value on the Player's board at generated row and (col PLUS 1) is less than  2)
                  set possibleMoves indexed at THREE equal  to new integer array containing generated
                      row  and (col PLUS 1)

//Try next Possible direction
else
     set row equal to possibleMoves indexed at [moveIndex] [ZERO]
     set col equal to possibleMoves indexed at [moveIndex] [ONE]

     //if miss, eliminate all possible direction by setting it to NULL
     if(value on the Player's board at generated row and col is equal to 0)
        set the value at that position equal to 3
        set possibleMoves indexed at [moveIndex] equal to NULL

     //if hit, check next possible location in that direction
     else if(value on the Player's board at generated row and col is equal to 1)
             decrement squareRemainingPlayer by the factor of 1
             set the value at that position on Player's board equal to 2

     switch(moveIndex)
                    case 0:
                           if((generated row MINUS 1)is strictly less than 0)
                              set possibleMoves indexed at ZERO equal to NULL
                           else
                               if(value on the Player's board at generated (row MINUS 1) and col is less
                                      than  2)
                                  set possibleMoves indexed at ZERO equal  to new integer array
                                      containing generated (row MINUS 1) and col
                               else
                                   set possibleMoves indexed at ZERO equal to NULL

                          break

                    case 1:
```

```
                    if((generated row PLUS 1)is strictly greater than 9)
                       set possibleMoves indexed at ONE equal to NULL
                    else
                        if(value on the Player's board at generated (row PLUS 1) and col is less
                               than 2)
                           set possibleMoves indexed at ONE equal to new integer array
                               containing generated (row PLUS 1) and col
                        else
                            set possibleMoves indexed at ONE equal to NULL

                   break

              case 2:
                    if((generated col MINUS 1)is strictly less than 0)
                       set possibleMoves indexed at TWO equal to NULL
                    else
                        if(value on the Player's board at generated row and (col MINUS 1) is less
                               than 2)
                           set possibleMoves indexed at TWO equal to new integer array
                               containing generated row and (col MINUS 1)
                        else
                            set possibleMoves indexed at TWO equal to NULL

                   break

              case 3:
                    if((generated col PLUS 1)is strictly greater than 9)
                       set possibleMoves indexed at THREE equal to NULL
                    else
                        if(value on the Player's board at generated row and (col PLUS 1) is less
                               than 2)
                           set possibleMoves indexed at THREE equal to new integer array
                               containing generated row and (col PLUS 1)
                        else
                            set possibleMoves indexed at THREE equal to NULL

                        break
define allNull: boolean equal to true
for(index in range 0 to 3 in reverse order)
     if(possibleMoves at current index is NOT equal to NULL)
        set allNull equal to false
        set moveIndex equal to current index

//if all direction are checked reset moveIndex to -1 for next attack
if(allnull is equal to true)
   set moveIndex equal to -1

//if the player runs out of ships, then they LOSE the GAME
if(squareRemainingPlayer is equal to ZERO AND current mode does NOT equal the GAME_LOSE,
                        SelectionMode)
  Change the current mode to GAME_LOSE, SelectionMode
  Print "You Lose" message on the screen

  if(name of the current Player does NOT equal EMPTY_STRING)
     call recordStats method with parameters currentPlayerName and (squareRemainingCPU X -1))
```

Pre-Conditions: The value of squaresRemainingPlayer is equal to n, before the method execution.

Post-Conditions: The value of squaresRemainingPlayer is less than or equal to n, after the method execution.

PlayerMove(int row, int column)

Allows the Player to attack the computers board and checks if the player hit or miss the enemy battle ship after turn completion

This is done as follows:

```
Define boolean goodGuess equal to true

Print "Take a shot!"

if (value at specified row and column, in the AI's Table is equal to 0)
        set value of the specified row and column , in the AI's Table equal to 3

else if (value at specified row and column, in the AI's Table is equal to 1)
        set value of the specified row and column , in the AI's Table equal to 2
        squaresRemainingCPU is equal to squaresRemainingCPU - 1

else if (value at specified row and column, in the AI's Table is equal to 2)
        Print "Take a shot!"
        Set goodGuess as false
else if (value at specified row and column, in the AI's Table is equal to 3)
        Print "Take a shot!"
        Set goodGuess as false
if (squaresRemainingCPU is 0)
        Print "You win!"
```

Pre – condition: The number of squares remaining on CPU board, before the method execution, is equal to n
Post – condition: The number of squares remaining on CPU's board, after method execution, is less than n

---

startNewGame ()

This method will reset everything on the Player/CPU boards. It will delete the scores of the current game.

```
Set possiblemoves equal to a 2D Array with NULL
Set moveIndex equal to -1
Define a new MousePosition object with NULL value and row and column equal to ZERO
Set placement equal to  ZERO
Change the mode of the game to PRE_GAME
Set squareRemainingPlayer equal to 16
Set squareRemainingCPU equal to 16
```

Pre – condition: Let Score of the current game be **n**, before method execution
Post – condition: The Score of the current game, after method execution, is **0**

---

saveStats ()

This method will save the stats of the current player in a file. If the file does not exist, then it will provide with a terminal error message.

```
Define a String variable to store the name of the file, "record.ser"
Create a new fileOutputStream and objectOutputStream object
Write  the stats on the newly created file as output stream

Exception: if an IOException is caught print a message on terminal stating that an ERROR has occurred.
```

Pre – condition: The number of player entries is **equal** to **n**, before the method execution
Post – condition: The number of player entries is **greater than equal** to **n**, after the method execution

---

loadRecords ()

This method will load all the player stats that are stored in a file. If the file does not exist, then it will create a new file and save all the data in it.

```
Define a String variable to store the name of the file, "record.ser"
Create a new file object with the above given file name.
if (file with given name already EXISTS)
    Create a new fileOutputStream and objectOutputStream object
    Read  the stats from the previously opened file as readObject and store them in an ArrayList

Exception: if an IOException, ClassNotFoundException, or FileNotFoundException is caught call saveStats Method and save the current stats in a new file.
```

Pre – condition: The number of player entries is **equal** to **0** or **n**, before the method execution

Post – condition: The number of player entries is **equal** to **n, if initial n. and 1, if initial 0**, after the method execution

---

saveGameState ()

This method will save the current state of Player and CPU's boards in a file. It will save the current positions of the ships and all the attacks that were made before saving.

```
Define currentBoardState as a 3D array with dimensions of 2x10x10
for r: int from 0 to 9
    for c: int from 0 to 9
        currentBoardState with dimensions of 0 x r x c is equal to the value at (r,c) of
        playerBoard
        currentBoardState with dimensions of 1 x r x c is equal to the value at (r,c) of
        computerBoard

Define state: GameState taking the following input parameters (currentBoardState,
        currentPlayerName, placements, possibleMoves, moveIndex, mode)

Define fileName: String as "gamestate.ser"

Try: Writing state information to the file "gamesate.ser"

Exception: If data cannot be written to the file, then display "Error writing to file '" + filename
```

---

resumeGameState ()

This method will resume the previously saved state of Player and CPU's boards. It will reload all the data that was previously stored on a file, and allow the user to continue their game.

```
Define fileName: String as "gamestate.ser"

Define file: File with input parameter as fileName

if(file.exists is true )

    Define in: ObjectInputStream
    Define state: GameState
    Load the previous state
    Display "Game Loaded."

    Exception: If file not found, then display "Unable to Load" + fileName
               If input/ output operation failed, then display "Error reading file "+ fileName
```

```
else
        Display "No game to load."
```

---

recordStats(String name, int spread):NULL

Records Name, Wins, Losses, High Score, Best Score, and Average Score
This is archived by:

```
define boolean exists equal to false

// if the player record with same name already exists
for (All the recorded Players in the record database)
    if(name(Input Parameter) is equal to any previously recorded name)
        set exists equal to true
        run the recods corresponding to the input name through addMatch method with input
        parameter equal to spread

// if the player record with same name does NOT exist
if(exists is equal to false)
        create new Player records with the new name (Input Parameter)
        run addMatch method with the provided spread as the input parameter.
        Add the new Player stats to the database

Run the save method
```

Pre-condition: **Number of records** before method execution is **equal** to **n**

Post-condition: **Number of records** after method execution is **greater than or equal** to **n**

---

initComponents ()

This private method will initiate the Graphical User Interface for the game. It will setup the main game board seen by the User, including the menu.

```
Create new dialog box for newGameDialog
Create two grids, for each row/column ranging from 0 to 9, on the newGameDialog window
First Grid is labeled "Player" and second Grid is labeled "Computer"
Create a button labeled "H/V" for toggling between horizontal and vertical orientation of the ships
Create the menu bar with single field titled "File" with four subfields titled "New Game", "Player
Stats", "Save Game", and "Load Game"

Selecting a "New Game" option will open a new dialog box with dimensions 300 X 200 and a field
        Labeled "Enter Your Name. Leave blank to Play anonymously"
```

```
Create a text box – where user can enter their name.
Create a button labeled "Start" – will begin the game

Selecting a "Player Stats" option will open a new dialog box with a dimensions 500 X 500 and a table
            with field labeled "Name", "Wins", "Losses", "Average", and "Best"
Name will store the name of player entered by players when they started the New Game.
Other field will store the player stats based on the past wins and losses and the number of squares
that they won or lost.
Create two buttons titled 'Clear Player' – delete the selected player and 'Clear all' – deletes data of all
Players from the database.

Use action listener to get the appropriate click information from the main screen and pass the
information down to the methods that requires their use to provide with correct response.
```

---

hvToggleActionPerformed () //Toggles the orientation(Horizontal/ Vertical) of the ship

This private method will be initiated by pressing the button labeled "**H/V".** The main objective of this method is to toggle the orientation of the ship between Horizontal and Vertical. It is used when the player is placing the ships on their grid.

```
if(current selection mode is equal to HORIZONTAL_HIGHLIGHT)
    Change the selection mode  to VERTICAL_HIGHLIGHT)

else if(current selection mode is equal to VERTICAL_HIGHLIGHT)
        Change the selection mode  to HORIZONTAL_HIGHLIGHT)
```

---

startGameButtonActionPerformed () //Start Button in New Game Dialog Box

This private method will be initiated by pressing the button labeled "**Start",** can be seen after clicking **File -> New Game.** The main objective of this method is to get the name of the user, to store stats, and to provide user with the opportunity to place their ships

```
Set current Player Name equal to the name provided by the user in the text box.
Set current selection mode  to horizontal highlights
Close the "New Game" dialog box  and wait for player to start playing
```

---

newGameMenuItemActionPerformed () //New Game

This private method will be initiated by clicking **File -> New Game.** The main objective of this method is to reset every entry in the statistics table and Player and CPU board/table, and start a new game

```
for(all rows in the Player's board/table)
     for(all columns in the Player's board/table)
          set the value in Player's board/table, of all the cells to ZERO
         set the value in Computer's board/table, of all the cells to ZERO
```

```
run startNewGame Method
open "New Game" dialog box with dimensions 300 X 200
```

playerStatsMenuItemActionPerformed () //Print Stats

This private method will be initiated by clicking **File -> Player Stats.** The main objective of this method is to provide user with an organized table of statistics corresponding to a unique player name, chosen by them. The table contains **five** fields – **Name** – Player Name, **Wins** – Total Number of Wins for the Player, **Losses** – Total Nuber of Losses by the Player, **Best** – Best Score earned by the Player, and **Average** – Average Score received by the Player.

```
//Size of the Stats refers to the total number of Players names.
Create a new table, with dimensions (size of stats by 5), for the five fields – "Name", "Wins",
        "Losses", "Average", and "Best"
for(all Player Names in the size of stats)
    set the first column equal to the Player Names arranged according to their creation time. i.e.,
        older Players will appear at the top rows of the list.
    set the second column equal to the total number of Wins earned by the Current Player.
    set the third column equal to the total number of Losses earned by the Current Player.
    set the fourth column equal to the Average Score earned by the Current Player.
    set the fifth column equal to the Best Score earned by the Current Player.

Display the Print Stats dialog box
```

saveGameMenuItemActionPerformed () //Save Game

This private method will be initiated by clicking **File -> Save Game.** The main objective of this method is to save the current state of the game, for future **Gameplay**, by remembering the position of the current hits and misses.

```
If (current mode is equal to PLAYER_MOVE)
    Run saveGameState Method
```

loadGameMenuItemActionPerformed () //Load Game

This private method will be initiated by clicking **File -> Load Game.** The main objective of this method is to open a previously saved state of the game, for current **Gameplay**, by displaying the position of the past hits and misses.

```
Run resumeGameState Method
```

clearAllStatsButtonActionPerformed () //Clear All Stats

This private method will be initiated by clicking **File -> Player Stats -> Clear All Stats.** The main objective of this method is to remove all previously saved stats of all the Players.

```
Set stats equal to an ArrayList of length ZERO
Reset everything in the stats Table. i.e., create new stat object to store new Name, Wins, Losses, Average, and Best Scores

Save this new state of the game by calling saveStats Method
```

---

clearPlayerStatsButtonActionPerformed () //Clear Player Stats

This private method will be initiated by clicking **File -> Player Stats -> Clear Player Stats.** The main objective of this method is to remove the stats the selected Player.

```
if(selected row of the stat table is greater than or equal to ZERO)
    remove the stats of the selected row

set Stat Table model to the regular ‘5 column modal’ explained in the initCondition Method

for(all numbers in range 0 to size of the stats)
    set value in the table for cell with name at current index equal to ZERO
    set value in the table for cell with wins at current index equal to ONE
    set value in the table for cell with losses at current index equal to TWO
    set value in the table for cell with averageScore at current index equal to THREE
    set value in the table for cell with bestScore at current index equal to FOUR

call saveState Method
```

---

main() //Main Method

This private method will be initiated by clicking running the java executable file or java code file. The main objective of this method is to start the game and display form/boards.

```
Create “nimbus” look and feel

Exception: If classNotFound, instantiationException, or IllegalAccessException are found catch them

Set BattleshipGui to true and display the previously created form

Initialize the fields using startNewGame Method and load stats using loadRecord Method
```

Class: PlayerRecord

Variable:

name: **String**
Records the name

wins: **int**
Counts the number of wins

losses: **int**
Counts the number of losses

bestScore: **int**
Stores the highest score

averageScore: **int**
Calculates the average score

exists: **boolean**
Determines whether the records exist or not

spread: **int**
Number of squares the player had left when the game ended.

Access Program:

PlayerRecord(String name):NULL

This is a constructor method that declares default values for variables
This is achieved by:

name is **equal** to **name**
wins is **equal** to 0
losses is **equal** to 0
bestScore is **equal** to 0
averageScore is **equal** to 0

Pre-condition: Player record before the method execution is as follows:
name = NULL
wins = 0
losses = 0
bestScore= 0
averageScore = 0

Post-condition: Player record after the method execution is as follows:
name = name(input Parameter)
wins = 0
losses = 0

```
bestScore= 0
averageScore = 0
```

---

setName(String name) :NULL

Takes in a String parameter (name: **String**) and assigns it to name: **String**
This is achieved by:

```
name(current Object) is equal to name: String (parameter input)
```

Pre-condition: name before method execution is NULL
Post-condition: name after method execution is equal to name (Input Parameter)

---

addMatch(int score):NULL

Records wins, losses, high score and also average score
This is archived by:

```
if (score is greater than 0)
   wins is equal to wins + 1

if(score is greater than bestScore)
    bestScore is equal to score

else
      losses is equal to losses + 1

averageScore is equal to (averageScore x (wins + losses - 1) + score) / (wins + losses)
```

Pre-condition: let the records before execution of the method are as follows:
wins = w
losses = l
bestScore = b
averageScore = a

Post-condition: let the records after execution of the method are as follows:
wins = w+ 1 AND bestScore = b + score(Input Parameter) OR losses = l + 1 AND averageScore = a' (Updated average score), a (not necessarily)= a'

Class: GameState

Uses: BattleshipGUI

Variable:

boardState: **int[][][]**
Records the state of the board in a 3D Array where **first parameter** is **player type** (CPU/Player), **second parameter** is **row index**, and **third parameter** is **column index**

playerName: **String**
Name chosen by the Player at the start of the game

computerMoves: **int[][]**
Records the moves made by the Computer in a 2D Array where **first parameter** is **row** and **second parameter** is **column**

computerMoveIndex: **int**
the index determining the type of attack i.e., hit or miss

modeOfState: **SelectionMode**
Mode of state stores the current mode of the board using an enumerated type defined earlier called SelectionMode

shipsPlaced: **int**
Stores the number of ships already placed on the board

Access Program:

GameState(int[][][] boardState, String name, int shipsPlacement, int[][] cMoves, int cMoveIndex, SelectionMode mode):NULL

This is a constructor method that declares default values for variables
This is achieved by:

Current object's shipPlaced is **equal** to **shipPlacement**
Current object's boardState is **equal** to **boardState**
Current object's playerName is **equal** to **name**
Current object's computerMoves is **equal** to **cMoves**
Current object's computerMoveIndex is **equal** to **cMoveIndex**
Current object's mode is **equal** to **modeOfStates**

---

loadState():NULL

This is a constructor method that declares default values for variables
This is achieved by:

```
Create a new mouse position at row and column indexed at ZERO with table initialized as NULL
Set currentPlayerName equal  to  the playerName  of the current object
Set possibleMoves equal to the computerMoves of the current object
Set moveIndex equal to the computerMoveIndex of the current object
Set mode equal to the modeOfState of the current object
Set squaresRemainingPlayer equal to ZERO
Set squaresRemainingCPU equal to ZERO
Set placements to the shipPlaced of the current object

Define a 2D Array, pBoard and set it equal to the boardState of the current object indexed at ZERO
Define a 2D Array, cBoard and set it equal to the boardState of the current object indexed at ONE

for(all row ranging from 0 to 9)
    for(all column ranging from 0 to 9)
        set playerBoard indexed at (row, column) equal to the value at pBoard[row][column]
        set computerBoard indexed at (row, column) equal to the value at cBoard[row][column]

        if(the value at pBoard[row][column] is equal to 1)
            increment squareRemainingPlayer by factor of 1

        if(the value at cBoard[row][column] is equal to 1)
            increment squareRemainingCPU by factor of 1
```

<u>Class</u>: MousePosition

Variables:

table: JTable
Used to display two-dimensional table of cells.

highlightedRow: int
Currently selected row

highlightedColumn: int
Currently selected column

row: int
Row numbers from 0 to 9 on player's/AI's board

column: int
Column numbers from 0 to 9 on player's/AI's board

Access Program:

MousePosition(JTable table, int row, int column):NULL

This is a constructor method that declares default values for variables
This is achieved by:

```
Current object's table is equal to table(input parameter)
Current object's highlightedRow is equal to row
Current object's highlightedColumn is equal to column
```

Class: BoardRenderer

Variables:

table: **JTable**
Used to display two-dimensional table of cells.

value: **object**
Value/State Value present inside a particular cell in the Player's/CPU's Board

isSelected: **boolean**
Determines whether the current cell is selected or not.

hasFocus: **boolean**
Determines whether the current cell has hit value.

row: **int**
Row numbers from 0 to 9 on player's/AI's board

column: **int**
Column numbers from 0 to 9 on player's/AI's board

shipSize: **int**
size of a ship to be placed

shipSizes: **int** Array
An integer array containing the sizes for all ships.

placements: **int**
possible ship placement locations on the computer/Player boards

Access Program:

BoardRenderer()

This is a constructor method that declares default values for variables
This is achieved by:

setOpaque is **equal** to **true**

---

getTableCellRendererComponent(**JTable** **table**, **object** **value**, **boolean** **isSelected**, **boolean** hasFocus, **int** row, **int** column)

This Method is used to change/render the board according to the acquired inputs it will display changes such as colour of the cell. It will communicate the results of the attacks made by the user or the AI.

```
Define an int variable shipSize
Set the background of the form to GREY
Set the line border's on the form to BLACK

if(the value of placements is less than or equal to 5)
   set shipSize equal to global static variable shipSizes indexed at placements
else
     set shipSize equal to ZERO

if(current mode is HORIZONTAL_HIGHLIGHT AND current table is Player's Board)
   if(current table is position on table AND column is greater than or equal to highlighted column,
            AND column is less than (highlighted column PLUS shipSize)
            AND row is equal to highlighted row)
       Set background colour to RED, of that particular square/cell

else if(current mode is VERTICAL_HIGHLIGHT AND current table is Player's Board)
        if(current table is position on table AND row is greater than or equal to highlighted row,
                  AND row is less than (highlighted row PLUS shipSize) AND column is equal to
                   highlighted column)
           Set background colour to RED, of that particular square/cell

else if(current mode is PLAYER_MOVE)
        if(current table is position on table AND row is equal to highlighted row, with a ship, AND
                 column is equal to highlighted column)
           Set background colour to ORANGE, of that particular square/cell

/*  value: Interpretation
      1: UNHIT SHIP, there is a ship here which is NOT HIT yet.
      2: SHIP HIT, Attack HITs this ship
      3: MISSED HIT, There is NO ship here, so ATTACK MISSED
*/

if(the value is equal to 1)
   if(table is equal to Player's Table)
      set the background colour of the square to DARK GRAY

else if(the value is equal to 2)
        set the background colour of the square to RED

else if(the value is equal to 3)
        set the background colour of the square to BLUE

repaint the table AND return the newly Painted Table
```

Class: ComputerBoardClickListener

Extends: MouseAdapter

Variables:

table: JTable
Used to display two-dimensional table of cells.

row: int
Row numbers from 0 to 9 on AI's board

column: int
Column numbers from 0 to 9 on AI's board

Access Program:

mousePressed (MouseEvent e)

Allows the Player to select and target a box of their choosing on the computer's board.

```
set JTable table equal to the selected point            //returns a type object e.getSource
set int row equal to row value of the selected point
set int column equal to column value of the selected point

if (current selection mode is equal to PLAYER_MOVE)
        call the method playerMove using the current row and column values
```

Pre-condition: the **values** for **row and column** equal to **n** before the method execution.

Post-condition: The **row/column value** in the method **playerMove** is the same as the **corresponding values** in this **class** before the method **playerMove** is executed.

Class: PlayerBoardClickListener

Extends: MouseAdapter

Variables:

table: JTable
Used to display two-dimensional table of cells.

row: int
Row numbers from 0 to 9 on player's board

column: int
Column numbers from 0 to 9 on player's board

Access Program:

mousePressed (MouseEvent e)

Allows the Player to select an initial coordinate on their board for ship placement.

```
set JTable table equal to the selected point            //returns a type object e.getSource
set int row equal to row value of the selected point
set int column equal to column value of the selected point

if (current selection mode is equal  to HORIZONTAL_HIGHLIGHT)
        call the method placeShip using the current row/column  values and set          the
boolean isHorizontal equal to true

else if (current selection mode is equal  to VERTICAL_HIGHLIGHT)
        call the method placeShip using the current row/column  values and set          the
boolean isHorizontal equal to false
```

Pre-condition: the **values** for **row and column** equal to **n** before the method execution.

Post-condition: The **row/column value** in the method **placeShip** is the same as the **corresponding values** in this **class** before the method **placeShip** is executed.

Class: HoverMouseAdapter

Extends: MouseMotionAdapter

Variables:

table: JTable
Used to display two-dimensional table of cells.

highlightedRow: int
Currently selected row

highlightedColumn: int
Currently selected column

Access Program:

mouseMoved (MouseEvent e)

Highlights the box where the player's mouse is currently located.

```
set JTable table equal to the where the mouse is currently place
//returns a type object e.getPoint
set the current position to the corresponding location on the JTable table
set int highlightedRow equal to row value of the current position
set int highlightedRolumn equal to column value of the current position
update the current position by repainting the table AND return the newly repainted table
```

Pre-condition: the **values** for **highlightedRow** and **highlightedColumn** equal to **n** before the method execution.

Post-condition: only has one **highlighted area** on the board after every repaint.