Criterion C – Development

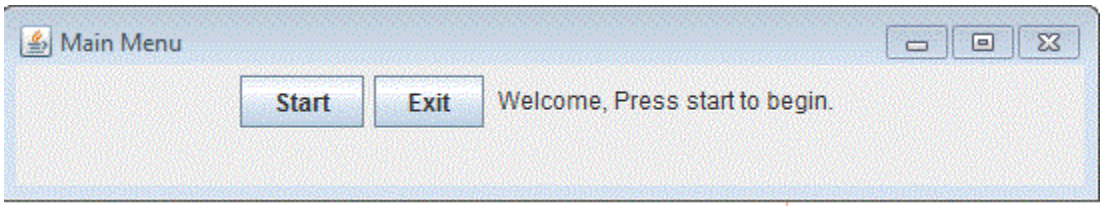
Techniques:

- Graphical Interface – JFrame
- Global Variables
- Global Arrays
- Matching of Parameters
- Legality of Moves - Gridworld

NOTE: Because the rules of Checkers was an integral component of the development of this product, a source offering information about the rules of checkers was referred to. The citation of this source is available in Appendix C, and the rules themselves are available in Appendix A. In addition, the citation to the Gridworld Case Study is available in Appendix C, as Gridworld was thoroughly implemented.

Graphical Interface

The graphical interface menus will be launched initially when the program is run. They will allow the user to input information before the actual game starts. These menus were created using Java's JFrame GUI component.

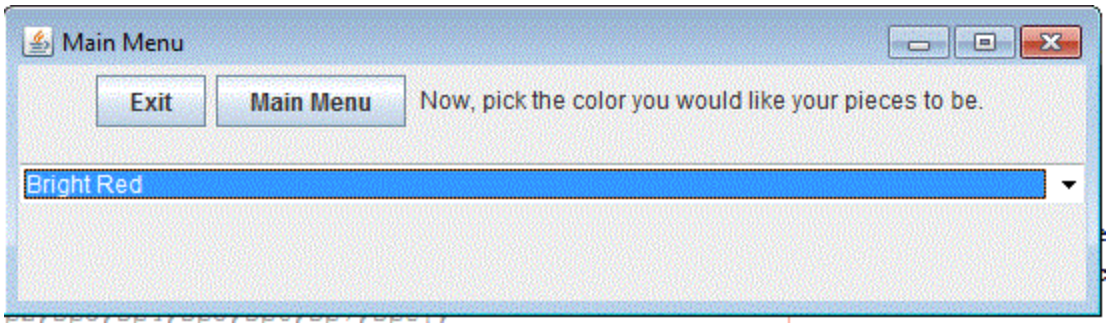


The game starts with the Start and exit buttons.

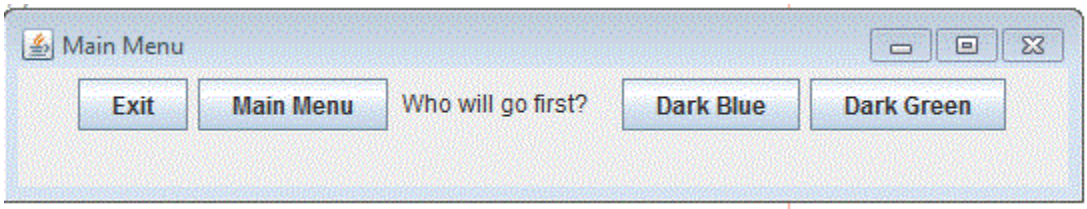
"START" – Starts the game. Only available at the main menu at the beginning of the game.

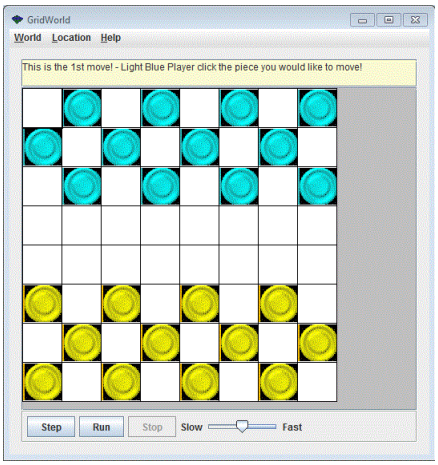
"EXIT" – Completely stops running the program. Is available throughout the program.

"MAIN MENU" – Re-Runs the program.



Once the start button has been pressed, the program proceeds to ask what colors the players' pieces will be. These colors are saved and used throughout the running of the program.





After the colors of both players have been picked, the program asks (by color) which player will go first. The colors that were saved beforehand are used as the fields for the buttons.

Once the colors have been picked, pieces of those colors are instantiated as GridWorld actors, and the GridWorld board is displayed. The pieces that will move first will always be displayed on the upper side of the board.

A separate class was made called CheckerWorld, which was an extension of ActorWorld, a given GridWorld class for actors.

Throughout the game, messages are displayed, such as "This is the 1st move…" here, prompting the users to move or sending them error messages.

Global Variables:

The use of global variables was an integral part of the development of the product. After a piece and the location to move it have been selected, the move is checked for legality. When a move is checked for legality, information is saved to the global variables. Then, if the move is legal, the player has the option of confirming or cancelling the selected move. If the move is confirmed, then global variables are checked to execute any post-move tasks. Therefore, the variables must be saved globally, so that they can be accessed through methods, without having to re-check conditions that were checked before.

```
if((setColor(piece).equals(name2))&&(loc.getRow()==start.getRow()-2)&&(loc.getCol()==start.getCol()-2))//
{
    taken = getGrid().get(new Location(loc.getRow()+1, loc.getCol()+1));                                /
    if(taken==null){return false;}
    if(((Piece)taken).getType().equalsIgnoreCase("King")){return false;}
    if((setType(taken).equalsIgnoreCase("Piece"))&&(!taken.getColor().equals(((Piece)piece).getColor())))
        take = true;
        return true;}
    return false;
}
```

Take is initially filled in isValidisCheckers.

For example, Take is checked for whether the move selected was a capturing move. If so, the captured piece is removed from the board. But this is determined before, in isValidinCheckers, so the variable must be made global.

```
            if(take==true)
            {
                taken = gr.get(getTaken(piece, piece.getLocation(),loc,setType(piece), setColor(piece)));
                out.println(taken+" was taken");
            }
```

Another example is hasClicked. Initially, when a piece and a move have been selected, hasClicked is set to True. If enter is pressed at any time, the selected move is executed, only if, of course, a move has been selected. Therefore, when enter is pressed, hasClicked is checked to determine whether a move has actually been selected.

```
        if(ans.contains("ENTER"))
        {
            if(hasClicked==false)
            {
                loccount=0;
                setMessage("Please only press enter after you have chosen a piece and a move");
                return false;
            }
```

The importance of the Global Strings Name1 and Name2 is that when printed out, C1 and C2 return the r-g-b values, rather than the name of the color. Therefore it is essential to create Name1 and Name2, separately from C1 and C2, to act as the colors names, for output purposes.

Global Arrays:

A global array is used for each player's pieces. Whenever a piece is converted to a king, the Actor at that spot in the array is changed to a king.

```
if((piece instanceof Piece)&&(!((Piece)piece).getType().equalsIgnoreCase("King")))
{
    if((setColor(piece).equals(name1))&&(piece.getLocation().getRow()==7))
    {
        for(int x = 0;x<8;x++)
        {
          if((white[x]!=null)&&(white[x]==piece))
          {
                replace = new Location(piece.getLocation().getRow()+(1-1),piece.getLocation().getCol()+(1-1));
                piece.removeSelfFromGrid();
                king = new King(c1);
                king.putSelfInGrid(gr,replace);
                white[x]=king;
                piece=null;
          }
        }
    }
```

Similarly, whenever a piece is captured, the Actor at that spot in the array is changed to null.

```
taken.removeSelfFromGrid();
for(int x = 0;x<8;x++)
{
    if(taken==white[x]){
        white[x]=null;
        break;
    }
    if(taken==black[x]){
        white[x]=null;
        break;
    }
}
```

Also, the global arrays are referred to in the canTake and canMoveAtAll methods. The methods determine whether the current player can take another piece, or move their pieces, respectively, and since they are supposed to check if the **player** can take or move a piece, the player's entire array of pieces, rather than a specific piece, must be checked.

```
public boolean canTake(Color color)
{
    if(color==c1){
     for(int i = 0;i<white.length;i++){
        for(int x = 0;x<8;x++){
        for(int y = 0;y<8;y++){
            if(white[i]!=null){
            if(white[i].getLocation()!=null){
            if((isValidinCheckers(white[i], white[i].getLocation(), new Location(x,y), setType(white[i]), name1)==true)&&(take==true)){
                System.out.println("The piece can move");
                return true;
            }}}}}}
    if(color==c2)
    {
     for(int i = 0;i<8;i++){
        for(int x = 0;x<8;x++){
        for(int y = 0;y<8;y++){
            if(black[i].getLocation()!=null){
            if((isValidinCheckers(black[i], black[i].getLocation(), new Location(x,y), setType(black[i]), name2)==true)&&(take==true)){
                System.out.println("The piece can move");
                return true;
            }}}}}}

    return false;
}
```

CanTake checks if **any** of the player's pieces can capture a piece.

```
public boolean canMoveAtAll()
{
    if(count%2==0){
        out.println("p1's move");
        for(int x = 0;x<8;x++){
            if(white[x]!=null){
            if(canMove(white[x],name1)==true){
                out.println("The piece can move");
                return true;
            }}}}
    else if(count%2==1){
        out.println("p2's move");
        for(int y = 0;y<8;y++){
            if(black[y]!=null){
            if(canMove(black[y],name2)==true){
                out.println("The piece can move");
                return true;}}}}
    out.println("Done");
    return false;
}
```

CanMoveAtALL checks if **any** of the player's pieces have a possible move.

Matching:

One of the most common conditions checked in the program is if parameters match up with each other. This is usually to ensure that the player is selecting, capturing, etc., the correct color piece.

```
if((count%2==1)&&(piece.getColor()==c1))
{
    piece =null;
    setMessage("It's "+name2+"'s move, not "+name1+"'s!");
    return false;
}
if((count%2==0)&&(piece.getColor()==c2))
{
    piece =null;
    setMessage("It's "+name1+"'s move, not "+name2+"'s!");
    return false;
}
```

Here, the program checks if the player clicked the right color piece. If count is odd, then it is player1's turn, so a piece of color C1 must be clicked, a piece of color C2 if count is even.

```
if(((setColor(piece).equals(name2))&&(loc.getRow()==start.getRow()-1))&&((loc.getCol()==start.getCol()-1)||(loc.getCol()==start.getCol()+1)))
{
```

Here (in isValidinCheckers), the program checks if the piece selected (which is already confirmed to be a regular piece) is of color c2 (or name2), and if so, then the move must be upwards ((loc.getrow=start.getrow-1) or (loc.getrow=start.getrow-2)).

```
if(((setColor(piece).equals(name1))&&(loc.getRow()==start.getRow()+1))&&((loc.getCol()==start.getCol()-1)||(loc.getCol()==start.getCol()+1)))
{
```

Similarly, if the piece is of color c1 (or name1) the move must be downwards ((loc.getrow=start.getrow+1) or (loc.getrow=start.getrow+2)).

A piece is changed to a king when it reaches the side opposite to its starting side. So in this case the color of the piece and its location, once again, must match.

```
if((setColor(piece).equals(name1))&&(piece.getLocation().getRow()==7))
```

If the piece is of c1, it started on rows 0-3, so it must reach row 7.

```
if((setColor(piece).equals(name2)) && (piece.getLocation().getRow() == 0))
```

Similarly, if the piece is of c2, it started on rows 4-7, so it must reach row 0.

Legality of Moves:

In isValidinCheckers, a move is constructed from the piece's starting location (start) and its final location (loc), and this move is checked for legality.

Regular pieces move forwards, and kings move in both directions.

Pieces of color c1 start between rows 0 and 3, so they move downwards, which means the row of loc must be greater than that of start.

```
if(((setColor(piece).equals(name1))&&(loc.getRow()==start.getRow()+1))
```

Pieces of color c2 start between rows 4 and 7, so they move upwards, which means the row of loc must be less than that of start.

```
if(((setColor(piece).equals(name2))&&(loc.getRow()==start.getRow()-1))
```

```
if((piece instanceof King)){
        if(((loc.getRow()==start.getRow()-1)&&((loc.getCol()==start.getCol()-1)||(loc.getCol()==start.getCol()+1)))
        {
            take = false;
            return true;
        }
```

As shown above, when a piece is a king its color is not checked, because color is supposed to determine whether a piece is supposed to move forwards or backwards, but kings can move in both directions.

```
if((loc.getRow()==start.getRow()-2)&&(loc.getCol()==start.getCol()-2))
{
    taken = getGrid().get(new Location(loc.getRow()+1, loc.getCol()+1));
    if(taken==null){return false;}
    if((!taken.getColor().equals(piece.getColor()))){
        take = true;
        return true;}
    return false;
}
```

If the move is a capturing move, the move has to be two upwards or downwards and a piece must be taken. The piece taken, therefore, will be directly in between loc and start. If taken=null or if the color of taken is the same as the color of the piece, the move is illegal.

If the move is not a capturing move, it must be 1 in the correct direction, and no piece is taken.

Word Count: 1102