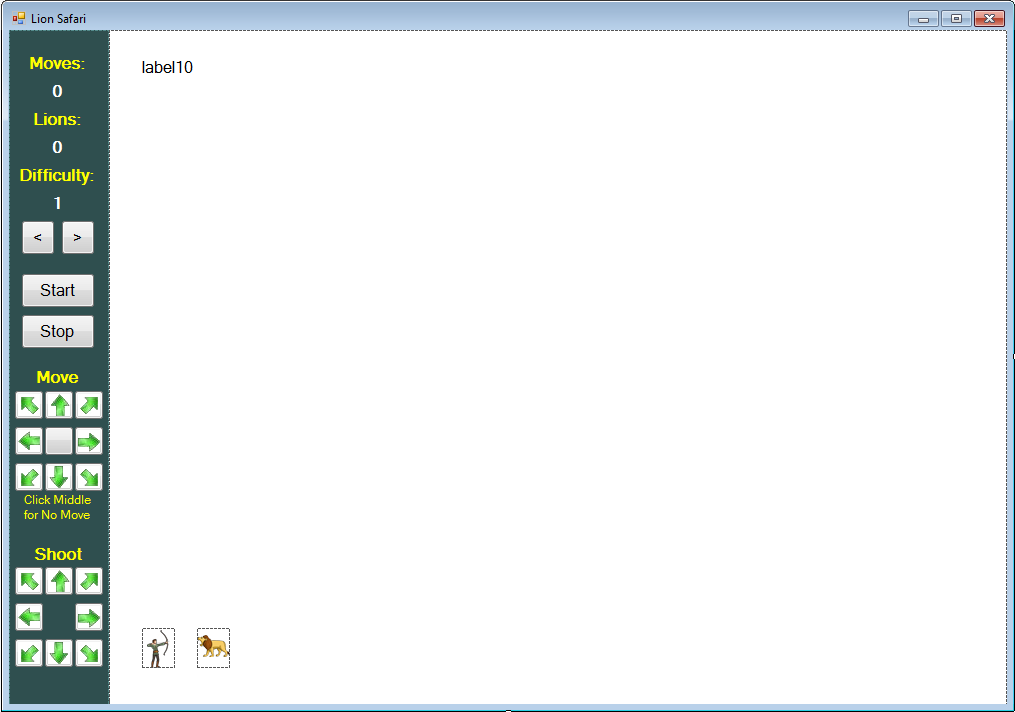
**Lion Safari**

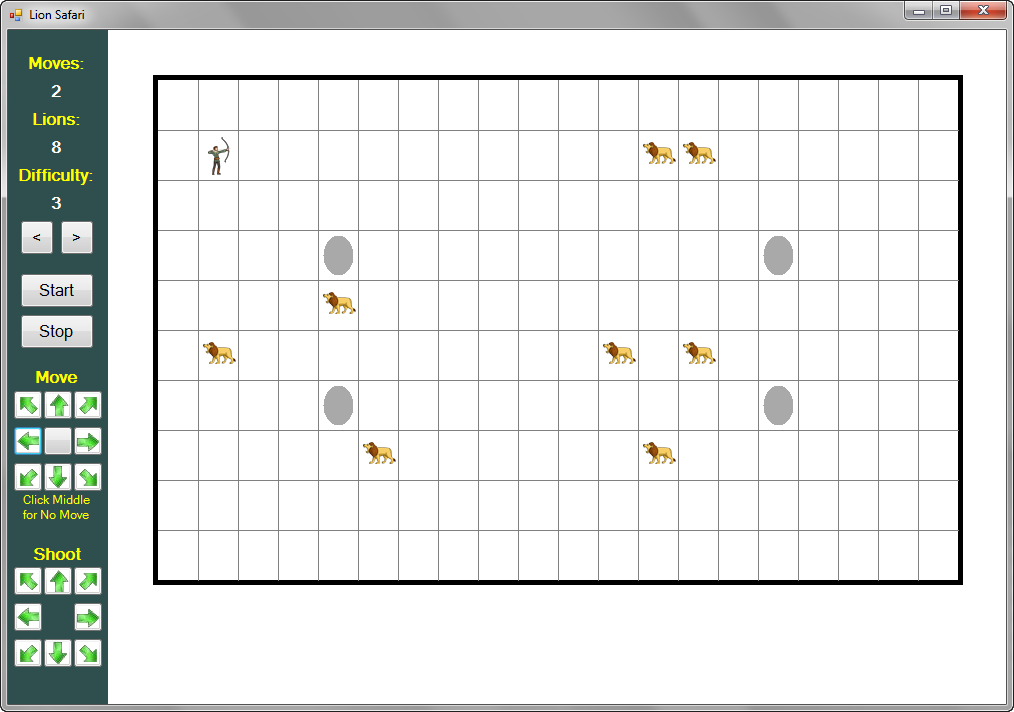
In this chapter, we will build the **Lion Safari** game. The Hunter is placed in a grid with a number of lions (and four rocks). The idea is for The Hunter to dispatch (eliminate) each lion by shooting an arrow at it. The game is over when The Hunter has dispatched all the lions or the moving lions reach The Hunter first.



Labels are used for instructions and to display game information (number of moves, number of lions, difficulty). There are many button controls. Two are used to change the difficulty, one to start the game and one to stop. Nine button controls (with arrows) are used to move The Hunter, while eight buttons (with arrows) are used to shoot arrows at lions. Picture box controls hold images for The Hunter and The Lions.

Run the program (press <**F5**>). The game will appear in its ‘initial’ state. A game description and program instructions are given. At this point, you can click the **Start** button to start the game, click arrow buttons to change the game **Difficulty** or click **Stop** to stop the program.

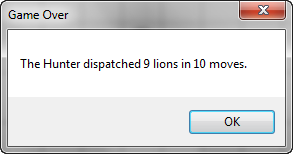
Change the **Difficulty** if you want. The higher the difficulty, the more lions that The Hunter must dispatch. I set the value to **3** by clicking the little arrows under **Difficulty**. Click the **Start** button to start playing. The program instructions disappear. Lions appear in the game grid This is the game’s ‘**HunterMove’** state:



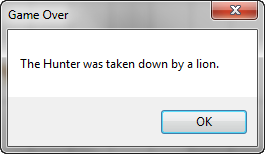
You can see The Hunter in the upper left corner of the grid. A number of lions are in the grid along with four rocks (gray ovals). In the upper left corner you are shown how many **Moves** you have taken and how many **Lions** remain. You have two choices. You can move The Hunter by clicking one of the arrows under **Move** or, you can shoot a arrow at a lion by clicking one of the green arrows under **Shoot**.

If you move The Hunter, he will move one square in the chosen direction (or click the middle to have him remain where he is). Following his move, all the lions will move closer to The Hunter, so be careful. Note lions cannot move through the rocks. If you shoot an arrow, it will travel in the selected direction. If it intercepts a lion, that lion is dispatched (taken out of the grid). An arrow will not move through a rock. The goal of the game is to dispatch all the lions before one reaches The Hunter.

You should see how the game is played. When a lion is in shot range, take a shot. If a lion gets stuck behind a rock, you need to move the hunter until the lion moves in range. At the end, you’ll get a message:



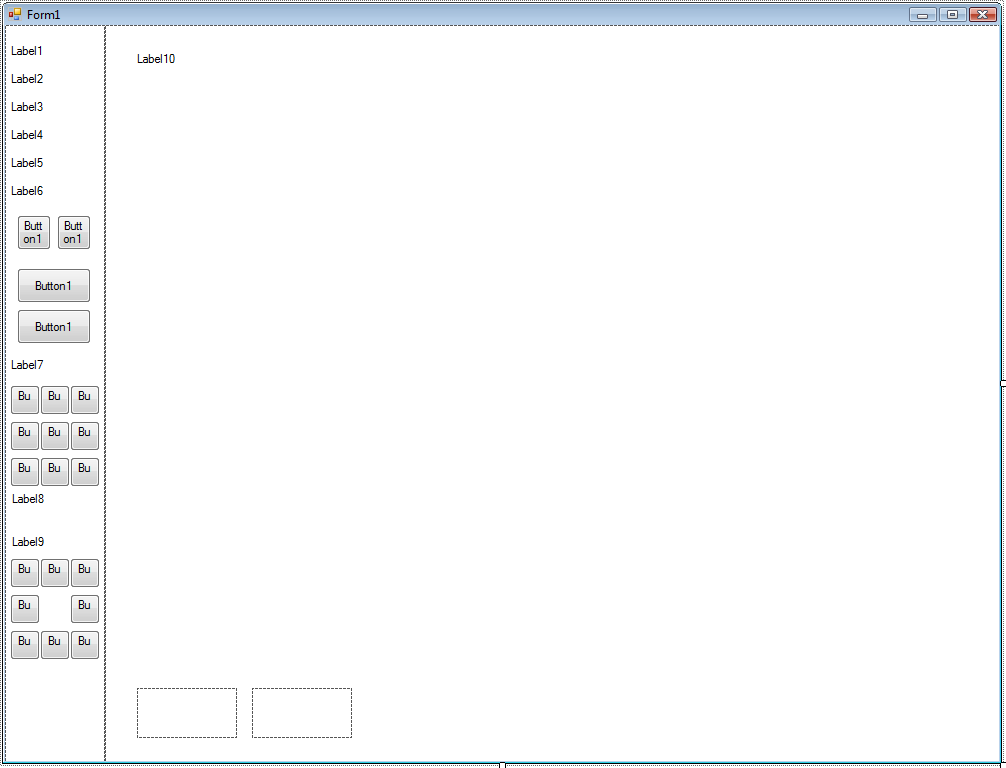
If you aren’t successful in dispatching all the lions, you will see this message:



When a game is over, you can change the **Difficulty**. You can also click **Start** to play again or click **Stop** to stop the program. Continue playing the game to understand its operation.

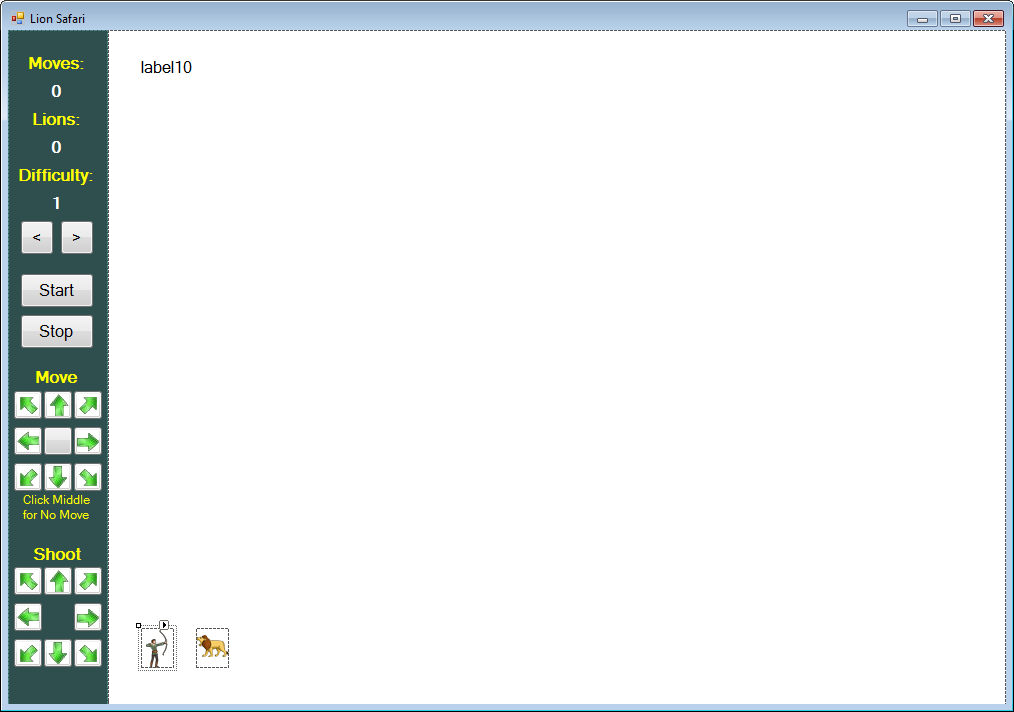
**Lion Safari Form Design**

We begin building the **Lion Safari** game project. Let’s build the form. Start a new project in Visual C# Express. Put two panel controls on the form (one 100 pixels wide on the left, one 900 pixels wide on the right). Put six labels and four buttons at the top of the left panel. Under these put another label, then a 3 x 3 grid of nine small buttons, followed by two labels. Lastly, place another grid of eight small button controls. Put a label control and two picture box controls on the right panel. The form should look like this:



The labels on the left are used to display game information. Two buttons are used to change the difficulty, one to start the game, and one to stop the program. Under the game control buttons are two sets of directional buttons. The first set of nine buttons are used to move The Hunter. The last set of eight buttons are used to fire arrows at the lions. The label control in the second panel is used to display game instructions. The picture boxes hold the images of The Hunter and a lion.

When done, the form should look like this:



We now begin writing code for the **Lion Safari** game. We will write the code in several steps. As a first step, we will write the code that starts the program and establishes its ‘initial’ state. Then, we look at how to go to the state allowing The Hunter to move following clicking of the **Start** button. We then write code to move The Hunter (using mouse), move the lions toward The Hunter and shoot arrow at the lions. During the code development process, recognize you may modify a particular procedure several times before arriving at the finished product.

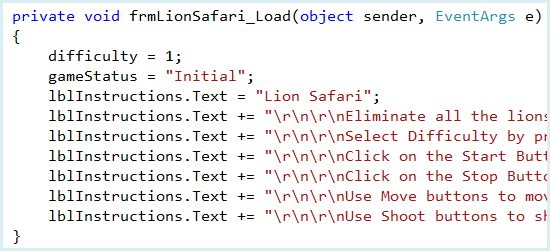
**Code Design – Initial State (Instructions)**

Any time we start a program, there are certain initializations that must take place. Let’s look at the initializations needed in the **Lion Safari** game. When the program first begins, we want to display the game instructions. We also initialize the difficulty (**difficulty**) and the game status (**gameStatus**). These initializations are in the forms **Load** procedure.

We declare two variables (**difficulty** and, **gameStatus**). Do this in the general declarations area:



Add this code to the **frmLionSafari** event procedure:

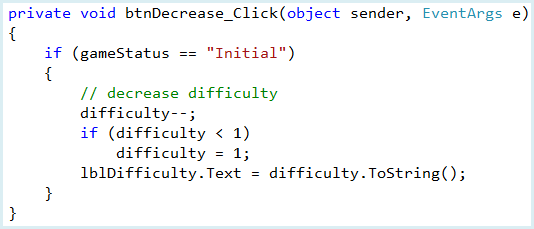


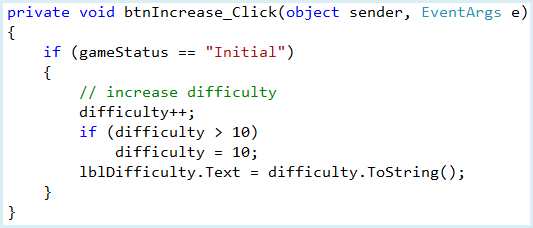
As mentioned, we set **difficulty** to **1** and the **gameStatus** to “**Initial**”. Next, using a series of statements, we write out the game instructions in **lblInstructions**.

**Save** and **Run** the program to see:

**Code Design – Select Difficulty**

The **difficulty** variable is used to determine how many lions are put on the game grid (the higher the difficulty, the more lions). To change the difficulty, a user clicks on either of the two arrow buttons. The code to do this goes in the **btnDecrease** and **btnIncrease** **Click** event procedures:





The code is straightforward. The difficulty can only be changed when the game is in “**Initial**” state. If the **<** button is clicked, we decrease the difficulty by one (minimum of 1). If the **>** button is clicked, we increase the difficulty (maximum of 10). The new value for the difficulty is then displayed in **lblDifficulty**.

**Save** and **Run** the program. Make sure the arrow buttons work

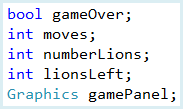
**Code Design – Draw Grid**

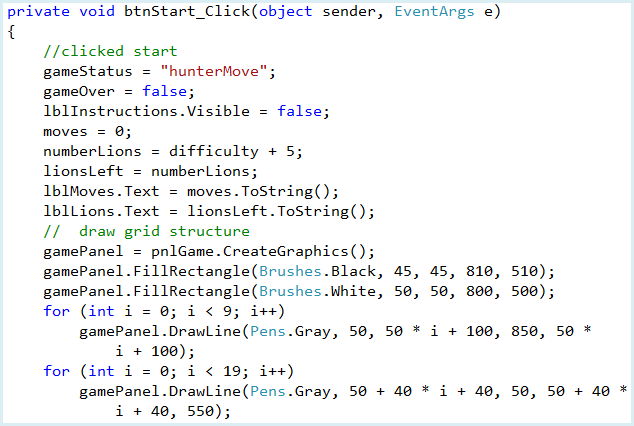
Once the difficulty is set, the next step is to click the **Start** button to allow the game to begin. Hence, we need code to move from the “**Initial**” state to what we’ll call “**HunterMove**” state. When the **Start** button is clicked in “**Initial**” state, we need to take the following steps:

* Change **gameStatus** to “**HunterMove**”
* Initialize variables
* Clear game screen
* Draw game grid with rocks
* Randomly place The Hunter on the grid
* Randomly place lions on the grid

There is a lot to do here, so we will take it in steps. First, we’ll initialize variables and draw the grid.

Several variables are used in game control. **gameOver** is a Boolean variable that tells us if the game has ended – it is initialized at **false**. **moves** (initialized at zero) tells us how many times The Hunter has moved. **numberLions** tells us how many lions are placed in the grid (initialized to **difficulty + 5**) and **lionsLeft** tells us how many lions remain in the grid. The game grid has 10 rows and 20 columns (each grid space 40 pixels wide by 50 pixels high). **gamePanel** is the graphics region we play the game in. Add these general declarations:

Add this code to the **btnStart** **Click** event procedure:



**Q1:** We use two **for** loops when drawing the grid structure. Why is does one have the condition *i < 9* while the other has the condition *i < 19*? **(2 points)**

**Q2:** In the first *gamePanel.DrawLine*, the third parameter reads *50 \* i + 100*.

1. What is this parameter for? **(1 point)**
2. What happens if you change 100 to 200? **(1 point)**
3. What happens if you change 50 to 100? **(1 point)**

You should be able to identify each step in this new code. We clear the area and set variables. The grid structure is established as a white rectangle with a black border. Horizontal lines are drawn every 50 pixels and vertical lines are drawn every 40 pixels.

**Save** and **Run** the program. Choose a difficulty and click. Notice the displayed values for **Moves** and **Lions**. Notice the nice grid structure.

**Code Design –Position Rocks**

Four rocks are placed in the grid. The Hunter can hide behind these rocks if needed for protection from the lions. Elliptical shapes are used for each of the rocks.

Recall the grid has 10 rows (numbered from 0 to 9) and 20 columns (numbered from 0 to 19). We use a two-dimensional array **A** to keep track of what is in each location within the grid. The first index in the array is the **row**, the second is the **column**, so the value at:

A[row, column]

tells us what is in that location. We use this system:

A[row, column] = 0 Nothing is there

A[row, column] = 1 Rock is there

A[row, column] = 2 Lion is there

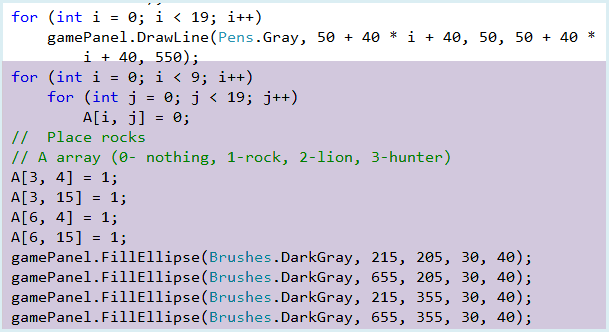
A[row, column] = 3 The Hunter is there

The rocks are in fixed locations: row 3, column 4 and column 15 and row 6, column 4 and column 15.

Add this line to the general declarations area to define the **A** array:



Add the shaded code to the **btnStart Click** event procedure to initialize the **A** array to all zeroes, then position the four rocks on the grid:



**Q3:** In this shaded code, we use nested for loops. Describe why two for loops are required, and what the line ***A[i, j] = 0;***means**. (3 points)**

The (x, y) coordinates used in the **FillEllipse** methods were found by knowing where the grid lines are located (by looking at the code that draws these lines). For example, the first rock is at row 3, column 4. The top left corner at that position is (**210**, **200**).

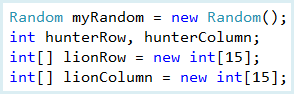
**Save** and **Run** the program. Click **Start**. The rocks will appear:

**Code Design – Position The Hunter and Lions**

The last initialization step is to randomly place The Hunter and the Lions (**numberLions**) onto the grid. The images stored in **picThe Hunter** and **picLion** are are used:

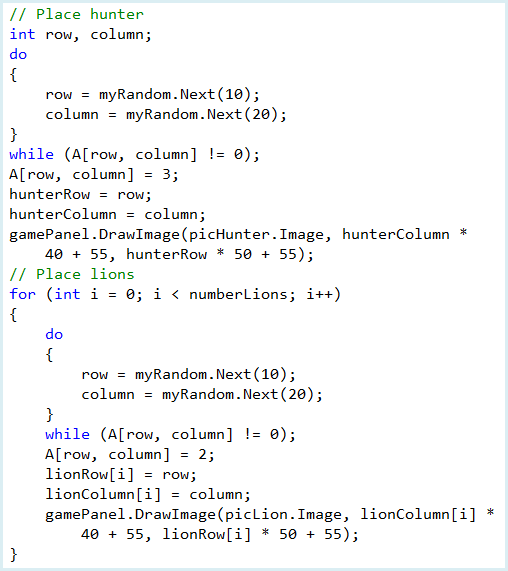
The code to position the Hunter in the grid is similar. We randomly pick a row and column. If nothing is already at that position, put it there. The variables **The HunterRow** and **The HunterColumn** tell us where The Hunter is. The arrays **lionRow** and **lionColumn** tell us where each lion is. Positioning coordinates (x, y) are determined like we did for rock placement.

Add these new variable declarations:



**myRandom** is needed for random number generation.

Add the shaded code to the **btnStart** **Click** event procedure under the place rock code:

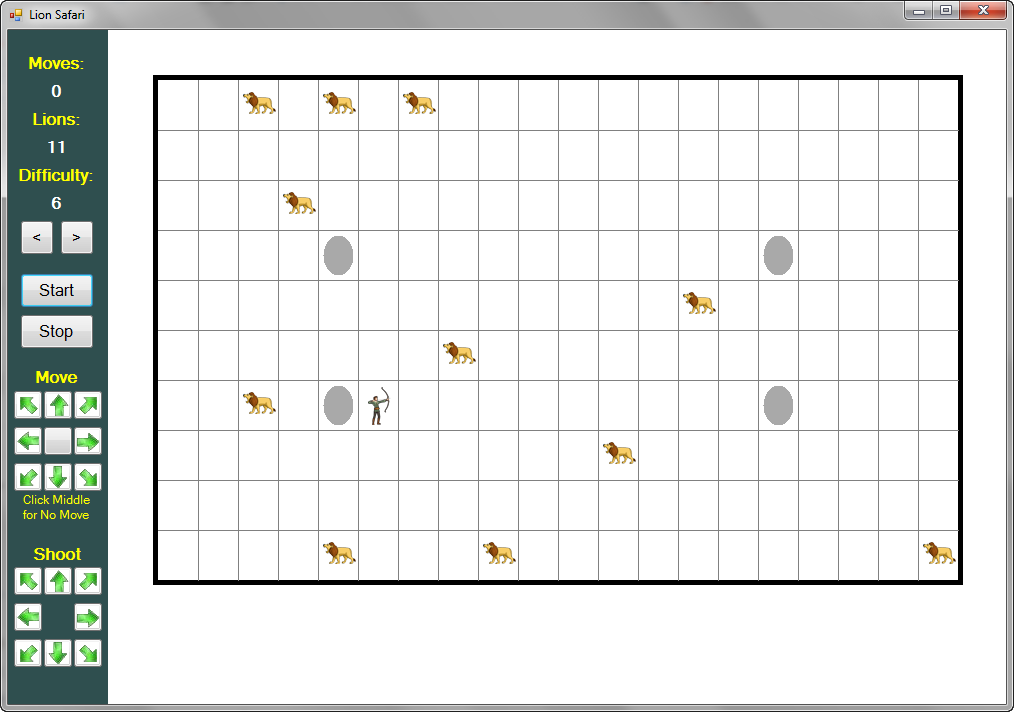


**Q4:** We place the hunter in a *do, while* loop. Explain why this is necessary to avoid any errors on our grid. Specifically, what does the line ***while (A[row, column] != 0);***mean? **(2 points)**

**Q5:** Place a comment next to each line in the **//Place Hunter** section to briefly explain what it does **(9 points)**

**Q6:** We use the same functionality to place the lions on our grid, however unlike the hunter, it is all placed within a for loop. Why? **(2 points)**

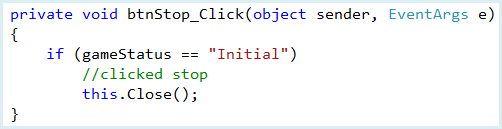
**Save** and **Run** the program. Click **Start**. You should now see the completed initialized game grid.



The next step is to either move The Hunter or shoot an arrow at a lion. We write code to do these steps soon. But first, let’s take care of one last thing you can do while in “**Initial**” state – stop the program.

**Code Design – Stop the Program**

One last thing you can do while the game is in “**Initial**” state is to click on the **Stop** button. This will stop the program. The code here is simple. Add these lines to the **btnStop** **Click** event procedure:



As desired, if the **Stop** button is clicked, the program ends.

**Save** and **Run** the program. Make sure the **Stop** button works. This completes the code for moving from “**Initial**” game status to ‘**The HunterMove**’ status. Now, let’s look at code to move The Hunter around the grid.

**Code Design – Moving The Hunter**

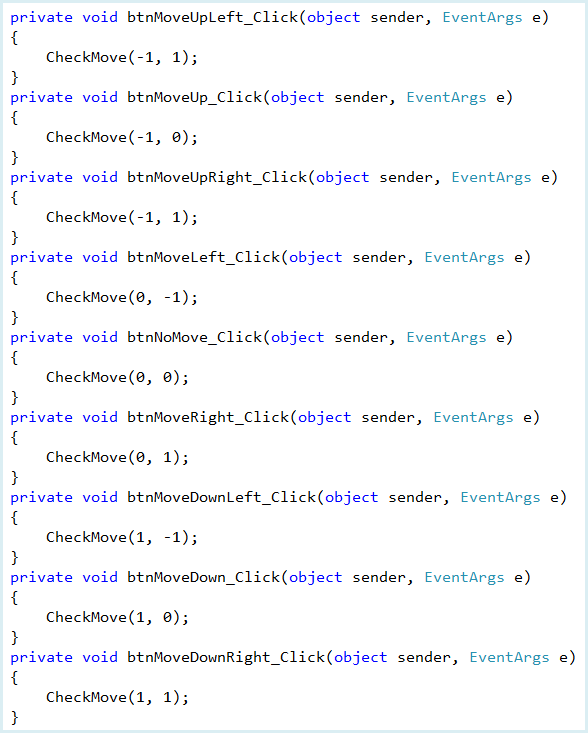
Once the game begins, you move The Hunter around the grid so he can begin shooting arrows at the lions. The Hunter can be moved by using the ‘buttons’ under the **Move** heading on the game screen:



Clicking on any of the arrow buttons will move The Hunter one grid square in the corresponding direction. Clicking the middle button will leave The Hunter at his current position.

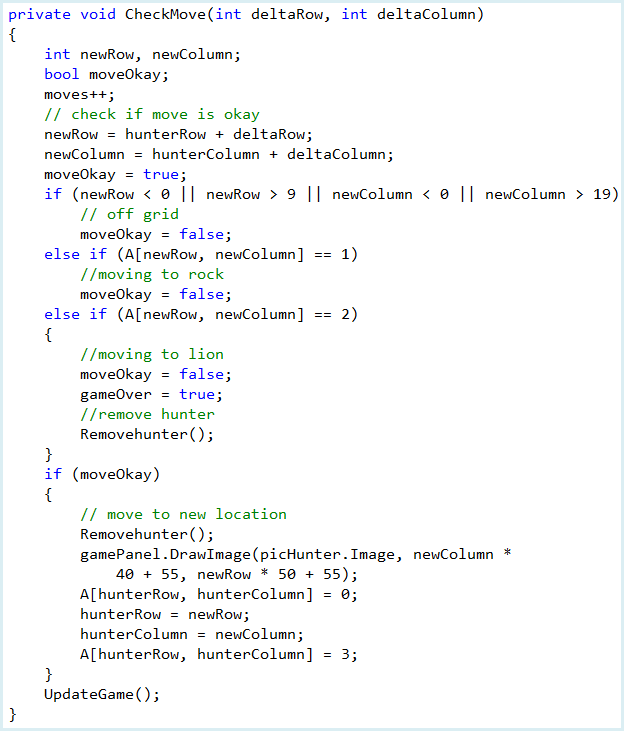
The code to detect clicking on any of these buttons is placed in the corresponding button’s **Click** event procedure. Once a move is made, we need to check to see if it is a valid move. Invalid moves include moving off the top, bottom, left or right side of the grid, moving onto a rock or moving into a lion. If an invalid move is attempted, it is not allowed, and, if a move is into a lion, the game ends!

We use a subroutine **CheckMove** (will be presented in a bit) to see if a move is valid. **CheckMove** requires two pieces of information: **DeltaRow** and **DeltaColumn**. **DeltaRow** tells us how many rows to move The Hunter – it will be either +1 (down one row), 0 (no row change), or -1 (up one row). Similarly, **DeltaColumn** tells us how many columns to move The Hunter - it will be either +1 (right one column), 0 (no column change), or -1 (left one column). You should be able to see that clicking any arrow button in the top row yields **DeltaRow** = - 1, any button in the middle row yields **DeltaRow** = 0 and any button in the bottom row yields **DeltaRow** = +1. Likewise, clicking any button in the left column gives **DeltaColumn** = - 1, any button in the middle column gives **DeltaColumn** = 0 and any button in the right column gives **DeltaColumn** = +1. Using this, we write the nine button **Click** event procedures:



In each procedure, **CheckMove** is called with the appropriate values for **deltaRow** (the first number) and **deltaColumn** (the second number). Now, let’s add **CheckMove**.

Add this subroutine to your program:

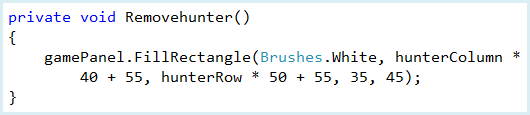


**Q7:** Explain how each condition works to determine that the move is *not* okay (i.e. each condition that leads to **moveOkay = false**) **(3 points)**

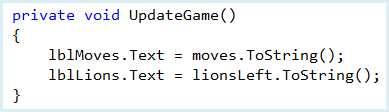
**Q8:** Place comments next to each line under **if (moveOkay)** to briefly explain what it does **(6 points)**.

We increment **moves** and set new values for The Hunter’s row and column (**newRow** and **newColumn**). We then check the validity of the move. If The Hunter moves off the grid, onto a rock, or into a lion, we set **moveOkay** to **false**. If The Hunter moves into a lion, we set **gameOver** to **true** and remove The Hunter from the grid. If we have a valid move, The Hunter is moved to his new location from his old location.

We use this little subroutine (**RemoveThe Hunter**) to clear The Hunter from his current location:



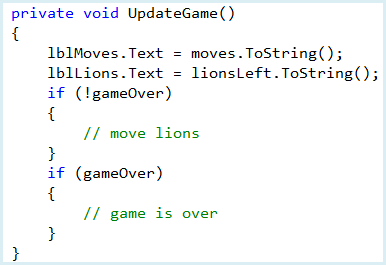
And, notice the last line in **CheckMove** is a call to another subroutine **UpdateGame**. This subroutine is used to change the game status after The Hunter moves or after The Hunter shoots a arrow (that code is added later). For now, **CheckMove** just updates the number of moves and number of lions. Add this subroutine to your code:



**Save** and **Run** the program. Try all the **Move** buttons to make sure they work properly. Try moving off the grid. Try moving onto a rock. Stay away from lions for now – we don’t have logic to end a game yet.

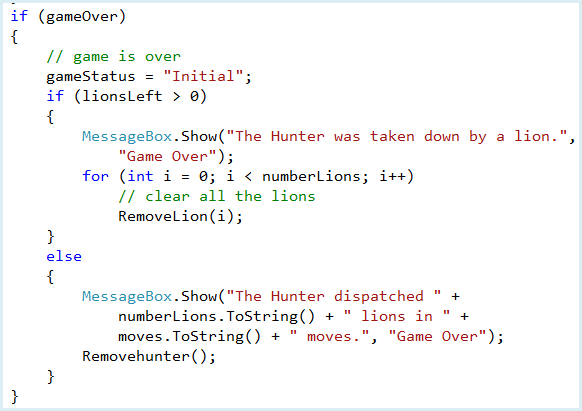
**Code Design – Moving Lions**

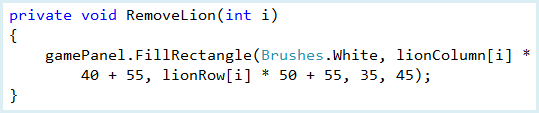
Once The Hunter makes a move, all the lions also make a move – closer to The Hunter! If The Hunter and a lion meet, the game ends. We put this basic logic in **UpdateGame** subroutine. Add the shaded framework:



If **gameOver** is **false**, we move the lions. If **gameOver** is **true**, we end the game. Let’s add the code for both options..

For now, we’ll program the game over logic. When the game ends, we change **gameStatus** to “**Initial**”. If there are lions remaining (**lionsLeft > 0**), The Hunter has met a lion. If no lions remain, The Hunter has dispatched all the lions with arrows. This logic is reflected in the UpdateGame:

This new code needs a subroutine (**RemoveLion**) that removes lion i from its current location:



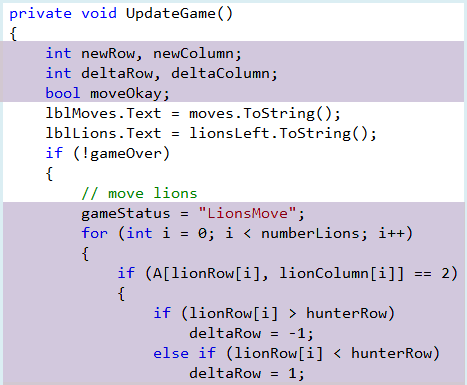
**Save** and **Run** the program. Click **Start**. Move The Hunter around until he meets a lion. You should see a messagebox appear.

You will see a different message if The Hunter dispatches all the lions. Before this can happen, though, we need code to move the lions, then code to shoot the arrows.

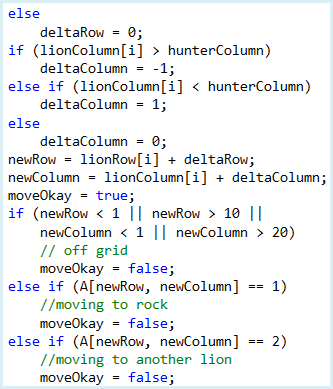
When the lions move, the game is switched to “**LionsMove**” status. The goal is move each lion one grid square closer to The Hunter. So, for each lion remaining, the steps taken are:

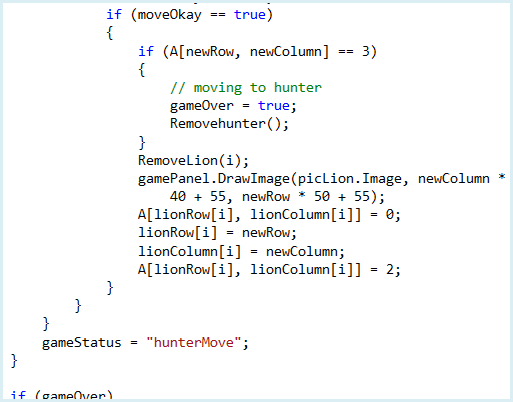
* If The Hunter is above lion, move lion up one grid square. If The Hunter is below lion, move lion down one grid square. If The Hunter and lion are in same row, do not move lion vertically.
* If The Hunter is to left of lion, move lion left one grid square. If The Hunter is to right of lion, move lion righ one grid square. If The Hunter and lion are in same column, do not move lion horizontally.
* Lion cannot move off of grid.
* Lion cannot move into rock or another lion.
* If lion moves into The Hunter, the game ends.
* Once move is complete, change **gameStatus** to “**The HunterMove**”

The code that implements these steps is placed in the top part of the **UpdateGame** subroutine. The code is shaded:



Continue with both of these sections of code:





We declare some new local variables at the top. We cycle through all the lions (**numberLions**) in a **for** loop. If a lion is still on the grid, we determine values for **deltaRow** and **deltaColumn** (how much a lion moves) and the proposed new position (**newRow** and **newColumn**). If it is an acceptable move, the lion is moved by first removing it from its current location, then redrawn at its new location. Notice if the lion moves into The Hunter, **gameOver** is set to **true** and The Hunter is removed from the grid. This will end the game.

**Save** and **Run** the program. See if the Lions move. Keep going until a lion takes you down.

If the Hunter could have shot some arrows, the lions wouldn’t have gotten to him. Let’s add that logic now.

**Code Design – Shooting Arrows**

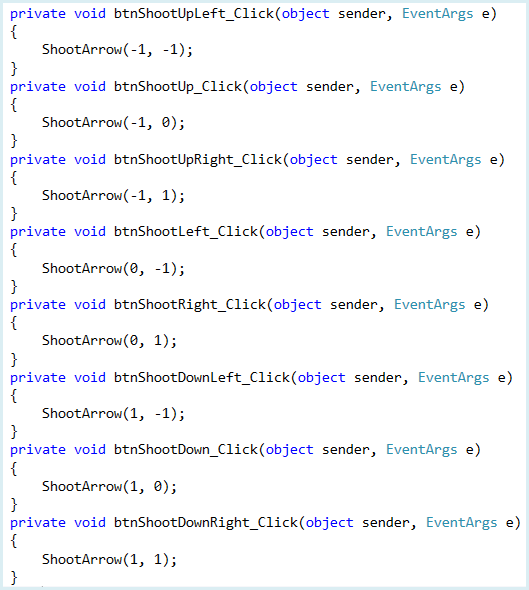
To keep the lions away, The Hunter can ‘shoot’ arrows at the lions to remove them from the grid. Arrows are shot by using the eight buttons under the **Shoot** heading on the game screen:



Clicking on any of the arrows will shoot an arrow in the corresponding direction. If the arrow intercepts a lion, it is removed from the grid. The logic used here is nearly identical to that used for the buttons used to move The Hunter.

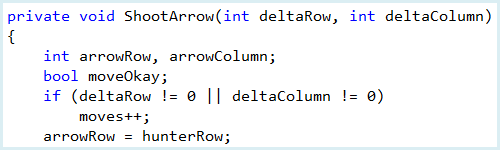
The code to detect clicking on any of these buttons is placed in the corresponding button’s **Click** event procedure. Once a shot is taken, we move in the corresponding direction until we hit a rock, move off the grid or hit a lion. If a lion is hit, that lion is removed. Once all lions are removed, the game ends! Let’s do the code.

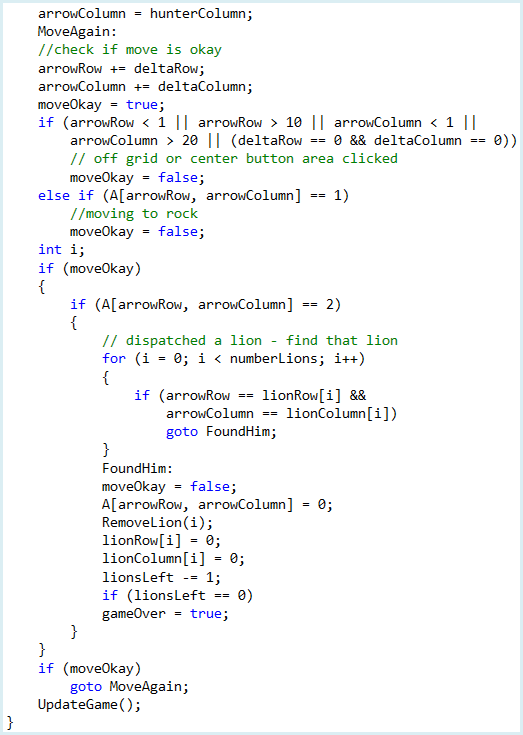
We use a subroutine **ShootArrow** (will be presented in a bit). Like the similar **CheckMove** subroutine, **ShootArrow** requires two pieces of information: **deltaRow** and **deltaColumn**. These tell us the direction of the arrow. **deltaRow** tells us how the vertical direction – it will be either +1 (down), 0 (no vertical motion), or -1 (up). Similarly, **deltaColumn** the horizontal direction - it will be either +1 (right), 0 (no horizontal motion), or -1 (left). You should be able to see that clicking any arrow button in the top row yields **deltaRow** = - 1, either button in the middle row yields **deltaRow** = 0 and any button in the bottom row yields **deltaRow** = +1. Likewise, clicking any button in the left column gives **deltaColumn** = - 1, either button in the middle column gives **deltaColumn** = 0 and any button in the right column gives **deltaColumn** = +1. Using this, we write the eight button **Click** event procedures:



In each procedure, **ShootArrow** is called with the appropriate values for **deltaRow** (the first number) and **deltaColumn** (the second number). Now, let’s add **ShootArrow**.

Add this subroutine to your program:





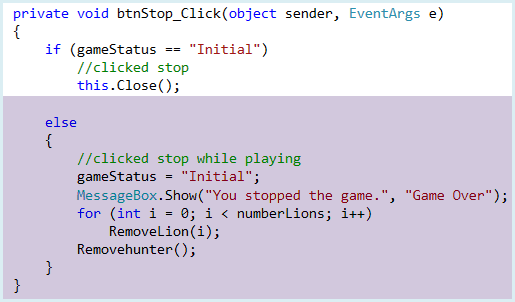
**Q9:** We use a ***goto***statement inside three nested if statements and a for loop. Why do we use this? If you are unfamiliar with ***goto*** statements, try looking them up [here](https://msdn.microsoft.com/en-us/library/13940fs2.aspx). **(2 points)**

**Q10:** Place a comment next to each line underthe **FoundHim** labelto briefly explain what it does. **(9 points)**

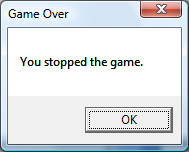
We initialize **arrowRow** and **arrowColumn** to The Hunter’s position (where the arrow starts). From that point on, we change **arrowRow** and **arrowColumn** (by the known **deltaRow** and **deltaColumn** values) until the arrow moves off the grid, hits a rock or hits a lion. If a lion is hit, we determine which lion and remove it from the grid. As a last step, we call **UpdateGame**.

**Code Design – Stop the Game**

One last step and we’re done. You may want to stop a game before dispatching all the lions. To do this, you can click the **Stop** button while the game is in “**The HunterMove**” or “**LionsMove**” status. The code here is simple. Add the few shaded lines to the btnStop Click event procedure:



**Save** and **Run** the program. Click **Start**, then click **Stop**. You should see this message:



All the lions and The Hunter should be removed from the screen. The game is now complete.