Assignment Bonus Battleship AI logic:

1. Develop an AI BattleShip Playing Class.
    a. Name your class with your three initials and the year you graduate, ie cwh1986
    b. Implement the interface Bai2 as described below
    public interface Bai2 {
    public ArrayList placeShips();//ArrayList of Locations and Directions
    public Location guess(); //done each turn
    public void didIHit(boolean flag, boolean sunk, int sizeSunk);
    public Image represent();//return an image of yourself to be displayed while playing
    public void who(String name); //done once
    public void enemyGuess(Location loc); //done each turn
    }
    The ArrayList should consist of 10 objects.
    The 10 objects should be;
    Location of starting point for Carrier (see attached Location class)
    Direction from starting point (see attached Direction class)
    Location of starting point for Battleship (see attached Location class)
    Direction from starting point (see attached Direction class)
    Location of starting point for Destroyer (see attached Location class)
    Direction from starting point (see attached Direction class)
    Location of starting point for Sub (see attached Location class)
    Direction from starting point (see attached Direction class)
    Location of starting point for PT (see attached Location class)
    Direction from starting point (see attached Direction class)

    For example, The following Layout would have this ArrayList
    ……C...
    ……C...
    ……C...
    ……C...
    ……C...
    …….
    …….
    …….
    …….
    …….
    ArrayList
    0 (0,6)
    1 Direction.SOUTH,
    c. Placing ships outside the grid results in instant loss
    d. Placing guesses outside the grid results in instant loss
    e. Going 100 guesses without victory results in instant loss

RUBRIC:

Working game:	35 points
Winning playoff	25 points

Runner Up 15 points

LOCATION

```
public class Location implements Comparable
{
   // Instance Variables: Encapsulated data for each Location object
   private int myRow;           // row location in grid
   private int myCol;           // column location in grid

   /** Constructs a <code>Location</code> object.
    *  @param row    location's row
    *  @param col    location's column
    **/
   public Location(int row, int col)
   {
      myRow = row;
      myCol = col;
   }

  // accessor methods

   /** Returns the row coordinate of this location.
    *  @return       row of this location
    **/
   public int row()
   {
      return myRow;
   }

   /** Returns the column coordinate of this location.
    *  @return       column of this location
    **/
   public int col()
   {
      return myCol;
   }

   /** Indicates whether some other <code>Location</code> object is
    *  "equal to" this one.
    *  @param other    the other location to test
    *  @return     <code>true</code> if <code>other</code> is at the
    *              same row and column as the current location;
    *              <code>false</code> otherwise
    **/
   public boolean equals(Object other)
   {
      if ( ! (other instanceof Location) )
        return false;

      Location otherLoc = (Location) other;
```

```
        return row() == otherLoc.row() && col() == otherLoc.col();
    }

    /** Generates a hash code for this location
     *  (will not be tested on the Advanced Placement exam).
     *  @return     a hash code for a <code>Location</code> object
     **/
    public int hashCode()
    {
      return row() * 3737 + col();
    }

    /** Compares this location to <code>other</code> for ordering.
     *  Returns a negative integer, zero, or a positive integer as this
     *  location is less than, equal to, or greater than <code>other</code>.
     *  Locations are ordered in row-major order.
     *  (Precondition: <code>other</code> is a <code>Location</code> object.)
     *  @param other    the other location to test
     *  @return     a negative integer if this location is less than
     *              <code>other</code>, zero if the two locations are equal,
     *              or a positive integer if this location is greater than
     *              <code>other</code>
     **/
    public int compareTo(Object other)
    {
       Location otherLoc = (Location) other;
       if ( equals(other) )
          return 0;
       if ( row() == otherLoc.row() )
          return col() - otherLoc.col();
       return row() - otherLoc.row();
    }

    /** Represents this location as a string.
     *  @return        a string indicating the row and column of the
     *                 location in (row, col) format
     **/
    public String toString()
    {
       return "(" + row() + ", " + col() + ")";
    }

}
```

DIRECTION

```
public class Direction
{
   // Named constants for some common compass directions
   public static final Direction NORTH = new Direction(0);
   public static final Direction NORTHEAST = new Direction (45);
   public static final Direction EAST = new Direction(90);
   public static final Direction SOUTHEAST = new Direction (135);
   public static final Direction SOUTH = new Direction(180);
   public static final Direction SOUTHWEST = new Direction (225);
   public static final Direction WEST = new Direction(270);
   public static final Direction NORTHWEST = new Direction (315);

   /** Number of degrees in compass
    *  (will not be tested on the Advanced Placement exam).
    **/
   public static final int FULL_CIRCLE = 360;   // not tested on AP exam

   // Array of strings representing common compass points.
   private static final String[] dirNames = {"North", "Northeast", "East", "Southeast",
                                             "South", "Southwest", "West", "Northwest"};

   // Instance Variables: Encapsulated data for each Direction object
   private int dirInDegrees;   // represents compass direction in degrees,
                               // with 0 degrees as North,
                               // 90 degrees as East, etc.


  // constructors

   /** Constructs a default <code>Direction</code> object facing North.
    **/
   public Direction()
   {
      dirInDegrees = 0;    // default to North
   }

   /** Constructs a <code>Direction</code> object.
    *  @param degrees    initial compass direction in degrees
    **/
   public Direction(int degrees)
   {
      dirInDegrees = degrees % FULL_CIRCLE;
      if ( dirInDegrees < 0 )
         dirInDegrees += FULL_CIRCLE;
   }

   /** Constructs a <code>Direction</code> object.
    *  @param str    compass direction specified as a string, e.g. "North"
    *  @throws IllegalArgumentException if string doesn't match a known direction name
```

```
 **/
public Direction(String str)
{
   int regionWidth = FULL_CIRCLE / dirNames.length;

         for ( int k = 0; k < dirNames.length; k++ )
         {
      if ( str.equalsIgnoreCase(dirNames[k]) )
      {
          dirInDegrees = k * regionWidth;
          return;
      }
   }
   throw new IllegalArgumentException("Illegal direction specified: \"" +
                     str + "\"");
}


// accessor methods

 /** Returns this direction value in degrees.
 *  @return  the value of this <code>Direction</code> object in degrees
 **/
public int inDegrees()
{
   return dirInDegrees;
}

 /** Indicates whether some other <code>Direction</code> object
 *  is "equal to" this one.
 *  @param other   the other position to test
 *  @return       <code>true</code> if <code>other</code>
 *                represents the same direction;
 *                <code>false</code> otherwise
 **/
public boolean equals(Object other)
{
   if ( ! (other instanceof Direction) )
      return false;

   Direction d = (Direction) other;
   return inDegrees() == d.inDegrees();
}

 /** Generates a hash code for this direction
 *  (will not be tested on the Advanced Placement exam).
 *  @return     a hash code for a <code>Direction</code> object
 **/
public int hashCode()
{
  return inDegrees();
```

```
}

/** Returns the direction that is a quarter turn
 *  to the right of this <code>Direction</code> object.
 *  @return     the new direction
 **/
public Direction toRight()
{
   return new Direction(dirInDegrees + (FULL_CIRCLE / 4));
}

/** Returns the direction that is <code>deg</code> degrees
 *  to the right of this <code>Direction</code> object.
 *  @param deg    the number of degrees to turn
 *  @return     the new direction
 **/
public Direction toRight(int deg)
{
   return new Direction(dirInDegrees + deg);
}

/** Returns the direction that is a quarter turn
 *  to the left of this <code>Direction</code> object.
 *  @return     the new direction
 **/
public Direction toLeft()
{
   return new Direction(dirInDegrees - (FULL_CIRCLE / 4));
}

/** Returns the direction that is <code>deg</code> degrees
 *  to the left of this <code>Direction</code> object.
 *  @param deg    the number of degrees to turn
 *  @return     the new direction
 **/
public Direction toLeft(int deg)
{
   return new Direction(dirInDegrees - deg);
}

/** Returns the direction that is the reverse of this
 *  <code>Direction</code> object.
 *  @return     the reverse direction
 **/
public Direction reverse()
{
  return new Direction(dirInDegrees + (FULL_CIRCLE / 2));
}

/** Represents this direction as a string.
 *  @return       a string indicating the direction
```

```
 **/
public String toString()
{
   // If the direction is one of the compass points for which we have
   // a name, provide it; otherwise report in degrees.
   int regionWidth = FULL_CIRCLE / dirNames.length;
   if (dirInDegrees % regionWidth == 0)
      return dirNames[dirInDegrees / regionWidth];
   else
      return dirInDegrees + " degrees";
}

/** Rounds this direction to the nearest "cardinal" direction
 *  (will not be tested on the Advanced Placement exam).<br>
 *  The choice of possible cardinal directions depends on the number
 *  of cardinal directions and the starting direction.  For example,
 *  the two cardinal directions starting at NORTH are NORTH and SOUTH.
 *  The two cardinal directions starting at EAST are EAST and WEST.
 *  The four cardinal directions starting at NORTH are NORTH, EAST,
 *  SOUTH, and WEST.  The four cardinal directions starting from
 *  NORTHEAST are NORTHEAST, SOUTHEAST, SOUTHWEST, and NORTHWEST.
 *  (Precondition: 0 < numDirections <= 360)
 *  @param numDirections  the number of "cardinal" directions
 *  @param startingDir    the starting cardinal direction
 *  @return the current direction rounded to a "cardinal" direction
 **/
public Direction roundedDir(int numDirections, Direction startingDir)
{
   // Determine offset of this direction from startingDir.
   int degreesFromStartingDir = dirInDegrees - startingDir.inDegrees();

   // Divide the compass into regions whose width is based on the
   // number of cardinal directions we have.  Then determine how many
   // regions this direction is from the starting direction.
   int regionWidth = FULL_CIRCLE / numDirections;
   int numRegions = Math.round((float)degreesFromStartingDir/regionWidth);

   // Return the "cardinal" direction numRegions regions from the
   // starting direction.
   return startingDir.toRight(numRegions*regionWidth);
}


// methods not tied to any one Direction object

/** Returns a random direction.
 *  @return a direction
 **/
public static Direction randomDirection()
{
   Random randNumGen = RandNumGenerator.getInstance();
```

```
            return new Direction(randNumGen.nextInt(FULL_CIRCLE));
        }
    }
```

Recursion*Linear Search*Binary Search*Marine Biology Case Study*Infix*Postfix*Prefix*nlogn