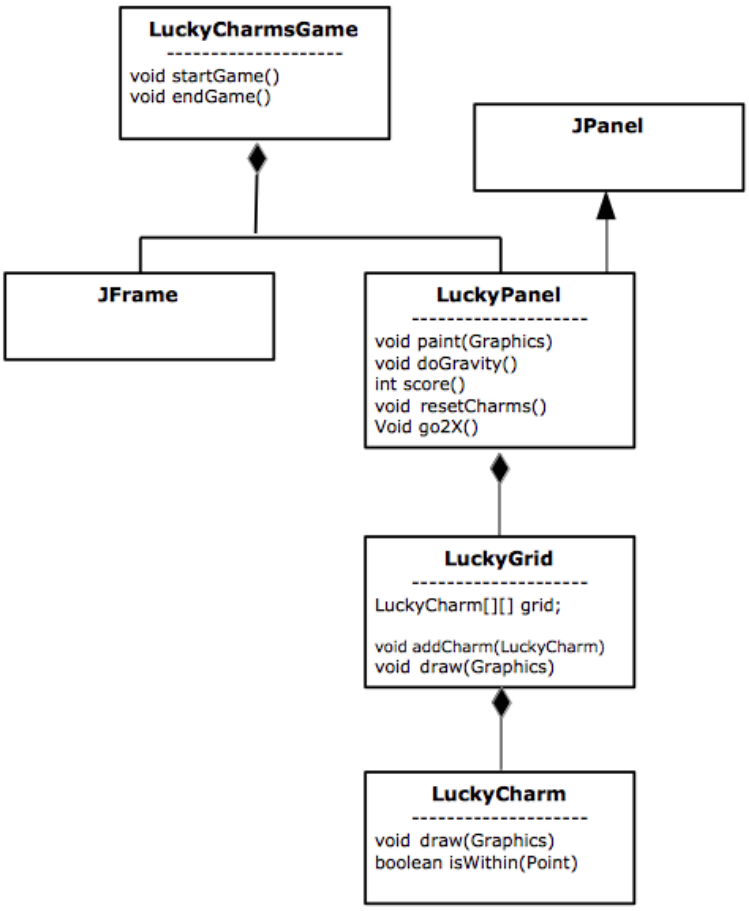
Program #4 - Design Notes

Mar 2015

A. UML Class Diagram

Here's my UML Class Diagram:



Additional diagram notes:

- LuckyCharmsGame adds an ActionListener to its JFrame to handle the menu bar
- LuckyCharmsGame adds a KeyListener to its JFrame to handle gravity and exiting
- LuckyPanel adds an **MouseAdapter** to itself (JPanel) to catch mouse clicks of charms

B. Smart coding!

You can win by chopping your problem into manageable chunks. Divide and conquer your program into simple steps.

Here's an example… the steps to getting to displaying your grid of LuckyCharms:

- Create Program04 class - main(); Hello, world
- Create LuckyCharmsGame class - create empty frame in ctor; show it in startGame()
- Create LuckyPanel - create empty panel; set its size and background color; in LuckyCharmsGame add the panel to your frame
- Draw one 100x100 orange square at (0,0) in your LuckyPanel paint(); draw another green one at (100,0)
- Create LuckyCharm class - replace your hard-coded squares with a charm
- Create LuckyGrid class - create empty 2D array in ctor; create addCharm() method to add charms to the grid; draw() is just a loop to draw each charm in the grid
- And so on...

Remember - Simplify Program #4 by setting the size of your standard game: 5x5 grid, 100 pixels per charm. And then, a 2X game is 10x10, 50 pixels per charm. Do channels (gray space between charms) later.

It's **SO important** to always **be green** with all your files… don't run with errors or warnings.
Create blocks of code (5-15 lines) with an inline comment describing what it does.
Good variable names.
It's OK to do your Javadoc later.
If you're stuck: 1) email me your question, 2) do something that's unrelated and really easy, like the menu bar.

C. LuckyCharm, LuckyGrid

My `LuckyCharm` objects have 3 states: normal, selected and gobbled.
The selected and gobbled states are stored with each `LuckyCharm` as a class variable.
The state of a `LuckyCharm` may alter how he is drawn…I did this by changing the `draw(Graphics)` method in `LuckyCharm`.
I can set and ask a LuckyCharm about its state using methods like:

```
void setGobbled(boolean)
boolean isGobbled()
```

`LuckyGrid` is a simple class. It's basically just a 2D array of LuckyCharm objects. Then, why bother? Why not just slap a 2D array in `LuckyPanel` directly?

`LuckyGrid` is there to make your `LuckyPanel` code smaller and easier to read. Instead of looping through all the `LuckyCharm` object in `LuckyPanel`, add a method to `LuckyGrid` to do it. An example: The score of a game equals the number of gobbled LuckyCharms. Add a method to do this count to your LuckyGrid:

```
int countGobbledCharms() { … }
```

I found many examples in my code where I queried all the charms, so I added the grid and it makes things a lot nicer and cleaner.

D. Selection and growing the selection

Selection is pretty easy. In the MouseAdapter of your LuckyPanel, grab the (x,y) point of the mouse click. Then, pseudo-code for LuckyGrid.selectCharm(Point p)…

```
for( row = 0 to size-1) {
      for( col = 0 to size -1) {
            charm = grid[row][col]
            if( point p is within charm borders) {
                  return charm;
            }
      }
}
```

Once one charm is selected, then you must “grow” the selection to neighboring charms that are adjacent *and* the same color. I’ll present 2 algorithms for you here.

D.1 Growing selection with a Queue

So, we have a charm selected and its color. Go!

```
create a Queue<LuckyCharm>
add the selected charm to the queue
while( queue is not empty) {
      curCharm = remove top of queue
      set curCharm as “selected”
      foreach neighbor of curCharm {
            if( neighbor is not selected and same color) {
                  add neighbor to queue
            }
      }
}
```

D.2. Growing selection with iteration

In this algorithm, we look at all charms in the grid in each iteration. We do this until no more charms are selected in an iteration.

```
mark selected charm as "selected"
while( true) {
      count = grow selection
      if( count == 0) break loop  // stop when no growth
}

int growSelection() {
      growthCount = 0;
      for( row = 0 to size-1) {
            for(col = 0 to size-1) {
                  charm = grid[row][col]
                  for each neighbor of charm {
                        if( neighbor is selected and same color) {
                              mark charm as "selected"
                              increment growthCount
                        }
                  }
            }
      }
      return growthCount
}
```

D.3. Conclusion of growing selection

At the conclusion of either of the algorithms above, you have a grid of charms, some of which are selected. If the number of selected charms >= 3, then gobble them and make them disappear.

E. Gravity

After you've mastered selection, growing your selection, and gobbling charms, you'll want charms to drop to the bottom, one space at a time. Charms fall down one spot if the charm below them has been gobbled.

Two things I can think of:

- To do one step of gravity, start at the bottom of your grid. If you start at the top, you'll bump into a charm again as you go down rows and columns.
- You have a choice when "moving" charms down: 1) move the LuckyCharm object itself, or 2) change the color and gobbled state of charms. I chose the latter because it seemed easier.