CS107 Handout 38
Spring 2007 May 30, 2007

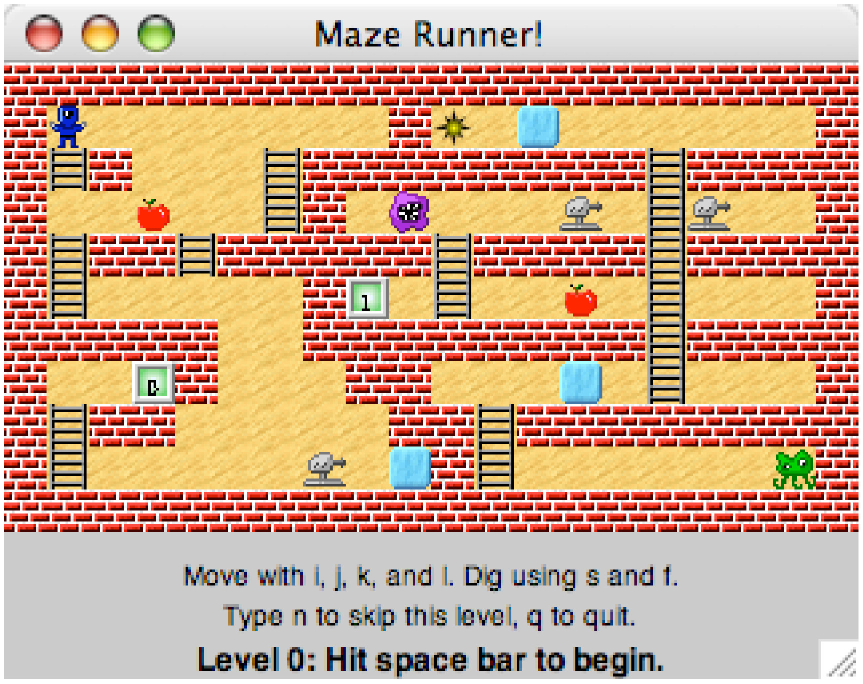
# Assignment 8: Maze Runner

This great assignment was written by Julie Zelenski.

Your final assignment is a chance to explore the wonderful world of object orientation and inheritance using Java. We've tested our solution on Solaris and on Mac OS X boxes, so we're confident it'll work there. You're free to download the most recent version of the jdk, or use Eclipse if you've used that before. PC folks concerned you're being left in the dark can follow Matt Spitz's Cygwin tutorial at **`http://cs107.stanford.edu`** so you can **`ssh`** into an **`elaine`** machine and render a virtual desktop on your own computer. You can work however, wherever using whatever tools you want to, but ultimately you need to port everything back to the elaines, confirm they work there, and submit.

**Due: Wednesday, June 6th at 11:59 p.m.**

You've mastered all things Kevin Bacon and RSS, so now you're ready for the ultimate programming challenge: video games! Your assignment should keep you entertained while giving you experience with designing and implementing objects that use inheritance and messaging to cooperate in one big multi-threaded object-oriented bowl of soup. When finished, it will look something like the following:



Move with i, j, k, and l. Dig using s and f.
Type n to skip this level, q to quit.
Level 0: Hit space bar to begin.

Many thanks to CS107 alum Peter Westen who created the nifty icons you see above.

As a means of gradual introduction, this handout points out some strategies for which classes to tackle first and some development hints that should gently lead you on your way.

**A Cold, Cruel, Cartesian World**

MazeRunner is an invented video game loosely based on arcade games like LodeRunner and PacMan. The maze world is a two-dimensional grid of squares, some are empty, and others contain bricks or freezers or other such things. The world is populated by a number of creatures roaming about. As the game proceeds, each creature runs separately in its own thread and as its gets processor time, a creature looks around the world and can take some action based on its local state of affairs. There is one human player who runs around in the maze, scurrying about, jumping through hypersquares, eating energy apples, and digging through bricks—all under keyboard control by the user. The user's goal is to navigate the maze and get to the goal square before being tagged by any of the bad guys. Each time the human makes it to the goal square, the game advances to a level that all the more treacherous.

**The Squares**

Brick
Hyper
Freezer
Ladder
Goal
Empty w/ apple

- Squares make up the fixed scenery of the world. Each square is either empty, or is occupied by a brick, a hyperjump, a freezer, a ladder, or the goal square. Squares never change for any given level.
- A square knows its location in the maze and is responsible for drawing itself and updating the display as needed when things in that square change. As part of drawing, the square draws the background, then asks each of the occupying creatures to draw themselves on top of that. Of course, it may need to draw an "energy apple" if the square contains one.
- A square can have any number of creatures occupying it at a time.
- A square knows whether or not a creature can enter it. For most squares, a creature can enter the square at any time. A creature cannot usually enter a brick square, but if the bricks have been dug out and have not yet closed up, a creature can enter.
- If a creature tries to enter a square already occupied by some other creatures, it is allowed to join the crowd and the creatures just are drawn one on top of another. However, if a human and an enemy creature try to occupy the same square simultaneously, the enemy will tag the human and the human dies. (Actually, tag is really a gentle description of what really happens. Most humans don't die from being tagged.)
- A square knows whether or not it can support its occupants. If a square doesn't provide support, then, like in the real world, a creature that enters that square will start falling downward until it lands. Ladders always support their occupants (you

can hang on a ladder square no matter what is below you), other squares need to check with the square below to see whether there is support from below. A creature can stand on top of a non-dug-out brick or a ladder. The edge of the board (below the bottommost square) can serve as support.

- Some squares thwart or aid a creature in getting from point A to point B. For example, when a creature enters a freezer square, the creature goes numb for a random amount of time ranging from 1 to 3 seconds. When a creature enters a hyperjump square, it is immediately transported to the next hyperjump square in the maze. The goal square has no interesting behavior when an enemy creature enters it, but if the human player enters the goal square, the human wins that level. Nothing special happens on entering an empty, dug-out brick, or ladder square. Action is taken whether a creature walks, ascends, or falls there. Falling through a freezer square will temporarily pause the creature on its way down. The hyperjump applies even if you climbed a ladder to get there.
- Each hyper jump square is assigned a unique number from 0 to 9 (or less if there are fewer hyper squares in this maze). The numbers identify the sequence for hyperjumping. When a creature enters hyperjump square numbered N, it is transported to hyperjump square N + 1. From the highest number hyperjump square, it wraps around and jumps to hyper square 0. Note that when the transported creature lands in the destination hyperjump square it doesn't hyperjump again. A creature would have to move out of the destination square and back in to trigger a hyperjump to the next square in sequence.
- Brick squares are somewhat unusual. In its normal state, a brick square is filled with bricks and no occupant can enter it. However, the human player can earn the ability to "dig" a hole in a brick square, which removes the bricks and draws it as though it were empty. When the square is in this "dug" state, a creature can enter the square (which is now more like a hole) and go through it. The square stays in the dug state until at least one creature enters and leaves it. If no creature enters the square after it was dug, the square just stays open. While any creature remains in the dug-out square, it will stay open. When the last creature leaves and there are no remaining occupants, the brick closes back up and returns to the normal state.
- Apples—more specifically, energy apples—are the official fruit of Mazerunner. Each maze starts with a few energy apples placed on various empty squares. When any creature enters a square, it takes any existing energy apple from the square. Some creatures may leave apples behind when moving. These tasty apples serve as food for the creatures that given them superpowers described below. A square can hold at most one energy apple at a time. Energy apples never move, die, or do anything interesting. They just sit around waiting for someone to eat them, as that's the only thing they look forward to. They are most easily represented as a **`boolean`** in the square rather than anything more fancy.

## Meet the Creatures

In addition to the largely inanimate squares, there are also a number of animating creatures that roam about in the maze. Each of these creatures runs in its own thread, allowing them all to run about concurrently. Each creature occupies a square and usually just moves one step (in any of the four major directions) at a time. A creature can only move to a square can that can be entered, according to the square constraints noted above.

Each creature occupies a square within the maze and has responsibility for drawing itself when needed (as in when asked by the square as part of drawing its occupants). All creatures have the same basic loop that continues as long as the game level is still in play and while the creature itself hasn't died. Inside the loop, each creature has its own particular strategy to follow about when to move, jump, sleep, etc. described below.

Human Rover Pacer Jumper

**Human** There is always exactly one Human player in the game. The human player has no inherent strategy of its own. Instead, it waits for instructions that come from the real human in the form of key presses. The user can indicate their desire to move in one of the four directions by using the keys 'j' for left, 'k' for down, 'l' for right, and 'i' for up. Whenever a human enters a square that contains an energy apple, it consumes the apple to gain special powers.

The player can also try to dig the square to the left or right. The 's' digs the square southwest of the player, the 'f' digs the square to the southeast. Usually the player cannot dig, but each energy apple they consume gives them the superpower to later dig one brick. Only a Brick square can be dug and it must not already be dug out and the square above it must be enterable and have no occupants (i.e. you can't dig the floor out from under someone else). After a successful dig, the human's energy stash decreases by one.

The human can also use the 'n' key to end this level right now and skip to the next and 'q' to end the entire game. The human's task is to make its way to the goal square on this level without being tagged by any of the enemies or getting trapped somewhere with no way out.

If the user attempts an invalid action, such as entering a non-enterable square or digging a square that is not diggable or when they don't have any energy, you print an error message on the display to point out the errors of their ways.

**Enemy** In general, the enemies exist to make life difficult for the human player. There are three types of enemy creatures: Rover, Pacer, and Jumper. The basic strategy for all enemies is to take a little pause (giving the human an advantage and a fighting chance to get away) and then make some sort of move to chase down the human. The different types of enemy creatures have various ways of trying to go after or annoy the human that are described below. Just like the human, when an enemy creature enters a square with an energy apple, it consumes the apple to gain special powers. After consuming an energy apple, instead of gaining the ability to dig, an enemy goes into a temporarily accelerated mode for the next 50 moves, where its usual pause between moves is cut by a factor of 4. For example, a Rover usually pauses 300 milliseconds between moves, after eating an energy apple it will only pause 75 milliseconds for the next 50 moves, speeding it up for a temporary period.

**Rover** The Rover enemy's strategy is to try to get close to the human player using a fairly brain-dead strategy of moving one square closer each opportunity. A Rover flips a coin to decide whether to try horizontal or vertical first. If trying horizontal, it tries to move sideways one step so that it closer to the column of the human player. If the move is blocked or the Rover is already in the same column as the human, it tries to close in on the human vertically (vice versa if trying horizontal first). If it can't make progress horizontally or vertically, it just randomly chooses a direction to move one step. Rovers are oblivious to the worthlessness of moves that cause them to get repeatedly stuck, fall off ledges, hop up and down, or otherwise appear stupid. Rovers pause 300 milliseconds between moves, which gives the human enough time to out-run them if it keeps moving.

**Pacer** Rather than directly chasing the human like a Rover, the Pacer strategy is to trace out a path back and forth horizontally and hope that it will bump into the human at some point (or at least make it difficult for the human to get past them.) A Pacer will move east as far as it can (stopping when it gets to a dead end or before falling off a ledge) and then turn around and move west as far as it can. It repeatedly paces back and forth like this. Pacers aren't very tough opponents since they move in such a predictable pattern. However, they are much faster than Rovers, they only pause 150 milliseconds between moves, which means you often have to get your timing just right to sneak past one. The Pacer is a little unique in that it has two different images, one drawn when it is walking west, and a different one for east. All other creatures draw using the same image no matter which direction they are moving.

**Jumper** The Jumper's strategy is to be unpredictable. A Jumper pauses a random amount of time (between 3 and 8 seconds) between moves. When moving, it picks a random enterable square on the board and jumps there. The enterable square doesn’t have to be supportable, so it may jump and then fall down to a supported position. Although it makes no direct attempt to go after the player, Jumpers do have the annoying habit of showing up in human's way at the most inopportune moments.

Jumpers are also the source for new energy apples. About 25% of the time when leaving a square to jump to a new location, the Jumper will leave behind an energy apple treat for the next occupant. If Jumper happens itself to jump to a square with an energy apple, it consumes it and not only does it do the usual enemy 4x speedup for the next 50 turns, it also decreases the drop percentage by 5%. That means from here on this Jumper will only leave apples 20% of the time. If it hits up another apple later, the percentage will go down to 15% and so on, until it bottoms out at 0%.

Note that when entering a square, all creatures, no matter what type, are subject to the action the square takes (such as hyper jumping it to a new location or temporarily freezing it). And all creatures are subject to gravity— whenever a creature enters a square that cannot support an occupant, the creature move one square downward once every 200 milliseconds until it comes to rest at a stable location somewhere below.

**What we give you: Location and Grid**

To manage the maze world, you're definitely going to need some facilities for tracking locations (a packaged-up row and column) and some sort of grid object for the board. These sort of simple utility classes aren't too interesting to build, so we just wrote them for you. As your first task, you should look through the code for these two classes and make sure you understand it and how you would use these objects and their methods. They really are pretty simple.

Our **`Location`** object wraps up a row and column into one object and has a few convenience functions to compare locations and find adjacent locations.

Our **`Grid`** class is a general-purpose 2-dimensional template collection indexed by location. A grid can be of any size, and the number of rows and columns are specified as arguments to the constructor. Once set, the number of rows and columns in a grid doesn't change. The grid operations include setting the element at a location, retrieving the element at a location, retrieving the number of rows and columns, checking if a location is in bounds, and choosing an element at random from the grid.

One thing to note about the grid and maze world: location (0,0) is the uppermost left corner. Column numbers increase to the right and row numbers increase as you move down, which might be a little backwards from what you expect.

**Partially implemented: the MazeRunner object**

Next, take a look at the beginnings of the MazeRunner game object we give you. This class is only partially implemented, some of the basic structure is there to get you started, but many of the details are left for you to fill in. You'll want to read the provided code over carefully to understand what is already written and what you will need to contribute to finish off its implementation.

This class is responsible for the overall game data. It keeps track of the grid, the display, and the squares and creatures in the maze world. The game object ties things together and will likely to serve as a bridge between things (such as when a creature wants to get a neighboring square from the grid or needs to check if the game is over).

At the beginning of each level, the MazeRunner class is responsible for reading the starting configuration from a data file, configuring the Grid and Display and then populating the maze with the squares and creatures. All creatures are created in the suspended state, when the level officially begins, the game will message all of the creatures to start them off animating in separate threads.

We've given you the code that reads in the maze data files and draws the world. Here is what a sample maze world file looks like (this is for the world pictured on the first page of the handout):

```
11
20
%%%%%%%%%%%%%%%%%%%%
%H       %! *      %
%#%   #%%%%%%%%#%%%%
%  E  #% J   P #P  %
%#%%#%%%%%#%%%%#%%%%
%#     %1 #  E #   %
%%%%%  %%%%%%%%#%%%%
%  0%   %%   * #   %
%#%%     %%#%%%%%%%%
%#     P *%#      R%
%%%%%%%%%%%%%%%%%%%%
```

The first line tells you the number of rows, the second the number of columns, then the contents of the maze is laid out line by line. The legend is as follows:

| | |
|---|---|
| `%` | brick square |
| `*` | freezer square |
| `#` | ladder square |
| `0-9` | hyperjump square (one number per square, to identify sequence of jumps) |
| `!` | goal square |
| | empty square (space character) |
| `H` | empty square with Human |
| `J` | empty square with Jumper |
| `P` | empty square with Pacer (all Pacers should start facing East, by the way) |
| `R` | empty square with Rover |
| `E` | empty square with Energy Apple |

The code we give you provides the beginnings of a Viewable hierarchy, with partial implementations of the concrete **`Human`**, **`Empty`**, and **`Freezer`** classes. As you code, you'll introduce your squares and creatures to the mix by integrating them into the code hierarchy and updating the **`MazeRunner`** class to create instances of them. The starter code lays down an empty square as a placeholder for character codes it doesn't implement classes for yet. The starter code compiles and runs just fine—the **`Human`** is enemy-less and all-powerful in that it can travel all over the grid. But that'll change as you introduce the other squares and creature types.

The game is also responsible for tracking when the game is running so that the creatures can check on this status and starting and stopping the creature threads at the beginning and end of the levels. Most of the code is already in place, you mostly just need to make the appropriate calls at the right time. For example, when the human makes it to the goal or the human dies, it will call the method to end the current level and advance or quit. If advancing to the next level, the game clears out any state from the current level, reads the next level's maze file, and starts the next run.

**From here: the Squares**

The maze world has several different types of squares, each having some features in common with the others, while other features are unique. A common parent class can be used to capture the common elements, while inheritance and overriding can be used to produce specialized subtypes.

Start by identifying what the various squares have in common: they all know their location in the maze, all track a collection of occupants, all know the image used to draw, and all have operations that allow you display the square, add an occupant, ask if the square can be entered, and so on. This will form the basic behavior in the base square class. Now consider the variations in the operations, too, e.g. a hyperjump square draws a little differently, a freezer square takes its own special action when entered, a ladder can support a creature, a brick can be dug out, etc. These differences will be factored out into specialized square subclasses that override to replace or refine the default behavior.

The **`Square`** classes form a nice exercise in designing a small inheritance tree. The most important part here is learning how to enable code re-use by factoring shared code upwards in the hierarchy. Although many of the Square methods will be quite short (just a line or two), don't give in to the temptation to just copy and paste code between similar methods. Committing yourself to making inheritance work for you on the small things will help you reap the larger benefits on inheritance when designing more complex hierarchies later. Some of the initial design decisions have been made for you: We provide an abstract **`Viewable`** superclass, an abstract **`Square`** subclass, and concrete **`Empty`** and **`Freezer`** classes. You may or may not need to add new methods to these, and you may or may not want to refactor the starter code. What needs to happen will become clear as you implement more classes.

**Finally: the Creatures**

Save the creatures for last since they are the most complicated objects in the game. Like the squares, you need to be very conscious of designing your classes to enable as much code sharing as possible. All creatures occupy a square, all creatures draw, all creatures move, all creatures fall, etc. so these operations should be factored up to the parent and used, overridden, and extended in the subclasses as needed.

We've already started you off with a partial `Creature` hierarchy as well: `Viewable` is subclassed to give you `Creature`, which is further subclassed to give you `Human`. Work on perfecting the `Human` class as well as solidifying the interaction between creatures and the various square types. Does the human freeze when entering a freezer? Fall when jumping off a ledge? Hyperjump to the next square in sequence? Pick up energy apples from the squares? Win when they reach the goal?

After you have the basic human-square interactions working, you are ready to add in the enemy creatures one by one. The `Jumper` and `Pacer` classes are probably a bit easier than the `Rover` so you may want to leave the `Rover`s until last.

**Designing an object hierarchy**

One of your most important jobs is going to be laying out the inheritance tree and figuring out what data and methods go with each object. We encourage you to sketch things out on paper and plan your strategy before writing a lot of code since you want to avoid committing to a design that would be difficult to implement.

When classes are strongly similar, you want to maximize code sharing through inheritance. The different squares and creatures have a lot of behavior in common—the challenge of the assignment is structuring your classes so as to avoid repeating code.

You should also be able to add other types of squares and creatures without a lot of modification. This means you will want many small well-named methods. Some of these methods may have just one or two lines of code, but by factoring into methods of their own, you enable subclasses to override the method and easily extend or change details while still inheriting all the common behavior with no effort.

Where code is not quite the same, you do not want to copy and paste methods from class to class, making small modifications. Instead factor up all the common parts and move the different pieces to small helper methods that can be overridden as needed.

With all common behaviors factored out to the parent class(es), only the extensions and differences appear in the specialized subclasses. The parent class declares the common methods, those that can be reasonably sent to any of the subclass varieties. Whether or not there is a good default implementation will decide whether the method is abstract.

There is a fair amount of latitude in the way you can construct a reasonable set of classes. The most important expectation is that your code uses sensible object-oriented design. Your classes should present a clean abstraction that successfully coordinates with other classes through a reasonable interface. Classes should take responsibility for their own behaviors (displaying, changing status, etc.) rather than being externally manipulated. Code should not be repeated between methods or classes. And so on...

**Drawing and the Display class**

How do all the fancy graphics occur? Well, in true object-oriented fashion, each type of thing that needs to draw (squares, creatures, etc.) should be responsible for messaging the display to draw itself. A square should know how to draw itself and then message any related components that need to draw (i.e. its occupants) and so on.

Since drawing is usually expensive, you want to draw only what is needed. At the beginning of the game, you draw the entire maze once, but after that, you should just re-draw the things that change as creatures move around.

We provide you with the **`Display`** class that gives you a simplified interface for drawing images in a two-dimension grid as well as printing status messages. Due to the magic of Java's Abstract Windowing Toolkit (**`awt`**) the graphics support is **supposed** to work on any JDK-compliant platform— Mac OS X, XWindows, Vista, whatever. We tested the Display class under the JDK on Solaris and on Mac OS X and it worked just fine, which means it probably works elsewhere, but there is no guarantee. If you have problems with it elsewhere, please just come back to the platform where we know it works.

The **`Display`** provides a minimal collection of drawing functionality that is specifically designed to support the maze world. The methods you will need to interact with are:

- The **`configureForSize(numRows, numCols)`** method reconfigures the board and window to accommodate a maze world of the given number of rows and columns. This is usually called once per level when reading the maze file. This is called by the starter code and you won't need to call it anywhere else.
- The **`drawAtLocation(name, location)`** method takes a **`String`** name and a Location object and draws that named thing at that location. The names of known things are "**`Empty`**", "**`Brick`**", "**`Hyper`**", "**`Freezer`**", "**`Goal`**", "**`Human`**", "**`Rover`**", "**`PacerEast`**", "**`PacerWest`**", "**`Jumper`**"", and "**`Energy`**". Each of these corresponds to an image file that is drawn at the specified location. (Debugging tip: when an image can't be found or accessed, it draws as a gray square, check to make sure the files are there and you have the name correctly spelled, etc.) In order to draw the blank background at a location square, you just draw the name "**`Empty`**".
- The **`drawAtLocation(name, char, location)`** method works similarly to the above method, but takes an additional parameter of the character to overlay on top of the image. This is used by the hyperjump squares that want to draw the hyper jump image with an additional label to identify which numbered square it is.
- The **`drawStatusMessage(msg)`** method takes a **`String`** and just prints that message at the bottom of the display window, replacing any previous message. You use the message field for any type of status message you'd like to communicate to

the user during the play of the game such as complaining when they try to move to a non-enterable square, showing updates to the human's energy store, or announcing when the game is won or lost.

- The **`getKeyFromUser()`** method is used to get keystrokes from the user for the human's strategy. It uses an efficient blocking mechanism that doesn't waste processor time, but instead quietly waits until the user types a key and wakes up to return it.

All of these methods are **`static`** methods, so you send them to the **`Display`** class itself. Only one **`Display`** is created and used at a time.

**A few miscellaneous details**

- The starter code (and the code you'll submit) uses one deprecated method in the **`Thread`** class. It's a long story, but it's much easier to use this one deprecated feature at the moment, so you can ignore javac's complaints about it. Here's what a "clean" compile on my Mac says:

  ```
  jerry-cains-computer:~/Development/cs107/java/maze-runner jerry$ make
  javac -g MazeRunner.java
  Note: MazeRunner.java uses or overrides a deprecated API.
  Note: Recompile with -Xlint:deprecation for details.
  ```

  You don't need to worry about this.

- At some point, you may need to convert from a char to an int. You do this the same way you do in C and C++— characters are represented using their ASCII numeric values and can be manipulated arithmetically when needed. The only difference you need to be aware of is that you cannot directly assign an **`int`** to a **`char`** in Java. Since a narrowing conversion truncates and potentially loses information, Java requires an explicit cast for such cases. For example, here is how to convert the character **`'3'`** to the numeric value **`3`** and back.

  ```
  int num;
  char ch;

  ch = '3';
  num = ch - '0';  // subtract ascii values to get distance from '0'
  ch = (char)('0' + 3);
  ```

- Note that if one of the creature threads runs into a fatal problem (such as using a **`null`** object reference or accessing an array out of bounds), only that thread is stopped. The rest of program and the other threads keep going merrily along. So if one of your creatures appears to get stuck as though its thread died, you might want to check the output window to see if a fatal exception was reported for that thread so you know what you need to fix.

- When the game ends (either from quitting, winning or losing), the program will just sit there with the last maze window on-screen. In order to terminate the Java process, you need to click the close box on the window. This action tells Java you are done with the window and it can exit.

- The convention in Java is to name classes starting with a capital letter (**`Grid`**, **`Location`**), and to name variables and methods starting with a lowercase letter and capitalizing any further words (**`numRows`**, **`getLength()`**). Constants (i.e. **`static final`** variables) are usually either capitalized or in all capitals.
- If you are using descriptive variable and method names and taking care to write clean, readable code, you will find there is less motivation for extensive commenting. The control flow may be entirely obvious from the code itself and will not require a lot of additional explanation. Just how much commenting does a method like **`getNumRows`** really need? You should provide class overview comments and specific comments for anything tricky or unusual.
- The UNIX **`javac`** compiler (where we will be compiling and testing your programs) absolutely insists that every class be in its own file. Other more lax compilers may not enforce this requirement, but your classes must be properly separated into files in order to compile and run where we will test. Many of your classes will be quite small and this will lead to a proliferation of short files. This is to be expected and nothing to worry about.

**Don't sweat the details! We mean it!!!**

This handout tries to give the basics of the game and its rules, but there are likely to be small details of the game that aren't completely spelled out. Feel free to just make a reasonable decision about such cases, document your choice, and move on. We're not all that interested in the minutiae of the game and we don't want you to get stressed about it either. This assignment is about learning object-oriented class hierarchies and to that end, we want you to spend your time thinking about sensible class design and learning how to make good use of the Java built-ins, not fussing over game rules. Examples of the kind of things you shouldn't stress about:

- How to choose a random location for the Jumper
- Which creature gets started first
- What happens if you enter the goal square and an enemy is already there (do you win or lose?)
- What happens when an enemy eats a second apple before finishing an earlier one (does it accelerate even more? Continue for more turns? Is the second apple completely ignored?)
- Is the human's energy stash preserved between levels or does it start back at 0?

Just decide for yourself what is easiest to do or what makes the game play best and go with it. Document your choice so we know what to expect. Also, for parameters we give, such as how long enemies pause between turns, how fast you fall, the percentages of apples dropped by the jumper, etc. you are welcome to change values as you like to make the game more playable. The primary point of the assignment is to get you to design class hierarchies. Decisions you make about arbitrary details of the game are unlikely to influence your hierarchy and class design. If no one answer causes a shift in your hierarchy, then the answer doesn't really matter, so just pick one.