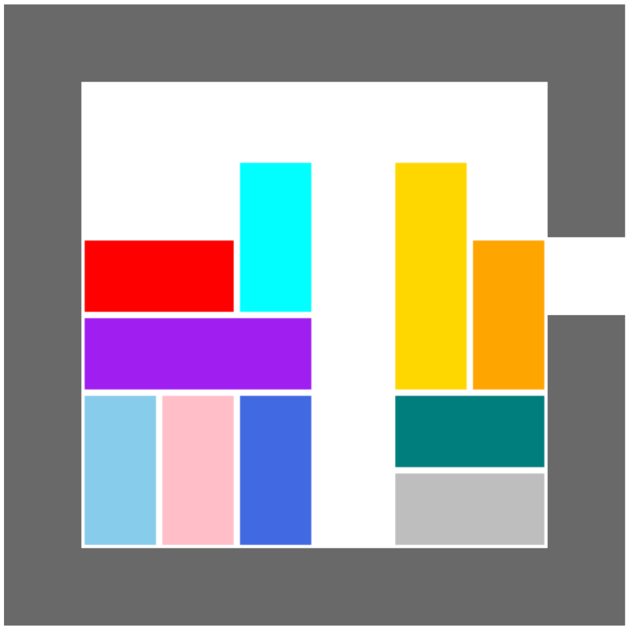
- When evaluating the result, the computer on which this program will run has no support for negative numbers. Design a `NoNegativeResults` visitor that visits an `IArith` and produces a `Boolean` that is `true`, if a negative number is never encountered at any point during its evaluation.

Problem 2: Rush Hour, Part 1

We're going to build *Rush Hour*, a logic/puzzle game invented in the 1970s, over the next few weeks. There are many variations on this game in existence, so we'll be adding features as we go. Adding features may mean that you have to redesign earlier parts of your code, so the cleaner your initial design is, the easier it will be to modify later. Spending time early on careful, clean design will pay off!

Note: This assignment is not written in the same style as other problems; we are not specifying class or interface names for you, or demanding a specific set of method signatures on them. Instead, we are giving you Exercises of the functionality to design and implement this week. Don't try to implement "the whole game" this week, since those functionality requirements haven't been specified yet.

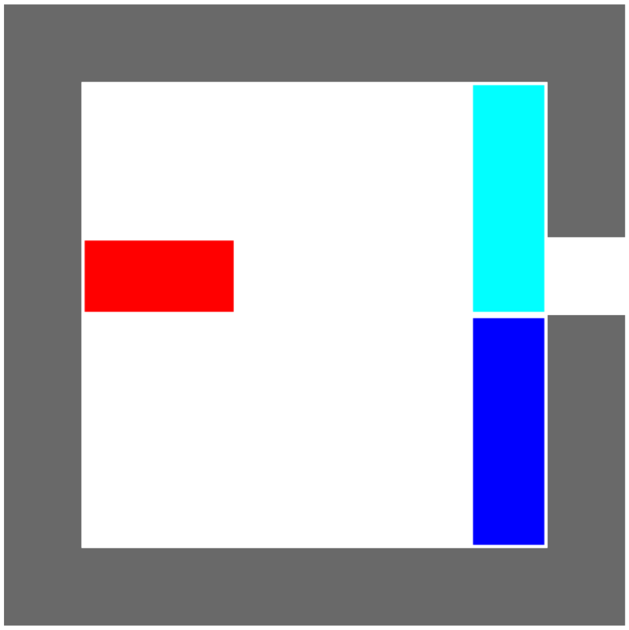
A Rush Hour level is played on a rectangular grid of square cells. For now we'll only consider a few kinds of pieces; we'll add more later. Here is an example board:



The board (currently) consists of cars and trucks stuck in a parking lot. Each car is two squares long; each truck is three squares long. All vehicles can only move forwards or backwards — they can't "change lanes", nor can they turn. As in real life, two vehicles can't share the same space. The gray area shows the boundaries of the parking lot, and the goal of the game is to get the target car (usually shown in red) out of the traffic jam and through the parking lot's exit; no other vehicles can exit the parking lot. You can play a version of this game online at https://www.thinkfun.com/rush-hour-online-play/. Be aware that our game may (currently or eventually) differ in some details from this online version.

In fact, the general case of the game is provably known to be *very* hard indeed...

It's worth pointing out that not all possible levels of the game are solvable; here is a trivial traffic jam:



You do not have to determine if a given game is in fact solvable!

Exercise

Design data definitions sufficient to describe the board state. You will need several helper data definitions. Note that different boards could easily be of different sizes, so your definitions should not limit themselves to a specific size of board. Be sure to follow the design recipe for all of them, and create sufficient examples. Your data definitions will need to accommodate new features in the future, so be careful not to make your designs too inflexible. Make sure your data definitions have clear interpretations, so that readers of your code can understand and use your definitions.

It will be convenient to have an easy way to write down board levels as text.

- The letter `"T"` will represent the start of a vertical truck, and the letter `"t"` will represent the start of a horizontal truck.
- The letter `"C"` will represent the start of a vertical car, and the letter `"c"` will represent the start of a horizontal car.
- The letter `"X"` will represent the exit.
- For visibility, the characters `"+"`, `"-"` and `"|"` will be used to indicate the walls of the parking lot. (They all mean the same thing, but it's useful to have all three for ASCII-art purposes, as visible below.)
- Linebreaks `"\n"` separate rows within the level.
- Spaces `" "` indicate nothing is in a given cell.

So for instance, the first level above can be written as

```
String demoLevel =
        "+------+\n" +
        "|      |\n" +
        "|  C T |\n" +
        "|c    CX\n" +
        "|t     |\n" +
        "|CCC c |\n" +
        "|    c |\n" +
        "+------+";
```

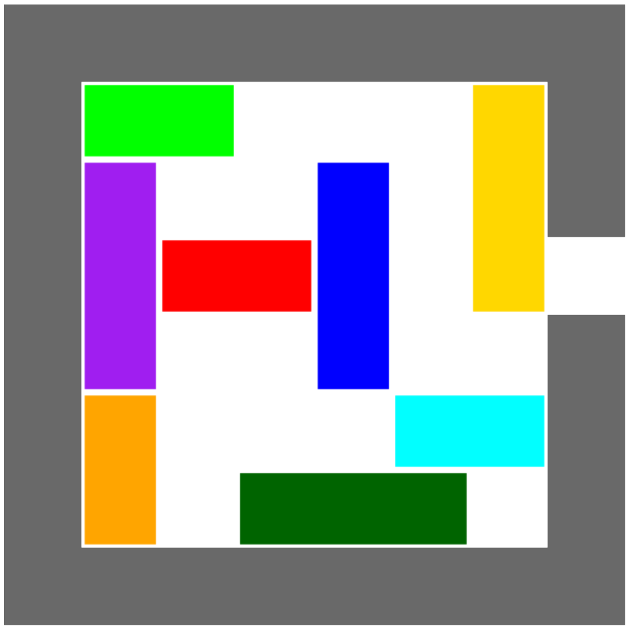
(Note that this string doesn't by itself provide enough information to indicate which is the target car.)

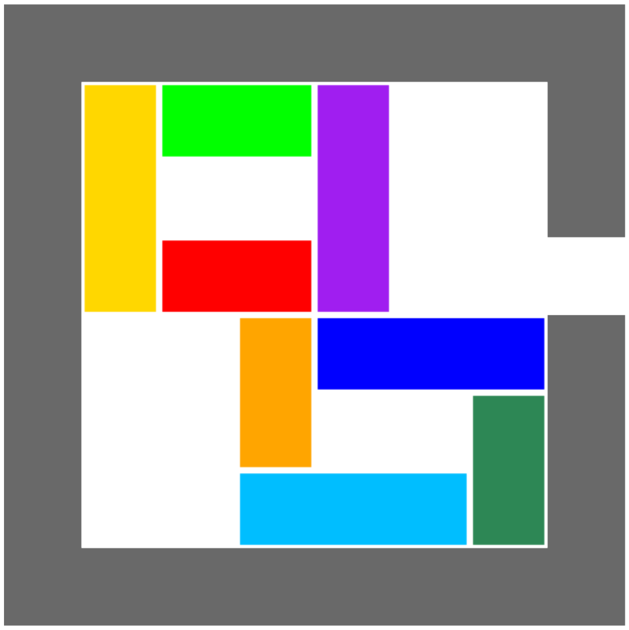
> **Exercise**
>
> Design a class to represent a Rush Hour level. One of the constructors of this class should take in a level-description string like the one above (plus additional information, if you think it's necessary), and use it to configure the level — it will likely be much easier than trying to write out the `IList` of contents by hand. (*Reminder:* not all strings describe valid levels! You do not need to determine if a level is solvable, but you should double-check that the string doesn't describe a logically-invalid level.)

Hint: There are many plausible ways to represent your board. Don't assume that your first-draft idea is the best or only way to do it!

Test this constructor carefully on a few examples, and then you can use this constructor to help build your other examples and other test cases. You will likely want to build a utility class, with a method on it that takes the string above and produces the list of game items in it... You will likely need some helper methods, and likely need to use the `substring` method of `String` to examine the contents of the string.

Here are some other levels that may be fun to play. In each of them, the red car is the one that must escape:





Assignment 7: Mutable Worlds

Goals: Further practice with `ArrayLists`; mutable worlds.

Instructions

This assignment is long. Start early.

As always, be very careful with your naming conventions.

The submissions will be organized as follows:

- **Homework 7 Problem 1:** Rush Hour, part 2.
- **Homework 7 Problem 2:** The `Automata.java` file.

Due Date: Tuesday, March 12th at 9:00pm

Problem 1: Rush Hour, Part 2

In this assignment, you'll add enough behavior to Rush Hour to make it a playable game. Plus, you'll add a new extension to the game so far.

7.1 Design reflection:

You and your new partner each have an implementation of part 1 of the project. You will *merge* the best parts of your two designs together, as the starting point for this part of the project. You are allowed to revise any part of your prior designs: you can change data definitions, implementations, etc. You may change from Funworld to Impworld (see problem 2 below). You may use mutation...

...carefully! You are still expected to *design* your code well, meaning it should still be testable, readable, and understandable. Gratuitous use of mutability, untested `void` methods, etc., will be considered *bad* design. You now have lots of tools to express yourselves: part of the challenge of this problem is to express your design *clearly*.

Exercise

In a file `Merging.txt`, describe the similarities and differences between your two implementations. Explain which parts of whose implementations were kept (and why), and which parts were discarded (and why). Explain what primary changes you plan to make to your prior designs.

Exercise

Update your `Design.txt` file to describe the *current design* that you're now using.

7.2 Design cleanup:

You should likely have noticed that cars and trucks are nearly identical in terms of their behavior. Revise your merged code as needed to eliminate as much duplicate code as possible.

You likely had one of two representations for the space your vehicles occupy:

- Either a `Posn` representing one of the corners, as well as a `width` and a `height`,
- Or a `Posn` representing one corner and a second `Posn` representing the diagonally-opposite corner.

You might have consolidated these representation choices into an `Area` class, such that every vehicle has an `Area` field.

> **Exercise**
>
> If you have not done so already, you should do so: create a class to represent a rectangular area of the board, and move into that class all the arithmetic for computing overlaps between areas. You will likely want to add more methods to this new class, in support of the functionality below. **Make sure** that regardless of which representation choice you made above, you *document clearly* whether the coordinates represent the cells or their corners, and whether the area is *open*, *closed*, or *semi-open* – review Lecture 22: ArrayLists for why these distinctions matter.

7.3 Player motion:

In the previous part of the game, you implemented the ability for players to select and deselect vehicles in the game. Now, you should add the ability to play the game:

- A player can use the arrow keys to move the selected vehicle one cell.
- Vertical vehicles will ignore being moved sideways; horizontal vehicles will ignore being moved up or down.
- The player cannot move a vehicle such that it would overlap another vehicle or wall.
- The game should end and the player should win if the vehicle makes it through the exit.

> **Exercise**
>
> Implement any methods needed to produce the above functionality and make the game playable. Update your `UserGuide.txt` file to more-fully explain how a player might play your game.

7.4 Obstacles:

While parking lots are often just empty wide-open expanses of pavement, there are usually obstacles that make driving trickier. (You can think of these as traffic cones, or potholes, or just as pillars in a parking garage.)

Exercise

Extend your design from part 1 to support placing obstacles anywhere within the game area. These 1x1-cell obstacles should prevent vehicles from driving through them in any direction. (Hint: do you also need to update the string-to-level convenience constructor? Or does it/could it already work with this new requirement?)

Exercise

In a file `Changes.txt`, explain:

- What changes did you have to make to your data definitions from the previous assignment, and why?
- Had you known in the prior assignment that obstacles would be arriving now, would you have designed your data differently?
- If we were to enhance the game with new kinds of items in the next assignment, would your revised data definitions accommodate those changes more easily?

7.5 Irregularly shaped levels:

Not all parking lots are rectangular.

Exercise

Design two example levels whose playable area is not a straightforward rectangle. In your `Design.txt` file, explain what new implementation support, if any, you needed to build to make this work.

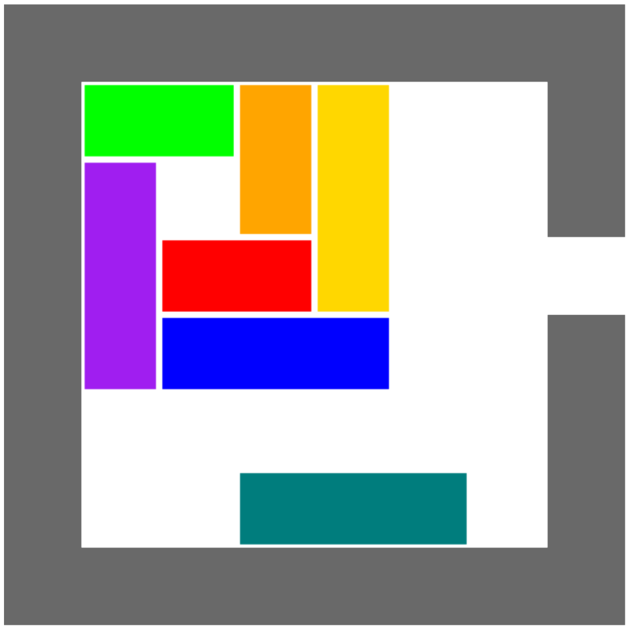
7.6 Submission notes:

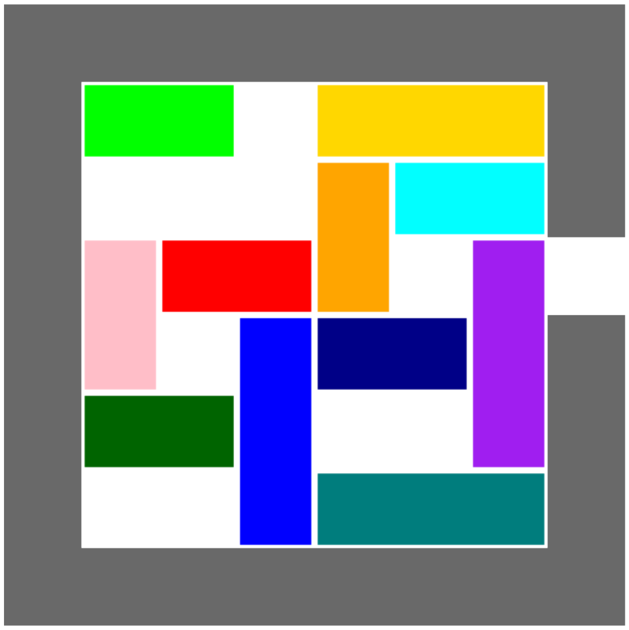
Include all the code needed for your game, and any image files you might be using.

Include the `Merging.txt`, `Design.txt`, `Changes.txt` and `UserGuide.txt` files.

Problem 2: Cellular Automata and Mutable World Programs

A *cellular automaton* is a simplified model of a biological cell: it has a *state*, a short lifespan, and can produce a child cell in some state in response to its own state and to the state of its neighbors. When a collection of cellular automata are arranged in some fashion (in a line, or on a plane, or in a 3-d grid...), interesting collective behaviors can emerge. In this problem, we're going to work with the simplest kind of cellular automaton, which has only two states (on or off), and we're only going to





Exercise

Design a method on your class to render it as an image. You do not need to reproduce exactly the renderings above, but your level should be recognizable.

Exercise

Design a method to detect if two vehicles are overlapping. (*Hint:* Consider the simpler case of *one dimensional* intervals (l, h), where l is the lower endpoint of the interval and h is the higer endpoint. To check if two intervals (l_1, h_1) and (l_2, h_2) are overlapping, you should check whether either l_1 is between l_2 and h_2, or l_2 is between l_1 and h_1. Another, more concise way to phrase this is whether the *larger* of l_1 and l_2 is less than the *smaller* of h_1 and h_2. Vehicles are *two-dimensional*, in that they take up width and height, so you should generalize the idea above somehow.)

Exercise

Design a method to detect if the player has won the level, by getting the target vehicle to the exit.

Exercise

Design a class that subclasses `javalib.funworld.World`, that contains a starting Rush Hour level. Design a mouse-click method (read the documentation, or refer back to Lab 1) that detects when the player clicks on a vehicle, and draws that vehicle highlighted yellow instead of its usual color. Clicking on another vehicle should switch the selection; clicking on a cell that has no vehicle in it should deselect the currently-selected vehicle.

Hint: It *likely* makes sense to have two separate classes: one that subclasses `World` and handles mouse-clicks and keypresses and such, and one that represents your game logic. Combining these into a single class risks making that class bloated and hard to read. It is not *absolutely necessary* to have this split, but it likely will simplify your code.

Submission notes

Don’t make the game overly elaborate; we are more interested in your program’s design than your graphic design!

You may, if you wish, separate out different classes into separate files. Make sure you submit them all, though.

Include in your submission two additional files:

- `UserGuide.txt` should be a short file describing how to play the game (so far! – you’ll update this in future parts as more of the game gets implemented).
- `Design.txt` should describe each of the interfaces and classes in your design, and what they all do. If you think it’s helpful, include an ASCII-art class diagram to show the relationships between the classes.

Be sure to include all the needed files — source file(s), documentation, etc — or else we won’t be able to run your game. You should submit the files as a single `.zip` file. To make a zip file, follow these instructions: open Explorer or Finder, select all the relevant files, right-click and choose “Add to zip” or “Compress” or some similarly-named menu item. *Do not include* the `out/` or `target/` directories where Eclipse puts your compiled code, or any of the `.class` files; they are not necessary.

A note about randomness

There are two ways to generate random numbers in Java. The easiest is to use `Math.random()`, which generates a `double` between 0 (inclusive) and 1 (exclusive). You can multiply this number by some integer to make it bigger, then coerce to an `int` to produce a random integer in the range you wish. However, this is not easily testable: you'll get different random values every time.

The better way to generate random numbers is: First, `import java.util.Random` at the top of your file. Next, create a `new Random()` object, and use its `nextInt(int maxVal)` method, which will give you a random integer between zero (inclusive) and `maxVal` (exclusive).

> This is known as a "pseudorandom number generator", since the numbers aren't really random if they can be reliably repeated...

The reason this is better is because there is a second constructor, `new Random(int initialSeed)`, which has the property that every time you create a `Random` with the same initial seed, it will generate the same "random" numbers in the same order every time your program runs. You should therefore design your world classes with two constructors:

- One of them, to be used for testing, should take in a `Random` object whose seed value you specify. This way your game will be utterly predictable every single time you test it.
- The second constructor should *not* take in a `Random` object, but should call the other constructor, and pass along a really random object:

  ```
  import java.util.Random;

  class YourWorld {
    Random rand
    // The constructor for use in "real" games
    YourWorld() { this(new Random()); }
    // The constructor for use in testing, with a specified Random object
    YourWorld(Random rand) { this.rand = rand; ... }
  }
  ```

  Now, your tests can be predictable while your game can still be random, and the rest of your code doesn't need to change at all.

Problem 2: Rush Hour, Part 3

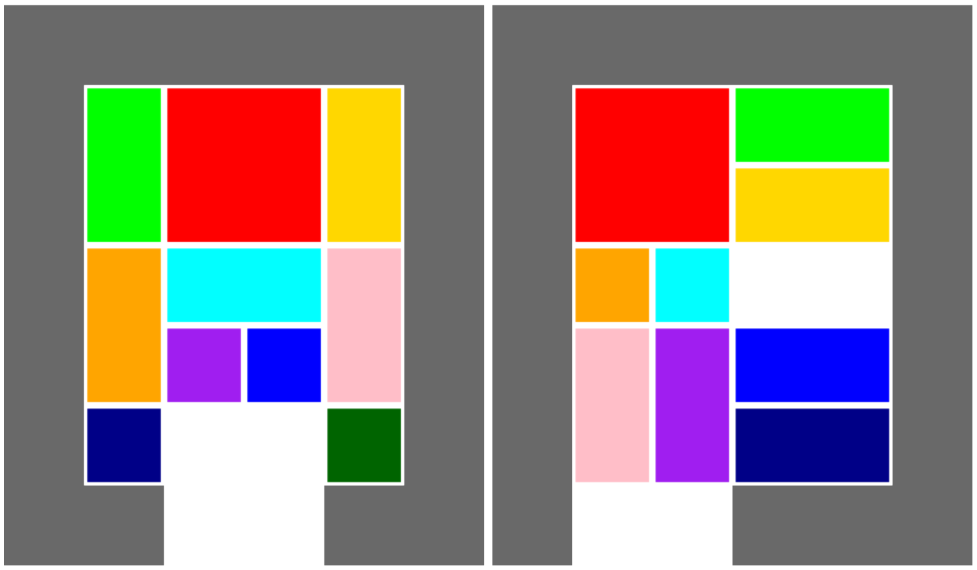
In the previous part of Rush Hour, you implemented player movement rules, added obstacles and irregularly-shaped levels. In this week's exciting conclusion, you'll create a variation on the game, and add a way to fix mistakes.

8.1 Klotski

The general mechanic of "moving rectangular tiles in straight lines and trying to get a goal piece out the exit" has come up in lots of varieties over time. This week, you will *extend* your implementation to support Klotski, whose rules are similar to Rush Hour but with a few variations:

- Tiles can be 1x1, 1x2 or 2x2 cells large.
- Tiles can move in all four directions: left, right, up and down. They cannot turn or move diagonally.
- The goal is to get the 2x2 square out the exit.
- No other tiles can be removed from the exit. (This would make the game somewhat trivial!)

Here are two Klotski boards to get you started:



In both of these levels, the goal is to get the red square out the exit.

To enter these worlds as strings, extend your string-parsing such that

- `"."` represents a 1x1 piece
- `"S"` represents a 2x2 piece

NOTE: you will *still* need to support standard Rush Hour rules! You must be able to start a Rush Hour level and play by Rush Hour rules (where pieces can only move in one dimension) as well as be able to start a Klotski level and play by Klotski rules (as described above).

Exercise

Extend your implementation of Rush Hour to now support Klotski as well. Hint: You will likely need at least two `World` subclasses, to represent your distinct rulesets. Be sure to avoid as much code duplication as possible: your code will still be graded on its design and clarity.

8.2 Keeping score

Exercise

Add a move-counter to keep track of the number of moves a player has made. A *move* should count how many pieces are selected and moved —merely selecting and deselecting a piece should not count as a move, and multiple consecutive moves of the same piece should count as only one move.

Show the score somewhere on screen. Lower scores are better!

8.3 Fixing mistakes:

Exercise

It's far too easy to make mistakes in this game, and it's reasonable for a player to try out some ideas and then want to "go back" to a previous board state. Add an undo feature, that lets the player type `"U"` in order to rewind the game by one step. (Note that each undo should *increase* the player's score! Undoing a bad move should *not* also rewind their score to something better.) Make sure that the game does not crash if the player tries to undo when there are not currently any moves to undo.

Have fun!

You can find a few more levels at https://josephpetitti.com/klotski. You might also try designing a few levels of your own – see if you can combine both Rush Hour mechanics (like trucks and obstacles and irregular board shapes) and Klotski rules, or Klotski shapes (1x1 and 2x2 pieces) with Rush Hour rules.

8.4 Submission notes:

As with prior parts, you should submit an updated `UserGuide.txt` and `Design.txt` that describe how a user should play your game, and how a developer should read and understand your code. Make sure you submit all the necessary source and image files that your graders need in order for them to run your games.

There will likely be a *self-eval* questionnaire the day after this assignment is due. Much like the *peer* evaluations asked you to review each other's code, this self-evaluation will ask you to justify your design choices against your `Design.txt` explanations, and to guide your graders to understand where to find particular test cases, examples, or other implementation choices.