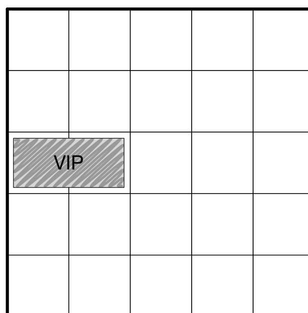
Solutions to Quiz 1 (October 24, 2018)

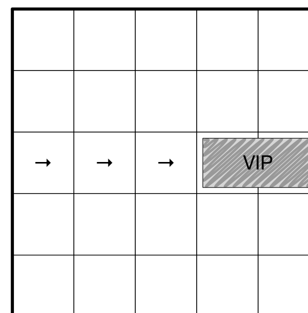
For this quiz, *Mini Rush Hour* is a sliding block puzzle played on a 5×5 grid of cells numbered 1 to 25:

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

← The board has an *exit* to the right of cell 15.

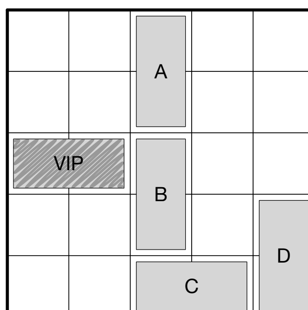
Cars are 1-cell × 2-cells pieces placed horizontally or vertically on the grid. Every Mini Rush Hour puzzle starts with the *VIP car* on cells 11 and 12. The goal is to move the VIP car to the right so it reaches cells 14 and 15, at which point it can exit the board:





← Winning!

Blocking the VIP car's way are other cars, for example:



While they are labeled in this diagram, cars other than the VIP car are not differentiated except by their placement on the board, they have no distinguishing characteristics. All cars in the game move only along their axis of orientation: horizontal cars move left and right, vertical cars move up and down. Cars never overlap. They can only move through and stop on empty cells.

The player wins by making a series of moves that allow the VIP car to reach the exit. The following page shows a solution to this example puzzle, plus another example.

The problems in this quiz refer to the code for two different Mini Rush Hour ADTs, `RushHourPuzzle` on page 6 and `RushHourGame` on page 7, which you may detach.

Notes which are not relevant to this quiz:

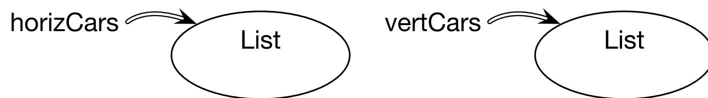
- *Rush Hour*® is a trademark of ThinkFun, Inc. The full game uses varying-size cars on a larger board.
- On arbitrarily large grids, *Rush Hour* and *Size-2 Rush Hour* are PSPACE-complete (Flake & Baum, 2002; Tromp & Cilibrasi, 2005).

Problem 1 (AFs & RIs) (**18 points**).
Immutable `RushHourPuzzle` represents the starting layout of a Mini Rush Hour puzzle.

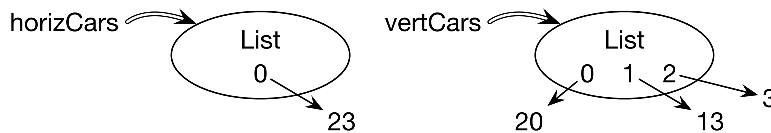
Ben proposes the following rep for this type:

```
private final List<Integer> horizCars;
private final List<Integer> vertCars;
```

He will use empty lists to represent the minimal starting puzzle, with just the VIP car in cells 11 and 12:



And here is how he intends to represent the example puzzle from page 1:



Help construct the abstraction function and rep invariant for this implementation.

(a) Write a concise but complete abstraction function for this rep, consistent with Ben's examples and with your (partial) rep invariant below.

Solution.

AF(`horizCars`, `vertCars`) = the Mini Rush Hour puzzle with the VIP car in cells 11 & 12,
horizontal cars in cells i & $i+1$ for all i in `horizCars`,
and vertical cars in cells j & $j+5$ for all j in `vertCars` ■

In each box below, write one good part of the rep invariant that is **required** for your chosen abstraction function. These statements alone do **not** need to combine to form the complete rep invariant.

(b) Assumed in 6.031:

Solution. `horizCars` and `vertCars` are not `null` and do not contain `null`. ■

(c) One statement required for your AF that involves only `horizCars`:

Solution. For example, $i \in$ `horizCars` $\Rightarrow i+1 \notin$ `horizCars`; or, $\forall i \in$ `horizCars`, $i \% 5 \neq 0$. ■

(d) One statement required for your AF that relates `horizCars` and `vertCars`:

Solution. For example, $i \in$ `horizCars` $\Rightarrow i \notin$ `vertCars`. ■

Problem 2 (ADTs) (**32 points**).
Ben is trying to implement equality for immutable `RushHourPuzzle` using a `sameValue(..)` helper:

```
  private boolean sameValue(RushHourPuzzle that) {
      return horizCars.equals(that.horizCars) && vertCars.equals(that.vertCars);
  }
```

(a) Unfortunately, this implementation requires a stronger rep invariant in order to work. Use Python list notation to give a plausible example of two different reps, containing **as few cars as possible**, where Ben's implementation would return the wrong result:

Solution. For example, `horizCars = [1, 3]` vs. `[3, 1]`, with `vertCars = []` in both. ■

(b) Ben does not want to strengthen the RI, and plans to fix `sameValue` instead. While he does that, implement a valid `hashCode` that will also work without strengthening the RI. **Do not return a constant.**

```
@Override public int hashCode() {

}
```

Solution. For example, return `horizCars.size()`, or the sum of values in `horizCars` and/or `vertCars`.

■

(c) For each of these operations of `RushHourPuzzle`, complete its type signature, and write what kind of operation it is in our taxonomy of ADT operations.

	inputs		outputs	kind
withHorizontal :		→		is a
difficulty :		→		is a

Solution.

`withHorizontal` : `RushHourPuzzle` $\times$ `int` $\rightarrow$ `RushHourPuzzle`, producer
`difficulty` : `RushHourPuzzle` $\rightarrow$ `int`, observer ■

Alyssa proposes a different rep for `RushHourPuzzle`, using the `Direction` type at the top of page 7:

```
  private final Map<Integer, Direction> cars;
```

(d) What do you know about this rep that provides safety from rep exposure?

Solution. `cars` is private (and `final`), `Integer` and `Direction` are immutable. ■

(e) What one additional assumption about this rep completes the SRE argument, without any reference to methods or their signatures? (Partial credit: complete the argument with an assumption about the methods.)

Solution. Assume the `Map` is immutable.

Otherwise, assume creators that take in a `Map` copy it, and other operations that return the map or related mutable types (for example, the `Set` of keys) use copies or immutable wrappers. ■

Problem 3 (Specs I) (**20 points**).
Alyssa realizes that the spec for `withHorizontal(..)` does not account for attempts to add an overlapping car (`withVertical` has a similar problem, and both may have other problems).

Given the following changes, where the rest of the spec is unchanged in each:

- Pick **one** best solution to the specific problem of overlapping cars in `withHorizontal`. Circle "YES" and explain in one sentence what properties make it the best solution.
- For every other option, circle "NO" and explain in one sentence why it is not a good solution.

(a)

```
* ...
* @param left requires 1 <= left <= 25, and adding left to horizCars does not
*             violate the RI above
* ...
```

Solution. NO.

The rep invariant is internal to the implementation, it is not part of the specification. ■

(b)

```
* Make a Mini Rush Hour puzzle identical to this but with a horizontal car as
*   close as possible to cell left.
* ...
* @return this puzzle with an additional horizontal car in left and left+1 if
*         possible, or in the nearest empty 2-cell-wide space otherwise
```

Solution. NO.

The underdetermined postcondition is both hard for clients to use and, since the board may have no empty 2-cell-wide spaces, cannot always be satisfied. ■

(c)

```
* ...
* @return this puzzle with an additional horizontal car in left and left+1
* @throws OverlapException if another car occupies cells left or left+1
*/
```

Solution. YES.

The spec is appropriately deterministic and handles the exceptional case with an (unchecked) exception. ■

(d)

```
* ...
* @return true if and only if left and left+1 were empty and the car was added
*/
public boolean withHorizontal(int left)
```

Solution. NO.

`RushHourPuzzle` is immutable, so this producer must return the new puzzle. ■

Problem 4 (Specs II) (**15 points**).
Mutable `RushHourGame` (on page 7) allows the client to play a Mini Rush Hour game, starting from an initial `RushHourPuzzle` and making a series of moves with the `move(..)` method.

For each of the following changes to the spec of `move`, where the rest of the spec is unchanged in each:

- Circle "STRONGER," "WEAKER," or "INCOMPARABLE" to indicate whether the new spec is stronger than, weaker than, or not comparable to the original spec of `move`.
- Explain why in one sentence that mentions the **pre- and postconditions** and uses them to draw a conclusion.

(a)
```
/**
 * Move the car that occupies cellNum one cell in the given direction, if possible.
 * ...
```

Solution. INCOMPARABLE.

The precondition is unchanged, and the postcondition requires different outputs for the same inputs. ■

(b)
```
* ...
* @param cellNum indicates car to move, requires 1 <= cellNum <= 25
* ...
* @throws IllegalArgumentException if no car occupies cellNum
*/
```

Solution. STRONGER.

The precondition is weaker, and the postcondition is the same for inputs that satisfy the original stronger precondition. Any implementation of this spec will also satisfy the original spec, where the behavior for inputs that do not indicate a car is undefined. ■

(c)
```
* ...
* @param direction direction to move, must be LEFT/RIGHT or UP/DOWN when the car
*                  is horizontal or vertical, respectively
* ...
```

Solution. WEAKER.

The precondition is stronger, and the postcondition is unchanged. Any implementation of the old spec will also satisfy this spec. ■

Problem 5 (Testing) (**15 points**).
Using the specs for mutable `RushHourGame` on page 7, start devising a testing strategy for `move(..)`.

Each of your partitions below should divide the space into 2 parts.

(a) Write one correct and useful 2-part partitioning of the input space on input `this` alone:

Solution. For example:

The game puzzle is solvable / unsolvable. The game is won / not won.
The game has the VIP car plus no other cars / plus 1 or more other cars. ■

(b) Write one correct and useful 2-part partitioning on only `this` and `cellNum`:

Solution. For example:

The car in `cellNum` is horizontal / vertical.
The car in `cellNum` can move in some direction / cannot move in any direction. ■

(c) Write one correct and useful 2-part partitioning on all of the inputs together:

Solution. For example:

The car in `cellNum` cannot move any cells / can move one or more cells in `direction`.
The move is game-winning / not game-winning. ■

You may detach this page. Write your username at the top, and hand in all pages when you leave.

```
1  /** Immutable starting layout of a Mini Rush Hour puzzle. */
2  public class RushHourPuzzle {

3      // ... rep ...

4      /**
5       * Make a Mini Rush Hour puzzle with the VIP car in the starting location
6       *   and no other cars on the board.
7       */
8      public RushHourPuzzle() { ... }

9      /**
10      * Make a Mini Rush Hour puzzle identical to this but with a horizontal car
11      *   whose left half is in cell left.
12      * @param left  requires 1 <= left <= 25
13      * @return this puzzle with an additional horizontal car in left and left+1
14      */
15     public RushHourPuzzle withHorizontal(int left) { ... }

16     /**
17      * Make a Mini Rush Hour puzzle identical to this but with a vertical car
18      *   whose top half is in cell top.
19      * @param top  requires 1 <= top <= 25
20      * @return this puzzle with an additional vertical car in top and top+5
21      */
22     public RushHourPuzzle withVertical(int top) { ... }

23     /**
24      * @return the difficulty of this puzzle, measured as the minimum number of
25      *         moves required to solve it; or -1 if this puzzle is unsolvable
26      */
27     public int difficulty() { ... }

28     /**
29      * @return a string representation of this puzzle that describes all the
30      *         cars on the board
31      */
32     @Override public String toString() { ... }

33     // ... other operations ...
   }
```

```java
public enum Direction { LEFT, RIGHT, UP, DOWN }
```

```java
/** Mutable Mini Rush Hour game. */
public class RushHourGame {

    // ... rep ...

    /**
     * Make a new Mini Rush Hour game starting from the given puzzle layout.
     * @param starting  puzzle to play
     */
    public RushHourGame(RushHourPuzzle starting) { ... }

    /**
     * Move the car that occupies cellNum as many cells as it can move in the
     *   given direction.
     * @param cellNum indicates car to move, requires 1 <= cellNum <= 25, and a
     *                car must occupy that cell
     * @param direction  direction to move
     * @return true if-and-only-if the VIP car is in the game-winning position
     */
    public boolean move(int cellNum, Direction direction) { ... }

    // ... other operations ...
}
```

For reference, the grid and the example puzzle:

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25