

Problem Solving Session

- The remainder of today's class will comprise the **problem solving session (PSS)**.
- Your instructor will divide you into **teams of 3 or 4 students**.
- Each team will **work together** to solve the following problems over the course of **20-30 minutes**.
 - You may work on paper, a white board, or digitally as determined by your instructor.
 - You will submit your solution by pushing it to GitHub before the end of class.
- Your instructor will go over the solution before the end of class.
- If there is any time remaining, you will begin work on your homework assignment.



Class participation is a significant part of your grade (20%). This includes in class activities and the problem solving session.

Your Course Assistants will grade your participation by verifying that you pushed your solutions before the end of the class period each day.

Problem Solving Team Members



Record the name of each of your problem solving team members here.

Do not forget to **add every team member's name!**
Your instructor (or course assistant) may or may not use this to determine whether or not you participated in the problem solving session.

Denis Kešelj
Ivano Maragretić
Gabriel Muskaj

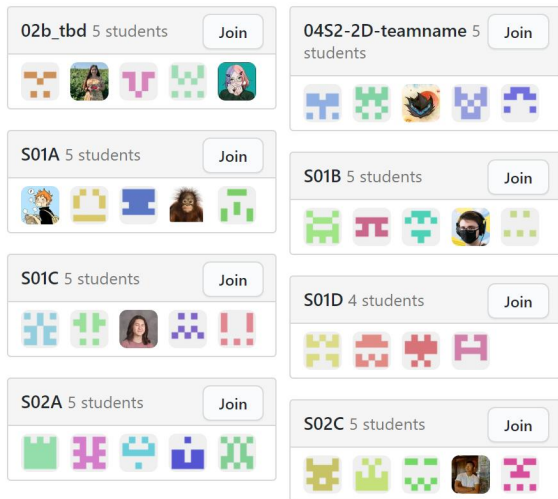
Rush Hour

- **Rush Hour** is a sliding block logic game.
- The goal is to get the red vehicle out the one and only exit.
- The game begins with the vehicles gridlocked in a puzzle pattern.
- A vehicle is aligned either horizontally or vertically and can move in both directions in that orientation. That is, a vehicle aligned vertically can move only up or down. A vehicle aligned horizontally can move only left or right.
 - Vehicles cannot move sideways or diagonally.
- The red vehicle is always aligned with the exit.
- The player moves one vehicle at a time in any valid direction one space at a time.
- Play continues until the red vehicle is at the exit.
- Take a few minutes to give the game a try:
<https://www.thinkfun.com/rush-hour-online-play/>



Join the Team

Join an existing team



OR Create a new team

Create a new team

+ Create team

Do not race each other to click the link! Choose **one** team member to create the team. Be sure to follow any **team naming guidelines** provided by your instructor.

Your instructor will share a GitHub Classroom assignment invitation with you. Unlike the other assignments you have done for this course, the project will require you to create or join a **team**.

Before proceeding further ask your instructor for guidelines regarding team names.

Select **one** member of your team to be the **first** to click on the assignment invitation. GitHub will present that person with the option of creating or joining a new team. This person should **create the new team**. This will automatically create a new repository with any starter code for the team to use. Be sure to name the team according to your instructor's guidelines!

Only after the team has been successfully created should the remaining team members click the assignment link and join the team.

Once everyone on the team has accepted the assignment, you should each clone a copy of the GitHub repository to the computer that you will be using for the rest of today's exercises.

You should then **take turns** working through the following steps (**do not** try to do this all at once):

- Sit so that you can see each other's screens.
 - If working remotely, **share** your screen with your team so that they can see while you work.
- **Pull** the latest version of the repository. If you do not pull first, you may get an error.
- Add your **name** and **email** to a file named "**readme.txt**" in the `data` directory of your repository.
- **Commit** and **push** the file to the repository.

Repeat until every team member has added their name and email to the repository.

Teamwork

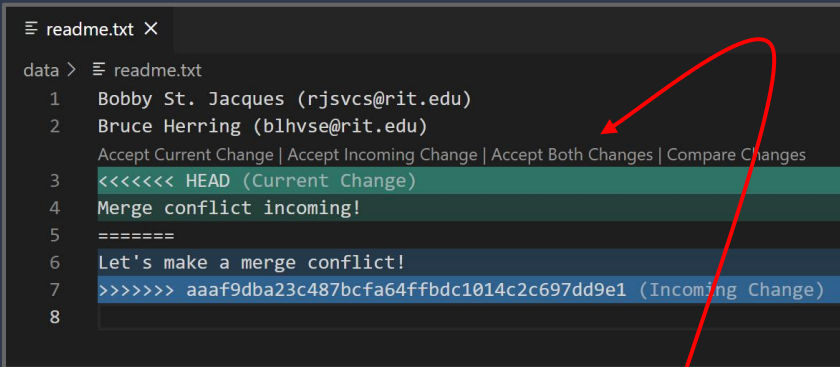


The purpose of this exercise is to make sure that **everyone** on the team has access to clone, pull from, and push to the repository.

Please **take turns** and work so that your team members can **see your screen**.

Merge Conflicts

If a file with a **merge conflict** is opened in VS Code, it will helpfully highlight the conflict and provide links to fix the problem quickly.



```
readme.txt X
data > readme.txt
1 Bobby St. Jacques (rjsvcs@rit.edu)
2 Bruce Herring (blhvse@rit.edu)
   Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes
3 <<<<<< HEAD (Current Change)
4 Merge conflict incoming!
5 =====
6 Let's make a merge conflict!
7 >>>>>> aaaf9dba23c487bcfa64ffbdc1014c2c697dd9e1 (Incoming Change)
8
```

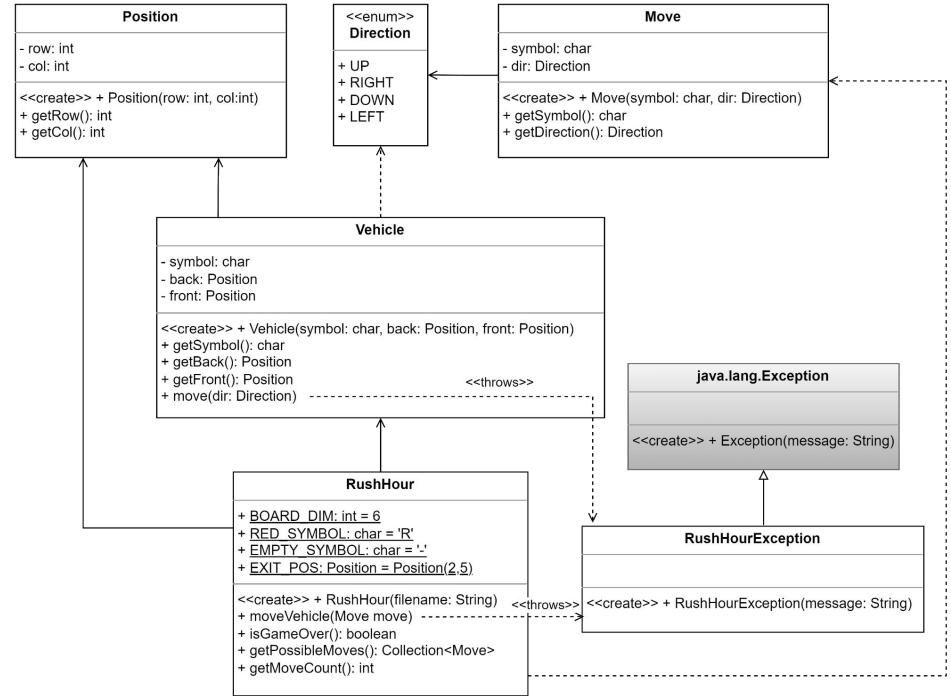
For today, choose the **Accept Both Changes** option.

In the event that **multiple merge conflicts** occur (in the same or different files), you will need to resolve them all before you can push your code.

- Practice resolving merge conflicts by following these steps:
 - Make sure that everyone on the team has pulled the **latest version** of the repository.
 - Two team members should **simultaneously** edit the `readme.txt` file and **add a new line to the bottom** of the file. It does not matter what the text says as long as each person types something **different**.
 - One team member should push their code **first**. **Wait** until the push completes.
 - The **second** team member should then try to push their changes. The push should be **rejected** forcing them to **pull first**.
 - The pull will cause a **merge conflict**.
 - **Resolve** the conflict by opening the file, scrolling to the conflict (if necessary), and **accepting both changes** (see left).
 - Follow the Git workflow to push the merged file to GitHub.
- Repeat the above until every member of the team has had a chance to resolve at least one merge conflict.

A Partial Design

Refer to this full UML class diagram as needed while you and your team work through the rest of this assignment.



- **Pair programming** is a technique during which two developers collaborate to solve a software problem by writing code together.
- One developer takes on the role of **the driver**.
 - Shares their screen.
 - Is actively writing code.
- The other developer(s) takes on the role of **the navigator**.
 - Watches while the driver codes.
 - Takes notes.
 - Asks questions.
 - Points out potential errors.
 - Makes suggestions for improvements.
- The driver and navigator regularly **switch roles**, e.g. every **10-20 minutes**.
 - Set a timer!
 - **Push your code!**
- For the rest of today's problem solving session, you and your team will practice pair programming with **one** team member acting as the driver and the **remaining** team members acting as the navigators.

Pair Programming

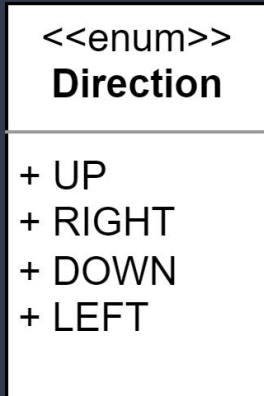
Everyone on the team should take a moment to watch [this short video on pair programming](#).

zoom



When you are the driver, you should be using Zoom or Discord to **share your screen**. Be sure to **push your code** and **switch roles** every 10-20 minutes!

Direction



While you are **required** to include the values in the diagram above, you may add additional state and behavior if you find it necessary to do so.

- Every vehicle on the Rush Hour board can move in one of two directions, assuming no obstruction like another vehicle or the edge of the board.
 - A **horizontal** vehicle can move only **left** or **right**.
 - A **vertical** vehicle can move only **up** or **down**.
- Using the UML to the left, implement a **Direction** class in the `rushhour.model` package.
- You **must** include the values in the UML.

Consider:

 - Will you need additional attributes associated with each value?
 - Will you need additional methods, non-static or static?
 - Will you want to print a nicely formatted version of the enumeration?

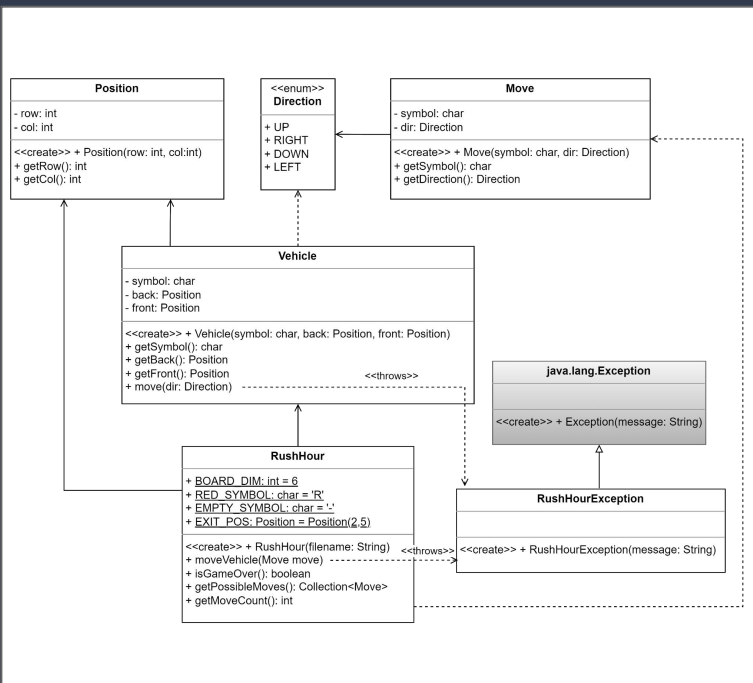
Position

Position
- row: int - col: int
<<create>> + Position(row: int, col:int) + getRow(): int + getCol(): int

While you are **required** to include the state and behavior in the diagram above, you may add additional state and behavior if you find it necessary to do so.

- Every position on the Rush Hour board can be addressed using its **row and column location**.
- Both rows and columns are numbered from **0 to length-1**.
 - Given the Rush Hour board is a fixed 6x6 board, both rows and columns are always numbered 0 to 5.
- Using the UML to the left, implement a **Position** class in the `rushhour.model` package.
- You **must** include the methods and fields in the UML, but you **may** add additional state and behavior as you see fit. Consider:
 - Will you need to determine if two different **Position** objects represent the same position on the board?
 - Will your **Position** class need to work with hashing data structures like `HashSet` or `HashMap`?
 - Will you want to print a nicely formatted version of the class?

RushHour

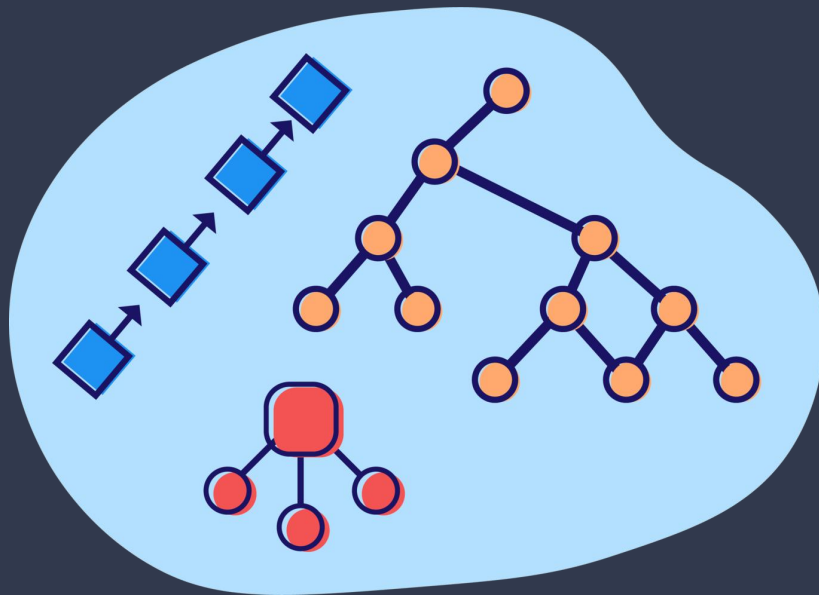


The classes that you've implemented today are only **part** of what will be the full solution to this part of the project.

- Over the next few slides, and using the UML to the left as a guide, you will begin to implement the remaining **required** classes and enumerations with your team.
 - The **RushHourException** class, which is a custom exception that may be thrown by a **Vehicle** or **RushHour** object if a move cannot be completed.
 - When would a move be invalid?
 - This class will be a **checked exception** and so should extend the `java.lang.Exception` class.
 - The **Vehicle** class represents a vehicle on the board.
 - In the game, a **Vehicle** will span 2 or more positions on the board.
 - The **Move** class specifies which vehicle is moving and in what direction.
 - The **RushHour** class is the main class that implements the game rules and maintains the game state.
- All new types should be in the `rushhour.model` package.
- Refer to the full UML diagram as needed.

- Choosing the right data structures for your Rush Hour game will be critical for **efficient implementation** and program **execution**.
- As you may have guessed from mentions of a `model` package, you will be implementing a **Model-View-Controller** (MVC) design pattern.
 - One implication of this is that classes in your `model` package must be designed and implemented for efficient game execution and state management without regard to user interface (UI) needs.
- Work with your team to decide on the **data structures** and **attributes** that will be needed by the `RushHour` and `Vehicle` classes. Consider:
 - Does a `Vehicle` need to maintain all of the positions on the board that it occupies?
 - Does the `RushHour` class need a data structure or class that represents a board?
 - How will attributes be initialized?
- After agreeing on the initial data structures
 - Create new classes, interfaces, and enumerations as necessary
 - Add attributes to your `Vehicle` and `RushHour` classes.
 - Work on the constructor and other initialization private methods.

Data Structures



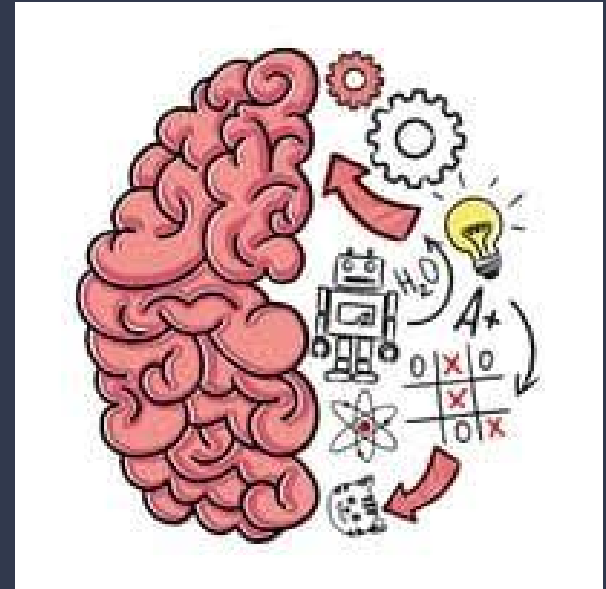
Spending time **upfront thinking** about data structures can prevent kludgy code, poor code communication, and rework down the line.

- Now that you have a good starting point of data structures and attributes, discuss the **logic** needed to implement the **game play** and maintain the **game state**.

How will your `RushHour` class

- Determine that a vehicle is able to move in the given direction?
 - Manage game state?
 - Return a representation of the vehicles to the UI?
 - Determine and return all possible vehicle moves to the UI?
 - Reset a game?
- Once your team feels comfortable with the **design decisions** made, begin implementing the methods in the UML diagram.
 - Add attributes and methods as needed.

Thinking Logically



Resist the urge to zip by design discussions and dive into coding. The further into a project a design flaw is discovered, the more costly it is to fix.

Meeting Times

This project counts for **significantly more** than a normal homework assignment. You and your team should plan on spending more time than usual working on it.

You should also plan to **work together** as much as you can, even if that means setting up a remote meeting using Zoom or Discord.

Plan **at least 2 meetings** over the course of this part of the project. Each meeting should be at least **1 hour**. Following the example on the right, put the information for all scheduled meetings in a *meetings.txt* file and **push to the data directory of your repository**.

Date/Time	Location	Area of Focus
4/1 @ 1:00PM	Discord	Vehicle Class
28/3 @ 6:00PM	WhatsApp	Sharing tasks for the project

It is **highly recommended** that you also establish a means of **asynchronous communication**, e.g. Discord channel, Slack channel, group text, etc, where you can post updates, ask questions, and coordinate.

If You Made it This Far...

Your team is off to a good start, but you are **not quite** finished with **Part 1** of the Project yet.

You still need to create a **working implementation** of Rush Hour that can be played by a single human user via a **command-line interface**.

If you have time remaining in class, you should begin reading the **full project description**, which you will find on MyCourses.

