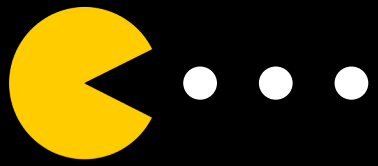
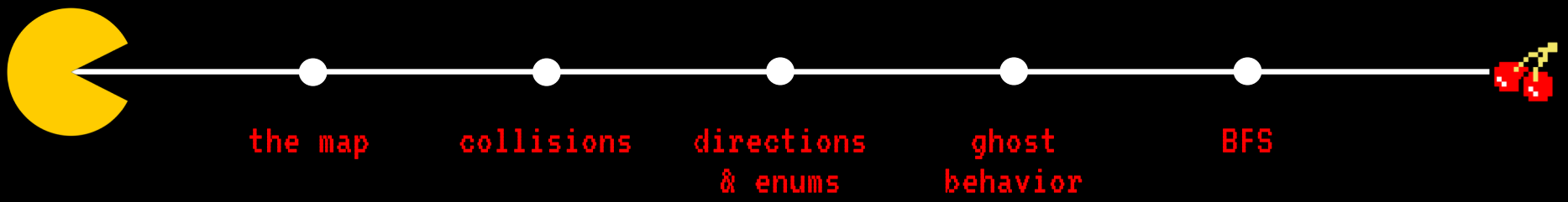
PACMAN HELP SLIDES



presented by: anna, cece, jake, & victor

design overview

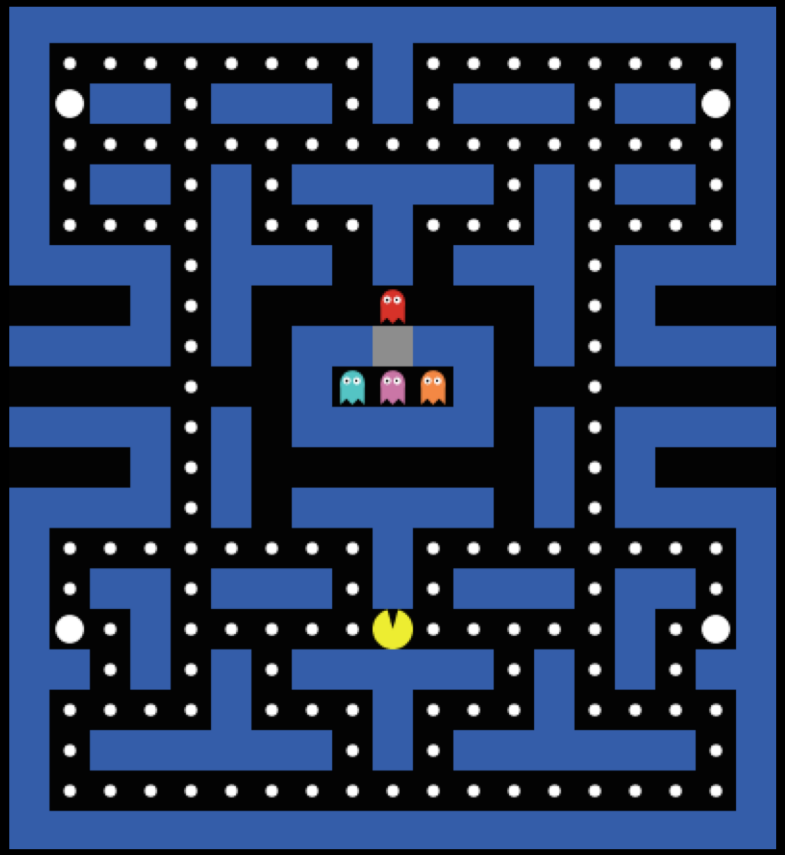


the map

- Use our support map to generate your own **object-oriented** map

- What does this mean?
  - Instead of modeling each square of the map as a simple **enum**, design an object to keep track of the state of the square and encapsulate other helpful functions!

- What **data structure** will you use to organize these map square objects?



the map: squares

- Each map square should know if it is a wall and what elements are inside of it at any moment (ghosts, pellets, energizers, etc.)

- Have a “smart square” class, and have it contain an ArrayList!
  - What type will this ArrayList hold?

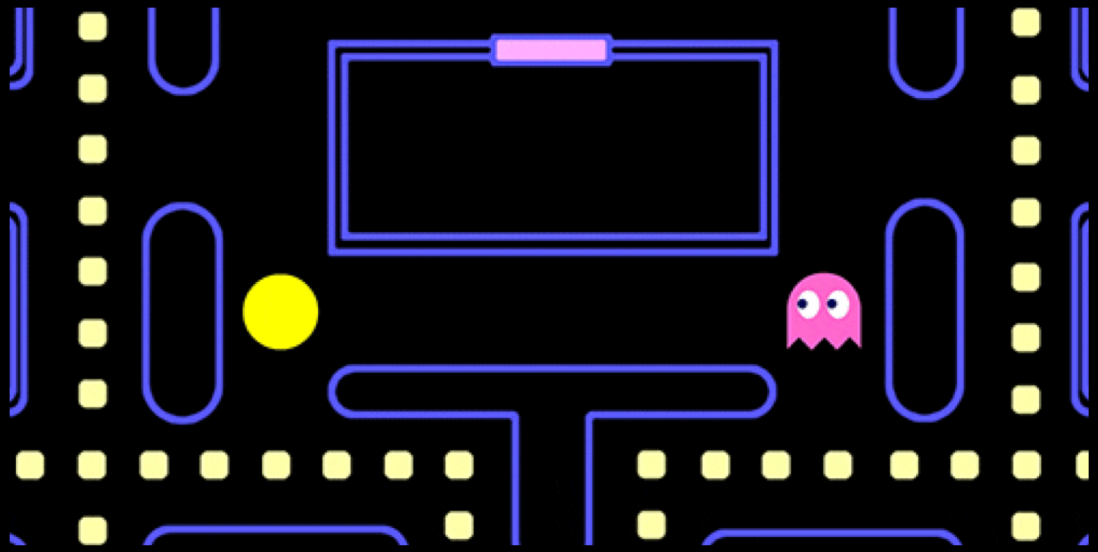
  - **Beware:** If you try to remove or add something to an Arraylist while iterating through it, you will get a ConcurrentModificationException!
    - *There is a way to design your program so that you never run into this problem!*

the map: squares

Should this list contain Pacman?

- Let’s think about how we would handle collisions both ways. Maybe that will help us answer this question.

collisions



Collisions occur when Pacman and an object are in the same square!

collisions

Question: Should we store Pacman in a Square's ArrayList?

- **Benefits**: We treat Pacman like we do other maze objects!
- **Downsides**: Pacman isn't always like other maze objects!
  - We want our Pacman to collide with everything in the Square, but we don't want Pacman to collide with Pacself 😉
  - To check, we might have to write isPacman() methods that return false in the pellets, which isn't exactly logical...😬

collisions

Solution: Pacman is not directly stored in Square's ArrayList

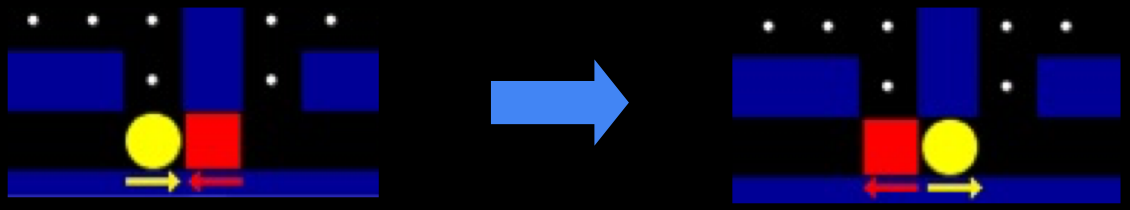
- When Pacman enters a Square, iterate over that Square's ArrayList and tell each element to perform its collision action
- We can even delegate the collision actions to the other elements of the game — a little counter-intuitive, but super elegant!

collisions

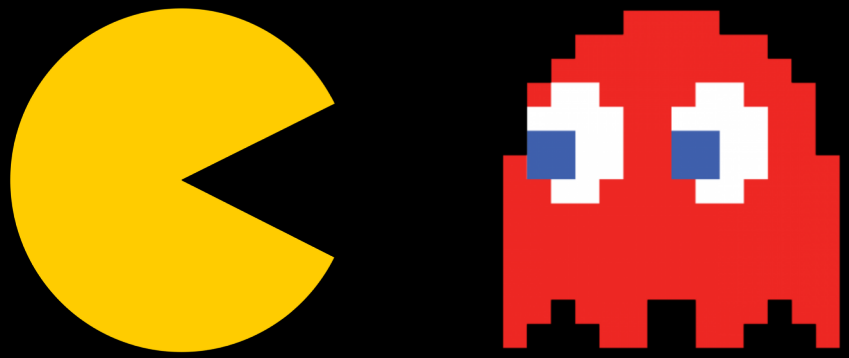
So, if collisions occur only when **Pacman** and a maze object are in the same cell...

There's an edgecase!

- If **Pacman** and a Ghost are moving towards each other with the right timing, it's possible for them to switch cells without colliding



collisions



CHECK TWICE!

Move Pacman → Check for collisions → Move ghosts → Check for collisions

board coordinate vs. Point2D

In this project, you'll be dealing with many, many points on the board

Should we use a `javafx.geometry.Point2D` to represent all of these points?

- The `Point2D` constructor takes in two doubles, meaning
  - you could pass in *negative* numbers (we want only 0, 22)
  - you could pass in *decimal* numbers (we only need integers)

- What if you get an `ArrayIndexOutOfBoundsException`?
  - You might know what *line* of code caused it, but will you know *when* you created that point?

How can we fix this?

stencil/support code: board coordinates

```
private void checkValidity(int row, int column) {
    if (row < 0 || column < 0) {
        throw new IllegalArgumentException(
                "Board Coordinates must not be negative: "
                        + " Given row = " + row + " col = " + column);
    } else if (row > _ROW_MAX || column > _COL_MAX) {
        throw new IllegalArgumentException(
                "Board Coordinates must not exceed board dimensions: "
                        + " Given row = " + row + " col = " + column);
    }
}
```

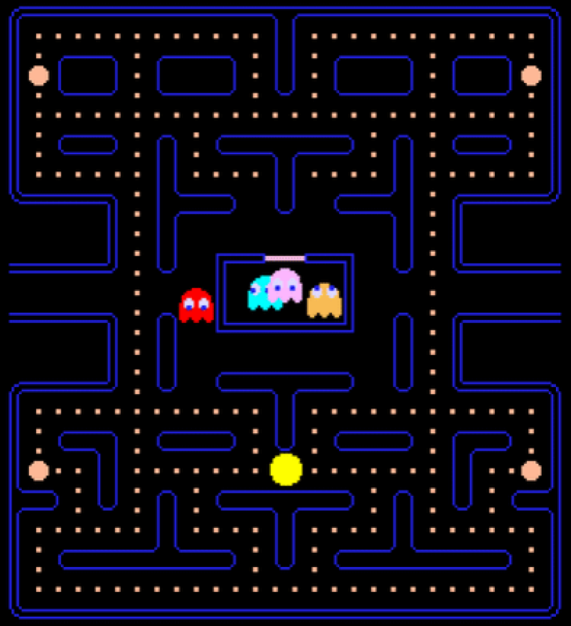
- Throw exceptions when your BoardCoordinate receives bad coordinates (then print a descriptive error message)
  - Prevent bugs from appearing in your algorithm by catching them here!
  - Ensures that you can't make a BoardCoordinate instance that would exist outside of the board!
  - Be consistent with row, column indexing in your board!

stencil/support code: board coordinates

```
private void checkValidity(int row, int column) {
    if (row < 0 || column < 0) {
        throw new IllegalArgumentException(
                "Board Coordinates must not be negative: "
                        + " Given row = " + row + " col = " + column);
    } else if (row > _ROW_MAX || column > _COL_MAX) {
        throw new IllegalArgumentException(
                "Board Coordinates must not exceed board dimensions: "
                        + " Given row = " + row + " col = " + column);
    }
}
```

- *BUT* the BoardCoordinate constructor will *NOT* call checkValidity if the constructor boolean **isTarget** is true, so the exception will not be thrown
  - This allows you to create chase targets based on offsets to Pacman while still using this class. You should only set isTarget to true if you are actually making a target square!

directions & enums



How can you easily represent the directions that Pacman and the ghosts can move in?

directions & enums

enums! A data type whose fields consist of a fixed set of constants

- Keep in mind that there are other ways to store directions, but enums are a very clean way to do so:
  - You might have used static constants in a similar way during Tetris, but this will be much more complicated in Pacman
  - A totally ***unacceptable*** (though maybe functional) solution is to use integers or strings that are never defined as constants. This is extremely buggy. DO NOT DO THIS!!!

directions & enums

Here is an example using enums to represent the directions left and right

```
/* Has its own .java file just like an interface or class */
public enum Direction {
        //The types of directions, MUST come first
        LEFT, RIGHT;

        //enums can have methods just like classes do
        public Direction getOpposite() {
                //enums are comparable just like ints
                switch (this){
            case LEFT:
                                return RIGHT;
                        case RIGHT:
                                return LEFT;
                }
        }
}
```

directions & enums

- To get a specific value…
  - use `Direction.LEFT` or `Direction.RIGHT`
- You cannot use the `new` keyword with `enum`s
  - The valid way of initializing `enum`s would look like:

```
Direction dir = Direction.LEFT;
```

- All `enum`s have a `static` method to return their values as an array
  - In code, this would look like:

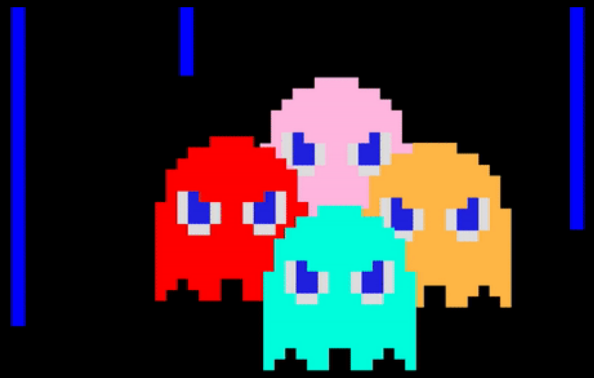
```
Direction[] allDirections = Direction.values();
```

directions & enums

Here is an example of a method that uses the Direction enum

```
        // method to check before crossing street
        public boolean isSafeToCrossStreet() {

                Direction[] directions = Direction.values();

                for (int i = 0; i < directions.length; i++) {
                        // check if a car is coming from that direction
                        if (this.isCarDrivingFrom(directions[i]){
                                return false;
                        }
                }
                return true;
        }
```

ghost behavior



Ghosts move differently based on the current status of the game. Ghosts can have 3 different modes: *frightened*, *scatter*, and *chase*

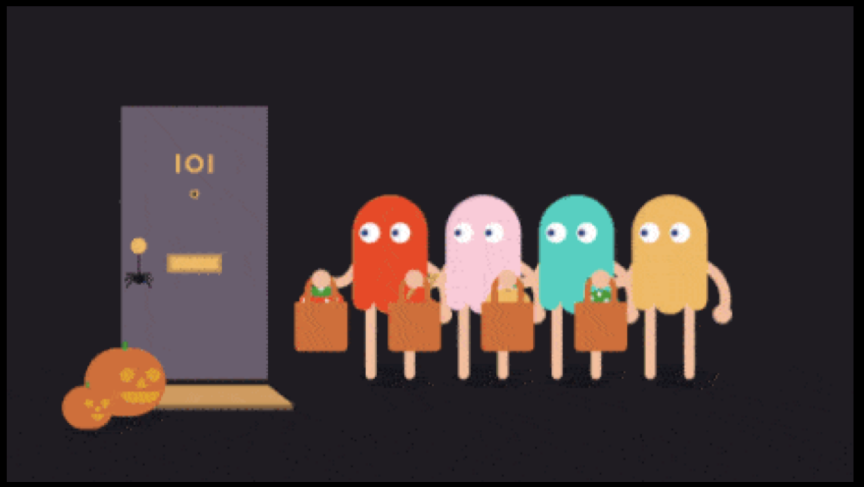
- How will you handle switching between modes? How many Timelines do you need?
- How will you represent each ghost's current mode? (***Hint:*** *What did you just learn?*)

ghost behavior

Frightened Mode

- Starts when Pacman eats an energizer
- 7 seconds in which Pacman can ***eat the ghosts***
- Ghosts will move ***randomly***
    - At every intersection, choose a random direction for the ghost to move.

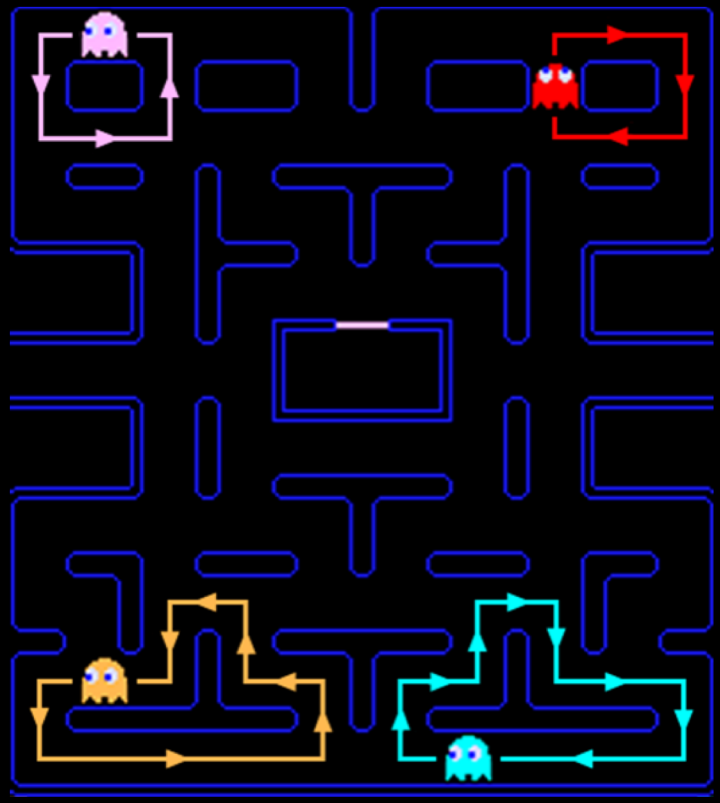
(***Note**: Ghosts should never do a 180 degree turn!)*



ghost behavior

Scatter Mode

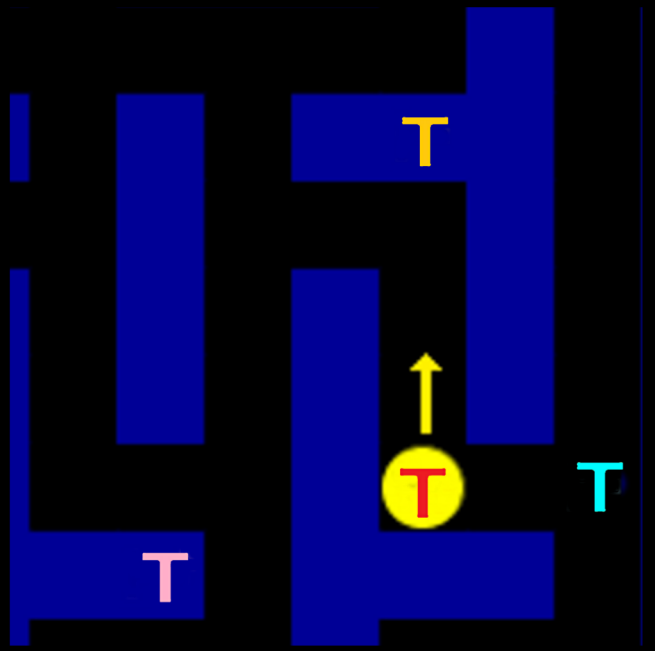
- Each ghost should target a ***different corner*** of the board
- Ghosts will run in circles during scatter mode (in the corners)



ghost behavior

Chase Mode

- The targets should be ***relative*** to Pacman's current location (***Note**: the target is constantly changing!*)
- The ghosts should all be unique – this means they have ***different targets*** during chase mode
- The original game's targeting is complicated (*see extra credit if you're interested*)
  - Instead, use different target locations:
    - Pacman's location.
    - 2 spaces to the right of Pacman's location.
    - 4 spaces above Pacman's location.
    - 3 spaces to the left and 1 space down from Pacman's location.



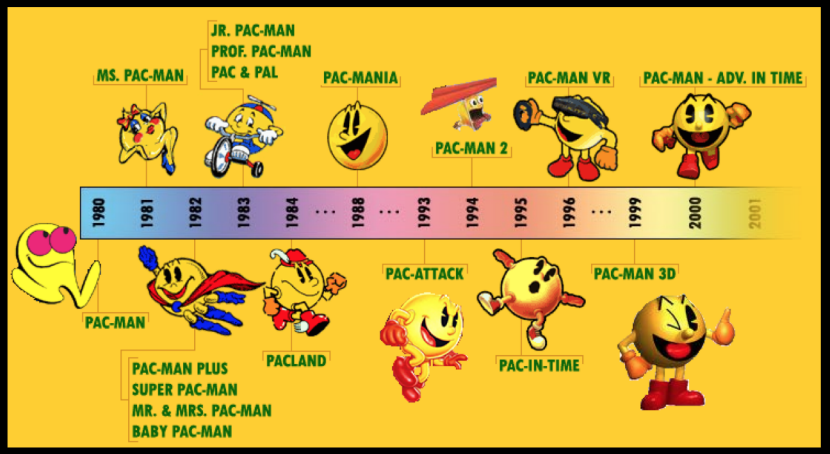
timeline & movement

- What is a counter?
  - An variable (int) that is incremented during a TimeHandler
  - Oftentimes, you check the counter to see if it is equal to some value, and based off of whether that condition is met, perform some game logic
- Think about how this applies to certain modes
  - How do you know when frightened mode (7 seconds) is over?
- Note: you can use multiple counters in the same TimeHandler!

ghost pen

- The ghost pen is the place the Ghosts ***start*** in, and where they ***return to*** when they are eaten by Pacman
- At start of a game, we want Blinky to be ***outside of the pen***
- Afterwards, we want Pinky, then Inky, and then Clyde to exit the pen.

ghost pen



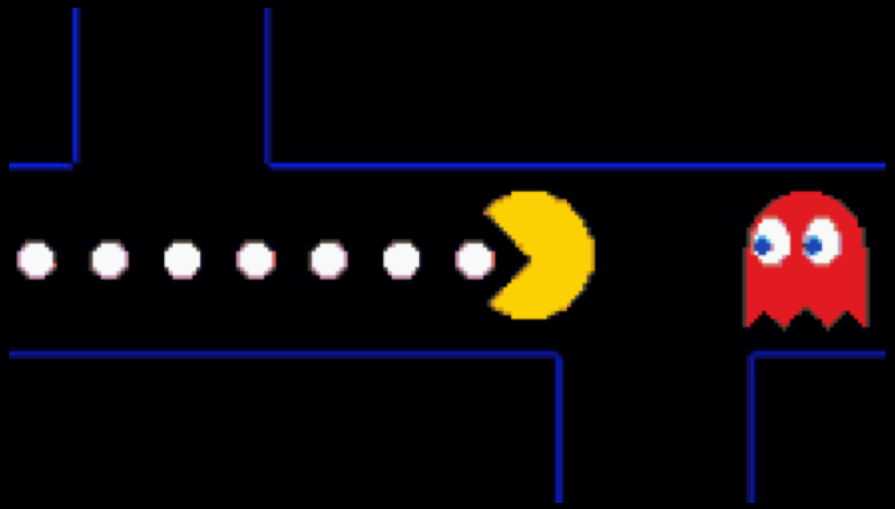
- Also, when two ghosts get eaten, we want the first one eaten to be the first one to exit. *What does this sound like?*
- Great, so now we know which ghost should exit the pen, but when should it exit? (***Hint**: How can we track of a duration of time?)*
- Use a `Timeline`! And specifically, the pen should have its own `Timeline` to make your lives easier!

the ghost's search algorithm

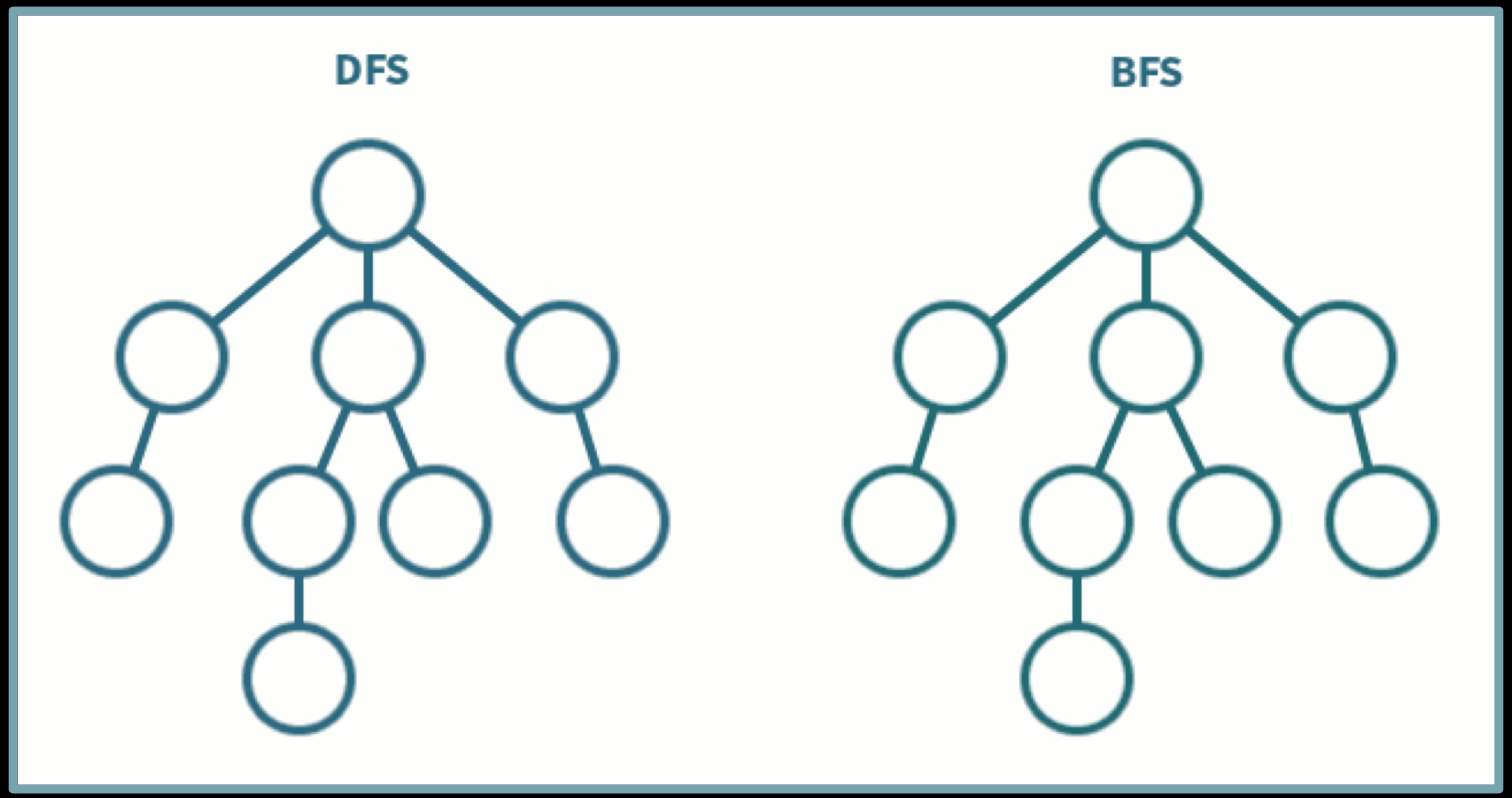
breadth-first search

Search algorithms are often used in Computer Science, and Breadth-First Search is one (of many) popular searches that are used.

In Pacman, we will be adding functionality to a standard Breadth-First Search to implement Ghost smart turning.



breadth-first search

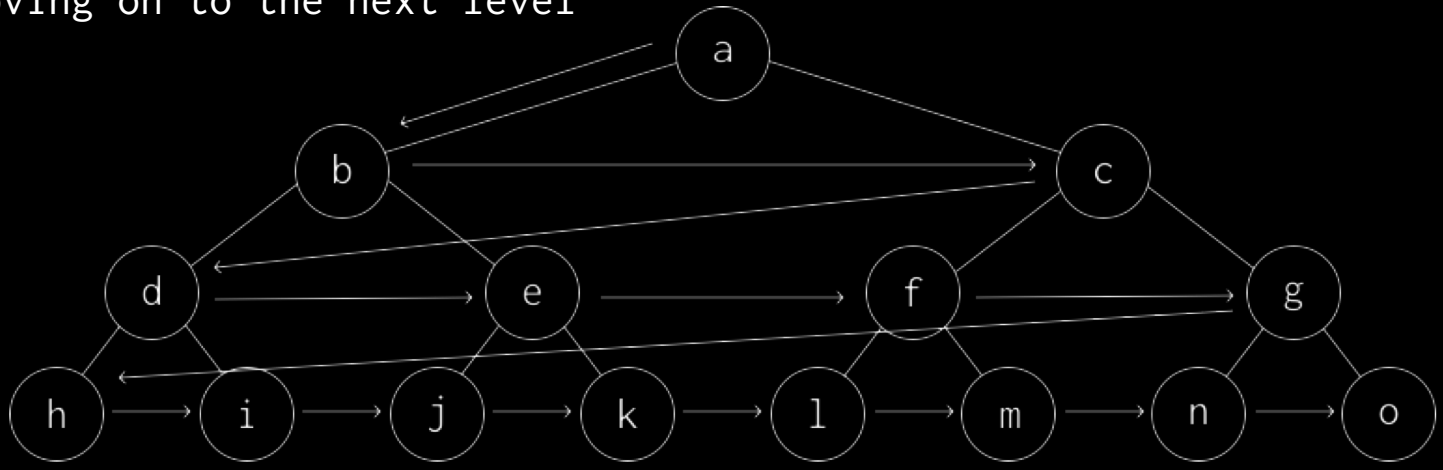


breadth-first search

What is a **Breadth-First Search**?

Definition of “Breadth”: *the distance or measurement from side to side of something; width*

A **Breadth-First Search** means: we will examine ***all*** neighbors of a given node ***before*** moving on to the next level



breadth-first search

Characteristics of a **Breadth-First Search (BFS)**:

Using a BFS ***GUARANTEES*** that we will find the shortest path between a starting node and a specified end node.

Here is a standard BFS:

```
bfs(root):
    Q = new Queue()
    enqueue root
    while Q is not empty:
        current = Q.dequeue()
        enqueue current's neighbors if they haven't been visited
```

the ghost's breadth-first search

- **Goal**: When the Ghost must make a directional decision at a turn, use a BFS to determine the quickest direction towards Pacman

- We only want to determine the Direction to turn, we ***do not*** want to find the path to take

- This is true because Pacman is always moving around and the Ghosts should move freely in an informed (BFS) way

the ghost's breadth-first search

So, what needs to change to make a BFS work for the Ghosts?

1. `root` needs to be the Ghost's current `BoardCoordinate`

1. Need a variable `target`, which would be the Ghost's frightened location or some offset on `Pacman`'s location.
    - Because of offsets, the target may occasionally be a wall. In order to protect against this, we'll search the entire board and just use the closest square to the target.
    - Therefore, we also need a `closestSquare`!

1. Need to keep track of which direction to take!

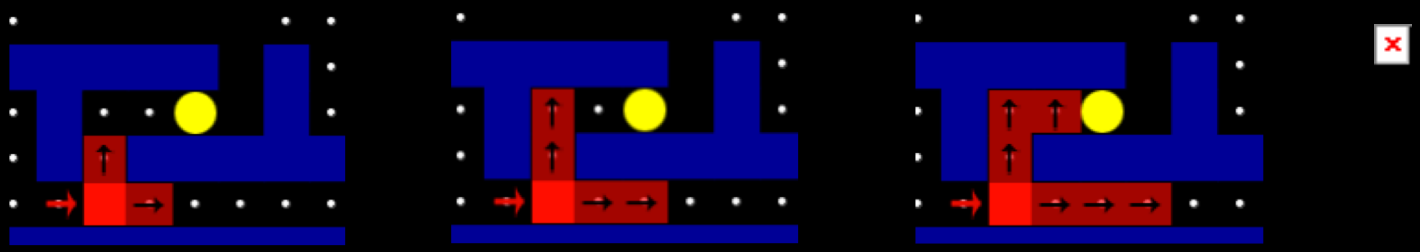
the ghost's breadth-first search

```
ghostBFS(target):

    Q = new Queue()

    initialize variables to track closest square and directions

    enqueue initial turn options and document the directions

    while Q is not empty:

        current = Q.dequeue()

        if current is the closest to target:

            update relevant closest information

        if current's neighbors haven't been visited

                enqueue them

            mark that you're visiting them via the initial
            direction that lead you to dequeue current


        return the direction to turn
```

the ghost's breadth-first search

How do we keep track of the directions? A directions array!

- We use a 2D array of Directions to keep track of the initial direction the Ghost would need to take in order to get to the current square
  - Remember: a Direction is an enum!
- **Goal:** determine direction the Ghost should turn to reach Pacman soonest
  - How do we find this given a Direction[][]?
    - The Direction marked on the “closest cell” indicates the best initial turn!



the ghost's breadth-first search

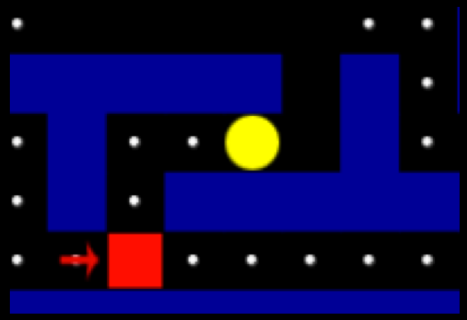
We will use a few different data structures to perform the BFS

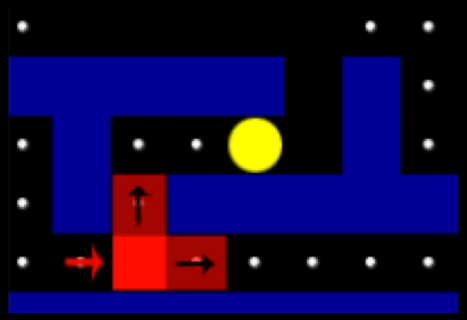
- Use BoardCoordinates to represent squares on the board.
- A 2D Array of Directions to mark the initial direction taken that leads to a given cell.
  a. Cells can be indexed into by BoardCoordinate location.
  b. The BoardCoordinate that represents the closest square will allow us to index into the 2D Array to find the *best initial direction*.
- Queue of BoardCoordinates can be used to make sure we visit *all* squares.
  a. Add each cells by enqueuing their BoardCoordinate location.
  b. We add the "neighbors" at each iteration (*ensures we look at the closest cells first).*

the ghost's breadth-first search

Step 0: Initial population of the queue

- Enqueue the initial neighbors of the cell containing the ghost
  - *Cannot* be a wall nor a cell in the direction opposite to the direction of the ghost's movement
- Mark each cell with their *initial direction*
  - This direction is the direction the ghost must go in order to reach the cell
- Do we need to do anything for the cell in which the ghost currently lies?

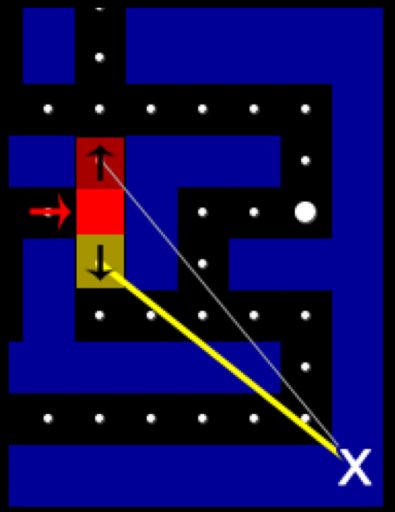




the ghost's breadth-first search

Step 1: Check the current cell

- While the queue of BoardCoordinates is not empty, dequeue to get current cell
  - When the queue is empty, we have visited all possible cells and have found the optimal direction!
- Calculate the distance between the **current** cell and the **target** cell
  - This can just be the Euclidean Distance between the BoardCoordinates
- If the **current** cell is closest to the **target** cell:
  - Update the closest reference cell
  - Update the smallest distance to the target



the ghost's breadth-first search

Step 2: **Marking the neighbors**

- Determine which neighbors are valid:
  a. Cannot be a wall.
  b. Must be unvisited.
     - *How do we check if something has been visited?* Whether or not a **direction** has been **assigned** to a cell!
- We need to store the directions of the cells we have marked in some data structure…
     - Remember the 2D array of Directions?
- Enqueue the *valid* neighbors so that they can be checked late.

the ghost's breadth-first search

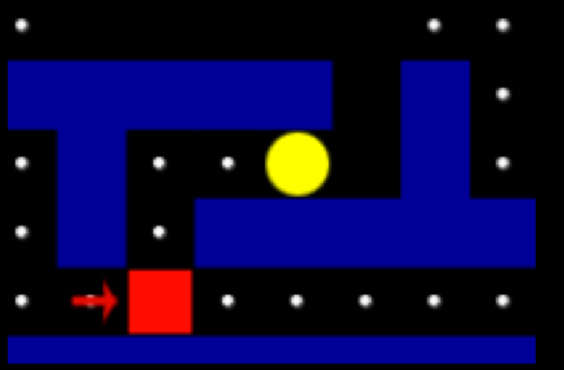
Step 3: **Repeat**

- When we finish with the **current** cell, move on to the next cell
  - The unvisited cells are stored in the queue!
  - We dequeue the next unvisited cell as the current cell

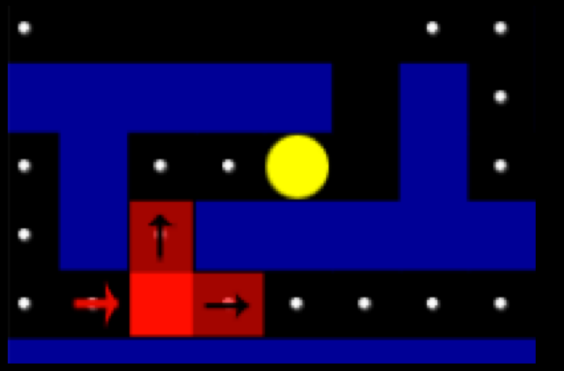
- **When do we stop?**
  - When we have no more cells to check (the queue is empty)
  - ...So we need to repeat *while the queue is not empty!*

the ghost's breadth-first search

Hand Simulation of Ghost's Breadth First Search



- Let's consider a simple case:
  - The Ghost can make two choices
  - Look at a small game board
- **Step 0: Initial population**
  - Create an empty queue of BoardCoordinates
  - Add the neighbors of the Ghost's starting location with their corresponding directions
- **Step 1: Check the current cell**
  - Dequeue to get the current cell
  - Calculate the distance of the current cell from the target
  - Update closest cell and smallest distance references if necessary
- **Step 2: Mark the neighbors**
  - Enqueue the valid neighbors of the current cell
- **Step 3: Rinse and repeat!**
  - Repeat until the queue of BoardCoordinates is empty

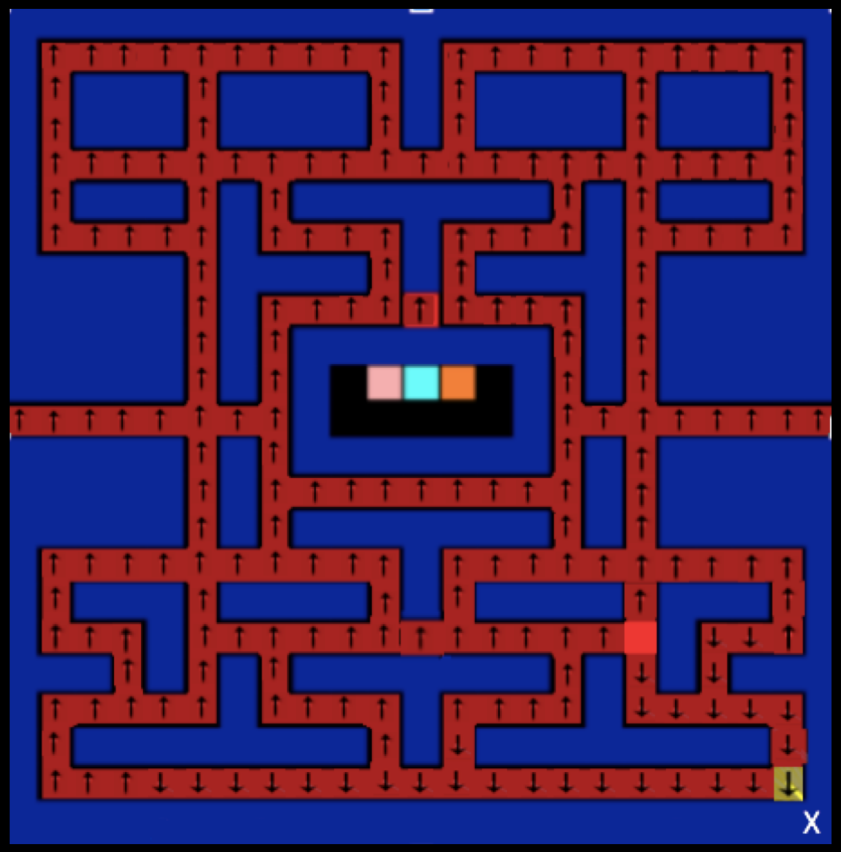


the ghost's breadth-first search

- The Breadth First Search should continue until you have a direction associated with every valid square in the board array

- In other words, every square in the paths has a direction stored at that location in the direction array

This picture shows a completed round of BFS



coding incrementally

1. Get your board to show up with all of the **Dots** and **Energizers**
2. Get **Pacman** to move correctly - this means both time and key input!
3. Make sure your move validity works, and be sure to check the validity
4. Get collisions to work — **Pacman** should collide with **Dots** and **Energizers** and the score should increment properly
   — Think about implementing a **Collideable** interface!
5. Get a **Ghost** to BFS properly.
   a. Make sure your move validity works properly!
   b. Consider testing in **Scatter Mode**, or **Blinky** in **Chase Mode** (as **Blinky** goes right to **Pacman**)
6. Get **Ghosts** to appear and leave the **Ghost Pen**

GOOD LUCK!

javafx.animation.Timeline<FinalProject>

- Sign up for design discussion time by **Saturday, 11/30 at 5 PM**
  - Signups will go out next week
- Mini-Assignment will be due **at Design Discussion**
- Design Discussions will be **Tuesday, 12/3 - Thursday 12/5**
- Handin Deadlines
  - **On-time**: **Sunday 12/14, 11:59 PM**
  - **Late**: **Tuesday 12/16, 5 PM** (you *cannot* use a late pass for the final project)

reminder: no class on Tuesday, 11/26

HAPPY THANKSGIVING!