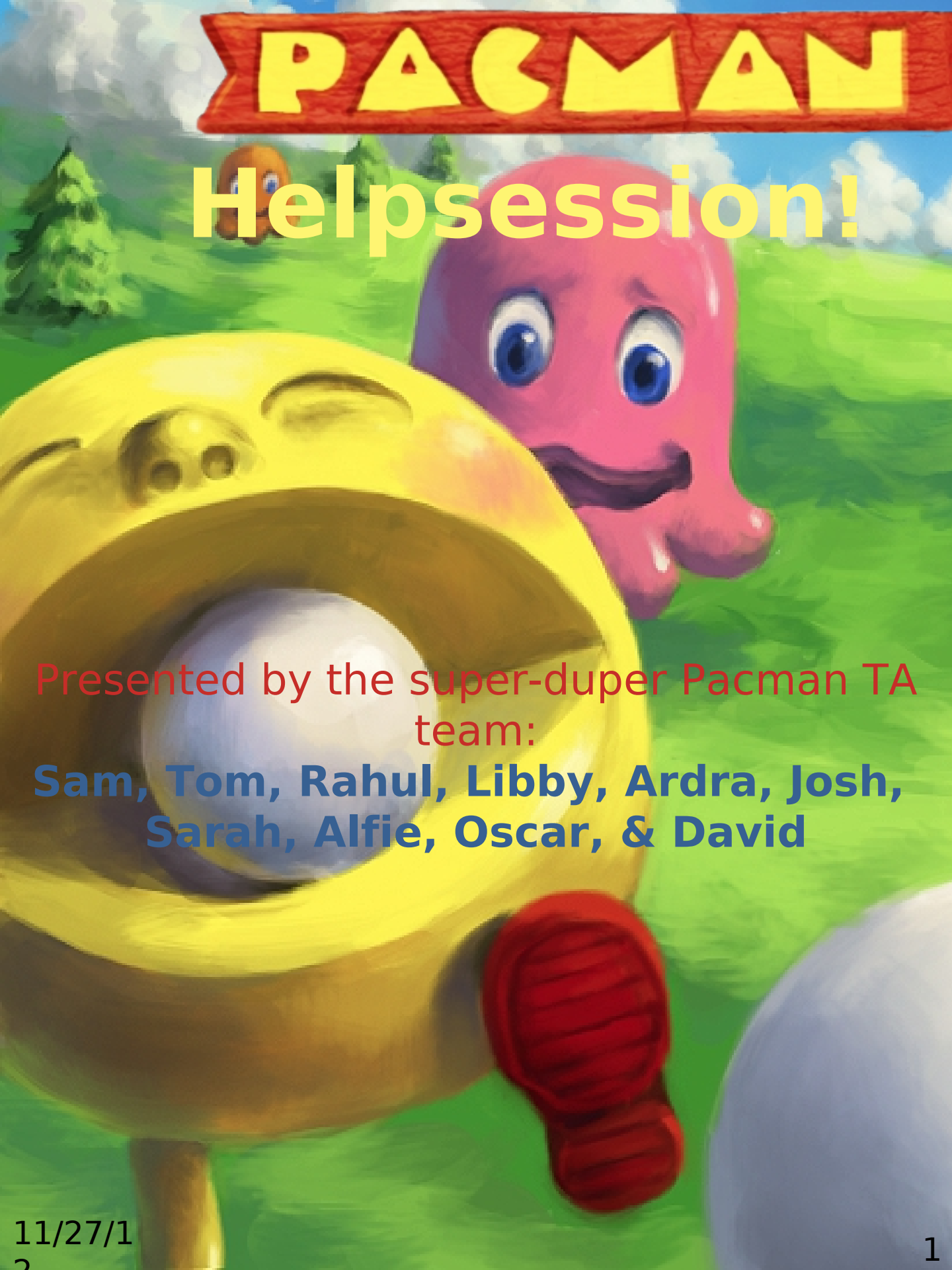


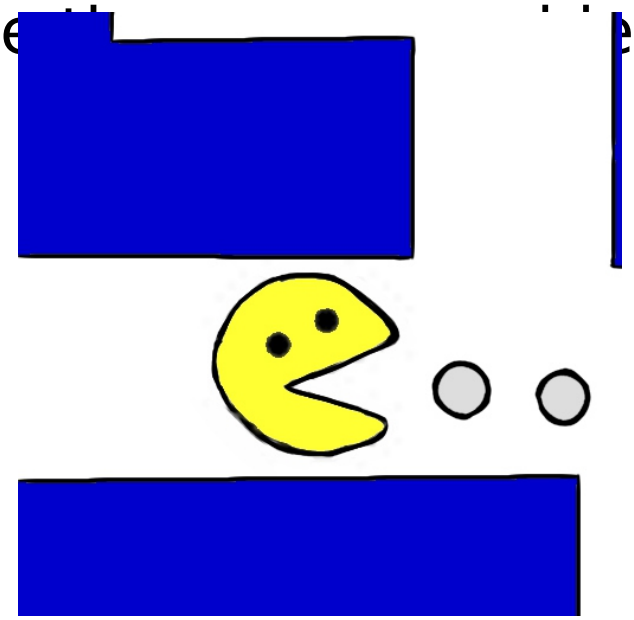
Helpsession!

Presented by the super-duper Pacman TA team:

Sam, Tom, Rahul, Libby, Ardra, Josh, Sarah, Alfie, Oscar, & David

THE MAP

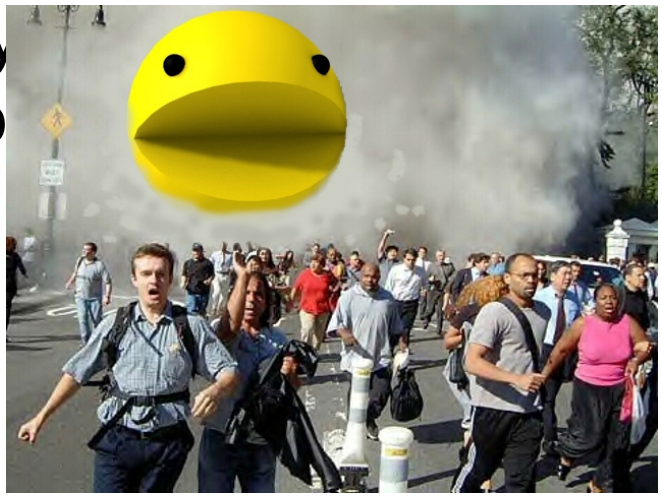
- Use our support map to generate your own **object-oriented** map.
- What does this mean?
  - Instead of modeling each square of the map as a simple integer, you should design an object to keep track of the state of the square and encapsulate other helpful functions.
- What data structure will you use to organize [illegible]ects?



MAP SQUARES

- Each map square should know about what elements are inside of it at any moment (ghosts, pellets, energizers, etc.)
  - use a **`java.util.ArrayList`**!
- What **type** will this ArrayList hold?
- **TRICKY QUESTION**: Should this ArrayList also contain Pacman?
- Beware: If you try to remove or add something to a `ArrayList` while iterating through it, you might get a `ConcurrentModificationException`!

Think of how y
problem in Do



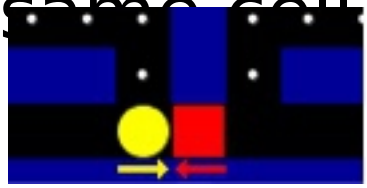
COLLISIONS

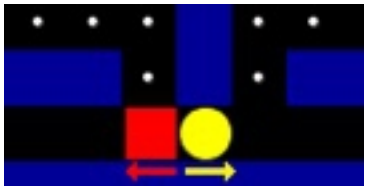
Remember: a collision occurs when Pacman and another object occupy the same map square.

- How to handle collisions:
  - When Pacman enters a square, iterate over that square's ArrayList and tell each element to collide with Pacman.
  - If each element in the ArrayList knows how to collide with Pacman, then Pacman has one less thing to worry about
  - Is it useful to pu e ArrayList as wel

COLLISIONS

- Collisions occur when Pacman and a maze object are in the same cell





- But there's a bug -- if Pacman and a ghost are moving towards each other with the right timing, it's possible for them to switch colliding.

- **How can we fi**

  this??

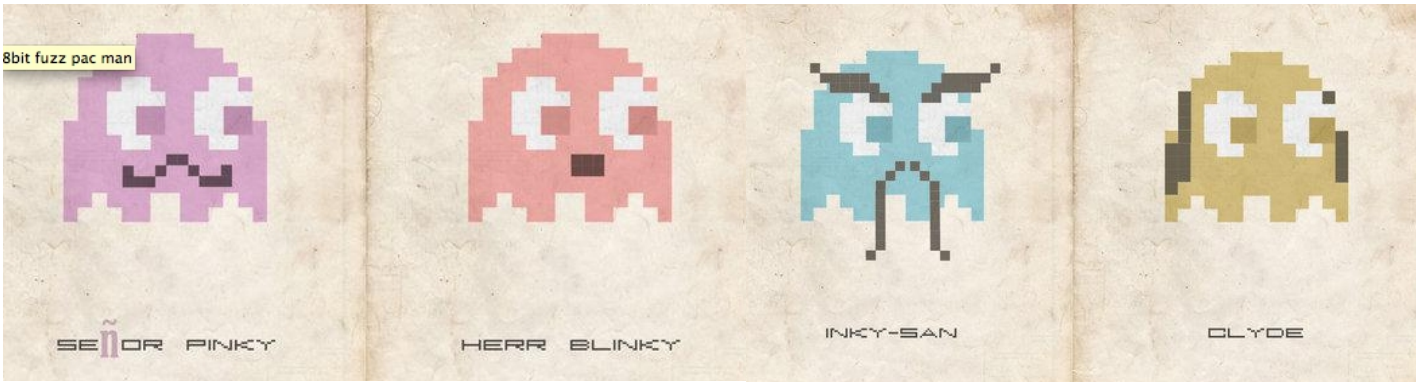
COLLISIONS

Check twice!!

- Move Pacma
- Check for collisio
- Move ghosts
- Check for collisio

DIRECTIONS & enumS

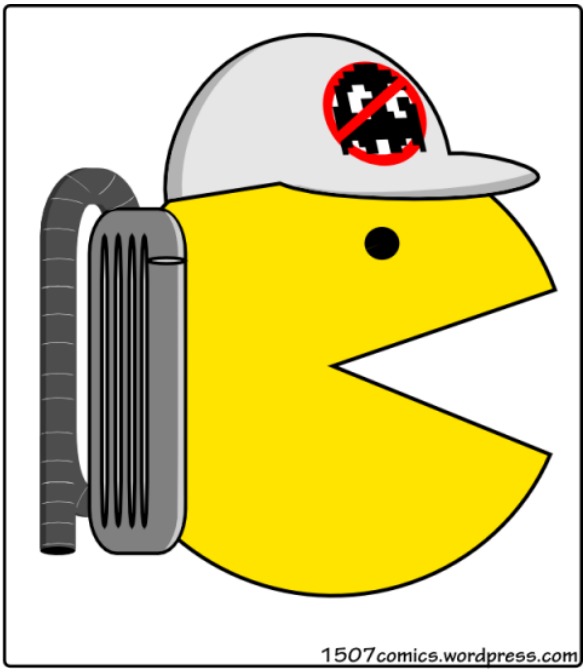
- How can you easily represent the directions that Pacman and the ghosts can move in?
- **enum**s! A data type whose fields consist of a fixed set of constants
- Keep in mind that this is only one way to do directions:
  - You might have used static constants in a similar way during Tetris, but this will be much more complicated in Pacman
  - A workable, but totally unacceptable, way is to use integers or strings that are never defined as constants
  - (Use **enum**s)



USING enumS

- Here is an example using **enum**s to represent the directions left and right:

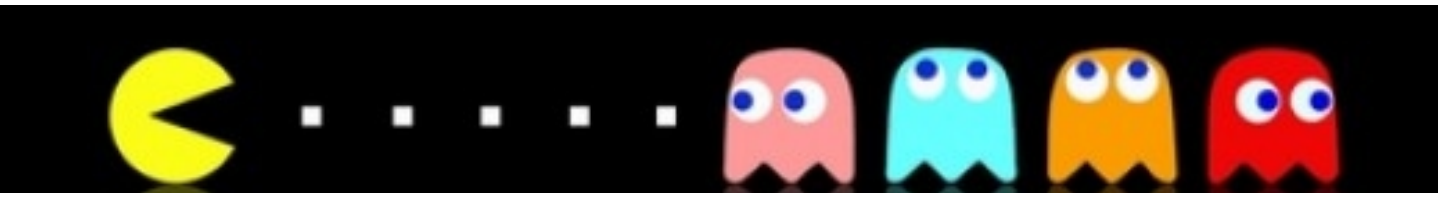
```
/* Has its own .java file just like an
 * interface or class */
public enum Direction {
    //The types of directions, MUST come first
    LEFT, RIGHT;

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



The Original Ghostbuster

USING enumS

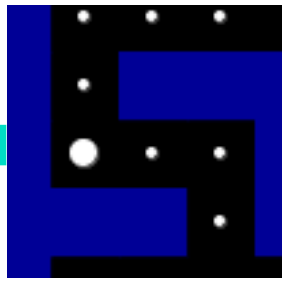
- To get a specific value, use:
  `Direction.LEFT` or `Direction.RIGHT`
- You cannot `new` enums
- So, in code this would look like:
  `Direction dir = Direction.LEFT;`
- All `enum`s have a `static` method to return their values as an array:
  `Direction[] directions = Direction.values();`



- Example of a method that uses the `Direction` `enum`:

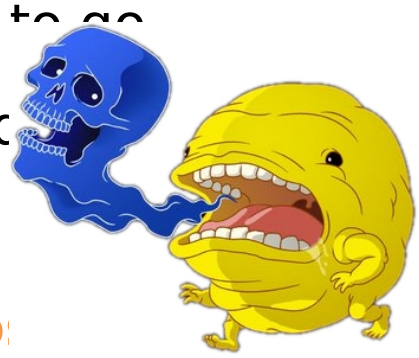
```
// method to check before crossing street
public boolean is isSafeToCrossStreet() {
  Direction[] directions = Direction.values();

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

GHOST BEHAVIOR



- 3 modes
  - **Chase**, **Scatter**, **Frightened**
- Frightened mode
  - Starts when Pacman eats an energizer
  - Lasts about 7 seconds
  - Ghosts will move **randomly**
  - How it works: at every intersection, choose a random direction to go
  - note: ghosts should never d

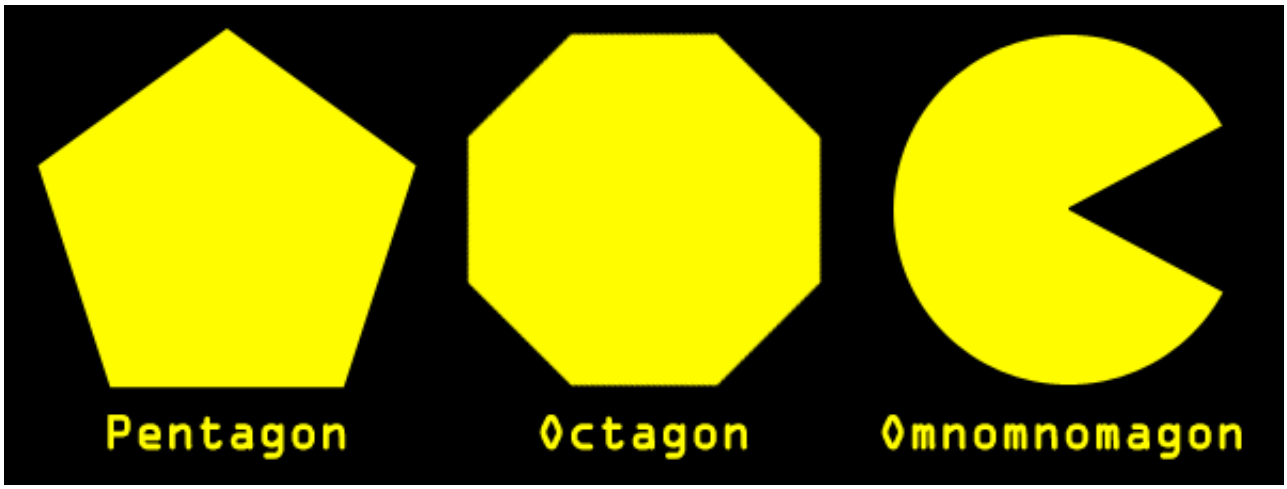
    degree turns!
  - **Pacman can eat the gho**



- How will you handle switching between modes? How many timers do you need?
- How will you represent each ghost's current mode? (hint: WHAT DID YOU JUST LEARN??)

GHOST TARGET LOCATIONS

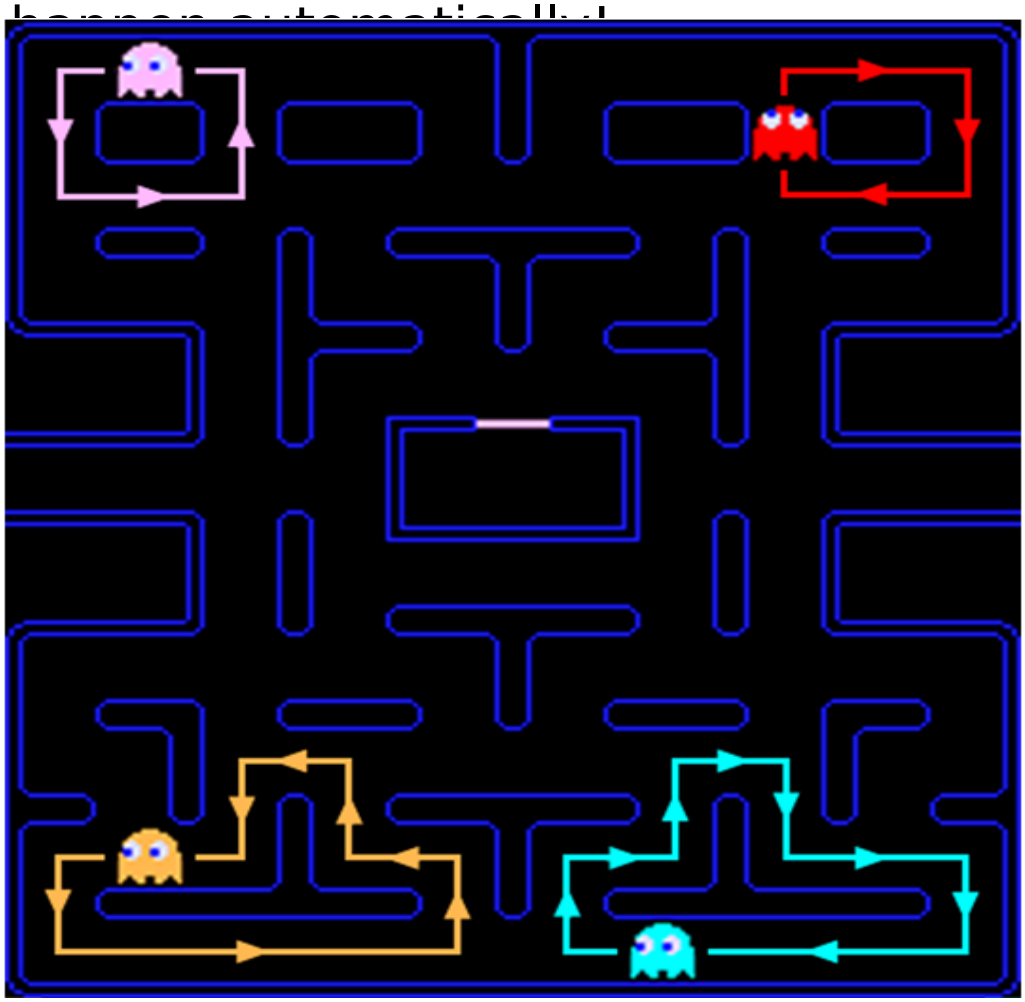
- Chase and Scatter
  - Ghosts switch between these two in normal game play (Ex: 20 seconds chase, 7 seconds scatter)
  - **Ghosts can eat Pacman**
  - How it works: ghosts use BFS to find the



- The breadth first search needs a target
  - We'll explain BFS in the next slides

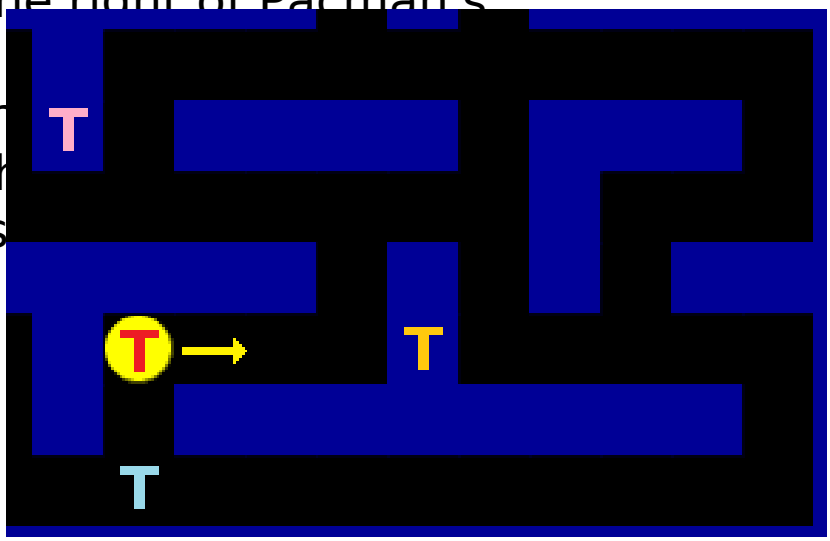
Ghost Target Locations

- Scatter mode
  - Each ghost should target a different corner of the board
  - Ghosts will run in circles during scatter mode (in their corners)-- if you implement BFS correctly, this will happen automatically!



GHOST TARGET LOCATIONS

- Chase mode
  - The targets should be **relative to Pacman's current location**
    - note: the target is constantly changing!
  - The ghosts should all have **unique targets.**
  - Original game's targeting is complicated
    - See extra credit if you're interested
  - Sample target locations:
    -Pacman's location
    - 2 spaces to the right of Pacman's location
    - 4 spaces in fr
    - 3 spaces to th from Pacman's

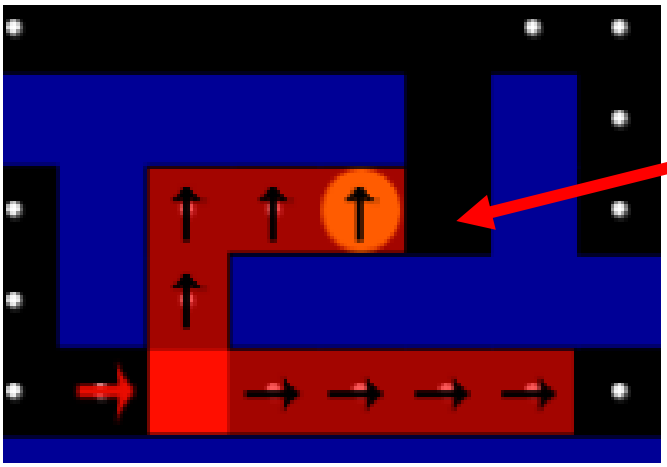


BREADTH-FIRST SEARCH

- Ghosts will use this method to determine where to turn
- In brief, the algorithm involves these main steps:
  - Look at a cell (the current cell), and check if it is the closest to the target
  - For each neighbor of the cell, ***mark*** that neighbor with a ***direction***
  - Repeat these steps with the neighbors
- We'll go through each of the steps in more detail later on
- **Breadth-first** means we check all the neighbors at a particular "depth" before moving on to their neighbors (a level deeper). How can we accomplish this?

DATA STRUCTURES

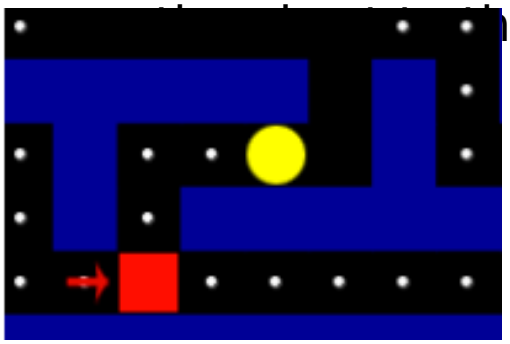
- Use a **queue** to keep track of what cells to check
  - We add the neighbors of the current cell at each iteration
  - Add cells by their *location*
  - Assures that we look at closest cells first
- Use an **array** to store the **initial direction that led to the cell**
  - Cells indexed by location
  - Access best location directly from

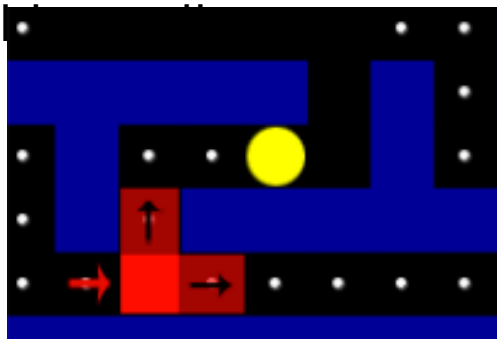


Enqueue this square and add to array — what direction value?

BFS

- **Step 0**: Put *initial* neighbors in the array and the queue
- These cells are neighbors of the cell containing the ghost
  - Cannot be a wall, and **CANNOT** be in the direction opposite the ghost's direction of movement
- These cells need to marked with the *initial* direction
  - In this case, the direction *is* direction from the neighbor to the neighbor cell



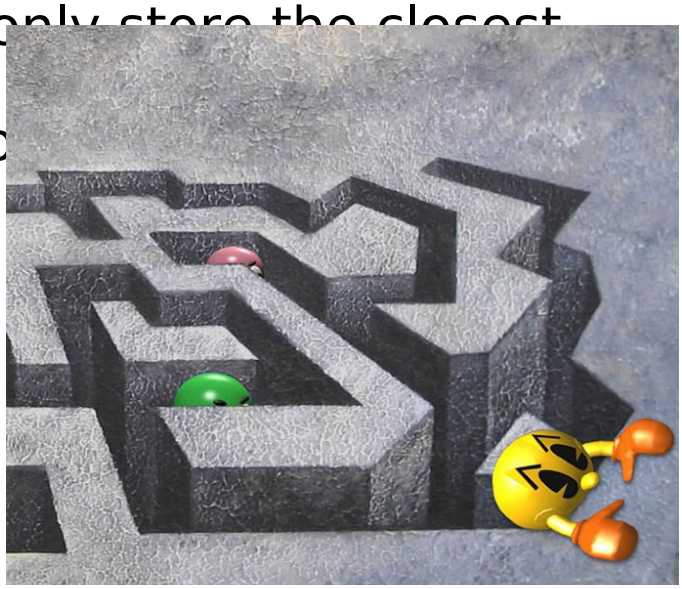


- One more little detail: what do we do about the ghost's cell?

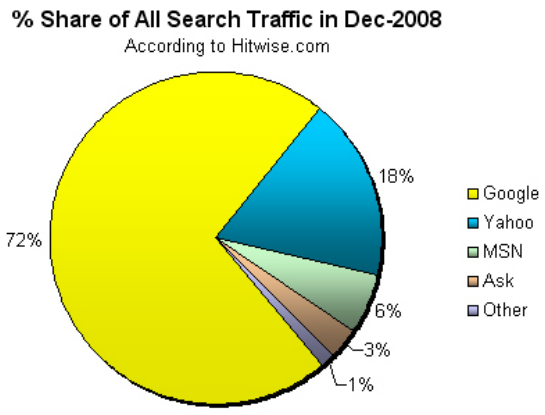
BFS

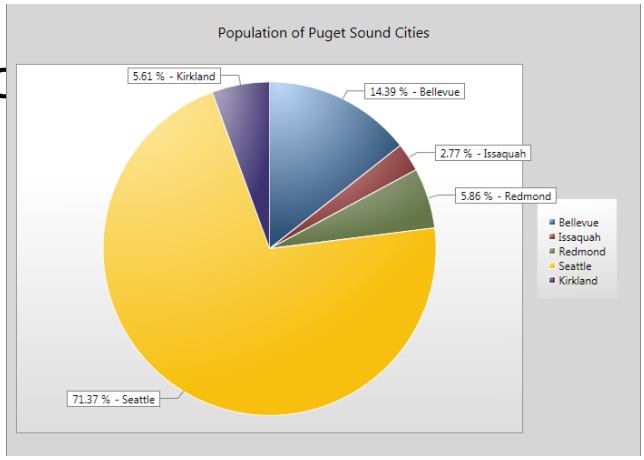
- Now, the steps of our breadth-first search:
- **Step 1**: Check if current cell is closest to the target
  - The distance between two points is easy to calculate
  - How do we keep track of the closest to the target we've gotten? We should use a variable…should it be instance or local?
  - We need not only store the closest ***distance***, we ***cell*** that is clo

will this?



- **Step 2**: Mark neighbors
- First need to determine which neighbors are valid:
  - Cannot be a wall (this is essentially checking for a collision)
  - Must be unvisited
- How do we know if a cell has been visited?
  - Whether or not it is marked with a direction
- Once marked, we need to remember to check it later





BFS

- **Step 3**: Repeat
- When we finish with the current cell, we should move on to the next cell
  - Where is this next cell stored? In the queue!
  - So we need to dequeue, and use that cell as the current cell
- When should we stop?
  - When there are no more cells to check, i.e. the queue is empty
  - How can we rep…s until the queue…

BFS

- Summary: **pseudocode**

for each of the ghost's valid neighboring squares:

    add the square's location (java.awt.Point) to the queue and store the square's direction in the 2D array, indexed by its location

while the queue isn't empty:

    dequeue the next square location

    update the closest distance and the closest point, if necessary

    for each valid neighbor of the current square:

        if the neighbor has not been visited, add its location to the queue and store the current square's direction in the array, indexed by the neighbor's location

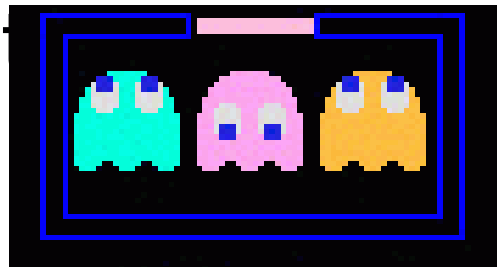
return the direction of the closest point

- This is a complicated algorithm. Make sure you understand it completely before attempting to implement it!

- Note: You do NOT need to write your own queue class! We suggest using the `java.util.LinkedList`, which has `addLast(Object o)` and `removeFirst()` methods (analogous to enqueue and dequeue)

GHOST PEN

- The ghost pen is the place the ghosts start in and where they are returned to when they are eat
Pac-man



Oh hey there's one now!

- The first challenge is to know which Ghost should exit at a given point.

For example, in the beginning of the game we want
Pinky then Inky and then Clyde to exit
n

GHOST PEN

- Also, when two ghosts get eaten, we want the first one eaten to be the first one to exit. What does that sound like?

- Great, so now we know which ghost should exit the pen, but when should it exit?
  - Hint: tick tick tick

- Use a timer! And specifically the pen should have its own timer to make your lives easier

GOOD LUCK!

PAC-MAN