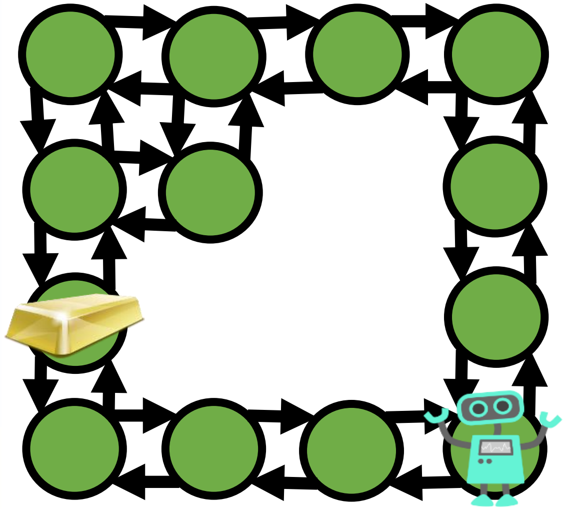
Class Design

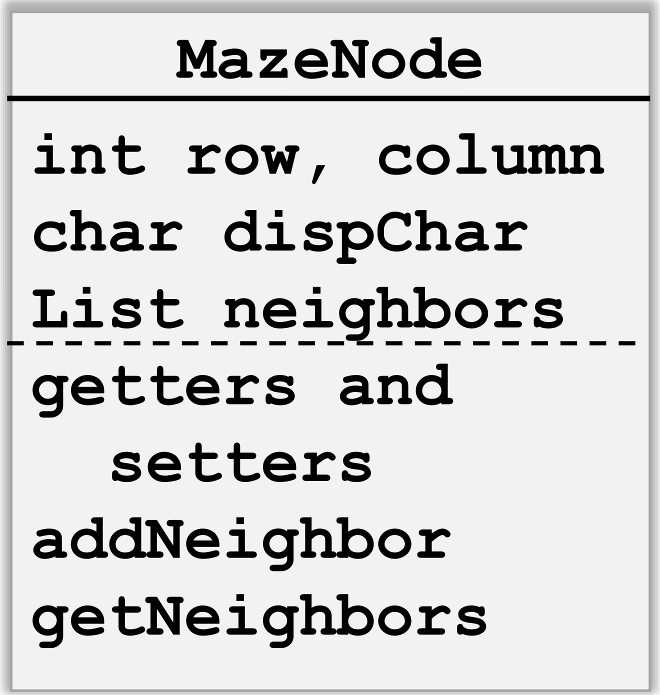
Part 2: Refactoring for better classes

By the end of this video you will be able to…

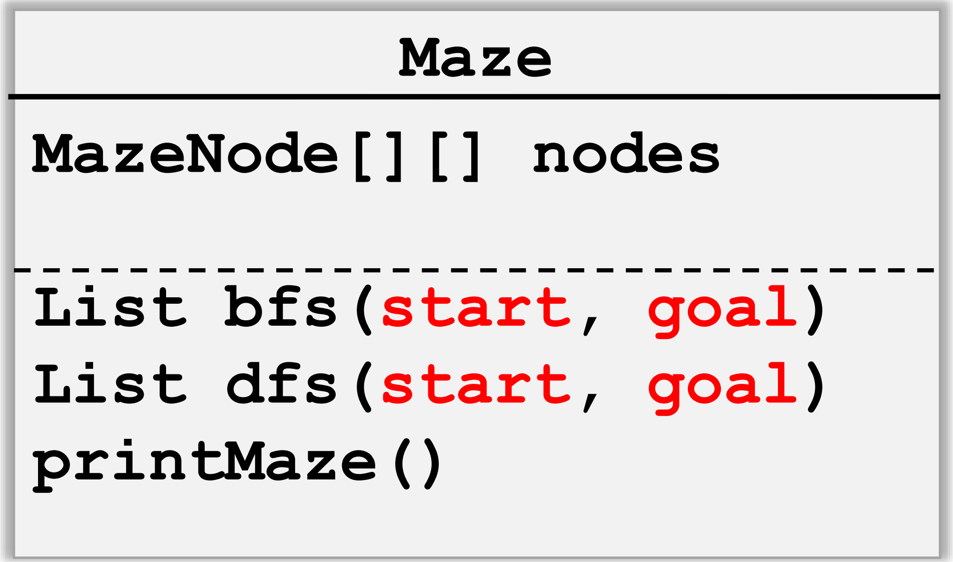
- Explain the concepts of redesign and refactoring
- Critique aspects of code design
- Redesign and refactor code for better design



Check out our code!



```
MazeNode
int row, column
char dispChar
List neighbors
----------------------
getters and
  setters
addNeighbor
getNeighbors
```



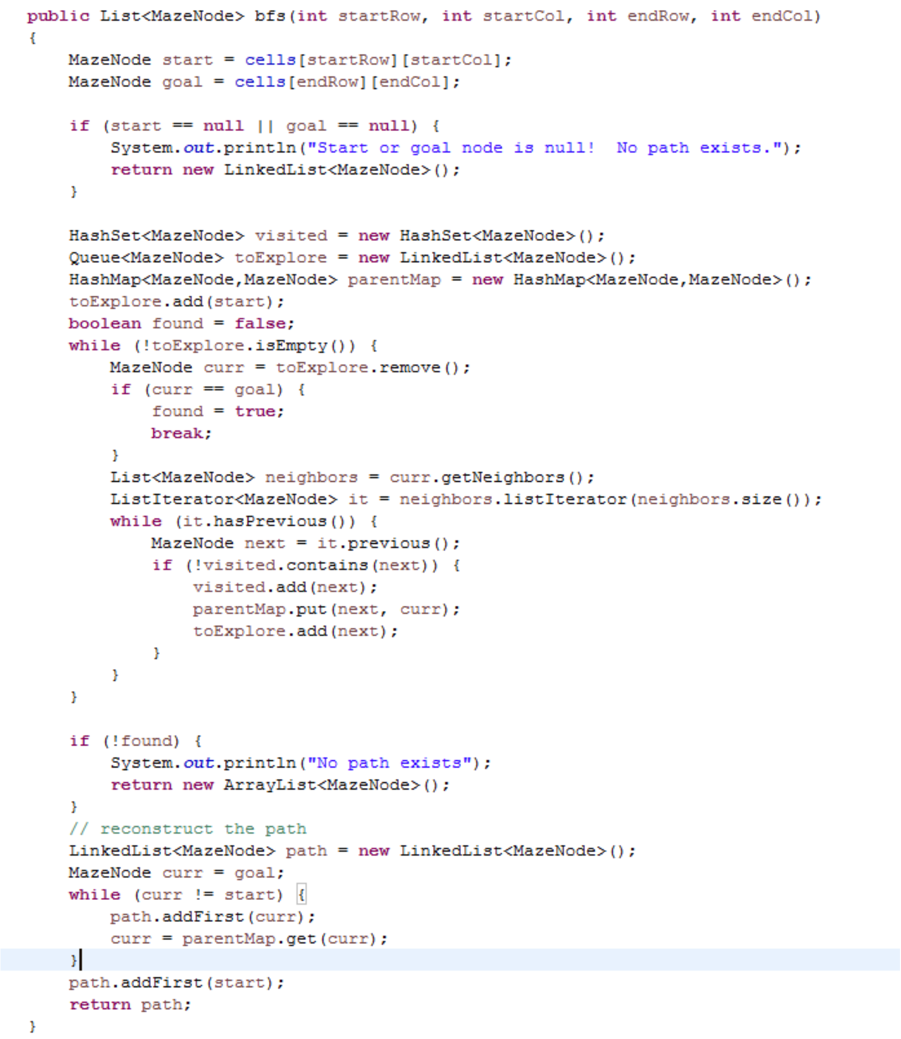
```
Maze
MazeNode[][] nodes

----------------------------
List bfs(start, goal)
List dfs(start, goal)
printMaze()
```

What to look for in a good design

- Objects that make sense, whose data and methods go together
- Interfaces are clean; private data (or data structures) are not exposed
- It's easy and fast to do the operations you want to do
- Methods are short and easy to read and understand

Methods are short and easy to read and understand



```
public List<MazeNode> bfs(int startRow, int startCol, int endRow, int endCol)
{
    MazeNode start = cells[startRow][startCol];
    MazeNode goal = cells[endRow][endCol];

    if (start == null || goal == null) {
        System.out.println("Start or goal node is null!  No path exists.");
        return new LinkedList<MazeNode>();
    }

    HashSet<MazeNode> visited = new HashSet<MazeNode>();
    Queue<MazeNode> toExplore = new LinkedList<MazeNode>();
    HashMap<MazeNode,MazeNode> parentMap = new HashMap<MazeNode,MazeNode>();
    toExplore.add(start);
    boolean found = false;
    while (!toExplore.isEmpty()) {
        MazeNode curr = toExplore.remove();
        if (curr == goal) {
            found = true;
            break;
        }
        List<MazeNode> neighbors = curr.getNeighbors();
        ListIterator<MazeNode> it = neighbors.listIterator(neighbors.size());
        while (it.hasPrevious()) {
            MazeNode next = it.previous();
            if (!visited.contains(next)) {
                visited.add(next);
                parentMap.put(next, curr);
                toExplore.add(next);
            }
        }
    }

    if (!found) {
        System.out.println("No path exists");
        return new ArrayList<MazeNode>();
    }
    // reconstruct the path
    LinkedList<MazeNode> path = new LinkedList<MazeNode>();
    MazeNode curr = goal;
    while (curr != start) {
        path.addFirst(curr);
        curr = parentMap.get(curr);
    }
    path.addFirst(start);
    return path;
}
```

DFS: *not short!*

Solution: Refactor! Restructure code without changing functionality

Methods are short and easy to read and understand

```
public List<MazeNode> dfsRefactored(int startRow, int startCol,
                                    int endRow, int endCol) {
    MazeNode start = cells[startRow][startCol];
    MazeNode goal = cells[endRow][endCol];

    if (start == null || goal == null) {
        System.out.println("No path exists");
        return new LinkedList<MazeNode>();
    }

    HashMap<MazeNode,MazeNode> parentMap = new HashMap<MazeNode,MazeNode>();
    boolean found = dfsSearch(start, goal, parentMap);

    if (!found) {
        System.out.println("No path exists");
        return new LinkedList<MazeNode>();
    }

    return constructPath(start, goal, parentMap);
}
```

DFS: *short!*

Interfaces are clean
Private data (or data structures) are not exposed

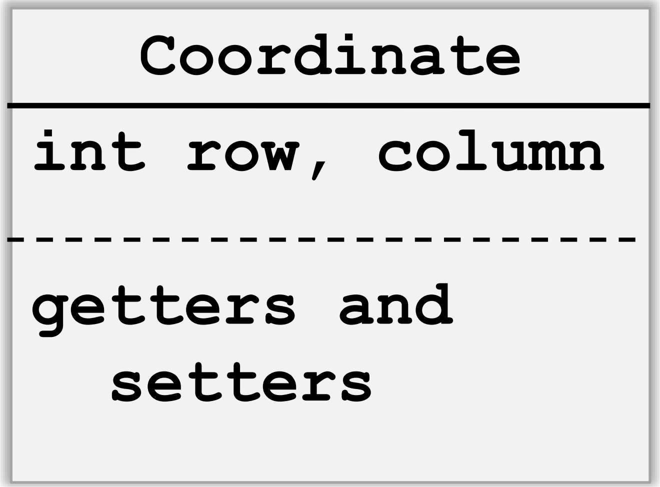
```
public class Maze
{
      ...

      /** Return the path from start to finish */
      public List<MazeNode> bfs(int startRow, int startCol, int endRow, int endCol)
      {
```

Interfaces are clean
Private data (or data structures) are not exposed

```
public class Maze
{
    ...

    /** Return the path from start to finish */
    public List<MazeNode> bfs(int startRow, int startCol, int endRow, int endCol)
    {
```



Coordinate

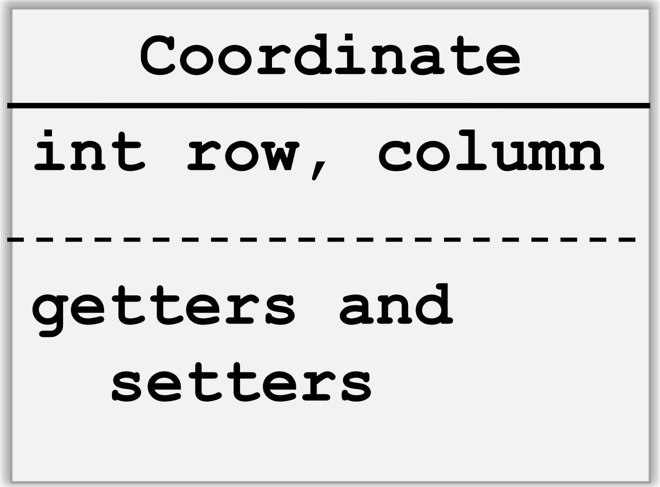
int row, column

getters and setters

Interfaces are clean
Private data (or data structures) are not exposed

```
public class Maze
{
    ...

    /** Return the path from start to finish */
    public List<Coordinate> bfs(Coordinate start, Coordinate end)
    {
```



Coordinate

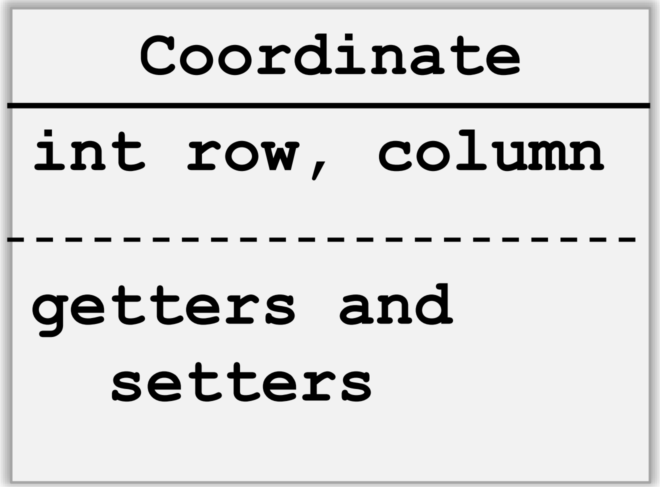
int row, column

getters and setters

Example of Code Redesign (changes the interface)
OK during development. Difficult after release.

```
public class Maze
{
     ...

     /** Return the path from start to finish */
     public List<Coordinate> bfs(Coordinate start, Coordinate end)
     {
```



Coordinate

int row, column

getters and setters

What to look for in a good design

- Objects that make sense, whose data and methods go together
- Interfaces are clean; private data (or data structures) are not exposed
- It's easy and fast to do the operations you want to do
- Methods are short and easy to read and understand

Don't be afraid to redesign and refactor as you go!