Abstract Classes
Mazes and Recursion
Review

Review

- Stacks -- ch 5,6
- Queues -- ch 7,8
- Recursion -- ch 9
  - Search, Binary Search -- ch 19

- Java
  - Comparable -- Interlude 5
  - Abstract Classes & Interfaces -- Interlude 7

Abstract classes

- A class that should/can NEVER be instantiated.
  - From the Pets example
    - Pet, Dog should be defined as abstract classes
      - The only instances of each of these should be from more specific classes.
      - In taxonomy kingdom, phylum or division, class, order, family, and genus should all be abstract
        - only species should have instance

AbStract Classes

Pt 2

	Interface	Abstract Class	Class
stub methods	YES	YES	NO
full methods	NO	YES	YES
Instance Variables	NO	YES	YES
Multiple inheritance from	YES	NO	NO
Instantiatable	NO	NO	YES
Has Constructors	NO	NO	YES
May implement interfaces	YES	YES	YES
May extend classes	NO	YES	YES

Recursion

- Write a function to find the maximum value in an ArrayList (List151Impl) of Integers
- Do this using recursion
- Start with `public int maxValue(ArrayList<Integer> lstOfInts)`
    - Use a private utility function to actually do the recursion
- Show all of the function calls and returns (program stack operations) given an ArrayList containing [7,9,3,5,6]

Abstract Class

AbstractExample

```java
public abstract class AbCl {
    private double km;

    public double getKM () {
        return km;
    }

    public double getMiles() {
        return km * 1.62;
    }

    /**
     * A really long comment so that implementers know exactly what to do
     * @param aaa
     * @param bbb
     */
    public abstract void populate(int aaa, int bbb);
}
```

Recursion and Backtracking

- All problems considered so far progress steadily towards an answer.
- Consider a maze. Sometimes you need to “backtrack”.
    - RECURSION makes backtracking easy!
- Idea:
    - 1. Somehow make a copy of where you are
    - 2. Identify all of the possible moves you can make
    - 3. Try to go forward one step.
        - A. If you can go forward … ,
            - If needed, mark your step on the copy.
            - return to step 1
        - B. If failure --
            - try a different forward step
    - 4. If you run out of forward steps, backtrack

- Twiddle
    - especially with mazes mark places you have been so you do not retry failed paths

Maze

basics

```java
public class Maze {
    // Letters indicating state in the maze
    public static final char START = 's';
    public static final char FINISH = 'e';
    public static final char WALL = '*';
    public static final char PATH = ' ';
    public static final char USED_PATH = 'A';

    /**
     * The internal representation of the maze
     */
    char[][] mazeRep;

    /**
     * The directions of allowed movement in the maze There are two set of
     * directions
     */
    Coordinate[] diagonal = { new Coordinate(-1, -1),
new Coordinate(1, -1), new Coordinate(1, 1),
            new Coordinate(-1, 1) };

    Coordinate[] updownrl = { new Coordinate(1, 0),
new Coordinate(-1, 0), new Coordinate(0, 1),
            new Coordinate(0, -1) };

    Coordinate[] moves = diagonal;
```

```java
public class Coordinate {
   private final int d1;
   private final int d2;
   public Coordinate(int d1, int d2) {
        this.d1 = d1;
        this.d2 = d2;
    }
   public Coordinate(Coordinate starter, Coordinate adder) {
        this.d1 = starter.getD1() + adder.getD1();
        this.d2 = starter.getD2() + adder.getD2();
    }

    public int getD1() {
        return d1;
    }

    public int getD2() {
        return d2;
    }

    @Override
    public String toString() {
        return "<" + d1 + "," + d2 + ">";
    }

}
```

Maze

Constructors

```
// read maze from a file

public Maze(String fileName) throws FileNotFoundException,
ImproperMazeException, IOException {
        // first read the file into an array list to get the dimensions of
the maze
        ArrayList<String> tmaz = new ArrayList<>();
        BufferedReader br = new BufferedReader(new FileReader(fileName));
        int wid = -1;
        String lin;
        while ((lin = br.readLine()) != null) {
            if (wid > 0 && lin.length() != wid) {
                throw new ImproperMazeException(
                        "Maze must be a rectangle current width=" + wid +
" new line width=" + lin.length());
            }
            if (wid < 0)
                wid = lin.length();
            tmaz.add(lin);
        }
        br.close();

        // with the file read complete, fill in the internal
representation
        mazeRep = new char[tmaz.size()][wid];
        for (int i = 0; i < tmaz.size(); i++) {
            String s = tmaz.get(i);
            for (int j = 0; j < s.length(); j++) {
                if (s.charAt(j) == WALL || s.charAt(j) == START ||
s.charAt(j) == FINISH || s.charAt(j) == PATH) {
                    mazeRep[i][j] = s.charAt(j);
                } else {
                    throw new ImproperMazeException("Maze contains an
unknown character ||" + s.charAt(j) + "||");
                }
            }
        }
    }
```

```
// Copy Constructor


public Maze(Maze oldMaze) {
        mazeRep = new char[oldMaze.getDim1()][oldMaze.getDim2()];
        for (int i = 0; i < getDim1(); i++) {
            for (int j = 0; j < getDim2(); j++) {
                mazeRep[i][j] = oldMaze.mazeRep[i][j];
            }
        }
    }
```

Maze

Utilities

```
public void setCoordValue(Coordinate c, char val) {
    mazeRep[c.getD1()][c.getD2()] = val;
}

/** The size in the first dimension of the maze */
private int getDim1() {
    return mazeRep.length;
}

/** The size in the second dimension of the maze */
private int getDim2() {
    return mazeRep[0].length;
}

  /**
 * Find the location of the starting point
 *
 * @return the location of the starting point
 */
public Coordinate getStart() {
    for (int i = 0; i < mazeRep.length; i++)
        for (int j = 0; j < mazeRep[i].length; j++)
            if (mazeRep[i][j] == START)
                return new Coordinate(i, j);
    return null;
}
```

Maze

Solver

```java
public void solve() {
    Coordinate mstart = getStart();
    if (mstart != null) {
        setCoordValue(mstart, PATH);
        Maze zz = solveUtil(mstart, 0);
        if (zz != null) {
            System.out.println(zz);
            return;
        } else {
            return;
        }
    } else {
        System.out.println("No starting point");
        return;
    }

}
```

```java
private Maze solveUtil(Coordinate c, int depth) {
    // System.out.println("Considering " + c);
    if (c.getD1() < 0 || c.getD2() < 0 || c.getD1() >= getDim1() || c.getD2() >= getDim2()) {
        // System.out.println(" Off Grid");
        return null;
    }
    if (mazeRep[c.getD1()][c.getD2()] == FINISH) {
        setCoordValue(c, USED_PATH);
        System.out.println("Solved in " + depth + " steps!!!");
        return this;
    }
    if (mazeRep[c.getD1()][c.getD2()] == WALL) {
        // System.out.println(" Wall");
        return null;
    }
    if (mazeRep[c.getD1()][c.getD2()] == PATH) {
        setCoordValue(c, USED_PATH);
        for (Coordinate mv : moves) {
            Maze mm = (new Maze(this)).solveUtil(new Coordinate(c, mv), depth + 1);
            if (mm != null)
                return mm;
        }
    }
    // System.out.println(" No way forward");
    return null;
}
```