CS 2510 Exam 3 *SOLUTION* Spring 2011

Name: _____

Student Id (last 4 digits): _____

Section (Proulx / Chadwick): _____

• ***Read the questions carefully*** and write down your answers in the space provided.
• You may use all parts of the Java language we have learned. If you need a method and you don't know whether it is provided, define it. You do not need to include the curly braces for every **`if`**/**`else`**, as long as the statements you write are correct in standard Java.
• For tests you only need to provide the expression that computes the actual value, connecting it with an arrow to the expected value. For example `s.method() -> true` is sufficient.
• Remember that the phrases "design a class" or "design a method" mean more than just providing a definition. It means to design them according to the **design recipe**. You are *not* required to provide a method template unless the problem specifically asks for one. However, be prepared to struggle if you choose to skip the template step.
• We will not answer *any* questions during the exam. If a problem seems ambiguous, make an intelligent decision and document your assumptions.

Good luck.

Problem	Points	/
1		/ 6
2		/ 12
3		/ 12
4		/ 16
5		/ 6
Total		/ 52

A startup game-design company, *B-Mazing*, is working on a simple maze game that teaches children (or undergraduates) to follow the directions. According to the CEO, a *maze* is a rectangular grid with a path that the player must follow. The *path* is a series of moves that starts in the upper-left corner of the grid (where the player starts) and ends in the lower-right corner. To keep things simple (for the undergrads) the paths will never contain loops, short-cuts, or dead-ends: there is only one *next* direction along the path (though you will not need to enforce this).

During the game, the player hits one of the arrow keys (`"up"`, `"down"`, `"left"`, or `"right"`). If the key corresponds to the next direction on the path then player advances along the path, otherwise the player looses a *life*. The game ends when the player has reached the end of the path (lower-left corner), or when the player runs out of lives.

The three pictures below show the prototype implementation at the start, middle, and completion of a successful game:

Unfortunately, the prototype is very limited, so the company wants you to design classes and methods to help implement the game. Fortunately, your manager understands the power of programming to interfaces and she has designed the interface on the next page to represent maze-paths in the game.

```
interface IMaze extends Iterable<String>{
   /** Returns an iterator for the directions/path through
    *    this maze.  The iterator specifies the directions
    *    (in order) that the player must move.
    *  Repeated obligation from Iterable<String> */
   Iterator<String> iterator();

   /** Is this maze's path valid for its width and
    *    height? (Is it completely on the grid?) */
   boolean isValid();

   /** Returns the width of this maze grid */
   int getWidth();

   /** Returns the height of this maze grid */
   int getHeight();

   /** Set the lives for the player */
   void setLives(int lives);

   /** Returns the player's lives */
   int getLives();

   /** Record a bad move by decreasing the player's lives */
   void badMove();
}
```

Problem 1 6 Points

Your team has decided that you will (first) be responsible for the *model* portion of the game: representing the size of the grid and the path that a player will follow. Design the class `Maze` that implements the `IMaze` interface.

For this problem you only need to give a data definitions and examples (class definition(s), fields, constructor(s), and instances). One of your examples **must** be the small maze/path represented in the earlier pictures.

Hint: Read the game description again, *carefully*, in order to decide what/how information should be represented in your class. Think about the methods that must be implemented from the interface (for later questions). Using/defining/designing helper classes is Ok.

Solution

```
class Maze implements IMaze{
    int width;
    int height;
    int lives;
    ArrayList<String> path;

    Maze(int width, int height, int lives, ArrayList<String> path){
      this.width = width;
      this.height = height;
      this.lives = lives;
      this.path = path;
    }
}

class Examples{
   IMaze smallmaze;

   void init(){
     ArrayList<String> smallpath = new ArrayList<String>();
     smallpath.add("down");
     smallpath.add("down");
     smallpath.add("right");
     smallpath.add("up");
     smallpath.add("right");
     smallpath.add("right");
     smallpath.add("down");
     smallpath.add("down");
     this.smallmaze = new Maze(4, 4, 3, smallpath);
   }
}
```

... This page is intentionally blank for your work ...

Problem 2 12 Points

Design the method `isValid` for your `Maze` class. A valid maze path ends in the lower-right corner and never leaves the bounds of the game grid (when starting from the upper-left corner). Again, *you may assume the path is free from loops/dead-ends*, you should only check the bounds.

Note 1: You should be able to design this method using only the methods provided by the `IMaze` interface.

Note 2: It may be useful to design a helper class to keep track of the *row* and *column* of the maze path — feel free to add it to your design.

Solution

```
boolean isValid(){ ... }
```

... This page is intentionally blank for your work ...

Problem 3 — 12 Points

Design the remaining methods from the `IMaze` interface (i.e., `iterator`, `getWidth`, `getHeight`, `setLives`, `getLives`, and `badMove`).

Solution

```
/** Returns an iterator for the path through this maze. */
Iterator<String> iterator(){ return this.path.iterator(); }

/** Returns the width of this maze grid */
int getWidth(){ return this.width; }

/** Returns the height of this maze grid */
int getHeight(){ return this.height; }

/** Set the lives for the player */
void setLives(int lives){ this.lives = lives; }

/** Returns the player's lives */
int getLives(){ return this.lives; }

/** Record a bad move by decreasing the player's lives */
void badMove(){ this.lives--; }
```

... This page is intentionally blank for your work ...

Problem 4 16 POINTS

Now that you have an implementation of `IMaze`, you are ready to design some of the behavior of the *game*. The designer of the drawing method needs your game class to contain (1) an `IMaze` so that she can correctly draw the grid/path, (2) a representation of the rest of the path to be traveled, and (3) the player's current location. But, you may add other fields as needed.

The display designer will provide you with a **`void`** method `showMove()` method, your task is to design the class `MazeGame`, and two methods: `onKey` and `stopWhen`.

A. Design `MazeGame` and provide examples.

Solution

```
class MazeGame{
   IMaze maze;
   ...
}
```

B. Design the method **`void`** `onKey(String)`, that responds to key events as follows:

- The game should only respond to the four arrow keys: `"up", "down", "left", "right"`.
- If the key is in the direction of the next maze path, move the current location accordingly and invoke the provided `showMove` method. *Note*: you need to keep track where you are on the path along the maze.
- If the player moves in a direction other than given by the next maze path, leave the player at the current location, and invoke the `badMove` method on the game's *maze*.

Solution

```
void onKey(String key){ ... }
```

C. Design the method: **`boolean`** `stopWhen()` that produces a boolean value **`true`** when the game has ended. There are possible endings to the game:

- The player reaches the end of the maze path successfully, or
- The player has no lives left

Solution

```
boolean stopWhen(){ ... }
```

... This page is intentionally blank for your work ...

... This page is intentionally blank for your work ...

Problem 5 6 Points

Implement the method **`boolean noLoops()`** for your `Maze` class that verifies that this maze's path does not contain any loops, i.e., it never goes through the same grid point twice.

Library classes/interface reminders:

```
/*** class ArrayList<T>: ***/

    /** Remove all elements from this list */
    void clear();

    /** Add the element at the end of this list */
    boolean add(T t);

    /** Add the element at the given index, shifting all
     *    items after to the right */
    void add(int index, T t);

    /** Get the element at the given index */
    T get(int index);

    /** Set the element at the given index to the given
     *    value and return the index's previous value */
    T set(int index, T t);

    /** Returns true if this list contains no elements */
    boolean isEmpty();

    /** Removes (and returns) the element at the given index */
    T remove(int index);

    /** Returns the number of elements in this collection */
    int size();

/*** interface Iterable<T>: ***/

    /** Returns the iterator for this data set */
    Iterator<T> iterator();

/*** interface Iterator<T>: ***/

    /** Are there additional items available this data set */
    boolean hasNext();

    /** Produce the next item in this data set
     * EFFECT: modify this iterator to point to the next item */
    T next();
```