# CS100M Spring 2004
# Assignment 6: Return of The ADT

## 0. Introduction

### 0.1 Goals

This assignment will help you develop further OOP skills with arrays, strings, and subtyping. Be sure to develop your code by stubbing classes and methods. Write each method one at a time, testing your code along the way.

### 0.2 Instructions

Be sure to read the *entire* assignment before answering the questions! Do the tasks in the following sections. You may work with one partner or by yourself for this assignment.

### 0.3 Submission

Follow the **Submission Format Requirements** at the CMS Info link on the course website. The last section describes what to submit on CMS.

### 0.4 Grading

All code that you submit must run without warnings and errors. Otherwise, you will receive a zero. This assignment's weight counts as 1.5 times the amount of a regular assignment. According to the Syllabus, you may not drop this assignment. There are several bonus point options as well.

### 0.5 Academic Integrity

You must abide by the Code of Academic Integrity, which is provided for CS100M, the Department of Computer Science, and Cornell University on our course website. Refer to the link called Academic Integrity.

## 1. Background

There is a classic game that involves a square grid of consecutively numbered tiles that slide in perpendicular directions, because one spot is ***blank***. This blank tile is really just an open space. For example, a $3 \times 3$ grid from initial state, ***solved state*** might look something like this:

```
123
456
78
```

Note that a blank space represents the blank tile location. To scramble, the puzzle, you are not allowed to exchange any tile by lifting it off of the board. Instead, you must slide a tile in the blank. Then, slide a tile into a new blank, and so forth. For example, three scrambling moves in the above puzzle would produce this ***scrambled state***:

```
1 3
425
786
```

Can you see the moves that produced this state? My sequence of moves was 6-down, 5-right, and 2-down. To condense this language, I will say the following directions:

- **N**: move a tile ***north***, or up, into a blank space.
- **S**: move a tile ***south***, or down, into a blank space.
- **E**: move a tile ***east***, or right, into a blank space.
- **W**: move a tile ***west***, or left, into a blank space.

So, moving a tile means that you find a tile *adjacent* to the blank and then exchange places with the blank.

As you scramble more and more tiles, the solution usually becomes more and more difficult for someone who does not know the scrambling sequence. For more information about the history and national obsession that struck about 100 years ago, check out http://www.holotronix.com/samlloyd15.php. You will also discover an explanation why an illegal exchange of tiles can prevent a solution from ever being found. This toy makes an simple and elegant computer game, which you will design in this assignment.

## 2. Puzzle ADT

There are numerous ways to design the software for this assignment. We need you to practice using interfaces and inheritance, where you try various implementations of the puzzle ADT. Recall that an ADT is a high-level specification of the data and operations that are associated with a type that you intend to define and ultimately implement as code.

Puzzle data:

- grid size
- numbered tiles based on grid size
- others?

Puzzle operations:

- move a tile into the blank (and the blank to the previous location of that tile)
- check if a puzzle is solved
- scramble a puzzle
- reset the puzzle
- display the current puzzle
- others?

So, you will need to design a class with fields for the data (and possibly other items) and methods for the operations (and possible other actions).

You have a slight problem, however. How should the physical state of the puzzle be represented? The quickest, most obvious answer is an array because of the grid. However, lest you be tempted by the Darkside of The Array, which is quicker and easier, there are other structures that can store the same information with less memory consumption. What about a **`String`**? Rather than worry about memory consumption for several elements, you can actually store the entire puzzle state as a single value. In fact, since **`String`**s are objects, you can make the memory management even more streamlined by using an integer, which is given as bonus problem.

Regardless of the choice of internal data representation, the principle of abstraction dictates that your “connection” to an external client (another class) should remain unchanged. So, the programmer of the Main Class should not have to worry how the move, check, scramble, reset, and display methods all access and manipulate the internal representation of the Puzzle. Instead, the client should simply know the purpose and header of the methods in case the server (the Puzzle class) should change the internal representation of its data.

## 3. Implementation: Class Stubs

Given the requirement that a programmer must be given a collection of known, unchangeable method headers to design the Main Class, we define the following interface **`IPuzzle`** for all Puzzle classes:

```
public interface IPuzzle {
   public void reset();            // restore puzzle to the sorted state
   public boolean isSolved();      // return true if puzzle is in the sorted state
   public boolean move(char dir);  // move a tile into the blank position
   public void scramble();         // scramble the puzzle
   public void display();          // print out current puzzle
}
```

All puzzle classes must abide by this contract so that Main Class can keep using the same methods. So, we provide rough class stubs for the two Puzzle representations that we want you to use. In each class, below, you will see that the class implements the **`IPuzzle`** interface, which means that the class must provide methods with the headers as those in the interface:

```
class PuzzleAsArray implements IPuzzle {
   private char[][] puzzle;
   // more fields
   // constructors
   // methods from IPuzzle
   // utility methods
}

class PuzzleAsString implements IPuzzle {
   private String puzzle;
   // more fields
   // constructors
   // methods from IPuzzle
   // utility methods
}
```

The integer representation of a Puzzle is given as a bonus problem. Note that the Main Class does not need to implement the **`IPuzzle`** interface, because the class uses the other Puzzle classes but not actually represent a Puzzle itself!

```
public class PlayPuzzle {
   // fields?
   public static void main(String[] args) {
   // statements
   }
   // constructors?
   // methods?
}
```

# 4. Implementation: `IPuzzle` Methods

This section provides a bit more detail about each method in **`IPuzzle`** that you will be implementing in your Puzzle classes. Note that you may need additional helper, or ***utility***, methods in the classes to assist with the methods required by the interface.

## 4.1 Method `reset`

The reset method restores the puzzle to the solved state:

- For **`PuzzleAsString`**, the solved state will be **`"12345678 "`**, where **`" "`** (a string composed of the blank character) represents the blank.
- For **`PuzzleAsArray`**, use an array of *characters* with a space character representing the blank. The array will have a grid pattern that resembles the grid in Section 1.

## 4.2 Method `isSolved`

The **`isSolved`** method compares the current state of the Puzzle to the solved state and returns a boolean value.

## 4.3 Method `move`

As explained in Section 1, the **`move`** method takes the input move called **`dir`** and moves a tile that is adjacent to the blank into the blank's position. The position of the tile that was moved is now the new position of the blank. Section 1 gives additional terminology for the direction of moves. If the move that is attempted is impossible, the method returns **`false`**. Otherwise, if the method succeeds, it returns **`true`**.

## 4.4 Method `scramble`

The **`scramble`** method scrambles the Puzzle by making a sequence of random, but *legal*, moves based on a difficulty that user sets (described in Section 5). You do *not* have to check for poor selections of random moves, as in moves that inadvertently reverse other scrambles. Note that you need to store the sequence of moves to scramble to Puzzle if you need to reverse these moves. By "remembering" the sequence of moves made by scrambling combined with the moves made by the user, you easily solve the puzzle!

## 4.5 Method `display`

Display the Puzzle by printing it out to the screen. For example, the $3 \times 3$ puzzle in the solved state would output as follows:

```
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
| 4 | 5 | 6 |
+---+---+---+
| 7 | 8 |   |
+---+---+---+
```

Adventurous students who wish to improve the interface may do additional work for bonus points by developing a GUI. Refer to the Section 7 for more information.

# 5. Main Class

As introduced in Section 3, **`PlayPuzzle`** is the Main Class. We divide this section into two parts:

- How to call **`PlayPuzzle`** from the command-line to set various parameters
- How the game should play based on your chosen parameters.

## 5.1 Setting Game-Play Parameters

Your program will use either command-line arguments or default settings to select parameters that determine the following:

- *type* of game to create: **`"array"`** or **`"string"`**.
- *level* of difficulty: the number of moves to randomly scramble the Puzzle.
- *size* of grid: 2 or 3.

For example, the command-line

```
> java PlayPuzzle array 10 3
```

uses **`PuzzleAsArray`**, which starts with a $3 \times 3$ Puzzle that has been scrambled 10 times. If the user specifies no command-line arguments, your program will set defaults of type **`"array"`**, level **`3`**, and size **`3`**. You must check the inputs for illegal values. If the user enters illegal input(s), alert the user and exit the program. You may also to use **`SavitchIn`** to prompt the user to fix the input.

## 5.2 Game Play

Assuming the user supplied a legal inputs, the program scrambles the *solved* puzzle (the initial state, which is not displayed), and displays it the user. The user can then enter a few different inputs:

- **`N`**, **`S`**, **`E`**, or **`W`**: Move a tile. Refer to Section 1.
- **`Q`**: Quit the game immediately.
- **`R`**: Restart the game by re-scrambling the Puzzle.
- **`A`**: Automatically solve the Puzzle from the current state.
- **`?`**: Help! Display an explanation of these input selections.

If the user makes an illegal selection or move, the program will prompt the user to enter another input. The input by the user should not be case-sensitive. Do not let the user enter anything about single letters.

Example session:

```
> java PlayPuzzle string 2 2

Welcome to the Puzzle Game!
Current state of puzzle:
   +---+---+
   |   | 2 |
   +---+---+
   | 1 | 3 |
   +---+---+
Input [NSEW QRA?]> N
```

```
Current state of puzzle:
   +---+---+
   | 1 | 2 |
   +---+---+
   |   | 3 |
   +---+---+

Input [NSEWQRA]? W

Current state of puzzle:
   +---+---+
   | 1 | 2 |
   +---+---+
   | 3 |   |
   +---+---+

Congratulations! You solved the puzzle. Good-bye!
```

## 6. Automatic Solution

As part of the allowed input, the user can enter **A**, which activates the ***automatic solution***. The program will then backtrack all of the moves made since the solved state.

How does the program know how to automatically solve the program from the current state? During the scrambling, you can keep track of each move in a dynamic data structure. By keeping track of all the moves performed on the puzzle (from scrambling to the user's moves), you can simply reverse each move. Start from the last move and work your way all back to first move, in order.

How should your program "remember" the entire collection of moves? Since you don't know for certain the number of scrambles combined with the number of user moves, you need a dynamic data structure! So, you can use a **`Vector`**, **`List`**, and even a **`Stack`**, so long as you maintain a LIFO structure. Your program does *not* need to be smart about how the user made their guesses. For example, if the user goes **NSNSNS**…, your program will go **SNSNSN**… to reverse those moves.

Each time the program reverses a move, the program must display the new state of the puzzle. You should introduce a brief pause between each display so that the user can see order of moves being reversed.

## 7. Bonus Opportunities

For each of the following tasks, we will not award partial credit. So, to receive bonus points on a particular task, your work must be very solid and well tested. You must also provide a readme.txt file that explains what you are submitting for extra credit:

- 20 points: Include class **`PuzzleAsInt`**, using an integer representation of a Puzzle. The command-line will accept **`int`** to choose this Puzzle type.

- 20 points: Write another program **`PuzzleGUI.java`** that uses a GUI to play the game using the array representation of the Puzzle. Include several buttons that provide flexibility to the user for a variety of modes.
- 20 points: Animate the solution process in your GUI.
- 20 points: Allow the user to upload a figure, which is divided into the tiles instead of using numbers.
- 20 points: Bonus-bonus points! If you do all 80 bonus points completely, you will receive another 20 bonus points.

## 8. Submitting Your Work

Submit a zip file called **`a6.zip`** that contains these files:

- **`PlayPuzzle.java`**
- **`PuzzleAsArray.java`**
- **`PuzzleAsString.java`**
- optional: bonus work

Do not submit **`IPuzzle.java`**, though we will test your code with our version. Do not submit any other files! As a reminder (see first page), refer to **Submission Format Requirements** on CMS Info on the course website before submitting any work! You will find some differences for Java programs.