# ITMD 510 MP 1 – Monopoly

Due: October 29, 2015 6:25pm (**No Late Submissions**)

## 1 Objectives

The objectives of this MP are to demonstrate mastery of the following course objectives:

- Implement Java applications that leverage the object-oriented features of the Java language
- Understand and implement applications that use the Java collections framework.

## 2 Introduction

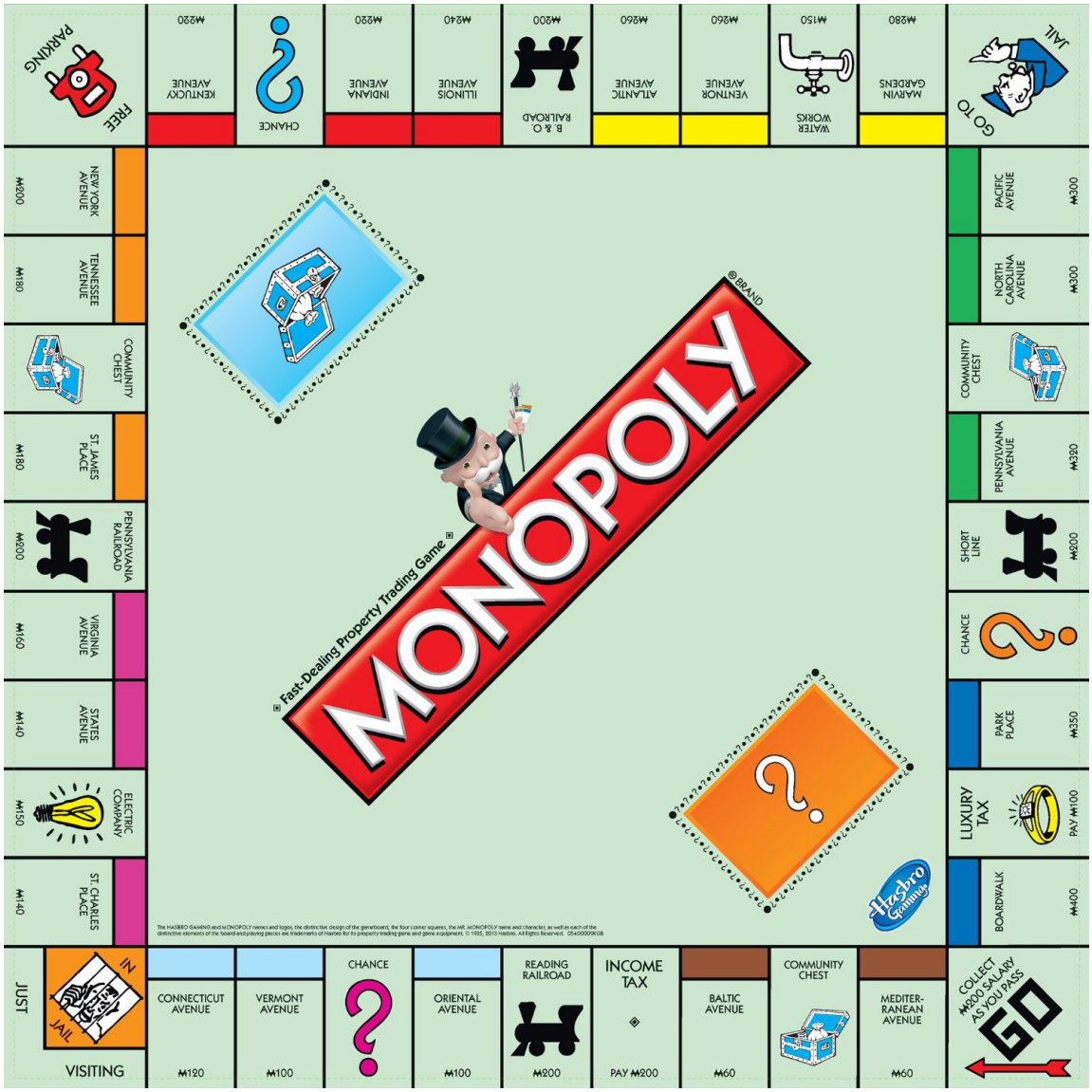
Themed slot machines have been a staple of casinos for decades. One such popular theme is the game of Monopoly. Each Monopoly themed slot machine has a bonus round that is triggered by certain symbols showing up on the reels during a spin.

A favorite bonus round of slot machine enthusiasts is the Once Around Bonus. In this bonus, a player token starts at Go and Mr. Monopoly rolls the dice to move the token around the board. Instead of the player paying for properties, taxes, utilities, etc, the game pays the player based on the type of space the player lands on. When the player reaches or passes Go, the bonus round is over and the player collects his winnings.

For this MP, you will implement the logic for the Once Around Bonus game.

## 3 Rules

The Monopoly board used for the Once Around Bonus game looks like the following:




The bonus round proceeds as follows:

1. At the start of the bonus round, the player's token is at Go.
2. Two six-sided dice are rolled and the total of the dice is the number of spaces the player's token is moved. For example, if the dice are 2 and 3, the token is moved 5 spaces from Go to Reading Railroad.
3. Based on the type of space the player lands on, the following could occur:
   (a) Property spaces, railroads, utilities, Income Tax, Luxury Tax, and Go spaces all pay some number of credits based on the rules for that space, detailed below.

(b) Landing on Chance or Community Chest results in some random action that could pay the player credits, move the token back three spaces, go to the Just Visiting/Jail space, etc.

4. Steps 2 and 3 are repeated until the player passes Go. When the player passes Go, the player is awarded 200 extra credits and the bonus round ends.

5. The player is then awarded the sum total of all credits awarded during all of the player's rolls.

## 3.1 Rules for Spaces

**Property Spaces**

The spaces with the colored bar in the top of the space is a property space (for example, Mediterranean Avenue). The property spaces pay the player an amount of credits based on the table below:

| Name | Credits |
|---|---|
| Mediterranean Avenue | 60 |
| Baltic Avenue | 60 |
| Oriental Avenue | 100 |
| Vermont Avenue | 100 |
| Connecticut Avenue | 120 |
| St. Charles Place | 140 |
| States Avenue | 140 |
| Virginia Avenue | 160 |
| St. James Place | 180 |
| Tennessee Avenue | 180 |
| New York Avenue | 200 |
| Kentucky Avenue | 220 |
| Indiana Avenue | 220 |
| Illinois Avenue | 240 |
| Atlantic Avenue | 260 |
| Ventnor Avenue | 260 |
| Marvin Gardens | 280 |
| Pacific Avenue | 300 |
| North Carolina Avenue | 300 |
| Pennsylvania Avenue | 320 |
| Park Place | 350 |
| Boardwalk | 400 |

**Railroads**

The railroad spaces on the board are Reading Railroad, Pennsylvania Railroad, B&O Railroad and Short Line Railroad. When the player lands on a railroad space the first time, the player is paid 200 credits. However, the second time the player lands on a railroad space, the player is paid 400 credits. The third time, the player is paid 600 credits for landing on the space. The fourth time, the very lucky player is paid 800 credits for landing on the last railroad space which is Short Line.

**Income Tax, Luxury Tax**

Landing on either the Income Tax or Luxury Tax spaces pays the player the following credits, based on the space landed:

| Name | Credits |
|---|---|
| Income Tax | 200 |
| Luxury Tax | 100 |

**Electric Company, Water Works**

When the player lands on these utility spaces, the player is awarded the number specified in the table below for the space times the total of the roll of two six sided dice. For example, if the player lands on Electric Company and rolls 2 and 4 the player is awarded 40 credits.

| Name | Multiplier |
|---|---|
| Electric Company | 5x |
| Water Works | 10x |

**Go To Jail**

When the player lands on this space, the player's token is moved back to the Just Visiting/In Jail space. The player does not pass the Go space when this happens so the player does not win 200 credits. Play proceeds as normal after the player token is moved.

**Just Visiting/In Jail, Free Parking**

These spaces do not pay the player any credits nor do they move the player token. Play proceeds as normal after landing on these spaces.

**Chance**

When the player lands on Chance, the player draws a random Chance card from the available Chance cards below. The player could win credits or move to another spot on the board. Here are the Chance cards available:

- "Go directly to Jail" – The player token moves directly to Just Visiting/In Jail. The player does not pass Go nor does the player collect 200 credits. Play proceeds normally.
- "Bank pays you dividend of 50 credits" – The player is awarded 50 credits. Play proceeds normally.
- "Go back 3 spaces" – The player is moved back three spaces. Play proceeds normally.
- "You have won a crossword competition. Collect 10 credits." – The player is awarded 10 credits. Play proceeds normally.
- "You have been elected Chairman of the Board. Collect 100 credits." – The player is awarded 100 credits. Play proceeds normally.

**Community Chest**

When the player lands on Community Chest, the player draws a random Community Chest card from the available Community Chest cards below. The player could win credits or move to another spot on the board. Here are the Community Chest cards available:

- "Bank error in your favor. Collect 200 credits" – The player is awarded 200 credits. Play proceeds normally.
- "From sale of stock, you get 50 credits." – The player is awarded 50 credits. Play proceeds normally.
- "Go directly to Jail" – The player token moves directly to Just Visiting/In Jail. The player does not pass Go nor does the player collect 200 credits. Play proceeds normally.
- "Holiday Fund matures. Receive 100 credits." – The player is awarded 100 credits. Play proceeds normally.
- "Income tax refund. Collect 20 credits" – The player is awarded 20 credits. Play proceeds normally.

**Go**

Landing on Go pays the player 200 credits and the bonus round is complete.

# 4 Requirements

## 4.1 Functional Requirements

You will implement a Java program that will play the Monopoly Once Around Bonus. The program is started by running the Java interpreter and providing the classpath and main class name:

```
java -classpath src Monopoly -dice
```

`-dice` is an optional command line argument that may or may not be on the command line. If `-dice` is on the command line, the player should be prompted for the values of two six sided dice when it is time for the player to move.

The program plays the bonus game as follows:

1. If `-dice` is on the command line, the player is prompted to enter the value of the two six sided dice, separated by a space:

   ```
   Please enter your dice roll: 5 6
   ```

2. If `-dice` is not on the command line, the program randomly generates the values of two six sided dice and prints out the result of the roll:

   ```
   Player rolled 5 and 6 = 11
   ```

3. The program then moves the player token the number of spaces determined by the total of the dice. The picture of the board above should be used for the layout of the board and the position of the spaces.

4. If the player passed Go, the program prints out:

```
Player passes Go and is awarded 200 credits.
```

The program then prints out the total number of credits won:

```
Total Credits Won: 1234 credits
```

and the program exits.

5. If the player did not pass Go, the program prints out the name of the space the player landed on:

```
Player landed on St. Charles Place
```

6. The program then takes the action based on the type of space the player lands on.

   (a) For property spaces, Income Tax, Luxury Tax and Go print out the name of the space and how many credits are awarded:

   ```
   St. Charles Place awards 140 credits.
   ```

   (b) For railroads, print out the name of the space, how many railroads have been landed on and the total number of credits awarded:

   ```
   Pennsylvania Railroad awards 2 x 200 = 400 credits.
   ```

   (c) For Electric Company and Water Works, print out the name of the space, the values of the two randomly rolled dice, their total and then the total number of credits won:

   ```
   Electric Company awards (1 + 6) x 5 = 35 credits.
   ```

   (d) For Go to Jail, print out the following message and move the player token to the Just Visiting/In Jail space:

   ```
   Go directly to Jail. Do not pass Go, do not collect 200 credits.
   ```

   (e) For Just Visiting/In Jail and Free Parking, do nothing and print nothing extra.

   (f) For Chance and Community Chest, print the text of the randomly selected card and perform the action described in the rules above. For example:

   ```
   Bank error in your favor. Collect 200 credits.
   ```

7. The program then prints out the total number of credits awarded so far:

```
Total Credits Won: 1234
```

8. If the player landed on Go, the program then exits. Otherwise, repeat steps 1-7.

## 4.2 Non-Functional Requirements

**Implementation Requirements**

1. Implementation should consist of a good object-oriented design where:

   (a) The program is decomposed into multiple classes.

   (b) Those classes use encapsulation, inheritance and polymorphism to implement the game play.

2. The class should not depend on any third-party libraries or classes other than what comes in the Java SE API.

3. The source code should follow standard Java conventions and style:

   (a) Proper indentation and brace usage.

   (b) Proper use of vertical spacing. Each method definition should be separated by a space.

   (c) Variable names should be descriptive and relevant. One letter variable names are only appropriate in loops.

   (d) Methods should be:

      i. Not overly long. More than a half a screenful is probably too long. Break it up into smaller methods.

      ii. Named properly. Method names should be verbs or predicates.

   (e) Comments should be used liberally to explain your code or what the program is doing.

**Unit Testing Requirements**

1. You will create a set of unit tests using the JUnit framework that will be executed to test that you have implemented the rules for Monopoly properly.

2. You should have a unit test for the following types of spaces to test the outcome of landing on those spaces. Note: there is no need to create an individual unit test for each space in that type. In other words, you don't need a unit test each for Baltic Avenue, Connecticut Avenue, etc.

   (a) Property spaces

   (b) Railroad spaces

   (c) Luxury Tax/Income Tax spaces

   (d) Go Directly To Jail space

   (e) Go space

3. The unit tests should be implemented in a class called `MonopolyTest.`

4. Unit tests must be annotated with the `@Test` annotation so they can be run via the JUnit test runner.

5. Unit tests show follow the same Java coding convention and style as the implementation. Additionally,

   (a) Unit tests should be named with what test is being performed (i.e. `landingOnStCharlesPlacePays140Credits`, `landingOnGoToJailMovesPlayerToJustVisiting`, etc.)

   (b) The body of the unit test should follow the **given-when-then** format discussed in class with comments showing which part is the "given", which part is the "when" and which part is the "then".

   (c) The "then" section should use JUnit's `Assert` class to assert the outcome of the code that was executed in the "when" section of the code based on the test.

**Documentation Requirements**

1. In addition to the source code, you will also create a `README.txt` file with the following contents:
   (a) Your name and student ID number
   (b) Answers to the following questions:
      i. How do you run your program (i.e. what is the command line)?
      ii. Describe your object-oriented design for the program:
         A. How are you implemeting the game board?
         B. How are you implementing the spaces and the types of spaces?
         C. How are you implementing the Chance and Community Chest cards?
      iii. What specific problems or challenges did you have implementing your solution? For example, was there a particular requirement that you had difficulty implementing? Or perhaps there was a particularly nasty bug in your implementation you had problems tracking down.
      iv. Were there any requirements that were not implemented or not implemented properly in your solution? If so, please elaborate.
      v. Were there any requirements that were vague or open to interpretation that you had to make a decision on how to implement? How did you elect to interpret them?
      vi. How would you rate the complexity of this MP on a scale of 1 to 10 where 1 is very easy and 10 is very difficult. Why did you give this rating?

**Submission Requirements**

To submit your solution to this MP, please place the following files into a file in the ZIP file format (i.e. I should be able to unzip the file using Windows' extract tool, or the `unzip` command in Linux):

- `README.txt`
- A `src` directory containing the directory structure that has all of your `.java` source code files. If you use `package` declarations, make sure your `.java` files are in a directory structure that reflects the `package` declarations (i.e. a class called `Monopoly` that is in the `edu.iit.itmd510` package should have a path of `src/edu/iit/itmd/Monopoly.java` in you zip file).
- A `bin`, `target` or `classes` directory (depending on your IDE) containing all of the compiled `.class` files for your program. If you use `package` declarations, make sure your `.class` files are in a directory structure that reflects the `package` declarations (i.e. a class called `Monopoly` that is in the `edu.iit.itmd510` package should have a path of `bin/edu/iit/itmd/Monopoly.class` in your zip file).

Please name this zip file in the following format `Name-StudentID.zip`.

# 5 Grading

Your implementation will be graded on a scale of 100 points. The breakdown of the points is as follows:

- **35 points for correctness**. Correctness is whether or not your program compiles, executes with no errors and implements the requirements correctly.

- **35 points for completeness**. Completeness is whether or not your program implements all of the rules and functional requirements above.
- **15 points for object-oriented design**. Program should use classes, objects, encapsulation, inheritance and polymorphism where appropriate.
- **10 points for unit testing**. Unit tests are implemented per the requirements above, compile, can be executed and pass. Unit test must assert something using JUnit Assert to be a valid unit test.
- **5 points for coding conventions, style and documentation**. Submission follows coding standards per requirements above, follows the documentation requirements and the submission requirements.

# 6 Plagarism and Copying

- **All solutions will be checked for plagiarism and copying via automated tools and manual inspection.**
- **Students found to have copied code from another student or the Internet will receive a zero for this assignment and will be reported for an Academic Code Violation.**
- **This is not a group assignment, this is an individual assignment.** You may discuss the assignment with other students but you are not allowed to give code to, share code with or show your code to your fellow ITMD 510 students. When code is shared, there is no way for me to determine which student copied from whom so all parties involved might receive a zero.

# 7 Suggestions

- Start working on the MP early so that you have enough time to ask questions during class, during office hours, on Blackboard, etc.
- Come up with an OO design before you start coding. Identify possible classes and method by reading the description of the rules and the functional requirements.
- Don't create a different class for each space on the board (i.e. `StCharlesPlaceSpace, ElectricCompanySpace, etc.`). Think about what common data and functionality the different spaces have and see if you can encapsulate them in a reusable class.
- Something that might help you implement the outcome of landing on a space is to encapsulate the game or player state in an object which is given to the code that implements the outcome. The code that implements the outcome calls methods on the player state to change the player state based on the desired outcome.