CPE 102 Project 4
Maze Game, Part 3

Submission Instructions

The following files must be submitted on PolyLearn.

Files: DelimitedTextIO.java, Drawable.java, DrawableExplorer.java, DrawableMonster.java, DrawableSquare.java, DrawableTreasure.java, DrawableMaze.java, Explorer.java, Maze.java, MazeReadException.java, Monster.java, Occupant.java, RandomOccupant.java, Square.java, Treasure.java

Submission Dates:
Full Credit: Due on Friday (11/18)
90%: Due on Saturday (11/19), and
80%: Due on Sunday (11/20).

Testing With the Provided Test Driver

1. The test driver will be published ***at 6am on the day of the 100% due date for the assignment***.
2. You should develop and use your own tests prior to using the provided test driver. Do not use the provided test driver until your solution is complete and you believe it is correct or you are likely to be overwhelmed with error messages and will spend unnecessary time just trying to understand the test driver - a frustrating and inefficient way to approach problem solving with computers!
3. Download P4TestDriver.java (to be published on the first due date) in the same directory as all of the other source files (.java files)
4. Compile the P4TestDriver.java, all of your source files (.java files) and run P4TestDriver. Remember that your code will be graded on unix1 (2, 3, or 4) so, to avoid unpleasant grading surprises be sure to test on one of those machines just before handing it in.

Mini Test Driver

Testing a project with exceptions and file I/O can be tricky. P4TinyDriver.java is a "starter" test driver to help you begin testing. This driver is available now and you can use it to build your own tests.

Learning Objectives

- More practice writing your own exceptions.
- More practice handling exceptions in code you write.

- Practice working with file-based streams.
- To solve the issues that come up while reading and writing comma delimited data.
- More practice with inheritance and interfaces.
- More practice using Java Standard Library classes.

Resources

1. Java Standard API

2. P4TestDriver.java - to be released 11/18 (at 6am)

3. Necessary input files for the test driver. The first three files are a well-formed comma delimited Maze files. The others contain errors of some sort:

    - mazeGood.txt
    - mazeGood2.txt
    - mazeGoodOutOfOrder.txt
    - mazeGoodDiffRowsCols.txt
    - mazeBadRowCol.txt
    - mazeUnknownType.txt
    - mazeDupSquare.txt
    - mazeBadLineFormat.txt
    - mazeBadLineFormat2.txt
    - mazeGood3.txt
    - mazeGood3a.txt

Problem Description

You will be modifying and enhancing your Project 3 source code in several ways. You will also be working with character streams to read and write comma-delimited text files that contain complete maze specifications. Additionally, you will create an exception class and throw exceptions when various errors occur.

IMPORTANT: When running the provided test driver you will need all the text files (see Resources section above for links) to be in the directory where the .class files of your solution are found.

IMPORTANT #2: You will need to use the .java versions of **all** your Project3-Part 1 files, including the Drawable*.java files. Your draw() methods will need to be empty to compile with a regular java compiler.

Suggestions:

1. Make sure your Project 3 - Part 1 code successfully passes all of the Project 3 tests before moving on to the Program 4 modifications.

2. Then, and only then, **COPY** all Program 3 source files (.java files) to a new project and begin making the Program 4 modifications.

3. Test as you go. Compile after *every* change. At some point you will be updating your game so that you can store it out to a file and read it back in from a file. I recommend writing all the code to write your maze to a file first (the "toText" method and "writeMazeToFile" method). Test that and make sure it works to your liking. Then work on the code to read in from the file (the "toObject" method and "readMazeFromFile" method).

Specification

1. You must implement the classes, interfaces, and methods exactly as described or the provided test driver will not work!

2. Do not reinvent the wheel. Demonstrate knowledge of the Java Standard Library classes you are working with. When and wherever appropriate use code that has already be written and tested for you! Your program will be graded with this in mind!

3. Add a default constructor the **Maze** class. Create the **ArrayList** of **RandomOccupants** inside the constructor.

4. Add a constructor to **Explorer** that only takes a **Maze** as a parameter. Do not do anything in the constructor except set the **Maze** instance variable to the parameter passed in.

5. Add a constructor to **Square** that only takes a row and a column.

6. In the **Treasure** class, override the `moveTo` method of **Occupant**. Check to see if this Treasure's *current* location is null, if not, set the **Treasure** at the current location to null. (Remove treasure from that location because we are moving.) Call the parent class `moveTo` method to actually do the moving to the new location. Don't forget to set the **Treasure** in your new location to 'this'.

7. Create the following ***checked*** exception:

MazeReadException - This exception should take a **String** message, **String** line, and int lineNumber in its constructor. Pass the message on to the parent class. Store the line and lineNumber. Create query methods called `getLine()` and `getLineNum()`.

8. Write a new Java interface called ***DelimitedTextIO*** with the following methods:

- String toText(char delimiter) - This method returns a String containing all the data of the implementing class as text and with each element separated by the provided delimiter.

- void toObject(Scanner input) - This method uses the provided Scanner input to parse delimited text representing the data for the implementing class and initializes the objects instance variables with the parsed values. The delimiter to use must be specified for the Scanner input before calling this method.

9. Implement the ***DelimitedTextIO*** interface in the **Occupant** class and the **Square** class. Implement its methods in **Occupant** and **Square** and override them in an necessary subclass as follows:
   - The toText method should return a delimited **String** (a comma in this example) in the order specified below for each specific occupant. Replace the names of the variables with the object's actual values. You must demonstrate that you understand inheritance and overriding methods in your implementation - this means you must not use any accessor methods (other than toText) of **Occupant** or any of its subclasses. You may use the row() and col() methods of Square from within Occupant. Recall that you can use the getClass().getName() call to obtain the name of any class at runtime.

```
"Square,row,col,wall[UP],wall[RIGHT],wall[DOWN],wall[LEFT],seen,inView"
"Explorer,row,col,name"
"Treasure,row,col,found"
"Monster,row,col"
```

Notes:

- Not every subclass of **Occupant** will need a toText() method. For example, **RandomOccupant** does not override **Occupant's** toText() because it has nothing to add.
- Think carefully about which classes have the instance variables that need to be added to the text String.
- The toObject method should use the provided **Scanner** to reverse the process of toText above. For example during Maze's readMazeFromFile, if it read a line that contained the following:

  ```
  Treasure,2,3,false
  ```

  the Maze would read the class name from the line and construct a Treasure object and call Treasure's toObject with the Scanner still containing the rest of the line, "2,3,false". The Treasure would pass that call to its parent (that contains a Maze reference so it could get the proper Square and move there). Then the Treasure would finish parsing the line so that it could set its "found" variable to false. You must demonstrate that you understand inheritance and overriding methods in your implementation - this means you must not use any methods (other than toObject) of **Occupant**, any of its subclasses, or **Square**.

Notes:

- The object class name will already have been removed from the input **Scanner** when it gets passed to the toObject method of each class. Additionally, the row and col will have already been removed for **Square**.
- The **Scanner** will already have its delimiter set appropriately (to parse lines with commas in them) when it is passed to this method.
- Think carefully about how you will read the row and column and use that information to set the location of the particular occupant that you are "building". You can use the getSquare() method of the **Maze** class to get the correct **Square** to moveTo. Note that Occupant doesn't have access to the **Maze** so you can't do it there.

10. Add the following new methods to the **Maze** class:

public void writeMazeToFile(String fileName) throws IOException

This method must write every **Square** and **Occupant** in the **Maze** to a text file as comma-delimited text with one **Occupant/Square** per line. The first line of the file will be the "rows,cols" of the **Maze**, then all the **Squares** should be written to the file, and finally it will write all the **Occupants**. It ***must*** make use of the toText method written in **Occupant**, its subclasses, and **Square**. If, after thinking about it sufficiently, you do not know how to do this you may ask you instructor for clarification. You can examine the provided comma-delimited input files for examples of what your output should look like (see Resources section above).

Notes:

- The data must be written in ***exactly the order specified*** depending on **Occupant** or **Square** class type (i.e. **Treasures** must first write "Treasure", then row, then col, then whether it's been found or not). The only **String** formed in **Maze** is the first line containing the row and cols of the **Maze**. All other **Strings** written to the file are formed completely by the toText() method of each **Object**.

public void readMazeFromFile(String fileName) throws IOException, FileNotFoundException, MazeReadException

This method must read comma-delimited text files in the format specified for the toText method of **Occupant** class, its subclasses, and **Square** class. It must make use of the toObject method written in **Occupant**, its subclasses, and **Square**. If, after thinking about it sufficiently, you do not know how to do this you may ask you instructor for clarification.

1. The **Maze** will be constructed using the new default constructor. It's **Square[][]**, **Explorer**, and **ArrayList<RandomOccupant>** will all be constructed from the file.

2. Read in the rows and cols of the **Maze** to create a **Square[][]** of the appropriate size. Then construct and read in all the **Squares/Occupants**.

For **Squares**, the **Maze** will first determine that the line starts with "Square". It will then read in the row and col of the **Square** and use that information to construct a **Square** object using the constructor that takes a row and col. Finally it will pass the rest of the **Scanner** to the **Square's** toObject method so it can initialize itself.

For all other **Occupants**, the **Maze** will determine what kind of **Occupant** it is and construct the appropriate object using the constructor that only takes a **Maze**. It will *not* read the row or the col from the **Scanner**, but simply pass the **Scanner** on to the toObject method of the newly created object. Don't forget to add RandomOccupants to the ArrayList once they are fully initialized.

3. When reading in data and initializing itself from the file, the readMazeFromFile method will throw several exceptions.

    - Should a **FileNotFoundException** or **IOException** occur, it will do nothing and simply let the exception propagate to the method caller.
    - Should the first line of the file not contain the row and col, it will throw a **MazeReadException** with the message "Rows and columns not specified." It will additionally store the contents of the line the error occurred on and the line number.
    - Should it read in a Square with a row and col that has already been initialized, it will throw a **MazeReadException** with the message "Duplicate square." It will additionally store the contents of the line the error occurred on and the line number.
    - Should the first it read a line where the first word is not a valid class, it will throw a **MazeReadException** with the message "Unknown type." It will additionally store the contents of the line the error occurred on and the line number.
    - Should any exceptions get thrown from a toObject() method call, it will catch the exception and then it will throw a **MazeReadException** with the message "Line format or other error." It will additionally store the contents of the line the error occurred on and the line number.

Hint: The instructor solution uses two **Scanners**, one to read lines from the file, the second to parse each line. Don't forget to set the delimiter of the second **Scanner**!

Last updated 5/13/16 by Julie Workman