

## CPE 102 Program 3 – Maze Game

### Learning Objectives:

- To be able to **design, code, and test** a Java **abstract class**.
- To be able to **design, code, and test** a collection of Java **classes** that constitute a **class hierarchy**.
- To be able to **design, code, and test** Java **classes** that **implement an interface**.
- To be able to **use objects and methods** of the Processing graphics library to draw graphic shapes.
- To be able to **design, code, and test** a Java **class** that uses an array of **polymorphic object references**.
- To gain further practice using java arrays and ArrayLists.

### Due Dates and Submission Instructions

**Important:** This Project will have a test driver that you must pass 100% to get any credit. This driver will become available to you at 6am the day the project is due. Do your own testing! Project

**Submission:** 100 Points. Due 100% on Friday (11/4), 90% on Saturday (11/5), and 80% on Sunday (11/6).

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

Submit all java files to the PolyLearn.

### Testing With the Provided Test Driver

1. The test driver will be published at 6am on the due date.
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. Save [P3Tests.java](#) in the same directory as all of the source files (.java files).
4. 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.

## Problem Description

In this project you will be implementing a maze game. Your game will consist of a maze filled with treasures and monsters. An explorer must navigate the maze and find all the treasures before it runs into a monster.

To implement the game you must do the following:

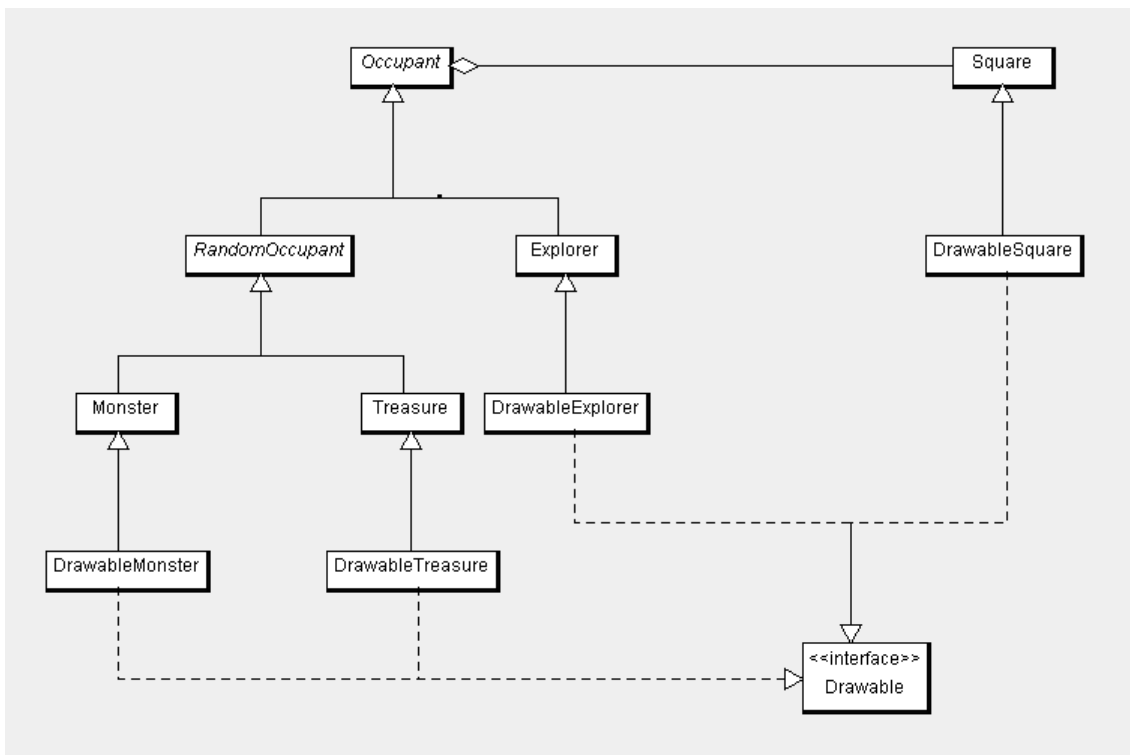
- You will implement a *hierarchy of classes* that represent **Occupants** of the maze game including **Explorer**, **Monster**, and **Treasure**. Objects constructed from these classes have some common properties inherited from the **Occupant** class such as a location in the maze (**Square**). **Explorer** objects also have a name property and the ability to move at the direction of the game player. **Treasures** and **Monsters** can have randomly generated starting locations in the maze.
- Additionally, you will implement the following classes: **DrawableSquare**, **DrawableExplorer**, **DrawableMonster**, and **DrawableTreasure**. Objects constructed from these classes "know how to draw themselves", that is they have a method named `draw` that can be called to draw their image in the **Maze**. Although there are three different concrete classes descendant from the **Occupant**, we wish them all to have the same method specification for drawing themselves. This is necessary so that a collection (array) of different classes of **Occupant** objects may all be drawn from the **Maze** class with the same method call within a loop. To do this you will need to *implement an interface* for that drawing method.
- You will additionally design a **Maze** class that consists of a 2D array of maze **Squares**. The **Maze** class contains the basic game logic and knows how to move its **Occupants** around the **Maze**. A **DrawableMaze** has a method to command all of its **Squares** and **Occupants** to draw themselves.
- You will first implement all the non-drawable versions of each class. All of the game logic is contained in these classes. The only difference is that they do not know how to draw themselves in the Processing environment. However, the `P3Tests.java` expects all of your classes to be present, but does nothing to test the `draw` method of the **Drawable** classes. We are not testing the draw part.
- A goal of your learning for this assignment will be how to *reuse class specifications through inheritance* and how to use a standardized *interface* for the drawing operation. To complete the program and its classes you will be required to use abstract classes, as well.

# Specification

These are the requirements for the classes that you must implement. I **highly** recommend creating skeletons of all the non-drawable classes, with stubs for the required methods, before you do any implementation. **Get your entire system compiling before you fill in any one class.** Then compile and test your classes as you go.

Get your non-drawable classes implemented first! Then create the drawable versions with empty draw methods.

To give you an overview of some of the classes you will be designing, here is a UML diagram of the system. Note that the Maze and DrawableMaze classes are not pictured. Also, the abstract classes are shown in italics.



## Square

Develop a class to represent a **Square** in the **Maze**. To be compatible with the test drivers that will be used to grade your program you must name this class **Square**.

The specifications for this class are:

### Named Constants:

- A public int SQUARE\_SIZE set to 50 (number of pixels per **Square**)

- A public int UP set to 0
- A public int RIGHT set to 1
- A public int DOWN set to 2
- A public int LEFT set to 3

#### Fields (instance variables):

- A java array of four boolean values to represent the walls of the **Square**. (i.e. Does the **Square** have a wall in that direction?)
- A boolean to keep track of whether the **Square** has been seen or not
- A boolean to keep track of whether the **Square** is currently in view or not
- Two integers to keep track of the row and column of the **Square**
- A **Treasure** reference to keep track of the treasure (if any) in that **Square**.

#### Constructors:

- `public Square(boolean up, boolean right, boolean down, boolean left, int row, int col)`

A constructor to initialize the walls array and the row and column. The booleans to keep track of "seen" and "inView" should default to false. The Treasure reference starts out null.

#### Methods:

- `public boolean wall(int direction)` - a query for a wall in the given direction.
- Create query methods for seen, inView, row, col, and treasure. Name them `seen()`, `inView()`, `row()`, `col()`, and `treasure()`.
- `public int x()` - a query for the x-value (in pixels) of the upper left corner of the **Square**. Use row and/or column, and `SQUARE_SIZE` to determine this.
- `public int y()` - a query for the y-value (in pixels) of the upper left corner of the **Square**. Use row and/or column, and `SQUARE_SIZE` to determine this.
- `public void setInView(boolean inView)` - a command to tell the **Square** if it is currently in view of the **Explorer**. This command should also set the `seen` variable to true if the value of `inView` is true. (If it has been in view once then it logically has been seen.)
- `public void setTreasure(Treasure t)` - sets the treasure reference to the parameter `t`.
- `public void enter()` - a command to tell the **Square** that the **Explorer** has entered it. Simply check to see if **Square** contains a **Treasure**, and if it does, invoke the treasure's `setFound()` method.

## <<abstract>> Occupant

Develop an abstract class that includes properties common to all maze occupants. To be compatible with the test drivers that will be used to grade your program you must name this class **Occupant**. This class is designed to be extended, and as such, is abstract even though it contains no abstract methods.

The specifications for this class are:

### **Fields (instance variables):**

- a **Square** (see Square class above) representing the location of the occupant in the maze

### **Constructors:**

- **public** `Occupant ()` - a default constructor that does not initialize anything
- **public** `Occupant (Square start)` - a constructor that accepts a **Square** argument to initialize the **Occupant's** location

### **Methods:**

- **public** `Square location ()` - a query method for the location
- **public void** `moveTo (Square newLoc)` - a command to change the Occupants location

## Explorer

Develop a *class* that *extends* **Occupant** to model an **Explorer**. To be compatible with the test drivers that will be used to grade your program you must name this class **Explorer**. Because an **Explorer is-an Occupant**, you can query it for `location ()` and move it to a new location using `moveTo ()`.

The specifications for this class are:

### **Fields (instance variables):**

- a **String** name
- a reference to the **Maze** that it inhabits so that it may locate itself and move about the **Maze**.

### **Constructors:**

- **public** Explorer(Square location, Maze maze, String name) - a constructor to initialize all the instance variables. Lastly, be sure to call the lookAround method of the **Maze** to reveal the squares that the **Explorer** can initially see.

#### Methods:

- **public** String name() - query for the name.
- **public void** move(int key) - command the **Explorer** to move a direction in the **Maze** based on a key press from the user. The **Explorer** does not move itself randomly around the **Maze** like a **RandomOccupant**, but instead is built to respond to key presses from the user. This key press information is passed to the move method of the **Explorer**. Key presses in this game will be arrow keys, which have pre-defined values in the **KeyEvent** class. Look up the values of the *named constants* for the non-numpad and numpad arrow keys ([in the java class libraries](#)). Steps to accomplish the move method:
  1. Get the row and column of the current location.
  2. Determine the key pressed, if not an arrow key, do nothing. If it was an arrow key, figure out which direction and adjust the row and column numbers accordingly.
  3. Check the **Explorer**'s current location to see if there is a wall in the chosen direction. If not, get the desired **Square** from the **Maze** and call the **Explorer**'s moveTo method. If there is a wall in the chosen direction, do nothing.

Hints (*some* of the methods you will need):

**Maze**'s getSquare method.  
**Square**'s wall method  
**Explorer**'s moveTo method.

- **void** moveTo(**Square** s) - command the **Explorer** to move to another **Square** in the **Maze**. The actual moving of the **Explorer** is done via its parent's moveTo method. Be sure to call that method within this one. In addition to actually changing locations, the **Explorer** needs to notify the given **Square** that it has been entered by the **Explorer**. Call the appropriate method of the **Square** class to do this. Finally, inform the **Maze** that the **Explorer** would like to lookAround from the newly entered **Square**.

## <<abstract>> RandomOccupant

Develop an abstract class that *extends* **Occupant** to include properties common to all not human-controlled maze occupants, namely: know how to generate a random **Square** as its initial location and know how to move randomly around the **Maze**. To be compatible with the test drivers that will be used to grade your program you must name this class **RandomOccupant**.

The specifications for this class are:

### **Fields (instance variables):**

- a reference to a **Random** object used to generate random numbers for movement and initial location
- a reference to the **Maze** that it inhabits so that it may locate itself and move about the **Maze**.

### **Constructors**

- **public** RandomOccupant (Maze maze) - a constructor to initialize the maze variable and randomly set its location in the maze. To do so, it will need the following methods:

From **Maze**: rows, cols, getSquare

From **Occupant** (its parent): default constructor, moveTo

- **public** RandomOccupant (Maze maze, long seed) - a constructor to initialize the maze variable and randomly set its location in the maze, just like the above constructor. Additionally, this constructor has a seed to use when constructing the Random number generator. The test driver relies on this constructor so be sure to call the constructor for the Random Number generator that takes a seed. Additionally, to pass the test driver, you must select a random row first, then a random column.
- **public** RandomOccupant (Maze maze, Square location) - a constructor to initialize the maze variable and the location of its parent. (This constructor is used if the location of the RandomOccupant needs to not be random.)

### **Methods:**

- **public void** move () - this method is responsible for moving the **RandomOccupant** in a random fashion. The move must be legal, however, so that it does not move through any walls in its current location. Steps to accomplish this task:
  1. Get the row and column of the current location.
  2. Generate a random integer between 0-3 (inclusive) to represent a direction to move. (Note that these directions are named constants in the **Square** class). Repeat until a valid direction is generated. A valid direction is one in which no wall is in the way.
  3. Adjust the row and column numbers accordingly depending on the direction moved. You must use the named constants in the **Square** class instead of the literals 0, 1, 2, and 3.
  4. Get the **Square** from the **Maze** that is in the row and column the **RandomOccupant** is supposed to move to and then move the **RandomOccupant** to that location. You will need **Maze**'s `getSquare` method.

## Monster

Develop a *class* that *extends* **RandomOccupant** to model a **Monster**. To be compatible with the test drivers that will be used to grade your program you must name this class **Monster**. Usually **Monsters** begin the game in a random location, although you will also provide a constructor that can place the **Monster** in a specific location in the **Maze**. **Monsters** then move randomly about the **Maze**. If they run into the **Explorer**, the **Explorer** dies and the game is over. The random initial placement and the random movement is taken care of by **Monster**'s parent class, **RandomOccupant**. The **Monster** class exists for two reasons:

- To be able to differentiate between other **RandomOccupants** that do not kill **Explorers**.
- To extend to the **DrawableMonster** class.

The specifications for this class are:

### **Fields:**

- None

### **Constructors:**

- **public** Monster(Maze maze) - constructor to randomly locate itself in the **Maze**. (Remember, the parent of this class is **RandomOccupant**.)
- **public** Monster(Maze maze, long seed) - constructor to randomly locate itself in the **Maze** with a seed for the Random class.



- **public** Monster(Maze maze, Square location) - constructor to generate a **Monster** in a specified location in the **Maze**.

## Treasure

Develop a *class* that *extends* **RandomOccupant** to model a **Treasure**. To be compatible with the test drivers that will be used to grade your program you must name this class **Treasure**. Usually **Treasures** begin the game in a random location, although you will also provide a constructor that can place the **Treasure** in a specific location in the **Maze**. Even though **Treasures** extend **RandomOccupant**, they may not move about the Maze, so you will override the Treasure's inherited move method. The **Explorer** must collect all the **Treasures** in the Maze before encountering a **Monster** to win the game. The random initial placement is taken care of by **Treasure**'s parent class, **RandomOccupant**. **Treasure** must simply add code to override the move method..

The specifications for this class are:

### Fields:

- a boolean representing whether or not the treasure has been found by the **Explorer**

### Constructors:

- **public** Treasure(Maze maze) - constructor to randomly locate itself in the **Maze**. (Remember, the parent of this class is **RandomOccupant**.) The found variable should be initialized to false. In the constructor, the object must also send itself (using `this`) to the `setTreasure` method of the **Square** it resides in.
- **public** Treasure(Maze maze, long seed) - constructor to randomly locate itself in the **Maze** using a seed. (Remember, the parent of this class is **RandomOccupant**.) The found variable should be initialized to false. In the constructor, the object must also send itself (using `this`) to the `setTreasure` method of the **Square** it resides in.
- **public** Treasure(Maze maze, Square location) - constructor to generate a **Treasure** in a specified location in the **Maze**. The found variable should be initialized to false. In the constructor, the object must also send itself (using `this`) to the `setTreasure` method of the **Square** it resides in.

### Methods:

- `public boolean found()` - query method for the found value
- `public void setFound()` - sets the found value to `true`
- `public void move()` - override the move method of **RandomOccupant** so that it does nothing. Note that a **Treasure** may still be moved via the `moveTo()` method, but cannot move itself randomly about the **Maze**.

## Maze

Complete the [implementation](#) of the **Maze** class. This class contains most of the game logic. Complete all the code where I say "CHANGE" in a comment. I recommend you remove the word "CHANGE" whenever you implement a part of it, so you know you are done when all the CHANGES are gone. The instructions for this class are in the code.

## Drawable

Define the *Drawable* interface. This must be placed in a file named **Drawable.java**.

### Methods:

- `public void draw();`

## STOP and READ before you move on:

The following specifications are for the *Drawable* versions of your classes. When you first create these classes, leave the `draw()` method empty. The specifications below include how you should draw each item *but we are not testing the actual draw*.

## DrawableSquare

Extend the **Square** class and implement the *Drawable* interface so that the **DrawableSquare** knows how to draw itself in the Processing window.

The specifications for this class are:

### Constructors:

- `public DrawableSquare(boolean up, boolean right, boolean down, boolean left, int row, int col)` - a constructor to initialize all the data needed in the parent class.

### Methods:

- **public void** draw() - Instruct the **DrawableSquare** to draw itself *in the appropriate spot on the screen.*

If the **DrawableSquare** *has not* been seen: Don't draw anything.

If the **DrawableSquare** *is currently in view*: Draw the entire square in the color scheme of your choice. Be sure to draw walls in the proper places on your square in the theme of your choice.

If the **DrawableSquare** *has been seen, but is not currently in view*: Draw the entire square in the color scheme of your choice, but make sure the color scheme is darker than you would draw for a square that is currently in view. Be sure to draw the walls here, as well. Your explorer has a good memory!

### Hints:

Use the `x()` and `y()` query methods to get the x and y coordinate of the **DrawableSquare's** upper left pixel. Derive the rest of your pixel calculations from there.

## DrawableExplorer

Extend the **Explorer** class and implement the **Drawable** interface so that the **DrawableExplorer** knows how to draw itself in the Processing window.

The specifications for this class are:

### Constructors:

- **public** DrawableExplorer(Square location, Maze maze, String name) - a constructor to initialize all the data needed in the parent class.

### Methods:

- **public void** draw() - Instruct the **DrawableExplorer** to draw itself *in the appropriate spot on the screen.*

Draw your explorer in the appropriate place on the screen using methods of the Processing Development Environment. Be sure your explorer matches your chosen theme.

## DrawableMonster

Extend the **Monster** class and implement the *Drawable* interface so that the DrawableMonster knows how to draw itself in the Processing window.

The specifications for this class are:

### Constructors:

- **public** DrawableMonster(Maze maze) - a constructor to initialize all the data needed in the parent class for a Monster with a random location.
- **public** DrawableMonster(Maze maze, long seed) - a constructor to initialize all the data needed in the parent class for a Monster with a random location and a seeded Random number generator.
- **public** DrawableMonster(Maze maze, Square location) - a constructor to initialize all the data needed in the parent class for a Monster with a given location.

### Methods:

- **public void** draw() - Instructs the **DrawableMonster** to draw itself in the appropriate spot on the screen -- if the **Square** it currently occupies is inView of the explorer.

## DrawableTreasure

Extend the **Treasure** class and implement the *Drawable* interface so that the DrawableMonster knows how to draw itself in the Processing window.

The specifications for this class are:

### Constructors:

- **public** public DrawableTreasure(Maze maze) - a constructor to initialize all the data needed in the parent class for a Treasure with a random location.
- **public** public DrawableTreasure(Maze maze, long seed) - a constructor to initialize all the data needed in the parent class for a Treasure with a random location and a seeded Random number generator.
- **public** public DrawableTreasure(Maze maze, Square location) - a constructor to initialize all the data needed in the parent class for a Treasure with a given location.

### Methods:

- **public void** draw() - Instructs the **DrawableTreasure** to draw itself if the **Square** it currently occupies *has been seen* by the **Explorer**.

### DrawableMaze

Extend the **Maze** class and implement the **Drawable** interface so that the **DrawableMaze** knows how to draw itself in the Processing window.

The specifications for this class are:

### Constructors:

- **public** DrawableMaze (DrawableSquare[][] maze, int rows, int cols)- a constructor to initialize all the data needed in the parent class. Notice that the array of **Squares** required for **Maze** is an array of **DrawableSquares** for **DrawableMaze**. Can we pass the array of **DrawableSquares** to the parent constructor?

### Methods:

- **public void** draw() - Instruct the **DrawableMaze** to draw itself *in the appropriate spot on the screen*. Do this by drawing a rectangle the size of the entire maze in the color of your choice that matches your theme. The color you chose should represent unseen portions of the maze (e.g. dark green in the instructor version).

Next, instruct each **Square** in the **Maze** to draw itself. Note that **Squares** don't have a draw method, and the code won't compile if we try to call the draw method on a **Square** retrieved via the `getSquare` method. But, we know they must really be **DrawableSquares**. What can we do to call the draw method of these **DrawableSquares**?

Then, instruct all the **RandomOccupants** in the **Maze** to draw themselves. We run into the same problem as above. Note that you do NOT have to determine whether each **RandomOccupant** is really a **Treasure** or a **Monster**.

Finally, instruct the **Explorer** to draw itself.

## Documentation Requirements

Your class files **MUST** include the following documentation as Java comments.

==> The grader will deduct points from your grade if any of these are missing.

**File header comments:**

1. **Filename**
2. **Class Title and description:** one sentence description of what the program/class does
3. **Author's name and section** -- use Javadoc tag `@author`
4. **Date** -- use Javadoc tag `@version`

## Running the Test Driver

Implement all the non-drawable classes first (and the Drawable classes with the draw() method empty) and test them with the provided P3Tests. This will get you most of the points available for the project.