

## Problem 1: NRooks

NQueens is the problem of how to put N chess queens on an N board such that no queen can see the others according to standard chess rules. However, for simplicity lets assume that the board is filled with Rooks. A rook can see another rook if it shares the same x coordinate or y coordinate In this question we will represent a 2D board that can validate.

```
public static class NRooks {
    public static void main(String args[]) {
        boolean board[][] = createBoard(8);
        set(board, 3,3);

        boolean seesNothing = visibility(board, 3,3);
        System.out.println(isValid(board) && isComplete(board));
        board.set(0,1);
    }
}
```

- (1) Implement a function that creates a square board filled with false to represent there are no rooks in any positions
- (2) Implement a function that sets a single value in the current board to indicate there is a rook there called set.
- (3) Implement a function that checks whether a rook can see another rook called visibility .
- (4) Implement a function that checks whether the board has enough rooks.
- (5) Implement a function that checks whether none of the rooks on the board can see one another

```
}
```

## Problem 2: String to Chars

Implement a function that converts a list of strings to two dimensional array of characters without using String.toCharArray().

```
public static char[][] toCharArrays(List<String> sentence) {
```

- (1) Construct the two-dimensional array that will be returned to an array called returnData.

```
    for(int j = 0; j < sentence.size(); ++j) {
```

- (2) Convert each String in sentence into a single-dimensional array in returnData.

```

    }

    return returnData;
}

```

### Problem 3: Bookstore

Implement a class that represents some basic details about a book in a bookstore: its title, author, price, and number of pages.

```
class Book {
```

- (1) Create members to represent the aforementioned details
- (2) Implement a constructor that lets the user pass in everything
- (3) Implement a constructor that doesn't require passing in the price, but allows for a default price of 9.99.
- (4) Implement getters or setters to indicate which properties can or cannot be changed (only the price can be changed).

}

The bookstore decides it also wants to be a library. Make a new class called `BorrowableBook`. A borrowable book always has a price of 0, and needs to store by some day and who borrowed it. Assume the day is just an integer (say, number of days since January 1 1970). Don't forget that a `BorrowableBook` is a sort of `Book`.

```
public class BorrowableBook
```

- (5) Create members to represent the aforementioned details

(6) Implement a constructor that passes forward all of the data to the Book constructor.

(7) Implement getters and setters for the aforementioned data.

(8) Implement a function that, when passed in the current date, determines if a book is late.

```
// System.out.println(borrowableBook.isLate(currentDate));
```

```
}
```

Now lets represent the entire inventory of books in a store

```
public class Inventory {
```

(9) Create members to represent the books in the store and the money in the cash register.

(10) Implement a method for adding a book to the inventory.

(11) Implement a method for buying a book to the inventory where the cost of the book is its price.

- (12) Implement a function for selling a book from its title. If the book is borrowable then it is merely borrowed, so this function needs to know the name of the person buying the book and the current date, which should both be passed as a parameter.

```
}
```

## Problem 4: Duck Duck Goose

We want to find the set of geese in a collection of fowls using inheritance. In this problem we will not care about the details of a class, just write ‘...’ to indicate the class body. For example, the ‘Fowl’ class is written as

```
public class Fowl {  
    ...  
}
```

- (1) Define classes for ‘Duck’ and ‘Goose’ that inherit from ‘Fowl’.
- (2) Implement a function that takes in an input parameter  $N$  and returns an array of  $N$  Fowl objects, where  $\frac{1}{5}$ th of the Fowls are randomly geese.
- (3) Implement a function that takes in an array of ‘Fowl’ objects and returns an array of ‘Goose’ objects that were found in the input array.

## Problem 5: Spreadsheet

In this problem we will define a basic spreadsheet data structure, where each cell can contain different types of data. The most basic type will be a Cell and we will represent several basic types like NumericCell, StringCell, and LogicalCell.

These derived types of cells will each store some data and pairs of Cells will be able to do some basic operations with one another:

- Add two cells together. Each type of cell can only be added with another cell of the same type.
  - Generate a string representation that can be printed to a screen.
  - An overload of `Object.toString()` for debug visualization.
- (1) Implement the class Cell, which represents an empty cell. It should have a single function called `add` which takes another cell and returns a new Cell instance.

```
public class Cell {
```

```
}
```

- (2) Implement the other types of cell and their data. Make sure that they know how to add with one another using overloading.:

- (3) Implement a spreadsheet class that initializes a grid of Cell objects and has a function add that takes in three cell coordinates

```
//sheet.add(x0,y0,x1,y1,x2,y2) // -> adds cell at [x0,y0] with cell at [x1,y1] and writes the  
result to [x2,y2]
```

## Problem 6: Getting Packages

Implement a class that represents a package being shipped. Each package has a unique ID, current location, and a destination, where the ID is a single integer, locations can be represented by a single string. This question focuses around the task of giving each package a unique ID without having to pass the ID in as a parameter.

```
class Package {
```

- (1) Create the data members.

- (2) Create a variable that stores an ID that has not been used yet. This variable is shared by every instance of the class Package.

Every time a new Package is constructed this variable is incremented. After  $n$  packages have been created the IDs  $0, 1, 2, \dots, n-1$  will have been used and so the next package can use the ID  $n$ .

```
// Package p0 = new Package(); //p0.ID = 0
// Package p1 = new Package(); //p1.ID = 1
// Package p2 = new Package(); //p2.ID = 2
// ...
// Package pn = new Package(); //pn.ID = n-1
```

- (3) Implement a constructor for the package where every parameter is passed in.
- (4) Implement a constructor for the box that gets passed every piece of data except the ID, but uses the aforementioned special variable to set a unique ID parameter.
- (5) Implement a constructor that doesn't require knowing the current location and will automatically set that to be "unknown".
- (6) Implement getters or setters to indicate which properties can or cannot be changed (only the current location can be changed).
- (7) Implement a function that returns true if the package has been lost (its destination is "unknown").

}

## Problem 7: Package Factory

Another way for giving each package its own ID is to use what's called a *factory* class. Rather than

use the shared variable as above we can let another class manage which IDs have already been used. We'll call this class PackageFactory.

```
class PackageFactory {
```

- (1) Implement a member that stores an ID that has not been used thus far and give it a default value of 0.
- (2) Implement a function called create that returns a Package with a unique ID according to the same scheme as before. This method is NOT allowed to use the shared variable from the previous problem. The following code should work:

```
PackageFactory factory = new PackageFactory();
Package a = factory.create(currentLocation, destination);
Package b = factory.create(currentLocation2, destination2);
// a.getId() == 0;
// b.getId() == 1;
```

```
}
```

## Problem 8: Bounding Boxes

In graphical applications it's common to have multiple ways to make sure that assets do not overlap with one another. You may assume this Point class exists without setters/getters to reduce writing required:

```
public class Point {
    public int x,y;
    Point(int x, int y) { this.x = x; this.y = y; }
}
```

Also we can assume that we have a simple

```
public class Polygon {
    // returns the vertices of the polygon
    public Point[] getPoints();
}
```

We begin by defining a BoundingBox class that can be incrementally expanded by adding points. The corners of the bounding box of a set of points are defined by

$$(\min_i p.x, \min_i p.y) \text{ and } (\max_i p.x, \max_i p.y).$$

This can be *incrementally* generated by creating an “empty” bounding box and adding each point to it one at a time:

```
BoundingBox b = new BoundingBox();
b.expand(new Point(2,3)); // min: 2,3 max: 2,3
b.expand(new Point(0,0)); // min: 0,0 max: 2,3
b.expand(new Point(10,0)); // min: 0,0, max: 10,3
b.expand(new Point(-1,5)); // min: -1,0 max: 10,5
```

For simplicity we will assume that the  $x$  and  $y$  coordinates all lie within  $[-10000, 10000]$ .

```
public class BoundingBox {
```

- (1) Define members that can store the smallest coordinate and largest coordinate in the box as well as a polygon.
- (2) Define members of the class that store the min and max values allowed for a bounding box (-10000 and 10000).
- (3) Define constructor that initializes an empty bounding box whose min point is (10000,10000) and max point is (-10000,-10000) without typing those numbers explicitly.
- (4) Define a function that takes in a single Point and expands the bounding box to include that point.
- (5) Define a function that takes in a single Polygon and expands the bounding box to include that polygon.
- (6) Define a function that returns true if the bounding box is empty (this is equivalent to the box having negative volume).
- (7) Define a function that identifies the intersection between this bounding box and another bounding box.
- (8) Define a function that identifies whether two bounding boxes intersect with one another (this is equivalent to checking if the intersection is empty)



## Problem 9: Entity Handles

Say our game engine allows for the creation of a large number of Entity objects - that mainly represent any sort of thing in the game that can be drawn (characters, buildings, etc). Every entity that is created needs to be searchable by its name, so we have to keep a collection of them. We will create a class that keeps its own list of existing entities to allow for lookup.

- (1) Define the 'Entity' class that has two members, one called name which is the name, and another that stores the entity's bounding box.
- (2) Define getters and setters for its members.
- (3) Define a property held by the Entity class that contains a dynamic array of every Entity that is created.
- (4) Define a constructor for Entity that sets its name and bounding box, and then appends this entity to the list of entities.
- (5) Define a function findEntity that looks for an entity with a particular name. If no name is found this function should return null.
- (6) Define a constructor for Entity that takes no arguments, but declares a name like Entity 1, Entity 23, etc. This name cannot have been used thusfar (try using findEntity to help with this process).
- (7) Write a function that checks if two entities collide with one another.
- (8) Define a class called Sphere that has a specified center and radius, as well as a custom getter for the bounding box that doesn't use the internally held member, instead uses the current center and radius.
- (9) Define a class called PointCloud that stores a list of points, as well as a custom getter that doesn't use the internally held member, instead uses the current positions of the points.
- (10) Define a class called ConvexPolygon that stores a list of points in counter-clockwise order, as well as a custom getter that computes the bounding box according to the current positions of its vertices. You may not implement this function, and note that a Polygon is a sort of point cloud with connections between points.