```
// Some useful java functions

// initialize just the outer dimension
int[][] x = new int[4][];
x[3] = new int[]{1,2,3};
int length = x.length;
// grid of values
int[][] y = new int[4][5];


class ArrayList<T> {// same for List
    T at(int index);
    void add(T newValue);
    void remove(int index);
    int length();
}

public class String {
    char charAt(int index);
    int length();
    char[] toCharArray();
}


public class Random {
    // returns a value between 0 and bound (including 0 but not including bound)
    int nextInt(int bound);
}
```

Problem 1: 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 2: Variant

Sometimes we need to use a single variable to store multiple types of values, like with Vulkan or Google's protobuf libraries. In this problem we will creat a single class that can store different types of object internally.

For simplicity, we will only keep track of three potential types: Character,Integer,Double. The full implementation of this problem would require writing a version of certain functions for each type (like getters and setters), but to save on writing you will only need to write one implementation to show you know what to do. Once you've written a function like isInteger you can assume that isCharacter and isDouble exist as well.

```
// Variant charVar = new Variant(new Character('a'));
// Variant intVar = new Variant(new Integer(3));
// Variant doubleVar = new Variant(new Double(1.5));
public class Variant {
```

(1) The variant class stores two pieces of data: an object and an integral index to indicate which type of object it is. Create these members.

(2) Rather than keep track of which indices are associated with which types, declare some publicly visible members shared between every Variant called CHARACTER_INDEX, INTEGER_INDEX, DOUBLE_INDEX. You need to specify the values of these numbers, but just about any numbers will work.

(3) Write a getter for the internally held index.

(4) Write a function called isInteger that checks if the internally held index indicates the internally held object is of type Integer using the numeric variable defined previously.

(5) Write a function called isValid that uses a switch/case to identify if the internally held index is set appropriately (for example it can only be INTEGER_INDEX if the object member is an instance of Integer).

(6) Write a sort of getter called getCharacter that returns the internally held object. This returned object should be properly converted to a Character, but only if the held index is set to CHARACTER_INDEX. If it isn't set to that this function returns null.

```
// Character c = charVar.getCharacter(); <- gets the Character holding 'a'
// Character c2 = intVar.getCharacter(); <- returns null
```

(7) Write two constructors: one that takes in a Double and one that takes in a Character. Even though these constructors only take in one argument they must set all members properly.

```
// Variant c = new Variant(new Character('c'));
// Variant d = new Variant(new Double(3.0));
```

Problem 3: Fractions

We've seen float and double types but unfortunately they can't accurately represent fairly common numbers like $1/3$ and $1/10$. In this problem we will represent such numbers by storing the numerator and denominator of integral numbers.

```
class Fraction {
```

(1) Declare members for the numerator and denominator of a number.

(2) Define getters and setters for the members. The denominator cannot be 0. If the denominator is invalid it is set to 1.

(3) Define a constructor that takes in a numerator and denominator, making sure any constraints on the denominator are preserved.

```
// Fraction a = new Fraction(3,5); // stores fraction 3/5
// Fraction b = new Fraction(3,0); // stores fraction 3
```

(4) Define a constructor that only takes in an integral number. Note that any number n can be represented as a fraction with $n/1$.

```
// Fraction c = new Fraction(10); // stores fraction 10/1
```

(5) Write a member function add that takes in another fraction and returns the sum of the two. Note that division cannot be used here because we storing integer values.

```
// Fraction d = c.add(a); // stores fraction (10*5+3*1)/(5*1)
```

(6) Write a member function compare that returns a negative number if $a < b$, 0 if $a == b$, and a positive number if $a > b$. Note that division cannot be used here because we storing integer values so you will want to use a bit of algebra to transform the equation into one without division. Also, you are free to pick any positive/negative numbers you want for this comparison operator.

```
// -12 == a.compare(b)
// 0 == c.compare(c)
// 47 == c.compare(a)
```

(7) Write a member function round that converts the number to a double. Note that the fraction $1/2$ should return .5 and not 0.0.

```
// 3.0 == b.round()
// .59999 == a.round()
```

Problem 4: Pascal's Triangle

Pascal's triangle is a generally useful structure in a variety of mathematical areas and in this problem you will implement a class that stores it and uses it to solve some problems. It starts off something like

```
     1        //row 0
    1 1       //row 1
   1 2 1      //row 2
  1 3 3 1     //row 3
1 4 6 4 1     //row 4
...
```

The entries of each row is constructed by taking the sum of the two values "above" it.

```
public class PascalTriangle {
```

(1) Declare a member that can store a number of rows of Pascal's triangle. Note that we will not be changing the size of the triangle after constructing it.

(2) Implement a constructor that takes in the number of rows to be constructed in the triangle. The constructor should fill the held triangle up to the number of rows requested.

```
// PascalTriangle pt = new PascalTriangle(3);
// pt.getValue(2,1) == 2
// pt.getValue(3,0) == -1
```

(3) Implement a function called getValue that returns the value of the triangle at the specific row and position. If this value is outside of the range of the triangle stored then this function should return -1.

```
// pt.getValue(2,1) == 2
// pt.getValue(3,0) == -1
```

Problem 5: 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 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 6: 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).

```
public class BorrowableBook
```

(5) Create members to represent the aforementioned details

(6) Implement a constructor that passes all of the forward 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, produces a brief email detailing how late the book is.

```
// System.out.println(borrowableBook.getLateEmail(currentDate));
// Says: "Hi BorrowerName, you borrowed ThisBookName, which was due OnThisDate.
//You are ThisLate." where BorrowerName/ThisBookName/OnThisDate/ThisLate
// are filled out properly
```

}

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. To make the bookstore financially viable the book should cost 1 penny more when ti is sold.

(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 7: 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 $0, 1, 2, \ldots, n-1$ 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 8: 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 9: Photoshop

Photoshop provides tools to arrange assets like images and text to generate a new image. In this problem we will create primitive representations of assets like text boxes and images and a project that contains these assets.

```
public class Asset {
```

(1) Specify the member data for an asset. An asset stores two numbers to represent the bottom-left corner where the asset should be drawn and two numbers to represent the width and height of the asset.

(2) Specify a constructor that takes in each member as a parameter.

```
// // starting from 20,30 and width 100 height 150
//Asset A = new Asset(20,30,100,150);
```

(3) Define a constructor that takes no parameters. It should by default set the bottom-left corner to 0,0 and the height and width should be 100,150.

```
// // starting from 0,0 and with 100 height 150
//Asset B = new Asset();
```

(4) Write getters and setters for the above members

(5) Make functions called getTop and getRight that report the top and left extents of the image (the top is bottom + height and the right is left + width.

}

The first type of asset we will represent is an Image. An image asset stores the path to an image that will be drawn in the region defined by the asset. The path is to be stored a plain string.

(6) Declare the Image class that is derived from the Asset class and its single member.

(7) Write a constructor that takes in the path to an image, its bottom-left corner, and its dimensions. This constructor should call the Asset constructor appropriately.

```
// Image image = new Image("/usr/include/img.png", 20,30,100,150);
```

(8) Write a getter and setter for the path to the image. The image path should not be changed after construction but the setter should print out a warning message if the path is empty.

}

The second type of data we will represent is a text box. A text box is an asset that also stores some text data, a font name, and a font size.

(9) Declare the Textbox class that is derived from the Asset class and its members.

(10) Define a constructor that populates a text box but uses the default constructor the Asset class.

(11) Write getters an setters for the members of the class. Any value is allowed to change, but the font must be at least 6 - if the value is less than 6 then it is set to 6.

```
// TextBox textBox = new TextBox("Hello world!", "Helvetica", 12);
```

}

Now we can represent a project, which just contains a dynamic list of assets.

```
class Project {
```

(12) Define a member that stores a dynamic list of assets.

(13) Define a member function addImage that takes in the parameters for constructing an image, constructs one, and generates an image, and adds it to the dynamic list of assets.

```
// Project project = new Project();
// project.addImage("/tmp/text.gif",100,200,30,40);
```

(14) Define a member function addTextBox that only takes in the text data, constructs a text box whose bottom-left corner is 500,500 and width,height are 100,500, and adds it to the dynamic list of assets.

```
// project.addTextBox("Hello there");
// project.addTextBox("Bye");
```

(15) Define a member function printTextBoxes that iterates through the set of assets, and if an asset is a TextBox prints out the text data of the TextBox. Images should not generate any text.

```
// project.printTextBoxes();
// // prints:
// // Hello there
// // Bye
```

```
}
```

Problem 10: Dining Menu

Restaurants present the dishes they produce in a menu with some information that helps customers pick the items they want. Although historically menus could be made on a single Word document or simply written on a blackboard, in this modern era where we can order dishes online restaurants now need to present their menus on paper, on a website, and on an app. In this problem we will develop a simple tool for building, storing, and manipulating a menu. These menus will comprise of a list of dishes and only be active during a specific interval of time. We'll build several classes to help represent them.

```
public class Dish{
```

(1) In our menu we will keep track of several pieces of data for each item in the menu: its name, a description, its price (in double), if the dish is gluten free, and if the dish is vegan. Declare these members.

(2) Declare getters and setters for each of these members. We will allow for every member to be changed by others.

(3) Declare a constructor that allows a user to specify all of these members.

```
// //vegan, gluten free
//Dish injera = new Dish("Injera",
        "A flat-bread made with teff flour.", 3.99, true, true);

// //not vegan, gluten free
//Dish omelette = new Dish("Omelette",
        "Mushrooms and vegetables wrapped in an egg", 6.99, true, false);
```

(4) Declare a constructor that by default specifies that a dish is not vegan and has gluten.

```
// //not vegan, not gluten free
//Dish meatloaf = new Dish("Meatloaf",
        "A combination of beef, eggs, and flour", 10.99);

//not vegan, not gluten free
//Dish burger = new Dish("Burger",
        "A beef patty put between two slices of bread", 8.99);
```

}

A menu is comprised of a list of dishes and the hours that the dishes are available. For this problem we will assume that the number of dishes in a menu does not change.

```
class Menu {
```

(5) Specify the members of this menu. The hours that the dishes are available should be represented by two numbers between 0 and 24. Being available from ‘10‘ to ‘15‘ means available from 10am to 3pm, not including 3pm.

(6) Write a constructor that takes in the members of the class.

```
// Dish[] dishes = {...};// somehow a menu is constructed
// // this menu serves the dishes and is available from 6am to 11am
// Menu breakfast = new Menu(dishes, 6,11);
```

(7) Write a getter for the name of a dish that takes in an argument and a size() function that returns the number of dishes.

```
// String dishName = breakfast.getDishName(0);
// int numDishes = breakfast.size();
```

(8) Implement a member function getDish that takes in the index of a dish as input.

```
// Dish firstDish = breakfast.getDish(0);
```

(9) Write a function called isActive which takes in an hour and determines if the menu is currently active

```
// true == breakfast.isActive(6);
// true == breakfast.isActive(10);
// false == breakfast.isActive(11);
// false == breakfast.isActive(15);
```

(10) Write a function called getPrice that takes in the name of a dish and returns its price. We'll assume that the user of this function always correctly types in the name of a dish that exists in the menu.

```
// 8.99 == breakfast.getPrice("Burger");
// 3.99 == breakfast.getPrice("Injera");
```

(11) Write a function called getPrice that takes in a list of dish names and returns the total price. We'll assume that the user of this function always correctly types in the name of dishes that exists in the menu. This function is not allowed to access the list of dishes directly.

```
// String[] dishes = {"Burger", "Omelette"};
// 12.98 == breakfast.getPrice(dishes);
```

(12) Write a function called getMenu that takes in a list of dish names and returns their total price. This function does not require a Menu object to be constructed to call it, but it takes constructed Menu objects as inputs.

```
// // The breakfast menu, lunch menu from 11am–3pm, dinner menu from 3–8pm
// Menu[] menus = {...};
// Menu breakfastMenu = Menu.getMenu(menus,8);// returns the breakfast menu
// Menu lunchMenu  Menu.getMenu(menus,14);// returns the lunch menu
```

Problem 11: 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 12: 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 $0, 1, 2, \ldots, n-1$ 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 13: 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 14: Expression Representation

Computers are tasked with representing and evaluating complex sorts of expressions, and these representations are usually done with classes. Expressions are typically defined in terms of other expressions. Say we have an expression A and an expression B. Then their sum $A + B$ is an expression, as is the negation of A, $-A$. In this problem we will implement a simple expression representation for basic arithmetic. The basic object of study will be a Expression object that by default just returns 0, but in every class we will write in this problem we will overload this expression.

```
class Expression {
    public double evaluate() {return 0.0; }
}
```

This expression system will work by declaring variables, assigning those variables, and then evaluating expressions that use those variables.

```
Variable x = new Variable(3.0);
Variable y = new Variable();
y.setValue(5.0);

Sum sum = new Sum(x,y);// sum.evaluate() is 8.0
Multiplication prod = new Multiplication(x,sum); // prod.evaluate() is 24.0
Negate neg = new Negate(prod); // neg.evaluate() is -24.0
```

(1) We can treat a variable as a sort of expression that stores a value and can be changed. Finish declaring this as an expression and define a single value as a member.

```
class Variable
```

(2) Implement a getter and setter for this function. Note that the only "getter" required is just the evaluate function.

(3) This variable class should also have two constructors: one that takes in no argument but sets the default value to 0.0. Implement them.

```
}
```

(4) Negating an expression is yet another expression whose evaluation is the negation of the evaluation of the sub-expression. Implement an expression that negates another one by first finishing the class statement and declaring its member, which is a single expression.

```
class Negate
```

(5) Implement a constructor that takes in the sub-expression.

(6) Implement getters for the member and the evaluate function which should call the underlying operation and negate its value.

}

(7) Adding two expressions is an expression that computes the sum of the evaluation of the two sub-expressions. Start off by declaring two members, which are also expressions.

```
class Sum
```

(8) Implement a constructor that takes in the two sub-expressions.

(9) Implement a getters for the members and the evaluate function which should call the underlying operation and negate its value.

}