

WES-CS GROUP MEETING #12

Exercise 1: Practice with Inheritance

The code shown below defines three classes: Drink, CocaCola, and Milk. Coca-Cola and Milk are each subclasses of Drink.

```
public class Drink {
    private double myPrice;
    public Drink(double price) { myPrice = price; }
    public double getPrice() { return myPrice; }
    public String getName() { return "Drink"; }
    public void printLabel() {
        System.out.println( getName() );
        System.out.println("Price: " + getPrice());
    }
}

public class CocaCola extends Drink {
    private static final double PRICE = 1.25;
    public CocaCola() { super(PRICE); }
    public String getName() { return "Coca-Cola Classic"; }
}

public class Milk extends Drink {
    private static final double PRICE = .75;
    private int percentFat; // either 1 or 2
    public Milk( int fat ) {
        super(PRICE);
        percentFat = fat;
    }
    public String getName() { return "Milk"; }
    public int getFat() { return percentFat; }
}
```

Your job is to complete the `DrinkTester` class below by writing the `getDrink` method and adding the code in the `main` method that prints the fat content if the drink is milk.

```
import java.util.*;

class DrinkTester {
    // WRITE THE getDrink METHOD HERE
    public static void main(String[] args) {
        for (int k=1; k<6; k++) {
            Drink drink = getDrink();
            drink.printLabel();
            // ADD CODE HERE TO PRINT THE FAT CONTENT IF drink IS MILK
        }
    }
}
```

Part (a)

The `getDrink` method should create and return a `Drink`, randomly choosing between `CocaCola` and `Milk`. If it chooses `Milk`, it should randomly choose between 2% and 1% for the fat content. Before writing the code, think about the following questions:

- Should the `getDrink` method be a *static* or a *non-static* method?
- What are the possible types that will actually be returned by the `getDrink` method?
- What should the return type of the `getDrink` method be?
- When `printLabel` is called (just after the call to `getDrink`), it causes a call to `getName`. There are three versions of that method, defined in the `Drink`, `Coca-Cola`, and `Milk` classes. What determines which version is actually called?

Part (b)

What code could we add at the end of the `main` method that would figure out whether `drink` is a `Milk` object, and if so, would print its fat content?

Exercise 2: Two-D Arrays

Most groups did not get to do this exercise last time, so here it is again.

For many board games (e.g., chess, checkers, go) the board can be represented using a 2-dimensional array of integers, where the value in position `[j][k]` tells what piece (if any) is currently at that position. When deciding whether a player has won the game, it is often useful to look for certain patterns on the board.

These ideas lead to the following (partially specified) class.

```
public class BoardGame {
    /*** fields ***/
    private int[][] board;

    /*** public methods ***/
    public BoardGame() { ... } // constructor
    public boolean hasPattern( int[][] pattern ) {
        // part (b)
    }

    /*** private methods ***/
    private boolean isMatch(int[][] pattern, int row, int col) {
        // part (a)
    }

    private boolean isMatchIgnoreZeros(int[][] pattern,
                                       int row, int col) {
        // part (c)
    }
}
```

Part (a)

The `isMatch` method of the `BoardGame` class should return `true` iff the given pattern matches the portion of the board that has the given row and column numbers as its upper-left corner.

For example, assume that the board array is as follows:

1	2	3	4
5	6	7	8
9	10	11	12

Below are the results of some calls to `isMatch`.

pattern array	row	col	value returned by the call <code>isMatch(pattern, row, col)</code>						
<table border="1"><tr><td>1</td><td>2</td><td>3</td></tr><tr><td>5</td><td>6</td><td>7</td></tr></table>	1	2	3	5	6	7	0	0	true
1	2	3							
5	6	7							
<table border="1"><tr><td>1</td><td>2</td><td>3</td></tr><tr><td>5</td><td>6</td><td>7</td></tr></table>	1	2	3	5	6	7	0	1	false
1	2	3							
5	6	7							
<table border="1"><tr><td>6</td><td>7</td></tr><tr><td>10</td><td>11</td></tr></table>	6	7	10	11	1	1	true		
6	7								
10	11								
<table border="1"><tr><td>6</td><td>7</td></tr><tr><td>10</td><td>11</td></tr></table>	6	7	10	11	0	1	false		
6	7								
10	11								
<table border="1"><tr><td>6</td><td>7</td></tr><tr><td>10</td><td>14</td></tr></table>	6	7	10	14	1	1	false		
6	7								
10	14								

Write method `isMatch`. Assume that both the `board` field and the `pattern` parameter are non-empty, rectangular arrays

Part (b)

The `hasPattern` method of the `BoardGame` class returns `true` iff the given pattern occurs somewhere in the board array.

For example, assume again that the board array is as follows:

1	2	3	4
5	6	7	8
9	10	11	12

Below are the results of some calls to `hasPattern`.

pattern array	value returned by the call <code>hasPattern(pattern)</code>						
<table border="1"><tr><td>1</td><td>2</td><td>3</td></tr><tr><td>5</td><td>6</td><td>7</td></tr></table>	1	2	3	5	6	7	true
1	2	3					
5	6	7					
<table border="1"><tr><td>6</td><td>7</td></tr><tr><td>10</td><td>11</td></tr></table>	6	7	10	11	true		
6	7						
10	11						
<table border="1"><tr><td>12</td></tr></table>	12	true					
12							
<table border="1"><tr><td>6</td><td>2</td></tr><tr><td>7</td><td>3</td></tr></table>	6	2	7	3	false		
6	2						
7	3						
<table border="1"><tr><td>1</td><td>3</td></tr><tr><td>5</td><td>7</td></tr></table>	1	3	5	7	false		
1	3						
5	7						

Write method `hasPattern`. Assume that both the board field and the pattern parameter are non-empty, rectangular arrays

Part (c)

Sometimes when you look for a pattern you only care about matching some parts of the pattern. For example, to look for a diagonal win for X in tic-tac-toe (three X's in a row on the diagonal, using 1's to represent X's), all you care about is whether there are three 1's in a row on a diagonal-- you don't care what values if any are around them. You'd like to be able to use patterns like these:

1	0	0
0	1	0
0	0	1

0	0	1
0	1	0
1	0	0

where the ones need to be matched, but the zeros in the pattern can correspond to any values in the board array.

Write the `isMatchIgnoreZeros` method so that only non-zero entries in the pattern have to match the corresponding positions on the board.

Exercise 3: Programming Connect-4

For this exercise, you will write a class called `Connect4` that can be used to allow two people to play a game of Connect-4 against each other, using your program instead of the actual Connect-4 game.

You should write the `Connect4` class so that it is possible to create a `Connect4` object and to play one game by calling the object's `play` method. The `Connect4` class should include methods to do the following tasks; some of the methods will include calls to other methods.

- Play an entire game.
- Print the current gameboard.
- Print a message asking the next player (player 1 or player 2) to enter their next move; make sure it's a legal move (giving an error message and asking for the move again if not); update the gameboard to reflect the move. Note: It is a good idea to write methods to handle the various sub-tasks that this method will need to do. Also, you will need to think about how a player should specify their next move.
- Determine whether the game is over, and if so, who won (or is it a tie). Think about how to use the `isMatchIgnoreZeros` method that you wrote for Exercise 2 (and a `hasPatternIgnoreZeros` method that calls `isMatchIgnoreZeros`) to check for a winner.

Part (a)

Start by working with a partner to design the `Connect4` class:

- Write a list of the fields that a `Connect4` object should have, and what their types should be (you might find it easier to decide on the fields as you design the methods).
- Write appropriate names for each of the methods described above, and also for any “helper” methods that you think should be included in the class.
- Decide how many parameters each method should have (and what the parameters’ types should be), as well as whether the method should be `void` or should return a value (of what type?).
- Write pseudo-code for each method that makes it clear how the method will work. For example, will it have a loop? If so, under what conditions will the loop end?
- Decide what parameters the constructor should have and what it should do.

When you’re finished, compare your design with another pair’s design. What parts of the design are different? What aspects of each design seem particularly good?

Decide on a final design (perhaps combining some of your ideas with those of another pair).

Part (b)

Write the actual code for the `Connect4` class and test it! It is a good idea to do lots of little tests as you go along, instead of writing the whole class and then trying to test it. For example, you might start by writing the constructor and the method that prints the board. Make sure those work. Then write the method that figures out whether a particular player has won, and test that method (to do the test, you will probably need to modify the constructor so that it creates a non-empty gameboard).