WES-CS GROUP MEETING #10

Exercise 1: Bad Loop! (Code Tracing and Arrays)

We've seen that code tracing is an important technique to figure out what code does. We'll see in this exercise that we also can use this technique to figure out why code isn't working as desired. When tracing code, we are the computer! We do a line-by-line execution of the code accompanied by drawing and changing memory diagrams as well as writing the output that would be displayed. This helps us see what is happening to the information being used by and generated in a program.

The code shown below was supposed to set the `characters` array to hold the characters

```
a * b * c * d * e * f * g * h * i * j *
```

However, the code does not work correctly.

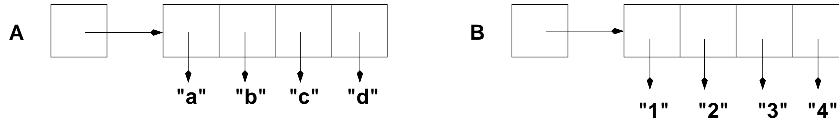
```
char [] characters = new char [20];
for (int n = 0; n<characters.length; n++) {
    characters[n] = '*';
}

char a = 'a';
for (int n = 0; n < characters.length; n++) {
    while (a != 'k') {
        if (n%2==0) characters[n] = a;
        a = (char)(a + 1);
    }
}
```

First, trace the code to find out what the final values in the `characters` array actually are.

Then, fix the code so that it works as intended.

Exercise 2: What's Mine is Yours (Arrays and Aliasing)

Assume that you have two `String` arrays, `A` and `B`, initialized as follows.



For each of the pictures shown below, write code that would change arrays `A` and `B` from their initial values as shown above, to the new value shown in the picture. If there is no such code, explain why.

Picture 1	**Picture 2**
A "a" "b" "c" "d" B	A "a" "b" "c" "d" B
Picture 3	**Picture 4**
A "a" "b" "c" "d" B	A "b" "c" "d" "a" B "2" "3" "4" "1"

Exercise 3: Lights-Out! (Logic & Class Design)

For this exercise you will program a game for one player. The game is played on a row of lighted buttons, and the aim is to turn all the lights off. The tricky part is that pushing a button changes the on/off state of that button plus its two neighbors. Also, the first and last buttons are considered to be neighbors.

Below is an example of two rounds of the game to show how your program should work. An `O` is used for a button that is lit up, and an `X` is used for a button whose light is off.

```
Enter number of buttons (3 or more): 4

 1  2  3  4
 ----------
 O  O  O  O

Enter which button to push: 3

 1  2  3  4
 ----------
 O  X  X  X


Enter which button to push: 1

 1  2  3  4
 ----------
 X  O  X  O
```

You will write a `Game` class and a `PlayGame` class. The `PlayGame` class will have a main method that works as follows:

1. Ask the user for the number of buttons.
2. Create a `Game` class for that many buttons.
3. Display the row of buttons on the screen with the button numbers.
4. Repeat until all lights are off:

   Ask the player to choose which button to push.

   Read the number entered by the player.

   Update the row of buttons to reflect the result of pushing the chosen button, then print the row of buttons.

5. When all lights are off, print "`Congratulations you win!`"

Part (a)

Work in three groups to design the `Game` class. What fields and methods should it have? Should they be public or private, static or non-static?

If you finish early, see if you can come up with a strategy to win the game for a row of 4 buttons. How about a row of 5 buttons? A row of N buttons?

When everyone is ready, compare your class designs. If you want to change your design based on what another group did, that's fine.

Part (b)

Now each group use one of the laptops to write the methods for the `Game` and `PlayGame` classes. Test your program.

Part (c)

Swap laptops with another group. See if you can make the other group's program do something unexpected. If you can, show the designers and ask them to fix their code.

Exercise 4: My Favorite People (Arrays of Objects)

Assume that the following `Person` class has been defined.

```
public class Person {
    private String name;
    private int age;
    private String eyeColor;

    // constructor
    public Person(String aName, int anAge, String aColor) {
        name = aName;
        age = anAge;
        eyeColor = aColor;
    }

    // accessor methods
    public String getName() { return name; }
    public int getAge() { return age; }
    public String getEyeColor() { return eyeColor; }
}
```

First, choose four people in your group and draw an array called *people* that contains four *Person* objects representing those four people (use strings with all capital letters for the names and eye colors).

Now (in groups of two or three) play a game of *concentration* using one of the sets of cards. One pack in each set has a Java expression. The other pack in each set has a value. Start with all of the cards upside down. When it's your turn, you turn over one card of each color. If they match you keep those two cards, and keep going; otherwise, your turn is over.

The game ends when you run out of expression cards or the remaining value cards don't match any expression cards.

Whoever has the most cards wins!

Exercise 5: Elementary my Dear Watson! (Logical Thinking)

As you probably know by now, sometimes errors in your program cause exceptions to be thrown when the program is run (e.g., you may get a *NullPointerExecption* if you forget to initialize some object, or you may get an *ArrayIndexOutOfBoundsException* if you use an index that is too large or too small).

One way to debug your program is to "think backwards" from the point of the exception: what could have happened just before the exception was thrown, and what could have happened just before that, and so on, until you get to the point of the actual mistake in your code.

For this exercise, we will practice thinking backward in the context of a chess game. Keep the following in mind:

- A player is not allowed to move into check. (Among other things, this means that two kings can never be next to each other horizontally, vertically, or diagonally.)
- A pawn normally moves one space forward. However, on its very first move it is allowed to move two spaces forward, and to capture an opponent's piece it moves one space forward *diagonally*.
- If a pawn arrives at the end of the board, it is *promoted*: replaced by a queen, castle, bishop, or knight.

Part (a): Look at the board below, which shows a position in a game of chess. You are told that black moved last. Your job is to figure out what black's last move was, and what was white's move before that. There are two solutions: the easier solution only works if the person playing white is sitting at the *north* end of the board (i.e., the white pawn is moving south); the other (more difficult) solution works whether the person playing white is sitting at the north or the south end of the board (i.e., the white pawn can be moving either way). Note that the moves are not necessarily good ones, but they are all legal.

NORTH

black king		white king					
							white pawn
						white bishop	

SOUTH

Part (b): Now look at the board below, which shows the final position of a game (the game is over because white has check-mated black).

black king		white king					
				white queen			
		white bishop		white pawn			
			white pawn				
							white pawn
							white bishop

This time, your job is to figure out whether the person playing white was sitting on the north or the south side of the board. To do that, you will need to figure out what white's last move was. There are several possibilities, so to narrow it down, you'll need to figure out what black's last move before that could have been in each case. In all but one case you should find that in fact there would have been *no* possible move for black, and that should let you figure out where the person playing white was sitting.

Good luck!

Green/Blue/Pink Cards

```
people[1].getAge()
```

```
(people[0].getEyeColor().compareTo(people[3].getEyeColor())) < 0
```

```
people[people.length-1].getAge() > 18
```

```
people[2].getName().length() < 6;
```

```
(people[2].getName().substring(0,1)).equals((people[3].getEyeColor().substring(0,1)))
```

```
people[1].getEyeColor().charAt(1)
```

18	19	20
18	true	false
true	false	true
false	true	false
L	R	A