Introduction to data structures, algorithms, and object-oriented programming

DAY 8
Today in lab, you will take a practice exam that is in the format of the on-line exam you'll take on the 29th. You should try to solve these problems on your own (but you can ask me and the coaches questions that we might not be able to answer during the actual exam).

My interest in giving you this practice exam is to find out how many problems most of you can finish in 75 minutes. The lab will not be posted until lab time starts. If you don't finish, you will have time to complete the problems as practice for the midterm.

If one of the problems stumps you, got on to the next problem.
Reverse a String

Since Strings are numbered like arrays, and because the String class has a function length(), it is easy to use a for loop to build and return the String in reverse.

(in class exercise)

Convert the method to a palindrome checker.
Write a JUnit tester class

After writing the reverseString class, write a tester class to test various String inputs.

Any method with deterministic operation on a given input produces the same output and that output is testable. Deterministic means there are no random actions in the method.
Making many nested decisions

WorkTime:
This program determines whether you should be working or playing based on day (SUNDAY (0) through SATURDAY (6)), hour (0 through 23), minute (0 through 59) and personality: GEEK, DORK or NERD.

A dork works 9am to 5pm, Monday through Friday. Nerds work like dorks but also put in extra hours on Sundays from 1pm to midnight. Geeks (aka graduate students) don’t recognize the difference between work and play, but they do sleep each morning from 3am to 11am. Let’s say a geek is working whenever he/she is awake. Complete the definition of the workTime method in this project to return true or false depending on the input parameters. Make appropriate use of switch and if-else statements.
The parameter to the main method is now recognizable to you as an array of Strings.

If you compile a program that uses the input array args and then run it at the command line, you must enter strings to be used in the args array inside the class.
public class Main {
    public static void main(String[] args) {
        if (args.length == 0) {
            System.out.println("No Command Line arguments");
        } else {
            System.out.println("You provided " + args.length + " arguments");
            for (int i = 0; i < args.length; i++) {
                System.out.println("args[" + i + "]: " + args[i]);
            }
        }
    }
}

Reading Strings from the command line
From outside classes:

All public methods and data fields, both static and non-static (instance fields), of a class are accessible through an object created from a particular class.

Within one class:

Static (aka class) methods can access static data fields within the same class directly, but must access instance variables or instance methods through objects.

Instance methods can access all types of data fields and static methods.
Sometimes referred to as **data** hiding, is the mechanism whereby the implementation details of a class are kept hidden from the user. The user can only perform a restricted set of operations on the private members of the class by executing public methods.
Data Encapsulation Example

```csharp
class Program {
    public class Account {
        private decimal accountBalance = 500.00m;
        public decimal CheckBalance() {
            return accountBalance;
        }
    }
    static void Main() {
        Account myAccount = new Account();
        decimal myBalance = myAccount.CheckBalance();
        /* This Main method can check the balance via the public
         * "CheckBalance" method provided by the "Account" class
         * but it cannot manipulate the value of "accountBalance" */
    }
}
```
Passing data into methods

Primitive types:
• passed into methods using a "pass-by-value" scheme.
• local copies of primitive parameters are created, even if the parameters and arguments have the same name.

Object types:
• passed into methods using "pass-by-value".
• actually passing in only the address(a reference) to the object.
• changes made to the object inside (local to) the method will also be made to the object in the larger scope in which the call is made.
(1) class Octopus {

    // instance data field
    private int numLegs;

    // class (static) data field
    private static final int MAX_LEGS = 8

    // note use of this keyword in next 3 methods
    public Octopus(int numLegs) {
        this.numLegs = numLegs;
    }

    public void setNumLegs(int numLegs) {
        this.numLegs = numLegs;
    }

    public int getNumLegs() {
        return this.numLegs;
    }

}
Passing data into methods

Ex: Suppose we had methods (2) through (5) declared in the Octopus class:

(2) public static void main(String[] args) {
    Octopus octy = new Octopus(MAX_LEGS);
    // sending an object into a method
    removeLegs(octy);
    Octopus octet;
    // returning an object from a method
    octet = mutantOctopus();
}

(3) public static void removeLegs(Octopus octo) {
    octo.setNumLegs(octo.getNumLegs() - 1);
}

(4) public static Octopus mutantOctopus() {
    Octopus newMutant = new Octopus(getRandomLegs());
}

(5) public static int getRandomLegs() {
    java.util.Random generator = new java.util.Random();
    return 1 + generator.nextInt(MAX_LEGS);
}
Objects can be created inside methods and returned just like primitive types. For example, the mutantOctopus method (4) has return type Octopus. An Octopus object is created inside the mutantOctopus method and "newMutant" is the local name for this object. When control is returned to the main program, the Octopus object reference "octet" is set to point at the returned object.
Arrays of Objects

Person[] people = new Person[10];

for (int i = 0; i < people.length; i++) {
    String name = javax.swing.JOptionPane.showInputDialog(null, "Enter name "+i+" : ");
    int age =
        Integer.parseInt(javax.swing.JOptionPane.showInputDialog(null, "Enter age of "+i+" : "));

    // This is the step most often forgotten -
    // creating each individual object in the array.

    people[i] = new Person(name, age, gender);
}
Arrays of objects

IMPORTANT: Every time you create an array of objects you must use the new operator once for the array declaration and once for each object in the array.

E.g., if the array length is 10, this means the new operator would be used 11 times to create all the objects in the array.