

Object-Oriented programming

SIS #2 (Deadline – 7th week)

Problem #1

When electricity moves through a wire, it is subject to electrical friction or *resistance*. When a resistor with resistance R is connected across a potential difference V , Ohm's law asserts that it draws current $I = V / R$ and dissipates power V^2 / R . A network of resistors connected across a potential difference behaves as a single resistor, which we call the *equivalent resistance*.

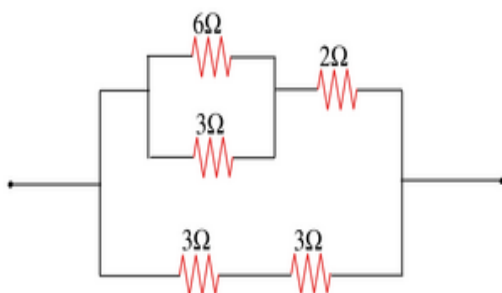
You should have: an abstract superclass **Circuit** that encapsulates the basic properties of a resistor network. For example, each network has a method **getResistance** that returns the equivalent resistance of the circuit.

```
public abstract class Circuit {
    public abstract double getResistance();
    public abstract double getPotentialDiff();
    public abstract void applyPotentialDiff(double V);

    public double getPower() {
        ///your code
    }

    public double getCurrent() {
        ///your code
    }
}
```

A series-parallel resistor network is either (i) a single resistor or (ii) built up by connecting two resistor networks in series or parallel. So create three **Circuit** class subclasses **Resistor**, **Series**, and **Parallel**. Your goal is to be able to compose circuits as in the following code fragment, which represents the circuit depicted below.



```
Circuit a = new Resistor(3.0);
Circuit b = new Resistor(3.0);
Circuit c = new Resistor(6.0);
Circuit d = new Resistor(3.0);
Circuit e = new Resistor(2.0);
Circuit f = new Series(a, b);
Circuit g = new Parallel(c, d);
Circuit h = new Series(g, e);
Circuit circuit = new Parallel(h, f);
double R = circuit.getResistance();
```

The class **Resistor** contains a constructor which sets the resistance in Ohms and an accessor method to return it. It also has a private field `pottentialDifference` and `get/set` mehods to it.

The class **Series** contains a constructor which takes two resistor circuit objects as inputs and represents a circuit with the two components in series.

The class **Parallel** is almost identical to **Series** except that it uses the reciprocal rule instead of the additive rule to compute the equivalent resistance.

The potential difference across each section depends on whether the circuit is series or parallel. For a parallel circuit, the potential difference across each branch is equal to the potential difference across the whole parallel circuit. For a series circuit, first find the current (I) from Ohm's law $I = V/R$, where V is the potential difference across the series circuit and R is its total resistance. The potential difference across each resistor is equal to the current times the resistance of that resistor.

Problem #2

```
public class Account
{
    private double balance; //The current balance
    private int accNumber; //The account number

    public Account(int a)
    {
        balance=0.0;
        accNumber=a;
    }

    public void deposit(double sum) { , , , }

    public void withdraw(double sum) { , , , }

    public double getBalance() { , , , }

    public double getAccountNumber() { , , , }

    public void transfer(double amount, Account other){}

    public String toString() {
        ""
    }

    public final void print()
    {
        //Don't override this, override the toString method
        System.out.println( toString() );
    }
}
```

Write the **Account** class and using it as a base class, write two derived classes called **SavingsAccount** and **CheckingAccount**.

A **SavingsAccount** object, in addition to the attributes of an **Account** object, should have an interest rate variable and a method which adds interest to the account. A **CheckingAccount** object (here there is a charge for each transaction), in addition to the attributes of an **Account** object, should have a counter variable, that will store the number of transactions done by user, and variable **FREE_TRANSACTIONS** – number of free transactions. Here you also will have a method **deductFee()**, that withdraws money for made transactions from account (suppose there is \$0.02 for each transaction - withdraw or deposit). Ensure that you have overridden methods of the **Account** class as necessary in both derived classes.

- Now create a **Bank** class, an object of which contains a **Vector** of **Account** objects. Accounts in the **Vector** could be instances of the **Account** class, the **SavingsAccount** class, or the **CheckingAccount** class. Create some test accounts (some of each type).

- Write an `update` method in the `Bank` class. It iterates through each account and deposits/withdraws money from accounts. After that Savings accounts get interest added (via the method you already wrote); CheckingAccounts get fees deducted .
- The `Bank` class requires methods for opening and closing accounts.

Problem #3

Create abstract class **MyCollection** with all the methods below. Then, implement **MyVector** (implements MyCollection interface) class to represent a growable array of objects. Like an array, it contains components that can be accessed using an integer index. Your class will store integers.

Constructors:

MyVector() – constructs an empty vector.

MyVector(int [] a) – constructs a vector containing all integers from array a.

Methods:

add(int element)

appends the specified element to the end of this Vector.

add(int index, int element)

inserts the specified element at the specified position in this Vector. Do not forget to throw Exception if index is more than size.

clear()

removes all of the elements from this Vector.

contains(int o)

returns true if this vector contains the specified element.

get(int index)

returns the element at the specified position in this Vector.

indexOf(int o)

returns the index of the first occurrence of the specified element in this vector, or -1 if this vector does not contain the element.

insertElementAt(int element, int index)

inserts the specified integer vector at the specified index.

isEmpty()

tests if this vector has no components.

removeAt(int index)

removes the element at the specified position in this Vector.

remove(int element)

removes the first occurrence of the element in this Vector.

removeAll(int element)

removes all occurrences of element in this Vector.

reverse()

reverses the elements in the array

set(int index, int element)

replaces the element at the specified position in this Vector with the specified element.

size()

returns the number of components in this vector.

sort()

Performs sorting of the elements in the array. Do not use sort() method of java API.

toArray()

returns an array containing all of the elements in this Vector in the correct order.

toString() returns a string representation of this Vector

GOOD LUCK ☺

By Pakita Shamo, CE, FIT