KAZAKH BRITISH TECHNICAL UNIVERSITY

Object-Oriented programming and design
SIS #2 *(5 pts, deadline – 12th week)*

No late submissions will be accepted!

SIS requires much more effort and time, rather than lab. Plan your time carefully.

```
If (you can't recreate "your" code) point = 0;
```

Problem #1 (2 points)

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*.

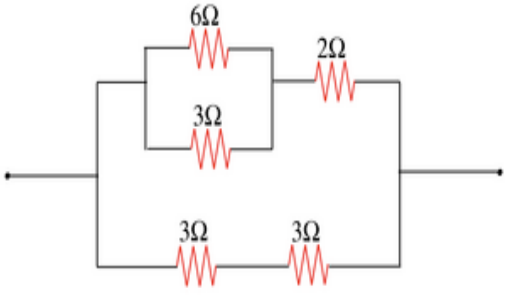
You should have: an abstract superclass **`Circuit`** that encapsulates the basic properties of a resistor network. For example, each network has a method **`getResistance`** that returns the equivalent resistance of the circuit.

```
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 (1.5 points)

```
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 (0,5 points)

Write a clone() method for the Employee and Manager classes you created in previous labs. In addition, provide 2 comparators – one to sort by name and the other to sort by hire date.

Check that your implementations are working fine.

Problem #4 (1 point)

Write a class **Chocolate** with fields **weight** and **name** (e.g., Twix) and method **toString**(). Implement **Comparable** interface (chocolates must be compared by **weight**).

Implement **Comparable** interface for a **Time** class you created in Lab2. Then import this class to your current project.

Next, implement a **Sort** class that will be able to sort anything that can be compared to each other (e.g. anything that is **Comparable**). You have to have a **swap** method that is commonly used in sorting algorithms (this ensures that you will not duplicate this code each time you need to swap elements) and 2 methods for sorting. USE ANY SORTING ALGORITHMS YOU LIKE. Bubble Sort and Merge Sort are given just for example.

```
public class Sort {
     static <E> void swap(E [] array, int i, int j) {
        ///..
    }
//this means that E is an arbitrary data type

     static <E extends Comparable<E>> void bubbleSort(E []
array) {
       ///…
/////this means that E must be a type whose instances can be
compared to each other (e.g. they have a compareTo method).


}
     static <E extends Comparable<E>> void mergeSort(E []
array) {

}

}
```

Finally, write a Test class and check that your Sort class is capable of sorting chocolates, times and employees (from prev. lab).

GOOD LUCK ☺

By Pakita Shamoi