Problem 1

Create three classes

- Product with fields name (String) and quantity (of type int);
- Box with fields id (String) and prods – array of Products;
- Storage with only one field boxes – array of Boxes.

All fields must be private. Provide necessary constructors and getters.
In class Storage add a method totQuant which counts (and returns as an int) the total quantity of product the name of which (String) is passed as an argument to the method. In a separate class define function main

download *ObjsII2Sto.java*

```
public static void main (String[] args) {
    Box box1 = new Box("Box1",
        new Product[]{
            new Product("Carrot",15),
            new Product("Apples",20)
        });
    Box box2 = new Box("Box2",
        new Product[]{
            new Product("Potato",10),
            new Product("Carrot",12)
        });
    Storage sto = new Storage(
            new Box[]{box1,box2});
    System.out.println("Tot. quantity of product: " +
                                    sto.totQuant("Carrot"));
}
```

which, for this example, should print

```
Tot. quantity of product: 27
```

Problem 2

Create two classes

- Person representing a person with fields name of type String and cars of type Car representing cars which he/she possesses; this can be null if he/she doesn't have any cars;
- Car representing a car with fields make (String), price (int) and next of type Car which is the reference to the next car possessed by a person (possibly null, then *this* car is the last).

All fields in both classes should be private!

The class Car contains the following member functions (constructors and methods):

```
public Car(String m, int p, Car n) { ... }
public Car(String make, int price) { ... }
public String getMake()    { ... }
public    int getPrice()   { ... }
public    Car getNext()    { ... }
public void   showCars()   { ... }
public void showCarsRev()  { ... }
@Override
public String toString()   { ... }
```

where

1. the first two member functions are constructors: one taking values of all fields, and one taking a make and a price and setting next to null;
2. the next three methods are just accessors which return values of the corresponding fields;
3. showCars prints (using System.out.print – see below) information on all cars: *this* car, then the one referenced to by its field next, then next of this next e.t.c, until null is encountered. Information on consecutive cars should be printed in one line;
4. showCarsRev prints (using System.out.print – see below) information on all cars, as does showCars, but in the reverse order: it has to be recursive and must not use any loops, auxiliary arrays or additional fields;
5. toString overrides toString from class Object and returns a string with make and price of *this* car. In this way a reference to Car may be used as an argument to System.out.print yielding a sensible string.

The class Person contains the following member functions (constructors and methods):

```
public Person(String name) { ... }
public Person buys(String make, int price) { ... }
public String    getName() { ... }
public void     showCars() { ... }
public void  showCarsRev() { ... }
public int getTotalPrice() { ... }
public boolean hasCar(String make) { ... }
public Car mostExpensive() { ... }
```

where

1. the first member is the constructor taking name only; field cars will be initialized with null);

2. **buys** takes a **String** and an **int**, creates a **Car** with the given make and price and sets it as the first car possessed by *this* person; the car which was the first (possibly **null**) becomes the second, i.e., it is referenced by the field next of the newly created car. The method returns the reference to *this* person (the one the method has been invoked on);
3. **getName** is a simple getter method;
4. **showCars** prints all cars owned by *this* person using the corresponding method in class **Car**;
5. **showCarsRev** does the same thing, but utilizing the corresponding recursive method in **Car**;
6. **getTotalPrice** returns the total price of all cars owned by *this* person;
7. **hasCar** takes a **String** and returns a **boolean** stating if *this* person owns a car of the given make (use **equalsIgnoreCase** from **String** to make string comparison case insensitive);
8. **mostExpensive** returns the most expensive car owned by *this* person (or **null** if *this* person doesn't have a car).

The following **main** function (in another class)

download *PersonCars.java*

```
public static void main (String[] args) {
    Person john = new Person("John");
    john.buys("Ford", 20000)
        .buys("Opel", 16000)
        .buys("Fiat", 12000)
        .showCars();
    System.out.println();

    john.showCarsRev();
    System.out.println();

    System.out.println("Total price of " +
            john.getName() + "'s cars: " +
            john.getTotalPrice());
    System.out.println("Does " + john.getName() +
            " have a ford? " + john.hasCar("ford"));
    System.out.println("Does " + john.getName() +
            " have a bmw?  " + john.hasCar("bmw"));
    System.out.println(john.getName() + "'s most " +
            "expensive car is " + john.mostExpensive());
}
```

should print

```
Fiat(12000) Opel(16000) Ford(20000)
Ford(20000) Opel(16000) Fiat(12000)
Total price of John's cars: 48000
Does John have a ford? true
Does John have a bmw?  false
John’s most expensive car is Ford(20000)
```

Problem 3

Create a class RPNStack which represents a stack of objects of type Node. Class RPNStack contains only one private field top of type Node. Objects of type Node represent data that are pushed on the stack: each object contains in its field val a double and in field next a reference to the next node (as in a singly-linked list — top plays here the rôle of the „head”). Class RPNStack offers three methods:

- method `public void push(double d)` pushing new object of type Node on top of the stack (i.e., it becomes the new top);
- method `public double pop()` removing the top node (so the next node becomes the new top) and returning val from the removed node;
- method `public boolean empty()` returning true if and only if the stack is empty (top is null); otherwise false is returned.

Note that stack is a singly-linked list where adding and removing elements is always performed at the beginning.

The main program reads a file with data representing arithmetic expressions in the Reverse Polish Notation (RPN), for example:

```
2 7 5 + * 20 - 1 4 / /
```

After reading each line, it is split (using spaces as separators) and for each token:

- if it is string `"+"`, we pop two numbers from the stack, add them and push the result on the stack;
- if it is string `"*"`, we do the same but myltiplying the numbers instead of adding them;
- if it is string `"-"`, we pop two elements, subtract the one popped as the first from the one popped later and push the result on the stack;
- if it is string `"/"`, we do the same but dividing the numbers instead of subtracting them;
- if it is not `"+"`, `"*"`, `"-"` or `"/"`, we interpret it as a number of type double and we push it onto the stack.

After all tokens from the line have been processed, we pop the remaining number off the stack, which should be the value of the whole expression. We then print the line and the result. We also check if the stack is now empty; if not, we inform the user about this fact, we clear the stack and continue with the next line of the input file.