Exercises for Automous Practice

Exercise E1

In the construction of residential buildings, the various professionals need to manage information on each housing unit (house or apartment) and extract various properties. Some classes are provided to represent points in Cartesian space (`Point`), housing units (`House`) and rooms within a house (`Room`), as well as a class for testing (`TestHouse`). It is assumed that each room will have a rectangular shape, aligned with the axes of a given coordinate system. Coordinates and distances are specified in meters.

You should incorporate the following new features in the program:

a. As you can see through the methods of class `House`, the rooms of a house are stored in an array. In particular, the method `addRoom(room)` in the class `House` adds a room to the array. Change this method so that it returns the index of the element of the array in which the room was stored. This index will serve as an identifier of the room.

b. Regarding the addition of doors to a house, complete the definition of class `Door` with the following methods (at least):

- `Door(RoomId1,RoomId2,Width,Height)` - constructor that receives the identifiers of the two rooms connected by this door, as well as the dimensions of the door.
- `area()` - a method that returns the area of the door.

c. The constructor of `House` creates an array for storing the doors. The capacity of this array is equal to the initial capacity of the array of rooms. The method `addDoor(Door)` in class `House` adds a new door, but fails when the doors array is full. Change this method so that the array capacity is extended with an extra `extensionSize` elements whenever that happens.

d. Create a method `roomClosestToRoomType(roomType)` in class `House` that, given a type of room, returns the identifier of the room closest to any room of that type. Consider the distance in a straight line between the centers of the rooms.

e. Create a method `maxDoorsInAnyRoom()` in class `House` that returns the maximum number of doors in any room of the house.

Exercise E2

[1] The files `JogaJogoDoGalo.java` and `jogos/JogoDoGalo.java` define a program and a module, respectively, implementing a “Tic-Tac-Toe” game. Intentionally, some errors were introduced.

[1]Problem from the AIP of 2009-2010.

a. Correct module `JogoDoGalo.java` in order to remove its syntactical (compilation) errors.

   - For compiling it use the following command: `javac JogaJogoDoGalo.java`

b. The main program `JogaJogoDoGalo.java` contains a semantic error. Find and correct it.

   - You may execute it with: `java -ea JogaJogoDoGalo`
   - You may execute a correct version with: `java -ea -jar JogaJogoDoGalo.jar`

```
    1   2   3
 1  X |   | O
   ---+---+---
 2    | X | O
   ---+---+---
 3    |   | X



Jogador X ganhou!
```

Note: The sentence in the output means "Player X won!"

c. Make the program robust regarding the module exploitation (exceptions are not needed).

d. Change the program `JogaJogoDoGalo.java` in order to deploy championships up to 10 games, finishing when one of the players scores 3 victories. At the end of each game the score of each player should be displayed.

Exercise E3

Available only in Portuguese.

Exercise E4

Implement a recursive function `factors` receiving an integer number as argument and returning a `String` with the product of its factors. For example, the following program invocation

```
java -ea Factors 0 1 10 4 10002
```

should output:

```
0 = 0
1 = 1
10 = 2 * 5
4 = 2 * 2
10002 = 2 * 3 * 1667
```

Exercise E5

Available only in Portuguese.

Exercise E6

Write a program that receives an integer number as argument and writes all its divisors other than itself and the unity and, recursively, does the same for all those divisors.

Here are some use cases of the intended program:

```
java -ea AllDivisors 12  java -ea AllDivisors 23  java -ea AllDivisors 81  java -ea AllDivisors 32
12                       23                       81                       32
   6                                                 27                       16
      3                                                 9                        8
      2                                                    3                        4
   4                                                    3                              2
      2                                              9                              2
   3                                                    3                        4
   2                                                 3                              2
                                                                                 2
                                                                              8
                                                                                 4
                                                                                    2
                                                                                 2
                                                                              4
                                                                                 2
                                                                              2
```

Exercise E7

Write a program that receives a rational number belonging to $]\,0, 1\,[$, expressed as a fraction (n/d), and write that fraction as the sum of unitary fractions (with unity as numerator) with different denominators[2]. The program to develop must use a recursive algorithm.

Here are some use cases of the program:

`java -ea UnitaryFractionSum 3 4`	`3/4 = 1/2 + 1/4`
`java -ea UnitaryFractionSum 3 7`	`3/7 = 1/3 + 1/11 + 1/231`
`java -ea UnitaryFractionSum 1 8`	`1/8 = 1/8`
`java -ea UnitaryFractionSum 2 20`	`2/20 = 1/10`

To solve the problem consider the following strategy (called "greedy" and proposed by Fibonacci in the 13th Century):

a. Attempt to subtract from the fraction the highest possible unitary fraction. To find out that unitary fraction $(1/d)$, with the lowest possible d, consider the following expression:

$$\frac{\text{num}}{\text{den}} - \frac{1}{d} \geq 0 \quad \Leftrightarrow \quad d \geq \frac{\text{den}}{\text{num}} \quad \Rightarrow \quad d = \left\lceil \frac{\text{den}}{\text{num}} \right\rceil$$

b. The fraction will be the sum of the unitary fraction $1/d$ added to the unitary fraction obtained from the fraction resulting from the difference (for which you should apply the same algorithm);

[2]Fibonacci has demonstrated that any rational number can be expressed by a finite sum of unitary fractions with different denominators.

c. The process ends when the numerator is divisor of the denominator (an evidence that it is already an unitary fraction).

Exercise E8

Available only in Portuguese.

Exercise E9

Program `ArraySorting.java` contains the implementation of several sorting algorithms as well as a `main()` function that applies these algorithms to randomly generated arrays of numbers (by the function `randomArray()`. The implementations of the sorting algorithms include some assertions to check pre- and post-conditions. If the program is run with assertions enabled (option `-ea`), normal termination of the program indicates that the sorting algorithms correctly sorted the randomly generated arrays. Revise the whole implementation and correct any errors that you find.

Exercise E10

Create a `LeakyQueue` module, based on the queue data structure, in order to enable program `ProgX` to work properly[3].

A N-long leaky queue is a queue-based data structure that keeps only the last N inserted values. When the queue is full (with N elements), inserting a new one implies dropping the first one from the queue.

Here are some use cases (with $N = 3$) and expected results:

```
java -ea ProgX 1 2 3 4 5 6
i = 0    1.0                (Min = 1.0)
i = 1    1.0  2.0           (Min = 1.0)
i = 2    1.0  2.0  3.0      (Min = 1.0)
i = 3    2.0  3.0  4.0      (Min = 2.0)
i = 4    3.0  4.0  5.0      (Min = 3.0)
i = 5    4.0  5.0  6.0      (Min = 4.0)
```

```
java -ea ProgX 9 8 7 6 5 4 3 2 1
i = 0    9.0                (Min = 9.0)
i = 1    9.0  8.0           (Min = 8.0)
i = 2    9.0  8.0  7.0      (Min = 7.0)
i = 3    8.0  7.0  6.0      (Min = 6.0)
i = 4    7.0  6.0  5.0      (Min = 5.0)
i = 5    6.0  5.0  4.0      (Min = 4.0)
i = 6    5.0  4.0  3.0      (Min = 3.0)
i = 7    4.0  3.0  2.0      (Min = 2.0)
i = 8    3.0  2.0  1.0      (Min = 1.0)
```

```
java -ea ProgX 1 3 - 5 7 - 9 11 -
i = 0    1.0                (Min = 1.0)
i = 1    1.0  3.0           (Min = 1.0)
i = 2    3.0                (Min = 3.0)
i = 3    3.0  5.0           (Min = 3.0)
i = 4    3.0  5.0  7.0      (Min = 3.0)
i = 5    5.0  7.0           (Min = 5.0)
i = 6    5.0  7.0  9.0      (Min = 5.0)
i = 7    7.0  9.0 11.0      (Min = 7.0)
i = 8    9.0 11.0           (Min = 9.0)
```

```
java -ea ProgX 2 - - 4 - 6 8
i = 0    2.0                (Min = 2.0)
i = 1
i = 2
i = 3    4.0                (Min = 4.0)
i = 4
i = 5    6.0                (Min = 6.0)
i = 6    6.0  8.0           (Min = 6.0)
```

[3] You cannot use the modules from package `exameP2` in this problem.

Exercise E11

The program `ProgX` verifies if an arithmetic expression (formed by algarisms, elementary operations and parenthesis) is syntactically correct. Write the module `PilhaX`, based on the stack data structure, in order to enable this program to work properly[4].

Here are some use cases and expected results:

```
java -ea ProgX "2+2"
  PUSH: D
REDUCE: e
  PUSH: e+
  PUSH: e+D
REDUCE: e+e
REDUCE: e
Correct expression!
```

```
java -ea ProgX "2+(2-3)"
  PUSH: D
REDUCE: e
  PUSH: e+
  PUSH: e+(
  PUSH: e+(D
REDUCE: e+(e
  PUSH: e+(e-
  PUSH: e+(e-D
REDUCE: e+(e-e
REDUCE: e+(e
  PUSH: e+(e)
REDUCE: e+e
REDUCE: e
Correct expression!
```

```
java -ea ProgX "3*(4/(3))"
  PUSH: D
REDUCE: e
  PUSH: e*
  PUSH: e*(
  PUSH: e*(D
REDUCE: e*(e
  PUSH: e*(e/
  PUSH: e*(e/(
  PUSH: e*(e/(D
REDUCE: e*(e/(e
  PUSH: e*(e/(e)
REDUCE: e*(e/e
REDUCE: e*(e
  PUSH: e*(e)
REDUCE: e*e
REDUCE: e
Correct expression!
```

```
java -ea ProgX "2+"
  PUSH: D
REDUCE: e
  PUSH: e+
Bad expression!
```

```
java -ea ProgX "(3*(2+4)+5))"
  PUSH: (
  PUSH: (D
REDUCE: (e
  PUSH: (e*
  PUSH: (e*(
  PUSH: (e*(D
REDUCE: (e*(e
  PUSH: (e*(e+
  PUSH: (e*(e+D
REDUCE: (e*(e+e
REDUCE: (e*(e
  PUSH: (e*(e)
REDUCE: (e*e
REDUCE: (e
  PUSH: (e+
  PUSH: (e+D
REDUCE: (e+e
REDUCE: (e
  PUSH: (e)
REDUCE: e
  PUSH: e)
Bad expression!
```

```
java -ea ProgX "2+4*(4++5)"
  PUSH: D
REDUCE: e
  PUSH: e+
  PUSH: e+D
REDUCE: e+e
REDUCE: e
  PUSH: e*
  PUSH: e*(
  PUSH: e*(D
REDUCE: e*(e
  PUSH: e*(e+
  PUSH: e*(e++
  PUSH: e*(e++D
REDUCE: e*(e++e
  PUSH: e*(e++e)
Bad expression!
```

Exercise E12

Write a program (`JustifiedText.java`) for aligning a text to both margins, simultaneously (justified alignment). The program receives as parameters the line length and the name of the file with the text to align, which should be written in the standard output.

To tackle this problem you should use at least one suitable data structure from package `exameP2.jar`.

For instance, given the following text on the file `texto.txt`:

[4]You cannot use the modules from package `exameP2` in this problem.

```
If one cannot enjoy reading a book over and over again, there is no use
in reading it at all.
Perfect day for scrubbing the floor and other exciting things.

You
are      standing   on my
toes.  You have taken yourself too seriously.
```

these are two use cases of the program:

```
java -ea JustifiedText 40 texto.txt
```

```
If one cannot enjoy reading a book  over
and over  again,  there  is  no  use  in
reading  it  at  all.  Perfect  day  for
scrubbing the floor  and other  exciting
things.

You are standing  on my  toes. You  have
taken yourself too seriously.
```

```
java -ea JustifiedText 30 texto.txt
```

```
If one cannot enjoy reading  a
book  over  and  over   again,
there is no use in reading  it
at  all.   Perfect   day   for
scrubbing the floor and  other
exciting things.

You are standing  on my  toes.
You have  taken  yourself  too
seriously.
```

Details to take into consideration:

- Each output line must contain the maximum possible words without trespassing the maximum line length. A “word” should be considered any sequence of characters delimited by white spaces (space characters, tab characters, etc.).
- Words cannot be joined or merged (null spacing) nor split across lines.
- The lengths of spaces between words of the same line should differ at most by one character.
- The last line of each paragraph must be left aligned (with a single space between words). Consider that a paragraph terminates with an empty line or with the end of the file.

Exercise E13

Available only in Portuguese.

Exercise E14

`MainTrain` is a program that demonstrates the usage of a data structure for managing the loading and unloading of wagons on a freighter train. Create the `Train` module in a way that allows this program to compile and work properly[5].

An object belonging to the Train class represents a train composed by different freighter wagons in bulk. When a train is created, it is necessary to specify the capacity of each wagon, and the total capacity supported by the train, both in tons. You can add a wagon with a certain amount of cargo to a train (addWagon) or you can remove a wagon from its tail (removeWagon), according to a LIFO policy (the Last one In is the First one Out). Naturally, the load of a wagon cannot surpass its capacity and you can only add a wagon

[5]You cannot use the `exameP2` modules in this problem.

which doesn't overcome the maximum total cargo of the train. It is also possible to request the unload (unload) of a certain amount. This can be accomplished through completely unloading and removing zero or more wagons from the tail, as well as partially unloading another wagon to complete the requested amount. At any time it is possible to obtain a list with the cargo list of the train wagons (list); know the number of wagons (size) or the total transported cargo (totalCargo).

Usage examples and expected results:

```
java -ea MainTrain 10 100 1 2 3 R R 4.5 0.1

(Wagons capacity: 10.0 ton.)
(Train capacity: 100.0 ton.)
args[2]="1": Joins wagon with 1.0 ton
  (1 wagons, 1.0 ton): LocO_[1.0]
args[3]="2": Joins wagon with 2.0 ton
  (2 wagons, 3.0 ton): LocO_[1.0]_[2.0]
args[4]="3": Joins wagon with 3.0 ton
  (3 wagons, 6.0 ton): LocO_[1.0]_[2.0]_[3.0]
args[5]="R": Removes wagon with 3.0 ton
  (2 wagons, 3.0 ton): LocO_[1.0]_[2.0]
args[6]="R": Removes wagon with 2.0 ton
  (1 wagons, 1.0 ton): LocO_[1.0]
args[7]="4.5": Joins wagon with 4.5 ton
  (2 wagons, 5.5 ton): LocO_[1.0]_[4.5]
args[8]="0.1": Joins wagon with 0.1 ton
  (3 wagons, 5.6 ton): LocO_[1.0]_[4.5]_[0.1]
```

```
java -ea MainTrain 10 100 4 2 5 7 -2 -11 -1

(Wagons capacity: 10.0 ton.)
(Train capacity: 100.0 ton.)
args[2]="4": Join wagon with 4.0 ton
  (1 wagons, 4.0 ton): LocO_[4.0]
args[3]="2": Join wagon with 2.0 ton
  (2 wagons, 6.0 ton): LocO_[4.0]_[2.0]
args[4]="5": Join wagon with 5.0 ton
  (3 wagons, 11.0 ton): LocO_[4.0]_[2.0]_[5.0]
args[5]="7": Join wagon with 7.0 ton
  (4 wagons, 18.0 ton): LocO_[4.0]_[2.0]_[5.0]_[7.0]
args[6]="-2": Unloads 2.0 ton and removes 0 empty wagons.
  (4 wagons, 16.0 ton): LocO_[4.0]_[2.0]_[5.0]_[5.0]
args[7]="-11": Unloads 11.0 ton and removes 2 empty wagons.
  (2 wagons, 5.0 ton): LocO_[4.0]_[1.0]
args[8]="-1": Unloads 1.0 ton and removes 1 empty wagons.
  (1 wagons, 4.0 ton): LocO_[4.0]
```

```
java -ea MainTrain 10 20 2 10 11

(Wagons capacity: 10.0 ton.)
(Train capacity: 20.0 ton.)
args[2]="2":  Join wagon with 2.0 ton
  (1 wagons, 2.0 ton): LocO_[2.0]
args[3]="10": Join wagon with 10.0 ton
  (2 wagons, 12.0 ton): LocO_[2.0]_[10.0]
args[4]="11": ERROR: Wagon overload!
```

```
java -ea MainTrain 10 20 5 7 9

(Wagons capacity: 10.0 ton.)
(Train capacity: 20.0 ton.)
args[2]="5": Join wagon with 5.0 ton
  (1 wagons, 5.0 ton): LocO_[5.0]
args[3]="7": Join wagon with 7.0 ton
  (2 wagons, 12.0 ton): LocO_[5.0]_[7.0]
args[4]="9": ERROR: Train overload!
```