Lab01 - Kalkulačka

Template pro domácí úkol ke stažení zde.

Vytvořte jednoduchou kalkulačku. Program se postupně uživatele dotáže na operaci tak, že 1 odpovídá součtu, 2 rozdílu, 3 součinu a 4 podílu. Po volbě operace se program dotáže na dva operandy, které korektně označí podle zvolené operace - tj. pro součet se dotáže na sčítance, pro rozdíl na menšence a menšitele, pro součin na činitele a pro podíl na dělence a dělitele. Posledním vstupem programu je počet desetinných míst, která se použijí na tisk výsledku.

Volba operace očekává celé číslo (1, 2, 3, 4). Operandy jsou reálná čísla. Počet desetinných míst je celé ~~kladné~~nezáporné číslo.

Program musí ošetřit případné dělení nulou, či volbu nepodporované operace.

Formát výstupu je: "operand1 operátor operand2 = výsledek" Tak, že operandy i výsledek jsou formátovány na zadaný počet desetinných míst, a výstup je **ukončen znakem nového řádku**.

Odevzdávejte zazipované soubory `Lab01.java` a `Start.java`

Program implementujte jako metodu pojmenovanou `homework`() a zajistěte, že bude tato metoda volána po spuštení programu. Dodržte prosím doporučené textové výpisy dle ukázek níže. Program bude automaticky zkontrolován v odevzdávacím systému.
Za textem výběru operace není mezera, naopak za textem zadávání operandů mezera je. Např.:

```
Vyber operaci (1-soucet, 2-rozdil, 3-soucin, 4-podil):

Zadej scitanec:_

Zadej scitanec:_

Zadej pocet desetinnych mist:_
```

Zde znak podtržítka "_" reprezentuje mezeru.

Příklad komunikace programu

```
Vyber operaci (1-soucet, 2-rozdil, 3-soucin, 4-podil):

10

Chybna volba!

Vyber operaci (1-soucet, 2-rozdil, 3-soucin, 4-podil):

1

Zadej scitanec:

10.1234

Zadej scitanec:
```

```
42
Zadej pocet desetinnych mist:
2
10.12 + 42.00 = 52.12
Vyber operaci (1-soucet, 2-rozdil, 3-soucin, 4-podil):
2
Zadej mensenec:
1.23456
Zadej mensitel:
12.345678
Zadej pocet desetinnych mist:
3
1.235 - 12.346 = -11.111
Vyber operaci (1-soucet, 2-rozdil, 3-soucin, 4-podil):
3
Zadej cinitel:
12.345678
Zadej cinitel:
1.23456
Zadej pocet desetinnych mist:
3
12.346 * 1.235 = 15.241
Vyber operaci (1-soucet, 2-rozdil, 3-soucin, 4-podil):
4
Zadej delenec:
10
Zadej delitel:
0
Pokus o deleni nulou!
Vyber operaci (1-soucet, 2-rozdil, 3-soucin, 4-podil):
4
Zadej delenec:
```

```
10
Zadej delitel:
2
Zadej pocet desetinnych mist:
5
10.00000 / 2.00000 = 5.00000
Vyber operaci (1-soucet, 2-rozdil, 3-soucin, 4-podil):
1
Zadej scitanec:
10.234
Zadej scitanec:
1
Zadej pocet desetinnych mist:
0
10 + 1 = 11
Vyber operaci (1-soucet, 2-rozdil, 3-soucin, 4-podil):
1
Zadej scitanec:
10
Zadej scitanec:
20
Zadej pocet desetinnych mist:
-1
Chyba - musi byt zadane kladne cislo!
```

Lab02 - Výpočet statistiky číselné posloupnosti

Template domácího úkolu ke stažení zde. Ke kontrole vstupů použijte funkce z třídy TextIO.

21.3.2018: Zamyslete se nad tím, že délka posloupnosti není omezená a není tedy vhodné vstupní data ukládat do paměti.

Úkoly:

Napište program, který vypočte průměrnou hodnotu a směrodatnou odchylku z posloupnosti čísel zadaných na standardní vstup. Při implementaci se inspirujte příkladem přesměrování standardního vstupu ze souboru. Testovací soubor obsahuje na každém řádku jedno číslo (nebo nečíselnou hodnotu, viz dále).

- Rozšiřte tento program pro výpočet statistiky (průměr a odchylka) z každých 10 vstupních čísel. Tyto dvě hodnoty vypište na jeden řádek standardního výstupu na tři desetinná místa a čísla oddělte mezerou, tj. formátování "`%.3f %.3f`".
- Na začátek řádku vypište počet hodnot, ze kterých jsou průměr a odchylka vypočteny na dvě místa, tj. formátování "`%2d`".
- Při detekci konce vstupního souboru vypište dílčí výsledek z příslušného počtu hodnot, ale pouze pokud je počet hodnot použitých k výpočtu vyšší než 1.
- Rozlište mezi výstupem na standardní výstup a na standardní chybový výstup. Program implementujte jako část třídy Lab02 v metodě pojmenované `homework`.
- Detekci konce vstupu indikujte výpisem 'End of input detected!' na standardní chybový výstup.

Odevzdávejte soubory `Lab02.java`, `Start.java` a `TextIO.java`.

V případě, že na řádku není detekováno číslo, program řádek přeskočí a pokračuje ve čtení vstupu do konce souboru. Tuto situace indikujte výpisem na standardní chybový výstup s uvedením čísla řádku, na kterém k této události došlo.

Pro výpočet směrodatné odchylky lze použít průměrnou hodnotu a průměr z mocniny sledované veličiny viz http://en.wikipedia.org/wiki/Standard_deviation.

Ukázka výstupu standardního a chybového výstupu po zpracování souboru test_input.txt. Dbejte prosím na formátování výstupu: každý - i ten poslední - řádek by měl být zarovnaný správně a **ukončen znakem nového řádku** (viz. ukázka).

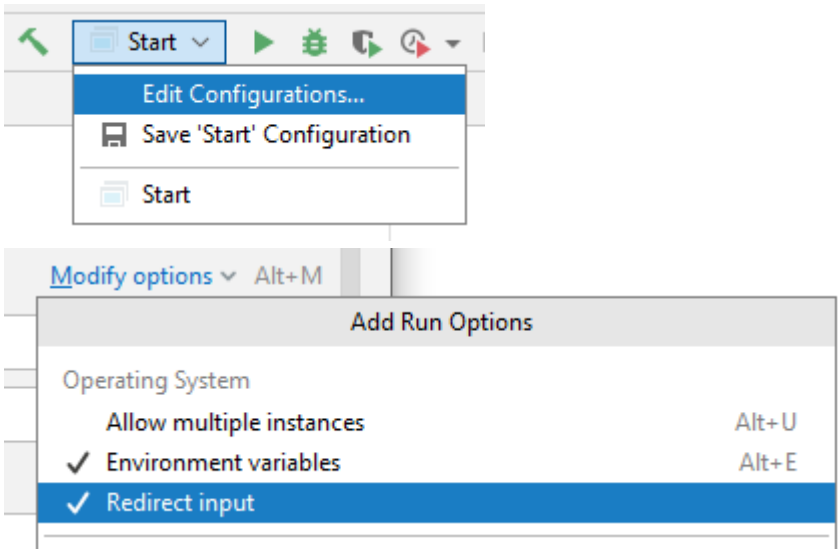
```
java -jar dist/lab02.jar <long_input.txt

A number has not been parsed from line 6

10 53,500 23,851

10 56,000 26,446

10 34,400 19,184

10 53,900 26,121
```

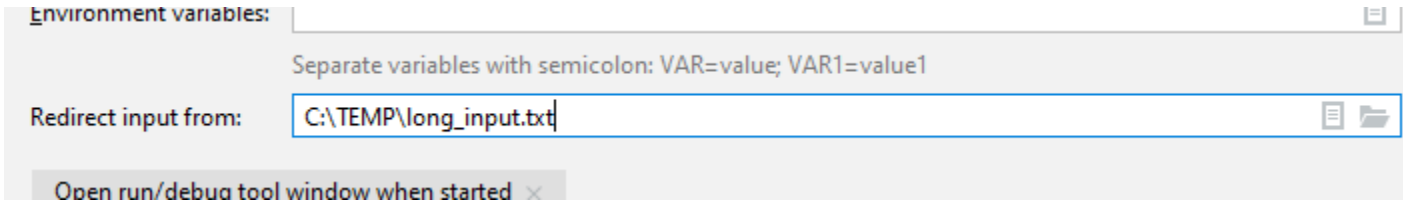
```
10 63,400 32,116

10 58,200 28,071

10 65,800 32,508

10 44,200 24,891

A number has not been parsed from line 86

10 53,400 29,063

End of input detected!

 8 39,875 23,235
```

Tato ukázka využívá přesměrování textového suouboru do `System.in`, t.j. Váš program bude číst vstupy způsobenm, jako kdyby je zadával uživatel z klávesnice. Zajistí to znak přesměrování: `<`.

Alternativní způsoby přesměrování obsahu souboru na standardní vstup:

IntelliJ Idea





Netbeans

Pro účely testování přidejte do metody `main` kód:

```
public static void main(String[] args) throws FileNotFoundException {

  FileInputStream is = new FileInputStream(new File("C:\\TEMP\\
long_input.txt"));

  System.setIn(is);
```

```
  //...
```

Před odevzdáním řešení do BRUTE **tento kód pochopitelně odstraňte**!

Rada - spusteni jar souboru bez manifestu z prikazove radky:

```
java -cp ./soubor.jar cz.cvut.fel.pjv.Main
```

Lab03 - Kruhová fronta

Template domácího úkolu je ZDE

Napište program, který bude reprezentovat cyklickou frontu uchovávající hodnoty typu `String`. Kapacita fronty bude parametrem konstruktoru. Pokud bude použit bezparametrický konstruktor, vytvořte frontu o **konstantní velikosti 5**. Dále naimplementujte metody do připravené třídy `CircularArrayQueue`. Co mají jednotlivé metody dělat dozvíte v dokumentaci v interface `Queue.java`. Všimněte si, že dokumentace některých metod je velmi podobná (někdy i stejná) dokumentaci metod ve třídě Queue ve standardním java frameworku. Toto není náhoda, autoři úkolu se tímto javadocem inspirovali 🙂.

Odevzdávejte pouze soubor `CircularArrayQueue.java`.

Frontu implementujte **pomocí pole statické délky** se dvěma indexy ukazujícími na začátek a konec pole. Bližší informace naleznete na wikipedii. **Jiná řešení nemusí být přijata**, v takových případech Vám cvičící po kontrole odevzdaného kódu sníží bodový zisk v BRUTE na 0.

Součástí hodnocení může být i namátkové manuální subjektivní hodnocení kvality kódu cvičícím. Dejte si tedy záležet.

Ve třídě `Start` je připraven kód, na kterém můžete funkčnost implementace kruhové fronty vyzkoušet

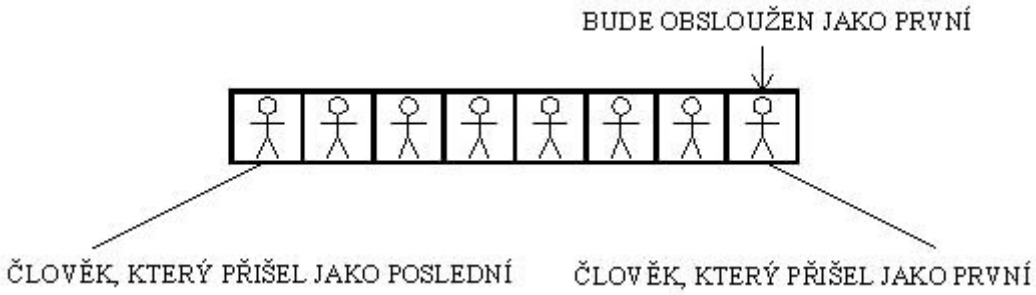
```
--- Příklad očekávaného výstupu programu

size: 6

value dequeued from CircularArrayQueue: Starkiller

printing all elements:

C-3PO

Jabba the Hutt

HK-47

Darth Nihilus

Count Dooku

size: 6
```

Fronta

Fronta je dynamická množina (datová struktura), u které jsou specificky definovány operace výběru a vložení prvku. Operace výběr z fronty vybere prvek, který jsme vložili do fronty jako první. Při vkládání prvků do fronty se vkládaná položka vloží na jeho konec. (anglicky enqueue a dequeue)

Tato struktura se také někdy označuje termínem FIFO (first-in first-out).

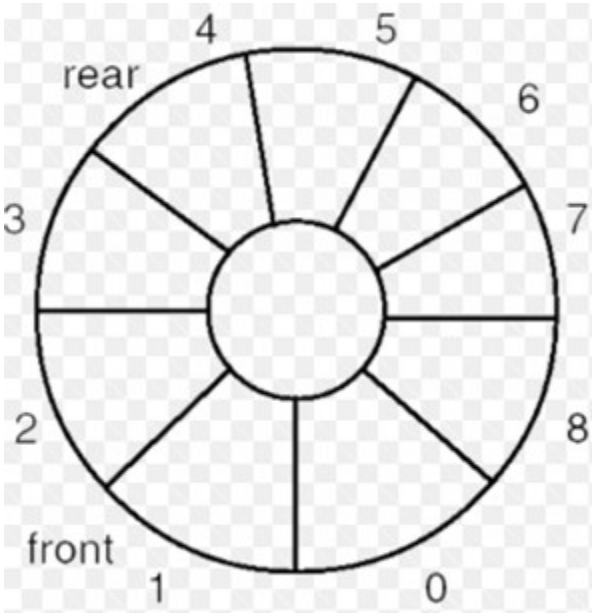
Fronta se dá implementovat polem a to buď polem statické délky s explicitním omezením na počet vložených prvků nebo polem dynamické délky. Alterantivně se dá také realizovat datovou strukturou nazývanou spojový seznam, se kterou se seznámíme na dalším cvičení.



Doplňující informace na wiki

Kruhová fronta

V případě omezené kapacity fronty, například realizované polem statické délky, můžeme využít takzvanou kruhovou frontu. V té se začátek fronty pohybuje po jednotlivých prvcích pole tak jak jsou postupně prvky vkládány a odebírány. Praktické použití takové fronty si můžeme představit v případech, kdy do fronty jsou dávány jednotlivé požadavky na obsloužení, které v průměru nepřicházejí častěji než je rychlost obsluhy. V případě, že obsloužení konkrétního požadavku trvá déle, jsou další požadavky uloženy ve frontě a po vyřízení náročnějšího požadavku jsou pak ostatní, méně náročné, požadavky obslouženy rychleji a fronta je rychle vyprázdněna. Analogickou situaci můžeme začít například v obchodě, ve kterém je kapacita fronty omezena velikostí obchodu.



Kruhová fronta může explicitně hlídat, zda-li je možné nový požadavek do fronty vložit a v případě zaplnění fronty je možné požadavek na vložení zamítnout. Na druhé straně můžeme také najít připady, kdy do kruhové fronty dáváme požadavky a pokud je plná, jsou ty nejstarší požadavky přepisovány. To může například nastat v případě zpracování senzorických dat, ve kterém předpokládáme, že pokud se data nestihla zpracovat, jsou pravděpodobně již zastaralá a dáváme přednost zpracování aktuálnějších informací.

Lab04 - Lámání hesel

Template domácího úkolu je ZDE

Povedl se Vám nečekaně cenný úlovek - získali jste do rukou zadání testů z PJV. Bohužel, tyto testy jsou uzavřeny v sejfu s elektronickým zámkem. K tomuto zámku dokážete připojit svůj počítač, pomocí kterého můžete zkoušet různá hesla. Jednou z metod prolamování hesel je intuitivní tzv. brute force attack, tedy útok hrubou silou. Při této metodě útočník zkouší postupně všechny možné kombinace znaků, dokud neuhodne heslo.

Domácí úkol sestává ze tříd `Test, Thief a BruteForceAttacker`. Vy budete pracovat pouze se třídou `BruteForceAttacker`. Zde doplňte kód do metody `public void breakPassword(int sizeOfPassword)`, který se pokusí prolomit heslo o délce `sizeOfPassword` (heslo je dlouhé přesně zadaný počet znaků, tedy ani kratší ani delší). K dispozici máte následující metody:

- `char[] getCharacters()` tato metoda vrací seznam znaků z jejichž podmnožiny je složeno heslo
- `boolean tryOpen(char[] password)` tato metoda zkusí otevřít sejf. Pokud se podaří vypíše hlášku a vrací `true`, jinak `false`. Jakmile je sejf otevřen, již není třeba se snažit dále. Navíc, pokud budete dále zkoušet jiná hesla, sejf se opět zamkne.

Téma cvičení - rekurze - vám napovídá, že tento úkol je **povinné** řešit pomocí rekurze 🙂. Nicméně, je dost možné, že metoda `void breakPassword(int sizeOfPassword)` nebude rekurzivní a bude volat jinou metodu, která již rekurzivní bude.

Váš algoritmus si můžete otestovat. Viz kód ve třídě `Test.java`. V tomto kódu se sejf nastaví na heslo `abcdaaaddb` a množina znaků na `{'a', 'b', 'c', 'd'}`. Podle tohoto vzoru si můžete vyzkoušet i jiná hesla.

Do Upload Systemu nahrávejte soubor `BruteForceAttacker.java` (zabalený v archivu), který obsahuje vaše řešení.

Lab05 - Binární strom

Implementujte dodané interface `Tree` a `Node` třídami `TreeImpl` a `NodeImpl`.
Třída `TreeImpl` musí obsahovat defaultní konstruktor (bez parametrů). Metody a proměnné pojmenovávejte anglicky. Nepoužívejte javovské kolekce; potřebujete pouze pole, které dostanete jako parametr `setTree`.

Implementované třídy `TreeImpl` a `NodeImpl` umístěte do stejného balíčku jako jsou dodané interfacy.

Tree reprezentuje binární strom, který ve všech uzlech obsahuje celočíselná data. Každý uzel stromu je reprezentován třídou implementující interface `Node`. Tree obsahuje následující metody:

- `void setTree(int[] values)`
  - nastaví strom, tak aby obsahoval hodnoty z pole values
  - pokud je délka pole lichá, kořen obsahuje prostřední číslo, jinak obsahuje první číslo za polovinou posloupnosti (na jejich hodnotách tedy nezáleží, pouze na jejich pozici v poli)
  - levá část podstromu pak obsahuje prvky pole před tím prostředním prvkem a pravé prvky za ním
  - obdobně to platí i pro podstromy
- `Node getRoot()`
  - vrátí kořen stromu
- `String toString()`
  - vrátí řetězcovou reprezentaci stromu vhodnou k výpisu v následujícím formátu
    - každá hodnota je na jednom řádku, předchází ji počet mezer odpovídající hloubce uzlu (0 pro kořen) a '- '
    - na prvním řádku je hodnota kořenu
    - hodnotu uzlu následuje výpis levého podstromu a pak pravého podstromu
    - každý řádek (vč. posledního) je ukončen novým řádkem ('\n')
  - Příklad pro strom vytvořený pro pole [1, 2, 3, 4, 5, 6, 7]:

```
- 4
 - 2
  - 1
  - 3
 - 6
  - 5
  - 7
```

Cílem není implementovat binární *vyhledávací* strom; na samotných hodnotách tedy **nezáleží**, pouze na jejich pořadí ve vstupním poli!

Ukázka výstupu metody `toString` pro stromy vytvořené z posloupností [1], [1, 2], … , [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

```
- 1


- 2
 - 1


- 2
 - 1
 - 3


- 3
 - 2
  - 1
 - 4


- 3
 - 2
  - 1
 - 5
  - 4


- 4
 - 2
  - 1
  - 3
 - 6
  - 5


- 4
 - 2
  - 1
```

```
  - 3
 - 6
  - 5
  - 7


- 5
 - 3
  - 2
   - 1
  - 4
 - 7
  - 6
  - 8


- 5
 - 3
  - 2
   - 1
  - 4
 - 8
  - 7
   - 6
  - 9


- 6
 - 3
  - 2
   - 1
  - 5
   - 4
 - 9
  - 8
```

```
  - 7

- 10
```

Odevzdávejte následující soubory: `NodeImpl.java`, `TreeImpl.java`