Laboration 3 TDA143

Syfte

Öva på att använda nya klasser från standardbiblioteken. Öva på textbehandling, arrayer och Listor. Öva på att implementera klasser och metoder utifrån informella beskrivningar, och på att konstruera metoder utifrån testfall som beskriver hur de ska fungera. Öva på att testa att klasser fungerar som de ska. Öva på att kasta och fånga undantag.

Redovisning

Varje deluppgift ska redovisas för kursassistent på labbpass, för varje redovisning får ni en fembokstavskod. Därefter ska ni skicka in alla .java filer ni har skrivit eller ändrat, och fembokstäverskoderna. Följ instruktionerna på inlämningssidan på kurshemsidan. Sammanlagt ska ni skicka in 4 koder för Laboration 3.

Detaljerade uppgifter och tips

Uppgiftstexterna i den här labben är mer kortfattade än i tidigare labbar. Att lösa dem kräver att ni själva listar ut hur ni kan bryta ner uppgiften i delar.

I filen Labb3Tips.pdf finns betydligt mer detaljerad uppdelning av varje uppgifter i deluppgifter. Har ni svårt att komma igång eller kör fast så läs filen.

Uppgifter

Uppgift 1: Stora tal

Datatypen int kan lagra heltal i intervallet [-2147483648, 2147483647] (dvs [-2^{31}, $2^{31}-1$]). Typen long har ett större men ändå ändligt intervall. Om man behöver göra exakta beräkningar med heltal av godtycklig storlek kan man använda klassen BigInteger som finns i Javas API: https://docs.oracle.com/javase/8/docs/api/java/math/BigInteger.html

a) Skriv ett Javaprogram (Calculator.java) som genom dialogrutor läser in enkla aritmetiska uttryck endera på formen tal * tal eller formen tal + tal. Talen kan vara godtyckligt stora. Ni kan anta att det alltid är mellanslag mellan talen och operatorn.

Resultatet av uttrycket ska sedan beräknas och skrivas ut i en dialogruta.

Tips: Scanner har en metod nextBigInteger för att läsa in stora heltal. Metoden next i Scanner läser in ett ord med vilka bokstäver/siffror/symboler som helst.

b) Innan redovisning: Se till att er kod är körbar, snyggt indenterad, har väl valda namn och kommentarer där det behövs.

Uppgift 2: Matcha parenteser.

I den här uppgiften ska ni skriva en metod för avgöra om en text har matchande parenteser. Texterna innefattar flera olika sorters parenteser. Alla tecken som inte är parenteser ska ignoreras. För att räknas som matchande måste följande gälla:

1. Varje startparentes motsvaras av en slutparentes av samma sort, och vice versa
2. Parentesuttryck får inte vara delvis överlappande. Varje parentesuttryck måste alltså vara helt inuti ett annat parentesuttryck, eller helt utanför. Det här är till exempel inte tillåtet: "{(})", men däremot är både "(){}" och "({})" tillåtna.

De parentessorter ni ska ha stöd för inledningsvis är: (), [], {} och <>, men det ska vara så enkelt som möjligt att lägga till nya parentespar, helst bara att lägga till två tecken till källkoden (ett för start och ett annat för slut).

a) Skapa en klass Parenthesis. Lägg till en metod:

```
public static boolean matches(String input)
```

Som svarar på om strängen input har matchande parenteser. Första steget är att få den att loopa igenom strängens tecken från början till slut (och kanske bara skriva ut alla tecken till att börja med).

För att testa metoden på några enkla exempel behöver ni anropa matches i main, och skriva ut resultatet.

Föreslagen algoritm: Observera att om man går igenom strängen från början till slut, så finns det vid varje givet tillfälle bara en (eller ingen) tillåten slutparentes, nämligen den som motsvarar den senaste startparentesen som ännu inte slutits. Mer allmänt finns det alltid en serie förväntade slutparenteser, med en viss ordning. Till exempel efter att ha läst in tecknen "**([(**" så förväntas '**)**', '**]**' och '**)**' i den ordningen. Om nästa tecken är till exempel '**{**' så ska '**}**' läggas "överst" på de förväntade tecknen. Om istället '**)**' läses in ska tecknet raderas från förväntade tecken. Om '**>**' läses in matchar inte strängen och metoden ska returnera false.

Ni kan använda en lista på tecken för att hålla reda på vilka slutparenteser ni förväntar er och i vilken ordning. Det sista elementet i listan är alltid den enda tillåtna slutparentesen. När ni läser in en startparentes lägger ni till dess motsvarande slutparentes sist i listan (så ni behöver kunna hitta slutparentesen som motsvarar en startparentes på något sätt). När ni läser in en slutparentes tar ni bort det sista elementet i listan och kontrollerar att det stämmer med den inlästa slutparentesen, annars returnerar ni false.
Fundera på: Hur ska listan se ut när ni läst in tecknen "**([<>{**"? Vad behöver ni kontrollera när teckenströmmen är slut?

Testning:

Exempel på giltiga parentesuttryck: "([-])" och "<>{x(**{{[a]}b})}c[_]"

Exempel på ogiltiga parentesuttryck: "(]", "(", ")", "(()" och "([)]".

b) Skriv en main-metod som testar alla exemplen ovan, och skriver ut resultaten från matches för var och en (true,true,false,false,false,false,false).

Uppgift 3 : Avståndsenheter

I den här uppgiften ska du skriva och testa ett par klasser för att konvertera mellan olika avståndsenheter.

a) Ladda ner TestUnits.java från kurshemsidan och lägg den i en egen mapp/eget Eclipse-projekt. Skapa en klass DistanceUnit i samma mapp/projekt. Klassen är till för att modellera olika avståndsenheter. Varje instans motsvarar en specifik avståndsenhet.

Klassen ska ha en konstruerare `public DistanceUnit(double metersPerUnit)`. Parametern är (som namnet anger) hur många meter det går på en enhet av den modellerade avståndsenheten.

Vidare ska den ha metoder toMetric och fromMetric för att konvertera ett värde i enheten till/från meter (se TestUnits för exempel på användning).

Till slut ska den ha konstanter METER, FOOT, PARSEC och PLANCK för avståndsenheterna meter, fot, parsec (jättelångt) och plancklängder (jättekort). Exempel på hur konstanterna kan användas finns i metoden **`testDistanceUnit`** i TestUnits.java. Det verkar kanske underligt att ha ett objekt för att konvertera från meter till meter, men syftet blir förhoppningsvis tydligare i nästa del av uppgiften. Använd följande tabell för att skapa konstanterna:

1 Fot	0.3048 meter
1 Parsec	$3.08567758 \times 10^{16}$ meter
1 Plancklängd	1.616229×10^{-35} meter

b) Skapa en ny klass DistanceConverter. Den här klassen modellerar konverteringar mellan olika enheter. Varje instans har en från-enhet och en till-enhet. Den ska ha två konstruerare:

```
public DistanceConverter(DistanceUnit from, DistanceUnit to)
public DistanceConverter(DistanceUnit from)
```

Den första utför konvertering från enheten **`from`** till enheten **`to`**. Den andra utför konvertering från enheten **`from`** till meter.

Metoderna ska vara **`convert`**, som utför själva konverteringen och **`reverseDirection`** som ger en ny konverterare som fungerar i motsatt riktning. Metoden **`testDistanceConverter`** i TestUnits.java ger ett par exempel på hur den används, men notera att metoden inte är fullständig, ni måste fixa till ett par "TODO"-markerade uppgifter.

När ni kör testerna bör ni kommentera bort anropet till `testDistanceUnit` i main-metoden, så ni slipper få så mycket utskrifter.

c) Innan redovisning, se till att ni utfört och raderat alla TODOs i kommentarerna i TestUnits.java och att de olika testmetoderna kan köras med de förväntade utskrifterna.

Uppgift 4: Persondatabas

Ni ska skriva en klass Person som håller reda på namn, personnummer och vem (om någon) de är gifta med. Personnummer ska lagras som ett heltal, men tänk på att typen int nog är för liten för personnummer. Alla instansvariabler ska vara privata.

a) Klassen ska ha en konstruerare som tar namn och personnummer. Konstrueraren ska krascha med ett RuntimeException ifall personnumret har en ogiltig kontrollsiffra, se:
https://sv.wikipedia.org/wiki/Personnummer_i_Sverige#Kontrøllsiffran
Kika även på:
https://sv.wikipedia.org/wiki/Luhn-algoritmen#Kontroll_av_nummer

b) Klassen ska ha två publika instansmetoder: marry(Person otherPerson) och divorce().

Ett anrop som x.marry(y) gifter person x med person y (samma sak som y.marry(x)). Ett anrop som x.divorce() skiljer x från dess partner (om den har någon).

Det är viktigt att er persondatabas alltid uppfyller följande regel: En person får aldrig vara gift med en person som är ogift eller som är gift med någon annan. Det betyder att:

1. Marry ska krascha om någon av de inblandade personerna redan är gift.
2. Marry måste ändra båda personernas (this och otherPersons) relationsstatus.
3. Divorce måste ändra båda personernas relationsstatus.

Vidare ska klassen ha en toString-metod som skriver ut personens namn, personnummer och namnet på personen de är gifta med (om någon).

c) Skriv en testmetod (en main-metod) som skapar, gifter och skiljer ett par personer. Argumentera för att ni har testat alla viktiga fall och ange i kommentarer vad det är ni testat i varje fall, och vad den förväntade utskriften är.

Tips: Använda try/catch där ni förväntar er att fel ska uppstå. Ungefär så här (`1111111111L` är en konstant av typen long):

```
try {
    new Person("X",1111111111L);
    System.out.println("SOMETHING HAS GONE HORRIBLY WRONG!");
} catch (RuntimeException e) {
    System.out.println(e.getMessage());
}
```

Om konstrueraren inte kraschar (den ska krascha) skrivs SOMETHING HAS GONE HORRIBLY WRONG ut, annars skrivs bara felmeddelandet ni valt ut.

d) Skapa en klass Employee som ärver från Person och utökar den med två metoder: setSalary och getSalary för att ändra/läsa personens lön. Skapa en testmetod som visar att det till exempel går att gifta en Person med en Employee och så vidare.