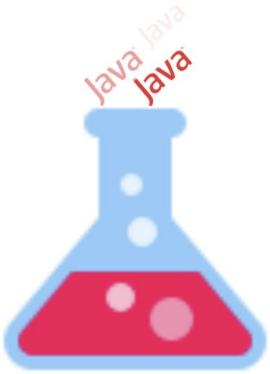


TDDC77 - Objektorienterad programmering

Laboration 3

Institutionen för datavetenskap
Linköpings universitet

Det är dags för Linus och Linnea att ta sig än den avslutande laborationsuppgiften i period 1. Sista delmomentet är dessutom individuellt så att ni alla får chansen att visa på era nyvunna kunskaper. Innan ni börjar med uppgiften är det viktigt att ni är bekanta med Javas API samt att ni har förstått och övat på följande begrepp:

- Primitiva typer (int, char ...)
- Strängar
- Villkorliga hopp
- Loopar (for, while ...)
- Metoder och metodanrop
- Arrayer
- Klasser och instansiering

VIKTIGT! Ni ska i den färdiga lösningen inte använda andra klasser än inmatning och utmatnings klasser, wrapper-klasser, String, Arrays, Queue och Stack. I början av en grundkurs som denna ligger fokus på att använda enkla byggstenar för att bygga upp datastrukturer och funktionalitet, inte att fullt ut använda Javas klassbibliotek.

3.1 Infix- och postfix-notation

Kalkylatorprogrammet som Linus och Linnea tänker skriva ska kunna beräkna enkla aritmetiska uttryck. Man ska kunna skriva in ett uttryck och slå RETURNtangenten, varvid uttryckets värde beräknas och skrivs ut. När programmet körs har de tänkt sig att det skulle se ut enligt följande.

```
[john@emil23 java]$ java LinCalc
LinCalc, mata in matematiska uttryck:
> 1 + 2.5
Resultat: 3.5
> 4*3.1+2
Resultat: 14.4
> 1-8/(4+2)
Resultat: -0.33333333333333326
```

Linnea (som inte bara sitter och hackar) har berättat för Linus att hon läst att ett program lätt kan beräkna ett aritmetiskt uttryck, om det är på en form som kallas postfix notation (omvänd polsk notation). I denna notation skrivs operatorerna (t.ex. + och -) efter sina operander (t.ex. 23 och 1.56). Uttrycken ovan är skrivna i så kallad infix-notation. För de uttrycken har Linus kommit fram till att motsvarande postfix uttryck bör vara följande:

Infix-notation	Postfix-notation
1 + 2.5	1 2.5 +
4 * 3.1 + 2	4 3.1 * 2 +
1 - 8 / (4 + 2)	1 8 4 2 + / -

Det finurliga med postfix-notationen är att den kan läsas från vänster till höger, och att deluttryck då successivt kan beräknas (slutligen hela uttryckets värde), utan att programmet behöva veta vad som kommer längre fram! För det andra postfix-uttrycket ovan skulle detta innebära, steg för steg:

1. Läs och spara 4
2. Läs och spara 3.1
3. Läs *
4. Hämta 3.1
5. Hämta 4
6. Beräkna 4*3.1 (= 12.4)
7. Spara resultatet 12.4
8. Läs och spara 2
9. Läs +
10. Hämta 2
11. Hämta 12.4
12. Addera 12.4+2 (= 14.4)
13. Spara resultatet 14.4
14. Uttrycket är slut, dess värde är det sist sparade resultatet

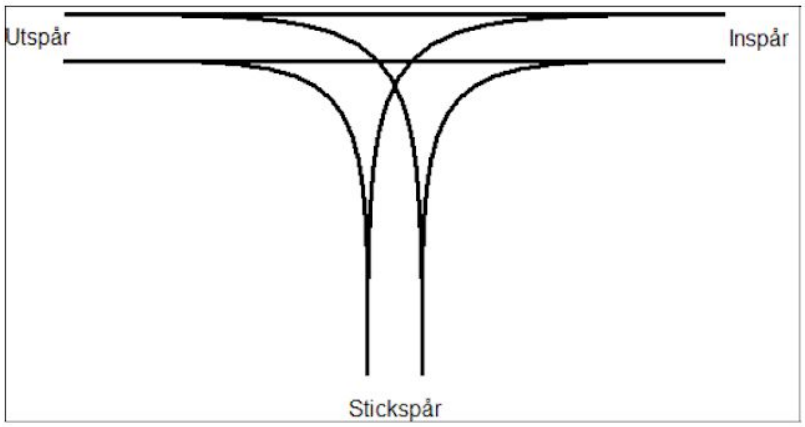
Programmet måste alltså kunna spara deluttryckens värden, i väntan på att de ska ingå i en efterföljande beräkning. När Linnea tittar på det sista postfix-uttrycket inser hon att många värden kan behöva sparas, i väntan på att ingå i en beräkning, i detta fall fyra stycken (1, 8, 4 och 2). Det verkar vara så att värdena ska hämtas i omvänd ordning, i förhållande till den de sparats undan i.

3.2 Järnvägs Algoritmen

Linus (som nu också börjat studera litteratur) har hittat en algoritm som kallas järnvägs algoritmen. Den kan användas för att göra om infix-uttryck till postfix-uttryck. Algoritmen har fatt sitt namn av att den kan illustreras som en enkel rangerbangård.

Operander och operatörer körs in från höger. Operander körs alltid rakt fram, till utspåret. Operatörer (och i förekommande fall vänsterparenteser) körs alltid ner på stickspåret, men dessförinnan ska i vissa fall (ledtråd: prioritet) operatörer som redan finns där köras till utgången, innan den inkommande operatören körs ned på stickspåret. Symboler kan aldrig köras tillbaka från utspåret och inte heller köras tillbaka från sticksparet till inspåret. Det finns några ganska enkla regler för när de här olika sakerna ska ske. Ta reda på de reglerna genom att prova med nagra exempel på papper. Jämför era resultat med Linus i tabell på förra sidan.

Figur 1: Järnvägs algoritmen med in-, ut- och stickspår.



3.3 Programmeringen

Nu har Linus och Linnea hittat ett sätt att konstruera sitt kalkylatorprogram. De har dock fått oväntad hjälp av en bekant som heter John. Enligt ryktet har han implementerat en kalkylator. Tyvärr har de inte tillgång till källkoden utan bara hans kompilerade kod (så kallad bytekod). Men Linnea tycker det borde gå att successivt byta ut Johns metoder mot egna. På så sätt har man också chansen att bygga ut kalkylatorn med funktioner som John inte har tänkt på än om man vill. Det är alltså tänkt att slutresultatet ska vara ett program med precis samma delar som i Johns:

1. Dela upp inmatningen i tal och operatorer
2. Överför från infix till postfix
3. Beräkna värdet

3.3.1 Steg 1 - Få igång Johns kod

För att kunna använda Johns kod så måste ni först göra ett par inställningar. Öppna (eller skapa) ert Java projekt med Eclipse. Högerklicka på ditt projekt i *projekt explorer*, välj *properties*. Klicka på *Java build path* till vänster, tabben *libraries*, *Add external JARs*, välj *"LinCalcJohn.jar"*. *LinCalcJohn.jar* finns i följande mapp: https://www.ida.liu.se/~TDDC77/material/labbar/lab3

Note: Kopiera först *LinCalcJohn.jar i projektmappen.*

Fortfarande i tabben *Libraries*, klicka på *add library*, välj *JUnit*, välj 4 (OBS: version 3 kommer inte att fungera). Färdigt, stäng inställnings fönstren.

I samma mapp som jarfilen finns även två javafiler. Den första heter Lin-Calc.java och innehåller ett kodskelett, den andra LinCalcTest.java innehåller testfall. Kopiera dessa till ert projekt. För att anropa en av Johns metoder skriver man helt enkelt:

```
LinCalcJohn.metodensNamn(argument);
```

För att istället köra sin egen metod så byter man ut anropet till Johns metod mot kod som ska göra motsvarande sak. Linus och Linnea bestämmer sig för att först testa den givna miniräknaren med några längre matematiska uttryck. Kan ni hitta några fel i Johns program?

Tutorial om hur man kan komma igång med Eclipse för att göra labb 3:
https://www.youtube.com/watch?v=kzaojdvQCXc&feature=youtu.be

Följande tas upp i videon:
1. Hur man kan skapa ett nytt projekt och samtidigt undvika module-info.java.
2. Hur man kan lägga till LinCalcJohn.jar och JUnit4 till Classpath.
3. Hur ”default package” fungerar.
4. Hur man kan skapa och använda andra typer av filer såsom text-filer i projektet.
5. Hur man kan skapa egen paketstruktur.

3.3.2 Steg 2 - Ta emot ett infix-uttryck

Eftersom Linus och Linnea bestämt sig för att de kan skriva ett minst lika bra program som John så börjar de med att ta emot ett infix-uttryck. Ett uttryck som läses in från tangentbordet lagras i en typ som kallas för String. Detta är ju bara en sekvens av tecken, som i sig inte har någon betydelse för Java. På något sätt måste denna sträng spjälkas upp i enheter så som operatorer eller tal/variabler. T.ex. så skulle strängen "123+asdf12*plu(3003 45" kunna delas upp i följande enheter "123", "+", "asdf12", "*", "plu", "(", "3003" och "45".

Våra två programutvecklare tycker att det vore praktiskt om man hade en metod som gjorde denna uppspjälkning. Givet en sträng så kunde metoden returnera en array av lexikaliska enheter (delsträngar). För att se att allt blivit rätt så provar man att skriva ut resultatet från uppspjälkningen i skalfönstret.

När de har skrivit klart sin egen spjälkare så borde miniräknaren fortfarande fungera men utan att anropa LinCalcJohn.lex().

Linus inser att hanteringen av unära minus, dvs negativa tal, blir lite speciell. Järnvägsalgoritmen bara fungerar nämlingen för binära operatorer (alltså operatorer som opererar på två operander, dvs siror). I indatan kan det tyvärr förekomma två sorters minustecken, både de vanliga binära och unära.

- Uttryck med enbart unära minusoperatorer: -1, -3+-3, 3*-2
- Uttryck med enbart binära minusoperatorer: 3*5-2, 1-1
- Hur identierar man unära minus? (Ledtrad: vad står innan?)

För att järnvägsalgoritmen ska fungera så finns det era sätt att hantera unära minus (välj själv vilken som verkar lättast). En lösning går ut på att internt - i sitt program - konvertera unära minus till en ny operator, säg (~) med högsta prioritet och låta denna endast ta ett tal från operatorstacken istället för två.

Det går även att redan under spjälkningen se till att unära minus hålls samman med sina siror när de sedan konverteras till yttal av java så blir de negativa och allting fungerar som förväntat.

3.3.3 Steg 3 - Omvandling till postfix och beräkning av + och -

Nu tänker Linus och Linnea skriva en metod som omvandlar infix till postfix. De börjar med att endast hantera heltal och operatorerna + och -. Miniräknaren bör nu fungera utan att anropa LinCalcJohn.toPostfix() men än så länge bara med heltal, addition och subtraktion.

3.3.4 Steg 4 - Omvandling till postfix och beräkning av * och /

Operatorerna * och / införs. De operatorerna har ju högre prioritet än + och -, vilket innebär att den fullfjädrade järnvägs algoritmen nu måste tas i bruk. Linnea och Linus tuffar vidare. Nu bör miniräknaren kunna räkna med alla fyra räknesätten.

3.3.5 Steg 5 - Parenteser

Parenteser läggs till, så att man kan styra i vilken ordning deluttryck i sammansatta uttryck ska beräknas. Miniräknaren bör kunna räkna godtyckliga uttryck med heltal, parenteser och de fyra räknesätten. Linus och Linnea testar med några lite mer komplicerade uttryck som t.ex. 6-3*(15/15*2) som ju borde bli noll.

3.3.6 Steg 6 - Reella värden

Möjligheten att ange reella värden på fixpunktsform införs. För reella värden ska gälla, att det måste vara minst en siffra före respektive efter decimaltecknet. Fixpunktsformatet har ingen exponentdel.

3.3.7 Steg 7 - Beräkning av postfixuttryck (individuell uppgift)

Nu återstar bara att byta ut metoden LinCalcJohn.calc() sen har Linus och Linnea programmerat en hel miniräknare själva. Därför bestämmer de sig för att göra det var för sig för att visa att de verkligen har lärt sig programmera. Med några exempel på papper så inser de snart hur ett uttryck på postfixform kan beräknas av datorn. Läs instruktionerna nedan angående inlämning.

3.4 Inlämning av LinCalc

3.4.1 Generella krav på kodkvalitet

Ett viktigt mål vid programmering är att lätt kunna skriva, läsa, förstå och underhålla programkod, även sådan andra programmerare skrivit. Därför ställs krav att varje programmerare skriver välstrukturerad, kommenterad programkod. Det gäller naturligtvis även er. Vid redovisning ställs nedan krav på er kod.

3.4.2 Testning

För att verifiera att er kalkylator fungerar behöver ni köra ett antal testfall och se att den räknar rätt. När ni skapar testfall, försök hitta vissa kombinationer av indata som ni tror kan orsaka problem, men glöm inte testa med "vanliga" indata också. Här nedan finns exempel på hur testfallen för LinCalc kan se ut:

- `3,5-3,5{3,5 = 3,5`
- `1+(1,9*(2,9*(3,9*(4,9*5,9)))) = 622.24699`
- `(((123,123))) = 123.123`
- `2,0/2,0*5 = 5.0`
- `5*-(6/2) = -15`
- `1*2/3*4 = 2.666666666667`
- `-1*7/-2/7*2 = 1`
- `(1+2)*(3-1) = 6.0`
- `18`
- `2,0/2,0*5 = 5.0`
- `1-2*3+4 = -1.0`

3.4.3 Metodstorlek

Era metoder bör inte vara större än 30-40 rader. Annars går det inte att överblicka vad ni gjort. Blir metoden större då är det dags att antingen generalisera (dvs minska antalet specialfall) eller dela upp metoden i flera metoder.

3.4.4 Loopar

Ni behöver i princip aldrig skriva mer än en loop nästlad i en annan, dvs en loop i en annan loop. Fler loopar än så i varann är inte bara onödigt utan ineffektivt, svårförståeligt och riskabelt när det gäller buggar. Ni bör heller aldrig joxa (räkna upp, räkna ner, återställa) med iterationsvariablerna mitt i en for-loop. Detta för att slutvillkoret bara testas i slutet av varje slinga. Joxar ni så riskerar ni att hamna utanför villkoret mitt i loopen. Inget joxande alltså.

3.4.5 Villkorssatser

Långa kaskader av `if-else if-else if-else if` är sällan eller aldrig en bra ide. Det tyder på att ni har uppfunnit en massa specialfall som alldeles säkert går att slå ihop till ett par generella fall istället. Och långa logiska villkor i stil med `if(bla && bla && bla && bla || (bla || bla || bla))` är inte heller en bra ide. Det tyder också på en massa specialfall som kan generaliseras.

3.4.6 Radlängd

Undvik kodrader längre än 80 tecken. Många textbaserade verktyg bygger på uppfattningen att en rad är 80 tecken lång. T.ex. startar kommandoskal såväl som Atom med radlängden 80 tecken. Skriver ni längre rader kommer ni få automatiska radbrytningar som gör koden grötig, svårläst och oöverskådlig. Ibland blir det även så illa att för långa rader inte kommer med helt på utskrifter. De flesta programkonstruktioner går att dela upp eller radbryta och indentera på ett lämpligt sätt för att undvika långa rader.

3.4.7 Kommentering

Kommentera två saker: Innan era metoddeklarationer ska ni kommentera vad metoden gör, vad den tar för inparametrar och vad den returnerar. Där ni gör något intressant/klurigt/svårtolkat ska ni kommentera framför allt varför ni gör det. Exempel på sådana ställen är algoritmer eller komplex hantering av data. Självklart ska ni inte ha kommentarer som

```
// Ökar variabeln i med ett
// En array av strängar. Snarare kommentarer som
// Behandlar operatorerna först för att slippa detektera unära minus senare.
```

Kommentarer är till för att öka läsbarheten och förståelsen av koden, inte för att undervisa personer som inte kan programmera. En bra kommentar mitt i koden är inte längre än två rader. En metodkommentar kan vara längre.

3.4.8 Indentering

Eclipse är snäll mot er och hjälper till att indentera koden korrekt. Använd ctrl-shift-f.