TDDC77 Objektorienterad programmering
Laborationsinstruktioner period 1

Ahmed Rezine
ahmed.rezine@liu.se
Institutionen för datavetenskap
Linköpings universitet
Bearbetning av tidigare versioner av Johan Jernlås, John Wilander och Jonas Wallgren

6 september 2015

Allmänt om laborationerna

Här följer den information ni behöver för labbserien i period 1 av TDDC77 Objektorienterad programmering.

LÄS NOGA IGENOM DETTA AVSNITT!

Att labba, öva och ha kul

Programmering är kreativt, spännande och utmanande. Det handlar om att experimentera, fundera och allra mest **prova**. Datorn tröttnar aldrig på er. Den låter er gladeligen vrida och vända på era program för att testa hur det hela fungerar. Därför uppmuntrar jag er till att söka uppgifter i böcker och på nätet, och att utforska uppgifterna i häftet tills ni förstår dem.

Utseende på programmen

Första periodens labbserie leder fram till *LinCalc*. Ni skall demonstrera först och sen lämna in (per email) de *-märkta uppgifter och LinCalc. Det ställs krav på att er programkod är välskriven. Därför finns tips och krav i sista kapitlet av detta kompendium.

Deadline

Senaste datum för examination på laborationerna för period 1 finns angivet på kurshemsidan. Det betyder att hela laborationsuppgiften LinCalc (se avsnitt 3) och de *-märkta uppgifter ska demonstreras för assistenten och lämnas in (per mail) senast då (både de delar ni skrivit i par och den del ni skrivit individuellt). De uppgifter som demonstreras och lämnas in i tid kommer gås igenom av assistenten och ni har chans att rätta eventuella fel i dem.

Anmäl er till labbarna

Ni måste först se till att dela upp er två och två i labbpar. Om ni är udda eller har särskilda skäl att labba ensamma, kontakta kursansvarig. Större grupper än två och två är inte tillåtna.

När ni hittat någon att labba med så anmäler ni er enligt länk på kursens webb (se webreg). Nu är det upp till er att planera och lägga upp labbandet så att ni hinner i tid till deadline. Observera att det kan behövas mer labbtid än vad som är schemalagt. Med andra ord är det bara att leta reda på lediga datorer på IDA eller labba hemma.

I den här kursen är det meningen att ni ska turas om att skriva programkod och redovisa resultaten. Det vill säga ni ska byta plats framför datorn mellan varje labb, eller oftare, (i sista labben gäller det mellan redovisningarna). Det är nämligen oerhört viktigt att alla har provat på ordentligt både att skriva och att granska kod. Sista delmomentet är dessutom individuellt vilket förutsätter att alla kan skriva och förklara Javakod på egen hand.

Innan första labben

Första gången föreslår jag att ni börjar med att skapa en katalog för den här kursen genom följande kommandon i ett skalfönster:

```
> cd
> mkdir tddc77
> cd tddc77
```

Laborationstillfällen

Det finns ett antal laborationsfällen i period 1. Dessa är inte tänkta att direkt motsvara laborationsuppgifter, det är upp till varje laborationsgrupp att disponera sin tid. Det är troligt (och meningen) att ni kommer att behöva använda icke schemalagd tid för att hinna med.

Observera att sista laborationsuppgiften, LinCalc, är mycker omfattande och kommer att kräva en hel del arbete. Den skall inte ses som ”Nu är det bara en laboration kvar”, utan den kommer att kräva det mesta arbetet.

Inför varje laborationstillfälle så skall ni se till att vara pålästa/förberedda. Ibland säger man att programmering sker vid skrivbordet, inte vid tangentbordet. Ni skall alltså komma till labblokalen och veta vad ni skall göra. Om ni inte har kommit på en komplett, korrekt lösning så bör ni i alla fall veta vilka varianter ni tycker att ni bör testa. Båda studenterna i ett par bör vara förberedda så att samarbetet flyter och båda lär sig.

Dags att köra igång – lycka till!

1 Laboration 1

1.1 Förberedelseuppgift

Fyll i luckorna och svaren på följande frågor. Kan du inte svaren får du läsa på i boken. Redovisa för labbhandledaren.

För att lagra värden använder man en ...

och den måste alltid ha en ...

`tal = 12;` kallas att göra en ...

och det måste man alltid göra innan man ...

Skriv en villkorssats som testar om `int tal` är större än 10 och i så fall skriver ut "Större än 10", annars skriver ut "Ej större än 10". Utskrift sker med `System.out.println("texten");`.

Svar:

Skriv en for-loop som skriver ut alla siffror från 0 till 9. Utskrift sker med `System.out.println(siffran);`.

Svar:

1.2 Att skriva Java-kod

När ni är redo att programmera så startar ni editorn Emacs. I ett skalfönster kan det ske som följer:

```
> emacs NamnetPåErJavaklass.java &
```

Där det står `NamnetPåErJavaklass` ska ni skriva in vad ni tänkt döpa er Javaklass till (namnet ska vara kopplat till det programmeringsproblem ni ska lösa). Observera att er fil *måste* heta samma sak som klassen/programmet fast med ".java" på slutet. &-tecknet gör att Emacs startas i en separat process och därför inte låser skalfönstret.

I Emacs kan ni få stöd genom så kallad *syntax highlighting*. Det betyder att Emacs känner igen t.ex. alla reserverade ord i Java och automatiskt sätter olika färger på olika delar av programkoden. På det viset blir det mycket lättare att läsa sina Javaprogram. Om det inte är så när ni startar Emacs kan ni starta det genom att klicka er fram till följande menyval:

Options → Global Font Lock

Nu är det bara att skriva in Javakod i Emacs. När ni vill prova att kompilera och köra ert program så sparar ni först koden i Emacs (Ctrl-x Ctrl-s) och skriver sen följande i skalfönstret:

```
> javac NamnetPåErJavaklass.java
> java NamnetPåErJavaklass
```

Om något inte var korrekt i er programkod så kommer kompilatorn (javac) att ge er felmeddelanden. Då kommer heller ingen körbar fil att skapas vilket betyder att ni inte kan starta programmet med kommandot `java NamnetPåErJavaklass`. Istället får ni läsa felmeddelandet och sen återgå till Emacs och försöka rätta till felen som kompilatorn rapporterade.

1.3 Första programmet ...

I de följande labbarna kommer ni dels lära er hur information kan flyttas mellan er och ert Javaprogram, dels prova på några av Javas konstruktioner för att styra programflödet. De labbar som markerats med en asterisk (*) ska ni redovisa för labbhandledaren och sen skickas per mail till honom/henne.

Inmatning kommer ske via tangentbordet och utmatning via monitorn. Om ni vill prova på att få igång ett Javaprogram direkt så ta något ur er lärobok eller skriv av programmet Backwards (döp filen till Backwards.java):

```
import java.util.Scanner;

class Backwards {
  public static void main(String[] args){
    System.out.println("Ange text:");
    Scanner in = new Scanner(System.in);
    String inputString = in.nextLine();

    for(int i=inputString.length()-1; i>=0 ; i--){
      System.out.print(inputString.charAt(i));
    }
  }
}
```

Fundera på vad som händer på varje rad.

Ändra programmet så att en tom rad skrivs ut efter att baklängesutskriften är klar.

Nu är det dags att börja lösa labbuppgifter!

1.4 Hur gammal är du?

Skriv ett program som frågar användaren hur gammal han/hon är och sen skriver ut åldern på skärmen.

En programkörning skulle kunna se ut så här:

```
[john@emil23 java]$ java HurGammal
Hej! Hur gammal är du?
27
Du är 27 år gammal.
[john@emil23 java]$
```

1.5 De fyra räknesätten

Skriv ett program som ber användaren mata in två tal (behöver inte vara heltal) och sen presenterar resultatet av de fyra räknesätten.

En programkörning skulle kunna se ut så här:

```
[john@emil23 java]$ java FyraRaknesatt
Mata in ett tal:
1.6
Mata in ett tal:
23
1.6 + 23.0 = 24.6
1.6 - 23.0 = -21.4
1.6 * 23.0 = 36.800000000000004
1.6 / 23.0 = 0.06956521739130435
[john@emil23 java]$
```

1.6 Vad heter du?

Skriv ett program som frågar användaren vad han/hon heter och sen hälsar på honom/henne.

En programkörning skulle kunna se ut så här:

```
[john@emil23 java]$ java VadHeterDu
Vad heter du?
John Wilander
Hej, John Wilander!
[john@emil23 java]$
```

1.7 Största tal

Skriv ett program som ber användaren om två heltal och sen skriver ut det största av de två talen.

En programkörning skulle kunna se ut så här:

```
[john@emil23 java]$ java StorstaTal
Mata in ett heltal:
34
Mata in ett till heltal:
3
34 är störst.
[john@emil23 java]$
```

1.8 Räkna växelpengar *

Låt oss nu lösa en lite mer komplicerad programmeringsuppgift. Ni ska skriva ett program som räknar ut växel för svenska pengar. Uppgiften är uppdelad i små steg som följer:

1. Skriv ett program som räknar ut växel på en tiokrona i femkronor och enkronor. Användaren anger ett pris mellan en och tio kronor och ert program svarar med vilken växel han/hon får tillbaka.

2. Ändra till växel på en femhundralapp i hundralappar, femtiolappar, tjugor, tiokronor, femkronor och enkronor. Användaren får nu ange ett pris mellan en och 500 kronor.

3. Lägg till att användaren får ange både priset och betalningssumman. Utöka programmet så att det hanterar femtioöringar.

   En programkörning efter steg 3 skulle kunna se ut så här:

   ```
   [john@emil23 java]$ java Pengar
   Ange pris (kr):
   124.50
   Du betalar (kr):
   500
   Du får tillbaka 3 hundralappar, 1 femtiolappar, 1 tjugor, 0 tior,
   1 femmor, 0 enkronor och 1 femtioöringar.
   [john@emil23 java]$
   ```

4. Studera programmet. Flera steg är troligen väldigt snarlika. Samma operationer utförs med olika data. Ändra er lösning så att ni använder arrayer för att spara datat i varje steg och en loop för att gå igenom alla steg.

När ni är klara med programmet tänk sedan igenom och testa följande:

- Vad händer om man betalar för lite?
- Vad händer om man matar in ett negativt pris eller en negativ betalningssumma?
- Vad händer om man matar in bokstäver istället för siffror, t.ex. priset ”tio”?
- Får man någon växel om man matar in priset 5.60 kr och betalar 6 kr?
- Om man ska få 60 kr i växel vill man kanske ha det i form av tre tjugor. Får man det via ert program? Om inte, hur skulle det lösas?

Nu har ni genomfört ert första test av ett program! Det är alltid bra att utsätta sina lösningar för olika test och specialfall. Man upptäcker alltid saker man glömt eller inte förutsett. Redovisa programmet för handledaren.

2 Laboration 2

2.1 Förberedelseuppgift

Fyll i luckorna och svaren på följande frågor. Kan du inte svaren får du läsa på i boken. Redovisa för labbhandledaren.

En uppsättning med flera variabler i rad kallas en ...

och man indexerar en sådan från till

Man kan dela upp funktionaliteten i sitt program/sin klass i flera ...

och sådana kan något till den som anropade.

Skriv en **while-loop** som skriver ut alla tecken i `char[] tecken = {'a', 'b', 'c'};`. Utskrift sker med `System.out.println(tecknet);`.

Svar:

Skriv en **for-loop** som skriver ut alla tecken i en sträng var för sig, t.ex. ur `String sträng = "Veckans bubblare";`. Utskrift sker med `System.out.print(tecknet);`.

Svar:

(Variabelnamnet `sträng` innehåller ju ett `ä`. Det borde vara så att olika nationella tecken fungerar utan problem i Java. Om ni alltid använder samma dator/system så borde det fungera, men om ni t.ex. skall flytta kod mellan hemmadatorn och IDA så kan det uppstå problem. Försök undvika variabelnamn som `räksmörgås` och `Dvořákçāõ`.)

2.2 Få igång Eclipse IDE

Från och med nu får ni (om ni så önskar) utveckla Java-program i ett program som heter *Eclipse*. Eclipse är en *integrerad utvecklingsmiljö* som erbjuder er många fördelar när ni programmerar. Den har bland annat *syntax highlighting*, *inkrementell kompilering* (kompilering görs hela tiden i bakgrunden), samt stöd för *debuggning* och *refactoring*. Eclipse är dessutom världens mest populära miljö för Java så chansen är stor att ni kommer arbeta med Eclispe i framtiden.

Eclipse är gratis och finns för Windows, Mac, Linux, Solaris med mera. Alltså går det utmärkt att arbeta hemma – gå bara till `www.eclipse.org` och ladda ner rätt version för er dator.
Från terminalen kan ni starta Eclipse med:

```
> eclipse &
```

Ni kommer att bli tillfrågade om var ni vill spara er Eclipse-arbetsyta (workspace). Välj `er_hemkatalog/workspace` och tryck OK.

2.3 Bekanta dig med laborationsmiljön

Denna övning går ut på att lära er in- och utmatning av text samt bekanta er med Eclipse. Ju mer ni upptäcker och testar i Eclipse nu desto snabbare kommer resten av labbserien gå. Längs vägen kommer ett par frågor som ni ska besvara (labbassistenten går runt och tittar så att era svar verkar rätt).

Starta Eclipse om ni inte har gjort det. Välj *Workbench*. Till vänster finns en flik med namnet *Package Explorer*. Högerklicka i den tomma, vita ytan och välj *New* → *Project...* . I fönstret som kommer upp väljer ni *Java Project*-wizarden och trycker *Next*. Döp ert projekt till något lämpligt i stil med TDDC77Labs och tryck sen *Finish*.

Nu ska en projektmapp med ert namn ha kommit upp i *Package Explorer*. Högerklicka på projektmappen och välj *New* → *Package...* . Döp paketet till exempelvis lab2 eftersom det är lab 2 ni ska börja att arbeta med. Observera att paketnamnet precis som metoder och variabler ska börja med liten bokstav. Om ni börjar paketnamnet med stor bokstav kommer Eclipse att varna er – "Discouraged package name. By convention, package names usually start with a lowercase letter". När ni döpt paketet trycker ni *Finish*. Nu ska ni ha en liten vit paketsymbol i er projektmapp.

Ni kommer att skriva klassen `AreaCalculator` för att bekanta er med Eclipse.

2.4 AreaCalculator

Klassen `AreaCalculator` har som uppgift att beräkna arean av en triangel givet dess bas och höjd, samt beräkna arean av en cirkel givet dess radie.

2.4.1 Sätt upp ett projekt i Eclipse

Högerklicka på den vita paketsymbolen och välj *New* → *Class...* . Nu får ni upp ett fönster med en massa valmöjligheter. Två av textfälten är redan ifyllda – "Source folder" och "Package". Det är för att ni markerade det paketet i den projektmappen när ni valde att skapa en ny klass.

Döp er klass till exempelvis AreaCalculator. Notera att klassnamnet börjar med stor bokstav och att det är ett substantiv. Klassnamn bör vara substantiv eftersom det känns naturligt att instansiera objekt som är någon sorts sak, dvs substantiv. Ni kan också kryssa för att ni vill skapa en *metodstubbe* (eng. method stub) för metoden `public static void main(String[] args)`. Tryck nu på *Finish*.

Ni bör nu ha fått upp källkoden till er nya klass under fliken AreaCalculator.java:

```
package lab1;

public class AreaCalculator {
  /**
  * @param args
  */
  public static void main(String[] args) {
    // TODO Auto-generated method stub
  }
}
```

Lägg till kod för att skriva ut någon text till skärmen.
Prova att köra igång programmet genom att klicka på menyvalet *Run* → *Run As* → *Java Application*. Under källkoden finns en flik som heter *Console*. Den fungerar som ett skalfönster, dvs det är där programmet skriver saker och det är där användare kan skriva in saker om programmet ber om det.

När ni skriver kod kan det hända att ni får ett kompileringsfel. Eclipse kompilerar hela tiden er kod i bakgrunden och så fort ni får ett kompileringsfel så markeras det med röd understrykning och en glödlampa med ett rött kryss. På samma sätt som med varningar (gul understrykning) så kan ni få reda på vad som är fel genom att hålla muspekaren över felmarkeringen.

2.5 GissaTal – ett litet spel *

Nu över till en lite mer intressant uppgift. Ni ska programmera ett litet spel som heter GissaTal. Första spelaren matar in ett heltal mellan noll och 100. Sen får den andra spelaren gissa vilket tal det var. För varje gissning får spelare två reda på om gissningen var rätt, för liten eller för stor.

1. Skapa ett program som ber spelare ett att mata in ett heltal mellan noll och 100. Därefter ska programmet tömma skalfönstret genom att skriva ut radbrytningar tillräckligt många gånger.

2. Nu ska programmet be spelare två att gissa på ett heltal mellan noll och 100. Om gissningen är fel så ska programmet be om ett nytt heltal. Om gissningen är rätt ska programmet gratulera till vinsten, skriva ut hur många gissningar som behövdes för att hitta rätt tal och sen avslutas.

3. Sist ska ni införa ledtrådar. Spelare två ska få veta om hans/hennes gissning är för liten eller för stor.

En programkörning efter steg 3 skulle kunna se ut så här:

```
[john@emil23 java]$ java GissaTal
Spelare ett, Mata in ett heltal mellan 0 och 100: 18

    *** Skärmen töms ***

Spelare två, chansa på ett heltal mellan 0 och 100: 20
För stort!
Spelare två, chansa på ett heltal mellan 0 och 100: 5
För litet!
Spelare två, chansa på ett heltal mellan 0 och 100: 15
För litet!
Spelare två, chansa på ett heltal mellan 0 och 100: 18
Rätt! Du behövde 4 gissningar.
[john@emil23 java]$
```

Kör programmet för handledaren. Kan ni komma på någon effektiv gissningsstrategi? Hur många gissningar behöver man maximalt med er strategi? Skulle man kunna skriva ett Javaprogram som spelade med er strategi, dvs där en människa matar in ett heltal och sen låter programmet försöka hitta talet med så få gissningar som möjligt (utan att fuska)? Prova!

2.6 Textstatistik – ett ordbehandlingsverktyg *

Förr i tiden skrev man långa dokument på skrivmaskin (har ni provat någon gång?). Fördelarna med att skriva på dator är många och stora. Några av fördelarna är att datorn mycket snabbt kan gå igenom texten och hjälpa till med rättstavning, grammatik, avstavning och statistik.

Det sistnämnda ska ni nu prova på att programmera – ett program som tar fram statistik på antal ord, antal meningar, antal bokstäver, antal skiljetecken samt medellängd på orden. Som indata tar ni en vanlig textfil som ni skrivit (t.ex. i Emacs). En körning av programmet skulle kunna se ut så här:

```
[john@emil23 java]$ java Textstatistik MinTextfil.txt
Antal ord: 26
Antal meningar: 6
Antal bokstäver: 97
Antal skiljetecken: 8
Medellängd på ord: 3.73 bokstäver
[john@emil23 java]$
```

(Obeservera att meddellängden är avrundad till ett vettigt antal decimaler.)

Ni får också några tips: Se till att avsedda delar av er kod ser ut som nedan. Ta sedan reda på om det finns några bra metoder i klassen `BufferedReader` (se API:et). Kör programmet för handledaren.

```
import java.io.*;

class Textstatistik {
  public static void main(String[] args) throws IOException {

  // Läser in filen som gavs som första argument
  BufferedReader input = new BufferedReader(new FileReader(args[0]));
```

Skriv en funktion som tar reda på om ett visst tecken finns i en uppsätttning av tecken: `public static boolean memberOf(char,String)`. Använd funktionen i programmet.

Det är inte tillåtet att först läsa in hela filen på en gång. Ni ska läsa ett tecken i taget.

3 Laboration 3

Linus och Linnéa ska bygga en kalkylator. Emil och Emilia på Chalmers, Osquar och Osqulda på KTH samt Truls och Trula från Lund kan också bygga kalkylatorer. Frågan är vilka som är bäst?

Det är dags att ta sig an 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

VIKTIGT! Ni ska i den färdiga lösningen inte använda andra klasser än **String**, **Integer**, **Double** och **Character**. Ni kan också använda Arrayer. 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 postfixnotation

Kalkylatorprogrammet som Linus och Linnéa tänker skriva ska kunna beräkna enkla aritmetiska uttryck. Man ska kunna skriva in ett uttryck och slå RETURN-tangenten, varvid uttryckets värde beräknas och skrivs ut. När programmet körs har de tänkt sig att det kunna 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
```

Linnéa (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 infixnotation. För de uttrycken har Linus kommit fram till att motsvarande postfixuttryck bör vara följande:

Infixnotation	Postfixnotation
`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 postfixnotationen 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 postfixuttrycket ovan skulle detta innebära, steg för steg:

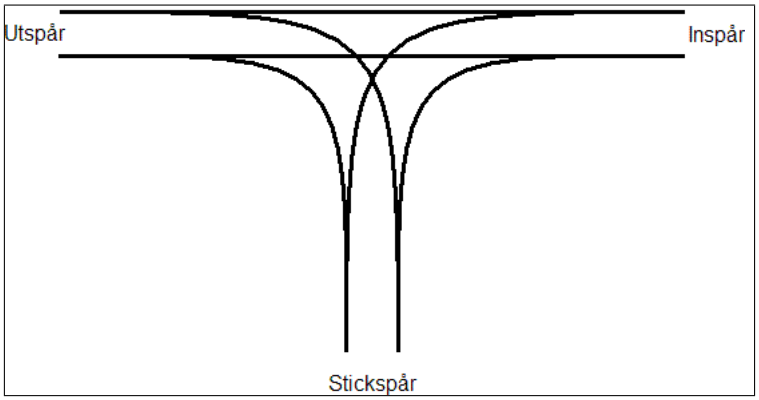
1. Läs och spara 4
2. Läs och spara 3.1
3. Läs *
4. Hämta 3.1
5. Hämta 4
6. Beräkna 4*3.1 (= 12.4)
7. Spara resultatet 12.4
8. Läs och spara 2
9. Läs +
10. Hämta 2
11. Hämta 12.4
12. Addera 12.4+2 (= 14.4)
13. Spara resultatet 14.4
14. Uttrycket är slut, dess värde är det sist sparade resultatet

Programmet måste alltså kunna spara deluttryckens värden, i väntan på att de ska ingå i en efterföljande beräkning. När Linnéa tittar på det sista postfixuttrycket 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 Järnvägsalgoritmen

Linus (som nu också börjat studera litteratur) har hittat en algoritm som kallas järnvägsalgoritmen. Den kan användas för att göra om infixuttryck till postfixuttryck. Algoritmen har fått sitt namn av att den kan illustreras som en enkel ranger-bangård.

Operander och operatorer körs in från höger. Operander körs alltid rakt fram, till utspåret. Operatorer (och i förekommande fall vänsterparenteser) körs alltid ner på stickspåret, men dessförinnan ska i vissa fall (ledtråd: prioritet)



Figur 1: Järnvägsalgoritmen med in-, ut- och stickspår.

operatorer som redan finns där köras till utgången, innan den inkommande operatorn körs ned på stickspåret. Symboler kan aldrig köras tillbaka från utspåret och inte heller köras tillbaka från stickspåret 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 några exempel på papper. Jämför era resultat med Linus i tabell på förra sidan.

3.3 Programmeringen

Nu har Linus och Linnéa 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 Linnéa 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:

- Dela upp inmatningen i tal och operatorer
- Överför från infix till postfix
- 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 i projektet. Högerklicka på ditt projekt i "projekt explorer", välj properties.

Klicka på Java Compiler, kryssa i "Enable project specific settings", välj "1.5" (eller "5.0" de syftar till samma sak) eller "1.6" i rullgardinsmenyn nedanför.

Klicka nu på Java build path till vänster, tabben libraries, Add external JARs, välj "/home/TDDC77/pub/LinCalcJohn.jar". (Alternativt, för er som

jobbar hemma, lägg till filen i ditt projekt först och välj sedan Ädd JARs".) Fortfarande i tabben Libraries, klicka på "add library", välj JUnit, välj 4 (OBS: version 3 kommer inte att fungera). Färdigt, stäng inställningsfönstren.

Skapa ett paket som heter lincalc (klicka på den bruna boxen med ett gult plustecken på).

I samma map som jarfilen finns även två javafiler. Den första heter LinCalc.java och innehåller ett kodskelett, den andra LinCalcTest.java innehåller testfall. Kopiera dessa till ert lincalc-paket.

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 Linnéa bestämmer sig för att först testa den givna miniräknaren med några längre matematiska uttryck. Kan ni hitta några fel i Johns program?

3.3.2 Steg 2 – Ta emot ett infixuttryck

Eftersom Linus och Linnéa bestämt sig för att de kan skriva ett minst lika bra program som John så börjar de med att ta emot ett infixuttryck. 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 siffror). 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 identifierar man unära minus? (Ledtråd: vad står innan?)

För att järnvägsalgoritmen ska fungera så finns det flera 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 siffror när de sedan konverteras till flyttal 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 Linnéa 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 järnvägsalgoritmen nu måste tas i bruk. Linnéa 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 Linnéa 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 återstår bara att byta ut metoden `LinCalcJohn.calc()` sen har Linus och Linnéa programmerat en hel miniräknare själva. Därför bestämmer de sig för att göra det var för sig för att vissa 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 kodkvalite

Ett viktigt mål vid programering 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 speciella kombinationer

indata som ni tror kan orsaka problem, men glöm inte testa med ”vanliga”indata. Här nedan finns exempel på hur testfall 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
- 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 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 emacs 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 lämplig 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 eller // 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.