Laboration 1

Objektorienterad Programmering (OOP)

Sektionen för IT
Högskolan i Borås
501 90 Borås

Patrick Gabrielsson
patrick.gabrielsson@hb.se
Rum: L436

Innehåll

1 Syfte

Laborationen behandlar grundstenarna i objektorienterad programmering (inkapsling, arv, polymorfism, metoder, attribut, klasser, objekt) samt generiska samlingsklasser, undantag och I/O i Java.

2 Uppgift

Ni har fått i uppdrag av en kund att skapa en simpel RPN-kalkylator, där RPN står för *Reverse Polish Notation* (https://en.wikipedia.org/wiki/Reverse_Polish_notation). Vad är då en RPN-kalkylator för något? Från matematiken är vi vana vid att använda artimetiska uttryck med *infix* notation. Exempelvis kan vi uttrycka summan av två tal, *A* och *B*, med infix notation som ”*A* + *B*”. I detta fallet är ”plus” *operatorn* +, ”fixerad inne i uttrycket” mellan dess två *operander A* och *B*. Därför kallar vi denna notationen för *infix*. På liknande sätt kan vi bilda ett ekvivalent matematiskt uttyck för summan av två tal med *prefix* eller *postfix* notation. Med prefix notation kommer operatorn före sina två operander ”+ *A B*” medans med postfix notation kommer operatorn efter sina två operander ”*A B* +”. Prefix notationen kallas för *Polish Notation* (*PN*) medans postfix notationen kallas för *Reverse Polish Notation* (*RPN*). Eran kalkylator kommer att använda *postfix* notationen.

Ovanstående Wikipedia-länk innehåller pseudokod för RPN-evalueringsalgoritmen, som använder en *stack* som datastruktur för att underlätta evalueringen av ett *postfix* uttryck. Ni skall också använda er av en *stack*, fast med en något annorlunda algoritm så att ni bättre kan öva er på objektorienterad programmering, vilket är huvudsyftet med denna laborationen.

Ni skall beräkna postfix uttryck med hjälp av en stack på följande sätt:

1. Uttrycket ”3 4 5 * -” läses från vänster till höger, dvs först ”3” och sist ”-”

2. Om man ”*splittar*” strängen på blankstegstecken, så fås följande symboler (*tokens*): (3, 4, 5, *, -). Dessa ”*pushas*” sedan på stacken i samma ordning som de läses:

   Här är alltså ”3” längst ner i stacken och ”-” längst upp i stacken. Observera att ni skall ”*pusha*” samtliga symboler (*tokens*) i en RPN-sträng till stacken.

3. Därefter poppas elementen från stacken en i taget medans beräkningar utförs. När stacken är tom, har det korrekta svaret beräknats.

    - pop ”-” (denna operatorn behöver två operander innan något beräknas)
    - pop ”*” (denna operatorn behöver två operander innan något beräknas)
    - pop ”5” (detta är ”*” operatorns andra operand)
    - pop ”4” (detta är ”*” operatorns första operand)
    - Nu har ”*” operatorn båda sina operander så följande beräknas: (4 * 5) = 20 (om vi ersätter ”4 5 *” i det ursprungliga uttrycket med ”20” får vi ”3 20 –”)
    - pop ”3 (detta är ”-” operatorns första operand)
    - Nu har ”-” operatorn båda sina operander (”20” är ”-” operatorns andra operand) så följande beräkning utförs (3 - 20) = -17
    - Eftersom stacken är tom, returneras resltatet -17

3 Kundens beskrivning av applikationen

Applikationens huvudklass skall heta ***Calculator*** och skall acceptera antingen 0 eller 2 kommandoradsväxlar (*command line arguments*). Om applikationen startas med 0 växlar, skall användaren presenteras med meddelandet ”*Ange RPN uttryck <retur> (tom sträng = avsluta):*” via standardströmmen för utmatning (*System.out*), varpå användaren matar in ett RPN-uttryck och trycker på <retur>-tangenten. Om uttrycket är giltigt, skall svaret presenteras till användaren via *System.out* enligt formatet: ”*Resultat: xx*” där *xx* är svaret. Om något fel uppstår under beräkningen av svaret, skall ett felmeddelande visas istället. I vilket fall som helst, så presenteras användaren återigen med meddelandet ”*Ange RPN uttryck <retur> (tom sträng = avsluta):*”. Denna loopen mellan inmatning av RPN uttryck och svar fortsätter tills användaren anger en tom sträng som RPN-uttryck, varpå applikationen avslutas med meddelandet ”*Användaren avslutade applikationen*”. En session skulle kunna se ut enligt nedan:

Ange RPN uttryck <retur> (tom sträng = avsluta): 3 4 5 * -
Resultat: -17.0
Ange RPN uttryck <retur> (tom sträng = avsluta): 3 4 5 w -
InvalidTokenException: w
Ange RPN uttryck <retur> (tom sträng = avsluta): 3 4 +
Resultat: 7.0
Ange RPN uttryck <retur> (tom sträng = avsluta):
Användaren avslutade applikationen

I ovanstående session startades applikationen med ”*java Calculator*”, dvs utan några kommandoradsväxlar. Därefter matade användaren in uttrycket ”3 4 5 * -”, vilket beräknades till 3 - (4 * 5) = 3 – 20 = -17 och visades för användaren. Nästa inmating innehöll ett ogiltigt tecken ”*w*”, vilket orsakade att undantagsfelet *InvalidTokenException* kastades. Därefter matades återigen ett giltigt uttryck in ”3 4 +” som beräknades till (3 + 4). Slutligen matade användaren in en tom sträng (dvs tryckte på <retur> utan att ange någon sträng), vilket fick applikationen att terminera med meddelandet ”*Användaren avslutade applikationen*”. Om applikationen startas med 2 giltiga växlar enligt ”*java Calculator input.txt output.txt*” så skall RPN-uttryck läsas in rad för rad från filen *input.txt* och resultaten skall skrivas till filen *output.txt*. Nedan anges ett exempel på källfil med tillhörande destinationsfil:

***input.txt* (källfil)**	***output.txt* (destinationsfil)**
3 4 +	7.00
3 4 -	-1.00
4 3 -	1.00
3 4 *	12.00
3 4 /	0.75
4 3 /	1.33
3 4 %	3.00
4 3 %	1.00
34 56 +	90.00
3 4 5 * +	23.00
3 4 5 * -	-17.00
3 4 + 5 *	35.00
3 4 - 5 *	-5.00
3 4 + 5 6 + *	77.00
3 4 5 + * 2 *	54.00
3 4.8 5.7 + * 2.5 *	78.75
100 2 4 / 5 6 + * -	94.50
239 100 4 2 / 5 6 + * - %	5.00
4 0 /	DivideByZeroException: 4.00/0.00
4 0 %	DivideByZeroException: 4.00/0.00
4 / 3	InvalidOperationException
A + B	InvalidTokenException: A
5 & 6	InvalidTokenException: &
1 2 +	3.00

På rad 5 och 6 nerifrån har undatagsfelet *DivideByZeroException* kastats eftersom vi försökte dela fyra med noll (4 / 0) samt försökte beräkna fyra modulus 0 (4 % 0). På rad 4 nerifrån har undantagsfelet *InvalidOperationException* kastats eftersom vi har använt *infix* notation (4 / 3) istället för *postfix* (4 3 /). På rad 2 och 3 nerifrån har undantagsfelet *InvalidTokenException* kastats eftersom t.ex. ”*A*” inte är en giltig operand (tal) och ”&” inte är en supportad aritmetisk operator (+ - * / %).

Åtminstone nedanstående aritmetiska operatorer skall supportas av applikationen:

- + (summan av två operander A och B, exempelvis ”A B +” = A + B)
- - (skillanden mellan två operander A och B, exempelvis ”A B -” = A - B)
- * (produkten av två operander A och B, exempelvis ”A B *” = A * B)
- / (kvoten mellan två operander A och B, exempelvis ”A B /” = A / B)
- % (modulus mellan två operander A och B, exempelvis ”A B %” = A % B)

Heltal och reella tal (positiva och negativa) skall supportas som operander.

Om applikationen startas med något annat än 0 eller 2 växlar, skall användaren presenteras med meddelandet ”*Syntax: java Calculator [källfil destinationsfil]*” via standardkonsollen för utmatning (*System.out*), varpå applikationen avslutas.

4 Arkitektur

Kunden vill att ni använder en MVC (*Model View Controller*) arkitektur (https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller).
I MVC arkitekturen separeras samtliga Java-klasser in i tre övergripande delar:

- **Model**
  klasser som har att göra med själva domänen (logik och data), som i detta fallet utgörs av själva kalkylatorn med sin data (stack) och logik (metoder) för att beräkna uttryck.

- **View**
  klasser som har att göra med användargränssnittet/presentationen i applikationen.

- **Controller**
  klasser vars syfte är att separera modellen (Model) från vyn (View) samt att driva själva applikationen.

Eftersom vi inte har behandlat designmöster ännu, innebär detta följande för er:

- Följande struktur på era Java paket (*packets*) skall användas
  1. Högsta nivån i eran applikations paketstruktur skall utgöras av paketet ***gruppXX*** där *XX* är erat gruppnummer.
  2. Paketet ***gruppXX.calculator*** skall innehålla eran huvudklass ***Calculator*** som i sin tur innehåller eran ***main*** metod.
  3. Paketet ***gruppXX.calculator.model*** skall innehålla samtliga Java klasser som har att göra med kalkylatorns logik och data (beräkningar och eran *stack*).
  4. Paketet ***gruppXX.calculator.view*** skall innehålla samtliga Java klasser som har att göra med användargränssnittet (inmatning av RPN-rader och presentation av resultat samt felmeddelanden).
  5. Paketet ***gruppXX.calculator.controller*** skall innehålla en klass ***Controller*** som driver själva applikationen. Dessutom ser denna klassen till så att model-klasserna är separerade från view-klasserna, dvs ingen model-klass skall ha en referens till en view-klass samt ingen view-klass skall ha en referens till en model-klass. Endast ***Controller*** klassen skall ha referenser till model-klasser och view-klasser.
  6. Givetvis får ni gärna utöka denna strukturen, exempelvis med ett paket för egendefinierade undantag t.ex. ***gruppXX.calculator.exceptions*** eller ett modell-subpaket för t.ex. operatorer och operander ***gruppXX.calculator.model.tokens***.

- Klassen ***Calculator*** i paketet ***gruppXX.calculator*** skall skapa en instans av klassen ***Controller*** samt anropa en metod som exempelvis heter ***run***.

- ***Controller*** klassen i paketet ***gruppXX.calculator.controller*** skall innehålla en metod, som exempelvis heter ***run***, som itererar igenom nedanstående loop tills antingen en tom sträng matas in av användaren eller samtliga rader i källfilen har lästs:

1. Loopa
2. Hämta nästa RPN-rad (från användaren eller källfilen)
3. Om RPN-raden är null eller en en tom sträng
4. Avsluta
5. Annars
6. Be kalkylatorn att beräkna resultatet av RPN-uttrycket
7. Presentera resultatet (till användaren eller destinationsfilen)

På så sätt driver *Controller* klassen applikationen samt skapar en separation mellan modell-klasserna samt vy-klasserna (detta underlättar, bland annat, underhåll av koden).

Själva RPN-kalkylatorn skall kunna ta emot ett RPN-uttryck i form av en sträng, splitta denna till ***Tokens*** (operatorer och operander) samt ”*pusha*” dessa på *stacken*. För att beräkna resultatet, ”*poppas*” sedan dessa från *stacken* medans de aritmetiska beräkningarna utförs. Resultatet returneras sedan till *Controller*n (punkt 6 ovan). Observera att eran *stack* endast skall känna till ***Tokens***! En ***Token*** kan i sin tur antingen vara en ***Operator*** eller ***Operand***, där en ***Operand*** är ett tal. En ***Operator*** i sin tur kan antingen vara *summationsoperatorn*, *differensoperatorn*, *produktoperatorn*, *kvotoperatorn* eller *modulusoperatorn* (samtliga operatorer är binära). Skapa en objektorienterad design för detta, där en viss operator känner till hur resultatet skall beräknas för dess två operander. Använd en genrisk samlingsklass (*generic collection*) för eran stack. Dessutom vill kunden lätt kunna byta ut implementationen för stacken i en framtida version av applikationen utan att ändra på dess publika interface.

5 Kravspecifikation

Designdokumentation gällande samtliga klasser skall lämnas in tillsammans med koden. Börja med att fundera igenom vilka klasser som skall ingå i lösningen samt ta fram specifikationer över klasserna samt de metoder som inte är triviala.

1. Metoden ***main*** skall ligga i klassen ***gruppXX.calculator.Calculator***. Denna klassen skall skapa en instans av ***gruppXX.calculator.controller.Controller***.
2. Programmet skall vara en applikation, naturligt uppdelad i klasser. Varje klass skall vara deklarerad i en egen fil. Arv och polymorfism skall utnyttjas.
3. När objekt skapas skall nödvändig information skickas med till konstruktorn.
4. Varje operator och operand skall ha en metod ***toString*** som returnerar en sträng som beskriver objektet.
5. Programmet skall innehålla felhantering så att det inte avbryts då fel uppstår. Använd egendefinierade Exception-klasser vid behov.
6. Alla ***Tokens*** för en RPN-rad skall pushas på *stacken* innan resultatet beräknas
7. De två användargränssnitten (en textbaserad och en filbaserad) skall supportas
8. Åtminstone de fem binära operanderna (+ - * / %) skall supportas
9. MVC arkitekturen skall följas och ***Controller***n skall innehålla en motsvarande ***run*** loop som beskrevs ovan för att driva själva applikationen
10. Implementationen för stacken skall lätt kunna bytas ut mot en annan i framtiden utan att ändra på dess publika interface

6 Övrig information

Arbeta i grupper om 2-3 studenter. Ni väljer grupp på pingpong (Innehåll/Gruppindelning).

6.1 Handledning

Handledning sker vid tre tillfällen enligt de tider som finns i schemat. Tänk på att komma till handledningen med väl förberedda frågor. Bokningslistor för handledningen finns i PingPong under Aktivitet/Innehåll/Handledning/Laboration 1. Där hittas Doodle länkar för varje handledningstillfälle (OBS! En Doodle länk för bokning av ett visst handledningstillfälle blir först synlig i PingPong dagen efter det förra handledningstillfället).

För varje handledningstillfälle (Handledning 1, 2 samt 3), boka endast **ett** handledningspass per grupp. Med reservation för ändringar (kolla alltid schemat), så gäller för närvarande nedanstående datum (och rum) för varje handledningstillfälle:

- Handledning 1 sker fredagen den 11 september i rum L433 13:00-15:30.
- Handledning 2 sker tisdagen den 15 september i rum L433 13:00-15:30.
- Handledning 3 sker måndagen den 21 september i rum L433 09:00-11:30.

6.2 Redovisning

Laborationen redovisas via PingPong, under Aktivitet/Inlämning/Laboration 1, där samtliga filer lämnas in som **en** arkivfil (zip-fil eller rar-fil). Följande filer skall finnas med i arkivfilen:

- Designdokumentation (UML klassdiagram)
- Javadoc (klassspecifikation)
- Källkod (beskriv relevanta metoder m.h.a. kommentarer i koden)
- En PDF (döp filen till *bidrag.pdf*) som listar vilka metoder varje gruppmedlem har skrivit. Varje metod skall dedikeras till en person.

Notera att det är varje enskild person ansvar att denna bidrar med relevant kod. Gruppen måste dock ge alla möjlighet att göra relevanta bidrag. Medlemmar som inte bidrar till laborationens lösning kommer underkännas även om gruppen blir godkänd. Uppstår det konflikter i gruppen skall detta tas upp med ansvarig lärare direkt och inte efter att laborationen lämnats in.

Inlämning och examination av laborationen sker vid tre tillfällen enligt schemat (OBS! En länk för ett visst inlämningstillfälle blir först synlig i PingPong dagen efter det förra inlämningstillfället). Samtliga tre tillfällen får utnyttjas. Om ni blir underkända på någon uppgift efter första och andra examinationstillfället skall denna vara åtgärdad innan nästa examinationstillfälle. Nedanstående datum gäller för varje examinationstillfälle:

- Deadline för inlämningstillfälle 1 är fredagen den 25 september 09:00.
- Deadline för inlämningstillfälle 2 är fredagen den 9 oktober 09:00.
- Deadline för inlämningstillfälle 3 är måndagen den 18 januari 09:00.

6.3 Betyg

Endast betygen Underkänd eller Godkänd förekommer på laborationen.