Prova på-laboration i Ruby

Johan Sjöholm
johsj@ida.liu.se
Institutionen för datavetenskap, Linköpings universitet
2008-08-06

1. Introduktion till objektorienterade språk

Programmeringsspråket Ruby började utvecklas under första halvan av 1990-talet av japanen Yukihiro "Matz" Matsumoto i syfte att skapa ett programmeringsspråk som balanserade imperativ och funktionell programmering, men samtidigt var ordentligt objektorienterat. Vad Ruby främst har kommit att användas till är så kallade domänspecifika språk och till webprogrammering, främst tack vare ramverket Ruby on Rails.

Ruby är som sagt objektorienterat och det innebär att man betraktar programmet som ett antal objekt som interagerar med varandra genom att skicka meddelanden. Istället för att, som i Ada, använda en extern procedur för att ändra programmets generella state, eller som i Lisp, skicka datan till en funktion som ändrar datan och returnera den, så ber man ett objekt, som datan är inkapslad i, att ändra på datan.

Objektorientering kan ses som en utbyggnad av imperativ programmering där varje objekt har sin egen state mer eller mindre oberoende av andra objekt och olika objekt påverkar varandra genom att skicka meddelanden. Alla objekt innehåller ett antal metoder, eller medlemsfunktioner, som själva tar hand om operationerna. För att ta ett enkelt exempel. Istället för att anropa en procedur som sorterar en lista, som i imperativ programmering, eller skicka en osorterad lista till en funktion som sorterar den, som i Lisp, så säger man till listan att sortera sig själv genom att anropa en metod inbyggd i själva listan.

Ett annat sätt man kan beskriva hur objektorienterade program är att medan imperativ och funktionell programmering kan beskrivas som en serie instruktioner till datorn, där alltså procedurerna och funktionerna står i centrum, så kan objektorienterad programmering beskrivas som olika data, olika objekt, som interagerar med varandra. Datan står alltså i centrum.

Klasser

Alla objekt i Ruby tillhör någon klass. En klass är snarast en beskrivning av hur ett objekt ska fungera, vad det ska innehålla och vilka metoder som ska finnas. När man skapar ett objekt av en viss klass säger man att objektet är en instans av den klassen. Till exempel är talet 12 en instans av klassen Integer i Ruby. Klasser är dock mer än de datatyper vi stött på i tidigare labbar, klasser är ett sätt att binda ihop data med de metoder som opererar på datan. En klass kan vara mycket enkel och innehålla en enda enkel rådata, som klassen Integer, eller

avancerad och innehålla många instanser av andra klasser i invecklade hierarkier. Ofta brukar ett program beskrivas som en huvudklass som binder ihop alla andra klasser, detta är speciellt sant i språk som Java där alla program måste vara i formen av en klass.

Metoder

I klassen skapar man så kallade metoder som är procedurer och funktioner som opererar på objekt av den klassen. Till exempel skulle en klass som hanterar heltal kunna ha metoder som returnerar talet i olika talbaser, till exempel i binär eller hexadecimal form eller metoder som returnerar absolutvärdet av talet. Det finns såväl klassmetoder som instansmetoder. Instansmetoder opererar på objekt medan klassmetoder opererar på klasser i sig.

Arv

I objektorientering är arv ett viktigt begrepp. Man säger att olika klasser ärver av varandra. Arv kan till en början vara lite svårt att förstå men när man väl förstått det kan det vara ett kraftfullt sätt att skapa abstraktion. Oftast brukar man introducera arv med någon form av exempel. Till exempel kan man tänka sig Klassen fordon som beskriver ett fordon och innehåller data om fordonets nuvarande hastighet och riktning samt metoder för att öka och sänka hastigheten samt ändra riktningen. Man kan tänka sig att man skapar klassen Lastbil som ärver av fordon, och således har tillgång till all data och alla metoder som fordon har, men som också innehåller data om antal släp, nuvarande last samt metoder för att koppla på och lossa släp, öppna och stänga lastluckor osv. Vi kan också skapa klassen Segelbåt som också ärver av fordon och innehåller data om antal segel, vilka segel som är uppe samt metoder för att hissa och ta ner segel med mera. Om en klass B ärver av klassen A säger man att B är en subklass till A och att A är en superklass till B.

Polymorfism

Starkt kopplat till arv är begreppet polymorfism som innebär att metoder med samma namn, och ofta samma funktionalitet, finns i flera klasser. I exemplet ovan med fordon har vi till exempel metoder för att ändra riktning. Att ändra riktning med en segelbåt är dock väldigt annorlunda från att ändra riktning med en lastbil och därför kommer samma metod behöva implementeras väldigt olika. Lastbilen behöver ha metoder som vrider på hjulen i en viss vinkel och under en viss tid för att ändra riktning medan segelbåten måste vrida på roder samt eventuellt ändra på segeluppsättningen. Om man däremot anropar metoden som ändrar riktning ska man inte behöva bry sig om dessa detaljer då objekten ska ta hand om det själva via sin implementation av metoden.

Förvirrande?

Mycket av detta kan verka förvirrande och svårt att förstå till en början men i övningarna nedan kommer ni få prova hur det fungerar, det är dock inte tänkt att ni ska förstå och kunna objektorientering bara genom att göra den här labben så bli inte avskräckta om ni inte förstår allt.

2. Exempel på Ruby-program

Ruby är ett så kallat script-språk vilket innebär att man mer eller mindre kan skriva en serie instruktioner uppifrån och ner i en fil (ett script) utan att innesluta dem i klasser eller procedurer. Nedan kommer just ett sådant script:

```
# Det här är en kommentar
# Programmet nedan skapar en array och adderar sedan ihop alla element
a = Array.new
a.push(1, 2, 3, 4)
resultat = 0
a.each {|x| resultat += x}
puts resultat
```

Ovanstående program, eller script, skapar ett objekt av klassen `Array` och lagrar i variabeln `a` genom att skriva `a = Array.new`. `Array` är en klass som innehåller en lista av andra objekt, varje objekt i en `Array` kallas för ett *element*.

För att skapa ett objekt av klassen `Array` anropar man klassmetoden `new` med hjälp av så kallad *punktnotation*. Punktnotation är ett vanligt sätt att anropa metoder i flera objektorienterade språk eftersom man tydligt ser vilket objekt, eller vilken klass, metoden man anropar opererar på.

Sedan anropar vi instansmetoden `push` med argumenten 1, 2 , 3 och 4, vilket innebär att dessa objekt stoppas in i `a`. Vi har nu en lista ser ut som följande, `[1, 2, 3, 4]`. Vi hade också kunnat skapa arrayen direkt genom att skriva `a = [1, 2, 3, 4]`.

Som synes behöver vi inte deklarera vilken typ variabeln `a` har efter som Ruby är dynamiskt typat, vi kan alltså stoppa in data av vilken klass som helst i vilken variabel som helst.

Sedan skapar vi variabeln `resultat` och tilldelar den värdet 0 genom att skriva `resultat = 0`. Efter det använder vi något som kallas en *iterator*. Iteratorer är ett speciellt sätt att iterera genom exempelvis `Array`-objekt. I Ruby finns ett antal olika iteratorer men den enklaste är `each`. Iteratorn each tar ett *block* som argument och applicerar det på varje element i `Array`-objektet.

Ett *block* är en konstruktion i ruby som kan beskrivas som en liten, icke namngiven, procedur. Den består av ett par *måsvingar* (”{ ”och ”}”) inom vilket det finns en *parameterlista* (parameter = variabel för indata) inneslutet mellan *pipes* (”|”) och sedan ett antal instruktioner.

När ett block används med iteratorn `each` kommer parametern, i det här fallet x, innehålla det aktuella elementet. Vad koden a.each {|x| resultat += x} gör är alltså att gå igenom `a` och addera varje element till resultat-variabeln.

Den sista raden, `puts resultat`, skriver ut resultat-variabeln på skärmen.

3. Att använda Ruby

Ruby är ett *interpreterat* språk vilket innebär att man till skillnad från i Ada inte behöver kompilera koden innan man kör den. För att köra kod i den här labben kommer vi använda terminalprogrammet `irb`. Programmet `irb` låter er skriva och köra Ruby-kod i en interaktiv miljö.
För att kunna använda ruby måste du först ladda in rätt modul, i terminalen skriver du:

```
$ module add prog/ruby
$ module initadd prog/ruby
```

För att starta `irb` skriver man helt enkelt `irb` i terminalen och får upp en prompt som ser ut ungefär så här:

```
irb(main):001:0>
```

I den här prompten kan man sedan skriva in Ruby-kod direkt:

```
irb(main):001:0> 1 + 3
=> 4
irb(main):002:0> puts "Sweet"
Sweet
=> nil
```

Här ser vi att det som returneras från körningarna skrivs ut efter symbolen => medan eventuella utskrifter som görs skrivs ut direkt. Man kan också ladda in kod från filer med Ruby-kod genom att skriva:

```
irb(main):003:0> load "rubyfil.rb"
=> true
```

Gör man så kan man sedan använda sig av klasser och metoder som finns i filen *rubyfil.rb*. Är filen ett script kommer det köras och resultatet skrivas ut. Om filen rubyfil.rb till exempel innehåller följande:

```
puts "Det funkar!"
```

Kommer resultatet bli:

```
irb(main):001:0> load "rubyfil.rb"
det funkar
=> true
```

Innehåller koden i filen fel returnerar load istället false:

```
irb(main):001:0> load "filmedfel.rb"
```

```
=> false
```

och man kommer då inte åt filens innehåll.

4. Övningar

De första två uppgifterna kan lösas helt i Ruby-tolken, i senare uppgifter rekomenderas att ni skapar en fil, exempelvis prova.rb, i vilken ni skriver era lösningar.

Övning 1 – Message passing och block

Att anropa metoder i ett objektorienterat språk innebär att man skickar säger åt objektet i fråga att köra en av sina metoder. Det kallas *message passing* och i Ruby finns ett antal sätt att göra detta. Det vanligaste och enklaste sättet är dock att anropa metoden med så kallad punktnotation [se avsnitt 2].

Skapa arrayen [2, 7, 4, 8, 1, 3] och sortera sedan arrayen med hjälp av metoden `sort`. Prova sedan att lägga till element med hjälp av metoden `push`. Lägg till och sortera med lite olika element och se vad som händer.

Använd sedan metoden `delete_if` för att ta bort objekt. Metoden `delete_if` är en form av iterator som tar ett block som returnerar ett sanningsvärde och tar bort alla element för vilket blocket returnerar true. Uttrycket

```
a.delete_if {|x| x == 1}
```

tar till exempel bort alla element som är lika med 1 ur `a`. Notera att vi använder två =-tecken för jämförelser eftersom ett enskilt används för att tilldela värden till variabler.

Kopiera och spara undan era testkörningar och deras resultat i en fil så att ni kan redovisa dem för er assistent.

Övning 2 – Polymorfism

I den här uppgiften ska ni prova på några olika inslag av polymorfism i Ruby.

Vi ska börja med +-operatorn. Den kan i Ruby användas för flera olika saker beroende på vilka objekt man använder den på. Prova följande

```
1 + 2                    #heltal + heltal
"Hello, " + "World!"     #sträng + sträng
[1, 2, 3] + [4, 5, 6]    #array + array
```

Prova också följande

```
"1 + 2 = " + 3
```

Vad händer? Fundera över varför. Prova också att anropa metoden `class` på ett heltal (Integer), en sträng (String) och en array (Array). Vad gör metoden? Prova sedan att anropa metoden `length` på en sträng och en array.

Övning 3 – Skapa en klass och två subklasser från grunden

Från den här uppgiften kan det vara lämpligt att skriva koden i någon texteditor (exempelvis Emacs) och sedan ladda in den i Ruby-tolken med `load` istället för att skriva in den direkt i tolken.

Skapa en ny klass `Pet` som har instansvariabeln `name` som ska innehålla namnet på ett husdjur. Klassen ska även innehålla metoderna `get_name` och `set_name` som ska returnera respektive ändra namnet på husdjuret.

För att skapa en klass i Ruby skriver man en så kallad klassdefinition. Inom klassdefinitonen skapar man sedan alla medlemsvariabler och metoder. Exempel:

```
class Klassnamn
     # Instansvariabler och metoder
end
```

Instansvariabler är variabler som ska finnas i varje instans av en viss klass och de skrivs på följande sätt: `@variabelnamn`. Snabel-a:t indikerar att det är en instansvariabel. Metoder skapas genom att man skriver en metoddefiniton. Inom metoddefinitionen skriver man sedan alla operationer som metoden ska utföra. En metod är i princip samma sak som en procedur eller funktion i Ada med den skillnaden att den är bunden till en viss klass. En metoddefinition ser ut såhär:

```
def metodnamn (parametrar)
     # Operationer
end
```

Om metoden ska returnera något värde så sätter man en return-sats i slutet av metoddefinitionen, på sista raden innan `end`. För att till exempel returnera en instansvariabel skriver man:

```
return @variabelnamn
```

Skapa sedan en klass `Dog` som har metoden `speak` som skrivet ut ”Woof!” på skärmen, klassen `Dog` ska ärva av `Pet`. Skapa också klassen `Cat` som också har en metod `speak`, vilken skriver ut ”Meow!” på skärmen, och ärver av `Pet`.

För att skapa en subklass lägger man till ett arv till klassdefinitionen med hjälp av <-tecknet. Exempel:

```
class Subklass < Superklass
          # Instansvariabler och metoder
end
```

Prova sedan att skapa ett `Dog`-objekt och anropa `set_name`, `get_name` och `speak` på det. Skapa ett `Cat`-objekt och gör samma sak.

Övning 4 – Konstruktorer och publika variabler

Skapa en klass `Person` med instanssvariablerna `name`, `age` och `gender`. Klassen ska ha en konstruktor som tar ett namn, en ålder och ett kön som input och skapar ett objekt med given input i instansvariablerna. Instansvariablerna `name`, `age` och `gender` ska gå att läsa och ändra utifrån utan att skapa explicita get- och set-metoder.

En konstruktor är den metod som anropas när man skriver Klassnamn.new, skapar man ingen egen konstruktor skapar Ruby en mycket enkel konstruktor själv. En klass kan ha flera olika konstruktorer som tar olika eller inga variabler och Ruby räknar själv ut vilken som ska användas baserat på vad `new` får för argument. Alla konstruktorer till en klass ska vara metoder med namnet `initialize` och dessa skapas precis som andra metoder.

För att göra instansvariabler möjliga att hämta och ändra utan speciella get- och set-metoder kan man använda `attr_accessor`-funktionen. Den fungerar på följande sätt:

```
attr_accessor :instansvariabelnamn1, :instansvariabelnamn2
```

Notera att vi måste använda kolon istället för snabel-a här.

*Övning 5 – Klasser i klasser

Skapa en klass `Family` med instansvariablerna `grown_ups`, `children` och `pets` som ska vara av klassen `Array`. Arrayerna ska sedan innehålla `Person`-, `Cat`- och `Dog`-objekt. Klassen ska ha metoder för att lägga till vuxna, barn och husdjur, det ska också finnas metoder för att ta bort familjemedlemar utifrån deras namn. Metoderna ska kontrollera att objekten man försöker lägga in är av rätt klass, till exempel att man inte försöker stoppa in ett `Person`-objekt bland husdjuren. Ni kan förutsätta att inga familjemedlemmar har samma namn.

För att testa att ett objekt tillhör en viss klass kan man använda metoden `is_a?` som finns för alla klasser i Ruby oavsett om de är inbyggda eller inte. Ni behöver alltså inte själva implementera den metoden. Metoden `is_a?` returnerar också sant om objektet är av en subklass till klassen man testar. Exempelvis är `Array` en subklass till klassen `Enumerable` vilket testas nedan. Metoden `is_a?` används på följande sätt:

```
irb(main):001:0> A = Array.new
=> []
irb(main):002:0> a.is_a?(Array)
=> true
irb(main):002:0> a.is_a?(Enumerable)
=> true
irb(main):002:0> a.is_a?(String)
=> false
```

I den här uppgiften kommer ni också behöva `if`-satsen. Den fungerar såhär:

```
if sanningsuttryck # exempelvis variabel.is_a?(String)
     # kod som körs om sanningsuttrycket är sant
else
     # kod som körs om sanningsuttrycket är falskt
end
```

*Övning 6 – Mer avancerade metoder

Skapa en `metod generate_shopping_list` i klassen `Family` som genererar en shoppinglista för familjen. För varje `Person` ska det stå ”Dinner for *personname*”, för varje `Dog` ska det stå ”Dog food for *dogname*” och motsvarande för varje `Cat`. För att infoga en radbrytning i en sträng lägger man till `\n`, som betyder newline, i strängen.

*Övning 7 – Mer avancerad abstraktion

Skapa en metod `pass_year` i klassen `Family` som åldrar alla `Person`-objekt i familjen med ett år. När ett barn åldras ska det påverkas på följande sätt. När barnet fyller 18 ska det vara en 50% chans att barnet flyttar hemifrån. Om barnet flyttar ska det tas bort från `children`. Om barnet bor hemma tills det är 24 ska det automatiskt flyttas ut när det fyller 25. Skriv metoder till klasserna `Family` och `Person` som hanterar de olika problemen i den här uppgiften. Försök tänka efter vad som borde göras i vilken klass.

För att slumpa fram ett tal använder man `rand` som tar ett heltal n som argument och slumpar fram ett tal bland de n första heltalen (inklusive 0). Exempel

```
irb(main):001:0> rand(3)
=> 0
irb(main):002:0> rand(3)
=> 1
irb(main):003:0> rand(3)
=> 1
irb(main):004:0> rand(3)
=> 0
irb(main):005:0> rand(3)
=> 2
irb(main):006:0> rand(3)
=> 1
```

För att testa något med 50% chans kan man alltså slumpa fram ett tal bland de två första heltalen (0 och 1) och undersöka vilket det blev.

*Övning 8 – Att ändra inbyggda klasser

Något som är häftigt i Ruby är att man kan ändra och bygga ut inbyggda klasser utan att skapa en ny subklass. Det är alltså möjligt att ändra på funktionaliteten i metoder som är inbyggda. Ta som exempel en `Array`. När en `Array` ska skrivas ut på skärmen anropas metoden `inspect`. Metoden `inspect` finns i klassen `Object` i Ruby och ärvs av alla andra klasser, den fungerar dock lite olika i olika klasser, dess syfte är dock alltid att returnera en utskriftsvänlig representation av objektet. Att vi ser en returnerad `Array` på skärmen som exempelvis [1, 2, 3] är tack vare `inspect`-metoden.

Vad ni ska göra är att ändra `inspect`-metoden så att den istället skriver ut en lista med ett element per rad. För att göra det gör ni en klassdefinition av `Array`-klassen och skapar metoden `inspect`. Ruby kommer automatiskt se till att den nya metoden ersätter den gamla under resten av programmets körning.

5. Referenser

Framtida kurser

Programspråket Ruby återkommer bland annat i kurserna *Konstruktion av datorspråk* och *Projekt: datorspråk* för IP-programmet. Kurser i objektorienterad programmering finns dock för de flesta programmen på LiTH, bland annat *Objektorienterad programmering och Java* för C- och D-programmen

Litteratur

Är du intresserad av Ruby kan man läsa mer på www.ruby-lang.org. Det finns också en ”standardbok” för Ruby som heter *Programming Ruby: The Pragmatic Programmers' Guide,* även kallad *Pickaxe* efter bilden på framsidan, vilken går att läsa på http://www.whytheluckystiff.net/ruby/pickaxe/. Språket Ruby tas även upp, tillsammans med många andra, i boken Concepts of Programming Languages där Ruby används som exempel på ett objektorienterat språk.

En lite mer annorlunda bok om Ruby är *why's (poignant) guide to Ruby* som är släppt under Creative Commons-licens och kan läsas på http://poignantguide.net/ruby/.

6. Frågor att fundera över

För dig som är nybörjare på programmering

Programmering är en process som består av flera faser. Först måste man förstå vad det är för problem som man ska lösa. Därefter ska man försöka designa en lösning. Denna lösning ska *implementeras*, dvs man ska skriva själva programmet. Sist men inte minst måste det färdiga programmet testas. Oftast går dock inte processen så här rakt och tydligt. Många gånger kan man tjäna på att experimentera lite, utan att ha förstått själva problemet. En del hävdar till och med att det är först när man har utformat lösningen som man verkligen har förstått problemet. För att bli en bra programmerare krävs lång träning – mycket längre än vad ens en högskoleutbildning kan ge. Man måste lära känna de olika byggstenarna som finns i programspråket, men också lära sig när och hur man ska använda dem.

Hur gjorde du när du löste övningarna i den här laborationen? Kan du känna igen de steg som beskrivs ovan? Känner du att du har förstått åtminstone lite grann av vilka byggstenar som finns i Ruby? Kändes det svårt att försöka formulera lösningar på problem i ett formellt programmeringsspråk?

För dig som programmerat en del förut

Liknar Ruby något annat språk som du har erfarenhet av? Vad känns svårare eller lättare att göra i Ruby, enligt din bedömning?