Norsk Ruby-Nuby - En introduksjon til språket Ruby

Kent Dahl < kentda@pvv.org > v1.0, Mars 3, 2002

En introduksjon til språket Ruby.

1 Bakgrunnsinformasjon

Kort om Ruby før vi titter på litt kode.

1.1 Hva er Ruby?

- et objekt-orientert programmeringsspråk
- et skript språk
- dynamisk ("svakt") typet
- lett å begynne med
- lesbart og samtidig enkelt å uttrykke seg i
- føles "kjent" (PoLS - Principle of Least Surprise)

1.2 Hvor kommer Ruby fra?

- Japan
  - Ruby større enn Python i Japan (sier ryktene)
  - 15(+) japanske bøker om Ruby
- Den geniale hjernen til Yukihiro Matsumoto ("Matz")
- En smeltedigel av features fra forskjellige språk:
  - Perl
  - Smalltalk
  - Python
  - CLU

1.3 Hvorfor Ruby?

- mer lesbart enn Perl.
- mer objekt-orientert enn Python og Perl.
- en tradisjonell og vanlig syntaks sammenlignet med f.eks. Smalltalk.

- tar de beste sidene ved flere språk og kombinerer dem.
- dynamisk/svak typing.

... og noen ting bare føles riktig!

1.4 Hvor finner jeg Ruby?

På *Ruby sin hjemmeside* `<http://www.ruby-lang.org/>` finner du kildekode, dokumentasjon til standard-biblioteket, bindinger til andre bibliotek og programmer, slides, etc.

Online engelsk bok `<http://www.rubycentral.com/>` om Ruby, skrevet av *The Pragmatic Programmers* `<http://www.pragmaticprogrammer.com/>`, som er veldig bra som introduksjon. Der finner du også en "Ruby for Windows" installer.

Ruby har også en mailingliste som blir speilet på nyhets-gruppen `comp.lang.ruby`

Den norske brukergruppen (Ruby User Group) heter *NorwayRUG* `<http://www.rubygarden.org/ruby?NorwayRUG>`. En kjekk ressurs når engelsk/japansk kunnskapene ikke strekker helt til.

Det siste året har det kommet flere engelske bøker om Ruby, bl.a.:

- *Ruby Developer's Guide* - Michael Neumann, Robert Feldt, Lyle Johnson
- *The Ruby Way* - Hal Fulton
- *Teach Yourself Ruby in 21 Days* - Mark Slagell
- *Ruby in a Nutshell* - Matsumoto Yukihiro

2 Vår første smakebit av Ruby-kode.

Eksemplene vises med linjenummer. (Disse er *ikke* en del av kildekoden.)
Det vil si, når det står

```
12| puts "Hello"
```

er det

```
puts "Hello"
```

som er kildekoden.

Kode-linjer er ofte annotert med en kommentar, `#=>`, som viser hva resultatet vil bli. (eller hva som skrives ut) For eksempel:

```
5 + 2 #=> 7
```

2.1 Hei verden!

Hello World begynner å bli gammel og tynnslitt, så her har vi en litt mer komplisert variant:

[code/heiverden1.rb]

```
# Ett klassisk eksempel
class Verden             # Definer en klasse...
  def si_hei             # med en metode...
    puts 'Hei verden!'   # som skriver ut litt tekst.
  end
end

verden = Verden.new      # Lag en instans av klassen.
verden.si_hei            # Kall en metode på objektet.
```

2.1.1 Kjøre heiverden1.rb

Det er flere måter å kjøre Ruby-kode på, bl.a.:

- Fra kommandolinjen:
  `ruby heiverden1.rb`
- Fra "Interactive Ruby" (IRB):
  `irb`
  irb(main):001:0> `load "heiverden1.rb"`
  Hei verden!
  true
- I Windows: Dobbelt-klikke på `heiverden1.rb` filen.
- I Unix: Gjøre skriptfilen kjørbar ved å legge til
  `#!/usr/bin/env ruby`
  på toppen av skriptet, og gjøre filen kjørbar med `chmod a+x heiverden1.rb`

2.2 Variabler

I Ruby har ikke en variabel noen statisk type. En variabel er bare et navn man forbinder med en referanse til et objekt. (Dette er den korte forklaringen på hva dynamisk typing er.)

[code/variabler1.rb]

```
x = 'Norge'  # Se, jeg er en tekst-streng (String).
x = [1,2,3]  # Øh, jeg mener en Array.
x = 5        # Ups, nå er jeg en Fixnum.

# Tilordninger kan lenkes
a = b = c = d = 5

```



```
# Du trenger ikke ekstra variable for å bytte to verdier
x = 5
y = 3
x, y = y, x  # Nå er x = 3 og y = 5

# Denne er kjekk å ha når en metode vil returnere flere verdier
a,b,c = ['a','b','c']

# Ruby bruker prefiks for å angi variabel skop
$global_variabel   = 'alle kan se meg!'
lokal_variabel     = 'sånn som x, y, a, b, c etc.'
@instans_attributt = 'jeg tilhører det gjeldende objekt.'
@@klasse_attributt = 'jeg er felles for mange objekter.'
KONSTANT_VARIABEL  = 'en selvmotsigelse?'
```

Attributter kalles også instansvariable eller felt.

Enhver variabel som begynner med stor forbokstav er konstant i Ruby. Konstant vil si at variabel-referansen er konstant, ikke at objektet ikke kan endre tilstand.

2.3 Tekststrenger

Vi kunne ikke skrive ut "Hei verden!" uten å ha litt tekst.
I Ruby er tekststrenger instanser av klassen String.

[code/tekst1.rb]

```
# String objekter kan instansieres
navn = String.new("Ruby") # veldig eksplisit, eller
navn = "Ruby"             # implisitt

# Tekst kan adderes...
fornavn   = 'Ola'
etternavn = 'Nordmann'
fullt_navn = etternavn + ', ' + fornavn
puts fullt_navn #=> "Nordmann, Ola"

#...multipliseres...
"Ah!" * 2 + " Tsjo!" #=> "Ah!Ah! Tsjo!"

#...manipuleres...
"Karakter fire".sub("fire", "en") #=>"Karakter en"
"14.99".to_f #=> 14.99

#...inspiseres...
"TEAMWORK".include?("I")  #=> false
"Kulturuke".index("tur")  #=> 3
```

Legg merke til at man kan skrive strenger både `"slik"` (med kråketær) og `'slik'` (med apostrof). Disse to versjonene er ikke bare for syns skyld.

2.3.1 Mer om String

String-klassen i Ruby fungerer ikke slik som Java sin String type, hvor man ikke kan endre et streng-objekt.

[code/tekst2.rb]

```
# String objekter kan endres:
navn = "Ola Nordmann"

# Kjønnsoperasjon?
nyttnavn = navn.sub("Ola", "Oline")

# sub lager ett nytt String objekt.
puts nyttnavn #=> "Oline Nordmann"
puts navn     #=> "Ola Nordmann"

# sub! (sub-bang) endrer selve streng objektet
nyttnavn = navn.sub!("Ola", "Oline")
puts nyttnavn #=> "Oline Nordmann"
puts navn     #=> "Oline Nordmann"
```

Legg merke til at sub generer et helt nytt String objekt, mens sub! endrer det faktiske objektet.
Metoder som slutter med utropstegn kalles **"bang"** eller **"destruktive"** metoder, da de som oftest endrer objektet.

Ruby bruker denne konvensjonen for navngivelse på mange metoder.

2.3.2 Skrive ut tekst

Det er mer enn en måte å flå en katt på, og der er også mer enn en måte å skrive ut tekst i Ruby.

[code/utput1.rb]

```
# Tekst kan skrives ut
navn = 'Kent'
puts 'Jeg heter ' + navn

# Liker du ikke puts?
print "Jeg heter " + navn + ".\n"

# Liker du C?
printf "Jeg heter %s!\n", navn

```



```
# Forelska i C++?
STDOUT << "Dahl... " <<
  navn << " Dahl.\n"

# Tekst-streng interpolering.
puts "Jeg heter fremdeles #{navn}"
puts "Jeg blir #{2002-1978} år i år"
```

Her ser vi hvorfor vi både bruker "kråketær" og 'apostrof' for å angi tekststrenger. Med "kråketær", kan vi escape tegn (slik som `\n` som betyr ny linje), og kjøre Ruby uttrykk inne i tekststrengen.

2.4 Tall.

Selv om tall også er objekter i Ruby, oppfører de seg normalt, og du må ikke vri hjernen din rundt sær syntaks.

[code/tall1.rb]

```
# Heltall
a = 3
b = 2
c = a + b    #=> 5
d = c / b    #=> 2

# Flyttall
f = c.to_f / b #=> 2.5

#Tall er også objekter
puts "tre og to er " + c.to_s
-25.abs     #=> 25
a.zero?     #=> false
0.zero?     #=> true
a.next      #=> 4
```

Ruby har flere tallklasser:

- Fixnum - begrenset heltall
- Bignum - ubegrenset heltall (forutsatt uendelig minne ;-)
- Float - flyt-tall med begrenset presisjon

Men du trenger ikke å tenke så mye på dem. Dersom du gjør noe med Fixnum som blir for stort, får du automatisk en Bignum ut. Du slipper å tenke på overflyt. Flyttall er derimot like unøyaktige som i de fleste språk og avrundingsproblematikk slipper du ikke unna.
For å lagre brøker, kan du bruke Rational-klassen. (`require 'rational'`)

to_s
: to string - lag en tekstlig representasjon av objektet, ikke ulikt Java sin `toString()`.

3 Flyt-kontroll

Logiske uttrykk skiller seg fra en del andre språk på et viktig område:
Kun `nil` og `false` evalueres til usant.
0, "", [], {} og andre 'tomme' objekter evaluerer til sant.
(For folk vant til C, Perl og/eller Python, er dette en av de vanligste kildene til feil.)

3.1 Hvis, dersom...

If-setningen byr ikke på så mange overraskelser i Ruby.

```
[ code/hvis1.rb ]
# Spør først om alderen.
print "Hvor gammel er du?: "
alder = gets.to_i

if alder < 1 then puts "Nå tuller du vel?"; exit end

if alder >= 18
  puts "Du er myndig."
elsif alder >= 16
  puts "Du er lovlig."
else
  puts "Småen!"
end

# 'if' kan også returnere en verdi, så du slipper
# å bruke '?:' operatoren hvis du ikke liker den.
drikkevare =
  if alder >= 60
    "Sviskejuice"
  else
    if alder >= 20
      "Sprit"
    elsif alder >= 18
      "Øl og vin"
    else
      "Brus"
    end
  end
puts "Kjøp deg litt #{drikkevare}"
```

Merk at `then` nøkkelordet ikke er nødvendig når man skriver if-setningen over flere linjer. Noen ting er valgfrie i Ruby forutsatt at uttrykket ikke blir tvetydig.

gets

get string - henter en tekststreng ifra standard input.

to_i

to integer - forsøker å gjøre objektet om til et heltall.

3.1.1 Forutsatt, med mindre...

Mulighetene til å skrive logiske uttrykk som er nærmere hvordan vi snakker, gjør Ruby mer lesbart, men kan også forvirre.

```
[ code/hvis2.rb ]
print "Liker du Ruby? [ja/nei]:"
svar = gets.chomp.downcase

# if kan også brukes etter uttrykk
puts "Jeg liker også Ruby!" if svar=="ja"

# 'unless' er det motsatte av 'if'
puts "La oss kode litt Ruby." unless svar=="nei"

# men bør brukes forsiktig
unless svar[0] == ?j
  puts "Mener du at du ikke liker Ruby?"
else
  puts "Doble negasjoner er forvirrende..."
end
```

chomp

fjerner et eventuelt newline-tegn i fra slutten av strengen.

downcase

gjør store bokstaver om til små.

?j

tallverdien til tegnet 'j'

Merk at chomp og downcase ikke har noen '!', så de returnerer kopier som har blitt modifisert. De endrer ikke objektet de blir kalt på.

3.1.2 Case

Ruby har også case-konstruksjonen, som ofte er et bedre valg enn en rekke elsif'er mot samme variabel.

```
[ code/case1.rb ]
print "Er du gutt eller jente?: "
svar = gets.downcase.chomp

```

```
# case er også kjent som switch/case i andre språk
case svar
when "intetkjønn"
  puts "Hei!"
when "jente", "kvinne", "dame"
  puts "Heisann søta!"
when "gutt", "mann", "herre"
  puts "Heisann kjekken!"
else
  puts "God dag herr/fru?"
end
```

Legg merke til at en **when** blokk kan slå ut på flere oppgitte verdier. Man kan også bruke regulære uttrykk, Range objekter, klasser etc. Du kan også lage dine egne objekter som kan brukes her ved å implementere === operatoren, også kalt "relationship operator". (Ja, det er 3 likhetstegn.)

3.2 Løkker - while

Hvis ikke du er fra Bergen, så har du vel falt for denne spøken en gang...

```
[ code/gjenta1.rb ]
# while - gjenta så lenge noe er sant
begin
  print "En gutt og ei jente satt i ett tre. \n"
  print "Så falt gutten ned. \n"
  print "  Hvem satt igjen? :> "
  svar = gets.chomp.downcase
end while svar.index("jenta")
```

index

gir indeksen til hvor i strengen det gitte argumentet finnes eller nil om det ikke finnes.

3.2.1 Until

```
[ code/gjenta2.rb ]
# Litt mer gøy med matte.
a = nil

# until - gjenta inntil noe blir sant
until a == 0
  print "Skriv inn ett tall (0 avslutter): "
  a = gets.to_i
  puts "#{a} opphøyd i 2      = " + (a**2).to_s
  puts "Kvaderatroten av #{a} = " + (Math.sqrt(a)).to_s
end
```

Merk at når løkke-uttrykket er foran koden som skal gjentas, droppes begin-nøkkelordet.

3.3 Iterasjon - Iterator-pattern og 'yield'

For løkken finnes fremdeles i Ruby, men hvor Python har gjort for-løkken glupere, har Ruby gått videre og tatt i bruk Iterator-pattern'et.

```
[ code/iter1.rb ]
# La oss skrive ut 3-gange-tabellen
tall = 3

# Ruby har for-løkker som de fleste språk
for i in (1..10)
  puts "#{i} gange #{tall} er #{i*tall}"
end

# 5-gange-tabellen
tall = 5

# men for-løkkens dager er talte.
# for-løkken over er syntaktisk sukker for
# følgende bruk av iterator-metoden each.
(1..10).each do |i|
  puts "#{i} gange #{tall} er #{i*tall}"
end
```

`(1..10)` lager et Range-objekt, som spenner i fra og med 1, til og med 10. Dersom du ikke ønsker å inkludere 10, kan du bruke 3 punktum, for eksempel så spenner `(1...10)` i fra 1, til og med 9.

3.3.1 Iterere over en datastruktur

Når man skal iterere over datastrukturer, blir indekser lett i veien.
Såkalte "off-by-one" feil er ganske vanlige.
Men hvorfor ikke la datastrukturen stå for itereringen?

```
[ code/iter2.rb ]
personer = [ "Ola", "Per", "Jan", "Line"]

# Den "gamle" naive måten
for i in (0...personer.size)
  puts "Hei " + personer[i]
end

# Alle objekter som implementerer 'each' kan itereres over
for person in personer
  puts "Er #{person} tilstede?"
end

```

```
# Ruby-måten: Bruke Iterator og en block
personer.each do |person|
  puts "Velkommen #{person}"
end
```

"Hva er den |person|greia?"
Det er nesten som en argumentdeklarasjon, men ikke til en funksjon.
`do |person|; end` er en **block**, et veldig viktig konsept i Ruby.

3.3.2 Yield og blocks.

En **block** er en kode-bit, som kan motta argumenter og returnere en verdi. Der slutter likhetene med en metode.

En block holder også tak i den omliggende konteksten og bindingen. Det betyr at lokale variable er tilgjengelig i block-koden, noe som gjør den perfekt til callback, f.eks. i grafiske brukergrensesnitt.

```
[ code/block1.rb ]
# En enkel, naiv iterator metode.
def tell_fingre
  yield "Tommel"
  yield "Peke"
  yield "Lange"
  yield "Ringe"
  yield "Lille"
end

# Blocken har tilgang til lokale variabler.
postfix = "finger..."

# Vi sender med en block når vi kaller iterator-metoden.
tell_fingre do |finger|
  puts finger + postfix
end
```

En block er ikke et objekt av effektivitetshensyn, men kan innkapsles i et Proc-objekt. (Via Proc.new, nøkkelordene proc og lambda, eller via bruk av & prefikset i argumentlista til metode-definisjonen)

Blocks for håndtering av ressurser Minne-håndtering i Ruby ordnes via garbage collection, men en del andre ressurser krever eksplisitt lukking. Åpne filer, database-tilkoblinger og andre ressurser som tar opp mer enn minne, har det bedre med en eksplisitt lukking. Men slikt glemmer man lett...

```
[ code/block2.rb ]
# Hent filnavn fra kommandolinjen.
filnavn = ARGV[0]

```



```
# Gamle måten.
fil = File.open( filnavn, "r" )
linjenummer = 0
fil.readlines.each{ |linje|
  linjenummer +=1
  print "#{linjenummer}: #{linje}"
}
fil.close   # Lukker filen eksplisitt

# Bruk block til ressurs styring.
File.open( filnavn, "r") { |fil|
  linjenummer = 0
  fil.readlines.each{ |linje|
    linjenummer += 1
    print "#{linjenummer}>: #{linje}"
  }
}
# File.open lukker filen etter å ha  kjørt koden i blocken.
```

Proc-objekter Et Proc-objekt innkapsler som oftest en block.

De kan lages via Proc.new, nøkkelordene proc og lambda, eller via bruk av & prefikset i argumentlista til metode-definisjonen.

```
[ code/block3.rb ]
# Lager ett Proc-objekt av en block.
p = proc {|i|
  puts "Hei #{i}!"
}

# Vi kan kalle Proc'en eksplisitt...
p.call('Jens') #=> "Hei Jens!"

# Bruke den som block...
[1,2,3].each( &p )

# & prefikset gjør en evt. block om til et Proc-objekt.
def tar_block( a, &block )
  block.call( a )
end

tar_block(5){|b|
  puts  "Hallo #{b}."
} #=> "Hallo 5."
```

4 Metoder, klasser og objekter.

Ruby er et særdeles objekt-orientert språk. Metoder, klasser og objekter er de grunnleggende byggesteinene.

4.1 Metoder

Ruby har verken prosedyrer eller funksjoner; kun metoder som kalles på objekter.

```
[ code/metoder1.rb ]
# Husker du denne?
def si_hei
  puts "Hei verden!"
end

# Hva er vel en funksjon uten argumenter?
def si_hei_til( hva )
  puts "Hei #{hva}"
end

si_hei_til("Trondheim!") #=> "Hei Trondheim!"

# Funksjoner kan ta flere argumenter og de kan ha default verdier
def send_julegave( til, fra="nissen" )
  puts "God jul, #{til}. Hilsen #{fra}."
end

send_julegave("Junior" )    #=> "God jul, Junior. Hilsen nissen."
send_julegave("Ola", "far") #=> "God jul, Ola. Hilsen far."
```

Du syntes kanskje det så lite objekt-orientert ut? Ikke var metodene definert i noen klasse, og ikke kalte vi dem på noe objekt heller. Det tror du. Alle metoder som defineres på toppnivå defineres i Object-klassen, og vi har implisitt en toppnivå Object-instans. (Prøv `self.type` og se selv.)

4.1.1 Retur-verdier

Ruby returnerer normalt den siste verdien i metoden, hvis ikke `return` kalles eksplisitt.

```
[ code/metoder2.rb ]
def legg_sammen( a, b)
  a + b   # det siste uttrykket returneres
end
puts legg_sammen( 9, 6 ) #=> 15

# fibonacci
def fib( i )
  if i <= 1
```



```
    return 1   # vi kan returnere eksplisitt
  end
  return fib( i-1 ) + fib( i-2 )
end
puts fib( 3 ) #=>  3
puts fib( 5 ) #=>  8
```

4.1.2 Spesielle argumenter

```
[ code/metoder3.rb ]
# * prefikset brukes for å pakke argumentlista inn i en Array
def list_opp( og_frase, *args )
  puts args[0..-2].join(", ").capitalize +
    og_frase + args[-1] + '.'
end

list_opp( " og ", "epler", "pærer", "bananer" )
  #=> "Epler, pærer og bananer."

# eller pakke opp en Array for å bruke elementene som argumenter
a =  [ " and ", "apples", "pears", "bananas" ]
list_opp( *a ) #=> "Apples, pears and bananas."
```

4.2 Klasser

Som ethvert objekt-orientert språk har Ruby klasser.

```
[ code/klasse1.rb ]
# En enkel klasse.
# Klassenavn må begynne med stor bokstav.
class Person
  # Person.new videresender argumentene til initialize
  def initialize( etternavn, fornavn, alder = 0 )
    # attributter prefikses med @
    @etternavn  = etternavn
    @fornavn    = fornavn
    @alder      = alder
  end

  # en vanlig instansmetode
  def to_s
    "#{@fornavn} #{@etternavn} er #{@alder} år."
  end
end

```

```
if __FILE__ == $0 # Kun når vi kjører denne filen:
  p = Person.new("Nordmann", "Ola", 23)
  puts p #=> "Ola Nordmann er 23 år."
end
```

4.2.1 Attributter

Ruby lar deg ikke få tak i et objekts attributter (felter, dataverdier, instansvariable) direkte. Alle attributter er "private". Enhver tilgang går via metodekall, såkalte get/set metoder.

[code/klasse2.rb]

```
# Vi vil bruke Person-klassen videre
require 'klasse1.rb'

# Klasser er "åpne" skop, og kan enkelt utvides.
class Person
  # Ruby tillater deg ikke å få tak i attributtene
  # fra utsiden av objektet. Du må gå via metodekall.

  #  get-metode
  def alder
    @alder
  end
  # set-metode
  def alder=( ny_alder )
    @alder = ny_alder
  end

  # tungvint? Jepp, så Ruby har en snarvei:
  attr_accessor :alder     # definerer metodene over automatisk

  # Vi vil også gjerne kunne lese navnene til personen
  attr_reader   :etternavn, :fornavn

end

if __FILE__ == $0 # Kun når vi kjører denne filen:
  p = Person.new( "Nordmann", "Baby" )
  p.alder = 3     # Vi setter alderen
  puts p.alder    #=> 3
  p.alder += 1    # Øk alderen med et år
  puts p.alder    #=> 4
  puts p.fornavn  #=> "Baby"
end
```


:alder, :etternavn, :fornavn

Disse er symboler. (instanser av Symbol-klassen) De ligner litt på String, men kan ikke endres, de er "internert" og begrenser seg til lovlige navn på klasser, metoder, variabler o.l. (Symbolet for instansvariabelen @alder, metoden alder og den lokale variabelen alder er alle sammen :alder.)

attr_accessor, attr_reader, attr_writer

Dette er metoder i Module-klassen som lager get/set metoder for deg. Som argument tar de symbolene til attributtene du vil lage get/set metoder for.

4.2.2 Arv

[code/klasse3.rb]

```
# Fortsetter der vi slapp...
require 'klasse2.rb'

# Arv - alle studenter er en submengde av alle personer
class Student < Person
  def initialize( etternavn, fornavn, alder = 0,
              studiested = "NTNU" )
    # kall super-klassens versjon av metoden
    super( etternavn, fornavn, alder )
    @studiested = studiested
    @karakterer = [] # Eventuelt Array.new
  end

  # redefinerer Person#to_s metoden
  def to_s
    "#{@etternavn}, #{@fornavn} - studerer ved #{@studiested}."
  end

  def ta_eksamen( karakter )
    @karakterer.push karakter
  end

  def karaktersnitt
    sum = 0
    @karakterer.each{ |karakter|
      sum += karakter
    }
    sum.to_f / @karakterer.size
  end

end

if __FILE__ == $0 # Kun når vi kjører denne filen:
  flinkis = Student.new("Einstein", "Al", 128, "Mensa")
  flinkis.ta_eksamen( 1.0 )
```

```
  flinkis.ta_eksamen( 2.0 )
  puts flinkis #=> "Einstein, Al - studerer ved Mensa."
  puts flinkis.karaktersnitt #=> 1.5
end
```

super

et alias for superklassens versjon av den metoden vi er i nå.

Multippel arv Beklager. Det er ikke lov å la en klasse arve fra mer enn en superklasse i Ruby.

Derimot har Ruby **mixin**, som kan legge til funksjonalitet fra flere moduler inn i en klasse. Dvs, du kan bare arve fra en klasse, men kan blande inn funksjonalitet i fra flere moduler.

```
[ code/modul1.rb ]
class Familie
  # Vi inkluderer funksjonalitet fra modulen kalt Enumerable.
  include Enumerable

  # Enumerable forventer at each metoden yield'er
  # det som skal itereres over.
  def each
    yield "Far"
    yield "Mor"
    yield "Sønn"
    yield "Datter"
  end
end

f = Familie.new

# include? og sort metodene er mikset inn fra Enumerable.
puts f.include?("Sønn") #=> true
puts f.sort.join(", ")  #=> "Datter, Far, Mor, Sønn"
```

I kontrast til Java tilbyr dette multippel arv av funksjonalitet. Java bruker **interface** til å "etterligne" multippel arv, men tilbyr ikke arv av implementasjon og løser noe som er et ikke-problem når man har dynamisk typing.

4.2.3 Klassevariabler

Klassevariabler (tilsvarende **static** variable i Java) er variabler som deles mellom alle instanser av klassen, samt instanser av sub-klasser.


```
[ code/klassevar1.rb ]
class Bil
  # En klassevariabel for å telle antall biler i verden.
  @@num_biler = 0
  def initialize
    @@num_biler += 1
  end
  def Bil.antall
    @@num_biler
  end
end

class Lada < Bil
end

class Yugo < Bil
  def krasj
    # klassevariabelen er felles for alle instanser av Bil,
    # samt instanser av subklasser av bil
    @@num_biler -= 1
  end
end

lada = Lada.new
yugo = Yugo.new
puts Bil.antall #=> 2

yugo.krasj
puts Bil.antall #=> 1
```

4.2.4 Død og begravelse: Finalize

Objekter fødes, brukes og dør. I Ruby dør objektene når de hentes av søppeltømmeren. (garbage collector) Når det skjer, er usikkert. Ingen referanser til objektet må eksistere og garbage collectoren må startes eksplisitt eller implisitt, f.eks. når det begynner å bli lite ledig minne.

```
[ code/gc1.rb ]
streng = "Hvil i fred."

# Vi gir en block som skal kjøres når streng objektet dør.
ObjectSpace.define_finalizer(streng){|id|
  puts "Objektet med ID=#{id} er nå dødt. "
  puts "Rest in peace."
}

```

```
# Starter søppeltømmeren eksplisitt.
puts "Henter søppel!"
GC.start
# Men ingenting skjer, da det ennå er en referanse til strengen.

# Prøver en gang til...
streng = nil
puts "Henter mer søppel!"
GC.start
# finalizer blocken blir kjørt.
```

Legg merke til at objektet allerede er dødt når finalizer block'en kalles. Ressurser som må lukkes eksplisitt, holdes via bindingen til block'en. (Bare pass på at bindingen ikke også holder en referanse til objektet.)

Flere finalizers kan registreres på et objekt.

5 Ting vi nesten glemte...

5.1 Exceptions: Feil og unntak. Når ting går galt.

For all sin glitrende magi, kan ikke Ruby beskytte deg fra å gjøre feil og heller ikke fra alt som kan gå galt.

```
[ code/unntak1.rb ]
$livvakter = true

def hent_kongen
  if $livvakter
    puts "Kanskje kommer Kongen..."
  else
    raise SecurityError, "Redd for bløtekake."
  end
end

begin
  hent_kongen #=> "Kanskje kommer Kongen..."
  $livvakter = false
  hent_kongen #=> "Niks: Redd for bløtekake."
rescue SecurityError => error
  puts "Niks: #{error}"
end
```

5.2 Regulære uttrykk

Regulære uttrykk (regular expressions) er en gjenganger blant skriptingsspråkene som gjør dem så perfekte til tekst-manipulering.

Klassen **Regexp** samler denne funksjonaliteten på en ryddig måte, men Ruby lar deg også bruke regexp literaler.

```
[ code/regexp1.rb ]
# Litt HTML tekst å lete i
html = '<UL>
<LI><IMG SRC="next.gif"></LI>
<LI><A HREF="index.html">Hei</A></LI>
</UL>'

# Den ryddige måten å lage Regexp'er på:
r1 = Regexp.new( '<IMG SRC=".*">' )
puts r1.match( html ).to_s #=> '<IMG SRC="next.gif">'

# Regexp-literaler
r2 = /<A HREF=".*">.*<\/A>/ # slash må escapes
puts r2.match( html ).to_s #=> '<A HREF="index.html">Hei</A>'

# String-klassen har også en del metoder som tar imot
# ett Regexp-objekt, deriblant sub og gsub
antall_e = 0
"Hvor mange e'er er det i denne setningen?".scan(/e/){ |match|
  antall_e += 1
}
puts "Totalt #{antall_e} e'er." #=> Totalt 9 e'er.
```

Regulære uttrykk er nesten en vitenskap i seg selv. Dessuten er de omtrent like lesbare som det norske lovverket etter å ha gått igjennom en makuleringsmaskin og bør derfor brukes med omhu.

6 Større eksempler

6.1 Gjettekonkurranse

Om du har hatt en Commodore 64, har du kanskje sett en variant av følgende spill i manualen. Målet er å gjette et tall med minst mulig forsøk, gitt litt tilbakemelding på hvor nær man gjettet.

```
[ code/gjett.rb ]
#!/usr/bin/ruby
class Gjett  # Et lite spill
  def initialize
     @max = 100
     @min = 1
     @forsoek = 0
     @ferdig = false
     @fasit_svar = ((@max-@min)*rand).to_i + @min
  end
```

```
  def spill
    puts "\nGjettekonkurranse!"
    while @forsoek < 10 and not @ferdig
      gjett( hent_svar )
    end
  end
  def gjett( svar )
    diff = (svar - @fasit_svar).abs
    if diff > 25
      print "Mye "
    elsif diff > 15
      print "Endel "
    elsif diff > 0
      print "Litt "
    end
    if svar > @fasit_svar
      puts "lavere. "
    elsif svar < @fasit_svar
      puts "høyere. "
    end
    if diff == 0
      @ferdig = true
      puts "Du klarte det på #@forsoek forsøk!"
    end
  end
  def hent_svar
    begin
      print "Gjett ett tall mellom #@min og #@max >:"
      svar = gets.to_i
    end until svar >= @min and svar <= @max
    @forsoek += 1
    svar
  end
end

# La oss unngå å bruke de kryptiske globale variabelnavnene.
require 'English'
if __FILE__ == $PROGRAM_NAME then
  begin
    g = Gjett.new
    g.spill
    print "Spille en gang til? [j/n]: "
    svar = gets.downcase
  end while svar[0] == ?j
end
```