Introduktion till Ruby

Vad är Ruby

Ruby är ett dynamiskt programmeringsspråk, distribuerat som öppen källkod med fokus på enkelhet och produktivitet.
Ruby skapades av Yukihiro "matz" Matsumoto och släpptes för allmänheten 1995 och idag är Ruby ett av de stora programmeringsspråken och popularitet bara ökar.
Det tog lite tid för folk utanför Japan för att upptäcka och arbeta med Ruby, delvis på grund av bristen på dokumentation.
Ruby är också en plattformsoberoende programmeringsspråk, vilket innebär att du kan utveckla och köra Ruby skript på Windows, Mac OS X, UNIX, Linux och många andra operativsystem.

Installera Ruby

Linux

Ruby finns till alla större Linuxdistributioner som förkompilerad paket.
För exempel att installera på Ubuntu används följande kommando:
sudo apt-get install ruby irb rdoc

Windows

För Windows finns det en "enklicksinstallerare" som installerar Ruby precis som vilket Windows program som helst. http://rubyinstaller.org/.

Mac OS X

Mac OS X levereras med Ruby förinstallerat.

Testa Ruby

Skriv in följande kommando i Terminalen *ruby —version*
Detta kommando kommer att bekräfta att Ruby har installerats genom att visa vilken version av Ruby som har installerats.
Nästa gång börjar vi lära oss programmera i Ruby.

Interactive ruby shell (IRB)

När installationen är klar är det dags att ta de första stegen in i Rubys värld.
Öppna terminalen och skriv in följande:

```
$> irb
>>
```

All inmatning i irb avslutas med returtangenten och det inskrivna kommandot utförs direkt.

Miniräknare

```
>> 2 + 3
=> 5
>> 4 - 3
=> 1
>> 30 * 2
=> 60
>> 20 / 10
=> 2
```

Alla utryck returnerar ett värde vilket visas efter =>.

Variabler
En variabel är ett namn som vi kan använda för att lagra information i och hämta den senare.

Exempel:

```
>> x = 10
=> 10
>> y = 20
=> 20
>> x + y
=> 30
```

x = 10 betyder att vi har skapat variabel x som innehåller talet 10 och variabeln y innehåller talet 20.
När vi skriver x + y kommer Ruby att använda talen som x och y refererar till och addera dem.

Några exempel till:

```
>>var1 = 2
=> 2
>>var2 = 3
=> 3
>> var3 = var1 + var2
=> 5
>> var4 = "Hej "
=> "Hej "
>> var4 = var4 * var3
=> "Hej Hej Hej Hej Hej"
```

Konstanter
Konstanter är variabler som inte ändrar sitt värde under programmets gång.
Du definerar konstanter precis som variabler, förutom att den första bokstaven är versal.

```
>> Stad = "Helsingborg"
=> "Helsingborg"
>> Stad = "Malmö"
(irb):19: warning: already initialized constant Stad
=> "Malmö"
>>
```

Arbeta med strängar

En sträng är en grupp av tecken, bokstäver eller siffror.
I Ruby skapar man en sträng med enkla citationstecken ' ' eller dubbla citationstecken " ".
Skillnaden mellan enkla och dubbla citationstecken är den att inom dubbla citationstecken kommer variablerna i strängen att bytas ut mot sitt värde.

```
>> "Summan av 33 + 2 är #{33+2}"
=> "Summan av 33 + 2 är 35"
>> 'Summan av 33 + 2 är #{33+2}'
=> "Summan av 33 + 2 är \#{33+2}"
```

Here Documents
Here-documents inleds med << och följs omedelbart av avgränsare.
Om du vill avsluta strängen skrivs avgränsaren ensam på en ny rad.

```
>> rubylove = <<EOS
Jag
älskar
att
programmera
i
språket
Ruby
EOS
=> "Jag\nälskar att\nprogrammera \ni\nspråket\nRuby\n"
```

Det går också att starta strängar med %-tecken.
%Q eller % följt av en avgränsare är samma sak som att använda dubbla citationstecken

```
>> rubylove = %{Jag
älskar att
programmera
i
språket
Ruby
}
=> "Jag\nälskar att\nprogrammera\ni\nspråket\nRuby \n"


>> rubylover = %Q{Jag
älskar att
programmera
i språket
Ruby}
=> "Jag\nälskar att\nprogrammera\ni språket\nRuby"

>> rubylover = %Q{3 + 2 är #{3 +2}}
=> "3 + 2 är 5"
```

%q skapar en sträng som om du hade använt enkla citationstecken.

```
>> rubylover = %q{Jag
älskar att
programmera
i språket
Ruby}
=> "Jag\nälskar att\nprogrammera\ni språket\nRuby"

>> rubylover = %q{3 + 2 är #{3 +2}}
=> "3 + 2 är \#{3 +2}"
```

Metoder för String-klassen

```
>> "ruby".capitalize
=> "Ruby"
>> "ruby".reverse
=> "ybur"
>> "Ruby".next
=> "Rubz"
>> "ruby".upcase
=> "RUBY"
>> "RUBY".downcase
=> "ruby"
>> "Ruby".swapcase
=> "rUBY"
=> "Ruby programmering".length
=> 18
>> "".empty?
=> true
>> "Ruby".empty?
=> false
```

Escape-tecken

Teckenkombination	Skrivs ut som
\'	'
\"	"
\\	\
\n	Ny rad
\t	Tab

Enkla utskrifter med puts och print

puts skriver till skärmen med radbrytning i slutet.
print gör samma sak fast utan radbrytning.

```
>> puts "Hej från Ruby"
Hej från Ruby
=> nil
>> print "Hej från Ruby"
=> Hej från Ruby=> nil
>> puts "Hej\nfrån\nRuby"
Hej
Från
Ruby
=> nil
```

nil är ett objekt precis som allt annat i Ruby och representerar ingenting.

```
>> x = 10            # variabeln x innehåller talet 10
=> 10
>> y = 2             # variabeln y innehåller talet 2
=> y
>> puts x + y # skriv ut resultatet av x + y
12
=> nil
>> puts "variabeln x innehåller talet #{x}"
x innehåller talet 10
=> nil
```

"gets"
Vi använder metoden gets för att få indata från användaren.

```
>> puts "Vad heter du?"
Vad heter du?
=> nil
>> namn = gets
Ruby
=> "Ruby\n"
>> puts "Hej " + namn
Hej Ruby
=> nil
```

Fler exempel

```
>> namn = gets
Ruby
=> "Ruby\n"
>> namn = gets.chomp  # chomp metoden ger dig tillbaka strängen fast utan radbrytningen
\n
Ruby
=> "Ruby"
>> tal = gets
20
=> "20\n"
>> tal = gets.to_i  # konvertera resultatet från gets till heltal
20
=> 20
>> tal = gets.to_f # konvertera resultatet från gets till flyttal
20
=> 20.0
```

Villkorssatser

Villkorssatser använder man så fort man måste hantera fler än ett alternativ.

If-satsen
If-satsen utför olika saker beroende på om ett villkor är sant eller falskt.

Exempel

```
>> name = "Ruby"
=> "Ruby"
>> if name == "Ruby"
>>    puts "Älskar Ruby"
>> end
Älskar Ruby
=> nil
```

Detta program skriver ut meddelandet eftersom variabeln name innehåller strängen Ruby.

```
>> name = "Rails"
=> "Rails"
>> if name == "Ruby"
>>    puts "Detta kommer inte att skrivas ut"
>> end
=> nil
```
Detta program skriver inte ut meddelandet eftersom name innehåller inte strängen Ruby.

If-else
Oftast vill man göra någonting om villkoret inte är sant.
Då använder man sig av en else-sats.

```
>> name = "Rails"
=> "Rails"
>> if name == "Ruby"
>>    puts "Ruby"
>> else
?>    puts "Ruby on Rails"
>> end
Ruby on Rails
=> nil
```
Om villkoret i if-satsen är falskt kommer alla satser mellan else och end att utföras.
elsif används för att lägga till fler än ett villkor.

Exempel
```
>> name = "PHP"
=> "PHP"
>> if name == "Ruby"
>>    puts "Ruby"
>> elsif name == "Rails"
>>    puts "Rails"
>> elsif name == "Merb"
>>    puts "Merb"
>> else
?> puts "Du har valt ett annat namn"
>> end
Du har valt ett annat namn
=> nil
```
Ett villkor i taget kommer att kontrolleras och när ett villkor är sant kommer det övriga inte att kontrolleras. Om inget av villkoren är sant kommer satsen i else att utföras.

= tilldelnings operatör
== lika meda == b
!= inte lika med a != b
> större äna > b
< mindre äna < b
>= större än eller lika meda >= b

```
>> 1 == 1
=> true
>> "a" != "b"
=>  true
>> "a" == "A"
=> false
>> 10 > 2
=> true
>> 2 < 10
=> true
>> 10 >= 10
=> true
>> 10  true
>> 10  true
```

Case
Vi använder en case-sats för att testa en sekvens av villkor.

```
>> name = "PHP"
=> "PHP"
>> case name
>> when "Ruby"
>>    puts "Hej Ruby"
>> when "Rails"
>>    puts "Hej Rails"
>> when "Merb"
>>    puts "Hej Merb"
>> else
?>    puts "Jag känner inte dig"
>> end
Jag känner inte dig
=> nil
```

case-satsen jämför värdet på en variabel eller resultatet av ett utryck mot flera alternativ.
case-satsen kan ha så många when jämförelser som du vill.

Intervallen (Ranges) är kanske det mest vanliga användningsområdet för case-satsen.
Intervall (Range) är en datatyp som representerar ett helt intervall med start och slut punkt.
Om två punkter används ingår både start-och slutvärde.
Om tre punkter används ingår startvärdet men inte slutvärdet.

```
>> 1..10  # skapar ett intervall från 1 till 10
>> 1...10  # skapar ett intervall från 1 till 9
>> betyg = 80
=> 80
>> resultat = case betyg
>> when 0..40: "Ej godkänt"
>> when 41..60: "Godkänt"
>> when 61..70: "Väl godkänt"
>> when 71..100: "Mycket väl godkänt"
>> else "Okänt"
>> end
=> "Mycket väl godkänt"
>> puts resultat
Mycket väl godkänt
>> nil
>>
```

Unless
Unless är motsatsen till if och koden utförs så länge satserna mellan unless och end är false eller nil.

```
>> tal1 = 10
=> 10
>> tal2 = 20
=> 20
>> unless tal1 == tal2
>>    puts "innehåller inte samma tal"
>> end
innehåller inte samma tal
=> nil
>> tal1 = 10
=> 10
>> tal2 = 10
=> 10

>> unless tal1 == tal2
```

```
>>    puts "innehåller inte samma tal"
>> end
```
Meddelandet i detta exempel skrivs inte ut eftersom unless är true båda talen innehålle samma tal.

Ternär operatorn
Ternär operatorn tillhandahåller en genväg för att fatta beslut beroende på om ett villkor är sant eller falskt. Syntaxen för ternära operatorn är följande:

[villkor] ? [true uttryck] : [false uttryck]

```
>> namn = "Ruby"
=> "Ruby"
>> namn == "Ruby" ? "Hej Ruby" : "Känner inte dig"
=> "Hej Ruby"
>> namn = "PHP"
=> "PHP"
>> namn == "Ruby" ? "Hej Ruby" : "Känne inte dig"
>> "Känner inte dig"
```

Introduktion till loopar och iterationer
Loopar i Ruby används för att utföra samma kodblock ett visst antal gånger.

While-satsen
While-satsen kommer att fortsätta upprepa koden så länge ett villkor är sant.

```
>> x = 1
=> 1
>> while x < 10
>>    print x
>>    x += 1
>> end
123456789=> nil
```
Så länge villkoret x < 10 är sant kommer satserna mellan while och end att utföras. Satsen x += 1 lägger till ett till variabeln.

Until-satsen
Until är nästan identisk med while förutom att until repeterar koden så länge villkoret är falskt.

```
>> x = 1
=> 1
>> until x > 10
>>   print x
>>   x += 1
>> end
12345678910=>nil
```

For-loopen
En for-loop används då antalet repetitioner är förbestämt.

```
>> for x in 1..10 do
?>   print x
>> end
12345678910=> 1..10
```
do är frivilligt att använda om inte koden är placerad på en enda rad.

```
>> for x in 1..10 do print x end
12345678910=> 1..10
```
Det går också att bryta ut sig ur en for-loop genom att använda break.

```
>> for x in 1..10
>>   print x
>>   break if x == 4
>> end
1234=>nil
```

Iteratorer
En iterator är en speciell metod som utför kod ett visst antal gånger.

Each
Det enklaste och mest använda iteratorn är each.

```
>> namn = ["PHP","Ruby","Rails","Merb"]
=> ["PHP","Ruby","Rails","Merb"]
>> namn.each do |x|
?>   puts "Hej #{x}"
>> end
Hej PHP
Hej Ruby
Hej Rails
Hej Merb
=> ["PHP","Ruby","Rails","Merb"]
```

Times,downto, upto
Times metoden är ett alternativ till for-loopen och används för att repetera kod ett visst antal gånger.

```
>> 3.times do
?>   print "Hej "
>> end
Hej Hej Hej => 3
```

Upto metoden fungerar ungefär som en for-loop.

```
>> 0.upto(10) do |x|
?>   print x
>> end
012345678910=> 0
```

Downto metoden liknar upto metoden undantaget är att det börjar vid ett värde och räknar ner snarare än upp.

```
>> 10.downto(5) {|x| print x, " "}
10 9 8 7 6 5 => 10
```

Arrayer
En array är en variabel som kan innehålla flera olika objekt.
Det finns många sätt att skapa eller initiera en array på.
Ett sätt är med klass metoden new:

```
>> namn = Array.new  # en tom array
=> []
>> namn = Array.new(4)
=> [nil, nil, nil, nil]
```

Arrayen namn har nu en storlek eller längd av 4 element.
Samtliga element är satta till objektet nil som representerar ingenting.
Du kan också skapa en array med hakparenteser:

```
>> namn = ['PHP','Ruby','Rails','Merb']
=> ["PHP","Ruby","Rails","Merb"]
```

Arrayen namn lagrar fyra strängar i nummerordning från 0 till 3.
En array kan även skapas med %w eller %W

```
>> namn = %w{PHP Ruby Rails Merb}
=> ["PHP","Ruby","Rails","Merb"]
>> namn[2]
=> "Rails"
```

För att hämta element ur arrayen namn används hakparenteser med index, där index 0 är första elementet.

```
>> namn[3]  # Fjärde elementet
=> "Merb"
>> namn[2]  # Tredje elementet
=> "Rails"
>> namn[-2] # näst sista elementet
=> "Rails"
>> namn[2,2] # två element från index 2
=> ["Rails","Merb"]
>> namn[1,3] # tre element från index 1
=> ["Ruby","Rails","Merb"]
```

Du kan skapa en array av siffror eller blandad med siffror och strängar:

```
>> siffror = [1,2,3,4,5]
=> [1,2,3,4,5]
>> blandad = [1,2,"Ruby",3,"PHP"]
=> [1,2,"Ruby",3,"PHP"]
```

Ändra ett element

```
>> namn = ["PHP","Ruby","Hej","Merb"]
=> ["PHP","Ruby","Hej","Merb"]
>> namn[2] = "Rails" # ändrar element 2 (Hej) till Rails
=> "Rails"
>> namn
=> ["PHP","Ruby","Rails","Merb"]
```

Det går att lägga ihop två arrayer genom att använda addition

```
>> siffror = [1,2,3]
=> [1,2,3]
>> siffror2 = [4,5,6,7]
=> [4,5,6,7]
=> tal = siffror + siffror2
=> [1,2,3,4,5,6,7]
```

Metoden delete raderar ett element ur arrayen

```
>> siffror = [1,2,3,4,5,6,7]
=> [1,2,3,4,5,6,7]
>> siffror.delete(7)
=> 7
>> siffror
=> [1,2,3,4,5,6]
>> namn = ["PHP","Ruby","Rails","Merb","Hej"]
=> ["PHP","Ruby","Rails","Merb","Hej"]
>> namn.delete("Hej")
=> "Hej"
>> namn
=> ["PHP","Ruby","Rails","Merb"]
```

Metoden sort används för att sortera arrayen.

```
>> namn = ["PHP","Ruby","Rails","Merb"]
=> ["PHP","Ruby","Rails","Merb"]
>> namn.sort
=> ["Merb", "PHP", "Rails", "Ruby"]
>> namn # Gamla namn arrayen
>> ["PHP","Ruby","Rails","Merb"]
>> namn.sort!  # skriver över namn
>> ["Merb", "PHP", "Rails", "Ruby"]
>> namn
>> ["Merb", "PHP", "Rails", "Ruby"]
```

Skapa flerdimensionella arrayer
En flerdimensionella array är en samling av arrayer.

```
>> multi_array = [[1,2,3,4],["a","b","c"]]
=> [[1,2,3,4], ["a","b","c"]]
```

multi_array är en multidimensionell array vars yttersta array har två delar. Den första delen multi_array[0] är en samling av fyra heltal och den andra delen multi_array[1] är en samling av tre strängar.
För att komma åt ett objekt i multidimensionella arrayer används två eller flera uppsättningar av hakparenteser.

```
>> multi_array[0][2]
=> 3
# hämtar element 3 från första delen av arrayen
>> multi_array[1][1]
=> "b"
# hämtar element 2 från andra delen av arrayen
```

Array metoder
array.at(index) Returner elementet vid index
array.clear Tar bort alla element från arrayen
array.collect Utför blocket för varje element
array.compact Returnerar en kopia med alla nil borttagna
array.compact! Tar bort nil element från arrayen. skriver över arrayen.
array.concat(array2) Lägger till array + array2
array.delete(obj) Tar bort obj ur array
array.delete_at(index) Tar bort elementet vid index.

array.delete_if{|item| block} Tar bort alla element med blockvillkor.
array.each Returnerar ett element i taget
array.each_index Samma som Array#each, returnerar ett index i taget till bifogat index
array.empty? Returnerar sant om array är tom
array.eql?(array2) Returnerar sant om array och array2 innehåller samma element
array.fetch Hämtar element ur array
array.fill Lägger element i array
array.first(n) Returnerar det första elementet eller den första element n.
array.flatten Returnerar en ny array som är en endimensionell med samma element som i array
array.flatten! Som flatten men skriver över array
array.frozen? Returnerar sant om array är fryst
array.hash Beräknar ett hash-kod för array.
array.include?(obj) Returnerar sant om obj finns i array, falskt annars
array.index(obj) Returnerar först index i array
array.replace Ersätter element i array
array.insert(index, obj…) Sätter in obj i array vid index i
array.inspect skapar en utskrivbar version av array
array.join sätter ihop element i array som en sträng
array.last(n) Returnerar det sista element(n) på array
array.length Returnerar antalet element i array
array.map Se collect
array.nitems Returnerar antalet element som inte är nil
array.pack Omvandlar array i en binär sekvens.
array.pop Tar bort det sista elementet ur array och returnerar det
array.push(obj,…) Lägger till obj sist i array
array.reverse Returnerar en ny array med arrays element i omvänd ordning
array.reverse! som reverse men skriver över array
array.reverse_each Samma som Array#each men itererar i omvänd ordning
array.rindex Samma som index men returnerar sista objektet i array
array.select Väljer ut element
array.shift Returnerar den första elementet,tar bort det och flyttar alla andra element ett steg nedåt.
array.size Samma som length, Returnerar längden på array
array.slice(start, slut) Returnerar element mellan start och slut
array.slice!(start, slut) Tar bort element mellan start och slut
array.sort Sorterar array
array.sort! Samma som sort men skriver över array
array.to_a Returnerar array
array.to_ary Returnerar array
array.to_s Returnerar array.join
array.transpose Förutsätter att self är en array och införlivar rader och kolumner
array.uniq! Tar bort dubbla element i array

array.unshift(obj,…) Sätter in obj i array och skjuter alla andra element ett steg upp

Symboler
Symboler är namn på t.ex. variabler, namn på metoder, namn på klasser.
Du behöver inte deklarera en symbol i förväg och en symbol är alltid unik.
Symboler är effektivare än strängar. Två strängar med samma innehåll är två olika objekt men för ett visst namn finns det bara en symbol objekt och detta kan spara både tid och minne.
En symbol inleds med ett kolon följt av symbolens namn:

```
>> symbol = :text
```

Detta skapar symbol-objektet med namnet text och tilldelas variabeln symbol.
Öppna din favorit texteditor och skriv in följande:

```
puts "Ruby".object_id
puts "Ruby".object_id
puts :ruby.object_id
puts :ruby.object_id
```

Spara filen som sym.rb och det är viktigt att texteditorn sparar texten som ren text och inte lägger till någon annan ändelse.

Öppna terminalen och skriv:
$> ruby sym.rb

Tryck retur och resultatet skrivs ut:

```
84410
84390
# båda strängarna har olika object_id och är separata objekt
102018
102018
# symbolerna har samma object_id och pekar på samma objekt
```

Ett annat exempel:

```
namn = :ruby
if namn == :ruby
  puts 'Hej Ruby'
else
  puts 'Du heter inte Ruby'
end
```

Spara filen som sym2.rb, öppna terminalen och skriv:

$> ruby sym2.rb
Utskriften blir
Hej Ruby

Hashtabeller
En hashtabell liknar en array men man använder index-nycklar istället för numeriska index som i arrayer.
Som arrayer finns det många olika sätt att skapa en hashtabell på.
Du kan skapa en hashtabell med klass metoden new

```
>> namn = Hash.new # skapar en tom hashtabell
=> {}
```
För att lägga värde i hashtabellen används nyckeln som index istället för en siffra

```
>> namn["stad"] = 'Helsingborg'
=> "Helsingborg"
>> namn
=>  {"stad"=>"Helsingborg"}
```

Du kan också skapa en hashtabell med {}:

```
>> namn = {}
=> {}
```
För att lägga värde i hashtabellen används nyckel/värde separerade med =>:

```
>> namn = {"Pspråk" => "Ruby","Stad" => "Helsingborg"}
=> {"Pspråk"=>"Ruby","Stad"=>"Helsingborg"}
```

För att hämta data ur en hashtabell används hakparenteser med nyckel som argument:

```
>> ht = {"Stad"=>"Helsingborg","Psprak"=>"Ruby"}
=> {"Psprak"="Ruby","Stad"=>"Helsingborg"}
>> ht["Stad"]
=> "Helsingborg"
>> ht["Psprak"]
=> "Ruby"
```

Ändra ett element

```
>> ht = {"Stad" => "Vet ej", "Psprak" => "Ingen"}
=> {"Stad"=>"Vet ej", "Psprak"=>"Ingen"}
>> ht.replace({"Stad" => "Helsingborg", "Psprak" => "Ruby"})
=> {"Stad"=>"Helsingborg", "Psprak"=>"Ruby"}
>> ht
=> {"Stad"=>"Helsingborg", "Psprak"=>"Ruby"}
```

replace ersätter innehållet av ht helt och hållet.

Metoden delete används för att ta bort ett värde ur en hashtabell.

```
>> ht = {"Stad" => "Helsingborg", "Psprak" => "Ruby"}
=> {"Stad"=>"Helsingborg", "Psprak"=>"Ruby"}
>> ht.delete("Stad")
=> "Helsingborg"
>> ht
=> {"Psprak"=>"Ruby"}
>>
```

Symboler används ofta i hashtabeller

```
>> ht = {:stad => "Helsingborg",:psprak => "Ruby",:alder => 26}
=> {:stad=>"Helsingborg", :psprak=>"Ruby", :alder=>26}
>> ht[:stad]   # Hämta data ur hashtabellen
=> "Helsingborg"
>> ht.delete(:alder) # ta bort värde ur hashtabellen
=> 26
>> ht
=> {:stad=>"Helsingborg",:psprak=>"Ruby"}
```

has_key? metoden returnerar true om den givna nyckeln finns i hashtabellen annars false:

```
>> ht = {:stad => "Helsingborg", :psprak => "Ruby"}
=> {:stad=>"Helsingborg", :psprak=>"Ruby"}
>> ht.has_key?(:stad)
=> true
>> ht.has_key?(:alder)
=> false
```

has_value? metoden returnerar true om värdet finns i hashtabellen annars false:

```
>> ht = {:stad => "Helsingborg", :alder => 26}
=> {:stad=>"Helsingborg",:alder=>26}
>> ht.has_value?(26)
=> true
>> ht.has_value?("Ruby")
=> false
>> ht.has_value?("Helsingborg")
=> true
```

En fullständig lista över Rubys Hash metoder hittar du här:
http://ruby-doc.org/core/classes/Hash.html

Metoder

Ruby metoder liknar funktioner i andra programmeringsspråk. Metod namn bör inledas med en liten bokstav. Om du börjar en metod namn med en versal kanske Ruby tolkar den som en konstant. Syntaxen för en Ruby metod är följande:

```
def metodnamn(arg1, arg2,..)
  Ruby kod
end
```

Exempel

```
# Enkel metod
>> def hej
>>   puts 'Hej'
>> end
=> nil
>> hej       # anropar metoden
Hej
=> nil

# Metod med ett argument
>> def hej1(namn)
>>   puts 'Hej ' + namn
>> end
>> hej1('Hayri')
```

Utskriften blir

```
Hej Hayri
```

```
nil
=> nil
```
Ruby tillåter oss att skriva metoder som accepterar varierande antal parametrar.

```
>> def antalparam(*args)
>>   args.each{ |x| puts x }
>> end
=> nil
>> antalparam("Ruby")
Ruby
=> ["Ruby"]
>> antalparam(1,"Ruby","Helsingborg")
1
Ruby
Helsingborg
=> [1, "Ruby", "Helsingborg"]
```

I Ruby kan man skapa alias för metoder, vilket skapar en kopia av en metod med ett annat namn:

```
>> def tal(n1,n2)
>>    summan = n1 + n2
>>    return summan
>> end
=> nil
>> alias nummer tal
=> nil
>> nummer(10, 20)
=> 30
>> tal(10,20)
=> 30
```

Klasser

I Ruby börjar en klass med nyckelordet class och slutar med en matchande end.

Detta är en simpel klass

```
class MinKlass
end
```

Så här skulle jag skapa ett användbart objekt från det:

```
objekt = MinKlass.new
```
För att göra MinKlass mer användbart måste jag ge det en metod eller två.
I detta exempel har jag lagt till en metod som kallas sag_hej:

```
class MinKlass
 def sag_hej
  puts "Hej"
 end
end
```
Nu när jag skapar en MinKlass objekt kan jag kalla denna metod för att få det att säga “Hej”:

```
objekt = MinKlass.new
objekt.sag_hej
```
När vi kör programmet blir utskriften:

```
$> ruby minklass.rb
Hej
$>
```

Instans Variabler
Instans variabler skapas för varje klass instans och är bara tillgängliga i den instansen eller genom de metoder som föreskrivs i denna instans. Instans variabler nås med hjälp av @-operatorn.

```
class MinKlass
 @ett = 1
 def gor_nagot
  @one = 2
 end

 def utskrift
  puts @one
 end
end

instans = MinKlass.new
instans.utskrift
instans.gor_nagot
instans.utskrift
```
När vi kör programmet blir utskriften:

```
$> ruby instans_var.rb
nil
```

```
2
$>
```

Klass variabler

En klassvariabel anges med @@-operatorn.Dessa variabler är förknippade med klassen snarare än ett objekt instans av klassen och är samma för alla objekt instanser.

```
class MinKlass
 @@klass_variabel = 1
 def lagg_till_ett
  @@klass_variabel = @@klass_variabel + 1
 end

 def resultat
  @@klass_variabel
 end
end
instansEtt = MinKlass.new
instansTva = MinKlass.new
puts instansEtt.resultat
instansEtt.lagg_till_ett
puts instansEtt.resultat
puts instansTva.resultat
```

Utskrifen:

```
$> ruby klass_variabel.rb
1
2
2
$>
```

Klassmetoder

Klassmetoder definieras genom att ange klassnamnet och en punkt före metodnamnet.

```
class MinKlass
  def MinKlass.min_metod
end
end
```

Ett annat sätt att skapa en klassmetod är att använda nyckelordet self.

```
class MinKlass
def self.min_metod
end
end
```

Nu kan du kalla min_metod via klassen antingen MinKlass.min_metod eller MinKlass::min_metod.

```
class MinKlass
 def MinKlass.min_metod
  puts 'Hej Hej'
 end
end

MinKlass.min_metod
```

Utskriften blir:

```
$> ruby klassmetod.rb
Hej Hej
$>
```

Arbeta med filer

Nya filer i Ruby skapas med new-metoden i File-klassen.
New-metoden tar två argument, det första är namnet på filen som ska skapas och det andra är det läge som filen ska öppna.

Här är en lista över olika sätt att öppna en fil på:
r Endast läsning
r+ Läs och skrivrättigheter.
w Endast skrivning.
w+ Läs och skrivning
a Endast skrivning vid slutet av filen.
a+ Läs-och skrivrättigheter. Skrivning läggs till vid slutet.
b Binär fil. Endast för Windows/Dos
Med denna information i åtanke kan vi därför skapa en ny fil i “write”-läge enligt följande.

```
>> File.new("test.txt","w")
=> #<File:test.txt>
```

Öppna filer
Med hjälp av open-metoden av File-klassen kan man öppna befintliga filer:

```
>> fil = File.open("test.txt")
=> #<File:test.txt>
```

Observera att befintliga filer kan öppnas i olika lägen som anges i tabellen ovan. Till exempel kan vi öppna en fil i skrivskyddat läge:

```
>> fil = File.open("test.txt","r")
=> #<File:test.txt>
```
Det är också möjligt att fastställa om en fil redan är öppen med closed? metoden:

```
>> fil.closed?
=> false
```
Slutligen kan vi stänga en fil med close-metoden:

```
>> fil = File.open("test.txt","r")
=> #<File:test.txt>
>> fil.close
=> nil
>> fil.closed?
=> true
```

Läsa och skriva filer

När vi har öppnat en befintlig fil eller skapat en ny fil måste vi kunna läsa från och skriva till den filen. Vi kan läsa rader från en fil med antingen readline eller each metoder:

```
>> fil = File.open("test.txt")
=> #<File:test.txt>
>> fil.readline
=> "Massor med text i filen"
>>
```
Med each-metoden kan vi läsa hela filen:

```
>> fil = File.open("test.txt")
=> #<File:test.txt>
>> fil.each { |x| print x }
Massor med text i filen
Ruby kurs för nybörjare
Ruby on Rails för nybörjare kommer snart
=> #<File:test.txt>
>>
```

Med metoden puts kan vi skriva en rad i taget till en fil.

```
>> fil = File.new("test.txt","w+")
=> #<File:test.txt>
>> fil.puts("Detta fungerar ju")
=> nil
>> fil.puts("Rad tva")
=> nil
```

```
>> fil.puts("Rad tre")
=> nil
>> fil.puts("Och rad fyra")
=> nil

>> fil = File.open("test.txt")
=> #<File:test.txt>
>> fil.each { |x| print x}
Detta fungerar ju
Rad tva
Rad tre
Och rad fyra
=> #<File:test.txt>
```