A Ruby programozási nyelv
Gyakorlat

Kovács Gábor

2019. február 12.

Amire szükségünk lesz a gyakorlat során: egy telepített Ruby környezet [1]. A Ruby programokat a `ruby` értelmező futtatja, aminek paramétere a futtatni kívánt szkript fájl.

A Hello, World! fájl felépítése az alábbi kódrészletben látszódik. Az első sornak UNIX rendszereken van jelentősége, ott ugyanis a futtatókörnyezet képes ez alapján kiválasztani fájl értelmezőjét különben ez csak egy komment. A második amúgy szintén komment sor a fájlban használt karakterkódolást specifikálja, mivel az alapértelmezett kódolás a 2.0-s verzió előtt az ASCII, ha ékezetes karaktereket akarunk írni a forrásba, ezt minden az elejére be kell szúrnunk. Ruby 2.0-tól UTF-8 lett az alapértelmezett. A gyakorlaton 2.3.3-as verziót használunk a félév során, ezért erre már nem lesz szükségünk. A fájl következő része azon függvénykönyvtárakat adja meg, amelyekből osztályokat kívánunk felhasználni ebben a fájlban. A függvénykönyvtár lehet egy Ruby modul vagy natív kiterjesztés. Ez körülbelül a C `#include`-nak vagy a Java `import`-nak felel meg. A fájl értelmezendő része a fájl végéig vagy az `__END__` sorig tart. Az utóbbi után a fájl még tartalmazhat belső feldolgozásra adatokat.

```
#!/usr/bin/ruby
# Encoding: UTF-8
require 'socket'

__END__
Itt még van valami
```

A gyakorlaton az `irb` értelmezőt használjuk a Ruby értelmező demonstrálására. A dokumentációt az `ri` paranccsal olvashatjuk, amelynek a megte-

[1] `http://www.ruby-lang.org/en/downloads/`

kinteni kívánt osztály (pl. `$ri String`) vagy metódus nevét kell megadnunk (pl. `$ri String.length`).

Írjuk meg mindjárt a szokásor Hello, world-öt. Ruby-ban a visszatérési érték az utolsó állítás, ami az interaktív értelmezőben a => szimbólum után látható.

```
kovacsg@debian:~> irb
irb(main):001:0> puts 'Hello, world!'
Hello, world!
=> nil
irb(main):002:0> 'Hello, world!'
=> "Hello, world!"
```

Programozási nyelvekben a következő lexikai elemekkel találkozhatunk: kulcsszavak, literálok, operátorok, azonosítók, üres karakterek és kommentek. A kulcsszavakkal menet közben ismerkedünk meg, az üres karakterek pedig a szokásosak: szóköz, tabulátor. A soremelésnek mint üres karakternek külön jelentése van, terminál egy állítást az értelmező számára.

Literálok Rubyban:

- egész szám (6. sor és 43. sor)
- lebegőpontos szám (7. sor)
- Ruby 2.0-tól létezik a racionális és képzetes literál is, amikkel komplex számokat írhatunk le (5. sor)
- string, amit megadhatunk ' vagy " szimbólumok között (8. és 9. sor)
- reguláris kifejezés (10. sor)
- szimbólum, ami nem más, mint a Ruby értelmezőn belül azonosítóként használt lebutított string, és amelyet hash-ekben előszeretettel alkalmazunk kulcsként (11. sor)
- tömb, ami egy szögletes zárójelek között megadott vesszővel elválasztott lista (12. sor)
- hash, ami kapcsos zárójelek között tartalmaz kulcs-érték párokat => vagy : szimbólummal elválasztva (13. sor)
- tartomány, ami a két szélső értékkel megadott egymás utáni egész értékek halmaza (14-15. sorok)
- boolean literálok (16-17. sorok)

- típus és érték nélküli literál (18. sor)

```
irb(main):005:0> (1+0i)
=> (1+0i)
irb(main):006:0> 1
=> 1
irb(main):007:0> 1.0
=> 1.0
irb(main):008:0> "String"
=> "String"
irb(main):009:0> 'String'
=> "String"
irb(main):010:0> /[a-z]/
=> /[a-z]/
irb(main):011:0> :sym
=> :sym
irb(main):012:0> [ 1, 2 ]
=> [1, 2]
irb(main):013:0> { "1" => 1, 1.0 => "egy" }
=> {"1"=>1, 1.0=>"egy"}
irb(main):014:0> 'a'..'f'
=> "a".."f"
irb(main):015:0> 'a'...'f'
=> "a"..."f"
irb(main):016:0> true
=> true
irb(main):017:0> false
=> false
irb(main):018:0> nil
=> nil
irb(main):043:0> 1_000_000_000_000_000_000_000_000.class
=> Bignum
```

Egysoros megjegyzést a `#` szimbólum után tehetünk. Többsoros megjegyzés tételére két módunk van, vagy folytatjuk a sor elejére tett `#` használatát, vagy `=begin` és `=end` közé helyezzük a figyelmen kívül hagyandó kódszészletet.

```
irb(main):001:0> # komment
irb(main):002:0> =begin
irb(main):003:0= akarmi
irb(main):004:0= =end
```

Ötféle – objektum– azonosítót különböztethetünk meg:

- konstans: nagybetűvel kezdődik
- (lokális) azonosító: kisbetűvel vagy _ szimbólummal kezdődik
- globális azonosító: $ szimbólummal kezdődik
- példányváltozó azonosító: @ szimbólummal kezdődik
- osztályváltozó azonosító: @@ szimbólumokkal kezdődik

A Ruby megkülönbözteti a kis- és nagybetűket. Konstans módosításakor figyelmeztetés kaptunk, akárcsak osztályváltozókhoz (ld. később) való jelöletlen hozzáférésnél.

```
irb(main):019:0> a = 1
=> 1
irb(main):020:0> Konstans = 2
=> 2
irb(main):021:0> Konstans = 3
(irb):21: warning: already initialized constant
   Konstans
(irb):20: warning: previous definition of Konstans was
   here
=> 3
irb(main):022:0> @a = 2
=> 2
irb(main):023:0> @@a = 2
(irb):23: warning: class variable access from toplevel
=> 2
irb(main):024:0> $a = 1
=> 1
```

Egy állítás jellemzően a következő soremelésig tart, kivéve, ha a sor utolsó lexikai eleme egy operátor, ami lehet az üzenetküldés operátor (. vagy ::) is, vagy épp egy utasításblokk közepén tartunk. Függvényhívások esetén konvenció szerint a . operátort használjuk, míg konstansok, osztálymetódusok (lásd később) elérésére, illetve névterek megkülönböztetésére a :: operátort. A Ruby operátorkészlete és azok precedenciái nagyrészt megegyeznek a C vagy Java operátorkészletével, azt néhány további operátorral kiegészítve. Érdekesség, hogy az egész számok osztásánál a Ruby nem 0 felé, hanem mindig mínusz végtelen felé kerekít.

```
irb(main):025:0> 1 + 2
=> 3
irb(main):026:0> 1 +
irb(main):027:0* 2
=> 3
irb(main):028:0> 1
=> 1
irb(main):029:0> + 2
=> 2
irb(main):030:0> 1 \
irb(main):031:0* +2
=> 3
irb(main):032:0> 3/2
=> 1
irb(main):033:0> -3/2
=> -2
irb(main):034:0> -(3/2)
=> -1
irb(main):035:0> 2**3
=> 8
irb(main):051:0> 'hello' =~ /h/
=> 0
irb(main):052:0> 'hello' =~ /k/
=> nil
```

Rubyban minden objektum, még az elemi típus literáljai is. Kétféle egész létezik `Fixnum` és `Bignum`. Objektumokon az üzenetküldés operátorokkal művelet végezhetünk. Azt, hogy egy objektum képes-e egy adott művelet végrehajtására a `respond_to?` függvénnyel kérdezhetjük le, amelynek a paramétere egy függvényazonosító, ami egy string.

```
irb(main):042:0> 1.class
=> Fixnum
irb(main):043:0> 1_000_000_000_000_000_000_000_000.class
=> Bignum
irb(main):044:0> 1.0.class
=> Float
irb(main):044:0> true.class
=> TrueClass
irb(main):045:0> (1+2i).class
=> Complex
```

```
irb(main):030:0> "egy".class
=> String
irb(main):120:0> a = 1
=> 1
irb(main):129:0> a.methods
=> [:%, :&, :*, :+, :-, :/, :<, :>, :^, :|, :~, :-@,
    :**, :<=>, :<<, :>>, :<=, :>=, :==, :===, :[], :
    inspect, :size, :succ, :to_s, :to_f, :div, :divmod,
    :fdiv, :modulo, :abs, :magnitude, :zero?, :odd?, :
    even?, :bit_length, :to_int, :to_i, :next, :upto, :
    chr, :ord, :integer?, :floor, :ceil, :round, :
    truncate, :downto, :times, :pred, :to_r, :numerator,
     :denominator, :rationalize, :gcd, :lcm, :gcdlcm, :+
    @, :eql?, :singleton_method_added, :coerce, :i, :
    remainder, :real?, :nonzero?, :step, :positive?, :
    negative?, :quo, :arg, :rectangular, :rect, :polar,
    :real, :imaginary, :imag, :abs2, :angle, :phase, :
    conjugate, :conj, :to_c, :between?, :f, :t, :
    instance_of?, :public_send, :instance_variable_get,
    :instance_variable_set, :instance_variable_defined?,
     :remove_instance_variable, :private_methods, :
    kind_of?, :instance_variables, :tap, :method, :
    public_method, :singleton_method, :is_a?, :extend, :
    define_singleton_method, :to_enum, :enum_for, :=~,
    :!~, :respond_to?, :freeze, :display, :object_id, :
    send, :nil?, :hash, :class, :singleton_class, :clone
    , :dup, :itself, :taint, :tainted?, :untaint, :
    untrust, :trust, :untrusted?, :methods, :
    protected_methods, :frozen?, :public_methods, :
    singleton_methods, :!, :!=, :__send__, :equal?, :
    instance_eval, :instance_exec, :__id__]
irb(main):130:0> a.respond_to? "+"
=> true
irb(main):131:0> a.class
=> Fixnum
```

Az azonosítók másik csoportja a függvény-, vagy másnéven metódusazonosítók. Függvényt a `def` kulcsszó után definiálhatunk a függvény azonosítója (36.sor), a formális paraméterlista és a törzs megadásával. A függvény törzse a `def`-fel egy szinten lévő `end`-ig tart (38.sor). A függvény azonosítója a konvenció szerint kisbetűvel kezdődik. Függvényhívásnál, ami nem

más mint egy üzenetküldés egy objektum számára, ameddig a paraméterlista egyértelműen meghatározható, a zárójelek elhagyhatók, így a függvényazonosítók után álló szóközökre különösen figyelnünk kell. A `defined?` operátor egy azonosítóról mondja meg, hogy az miként lett definiálva. Speciális függvényazonosító lehet a `!`-re végződő, ami azt jelenti, hogy a függvény megváltoztatja a hivatkozott objektum állapotát (valamely tagváltozójának értékét), a `?`-re végződő, ami azt jelöli, hogy kétértékű kifejezéssel tér vissza a függvény, és az `=`-re végződő, ez utóbbira az osztályok tárgyalásánál még visszatérünk.

```
irb(main):036:0> def f(n)
irb(main):037:1> n*2
irb(main):038:1> end
=> :f
irb(main):039:0> f(1+2)+3
=> 9
irb(main):040:0> f (1+2)+3
=> 12
irb(main):041:0> f 2
=> 4
```

Egy azonosító által reprezentált értékre `#{}` szintakszissal a kapcsos zárójelen belül hivatkozhatunk egy idézőjelben lévő stringben. A String sok tekintetben hasonlóan viselkedik, mint egy tömb. A karakterlánc karakterei tömbhozzáféréssel elérhetők, a negatív index a string végéről számol vissza, tartomány megadása esetén részstringet kapunk vissza.

```
irb(main):046:0> $a = 'lo'
=> "lo"
irb(main):047:0> "Hel#{$a}"
=> "Hello"
irb(main):048:0> 'Hel#{$a}'
=> "Hel\#{$a}"
irb(main):082:0> a[0]
=> "H"
irb(main):083:0> a[4]
=> "o"
irb(main):084:0> a[5]
=> nil
irb(main):085:0> a[-4]
=> "e"
irb(main):086:0> a[-33]
```

```
=> nil
```

A hash egy kulcs-érték párokat tartalmazó halmaz, leginkább a PHP asszociatív tömbhöz hasonlít. A Ruby tömb tetszőleges típusú értékekből összeállított lista, valójában egy hash, aminek az indexei automatikusan növelt egészek.

```
irb(main):087:0> arr = [ 1, 1.0, "egy" ]
=> [1, 1.0, "egy"]
irb(main):088:0> arr[0]
=> 1
irb(main):089:0> arr[-1]
=> "egy"
irb(main):090:0> arr * 2
=> [1, 1.0, "egy", 1, 1.0, "egy"]
irb(main):091:0> arr.join(':')
=> "1:1.0:egy"
irb(main):092:0> h = { "egy" => 1 }
=> {"egy"=>1}
irb(main):093:0> h["egy"]
=> 1
irb(main):094:0> h[1]
=> nil
irb(main):095:0> h[1.0] = 1
=> 1
irb(main):096:0> h
=> {"egy"=>1, 1.0=>1}
```

Speciális lexikai elem a tartomány, amely egész vagy karakter literálok egymás utáni elemeiből álló halmaz. A két ponttal definiált range a jobb oldali elemet is tartalmazza, a három ponttal definiált range a jobb oldali elemet már nem tartalmazza. A felsorolás típusokban (tömb, hash, tartomány) elérhető `each` metódussal ellenőrizzük, hogy ez valóban így van-e. Az `each` a felsorolás minden egyes elemére végrehajtja a függvényhíváshoz kapcsolt utasításblokkot, amelyet kapcsos zárójelek között vagy `do-end` kulcsszópár között helyezünk el.

```
irb(main):097:0> r = 'a'..'f'
=> "a".."f"
irb(main):098:0> r.each { |l| print "␣#{l}␣" }
 a  b  c  d  e  f => "a".."f"
irb(main):099:0> r = 'a'...'f'
=> "a"..."f"
```

```
irb(main):100:0> r.each { |l| print "␣#{l}␣" }
 a  b  c  d  e => "a"..."f"
```

Két boolean literál létezik a `true` és a `false`. A `nil` a nem definiált pointernek felel meg. A `false` `nil`-lé konvertálódik boolean kifejezésekben, e két értéken kívül minden más pedig `true`-vá.

```
irb(main):016:0> true
=> true
irb(main):017:0> false
=> false
irb(main):018:0> nil
=> nil
```

A Ruby kétféle vezérlési szerkezetet nyújt a programkód feltételes elágaztatására, amelyek csak szintakszisukban térnek el a C-ben megismertektől: `if`/`unless`, illetve `case`. Az `if` szerkezet formálisan:

```
if <feltetel> then
 <blokk>
{elsif <feltetel> then <blokk>}*
[else <blokk>]
end
```

. A `then` minden esetben helyettesíthető soremeléssel vagy pontosvesszővel karakterrel. Az `if` feltételének negálása helyett használható az `unless`. Az `if` és az `unless` blokkja kiemelhető a sor elejére, akkor azok őrfeltételként viselkednek. Többszörös elágazást a `case` szerkezettel hozható létre:

```
case <objektum>
{when <kifejezes> <blokk>}+
[else <blokk>]
end
```

Átfedő `when` értékek esetén az első illeszkedő ág hajtódik végre.

```
irb(main):057:0> if i > 1 then print "nagy" elsif i ==
  1 then print "egy" else print "kicsi" end
nagy=> nil
irb(main):058:0> if i> 1
irb(main):059:1> print "nagy"
irb(main):060:1> elsif i == 1
irb(main):061:1> print "egy"
irb(main):062:1> else print '
```

```
irb(main):063:1' kicsi'
irb(main):064:1> end
nagy=> nil
irb(main):065:0> print "nagy" if i > 1
nagy=> nil
irb(main):066:0> print "nagy" unless i > 1
=> nil
irb(main):067:0> i = 2
=> 2
irb(main):068:0> case i
irb(main):069:1> when 1
irb(main):070:1> print "egy"
irb(main):071:1> when 2..10
irb(main):072:1> print "nemegy"
irb(main):073:1> end
nemegy=> nil
```

Kétféle ciklus áll rendelkezésre: a `while`/`until`, ami megfelel a C while ciklusának, illetve a `for`, ami egy felsorolás (tömb, hash, tartomány) összes elemére hajtja végre a belső blokkot, és a más szkriptnyelvek `foreach` ciklusának felel meg.

```
irb(main):045:0> i = 1
=> 1
irb(main):049:0> while i < 3 do print "#{i=i+1}" end
23=> nil
irb(main):050:0> i = 1
=> 1
irb(main):051:0> until i > 3 do print "#{i=i+1}" end
234=> nil
irb(main):052:0> i = 1
=> 1
irb(main):053:0> print "#{i = i+1}" while i < 3
23=> nil
irb(main):054:0> 1..3
=> 1..3
irb(main):055:0> for i in 1..3 do print "#{i}" end
123=> 1..3
irb(main):056:0> i = 2
=> 2
```

Ruby-ban az azonosítók referenciaként viselkednek. Típusuk nem dekla-

ráció során, hanem az első használatkor dől el. Ha egy objektumhoz több referenciát rendelünk, akkor annak értéke referencián keresztül megváltoztatható. Az objektumok egyenlősége vizsgálható érték szerint (==) és referencia szerint (equal?) metódus. Az eql? metódus érték szerinti egyenlőséget vizsgál azonban nem végez típuskonverziót. A === operátor case ágaiban való illeszkedést vizsgál, alapértelmezés szerint úgy működik, mint a == operátor.

```
irb(main):115:0> a
=> "Hello"
irb(main):116:0> b = a
=> "Hello"
irb(main):117:0> b
=> "Hello"
irb(main):118:0> a.reverse!
=> "olleH"
irb(main):119:0> b
=> "olleH"
irb(main):120:0> a = 1
=> 1
irb(main):121:0> b = 1.0
=> 1.0
irb(main):122:0> a == b
=> true
irb(main):123:0> a.eql? b
=> false
irb(main):124:0> a.equal? b
=> false
```

Az objektumok konvertálhatók más típusúvá. A konverzió történhet automatikusan, vagy explicit módon a to_s, to_i vagy to_f stb. metódusokkal. A nem nil és nem false objektumok feltételes kifejezésekben true értékké konvertálódnak, míg a két megnevezett objektum false értékké.

```
irb(main):125:0> 1.to_f
=> 1.0
irb(main):126:0> 1.to_s
=> "1"
irb(main):127:0> "1".to_f
=> 1.0
```

Mivel egy azonosító bármilyen típust jelenthet, nem lehetünk mindig biztosak abban, hogy az adott objektumra vonatkozóan egy-egy metódus értelmezett-e. Ekkor futás közben a respond_to? metódussal állapítható

meg, hogy kaphatunk-e választ egy hívásra vagy sem. Fejlesztési időben ugyanebben a dokumentáció segíthet nekünk, amelyet az `ri` paranccsal érünk el konzolon, például `ri String`.

```
irb(main):154:0> a.respond_to? '+'
=> true
irb(main):155:0> "string".respond_to? '+'
=> true
```

Blokkot kétféle szintakszissal definiálhatunk: vagy `do-end` párok között vagy kapcsos zárójelekben. A blokkoknak speciális szerepük is lehet Rubyban. Függvényhívásoknak paraméterül adható egy procedurális blokk, ami akkor hívódik meg, ha a függvény törzse anonim esetben `yield`, nevesített esetben `block.call` sorhoz érkezik (a `block` itt a blokk nevesített azonosítója).

A 100. sorban (feljebb) bemutatott tartomány típus `each` metódusa is ilyen. A 101-104. sorokban a `t` metódus definíciója meghívja a metódushívás paramétereként a 106. sorban átadott blokkot. A blokkban definiált procedúra paraméterezhető, ahogy azt a 107-109. sorok mutatják. A 110. sorban meghívjuk a 107. sorban definiált `t` azonosítójú függvényt egy string paraméterrel. A függvény törzsében, vagyis a 108. sorban átadja a vezérlést a 110. sor blokkja törzsének átadva az `a` értéket. A 110. sor procedúrája a paraméterül kapott értékre a lokális `l` azonosítóval hivatkozik, amely a törzsben felhasználható. A törzs végével a vezérlést visszakapja a hívott metódus, vagyis a végrehajtás a 108. sorban, a `yield` után folytatódik. A `yield`-et tartalmazó metódushívás blokk argumentuma nem hagyható el, lásd 105. sor! Egy függvénynek így egy blokk adható át paraméterül. A blokknak tetszőleges számú paramétere lehet, azokat vesszővel elválasztott listában kell megadnunk a `|` jelek között.

```
irb(main):101:0> def t
irb(main):102:1> yield
irb(main):103:1> yield
irb(main):104:1> end
=> :t
irb(main):105:0> t
LocalJumpError: no block given (yield)
        from (irb):102:in `t'
␣␣␣␣␣␣␣␣from␣(irb):105
␣␣␣␣␣␣␣␣from␣/usr/bin/irb:11:in␣`<main>'
irb(main):106:0> t { print "a" }
aa=> nil
```

```
irb(main):107:0> def t(a)
irb(main):108:1> yield a
irb(main):109:1> end
=> :t
irb(main):110:0> t('hello') do |l| print "#{l}" end
hello=> nil
irb(main):111:0> def t(a)
irb(main):112:1> yield a*3
irb(main):113:1> end
=> :t
irb(main):114:0> t('ho') do |l| print "#{l}" end
hohoho=> nil
```

A Ruby programnyelv objektumorientált, építőkockái az osztályok és a modulok, amelyek egymással a nyilvános felületükön definiált metódusaikkal kommunikálnak egymással. A 140. sorban a `Math` modul `sqrt` metódusát hívjuk meg, a híváskor a `.` vagy a `::` operátort használhatjuk. Az üzenetek egymásba ágyazhatóak.

```
irb(main):128:0> Math::sqrt 4
=> 2.0
irb(main):141:0> class A
irb(main):142:1> end
```

Az osztály közös tulajdonságokkal és viselkedéssel bíró objektumokról képez mintát. A Ruby osztály egységbe zárja a tulajdonságokat (attribútumok), és a rajtuk végzett műveleteket, a viselkedést (metódusok), bár az utóbbiak nem érhetők el az osztály példányának referenciáján keresztül.

A 133-137. sorban (fent) egy `A` azonosítójú osztályt definiálunk, amelyet a 138. sorban példányosítottunk. A típusok azonosítóit Ruby-ban konvenció szerint nagybetűvel kezdjük. Az osztály a 134-136. sorban lévő `m` azonosítójú metódusa összeadja a két paraméterül adott két számot. Ha egy osztálydefiníció során egy már létező osztály azonosítóját adjuk meg, akkor az a definíció kibővíti vagy felüldefiniálja az osztály viselkedését. A 141-143. sorban bővítjük, a 147-149. sorban felüldefiniáljuk a viselkedést. A 138. sorban létrehozunk egy példányt az implicit ősosztályból, vagyis az `Object`-ből módon örökölt `new` metódussal, ekkor láthatjuk az objektum memóriabeli címé. A 139. és a 152. sorban meghívjuk az `m` metódus két változatát ugyanazokkal a paraméterekkel.

```
irb(main):133:0> class A
irb(main):134:1> def m(p1,p2)
irb(main):135:2> p1 + p2
```

```
irb(main):136:2> end
irb(main):137:1> end
=> :m
irb(main):138:0> a = A.new
=> #<A:0x0056512e7224c0>
irb(main):139:0> a.m 2,3
=> 5
irb(main):140:0> class A
irb(main):141:1> def initialize
irb(main):142:2> @a = 2
irb(main):143:2> end
irb(main):144:1> end
=> :initialize
irb(main):145:0> a = A.new
=> #<A:0x0056512e705e38 @a=2>
irb(main):146:0> class A
irb(main):147:1> def m(p1,p2)
irb(main):148:2> @a + p1 + p2
irb(main):149:2> end
irb(main):150:1> end
=> :m
irb(main):151:0> a = A.new
=> #<A:0x0056512e674578 @a=2>
irb(main):152:0> a.m 2 , 3
=> 7
```

A kezdeti viselkedést az `initialize` metódus (felül)definiálásával határozhatjuk meg. A 141-143. sorokban az `A` osztály `@a` azonosítójú példányváltozóját 2-re állítjuk be. A `@a` példányváltozót nem kell külön deklarálnunk, az az első hivatkozás hatására létrejön. A `@a` példányváltozó nem férhető hozzá kívülről, azonban az osztályon belül használható, ahogy azt a 147-149. sorban felüldefiniált `m` metódusban megtesszük.

A példányváltozókhoz úgy nevezett setter és getter metódusokkal férhetünk hozzá. A setter jellemzője, hogy a változó azonosítója mögé egy egyenlőségjelet írunk (154-156. sor), a getter azonosítója pedig maga a példányváltozó azonosítója `@` szimbólum nélkül (157-159. sor). Használatukat a 162-164. sorok mutatják. A 161. sor visszatérési értékében láthatuk, hogy a példányváltozó létrejött az első használatkor, vagyis a `initialize` metódus lefutott.

```
irb(main):153:0> class A
irb(main):154:1> def a=(val)
```

```
irb(main):155:2> @a = val
irb(main):156:2> end
irb(main):157:1> def a
irb(main):158:2> @a
irb(main):159:2> end
irb(main):160:1> end
=> :a
irb(main):161:0> a = A.new
=> #<A:0x0056512e6c8a60 @a=2>
irb(main):162:0> a.a = 3
=> 3
irb(main):163:0> a.m 2 , 3
=> 8
irb(main):164:0> a.a
=> 3
```

A Ruby egyszerűbb módot is nyújt a setterek, getterek létrehozására. Az `attr_accessor` mind a settert, mind a gettert automatikusan létrehozza a paraméterül adott szimbólum string reprezentációjának megfelelő azonosítóhoz, az `attr_reader` csak a gettert, az `attr_writer` csak a settert hozza létre. Az üzenetek létrejöttének igazolását, illetve hiányát a 166-176. sorok mutatják.

```
irb(main):166:0> class A
irb(main):167:1> attr_accessor :b
irb(main):168:1> attr_reader :c
irb(main):169:1> attr_writer :d
irb(main):170:1> end
=> nil
irb(main):171:0> a = A.new
=> #<A:0x0056512e4e3ab0 @a=2>
irb(main):172:0> a.b = 2
=> 2
irb(main):173:0> a
=> #<A:0x0056512e4e3ab0 @a=2, @b=2>
irb(main):174:0> a.c
=> nil
irb(main):175:0> a.c=3
NoMethodError: undefined method `c=' for #<A:0
    x0056512e4e3ab0 @a=2, @b=2>
Did you mean? c
        from (irb):175
```

```
        from /usr/bin/irb:11:in `<main>'
irb(main):176:0> a.d
NoMethodError: undefined method `d' for #<A:0
    x0056512e4e3ab0 @a=2, @b=2>
Did you mean?  d=
        from (irb):176
        from /usr/bin/irb:11:in `<main>'
irb(main):177:0> a.d=3
=> 3
```

Metódust definiálhatunk egy-egy példányra specializálva is, ekkor azt a metódus szingletonnak nevezzük. A 178-180. sorban kizárólag az `a` példányra vonatkozóan definiálunk egy metódust, ami az `A` osztály más példányára már nem hozzáférhető, illetve elfedi az osztály azonos nevű metódusát.

```
irb(main):178:0> def a.m2(val)
irb(main):179:1> @a + val
irb(main):180:1> end
=> :m2
irb(main):181:0> a.m2 3
=> 5
irb(main):182:0> b = A.new
=> #<A:0x0056512e7e9c28 @a=2>
irb(main):183:0> b.m2
NoMethodError: undefined method `m2' for #<A:0
    x0056512e7e9c28 @a=2>
Did you mean?  m
        from (irb):183
        from /usr/bin/irb:11:in `<main>'
```

Ruby-ban is definiálhatók úgy nevezett osztálymetódusok (186-188. sor) a metódus az osztály azonosítójával való nevesítésével. Az osztálymetódusok az osztály példányainak létezése nélkül is meghívhatók az osztály nevére való hivatkozással, illetve közösek, azonos viselkedést nyújtanak az osztály összes példányára vonatkozóan. Osztálymetódus az osztálydefiníción belüli az aktuális példány referenciájára (`self`) vonatkozó szingleton metódus definíciójával azonos hatást érhetünk el. Osztályváltozók (212. sor), vagyis a `@@` prefixű azonosítóval rendelkező változók, csakis osztálymetódusokon belül használhatók, és az első használatkor inicializálandók, különben hozzáféréskor hibát kapunk.

```
irb(main):185:0> class A
irb(main):186:1> def A.m3
```

```
irb(main):187:2>  "hello"
irb(main):188:2>  end
irb(main):189:1>  end
=>  :m3
irb(main):190:0>  A.m3
=>  "hello"
irb(main):191:0>  A::m3
=>  "hello"
irb(main):210:0>  class A
irb(main):211:1>  def A.m31
irb(main):212:2>  @@a = 2
irb(main):213:2>  @@a
irb(main):214:2>  end
irb(main):215:1>  end
=>  :m31
irb(main):216:0>  A.m31
=>  2
```

A Ruby egyik kellemes tulajdonsága a hash, amelyet metódus formális paramétereként felhasználva a formális paramétereket opcionálissá tehetjük, valamint tetszés szerinti sorrendben adhatjuk meg őket. Ez a metódus definíciója során többletmunkát igényel, viszont megkönnyíti a használatot. A 217-224. sorban egy ilyen definíciót látunk. A metódus paraméterét mint hash objektum kezeljük, amelynek az `:n` szimbólummal jelölt értékét hozzárendeljük az `n` lokális változóhoz, vagy ha az `:n` szimbólum nem szerepel a hívás argumentumai között mint kulcs, akkor 0-ra inizicializáljuk. A másik két lokális változó definíciója hasonlóképp történik. A hash argumentummal definiált metódusok hívására a 226. sor mutat példát, ahol azt láthatjuk, hogy az egyetlen formális paraméthez egy aktuálisparaméter-listát rendelünk, ami hash-sé transzformálódik.

A `:m` egy speciális lexikai elem Ruby-ban, ún. szimbólum, ami egy konstans `String`-et takar. A szimbólumok és a stringek kölcsönösen átalakíthatók egymásba, a stringből szimbólumba való irány implicit.

Változó hosszúságú paraméterlistát tömbként definiált paraméterrel hozhatunk létre, az ilyen paraméterek elé `*`-ot teszünk.

```
irb(main):217:0>  class A
irb(main):218:1>  def m4(a)
irb(main):219:2>  n = a[:n]  ||  0
irb(main):220:2>  m = a[:m]  ||  1
irb(main):221:2>  l = a[:l]  ||  0
irb(main):222:2>  l + m + n
```

```
irb(main):223:2> end
irb(main):224:1> end
=> :m4
irb(main):225:0> a = A.new
=> #<A:0x0056512e4f4ce8 @a=2>
irb(main):226:0> a.m4 :n => 3, :l => 4
=> 8
irb(main):227:0> *a = 1,2
=> [1, 2]
irb(main):228:0> a
=> [1, 2]
```

Az osztálydefiníció metódusai alapértelmezés szerint nyilvánosak (`public`), vagyis bármely példányon keresztül elérhetők. A Ruby két másik láthatósági szintet is definiál, a `protected` csak az adott osztály, illetve a leszármazott osztályok számára hozzáférhető, míg a `private` csak az adott osztályban használható. Az oszlálydefinícióban a láthatósági módósítók közötti blokkban lévő összes metódus az aktuális láthatósággal bír.

```
irb(main):229:0> class A
irb(main):230:1> def initialize
irb(main):231:2> @a = 2
irb(main):232:2> end
irb(main):233:1> def m(p1, p2)
irb(main):234:2> p1 + p2 + @a
irb(main):235:2> end
irb(main):236:1> protected
irb(main):237:1> def a
irb(main):238:2> @a
irb(main):239:2> end
irb(main):240:1> private
irb(main):241:1> def a=(val)
irb(main):242:2> @a = val
irb(main):243:2> end
irb(main):244:1> end
=> :a=
irb(main):245:0> a = A.new
=> #<A:0x0056512e7e8a58 @a=2>
irb(main):246:0> a.a
NoMethodError: protected method `a' called for #<A:0
    x0056512e7e8a58 @a=2>
        from (irb):246
```

```
        from /usr/bin/irb:11:in '<main>'
irb(main):247:0> a.a=3
NoMethodError: private method 'a=' called for #<A:0
    x0056512e7e8a58 @a=2>
Did you mean?  a
        from (irb):247
        from /usr/bin/irb:11:in '<main>'
irb(main):248:0> class B < A
irb(main):249:1> end
=> nil
irb(main):250:0> class B < A
irb(main):251:1> def a=(val)
irb(main):252:2> @a = val*2
irb(main):253:2> end
irb(main):254:1> end
=> :a=
irb(main):255:0> b = B.new
=> #<B:0x0056512e70de08 @a=2>
irb(main):256:0> b.a
NoMethodError: protected method 'a' called for #<B:0
    x0056512e70de08 @a=2>
Did you mean?  a=
        from (irb):256
        from /usr/bin/irb:11:in '<main>'
irb(main):257:0> b.a=3
=> 3
```

Az osztályok a `<` operátorral specializálhatók. A leszármazott osztály kiegészítheti vagyis specializálhatja, illetve felüldefiniálhatja az ősosztály viselkedését. A 250-254. sorban a `B` osztályt definiáljuk, amely rendelkezik az `A` osztály összes `public` és `protected` metódusával, illetve példányváltozójával. A 251-253. sor felhasználja az `A` osztály `protected` láthatósággal rendelkező `a` azonosítójú metódusát. A felüldefiniált metódusok láthatósága eltérhet egy leszármazott osztályban.

Metódusokhoz hasonlóan operátorok is (felül)definiálhatók egy osztályon belül (260-263. sor), mivel az operátorok önmaguk is metódushívások, lásd 258. és 265. sorok.

```
irb(main):258:0> 1.+(2)
=> 3
irb(main):259:0> class A
irb(main):260:1> def +(val)
```

```
irb(main):261:2> @a + val
irb(main):262:2> end
irb(main):263:1> end
=> :+
irb(main):264:0> a = A.new
=> #<A:0x0056512e6f3da0 @a=2>
irb(main):265:0> a.+(2)
=> 4
```

A Ruby másik egysége az osztály mellett a modul. A modul, akárcsak az osztály, egységbe zár attribútumokat és metódusokat. Modulba olyan metódusokat helyezhetünk el, amelyek több osztályra vonatkozóan közösek. A modul rokonságot mutat a Java interfész fogalmával, azzal a különbséggel, hogy a modul nemcsak deklarál egy bizonyos viselkedést a megvalósító osztály számára, hanem mindjárt specifikálja is. A modulnak nem lehet közvetlen peldánya, viszont létezhet objektum, amely rendelkezik a modulban definiált viselkedéssel.

A 266-273. sorok egy modult definiálnak egy példányváltozóval és egy setter-getter párral. A modul nem példányosítható (274. sor), viszont a modulban definiált metódusokkal bármely osztály viselkedését kibővíthetjük (275-277. sor) az `include` kulcsszóval példánymetódusként, az `extend` kulcsszóval osztálymetódusként. Egy osztály tetszőleges számú modult integrálhat magába.

```
irb(main):266:0> module Szin
irb(main):267:1> def szin=(val)
irb(main):268:2> @szin = val
irb(main):269:2> end
irb(main):270:1> def szin
irb(main):271:2> @szin
irb(main):272:2> end
irb(main):273:1> end
=> :szin
irb(main):274:0> m = Szin.new
NoMethodError: undefined method `new' for Szin:Module
        from (irb):274
        from /usr/bin/irb:11:in `<main>'
irb(main):275:0> class A
irb(main):276:1> include Szin
irb(main):277:1> end
=> A
irb(main):278:0> a = A.new
```

```
=> #<A:0x0056512e4fbb38 @a=2>
irb(main):279:0> a.szin = 'kek'
=> "kek"
```

A modulok emellett lehetővé teszik az esetleges osztálynév-ütközések elkerülését, névtereket definiálhatunk velük. Ekkor a logikailag összetartozó osztályok definícióját közös moduldefiníción belül helyezünk el.

```
irb(main):280:0> module M
irb(main):281:1> class A
irb(main):282:2> end
irb(main):283:1> end
=> nil
irb(main):284:0> b = M::A.new
=> #<M::A:0x0056512e4dbce8>
irb(main):285:0> a = A.new
=> #<A:0x0056512e4d3c00 @a=2>
```