A Ruby programozási nyelv
Gyakorlat

Kovács Gábor

2017. február 14.

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 1.9.3-as verziót használunk a félév során, ezért erre még szükségünk van. 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.

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):002:0> # Akarmi
irb(main):005:0> =begin
irb(main):006:0= Akarmi
irb(main):007:0= van
irb(main):008:0= itt
irb(main):009:0= =end
```

Literálok Rubyban:

- egész szám (10. sor, 43. sor)
- lebegőpontos szám (11. sor)
- string, amit megadhatunk ' vagy " szimbólumok között (12. és 13. sor)
- tömb, ami egy szögletes zárójelek között megadott vesszővel elválasztott lista (14. sor)
- hash, ami kapcsos zárójelek között tartalmaz kulcs-érték párokat => vagy : szimbólummal elválasztva (15. sor)
- reguláris kifejezés (16. sor)

- tartomány, ami a két szélső értékkel megadott egymás utáni egész értékek halmaza (17-18. sorok)
- szimbólum, ami nem más, mint egy lebutított string, és amelyet hashekben előszeretettel alkalmazunk kulcsként (36. sor)
- boolean literálok (19-20. sorok)
- típus és érték nélküli literál (21. sor)
- Ruby 2.0-tól létezik a racionális és képzetes literál is, amikkel komplex számokat írhatunk le

```
irb(main):010:0> 1
=> 1
irb(main):043:0> 1_000_000_000_000_000_000_000_000.class
=> Bignum
irb(main):011:0> 1.0
=> 1.0
irb(main):012:0> 'String'
=> "String"
irb(main):013:0> "String"
=> "String"
irb(main):014:0> [1, 1.0, "egy"]
=> [1, 1.0, "egy"]
irb(main):015:0> { "egy" => 1.0, 1 => "egy" }
=> {"egy"=>1.0, 1=>"egy"}
irb(main):016:0> /[a-z]/
=> /[a-z]/
irb(main):017:0> 1..6
=> 1..6
irb(main):018:0> 1...6
=> 1...6
irb(main):036:0> :f
=> :f
 irb(main):019:0> true
=> true
irb(main):020:0> false
=> false
irb(main):021:0> nil
=> nil
```

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.

```
irb(main):022:0> Konstans = 1
=> 1
irb(main):023:0> Konstans = 2
(irb):23: warning: already initialized constant
   Konstans
(irb):22: warning: previous definition of Konstans was
   here
=> 2
irb(main):024:0> v = 1
=> 1
irb(main):025:0> $globalis = 1
=> 1
irb(main):026:0> @peldanyvaltozo = 1
=> 1
irb(main):027:0> @@osztalyvaltozo = 1
(irb):27: warning: class variable access from toplevel
```

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 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):028:0> 1 + 2
=> 3
irb(main):029:0> 1 +
irb(main):030:0* 2
=> 3
irb(main):031:0> 1
```

```
=> 1
irb(main):032:0> + 2
=> 2
irb(main):033:0> 1 \
irb(main):034:0* + 2
=> 3
irb(main):045:0> 3/2
=> 1
irb(main):046:0> 3/2.0
=> 1.5
irb(main):047:0> -3/2
=> -2
irb(main):048:0> -(3/2)
=> -1
irb(main):049:0> 2**2
=> 4
irb(main):050:0> 2**3
=> 8
```

Rubyban minden objektum, még az elemi típus literáljai is. Kétféle egész létezik `Fixnum` és `Bignum`. 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):050:0> "string".class
=> String
irb(main):127:0> a = 1
=> 1
irb(main):138:0> a
=> 1
irb(main):135:0> a.respond_to? '+'
=> true
irb(main):136:0> a.respond_to? '-'
=> true
irb(main):137:0> a.methods
=> [:to_s, :inspect, :-@, :+, :-, :*, :/, :div, :%, :
```

```
modulo, :divmod, :fdiv, :**, :abs, :magnitude, :==,
:===, :<=>, :>, :>=, :<, :<=, :~, :&, :|, :^, :[],
:<<, :>>, :to_f, :size, :zero?, :odd?, :even?, :succ
, :integer?, :upto, :downto, :times, :next, :pred, :
chr, :ord, :to_i, :to_int, :floor, :ceil, :truncate,
 :round, :gcd, :lcm, :gcdlcm, :numerator, :
denominator, :to_r, :rationalize, :
singleton_method_added, :coerce, :i, :+@, :eql?, :
quo, :remainder, :real?, :nonzero?, :step, :to_c, :
real, :imaginary, :imag, :abs2, :arg, :angle, :phase
, :rectangular, :rect, :polar, :conjugate, :conj, :
between?, :nil?, :=~, :!~, :hash, :class, :
singleton_class, :clone, :dup, :taint, :tainted?, :
untaint, :untrust, :untrusted?, :trust, :freeze, :
frozen?, :methods, :singleton_methods, :
protected_methods, :private_methods, :public_methods
, :instance_variables, :instance_variable_get, :
instance_variable_set, :instance_variable_defined?,
:remove_instance_variable, :instance_of?, :kind_of?,
 :is_a?, :tap, :send, :public_send, :respond_to?, :
extend, :display, :method, :public_method, :
define_singleton_method, :object_id, :to_enum, :
enum_for, :equal?, :!, :!=, :instance_eval, :
instance_exec, :__send__, :__id__]
```

Az azonosítók másik csoportja a függvény-, vagy másnéven metódusazonosítók. Függvényt a `def` kulcsszó után definiálhatunk a függvény azonosítója (41.sor), a formális paraméterlista és a törzs megadásával. A függvény törzse a `def`-fel egy szinten lévő `end`-ig tart (43.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):037:0> def f(n)
irb(main):038:1> 2*n
```

```
irb(main):039:1> end
=> nil
irb(main):040:0> f(1+2)+3
=> 9
irb(main):041:0> f (1+2)+3
=> 12
irb(main):058:0> def f=
irb(main):059:1> end
=> nil
irb(main):060:0> def f?
irb(main):061:1> true
irb(main):062:1> end
=> nil
irb(main):063:0> def f!
irb(main):064:1> end
=> nil
irb(main):044:0> defined? f
=> "method"
```

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):051:0> "String".class
=> String
irb(main):052:0> $a = 'lo'
=> "lo"
irb(main):053:0> "Hel#{$lo}"
=> "Hel"
irb(main):054:0> "Hel#{$a}"
=> "Hello"
irb(main):055:0> 'Hel#{$a}'
=> "Hel\#{$a}"
irb(main):056:0> a = "Hel#{$a}"
=> "Hello"
irb(main):057:0> a[0]
=> "H"
irb(main):058:0> a[-1]
=> "o"
irb(main):059:0> a[-5]
```

```
=> "H"
irb(main):060:0> a[0..2]
=> "Hel"
irb(main):061:0> 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):062:0> arr = [1, 1.0, "egy"]
=> [1, 1.0, "egy"]
irb(main):063:0> arr[0]
=> 1
irb(main):064:0> arr[1]
=> 1.0
irb(main):065:0> arr[2]
=> "egy"
irb(main):066:0> arr * 2
=> [1, 1.0, "egy", 1, 1.0, "egy"]
irb(main):067:0> arr.join(':')
=> "1:1.0:egy"
irb(main):068:0> tmp = arr.join(':')
=> "1:1.0:egy"
irb(main):069:0> tmp.split(':')
=> ["1", "1.0", "egy"]
irb(main):070:0> h = { "egy" => 1.0, 1 => "egy" }
=> {"egy"=>1.0, 1=>"egy"}
irb(main):071:0> h[2] = "ketto"
=> "ketto"
irb(main):072:0> h
=> {"egy"=>1.0, 1=>"egy", 2=>"ketto"}
```

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.

```
irb(main):073:0> 1..6
=> 1..6
irb(main):074:0> 'a'..'f'
=> "a".."f"
```

```
irb(main):075:0> r = 'a'..'f'
=> "a".."f"
irb(main):076:0> r.each { |l| print "␣#{l}␣" }
 a  b  c  d  e  f => "a".."f"
irb(main):077:0> r = 'a'...'f'
=> "a"..."f"
irb(main):078:0> 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.

```
irb(main):015:0> true
=> true
irb(main):016:0> false
=> false
irb(main):017:0> 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` blokkja kiemelhető a sor elejére. 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):079:0> i = 2
=> 2
```

```
irb(main):080:0> if i > 1 then print "nagy" elsif i ==
   1 then print "egy" else print "kicsi" end
nagy=> nil
irb(main):081:0> if i > 1
irb(main):082:1> print "nagy"
irb(main):083:1> elsif i == 1
irb(main):084:1> print "egy"
irb(main):085:1> else
irb(main):086:1* print "kicsi"
irb(main):087:1> end
nagy=> nil
irb(main):088:0> unless i==1 then print "nemegy" else
   print "egy" end
nemegy=> nil
irb(main):089:0> case i
irb(main):090:1> when 1
irb(main):091:1> print "egy"
irb(main):092:1> when 2..10
irb(main):093:1> print "nemegy"
irb(main):094:1> end
nemegy=> nil
irb(main):095:0> i = 3 if i == 2
=> 3
irb(main):096:0> i = 3 unless i == 2
=> 3
```

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

```
irb(main):097:0> i = 1
=> 1
irb(main):098:0> while i < 3 do
irb(main):099:1* print "#{i=i+1}"
irb(main):100:1> end
23=> nil
irb(main):101:0> i = 1
=> 1
irb(main):102:0> until i > 3 do
irb(main):103:1* print "#{i=␣i+1}"
irb(main):104:1> end
234=> nil
```

```
irb(main):105:0> for i in 1..3 do print "#{i}" end
123=> 1..3
irb(main):106:0> for i in 1..3 do print "#{i}\n" end
1
2
3
=> 1..3
```

Ruby-ban az azonosítók referenciaként viselkednek. Típusuk nem deklaráció 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):118:0> a
=> "Hello"
irb(main):119:0> b = a
=> "Hello"
irb(main):120:0> a.reverse
=> "olleH"
irb(main):121:0> a
=> "Hello"
irb(main):122:0> a.reverse!
=> "olleH"
irb(main):123:0> a
=> "olleH"
irb(main):124:0> b
=> "olleH"
irb(main):125:0> a = b = c = 0
=> 0
irb(main):126:0> a == b
=> true
irb(main):127:0> a = 1
=> 1
irb(main):128:0> b = 1.0
=> 1.0
irb(main):129:0> a == b
=> true
irb(main):130:0> a.eql? b
=> false
```

```
irb(main):131:0> a.equal? b
=> false
irb(main):132:0> a == b
=> true
```

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):140:0> a == "1"
=> false
irb(main):141:0> a == "1".to_i
=> true
irb(main):142:0> a.to_s == "1"
=> true
irb(main):143:0> if a
irb(main):144:1> print "#{a}"
irb(main):145:1> end
1=> nil
```

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):135:0> a.respond_to? '+'
=> true
irb(main):136:0> a.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 107. sorban bemutatott tartomány típus `each` metódusa is ilyen. A 108-111. sorokban a `t` metódus definíciója meghívja a metódushívás paramétereként a 112. sorban átadott blokkot. A blokkban definiált procedúra paraméterezhető, ahogy azt a 113-115. sorok mutatják. A 116. sorban

meghívjuk a 113. sorban definiált `t` azonosítójú függvényt egy string paraméterrel. A függvény törzsében, vagyis a 114. sorban átadja a vezérlést a 116. sor blokkja törzsének átadva az `a` értéket. A 116. 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 115. sorban, a `yield` után folytatódik. A `yield`-et tartalmazó metódushívás blokk argumentuma nem hagyható el! 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):107:0> r.each { |l| print "␣#{l}␣" }
 a  b  c  d  e => "a"..."f"
irb(main):108:0> def t
irb(main):109:1> yield
irb(main):110:1> yield
irb(main):111:1> end
=> nil
irb(main):112:0> t { print 'a' }
aa=> nil
irb(main):113:0> def t(a)
irb(main):114:1> yield a
irb(main):115:1> end
=> nil
irb(main):116:0> t('hello') { |l| print "#{l}" }
hello=> nil
irb(main):117:0> t('hello') do |l| print "#{l}" end
hello=> 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):135:0> ::Konstans
=> 2
irb(main):140:0> Math::sqrt 4
=> 2.0
```

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 141-142. sorban egy `Szemely` azonosítójú osztályt definiálunk, amelyet a 143. sorban példányosítunk. A típusok azonosítóit Ruby-ban konvenció szerint nagybetűvel kezdjük. Az osztályt a 146-150. sorban kibővítjük egy `osszeadas` azonosítójú metódussal, ami 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 146-150. sorban bővítjük, a 157-159. sorban felüldefiniáljuk a viselkedést. A 151. sorban létrehozunk egy példányt az implicit ősosztályból, vagyis az `Object`-ből módon örökölt `new` metódussal, ekkor láthatjuk az objektum memóriabeli címét. A 152. sorban meghívjuk az `m` metódus két paraméterrel.

```
irb(main):141:0> class Szemely
irb(main):142:1> end
=> nil
irb(main):146:0> class Szemely
irb(main):147:1> def osszeadas(p1,p2)
irb(main):148:2> p1 + p2
irb(main):149:2> end
irb(main):150:1> end
=> nil
irb(main):151:0> a = Szemely.new
=> #<Szemely:0x007fb79306fa70>
irb(main):152:0> a.osszeadas 2, 3
=> 5
irb(main):153:0> class Szemely
irb(main):154:1> def initialize
irb(main):155:2> @a = 2
irb(main):156:2> end
irb(main):157:1> def osszeadas(p1,p2)
irb(main):158:2> @a + p1 + p2
irb(main):159:2> end
irb(main):160:1> end
=> nil
irb(main):161:0> a = Szemely.new
=> #<Szemely:0x007fb793113be8 @a=2>
irb(main):162:0> a.osszeadas 2, 3
=> 7
```

A kezdeti viselkedést az `initialize` metódus (felül)definiálásával határozhatjuk meg. A 154-156. sorokban az `Szemely` 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 157-159. sorban felüldefiniált `osszeadas` metódusban megtesszük.

A példányváltozókhoz úgy nevezett setter és getter metódusokkal férhetünk hozzá. A setter jellemzője, hogy a változó azonosítója mögé egy egyenlőségjelet írunk (168-170. sor), a getter azonosítója pedig maga a példányváltozó azonosítója `@` szimbólum nélkül (171-173. sor). Használatukat a 176., 177. és a 178. sor mutatja. A 175. sor visszatérési értékében láthatuk, hogy a példányváltozó létrejött az első használatkor, vagyis a `initialize` metódus lefutott.

```
irb(main):167:0> class Szemely
irb(main):168:1> def a=(val)
irb(main):169:2> @a = val
irb(main):170:2> end
irb(main):171:1> def a
irb(main):172:2> @a
irb(main):173:2> end
irb(main):174:1> end
=> nil
irb(main):175:0> a = Szemely.new
=> #<Szemely:0x007fb7920248b38 @a=2>
irb(main):176:0> a.a
=> 2
irb(main):177:0> a.a=3
=> 3
irb(main):178:0> a.a
=> 3
irb(main):179:0> a
=> #<Szemely:0x007fb7920248b38 @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 186-191. sorok mutatják.

```
irb(main):180:0> class Szemely
irb(main):181:1> attr_accessor :c
irb(main):182:1> attr_reader :d
```

```
irb(main):183:1> attr_writer :e
irb(main):184:1> end
=> nil
irb(main):185:0> a = Szemely.new
=> #<Szemely:0x007fb79207caa0 @a=2>
irb(main):186:0> a.c
=> nil
irb(main):187:0> a.c = 2
=> 2
irb(main):188:0> a.d
=> nil
irb(main):189:0> a.d=2
NoMethodError: undefined method `d=' for #<Szemely:0
    x007fb79207caa0 @a=2, @c=2>
        from (irb):189
        from /usr/bin/irb:12:in `<main>'
irb(main):190:0> a.e =3
=> 3
irb(main):191:0> a.e
NoMethodError: undefined method `e' for #<Szemely:0
    x007fb79207caa0 @a=2, @c=2, @e=3>
        from (irb):191
        from /usr/bin/irb:12:in `<main>'
```

Metódust definiálhatunk egy-egy példányra specializálva is, ekkor azt a metódus szingletonnak nevezzük. A 192-194. sorban kizárólag az `a` példányra vonatkozóan definiálunk egy metódust, ami az `Szemely` osztály más példányára már nem hozzáférhető, illetve elfedi az osztály azonos nevű metódusát (196-197. sor).

```
irb(main):192:0> def a.osszeadas(p1,p2)
irb(main):193:1> p1.to_s + p2.to_s
irb(main):194:1> end
=> nil
irb(main):195:0> a.osszeadas 2,3
=> "23"
irb(main):196:0> b = Szemely.new
=> #<Szemely:0x007fb793138808 @a=2>
irb(main):197:0> b.osszeadas 2,3
=> 7
```

Ruby-ban is definiálhatók úgy nevezett osztálymetódusok (199-201. 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 (222-224. sor). Osztályváltozók, vagyis a `@@` prefixű azonosítóval rendelkező változók, csakis osztálymetódusokon belül használhatók, és az első használatkor inicializálandók, különben hozzáféréskor hibát kapunk.

```
irb(main):198:0> class Szemely
irb(main):199:1> def Szemely.hello
irb(main):200:2> "hello"
irb(main):201:2> end
irb(main):202:1> end
=> nil
irb(main):203:0> Szemely.hello
=> "hello"
irb(main):204:0> class Szemely
irb(main):205:1> def self.hallo
irb(main):206:2> "hallo"
irb(main):207:2> end
irb(main):208:1> end
=> nil
irb(main):209:0> class Szemely
irb(main):210:1> def Szemely.hello
irb(main):211:2> @@a = "hello"
irb(main):212:2> @@a
irb(main):213:2> end
irb(main):214:1> end
=> nil
irb(main):215:0> Szemely.hello
=> "hello"
```

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 220-227. sorban egy ilyen definíciót látunk. A metódus paraméterét mint hash objektum kezeljük, amelynek az `:alapfizetes` szimbólummal jelölt értékét hozzárendeljük az `n` lokális változóhoz, vagy ha az `:alapfizetes` 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 229. sor mutat példát.

A `:azonosito` 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.

```
irb(main):220:0> class Szemely
irb(main):221:1> def jovedelem(a)
irb(main):222:2> n = a[:alapfizetes] || 0
irb(main):223:2> m = a[:extra] || 0
irb(main):224:2> l = a[:jutalom] || 0
irb(main):225:2> n + m + l
irb(main):226:2> end
irb(main):227:1> end
=> nil
irb(main):228:0>   a = Szemely.new
=> #<Szemely:0x007fb79180ccd0 @a=2>
irb(main):229:0> a.jovedelem :alapfizetes => 1, :extra
   => 3000000, :jutalom => 1000000
=> 4000001
```

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):230:0> class Szemely
irb(main):231:1> def initialize
irb(main):232:2> @a = 2
irb(main):233:2> end
irb(main):234:1> def osszeadas(p1,p2)
irb(main):235:2> p1 + p2 + @a
irb(main):236:2> end
irb(main):237:1> protected
irb(main):238:1> def a
irb(main):239:2> @a
irb(main):240:2> end
irb(main):241:1> private
irb(main):242:1> def a=(val)
```

```
irb(main):243:2> @a = val
irb(main):244:2> end
irb(main):245:1> end
=> nil
irb(main):246:0> a = Szemely.new
=> #<Szemely:0x007fb7920da0d8 @a=2>
irb(main):247:0> a.a
NoMethodError: protected method `a' called for #<
    Szemely:0x007fb7920da0d8 @a=2>
        from (irb):247
        from /usr/bin/irb:12:in `<main>'
irb(main):248:0> a.a=1
NoMethodError: private method `a=' called for #<Szemely
    :0x007fb7920da0d8 @a=2>
        from (irb):248
        from /usr/bin/irb:12:in `<main>'
irb(main):249:0> class Fonok < Szemely
irb(main):250:1> end
=> nil
irb(main):251:0> b = Fonok.new
=> #<Fonok:0x007fb792106b38 @a=2>
irb(main):252:0> b.osszeadas 2,3
=> 7
irb(main):258:0> class Fonok < Szemely
irb(main):259:1> def a=(val)
irb(main):260:2> @a = val
irb(main):261:2> end
irb(main):262:1> end
=> nil
irb(main):263:0> a.a=1
NoMethodError: private method `a=' called for #<Szemely
    :0x007fb7920da0d8 @a=2>
        from (irb):263
        from /usr/bin/irb:12:in `<main>'
irb(main):264:0> b = Fonok.new
=> #<Fonok:0x007fb792144988 @a=2>
irb(main):265:0> b.a = 2
=> 2
```

Az osztályok a < operátorral specializálhatók. A leszármazott osztály kiegészítheti vagyis specializálhatja, illetve felüldefiniálhatja az ősosztály vi-

selkedését. A 258-262. sorban a `Fonok` osztályt definiáljuk, amely rendelkezik az `Szemely` osztály összes `public` és `protected` metódusával, illetve példányváltozójával. A 259-261. sor felüldefiniálja az `a` attribútum setter metódusát, így az másképp fog viselkedni, mint azt a 242. sorban láttuk, `Fonok` példányaira ismét elérhető lesz az osztály példányán keresztül. A felüldefiniált metódus láthatósága eltérhet egy leszármazott osztályban.

Metódusokhoz hasonlóan operátorok is (felül)definiálhatók egy oszlályon belül (268-272. sor), mivel az operátorok önmaguk is metódushívások, lásd 275. sor.

```
irb(main):266:0> 1 + 2
=> 3
irb(main):267:0> 1.+(2)
=> 3
irb(main):268:0> class Szemely
irb(main):269:1> def +(val)
irb(main):270:2> @a + val
irb(main):271:2> end
irb(main):272:1> end
=> nil
irb(main):273:0> a = Szemely.new
=> #<Szemely:0x007fb792175740 @a=2>
irb(main):274:0> a+2
=> 4
irb(main):275:0> 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 (314. sor), viszont létezhet objektum, amely rendelkezik a modulban definiált viselkedéssel.

A 276-283. sorok egy modult definiálnak egy példányváltozóval és egy setter-getter párral. A modul nem példányosítható, viszont a modulban definiált metódusokkal bármely osztály viselkedését kibővíthetjük (316.) az `include` kulcsszóval példánymetódusként, az `extend` kulcsszóval osztálymetódusként. Egy osztály tetszőleges számú modult integrálhat magába.

```
irb(main):276:0> module Szin
```

```
irb(main):277:1> def szin=(val)
irb(main):278:2> @szin = val
irb(main):279:2> end
irb(main):280:1> def szin
irb(main):281:2> @szin
irb(main):282:2> end
irb(main):283:1> end
=> nil
irb(main):314:0> s = Szin.new
NoMethodError: undefined method `new' for Szin:Module
        from (irb):314
        from /usr/bin/irb:11:in `<main>'
irb(main):285:0> class Szemely
irb(main):286:1> include Szin
irb(main):287:1> end
=> Szemely
irb(main):288:0> a = Szemely.new
=> #<Szemely:0x007fb7921a6868 @a=2>
irb(main):289:0> a.szin="kek"
=> "kek"
irb(main):290:0> a.szin
=> "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):291:0> module Azenyem
irb(main):292:1> class Time
irb(main):293:2> end
irb(main):294:1> end
=> nil
irb(main):295:0> Azenyem::Time.new
=> #<Azenyem::Time:0x007fb7921bdf68>
```