A Ruby programozási nyelv
Gyakorlat

Kovács Gábor

2010. szeptember 17.

Amire szükségünk lesz a gyakorlat során: egy telepített Ruby környezet. A Ruby programokat a `ruby` értelmező futtatja, aminek paramétere a futtatni kívánt szkript fájl. A 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. 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 -w
# -*- coding: utf-8 -*-
require 'socket'

__END__
```

A gyakorlaton az `irb` értelmezőt használjuk a Ruby értelmező demonstrálására. A dokumentációt az `ri` paranccsal olvashatjuk, amelynek a megtekinteni kívánt osztály (pl. `$ri String`) vagy metódus nevét (pl. `$ri String.length`) kell megadnunk.

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

```
F:\kovacsg\munka\targyak\ror\RubyPeldak>irb
irb(main):001:0> "Hello, world"
=> "Hello, world"
```

```
irb(main):002:0> puts "Hello, world"
Hello, world
=> nil
```

Literálok Rubyban:

- egész szám
- lebegőpontos szám
- string, amit megadhatunk ' vagy " szimbólumok között
- reguláris kifejezés
- tömb, ami egy szögletes zárójelek között megadott vesszővel elválasztott lista
- hash, ami kapcsos zárójelek között tartalmaz kulcs-érték párokat => szimbólummal elválasztva

```
irb(main):003:0> 1
=> 1
irb(main):004:0> 1.0
=> 1.0
irb(main):005:0> 'String'
=> "String"
irb(main):006:0> "String"
=> "String"
irb(main):007:0> /[a-z]/
=> /\[a-z\]/
irb(main):008:0> [1, 2, 3]
=> [1, 2, 3]
irb(main):009:0> { 1 =>"egy", "ketto" => 2 }
=> {1=>"egy", "ketto"=>2}
irb(main):010:0> h = { 1 =>"egy", "ketto" => 2 }
=> {1=>"egy", "ketto"=>2}
irb(main):011:0> h[1]
=> "egy"
```

Négyféle azonosítót különböztethetünk meg:

- konstans: nagybetűvel kezdődik
- (lokális) azonosító: kisbetűvel vagy _ szimbólummal kezdődik

- globális azonosító: `$` szimbólummal kezdődik
- példányváltozó azonosító: `@` szimbólummal kezdődik

A Ruby megkülönbözteti a kis- és nagybetűket.

```
irb(main):012:0> Konstans = 1
=> 1
irb(main):013:0> Konstans = 2
(irb):13: warning: already initialized constant
    Konstans
=> 2
irb(main):014:0> Konstans
=> 2
irb(main):015:0> $globalis = "hello"
=> "hello"
```

Egysoros megjegyzést a `#` szimbólum után tehetünk.

```
irb(main):016:0> # akarmi
```

Többsoros megjegyzés tételére két módunk van:

```
=begin
akarmi
=end

#
# Dokumentacio
#
#
```

Egy állítás jellenzően a következő soremelésig tart, kivéve, ha a sor utolsó lexikai eleme egy operátor.

```
irb(main):017:0*
irb(main):018:0* 1 +
irb(main):019:0* 2
=> 3
irb(main):020:0> 1
=> 1
irb(main):021:0> +2
=> 2
irb(main):022:0> 1 \
irb(main):023:0* +2
```

```
=> 3
irb(main):024:0> "egy"
=> "egy"
irb(main):025:0> "egy".
irb(main):026:0* split("g")
=> ["e", "y"]
```

Függvényt a `def` kulcsszó után definiálhatunk a függvény azonosítója (27.sor), a formális paraméterlista és a törzs megadásával. A függvény törzse a `def`-fel egy szinten lévő `end`-ig tart (29.sor).

```
irb(main):027:0> def f(n)
irb(main):028:1> 2*n
irb(main):029:1> end
=> nil
```

Függvényhívásnál, ameddig a paraméterlista egyértelműen meghatározható, a zárójelek elhagyhatók.

```
irb(main):030:0> f(1+2)+3
=> 9
irb(main):031:0> f (1+2)+3
=> 12
irb(main):032:0> f 2
=> 4
```

Rubyban minden objektum, még az elemi típusok is. Kétféle egész létezik `Fixnum` és `Bignum`. A C-ben megszokott operátorkészlet áll itt is rendelkezésre, kiegészítve a hatványozás operátorával (`**`). Egész számok osztása egész számot ad eredményül, ha az eredmény negatív, akkor az értelmező mínusz végtelen felé kerekít.

```
irb(main):033:0> 2.class
=> Fixnum
irb(main):034:0> 1_000_000_000_000.class
=> Bignum
irb(main):035:0> 1.0.class
=> Float
irb(main):036:0> 3/2
=> 1
irb(main):037:0> 3/2.0
=> 1.5
irb(main):038:0> -3/2
=> -2
```

```
irb(main):039:0> -(3/2)
=> -1
irb(main):047:0> a =2
=> 2
irb(main):048:0> a**3
=> 8
```

Egy azonosító által reprezentált értékre `#{}` szintakszissal a kapcsos zárójelen belül hivatkozhatunk egy idézőjelben lévő stringben. A String sok tekintetben hasonlóan viselkedik, mint egy tömb.

```
irb(main):049:0> "String".class
=> String
irb(main):050:0> $a = 'lo'
=> "lo"
irb(main):055:0> "Hel#{$a}"
=> "Hello"
irb(main):056:0> 'Hel#{$a}'
=> "Hel\#{$a}"
irb(main):057:0> "Hel#{$a}"
=> "Hello"
irb(main):058:0> a[0]
=> 0
irb(main):059:0> $a[0]
=> "l"
irb(main):060:0> $a[-1]
=> "o"
irb(main):061:0> a = "Hello"
=> "Hello"
irb(main):062:0> a[-4]
=> "e"
irb(main):063:0> a[-5]
=> "H"
irb(main):065:0> a[-33]
=> nil
```

A Ruby tömb tetszőleges típusú értékekből összeállított lista. Gyakori feladat tömb elemeinek összefűzése (ld. PHP implode), illetve string objektumok szétválasztása valamely elválasztó karakterre tekintettel (ld. PHP explode).

```
irb(main):066:0> arr = [ "1", 2.0, 'Hello' ]
=> ["1", 2.0, "Hello"]
```

```
irb(main):067:0> arr[0]
=> "1"
irb(main):068:0> arr[1]
=> 2.0
irb(main):069:0> arr * 2
=> ["1", 2.0, "Hello", "1", 2.0, "Hello"]
irb(main):070:0> a *2
=> "HelloHello"
irb(main):071:0> arr.join (":")
=> "1:2.0:Hello"
irb(main):072:0> tmp = arr.join (":")
=> "1:2.0:Hello"
irb(main):073:0> tmp.split(":")
=> ["1", "2.0", "Hello"]
```

A hash egy kulcs-érték párokat tartalmazó halmaz, leginkább a PHP asszociatív tömbhöz hasonlít.

```
irb(main):074:0> h = { "egy" => 1 }
=> {"egy"=>1}
irb(main):075:0> h[2] = "ketto"
=> "ketto"
irb(main):076:0> h
=> {"egy"=>1, 2=>"ketto"}
```

Speciális lexikai elem a range, 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):079:0> r = 'a'..'f'
=> "a".."f"
irb(main):087:0> r.each {|l| print "␣#{l}␣" }
 a  b  c  d  e  f => "a".."f"
irb(main):088:0> r.each do |l| print "␣#{l}␣" end
 a  b  c  d  e  f => "a".."f"
```

Két boolean literál létezik a `true` és a `false`. A `nil` a nem definiált pointernek felel meg.

```
irb(main):089:0> true
=> true
irb(main):090:0> false
=> false
```

```
irb(main):091:0> 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
```

. 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
```

```
irb(main):092:0> i = 2
=> 2
irb(main):095:0> if i > 1 then print "nagy" elsif then
   i==1 print "egy" else print >
nagy=> nil
irb(main):100:0> i
=> 2
irb(main):101:0> if i > 1 then print "nagy" else print
   "kicsi" end
nagy=> nil
irb(main):104:0> unless i==1 then print "nemegy" else
   print "egy"
irb(main):105:1>
irb(main):106:1* end
nemegy=> nil
irb(main):107:0> i=2
=> 2
irb(main):116:0> case i
irb(main):117:1> when 1
irb(main):118:1> print 'egy'
```

```
irb(main):119:1> when 2..10
irb(main):120:1>  print 'nemegy'
irb(main):121:1> end
nemegy=> nil
irb(main):122:0> i
=> 2
irb(main):123:0> i = 3 if 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.

```
irb(main):124:0> i =1
=> 1
irb(main):125:0> while i < 3 do print "#{i=i+1}" end
23=> nil
irb(main):126:0> i = 1
=> 1
irb(main):127:0> until i > 3 do print "#{i=i+1}" end
234=> nil
irb(main):129:0> for i in 1..3 do print "#{i}"; print "
    \n" end
1
2
3
=> 1..3
```

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 87-88. sorokban bemutatott range típus `each` metódusa is ilyen. A 132-135. sorokban a `t` metódus definíciója kétszer hívja meg a metódushívás paramétereként a 136. sorban átadott blokkot. A blokkban definiált procedúra paraméterezhető, ahogy azt a 136-140. sorok mutatják. A 140. sorban meghívjuk a 137. sorban definiált `t` azonosítójú metódust egy string paraméterrel. A metódus törzsében, vagyis a 138. sorban átadja a vezérlést a 140. sor blokkja törzsének átadva az `a` értéket. A 140. sor procedúrája a paraméterül kapott értékre a lokális `l` azonosítóval hivatkozik, ami a törzs-

ben felhasználható. A törzs végével a vezérlést visszakapja a hívott metódus, vagyis a végrehajtás a 139. sorban folytatódik.

```
irb(main):130:0> r
=> "a".."f"
irb(main):131:0> r.each do |l| print "␣#{l}␣" end
 a  b  c  d  e  f => "a".."f"
irb(main):132:0> def t
irb(main):133:1> yield
irb(main):134:1> yield
irb(main):135:1> end
=> nil
irb(main):136:0> t { print 'a' }
aa=> nil
irb(main):137:0> def t(a)
irb(main):138:1> yield a
irb(main):139:1> end
=> nil
irb(main):140:0> t('hello') {|l| print "#{l}" }
hello=> nil
irb(main):141:0> def t(a)
irb(main):142:1> yield a*2
irb(main):143:1> end
=> nil
irb(main):144:0> t('hello') {|l| print "#{l}" }
hellohello=> nil
```

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 (`eql?`) metódus.

Mivel egy azonosító bármilyen típus jelenthet, nem lehetünk mindig biztosak abban, hogy az adott objektumra vonatkozóan egy-egy metódus értelmezett-e. Ekkor a `respond_to?` metódussal állapítható meg, hogy kaphatunk-e választ egy hívásra vagy sem.

```
irb(main):020:0> a = 'hello'
=> "hello"
irb(main):021:0> b = a
=> "hello"
irb(main):022:0> a.reverse!
```

```
=> "olleh"
irb(main):023:0> b
=> "olleh"
irb(main):145:0> a = b = c = 0
=> 0
irb(main):160:0> a == b
=> false
irb(main):161:0> a = 'lo'
=> "lo"
irb(main):162:0> a == b
=> true
irb(main):163:0> a.eql?b
=> true
irb(main):164:0> a =1
=> 1
irb(main):165:0> b = 1.0
=> 1.0
irb(main):166:0> a == b
=> true
irb(main):167:0> a.eql?b
=> false
irb(main):168:0> a.length
NoMethodError: undefined method `length' for 1:Fixnum
        from (irb):168
        from C:/Program Files/Ruby/bin/irb:12:in `<main
    >'
irb(main):169:0> a.class
=> Fixnum
irb(main):173:0> a.respond_to? "length"
=> false
```

Az objektumok konvertálhatók más típusúvá. A konverzió történhet automatikusan, mint például a 37. vagy esetleg az 57. sorban történik, vagy explicit módon a `to_s`, `to_i` vagy `to_f` metódusokkal.

```
irb(main):177:0> a.to_s
=> "2"
irb(main):179:0> s = a.to_s
=> "2"
irb(main):180:0> s.to_i
=> 2
irb(main):181:0> s.to_f
```

```
=> 2.0
irb(main):186:0> s::to_i
=> 2
```

A Ruby programnyelv objektumorientált, építőkockái az osztályok és a modulok, amelyek egymással a nyilvános felületükön definiált metódusaikkal kommunikálnak egymással. A 187. sorban a `Math` modul `sqrt` metódusát hívjuk meg, a híváskor a `.` vagy a `::` operátort használhatjuk.

```
irb(main):187:0> Math::sqrt 4
=> 2.0
```

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

A 188-192. sorban egy osztályt definiálunk, amelynek `m` metódusa összeadja a két paraméterül adott számot. A 193. sorban létrehozunk egy példányt az implicit ősosztályból, vagyis az `Object`-ből módon örökölt `new` metódussal, a 194. sorban meghívjuk az `m` metódus két paraméterrel.

```
irb(main):188:0> class A
irb(main):189:1> def m(p1,p2)
irb(main):190:2> p1+p2
irb(main):191:2> end
irb(main):192:1> end
=> nil
irb(main):193:0> a = A.new
=> #<A:0x19c3578>
irb(main):194:0> a.m 2, 3
=> 5
```

A kezdeti viselkedést az `initialize` metódus (felül)definiálásával határozhatjuk meg. A 195-199. 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 200-204. sorban felüldefiniált `m` metódusban megtesszük.

Ha egy osztálydefiníció során egy már létező osztály azonosítóját adjuk meg, akkor az a definíció kibővíti vagy felüldefiniálja az osztály viselkedését. A 195-199. sorban bővítjük, a 200-204. sorban felüldefiniáljuk a viselkedést.

```
irb(main):195:0> class A
irb(main):196:1> def initialize
```

```
irb(main):197:2> @a = 2
irb(main):198:2> end
irb(main):199:1> end
=> nil
irb(main):200:0> class A
irb(main):201:1> def m(p1,p2)
irb(main):202:2> p1+p2+@a
irb(main):203:2> end
irb(main):204:1> end
=> nil
irb(main):221:0> a.m 2, 3
=> 7
```

A példányváltozókhoz setter és getter metódusokkal férhetünk hozzá. A setter jellemzője, hogy a változó azonosítója mögé egy egyenlőségjelet írunk (223-225. sor), a getter pedig maga a változó azonosítója (230-232. sor). Használatukat a 227. és a 234. sor mutatja.

```
irb(main):222:0> class A
irb(main):223:1> def a=(val)
irb(main):224:2> @a=val
irb(main):225:2> end
irb(main):226:1> end
=> nil
irb(main):227:0> a.a=3
=> 3
irb(main):228:0> a.m 2, 3
=> 8
irb(main):229:0> class A
irb(main):230:1> def a
irb(main):231:2> @a
irb(main):232:2> end
irb(main):233:1> end
=> nil
irb(main):234:0> 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.

```
class A
  attr_accessor :c
  attr_reader :d
  attr_writer :e
end
```

Metódust definiálhatunk egy-egy példányra specializálva is, ekkor az a metódus szingletonnak nevezzük. A 235-237. sorban az `a` példányra vonatkozóan definiálunk egy metódust, ami az `A` osztály más példányára már nem hozzáférhető.

```
irb(main):235:0> def a.m2(val)
irb(main):236:1>   @a+val
irb(main):237:1> end
=> nil
irb(main):238:0> a.m2 2
=> 5
irb(main):239:0> b = A.new
=> #<A:0x10018f8 @a=2>
irb(main):245:0> a.m2 2
=> 5
irb(main):246:0> b.m2 2
NoMethodError: undefined method `m2' for #<A:0x10018f8
    @a=2>
        from (irb):246
        from C:/Program Files/Ruby/bin/irb:12:in `<main
    >'
```

Ruby-ban is definiálhatók osztálymetódusok (248-250.sor), amelyek az osztály példányainak létezése nélkül is meghívhatók (252.sor), illetve közösek az összes példányra vonatkozóan.

```
irb(main):247:0> class A
irb(main):248:1> def A.m3
irb(main):249:2>   "hello"
irb(main):250:2> end
irb(main):251:1> end
=> nil
irb(main):252:0> A.m3
=> "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 262-267. 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 269. és a 270. 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.

```
irb(main):261:0> class A
irb(main):262:1> def m4(a)
irb(main):263:2> n=a[:n] || 0
irb(main):264:2> m=a[:m] || 1
irb(main):265:2> l=a[:l] || 0
irb(main):266:2> n+m+l
irb(main):267:2> end
irb(main):268:1> end
=> nil
irb(main):269:0> a.m4 :n => 3, :l => 4
=> 8
irb(main):270:0> a.m4 :n => 3, :m => 4
=> 7
```

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.

```
class A
  def initialize
    @a = 2
  end
  def m(p1,p2)
    p1+p2+@a
  end
  protected
  def a
```

```
    @a
  end

  private
  def a=(val)
    @a = val
  end

end
```

Az osztályok a < operátorral specializálhatók. A leszármazott osztály kiegészítheti vagyis specializálhatja, illetve felüldefiniálhatja az ősosztály viselkedését. A 280-284. sorban a `B` osztályt definiáljuk, amely rendelkezik az `A` oszltály összes metódusával, illetve példányváltozójával. A 281-283. sor felüldefiniálja az `a` attribútum setter metódusát, így az másképp fog viselkedni, mint azt a 227. és 234. sorban láttuk.

```
irb(main):271:0> class B<A
irb(main):272:1> end
=> nil
irb(main):280:0> class B<A
irb(main):281:1> def a=(val)
irb(main):282:2> @a = val*2
irb(main):283:2> end
irb(main):284:1> end
=> nil
irb(main):285:0> b.a=2
=> 2
irb(main):286:0>
irb(main):287:0* b.a
=> 4
```

Metódusokhoz hasonlóan operátorok is (felül)definiálhatók egy oszlályon belül (290-292. sor), mivel az operátorok önmaguk is metódushívások, lásd 288. sor.

```
irb(main):288:0> 1.+(2)
=> 3
irb(main):289:0> class A
irb(main):290:1> def +(val)
irb(main):291:2> @a +val
irb(main):292:2> end
irb(main):293:1> end
```

```
=> nil
```

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 28-35. sorok egy modult definiálnak egy példányváltozóval és egy setter-getter párral. Ezzel a modullal bármely osztály viselkedését kibővíthetjük (37. sor). Egy osztály tetszőleges számú modult integrálhat magába.

```
irb(main):028:0> module Szin
irb(main):029:1> def szin=(val)
irb(main):030:2> @szin=val
irb(main):031:2> end
irb(main):032:1> def szin
irb(main):033:2> @szin
irb(main):034:2> end
irb(main):035:1> end
=> nil
irb(main):036:0> class A
irb(main):037:1> include Szin
irb(main):038:1> end
=> A
irb(main):039:0> a = A.new
=> #<A:0x13b03c0>
irb(main):040:0> a.szin="kek"
=> "kek"
irb(main):041:0> a.szin
=> "kek"
```

A modulok emellett lehetővé teszik az esetleges osztálynév-ütközések elkerülését, névtereket definiálhatunk velük. Ekkor az összetartozó osztályok definícióját a modul definícióján belül helyezzük el.