# A Ruby programozási nyelv
# Gyakorlat

Kovács Gábor

2011. szeptember 13.

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 az ASCII, ha ékezetes karaktereket akarunk írni, ezt minden egyes Ruby forrásunk elejére be kell szúrnunk. 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 megtekinteni kívánt osztály (pl. `$ri String`) vagy metódus nevét (pl. `$ri String.length`) kell megadnunk.

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

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"
```

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
- szimbólum, ami nem más, mint egy lebutított string, és amelyet hashekben előszeretettel alkalmazunk kulcsként

```
irb(main):003:0> 1
=> 1
irb(main):004:0> 1_000_000_000
=> 1000000000
irb(main):005:0> 1.0
=> 1.0
irb(main):006:0> "Hello, world"
=> "Hello, world"
irb(main):007:0> 'Hello, world'
=> "Hello, world"
irb(main):008:0> /[a-z]/
=> /[a-z]/
irb(main):009:0> [ 1, 1.0, "egy" ]
=> [1, 1.0, "egy"]
```

```
irb(main):010:0> h = { 1 => "egy", "ketto" => 2.0 }
=> {1=>"egy", "ketto"=>2.0}
irb(main):011:0> h[1]
=> "egy"
irb(main):012:0> :egy
=> :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):013:0> Konstans = 1
=> 1
irb(main):014:0> Konstans = 2
(irb):14: warning: already initialized constant
    Konstans
=> 2
irb(main):015:0> $globalis = "hello"
=> "hello"
irb(main):016:0> @a = "hello"
=> "hello"
```

Egysoros megjegyzést a # szimbólum után tehetünk.

```
irb(main):017:0> # komment
irb(main):018:0* @a
=> "hello"
```

Többsoros megjegyzés tételére két módunk van:

```
irb(main):019:0> =begin
irb(main):020:0= Akarmit
irb(main):021:0= irhatok
irb(main):022:0= ide
irb(main):023:0= =end
```

```
#
# Dokumentacio
#
#
```

Egy állítás jellenzően a következő soremelésig tart, kivéve, ha a sor utolsó lexikai eleme egy operátor, amely lehet az üzenetküldés operátor (. vagy ::) is.

```
irb(main):024:0> 1+2
=> 3
irb(main):025:0> 1+
irb(main):026:0* 2
=> 3
irb(main):027:0> 1
=> 1
irb(main):028:0> +2
=> 2
irb(main):029:0> 1 \
irb(main):030:0* +2
=> 3
irb(main):031:0> "egy".split('g')
=> ["e", "y"]
irb(main):032:0> a = 2.4
=> 2.4
irb(main):033:0> a**2.4
=> 8.175361775184633
```

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):034:0> def f(n)
irb(main):035:1>   2*n
irb(main):036:1> end
=> nil
```

Függvényhívásnál, ami nem más mint egy üzenet küldés egy objektum számára, ameddig a paraméterlista egyértelműen meghatározható, a zárójelek elhagyhatók.

```
irb(main):037:0> f(1+2)+3
=> 9
irb(main):038:0> f (1+2)+3
```

```
=> 12
```

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):039:0> 2.class
=> Fixnum
irb(main):040:0> 1_000_000_000.class
=> Fixnum
irb(main):041:0> 1_000_000_000_000.class
=> Fixnum
irb(main):042:0> 1_000_000_000_000_000.class
=> Fixnum
irb(main):043:0> 1.0.class
=> Float
irb(main):044:0> 3/2
=> 1
irb(main):045:0> -3/2
=> -2
irb(main):046:0> -(3/2)
=> -1
irb(main):047:0> 3.0/2
=> 1.5
```

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.

```
irb(main):048:0> "String".class
=> String
irb(main):049:0> $a = 'lo'
=> "lo"
irb(main):050:0> "Hel#{$a}"
=> "Hello"
irb(main):051:0> 'Hel#{$a}'
=> "Hel\#{$a}"
irb(main):052:0> a = "Hel#{$a}"
=> "Hello"
irb(main):053:0> a[0]
```

```
=> "H"
irb(main):054:0> a[-1]
=> "o"
```

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):055:0> arr = [ "1", 1.0, 1 ]
=> ["1", 1.0, 1]
irb(main):056:0> arr*2
=> ["1", 1.0, 1, "1", 1.0, 1]
irb(main):057:0> arr = [ 12, 38, 12.1]
=> [12, 38, 12.1]
irb(main):058:0> arr.join(":")
=> "12:38:12.1"
```

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

```
irb(main):059:0> h
=> {1=>"egy", "ketto"=>2.0}
irb(main):060:0> h[3] = 3
=> 3
irb(main):061:0> h[3]
=> 3
irb(main):062:0> h
=> {1=>"egy", "ketto"=>2.0, 3=>3}
```

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):063:0> r = 'a'..'f'
=> "a".."f"
irb(main):064:0> r1 = 'a'...'g'
=> "a"..."g"
irb(main):065:0> r.each {|l| print "␣#{l}␣" }
 a  b  c  d  e  f => "a".."f"
irb(main):066:0> r1.each {|l| print "␣#{l}␣" }
 a  b  c  d  e  f => "a"..."g"
```

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):067:0> true
=> true
irb(main):068:0> false
=> false
irb(main):069:0> 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
```

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

```
irb(main):077:0> a = 2
=> 2
irb(main):078:0> if a>1 then puts "nagy" elsif a==1
   then puts "egy"  else puts "kicsi" end
nagy
=> nil
irb(main):079:0> unless a==1 then puts "nemegy"
irb(main):080:1> end
nemegy
=> nil
irb(main):081:0> puts "nemegy" unless a==1
nemegy
=> nil
```

```
irb(main):082:0> case a
irb(main):083:1> when 1
irb(main):084:1> puts "egy"
irb(main):085:1> when 2..10
irb(main):086:1> puts "a tartomanyba esik"
irb(main):087:1> end
a tartomanyba esik
=> nil
irb(main):095:0> case i
irb(main):096:1> when 3
irb(main):097:1> puts "eloszor"
irb(main):098:1> when 3..4
irb(main):099:1> puts "masodszor"
irb(main):100:1> end
eloszor
=> nil
```

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):088:0> i = 1
=> 1
irb(main):089:0> while i < 3 do puts "#{i=i+1}" end
2
3
=> nil
irb(main):090:0> i= 1
=> 1
irb(main):091:0> until i > 3 do puts "#{i=i+1}" end
2
3
4
=> nil
irb(main):092:0> for i in 1..3 do puts "#{i}" end
1
2
3
=> 1..3
irb(main):093:0> i
=> 3
irb(main):094:0> for i in 1..3 do print "#{i}" end
```

```
123=> 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).

Ay 65-66. sorokban bemutatott tartomány típus `each` metódusa is ilyen. A 104-107. sorokban a `t` metódus definíciója kétszer hívja meg a metódushívás paramétereként a 108. sorban átadott blokkot. A blokkban definiált procedúra paraméterezhető, ahogy azt a 109-111. sorok mutatják. A 112. sorban meghívjuk a 109. sorban definiált `t` azonosítójú metódust egy string paraméterrel. A metódus törzsében, vagyis a 110. sorban átadja a vezérlést a 112. sor blokkja törzsének átadva az `a` értéket. A 112. sor procedúrája a paraméterül kapott értékre a lokális `l` azonosítóval hivatkozik, ami a törzsben felhasználható. A törzs végével a vezérlést visszakapja a hívott metódus, vagyis a végrehajtás a 110. sorban, a `yield` után folytatódik.

```
irb(main):101:0> r
=> "a".."f"
irb(main):102:0> r.each {|l| print "␣#{l}␣|" }
 a | b | c | d | e | f |=> "a".."f"
irb(main):103:0> r.each do |l| print "␣#{l}␣" end
 a  b  c  d  e  f => "a".."f"
irb(main):104:0> def t
irb(main):105:1> yield
irb(main):106:1> yield
irb(main):107:1> end
=> nil
irb(main):108:0> t { print "a" }
aa=> nil
irb(main):109:0> def t(a)
irb(main):110:1> yield a
irb(main):111:1> end
=> nil
irb(main):112:0> t('hello') {|l| print l}
hello=> 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 megváltoztat-

ható. Az objektumok egyenlősége vizsgálható érték szerint (==) és referencia szerint (`eql?`) metódus.

```
irb(main):117:0> a = 2
=> 2
irb(main):118:0> b = 2.0
=> 2.0
irb(main):119:0> a == b
=> true
irb(main):120:0> a.eql? b
=> false
irb(main):121:0> a = "hello"
=> "hello"
irb(main):122:0> b = a
=> "hello"
irb(main):123:0> a.eql? b
=> true
irb(main):124:0> a.reverse!
=> "olleh"
irb(main):125:0> b
=> "olleh"
irb(main):126:0> a = 2
=> 2
irb(main):127:0> b = "2"
=> "2"
irb(main):128:0> a==b
=> false
irb(main):129:0> b= 2.0
=> 2.0
```

Az objektumok konvertálhatók más típusúvá. A konverzió történhet automatikusan, mint például a 47. vagy esetleg az 53. sorban történik, vagy explicit módon a `to_s`, `to_i` vagy `to_f` metódusokkal.

```
irb(main):071:0> a = "1"
=> "1"
irb(main):072:0> a.to_i
=> 1
irb(main):073:0> a.to_f
=> 1.0
irb(main):074:0> a = 1.0
=> 1.0
```

```
irb(main):075:0> a.to_s
=> "1.0"
```

Mivel egy azonosító bármilyen típus 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):116:0> a.respond_to? '+'
=> true
```

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. Az üzenetek egymásba ágyazhatóak, lásd a 154. sort.

```
irb(main):187:0> Math::sqrt 4
=> 2.0
irb(main):154:0> A.new.a=3
=> 3
```

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 131-135. sorban egy osztályt definiálunk, amelynek `m` metódusa összeadja a két paraméterül adott számot. A 136. sorban létrehozunk egy példányt az implicit ősosztályból, vagyis az `Object`-ből módon örökölt `new` metódussal, a 137. sorban meghívjuk az `m` metódus két paraméterrel.

```
irb(main):131:0> class A
irb(main):132:1> def m(p1,p2)
irb(main):133:2> p1+p2
irb(main):134:2> end
irb(main):135:1> end
=> nil
irb(main):136:0> a = A.new
=> #<A:0x00000002078148>
irb(main):137:0> a.m 2,3
=> 5
```

A kezdeti viselkedést az `initialize` metódus (felül)definiálásával határozhatjuk meg. A 130-132. 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 127-129. 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 142-144. sorban bővítjük, a 139-141. sorban felüldefiniáljuk a viselkedést.

```
irb(main):138:0> class A
irb(main):139:1> def m(p1,p2)
irb(main):140:2> p1+p2+@a
irb(main):141:2> end
irb(main):142:1> def initialize
irb(main):143:2> @a=2
irb(main):144:2> end
irb(main):145:1> end
=> nil
irb(main):146:0> a = A.new
=> #<A:0x000000020557d8 @a=2>
irb(main):135:0> a.m 2,3
=> 7
```

A példányváltozókhoz setter és getter metódusokkal férhetünk hozzá. A setter jellemzője, hogy a változó azonosítója mögé egy egyenlőségjelet írunk (148-150. sor), a getter pedig maga a változó azonosítója (151-153. sor). Használatukat a 155. és a 156. sor mutatja.

```
irb(main):147:0> class A
irb(main):148:1> def a=(val)
irb(main):149:2> @a = val
irb(main):150:2> end
irb(main):151:1> def a
irb(main):152:2> @a
irb(main):153:2> end
irb(main):154:1> end
=> nil
irb(main):156:0> a.a=3
=> 3
irb(main):157:0> 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 163. és a 169. sor mutatja.

```
irb(main):158:0> class A
irb(main):159:1> attr_accessor :b
irb(main):160:1> end
=> nil
irb(main):161:0> a = A.new
=> #<A:0x00000002022860 @a=2>
irb(main):162:0> a.b=2
=> 2
irb(main):163:0> a.c=3
NoMethodError: undefined method `c=' for #<A:0
    x00000002022860 @a=2, @b=2>
        from (irb):163
        from /usr/bin/irb:12:in `<main>'
irb(main):164:0> class A
irb(main):165:1> attr_reader :d
irb(main):166:1> attr_writer :e
irb(main):167:1> end
=> nil
irb(main):168:0> a = A.new
=> #<A:0x00000002003dc0 @a=2>
irb(main):169:0> a.e
NoMethodError: undefined method `e' for #<A:0
    x00000002003dc0 @a=2>
        from (irb):169
        from /usr/bin/irb:12:in `<main>'
irb(main):170:0> a.d
=> nil
```

Metódust definiálhatunk egy-egy példányra specializálva is, ekkor az a metódus szingletonnak nevezzük. A 171-173. sorban az `a` példányra vonatkozóan definiálunk egy metódust, ami az `A` osztály más példányára már nem hozzáférhető, és elfedi az osztály azonos nevű metódusát.

```
irb(main):171:0> def a.m2(val)
irb(main):172:1>   @a = val
```

```
irb(main):173:1> end
=> nil
irb(main):174:0> a.a
=> 2
irb(main):175:0> a.m2 3
=> 3
irb(main):176:0> a.a
=> 3
irb(main):177:0> b = A.new
=> #<A:0x00000001fdfb00 @a=2>
irb(main):178:0> b.m2 3
NoMethodError: undefined method `m2' for #<A:0
    x00000001fdfb00 @a=2>
        from (irb):178
        from /usr/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):179:0> class A
irb(main):180:1>   def A.m3
irb(main):181:2>     "hello"
irb(main):182:2> end
irb(main):183:1> end
=> nil
irb(main):186:0> 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 188-193. 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 198-199. 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):187:0> class A
irb(main):188:1> def m4(a)
irb(main):189:2> n = a[:n] || 0
irb(main):190:2> m = a[:m] || 1
irb(main):191:2> l = a[:l] || 0
irb(main):192:2> n+m+l
irb(main):193:2> end
irb(main):194:1> end
=> nil
irb(main):195:0> a = A.new
=> #<A:0x00000001ec5c38 @a=2>
irb(main):198:0> h = {:n=>3, :l=>4}
=> {:n=>3, :l=>4}
irb(main):199:0> a.m4 h
=> 8
```

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):200:0> class A
irb(main):201:1> def initialize
irb(main):202:2> @a=2
irb(main):203:2> end
irb(main):204:1> def m(p1, p2)
irb(main):205:2> p1+p2+@a
irb(main):206:2> end
irb(main):207:1> protected
irb(main):208:1> def a
irb(main):209:2> @a
irb(main):210:2> end
irb(main):211:1> private
irb(main):212:1> def a=(val)
irb(main):213:2> @a=val
irb(main):214:2> end
irb(main):215:1> end
=> nil
```

Az osztályok a `<` operátorral specializálhatók. A leszármazott osztály kiegészítheti vagyis specializálhatja, illetve felüldefiniálhatja az ősosztály viselkedését. A 218-222. sorban a `B` osztályt definiáljuk, amely rendelkezik az `A` oszltály összes metódusával, illetve példányváltozójával. A 219-221. sor felüldefiniálja az `a` attribútum setter metódusát, így az másképp fog viselkedni, mint azt a 148. sorban láttuk.

```
irb(main):216:0> class B < A
irb(main):217:1> end
=> nil
irb(main):218:0> class B < A
irb(main):219:1> def a=(val)
irb(main):220:2> @a =  2*val
irb(main):221:2> end
irb(main):222:1> end
=> nil
irb(main):233:0> b = B.new
=> #<B:0x00000002080bb8 @a=2>
irb(main):234:0> b.a=1
=> 1
```

Metódusokhoz hasonlóan operátorok is (felül)definiálhatók egy oszlályon belül (248-250. sor), mivel az operátorok önmaguk is metódushívások, lásd 228. sor.

```
irb(main):247:0> class A
irb(main):248:1> def +(val)
irb(main):249:2> @a+val
irb(main):250:2> end
irb(main):251:1> end
=> nil
irb(main):254:0> a.+2
=> 4
irb(main):255:0> 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 229-236. sorok egy modult definiálnak egy példányváltozóval és egy setter-getter párral. A modulban definiált metódusokkal bármely osztály viselkedését kibővíthetjük (237. sor). Egy osztály tetszőleges számú modult integrálhat magába.

```
irb(main):256:0> module Szin
irb(main):257:1> def szin=(val)
irb(main):258:2> @szin=val
irb(main):259:2> end
irb(main):260:1> def szin
irb(main):261:2> @szin
irb(main):262:2> end
irb(main):263:1> end
=> nil
irb(main):264:0> class A
irb(main):265:1> include Szin
irb(main):266:1> end
=> A
irb(main):267:0> a = A.new
=> #<A:0x00000002008898 @a=2>
irb(main):268:0> a.szin="kek"
=> "kek"
irb(main):269:0> 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.