

A Ruby programozási nyelv Gyakorlat

Kovács Gábor

2016. február 23.

Amire szükségünk lesz a gyakorlat során: egy telepített Ruby környezet¹. 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 szúrunk. 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 szokásos 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 (3. sor, 4. sor)
- lebegőpontos szám (5. sor)
- string, amit megadhatunk ' vagy " szimbólumok között (6. sor)
- reguláris kifejezés (7. sor)
- tömb, ami egy szögletes zárójelek között megadott vesszővel elválasztott lista (10. sor)
- hash, ami kapcsos zárójelek között tartalmaz kulcs-érték párokat => vagy : szimbólummal elválasztva (11. sor)
- tartomány, ami a két szélső értékkel megadott egymás utáni egész értékek halmaza (8-9. sorok)
- szimbólum, ami nem más, mint egy lebutított string, és amelyet hash-ekben előszeretettel alkalmazunk kulcsként (13. sor)
- boolean literálok (14-15. sorok)
- típus nélküli literál (16. sor)
- Ruby 2.0-től létezik a racionális és képzetes literál is, amikkel komplex számokat írhatunk le

```

irb(main):003:0> 1
=> 1
irb(main):004:0> 1_000_000_000_000_000_000
=> 10000000000000000000
irb(main):005:0> 1.1
=> 1.1
irb(main):006:0> "Hello ,_world"
=> "Hello ,_world"
irb(main):007:0> /[a-z]/
=> /[a-z]/
irb(main):008:0> 'a'..'f'
=> "a".."f"
irb(main):009:0> 'a'...'f'
=> "a"..."f"
irb(main):010:0> [ 1, 1.1, 'egy' ]
=> [1, 1.1, "egy"]
irb(main):011:0> { 1 => 1.0, 'egy' => 1 }
=> {1=>1.0, "egy"=>1}
irb(main):013:0> :s
=> :s
irb(main):014:0> true
=> true
irb(main):015:0> false
=> false
irb(main):016:0> 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):026:0> Konstans = 1
=> 1

```

```

irb(main):027:0> Konstans = 2
(irb):27: warning: already initialized constant
  Konstans
(irb):26: warning: previous definition of Konstans was
  here
=> 2
irb(main):028:0> $globalis = 1
=> 1
irb(main):029:0> @peldany = 1
=> 1
irb(main):030:0> @@osztaly = 1
(irb):30: warning: class variable access from toplevel

```

Egy állítás jellemzően a következő soremelésig tart, kivéve, ha a sor utolsó lexikai eleme egy operátor, amely lehet az üzenetküldés operátor (`.` vagy `::`) is. 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. Érdekes, hogy az egész számok osztásánál a Ruby nem 0 felé, hanem mindig mínusz végtelen felé kerekít.

```

irb(main):031:0> 1+2
=> 3
irb(main):032:0> 1+
irb(main):033:0> * 2
=> 3
irb(main):034:0> 1
=> 1
irb(main):035:0> +2
=> 2
irb(main):036:0> 1\
irb(main):037:0> * +2
=> 3
irb(main):038:0> 3/2
=> 1
irb(main):039:0> -3/2
=> -2
irb(main):040:0> 2**3
=> 8

```

Egysoros megjegyzést a `#` szimbólum után tehetünk. Többsoros megjegy-

zés tételére két módunk van, vagy folytatjuk a sor elejére tett # használatát, vagy =begin és =end közé helyezzük a figyelmen kívül hagyandó kódszészletet.

```
irb(main):017:0> # akarmi
irb(main):020:0> =begin
irb(main):021:0= akarmi
irb(main):022:0= is
irb(main):023:0= kerül
irb(main):024:0= ide
irb(main):025:0= =end
```

Rubyban minden objektum, még az elemi típusok is. Kétféle egész létezik Fixnum és Bignum. Azt, hogy egy objektum képes-e egy adott művelet végrehajtására a respond_to? függvénnyel kérdezhajtjuk le, aminek a paramétere egy függvényazonosító, ami szimbólum.

```
irb(main):046:0> 2.class
=> Fixnum
irb(main):047:0> 1_000_000_000_000_000_000.class
=> Fixnum
irb(main):048:0> 1_000_000_000_000_000_000_000.class
=> Bignum
irb(main):049:0> 1.0.class
=> Float
irb(main):050:0> "string".class
=> String
irb(main):045:0> a.respond_to? "+"
=> true
irb(main):046:0> a.respond_to? "to_s"
=> true
irb(main):047:0> a
=> "egy"
irb(main):049:0> a.class
=> String
irb(main):050:0> a.respond_to?("to_s")
=> true
irb(main):136:0> 1.methods
=> [:to_s, :inspect, :-@, :+, :-, :*, :/, :div, :%, :
  modulo, :divmod, :fdiv, :**, :abs, :magnitude, :==,
  :===, :<=>, :>, :>=, :<, :<=, :~, :&, :|, :^, :[],
  :<<, :>>, :to_f, :size, :bit_length, :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?, :
remainder, :real?, :nonzero?, :step, :quo, :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, :
singleton_method, :define_singleton_method, :
object_id, :to_enum, :enum_for, :equal?, :!, :!=, :
instance_eval, :instance_exec, :__send__, :__id__]
irb(main):138:0> 1.methods.include? '+'
=> false
irb(main):139:0> 1.methods.include? :+
=> true

```

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 figyelniük 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 az objektuma 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):041:0> def f(n)
irb(main):042:1>   2*n

```

```

irb(main):043:1> end
=> :f
irb(main):044:0> f(1+2)+3
=> 9
irb(main):045:0> 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> $a = 'lo'
=> "lo"
irb(main):052:0> "Hel#{ $a }"
=> "Hello"
irb(main):053:0> 'Hel#{ $a }'
=> "Hel\#{ $a }"
irb(main):054:0> a = "Hel#{ $a }"
=> "Hello"
irb(main):055:0> a[0]
=> "H"
irb(main):056:0> a[-1]
=> "o"
irb(main):057:0> a[0..1]
=> "He"

```

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):058:0> arr = [ 1, 1.1, 'egy' ]
=> [1, 1.1, "egy"]
irb(main):059:0> arr*2
=> [1, 1.1, "egy", 1, 1.1, "egy"]
irb(main):060:0> arr << "Egy"
=> [1, 1.1, "egy", "Egy"]
irb(main):061:0> arr
=> [1, 1.1, "egy", "Egy"]
irb(main):062:0> arr[0]
=> 1
irb(main):063:0> arr.join(':')
=> "1:1.1:egy:Egy"
irb(main):064:0> h = { 1 => 1.0, 1.0 => 'egy' }
=> {1=>1.0, 1.0=>"egy"}
irb(main):065:0> h[1]
=> 1.0
irb(main):066:0> h[1.0]
=> "egy"
irb(main):067:0> h["ketto"] = 2
=> 2
irb(main):068:0> h
=> {1=>1.0, 1.0=>"egy", "ketto"=>2}
```

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):069:0> r = 'a'..'f'
=> "a".."f"
irb(main):070:0> r.each { |l| print "_#{l}_" }
a b c d e f => "a".."f"
irb(main):071:0> r = 'a'...'f'
=> "a"..."f"
irb(main):072:0> 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 szintaxisukban térnek el a C-ben megismertektől: `if/unless`, illetve `case`. Az `if` szerkezet formálisan:

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

. A `then` minden esetben helyettesíthető soremeléssel vagy pontosvessző 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):073:0> a = 2
=> 2
irb(main):074:0> if a > 2 then puts "nagy" elsif a==1
  then puts "egy" else puts "kicsi" end
kicsi
=> nil
irb(main):075:0> if a > 2
irb(main):076:1> puts "nagy"
irb(main):077:1> elsif a==1
irb(main):078:1> puts "egy"
irb(main):079:1> else
irb(main):080:1* puts "kicsi"
irb(main):081:1> end
kicsi
```

```

=> nil
irb(main):082:0> if a > 2 then puts "nagy" end
=> nil
irb(main):083:0> puts "nagy" if a > 1
nagy
=> nil
irb(main):084:0> puts "nagy" unless a > 1
=> nil
irb(main):085:0> case a
irb(main):086:1> when 1
irb(main):087:1> puts "egy"
irb(main):088:1> when 2..10
irb(main):089:1> puts "a_tartomanyban_van"
irb(main):090:1> end
a tartomanyban van
=> nil
irb(main):091:0> a = 3
=> 3
irb(main):092:0> case a
irb(main):093:1> when 3
irb(main):094:1> puts 3
irb(main):095:1> when 3..4
irb(main):096:1> puts "targyomany"
irb(main):097:1> end
3
=> 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):098:0> i = 1
=> 1
irb(main):099:0> while i < 3 do puts "#{i=_i+1}" end
2
3
=> nil
irb(main):100:0> i = 1
=> 1
irb(main):101:0> until i > 3 do puts "#{i=_i+1}" end
2
3

```

```

4
=> nil
irb(main):102:0> i = 1..3
=> 1..3
irb(main):103:0> for a in i do print "#{a}" end
123=> 1..3
irb(main):105:0> for a in arr do print "#{a}" end
11.1egyEgy=> [1, 1.1, "egy", "Egy"]
irb(main):106:0> for a in h do print "#{a}" end
[1, 1.0][1.0, "egy"]["ketto", 2]=> {1=>1.0, 1.0=>"egy",
  "ketto"=>2}

```

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):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 === b
=> true
irb(main):121:0> a.eql? b
=> false
irb(main):123:0> a = 'hello'
=> "hello"
irb(main):124:0> a.reverse!
=> "olleh"
irb(main):125:0> a
=> "olleh"
irb(main):126:0> "1.0" == 1
=> 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` értéké.

```
irb(main):127:0> "1.0".to_i == 1
=> true
irb(main):128:0> "1.0" == 1.to_s
=> false
irb(main):129:0> 1.to_s
=> "1"
irb(main):130:0> "1.0" == 1.0.to_s
=> true
irb(main):131:0> "1.0" == 1.to_f.to_s
=> true
irb(main):132:0> if a then puts "#{a}" end
olleh
=> nil
```

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):142:0> a::to_f
=> 1.0
irb(main):143:0> a.respond_to? "*"
=> true
irb(main):144:0> a.respond_to? "to_i"
=> true
```

Blokkot kétféle szintaxszissal definiálhatunk: vagy `do-end` párok között vagy kapcsos zárójelekben. A blokkoknak speciális szerepük is lehet Ruby-ban. 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 procedura 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, ami 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
=> :t
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
=> :t
irb(main):116:0> t('hello') { |l| print l }
hello=> nil
```

A Ruby programnyelv objektumorientált, építőkövei 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 `A` azonosítójú osztályt definiálunk, amelyet a 143. sorban példányosítunk. Az osztályt a 147-151. sorban kibővítjük egy `m` 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 147-151. sorban bővítjük, a 153-157. sorban felüldefiniáljuk a viselkedést. A 143. sorban létrehozunk egy példányt az implicit őszosztályból, vagyis az `Object`-ből módon örökölt `new` metódussal, a 152. sorban meghívjuk az `m` metódus két paraméterrel.

```

irb(main):141:0> class A
irb(main):142:1> end
=> nil
irb(main):143:0> a = A.new
=> #<A:0x0000000282e388>
irb(main):144:0> a.methods
=> [:nil?, :==, :=~, :!~, :eql?, :hash, :<=>, :class,
    :singleton_class, :clone, :dup, :taint, :tainted?, :
    untaint, :untrust, :untrusted?, :trust, :freeze, :
    frozen?, :to_s, :inspect, :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, :
    singleton_method, :define_singleton_method, :
    object_id, :to_enum, :enum_for, :==, :equal?, :!,
    :!=, :instance_eval, :instance_exec, :__send__, :
    __id__]
irb(main):145:0> class A < Object
irb(main):146:1> end
=> nil
irb(main):147:0> class A
irb(main):148:1> def m(p1, p2)
irb(main):149:2> p1+ p2
irb(main):150:2> end
irb(main):151:1> end
=> :m
irb(main):152:0> a.m 1,2
=> 3
irb(main):153:0> class A
irb(main):154:1> def m(p1, p2)
irb(main):155:2> p1 + p2 + @a

```

```
irb(main):156:2> end
irb(main):157:1> end
=> :m
```

A kezdeti viselkedést az `initialize` metódus (felül)definiálásával határozhatjuk meg. A 161-163. 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 154-156. sorban felüldefiniált `m` metódusban megteesszük.

```
irb(main):160:0> class A
irb(main):161:1> def initialize
irb(main):162:2> @a = 2
irb(main):163:2> end
irb(main):164:1> end
=> :initialize
irb(main):165:0> a = A.new
=> #<A:0x000000028a9880 @a=2>
irb(main):183:0> a.m 2,3
=> 7
```

A példányváltozókhoz setter és getter metódusokkal férhetünk hozzá. A setter jellemzője, hogy a változó azonosítója mögé egy egyenlőségjelet írunk (178-180. sor), a getter pedig maga a változó azonosítója (175-177. sor). Használatukat a 183. és a 184. sor mutatja.

```
irb(main):171:0> class A
irb(main):175:1> def a
irb(main):176:2> @a
irb(main):177:2> end
irb(main):178:1> def a=(val)
irb(main):179:2> @a = val
irb(main):180:2> end
irb(main):181:1> end
irb(main):182:0> a = A.new
=> #<A:0x00000002804088 @a=2>
irb(main):183:0> a.a
=> 2
irb(main):184:0> a.a=3
=> 3
irb(main):185:0> 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 197-200. sorok mutatják.

```
irb(main):186:0> class A
irb(main):187:1> attr_accessor :b
irb(main):188:1> end
=> nil
irb(main):189:0> a = A.new
=> #<A:0x000000027f3698 @a=2>
irb(main):190:0> a.b = 2
=> 2
irb(main):191:0> a.b
=> 2
irb(main):192:0> class A
irb(main):193:1> attr_reader :c
irb(main):194:1> attr_writer :d
irb(main):195:1> end
=> nil
irb(main):196:0> a = A.new
=> #<A:0x000000027e22f8 @a=2>
irb(main):197:0> a.c
=> nil
irb(main):198:0> a.d = 2
=> 2
irb(main):199:0> a.d
NoMethodError: undefined method 'd' for #<A:0
  x000000027e22f8@a=2,@d=2>
~~~~~
from (irb):199
~~~~~
from /usr/bin/irb:11:in '<main>'
irb(main):200:0> a.c = 2
NoMethodError: undefined method 'c=' for #<A:0
  x000000027e22f8@a=2,@d=2>
~~~~~
from (irb):200
~~~~~
from /usr/bin/irb:11:in '<main>'
```

Metódust definiálhatunk egy-egy példányra specializálva is, ekkor az a

metódus szingletonnak nevezzük. A 201-203. 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):201:0> def a.m2(val)
irb(main):202:1> @a = val
irb(main):203:1> end
=> :m2
irb(main):204:0> a.m2 3
=> 3
irb(main):205:0> a.a
=> 3
irb(main):206:0> a.m2 4
=> 4
irb(main):207:0> a.a
=> 4
irb(main):208:0> b = A.new
=> #<A:0x000000026c90d8 @a=2>
irb(main):209:0> b.m2 4
NoMethodError: undefined method 'm2' for #<A:0
  x000000026c90d8@a=2>
~~~~~from (irb):209
~~~~~from /usr/bin/irb:11:in '<main>'
```

Ruby-ban is definiálhatók osztálymetódusok (215-217. sor), amelyek az osztály példányainak létezése nélkül is meghívhatók, illetve közősek az összes példányra vonatkozóan. Osztálymetódus az osztálydefiníció belüli az aktuális példány referenciájára (`self`) vonatkozó szingleton metódus definíciójával azonos hatást érhetünk el (222-224. sor). Osztályváltozók (`@@` prefixú azonosítóval rendelkező változók) csakis osztálymetódusokon belül használhatók, és az első használatkor inicializálандók.

```
irb(main):214:0> class A
irb(main):215:1>   def A.m3
irb(main):216:2>     "hello"
irb(main):217:2>   end
irb(main):218:1> end
=> :m3
irb(main):219:0> A.m3
=> "hello"
irb(main):221:0> class A
irb(main):222:1>   def self.m4
```

```

irb(main):223:2 > "hello"
irb(main):224:2 > end
irb(main):225:1 > end
=> :m4
irb(main):230:0 > class A
irb(main):231:1 > def A.m3
irb(main):232:2 > @@a = 2
irb(main):233:2 > puts "#{@@a}"
irb(main):234:2 > end
irb(main):235:1 > end
=> :m3
irb(main):236:0 > A.m3
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 238-243. 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 inicializáljuk. A másik két lokális változó definíciója hasonlóképp történik. A hash argumentummal definiált metódusok hívására a 247. 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):237:0 > class A
irb(main):238:1 > def m5(a)
irb(main):239:2 > n = a[:n] || 0
irb(main):240:2 > m = a[:m] || 1
irb(main):241:2 > l = a[:l] || 0
irb(main):242:2 > l+m+n
irb(main):243:2 > end
irb(main):244:1 > end
=> :m5
irb(main):245:0 > h
=> {1=>1.0, 1.0=>"egy", "ketto"=>2}
irb(main):246:0 > h = { :n=> 3, :l => 4}
=> {:n=>3, :l=>4}
irb(main):247:0 > a.m5 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 osztálydefinícióban a láthatósági módosítók közötti blokkban lévő összes metódus az aktuális láthatósággal bír.

```
irb(main):261:0> class A
irb(main):262:1> def initialize
irb(main):263:2> @a = 2
irb(main):264:2> end
irb(main):265:1> def m(p1,p2)
irb(main):266:2> p1+p2+@a
irb(main):267:2> end
irb(main):268:1> end
=> :m
irb(main):269:0> class A
irb(main):270:1> protected
irb(main):271:1> def a
irb(main):272:2> @a
irb(main):273:2> end
irb(main):274:1> end
=> :a
irb(main):275:0> class A
irb(main):276:1> private
irb(main):277:1> def a=(val)
irb(main):278:2> @a = val
irb(main):279:2> end
irb(main):280:1> end
=> :a=
irb(main):281:0> a = A.new
=> #<A:0x00000002882050 @a=2>
irb(main):282:0> class B < A
irb(main):283:1> end
=> nil
irb(main):284:0> b = B.new
=> #<B:0x00000002876818 @a=2>
irb(main):285:0> a.a
NoMethodError: protected method 'a' called for #<A:0
x00000002882050 @a=2>
```

```

~~~~~from_(irb):285
~~~~~from_/usr/bin/irb:11:in_<'<main>'>
irb(main):286:0> a.a=1
NoMethodError: private method 'a='_called_for_#<A:0
  x00000002882050_@a=2>
~~~~~from_(irb):286
~~~~~from_/usr/bin/irb:11:in_<'<main>'>
irb(main):287:0> b.a
NoMethodError: protected method 'a'_called_for_#<B:0
  x00000002876818_@a=2>
~~~~~from_(irb):287
~~~~~from_/usr/bin/irb:11:in_<'<main>'>
irb(main):288:0> b.a=1
NoMethodError: private method 'a='_called_for_#<B:0
  x00000002876818_@a=2>
~~~~~from_(irb):288
~~~~~from_/usr/bin/irb:11:in_<'<main>'>
irb(main):289:0> b.m 1,2
=> 5
irb(main):290:0> class B < A
irb(main):291:1> def a=(val)
irb(main):292:2> @a = 2*val
irb(main):293:2> end
irb(main):294:1> end
=> :a=
irb(main):295:0> b = B.new
=> #<B:0x0000000285a910 @a=2>
irb(main):296:0> b.a=1
=> 1
irb(main):297:0> b.m 1,2
=> 5

```

Az osztályok a < operátorral specializálhatók. A leszármazott osztály kiegészítheti vagyis specializálhatja, illetve felüldefiniálhatja az őosztály viselkedését. A 290-294. sorban a B osztályt definiáljuk, amely rendelkezik az A osztály összes **public** és **protected** metódusával, illetve példányváltozójával. A 291-293. sor felüldefiniálja az a attribútum setter metódusát, így az másképp fog viselkedni, mint azt a 276. sorban láttuk, B példányaira ismét elérhető lesz. A felüldefiniált metódus láthatósága eltérhet egy leszármazott osztályban.

Metódusokhoz hasonlóan operátorok is (felül)definiálhatók egy osztályon

belül (299-301. sor), mivel az operátorok önmaguk is metódushívások, lásd 305. sor.

```
irb(main):298:0> class A
irb(main):299:1> def +(val)
irb(main):300:2> @a + val
irb(main):301:2> end
irb(main):302:1> end
=> :+
irb(main):303:0> a = A.new
=> #<A:0x000000028395a8 @a=2>
irb(main):304:0> a.+(2)
=> 4
irb(main):305:0> 2.+(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 közősek. 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 példánya (273. sor), viszont létezhet példány, ami rendelkezik a modulban definiált viselkedéssel.

A 306-313. 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):306:0> module Szin
irb(main):307:1> def szin=(val)
irb(main):308:2> @szin = val
irb(main):309:2> end
irb(main):310:1> def szin
irb(main):311:2> @szin
irb(main):312:2> end
irb(main):313:1> end
=> :szin
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):315:0> class A
irb(main):316:1> include Szin
irb(main):317:1> end
=> A
irb(main):318:0> a = A.new
=> #<A:0x00000002893a30 @a=2>
irb(main):319:0> a.szin = 'kek'
=> "kek"
irb(main):320:0> 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.

```

irb(main):321:0> module M
irb(main):322:1> class A
irb(main):323:2> def initialize
irb(main):324:3> @a = 3
irb(main):325:3> end
irb(main):326:2> def a
irb(main):327:3> @a
irb(main):328:3> end
irb(main):329:2> end
irb(main):330:1> end
=> :a
irb(main):331:0> M::A.new
=> #<M::A:0x000000028aa2f8 @a=3>
irb(main):332:0> A.new
=> #<A:0x0000000287ff08 @a=2>

```