

# A Ruby programozási nyelv Gyakorlat

Kovács Gábor

2015. szeptember 15.

Amire szükségünk lesz a gyakorlat során: egy telepített Ruby környezet<sup>1</sup>. 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 szúrni. 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-

---

<sup>1</sup><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 (6-7. sorok)
- reguláris kifejezés (8. sor)
- tömb, ami egy szögletes zárójelek között megadott vesszővel elválasztott lista (9. sor)
- hash, ami kapcsos zárójelek között tartalmaz kulcs-érték párokat => vagy : szimbólummal elválasztva (10. sor)
- tartomány, ami a két szélső értékkel megadott egymás utáni egész értékek halmaza (11-12. sorok)
- szimbólum, ami nem más, mint egy lebutított string, és amelyet hash-ekben előszeretettel alkalmazunk kulcsként (13. sor)
- boolean literál (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
=> 1000000000000000000
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> { 1 => "egy", 2.0 => "ketto" }
=> {1=>"egy", 2.0=>"ketto"}
irb(main):011:0> 1..6
=> 1..6
irb(main):012:0> 1...6
=> 1...6
irb(main):013:0> :egy
=> :egy
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):017:0> a = 1
=> 1
irb(main):018:0> @a
=> nil
irb(main):019:0> @@a
NameError: uninitialized class variable @@a in Object
      from (irb):19
      from /usr/bin/irb:12:in '<main>'
irb(main):020:0> _@a_=1
=>_1
irb(main):021:0> _$a_=1
=>_1
irb(main):022:0> _Konstans_=1
=>_1
irb(main):023:0> _Konstans_=2
(irb):23: warning: already initialized constant
  Konstans
=>_2

```

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):024:0> 3/2
=> 1
irb(main):025:0> -3/2
=> -2
irb(main):026:0> 2**3
=> 8
irb(main):027:0> 3 +
irb(main):028:0> 2
=> 5
irb(main):029:0> 3
=> 3
irb(main):030:0> +2
=> 2

```

```
irb(main):031:0> 3 \
irb(main):032:0* +2
=> 5
```

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):033:0> a = 1 # ez egy komment
=> 1
irb(main):034:0> =begin
irb(main):035:0= Akarmi
irb(main):036:0= van
irb(main):037:0= itt
irb(main):038:0= =end
```

Rubyban minden objektum, még az elemi típusok is. Kétféle egész létezik Fixnum és Bignum.

```
irb(main):039:0> a
=> 1
irb(main):040:0> a = "egy"
=> "egy"
irb(main):041:0> a.class
=> String
irb(main):042:0> 1_000_000_000_000_000_000.class
=> Fixnum
irb(main):043:0> 1_000_000_000_000_000_000_000.class
=> Bignum
irb(main):044:0> 1.0.class
=> Float
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
```

Az azonosítók másik csoportja a függvény-, vagy másnéven metódusazonosítók. Függvényt a `def` kulcsszó után definiálhatunk a függvény azonosítója (51.sor), a formális paraméterlista és a törzs megadásával. A függvény törzse a `def`-fel egy szinten lévő `end`-ig tart (53.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 figyelniünk 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, 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):051:0> def f(n)
irb(main):052:1>   n*2
irb(main):053:1> end
=> nil
irb(main):054:0> f("Haho")
=> "HahoHaho"
irb(main):055:0> f(2)
=> 4
irb(main):056:0> f(1+2)+3
=> 9
irb(main):057:0> 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 végétől számol vissza, tartomány megadása esetén részstringet kapunk vissza.

```
irb(main):065:0> $a = 'lo'
=> "lo"
irb(main):067:0> "Hel#{$a}"
=> "Hello"
irb(main):068:0> 'Hel#{$a}'
=> "Hel\#{$a}"
irb(main):069:0> $a = '"'
=> "\""
irb(main):070:0> 'Hel#{$a}'
=> "Hel\#{$a}"
irb(main):071:0> "Hel#{$a}"
=> "Hel\""
irb(main):072:0> a[0]
=> "e"
irb(main):073:0> a
=> "egy"
irb(main):074:0> a[2]
=> "y"
irb(main):075:0> a[-1]
=> "y"
irb(main):076:0> a[-3]
=> "e"
```

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):077:0> arr = [ 1, 1.0, "egy" ]
=> [1, 1.0, "egy"]
irb(main):078:0> arr[1]
=> 1.0
irb(main):079:0> arr[-1]
=> "egy"
irb(main):080:0> arr*2
=> [1, 1.0, "egy", 1, 1.0, "egy"]
irb(main):081:0> arr.join(':')
=> "1:1.0:egy"
irb(main):082:0> h = { 1 => "egy", 2.0 => "ketto" }
```

```

=> {1=>"egy", 2.0=>"ketto"}
irb(main):083:0> h[1]
=> "egy"
irb(main):084:0> h["harom"] = 3.0
=> 3.0
irb(main):085:0> h
=> {1=>"egy", 2.0=>"ketto", "harom"=>3.0}
irb(main):086:0> arr << 'egy'
=> [1, 1.0, "egy", "egy"]

```

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):087:0> r = 'a'..'f'
=> "a".."f"
irb(main):088:0> r1 = 'a'...'g'
=> "a"..."g"
irb(main):089:0> r.each do |l| print " #{l}" end
a b c d e f => "a".."f"
irb(main):090:0> r1.each do |l| print " #{l}" end
a b c d e f => "a"..."g"
irb(main):091:0> a = 2
=> 2

```

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 sosemeléssel 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):092:0> if a > 1 then puts "nagy" elsif a==1
  then puts "egy" else puts "kicsi" end
nagy
=> nil
irb(main):093:0> puts "nagy" if a > 1
nagy
=> nil
irb(main):094:0> puts "nagy" unless a <= 1
nagy
=> nil
irb(main):095:0> a
=> 2
irb(main):096:0> case
irb(main):097:1* when 1
irb(main):098:1> puts "egy"
irb(main):099:1> when 2..10
irb(main):100:1> puts "a_tartomanyban_van"
irb(main):101:1> end
egy
irb(main):116:0> i
=> 1
irb(main):117:0> i =3
=> 3
irb(main):118:0> case i
irb(main):119:1> when 3
irb(main):120:1> puts "eloszor "
irb(main):121:1> when 3..4
irb(main):122:1> puts "masodszor "
```

```
irb(main):123:1> 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):116:0> i = 1
=> 1
irb(main):117:0> while i < 3 do puts "#{i=i+1}" end
2
3
=> nil
irb(main):118:0> until i > 3 do puts "#{i=i+1}" end
4
=> nil
irb(main):119:0> i = 1
=> 1
irb(main):120:0> until i > 3 do puts "#{i=i+1}" end
2
3
4
=> nil
irb(main):121:0> for i in 1..3 do puts "#{i=i+1}" end
2
3
4
=> 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 `equal?` 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):132:0> 1 == 1.0
=> true
irb(main):133:0> 1.eql? 1.0
=> false
irb(main):134:0> 1 == 1..10
```

```

ArgumentError: bad value for range
      from (irb):134
      from /usr/bin/irb:12:in '<main>'
irb(main):135:0> a=2
=> 2
irb(main):136:0> b=2.0
=> 2.0
irb(main):137:0> a==b
=> true
irb(main):138:0> a.eql?b
=> false
irb(main):139:0> a="hello"
=> "hello"
irb(main):140:0> b=a
=> "hello"
irb(main):141:0> a.reverse
=> "olleh"
irb(main):142:0> a
=> "hello"
irb(main):143:0> a.reverse!
=> "olleh"
irb(main):144:0> b
=> "olleh"
irb(main):145:0> a
=> "olleh"
irb(main):146:0> a=2
=> 2
irb(main):147:0> b="2"
=> "2"
irb(main):148:0> a==b
=> false
irb(main):131:0> i=2
=> 2
irb(main):132:0> case i
irb(main):133:1> when 1..5
irb(main):134:1> print "igen"
irb(main):135:1> end
igen=> nil

```

Az objektumok konvertálhatók más típusúvá. A konverzió történhet automatikusan, vagy explicit módon a `to_s`, `to_i` vagy `to_f` stb. metódusok-

kal.

```
irb(main):147:0> b = "2"  
=> "2"  
irb(main):149:0> b.to_i  
=> 2  
irb(main):150:0> b.to_f  
=> 2.0
```

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):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 szintakszissal 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 122. sorban bemutatott tartomány típus `each` metódusa is ilyen. A 123-126. sorokban a `t` metódus definíciója meghívja a metódushívás paramétereként a 127. sorban átadott blokkot. A blokkban definiált procedura paraméterezhető, ahogy azt a 128-130. sorok mutatják. A 131. sorban meghívjuk a 128. sorban definiált `t` azonosítójú függvényt egy string paraméterrel. A függvény törzsében, vagyis a 129. sorban átadja a vezérlést a 131. sor blokkja törzsének átadva az `a` értéket. A 131. 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 129. sorban, a `yield` után folytatódik. A `yield`-et tartalmazó metódushívás blokk argumentuma nem hagyható el! Egy függvénynek így egy blokk adható át paraméterül.

```
irb(main):122:0> r.each do |l| print "_#{l}_" end
```

```

a b c d e f => "a".."f"
irb(main):123:0> def t
irb(main):124:1> yield
irb(main):125:1> yield
irb(main):126:1> end
=> nil
irb(main):127:0> t { print "a" }
aa=> nil
irb(main):128:0> def t(a)
irb(main):129:1> yield a
irb(main):130:1> end
=> nil
irb(main):131:0> 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 136. 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):136:0> 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 161-162. sorban egy osztályt definiálunk, amelyet a 168. sorban példányosítunk. Az osztályt a 163-167. 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 164-166. sorban bővítjük, a 178-180. sorban felüldefiniáljuk a viselkedést. A 168. sorban létrehozunk egy példányt az implicit őosztályból, vagyis az `Object`-ből módon örökölt `new` metódussal, a 173. sorban meghívjuk az `m` metódus két paraméterrel. Az osztálydefiníción a végzett módosításnak nincs hatása a korábban létrehozott példányra, az új viselkedés csak a módosított definíció után létrehozott példányokra vonatkozik.

```

irb(main):161:0> class A

```

```

irb(main):162:1> end
=> nil
irb(main):163:0> class A
irb(main):164:1> def m(p1,p2)
irb(main):165:2> p1+p2
irb(main):166:2> end
irb(main):167:1> end
=> nil
irb(main):168:0> a = A.new
=> #<A:0x000000011b0c28>
irb(main):173:0> a.m 2,3
=> 5

```

A kezdeti viselkedést az `initialize` metódus (felül)definiálásával határozhatjuk meg. A 175-177. 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 178-180. sorban felüldefiniált `m` metódusban megteesszük.

```

irb(main):174:0> class A
irb(main):175:1> def initialize
irb(main):176:2> @a = 2
irb(main):177:2> end
irb(main):178:1> def m(p1,p2)
irb(main):179:2> @a + p1+p2
irb(main):180:2> end
irb(main):181:1> end
=> nil
irb(main):182:0> a = A.new
=> #<A:0x000000015ab990 @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 (189-191. sor), a getter pedig maga a változó azonosítója (192-194. sor). Használatukat a 197. és a 198. sor mutatja.

```

irb(main):188:0> class A
irb(main):189:1> def a=(val)
irb(main):190:2> @a = val
irb(main):191:2> end

```

```

irb(main):192:1> def a
irb(main):193:2> @a
irb(main):194:2> end
irb(main):195:1> end
=> nil
irb(main):196:0> a = A.new
=> #<A:0x00000001574288 @a=2>
irb(main):197:0> a.a=3
=> 3
irb(main):198:0> a
=> #<A:0x00000001574288 @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 205-208. sorok mutatják.

```

irb(main):199:0> class A
irb(main):200:1> attr_accessor :b
irb(main):201:1> attr_reader :c
irb(main):202:1> attr_writer :d
irb(main):203:1> end
=> nil
irb(main):204:0> a = A.new
=> #<A:0x0000000152a0e8 @a=2>
irb(main):205:0> a.d
NoMethodError: undefined method 'd' for #<A:0
  x0000000152a0e8@a=2>
~~~~~
from (irb):205
~~~~~
from /usr/bin/irb:12:in '<main>'
irb(main):206:0> a.d=1
=> 1
irb(main):207:0> a.c
=> nil
irb(main):208:0> a.c = 1
NoMethodError: undefined method 'c' for #<A:0
  x0000000152a0e8@a=2,@d=1>
~~~~~
from (irb):208
~~~~~
from /usr/bin/irb:12:in '<main>'

```

Metódust definiálhatunk egy-egy példányra specializálva is, ekkor az a metódus szingletonnak nevezzük. A 220-222. 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):219:0> a
=> #<A:0x000000011c9bb0 @a=2>
irb(main):220:0> def a.m3(val)
irb(main):221:1>   @a = val
irb(main):222:1> end
=> nil
irb(main):223:0> a.a
=> 2
irb(main):224:0> a.m3 3
=> 3
irb(main):225:0> a.a
=> 3
irb(main):226:0> b = A.new
=> #<A:0x0000000162bf50 @a=2>
irb(main):227:0> b.m3 3
NoMethodError: undefined method 'm3' for #<A:0
  x0000000162bf50@a=2>
~~~~~from (irb):227
~~~~~from /usr/bin/irb:12:in '<main>'
```

Ruby-ban is definiálhatók osztálymetódusok (210-212.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.

```
irb(main):209:0> class A
irb(main):210:1>   def A.m2(val)
irb(main):211:2>     @@a = val
irb(main):212:2>   end
irb(main):213:1> end
=> nil
irb(main):214:0> A.m2(3)
=> 3
irb(main):201:0> self
=> main
irb(main):228:0> class A
```



```

irb(main):229:1 > def self.m4(val)
irb(main):230:2 >   @@a = val
irb(main):232:2 > end
irb(main):233:1 > end
=> nil

```

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 235-240. 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 244. 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):234:0 > class A
irb(main):235:1 > def m5(a)
irb(main):236:2 >   n = a[:n] || 0
irb(main):237:2 >   m = a[:m] || 1
irb(main):238:2 >   l = a[:l] || 0
irb(main):239:2 >   n+m+l
irb(main):240:2 > end
irb(main):241:1 > end
=> nil
irb(main):242:0 > a = A.new
=> #<A:0x000000015872e8 @a=2>
irb(main):243:0 > h = { n: 3, l: 4 }
=> {:n=>3, :l=>4}
irb(main):244:0 > 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):251:0> class A
irb(main):252:1>   def initialize
irb(main):253:2>     @a = 2
irb(main):254:2>   end
irb(main):255:1>   def m(p1,p2)
irb(main):256:2>     p1+p2+@a
irb(main):257:2>   end
irb(main):258:1>   protected
irb(main):259:1>   def a
irb(main):260:2>
irb(main):261:2*     @a
irb(main):262:2>   end
irb(main):263:1>   private
irb(main):264:1>   def a=(val)
irb(main):265:2>     @a = val
irb(main):266:2>   end
irb(main):267:1> end
=> nil
irb(main):234:0> a = A.new
=> #<A:0x000000025baac0 @a=2>
irb(main):268:0> class B < A
irb(main):269:1> end
=> nil
irb(main):270:0> class B
irb(main):271:1>   def a=(val)
irb(main):272:2>     @a = 2*val
irb(main):273:2>   end
irb(main):274:1> end
=> nil
irb(main):275:0> b = B.new
=> #<B:0x000000011b69e8 @a=2>
irb(main):276:0> b.a=1
=> 1
irb(main):277:0> a = A.new
=> #<A:0x000000011a58f0 @a=2>
irb(main):278:0> a.a=1
NoMethodError: private method 'a='_called_for_#<A:0
  x000000011a58f0_@a=2>
  ~~~~~_from_(irb):278
  ~~~~~_from_/usr/bin/irb:12:in_'<main>'

```

Az osztályok a < operátorral specializálhatók. A leszármazott osztály kiegészítheti vagyis specializálhatja, illetve felüldefiniálhatja az őosztály viselkedését. A 270-274. sorban a B osztályt definiáljuk, amely rendelkezik az A osztály összes `public` és `protected` metódusával, illetve példányváltozójával. A 271-273. sor felüldefiniálja az `a` attribútum setter metódusát, így az másképp fog viselkedni, mint azt a 276. sorban láttuk, B példányaira ismét elérhető lesz.

Metódusokhoz hasonlóan operátorok is (felül)definiálhatók egy osztályon belül (280-282. sor), mivel az operátorok önmaguk is metódushívások, lásd 285. sor.

```
irb(main):279:0> class A
irb(main):280:1>   def +(val)
irb(main):281:2>     @a+val
irb(main):282:2>   end
irb(main):283:1> end
=> nil
irb(main):284:0> a = A.new
=> #<A:0x000000015c34f0 @a=2>
irb(main):285:0> a.+(2)
=> 4
irb(main):286:0> 2.+(2)
=> 4
irb(main):287:0> 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 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 példánya (273. sor), viszont létezhet példány, ami rendelkezik a modulban definiált viselkedéssel.

A 293-300. sorok egy modult definiálnak egy példányváltozóval és egy setter-getter párral. A modul nem példányosítható, viszont a modulban definiált metódusokkal bármely osztály viselkedését kibővíthetjük (303.) az `include` kulcsszóval példánymetódusként, az `extend` kulcsszóval osztálymetódusként. Egy osztály tetszőleges számú modult integrálhat magába.

```
irb(main):293:0> module Szin
irb(main):294:1> def szin=(val)
```

```

irb(main):295:2> @szin= val
irb(main):296:2> end
irb(main):297:1> def szin
irb(main):298:2> @szin
irb(main):299:2> end
irb(main):300:1> end
=> nil
irb(main):301:0> s = Szin.new
NoMethodError: undefined method 'new' for Szin:Module
~~~~~from (irb):301
~~~~~from /usr/bin/irb:12:in '<main>'
irb(main):302:0> class A
irb(main):303:1> include Szin
irb(main):304:1> end
=> A
irb(main):305:0> a = A.new
=> #<A:0x0000000156f378 @a=2>
irb(main):306:0> a.szin = 'kek'
=> "kek"
irb(main):307:0> a.szin
=> "kek"

```

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

```

irb(main):288:0> module M
irb(main):289:1> class A
irb(main):290:2> end
irb(main):291:1> end
=> nil
irb(main):292:0> a = M::A.new
=> #<M::A:0x000000015a6e18>

```