A Ruby programozási nyelv
Gyakorlat

Kovács Gábor

2016. 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 a 2.0-s verzió előtt az ASCII, ha ékezetes karaktereket akarunk írni a forrásba, ezt minden az elejére be kell szúrnunk. Ruby 2.0-tól UTF-8 lett az alapértelmezett. A gyakorlaton 1.9.3-as verziót használunk a félév során, ezért erre még szükségünk van. A fájl következő része azon függvénykönyvtárakat adja meg, amelyekből osztályokat kívánunk felhasználni ebben a fájlban. A függvénykönyvtár lehet egy Ruby modul vagy natív kiterjesztés. Ez körülbelül a C `#include`-nak vagy a Java `import`-nak felel meg. A fájl értelmezendő része a fájl végéig vagy az `__END__` sorig tart. Az utóbbi után a fájl még tartalmazhat belső feldolgozásra adatokat.

```
#!/usr/bin/ruby
# Encoding: UTF-8
require 'socket'

__END__
Itt még van valami
```

A gyakorlaton az `irb` értelmezőt használjuk a Ruby értelmező demonstrálására. A dokumentációt az `ri` paranccsal olvashatjuk, amelynek a megte-

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

kinteni kívánt osztály (pl. `$ri String`) vagy metódus nevét kell megadnunk (pl. `$ri String.length`).

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

```
kovacsg@debian:~> irb
irb(main):001:0> puts 'Hello, world!'
Hello, world!
=> nil
irb(main):002:0> 'Hello, world'
=> "Hello, world"
```

Programozási nyelvekben a következő lexikai elemekkel találkozhatunk: kulcsszavak, literálok, operátorok, azonosítók, üres karakterek és kommentek. A kulcsszavakkal menet közben ismerkedünk meg, az üres karakterek pedig a szokásosak: szóköz, tabulátor. A soremelésnek mint üres karakternek külön jelentése van, terminál egy állítást az értelmező számára.

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 hashekben előszeretettel alkalmazunk kulcsként (13. sor)
- boolean literálok (14-15. sorok)
- típus nélküli literál (16. sor)
- Ruby 2.0-tól 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> 1.1
=> 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 jellenzően 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. Érdekesség, 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= kerul
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 kérdezhetjük 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_000.class
=> Fixnum
irb(main):048:0> 1_000_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 figyelnünk kell. A `defined?` operátor egy azonosítóról mondja meg, hogy az miként lett definiálva. Speciális függvényazonosító lehet a `!`-re végződő, ami azt jelenti, hogy a függvény megváltoztatja 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"
irb(main):061:0> a[-5]
=> "H"
irb(main):062:0> a[-33]
```

```
=> nil
```

A hash egy kulcs-érték párokat tartalmazó halmaz, leginkább a PHP asszociatív tömbhöz hasonlít. A Ruby tömb tetszőleges típusú értékekből összeállított lista, valójában egy hash, aminek az indexei automatikusan növelt egészek.

```
irb(main):063:0> arr = [1, 1.0, "egy"]
=> [1, 1.0, "egy"]
irb(main):065:0> arr[0]
=> 1
irb(main):066:0> arr[1]
=> 1.0
irb(main):067:0> arr.class
=> Array
irb(main):068:0> arr*2
=> [1, 1.0, "egy", 1, 1.0, "egy"]
irb(main):069:0> arr.join(':')
=> "1:1.0:egy"
irb(main):070:0> h = { "egy" => 1}
=> {"egy"=>1}
irb(main):071:0> h["egy"]
=> 1
irb(main):072:0> h[2] = "ketto"
=> "ketto"
irb(main):073:0> h
=> {"egy"=>1, 2=>"ketto"}
```

Speciális lexikai elem a tartomány, amely egész vagy karakter literálok egymás utáni elemeiből álló halmaz. A két ponttal definiált range a jobb oldali elemet is tartalmazza, a három ponttal definiált range a jobb oldali elemet már nem tartalmazza.

```
irb(main):074:0> r = 'a'..'f'
=> "a".."f"
irb(main):075:0> r = 'a'...'f'
=> "a"..."f"
irb(main):076:0> r = 'a'..'f'
=> "a".."f"
irb(main):077:0> r.each { |l| print "#{l}" }
abcdef=> "a".."f"
irb(main):078:0> r = 'a'...'f'
=> "a"..."f"
```

```
irb(main):079:0> r.each { |l| print "#{l}" }
abcde=> "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):011:0> true
=> true
irb(main):012:0> false
=> false
irb(main):013:0> nil
=> nil
```

A Ruby kétféle vezérlési szerkezetet nyújt a programkód feltételes elágaztatására, amelyek csak szintakszisukban térnek el a C-ben megismertektől: `if`/`unless`, illetve `case`. Az `if` szerkezet formálisan:

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

. A `then` minden esetben helyettesíthető soremeléssel vagy pontosvesszővel karakterrel. Az `if` feltételének negálása helyett használható az `unless`. Az `if` blokkja kiemelhető a sor elejére. Többszörös elágazást a `case` szerkezettel hozható létre:

```
case <objektum>
{when <kifejezes> <blokk>}+
[else <blokk>]
end
```

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

```
irb(main):086:0> i = 2
=> 2
irb(main):087:0> if i > 1 then print "nagy" elsif i ==1
    then print "egy" else print "kicsi" end
nagy=> nil
irb(main):088:0> unless i==1 then print "nemegy" else
   print "egy" end
nemegy=> 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):089:0> i = 1
=> 1
irb(main):090:0> case i
irb(main):091:1> when 1
irb(main):092:1> print "egy"
irb(main):093:1> when 2..10
irb(main):094:1> print "nemegy"
irb(main):095:1> end
egy=> 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):080:0> i = 1
=> 1
irb(main):081:0> while i < 3 do print "#{i=i+1}" end
23=> nil
irb(main):082:0> i = 1
=> 1
irb(main):083:0> until i > 3 do print "#{i=i+1}" end
234=> nil
irb(main):084:0> for i in 1..3 do print "#{i=i+1}" end
234=> 1..3
irb(main):085:0> for i in 1..3 do print "#{i}" end
123=> 1..3
```

Ruby-ban az azonosítók referenciaként viselkednek. Típusuk nem deklaráció során, hanem az első használatkor dől el. Ha egy objektumhoz több referenciát rendelünk, akkor annak értéke referencián keresztül megváltoztatható. Az objektumok egyenlősége vizsgálható érték szerint (==) és referencia szerint (`equal?`) metódus. Az `eql?` metódus érték szerinti egyenlőséget vizsgál azonban nem végez típuskonverziót. A `===` operátor `case` ágaiban való illeszkedést vizsgál, alapértelmezés szerint úgy működik, mint a `==` operátor.

```
irb(main):107:0> a = 'hello'
=> "hello"
irb(main):108:0> b = a
=> "hello"
irb(main):109:0> a.reverse!
=> "olleh"
irb(main):110:0> b
=> "olleh"
irb(main):111:0> a.reverse
=> "hello"
irb(main):112:0> b
=> "olleh"
irb(main):113:0> a = 1
=> 1
irb(main):114:0> b = 1.0
=> 1.0
irb(main):115:0> a == b
=> true
irb(main):116:0> a.eql? b
=> false
irb(main):117:0> a.equal? b
=> false
irb(main):118:0> a === b
=> true
```

Az objektumok konvertálhatók más típusúvá. A konverzió történhet automatikusan, vagy explicit módon a `to_s`, `to_i` vagy `to_f` stb. metódusokkal. A nem `nil` és nem `false` objektumok feltételes kifejezésekben `true` értékké konvertálódnak, míg a két megnevezett objektum `false` értékké.

```
irb(main):124:0> 1 == "1"
=> false
irb(main):125:0> 1 == "1".to_i
```

```
=> true
irb(main):126:0> 1.0 == "1".to_f
=> true
irb(main):127:0> 1.to_s == "1"
=> true
```

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):121:0> a.class
=> Fixnum
irb(main):122:0> a.respond_to? "bit_length"
=> true
irb(main):123:0> a.respond_to? "even?"
=> true
```

Blokkot kétféle szintakszissal definiálhatunk: vagy `do-end` párok között vagy kapcsos zárójelekben. A blokkoknak speciális szerepük is lehet Rubyban. Függvényhívásoknak paraméterül adható egy procedurális blokk, ami akkor hívódik meg, ha a függvény törzse anonim esetben `yield`, nevesített esetben `block.call` sorhoz érkezik (a `block` itt a blokk nevesített azonosítója).

A 96. sorban bemutatott tartomány típus `each` metódusa is ilyen. A 97-100. sorokban a `t` metódus definíciója meghívja a metódushívás paramétereként a 101. sorban átadott blokkot. A blokkban definiált procedúra paraméterezhető, ahogy azt a 105. sor mutatja. A 105. sorban meghívjuk a 102. sorban definiált `t` azonosítójú függvényt egy string paraméterrel. A függvény törzsében, vagyis a 103. sorban átadja a vezérlést a 105. sor blokkja törzsének átadva az `a` értéket. A 105. 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 104. 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):096:0> r.each { |l| print "#{l}" }
abcde=> "a"..."f"
irb(main):097:0> def t
```

```
irb(main):098:1> yield
irb(main):099:1> yield
irb(main):100:1> end
=> :t
irb(main):101:0> t { print 'a' }
aa=> nil
irb(main):102:0> def t(a)
irb(main):103:1> yield a*2
irb(main):104:1> end
=> :t
irb(main):105:0> t('hello') { |l| print "#{l}" }
hellohello=> nil
```

A Ruby programnyelv objektumorientált, építőkockái az osztályok és a modulok, amelyek egymással a nyilvános felületükön definiált metódusaikkal kommunikálnak egymással. A 135. sorban a `Math` modul `sqrt` metódusát hívjuk meg, a híváskor a `.` vagy a `::` operátort használhatjuk. Az üzenetek egymásba ágyazhatóak.

```
irb(main):129:0> a::to_s
=> "1"
irb(main):130:0> Math::sqrt 4
=> 2.0
irb(main):135:0> ::Konstans
=> 2
```

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 131-135. sorban egy `Ber` azonosítójú osztályt definiálunk, amelyet a 136. sorban példányosítunk. Az osztályban, a 147-151. sorban definiálunk egy `fizetes` azonosítójú metódust, amely 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 132-134. sorban bővítjük, a 142-144. sorban felüldefiniáljuk a viselkedést. A 136. sorban létrehozunk egy példányt az implicit ősosztályból, vagyis az `Object`-ből módon örökölt `new` metódussal, a 137. sorban meghívjuk a `fizetes` metódust két paraméterrel.

```
irb(main):131:0> class Ber
irb(main):132:1> def fizetes(jovedelem, jarulek)
irb(main):133:2> jovedelem + jarulek
```

```
irb(main):134:2> end
irb(main):135:1> end
=> :fizetes
irb(main):136:0> a = Ber.new
=> #<Ber:0x00000027723b8>
irb(main):137:0> a.fizetes 2, 2
=> 4
```

A kezdeti viselkedést az `initialize` metódus (felül)definiálásával határozhatjuk meg. A 139-141. sorokban az `Ber` osztály `@alapfizetes` azonosítójú példányváltozóját 1-re állítjuk be. Az `@alapfizetes` példányváltozót nem kell külön deklarálnunk, az az első hivatkozás hatására létrejön. Az `@alapfizetes` példányváltozó nem férhető hozzá kívülről, azonban az osztályon belül használható, ahogy azt a 142-144. sorban felüldefiniált `fizetes` metódusban megtesszük. Az osztály újbóli példányosítása után láthatjuk a különbséget a 137. és a 147. sor eredménye között.

```
irb(main):138:0> class Ber
irb(main):139:1> def initialize
irb(main):140:2> @alapfizetes = 1
irb(main):141:2> end
irb(main):142:1> def fizetes(juttatas, jarulek)
irb(main):143:2> @alapfizetes + juttatas + jarulek
irb(main):144:2> end
irb(main):145:1> end
=> :fizetes
irb(main):146:0> a = Ber.new
=> #<Ber:0x000000026f85e0 @alapfizetes=1>
irb(main):147:0> a.fizetes 2,2
=> 5
```

A példányváltozókhoz setter és getter metódusokkal férhetünk hozzá. A setter jellemzője, hogy a változó azonosítója mögé egy egyenlőségjelet írunk (149-151. sor), a getter pedig maga a változó azonosítója (152-154. sor). Használatukat a 158. és a 159. sor mutatja.

```
irb(main):148:0> class Ber
irb(main):149:1> def alapfizetes=(val)
irb(main):150:2> @alapfizetes = val
irb(main):151:2> end
irb(main):152:1> def alapfizetes
irb(main):153:2> @alapfizetes
irb(main):154:2> end
```

```
irb(main):155:1> end
=> :alapfizetes
irb(main):156:0> a = Ber.new
=> #<Ber:0x0000000265a9a8 @alapfizetes=1>
irb(main):158:0> a.alapfizetes = 3
=> 3
irb(main):159:0> a.alapfizetes
=> 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 166-171. sorok mutatják.

```
irb(main):161:0> class Ber
irb(main):162:1> attr_accessor :munkaado
irb(main):163:1> attr_reader :ado
irb(main):164:1> attr_writer :szemelyi_szam
irb(main):165:1> end
=> nil
irb(main):166:0> a = Ber.new
=> #<Ber:0x000000026372c8 @alapfizetes=1>
irb(main):167:0> a.munkaado="En"
=> "En"
irb(main):168:0> a
=> #<Ber:0x000000026372c8 @alapfizetes=1, @munkaado="En
    ">
irb(main):169:0> a.ado
=> nil
irb(main):170:0> a.szemelyi_szam
NoMethodError: undefined method `szemelyi_szam' for #<
    Ber:0x000000026372c8 @alapfizetes=1, @munkaado="En">
Did you mean?  szemelyi_szam=
        from (irb):170
        from /usr/bin/irb:11:in `<main>'
irb(main):171:0> a
=> #<Ber:0x000000026372c8 @alapfizetes=1, @munkaado="En
    ">
```

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

metódus szingletonnak nevezzük. A 176-178. sorban az `a` példányra vonatkozóan definiálunk egy metódust, ami a `Ber` osztály más példányára már nem hozzáférhető, és elfedi az osztály azonos nevű metódusát.

```
irb(main):175:0> a = Ber.new
=> #<Ber:0x000000025bdea0 @alapfizetes=1>
irb(main):176:0> def a.extra_jutalom(val)
irb(main):177:1> @alapfizetes + val
irb(main):178:1> end
=> :extra_jutalom
irb(main):179:0> a
=> #<Ber:0x000000025bdea0 @alapfizetes=1>
irb(main):180:0> a.extra_jutalom 3
=> 4
irb(main):182:0> b = Ber.new
=> #<Ber:0x00000024260b0 @alapfizetes=1>
irb(main):183:0> b.extra_jutalom 3
NoMethodError: undefined method `extra_jutalom' for #<
    Ber:0x00000024260b0 @alapfizetes=1>
        from (irb):183
        from /usr/bin/irb:11:in `<main>'
```

Ruby-ban is definiálhatók osztálymetódusok (185-187. sor), amelyek az osztály példányainak létezése nélkül is meghívhatók, illetve közösek az összes példányra vonatkozóan. Osztálymetódus az osztálydefiníción belüli az aktuális példány referenciájára (`self`) vonatkozó szingleton metódus definíciójával azonos hatást érhetünk el. Osztályváltozók (`@@` prefixű azonosítóval rendelkező változók) csakis osztálymetódusokon belül használhatók, és az első használatkor inicializálandók, különben hibát kapunk.

```
irb(main):184:0> class Ber
irb(main):185:1> def Ber.minimalber
irb(main):186:2> @@minimalber = 2
irb(main):187:2> @@minimalber
irb(main):188:2> end
irb(main):189:1> end
=> :minimalber
irb(main):190:0> Ber::minimalber
=> 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 defi-

níciója során többletmunkát igényel, viszont megkönnyíti a használatot. A 191-198. sorban egy ilyen definíciót látunk. A metódus paraméterét mint hash objektum kezeljük, amelynek az `:minimal` szimbólummal jelölt értékét hozzárendeljük az `minimal` lokális változóhoz, vagy ha az `:minimal` 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 200. 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):191:0> class Ber
irb(main):192:1> def osszetevok(a)
irb(main):193:2> minimal = a[:minimal] || 0
irb(main):194:2> jutalom = a[:jutalom] || 1
irb(main):195:2> extra = a[:extra] || 0
irb(main):196:2> minimal + jutalom + extra
irb(main):197:2> end
irb(main):198:1> end
=> :osszetevok
irb(main):199:0> a = Ber.new
=> #<Ber:0x00000026f8b80 @alapfizetes=1>
irb(main):200:0> a.osszetevok :minimal => 3, :extra =>
  4
=> 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):228:0> class Ber
irb(main):229:1> public
irb(main):230:1> def osszetevok(jovedelem, jarulek)
irb(main):231:2> jovedelem + @alapfizetes + jarulek
irb(main):232:2> end
irb(main):233:1> end
=> :osszetevok
irb(main):234:0> class Ber
```

```
irb(main):235:1> protected
irb(main):236:1> def alapfizetes=(val)
irb(main):237:2> @alapfizetes = val
irb(main):238:2> end
irb(main):239:1> end
=> :alapfizetes=
irb(main):240:0> class Ber
irb(main):241:1> private
irb(main):242:1> def alapfizete
irb(main):243:2> end
irb(main):244:1> end
=> :alapfizete
irb(main):245:0> class Ber
irb(main):246:1> private
irb(main):247:1> def alapfizetes
irb(main):248:2> end
irb(main):249:1> end
=> :alapfizetes
irb(main):250:0> a = Ber.new
=> #<Ber:0x00000002415d28 @alapfizetes=2>
irb(main):251:0> a.alapfizetes
NoMethodError: private method `alapfizetes' called for
   #<Ber:0x00000002415d28 @alapfizetes=2>
Did you mean? alapfizetes=
        from (irb):251
        from /usr/bin/irb:11:in `<main>'
irb(main):252:0> a.alapfizetes=3
NoMethodError: protected method `alapfizetes=' called
    for #<Ber:0x00000002415d28 @alapfizetes=2>
Did you mean? alapfizetes
              alapfizete
        from (irb):252
        from /usr/bin/irb:11:in `<main>'
irb(main):253:0> class MBer < Ber
irb(main):254:1> end
=> nil
irb(main):255:0> class MBer < Ber
irb(main):256:1> def alapfizetes
irb(main):257:2> @alapfizetes
irb(main):258:2> end
irb(main):259:1> end
```

```
=> :alapfizetes
irb(main):260:0> b = MBer.new
=> #<MBer:0x000000026ff2c8 @alapfizetes=2>
irb(main):261:0> b.alapfizetes
=> 2
```

Az osztályok a < operátorral specializálhatók. A leszármazott osztály kiegészítheti vagyis specializálhatja, illetve felüldefiniálhatja az ősosztály viselkedését. A 255-259. sorban a `MBer` osztályt definiáljuk, amely rendelkezik az `Ber` osztály összes `public` és `protected` metódusával, illetve példányváltozójával. A 256-258. sor felüldefiniálja az `alapfizetes` attribútum privát setter metódusát, így az másképp fog viselkedni, mint azt a 251. sorban láttuk, az `MBer` példányaira ismét elérhető lesz. A felüldefiniált metódus láthatósága eltérhet egy leszármazott osztályban.

Metódusokhoz hasonlóan operátorok is (felül)definiálhatók egy oszlályon belül (264-266. sor), mivel az operátorok önmaguk is metódushívások, lásd 262. sor és 269. sor.

```
irb(main):262:0> 1.+(2)
=> 3
irb(main):263:0> class Ber
irb(main):264:1> def +(val)
irb(main):265:2> @alapfizetes += val
irb(main):266:2> end
irb(main):267:1> end
=> :+
irb(main):268:0> a = Ber.new
=> #<Ber:0x0000000268d650 @alapfizetes=2>
irb(main):269:0> a+1
=> 3
```

A Ruby másik egysége az osztály mellett a modul. A modul akárcsak az osztály egységbe zár attribútumokat és metódusokat. Modulba olyan metódusokat helyezhetünk el, amelyek több osztályra vonatkozóan közösek. A modul rokonságot mutat a Java interfész fogalmával, azzal a különbséggel, hogy a modul nemcsak deklarál egy bizonyos viselkedést a megvalósító osztály számára, hanem mindjárt specifikálja is. A modulnak nem lehet közvetlen peldánya (273. sor), viszont létezhet példány, ami rendelkezik a modulban definiált viselkedéssel.

A 271-278. sorok egy modult definiálnak egy példányváltozóval és egy setter-getter párral. A modul nem példányosítható, viszont a modulban definiált metódusokkal bármely osztály viselkedését kibővíthetjük (280. sor) az

`include` kulcsszóval példánymetódusként, az `extend` kulcsszóval osztálymetódusként. Egy osztály tetszőleges számú modult integrálhat magába.

```
irb(main):271:0> module Felvetel
irb(main):272:1> def felvetel
irb(main):273:2> @felvetel
irb(main):274:2> end
irb(main):275:1> def felvetel=(val)
irb(main):276:2> @felvetel = val
irb(main):277:2> end
irb(main):278:1> end
=> :felvetel=
irb(main):279:0> class Ber
irb(main):280:1> include Felvetel
irb(main):281:1> end
=> Ber
irb(main):282:0> a = Ber.new
=> #<Ber:0x000000025c2630 @alapfizetes=2>
irb(main):283:0> a.felvetel=3
=> 3
irb(main):284:0> a.felvetel
=> 3
```

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):285:0> module En
irb(main):286:1> class Ber
irb(main):287:2> end
irb(main):288:1> end
=> nil
irb(main):289:0> a = En::Ber.new
=> #<En::Ber:0x0000000261e570>
irb(main):290:0>
```