# A Ruby programozási nyelv
# Gyakorlat

Kovács Gábor

2014. szeptember 24.

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):002:0> puts "Hello, world"
Hello, world
=> nil
irb(main):003:0> "Hello, world"
=> "Hello, world"
```

Literálok Rubyban:

- egész szám (4. sor, 11. sor)
- lebegőpontos szám (5. sor)
- string, amit megadhatunk ' vagy " szimbólumok között (7-8. sorok)
- tömb, ami egy szögletes zárójelek között megadott vesszővel elválasztott lista (9. sor)
- hash, ami kapcsos zárójelek között tartalmaz kulcs-érték párokat => szimbólummal elválasztva (10. sor)
- reguláris kifejezés (12. sor)
- tartomány, ami a két szélső értékkel megadott egymás utáni egész értékek halmaza (13-14. sorok)
- boolean literál (15-16. sorok)
- típus nélküli literál (17. sor)
- szimbólum, ami nem más, mint egy lebutított string, és amelyet hashekben előszeretettel alkalmazunk kulcsként (18. sor)

```
irb(main):004:0> 1
=> 1
irb(main):005:0> 1.0
=> 1.0
irb(main):007:0> "Hello"
=> "Hello"
```

```
irb(main):008:0> 'Hello'
=> "Hello"
irb(main):009:0> [ 1, "ketto", 3.0 ]
=> [1, "ketto", 3.0]
irb(main):010:0> { 1 => "egy", 2.0 => 2 }
=> {1=>"egy", 2.0=>2}
irb(main):011:0> 1_000_000_000_000_000_000_000
=> 1000000000000000000
irb(main):012:0> /[a-z]/
=> /[a-z]/
irb(main):013:0> 'a'..'f'
=> "a".."f"
irb(main):014:0> 'a'...'f'
=> "a"..."f"
irb(main):015:0> true
=> true
irb(main):016:0> false
=> false
irb(main):017:0> nil
=> nil
irb(main):018:0> :s
=> :s
```

Egysoros megjegyzést a # szimbólum után tehetünk.

```
irb(main):019:0> # komment
irb(main):020:0*
```

Többsoros megjegyzés tételére két módunk van:

```
irb(main):024:0> #
irb(main):025:0* # Dokumentacio
irb(main):026:0* #
irb(main):020:0> =begin
irb(main):021:1* ide
irb(main):022:1> akarmit is
irb(main):023:1> irok
irb(main):024:1> ,
irb(main):025:1* annak
irb(main):026:1> nincs jelentosege
irb(main):027:1> =end
=> nil
```

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):028:0> 1+2
=> 3
irb(main):029:0> 1+
irb(main):030:0* 2
=> 3
irb(main):031:0> 1
=> 1
irb(main):032:0> +2
=> 2
irb(main):033:0> 1\
irb(main):034:0* +2
=> 3
irb(main):035:0> 2**3
=> 8
irb(main):060:0> 3/2
=> 1
irb(main):061:0> -3/2
=> -2
irb(main):062:0> -3/2.0
=> -1.5
```

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):036:0> Konstans = 1
=> 1
irb(main):037:0> Konstans = 2
(irb):37: warning: already initialized constant
    Konstans
```

```
=> 2
irb(main):038:0> a = 1
=> 1
irb(main):039:0> a = 2
=> 2
irb(main):040:0> $globalis = 1
=> 1
irb(main):041:0> @a  = 1
=> 1
irb(main):042:0> @@a = 2
=> 2
```

Az azonosítók másik csoportja a függvény-, vagy másnéven metódusazonosítók. Függvényt a `def` kulcsszó után definiálhatunk a függvény azonosítója (43.sor), a formális paraméterlista és a törzs megadásával. A függvény törzse a `def`-fel egy szinten lévő `end`-ig tart (45.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.

```
irb(main):043:0> def f(n)
irb(main):044:1> n*2
irb(main):045:1> end
=> nil
irb(main):046:0> f(2)
=> 4
irb(main):047:0> f("ketto")
=> "kettoketto"
irb(main):048:0> "ketto".respond_to? "*"
=> true
irb(main):049:0> f (1+2) + 3
=> 12
irb(main):044:0> defined? f
=> "method"
```

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):050:0> 1.class
=> Fixnum
irb(main):053:0> 1_000_000_000_000_000_000_000_000.class
=> Bignum
irb(main):054:0> 1.0.class
=> Float
irb(main):055:0> "Hello".class
=> String
```

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):063:0> $a = 'lo'
=> "lo"
irb(main):064:0> "Hel#{$lo}"
=> "Hel"
irb(main):065:0> "Hel#{$a}"
=> "Hello"
irb(main):066:0> 'Hel#{$a}'
=> "Hel\#{$a}"
irb(main):067:0> s = "Hel#{$lo}"
=> "Hel"
irb(main):068:0> s[0]
=> "H"
irb(main):069:0> s[-1]
=> "l"
```

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):056:0> h = { 1 => "egy", 2.0 => 2 }
=> {1=>"egy", 2.0=>2}
irb(main):057:0> h[1]
=> "egy"
irb(main):058:0> t = [ 1, 1.0, "1" ]
=> [1, 1.0, "1"]
irb(main):059:0> t[1]
```

```
=> 1.0
```

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):073:0> r = 'a'..'f'
=> "a".."f"
irb(main):074:0> r.each { |l| print "␣#{l}␣" }
 a  b  c  d  e  f => "a".."f"
irb(main):075:0> r1 = 'a'...'f'
=> "a"..."f"
irb(main):076:0> r1.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):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
```

. 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):089:0> a = 2
=> 2
irb(main):090:0> if a>1 then puts "nagy" elsif a==1
   then puts "egy" else puts "kicsi" end
nagy
=> nil
irb(main):091:0> if a > 1
irb(main):092:1> puts "nagy"
irb(main):093:1> elsif a == 1
irb(main):094:1> puts "egy"
irb(main):095:1> else
irb(main):096:1* puts "kicsi"
irb(main):097:1> end
nagy
=> nil
irb(main):098:0> if a==2 then puts "ketto" end
ketto
=> nil
irb(main):099:0> puts "ketto" if a==2
ketto
=> nil
irb(main):100:0> unless a == 1 then puts "nemegy" end
nemegy
=> nil
irb(main):101:0> puts "ketto" unless a==2
=> nil
irb(main):102:0> puts "ketto" unless a
=> nil
irb(main):110:0> case a
irb(main):111:1> when 1
irb(main):112:1> puts "egy"
irb(main):113:1> when 2..10
irb(main):114:1> puts "a tartomanyban van"
irb(main):115:1> end
a tartomanyban van
=> nil
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):077:0> i = 1
=> 1
irb(main):078:0> while i < 3 do puts "#{i=i+1}" end
2
3
=> nil
irb(main):079:0> until i > 3 do puts "#{i=i+1}" end
4
=> nil
irb(main):080:0> i =1
=> 1
irb(main):082:0> until i > 3 do puts "#{i=i+1}" end
2
3
4
=> nil
irb(main):084:0> for i in 1..3 do puts "#{i}" end
1
2
3
=> 1..3
```

Az objektumok konvertálhatók más típusúvá. A konverzió történhet automatikusan, mint például ahogy az a 75. sorban történik, vagy explicit módon a `to_s`, `to_i` vagy `to_f` metódusokkal.

```
irb(main):104:0> 1.to_s
=> "1"
irb(main):105:0> "1".to_i
```

```
=> 1
irb(main):106:0> "1".to_f
=> 1.0
```

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):124:0> a
=> 2
irb(main):125:0> a.nil?
=> false
irb(main):126:0> 1 == 1.0
=> true
irb(main):127:0> 1 === 1.0
=> true
irb(main):128:0> 1.eql?(1.0)
=> false
irb(main):129:0> 1.equal?(1.0)
=> false
irb(main):130:0> a = "hello"
=> "hello"
irb(main):131:0> b = a
=> "hello"
irb(main):132:0> a.eql? b
=> true
irb(main):133:0> a.reverse!
=> "olleh"
irb(main):134:0> a
=> "olleh"
irb(main):135:0> b
=> "olleh"
```

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):159:0> a.respond_to? "to_str"
=> true
irb(main):160:0> a.respond_to? "*"
=> 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 136. sorban bemutatott tartomány típus `each` metódusa is ilyen. A 137-139. sorokban a `t` metódus definíciója meghívja a metódushívás paramétereként a 141. sorban átadott blokkot. A blokkban definiált procedúra paraméterezhető, ahogy azt a 137-139. sorok mutatják. A 141. sorban meghívjuk a 137. sorban definiált `t` azonosítójú függvényt egy string paraméterrel. A függvény törzsében, vagyis a 138. sorban átadja a vezérlést a 141. sor blokkja törzsének átadva az `a` értéket. A 141. 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 139. 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):136:0> r1.each { |l| print "␣#{l}␣" }
 a  b  c  d  e => "a"..."f"
irb(main):137:0> def t(a)
irb(main):138:1> yield a
irb(main):139:1> end
=> nil
irb(main):141:0> t("hello") { |l| print l }
hello=> nil
irb(main):142:0> def t(a)
irb(main):143:1> yield a
irb(main):144:1> yield a
irb(main):145:1> end
=> nil
irb(main):146:0> t("ho") { |l| print l }
hoho=> nil
irb(main):148:0> p = Proc.new { |l| print l }
```

```
=> #<Proc:0x0000000279c6e0@(irb):148>
irb(main):149:0> t("hello") { |l| b = 2; print l+b.to_s
    }
hello2hello2=> 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 150-155. 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 157. 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):150:0> def m(a)
irb(main):151:1> n = a[:n] || 0
irb(main):152:1> m = a[:m] || 1
irb(main):153:1> l = a[:l] || 0
irb(main):154:1> n+m+l
irb(main):155:1> end
=> nil
irb(main):156:0> h = { :n=> 3, :l => 4}
=> {:n=>3, :l=>4}
irb(main):157:0> m h
=> 8
```

A Ruby programnyelv objektumorientált, építőkockái 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 viselkedést (metódusok) és a tulajdonságokat (attribútumok), bár az utóbbiak nem érhetők el az osztály példányának referenciáján keresztül.

A 162-163. sorban egy osztályt definiálunk, amelyet a 164. sorban példányosítunk. Az osztályt a 165-169. 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 166-168. sorban bővítjük, a 173-175. sorban felüldefiniáljuk a viselkedést. A 170. sorban létrehozunk egy példányt az implicit ősosztályból, vagyis az `Object`-ből módon örökölt `new` metódussal, a 171. 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):162:0> class A
irb(main):163:1> end
=> nil
irb(main):164:0> a = A.new
=> #<A:0x00000002547890>
irb(main):165:0> class A
irb(main):166:1> def m(p1,p2)
irb(main):167:2> p1+p2
irb(main):168:2> end
irb(main):169:1> end
=> nil
irb(main):170:0> a = A.new
=> #<A:0x00000002536ae0>
irb(main):171:0> a.m 2,3
=> 5
```

A kezdeti viselkedést az `initialize` metódus (felül)definiálásával határozhatjuk meg. A 176-178. 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 173-175. sorban felüldefiniált `m` metódusban megtesszük.

```
irb(main):172:0> class A
irb(main):173:1> def m(p1,p2)
irb(main):174:2> p1+p2+@a
```

```
irb(main):175:2> end
irb(main):176:1> def initialize
irb(main):177:2> @a=2
irb(main):178:2> end
irb(main):179:1> end
=> nil
irb(main):180:0> a = A.new
=> #<A:0x00000029b6a98 @a=2>
irb(main):181:0> a.m 2,3
=> 7
```

A példányváltozókhoz setter és getter metódusokkal férhetünk hozzá. A setter jellemzője, hogy a változó azonosítója mögé egy egyenlőségjelet írunk (183-185. sor), a getter pedig maga a változó azonosítója (186-188. sor). Használatukat a 192. és a 193. sor mutatja.

```
irb(main):182:0> class A
irb(main):183:1>   def a=(val)
irb(main):184:2> @a =val
irb(main):185:2> end
irb(main):186:1> def a
irb(main):187:2> @a
irb(main):188:2> end
irb(main):189:1> end
=> nil
irb(main):190:0> a = A.new
=> #<A:0x00000029948f48 @a=2>
irb(main):191:0> a.a= 2
=> 2
irb(main):192:0> a.a= 3
=> 3
irb(main):193:0> 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 204-206. sorok mutatják.

```
irb(main):194:0> class A
irb(main):195:1> attr_accessor :b
```

```
irb(main):196:1> end
=> nil
irb(main):197:0> a = A.new
=> #<A:0x0000000292eb98 @a=2>
irb(main):198:0> a.b=2
=> 2
irb(main):199:0> class A
irb(main):200:1> attr_reader :c
irb(main):201:1> attr_writer :d
irb(main):202:1> end
=> nil
irb(main):203:0> a = A.new
=> #<A:0x00000002919798 @a=2>
irb(main):204:0> a.c
=> nil
irb(main):205:0> a.d =3
=> 3
irb(main):206:0> a.d
NoMethodError: undefined method `d' for #<A:0
    x00000002919798 @a=2, @d=3>
        from (irb):206
        from /usr/bin/irb:12:in `<main>'
```

Metódust definiálhatunk egy-egy példányra specializálva is, ekkor az a metódus szingletonnak nevezzük. A 207-209. 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):207:0> def a.m2(val)
irb(main):208:1> @a = val
irb(main):209:1> end
=> nil
irb(main):210:0> a.m2 5
=> 5
irb(main):211:0> a.a
=> 5
```

Ruby-ban is definiálhatók osztálymetódusok (213-216.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.

```
irb(main):212:0> class A
irb(main):213:1> def A.m3
irb(main):214:2> @@a = 5
irb(main):215:2> "hello"
irb(main):216:2> end
irb(main):217:1> end
=> nil
irb(main):218:0> class A
irb(main):219:1> def self.m4
irb(main):220:2> "hello"
irb(main):221:2> end
irb(main):222:1> end
=> nil
```

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):223:0> class A
irb(main):224:1> public
irb(main):225:1> def m(p1,p2)
irb(main):226:2> p1+p2+@a
irb(main):227:2> end
irb(main):228:1> protected
irb(main):229:1> def a
irb(main):230:2> @a
irb(main):231:2> end
irb(main):232:1> private
irb(main):233:1> def a=(val)
irb(main):234:2> @a=val
irb(main):235:2> end
irb(main):236:1> end
=> nil
irb(main):237:0> a = A.new
=> #<A:0x0000000278c1f0 @a=2>
irb(main):238:0> a.a
NoMethodError: protected method `a' called for #<A:0
    x0000000278c1f0 @a=2>
```

```
        from (irb):238
        from /usr/bin/irb:12:in '<main>'
irb(main):246:0> class B < A
irb(main):247:1> end
=> nil
irb(main):248:0> class B < A
irb(main):249:1> def a=(val)
irb(main):250:2> @a = 2* val
irb(main):251:2> end
irb(main):252:1> end
=> nil
irb(main):253:0> b = B.new
=> #<B:0x00000002554dd8 @a=2>
irb(main):254:0> b.m6
=> 4
irb(main):255:0> b.a=2
=> 2
irb(main):256:0> b.a
NoMethodError: protected method 'a' called for #<B:0
    x00000002554dd8 @a=4>
        from (irb):256
        from /usr/bin/irb:12:in '<main>'
```

Az osztályok a `<` operátorral specializálhatók. A leszármazott osztály kiegészítheti vagyis specializálhatja, illetve felüldefiniálhatja az ősosztály viselkedését. A 248-252. sorban a `B` osztályt definiáljuk, amely rendelkezik az `A` osztály összes metódusával, illetve példányváltozójával. A 249-251. sor felüldefiniálja az `a` attribútum setter metódusát, így az másképp fog viselkedni, mint azt a 238. sorban láttuk, `B` példányaira ismét elérhető lesz.

Metódusokhoz hasonlóan operátorok is (felül)definiálhatók egy oszlályon belül (258-260. sor), mivel az operátorok önmaguk is metódushívások, lásd 265. sor.

```
irb(main):257:0> class A
irb(main):258:1> def +(val)
irb(main):259:2> @a+val
irb(main):260:2> end
irb(main):261:1> end
=> nil
irb(main):262:0> a = A.new
=> #<A:0x00000002539128 @a=2>
irb(main):263:0> a.+(2)
```

```
=> 4
irb(main):264:0> 1+2
=> 3
irb(main):265:0> 1.+(2)
=> 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 (274. sor), viszont létezhet példány, ami rendelkezik a modulban definiált viselkedéssel.

A 266-273. sorok egy modult definiálnak egy példányváltozóval és egy setter-getter párral. A modulban definiált metódusokkal bármely osztály viselkedését kibővíthetjük (276. sor). Egy osztály tetszőleges számú modult integrálhat magába.

```
irb(main):266:0> module Szin
irb(main):267:1> def szin=(val)
irb(main):268:2> @szin=val
irb(main):269:2> end
irb(main):270:1> def szin
irb(main):271:2> @szin
irb(main):272:2> end
irb(main):273:1> end
=> nil
irb(main):274:0> sz = Szin.new
NoMethodError: undefined method `new' for Szin:Module
        from (irb):274
        from /usr/bin/irb:12:in `<main>'
irb(main):275:0> class A
irb(main):276:1> include Szin
irb(main):277:1> end
=> A
irb(main):278:0> a = A.new
=> #<A:0x00000002979530 @a=2>
irb(main):279:0> a.szin='kek'
=> "kek"
```

A modulok emellett lehetővé teszik az esetleges osztálynév-ütközések el-

kerülését, névtereket definiálhatunk velük. Ekkor az összetartozó osztályok definícióját a modul definícióján belül helyezzük el.

```
irb(main):263:0> module M
irb(main):264:1>   class A
irb(main):265:2>   end
irb(main):266:1> end
=> nil
irb(main):267:0> a = M::A.new
=> #<M::A:0x00000001d4d2c0>
```