Informatika 4.

Objektum-orientált programozás

Kottyán , László

Informatika 4. : Objektum-orientált programozás

Kottyán , László
Lektor : Cseri , Tamás

Ez a modul a TÁMOP - 4.1.2-08/1/A-2009-0027 „Tananyagfejlesztéssel a GEO-ért” projekt keretében készült. A projektet az Európai Unió és a Magyar Állam 44 706 488 Ft összegben támogatta.

v 1.0

Publication date 2010

Kivonat

A fejezet betekintést nyújt a programozás és az objektum-orientált paradigma alapjaiba a Ruby programozási nyelven keresztül.

Tartalom

A táblázatok listája

4. fejezet - Objektum-orientált programozás

1. 4.1 Bevezetés

A fejezet a programozás alapjainak elsajátításához nyújt segítséget a Ruby programozási nyelven keresztül.

A Ruby egy magas szintű programozási nyelv, amely világos, könnyen értelmezhető jelölésrendszerrel rendelkezik. Ezáltal, a kezdő programozók már az első lépéseknél sikerélményekkel gazdagodhatnak.

Terjedelmi korlátok miatt a Ruby programozási nyelvet teljes mélységében bemutatni nem tudjuk, ezért olyan alapismereteket kívánunk átadni, amelyre az olvasó építhet a későbbi programozási tanulmányai során.

Napjainkban, az alkalmazásfejlesztésben, az objektum-orientált szemlélet meghatározó jelentőséggel bír. A tananyagban az objektum-orientált programozás alapfogalmait ismertetjük. A Ruby objektum-orientált programozási nyelv, ezért alkalmas a paradigma gyakorlati alkalmazásának szemléltetésére.

A gyakorlati példák megoldásához az Olvasónak rendelkeznie kell a Ruby futtató környezettel. A Rubyt szabadon letöltheti a www.ruby-lang.org webhelyről, ahol a telepítés lépéseiről is tájékozódhat.

A továbbiakban az Olvasó

- áttekintést kap a programozási nyelvek általános jellemzőiről,
- megismerheti a Ruby programozási nyelv alapjait,
- és betekintést kap az objektum-orientált szemléletbe.

2. 4.2 A programozási nyelvek fejlődése

A CPU a számítógépnek az a része, amely a memóriában tárolt adatokat, utasításokat kiolvassa, értelmezi és végrehajtja. Az adatok és utasítások a memóriában kettes számrendszerben, bináris formátumban tárolódnak. A kettes számrendszer esetén az egyes helyi értékek kétféle értéket vehetnek fel, a 0-át és az 1-et. Ez hasonlít egy kapcsoló ki- vagy bekapcsolt állapotához. A memória is ilyen „kapcsolókból” épül fel, amelyekben az adatot a kapcsoló állapota határozza meg. Egy ilyen kapcsolót bitnek nevezünk. A memória kezelésekor egyszerre több bittel célszerű dolgozni, ezért a memória legkisebb elérhető egysége 8 bitből áll, amit byte-nak hívnak.

Egy, a számítógép számára értelmezhető és feldolgozható program byte-ok sorozatából áll, ezt gépi kódnak, vagy gépi nyelvnek nevezzük.

Egy gépi kód kettes számrendszerben (binárisan) megadva a következő:

```
10110000 01100001
```

A gépi kódot valójában tömörebb írásmóddal, hexadecimális (tizenhatos) számrendszerben ábrázolják. A fenti bináris kód hexadecimálisan ábrázolva:

```
B0 61
```

A gépi kódú programozás megtanulása nehézkes, nem elég csupán a programozási nyelvet megtanulni, de ismerni kell a hardvert is. Ezért a gépi kód utasításait, a byte-ok sorozatát csoportosították, ezeknek a csoportoknak könnyen megtanulható nevet adtak, így létre jött az Assembly programozási nyelv.

Egy tipikus Assembly kód az alábbi:

```
mov al, 061h
```

Egy Assembly kód több előre definiált gépi kódú utasítás sorozatnak felel meg, így a gépi kódú programozásnál könnyebbé és gyorsabbá válik a programozás.

A számítógéphez közel álló programnyelveket, mint a gépi kód vagy az Assembly, alacsony szintű nyelveknek nevezzük.

Az Assembly nyelv használatának azonban hátrányai is vannak, a programozónak ismernie kell a processzor működését. Amikor a processzor az adatokon dolgozik, akkor ideiglenes tárolókba, regiszterekbe helyezi azokat, majd műveleteket végez és a megfelelő számítógépes egységhez továbbítja az adatokat. További hátrány, hogy minden processzor típushoz más és más Assembly nyelv tartozik, ezért ezek a programok nem hordozhatóak, az egyik processzorra megírt programot nem tudjuk egy másikon futtatni.

Az Assembly nyelven írt programok bár gyorsan működnek, a nyelv nem ideális összetettebb programok elkészítésére. Annak érdekében, hogy a hardverrel kapcsolatos részletekkel a programozóknak ne kelljen foglalkozni, egyszerűbbé váljon a programozás, komplexebb feladatok is megvalósíthatók legyenek, megszülettek a magas szintű programozási nyelvek.

A magas szintű programozási nyelvekben magasabb absztrakciós szinten gondolkozunk - elvonatkoztatva a regiszterektől és a memóriacímektől -, az emberi nyelvhez hasonlító utasításokkal tudjuk a programot elkészíteni.

2.1. 4.2.1 Hello világ!

A „Hello World” hagyományosan az első program, amit a bevezető programozási tankönyvekben leírnak, megtanítanak a tanulóknak. A program Brian Kernighan és Dennise Ritchie The C Programming Language című könyvében jelent meg először 1978-ban és így nézett ki:

```
main() {
    printf("hello, world\n");
}
```

Ez a kód csupán annyit tesz, hogy kiírja a képernyőre az üdvözlő üzenetet. A rövid kódot szinte az összes programozási nyelven leírták már. Az idők során a nagybetűs és felkiáltójeles változat terjedt el.

Nézzünk néhány példát, a program megvalósítására különböző programozási nyelveken.

Egy Assembly változat:

```
mov ax,cs
mov ds,ax
mov ah,9
mov dx, offset Hello
int 21h
xor ax,ax
int 21h
Hello:
  db "Hello World!",13,10,"$"
```

Pascal nyelven:

```
program HelloWorld;
begin
  WriteLn('Hello World!');
end.
```

A Java kód:

```
class HelloWorld {
  static public void main( String args[] ) {
    System.out.println( "Hello World!" );
  }
}
```

A Ruby program:

```
puts 'Hello World!'
```

3. 4.3 A programozási nyelvek jellemzői

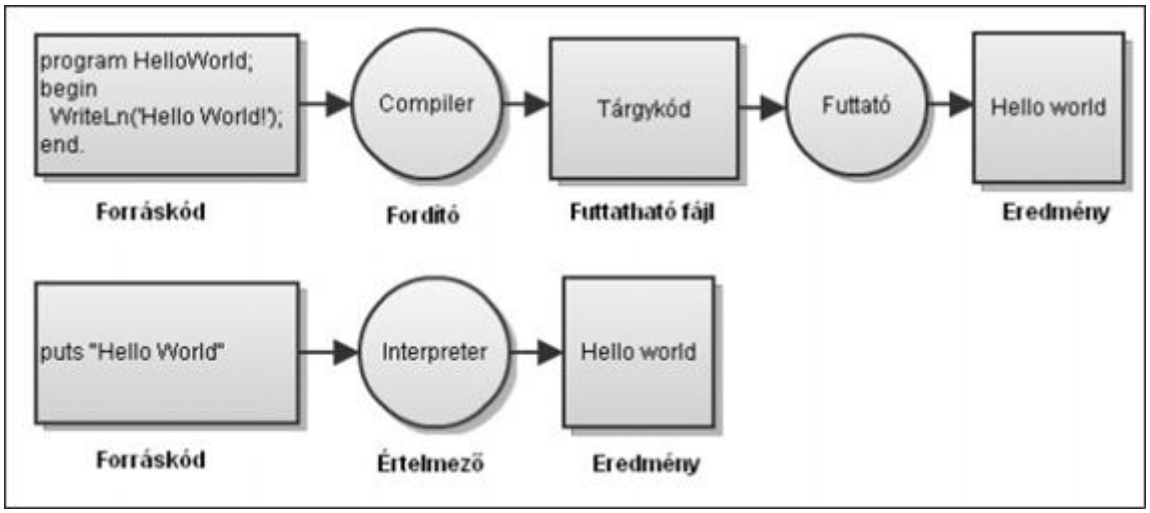
3.1. 4.3.1 A program végrehajtása

A programot, amit valamilyen szerkesztő eszközzel elkészítünk forráskódnak (vagy forrásprogramnak) hívjuk. Ez egy szöveges állomány, a választott programozási nyelven megfogalmazva és a programozási nyelvre jellemző kiterjesztéssel ellátva. A Ruby programok `.rb` kiterjesztésűek.

A számítógép csak a gépi kódot képes végrehajtani, ezért ha egy programot megírtunk, azt valamilyen módon a számítógép számára alkalmas formába kell átalakítani. Erre két alapvető technika létezik, az értelmezés és a fordítás.

Az értelmezett (interpretált, interpreteres) nyelvek esetén egy értelmező program a forráskód minden egyes sorát gépi kódra fordítja, amelyet a számítógép azonnal végrehajt.

A fordítás (compilation) során a fordító program (compiler) a teljes forráskódot egy lépésben alakítja át egy un. tárgykódra, amely egy végrehajtható állomány. A futtató környezet a lefordított állományt, tetszőleges alkalommal végrehajthatja (1. ábra).



1. ábra Fordítás és értelmezés

A lefordított programok előnye, hogy gyorsabban működnek, mert a végrehajtásuk előtt nem kell minden egyes utasítást újra gépi kódra alakítani. Hátrányuk, hogy sokkal nagyobb a méretük, mint az interpretált forráskód. Napjainkban, ez utóbbi jellemzőnek főként a mobil eszközöknél van jelentősége.

Az interpretálás egyik előnye a fejlesztés során mutatkozik, mert a forráskód bármely módosításának eredményét gyorsabban megtekinthetjük, nem kell minden változtatás után újabb és újabb futtatható állományt készíteni. Az interpretált nyelveket gyakran un. szkriptnyelvként is használják. Ez azt jelenti, hogy egy alkalmazás vagy rendszer működése a felhasználó által testreszabható, befolyásolható a szkriptnyelven írt

programokkal. Például, a Google Sketchup 3D modellező program a Rubyval szkriptelhető, a Microsoft Office alkalmazásokba a Visual Basic nyelv segítségével illeszthetünk utasításokat, un. makrókat.

3.2. 4.3.2 Változók

A programozás során a számítógépnek utasításokat adunk és meghatározzuk, hogy milyen adatokon milyen műveletet végezzen el.

Ahhoz, hogy ezt megvalósítsuk, időnként hivatkoznunk kell az adatainkra.

Tegyük fel, hogy egy egyszerű matematikai számítást szeretnénk elvégezi, például egy egyenlet eredményére vagyunk kíváncsiak, amelyet úgy kapunk meg, hogy egy számot megszorzunk kettővel és hozzáadunk hármat.

Ezt így írhatnánk le:

```
x = 2y+3
```

Ugyanez, egy programnyelven megfogalmazva nem sokban különbözik ettől. Először meg kell határoznunk `y` értékét – ez legyen most `4` -, majd el kell végezni a számítást. Ruby nyelven ez így fest:

```
y = 4
x = 2*y+3
puts x
```

Az eredmény:

```
11
```

A rövid programunkban két változót használtunk, `y` -t és `x` -et. A változóknak értékeket adtunk, vagy másképpen megfogalmazva a változóink hivatkoznak az értékekre. Végül a `puts` utasítás kiírta az eredményt a képernyőre.

Valójában az értékadással a memóriában lefoglaltunk egy területet, amely tárolja az adatot. Egy adat később, a korábban megadott változóval elérhető és felhasználható egyéb számítások elvégzésére. Erre egy példa, folytatva a programunkat, az alábbi:

```
y = 4
x = 2*y+3
puts x
a = x+y
puts a
```

Az eredmény:

```
11
15
```

Itt tehát, `x` és `y` változókkal számoltunk tovább és bevezettünk az a változót.

Változóinkat új értékekhez is rendelhetjük, megtehetjük a következőt:

```
y = 4
x = 2*y+3
puts x
a = x+y
```

```
puts a
x = 8
a = x+y
puts a
```

Most az eredmény:

```
11
15
12
```

Egy új értéket adtunk az `x` változónak, így folytattuk a programunkat, ezért az `a` változó értéke is módosult.

A változók fontos tulajdonsága, hogy különböző típusú értékekre hivatkozhatnak. Az előző példákban számokra hivatkoztak a változóink. A következő változó pedig egy szövegre hivatkozik:

```
sz = 'Ez egy szöveg'
```

A változókat a használatuk előtt értékekhez kell rendelni vagy az első használatkor szükséges ezt megtenni egy értékadó utasítással – mint például az `a=x+y` esetén tettük -, egyébként hibaüzenetet kapunk.

3.3. 4.3.3 Statikusan és dinamikusan típusos nyelvek

A változók különböző értékekre hivatkozhatnak, ezek az értékek lehetnek számok, szövegek vagy egyéb adatok. A programozási nyelvek egy részében egy változó csak egy előre meghatározott típusú értékre hivatkozhat.

A következő Pascal nyelvű programban, az `x` és `y` változókat meghatározzuk, azaz deklaráljuk a var kulcsszót követően, úgy, hogy azok egész szám, azaz `integer` típusú (integer típus: pl. a Pascalban -32768 és 32767 közötti egész számok) értékekre hivatkozhatnak.

```
program Pelda;
var
 x,y:integer;
begin
  y = 4;
  x = 2*y+3;
  writeln(x);
end.
```

Az eredmény:

```
11
```

A program a korábban látott feladatot végzi el, kiszámol egy egyszerű képletet, majd az eredményt kiírja a képernyőre. Ha a deklarálást elhagyjuk, akkor nem működik programunk, hibaüzenetet kapunk a fordítótól. Ha a programunkat további utasításokkal látnánk el `x` és `y` ugyan hivatkozhatna új értékekre, de továbbra is csak `integer` típusúakra.

Azokat a programozási nyelveket, amelyekben a változók előre meghatározott típusú értékekre hivatkozhatnak, statikusan típusos programozási nyelveknek nevezzük.

Azok a nyelvek, amelyeknél nincs ilyen megkötés, tehát a változók különböző típusú értékekre is hivatkozhatnak, dinamikusan típusos programozási nyelvek.

A Ruby a dinamikusan típusos programozási nyelvek közé tartozik. Nézzünk erre egy példát.

```
x = 42
puts x
x = 'negyvenkettő'
puts x
```

Az eredmény:

```
42
negyvenkettő
```

A programban az `x` változó először számra, majd szövegre hivatkozott, ez nem okozott problémát az értelmezőnek, a program működött, végrehajtódtak a kiíró utasítások.

3.4. 4.3.4 A forráskód

Vizsgáljuk meg általánosságban a forráskód tulajdonságait. A forráskódnak legalább két követelménynek kell megfelelnie, olvashatónak kell lennie a programozó számára és olvashatónak kell lennie az értelmező vagy a fordító számára.

Az első feltétel mondhatjuk, hogy automatikusan teljesül, mivel a programozó az általa ismert programozási nyelvet használja, tehát el is tudja olvasni, értelmezni is tudja azt. Valójában egy összetett program értelmezése nehézkessé válhat a nyelv ismerete ellenére is, ha a programozók nem követnek bizonyos konvenciókat. Ezen konvenciók - egyezmények – olyan szokások, amelyek következetes alkalmazása lehetővé teszi, hogy egy programozói közösségben egy kódot mindenki ugyanúgy értelmezzen, ezáltal a csapat hatékonyabban dolgozhasson együtt. Némelyik konvenció egy adott programozási nyelvhez kötődik, mások egy közösség által kialakítottak, vagy éppen az alkalmazott programozási módszerek, technikák következtében formálódnak.

Léteznek olyan egyezmények, amelyek általánosan elterjedtek a programozók között függetlenül attól, hogy milyen nyelvben programoznak, ilyen a forráskód dokumentálásának és a kód strukturálásának szükségessége.

A dokumentálás során megjegyzéseket (*comment*) írunk a kódba, amelyet az értelmező vagy a fordító figyelmen kívül hagy, de a programozót segíti a kód értelmezésében. A megjegyzéseket rendszerint valamilyen speciális karakter vagy karakterkombináció vezeti be, amelyet követ az emberi nyelven megfogalmazott mondandó.

A kód strukturálása az utasítások vagy az összetartozó utasításokból alkotott nyelvi egységek egyfajta hierarchikus elrendezését jelenti. Ez szintén a programozó számára fontos az olvashatóság miatt.

3.5. 4.3.5 A szintaktika

A programozási nyelveknek, mint az emberi nyelveknek van nyelvtana, ezt szintaktikának hívjuk. A szintaktika meghatározza az utasítások, kifejezések, nyelvi elemek leírásának módját.

Abban az esetben, ha a szintaktikai szabályokat megszegjük, az értelmező hibaüzenetet küld és a program futása leáll. A fordítót használó programozási nyelveknél a fordító küld hibaüzenetet és a fordítás lesz sikertelen.

A programozási nyelvekben léteznek olyan szavak, amelyek speciális jelentéssel bírnak az értelmező vagy a fordító számára, ezeket lefoglalt szavaknak vagy kulcsszavaknak nevezzük. A kulcsszavakat a felhasználó csak a rendeltetésüknek megfelelően használhatja, egyébként a fordítás vagy a program futása szintaktikai hibával megszakad. Ilyen kulcsszavak a Rubyban például a `while` , az `if` vagy az `end` .

3.6. 4.3.6 A program funkcionalitása

Egy program végrehajtható utasításokból épül fel. Alapvetően, a program végrehajtása utasításról-utasításra történik, abban a sorrendben, ahogyan ezt a forráskódban leírtuk. Ettől a sorrendtől azonban eltérhetünk, például ismétlődő utasításokat alkalmazhatunk vagy valamilyen logikai feltétel teljesüléséhez köthetjük az utasítások végrehajtását. Ezt vezérlési szerkezetekkel és metódusokkal valósíthatjuk meg.

3.6.1. 4.3.6.1 Vezérlési szerkezetek

A programunk funkcionalitásának, viselkedésének leírásakor lényegében algoritmust készítünk, algoritmizálunk. Az algoritmusok építőelemei a szekvencia, szelekció és iteráció, amelyek a programozási nyelvekben a nyelvi elemek segítségével, a nyelv által biztosított szintaktikával, kulcsszavakkal írhatók le.

A szekvenciát néhány nyelv nem jelöli külön, ugyanis ez csupán az egymás utáni utasítások sorozatát jelenti, más nyelvekben un. blokk utasításként adható meg a szekvencia, jelölve az utasítások csoportjának kezdetét és végét.

A szelekció és az iteráció vezérlési szerkezetekkel valósítható meg. A szelekciót feltételes szerkezetekkel, az iterációt ciklusokkal írhatjuk le. Ezek megvalósítása a különböző programozási nyelveken nagyon hasonló, szintaktikailag apróbb eltérések előfordulnak.

3.6.2. 4.3.6.2 Metódus

A program felépítésének kialakítása során a programozók gyakran döntenek úgy, hogy egy-egy algoritmust különálló egységként kódoljanak, amelyek később többször is felhasználhatóak. Az egyes programozási nyelvektől, a hagyományoktól és a programozási stílusoktól függően ezekre a programrészekre a következő elnevezések használatosak: rutin (*routin*), szubrutin (*subroutin*), eljárás (*procedure*), függvény (*function*), tagfüggvény (*member function*), metódus (*method*). A Ruby programozási nyelv a metódus elnevezést használja.

Egy metódus végrehajtása csak abban az esetben valósul meg, ha azt megadjuk a forráskódban, vagyis meghívjuk a metódust. A metódus meghívásakor végrehajtódnak a metódusban definiált utasítások, majd a program vezérlése visszatér arra a helyre, ahonnan a metódust meghívtuk és a következő utasítással folytatódik a program végrehajtása.

A metódusoknak lehetnek paraméterei és lehet visszatérési értéke. A Rubyban minden metódus rendelkezik visszatérési értékkel.

A programozási nyelvek futtató környezete (*runtime environment*) számos metódust biztosít, amelyet közvetlenül felhasználhat a programozó.

3.7. 4.3.7 Programozási paradigmák

A programozás fejlődése során különböző programozási stílusok, más néven paradigmák alakultak ki. A programozási paradigma (*programming paradigm*) egyfajta gondolkozásmód, amely a paradigma fogalmi keretrendszerét és a megfelelő programozási nyelvet felhasználva lehetővé teszi programok elkészítését az adott stílusban.

Mindegyik paradigma különböző nézőpontot, különböző gondolkodásmódot igényel egy programozási probléma megoldásához.

Napjainkban a legelterjedtebb az objektum-orientált (*object-oriented*) programozási paradigma, amelyet széleskörűen használnak különféle alkalmazások fejlesztéséhez.

A programozási nyelveket jellemzi, hogy mely stílusokat támogatják. A Pascal strukturált programozási nyelv, míg a Ruby objektum-orientált. Egy nyelv akár több paradigmát is támogathat, így a Ruby, bár alapvetően objektum-orientált, támogatja a funkcionális programozást és a metaprogramozást is.

4. 4.4 A Ruby programozási nyelv

A Ruby egy objektum-orientált, értelmezett, dinamikusan típusos, nyílt forráskódú programozási nyelv. A Ruby első nyilvános kiadása 1995-ben jelent meg, a nyelv alkotója Yukiro Matsumoto, a nicknevén említve Matz. A fejlesztés 1993-ban kezdődött, több nyelv - köztük a Python, a Perl és a Smalltalk - is hatással volt a Ruby kialakítására.

2004-ben David Heinemeier Hansson létrehozta a nyílt forráskódú Ruby on Rails (RoR) keretrendszert webalkalmazások fejlesztésére, amely nagy lendületet hozott a Ruby elterjedésében.

4.1. 4.4.1 Programok írása, futtatása

Ruby programok írásához egy (szöveg)szerkesztőre (*editor*) van szükségünk, amely azonnal rendelkezésünkre áll, ha Windowson dolgozunk és a One-Click Installer programmal telepítettük a Rubyt a gépünkre. A SciTE szerkesztő a Start menüből indítható. Természetesen, bármely más, egyszerű editor, akár a Jegyzettömb is, alkalmas a Ruby forráskód megírására. A forráskódot `.rb` kiterjesztésű állományként mentjük el, amelyet a Ruby értelmező végre tud hajtani.

Egy másik lehetőség Ruby programkódok írására és futtatására az interaktív Ruby értelmező (irb) használata. Windows környezetben az fxri programot is használhatjuk, amely az értelmezőn kívül a Ruby dokumentációját is tartalmazza. Az irb kiváló eszköz tanulási célokra ugyanis egy parancs futtatásának az eredményét azonnal láthatjuk a képernyőn. Az irb parancssorból az `irb` parancs kiadásával indítható.

```
irb(main):001:0> 2+2
=> 4
irb(main):002:0>
```

Az irb jelzései:

- a `001:0` azt jelenti, hogy az első sorba gépeltük be a parancsot és a kód, a tagolás szempontjából, a legfelső szinten található,
- a `=> 4` azt jelenti, hogy az értelmező kiértékelte a parancsot és ennek eredménye: `4`.

Az tananyagban bemutatott néhány soros kódrészleteket célszerű az irb-ben kipróbálni, míg a hosszabb programokat a SciTE szerkesztővel elkészíteni és `.rb` kiterjesztésű állományként menteni.

4.2. 4.4.2 A szintaktikáról

4.2.1. 4.4.2.1 Elnevezések

A nevekkel, más néven, azonosítókkal hivatkozunk az osztályokra, modulokra, változókra és a konstansokra.

A Rubyban az első karakter határozza meg a név használatát. Például az osztályok nevei nagy kezdőbetűvel, míg a változók nevei kis kezdőbetűvel írandók.

A nevekben használható karakterek a következők:

- az angol ABC kisbetűi, "a"-tól "z"-ig,
- az angol ABC nagybetűi, "A"-tól "Z"-ig,
- számok, "0"-tól "9"-ig,
- és az alábbi karakterek: _ , @ , $

A nevek kezdődhetnek kisbetűvel, nagybetűvel vagy aláhúzás karakterrel (_), amelyet a nevekben használható karakterek követhetnek.

Meghatározott jelentéssel bírnak a nevek első karaktereként használandó következő karakterek:

- `$` - a globális változó (*global variable*) nevének első karaktere
- `@` -a példányváltozó (*instance variable*) nevének első karaktere
- `@@` - az osztály változó (*class variable*) nevének első kettő karaktere

A nyelvi környezet által megkövetelt szabályok megsértése esetén az értelmező hibát jelez.

4.2.2. 4.4.2.2 Konvenciók

A nyelvi környezet által megkövetelt szabályokon felül az elnevezésekben léteznek a Ruby programozók körében kialakult konvenciók, más szóval megállapodások. Ezek betartása nem kötelező, de célszerű, mivel így könnyebben olvashatók a különböző programozók által írt kódok. Ezek a következők:

- A konstansok csupa nagy betűvel írandók.
- A többszavas elnevezések változók vagy konstansok esetén _ aláhúzás jellel tagoltak: `ez_egy_valtozo` `EZ_EGY_KONSTANS`
- Az osztályok és a modulok nevei, ha több szóból állnak, CamelCase írásmóddal tagoltak. A CamelCase írásmódnál a szavak nagybetűvel kezdődnek. Az elnevezés abból ered, hogy az íráskép "púpos", mint a teve.
- A metódusok neve utalhat a metódus jellegére, oly módon, ha a metódus neve:
  - felkiáltójelre ! (a metódus módosítja az objektum állapotát),
  - kérdőjelre ? (a metódus logikai értéket ad vissza),
  - vagy egyenlőségjelre = (értékadást végez a metódus) végződik.

4.2.3. 4.4.2.3 Megjegyzések a kódban

Megjegyzéseket a # karaktert követően írhatunk a kódba. A # karaktert követő szöveget az értelmező figyelmen kívül hagyja a sor végéig.

4.3. 4.4.3 A Ruby és az OO paradigma

Az objektum-orientált paradigma fogalmainak tárgyalására a jegyzet egy későbbi fejezetében térünk ki, azonban a Rubyval való ismerkedéskor már felhasználjuk a Ruby objektum-orientált jellemzőit.

Amikor a Rubyban számokkal, szövegekkel és egyéb adatokkal dolgozunk, akkor lényegében objektumokat használunk.

Korábban, azt állítottuk, hogy a változókkal különböző típusú értékekre hivatkozhatunk. Ezt egészítsük ki azzal, hogy a Rubyban ezek az értékek un. objektumok. Ez azt jelenti, hogy nem csupán egy adat áll a rendelkezésünkre, hanem több olyan előre elkészített algoritmus, amelyet a Ruby futtató környezet biztosít számunkra. Ezek az algoritmusok metódusokként kezelhetők. A metódusok alapvetően az objektumnév.metódusnév szintaktikával hívhatók meg. Nézzünk erre egy példát:

```
x = 'negyvenkettő'
```

Az `x` változó a `'negyvenkettő'` objektumra hivatkozik. Hívjuk meg az objektum egy metódusát, amely megadja a szöveg karaktereinek számát:

```
x.length
```

Az eredmény:

```
13
```

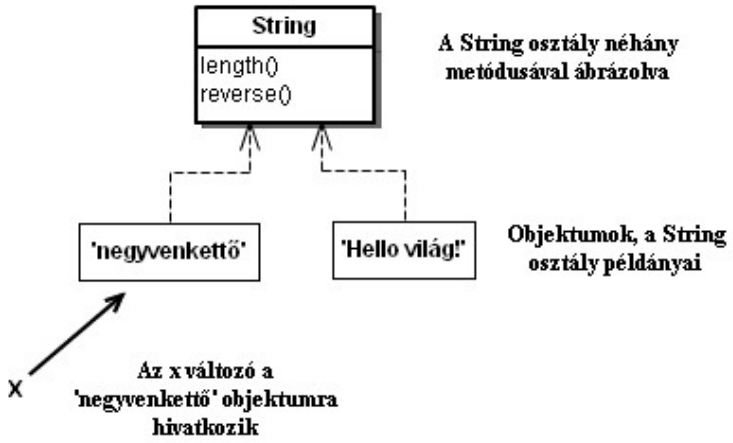
A `reverse` metódus fordított sorrendben adja vissza a szöveg karaktereit:

```
x.reverse
```

Az eredmény:

```
"őtteknevygen"
```

Azt, hogy egy objektumnak milyen metódusai léteznek, az objektum típusa, más néven, osztálya határozza meg. A `'negyvenkettő'` objektum szöveges típusú, ezt a típust a Rubyban String osztálynak nevezik. A 'negyvenkettő' tehát String osztály típusú, azaz a String osztály egy példánya, objektuma (2. ábra).



2. ábra A String osztály és kettő objektuma

4.4. 4.4.4 Számok

A számok a Rubyban egész számnak minősülnek, ha tizedespont nélkül vagy aláhúzás karakterrel tagoltan írjuk azokat. Egész számok például a következők:

```
1
234
1_000_000
1000000
```

A tizedesponttal megadott számokat a Ruby lebegőpontos számoknak (*floating point* vagy *float*), másképpen valós számoknak tekinti.

A számokkal példaként végezzük el a négy matematikai alapműveletet.

```
1+2
10-100_000
3*3
7/5
7.0/5.0
```

Az eredmény sorrendben:

```
3
-99990
9
1
1.4
```

Az első három művelet nem okozott meglepetést, azonban az osztás eredménye némi magyarázatra szorul. A `7/5` esetén egész számokat osztottunk, így az eredmény egy egész szám, az osztás eredményének egész része. A `7.0/5.0` esetén valós számokkal végeztünk műveletet, így az eredmény is valós szám lett.

Próbáljon ki néhány összetett műveletet, mint például `3*(12-5)+36/2` . A zárójelek megváltoztatják a kifejezés kiértési sorrendjét, ahogyan ezt el is várjuk.

A matematikai és logikai műveletekre tehát a kiértékelési sorrend, precedencia a jellemző. Az azonos precedencia szinten álló műveletek kiértékelése balról jobbra történik, a magasabb precedencia szintű műveletek előnyt élveznek a kiértékelésben és a zárójellel határolt műveletek külön egységként értékelődnek ki. Az 1. táblázat a Rubyban használható műveleteket mutatja, ahol az azonos sorban található műveletek azonos, a felsőbb sorokban található műveletek magasabb precedencia szintűek.

1. táblázat - Műveletek kiértékelési sorrendje

Művelet	Leírás
[] []=	hivatkozás, halmaz
**	hatványozás
! ~ + -	nem (ellentett), kiegészítés, pozitív előjel, negatív előjel
* / %	szorzás, osztás, modulo
+ -	összeadás, kivonás
>> <<	jobbra tolás, balra tolás
&	bitenkénti és
^ |	bitenkénti vagy, kizáró vagy
> >= < <=	összehasonlító operátorok
<=> == === != =~ !~	egyenlőségvizsgálat és minta illesztés
&&	logikai és
||	logikai vagy
.. ...	magában foglaló és kizáró tartomány
?:	háromtényezős (if-then-else) operátor
= %= /= -= |= &= <<= >>= *= &&= ||= **=	hozzárendelések
defined?	ellenőriz egy megadott elemet
not	logikai nem
or and	logikai vagy, logikai és

Az egész számok a Bignum vagy a Fixnum osztályok példányai. 32 bites Ruby futtató környezet esetén a -230 és a 230-1 közötti egész értékek Fixnum osztályúak, a tartományon kívül eső egész számok Bignum objektumok. A Ruby automatikusan a megfelelő típusúra konvertálja az objektumainkat. Így például, ha egy számítási feladatnál, az eredményünk nagyobb lesz, mint 230-1, automatikusan Bignum objektumot kapunk.

A valós számok a Float osztály objektumai.

A legtöbb matematikai és logikai művelet metódusként áll rendelkezésünkre. Azonban nem kell az objektumnév.metódusnév szintaktikát ráerőltetni a számításainkra használhatjuk a megszokott matematikai és logikai formulákat. Így is írhatnánk például az összeadást: `4.+5` , azonban a `4+5` formula kényelmesebben használható.

4.5. 4.4.5 Karakterláncok

A karakterláncok karakterek sorozatából épülnek fel, amelyeket szimpla idézőjelek (más néven: félidézőjelek) `''` vagy dupla idézőjelek `""` határolnak. A karakterek száma tetszőleges, ha nem adunk meg egy karaktert sem, akkor üres karakterláncot hozunk létre.

Például:

```
''
""
'Nyugat-magyarországi Egyetem Geoinformatikai Kar'
'123456789+!%/=()§'
"alma"
```

Előfordulhat, hogy olyan karaktert írnánk a karakterláncba, amely az értelmező számára valamilyen jelentéssel bír. A következő szövegnél nem lenne egyértelmű a fordító számára, hogy meddig tart a karakterlánc.

```
I'm Bond. James Bond.
```

Szimpla idézőjelek közé téve, az értelmező nem tudja megkülönböztetni a karakterlánc kezdetét jelölő idézőjelet a szövegben találhatótól. Az értelmező számára az idézőjelek a karakterlánc kezdetét és végét jelentik.

Erre egy megoldás a feloldójel karakter (*escape character*) használta, amely azt jelzi, hogy az utána következő karaktert másként kell értelmezni.

```
'I\'m Bond. James Bond.'
```

A feloldójel a fordított perjel \ (*backslash*) karakter, amely lehetővé teszi a szimpla idézőjel használatát a karakterláncon belül.

Egyszerűbb dolgunk van, ha dupla idézőjelekkel határoljuk a fenti szöveget. Ekkor az értelmező szimpla idézőjeleket vagy az aposztrófot `'` a karakterlánc részeként értelmezi.

```
"I'm Bond. James Bond."
```

A szimpla idézőjel használata tehát egyszerűbb megoldást kínál, a dupla idézőjelekkel határolt karakterláncok összetettebb írásmódot tesznek lehetővé. A feloldójel mellett használatos a feloldójel szekvencia (*escape sequence*), amely együtt jelöli a feloldójelet és a módosított karaktert. A dupla idézőjelek közötti karakterláncok esetében számos feloldó szekvencia használható. Például a `\n` egy új sor karaktert, míg a `\t` egy tabulátor karaktert jelent.

4.6. 4.4.6 Kiíratás a képernyőre

Ahhoz, hogy a karakterláncok használatát kipróbáljuk ismerkedjünk meg a képernyőre történő kiíratás lehetőségeivel.

Probáljuk ki:

```
print 2*5
print 'alma'
```

Az eredmény az irb-ben:

```
10=> nil
```

és

```
alma=> nil
```

Az eredmény, a kódot állományként (pl.: `ruby alma1.rb`)végrehajtva:

```
10alma
```

Próbáljuk ki:

```
puts 2*5
puts 'alma'
```

Az eredmény az irb-ben:

```
10
=> nil
```

és

```
alma
=> nil
```

Az eredmény, a kódot állományként (pl.: `ruby alma2.rb`) végrehajtva:

```
10
alma
```

A `print` és a `puts` metódusok egyaránt a képernyőre írnak. A `puts` a kiirandó szöveg után egy új sor karaktert is kiír, ezért az `alma` szöveg már egy sorral a `10` alatt olvasható. A kiíratandó adatokat a képernyőre írás előtt az értelmező szöveggé alakítja át. A `puts` könnyen megjegyezhető a *put string* kifejezésből.

Az irb kiértékelte az utasításokat, és `nil` értéket adott vissza. A `nil` jelentése: semmi. Ha a képernyőre írunk, akkor mindig `nil` értéket eredményez a művelet.

Az irb, mivel az utasításainkat sorról-sorra hajtja végre, nem jeleníthette meg a `10` -es számot és az `alma` szöveget közvetlenül egymás mellett vagy alatt. Ebben az esetben a `=>` karakterek helyzete jelezi a `print` és a `puts` közötti különbséget.

4.7. 4.4.7 Műveletek karakterláncokkal

A karakterláncok könnyedén összefűzhetők a konkatenáció művelettel, amelyet a `+` jel jelez.

```
puts 'alma'+'fa'
```

Az eredmény:

```
almafa
```

A karakterláncokat többszörözhetjük a szorzás `*` művelet használatával.

```
puts 'egy kettő  '*2
```

Az eredmény:

```
egy kettő egy kettő
```

Feloldó szekvenciák használata:

```
puts "A Ruby jellemzői:\n\tobjektum-orientált\n\tnyílt forráskódú\n\t..."
```

Az eredmény:

```
A Ruby jellemzői:
       objektum-orientált
       nyílt forráskódú
       ...
```

4.8. 4.4.8 Kifejezés-behelyettesítés

A dupla idézőjellel határolt karakterláncokba tetszőleges Ruby kifejezést illeszthetünk a `#{}` operátor segítségével.

```
x = 16
puts "x értéke: #{x}"
```

Az eredmény:

```
x értéke: 16
```

Az értelmező az `#{x}` formulát nem szöveges karakterekként értelmezte, hanem behelyettesítette az `x` változóhoz rendelt objektum értékét.

4.9. 4.4.9 Konstansok

Azok az értékek, amelyek a program futása során nem változnak, állandóként, más szóval, konstansként (*constant*) definiálhatók. A konstansok nevei nagy betűvel kezdődnek, ezt a nyelvi környezet megköveteli. Általában, megállapodás szerint, a konstansok csupa nagy betűvel írandók. A több szóból álló konstansokat aláhúzás `_` karakter választja el.

A Rubyban az osztályok és a modulok nevei is konstansok. Megállapodás szerint, ezek neveit nagy kezdőbetűvel írjuk, ha több szóból állnak, akkor a CamelCase írásmódot alkalmazzuk.

```
KONST = "ez egy állandó"
MERFOLD = 1609.3
CM_PER_INCH = 2.54
Osztalynev
ValamilyenOsztaly
```

Egy osztályban vagy modulban definiált konstans bárhol elérhető az osztályon vagy a modulon belül. Osztályon vagy modulon kívül a `::` operátorral hivatkozhatunk a konstansokra, megadva az osztály vagy a modul nevét, amely tartalmazza azt.

```
Math::PI
```

Konstansok nem definiálhatók a metódusokban, még az `initialize` metódusban sem. A konstans definíciók elhelyezhetők például az osztálydefiníció első soraiban, az `initialize` metódus előtt.

4.10. 4.4.10 Tartományok

A tartományok két megadott érték közötti objektumokat reprezentálnak. A magában foglaló tartomány esetén a megadott értékek is elemei a tartománynak, míg a kizáró tartomány esetén nem elemei.

```
1..10 #magában foglaló tartomány, elemei: 1,2,3,4,5,6,7,8,9,10
'a'..'z' #elemei karakterek
0...10 #kizáró tartomány, elemei: 2,3,4,5,6,7,8,9
0..."cat".length #valójában: 0...3, kizáró tartomány, elemei: 1,2
```

Példák:

```
sorozat1 = 1..10
sorozat1.max
=> 10
sorozat2 = 0...10
sorozat2.include?(10)
=> false
```

A tartományok a `to_a` metódussal tömbökké alakíthatók.

4.11. 4.4.11 Szimbólumok

A szimbólumok egyszerű adatszerkezetek, hasonlítanak a karakterláncokhoz, rendszerint valamilyen azonosító funkciót látnak el. Például az asszociatív tömböknél szimbólumokat alkalmazhatunk a kulcsok megadására.

```
:szimbolum
:nev
```

Két karakterlánc azonos tartalommal lehet két különböző objektum, míg két azonos tartalmú szimbólum mindig azonos objektumot jelent.

4.12. 4.4.12 Tömbök

Egy tömb objektum objektumok indexelt (sorszámozott) sorozatából épül fel. Egy tömbben bármilyen objektumot elhelyezhetünk. Az elemekre az indexekkel hivatkozhatunk, a `0` indexű elem a tömb első eleme, a `-1` indexű elem a tömb utolsó elemét jelenti. A tömb elemeire tehát, a tömb elejéről és végéről is hivatkozhatunk, így az `1` indexű elem a tömb második eleme, míg a `-2` indexű elem az utolsó előtti elemet jelenti.

```
tomb = Array.new vagy tomb = []
gyumolcsok = ['eper','narancs','szilva']
puts gyumolcsok[0] # az első elem: eper
puts gyumolcsok[1] # a második elem: narancs
puts gyumolcsok[-1]# az utolsó elem: szilva
honapok = Array(1..12)
=> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

4.13. 4.4.13 Asszociatív tömbök

Az asszociatív tömb kulcs – érték párok halmaza, az értékekre a kulcsokkal hivatkozhatunk. A Rubyban az asszociatív tömböt a Hash osztály definiálja.

```
szamok = Hash.new
szamok["egy"] = 1
szamok["ketto"] = 2
```

vagy

```
szamok = {:egy => 1, :ketto => 2}
```

Az alábbi példában a hash szerkezet a fájok nevét és az MD5 kódolással előállított ujjlenyomatát tárolja, az állomány eredetiségének, sértetlenségének ellenőrzése céljából. (Az MD5 algoritmus egy tetszőleges hosszúságú adatsorból állandó hosszúságú kimenetet eredményez.)

```
md5 = {"explorer.exe" =>"4f554999d7d5f05daaebba7b5ba1089d"}
```

Ha a fájl megváltozott, például vírusos, akkor az ismételten előállított ujjlenyomat eltér az eredetitől.

4.14. 4.4.14 Az objektum osztálya és metódusai

A `class` metódus meghívása visszaadja az objektum osztályát.

```
12345.class
=> Fixnum
"alma".class
=> String
```

A `methods` metódus meghívása visszaadja az objektum által használható metódusokat.

```
"alma".methods
=>["upcase!","zip","find_index","between?","to_f","minmax",
```

```
lines","sub","methods","send","replace","empty" (és így tovább)
```

4.15. 4.4.15 Logikai objektumok

A Rubyban a logikai értékek is objektumok, amelyek az alábbiak lehetnek:

- `true` - igaz
- `false` - hamis
- `nil` - jelentése: semmi.

Több metódus - pl.: `puts` , `print` stb. - `nil` értéket ad vissza.

4.16. 4.4.16 Algoritmusok a kódban

Az algoritmusok kódolására a programozási nyelvekben különböző vezérlési szerkezetek állnak rendelkezésre. A Rubyban a szelekciókat feltételes szerkezetekkel, az iterációkat ciklusokkal és iterátorokkal valósítjuk meg.

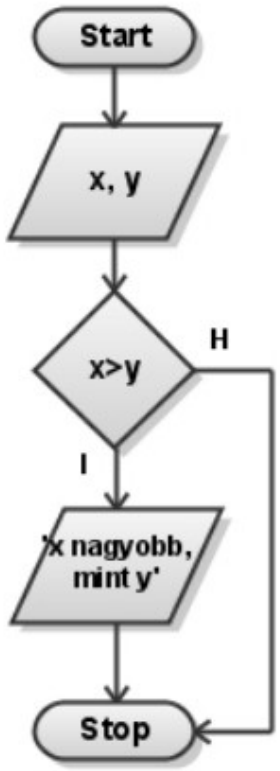
Tekintsük át a Ruby programozási nyelv által biztosított vezérlési szerkezeteket.

4.16.1. 4.4.16.1 Feltételes szerkezetek

Egy logikai kifejezés eredményétől függően hajtódnak végre az utasítások. Az algoritmus szelekció elemét valósítják meg.

if

A legegyszerűbb az `if` (ha) szerkezet, amely szerint, ha az `if` kulcsszót követő logikai kifejezés igaz (`true` értékű), akkor végrehajtódnak a szerkezetben megadott utasítások; amennyiben a logikai kifejezés értéke hamis, az utasítások nem hajtódnak végre.




```
if x > y

  puts 'x nagyobb, mint y'


end
```

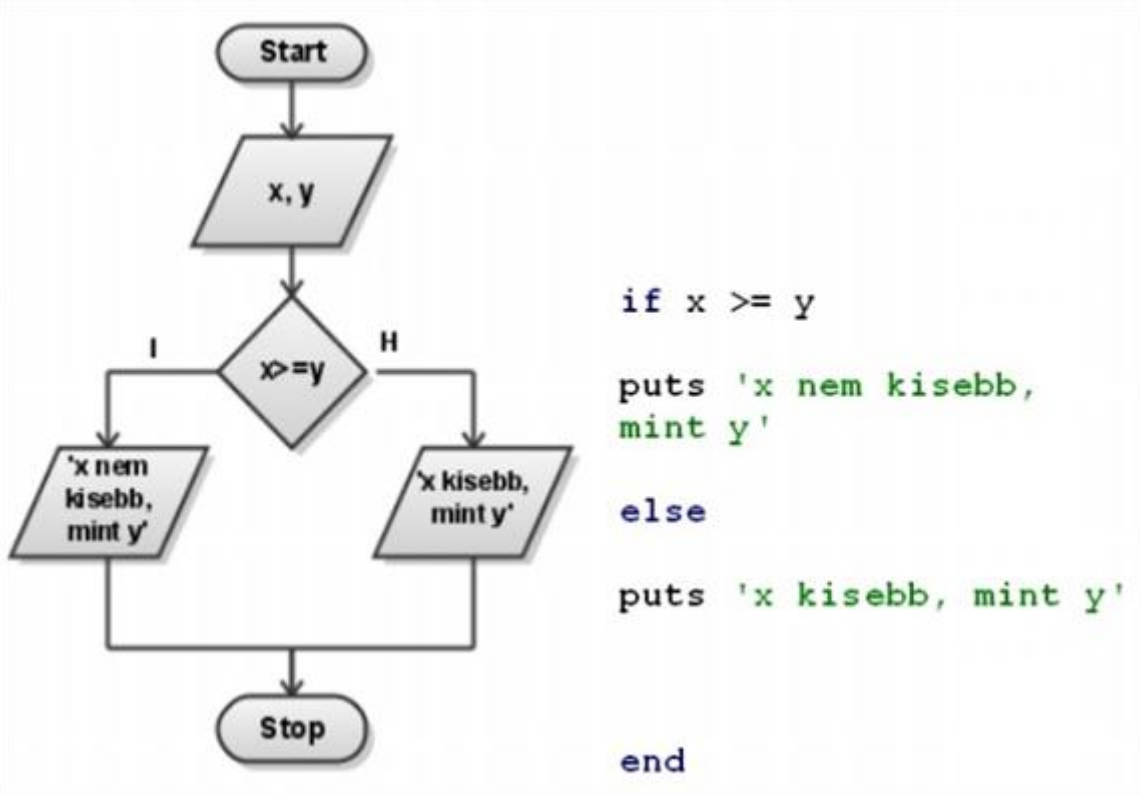
Az `if` szerkezet egy másik változata lehetővé teszi, hogy egyszerűen az utasítással azonos sorba, az utasítást követően írjuk a feltételt:

```
puts 'x nagyobb, mint y' if x > y
```

Az `if` ebben az esetben un. utasítás-módosító szerkezetként hajtódott végre. Az eredmény mindkettő példa esetén azonos, az `x` és az `y` változók értékétől függő.

if else

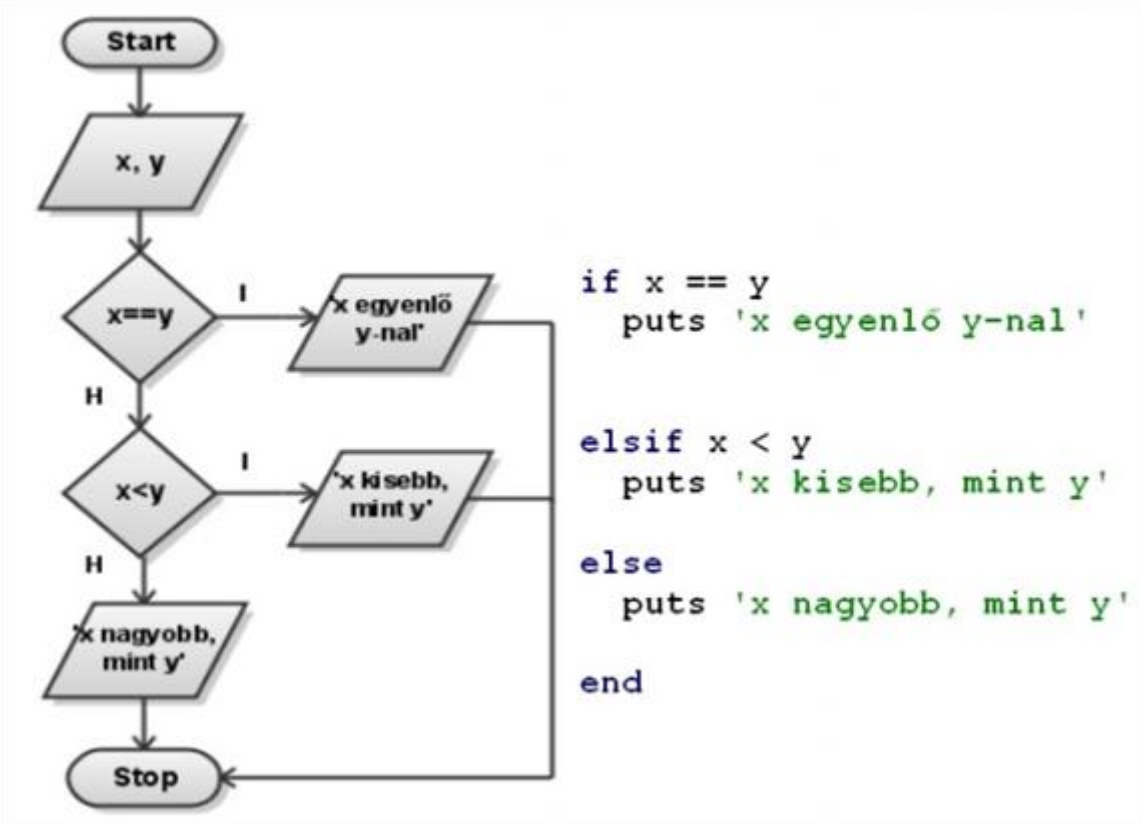
Az `if else` (ha, egyébként) szerkezet un. kétágú szelekció, a feltétel kiértékelésétől függően az igaz vagy a hamis ágon található utasítások hajtódnak végre. A hamis ágat az `else` kulcsszó vezeti be.



4. ábra Az if else szerkezet folyamatábrája és kódja

elsif

Az `elsif` szerkezet többágú szelekció, több feltétel tartalmaz. A feltételek kiértékelésétől függően az egyes ágakon hajtódnak végre az utasítások. A hamis ág, az `else` kulcsszót követően, akkor hajtódik végre, ha egyik feltétel sem teljesül.



5. ábra Az elsif szerkezet folyamatábrája és kódja

Az ábrán egy `elsif` ágat látunk, azonban több is szerepelhet a szerkezetben.

A Ruby nyelvi környezet lehetővé teszi, hogy a hamis ág, az `else` kulcsszóval, elhagyható legyen. Ekkor azonban, ha egyetlen feltétel sem bizonyul igaznak, nem történik kódvégrehajtás.

unless

Az `if` ellentettje, akkor hajtódnak végre az utasítások, ha a feltétel nem igaz. Az `if` szerkezetekhez hasonlóan használható, az `unless end` szerkezet vagy az utasítás végére írt `unless` megoldás, amely utasítás-módosítókét viselkedik.

```
puts 'x nagyobb, mint y' unless x <= y
```

unless else

Az `if else` szerkezethez hasonlóan használható az `unless else` szerkezet. A következő táblázatban található kódok az `if else` és az `unless else` használatát mutatják. A példa egy alkalmazás honosítását szimbolizálja, a felhasználó által választott nyelvtől függően jelenhetne meg a felhasználói felületen a kutya vagy a dog szó. Az `if else` szerkezetnél a `@d` példányváltozó akkor hivatkozik a `'kutya'` karakterláncra, ha a `lang` változó a `'hu'` objektumra mutat, vagyis teljesül a feltétel. Az `unless else` szerkezet ugyanazt eredményezi, de a feltétel teljesülése az `else` ág végrehajtásával jár.

2. táblázat - Az if else és az unless else szerkezetek

if else - "ha igaz, egyébként"	**unless else - "ha nem igaz, egyébként"**
`if lang = = 'hu'`	`unless lang = = 'hu'`

```
  @d = 'kutya'
else
  @d = 'dog'
end
```

```
  @d = 'dog'
else
  @d = 'kutya'
end
```

Az `elsif` , többágú szelekciós szerkezet `unless` esetén nem használható!

case

A `case` szerkezet az `elsif` szerkezethez hasonló többágú szelekció, esetszétválasztásnak is nevezik. Az `elsif` -nél tömörebb írásmódot tesz lehetővé.

```
lang = 'hu'
d = case lang
 when 'de' then 'Hund'
    when 'hu' then 'kutya'
 when 'fr' then 'chien'
 else           'dog'
 end
puts d
```

Az eredmény:

```
kutya
```

Az interpreter a `case` kulcsszót követő `lang` változóhoz rendelt karakterláncot sorban összehasonlítja a `when` kulcsszó után megadott karakterlánccal. Egyezőség esetén a `then` kulcsszó után megadott utasítás hajtódik végre, vagyis az interpreter a `d` változót a `'kutya'` karakterlánchoz rendeli. Ha nem adódott volna egyezőség, akkor az `else` kulcsszót követő utasítás hajtódott volna végre.

A következő példánál egy másik szintaktikát látunk, amely azt mutatja, hogy a then kulcsszó használatára nincs szükség, ha új sorba írjuk az utasításokat. A feltételeket itt tartományokkal adtuk meg.

```
pont = 99
case pont
  when 0..60
    puts "elégtelen"
  when 61..70
    puts "elégséges"
  when 71..80
    puts "közepes"
  when 81..90
    puts "jó"
  else
    puts "jeles"
end
```

Az eredmény:

```
jeles
```

4.16.2. 4.4.16.2 Ciklusok

A Ruby beépített ciklusai a `while` , az `until` és a `for` szerkezetek. A végrehajtandó kódok a ciklusmagban vagy más szóval ciklustörzsben találhatók. A `while` és az `until` feltételes ciklusok, a ciklusmag végrehajtása egy logikai feltétel kiértékelésének eredményétől függ. A `for` un. taxatív vagy számlálásos ciklus, a ciklusmag megadott számú lépésben ismétlődik.

Azt a változót, amelynek értéke, a feltétel teljesülése vagy a számlálás érdekében az egyes lépéseknél megváltozik ciklusváltozónak nevezzük.

while

A `while` ciklus esetén a `while` és a `do` kulcsszavak között megadott feltétel , amely általában egy logikai kifejezés, kiértékelését követően, ha a kifejezés értéke nem `false` vagy nem `nil` ,a ciklusmag végrehajtódik. A ciklusmag tehát az első kiértékeléstől függően nulla vagy több alkalommal hajtódhat végre. A ciklusmag utolsó utasítását követően a program vezérlése a ciklus feltételére ugrik, amely ismét kiértékelésre kerül. Ha a kiértékelés eredménye nem `false` vagy nem `nil` , a ciklusmag ismét végrehajtódik. A ciklus végrehajtása akkor fejeződik be, ha a logikai kifejezés `false` vagy `nil` értékűvé válik.

Egyszerűbben mondhatnánk azt, hogy amíg a feltétel igaz, addig végrehajtódik a ciklusmag. Ez az állítás azonban pontatlan, ugyanis a feltétel elvileg lehet egy objektum is - vagyis nem egy logikai kifejezés-, amely rendelkezik olyan értékkel, ami nem `false` , nem `nil` és nem `true` . Ekkor szintén végrehajtódik a ciklusmag.

```
x = 10            #A ciklusváltozó inicializálása
while x >= 0 do   #Ciklus, amíg x nagyobb vagy egyenlő, mint 0
  print "#{x} "   #x kiíratása
  x = x - 1       #x értékének csökkentése eggyel
end
```

Az eredmény:

```
10 9 8 7 6 5 4 3 2 1 0
```

A `while` utasítás-módosítóként is használható:

```
x = 0
print "#{x = x + 1} " while x < 10
```

Az eredmény:

```
0 1 2 3 4 5 6 7 8 9 10
```

until

Az `until` ciklus, a `while` ciklus ellentettje. Az `until` esetén az `until` és a `do` kulcsszavak között megadott feltétel kiértékelését követően, ha a kifejezés értéke `false` vagy `nil` ,a ciklusmag végrehajtódik. A ciklusmag tehát az első kiértékeléstől függően nulla vagy több alkalommal hajtódhat végre. A ciklusmag utolsó utasítását követően a program vezérlése a ciklus feltételére ugrik, amely ismét kiértékelésre kerül, ha a kiértékelés eredménye `false` vagy `nil` ,a ciklusmag ismét végrehajtódik. A ciklus végrehajtása akkor fejeződik be, ha a feltétel `false` vagy `nil` értéktől különböző értékűvé válik.

```
x = 0             #A ciklusváltozó inicializálása
until x > 10 do   #Ciklus amíg x > 10 false értékű
  print "#{x} "   #x kiíratása
  x = x + 1       #x értékének növelése eggyel
end
```

Az eredmény:

```
0 1 2 3 4 5 6 7 8 9 10
```

Az `until` utasítás-módosítóként is használható:

```
a = [1,2,3]               #Tömb létrehozása
puts a.pop until a.empty? #Kivesszük egyenként a tömb elemeit
                          #és kiíratjuk, amíg a tömb nem üres.
```

Az eredmény:

```
3
2
1
```

for/in

A `for` ciklus egy megszámlálható (*enumerable*) objektumot használ fel az iterációk végrehajtására. Ilyen objektum például egy tömb, egy tartomány vagy egy hash objektum. (Bármely olyan objektum használható, amely rendelkezik az `each` iterátor metódussal.) A ciklusváltozóhoz egyenként hozzárendelődik az objektum eleme és végrehajtódik a ciklusmag.

Egy tömb elemeinek kiíratása `for` ciklussal:

```
tomb = [1,2,3,4,5] #Tömb létrehozása
for elem in tomb   #Az elem a ciklusváltozó
  print "#{elem} " #Az elemek kiíratása
end
```

Az eredmény:

```
1 2 3 4 5
```

Egy hash kulcs-érték párjainak kiíratása:

```
hash = {:a=>1, :b=>2, :c=>3}
for kulcs,ertek in hash
  puts "#{kulcs} => #{ertek}"
end
```

Az eredmény:

```
a => 1
b => 2
c => 3
```

Számok kiíratása tartomány felhasználásával:

```
for i in 1..10
  print "#{i} "
end
```

Az eredmény:

```
1 2 3 4 5 6 7 8 9 10
```

4.16.3. 4.4.16.3 Iterátorok

A while, az until és a for ciklusok mellett speciális metódusok, un. iterátorok, más néven iterátor metódusok is használhatók az iterációk megvalósítására.

Iterátor metódusok, többek között, a `times` , az `each` és az `upto` metódusok. A metódusok hívásakor végrehajtódik a metódusokat követő kódrészlet, a blokk. A blokk utasításait vagy kapcsos zárójel pár `{}` , vagy a `do end` kulcsszó páros határolja.

times

Ha egy egész szám objektum `times` metódusát meghívjuk, akkor annyiszor hajtódik végre a metódushoz kapcsolódó blokk, amilyen értékkel az objektum rendelkezik.

```
3.times {print 'Ruby '}
```

Az eredmény:

```
Ruby Ruby Ruby
```

A fenti kód egy ékes példája annak, hogy a Ruby kódok "beszédesek". Ha egyes kódrészletek működésének megfogalmazás esetenként bonyolultnak tűnik, akkor sokat segíthet a Ruby programozási nyelv ezen jellemzője.

upto

Az `upto` metódust is az egész számokra alkalmazható. A metódus meghívja a kapcsolódó blokkot minden egész szám esetén, amely az objektum és az upto metódus paramétere között található, beleértve az objektumot és a paramétert is.

```
4.upto(6) {|x| print x}
```

Az eredmény:

```
456
```

A blokkban változót is definiálhatunk, amelynek a metódus átadja a paramétereket. A blokk változóját virgula jelekkel `||` határoljuk.

each

A for ciklus tárgyalásánál említésre került az `each` metódus, ugyanis a for ciklus a megszámlálható objektumok `each` metódusát használja fel. A Ruby programozók többsége szívesebben használja az `each` iterátor metódust a `for` ciklus helyett.

Az `each` metódus sorra veszi az objektum elemeit és átadja a blokk változójának.

Tömb elemeinek kiíratása az `each` metódussal:

```
[1,2,3].each do |x|
  print "#{x} "
```

```
end
```

Az eredmény:

```
1 2 3
```

4.17. 4.4.17 Metódusok készítése

4.17.1. 4.4.17.1 Metódus definiálása

A metódusokat a `def` és az `end` kulcsszavak között definiáljuk. A metódust paraméterekkel vagy paraméterek nélkül is megadhatjuk.

```
def metodus_neve(parameter, masik_parameter)
  #ide jön a metódus algoritmusa
end
```

A Rubyban minden metódus rendelkezik visszatérési értékkel. A visszatérési érték a `return` kulcsszót követően adható meg. A `return` kulcsszó hiányában a visszatérési érték megegyezik a metódus utolsó utasításának kiértékeléseként kapott eredménnyel.

4.17.2. 4.4.17.2 A metódusok elnevezése

A konvenciók szerint a felkiáltójelre végződő metódusok azt jelzik, hogy a metódus módosítja az objektumot. A kérdőjel azt jelenti, hogy logikai értéket ad vissza a metódus. Az egyenlőségjel az értékadást jelöli.

Példa kérdőjelre végződő metódusra:

```
tomb = [] #új tömb létrehozása
tomb.empty? #az empty? metódus true vagy false értéket ad vissza
```

Az eredmény:

```
true
```

Példa felkiáltójelre végződő metódusra:

```
tomb = [22,3,10,1]
puts tomb.sort #a sort rendezi a tömb elemeit, majd kiíratjuk
     #az eredmény: 1,3,10,22
puts tomb  #kiiratva az objektumot láthatjuk, hogy a tombben
       #az elemek sorrendje nem változott 22,3,10,1
puts tomb.sort! #a sort! megváltoztatja az objektum állapotát
puts tomb #az eredeti objektum elemei rendezettek: 1,3,10,22
```

5. 4.5 Az objektum-orientált paradigma

5.1. 4.5.1 Bevezetés

A paradigma tárgyalását kezdjük néhány egyszerű megállapítással.

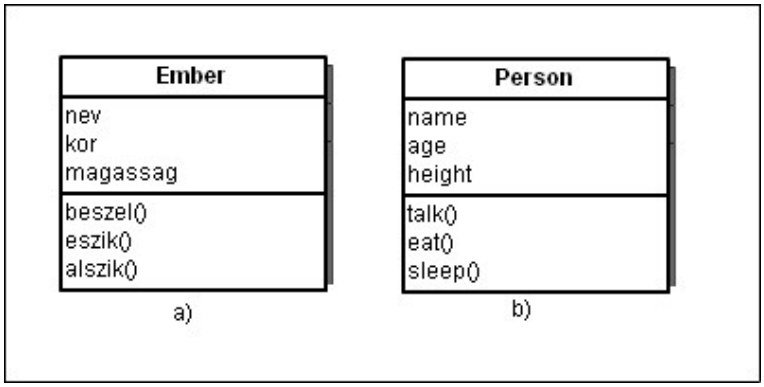
Példaként gondoljunk önmagunkra. Rendelkezünk számos tulajdonsággal, ami meghatároz bennünket, mint például: név, kor, magasság.

A felsorolt adatok, természetesen nem jellemeznek teljes mértékben, akár sok-sok további tulajdonságot leírhatnánk. Mindenesetre, ha a felsorolt jellemzőket gondolatban konkrét értékekkel töltenénk fel, önmagunkra ismernénk.

Azon felül, hogy tulajdonságokkal rendelkezünk, bizonyos tevékenységek elvégzésére is képesek vagyunk. Ezek közül néhány alapvető tevékenység a következő: beszéd, evés, alvás.

Ezen tevékenységek és tulajdonságok leírásával, lényegében megalkottuk az első objektumunkat. Megállapíthatjuk tehát, hogy egy objektum tulajdonságokkal és tevékenységekkel rendelkezik. Az objektumunk tulajdonságait attribútumoknak, tevékenységeit metódusoknak nevezzük.

Az imént felsorolt tulajdonságok és tevékenységek, láthatóan nem csak egy konkrét személyéhez köthetők, kiterjeszthetők a többi emberre is. Az objektumunk tehát egy kategóriába vagy típusba sorolható be, amit a paradigma terminológiája szerint osztálynak nevezünk. Így, az objektumunk az Ember osztályba tartozik, vagy másképpen megfogalmazva, Ember osztály típusú.



6. ábra Az Ember osztály

Az 6. ábra az Ember osztályt ábrázolja, felül az osztály neve, középen az attribútumok, alul a metódusok találhatók. A képen magyarul, a) és angolul, b) is látható az osztály. A magyar ékezetes betűket elhagytuk, mert a programunk írásakor sem használhatjuk azokat az osztályok, attribútumok, metódusok neveiben. Egy metódust, mivel tevékenységre, műveletre utal, igével fejezhetünk ki. A fenti ábrázolási módnál a metódusok neveinek végén zárójeleket () szokás feltüntetni.

Miután az Ember osztályt modelleztük, nézzük meg az osztály használatát Ruby programozási nyelven megfogalmazva.

```
valaki = Ember.new
valaki.alszik()
valaki.eszik(leves)
valaki.beszel
```

A kódban az `Ember` osztályból, a `new` metódus segítségével, létrehoztuk a `valaki` objektumot, és elvégeztettünk vele néhány tevékenységet. Az `eszik()` metódusnak zárójelek között egy paramétert - `leves` - is adtunk. A `beszel` metódusnál viszont elhagytuk a zárójelet. Mindegyik sor szintaktikailag helyesen megírt kód.

Természetesen ahhoz, hogy a metódusok végrehajtódjanak, az osztály forráskódjában meg kellene adni a metódusok algoritmusát, vagyis azt, hogy milyen utasítások hajtódjanak végre az `eszik()` , `alszik()` , `beszel()` metódusok meghívásakor.

A rövid bevezető példa alapján sejthetjük, hogy az objektum-orientált programunk vagy rendszerünk építőelemei az objektumok. Programjainkban objektumokat hozunk létre, amelyek különböző műveleteket hajtanak végre. Az objektum-orientált paradigma alkalmazása tehát azt jelenti, hogy objektumokban gondolkodunk. Az objektum-orientált szemlélet, bár a szoftverfejlesztésben bontakozott ki, számos egyéb területen, többféle célból használatos a valós világ modellezésére. A programozásban a cél: egy működő számítógépes program létrehozása vagy, ha a programunkra, mint termékre tekintünk, egy szoftver elkészítése.

5.2. 4.5.2 Objektum és osztály

Tekintsük át az osztályok és az objektumok jellemzőit, néhány definíció és példa kíséretében.

OO program

Egy objektum-orientált program egymással kommunikáló objektumok összessége. A kommunikáló objektumok üzeneteket küldenek egymásnak. Az üzenetek küldése a forráskódban metódushívásokkal valósul meg.

Programjaink elkészítéséhez rendszerint több osztályt készítünk és az osztályokat külön `.rb` állományokként mentjük. A programunk indításához a főprogramot tartalmazó állományt futtatjuk amelyben hivatkozunk a felhasználandó osztályokra.

A hivatkozást a főprogram első soraiban célszerű elhelyezni a következő módon:

```
require 'fajlnev'
```

A `'fajlnev'` az osztálydefiníciót tartalmazó állomány neve .rb kiterjesztés nélkül megadva.

A `require` kulcsszót minden esetben alkalmaznunk kell, ha egy osztályban olyan osztályra hivatkozunk, amelyet egy másik fájlban tároltunk.

Osztály

Az osztály egy minta, amely alapján objektumokat hozunk létre. Az objektum az osztályban definiált attribútumokkal és metódusokkal jön létre. Az objektumok létrehozását általában egy objektumkészítő metódus meghívásával kezdeményezzük. Az objektum elkészítésekor az objektum adatai kezdőértéket kapnak, erre azt mondhatjuk, hogy az objektum inicializálódik.

Egy osztályt a class kulcsszót követően hozunk létre. Egy rb állományban egy vagy több osztálydefiníciót is elhelyezhetünk.

```
class Ember #ez egy attribútumok és metódusok nélküli osztály
end
```

Egy objektum létrehozása:

```
valaki = Ember.new
```

Objektum

Az objektum az osztály példánya. Az objektum adatokat tárol és kérésre műveleteket végez. Az objektum adatait attribútumoknak, más néven példányváltozóknak nevezzük. A példányváltozó jelölése a Ruby kódban:

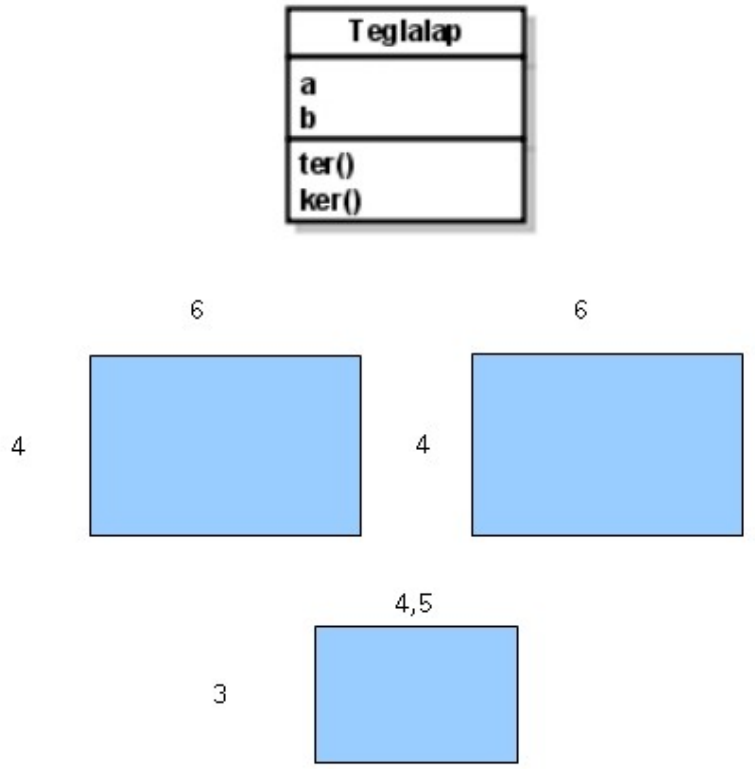
```
@valtozo
```

A műveletvégzést az objektum metódusai hajtják végre.

Az objektum különböző állapotokkal rendelkezik, a pillanatnyi állapotot az objektum attribútumainak pillanatnyi értéke határozza meg. Ha az objektum adatai egy metódus végrehajtása következtében megváltoznak, akkor megváltozik az állapota is.

Az objektumok egyértelműen azonosíthatók és az objektum azonossága független a tárolt értékektől. Vagyis, két azonos típusú és azonos állapotú objektum nem azonos egymással.

A következő ábrán egy `Teglalap` osztály látható `a` és `b` példányváltozókkal, valamint a `ker()` és `ter()` - kerület- és területszámító - metódusokkal. A téglalapok három különböző objektumot szimbolizálnak, utalva az objektumok állapotára és azonosságára.



7. ábra Teglalap objektumok

Példák osztály létrehozására és példányosításra

Az első példában az `Ember` osztály a `@nev` attribútumot tartalmazza, amelynek kezdőértéket adunk. Az attribútumot az `initialize` metódusban helyezzük el. Objektum létrehozása az osztály `new` metódusával valósítható meg. Mivel az objektum létrehozásakor a `new` metódus automatikusan meghívja az `initialize` metódust, így az objektumunk példányváltozója kezdőértéket kap.

```
#az Ember osztály definíciója
class Ember
  def initialize
    @nev = "Éva"
  end
  def alszik
    puts "#{@nev} alszik"
  end
end
#az osztály használata
valaki = Ember.new
valaki.alszik
```

Az eredmény:

```
Éva alszik
```

A második példában az objektum létrehozásakor adunk értéket a példányváltozónak. A `new` metódust paraméterrel hívjuk meg, így a paraméter értékét, amit a `nev` változóban tárolunk, az `initialize` metódusban átadjuk a `@nev` példányváltozónak. Az `eszik` metódust szintén egy paraméterrel hívjuk meg.

```
class Ember
  def initialize(nev)
    @nev = nev
  end
  def eszik(etel)
    puts "#{@nev} #{etel} eszik"
  end
end
#az osztály használata
valaki = Ember.new("Ádám")
valaki.eszik("levest")
```

Az eredmény:

```
Ádám levest eszik
```

5.3. 4.5.3 Alapkoncepciók

Az objektum-orientált programozás alapkoncepcióit, az egységbezárást (*encapsulation*), az öröklődést (*inheritance*) és a többalakúságot (*polymorphism*), gyakran az objektum-orientált programozás három pillérének nevezik.

Egységbezárás

Az egységbezárás azt jelenti, hogy az objektum az adatait és a viselkedésének (működésének) részleteit elrejti a külvilág elől.

Az objektum egy vagy több felületen keresztül érhető el a külvilág, vagyis a többi objektum számára. Ezt a felületet interfésznek (*interface*) hívjuk. Az interfész meghatározza azokat a szolgáltatásokat, amelyeket az objektum biztosít a külvilág számára. Az interfészek tulajdonképpen azok a metódusok, amelyeket más objektumok meghívhatnak.

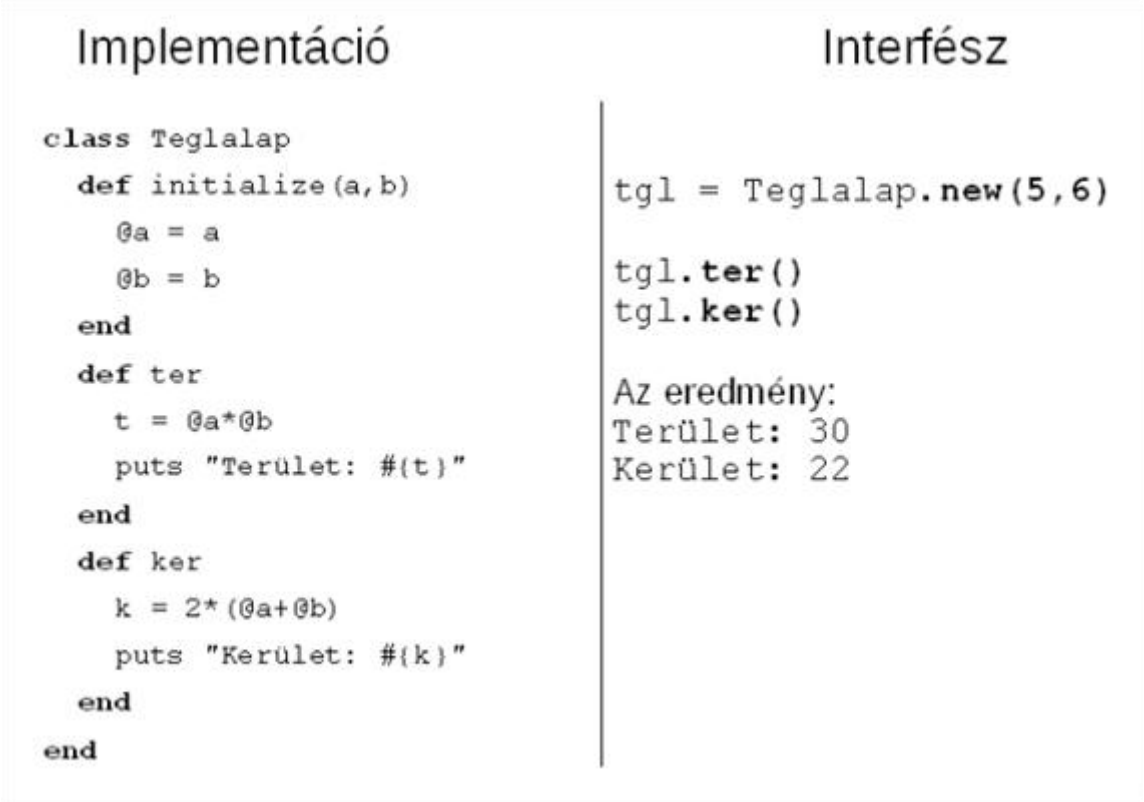
A következő példákban az interfészek a `reverse` , a `max` és az `alszik` metódusok.

```
# A String osztály egy interfésze
"HELLO".reverse
# A Range osztály egy interfésze
sorozat1 = 1..10
sorozat1.max
#Az Ember osztály egy interfésze
valaki = Ember.new
valaki.alszik()
```

A fenti objektumoknál nem biztos, hogy ismerjük a metódus végrehajtásának működését. Illetve, az objektum használatához nincs szükségünk a metódus működésének ismeretére, a metódus algoritmusára. Az objektum elrejti a részleteket, amely a felhasználás szempontjából nem fontos.

A metódus algoritmusát gyakran implementációnak (*implementation*) hívják. Az implementáció azt határozza meg, hogy hogyan hajtsa végre az objektum az általa biztosított szolgáltatásokat. Az implementáció tehát a kód, az algoritmus.

A következő ábra a `Teglalap` osztály implementációját és interfészeit mutatja.



8. ábra Implementáció és interfész

Tehát az interfészeket - `new` , `ter` , `ker` - a `Teglalap` osztály biztosítja számunkra a `tgl` objektumon keresztül.

A hozzáférés vezérlése

Egy metódus nem minden esetben interfésze egy objektumnak. Az, hogy más objektum eléri-e, az objektumot, azaz küldhet-e üzenetet az objektumnak, a hozzáférés vezérlés beállításaival szabályozható.

A Rubyban az alábbi beállításokat alkalmazhatjuk:

- A publikus (*public*) metódusok elérhetők más objektumok számára.
- A védett (*protected*) metódusokat csak az osztály és az alosztályok objektumai hívhatják meg.
- A privát (*private*) metódus csak az aktuális objektum számára elérhető.

A metódusok publikusak, ha másképpen nem jelöljük. A `def` kulcsszó előtt elhelyezett `protected` kulcszóval védett, a `private` kulcsszóval privát metódusok hozhatók létre.

A példányváltozók privátnak tekintendők, mert csak un. hozzáférési metódusokon keresztül érhetők el más objektumok számára.

Példányváltózó vs. attribútum

Szigorúan értelmezve különbséget tehetünk a példányváltozók és az attribútumok között.

A példányváltozó akkor válik attribútummá, ha hozzáférési metódusokon keresztül elérhető más objektumok számára. Egy hozzáférési metódus (*accessor method*) az attribútumot olvashatja vagy módosíthatja. A Ruby beépített mechanizmust biztosít a hozzáférési metódusok definiálására:

- `attr_reader` – az attribútum olvasására
- `attr_writer` – az attribútum módosítására

- `attr_accessor` – az attribútum olvasására vagy módosítására

A hozzáférési metódusok használatakor egy - egy interfészt kapunk, amely megegyezik a `@` jel nélküli attribútumnévvel. Az osztályban a hozzáférési metódusok definiálása a megfelelő kulcsszóval, majd az attribútumnév - `@` jel nélkül - szimbólumként történő megadásával történhet, például:

```
attr_reader :valtozo
```

vagy

```
attr_accessor :valtozo1, :valtozo2
```

Nézzük meg a hozzáférési metódusok létrehozását és használatát a Teglalap osztályban.

```
class Teglalap
  attr_reader :a, :b
  attr_writer :b
  # vagy lehetne @b esetén ugyanaz másképpen: attr accessor :b
  def initialize(a,b)
    @a = a
    @b = b
  end
end
#Használat
teglalap1 = Teglalap.new(5,6)
puts teglalap1.a           #@a olvasása
teglalap1.b = 8            #@b módosítása
puts teglalap1.b
```

Az eredmény:

```
5
8
```

Öröklődés

Az öröklődés egy olyan mechanizmus, amely lehetővé teszi egy osztály létrehozását egy már meglévő osztály alapján.

Az leszármaztatott osztályból létrehozott objektum használhatja az ős osztály metódusait.

Az ős osztály egyéb megnevezései: szuper osztály (*super class*), szülő osztály.

A leszármaztatott egyéb megnevezései: alosztály (*sub class*), gyerekosztály.

Az öröklődést a kódban a `<` jellel adjuk meg az alosztály definíciójában.

```
class Parent
  def say_hello
    puts "Hello from #{self}"
  end
end
# Alosztály
class Child < Parent    #a szuper osztály < jellel megadva
end
#Használat
p = Parent.new
p.say_hello
c = Child.new
```

```
c.say_hello
```

Az eredmény:

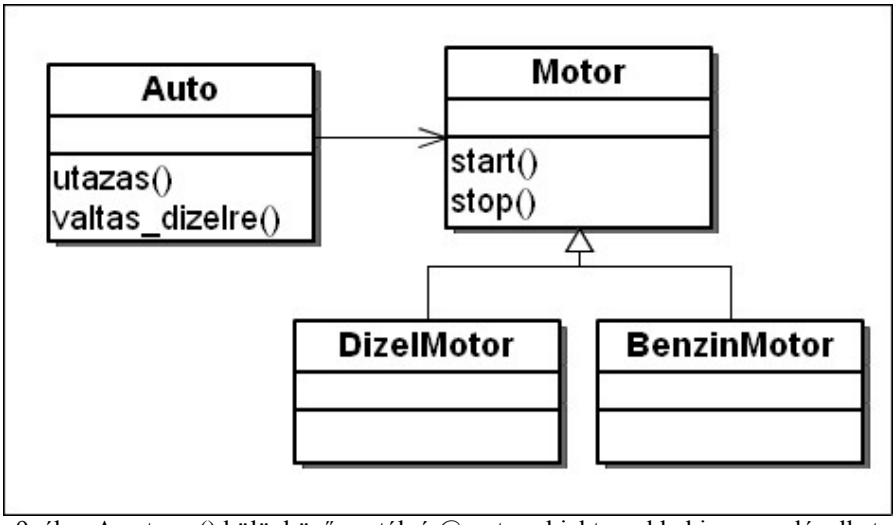
```
Hello from #<Parent:0x0a40c4>
Hello from #<Child:0x0a3d68>
```

A példában az öröklődés eredményeként a `Child` osztályból létrehozott `c` objektum is rendelkezik a `say_hello` metódussal. Ezt a `self` változóval mutattuk be. A `self` változó egy speciális változó amely mindig az aktuális objektumra hivatkozik.

Többalakúság

A többalakúság azt jelenti, hogy egy üzenetre a különböző objektumok különbözőképpen válaszolhatnak.

Egy név számos különböző viselkedést fejezhet ki. Vegyük a következő példát (9. ábra), amelyben az Auto osztály a Motor osztályhoz kapcsolódik. A Motor osztály a DizelMotor és a BenzinMotor osztályok szuperosztálya. A többalakúságot a start és a stop metódusokon keresztül vizsgáljuk meg.



9. ábra Az utazas() különböző osztályú @motor objektumokkal is megvalósulhat

Az `Auto` osztály forráskódjában azt olvashatjuk, hogy egy `Auto` objektum létrehozásakor egy `BenzinMotor` osztályú, a `@motor` példányváltozóval elérhető objektum is létrejön. A `valtas_dizelre` metódussal lehetőségünk van arra, hogy létrehozzunk egy `DizelMotor` osztályú objektumot, amelyre ezt követően a `@motor` változó hivatkozik. Az utazas metódus az aktuális `@motor` objektum `start` és `stop` metódusait hívja meg.

```
class Auto
  def initialize
    @motor = BenzinMotor.new
  end
  def utazas
    @motor.start
    #...ide jöhet az utazas kodja...
    @motor.stop
  end
  def valtas_dizelre
    @motor = DizelMotor.new
  end
end
```

Használjuk az `Auto` osztályt az alábbiak szerint:

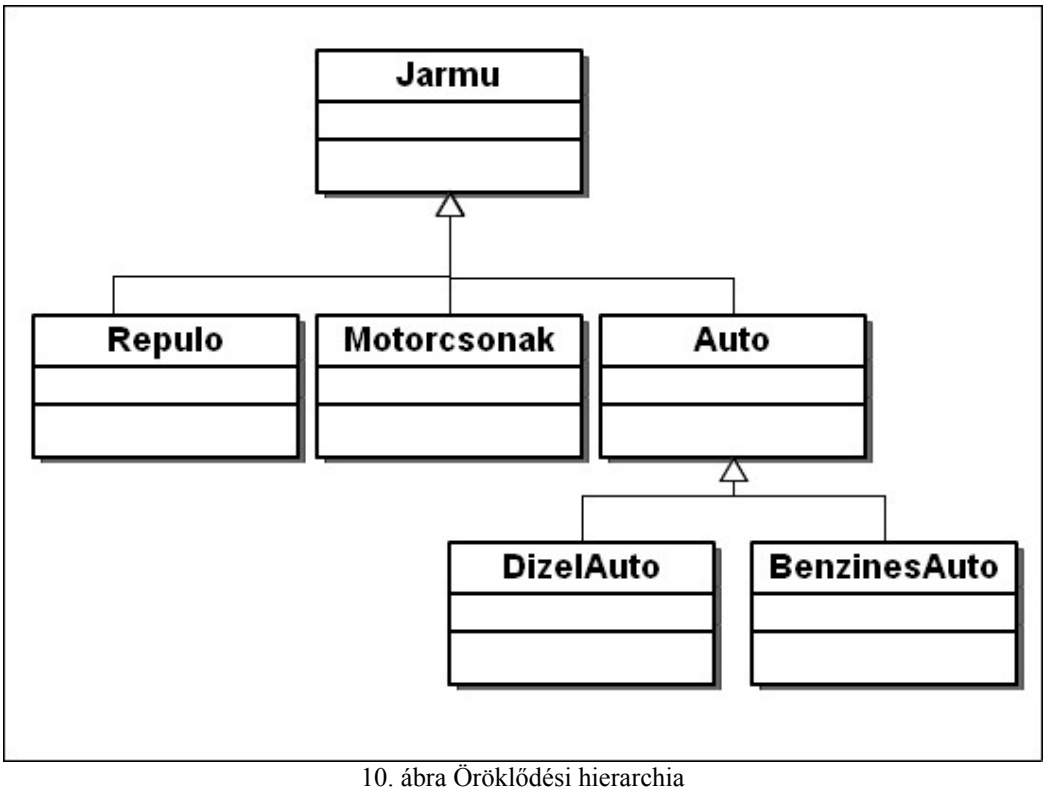
```
auto = Auto.new
auto.utazas
auto.valtas_dizelre
auto.utazas
```

Az `auto` objektum létrehozását követően az `utazas` metódusban `BenzinMotor` osztályú objektumot használunk. A `valtas_dizelre` metódus meghívása után az `utazas` metódus már egy `DizelMotor` osztályú objektumot kezel. Ugyanazon a néven (interfészen), a `start` és a `stop` metódusokkal, különböző objektumoknak küldünk üzenetet.

5.4. 4.5.4 Több osztály használata

Általánosítás-specializálás

Az öröklődés lényegében egy általánosítás-specializálás viszonyt fejez ki. Az 10. ábrán azt olvashatjuk, hogy a `Jarmu` egy specializált változata a `Repulo` , a `Motorcsonak` és az `Auto` . Az `Auto` speciális változatai a `DizelAuto` és a `BenzinesAuto` .

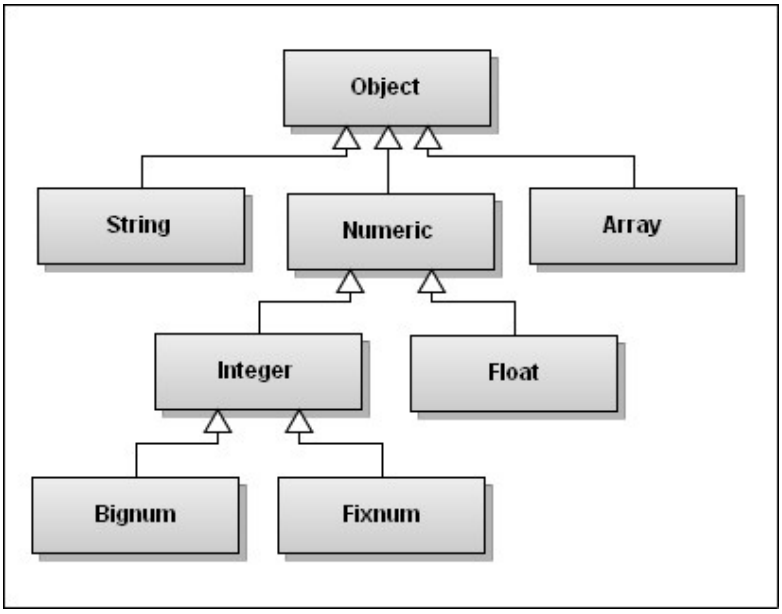


10. ábra Öröklődési hierarchia

Alulról olvasva a diagramot általánosítunk. A `BenzinesAuto` egy `Auto` , az `Auto` egy `Jarmu` . Angolul ezt *"Is-a"* kapcsolatnak nevezik (*Car is a Vehicle*).

Az öröklődés tehát hasznos, ha specializálunk, egy új viselkedést adunk meg. Illetve alkalmas a már meglévő kód újrafelhasználása, ugyanis a szuper osztály metódusai elérhetők az alosztályokból.

Az öröklődés előnyeivel már a Ruby első használatakor is éltünk, mert a Ruby alaposztályai egy öröklődési hierarchiába rendezettek, minden osztály az Object osztálytól származik. A következő ábra az osztályhierarchia egy részét ábrázolja.



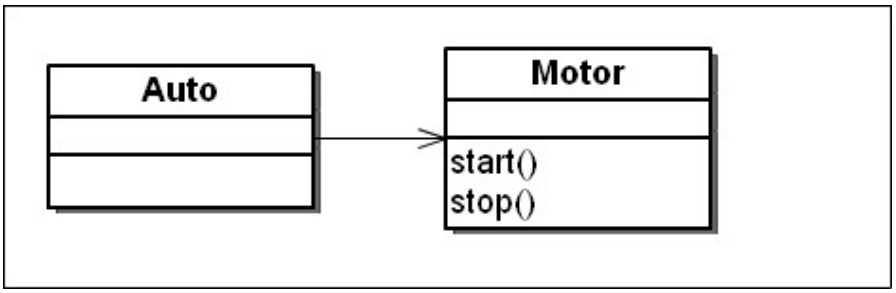
11. ábra A Ruby alaposztályainak egy része

A Rubyban egyszeres öröklődés lehetséges, azaz egy osztály csak egy közvetlen szuperosztállyal rendelkezhet. Más nyelvekben a többszörös öröklődés is megengedett, ekkor egy osztály közvetlenül több szuperosztálytól származhat.

Kompozíció

Az általánosítás-specializálás mellett létezik egy másik kapcsolat az osztályok között. Ez a kapcsolat azt fejezi ki, hogy egy objektum használ egy másik objektumot vagy egész-rész viszonyban van egy másik objektummal.

A szakirodalomban ezt a viszonyt asszociáció (*association*), kompozíció (*composition*) és aggregáció (*aggregation*) neveken említik. Bár ezek vizuális ábrázolásukban különböznek egymástól, a kódban nem. Általában az elméleti, objektum-orientált modellezéssel kapcsolatos szakirodalomban hosszan taglalják a különbséget a három eset között, a programozáshoz közelebb álló, gyakorlatias szakirodalom nagy része összefoglalóan kompozícióként hivatkozik erre a viszonyra.



12. ábra Kompozíció

A kompozíciót angolul gyakran *„Has-a”* kapcsolatként említik (*Car has an Engine*). Magyarul a következőket mondhatnánk:

- Az `Auto` nak van egy `Motor` ja.
- Az `Auto` osztály hivatkozik a `Motor` osztályra.
- Az `Auto` osztály objektuma tartalmazza a `Motor` osztály objektumát.

Ez a forráskódban gyakran úgy jelenik meg, hogy az egész-rész kapcsolat egész oldalán lévő osztály (`Auto`) példányváltozója a kapcsolat rész oldalán található osztály (`Motor`) objektumára hivatkozik.

```
class Auto
```

```
  def initialize
    @motor = Motor.new # a peldanyvaltozo egy Motor objektum
  end
  def utazas
    @motor.start
    #...ide jöhet az utazas kodja...
    @motor.stop
  end
end
```

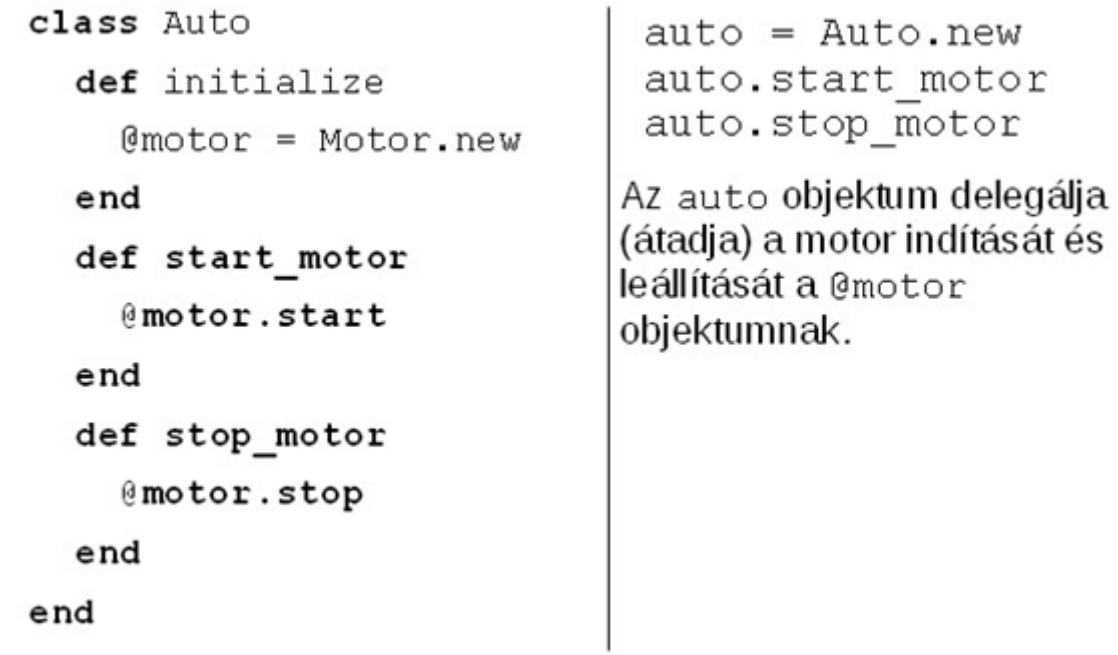
Delegálás

Ahogyan az öröklődés interfészt biztosít egy alosztályhoz, a delegálás is interfészt biztosít egy kompozícióval kapcsolódó osztályhoz.

A kompozíciónál ismertetett példánál azt mondtuk, hogy "Az `Auto` osztály objektuma tartalmazza a `Motor` osztály objektumát."

Ekkor a delegálás technika alkalmazása a következőket jelenti:

- a `Motor` osztály objektumának interfésze adható meg,
- az `Auto` osztály objektumának küldött üzenet továbbításra kerülhet `Motor` osztály objektumához.



13. ábra Delegálás

6. 4.6 Összefoglalás

A tananyagban a Ruby programozási nyelvről és az objektum-orientált programozás alapkoncepcióiról adtunk áttekintést. Terjedelmi korlátok miatt az áttekintés nem teljes.

Az érdeklődő olvasónak érdemes további ismereteket szereznie: a Ruby programozási nyelv osztályairól [1] , a grafikus felhasználói felületekről [2] , a Ruby szabadon elérhető - a közösség által közzétett - kiegészítő csomagjairól [3] vagy a szkriptelési lehetőségekről (pl. MS Office, Google Sketchup [4]).

[1] http://www.ruby-lang.org/en/documentation/
[2] Shoes: http://shoesrb.com/, WxRuby: http://wxruby.rubyforge.org/wiki/wiki.pl
[3] http://rubygems.org

Az objektum-orientált programozási stílus elsajátításához javasoljuk a nyílt forráskódú alkalmazások kódjainak böngészését és a szakirodalom tanulmányozását (tervezési minták, tervezési módszerek, szoftverfejlesztési módszerek).

A tananyag végére érve a következő ellenőrző kérdésekkel tesztelheti tudását:

1. Mi a különbség az alacsony szintű és a magas szintű programozási nyelvek között?
2. Mutassa be az programok végrehajtásának alapvető technikáit!
3. Melyek a jellemzői a dinamikusan és a statikusan típusos nyelveknek?
4. Mi a szintaktika?
5. A programok funkcionalitása milyen eszközökkel valósítható meg?
6. Mit tud a Ruby szintaktikájáról?
7. Milyen alapvető osztályokat ismer a Rubyban?
8. Milyen feltételes szerkezeteket biztosít a Ruby?
9. Az iteráció milyen szerkezetekkel valósítható meg a Rubyban?
10. Mi az osztály és az objektum definíciója?
11. Mit jelent az interfész és az implementáció?
12. Mi az öröklődés?
13. Mit jelent a többalakúság?
14. Milyen kapcsolatot jelent a kompozíció?
15. Mire használható a delegálás?

Irodalomjegyzék

Booch, G. et al.: *Object-Oriented Analysis and Design with Applications, Third Edition* Addison-Wesley 2009

Flanagan D. - Matsumoto Y.: *The Ruby Programming Language* O'Reilly Media, Inc. 2008

Pine, C.: *Learn to program, Second Edition* The Pragmatic Programmers, LLC. 2009

Talim, S.: *Study notes on Ruby* 2008 http://rubylearning.com

Olsen, R.: *Design Patterns in Ruby* Addison-Wesley 2008 http://designpatternsinruby.com/

[4] http://code.google.com/intl/hu/apis/sketchup/