Programozási nyelvek Java

Kitlei Róbert

Programozási nyelvek és Fordítóprogramok tanszék
ELTE Informatikai Kar

Mi mindent használunk programozáshoz?

- programozási nyelv
- ***könyvtár***ak (más terminológiával csomagok): könnyen újrafelhasználható programrészek
  - ◇ alapkönyvtárak: a programozási nyelvvel együtt települnek
  - ◇ "third party" könyvtárak
  - ◇ csomagkezelő (package manager): központilag elérhető csomagok keresése, (igény szerinti automatizált) telepítése
    - ▸ Java: Maven, Ant+Ivy, Gradle, jpm4j
- futtató környezet
  - ◇ lehet a fizikai gép
  - ◇ lehet virtuális gép
    - ▸ a virtuális gépet vezérelheti pl. böngésző vagy HTTP szerver, ekkor a programunk ***applet*** vagy ***szervlet***
- kiegészítő eszközök

Kiegészítő eszközök

- build rendszer: a programrendszer fordítását, tesztelését vezérli
- statikus ellenőrző eszköz (lint): hibákat keres a forráskódban
  - ◇ a fordítóprogram kiterjesztésének tekinthető
- debugger: futási idejű hibakereső
- profiler: futási időben méri, hol lassú/fogyaszt sok memóriát a program
- projekttámogató eszközök
  - ◇ verziókezelő (VCS, version control system)
  - ◇ project management: feladatok, határidők stb. kezelésére
  - ◇ hibakövető (bug tracker)
  - ◇ continuous integration: minden kód hamar az éles rendszerbe kerül
- fejlesztőkörnyezet (IDE): a fentiek közül sok eszköz beépített
  - ◇ kód módosításának támogatása (autokiegészítés, code snippets stb.)
  - ◇ kód áttekintésének támogatása (diagramok, navigáció stb.)

Programozási nyelvi szabványok

- a programozási nyelvek szabályait ***szabványok*** (standard) írják le
  - ◇ több száz/több ezer oldalas technikai dokumentumok
  - ◇ néha nyelv+évszám kombinációval szokás rájuk (informálisan) hivatkozni: ***Ada 2012***, ***C11***, ***C++11***, ***Haskell 98***
  - ◇ néha verziószámmal: ***Java 8***
    - ▸ a Java számozása furcsa: szokás Java 1.8-nak is nevezni
- a legtöbb megkötés teljesen egzakt
  - ◇ jobb platformfüggetlenség
    - ▸ kevesebb megkötéssel hordozhatóbb is lehet egy program...
      - pl. kevesebb memóriájú eszközön is futhat
    - ▸ ... viszont kompatibilitási problémák is felmerülhetnek
      - pl. az egyik gépen túlcsordul egy számláló, a másik gépen nem
  - ◇ a hatékonyság rovására mehet

Szabványok tartalma

- ***lexika***: a forráskód legkisebb alkotóelemeinek szabályait adja meg
  - ◇ pl. szavak, számkonstansok, operátorok alakját
- ***szintaxis*** (syntax): az alkotóelemekből épített struktúra szabályait adja meg
  - ◇ egy adott szöveg érvényes-e forráskódként?
  - ◇ milyen a szerkezete, pl. egy bonyolult kifejezés milyen zárójelezéssel értendő?
  - ◇ érvényes forráskódból ***szintaxisfa*** (syntax tree) építhető
- ***szemantika*** (semantics)
  - ◇ "értelmes-e" a forráskód?
    - ▸ pl. tartalmaz-e hivatkozást definiálatlan műveletre?
  - ◇ "mit jelent" a forráskód: hogyan kell végrehajtani? mi az eredménye?
  - ◇ lehet-e több különböző módon is lefordítani, van-e optimalizálási lehetőség?

Fordítás és futtatás

- fordítással (AOT, ahead of time compilation)
  1. ***fordítási idő*** (compile time): a ***fordítóprogram*** (compiler) lefordítja a ***forráskód***ot (source code) ***tárgykód***dá (object code)
     - ◇ ha a kód hibás, ***fordítási hibá***t ad (compile error)
     - ◇ a tárgykód ***gépi kód***ú (machine code) részeket tartalmaz
     - ◇ több tárgykódból a ***szerkesztő*** (linker) készíti el a végső programot
  2. ***futási idő*** (runtime): a gép közvetlenül hajtja végre a gépi kódot
- ***interpreter***rel (értelmező): egy külön program utasításonként fordítja le és hajtja végre a forráskódot
  - ◇ ***scriptnyelv***: olyan nyelv, amelyet jellemzően interpreter hajt végre
  - ◇ egy jellemző megközelítés: ***REPL*** (read–eval–print loop)
    1. read: a programozó megírja/átírja a kódot, ezt az interpreter beolvassa
    2. eval(uate): az interpreter végrehajtja (kiértékeli) a kódot
    3. print: az interpreter kiírja a végeredményt

Fordítás és futtatás

- JIT (just-in-time compilation)
  - ◇ a futtató rendszer újrafordíthatja a program részeit futás közben
  - ◇ ***Java virtuális gép*** (Java virtual machine, ***JVM***): legtöbbször ezen futtatjuk a Java programokat
    - ▸ ***bájtkód*** (bytecode): a Java virtuális gép gépi kódja

```
javac X.java
java X
java X param1 param2 param3
```

- a Java fordító egyszerre egy forrásfájlt fordít le
  - ◇ a benne szereplő osztályok közül legfeljebb egy lehet ***publikus***
    - ▸ ha van ilyen, annak a nevének (kis- és nagybetűre pontosan) meg kell egyeznie a forrásfájl nevével
  - ◇ a forrásfájl kiterjesztése ***.java***
  - ◇ a fordító a forrásfájlban található minden egyes osztályhoz létrehoz egy ***.class*** kiterjesztésű fájlt

Paradigmák

- ***paradigma***: a program elkészítését meghatározó elvek összessége
- ***imperatív*** programozás
  - ◇ a program ***állapot***át (state) és annak megváltoztatását helyezi előtérbe
  - ◇ a program ***utasítás***okat (statement) hajt végre ***sorban***
    - ▸ az ***értékadó utasítás***ok (assignment) változtatják meg az állapotot
    - ▸ a program ***vezérlése*** (control flow) az első utasításnál indul, és mindig pontosan meghatározott, merre megy tovább (melyik utasítást hajtja végre következőnek)
- ***strukturált*** programozás
  - ◇ olyan ***imperatív*** programozás, ahol a felhasználható ***vezérlési szerkezet***ek (control structure) korlátozva vannak
  - ◇ ***Böhm-Jacopini tétel*** (1966): minden imperatív program megalkotható ***szekvencia*** (sequence) ***elágazás*** (conditional), ***ciklus*** (loop) vezérlési szerkezetekkel
  - ◇ a ***goto*** utasítás (vezérlésátadás tetszőleges helyre) tiltott
- ***procedurális*** programozás: alprogramokat használunk, ezek kódja strukturált

Paradigmák

- ***deklaratív*** programozás
  - ◇ a vezérlés helyett a kiszámítandó érték szerkezetére koncentrál ("hogyan" helyett "mit")
  - ◇ pl. ***adatbáziskezelő*** nyelvek (SQL)
- ***logikai*** programozás
  - ◇ olyan ***deklaratív*** programozás, ahol tények és következtetési szabályok felhasználásával adódik a kívánt eredmény
  - ◇ legismertebb nyelv: ***Prolog*** (Colmerauer, 1972)

```
% tények
anyja(ed, eva). apja(sue, jim). apja(jim, ed). apja(eva, tom).

% következtetési szabályok
nagyapja(X,Y) :- apja(X,A),  apja(A,Y).
nagyapja(X,Y) :- anyja(X,A), apja(A,Y).

% lekérdezések
?- nagyapja(ed, X).
?- nagyapja(X, ed).
```

Paradigmák

- ***funkcionális*** programozás: olyan ***deklaratív*** programozás, amelyben az eredményt függvények (matematikai) kompozíciója adja
  - ◇ az imperatív nyelvekkel egykorú
    - modell: Turing-gép (Turing, 1936) $\leftrightarrow$ λ-kalkulus (Church, 1936)
    - nyelv: FORTRAN (Backus, 1957) $\leftrightarrow$ LISP (McCarthy, 1958)

```
-- prímszámok (Haskell)
prs = 2:filter (\n -> all (\p -> n `rem` p /= 0)
                          (takeWhile (<n`div`2) prs)) [3,5..]
```

- ***eseményvezérelt*** programozás
  - ◇ a program eseményekre (pl. egérkattintás) reagálva hajt végre műveleteket
  - ◇ a műveletek az említettek közül bármelyik paradigmájúak lehetnek
- ***objektum-orientált*** programozás
  - ◇ erről még sok szó lesz a továbbiakban
- ezek mellett sok egyéb paradigma is létezik

A Java paradigmái

- a Java főbb paradigmái
  - ◇ imperatív; strukturált; procedurális
  - ◇ eseményvezérelt
  - ◇ objektum-orientált
- a strukturált eszközök mellett a goto néhány korlátozott változata elérhető: `break`, `continue`, `return`
- funkcionális nyelvi elemek is elérhetőek
  - ◇ sok más imperatív nyelv is kezd átvenni funkcionális elemeket

Adatok a programban

- ***érték*** (value): futás közben megjelenő adat (data)
- az értékeket ***típus***okba soroljuk (type)
  - ◇ ***szigorúan típusos nyelv*** (strongly typed): a különböző típusok között éles a határvonal
  - ◇ ***gyengén típusos nyelv*** (weakly typed): a T típusba tartozó e érték könnyen T_2 típusba tartozónak tűnhet
    - ▸ a programnyelvek e két véglet között helyezkednek el
- ***literál*** (literal): primitív érték megjelenése a forráskódban
  - ◇ pl. `1`, `-52.2623`, `true`, `"abcd\txyz"`
  - ◇ a literálok formázásának szabályai rögzítettek
    - ▸ pl. az `1.` vagy a `.6` érvényes lebegőpontos literál-e?
    - ▸ pl. sorvége jel lehet-e szövegliterál közepén?
  - ◇ a literálok által jelölt érték típusa rögzített

Adatok a programban

- az adatok lehetnek primitív vagy összetett adatok
  - ◇ a Java primitív típusai
    - ▸ karakterek: ***char***
      - technikailag egész szám (a karakter kódja)
    - ▸ egész típusok: `byte`, `short`, ***int***, `long`
    - ▸ logikai típus: ***boolean***
    - ▸ lebegőpontos típusok: `float`, ***double***
  - ◇ Javában minden nem-primitív típus objektum

Adatok ábrázolása

- úgy tűnhet, hogy `int`$\equiv \mathbb{Z}$ és `double`$\equiv \mathbb{Q}$, esetleg `double`$\equiv \mathbb{R}$
  - ◇ … de véges tárterületen csak véges számokat tudunk ábrázolni
    - ▸ ***túlcsordulás*** (overflow): `(byte)(127+1) == (byte)(-128)`
  - ◇ jellemző reprezentációk
    - ▸ (előjeles) egész típusok: ***kettes komplemens*** (two's complement)
    - ▸ lebegőpontos típusok: ***IEEE 754*** szabvány
      - a `double` minden olyan értéket ábrázolni tud, amit az `int`
- a lebegőpontos kerekítések pontatlanságokat okoznak, ezt hosszú számításoknál figyelembe kell venni
  - ◇ ahol ez nem engedhető meg, pl. a pénzügyi szektorban, ***fixpontos*** számábrázolást használnak
- vannak tetszőleges méretű számokat ábrázoló típusok is, pl. `BigInteger`
  - ◇ kevésbé hatékony, de csak ritkán van rá szükség

Műveletek mint értékek

- vannak nyelvek, amelyekben a műveletek értékek ("first-class citizen")
  - ◇ ***closure*** (lezárt, zárvány, zárlat): alprogram, amely a környezetére hivatkozhat (a példában `F` a `C` változóra hivatkozik)

```
f(Par1, ParFun) ->  % Erlang nyelven írt függvény
  C = 36,
  F = fun(X) -> C*X end, % a fun...end az értékként adott fv.
  F(ParFun(Par1)). % az F által kötött fv. felhasználása
```

- Javában maguk a műveletek nem értékek: nem adható értékül változónak, nem lehet művelet visszatérési értéke vagy paramétere
  - ◇ … de Java 8-tól kezdve "majdnem" azok
    - ▸ ***szintaktikus cukor*** (syntactic sugar): egy bonyolultabb konstrukció egyszerűbb alakban is megjelenhet a forráskódban

```
int f(int par1, IntUnaryOperator parFun) {
                // ---------------- int->int típusú függvény
    int c = 36;
    IntUnaryOperator op = x -> c*x;
    return op.applyAsInt(parFun.applyAsInt(par1));   }
```

Program szerkezete

- a Java nyelv szerkezete vázlatosan a következő (más nyelveké többé-kevésbé hasonló):
  - ◇ csomag → osztály → metódus → utasítás → kifejezés
  - ◇ csomag → osztály → adattag → inicializáló kifejezés

```
package hu.valami.csom;
class Osztály {  // teljes neve: hu.valami.csom.Osztály
    int adattag;
    void ezEgyMetódus() {
        adattag = 28 * m2() + 456; // utasítás
                // --   ----    ---
                // ---------       az aláhúzottak
                // ---------------- mind
     // ------------------------- (rész)kifejezések
    }
    int m2() { return 123; }
}
```

A program szerkezete

- a program ***csomag***okból (package) áll
  - ◇ a csomag nevét pontok tagolják
  - ◇ ha egy weboldalhoz kötődő csomagról van szó, a csomag nevének kezdete jellemzően az URL fordítva (a példában `valami.hu`)
  - ◇ a könyvtárszerkezetnek illeszkednie kell a csomagszerkezethez
    - ▸ pl. az `abc.def.gh.ij` csomaghoz tartozó fájloknak az `abc/def/gh/ij` könyvtárba kell kerülniük
  - ◇ a ***package*** direktíva adja meg a csomag nevét
    - ▸ ez csak a `.java` fájl legelső eleme lehet
    - ▸ pl. `package abc.def.gh.ij;`
    - ▸ ha nincsen csomag feltüntetve a fájlban, a ***névtelen csomag***ba (default package) tartozik, és a forrásfájlok gyökérkönyvtárába kell kerülnie

A program szerkezete

- a csomagok ***osztály***okat tartalmaznak
  - ◇ az osztályok írják le az objektumok szerkezetét
    - ***adattag*** (data field) vagy ***mező*** (field)
  - ◇ és az objektumon végezhető ***művelet***eket
    - ***alprogram*** (subprogram): a művelet meghívása során lefutó, paraméterezhető kódrészlet
    - osztály műveletét ***metódus***nak (method) vagy ***tagfüggvénynek*** (member function) nevezzük
    - a ***művelet meghívása*** (method invocation) általában a lokális gépről történik, és csak egy alprogramhívást foglal magában
      - más megközelítéssel: a művelet meghívása során egy ***üzenet***et küldünk az objektumnak, amellyel utasítjuk a művelet végrehajtására, majd a visszatérési értéket szintén egy üzenetben kapjuk tőle vissza
    - a hívás történhet távoli gépről is (remote method invocation), hálózati kommunikációra támaszkodva
      - távoli hívás esetén az üzenetek explicit meg is jelennek

A program szerkezete

- a műveletek kódja ***utasítás***okból (statement) áll
  - ◇ az utasítások ***végrehajtás***ával (execution) működik a program
  - ◇ az utasítások lehetnek egyszerűek (pl. deklaráció, return, break, continue) vagy összetettek (elágazások, ciklusok)
  - ◇ minden kifejezés pontosvesszővel ellátva utasítás Javában
- a ***kifejezés***ek (expression) feladata jellemzően egy érték kiszámítása
  - ◇ amikor a vezérlés odajut, a kifejezéseket ***kiértékeljük*** (evaluate), azaz meghatározzuk a kifejezések értékét
  - ◇ a kifejezéseknek lehetnek ***részkifejezés***ei (subexpression), ezeket általában ki kell értékelni ahhoz, hogy a teljes kifejezés értékét meg tudjuk határozni
- sok nyelvben csak kifejezések vannak, utasítások nincsenek

Szavak

- ***kulcsszó*** (keyword): a nyelvben speciális célra használt szó
  - ◇ a jelentése a szabványban rögzített, nem változtatható meg
  - ◇ pl. Javában `class`, `new`, `final`, `static` stb.
  - ◇ másfajta szavaknál "erősebb"

```
int class = 3;  // tilos, a "class" kulcsszó
```

- ***azonosító*** (identifier): egy nevet vezet be a programban, és hozzárendeli azt egy programelemhez (csomag, osztály, metódus, adattag, változó)
  - ◇ innentől, ha a nevet leírjuk, arra a programelemre hivatkozunk vele

A program tagolása

- ***fehér szóköz*** (whitespace): szóköz, sorvége, tabulátor (és még néhány kevésbé fontos) karakter a forráskódban
  - ◇ Java programokban csak elválasztó szerepük van
  - ◇ akár egyetlen sorba is leírhatjuk a teljes programunkat
- ***tördelés*/*behúzás*/*indentálás*** (indentation): sor elejére elhelyezett fehér szóközök
  - ◇ osztály, metódus, összetett utasítás megnöveli a behúzást
    - ▸ jellemzően 4 (vagy 2, 3, 8) szóköznyivel nő a behúzás szintenként
  - ◇ jobban kiadja a program szerkezetét
  - ◇ a behúzott kódrészletet néha nyitó/csukó jelek (pl. *{*, *}*) veszik körül
    - ▸ ezek elhelyezésére több lehetőség van (előző sor végére, új sorba, új sorba külön behúzással)
- a szerkezeti egységeket (osztályok, függvények) érdemes legalább egy üres sorral is elválasztani
- ***behúzás-alapú*** (indentation based) nyelv: a sorok behúzásának mértéke befolyásolhatja a kód jelentését
  - ◇ pl. ilyen nyelvek: Python, Haskell

Kódolási konvenciók

- ***kódolási konvenció*** (coding convention): a nyelv követelményeinél szigorúbb szabályrendszert követünk
  - ◇ cél: jobb minőségű programkód
- mire vonatkoznak?
  - ◇ nevek: osztálynevek (`CamelCase`), változó- és metódusnevek (`camelCase`), `final` változók nevei (`ALL_CAPS`)
  - ◇ fehér szóközök elhelyezése, pl. indentálás
  - ◇ ***metrika*** (metrics): a kód minőségét jelző mérőszám
    - ▸ metódusok hossza: jó, ha viszonylag rövidek
      - egy-két képernyőnyinél nem jó, ha hosszabb
    - ▸ ***beágyazás mélysége*** (nesting): az egymásba skatulyázódó vezérlési szerkezetek (`if`, `for` stb.) mélysége
      - 2-3 fölött átláthatatlanná válik a program
    - ▸ segédprogramokkal mérhető, ezek rögtön jelzik, ha elromlik a kód
    - ▸ kiküszöbölés: a túl hosszú/mély kódrészek új metódusba kiemelése
      - ***refaktorálás***: a kód átszervezése (eszköztámogatással)

Kifejezések: operátorok

- ***aritás*** (arity): hány operandusa van
  - ◇ ***bináris*** (binary): a legtöbb operátor 2 aritású
  - ◇ ***unáris*** (unary): `!x`, `+x`, `-x`, `~x`, `++x`, `x++`
  - ◇ ***ternáris*** (ternary): `x?a:b`
- ***fixitás*** (fixity): az operátor helye
  - ◇ ***prefix*** (`++x`), ***posztfix*** (`x--`), ***infix*** (`x+y`), ***mixfix*** (`b?x:y`)
- ***precedencia*** (precedence): kif1 $\oplus$ kif2 $\odot$ kif3 hogyan értelmezendő?
  - ◇ kif1 $\oplus$ (kif2 $\odot$ kif3): $\odot$ precedenciája magasabb
  - ◇ (kif1 $\oplus$ kif2) $\odot$ kif3: $\oplus$ precedenciája magasabb
- ***asszociativitás*** (associativity): kif1 $\oplus$ kif2 $\oplus$ kif3 hogyan értelmezendő?
  - ◇ (kif1 $\oplus$ kif2) $\oplus$ kif3: $\oplus$ ***balasszociatív*** (left associative)
    - ▸ a legtöbb operátor balasszociatív (“balra köt”)
  - ◇ kif1 $\oplus$ (kif2 $\oplus$ kif3): $\oplus$ ***jobbasszociatív*** (right associative)
    - ▸ az értékadás a legtöbb nyelvben jobbasszociatív (“jobbra köt”)
    - ▸ mj.: az értékadás néhány nyelvben kifejezés, néhányban utasítás

Kifejezések: mellékhatásosság/tisztaság

- a kifejezések és a műveletek sokban hasonlítanak
  - ◇ mindkettőnek van neve (a kifejezés esetén az operátor)
  - ◇ mindkettőnek lehetnek paraméterei (kifejezés: operandusok)
  - ◇ vannak programozási nyelvek, ahol csak alig vannak megkülönböztetve
- kétfajta jellegű feladatuk szokott lenni
  - ◇ ***tiszta*** (pure): számítást végez
    - ▸ bemenő értékekből (operandus/paraméter) kimenő értéket állít elő
    - ▸ ha többször ugyanazt a bemenetet kapják, garantáltan ugyanazt a kimenetet adják (a matematikai függvényekhez hasonlóan)
    - ▸ a számítások során előállíthat átmeneti értékeket (pl. a részkifejezések értékeit), de ezeket a számítás végén túl nem tárolja
  - ◇ ***mellékhatás***os (side effect): tevékenységet végez
    - ▸ megváltoztatja a program ***állapot***át (state)
    - ▸ ... vagy kommunikációt folytat a program környezetével
- ***eljárás*** (procedure/subroutine): mellékhatásos művelet
- ***függvény*** (function): tiszta művelet
  - ◇ gyakran bármilyen műveletet függvénynek szokás nevezni

Kifejezések: mellékhatásosság/tisztaság

- a tiszta kódot könnyebb kezelni, mint a mellékhatásosat
  - ezért érdemes a legtöbb alprogramot tisztán tartani
  - sokkal könnyebben tesztelhető
- érdemes jól kiemelni a forráskódban, ha egy kódrészlet mellékhatásos
  - egy kifejezésben lehetőleg ne legyen egynél több mellékhatás
  - eljárásban minél kevesebb tiszta rész legyen, a számításokat külön függvények végezzék
- gyakori konkrét mellékhatások
  - globális/osztályszintű változó használata (értékének írása/olvasása)
  - a metódus paraméterének írása, ha az a metóduson kívül látható
  - I/O műveletek (sztenderd kimenet, fájlok stb. írása/olvasása)
  - hálózati kommunikáció
  - másik mellékhatásos művelet meghívása

Kifejezések: értékadás, ++

- prefix ++ (és --) esetén a kifejezés értéke a változó új értéke
- posztfix ++ esetén a növelés előtti érték

```
                  // i   a   b
int i  = 4;       // 4         // deklaráció kezdeti értékadással
    i += 2;       // 6
    i -= 3;       // 3
int a  = ++i;     // 4   4
int b  = i++;     // 5       4
```

- az értékadás mellékhatásos: beállítja a változó értékét
  - ◇ az értékadás kifejezés: van értéke (megegyezik a részkifejezése eredményével)
  - ◇ a legtöbb operátortól eltérően jobbra köt

```
int i;
int j;
i = j = 3 * f() + g();
i = (j = ((3 * f()) + g()));  // teljesen zárójelezve
```

Kifejezések kiértékelésének sorrendje

- Java: balról jobbra értékeljük ki egy kifejezés részkifejezéseit
- példa: `i+++ t[i]`
  - ◇ teljesen zárójelezett alakja: `(i++) + t[i]`
  - ◇ tfh `i` értéke kezdetben `6` és `t[7]` értéke `15`

  1. `i` (az `i++` kifejezésben), értéke: `6`
  2. `i++`, értéke: `6` (mellékhatás: `i` új értéke: `7`)
  3. `i` (a `t[i]` kifejezésben), értéke: `7`
  4. `t[i]`, értéke: `t[7]` értéke, `15`
  5. teljes kifejezés értéke: `6 + 15`, azaz `21`

- példa: `t[i] = i = 0`
  - ◇ teljesen zárójelezett alakja: `t[i] = (i = 0)`
  - ◇ tfh `i` értéke kezdetben `1`

  1. `i` a `t[i]` kifejezéshez, értéke: `1`
  2. `0`, értéke: `0`
  3. `i = 0`, értéke: `0` (mellékhatás: `i` új értéke: `0`)
  4. teljes kifejezés értéke: `0` (mellékhatás: `t[1]` új értéke: `0`)

  - ◇ a kétfajta mellékhatást nehéz követni (nem `t[0]` kap új értéket)

Kifejezések kiértékelésének sorrendje

- a részkifejezések kiértékelésének sorrendje különösen fontos, ha mellékhatásos kifejezéseink vannak
  - ◇ más nyelvekben a sorrendet bizonyos esetekben nem rögzíti a szabvány
    - ▸ különböző eredményt adó kód fordulhat ugyanabból a forrásból
    - ▸ tehát egyes forráskódok jelentése *szabvány szerint definiálatlan*

```
i = i++ + 1;
a = i++ + ++i;
b = f() + g();    // f és g mellékhatásosak
```

- Javában sem célszerű ilyen kódokat írni
  - ◇ több nyelvet használva nehéz követni, mi történik
  - ◇ nehezen észrevehetőek és követhetőek a mellékhatások
    - ▸ ugyanazon változó írása és használata (az új értéket olvassuk-e?)
    - ▸ több ütköző mellékhatás (milyen sorrendben hajtódnak végre?)
  - ◇ megoldás: több utasításra bontás

Kifejezések: lustaság/mohóság

- a tiszta kifejezésektől azt várjuk, hogy matematikai módon értékeket számítsanak ki bemenő paraméterekből
  - ◇ ... de dobhatnak ***kivétel***t (exception)
  - ◇ ... és kerülhetnek ***végtelen ciklus***ba (infinite loop)
    - ▸ ennek jelölése $\bot$ (bottom) vagy ∞
- az összetett kifejezéseket is befolyásolja a részkifejezéseiken keresztül
- ***lusta kiértékelés*** (lazy evaluation): a kifejezéseket csak akkor értékeljük ki, ha/amikor az értékükre szükség van
  - ◇ ***rövidzáras és*** (short-circuit): `false` && _ $\Rightarrow$ `false`, még akkor is, ha _ kiértékelése $\bot$-t adna, vagy kivételt váltana ki
    - ▸ trükkös dolgokra is alkalmas lehet (ritkán)

```
if (x != 0 && 100 / x < 5) { ... }
```

- ***mohó kiértékelés*** (eager/strict evaluation): mindig kiértékeli mindkét részkifejezését
  - ◇ `false` & $X \Rightarrow X$ ($\bot$, kivétel esetén egyaránt)
- az & operátor számokra is alkalmazható bináris és-ként: 13&24 $\Rightarrow$ 8

Kifejezések típusa

- ***statikusan típusos nyelv*** (statically typed): minden változóhoz és kifejezéshez fordítási időben típus van rendelve
  - ◇ már fordítási időben kiderülhet, ha bizonyos értékeket helytelenül használunk
  - ◇ ***manifest typing***: a programelemek típusát explicit fel kell tüntetni
  - ◇ ***típuskikövetkeztetés*** (type inference): a fordítóprogram deríti fel a változók/kifejezések/műveletek típusait
    - ▸ a legtöbb funkcionális nyelv támogatja
      - szinte soha nem kötelező a típusokat kiírni bennük
    - ▸ egyre többször jelenik meg imperatív nyelvekben is
- ***dinamikusan típusos nyelv*** (dynamically typed): csak a változókhoz/kifejezésekhez futási időben kapcsolódó értékek típusosak
  - ◇ egy változó teljesen különböző típusú értékeket is felvehet
  - ◇ fordítási időben nincsen védelem
    - ▸ sok ilyen nyelvhez elérhető statikus típusozó eszköz
  - ◇ valamivel gyorsabb fejlesztést tehet lehetővé
- ez a fogalompár más, mint az erősen/gyengén típusos nyelv!

Utasítások: for

```
for (int i = 0; i < 10; ++i) {
    System.out.printf("i értéke %d%n", i);
}
```

- a ciklusváltozó lokális a ciklusra
- meg lehet változtatni a ciklusváltozó értékét a ciklusmagon belül, de nagyon rossz ötlet

```
for (int i = 0; i < 10; i++) {
    System.out.printf("i értéke %d%n", i);
}
```

- hatékonysága ugyanaz, mint a fentinek, a fordítóprogram optimalizálja
  - ◇ felismeri, hogy `++i` illetve `i++` eredményét nem használjuk

Utasítások: for és while

- a for és a while ciklusok egymásra átírhatóak

```
for (kezdeti értékadás; feltétel; léptetés)  ciklusmag
 ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓
kezdeti értékadás;
while (feltétel) { ciklusmag; léptetés; }
```

```
while (belépési feltétel)     ciklusmag
 ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓
for (; belépési feltétel; )   ciklusmag
```

- végtelen ciklus

```
while (true) ciklusmag
for (;;)     ciklusmag

while (1)    ciklusmag   // hack kevésbé erősen
                         // típusos nyelveken
```

Utasítások: for és while

- mikor melyiket érdemes használni? (a jobb olvashatóság szempontjából)
  - ◇ `for`: leszámláló ciklus
    - előre tudjuk, hogy milyen intervallumot járunk be vele
    - adatszerkezet (tömb, fa, lista stb.) elemeinek érintésére
  - ◇ `while`: feltételes ciklus
    - előre nem ismert a lépések (maximális) száma
    - pl. a felhasználó adatokat visz be, de nem tudjuk előre, mennyit
  - ◇ `do..while`: utótesztelős ciklus
    - nehezebben olvasható a másik kettőnél

Utasítások: foreach

- bejárható adatszerkezet (gyűjtemény) elemeinek végigjárása

```
for (String arg: args) {
    System.out.println(arg);
}
```

- folyam
  - ◇ technikailag nem utasítás

```
Stream<String> argStream = Arrays.stream(args);

argStream.forEach((String arg) -> System.out.println(arg));
argStream.forEach(arg -> System.out.println(arg));
argStream.forEach(System.out::println);
```

- a folyamok mindenféle egyébre is képesek

```
int sumOfWeights = widgets.parallelStream()
                          .filter(w -> w.getColor() == RED)
                          .mapToInt(w -> w.getWeight())
                          .sum();
```

Utasítások: break és continue

- ***break***: ciklus befejezése, a program futása a ciklus vége után folytatódik
- ***continue***: a ciklusmag további része nem fut le, a ciklus következő iterációját kezdi meg
- általában a legbelső ciklusra hatnak
- a break/continue képesek külső ciklusra is hatni, ehhez címke kell
  - ◇ a címkére csak ilyen módon lehet ráugrani (általános goto nincsen)

```
külső:
for (.....) {  // <-- (3) ide adja a vezérlést
    for (.....) { // <-- (4) ide adja a vezérlést
        if (.....)   break külső;    // (1)
        if (.....)   break;          // (2)
        if (.....)   continue külső; // (3)
        if (.....)   continue;       // (4)
        // ide akkor jut a vezérlés, ha (1)-(4) nem hatott
    }
    // <-- (2) ide adja a vezérlést
}
// <-- (1) ide adja a vezérlést
```

Utasítások: blokk

- több utasítás egybefoglalása
- a benne deklarált változók használatának korlátozása
- legjellemzőbb használata: elágazások ágaként és ciklusok magjaként
  - ◇ a függvények törzse technikailag nem blokk utasítás (bár azt is kapcsos zárójel veszi körbe)
- változók láthatóságát is lehet vele korlátozni

```
{
    int tízSzámÖsszege = 0;
    for (int i = 0; i < 10; ++i) {
        tízSzámÖsszege += számotBeolvas();
    }
    // tízSzámÖsszege itt használható
}

// tízSzámÖsszege itt már nem használható
```

Utasítások: if

```
if (f1) if (f2) prg1 else prg2
```

- ***csellengő else*** (dangling else) probléma: két if és egy else szerepel egymás után
  - ◇ kérdés: melyik if-hez tartozzon az else?
  - ◇ a programnyelvek egységesen azt választották, hogy a belső (közelebbi) if-hez tartozzon (az ábrán balra)
  - ◇ ha azt szeretnénk, hogy a külsőhöz tartozzon, blokkot kell alkalmaznunk (az ábrán jobbra)

```
if (f1)
    if (f2)
        prg1
    else
        prg2
```

```
if (f1) {
    if (f2)
        prg1
} else
    prg2
```

Utasítások: if, for hibalehetőségek

- az üres utasítás egyetlen pontosvesszőből áll

```
// hibás                  // ezt jelenti valójában
if (...); {               if (...)    ;  // akármi is a feltétel,
    kód                                  // nem történik semmi
}                         { kód  } // a feltételtől függetlenül lefut

// hibás                  // ezt jelenti valójában
for (...); {              for (...)   ;  // az üres utasítást
    kód                                  // hajtjuk végre sokszor
}                         { kód  } // a kód pontosan egyszer fut le
```

- ha lemarad a blokk, mást jelent a kód, könnyű átsiklani felette
  - ◇ ha kicsit is kétséges, mi hová tartozik, tegyük ki a blokkot

```
// hibás                  // ezt jelenti valójában
for (...)                 for (...)
    utasítás1                 utasítás1    // ez a ciklusmag
    utasítás2             utasítás2  // az ut. pontosan egyszer fut le
```

Utasítások: switch

- minden ág végére "kötelező" break-et tenni
  - ◇ szinte sose arra van szükségünk, hogy "átcsorogjon" (fall through) a vezérlés a következő ágba
  - ◇ ... hacsak nem több értékre is pont ugyanazt szeretnénk végrehajtani

```
switch (kifejezés) {
case érték1: prg1;
             break;
case érték2: prg2;
             break;
default:     prgDef;
             break;
case érték3:
case érték4: prg3;
}
```

- az alapértelmezett ág opcionális, és nem feltétlenül a legutolsó címke