


# Programozási nyelvek és módszerek

## BEVEZETÉS

# Fontos

## Oktató

- Tornai Kálmán
- 232-es szoba
- tornai.kalman@itk.ppke.hu

## Tárgyadatok

- Programozási nyelvek és módszerek
- P-ITSZT-0007
- P-ITSZT-0045
- Hétfő: 10.15-13.00
  - Neumann ea.
  - Zárva van órákon kívül, kint kell várakozni.

## Követelmények

- Jegy: Szóbeli vizsga
  - Vizsga előfeltétele az érvényes (>1) gyakorlati jegy a Java programozási nyelv tárgyból!
- Az előadások látogatása legalább 80%-ban kötelező
  - Maximum két hiányzás!
- A félév során két hétfő oktatási szünet
  - Tavaszi szünet

# Irodalom

## Kötelező irodalom (részek)

- Sebesta, Robert W., Concepts of programming languages, Edinburgh, Pearson, 2013, ISBN 978-0-273-76910-1
- Nyékyné Gaizler Judit et al., (szerk.), Programozási nyelvek, Budapest, Kiskapu, 2003

## Ajánlott irodalom

- SCOTT, Michael L., Programming Language Pragmatics, Amsterdam, Morgan Kaufmann, 2009
- Comparative programming languages, Harlow, Addison-Wesley, 2001

# Tematika

- Bevezetés
- A programozási nyelvek elemei
- Beépített adattípusok, változók, kifejezések
- Utasítások
- Alprogramok és paramétereik
- Absztrakt adattípusok
- Sablonok
- Kivételek
- Objektum-orientált programozás
- Helyesség
- Párhuzamosság
- A könyvtártervezés elvei
- A funkcionális programozás alapjai

# A félév célja

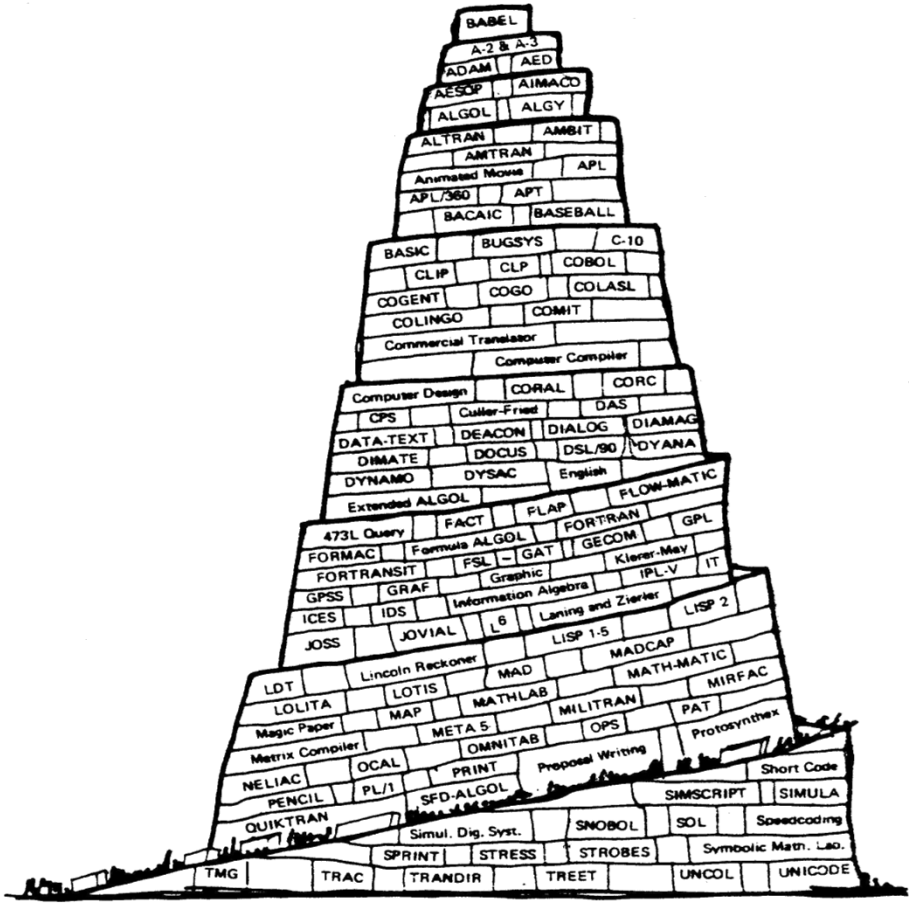
Programozási módszertan alapjai

Programozási nyelvek eszközkészlete

- Melyik nyelvet válasszuk egy feladatra?
- Milyen lehetőségek vannak?
- Milyen nyelvi stílusok vannak?
  - Ne ragadjunk bele a C++/Java logikájába
- Alapok későbbi programozási paradigmák, nyelvek elsajátítására
  - Mik a népszerű nyelvek most?
  - Mik lesznek?
    - Van még bőven idő a nyugdíjig …

# Ráhangolódás

# A nyelvek – Bábel tornya




Tower of Babel

# Mother Tongues

## Tracing the roots of computer languages through the ages

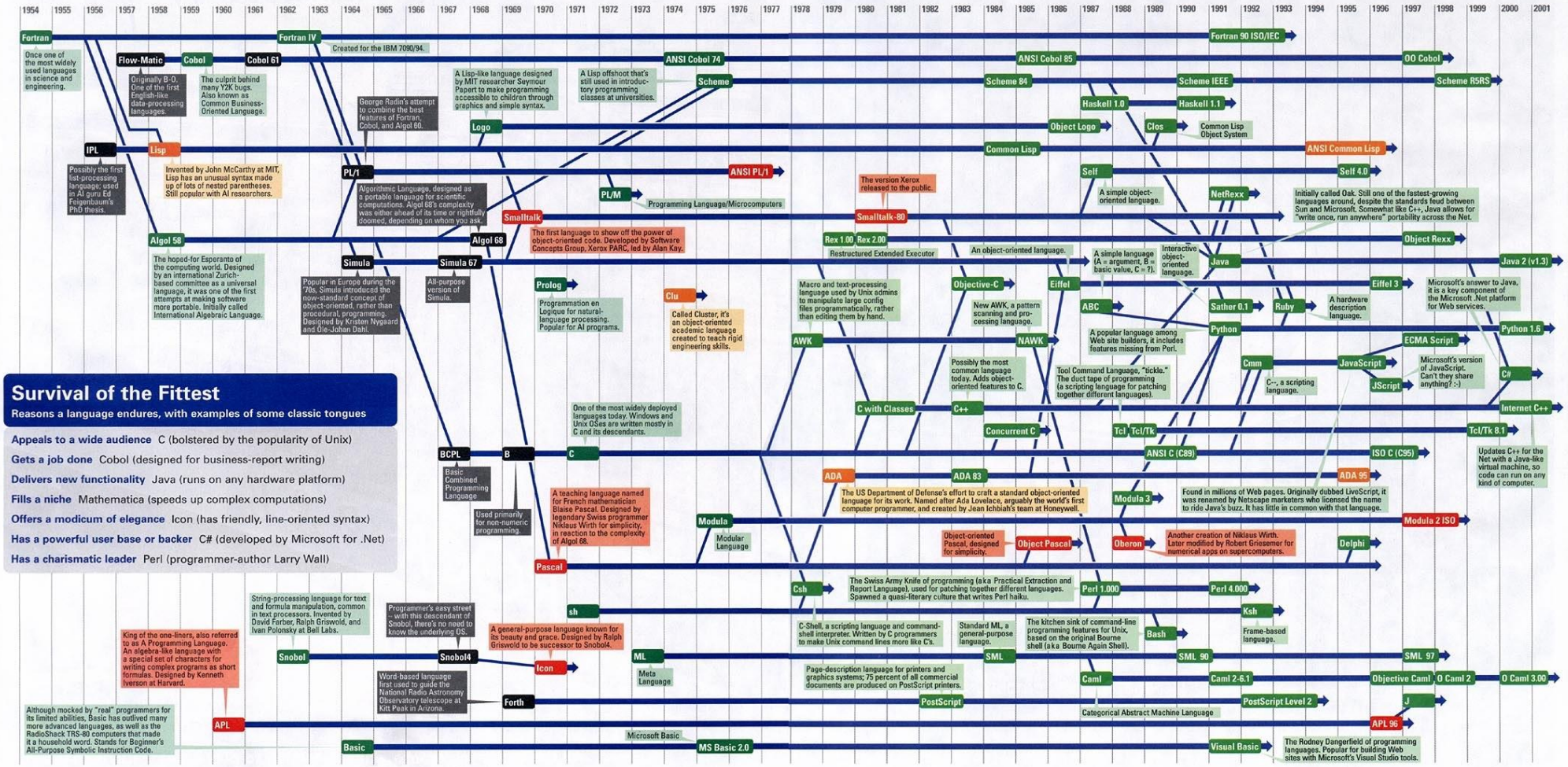
**Just like** half of the world's spoken tongues, most of the 2,300-plus computer programming languages are either endangered or extinct. As powerhouses C/C++, Visual Basic, Cobol, Java, and other modern source codes dominate our systems, hundreds of older languages are running out of life.

An ad hoc collection of engineers – electronic lexicographers, if you will – aim to save, or at least document, the lingo of classic software. They're combing the globe's 9 million developers in search of coders still fluent in these nearly forgotten lingua francas. Among the most endangered are Ada, APL, B (the predecessor of C), Lisp, Oberon, Smalltalk, and Simula.

Code-raker Grady Booch, Rational Software's chief scientist, is working with the Computer History Museum in Silicon Valley to record and, in some cases, maintain languages by writing new compilers so our ever-changing hardware can grok the code. Why bother? "They tell us about the state of software practice, the minds of their inventors, and the technical, social, and economic forces that shaped history at the time," Booch explains. "They'll provide the raw material for software archaeologists, historians, and developers to learn what worked, what was brilliant, and what was an utter failure." Here's a peek at the strongest branches of programming's family tree. For a nearly exhaustive rundown, check out the Language List at *www.informatik.uni-freiburg.de/Java/misc/lang_list.html*. **– Michael Menduno**

**Key**

- 1954 **Year Introduced**
- **Active:** thousands of users
- **Protected:** taught at universities; compilers available
- **Endangered:** usage dropping off
- **Extinct:** no known active users or up-to-date compilers
- **Lineage continues**



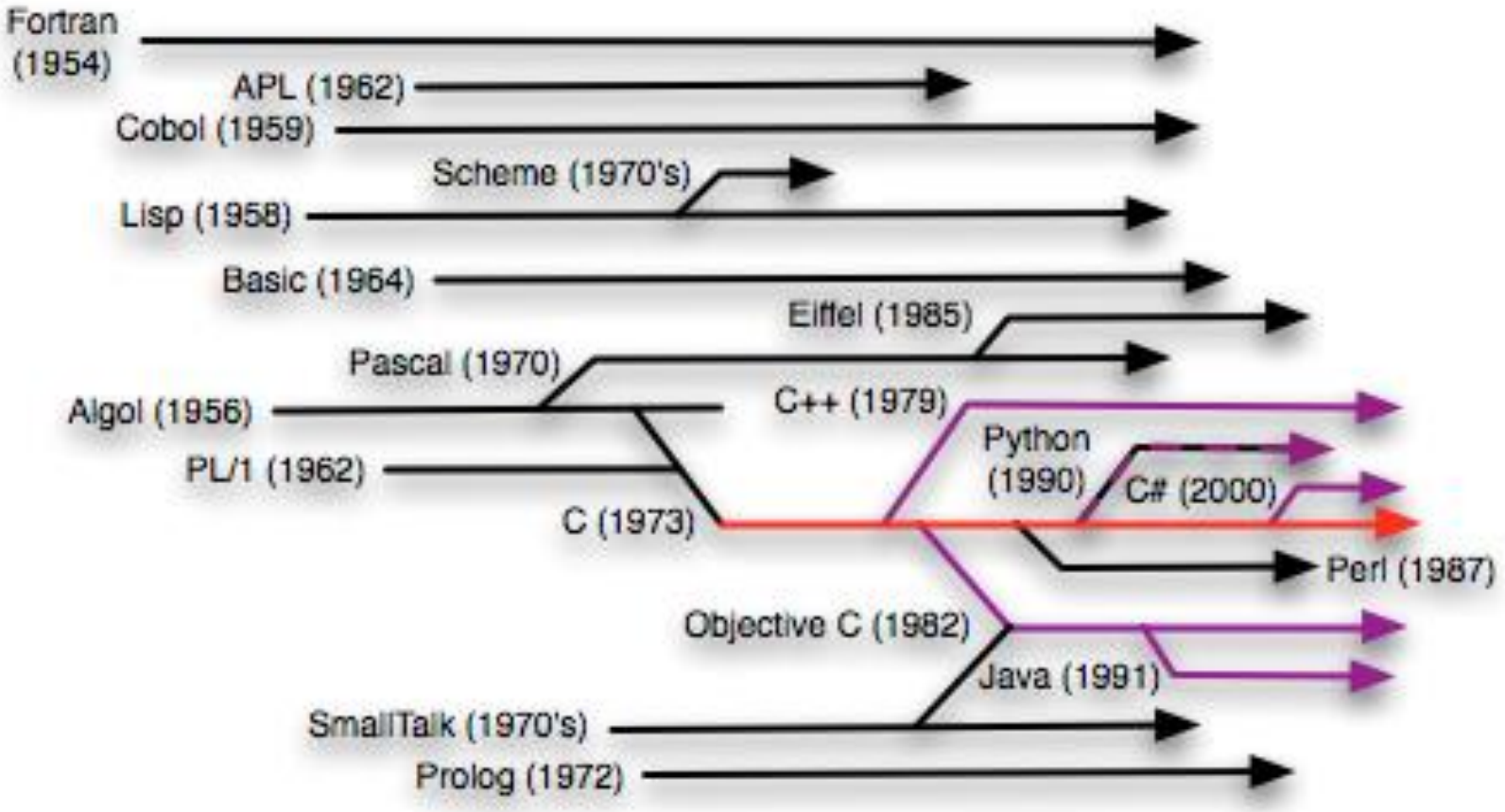

## Survival of the Fittest

**Reasons a language endures, with examples of some classic tongues**

- **Appeals to a wide audience** C (bolstered by the popularity of Unix)
- **Gets a job done** Cobol (designed for business-report writing)
- **Delivers new functionality** Java (runs on any hardware platform)
- **Fills a niche** Mathematica (speeds up complex computations)
- **Offers a modicum of elegance** Icon (has friendly, line-oriented syntax)
- **Has a powerful user base or backer** C# (developed by Microsoft for .Net)
- **Has a charismatic leader** Perl (programmer-author Larry Wall)

Sources: Paul Boutin; Brent Hailpern, associate director of computer science at IBM Research; The Retrocomputing Museum; Todd Proebsting, senior researcher at Microsoft; Gio Wiederhold, computer scientist, Stanford University

# „Fontosabb” nyelvek családfája



# Miért van ennyi?

Számos ok

- Evolúció
  - Újabb nyelvi konstrukciók, megközelítések
  - Pl: OOP, szkriptnyelvek
- Speciális célok
  - C: alacsonyszintű programozás
  - Awk: karakter alapú feldolgozás
  - Python: könnyű prototipizálás, számítások
- Személyes preferenciák
  - Rekurziós algoritmusok <-> Iteratív algoritmusok
  - Pointerek használata <-> automatikus (implicit) referencing

# Mitől sikeres egy nyelv?

Több ok

- Kifejező erő
  - Gyakori érv bizonyos nyelvek mellett / ellen, hogy „erős”
  - Matematikai értelemben nem igaz, mindegyik nyelv azonos kifejezőerővel bír, minden algoritmus leírható benne
    - Lásd később
    - A különbség a szintaktikai támogatottság
      - Le lehet írni, más kérdés, hogy mennyire bonyolultan
- Kezdők számára könnyű használat
- Könnyű implementálhatóság
  - Fordítóprogram, környezet
- Szabványosság
  - „Nyelvjárásoktól mentes”

# Mitől sikeres egy nyelv?

Több ok

- Nyílt forrás
  - Manapság szinte bármely nyelvre igaz, hogy van nyílt forrású implementáció
- Jó fordítóprogram
  - Hibamentes
  - Jó optimalizálás
    - Gyors kódot állít elő
  - Könnyű használhatóság
    - Toolchain

# Történelmi mérföldkövek

FORTRAN (1950-es évek közepe-vége)

- hangsúly a tudományos programozáson

LISP (1950-es évek vége)

- rekurzió, dinamikus helyfoglalás és felszabadítás

ALGOL 60 (1960)

- az adattípus fogalma
- érték- és név szerinti paraméterátadás

COBOL (1960)

- angol-szerű szintaxis

# Történelmi mérföldkövek

BASIC (1960-as évek közepe)
- Interpretált

PL/I (1960-as évek közepe)
- Általános célú
- Párhuzamosan végrehajtódó taszkok
- Kezeli a futási idejű kivételeket
- Pointerek mint adattípusok

APL (1960 körül)
- Tömb és mátrix kezelés

SNOBOL (1960-as évek közepe)
- Szövegkezelés, szövegminták hasonlítására

# Történelmi mérföldkövek

SIMULA 67 (1967)
- Osztály fogalmának bevezetése – egységbezárás
- Öröklődés

ALGOL 68 (1968)
- A legtöbb ezt követő programozás nyelv merített a tervezéséből

Pascal (1970-es évek eleje)
- Programozás oktatásra

C (1970-es évek eleje)
- rendszerprogramozásra

Prolog (1970-es évek közepe)
- Logikai nyelv

# Történelmi mérföldkövek

Ada (1970-es évek vége)

- A Pentagon megbízásából
- Kivételkezelés
- Sablonok
- Párhuzamosság támogatása (taszkok, randevúk)
- Helyességellenőrzés (2012-től)

Smalltalk (1980)

- Objektumorientált – máshogy

C++ (1980-as évek eleje óta)

- SIMULA 67 és a Smalltalk + C

# Történelmi mérföldkövek

Objective C (1980-as évek eleje)
- C és Smalltalk

Eiffel (1980-as évek eleje)
- Objektumorientált
- Tervezés szerződéssel a függvényhívó és a függvény között

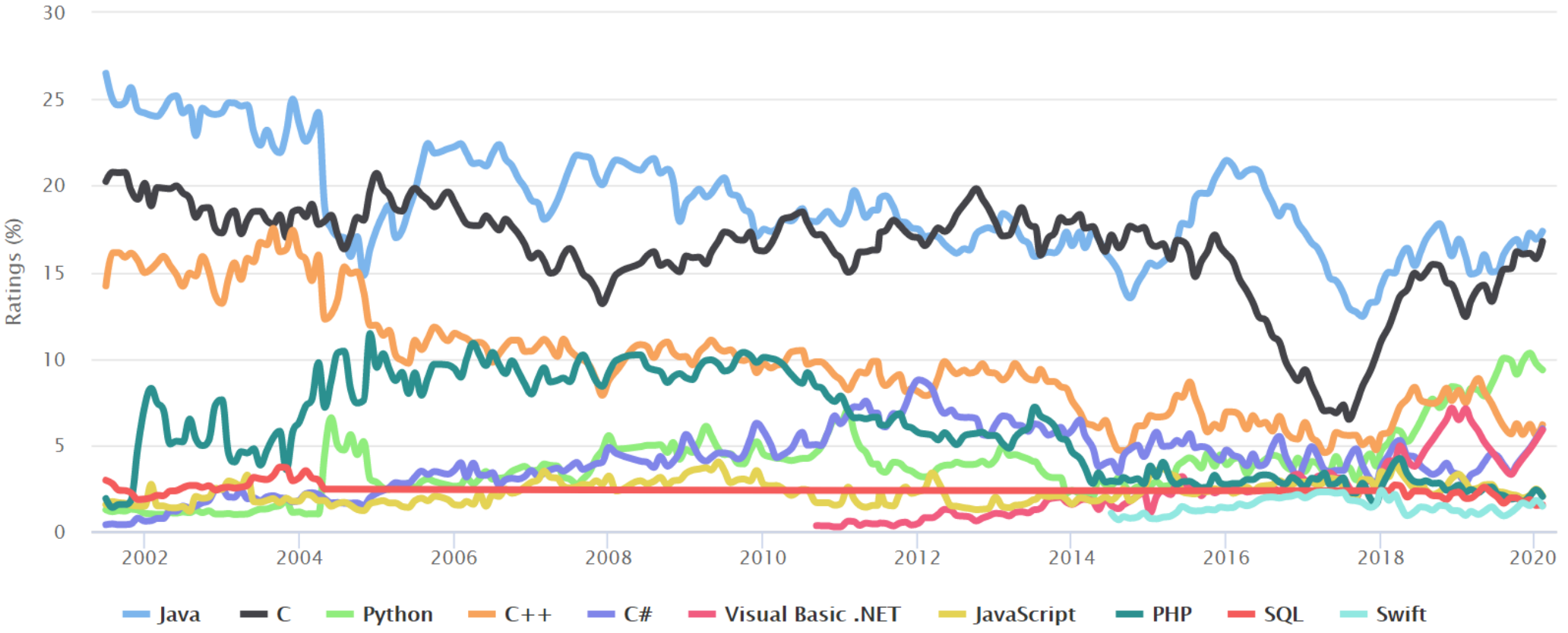
Java (1990-es évek közepe)
- OOP
- Párhuzamosság támogatása
- GC
- Kivételkezelés

C#
- „Microsoft Java-ja”
- .NET alapú fejlesztés támogatása

# Mitől fontos egy programozási nyelv?

HTTPS://WWW.TIOBE.COM/TIOBE-INDEX/
HTTPS://SPECTRUM.IEEE.ORG/COMPUTING/SOFTWARE/THE-TOP-PROGRAMMING-LANGUAGES-2019

# TIOBE Index


Ratings (%)
30
25
20
15
10
5
0
2002
2004
2006
2008
2010
2012
2014
2016
2018
2020
Java
C
Python
C++
C#
Visual Basic .NET
JavaScript
PHP
SQL
Swift

# TIOBE Index

| 2020 | 2019 | 2018 | 2017 | 2016 | 2015 | 2014 | 2013 | 2012 | 2011 | 2010 | 2009 | 2008 | Nyelv |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 2 | 2 | 1 | 2 | 1 | 1 | 1 | 1 | Java |
| 2 | 2 | 2 | 2 | 2 | 1 | 1 | 2 | 1 | 2 | 2 | 2 | 2 | C |
| 3 | 3 | 4 | 5 | 5 | 8 | 8 | 7 | 8 | 8 | 7 | 8 | 6 | Python |
| 4 | 4 | 3 | 3 | 3 | 3 | 4 | 4 | 4 | 3 | 3 | 4 | 3 | C++ |
| 5 | 7 | 5 | 4 | 4 | 5 | 5 | 5 | 5 | 4 | 6 | 7 | 8 | C# |
| 6 | 5 | 7 | 6 | 7 | 9 | 10 | 12 | 15 | 39 | | | | Visual Basic .NET |
| 7 | 6 | 6 | 7 | 9 | 6 | 9 | 11 | 11 | 10 | 11 | 9 | 9 | JavaScript |
| 8 | 8 | 9 | 10 | 6 | 7 | 6 | 6 | 6 | 5 | 4 | 3 | 5 | PHP |
| 9 | 9 | | | | | | | | | | | | SQL |
| 10 | 15 | 12 | 14 | 16 | | | | | | | | | SWIFT |

# TIOBE Index

| 2020 | 2019 | 2018 | 2017 | 2016 | 2015 | 2014 | 2013 | 2012 | 2011 | 2010 | 2009 | 2008 | Nyelv |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 11 | 16 | 19 | 13 | | | | | | | | | | Go |
| 12 | 14 | 15 | 9 | 13 | | | | | | | | | Assembly |
| 13 | 12 | 8 | 16 | 17 | 18 | 44 | | | | | | | R |
| 14 | 23 | | | | | | | | | | 12 | | D |
| 15 | 18 | 11 | 12 | 11 | 20 | 12 | 10 | 10 | 11 | 10 | 10 | 11 | Ruby |
| 16 | 11 | 18 | 19 | 18 | 17 | 14 | 18 | 20 | 22 | | | | MATLAB |
| 17 | 19 | 20 | 20 | 19 | 13 | 15 | 22 | 19 | 17 | 20 | 14 | 13 | PL/SQL |
| 18 | 17 | 13 | 11 | 10 | 11 | 20 | 15 | 12 | 12 | 12 | 11 | 10 | Delphi / Object Pascal |
| 19 | 13 | 10 | 8 | 11 | 12 | 13 | 9 | 9 | 9 | 9 | 6 | 7 | Perl |
| 20 | 10 | 16 | 18 | 14 | 4 | 3 | 3 | 3 | 6 | 8 | 18 | 42 | Objective-C |

# TIOBE hosszútávú összehasonlítás

| Nyelv | 2020 | 2015 | 2010 | 2005 | 2000 | 1995 | 1990 | 1985 |
|---|---|---|---|---|---|---|---|---|
| Java | 1 | 2 | 1 | 2 | 3 | - | - | - |
| C | 2 | 1 | 2 | 1 | 1 | 2 | 1 | 1 |
| Python | 3 | 7 | 6 | 6 | 22 | 21 | - | - |
| C++ | 4 | 4 | 4 | 3 | 2 | 1 | 2 | 12 |
| C# | 5 | 5 | 5 | 8 | 8 | - | - | - |
| Visual Basic .NET | 6 | 10 | - | - | - | - | - | - |
| JavaScript | 7 | 8 | 8 | 9 | 6 | - | - | - |
| PHP | 8 | 6 | 3 | 4 | 27 | - | - | - |
| SQL | 9 | - | - | 97 | - | - | - | - |
| Objective-C | 10 | 3 | 21 | 37 | - | - | - | - |
| Lisp | 31 | 18 | 16 | 13 | 14 | 5 | 3 | 2 |
| Ada | 35 | 29 | 24 | 15 | 15 | 6 | 4 | 3 |
| Pascal | 229 | 16 | 13 | 65 | 11 | 3 | 15 | 5 |

# IEEE Rank – 2019

| Rank | Language | Type | Score |
|---|---|---|---|
| 1 | Python | web, desktop, embedded | 100.0 |
| 2 | Java | web, mobile, desktop | 96.3 |
| 3 | C | mobile, desktop, embedded | 94.4 |
| 4 | C++ | mobile, desktop, embedded | 87.5 |
| 5 | R | desktop | 81.5 |
| 6 | JavaScript | web | 79.4 |
| 7 | C# | web, mobile, desktop, embedded | 74.5 |
| 8 | Matlab | desktop | 70.6 |
| 9 | Swift | mobile, desktop | 69.1 |
| 10 | Go | web, desktop | 68.0 |

# Miért tanuljunk (tovább) programozási nyelveket?

Egyben a tárgy célja is

- Megérteni a nyelvekben rejlő további funkcionalitást
  - C++ union típuskonstrukció
  - Változó számú argumentumok
- Alternatív módon kifejezni megoldásokat
  - Választott nyelvnek megfelelően
    - Matlab: mátrix műveletek
    - Python: list comprehensing
      - `x = [2 ** x for x in range(10) if x > 5]`
- Hibakeresők, assemblerek, …
  - Használatuk, céljuk megértése

# Miért tanuljunk (tovább) programozási nyelveket?

Egyben a tárgy célja is

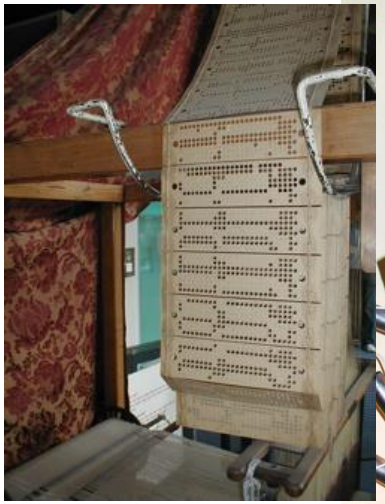
- Hatékony, hasznos nyelvi eszközök „átültetése”
  - Megvalósítás / Szimulálás más nyelven
- Szélesíteni a látókört a programnyelvek nyelvi eszközkészletét illetően
  - További konstrukciók
  - Megfelelő eszköz kiválasztása
  - Nem megrettenni egy új nyelvtől
    - Stratégia a megtanulásra

# Szoftverminőség

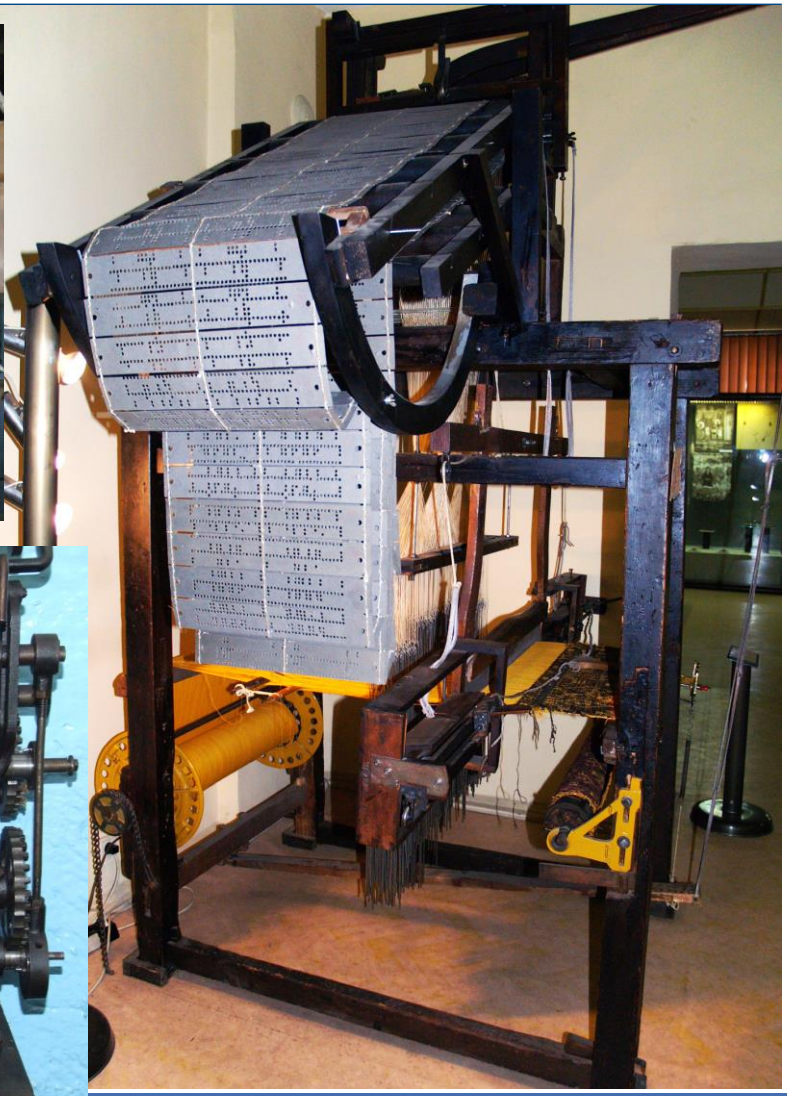
# Miért készítünk programokat?



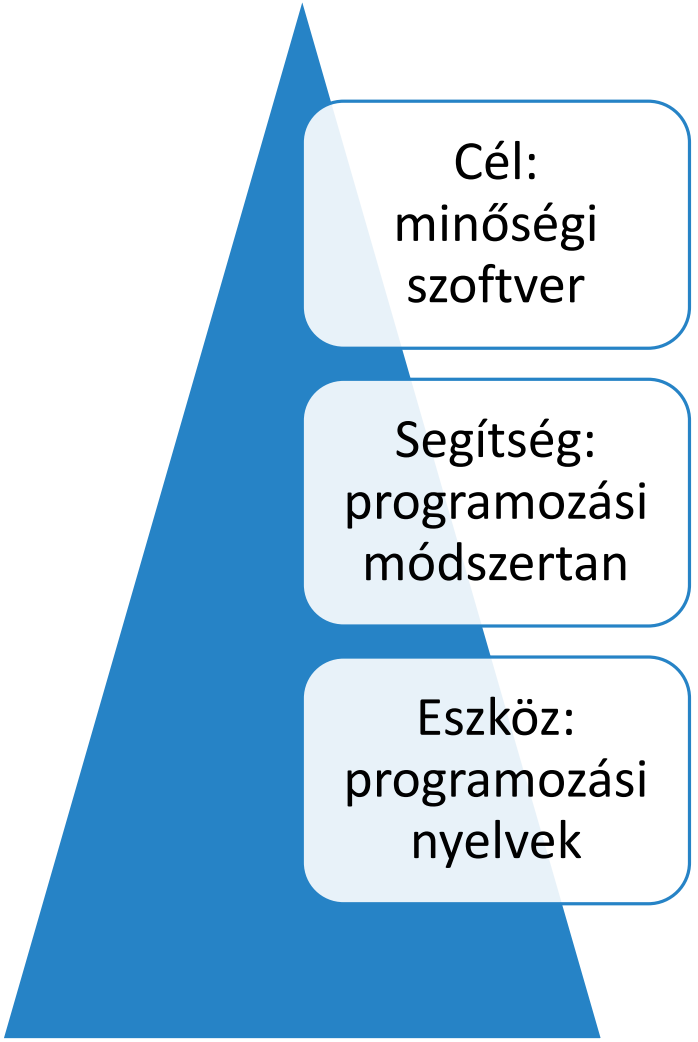
Joseph Marie Jacquard
(szövőgép)







# Minőségi szoftver




Szempontok

- Helyesség
- Megbízhatóság
- Karbantarthatóság
- Újrafelhasználhatóság
- Kompatibilitás
- Hordozhatóság
- Hatékonyság
- stb.

# Minőségi szoftver

## Helyesség

- A program helyes, ha pontosan megoldja a feladatot, megfelel a kívánt specifikációnak.
  - Ehhez szükséges a pontos és minél teljesebb specifikáció.

## Megbízhatóság

- A program megbízható, ha
  - *Helyes*, és
  - Abnormális helyzetekben sem történik katasztrófa
    - Valamilyen „ésszerű" működést produkál
    - Abnormális: a specifikációban nem leírt

# Minőségi szoftver

## Karbantarthatóság

- A karbantarthatóság annak mérőszáma, hogy milyen könnyű a programterméket a *specifikáció változtatásához* adaptálni.
  - Egyes felmérések szerint a szoftver költségek 70 %-át szoftver karbantartásra fordítják!
- Növelésének két alapelve
  - Tervezési egyszerűség
  - Decentralizáció – minél önállóbb modulok létrehozása

## Újrafelhasználhatóság

- A szoftver termék, vagy komponens egy másik, új alkalmazásban felhasználható.
  - Ez vonatkozik a szoftvertermék
    - Egészére
    - Részére

# Minőségi szoftver

## Kompatibilitás

- A kompatibilitás megmutatja, hogy mennyire könnyű a szoftver termékeket egymással kombinálni.
  - Nem légüres térben fejlesztünk!

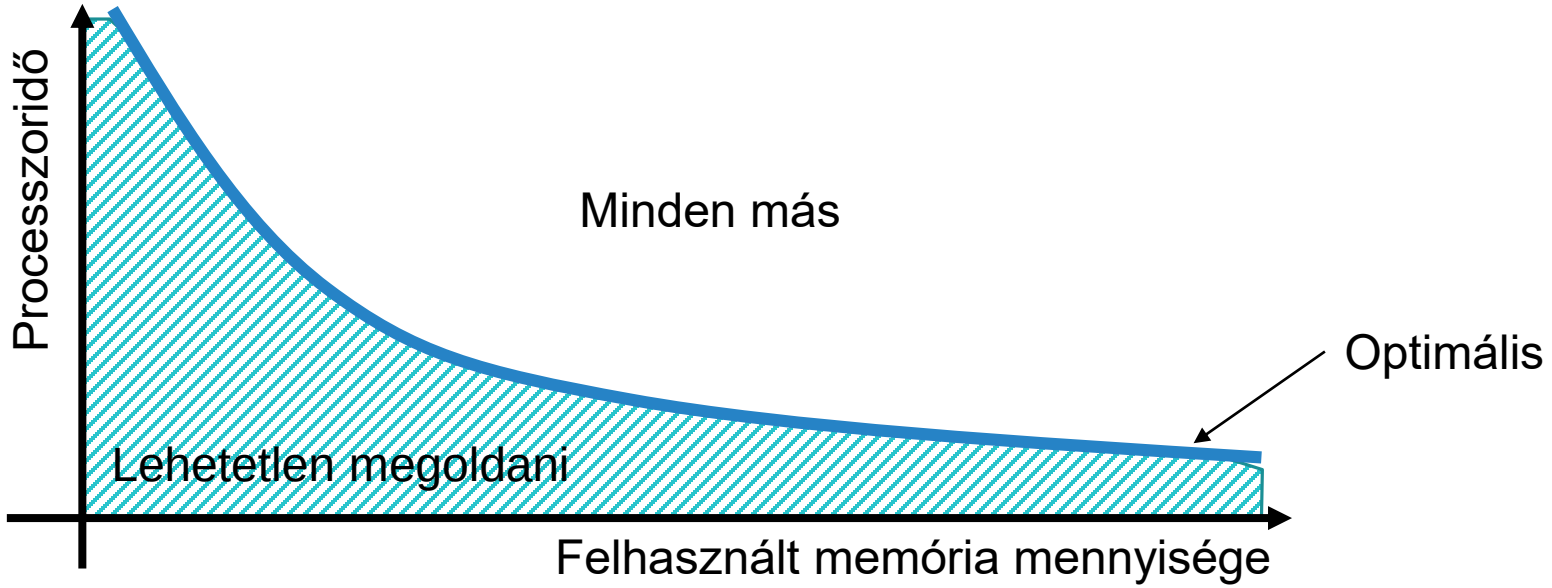
## Hordozhatóság

- A program hordozhatóságának mértéke mutatja meg, mennyire könnyű a programot átalakítani
  - Más géphez
  - Más konfigurációhoz
  - Más operációs rendszerhez
  - Általában más környezethez

# Minőségi szoftver

## Hatékonyság

- A program hatékonysága a futási idővel és a felhasznált memória méretével, illetve *további erőforrások* használatával arányos.
  - Egyszerűsítve: egy program hatékony, ha gyorsan lefut és kevés memóriát használ.



# Minőségi szoftver

## Barátságosság

- A program emberközelisége, barátságossága a felhasználó számára rendkívül fontos
  - Megköveteli, hogy legyen
    - az input logikus és egyszerű
    - az eredmények formája áttekinthető

## Tesztelhetőség

- A tesztelhetőség, áttekinthetőség a program karbantartói, fejlesztői számára fontos.

# Modularitás

# Programozási módszertanok

## Moduláris tervezés

- Programjainkat egymástól minél inkább független, de jól definiált kapcsolatban álló programegységek segítségével tervezzük.

## Paradigmák

- Procedurális
- Objektumorientált
- Funkcionális
- Logikai
- Szimbolikus
- …

# Moduláris dekompozíció

Jelentése

- A feladat több egyszerűbb részfeladatra bontása
- A részfeladatok megoldása egymástól függetlenül elvégezhető
- Ennek segítségével csökkentjük a feladat bonyolultságát.

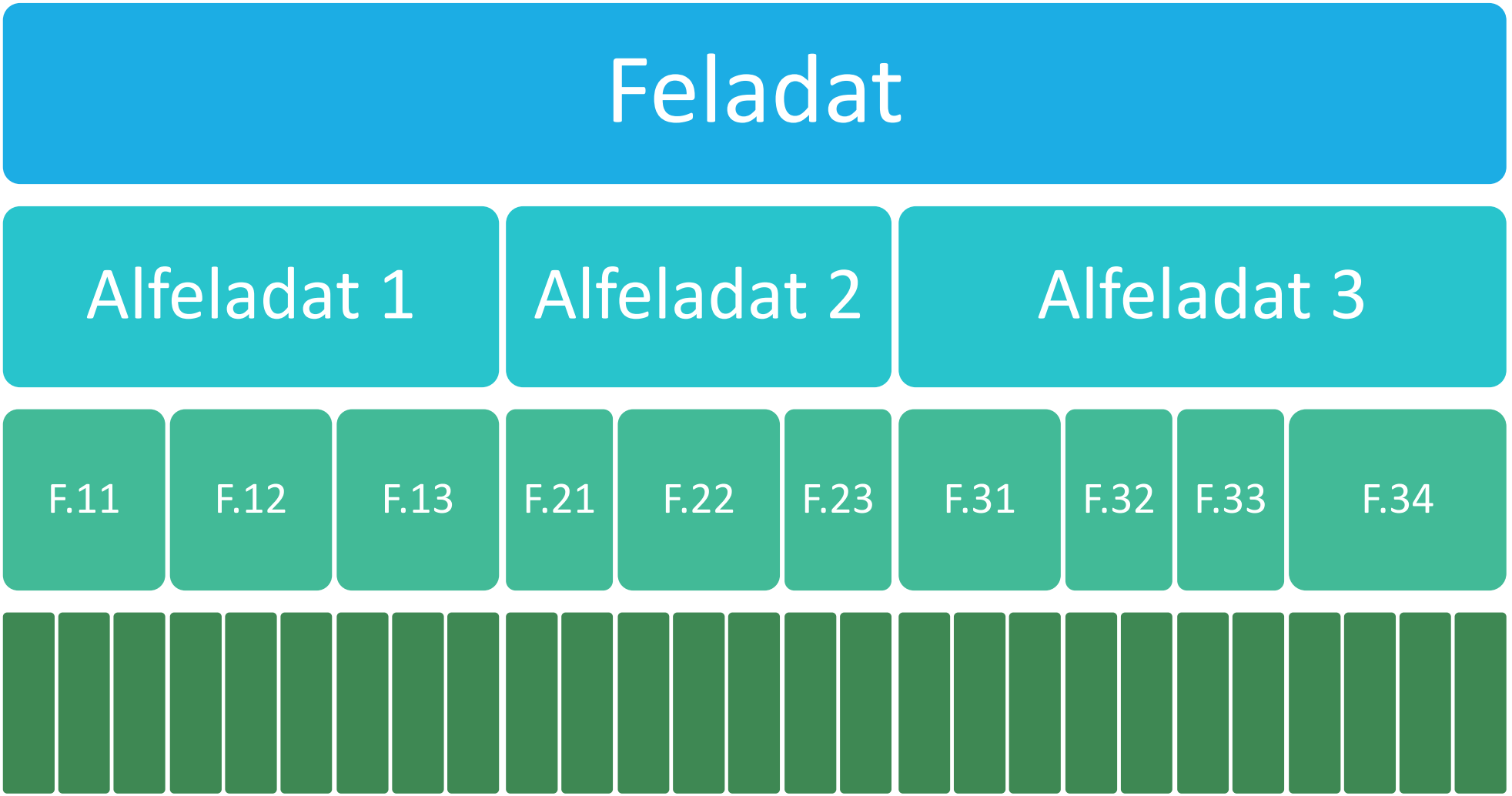
Általában a módszert ismételten alkalmazzuk

- Azaz az alrendszereket magukat is dekomponáljuk.
- Ez lehetővé teszi, hogy a feladat megoldásán egyszerre többen dolgozzanak

A módszer általi felbontás egy fával megadható.

- A fa csomópontjai az egyes dekompozíciós lépések

# Moduláris dekompozíció

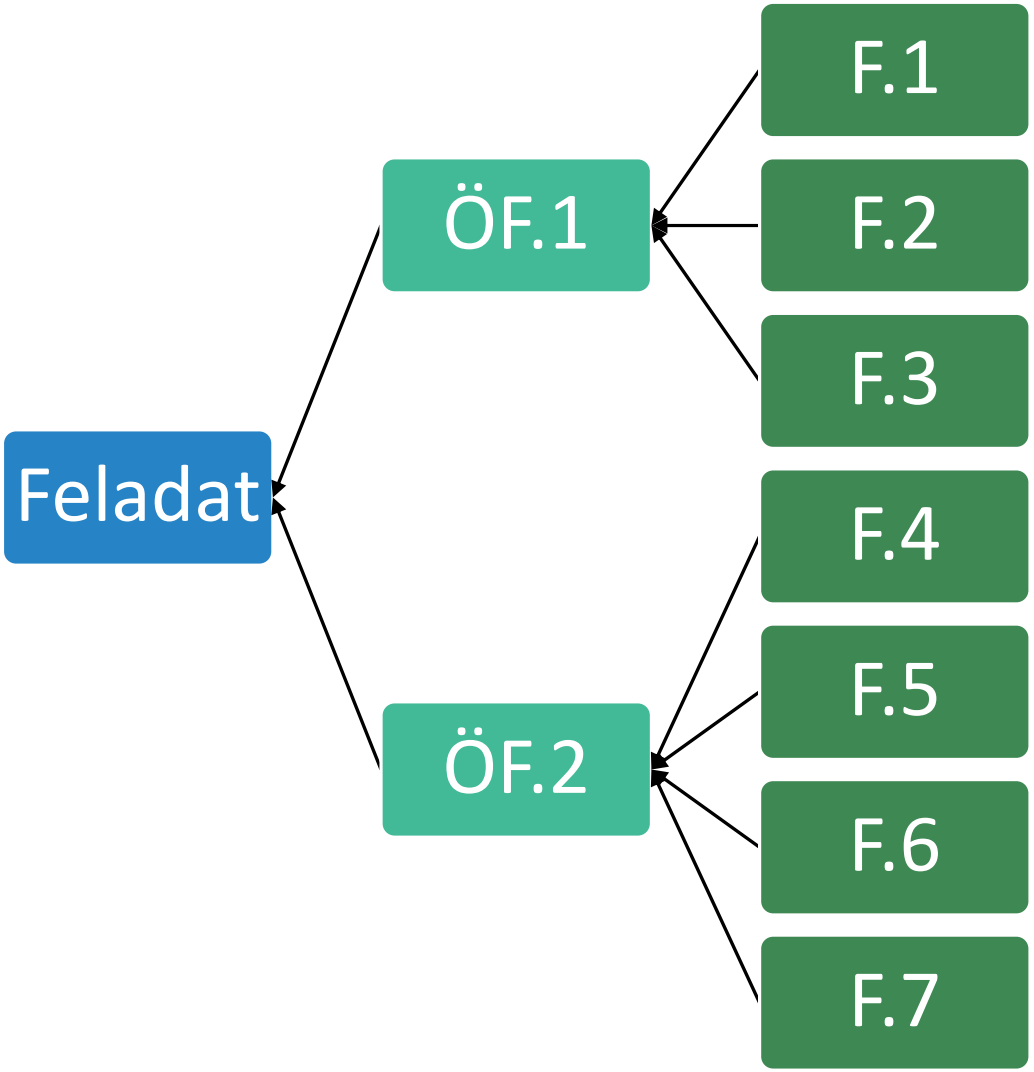


# Moduláris kompozíció

Olyan szoftver elemek létrehozását támogatja, amelyek szabadon kombinálhatók egymással.

- A programok lehetőleg a már meglévő programegységekből, mint építőkövekből épülnek fel.

# Moduláris kompozíció



# Modularitás

## Érthetőség

- A modulok önállóan is egy-egy értelmes egységet alkotnak
  - Megértésükhez minél kevesebb „szomszédos” modulra van szükség

## Folytonosság

- A specifikáció „kis” változtatása
  → a programban is csak „kis” változtatásra van szükség

## Védelem

- Célunk a program egészének a védelme az abnormális helyzetek hatásaitól.
  - Egy hiba hatása egy – esetleg néhány – modulra korlátozódjon

# A modularitás alapelvei

A modulokat nyelvi egységek támogassák

- A modulok illeszkedjenek a használt programozási nyelv szintaktikai egységeihez.

Kevés kapcsolat legyen

- Minden modul minél kevesebb másik modullal kommunikáljon!

Gyenge legyen a kapcsolat

- A modulok olyan kevés információt cseréljenek, amennyi csak lehetséges!

Explicit interface kell

- Ha két modul kommunikál egymással, akkor annak ki kell derülnie legalább az egyikük szövegéből.

# A modularitás alapelvei

## Információ elrejtés

- Minden információ egy modulról rejtett kell legyen, kivéve, amit explicit módon nyilvánosnak deklaráltunk.

## Nyitott és zárt modulok

- Zárt modul: más modulok számára egy jól definiált felületen keresztül elérhető, a többi modul változatlan formában felhasználhatja.
- Nyitott modul: kiterjeszthető
  - az általa nyújtott szolgáltatások bővíthetők vagy
  - hozzávehetünk további mezőket a benne levő adatszerkezetekhez, s ennek megfelelően módosíthatjuk eddigi szolgáltatásait.

# Újrafelhasználhatóság

A típusok változatossága
- A modulok többféle típusra is működjenek.

Adatstruktúrák és algoritmusok változatossága

Egy típus – egy modul
- Egy típus műveletei kerüljenek egy modulba.

Reprezentáció-függetlenség

# Programkönyvtárak

# Újrafelhasználható programkönyvtárak

## Tervezési igények

- Az alap-követelmények ugyanazok, mint  más szoftver tervezésénél
  - Helyesség
  - Megbízhatóság
  - Karbantarthatóság
  - Újrafelhasználhatóság
  - Kompatibilitás
  - Hordozhatóság
  - Hatékonyság
  - stb.
- A programkönyvtár felé az igények erőteljesebbek!

# Újrafelhasználható programkönyvtárak

Osztályokkal kapcsolatos további elvárások:

- Könnyű és intuitív használat
  - Könnyen megérthető legyen, néhány használat után jól megjegyezhető
- Azonnali, széleskörű felhasználhatóság
  - Szolgáltatásokat nyújt
- A lehető leghatékonyabb megvalósítás – state-of-the-art
- A tesztelés körültekintősége és az osztály megbízhatósága
- Magas szintű dokumentáció
  - Pontos, jól szervezett, hogy a felhasználó könnyen eligazodjon
- Platform-függetlenség
- Sokszor nyelv-függetlenség
- Meg kell hagyni a későbbi fejlesztés lehetőségét

# Újrafelhasználható programkönyvtárak

Néhány következmény

- Konzisztencia
  - a könyvtár minden komponense egy átfogó, koherens tervezésnek felel meg
  - számos szisztematikus, explicit és egységes konvenciót követ
- Jó minőségű, homogén komponensek kellenek
- OOP megközelítés az adatabsztrakció támogatása miatt különösen is alkalmas a könyvtárak készítésére

Ezek egyidejű teljesítése nehéz

- Lehetséges, hogy a legáltalánosabb algoritmus nem a leghatékonyabb sok speciális, de gyakran előforduló esetben.

# Újrafelhasználható programkönyvtárak

A könnyű felhasználhatóságnak különböző aspektusai vannak

- Az egyik ezek közül az osztályok elkülöníthetősége
  - Ha nagyon sok hasonló osztály található egy könyvtárban, akkor a felhasználó számára igen nehéz a pontosan megfelelő kiválasztása.
- Általában előnyösebb a kevés számú, majdnem teljesen ortogonális osztály definiálása
  - Minden esetben egy jól meghatározott szerep betöltésére
    - Az optimális megvalósítást követve

# Fejlesztő tulajdonságai

## A jó könyvtárfejlesztő tulajdonságai

- Van absztrakciós készsége
  - Képes arra, hogy az egyedi jelenségekből általánosítson
- Van érzéke a részletekhez
  - Egy könyvtárban minden kis tulajdonság számít.
  - Itt nincs olyan, hogy 'elég jó’
    - Teljes kell legyen a könyvtár
- Rend-mániákus
  - Képes az osztályozásra, hogy mindennek megtalálja a helyét.

# Fejlesztő tulajdonságai

Van irodalmi érzéke

- A leírásainak, a specifikációs részeket magyarázó megjegyzéseknek legyen stílusa és eleganciája

Elsőrendű programozó és tervező kell legyen

- Tudja átlátni a könyvtár későbbi használójának igényeit

„Ego-less” legyen

- Nem szabad, hogy egyéni stílus érvényesüljön
  - A cél a konzisztens stílus használatának lehetősége
- A kreativitása a könyvtár által nyújtott szolgáltatásokban jöjjön elő!

# Programkönyvtárak jellemzése

Egy programkönyvtár osztályok gyűjteménye.

- Egy osztályt az általa nyújtott szolgáltatások jellemeznek.

Ezen szolgáltatásokat a következő két módon lehet csoportosítani:

- I.
  - Számított (rutin)
    - Függvény
    - Parancs
  - Tárolt
    - Attribútum

# Programkönyvtárak jellemzése

Egy programkönyvtár osztályok gyűjteménye.

- Egy osztályt az általa nyújtott szolgáltatások jellemeznek.

Ezen szolgáltatásokat a következő két módon lehet csoportosítani:

- II.
  - Értéket visszaadó
    - Számított – függvény
    - Tárolt – attribútum
  - Parancs

# Programkönyvtárak jellemzése

Osztályok csoportosítási lehetősége

- csomagok (Java)
- clusterek (Eiffel)

használata.

A könyvtár és a csomag fogalmak kapcsolata lehet

- 1-sok
- sok-1

A csomagok gyakran a fizikai fájlrendszer struktúráját követik.

# Osztályhierarchia

OO esetben egy programkönyvtár elkészítése egy osztályhierarchia kialakítását is jelenti.

- A használhatóság szempontjából nem lényegtelen ezen hierarchia bonyolultsága vagy egyszerűsége.

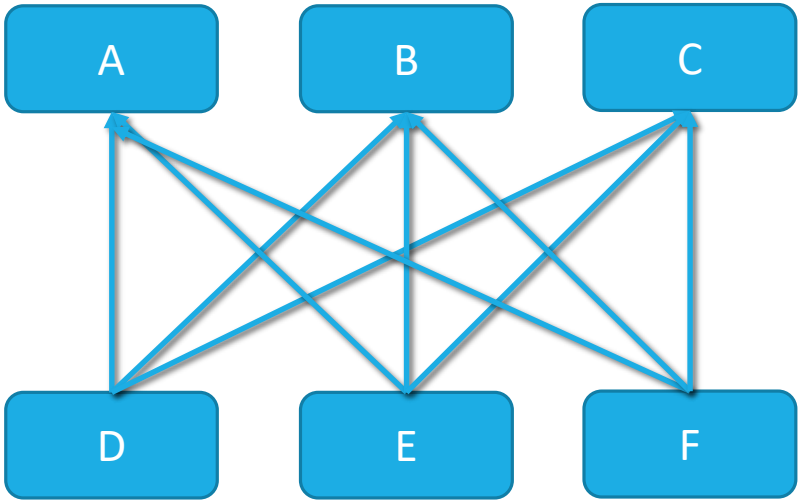
Az objektumorientált szemléletnek megfelelően

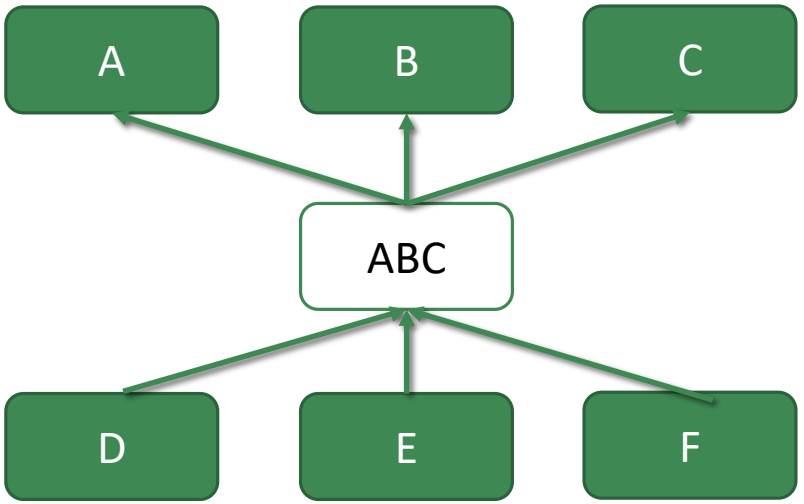
- igyekezzünk minél általánosabb osztályokat bevezetni (generalizáció)
- az öröklődést felhasználva hozzuk létre a szükséges osztályokat (specializálódás).

Ezáltal az osztályhierarchia karbantartása is leegyszerűsödik.

# Osztályhierarchia

Új osztály bevezetésének másik oka az osztályhierarchia egyszerűsítése is lehet:



# Biztonság

Egy sokak által használt programkönyvtár esetén fontos szempont a megbízhatóság és a biztonság.

- elő- és utófeltételek megadása
- invariánsok adásával

„Szerződés” létrejötte

- az előfeltétel a felhasználó kötelezettsége és a programozó biztosítéka
- az utófeltétel a programozó kötelezettsége és a használó biztosítéka
- Nem kell „defenzíven” programozni
  - minden lehetséges hibalehetőségre és bemenetre felkészülve
  - a kód hatékonyabb és egyszerűbb lesz

# Biztonság

## Két megközelítési mód

- Toleráns
  - A programkönyvtári rutinoknak nincs (vagy csak gyenge) előfeltételük
  - Minden lehetséges bemenetre valamilyen módon reagálnak
- Követelőző
  - Minden rutin szigorú előfeltételekkel rendelkezik
  - Ezek biztosítása a felhasználó felelőssége

## Az ajánlott megközelítés:

- Csak az absztrakt művelet logikai helyes elvégzéséhez szükséges előfeltételeket ellenőrizzük.
- Csak azt ellenőrizzük, ami, ha nem teljesülne, súlyosan befolyásolná a hatékonyságot.

# Biztonság

Az így létrejött szerződés bármilyen megsértése programhibát jelent

- Az előfeltétel megsérülése felhasználó oldali hiba
- Az utófeltétel vagy az invariáns sérülése programkönyvtár hiba

A programkönyvtár fejlesztése után az előfeltétel teljesülés vizsgálatát kell elvégezni

- A terjesztett változatban a hatékonyság érdekében.

# Dokumentáció

Egy programkönyvtár több fejlesztő munkája.

A felhasználhatóság szempontjából ezért igen fontos a jó dokumentáció.

Lehetséges megközelítési módok:

- „Forrásszöveg maga a dokumentáció, mi kell még?”
- Különálló dokumentáció írása

# Példák

```
/**
 * Always returns true.
 */
public boolean isAvailable() {
    return false;
}
```

```
for (j=0; j<arra_len; j+ =8) {
  total += array[j+0];
  total += array[j+1];
  total += array[j+2]; /* comment
  total += array[j+3];  * out
  total += array[j+4];  * some
  total += array[j+5];  * lines
  total += array[j+6];  */
  total += array[j+7];
}
```

```
#define TRUE FALSE
```

```
a=a++
```

```
*++b ? (*++b + *(b-1)) 0
```

- Nem definiált viselkedés, a precedencia miatt

# Híres példa – $1/\sqrt{(x)}$

```
float Q_rsqrt( float number )
{
    long i;
    float x2, y;
    const float threehalfs = 1.5F;

    x2 = number * 0.5F;
    y = number;
    i = * ( long * ) &y; // evil floating point bit level hacking
    i = 0x5f3759df - ( i >> 1 ); // WTF?
    y = * ( float * ) &i;
    y = y * ( threehalfs - ( x2 * y * y ) );
                        // 1st iteration
// y = y * ( threehalfs - ( x2 * y * y ) );
                        // 2nd iteration, this can be removed
    return y;
}
```

# Dokumentáció

A forrásszöveg nem elég absztrakt

- A felhasználó számára fontos információkon kívül az alacsonyszintű implementációt is tartalmazza
- A használata túl sok információ átfésülését jelentené
- Arra sarkallná a programozókat, hogy nem publikusnak szánt implementációs lehetőségeket is kihasználjanak
- Előnye, hogy mindig aktuális.

Különálló dokumentáció esetén nem biztosított a szoftver és a dokumentáció konzisztenciája.

- Előnye, hogy csak a felhasználó számára fontos dolgokat tartalmazza.

# Dokumentáció

Belső dokumentáció elve: a szoftver dokumentációja a programszövegbe legyen beágyazva.

Ennek előnyei:

- Forrásszöveg és dokumentáció mindig konzisztens marad.
  - Előbbi példa …
- Dokumentáció kinyerése automatizálható
  - javadoc

Eiffel flat-short forma

- az osztálynak egy olyan verziója, ami tartalmazza az öröklött és az itt bevezetett jellemzőket
  - figyelembe véve minden átnevezést és átdefiniálást
  - felépítve a teljes elő- és utófeltételt és a típusinvariánst

# Karbantartás

Egy programkönyvtár állandó fejlesztés alatt áll,

- Lehetőséget kell biztosítani újabb verziók problémamentes beépítésére.

A következő változtatások nem jelentenek problémát:

- Osztály bővítése új szolgáltatással.
- Egy szolgáltatás implementációjának lecserélése
- Előfeltétel gyengítése, illetve utófeltétel szigorítása

# Karbantartás

Szükség lehet egy szolgáltatás új verziójának bevezetésére, ha

- A szolgáltatás neve megváltozik
  - Például egységes névkonvenció kialakítása miatt
- Jobb szignatúra vagy specifikáció megadása
- Rutin elavulttá nyilvánítása.
  - Ez történhet nyelvi kulcsszavak használatával
    - Eiffel: obsolete
    - Java: deprecated
  - Vagy a régi változat átírásával
    - Az egyből az új változatot hívja meg (hívásátirányítás)
    - Esetleges figyelmeztető üzenet kiírása mellett

# Osztályok mérete

Egy osztály méretét a következő módokon lehet meghatározni:

- Sima (teljes) méret = szolgáltatások száma (öröklött + itt bevezetett).
- Közvetlen méret = közvetlen (nem absztrakt, illetve újradefiniált) szolgáltatások száma.
- Növekményes méret = közvetlen és újradefiniált szolgáltatások száma.

Másik szempont lehet, hogy tartalmazza-e a nem exportált jellemzőket:

- Külső méret
- Belső méret

# Osztályok mérete

A teljes méretre nyilván nincs felső határ

- A közvetlen, és a növekményes méret az, ami szerepel az osztály szövegében.
- De egy adott méreten felül az osztály kezelhetetlen.

Két megközelítési lehetőség

- Minimalista nézőpont
  - egy programkönyvtár osztálya a megvalósított típusnak csak a legalapvetőbb (atomi) szolgáltatásait tartalmazza,
  - redundáns (azaz ami atomi szolgáltatások használatával is megvalósítható) metódusokat pedig nem.
  - Ez persze a programkönyvtár írójának kedvez, hiszen kevesebb munkát és egyszerűbb karbantartást jelent.

# Osztályok mérete

- Bevásárlólista nézőpont: minden olyan szolgáltatást vegyünk bele az osztályunkba, ami az alábbi szempontoknak megfelel:
  - A szolgáltatás beleillik az absztrakt adattípus megvalósítási sémájába.
  - Nem rontja el az osztály helyességét, azaz az osztályinvariánst.
  - Hasznos funkcionalitást valósít meg.
  - Nem duplikál valamely már meglévő szolgáltatást.
  - Megfelel a könnyen kezelhetőség kritériumainak (lásd később).

# Osztályok mérete

A gyakorlatban általában

- a programozási nyelvtől elvárt szempont a minimalista nézőpont
  - azaz hogy műveletet csak minél kevesebb módon lehessen hatékonyan elvégezni,
- míg a programkönyvtárak tervezésekor a bevásárlólista nézőpont érvényesül
  - a felhasználás megkönnyítése érdekében

# Szolgáltatások száma

Túl sok szolgáltatás a programkönyvtár használatának megtanulhatóságát is kedvezőtlenül befolyásolja

Ezért

- Minden szolgáltatás egy jól meghatározott szerződéssel definiált
  - csak a specifikációjuk, nem pedig az implementációjuk alapján kerülnek felhasználásra.
- Az osztálydokumentáció egységes és könnyen áttekinthető formátumú legyen
  - feltüntetve minden szolgáltatás szerződését és elrejtve az implementációs részleteket
- A szolgáltatások nevei szigorú elnevezési konvenciót kövessenek.
- A szolgáltatásokat csoportosítsuk pontosan definiált kategóriák szerint, ezen kategóriák minden osztálynál egyezzenek meg
  - sorrendjük a dokumentációban legyen azonos
  - kategórián belül a szolgáltatások névsorrendben következzenek

# Példaadatok

Általában kijelenthető, hogy 80 szolgáltatásnál többet tartalmazó osztály esetén már meggondolandó, hogy nem lehetne-e az osztályhierarchiát tovább bontani.

# Szolgáltatások mérete

Fontos, hogy a használó pontosan ismerje egy szolgáltatás paramétereinek számát, típusát és sorrendjét.

- Ezért a felhasználás szempontjából egy szolgáltatás méretét
  - Tulajdonképpen a megadandó paraméterek száma
- Ez a méret erősen függ a programkönyvtár feladatától is,
  - Egy grafikus felületet megvalósító, vagy statisztikai számításokat végző osztály metódusai rengeteg argumentumot igényelhetnek.

# Szolgáltatások mérete

Nagyszámú argumentum esetén érdemes azokat két csoportra osztani:

- Paraméter: olyan argumentum, amelynek értékével valamilyen műveletet kell elvégezni.
- Opció: olyan argumentum, amely akár el is hagyható, ugyanis rendelhető hozzá egy alapértelmezett érték.
  - Csak a paraméterek feldolgozását befolyásolja.

Egy szolgáltatás fejlesztése során a paraméterlista lehetőleg ne változzon

- Új opciókat nyugodtan fel lehet venni.

# Szolgáltatások mérete

A szolgáltatás méretének csökkentése érdekében csak paramétereket vegyünk fel az argumentumlistába.

Az opciókat pedig külön attribútummal reprezentáljuk

- Adjunk hozzá új beállító/lekérdező szolgáltatásokat.
  - Opció beállítási technika.
- Növeli az osztály méretét
  - Esetleges új, globális opció-attribútum

# Példaadatok

Általában kijelenthető, hogy az a jó, ha a szolgáltatások átlagos paraméterszáma 0.5

- 5-nél több paraméter esetén meggondolandó a továbbontás

A szolgáltatások lekérdezés – parancs javasolt aránya

- 60 - 40%

# Elnevezés

Egy programkönyvtár használatát nagyban elősegíti az egységes elnevezési konvenció.

- Ez konzisztens névhasználatot jelent
- Kialakítása a megvalósított adattípusok közös tulajdonságain alapul

Például

- egy adott elem elérésére ( get )
- egy elem megváltoztatására ( put )
- egy elem törlésére ( remove )

Azonos neveket használunk különböző osztályokon belül.

- Mégsem vezet félreértéshez
  - a típusok különbözősége miatt (általában) a szignatúrák nem egyeznek meg

# Elnevezés

Standard elnevezések:

- `capacity` - adott struktúra kapacitását adja vissza.
- `count` - adott struktúra elemszámát adja vissza.
- `empty` - megadja, hogy az adott adatstruktúra üres-e.
- `full` - megadja, hogy az adott adatstruktúra tele van-e.
- `get` - struktúra adott elemét adja vissza.
- `has` - lekérdezi, hogy adott elem a struktúra eleme-e.
- `put` - struktúra adott elemét állítja be.
- `remove` - struktúra adott elemét kitörli.

# Elnevezés

Irányelvek:

- A név legyen rövid (rendszerint egy szó), de beszédes.
- Szolgáltatás neve ne tartalmazzon utalást a szolgáltatást tartalmazó osztályra
  - kivéve többszörös öröklődés esetén, ha átnevezés kell
- Osztály neve mindig főnév(i szerkezet) legyen.
- Parancsok (eljárások) nevei legyenek (felszólító módú) igék
  - Esetleges főnévi vagy értelmező jelzői kiegészítővel
- Nem logikai lekérdezések nevei legyenek főnevek vagy főnévi szerkezetek.

# Elnevezés

- Logikai lekérdezés neve legyen olyan melléknév, amely igen/nem választ sugall
  - Vagy használjuk az is_ (-e) szerkezetet
- A szolgáltatás nevét a célobjektum szemszögéből lehessen értelmezni
  - A szolgáltatás hívásakor mindig létezik egy objektum (a célobjektum) aminek a szolgáltatásáról szó van.
  - has
- Összetartozó művelet és lekérdezés párok elnevezéseinek szótöve legyen azonos
  - extendible - extend

# Osztályok fajtái

Konkrét típusok

- Minden programkönyvtár hierarchiában találunk olyan típusokat, melyek alapvető adatstruktúrákat reprezentálnak
- lista, vektor, string, dátum, komplex szám, …

# Osztályok fajtái

Konkrét típusok – tulajdonságok

- Szoros illeszkedés egy konkrét fogalomhoz és a megvalósítás módjához.
- Érthetőség, önálló használhatóság.
- Erősen implementációfüggők
  - Ezért hatékony a megvalósításuk
  - Bármilyen módosításukkor minden felhasználói kódot újra kell fordítani.
- Minimális mértékben függenek más osztályoktól.
- Különállóan (izoláltan) is fordíthatóak és használhatóa
- Nem, illetve ritkán lehet tőlük örökölni.

# Osztályok fajtái

## Absztrakt típusok

- Szerepük rendszerint egy adott specifikáció (akár interfész is lehetne) bevezetése. Pl. halmaz
- Az absztrakt típusokat konkrét típusok segítségével lehet implementálni.

## Absztrakt típusok – tulajdonságok

- Azonos elérési módhoz többfajta implementáció is létezhet.
- Hatékony tárkihasználás és elfogadható futási idő a virtuális metódusok segítségével.
- Az implementáció minimális mértékben függ más osztályoktól.
- Különállóan is lefordíthatóak – érthetőek önmagukban.

# Osztályok fajtái

## Csomópont típusok

- Az öröklési hierarchia belső pontjait alkotják.

## Jellemzőik

- Az ősosztályok szolgáltatásait implementálja,
  - Interfészét kibővíti virtuális metódusokkal is, melyeket maga is implementál.
- Függ az őseitől.
- Lehet belőle örökölni.
- Példányosítható.

# Osztályok fajtái

## Felületosztályok

- Egy felület igazítása úgy, hogy jobban illeszkedjen a felhasználó elvárásaihoz.
  - Például vektor ne 0-tól legyen indexelve.
- Ellenőrzött vagy korlátozott felületek biztosítása.

# Osztályok fajtái

„Kövér interfészek”

- Ez egy olyan típus, amely nem függ más osztályoktól, rengeteg szolgáltatást vezet be
  - ezek közül csak a legalapvetőbbekhez ad implementációt, míg a speciálisabb szolgáltatásokat virtuálisként deklarálja
  - Lehet példányosítani, ad hozzá egy üres implementációt, ami rendszerint egy hibaüzenet kiírását jelenti
    - Jelzi, hogy az adott szolgáltatás nem implementált.
    - Példa: általános tároló (container) osztály.

# Osztályok fajtái

Keret osztályok (Framework)

- Ezen absztrakt osztályok tulajdonképp egy mini alkalmazást valósítanak meg.
- Maga a programlogika van csak az osztályon belül implementálva
  - minden egyéb paraméter-bekérő és alapműveletet elvégző metódus virtuális
- Példa: szűrő osztály, amely egy bemeneti adatfolyam minden elemén végiglépkedve adott feltétel teljesülésekor bizonyos műveleteket elvégez.
- A bemeneti adatfolyamon történő végigléptetés, adott tulajdonság fennállásának ellenőriztetése,
  - A kívánt művelet meghívása és esetleges hibakezelés ezen osztályban van implementálva.

# Osztályok fajtái

## Leíró osztályok

- Felmerülhet az igény adott interfészen keresztül változó méretű osztályok objektumainak kezelésére is.
  - Erre ad megoldást a leíró osztályok használata.
- Ekkor ugyanis az implementáció két részre bomlik
  - az aktuális reprezentációra
  - a reprezentáció elérését biztosító leíró (handle) objektumra
  - Ez tulajdonképp a mutató típus objektumorientált megvalósításának tekinthető.
- Példa: fájlleíró osztályok

# Memóriakezelés

A programkönyvtár tervezés egyik kulcsfontosságú kérdése.

Két alapvető probléma:

- Memóriafoglalás új objektumok létrehozásakor.
- Memória felszabadítása.

Míg az első az operációs rendszer feladata, addig a második problémára két megoldási irányzat is született:

- Automatikus szemétgyűjtés (pl. Java).
  - Ez rendszerint referenciák bevezetését és a mutató típus megszüntetését jelenti.
- Objektumok kézzel történő felszabadítása.
  - Hatékonyabb, de sokkal veszélyesebb.

# Algoritmus, grammatika, fordítóprogramok

JÖVŐ HÉTEN




# Programozási nyelvek és módszerek

# Algoritmus

# Algoritmus

Abu Abdallah Muḥammad ibn Mūsā al-Khwārizmī

- 780 – ?845
- arabul alkotó perzsa tudós
- matematikus

Kidolgozta a matematikai algoritmusok fogalmát

- néhányan a számítástechnika nagyapjának

Az algoritmus szó a nevének félrefordított latin változatából ered



# Algoritmus

„Az algoritmus egyértelműen előírt módon  és sorrendben végrehajtandó tevékenységek véges sorozata egy feladat megoldására, ami véges idő alatt befejeződik.”

Hétköznapi életben is: egy feladat megoldási lépései teljes részletességgel

- Feladat: Süss egy gyümölcstortát!
  - Hogy írjuk le a feladatot?
  - Mi az input?
  - Mi az output?
- Itt az algoritmus: a recept

Nem minden feladatra adható algoritmus!

# Algoritmus

Megadható:

- Szövegesen, természetes nyelven
- Pszeudokóddal
- Grafikusan
  - Folyamatábrával
  - Struktogrammal
  - …
- Automatákkal vagy Turing-gép segítségével
- Táblázattal
- Programozási nyelv segítségével

# Algoritmus

Komponensei:

- Szekvencia  - parancsok egymásutánja
- Elágazás – valamilyen feltételtől függően alternatívák választása
- Ciklus – utasítások ismételt végrehajtása

Fontos kérdések

- Biztosan megoldja a feladatot?
  - Helyes – specifikációnak megfelel?
  - Mit jelent az, hogy megoldás?
    - Részletesen még jön!

# Feladat specifikációja

## Specifikáció megadása

- Szövegesen
  - Példa: számítsd ki **a** és **b** legnagyobb közös osztóját!
  - Számos információt „beleértünk”….
  - Néha félreértjük – véletlenül vagy szándékosan
    - „Találkozzunk a Keletinél a metró lépcsőjénél”
      - Pontosabb az, hogy „a Keletinél a metró új kijáratánál”?
    - „A királynőt megölni nem kell félnetek jó lesz”

# Feladat specifikációja

## Specifikáció megadása

- Formálisan
  - $I$ – a lehetséges bemenetek,
    $O$ – a lehetséges kimenetek
    végesen leírható halmaza
  - $F \subseteq I \times O$ reláció, ahol $i \in I$ és $o \in O$ esetén $(i,o) \in F$ azt jelenti, hogy $F$ az $i$ bemenethez hozzárendeli az $o$ kimenetet.
  - Például:
    - $I = N \times N$
    - $O = N$
    - $F = \{((a,b),c)$, ahol $a, b, c \in N \wedge c \mid a \wedge c \mid b \wedge \forall x : x \mid a \wedge x \mid b \rightarrow c \geq x\}$
    - $c \mid a$ jelentése: $a$ osztható $c$-vel.

# Feladat specifikációja

*Szerződés* a felhasználó és az implementáló között

- Előfeltétel: állítás, ami leírja azt a feltételt, ami szükséges a feladat helyes működéséhez
- Utófeltétel: állítás, ami leírja azt a feltételt, amit a függvény teljesít a helyes végrehajtás után

Helyesség a specifikáció figyelembevételével

- A program felhasználója teljesíti az előfeltételt
- A program lefut
- A program végeztével az utófeltétel igaz lesz.

Mit kell az implementációnak teljesíteni, ha a felhasználó megsérti az előfeltételt?

# Feladat specifikációja

Elő- és utófeltételekkel

- Állapottér: $\mathbb{N}\times\mathbb{N}\times\mathbb{N}$
  $a \quad b \quad c$
- Előfeltétel (Ef)
  - $a=a' \wedge b=b'$
- Utófeltétel (Uf):
  - $\text{Ef} \wedge c \mid a \wedge c \mid b \wedge \forall x : x \mid a \wedge x \mid b \rightarrow c \geq x$

# Programozási nyelv

# Mi a programozási nyelv?

Egy jelölés …

Eszköz

- a számítógép vezérlésére
- programozók közötti kommunikációra
- algoritmusok leírására
- magas szintű tervezésre
- a feladat bonyolultságának kezelésére

# Programozási nyelvek kialakulása

Gépi kód

Assembly nyelv

- Mnemonikok, címkék használata
- Assembler

Magas szintű nyelv

- Utasításkészlete független egy adott számítógép-architektúra utasításkészletétől
  - Végrehajtásuk előtt egy fordítóprogramnak kell fordítani.
    - Assembly kód -> assembler.
    - Közvetlenül gépi kód.

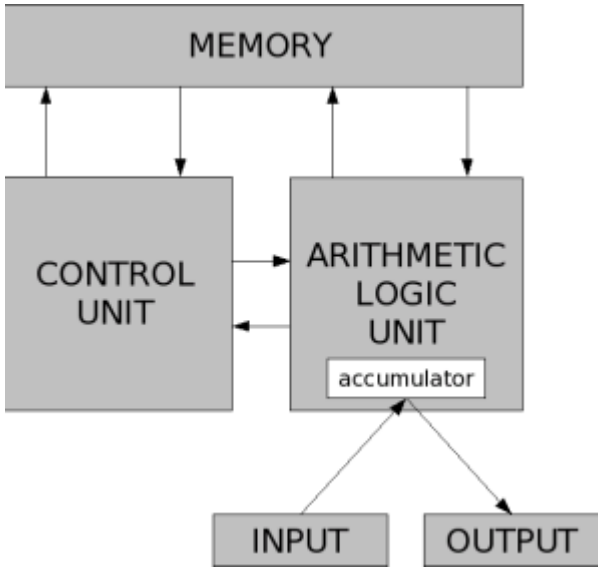
# Neumann János (1903 – 1957)



Neumann–elvek a számítógépről:

- Legyen univerzális
  - bármilyen feladat megoldására használható.
- Legyen soros működésű
  - az utasításokat sorban egymás után hajtsa végre
- Legyen teljesen elektronikus működésű
- Az egyes utasítások és adatok leírásához és feldolgozásához a kettes számrendszert használja.
- Legyen belső memóriája
- Tárolt program elven működjön
  - Egy program indításakor a programot alkotó utasítások és a működéshez szükséges adatok is kerüljenek be a belső memóriába
  - Az utasításokat a memóriából kiolvasva, sorban egymás után hajtsa végre a berendezés.

# Neumann János – 1946




A számítógép a következő egységekből álljon:

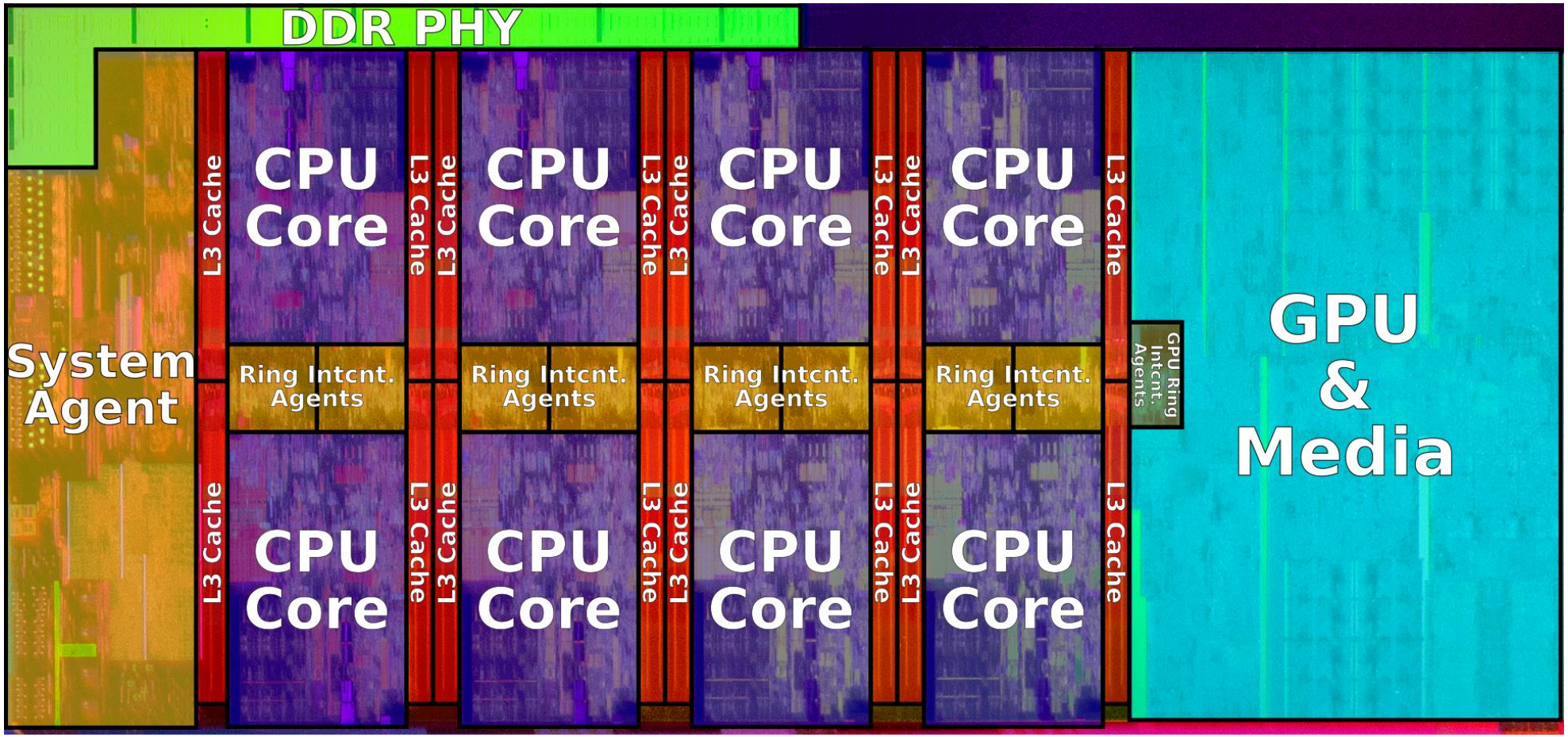
- Központi egység:
  - vezérlés, műveletvégzés, adatmozgatás
- Memória:
  - az adatok és a programok tárolása a program végrehajtása közben
- Bemeneti-kimeneti egység:
  - kommunikáció a külvilággal, az adatbevitel- és kivitel megvalósítása

# Egyszerűsített példa

HTTPS://GODBOLT.ORG/

# CPU – Core i7 9700K



# Program végrehajtása

Gépi kód: CPU / Vezérlő által értelmezett utasítások

```
55
48 89 e5
89 7d ec
89 75 e8
82 7d ec 00
79 03
f7 5d ec
83 7d e8 00
79 03
f7 5d e8
83 7d e8 00
7e 18
8b 45 e8
89 45 fc
8b 45 ec
99
f7 7d e8
89 55 e8
8b 45 fc
89 45 ec
eb e2
8b 45 ec
5d
c3
```

```
55
48 89 e5
b0 0a 00 00 00
bf 22 00 00 00
e8 ae ff ff ff
b8 00 00 00 00
5d
c3
0f 1f 44 00 00
```

Ember számára nehezen olvasható.
Számítógép közvetlenül végre tudja hajtani.

# Alacsonyszintű nyelv

Assemby – Ember számára is olvasható kód.

```
_Z4lnkoii:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    %edi, -20(%rbp)
        movl    %esi, -24(%rbp)
        cmpl    $0, -20(%rbp)
        jns     .L2
        negl    -20(%rbp)
.L2:
        cmpl    $0, -24(%rbp)
        jns     .L5
        negl    -24(%rbp)
.L5:
        cmpl    $0, -24(%rbp)
        jle     .L4
        movl    -24(%rbp), %eax
        movl    %eax, -4(%rbp)
        movl    -20(%rbp), %eax
        cltd
        idivl   -24(%rbp)
        movl    %edx, -24(%rbp)
        movl    -4(%rbp), %eax
        movl    %eax, -20(%rbp)
        jmp     .L5
.L4:
        movl    -20(%rbp), %eax
        popq    %rbp
        ret
```

```
main:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    $10, %esi
        movl    $34, %edi
        call    _Z4lnkoii
        movl    $0, %eax
        popq    %rbp
        ret
```

Ember számára olvashatóbb.
Közvetlen fordítható gépi kódra.

# Magasszintű nyelv

C++ – Ember számára olvasható kód.

```
int lnko(int a, int b) {
   int tmp;
   if (a<0) a= -a;
   if (b<0) b= -b;
   while(b > 0) {
      tmp = b;
      b = a % b;
      a = tmp;
   }
   return a;
}
```

```
int main()
{
    lnko(34, 10);
    return 0;
}
```

Ezt szoktuk meg.
A nyelv szabályait definiálni kell.
Fordítóprogramok kellenek.

# Modell

HTTP://WWW.PETERHIGGINSON.CO.UK/LMC/

# A „Little Man Computer”

Neumann elvek az oktatásban

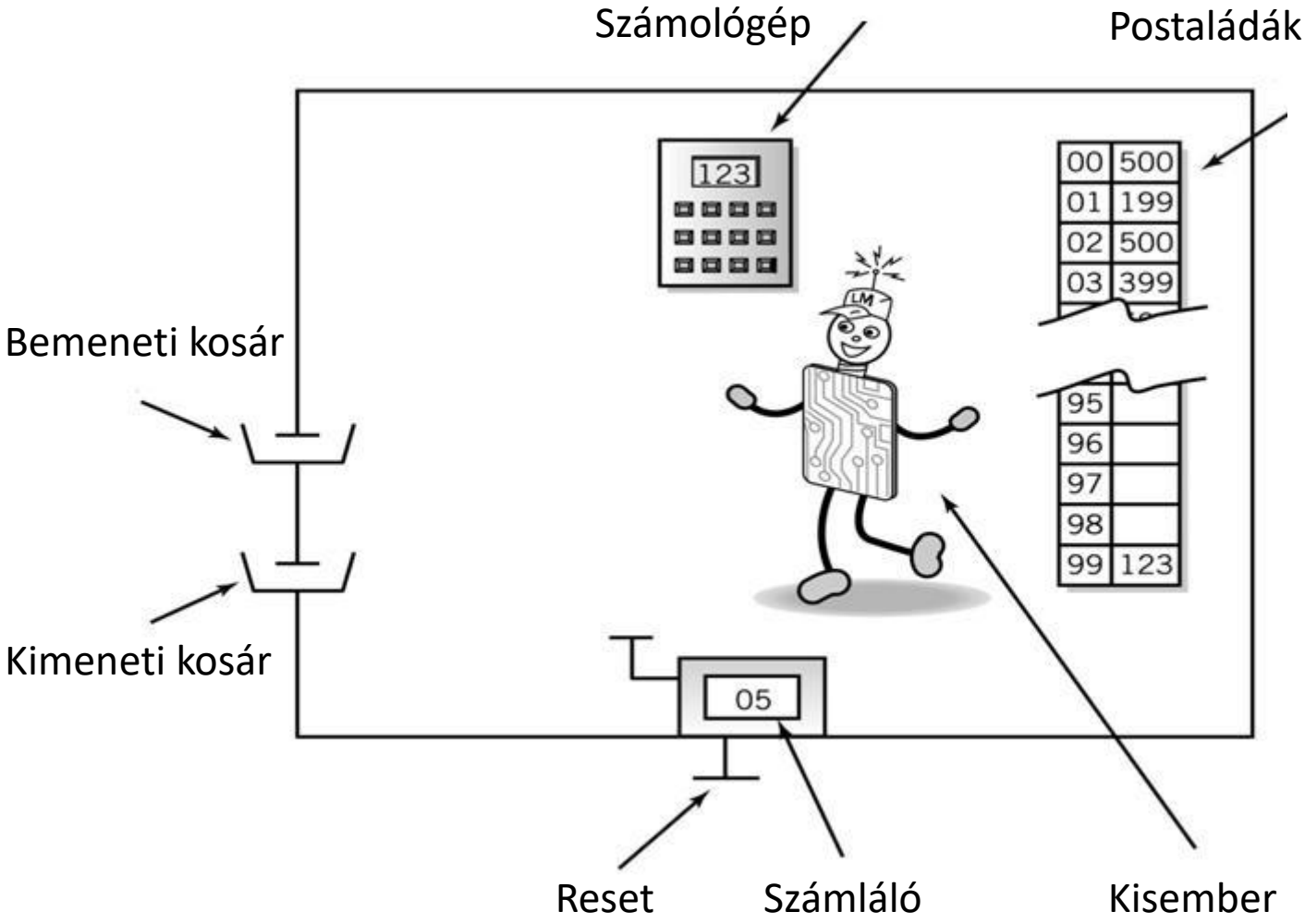
Stuart Madnick, MIT

Szoba, benne:

- Kisember
- 100 postaláda (00-99)
- Számláló (2)
- Bemenő és kimenő kosár
- Számológép

Nem foglalkozunk azzal, hogyan

- töltjük fel a postaládákat
- tegyünk adatot a bemenő kosárba



# LMC elemei

Bemeneti és kimeneti kosár:

- Kisember kap/beletesz egy 3 jegyű számot tartalmazó cédulát

Számláló:

- 0 és 99 között tud számolni
- Pedál megnyomására a számlálóban tárolt érték eggyel megnő
- Értékét nullára állíthatjuk egy külső „reset” gombbal

Számológép:

- Háromjegyű decimális számokat tud kezelni.
- Képes
  - Kivonni
  - Összeadni
  - A begépelt vagy a számítás eredményeként kapott értéket eltárolni.

# Postaládák

A postaládákban számokat tárolunk

- elérésük a postaláda címe (sorszáma) alapján

A címek folyamatosan egymás után jövő értékek

- 00 .. 99

A tartalom lehet

- Adat vagy
- Utasítás

| Cím | Tartalom |
|---|---|
| 00 | 522 |
| 01 | 123 |

# Utasítás esetén

Műveleti kód

Operandus

- A művelet operandusa: adat vagy az adat címe

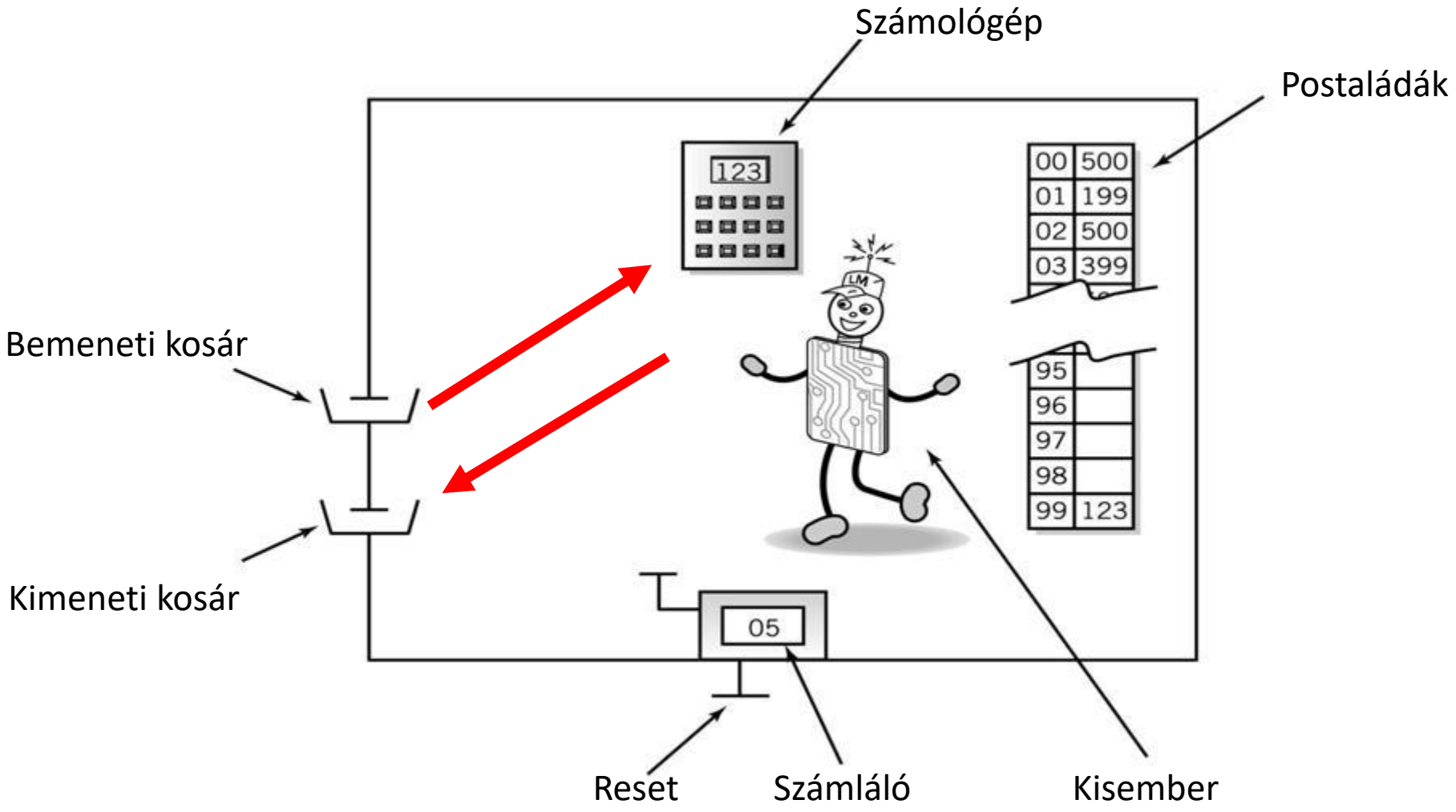
| Cím | Tartalom | |
|---|---|---|
| | Műv. kód | Operandus |
| | | |

# Utasításkészlet

Lehetséges utasításkészlet:

| | | |
|---|---|---|
| Aritmetikai | 1xx | Összeadás |
| | 2xx | Kivonás |
| Adat mozgató | 3xx | Tárolás |
| | 5xx | Betöltés |
| Input/Output | 901 | Beolvasás |
| | 902 | Kiírás |
| Vezérlés | 000 | Leállás |

# Input/Output

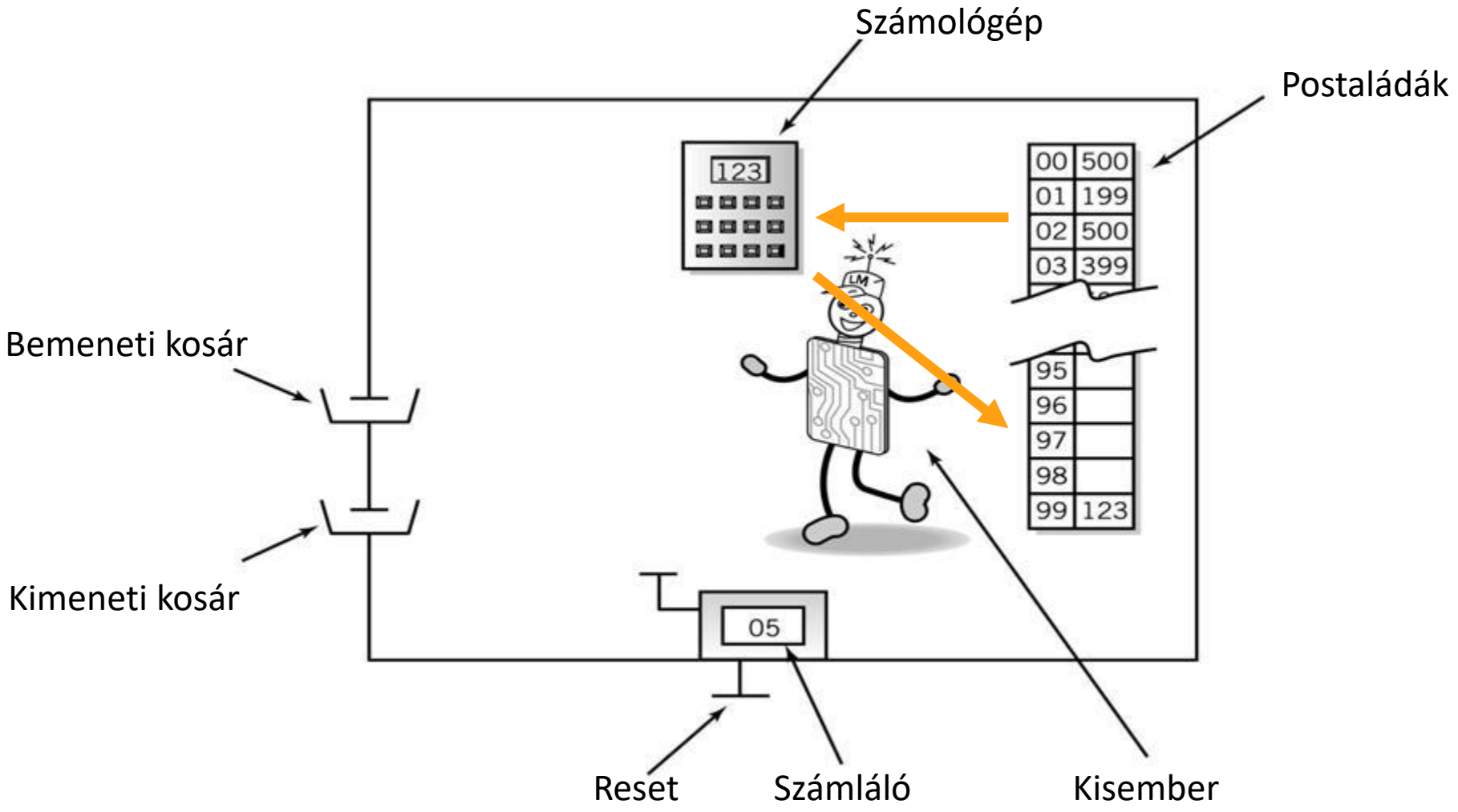


# Input/Output

Adatok mozgatása a számológép és a bemeneti/kimeneti kosarak között

| | Tartalom | |
|---|---|---|
| | Műveleti kód | Operandus (cím) |
| Beolvasás | 9 | 01 |
| Kiírás | 9 | 02 |

# LMC belső adatmozgatás



# Belső adatmozgatás

A postaláda és a számológép között

| | Tartalom | |
|---|---|---|
| | Műveleti kód | Operandus (cím) |
| Tárolás | 3 | xx |
| Betöltés | 5 | xx |

# Adatok tárolása

Az utasításokat és az adatokat tároló postaláda fiókok fizikailag azonosak

- Ezt valós esetekben elkülönítve tároljuk a memóriában

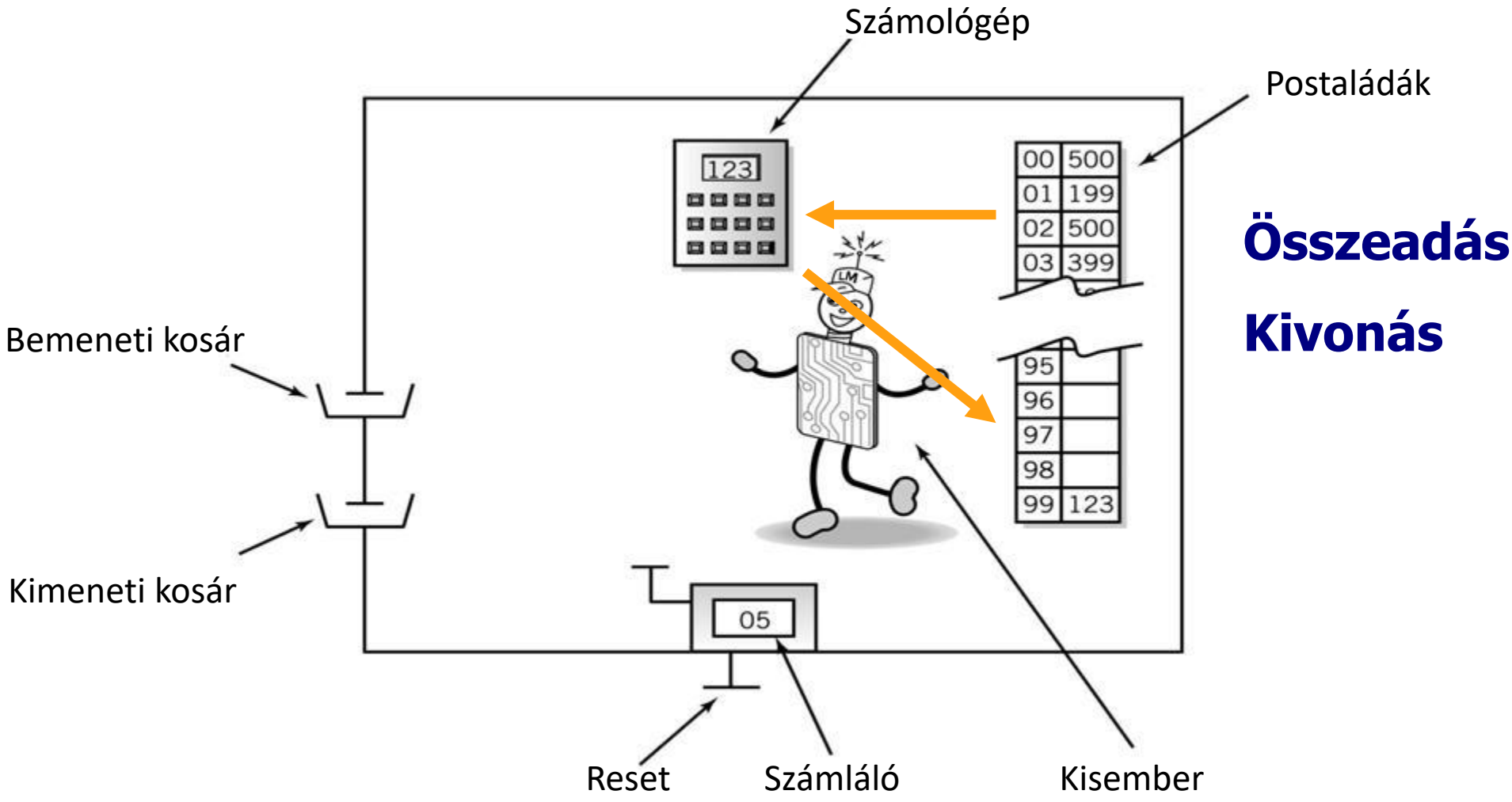
# Aritmetikai utasítások

Olvassuk el az operandus által meghatározott levelesládában lévő értéket

Hajtsuk végre a műveletet a számológéppel

| | Tartalom | |
|---|---|---|
| | Műveleti kód | Operandus (cím) |
| Összeadás | 1 | xx |
| Kivonás | 2 | xx |

# LMC Aritmetikai utasítások



# Utasítás-végrehajtási ciklus

Az utasítások végrehajtása

- Kikeresés (Fetch)
  - Kisember kikeresi, hogy melyik utasítást kell végrehajtani
- Értelmezi az utasítást (Decode)
  - Az utasítás kódja alapján eldönti, hogy mit kell pontosan csinálni
- Végrehajtás (Execute)
  - Kisember elvégzi a munkát.

A számítógép által egy-egy utasítás végrehajtásakor elvégzett tevékenység-sorozatot utasítás-végrehajtási ciklusnak nevezzük

- Mivel ezek ciklikusan ismétlődnek.

# Kikeresés és dekódolás

1. Kisember kiolvassa az utasításszámlálóból annak a postaládának a sorszámát (címét), ami az utasítást tartalmazza
2. Átsétál ahhoz a postaládához, ami megfelel a számláló értékének.
3. Kiveszi belőle a cédulát, elolvassa és dekódolja a számot, ami rajta van
   - A cetlit visszateszi, arra az esetre, ha esetleg később szükség lenne rá, hogy újra elolvassa

# Végrehajtás

1. Kisember odamegy a postaládához, aminek a sorszámát az éppen kikeresett utasítás tartalmazza.
2. Kiveszi belőle a cédulát és elolvassa a rajta lévő számot
   - nem felejti el a cédulát visszatenni, mert később is szükség lehet rá
3. A kiolvasott utasítás kódjának megfelelően elvégzi a teendőt az adattal
4. Odasétál a számlálóhoz és továbblöki, így folytathatja a következő utasítás kikeresésével.

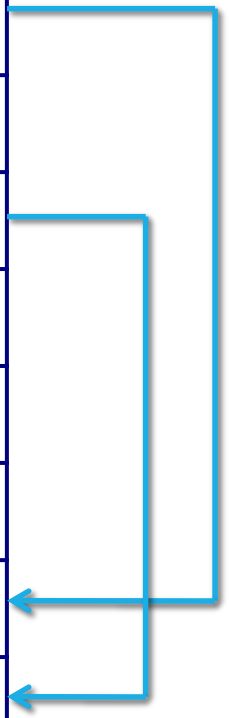
# Példaprogram

Adjuk meg két szám különbségét

- Olvassuk be az első számot
- Tároljuk el
- Olvassuk be a második számot
- Tároljuk el
- Vonjuk ki belőle az elsőt
- Írjuk ki az eredményt
- Fejezzük be a munkát

# LMC gépi kód

| Postaláda | Kód |
|---|---|
| 00 | 901 |
| 01 | 307 |
| 02 | 901 |
| 03 | 308 |
| 04 | 207 |
| 05 | 902 |
| 06 | 000 |
| 07 | 000 |
| 08 | 000 |



# LMC gépi kód

| Postaláda | Kód | Utasítás leírása |
|---|---|---|
| 00 | 901 | Első szám beolvasása |
| 01 | 307 | Szám tárolása |
| 02 | 901 | Második szám beolvasása |
| 03 | 308 | Szám tárolása |
| 04 | 207 | Az első kivonása a 2.-ból |
| 05 | 902 | Eredmény kiírása |
| 06 | 000 | Stop |
| 07 | 000 | Adatrekesz |
| 08 | 000 | Adatrekesz |

# Számítsuk ki két szám különbségének abszolút értékét!

1. Olvassuk be az első számot
2. Tároljuk el
3. Olvassuk be a másik számot
4. Tároljuk el
5. Vonjuk ki belőle az elsőt
6. Ha pozitív az eredmény, folytassuk a 9.–nél
7. (Különben) töltsük be az első számot
8. Vonjuk ki belőle a másodikat
9. Írjuk ki az eredményt
10. Fejezzük be a munkát

# Program vezérlés

## Elágazás utasítások

- Az utasítások egymásutánjából való kiugrás lehetősége
- Megváltoztatja a címet az utasítás számlálóban

## Leállás

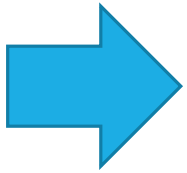
| | Tartalom | |
|---|---|---|
| | Műv. kód | Operandus (cím) |
| Ugrás az xx számú postafiókra | 6 | xx |
| Ha a számológép tartalma 0, ugrás az xx számú postafiókra | 7 | xx |
| Ha a számológép tartalma >0, ugrás az xx számú postafiókra | 8 | xx |

# LMC utasításkészlete

| | |
|---|---|
| Hozzáadás | 1xx |
| Kivonás | 2xx |
| Tárolás | 3xx |
| Betöltés | 5xx |
| Ugrás | 6xx |
| Ugrás 0-nál | 7xx |
| Ugrás pozitívnál | 8xx |
| Input | 901 |
| Output | 902 |
| Stop | 000 |

# Számítsuk ki két szám különbségének abszolút értékét!

| Postaláda | Kód |
|---|---|
| 00 | 901 |
| 01 | 307 |
| 02 | 901 |
| 03 | 308 |
| 04 | 207 |
| 05 | 902 |
| 06 | 000 |
| 07 | 000 |
| 08 | 000 |



| Postaláda | Kód |
|---|---|
| 00 | 901 |
| 01 | 310 |
| 02 | 901 |
| 03 | 311 |
| 04 | 210 |
| 05 | 808 |
| 06 | 510 |
| 07 | 211 |
| 08 | 902 |
| 09 | 000 |
| 10 | 000 |
| 11 | 000 |

# Számítsuk ki két szám különbségének abszolút értékét!

| | | |
|---|---|---|
| 00 | 901 | Első szám beolvasása |
| 01 | 310 | Szám tárolása |
| 02 | 901 | Második szám beolvasása |
| 03 | 311 | Szám tárolása |
| 04 | 210 | Első számot kivonjuk |
| 05 | 808 | Ha pozitív, ugorj a 08-ra (a kiírásra) |
| 06 | 510 | (különben) töltsd be az elsőt |
| 07 | 211 | Vond ki belőle a másodikat |
| 08 | 902 | Írd ki az eredményt |
| 09 | 000 | Stop |
| 10 | 000 | Az  első adat helye |
| 11 | 000 | A második adat helye |

# Nevezzük el az utasításkódokat!

| | | |
|---|---|---|
| Hozzáadás | 1xx | ADD xx |
| Kivonás | 2xx | SUB xx |
| Tárolás | 3xx | STO xx |
| Betöltés | 5xx | LDA xx |
| Ugrás | 6xx | BR xx |
| Ugrás 0-nál | 7xx | BRZ xx |
| Ugrás pozitívnál | 8xx | BRP xx |
| Input | 901 | INP |
| Output | 902 | OUT |
| Stop (Halt) | 000 | HLT |
| Adattároló | | DAT |

# Számítsuk ki két szám különbségének abszolút értékét!

| | | | |
|---|---|---|---|
| 00 | INP | 901 | Első szám beolvasása |
| 01 | STO 10 | 310 | Szám tárolása |
| 02 | INP | 901 | Második szám beolvasása |
| 03 | STO 11 | 311 | Szám tárolása |
| 04 | SUB 10 | 210 | Első számot kivonjuk |
| 05 | BRP 08 | 808 | Ha pozitív, ugorj a 08-ra (a kiírásra) |
| 06 | LDA 10 | 510 | (különben) töltsd be az elsőt |
| 07 | SUB 11 | 211 | Vond ki belőle a másodikat |
| 08 | OUT | 902 | Írd ki az eredményt |
| 09 | HLT | 000 | Stop |
| 10 | DAT 00 | 000 | Az  első adat helye |
| 11 | DAT 00 | 000 | A második adat helye |

# Adjunk nevet a változóknak és az utasításoknak!

| | | | | |
|---|---|---|---|---|
| | 00 | INP | 901 | Első szám beolvasása |
| | 01 | STO ELS | 310 | Szám tárolása |
| | 02 | INP | 901 | Második szám beolvasása |
| | 03 | STO MAS | 311 | Szám tárolása |
| | 04 | SUB ELS | 210 | Első számot kivonjuk |
| | 05 | BRP KI | 808 | Ha pozitív, ugorj a 08-ra (a kiírásra) |
| | 06 | LDA ELS | 510 | (különben) töltsd be az elsőt |
| | 07 | SUB MAS | 211 | Vond ki belőle a másodikat |
| KI | 08 | OUT | 902 | Írd ki az eredményt |
| | 09 | HLT | 000 | Stop |
| ELS | 10 | DAT 00 | 000 | Az első adat helye |
| MAS | 11 | DAT 00 | 000 | A második adat helye |

# Példa

Mit csinál?

```
     INP
     STO ELS
     INP
     ADD ELS
     OUT
     HLT
ELS  DAT 00
```

# Másik példa

És ez mit csinál?

```
      INP
LOOP  SUB  ONE
      OUT
      BRZ  QUIT
      BR   LOOP
QUIT  HLT
ONE   DAT  1
```

# Assembly nyelv

Jellemzői:

- CPU specifikus
- 1 - 1 megfelelés az assembly nyelvű utasítás és a gépi nyelvű utasítás között
  - makró assembly később alakult ki
- Mnemonikok reprezentálják az utasításokat
- Változók deklarálhatók
- Utasítások címkézhetők

Assembler lefordítja

- inputja az assembly nyelvű program
- outputja a gépi nyelvű

# Végeztünk?

# Assembly

Láthattuk, hogy bár az assembly mindenre jó, mégsem célszerű.

Szembetűnően hiányzik:

- Átláthatóság
  - Absztrakció hiánya
- Hordozhatóság
  - CPU, környezet specifikus
- Újrafelhasználhatóság
  - Magasabb szintű, nemcsak hívható alprogramok
- …

# Magasabb szintű nyelvek



# Magasabb szintű nyelvek

Láttuk, hogy a gépi kód egyértelmű, utasításokat adunk a CPU-nak

- Az Assembly ennek az olvasmányosabb változata
  - További funkciókkal
- A magasabb szintű nyelvek nyelvi elemei nem feleltethetők meg egy-egy módon ennek, fordítás vagy értelmezés szükséges
- **Szabályok kellenek, amik leírják az adott nyelvet:**
  - Mik a nyelvi elemek
  - Hogyan kell használni
  - Mi az értelmezése ezeknek

Szintaktika és szemantika

# Avram Noam Chomsky (1928-)



Amerikai nyelvész,

A generatív nyelvtan elméletének megalkotója

# Chomsky – Generatív grammatikák

Chomsky nyelvészként a természetes nyelveket vizsgálta

- Leírást adni a természetes nyelvek szabályaira
- Számos elmélet, cikk szerzője

Alapgondolat tömören

- Gyerek – véges számú mondatot hall
  - Képes tetszőleges nyelvtanilag helyes mondat megalkotására
  - Kell legyen egy szabályrendszer minden nyelvben!

A mesterséges nyelvek esetén nagyobb a siker …

# Generatív grammatika – Példa

Egyszerű példa, illusztrációra – kis nyelvtani emlékeztető.

- „ A haszontalan kisfiú ledobta a tányért.”

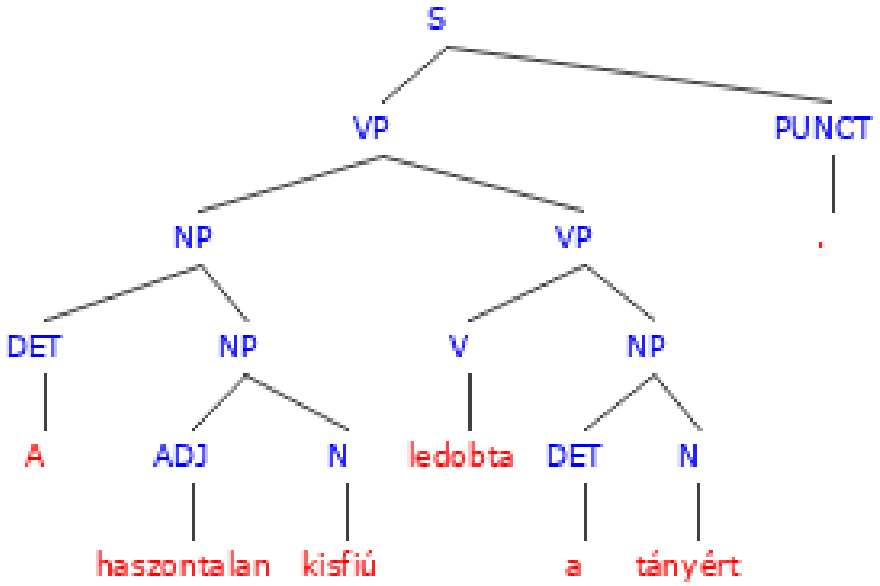
Ez egy mondat, ami felbontható

- Van alanyi és állítmányi részre
  - Az alanyi rész névelő, jelző és alany sorozata lehet
  - Az állítmányi rész az állítmány és a tárgy egymásutánja, stb.

A struktúra leírásakor „grammatikai jeleket” is használhatunk a mondatrészek, illetve szófajok jelzésére

- ezeket < > zárójelek közé tesszük

# Elemzési fa




| | |
|---|---|
| ADJ | melléknév (adjective) |
| DET | névelő (determiner) |
| N | főnév (noun) |
| NP | névszói csoport (noun phrase) |
| PRON | névmás (pronoun) |
| PUNCT | írásjel (punctuation) |
| S | mondat (sentence) |
| V | ige (verb) |
| VP | igei csoport (verbal phrase) |

https://www.webforditas.hu/elemzo

# Példa egyszerűsített szabályokra

| | |
|---|---|
| < S > | → < VP > < PUNCT > |
| < VP > | → < VP > < NP > |
| < VP > | → < V > < NP > |
| < NP > | → < DET > < NP> |
| < NP > | → < DET > < N > |
| < NP > | → < ADJ > < N > |
| | |
| < DET > | → a |
| < ADJ > | → haszontalan |
| < N > | → kisfiú |
| < N > | → tányér(t) |
| < V > | → ledobta |
| < PUNCT > | → . |

| | |
|---|---|
| ADJ | melléknév (adjective) |
| DET | névelő (determiner) |
| N | főnév (noun) |
| NP | névszói csoport (noun phrase) |
| PRON | névmás (pronoun) |
| PUNCT | írásjel (punctuation) |
| S | mondat (sentence) |
| V | ige (verb) |
| VP | igei csoport (verbal phrase) |

# Legbaloldalibb levezetés

Ha adott a fenti módon egy szabályhalmaz, akkor azt használhatjuk mondatok képzésére a következő módon:

- A < S > elnevezésű bal oldalon lévő grammatikai elemből kiindulva behelyettesíthetjük a szabály jobb oldalán álló jelsorozatot
- Majd ennek első elemét vesszük sorra, s írjuk át olyan szabály felhasználásával, ahol megfelelő részt bal oldalon találjuk.
- Ezt az eljárást folytatjuk, amíg még találunk mondatrészt a levezetett jelsorozatban, azaz, amíg át nem térünk a grammatikai egységekről a nyelv szavaira.
- A levezetés során az egyes grammatikai elemeket tetszés szerinti sorrendben helyettesíthetjük.

A következőkben a fenti grammatikai szabályok segítségével oly módon jutunk el az előző példamondatunkhoz, hogy a levezetés során mindig a lehetséges legbaloldalibb helyettesítést választjuk.

# Legbaloldalibb levezetés

< S > → < VP > < PUNCT >

< VP > < PUNCT > → < NP > < VP > < PUNCT >

< NP > < VP > < PUNCT > → < DET > < NP > < VP > < PUNCT >

< DET > < NP > < VP > < PUNCT > → A < NP > < VP > < PUNCT >

A < NP > < VP > < PUNCT > → A < ADJ > < N > < VP > < PUNCT >

A < ADJ > < N > < VP > < PUNCT > → A haszontalan < N > < VP > < PUNCT >

A haszontalan < N > < VP > < PUNCT > → A haszontalan kisfiú < VP > < PUNCT >

A haszontalan kisfiú < VP > < PUNCT > → A haszontalan kisfiú < V > < NP > < PUNCT >

A haszontalan kisfiú < V > < NP > < PUNCT > → A haszontalan kisfiú ledobta < NP > < PUNCT >

A haszontalan kisfiú ledobta < NP > < PUNCT > → A haszontalan kisfiú ledobta < DET > < N > < PUNCT >

A haszontalan kisfiú ledobta < DET > < N > < PUNCT > → A haszontalan kisfiú ledobta a < N > < PUNCT >

A haszontalan kisfiú ledobta a < N > < PUNCT > → A haszontalan kisfiú ledobta a tányért < PUNCT >

A haszontalan kisfiú ledobta a tányért < PUNCT > → A haszontalan kisfiú ledobta a tányért .

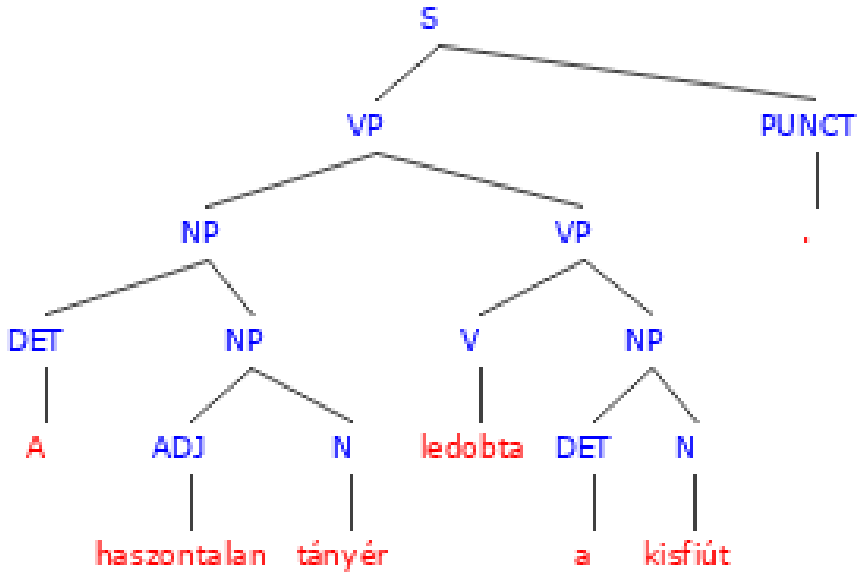
# Legbaloldalibb levezetés

A szabályok szerint lehetséges még:

- A haszontalan tányér ledobta a kisfiút.
- A haszontalan kisfiú ledobta a kisfiút.
- A haszontalan tányér ledobta a tányért.

Nem triviális.

- A továbbiakban szorítkozzunk a mesterséges nyelvekre.



# Mesterséges nyelvek

Magas szintű programozási nyelv esetén szükség van a forrásprogram „fordítására”, vagy „értelmezésére”

- Assemblyre
- Gépi kódra

Ehhez szükséges egy

- Fordítóprogram (compiler), vagy
- Értelmezőprogram (interpreter)

# A programok végrehajtása

Az interpreter egy utasítás lefordítása után azonnal végrehajtja azt

A fordítóprogram átalakítja a programot egy vele ekvivalens formára

- ez lehet a számítógép által (majdnem) közvetlenül végrehajtható forma, vagy
- lehet egy másik programozási nyelv

# Fordítás

A lefordítandó program a forrásprogram.

- Source program / sorce code

A fordítás eredményeként kapott program a tárgyprogram.

- Object program / object code

Az az idő, amikor az utasítások fordítása folyik, a fordítási idő.

- Compilation time

Az az idő, amikor az utasítások végrehajtása folyik, a végrehajtási idő.

- Execution time / runtime

# Szabályok

Szabályok kellenek, amik leírják

- Mik a nyelvben használható jelek (karakterek)
- Milyen nyelvi elemek vannak (formailag helyes karaktersorozatok)
- Jelentés szempontjából mik a helyes karaktersorozatok

A forráskódot író és az azt fordítóprogram ugyanazt kell, hogy értse a leírt kódon

- A gép a szabályok szerint értelmezett parancsainkat teljesíti, nem a kívánságainkat.

# Általános fogalmak

A nyelv szintaxisa azoknak a szabályoknak az összessége, amelyek az adott nyelven írható összes lehetséges, formailag helyes programot (jelsorozatot) definiálják.

- Reguláris kifejezések, BNF forma

Az adott nyelv programjainak jelentését leíró szabályok összessége a nyelv szemantikája.

# Szintaxis és szemantika

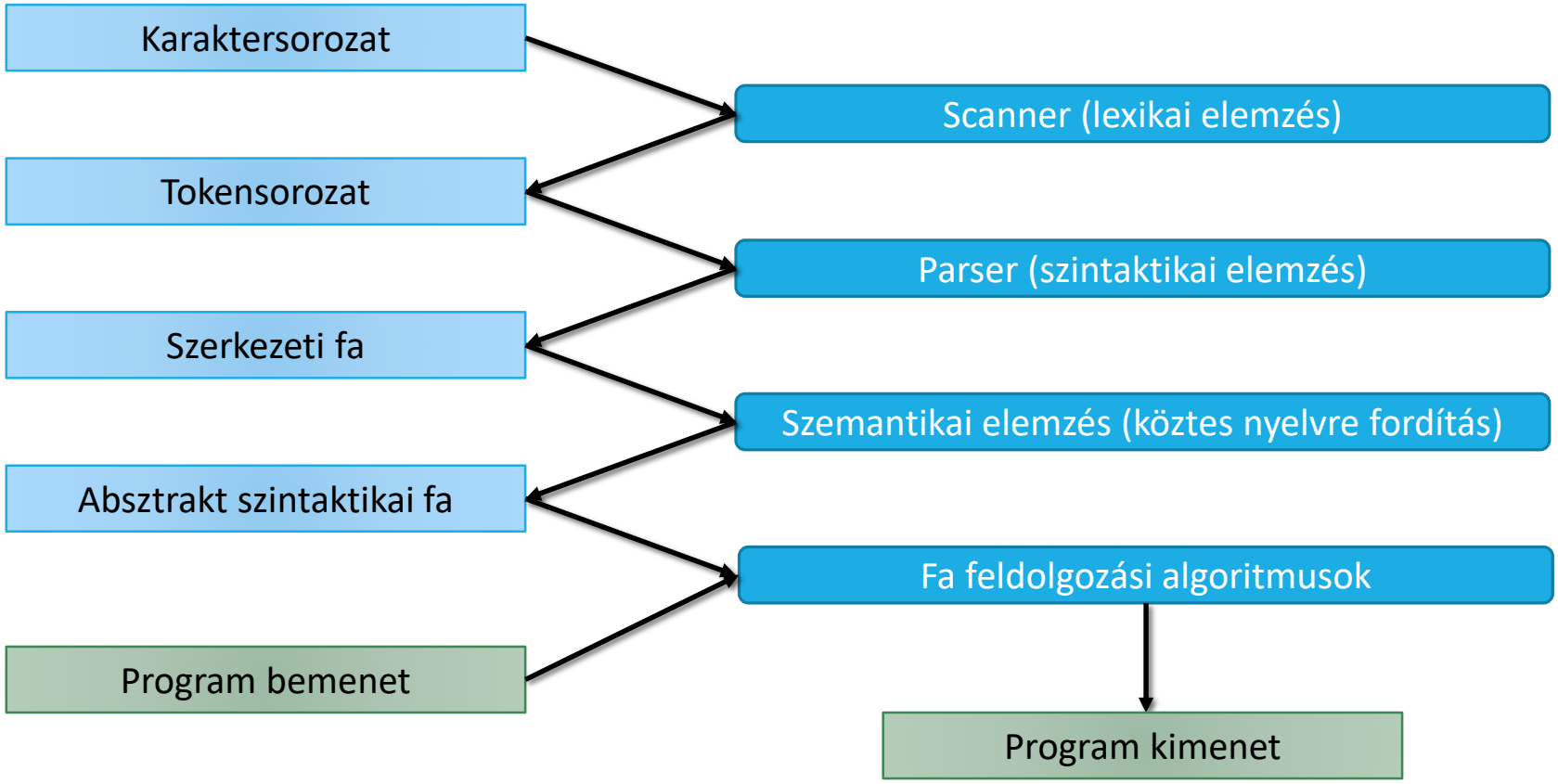
Példa:

- DD / DD / DDDD
- 02 / 11 / 2019

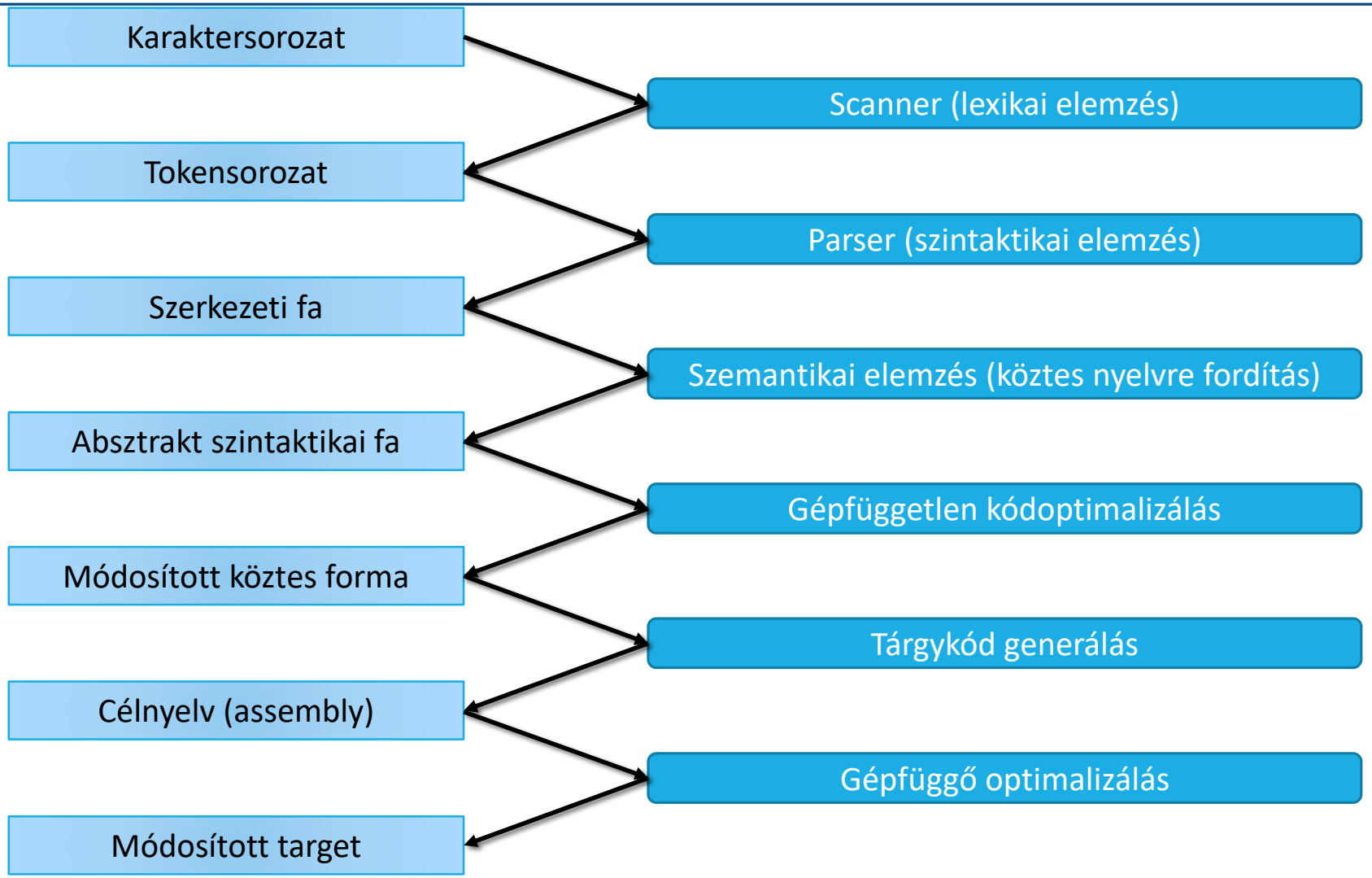
2019. február 11. vagy 2019. november 2.?

A szintaxis befolyásolja a programok megbízhatóságát

# Interpretálás



# Fordítás


Karaktersorozat
Scanner (lexikai elemzés)
Tokensorozat
Parser (szintaktikai elemzés)
Szerkezeti fa
Szemantikai elemzés (köztes nyelvre fordítás)
Absztrakt szintaktikai fa
Gépfüggetlen kódoptimalizálás
Módosított köztes forma
Tárgykód generálás
Célnyelv (assembly)
Gépfüggő optimalizálás
Módosított target

# Nyelv leírása szabályokkal

# Nyelv leírása a fordítónak

Formális nyelv

- Grammatika – generálja a nyelvet
  - A szabályok alapján a nyelv szerinti „mondatok” hozhatók létre
- Automaták – felismerik
  - A generált „mondatok” elemzése, ellenőrzése, feldolgozása

# Formális nyelvek – definíciók

**Σ:** az **ábécé** tetszőleges szimbólumok véges, nem üres halmaza

A **szó** az ábécé elemeiből képezett $a_1a_2...a_k$ alakú sorozat, ahol $k \geq 0$, $a_1, a_2, ..., a_k \in \Sigma$

A **Σ*** a $\Sigma$ ábécé elemeiből képezhető összes szavak halmaza:

- $\Sigma^* = \{a_1a_2...a_k \mid k \geq 0, a_1, a_2, ..., a_k \in \Sigma\}$
- Ha $k=0$, akkor a szó üres szó, jele $\varepsilon$ vagy $\lambda$

$\Sigma^+$ a nem üres szavak halmaza:

- $\Sigma^+ = \Sigma^* \setminus \{\lambda\}$

# Példa

$$\Sigma = \{a, b\}$$

$$\Sigma^* = \{\lambda, a, b, aa, ab, ba, bb, aaa, aab, \dots\}$$

$$\Sigma^+ = \Sigma^* - \lambda$$

# Grammatikák – definíciók

**Generatív nyelvtan (G)**

$G=(N, \Sigma, S, P)$, ahol

$N$ a nemterminálisok (grammatikai jelek) ábécéje

$\Sigma$ a terminálisok abc-je, $N \cap \Sigma = \varnothing$

$S \in N$ a kezdőszimbólum

$P$: $\alpha \rightarrow \beta$ alakú átírási szabályok véges halmaza

$\alpha$ a szabály baloldala, $\beta$ a szabály jobboldala

$\alpha, \beta \in (N \cup \Sigma)^*$

$\alpha$-ban van legalább egy nemterminális szimbólum

# Magyarázat

## Nemterminális

- Nem „célnyelvi” szavak, hanem a levezetési szabályok még behelyettesíthető szimbólumai
  - < ADJ > < NP > < PUNCT>

## Terminális

- „Célnyelvi” szavak, nem helyettesíthetők be tovább
  - Kisfiú tárnyért

## Kezdőszimbólum

- A startszimbólum, ezzel kezdjük meg a levezetést

# Grammatikák – definíciók

**Közvetlen levezetés (közvetlen levezetési reláció, $\Rightarrow G$)**

Legyen $\gamma, \delta \in (N \cup \Sigma)^*$

$\gamma \Rightarrow G\ \delta$, ha van olyan $\alpha \rightarrow \beta \in P$ szabály és

vannak olyan $\alpha', \beta' \in (N \cup \Sigma)^*$ szavak, hogy

$\gamma = \alpha'\alpha\beta'$ és $\delta = \alpha'\beta\beta'$

- $\Rightarrow G^n$ - n lépéses levezetés
- $\Rightarrow G^+$ - legalább egy lépéses levezetés
- $\Rightarrow G^*$ - levezethetőség (esetleg nulla lépésben is)

# Grammatikák – definíciók

## G grammatika által generált nyelv L(G)

- Legyen G=(N, $\Sigma$, S, P) generatív nyelvtan.
- L(G)={u$\in\Sigma^*$ | S $\Rightarrow_G^*$ u}
- L(G) = a G generatív grammatika által generált nyelv
  - összes lehetséges szó

## Ekvivalens nyelvtanok

- Legyenek G=(N, $\Sigma$, S, P) és G'=(N', $\Sigma$', S', P') generatív nyelvtanok.
  - G ekvivalens G'-vel, ha L(G)=L'(G').
  - Ekvivalens két nyelvtan, ha ugyanazt a nyelvet generálja

# Chomsky-féle nyelvosztályok

## 0. típusú nyelvtan (kifejezés struktúrájú)

- ha semmilyen korlátozás nincs
- Turing gép tudja feldolgozni

## 1. típusú nyelvtan (környezetfüggő)

- ha P-ben minden szabály $\alpha A\beta \rightarrow \alpha\delta\beta$ alakú
  - Itt $\delta \neq \lambda$, kivéve esetleg az $S \rightarrow \lambda$ szabályt, mely esetben $\alpha=\beta=\delta=\lambda$, de ekkor S nem szerepelhet egyetlen szabálynak sem a jobboldalán.
  - $(\alpha,\beta,\gamma,\delta \in (N \cup \Sigma)^*)$
- Turing gép tudja feldolgozni
- *Környezettfüggőség: a behelyettesítésnél figyelembe kell venni a behelyettesítendő szimbólum környezetében (szomszédságában) levő további szimbólumokat*

# Chomsky nyelvosztályok (folyt.)

## 2. típusú nyelvtan (környezetfüggetlen)

- ha P-ben minden szabály $A \rightarrow \alpha$ alakú
- ($A \in N$, $\alpha \in (N \cup \Sigma)^*$)
- Nemdeterminisztikus veremautomata tudja feldolgozni
- *Környezettfüggőség: a behelyettesítésnél nincs meghatározva megkötés a behelyettesítendő szimbólum környezetében levő további szimbólumokra*

## 3. típusú nyelvtan (reguláris, jobblineáris)

- ha P-ben minden szabály $A \rightarrow xB$, vagy $A \rightarrow x$ alakú.
- ($A, B \in N$, $x \in \Sigma^*$)
- Véges determinisztikus automata tudja feldolgozni

# Chomsky nyelvosztályok

**Nyelv típusa**

Egy $L \subseteq \Sigma^*$ nyelv i típusú (i=0,1,2,3),
ha van olyan i típusú nyelvtan, ami éppen L-et generálja.

$$\mathcal{L}_0 \supset \mathcal{L}_1 \supset \mathcal{L}_2 \supset \mathcal{L}_3$$

# Példák

Reguláris grammatika: $G = (\{S\}, \{a,b\}, S, P)$

P: $S \rightarrow abS$

$S \rightarrow a$

$S \Rightarrow a$

$S \Rightarrow abS \Rightarrow aba$

$S \Rightarrow abS \Rightarrow ababS \Rightarrow ababa = (ab)^2a$

$S \Rightarrow abS \Rightarrow ababS \Rightarrow abababS \Rightarrow abababa = (ab)^3a$

$S \Rightarrow abS \Rightarrow ababS \Rightarrow abababS \Rightarrow abababababS \Rightarrow abababababa = (ab)^4a$

$L(G) = \{(ab)^n a \mid \text{ahol } n \geq 0\}$

$L(G) = (ab)^*a$

# Példák

Környezetfüggetlen grammatika: $G = (\{S\}, \{a,b\}, S, P)$

P: $S \rightarrow aSb$

$S \rightarrow \lambda$

$S \Rightarrow \lambda$

$S \Rightarrow aSb \Rightarrow ab$

$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb = a^2b^2$

$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaabbb = a^3b^3$

....

$L(G) = \{a^n b^n \mid \text{ahol } n \geq 0\}$

# Példák és problémák

# Reguláris kifejezések

Definíció: Egy $S$ abc feletti reguláris kifejezések:

1. $S$ minden eleme.
2. $\lambda$ (vagy $e$)
3. $\emptyset$
4. Ha $a$ és $b$ reguláris kifejezések, akkor $(ab)$ is az.
5. Ha $a$ és $b$ reguláris kifejezések, akkor $(a \cup b)$ is az.
6. Ha a reguláris kifejezés, akkor $a^*$ is az.

Minden reguláris kifejezés reprezentál egy nyelvet.
Példa reguláris kifejezésre és nyelvre:

$(a^* bc)\ ^*ab$ $(\{a\}^*\{b\}\{c\}\ )^*\{a\}\{b\}$

# Reguláris kifejezések

Minták megadása – „hagyományos” jelölések a metakarakterekre:

- Karakter megfeleltetés
  - . (pont) Bármilyen karakter
    - Pl. .a. ⇔ ‘pap’, ‘rak’, ‘lap’, ‘kap’, stb.
  - [karakterek] A szögletes zárójelek között felsorolt karakterek valamelyikével megegyező karakter:
    - Pl. [abc] ⇔ ‘a’ vagy ‘b’ vagy ‘c’
    - A - (mínusz) jellel tartományt is megadhatunk.
    - Például [0-9] megfelel bármely számjegynek vagy [a-zA-Z] bármely kis vagy nagybetűnek.
  - [^karakterek] az előző operátor tagadása
    - Pl. [^a-d] : bármely karakter, ami nem az ‘a’, ‘b’, ‘c’ vagy ‘d’

# Reguláris kifejezések

- \d rövidítés: számjegyek 0-9
- \w rövidítés: betűk, számjegyek, ‘_’
- \s rövidítés: whitespace (space,tab,CR,LF)
- \D, \W, \S rövidítések: a fentiek negációi

Többszörözés - ismétlő operátorok a minták után:

- ? : 0 vagy 1 egyezés - pl. \d? ⇔ ∅, vagy egy számjegy
- * : 0 vagy több egyezés - pl. ab*a ⇔ aa, aba, abba, …
- + : 1 vagy több egyezés - pl. [0-9]+ ⇔ 1, 45, 460, ...
- {n} : n egyezés (n >= 1) - pl. .{3} ⇔ aaa, 888, ...
- {n,m} : n és m közötti egyezés (n>=1, n<=m) –
  pl. a{2,4} ⇔ ‘aaaa’, ‘aaa’ vagy ‘aa’
- {n,} : legalább n egyezés (n>=1) -pl. xa{2,}. ⇔ xaaf, xaaa, ...
- {,m}:  legfeljebb m egyezés

# Reguláris kifejezések

Diszjunkció

- „|” (pipe) szimbólum két minta között: „vagy”

  pl. a|A ⇔ ‘a’, ‘A’

- lehet sorozatban több is:

  pl. a|b|c ugyanaz, mint [a-c]

- a „|” -jel precedenciája a legalacsonyabb az operátorok között. Ezért:

  te|i ⇔ ‘te’, ‘i’

  t(e|i) ⇔ ‘te’, ‘ti’

# Reguláris kifejezések

## Horgonyok

- ^ : az elejére illeszkedik
  pl. ^M : <‘M’ a kezdete>
- $ : a  végére illeszkedik:
  pl. \.$ : <pont a végén>
- \b : szóhatárra illeszkedik (\w és \W karakterek közötti pozícióra, vagy \w és sztring kezdete/vége közé)
  pl. c\b : ‘abc’-re illeszkedik ‘abc  def’-ben
- \B: \b negáltja (karakter „szó” közepén)

## Speciális karakterek csak ‘\’ segítségével használhatók

- Pl. \. vagy \-

# Reguláris kifejezések

Csoportok

- A zárójel operátorokkal tetszőleges számú csoport kijelölhető a mintában:

  pl. a((bla)ko(k))(a)t
- A csoportokra alkalmazhatók az operátorok:

  pl. a(lm|kn)a

# Használat

Bármilyen keresésnél, formai ellenőrzésnél:

- Google-ben, ha nem tudjuk biztosan, vagy több írásmód is létezik:
  - "Handel", "Händel”  vagy "Haendel” ?
  - H(ä|ae?)ndel
- IP-cím formátum:
  - <3-jegyű_szám>.<3-jegyű_szám>.<3-jegyű_szám>.<3-jegyű_szám>
  - RegKif: \b\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}\b

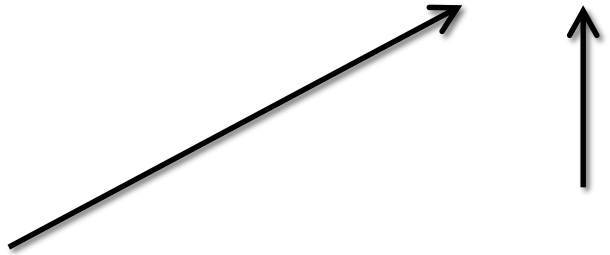
Fordítóprogramokban a lexikális elemző (lexer)

- Eldönthető, hogy érvényes szimbólumokból állnak-e a szavak
- változónév: [a-zA-Z][a-zA-Z0-9]*
- egész szám: (\+|\-)?[0-9]+
- valós szám: [0-9]+\.[0-9]*

# Context-Free Grammatika

$G=(N, \Sigma, S, P)$

ha P-ben minden szabály $A \rightarrow \alpha$ alakú.



Grammatikai jel

Grammatikai jelek és terminálisok sorozata

$(A \in N,\ \alpha \in (N \cup \Sigma)^*)$

# A grammatika által generált nyelv

$G$ grammatika

$S$ kezdőszimbólum

$$L(G) = \{w : \quad S \overset{*}{\Rightarrow} w, \quad w \in \Sigma^*\}$$

Terminálisok sorozata vagy a λ

# Context-Free nyelv

Egy L nyelv  környezetfüggetlen (CF) ha van egy G context-free grammatika, ahol

$$L = L(G)$$

# Levezetési fa

Definíció: Egy G CF grammatika szerinti levezetés egy fával ábrázolható

- A gyökere a kezdőszimbólum
- A levelek balról jobbra a levezetett szó betűi
- További csomópontjai nemterminálisok
- Az elágazások egy-egy alkalmazott szabálynak felelnek meg

Ez a levezetési fa.

# Példa

Context-free grammatika: $G$

$$S \to aSa \mid bSb \mid \lambda$$

Példa levezetések:

$$S \Rightarrow aSa \Rightarrow abSba \Rightarrow abba$$

$$S \Rightarrow aSa \Rightarrow abSba \Rightarrow abaSaba \Rightarrow abaaba$$

---

$$L(G) = \{ww^R : \quad w \in \{a,b\}^*\}$$

Páros hosszú palindromák

# Másik példa

Context-free grammatika : $G$

$$S \to aSb \mid SS \mid \lambda$$

Példa levezetések:

$$S \Rightarrow SS \Rightarrow aSbS \Rightarrow abS \Rightarrow ab$$

$$S \Rightarrow SS \Rightarrow aSbS \Rightarrow abS \Rightarrow abaSb \Rightarrow abab$$

---

$$L(G) = \{w \ : n_a(w) = n_b(w),$$

$$\text{és } \ n_a(v) \geq n_b(v)$$

$$\text{a } \ w \ \textit{bármely} \ v \ \textit{prefixében}\}$$

Leírja a jó zárójelezést: () ((( ))) (( ))

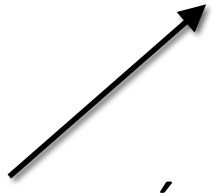
$a = ($, b $= )$

# A matematikai kifejezések grammatikája

$$E \rightarrow E + E \mid E * E \mid (E) \mid a$$

Példa sorozatok:

$$(a + a) * a + (a + a * (a + a))$$



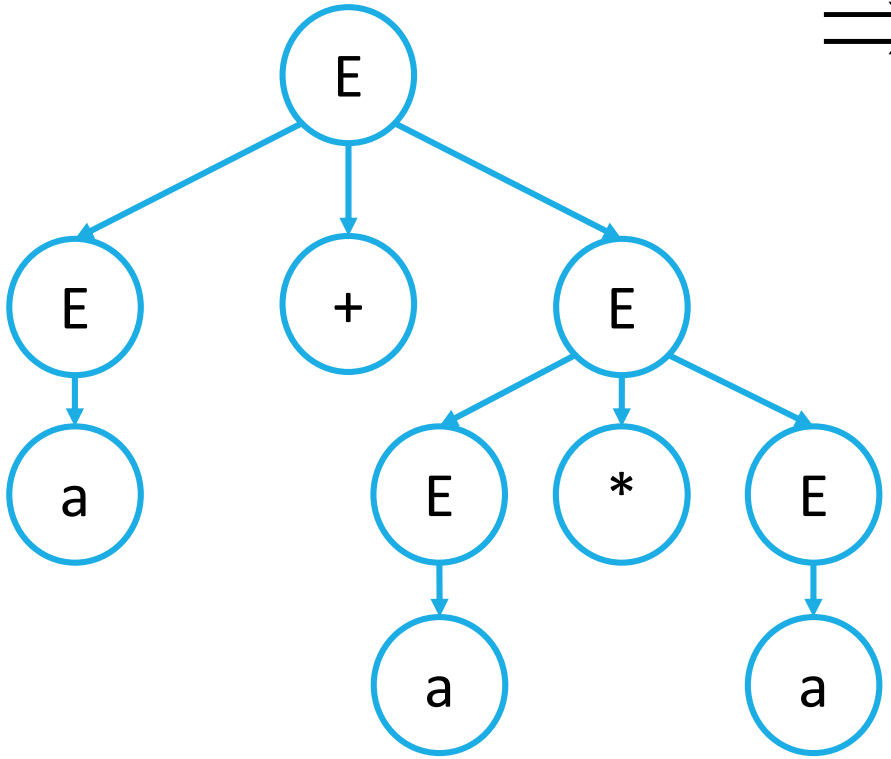
Tetszőleges szám

$$E \rightarrow E + E \mid E * E \mid (E) \mid a$$

$$E \Rightarrow E + E \Rightarrow a + E \Rightarrow a + E * E$$

$$\Rightarrow a + a * E \Rightarrow a + a * a$$

$$a + a * a$$




egy legbaloldalibb levezetése

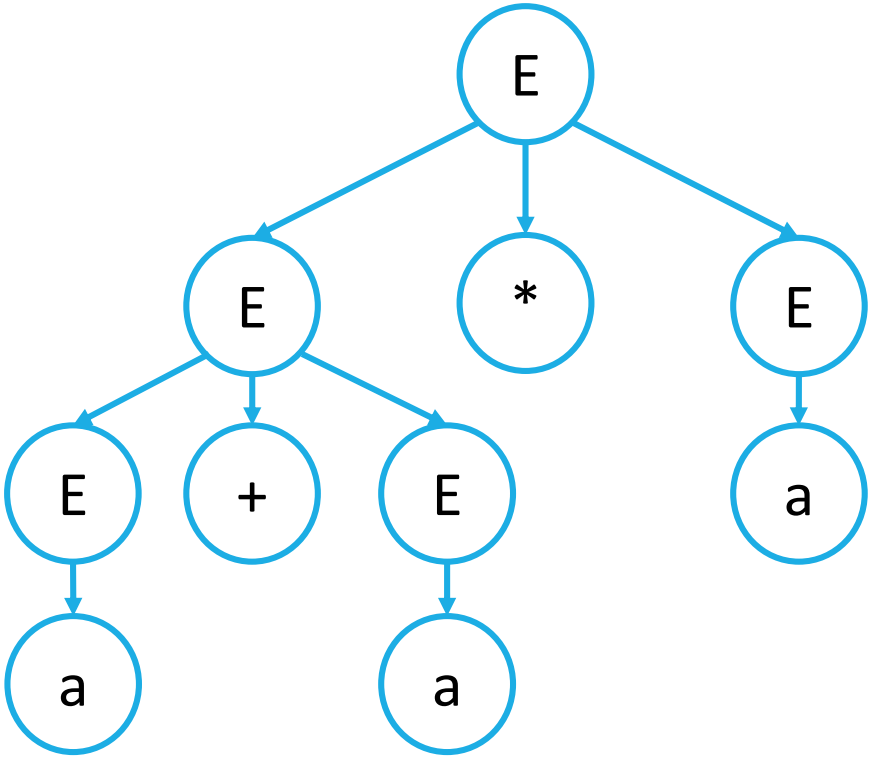
$$E \rightarrow E + E \mid E * E \mid (E) \mid a$$

$$E \Rightarrow E * E \Rightarrow E + E * E \Rightarrow a + E * E$$

$$\Rightarrow a + a * E \Rightarrow a + a * a$$

$$a + a * a$$

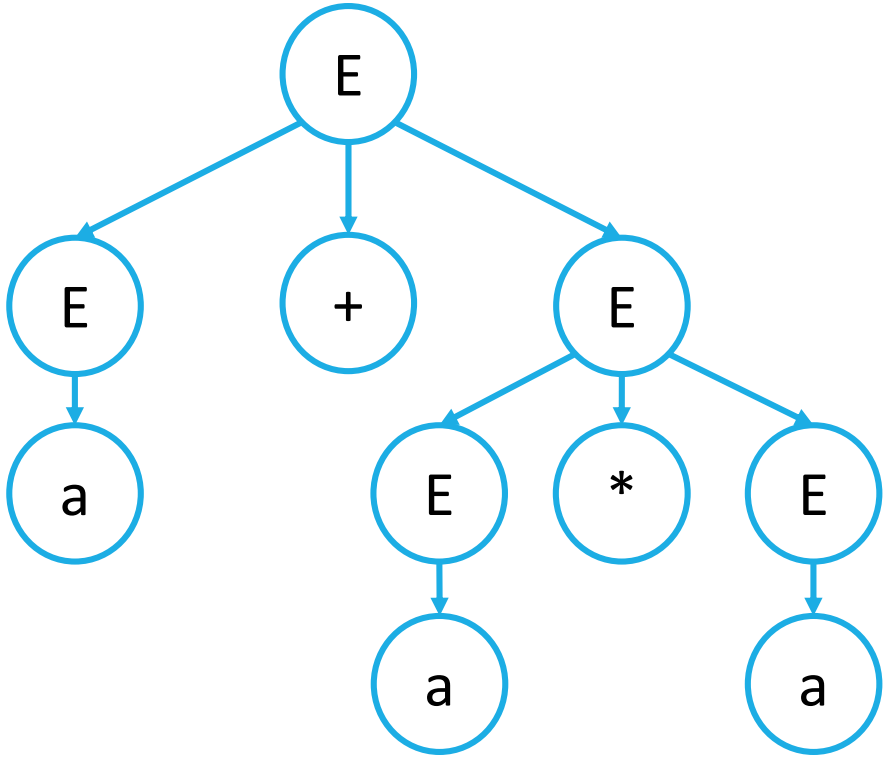
Egy másik legbaloldalibb levezetése

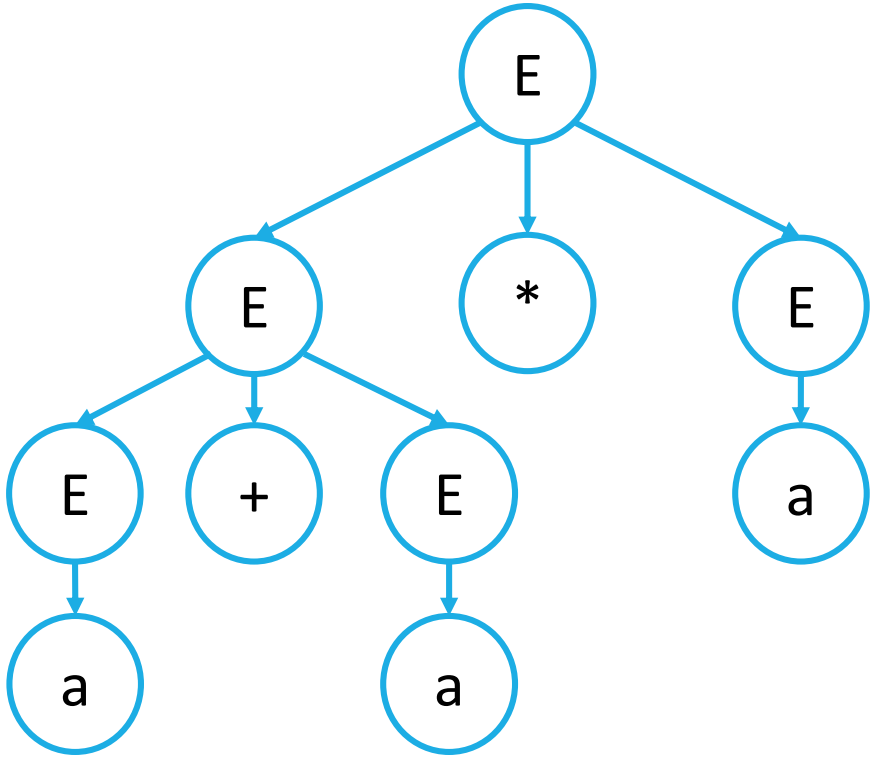


$$E \rightarrow E + E \mid E * E \mid (E) \mid a$$

$$a + a * a$$

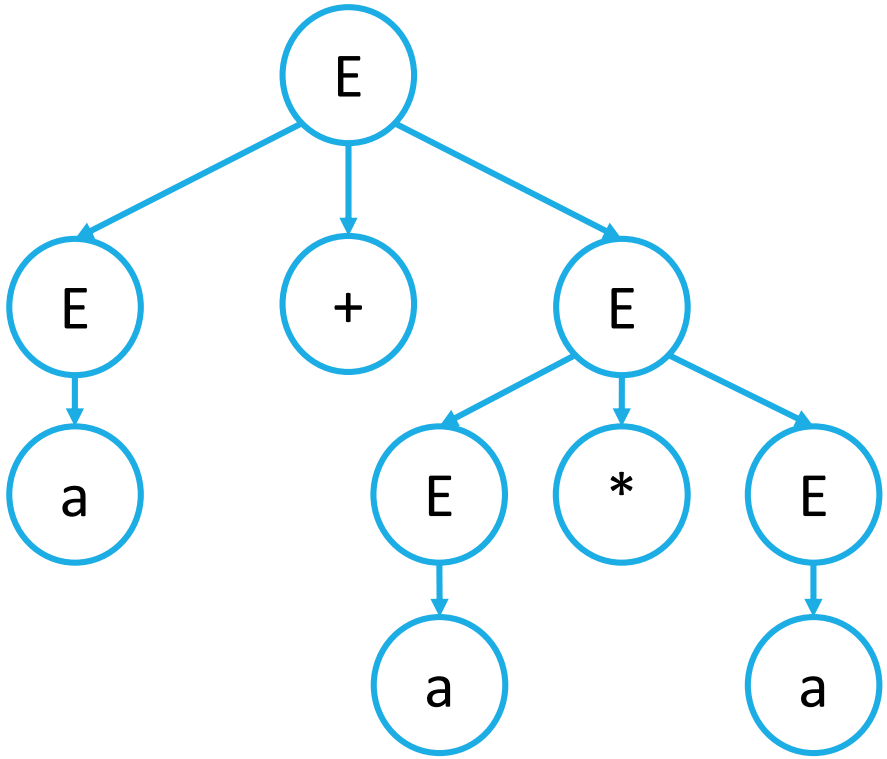
Két levezetési fa!

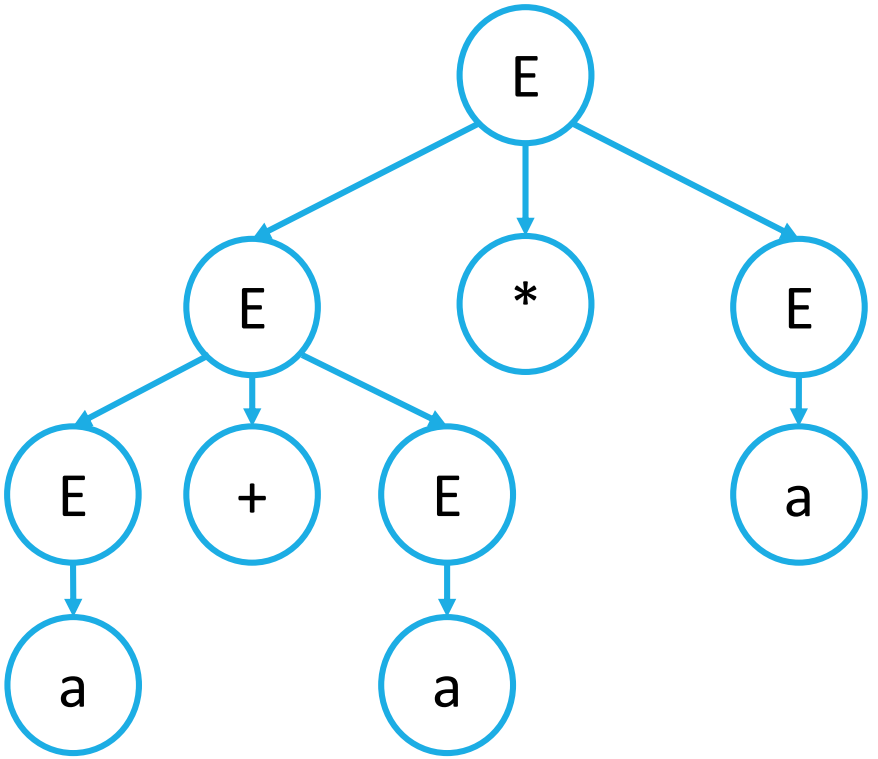


# Legyen a = 2

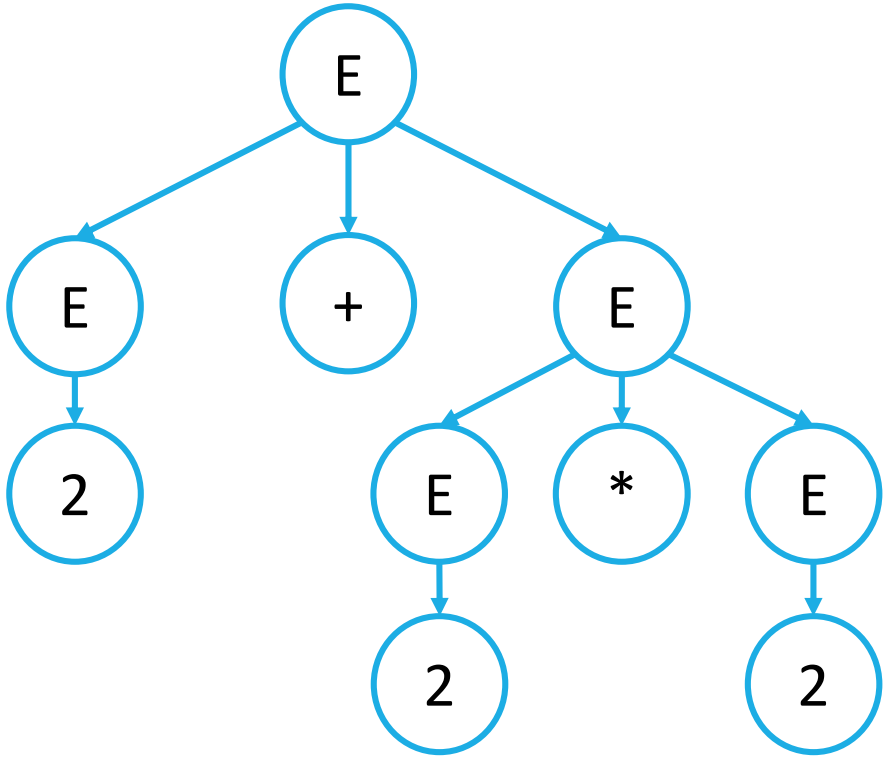
$$a + a * a = 2 + 2 * 2$$



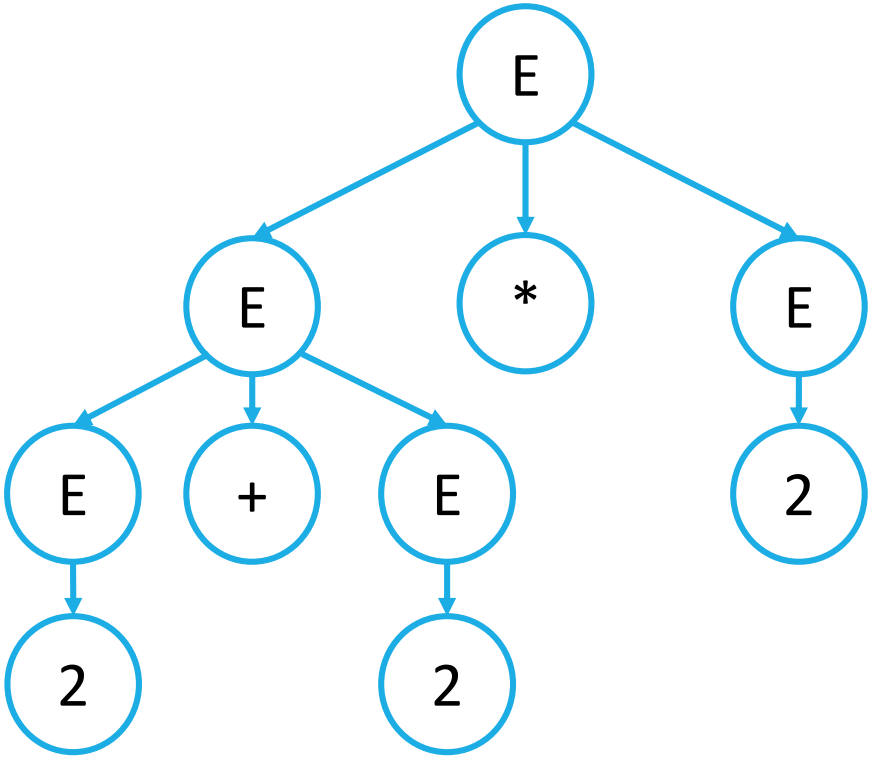
# Legyen a = 2

$2 + 2 * 2 = 6$




Számítsuk ki a kifejezés eredményét a fát használva

$2 + 2 * 2 = 8$



# Többértelmű grammatika

Egy G környezetfüggetlen grammatika **többértelmű**

ha van olyan $w \in L(G)$ amelyiknek:

- két különböző levezetési fája van, vagy
- két különböző legbaloldalibb levezetése van

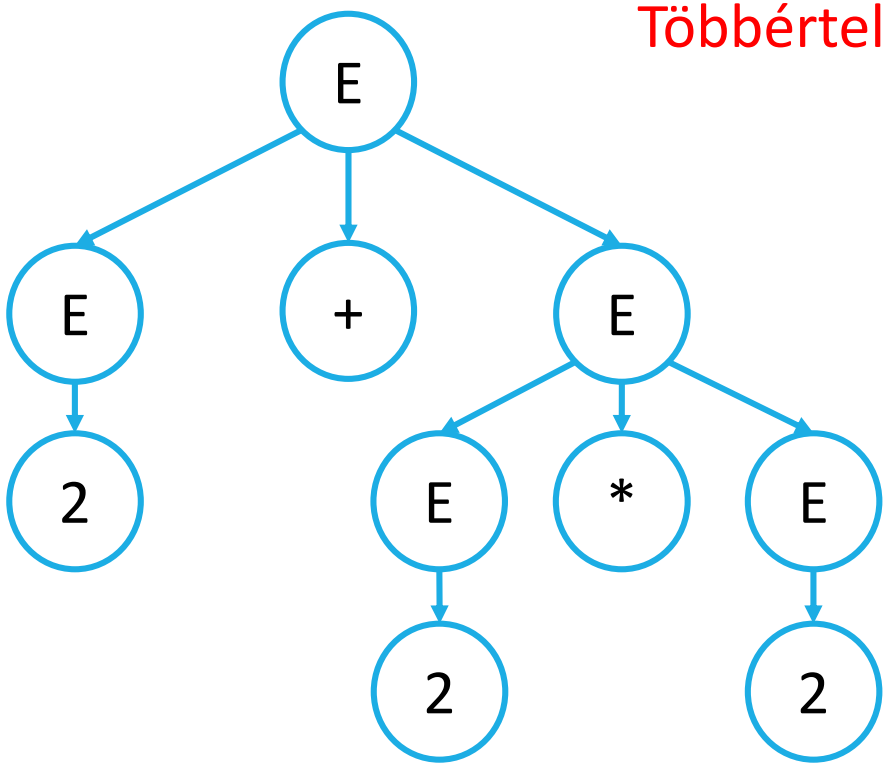
Két különböző levezetési fa problémákat okozhat

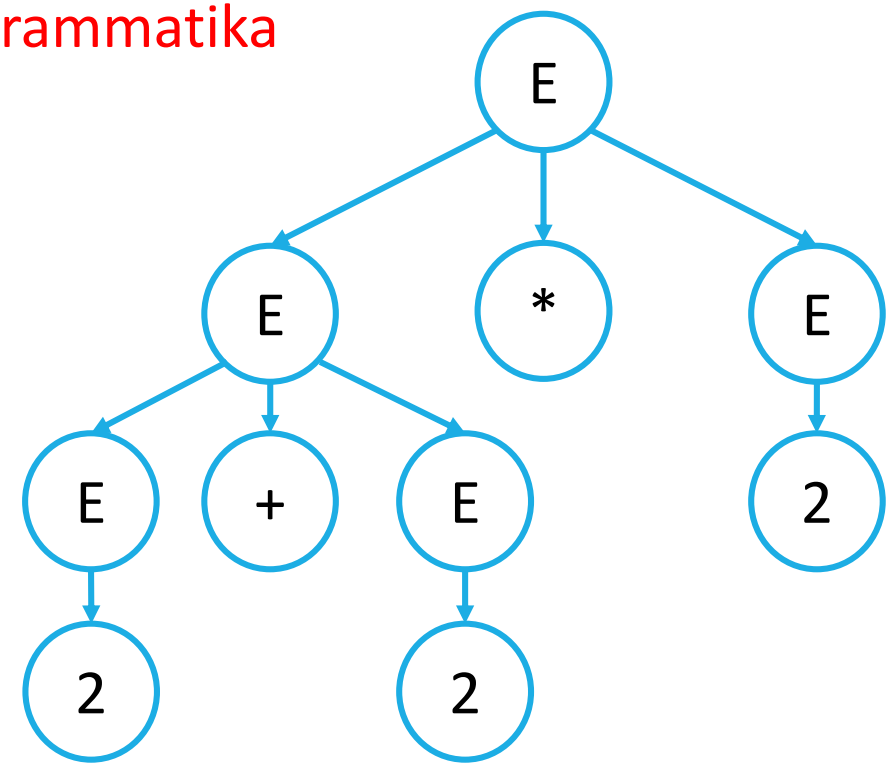
- kifejezések kiértékelésénél
- Általában - programozási nyelvek fordítóprogramjainál!

$$E \rightarrow E + E \mid E * E \mid (E) \mid a$$

$$a + a * a$$

Többértelmű a grammatika

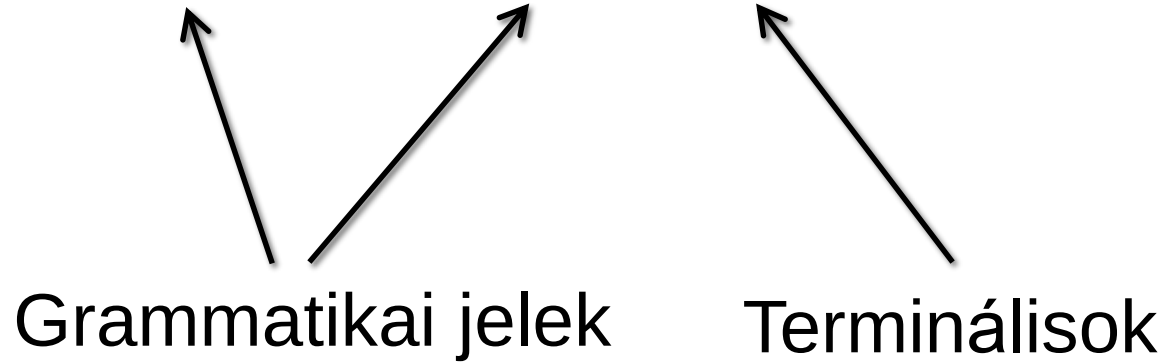


# Egy másik többértelmű grammatika

IF_STMT → **if** EXPR **then** STMT

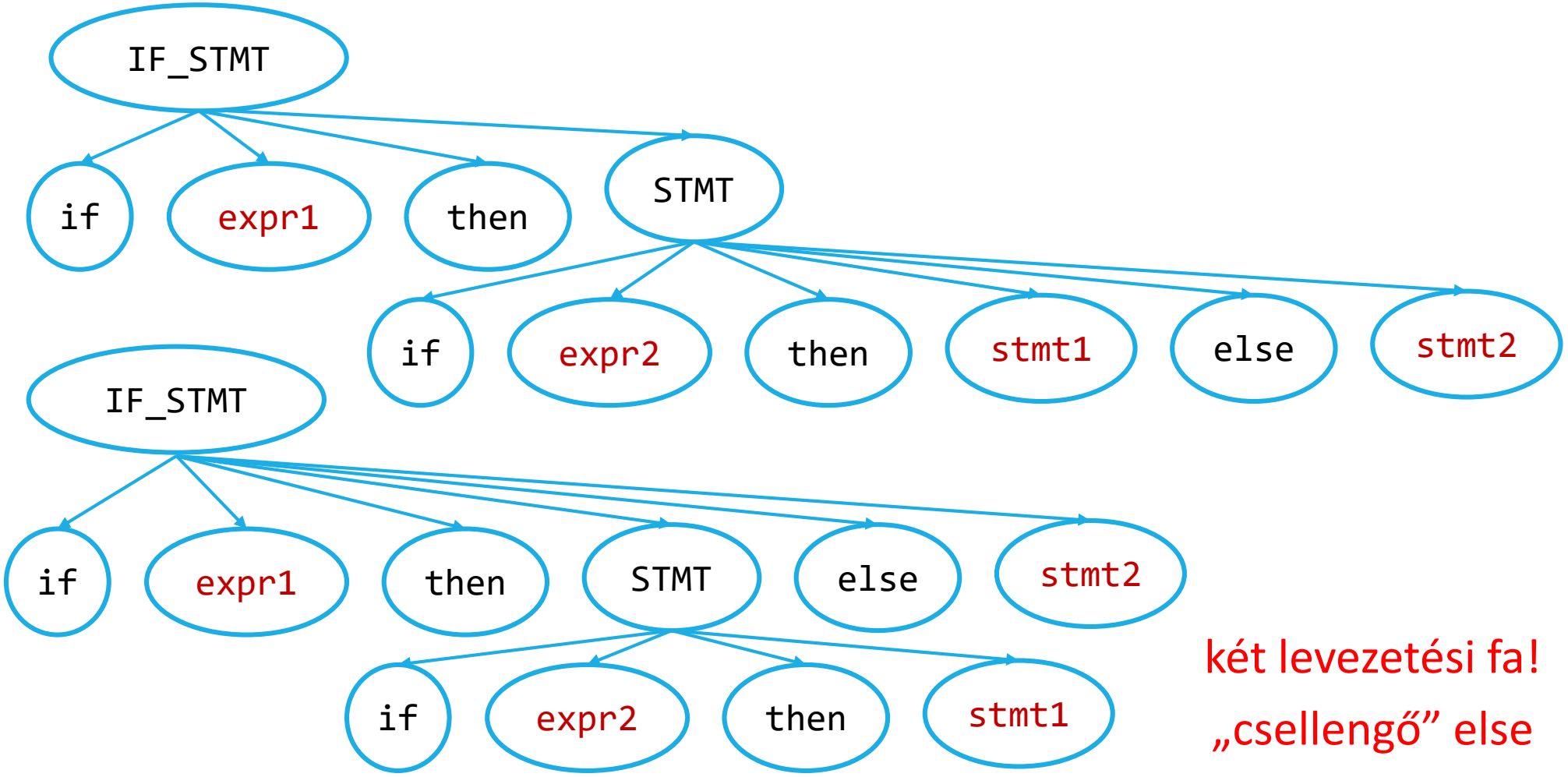
| **if** EXPR **then** STMT **else** STMT




A programozási nyelvek grammatikájának jellemző része

# Egy másik többértelmű grammatika

```
if expr1 then if expr2 then stmt1 else stmt2
```




két levezetési fa!

„csellengő” else

# Többértelműség

Általában a többértelműség rossz, és szeretnénk megszüntetni…

- Néha sikerül találni egyértelmű  (nem többértelmű) grammatikát a nyelvhez
- De általánosságban ezt nehéz elérni...

# Egy sikeres példa

Többértelmű grammatika:

$$E \to E + E$$
$$E \to E * E$$
$$E \to (E)$$
$$E \to a$$

Ekvivalens

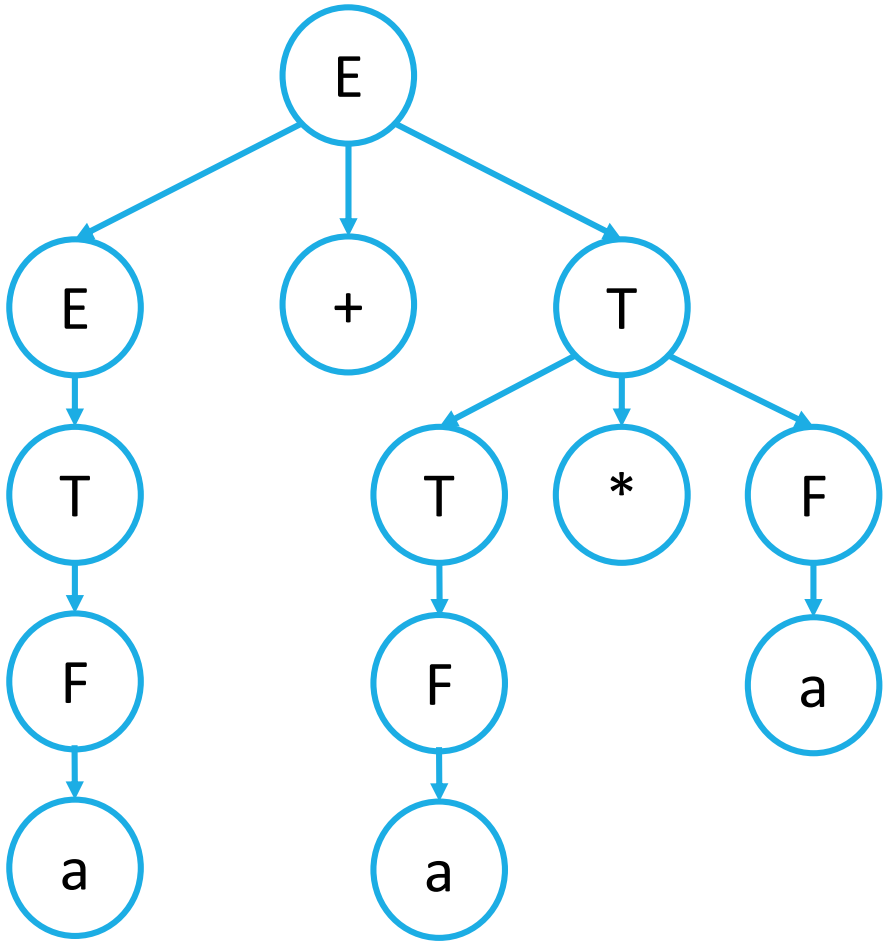
Nem-többértelmű grammatika

Ugyanazt a nyelvet generálja!

$$E \to E + T \mid T$$
$$T \to T * F \mid F$$
$$F \to (E) \mid a$$

$$E \Rightarrow E+T \Rightarrow T+T \Rightarrow F+T \Rightarrow a+T \Rightarrow a+T*F$$

$$\Rightarrow a+F*F \Rightarrow a+a*F \Rightarrow a+a*a$$




$$E \rightarrow E+T \mid T$$

$$T \rightarrow T*F \mid F$$

$$F \rightarrow (E) \mid a$$

$$a+a*a$$

Egyértelmű a levezetési fa

# Programozási nyelv megadása

# Programozási nyelv megadása

Formális szintaxis leírás

- ezt használja a fordítóprogram a nyelvi helyesség ellenőrzésére

Formális szemantika leírás

- a jelentés ellentmondásmentességének ellenőrzésére

# John Backus (1924-2007)



FORTRAN tervezője

"Munkám túlnyomó része egyszerűen a lustaságomból eredt. Amikor a rakéták röppályáját kiszámító programokon dolgoztam, hozzáfogtam egy programozási rendszer kidolgozásához, amelynek elsődleges feladata a munka megkönnyítése, leegyszerűsítése volt”

# Backus-Naur Form (BNF)

Nyelv a programozási nyelvek definiálására

- meta-nyelv, eredetileg 59-60-ból, az Algol és Fortran definiálására
- változatok ma is programozási nyelvek megadására

BNF formában adott szintaxishoz léteznek eszközök:

- automatikusan generálnak egy elemzőt
  - a szintaktikus helyesség ellenőrzésére
  - levezetési fa készítésére
  - adott esetben a program lefordítására

# Backus-Naur Form (BNF)

Példa

```
<loop statement> ::= <while loop> | <for loop>

<while loop> ::= while ( <condition> ) <statement>

<for loop> ::= for ( <expression> ; <expression>;
    <expression> ) <statement>

<assignment statement> ::=
    <variable> = <expression>
```

# BNF Linkek

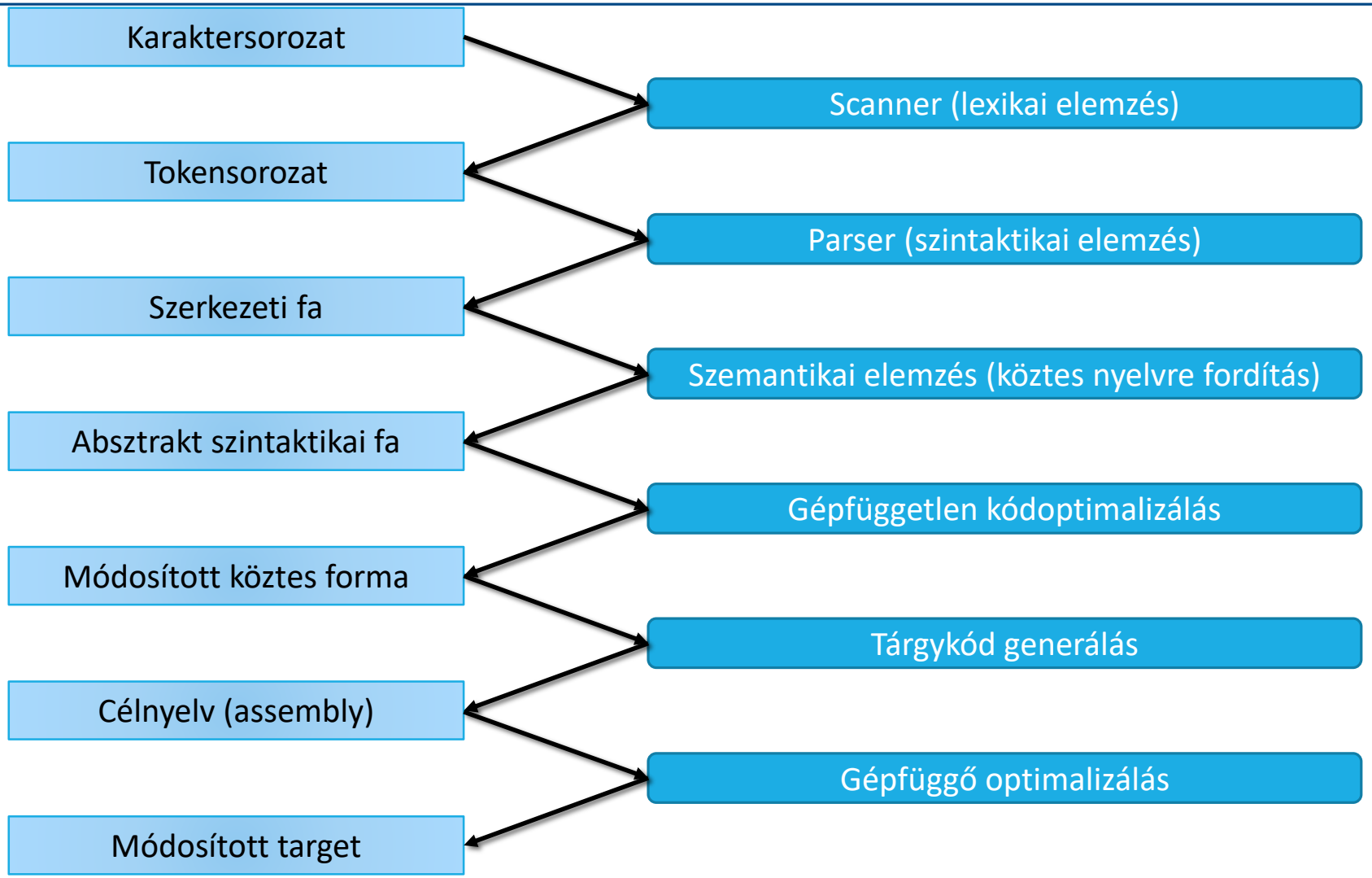
Java BNF

- https://docs.oracle.com/javase/specs/jls/se13/html/jls-19.html

Python BNF

- https://docs.python.org/3/reference/grammar.html

# Fordítás


Karaktersorozat
Scanner (lexikai elemzés)
Tokensorozat
Parser (szintaktikai elemzés)
Szerkezeti fa
Szemantikai elemzés (köztes nyelvre fordítás)
Absztrakt szintaktikai fa
Gépfüggetlen kódoptimalizálás
Módosított köztes forma
Tárgykód generálás
Célnyelv (assembly)
Gépfüggő optimalizálás
Módosított target

# Fordítóprogram

3. Lexikális elemzés
   - megadható reguláris nyelvtannal
2. Szintaktikus elemzés
   - megadható környezetfüggetlen nyelvtannal
1. Szemantikus elemzés
   - környezetfüggő
     - egy változó használatának helyessége függhet a deklarációjától, azaz a környezettől

A három lépés szétválasztásának oka
   - Egyszerű feladathoz ne használjunk bonyolult eszközöket.

# Programszerkezet

## Fordítási egység vagy modul

- A nyelvnek az az egysége, ami a fordítóprogram egyszeri lefuttatásával, a program többi részétől elkülönülten lefordítható
- Több modulból álló programok esetében a fordítás és a végrehajtás között: a program összeszerkesztése



# Lexikai elemek

JÖVŐ HÉTEN




# Programozási nyelvek és módszerek

## LEXIKAI ELEMEK, BEÉPÍTETT TÍPUSOK

# Programegység, fordítási egység

A programegység egy részfeladatot megoldó, tehát funkcionálisan összefüggő programrész.

- Részei
  - A specifikációs rész írja le az egységnek a más egységből elérhető “kapcsolódási pontjait".
  - A törzs tartalmazza az egység funkcióját megvalósító programot.

A fordítási egység programegységek halmaza.

# Ada példa

```
procedure Hello;
```



specifikáció

```
with Text_IO; use Text_IO;

procedure Hello is
   S: String(1..40):=(others=>' ');
   Last: Natural;
begin
   Put("Hogy hívnak?");
   Get_Line(S, Last);
   New_Line;
   Put("Hello "&S(1..Last));
end Hello;
```



törzs

# A törzs felépítése

A deklarációs részben (ha létezik külön)

- azonosítók deklarációit,
- konstansok, típusok, objektumok és programegységek definícióit

Az utasítássorozat részben

- utasításokat, amelyek végrehajtása nyomán a programegység kifejti a hatását.

# Deklarációk

A deklarációs rész feldolgozását, a definíciók értelmezését a deklarációs rész kiértékelésének mondjuk.

A deklarációs rész definícióinak kiértékelése történhet:

- Fordítási (pl, C, C++) vagy
  - statikus definíció-kiértékelés
- Futási (pl. PL/I, Ada) időben
  - dinamikus definíció-kiértékelés

# Deklarációk

Változódeklaráció – sokszor definíció is

- állapottér megadásához
- Formája általában
  - `típus azonosító`
  - `int i;`
- Vagy
  - `azonosító: típus`
  - `N: Natural;`

# Hatáskör, láthatóság

A programszövegnek azt a szakaszát, amelyen belül az adott deklaráció érvényben van, a deklaráció hatáskörének nevezzük.

A programszövegnek azt a szakaszát, ahol a deklarált azonosítóval az adott deklarációra hivatkozni lehet, a deklaráció láthatósági körének nevezzük.

# Blokkstruktúra

A blokkstruktúra a programegységek egymásba ágyazásával előálló hierarchikus struktúra.

Blokkstruktúrában egy programegység deklarációinak hatásköre kiterjed az összes tartalmazott programegységre is.

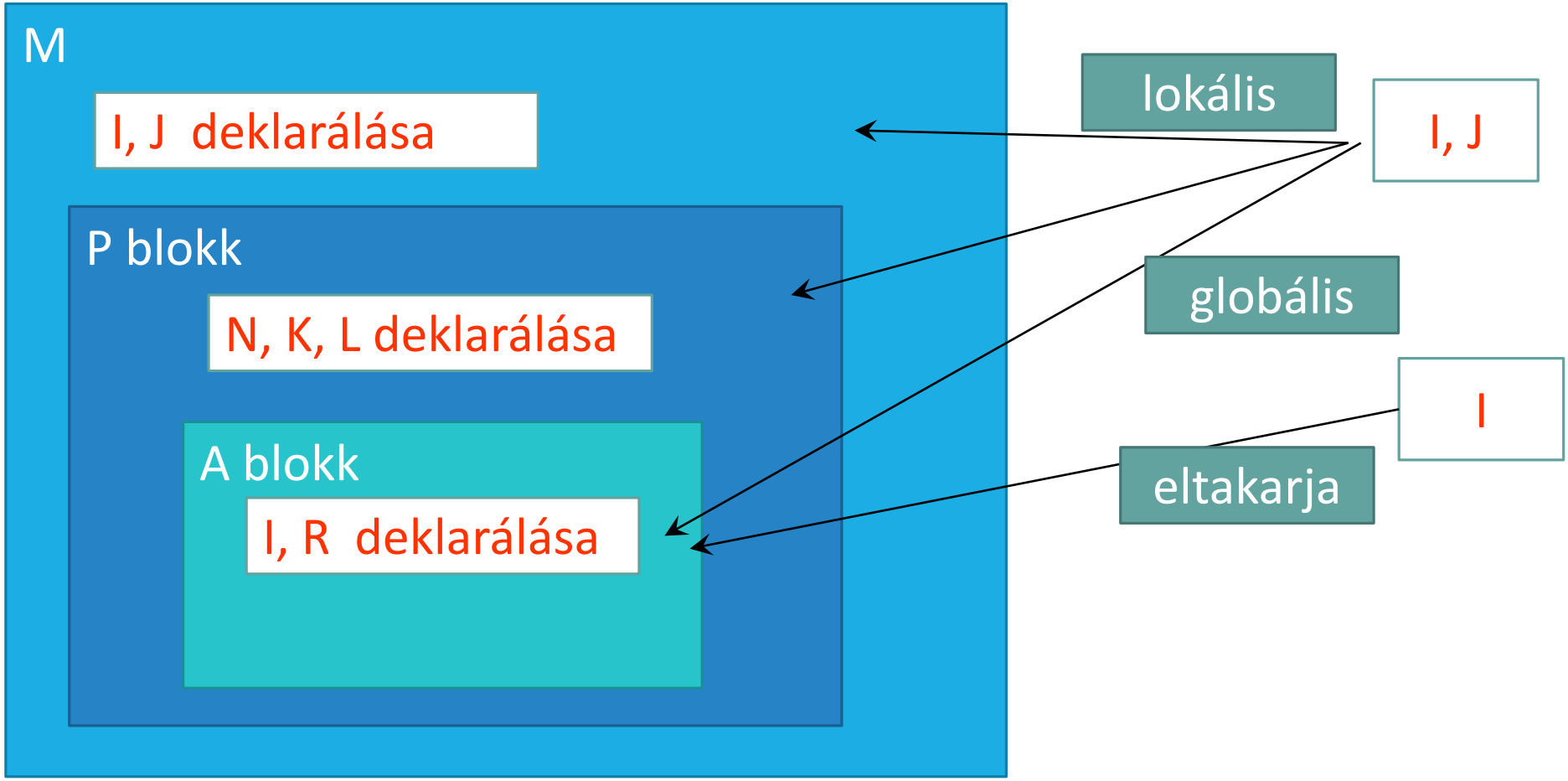
# Globális – lokális

Egy programegységben a tartalmazó programegységek által deklarált azonosítókat (a tartalmazott programegység szemszögéből) globális azonosítóknak nevezzük.

A programegységben deklarált azonosítókat magában a programegységben lokális azonosítóknak nevezzük.

Blokkstruktúra esetén egy programegységben egy globális azonosító újradeklarálható, ekkor a lokális deklaráció eltakarja a globális deklarációt.

# Példa


M
I, J deklarálása
P blokk
N, K, L deklarálása
A blokk
I, R deklarálása
lokális
I, J
globális
I
eltakarja

# Példa

```
procedure M is
  I, J:Integer;
  procedure P is
    N, K, L:Integer;
    procedure A is
      I, R:Integer;
      begin
         I:=3; J:=9;          -- A-beli I, M-beli J
         N:=1;                -- P-beli N
         M.I:=4;              -- minősített hivatkozás!
                              -- ...
      end;
  begin
    I:=5;                     -- M-beli I
    A;                        -- hívja A-t
                              -- ....
  end;

begin
  I:=3;
  P;                          -- hívja P-t ...
end;
```

# Azonosító túlterhelése

Egy azonosítónak ugyanabban a szövegrészben több definíciója is érvényben lehet

A túlterhelést a leggyakrabban alprogramok nevei esetében teszik lehetővé

- általában a paraméterek segítségével tudjuk megkülönböztetni
  - `write(2)`
  - `write("ez valami más")`
  - `2 + 3`
  - `2.1 + 3.0`

# Erősen típusos nyelv

Egy nyelv erősen típusos, ha minden érték, objektum, formális paraméter és függvény típusa fordítási időben egyértelműen meghatározható.

Statikus típusosság, ha a típusok ellenőrzése fordítási időben történik, míg a dinamikus típusosság esetén a típusellenőrzések (műveletekhez) futás időben történnek.

Az erősen típusos nyelv és a statikus típusosság összefonódása ebből következik.

# Memóriakezelés

A változóhoz a memóriaterület hozzárendelése (allokálása) történhet:

- automatikusan, a definíció kiértékelésekor, vagy
- a programozó rendelkezik róla

a változó által lefoglalt terület felszabadítása lehet:

- automatikus, vagy
- a programozó hatáskörébe tartozik.

# Élettartam

A változókhoz a szükséges memóriaterület lefoglalása (allokálása) és annak felszabadítása közötti időt a változó élettartamának nevezzük.

# Statikus változó

élettartama a program egész működése idejére kiterjed

elhelyezése mindig az úgynevezett statikus (main) memória-területre történik

a statikus terület egyszer, a program betöltésekor kerül lefoglalásra

# Dinamikus változó

a program explicit módon foglal le területet, a dinamikus (heap) memóriaterületen, amire a címével, úgynevezett pointerrel lehet hivatkozni.

egyes nyelvek tartalmaznak utasítást a dinamikus területek felszabadítására is.

blokkstruktúrában a programegységek (nem statikus) lokális változói az úgynevezett végrehajtási veremben helyezkednek el.

# Paraméterátadás

A programegységek legelterjedtebb fajtája az alprogram.

- A nyelvek kétféle alprogramot használnak, ezek az eljárások és a függvények.

Az alprogramoknál a hívóval való kommunikációt tervezéskor formális paraméterekkel írjuk le,

- Az alprogram aktivizálásakor (hívásakor) aktuális paramétereket kell megfeleltetni.

# Lexikális elemek

# A fordítási egység

A programok fordítási egységei lexikális elemek sorozatai.

- A lexikális elemek karaktersorozatok, határoló jelekkel elválasztva.

A nyelvekben általában megkülönböztetjük

- A grafikus karaktereket
  - a Latin-1 vagy az angol abc betűi, számjegyek és speciális karakterek
  - Például: `" # & ' ( ) , * - + / : ; < = _ [ ] { } | . -`
- Formátum vezérlő karaktereket
  - Például az ISO 6429 szabványban:
    - character tabulation (HT)
    - line tabulation (VT)
    - carriage return (CR)
    - line feed (LF)
    - and form feed (FF)
- Az egyéb vezérlő karaktereket
  - Ez implementáció függő

# Lexikális elemek

Lexikális elem

- Azonosító
- Numerikus literál
  - Numerikus érték leírására
- Karakter literál
  - ...
- String literál
- Határoló
- Megjegyzés

# A jelkészlet

A karakterkészlet  különböző karakterek egy halmaza.

- Nincs feltételezésünk a belső ábrázolásáról, rendezettségéről.
- Megadjuk a karakterek neveit és a karakter megjelenítését látható formában.
  - Tartalmazhat olyan karaktereket, amelyek bizonyos megjelenítéseknél ugyanúgy néznek ki, mégis logikailag különböző.
    - Három különböző karakter, ami nagyon hasonló megjelenésű:
      - a latin nagy „A”
      - a cirill nagy „A”
      - görög nagybetűs Alfa „A”
  - Példa
    - EXCLAMATION !
    - QUESTION_MARK ?
    - SEMICOLON ;

# Karakterkódok – táblázatok

A karakterkód egy leképezés:

- Kölcsönösen egyértelmű megfeleltetést ad a karakterkészlet karakterei és a nemnegatív egészek egy halmaza között
- Egy egyedi számkódot, egy kódpozíciót rendel a karakterkészlet minden karakteréhez.
  - Gyakran táblázat
  - http://unicode-table.com

# ASCII táblázat

| Decimal | Hex | Char | Decimal | Hex | Char | Decimal | Hex | Char | Decimal | Hex | Char |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | [NULL] | 32 | 20 | [SPACE] | 64 | 40 | @ | 96 | 60 | ` |
| 1 | 1 | [START OF HEADING] | 33 | 21 | ! | 65 | 41 | A | 97 | 61 | a |
| 2 | 2 | [START OF TEXT] | 34 | 22 | " | 66 | 42 | B | 98 | 62 | b |
| 3 | 3 | [END OF TEXT] | 35 | 23 | # | 67 | 43 | C | 99 | 63 | c |
| 4 | 4 | [END OF TRANSMISSION] | 36 | 24 | $ | 68 | 44 | D | 100 | 64 | d |
| 5 | 5 | [ENQUIRY] | 37 | 25 | % | 69 | 45 | E | 101 | 65 | e |
| 6 | 6 | [ACKNOWLEDGE] | 38 | 26 | & | 70 | 46 | F | 102 | 66 | f |
| 7 | 7 | [BELL] | 39 | 27 | ' | 71 | 47 | G | 103 | 67 | g |
| 8 | 8 | [BACKSPACE] | 40 | 28 | ( | 72 | 48 | H | 104 | 68 | h |
| 9 | 9 | [HORIZONTAL TAB] | 41 | 29 | ) | 73 | 49 | I | 105 | 69 | i |
| 10 | A | [LINE FEED] | 42 | 2A | * | 74 | 4A | J | 106 | 6A | j |
| 11 | B | [VERTICAL TAB] | 43 | 2B | + | 75 | 4B | K | 107 | 6B | k |
| 12 | C | [FORM FEED] | 44 | 2C | , | 76 | 4C | L | 108 | 6C | l |
| 13 | D | [CARRIAGE RETURN] | 45 | 2D | - | 77 | 4D | M | 109 | 6D | m |
| 14 | E | [SHIFT OUT] | 46 | 2E | . | 78 | 4E | N | 110 | 6E | n |
| 15 | F | [SHIFT IN] | 47 | 2F | / | 79 | 4F | O | 111 | 6F | o |
| 16 | 10 | [DATA LINK ESCAPE] | 48 | 30 | 0 | 80 | 50 | P | 112 | 70 | p |
| 17 | 11 | [DEVICE CONTROL 1] | 49 | 31 | 1 | 81 | 51 | Q | 113 | 71 | q |
| 18 | 12 | [DEVICE CONTROL 2] | 50 | 32 | 2 | 82 | 52 | R | 114 | 72 | r |
| 19 | 13 | [DEVICE CONTROL 3] | 51 | 33 | 3 | 83 | 53 | S | 115 | 73 | s |
| 20 | 14 | [DEVICE CONTROL 4] | 52 | 34 | 4 | 84 | 54 | T | 116 | 74 | t |
| 21 | 15 | [NEGATIVE ACKNOWLEDGE] | 53 | 35 | 5 | 85 | 55 | U | 117 | 75 | u |
| 22 | 16 | [SYNCHRONOUS IDLE] | 54 | 36 | 6 | 86 | 56 | V | 118 | 76 | v |
| 23 | 17 | [ENG OF TRANS. BLOCK] | 55 | 37 | 7 | 87 | 57 | W | 119 | 77 | w |
| 24 | 18 | [CANCEL] | 56 | 38 | 8 | 88 | 58 | X | 120 | 78 | x |
| 25 | 19 | [END OF MEDIUM] | 57 | 39 | 9 | 89 | 59 | Y | 121 | 79 | y |
| 26 | 1A | [SUBSTITUTE] | 58 | 3A | : | 90 | 5A | Z | 122 | 7A | z |
| 27 | 1B | [ESCAPE] | 59 | 3B | ; | 91 | 5B | [ | 123 | 7B | { |
| 28 | 1C | [FILE SEPARATOR] | 60 | 3C | < | 92 | 5C | \ | 124 | 7C | \| |
| 29 | 1D | [GROUP SEPARATOR] | 61 | 3D | = | 93 | 5D | ] | 125 | 7D | } |
| 30 | 1E | [RECORD SEPARATOR] | 62 | 3E | > | 94 | 5E | ^ | 126 | 7E | ~ |
| 31 | 1F | [UNIT SEPARATOR] | 63 | 3F | ? | 95 | 5F | _ | 127 | 7F | [DEL] |

# Karakterkódolás

A karakterkódolás egy *algoritmus* karakterek digitális formában történő megadására.

- Ez a karakterek kódszámainak sorozatait „oktetek” sorozatára képezi le.
  - Például az ISO 10646 karakterkódban az 'a'  az 'ä' és az ‰ karaktereknek megfelelő számkódok a 97, a 228 és a 8240.

A karakterkódolás adja meg, hogy a karakterkódokat *hogyan* képezzük le oktetek sorozatára.

- Az ISO 10646 egy lehetséges kódolásában minden karakter kódolásához két oktetet használva a kódolási algoritmus ezeknek a (0, 97), a (0, 228) és a (32,48) oktetpárokat felelteti meg.

# Milyen karakterek használhatók egy nyelvben?

## Példák

- Pascal (ISO 7185:1990):
  - Minimum követelmények
    - '0'..'9' rendezett és folytonos kell legyen
    - 'A'..'Z' rendezett, de nem feltétlenül folytonos
    - 'a'..'z' nem kötelező, de ha van, akkor mint a nagybetűk.
  - Nincs előírva a lehetséges karakterek halmaza
    - A különböző implementációk különböző kódolást használhatnak(!)
    - pl. EBCDIC vagy ASCII => különböző implementációk másképpen viselkedhetnek
      - '}' $\in$ 'A'..'Z'  EBCDIC-ben és '}' $\notin$ 'A'..'Z' ASCII-ben
      - ' A' < '1'  EBCDIC-ben és '1' < 'A' ASCII-ben
  - **Ez nem tesz jót a programok hordozhatóságának**

# Milyen karakterek használhatók egy nyelvben?

Példák:

- C++ - ISO/IEC 14882:2017
  - Az alap karakterkészlet 96 elemű
    - Space, horizontal tab, vertical tab, new-line, form feed
    - 91 grafikus karakter:
      - 'a'..'z' 'A'..'Z' '0'..'9'
      - _ { } [ ] # ( ) < > % : ; . ? * + - / ^ & | ~ ! = , \ " ’
  - Univerzális karakterek (UCS)
    - ISO 10646 szerint kódolva
    - Lényegében Unicode
  - **Ez precízebb régebbi verziókhoz viszonyítva:**
    - **Ahol nem volt a kód előírva**
    - **Nem lehetett UCS karaktereket használni**

# Milyen karakterek használhatók egy nyelvben?

Példák:

- CLU, Eiffel:
  - ASCII kód a megengedett.
- ADA (ISO/IEC 8652:2012(E))
  - A teljes ISO/IEC 10646:2011
- Java:
  - "Java programs are written using the Unicode character set”
- C#:
  - „A source file is an ordered sequence of Unicode characters.”
- Python:
  - „Python reads program text as Unicode code points; …”

# Elhatároló jelek

A legtöbb programozási nyelvben a helyköz (space), a tab, a sorvége elhatároló jel.

- Van sok egy karakterből álló elhatároló jel
  - Pascal: & ' ( ) + - * / : . ; < = >
  - C++: & % ^ ' ( ) + - { } | ~ [ ] \ " * / : . ; , < = > ! ?
  - Java: ( ) { } [ ] ; , . + - * / : < = > ! ~ ? : & | ^ %
  - ADA: & ' ( ) * + , - . / : ; < = > |
- Van számos több karakterből álló elhatároló jel is
  - Pascal: := >= <= <> << ** ..
  - C++: -> ++ -- .* ->* == <= >= && || << >> <<= >>= != .. += -= &= ^= |= :: *= /=
  - Java: == <= >= << != && || ++ -- >> >>> += -= *= /= &= |= ^= %= <<= >>= >>>=
  - ADA: => .. ** := /= >= <= << >> <> --

# Megkülönbözteti-e a nyelv a kis- és nagybetűket?

`Kutya = kutya = KUTYA = KuTya = …?`

Alapvetően két megközelítés

- Pascal, Fortran, CLU, ADA, … esetén ekvivalensek
- C, C++, Modula, Perl, Java, C#, … esetén minden betű különböző

De!

- Eiffel: a kis- és a nagybetűk ekvivalensek, de használhatjuk ugyanazt a nevet egy objektumra és a típusára
  - `a: A`
  - (itt az `A` az típus és `a` az objektum)

# Azonosítók

Milyen karakterek használhatóak az azonosítók leírására?

Mi az azonosító megengedett szintaxisa?

Van-e hosszúsági megkötés az azonosítókra?

Vannak-e kötött szavak?

Van-e különbség a kulcsszavak és az előre definiált szavak között?

# Milyen karakterek használhatóak azonosítókban?

Pascal (ISO 7185:1990):

- betűk ('A'..'Z', 'a'..'z') és számjegyek ('0'..'9')

C++ (ISO/IEC 14882:2017), CLU, Eiffel:

- betűk ('A'..'Z', 'a'..'z'), számjegyek ('0'..'9') és ‚_'

Perl:

- az előzőeken túl $, @ és % jellel is kezdődhetnek
  - $ a skalárokat, @ számmal indexelt tömböt, % asszociatív tömböt jelöl.

ADA

- Ada2012 szabvány : minden, aminek a nevében szerepel, hogy betű vagy számjegy! (Πλάτων, Чайковский stb. is)

# Milyen karakterek használhatóak azonosítókban?

Java:

- A Unicode betűi és számjegyei, a '$' és ‚_'

SWIFT:

- Identifiers begin with
  - an uppercase or lowercase letter A through Z,
  - an underscore (_),
  - a noncombining alphanumeric Unicode character in the Basic Multilingual Plane,
  - or a character outside the Basic Multilingual Plane that isn't in a Private Use Area.
- After the first character, digits and combining Unicode characters are also allowed.
  - `let π = 3.14159`
  - `let 你好 = "你好世界"`
  - `let 🐶🐮 = "dogcow"`

# Milyen karakterek használhatóak azonosítókban?

C#:

- Az azonosítókat a Unicode szabvány ajánlásának megfelelően kell írni.
- Érdekesség, hogy a kulcsszavak a @ bevezető jellel használhatóak azonosítóként is (például: `@bool`)
  - Ez a jel más azonosítót nem vezethet be.
- Ezt a lehetőséget a programok hordozhatósága illetve „átjárhatósága” miatt vezették be.
- A @ prefixű szimbólumok neve valójában @ nélkül kerül fordításra.
  - Így attól függetlenül, hogy egy másik nyelv nem tartalmazza például a „`sealed`” kulcsszót, és egy programban azonosítóként használják, C#-ban lehetőség van az azonosító közvetlen használatára.

# Mi az azonosítók megengedett szintaxisa?

A leggyakrabban:

- letter { letter | digit }
- néha az '_' beszúrása is megengedett.

Pascal: letter{letter|digit}

ADA: letter{[_]letter|[_]digit},

- itt a betűk halmaza nagyobb, minden, aminek a nevében szerepel az, hogy „letter”

Java: Java_letter{[_]Java_letter|[_]digit},

- αβγδε vagy El_Ňiňo

# Van-e hosszúsági megkötés az azonosítókra?

Érdekesség

- Fortran66, Fortran77: maximum. 6 jel
- Fortran90: maximum. 31 jel

Általában tetszőleges hosszúság de, vannak kivételek

- ADA: be kell férjen egy sorba

# Vannak-e „fenntartott” szavak?

Pascal, C, C++, Java, Perl, CLU, ADA, …

- a fenntartott szavak (kulcsszavak) nem használhatók azonosítóként
- Példa C++ további megkötésekre:
  - Each identifier that contains a double underscore __ or begins with an underscore followed by an uppercase letter is reserved to the implementation for any use
  - Each identifier that begins with an underscore is reserved to the implementation for use as a name in the global namespace

# Vannak-e „fenntartott” szavak?

ADA

- különbséget tesz a kulcsszavak és az előredefiniált szavak között, mint például: Integer, True, etc..
- Az előredefiniált szavak átdefiniálhatóak:
  - type Integer is range -999_999..+999_999;
  - így a program implementáció-független
  - de: True: Integer … -- nincs értelme…

Fortran, PL/1

- megengedett a kulcsszavakat azonosítóként használni!

# Literálok

Milyen numerikus literálok vannak?

Milyen más alapok megengedettek a 10-esen kívül?

Mi az egész, illetve valós literálok szintaxisa?

Többsoros string literálok megengedettek-e?

Vezérlőkarakterek string literálokban megengedettek-e, és hogyan írjuk le őket?

# Numerikus literálok

A gyakorlatban csak a '0'..'9’ és az 'A'..'F'  karakterek használhatók

- További lehetőségek:
  - '_' : 456_789
    - ADA, Perl, Eiffel
  - Exponenciális alak: 1E6
    - Ada
  - 'L’ ‘U’ a típus megadására (long, unsigned): 4L
    - C, C++, Java, C#

Mi a számok szerkezete?

- Egészek: decimális:  [-]digit{digit}
  - -123, 456789

# Numerikus literálok

Mi a számok szerkezete?

- Racionális: [-]digit{digit}.digit{digit}[exponent]
  - -123.456E+3
- Majdnem minden nyelv ad ehhez valami specialitást:
  - Pascal
    - [-]digit{digit}[exponent] 12E+3
  - ADA
    - ‘_’ is lehet: -1_234.0
  - Eiffel
    - [-]{digit}.{digit}[exponent] -1.
  - Java
    - „float_suffix” is lehet a végén

# Milyen alapok megengedettek?

Általában: decimális számok, néha más is

- Például 2, 8 16

Példák

- Pascal, Modula, Oberon: csak decimális számok
- C, C++, Java…: Oktális, decimális és hexadecimális
  - oktális: 0734
  - hexadecimális:  0xFF, 0xC0B0L.
- ADA: minden alap 2 és 16 között:
  - base#mantissa#[exponent]: 16#FFF.F#E1
- Mathematica: minden alap 2 és 36 között!
  - Minden számjegy és angol betű
  - basis^^mantissa : 30^^Mathematica = 13 207 019 439 499 57010

# Példa a szabályokra (Java)

IntegerLiteral:
- DecimalIntegerLiteral
- HexIntegerLiteral
- OctalIntegerLiteral
- BinaryIntegerLiteral

DecimalIntegerLiteral:
- DecimalNumeral [IntegerTypeSuffix]

HexIntegerLiteral:
- HexNumeral [IntegerTypeSuffix]

OctalIntegerLiteral:
- OctalNumeral [IntegerTypeSuffix]

BinaryIntegerLiteral:
- BinaryNumeral [IntegerTypeSuffix]

IntegerTypeSuffix:
- (one of)
- l L

DecimalNumeral:
- 0
- NonZeroDigit [Digits]
- NonZeroDigit Underscores Digits

NonZeroDigit:
- (one of)
- 1 2 3 4 5 6 7 8 9

Digits:
- Digit
- Digit [DigitsAndUnderscores] Digit

Digit:
- 0
- NonZeroDigit

DigitsAndUnderscores:
- DigitOrUnderscore {DigitOrUnderscore}

DigitOrUnderscore:
- Digit
- _

Underscores:
- _ {_}

# Karakterek és karaktersorozatok

Karakterek:

- 'A' vagy „escape szekvenciák”: '\n’, '\'’, '\ooo' stb.

Stringek "A”

Tárolásuk különböző lehet.

# Megjegyzések

Szerkezete befolyásolja a program megbízhatóságát

Szokásos lehetőségek:

- egy speciális oszloptól (jeltől) a sor végéig
- speciális jel(ek) elején – végén
- speciális jel(ek) elején – vége a sor vége

# Megjegyzések

## Példák

- Pascal:
  - (* és *)
  - { és }
- CLU, Matlab
  - %-tól sor végéig
- Python
  - #-tól a sor végéig
  - ’’ és ’’ között

## Példák

- C:
  - /* és  */
- C++: mint C-ben, és még
  - // -tól sor végéig
- Java: mint C++-ban és még
  - /** documentation */
- C#: mint C++-ban és még
  - /// egysoros dokumentációs megjegyzés
- ADA, Eiffel:
  - -- -tól sor végéig

# Beépített adattípusok, változók, kifejezések

# Mit jelent a nyelvben az adattípus?

Egy értékhalmaz és egy művelethalmaz

Specifikáció – Reprezentáció – Implementáció

Példa

- Specifikáció
  - egészek -∞ -től +∞ -ig, és a megengedett műveletek +, - *, stb.
- Reprezentáció
  - 8, 16, 32, 64-bit, előjeles kettes komplemens kóddal (megszorítások)
- Implementáció
  - a műveletek megvalósítása

# Típus a programozó szemszögéből

Típusspecifikáció („mit”)

- alaphalmaz: a valós világ minket érdeklő objektumainak halmaza
- specifikációs invariáns ($I_S$) – ezt a halmazt tovább szűkíti
- típusműveletek specifikációja – csak a „mit”!

# Típus a programozó szemszögéből

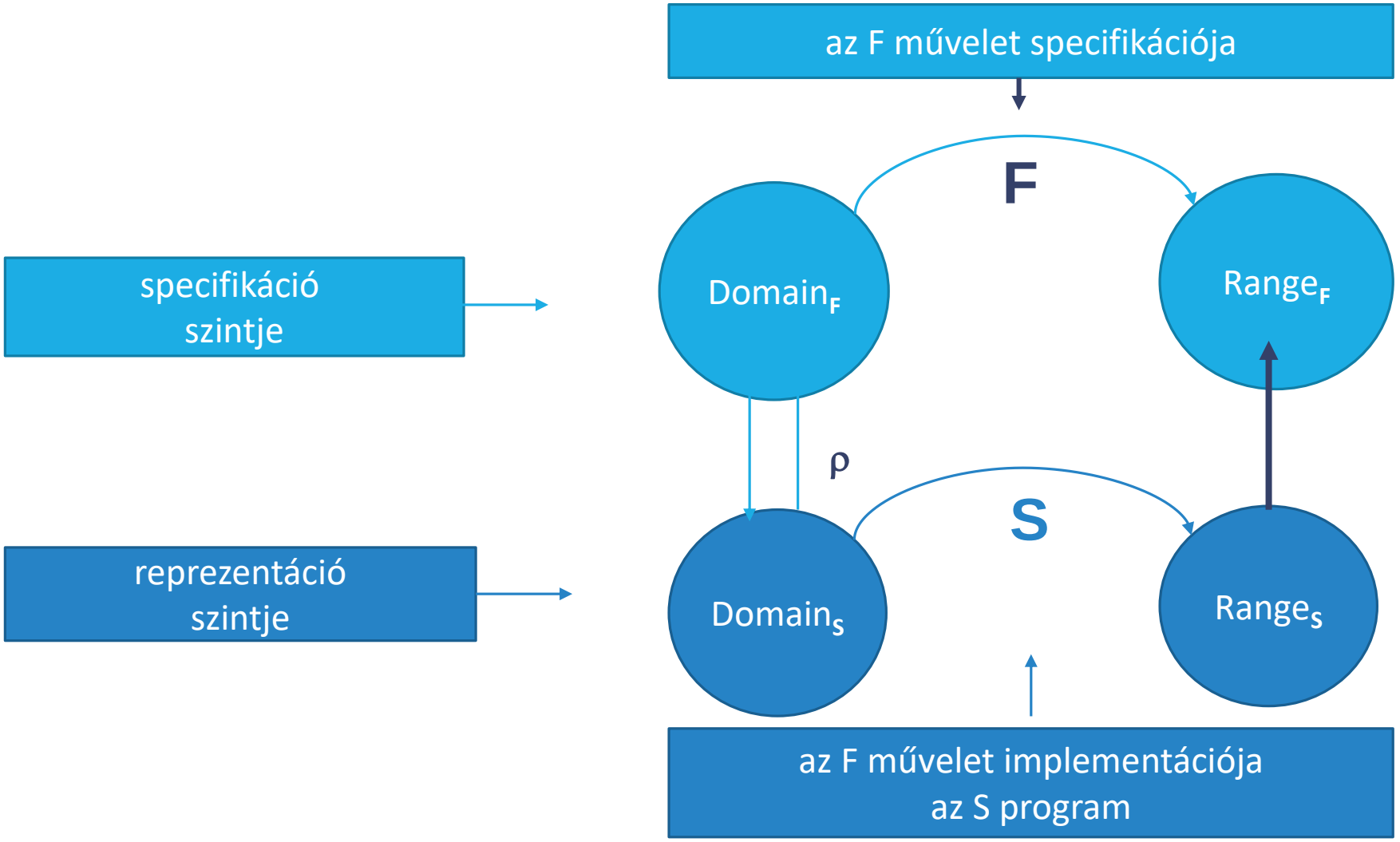
Típus megvalósítás („hogyan”)

- Reprezentációs függvény
- Típus invariáns
- Típusműveletek megvalósítása

Például

- Komplex számok
- Verem
- stb.

# A típus specifikáció és a típus kapcsolata



# Típusok támogatása

A legtöbb programozási nyelvben van valamilyen adattípusra támogatás.

Vannak „beépített” („built-in”) típusok, itt

- a specifikáció – a programozási nyelv referencia kézikönyvében
- a reprezentáció és az implementáció a fordítóprogrammal jön.
- mennyire szabványosak a beépített típusok?

Vannak megengedett típuskonstrukciók

- tömb, rekord, stb.

A programozási nyelvek gyakran lehetővé teszik új adattípusok tervezésekor, hogy önálló modulokban elrejtsük a reprezentációt és az implementációt

# A programozási nyelvek típusszerkezete

Hogyan osztályozhatjuk a típusokat?

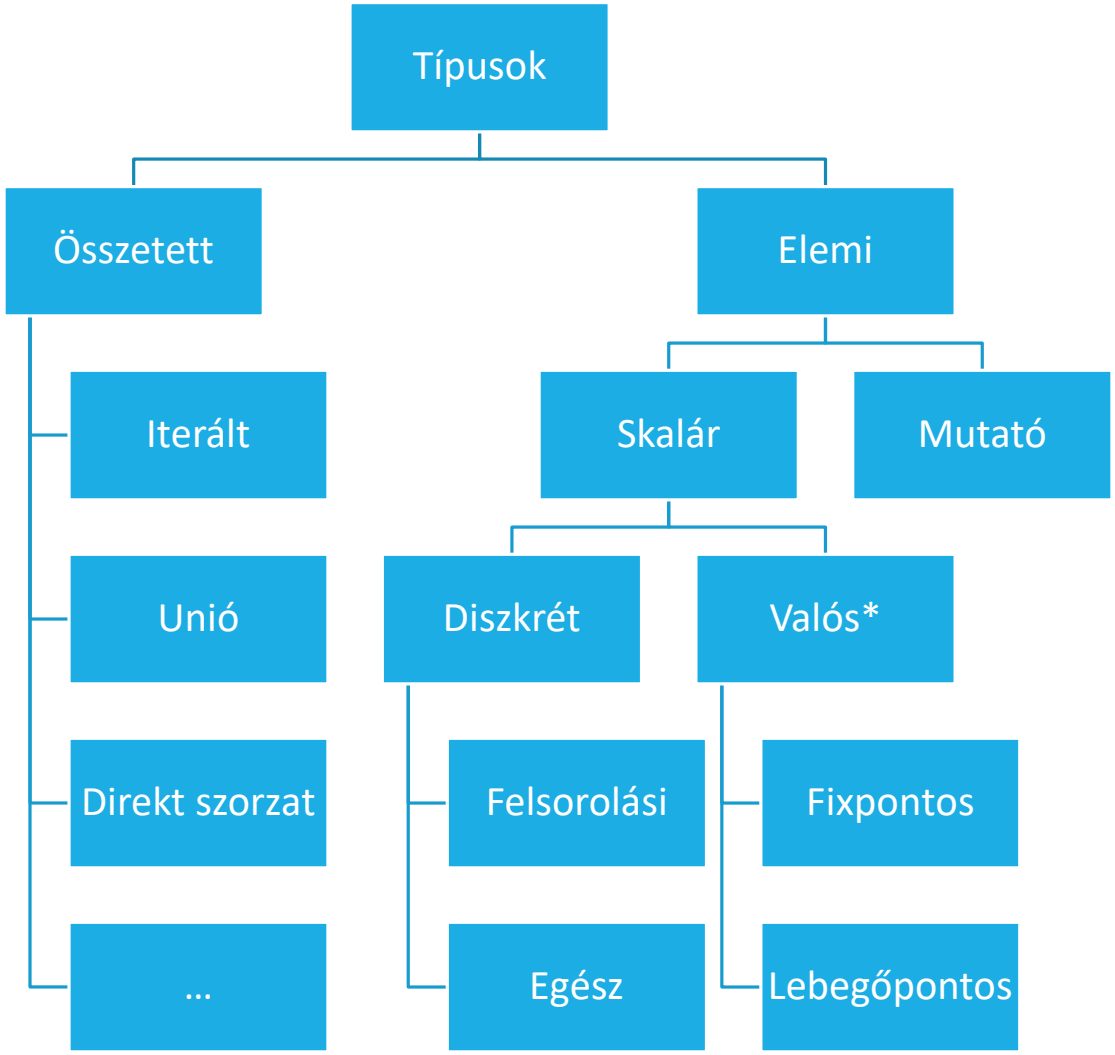
Vannak típusosztályok a nyelvben?

- Vannak típusosztályokra vonatkozó műveletek?

Elemi és összetett típus

- Elemi típusoknak nevezzük azokat a típusokat, amelyek logikailag felbonthatatlanok
- Az összetett típusokat már meglévő típusokból, mint komponensekből hozzuk létre

# Típusosztályok



# Példák

# Típusosztály műveletekre példa

Ada – felsorolási típus

- `type Hónapok is (Január, Február, Március,…., December);`
- automatikusan keletkezik:
  - `Hónapok'First, Hónapok'Last,`
  - `Hónapok'Range,`
  - `Hónapok'Min(x,y), Hónapok'Max(x,y)`
  - `Hónapok'Succ(x), Hónapok'Pred(x)`
  - `Hónapok'Image(x)`
  - `Hónapok'Value(x)`
  - `Hónapok'Val(x)`
  - `Hónapok'Pos(x)`
  - …

# Van-e speciális tulajdonsága a nyelv típusrendszerének?

## ADA

- Különbség az altípus és a származtatott típus között:
  - Egy típus altípusa a típus értékhalmazának részhalmazát jelöli.
  - Az altípusra alkalmazható a típus összes művelete.
- Például:
  - Dátum – a hónap napjai az 1 .. 31 intervallumba esnek.
  - Ehhez deklarálhatunk egy altípust:
    - `subtype Napok is Integer range 1 .. 31;`

# Van-e speciális tulajdonsága a nyelv típusrendszerének?

## ADA

- Új típus létrehozása már létező típusból: származtatás
  - itt kicsit mást jelent, mint az OOP-ben
  - `type Hosszúság is new Float;`
  - A származtatott típus átveszi az eredeti típus struktúráját, értékhalmazát és műveleteit, átveszi a kezdeti értékeit is.
    - De a két típus nem ekvivalens!
  - `type Terület is new Float;`
    - `H: Hosszúság; T: Terület;`
    - `H := T;` hibás! (már szintaktikailag!)
  - Konverzió lehetősége szükséges:
    - `function "*" (x, y: Hosszúság) return Terület is`
    - `  begin return Terület (Float(x) * Float(y)); end;`
    - `T:=H*H;` írható, de `T:=H;` vagy `T:=H/H;` továbbra sem!

# Van-e speciális tulajdonsága a nyelv típusrendszerének?

Java:

- Kétféle típus létezik: a primitív típusok (numerikus és logikai), és a referencia típusok.
- A felhasználó által definiált típusok az osztályok.
- A primitív típusoknak van „csomagoló” osztálya, boxing, unboxing lehetőségekkel

Eiffel:

- alapértelmezés szerint az értékek objektumokra vonatkozó referenciák, de definiálhatunk kiterjesztett (expanded) típusokat is, ezek változói maguk az objektumok.
- Az alap típusok mindig kiterjesztettek.
- Új típus = új osztály létrehozása.

C#: érték és referencia típusok.

- Érték típusok: egyszerű típusok (pl. char, int, float), felsorolási és struct típusok.
- Referencia típusok - osztály (class), interface típusok,  delegate és tömb típusok.
- Boxing – unboxing lehetőségek vannak itt is.

# Mik a beépített adattípusok?

Az alaptípusok általában, a szokásos műveletekkel

- az egészek
- a karakterek
- a logikai típus
- a valósak

# A skalár típusok osztálya

A skalár típusok a diszkrét és a valós típusok.

- Értékei rendezettek, így a relációs operátorok (<, <=, =, >=, >) előredefiniáltak ezekre a típusokra.
- ADA–ban számos attribútum:
  - `S'First, S'Last, S'Range`
  - `S'Min, S'Max`
  - `S'Succ, S'Pred (Constraint_Error)`
  - `S'Image , S'Width, S'Value stb.`
  - `Hónapok'First = Január`
  - `Hónapok'Succ(Március)= Április`

# A skalár típusok osztálya

Eiffelben ezek a típusok mindig kiterjesztettek

C++ – ezek az „integral” típusok

Perl

- Skalár adattípus
  - a neve kötelezően $ jellel kezdődik
  - számok, stringek (!) és referenciák tárolására, a típusok közötti (numerikus, string,...) konverzió automatikus
  - `$skalar1 = 'string';  # karakterlanc`
  - `$skalar2 = 1234;      # integer`
  - `$skalar3 = 4.5;       # float`
  - `$skalar4 = $skalar3;  # ertekadas`
  - `$skalar5 = \$skalar2; # referencia`

# A diszkrét típusok osztálya

A diszkrét típusok a

- felsorolási és az
- egész típusok

# Felsorolási típusok

A típusértékhalmaz megadható egy explicit felsorolással

- `type Days is (Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday);`

Gyakran a karakter és a logikai típus is előredefiniált felsorolási típus.

- A logikai értékek kezelésére szolgáló típust sok programozási nyelvben `Boolean`–nak hívják
  - két lehetséges értéke van, a `True` és a `False`,
  - a szokásos műveletek a `not`, `and`, `or` és a `xor`.
- De vannak kivételek
  - Például `Yes`, `No` Objective-C esetén

# Felsorolási típusok

Object Pascal

- Egy felsorolási típus definíciója:
- `type Flower = (Rose, Tulip, Violet, Geranium);`
- Az azonosítóknak különbözőeknek, és a programban egyedieknek kell lenniük.
- Standard függvények
  - `Pred(X)`
  - `Succ(X)`
  - `Dec(X)`
  - `Inc(X)`
  - `<`
  - …

# Felsorolási típusok

## Object Pascal

- Logikai
  - `Boolean`, `ByteBool` 1 byte
  - `WordBool` 2 byte
  - `LongBool` 4 byte
  - `ByteBool`, `WordBool`, `LongBool` : numerikus `boolean` típusok, (minden numerikus értéket logikaiként értelmez) a nemnulla a `True`
    - `-1 true`, `0 false`
- Karakter típusok
  - `ANSIChar` (1 byte),
    `WideChar` (2 byte, Unicode)

# Felsorolási típusok

ADA

- Minden felsorolási literál a felsorolási típus egy önálló típusértéke, és van egy pozíció száma.
  - Az első felsorolási literál pozíció száma 0.
  - A rendezési relációt a felsorolás sorrendje adja. Egy felsorolási típus értékei lehetnek azonosítók vagy karakterek.
  - `type Color is (Red, Blue, Green);`
  - `type Roman_Digit is ('I','V','X','L','C','D','M');`
- Megengedett a felsorolási nevek átlapolása, és ha szükséges, a típusnév segít a megkülönböztetésben: `Color'(Red)`.

# Felsorolási típusok

## ADA

- Három előredefiniált  karakter típus van
  - `Character` – ennek az értékei az ISO 10646 Basic Multilingual Plane (BMP) Row 00 256 kódpozíciójának (Latin-1)
  - `Wide_Character` típus értékei pedig az ISO 10646 Basic Multilingual Plane (BMP) 65536 kódpozíciójának
  - `Wide_Wide_Character` típus értékei pedig az ISO/IEC 10646:2011 2 147 483 648 kódpozíciójának felelnek meg.
- Logikai
  - `type Boolean is (False, True);`
    - speciális előredefiniált típus a szokásos műveleteken kívül az
    - "`and then`" és az
    - "`or else`" műveletekkel  (lusta kiértékelés).

# Felsorolási típusok

C++

- Az „enum” kulcsszót használják a felsorolási típusok, a megfelelő egész érték 0-val kezd, egyesével nő, kivéve, ha „= expr” szerepel valahol
  - `enum color {red, blue, green=20, yellow);`
- Az előredefiniált `char`, `signed char` és `unsigned char` típusok írják le a lehetséges karakter típusokat, sok probléma származik abból, hogy ezeket nem szabványosították igazán.
- Előredefiniált logikai típus a `bool` a `false` és `true` értékekkel, kifejezésekben a 0 értéket is hamis-ként kezeli
  - Minden nem 0-t igaznak.

# Felsorolási típusok

Java

- Nem volt felsorolási típus – a Java 5.0-ig!
- Az előredefiniált `char` típus a 16 bites Unicode character-készletet támogatja '\u0000' –tól '\uffff'-ig,
  azaz 0-tól 65535-ig.
- Előredefiniált logikai típus, a `boolean` a `false` és `true` értékekkel
- A műveletek
  - Relációs: `==` `!=`
  - not: `!`
  - and, or: `&&` `||`
  - and, or, xor (bitwise): `&` `|` `^`
  - feltételes kifejezés: `?` `:`

# Felsorolási típusok

Java

- `enum Season { WINTER, SPRING, SUMMER, FALL }`
- Lehet állapota is és műveletei is
  - https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html
  - Minden enum típusnak ez a (rejtett) közös őse
  - ```
    public abstract class Enum<E extends Enum<E>>
                  extends Object
                  implements Comparable<E>, Serializable
    ```
    ```

# Példa

```
import java.util.*;
public class Card {
      public enum Rank { DEUCE, THREE, FOUR, FIVE, SIX,
          SEVEN, EIGHT, NINE, TEN, JACK, QUEEN, KING, ACE }

      public enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES }

      private final Rank rank;
      private final Suit suit;
      private Card(Rank rank, Suit suit) {
          this.rank = rank;
          this.suit = suit;
      }

      public Rank rank() { return rank; }
      public Suit suit() { return suit; }
      public String toString() { return rank + " of " + suit; }

      private static final List<Card> protoDeck = new ArrayList<Card>();

       // Initialize prototype deck
      static {
          for (Suit suit : Suit.values())
             for (Rank rank : Rank.values())
                 protoDeck.add(new Card(rank, suit));
      }
```

Használja a Rank és a Suit toString-jét

Használja a Rank és a Suit values műveletét

# Tulajdonságok és viselkedés

```
public enum Planet {
 MERCURY (3.303e+23, 2.4397e6),
 VENUS (4.869e+24, 6.0518e6),
 EARTH (5.976e+24, 6.37814e6),
 MARS (6.421e+23, 3.3972e6),
 JUPITER (1.9e+27, 7.1492e7),
 SATURN (5.688e+26, 6.0268e7),
 URANUS (8.686e+25, 2.5559e7),
 NEPTUNE (1.024e+26, 2.4746e7);

 private final double mass; // in kilograms
 private final double radius; // in meters

 Planet(double mass, double radius) {
  this.mass = mass;
  this.radius = radius;
 }
 private double mass() { return mass; }
 private double radius() { return radius; }
```

# Tulajdonságok és viselkedés

```
// universal gravitational constant  (m3 kg-1 s-2)
public static final double G = 6.67300E-11;

double surfaceGravity() {
 return G * mass / (radius * radius);
}

double surfaceWeight(double otherMass) {
 return otherMass * surfaceGravity();
}
```

Próba

```
public static void main(String[] args) {
   double earthWeight = Double.parseDouble(20.0);
   double mass = earthWeight/EARTH.surfaceGravity();
   for (Planet p : Planet.values())
     System.out.printf("Your weight on %s is %f%n", p,
                                                  p.surfaceWeight(mass));
}
```

# Tulajdonságok és viselkedés – Absztrakt művelet

```
public enum Operation {
  PLUS   { double eval(double x, double y) { return x + y; } },
  MINUS  { double eval(double x, double y) { return x - y; } },
  TIMES  { double eval(double x, double y) { return x * y; } },
  DIVIDE { double eval(double x, double y) { return x / y; } };
   // Do arithmetic op represented by this constant
  abstract double eval(double x, double y);
}
```

Próba

```
public static void main(String args[]) {
  double x = Double.parseDouble(args[0]);
  double y = Double.parseDouble(args[1]);
  for (Operation op : Operation.values())
    System.out.printf("%f %s %f = %f%n", x, op, y, op.eval(x, y));

}
```

# Felsorolási típusok

## C# - Enum

- Rendelkezik alaptípussal – ez tetszőleges integral típus lehet
  - `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`
- Deklaráció példa:

```
enum Color: long {
    Red,
    Green,
    Blue
}
```

- Alapértelmezésben int
  - hozzárendelt egész-értékek 0-val kezdődnek, ez átállítható
- `System.Enum` a közös ősosztály

# Felsorolási típusok

C# logikai típus:

- `bool` a `true` és `false` értékekkel, szokásos műveletek
- Nincs szabványos konverziós lehetőség!

C# karakterek:

- char típus unsigned 16-bit egészek 0 és 65535 között.
- A Unicode karakter-halmaznak felel meg.
- Bár ugyanaz a reprezentációja, mint az ushort-nak, nem ugyanaz a művelethalmaz!
- A konstansokat kétféle módon lehet írni: karakter-literálként vagy egész literálként, explicit típuskényszerítéssel: `(char)10` ugyanaz, mint `'\x000A'`.

# Felsorolási típusok

SWIFT

- Alaptípus megadása
- Hozzárendelt reprezentációs érték megadható
  - Értékek lekérdezhetők
- Példa

```
enum Rank: Int {
    case Ace = 1
    case Two, Three, Four, Five, Six, Seven, Eight, Nine, Ten
    case Jack, Queen, King
}
let ace = Rank.Ace
let aceRawValue = ace.rawValue
```

# Egész típusok

Az egész típusok nagyon közel vannak a számítógépes reprezentációhoz.

A műveletek általában a szokásosak, néhol gond van az osztással és a hatványozással

# Egész típusok

Object Pascal lehetőségek:

- `ShortInt` signed8 bit
- `SmallInt` signed 16 bit (csak Delphi 2, 3)
- `Integer` signed system-dep. Delphi 1 – 16 bit
  Delphi 2-től – 32 bit
- `LongInt` signed32 bit
- `Byte` unsigned 8 bit
- `Word` unsigned 16 bit
- `Cardinal` unsigned system-dep. Delphi 1 – 16 bit
  Delphi 2-től – 32 bit

# Egész típusok

ADA:

- Előjeles egészek:
  - `type Page_Num is range 1 .. 2_000;`
  - `type Line_Size is range 1 .. Max_Line_Size;`
- Maradékosztályok:
  - `type Byte is mod 256; -- egy előjel nélküli byte`
  - `type Hash_Index is mod 97;`
- Minden egész típust úgy tekintenek, mint ami a névtelen, előredefiniált `root_integer` típusból lett származtatva.
- Egész literálok ennek az `universal_integer` osztályába tartoznak.
  - Ez a szigorú típusosság miatt fontos.

# Egész típusok

ADA:

- Az előredefiniált egész típus az Integer, van két előredefiniált altípusa:
  - `subtype Natural is Integer range 0 .. Integer'Last;`
  - `subtype Positive is Integer range 1 .. Integer'Last;`
- Az értékintervallumnak egy tetszőleges implementáció esetén tartalmaznia kell a –2**15+1 .. +2**15–1 –t.
- Megengedett, de nincs előírva:
  - `Short_Integer`
  - `Long_Integer`
  - `Short_Short_Integer`
  - `Long_Long_Integer`
  - …

# Egész típusok

ADA:

- A műveletek a szokásosak
  - minden attribútum,
  - relációs operátorok,
  - +, -,(unáris, bináris) *, / (csonkít),
  - rem (maradék),
  - mod (modulus),
  - abs,
  - ** (hatványozás Natural kitevőre)
- A modulo típusok műveletei a maradékosztályokon

# Egész típusok

C++:

- Az előredefiniált egész típusoknak 4 mérete lehet: `short int, int, long int, long long int`.
- „Longer integers provide no less storage than shorter ones.”
- Az `unsigned int`, `unsigned long int`, `unsigned short int` típusokat modulo $2^n$ aritmetikával használja (n a reprezentációban a bitek száma).
- A szabvány definiálja a minimum értékintervallumokat
- C++11 fix méretű típusok
  - `int8_t, int16_t, int32_t, int64_t`

# Egész típusok

C++

- A műveletek:

| | |
|---|---|
| Relációs | == , != < <= > >= |
| Unáris | * + - & |
| Multiplikatív | * / % |
| Additív | + - |
| incr. prefix postfix | ++ |
| decr. prefix postfix | -- |
| shift előjeles | << >> |
| Komplemens bitenként | ~ |
| Feltételes op. | ? : |
| Sizeof | |
| pointer_to_member | ->* .* |
| Értékadó op. | = *= /= %= += -= >>= <<= &= ^= != |

# Egész típusok

Java

- Az előredefiniált egész típusoknak előírt specifikációja van:

| | |
|---|---|
| byte | -128..127 |
| short | -32768..32767 |
| int | -2147483648.. 2147483647 |
| long | -9223372036854775808 .. 9223372036854775807 |
| char | '\u0000'..'\uffff', vagyis: 0..65535 |

# Egész típusok

Java

- Operátorok

| | |
|---|---|
| Relációs | == , != < <= > >= |
| Unáris | +- |
| Multiplikatív | * / % |
| Additív | + - |
| incr. Prefix postfix | ++ |
| decr. Prefix postfix | -- |
| shift előjeles, előjeln. | << >> >>> |
| komplemens bitenként | ~ |
| feltételes op. | ? : |

- További hasznos előredefiniált műveletek a csomagoló osztályokban

# Egész típusok

C#

- Előírt specifikáció van itt is
  - `sbyte` signed 8-bit $-128 \dots 127$
  - `byte` unsigned 8-bit $0 \dots 255$
  - `short` signed 16-bit $-32768 \dots 32767$
  - `ushort` unsigned 16-bit $0 \dots 65535$
  - `int` signed 32-bit $-2147483648 \dots 2147483647$
  - `uint` unsigned 32-bit $0 \dots 4294967295$
  - `long` signed 64-bit $-9223372036854775808 \dots 9223372036854775807$
  - `ulong` unsigned 64-bit $0 \dots 18446744073709551615.$
- Műveleteknél előírták a konverziókat is.

# Egész típusok

Eiffel:

- Az `INTEGER` kiterjesztett, a `COMPARABLE` és a `NUMERIC` osztályok leszármazottja.
- A reprezentáció legalább `Integer_bits` bitet használ, ez egy platform-függő konstans, amelyet a PLATFORM osztály definiál.
- A műveletek:
  - `< <= > >=` a `COMPARABLE`-ból,
  - `+ - * /` a `NUMERIC`-ból,
  - és az újak:
    - hatványozás `^`
    - egész osztás `//`
    - maradék `\\`

# Valós típusok

A valós típusok a valós számok közelítései.

- a lebegőpontos típusok - relatív pontosság
- fixpontos típusok - abszolút pontosság

A legtöbb programozási nyelv támogatja az előjeles lebegőpontos típusokat, ahol 1 bit az előjel, és a szám formátuma:




- mantissa * $10^{exponent}$, ahol  0<=abs(mantissa)<10.

A kitevőnek is lehet előjele

- A kitevőt általában az 'E' betű jelöli.

# Valós típusok

## Pascal

- A nyelv lebegőpontos valósakat használ.

| Típus: | Reprezentáció: |
|---|---|
| Single | 32 bit (1+23+8)<br>1.401E-45 .. 3.402E38 |
| Real | 48 bit (1+39+8)<br>2.9E-39 .. 1.7E38 |
| Double | 64 bit (1+52+11)<br>4.941E-324 .. 1.797E308 |
| Extended | 80 bit (1+64+15)<br>3.363E-4932 ..1.189E4932 |
| Comp | 64 bit (1+63)<br>-9.2E-18 .. 9.2E18 |

# Valós típusok

Object Pascal:

- `Single`, `Double`, `Extended`
  - IEEE nemzetközi szabvány szerint.
- Fixpontos valós típus 4 számjegy pontossággal:

| | | |
|---|---|---|
| Currency | -922337203685477.5808 .. 922337203685477.5807 | 8 byte |

  - Műveletek a szokásosak

# Valós típusok

ADA:

- A nyelv ad lehetőséget a lebegőpontos és a fixpontos típusok kezelésére, de csak egy előredefiniált lebegőpontos típus van, a `Float`.
- A lebegőpontos típusoknál a relatív pontosság, míg a fixpontos típusoknál az abszolút pontosság megadására van lehetőség. Megadható egy értékintervallum is:

```
type Real is digits 8;
type Coefficient is digits 10 range -1.0 .. 1.0;
type Mass is digits 7 range 0.0 .. 1.0E35;
```

# Valós típusok

ADA:

- Minden valós típust úgy tekintenek, mint egy előre definiált `root_real` típusból származtatott típust. A valós literálok ennek az osztályába tartoznak, így `universal_real` típusúak.
- Ha egy implementációban a lebegőpontos típusok pontossága legalább 6 számjegy, akkor a `Float` típus pontossága is legalább ennyi kell legyen.
- Megengedett, hogy egy implementáció támogasson további előre definiált lebegőpontos típusokat, pl.: `Short_Float`, `Long_Float`, `Short_Short_Float`, `Long_Long_Float`...
- A szokásos műveletek, minden attribútum, relációs operátorok, `+`, `-`,(unary, binary) `*`, `/` `abs`, `**` (hatványozás).

# Valós típusok

C++:

- Az előre definiált valós típusok a `float`, `double` és a `long double`.
- „The type double provides no less precision then float, and the type long double provides no less precision then double.” …

# Valós típusok

C++:

- Műveletek

| | |
|---|---|
| egyenlő, nem egyenlő összehasonlítás | == , != < <= > >= |
| indirekció, előjel, cím | * + - & |
| szorzás, osztás, moduló | * / % |
| összeadás, kivonás | + - |
| növelés prefix/postfix | ++ |
| csökk. prefix/postfix | -- |
| léptetés balra/jobbra | << >> |
| bitenkénti komplemens | ~ |
| feltételes kifejezés | ? : |
| obj/típus mérete | sizeof |
| tagkiválasztás | ->* .* |
| értékadások | = *= /= %= += -= >>= <<= &= ^= != |

# Valós típusok

Java:

- Az előredefiniált lebegőpontos típusoknak előírt pontossága van

| | |
|---|---|
| float | 32 bit IEEE 754 |
| double | 64 bit IEEE 754 |

- A `float` típus $s*m*2^e$ alakú, ahol
  - $s = +1$ vagy $-1$,
  - $m$ egy pozitív egész, kisebb, mint $224$
  - $e$ egy egész a $-149..104$ intervallumból.
- A `double` típus $s*m*2^e$ alakú, ahol
  - $s = +1$ vagy $-1$,
  - $m$ egy pozitív egész, kisebb, mint $253$
  - $e$ egy egész a $-1075..970$ intervallumból

# Valós típusok

Java:

- Érdekesség:
  - a pozitív és negatív végtelenek (`POSITIVE_INFINITY`, `NEGATIVE_INFINITY`)
  - speciális Not-a-Number (`NaN`) érték is szabvány.
    - Ha x a `NaN` mi az értéke az x != x -nek?
- Műveletek a `NaN`–ra nem értelmezettek
  - További hasznos műveletek a `Float`, `Double` és `Math` osztályokban.

# Valós típusok

Java:

- Műveletek a `NaN`–ra nem értelmezettek
  - További hasznos műveletek a `Float`, `Double` és `Math` osztályokban.

| | |
|---|---|
| Relációs | == , != < <= > >= |
| Unáris | +- |
| Multiplikatív | * / % |
| Additív | + - |
| Incr. prefix, postfix | ++ |
| Decr. prefix, postfix | -- |
| Shift előjeles és előjel nélküli | << >> >>> |
| Komplemens bitenként | ~ |
| Feltételes op. | ? : |

# Valós típusok

C#:

- Javához hasonló - 2 lebegőpontos típus: `float` és `double`

| | |
|---|---|
| float | 32 bit IEEE 754 |
| double | 64 bit IEEE 754 |

- Pozitív és negatív 0
  - legtöbbször ugyanaz, de osztásnál más lehet.
- Pozitív és negatív  végtelen (∞)
  - 1.0 / 0.0 a pozitív végtelen
  - –1.0 / 0.0 a negatív.
- A Not-a-Number érték (NaN).
  - 0.0/0.0

# Valós típusok

C#

- Decimal típus a pénzügyi számításokhoz:
  - 128-bites adattípus,
    értékei: $1.0 \times 10^{-28}$ … $7.9 \times 10^{+28}$,
    28-29 szignifikáns számjeggyel.
  - Lényeg, hogy a tízes számrendszerbeli törtértékeket is pontosan ábrázolja, műveleteknél szokásos módon kerekít.

# Valós típusok

Eiffel

- A REAL osztály kiterjesztett, a `COMPARABLE` és a `NUMERIC` osztályok leszármazottja.
- A reprezentáció legalább `Real_bits` bitet használ, ez egy platform-függő konstans, amelyet a `PLATFORM` osztály definiál.
- A műveletek
  - `< <= > >=` a `COMPARABLE`-ból,
  - `+ - * /` a `NUMERIC`-ból,
  - és az újak
    - hatványozás ^
    - stb.

# Típusok Vezérlési szerkezetek

KÖVETKEZŐ HÉTEN




# Programozási nyelvek és módszerek

# Pointerek

# Pointer és referencia típusok

Egy pointer (és egy referencia) egy olyan objektum, amely megadja egy másik objektum címét a memóriában

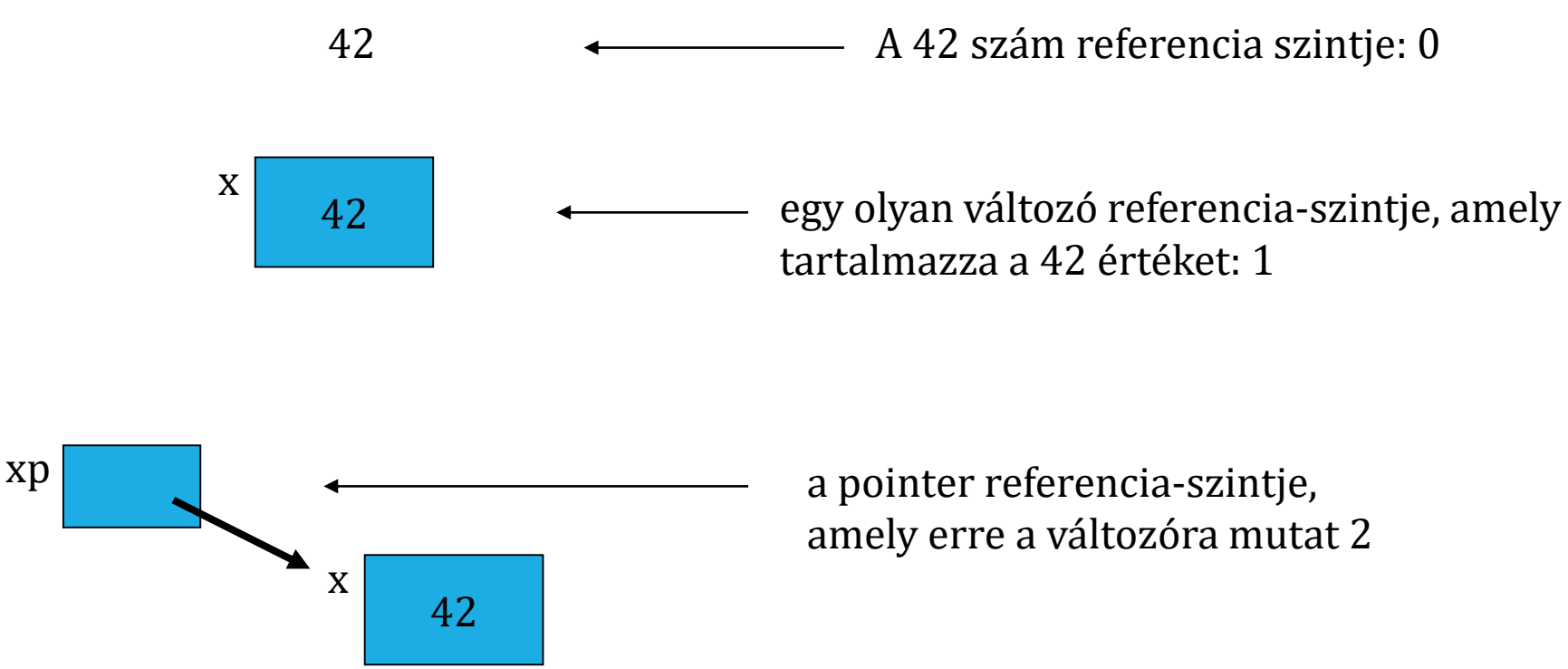
Egy pointer értéke egy memóriacím

- gépi nyelvekben az indirekt címzés lehetősége motiválta a pointerek létrehozását.

Vannak típusos és típus nélküli pointerek.

# Referencia szint

Pointerek segítségével magasabb referencia-szinten hivatkozhatunk objektumainkra.



# Mire kellenek a mutatók?

Hatékonyság

- ahelyett, hogy nagy adatszerkezeteket mozgatnánk a memóriában, sokkal hatékonyabb, ha az erre mutató pointert másoljuk, mozgatjuk.

x

| 23 | 10 | 11 | 17 | 0 | 34 | 28 | 22 | 55 | 88 |
|---|---|---|---|---|---|---|---|---|---|
| 4 | 7 | 0 | 0 | 6 | 8 | 1 | 9 | 10 | 3 |

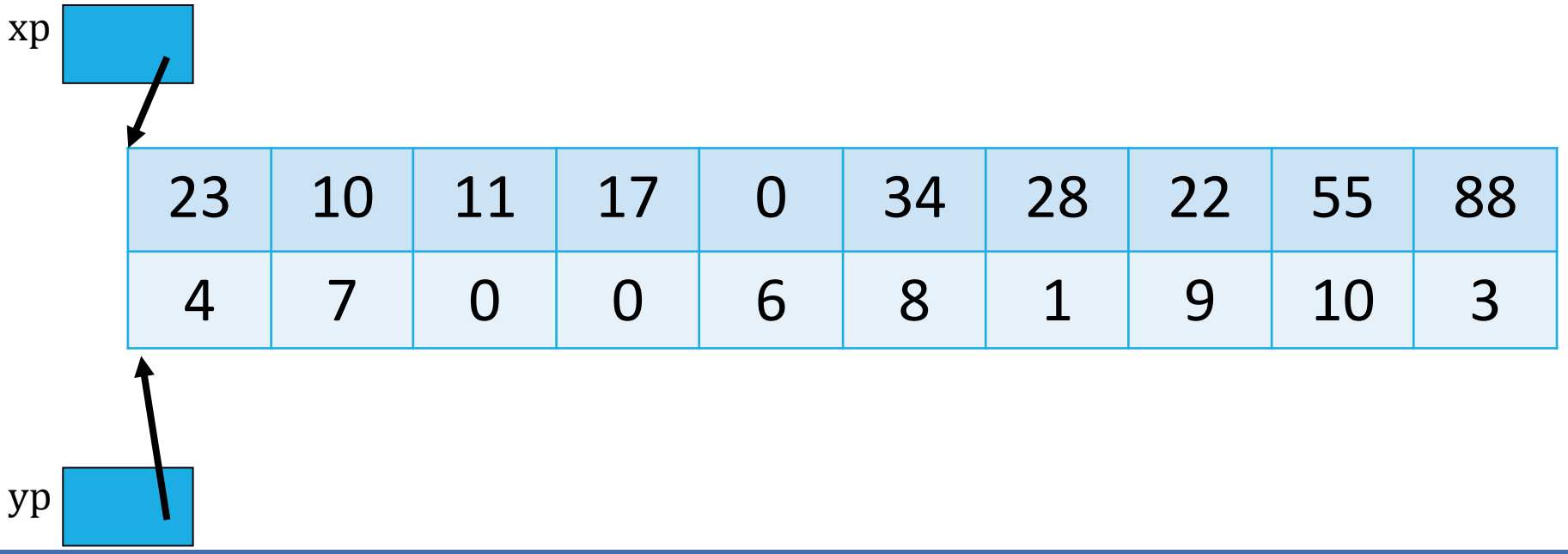
y

| 23 | 10 | 11 | 17 | 0 | 34 | 28 | 22 | 55 | 88 |
|---|---|---|---|---|---|---|---|---|---|
| 4 | 7 | 0 | 0 | 6 | 8 | 1 | 9 | 10 | 3 |

# Mire kellenek a mutatók?

Hatékonyság

- ahelyett, hogy nagy adatszerkezeteket mozgatnánk a memóriában, sokkal hatékonyabb, ha az erre mutató pointert másoljuk, mozgatjuk.
  - itt  vigyázni kell az osztott használatra! – jó, ha van read-only elérési lehetőség is



# Mire kellenek a mutatók?

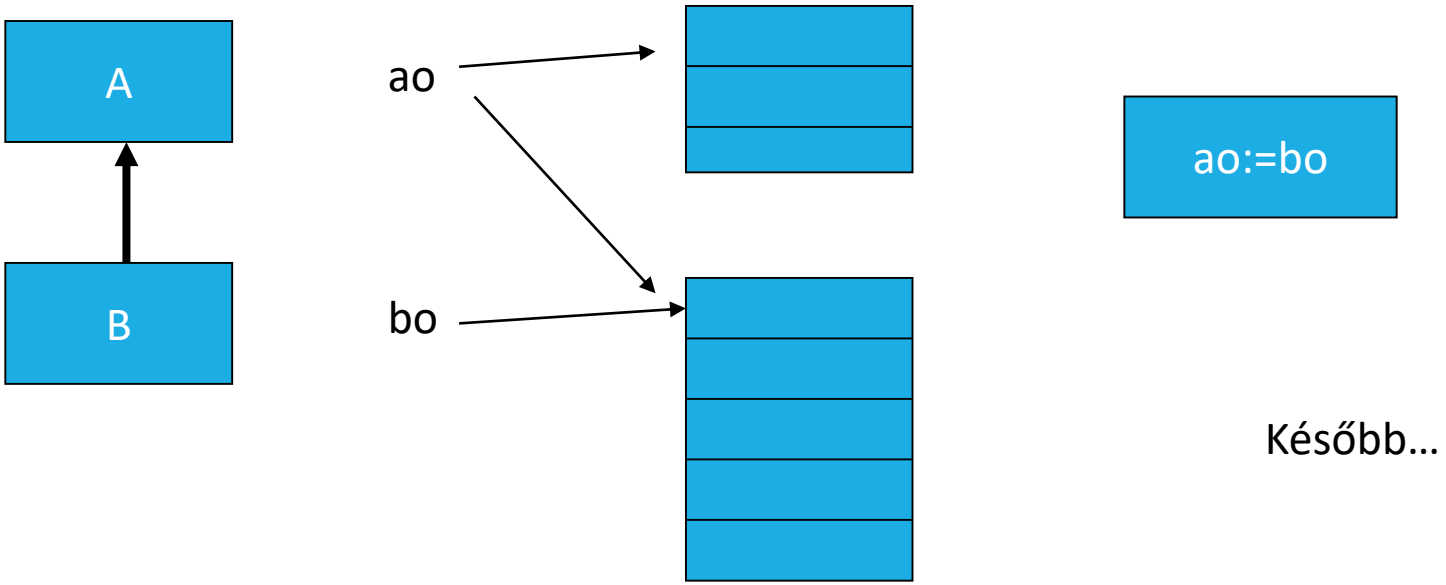
Dinamikus adatszerkezetek építéséhez



# Mire kellenek a mutatók/referenciák?

Objektumorientált funkciókhoz

- a programozási nyelvekben a polimorfizmust akkor tudjuk támogatni, ha a változók objektumokra való referenciákat tartalmaznak.



# A szokásos műveletek

Értékadás - pointerek között, hivatkozott objektumok között (copy, clone, deep-copy, deep-clone!)

Egyenlőség vizsgálat - ha két ugyanolyan típusú pointer ugyanarra az adatszerkezetre mutat (ez is több szinten lehet)

Dereferencing - a *mutatott* objektum részére vagy egészére való hivatkozás

Referencing - egy objektum címe

Új objektum dinamikus allokálása

Egy objektum deallokálása - explicit művelettel vagy implicit módon egy garbage collector-ral

Néha (pl. C, C++) összeadás, kivonás is megengedett

# „Csellengő” pointerek

„Csellengő” pointer: kísérlet olyan változó elérésére, ami már nem létezik.

```
#include <iostream>
int *r;
double *r2;
void f() {int v; r=&v; }
void g() {double v; v=2.1; r2=&v; }
int main() {
  f(); g(); *r=3; *r2=1.2;
  cout << "dangling *r="<<*r;
  cout << "\n *r2 " <<*r2 << ".\n";
}
```

**Compiler, builder:**
**0 error(s), 0 warning(s)**
**Az eredmény megjósolhatatlan!**

# „Csellengő” pointerek 2.:

```
int main() {
   int *j,*i;
   double *d;
   j=new int;
   *j=3;
   i=j;
   delete j;
   d=new double;
   *d=4.2;
   cout << *i;
}
```

**Compiler, builder:**
**0 error(s), 0 warning(s)**
**Az eredmény megjósolhatatlan!**

# Pointerek az egyes nyelvekben

A programozási nyelvek között a lehetséges különbségek:

- Csak konkrét típusra mutató pointerek megengedettek, vagy vannak típus nélküli pointerek is?
- Csak dinamikusan allokált objektumokra mutathat pointer, vagy „normál” változókra is?
- Lehetnek-e alprogramra mutató pointerek is?
- Milyen fajta konstans pointerek megengedettek?
  - Egy tömbnév C-ben egy konstans pointer a tömb objektum 0. elemére.
- Kötelező a pointer típusoknak önálló nevet adni, vagy csak a mutatott típust kell megadni?
  - `type PInteger is access to Integer;` (ADA-ban)
  - `int * x;` (C-ben)

# Pointerek az egyes nyelvekben

Milyen biztonságosan kezelhető a „csellengő” pointerek problémája?

Mi a megengedett műveletek halmaza?

Kapnak a pointer változók kezdeti (üres) értéket a deklarációnál?

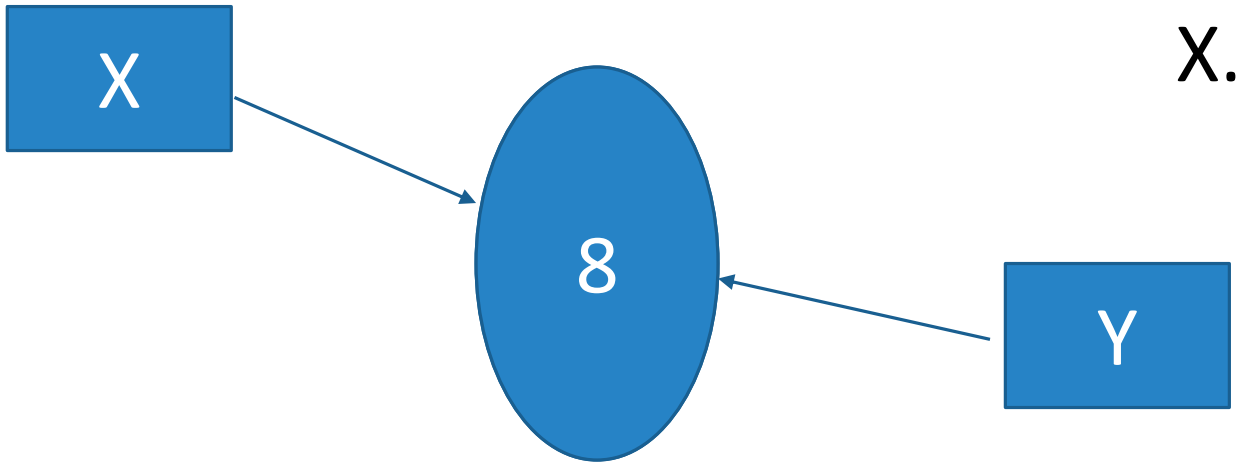
Lehetséges-e ugyanazt az adatot két (vagy több) pointeren keresztül is változtatni/elérni?

# Pointerek

CLU

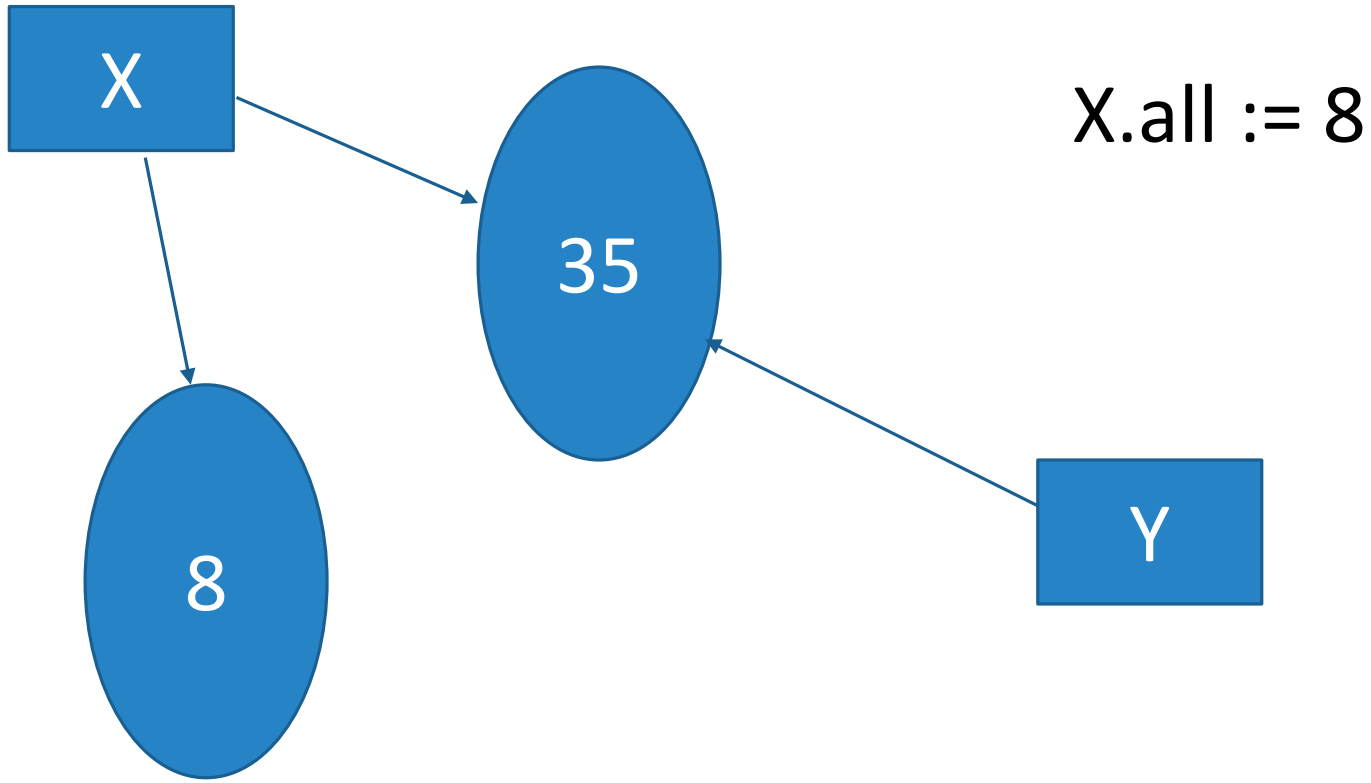
- A CLU-ban nincs hagyományos pointer típus.
- A program végrehajt műveleteket objektumokon
- Az objektumok mint egy univerzum részei léteznek, a program változói hivatkoznak ezekre az objektumokra
  - Garbage collection a felszabadításra
- A programban kétféle objektum lehet:
  - mindig ugyanaz az értéke (immutable
  - változhat az értéke (mutable)

# Mutable – változtatható




X.all := 8

# Immutable – nem megváltoztatható



# Pointerek

C++

- A legtöbb `T` típusra, `T*` a megfelelő pointer típus:
  - `int *p;`
- A tömbökre és függvényekre mutató pointereknek kicsit bonyolultabb jelölése van
  - `int (*vp)[10]; // pointer 10 int tömbjére`
  - `int (*fp) (char, char*);`
    - függvényre mutató pointer, amelynek
    - (char, char*) argumentumai vannak és egy egészet (int) ad vissza

# Pointerek

C++

- A megengedett műveletek:
  - Dereferencing: prefix *
  - Dinamikus allokálás `new`
  - Deallokálás `delete`
  - értékadás: `=`
  - egyenlőség: `==`
  - additív műveletek `+ -`
  - increment, decrement `++ --`
  - member ref. `.* ->*`

# Pointerek

C++

- van egy speciális operátor, az „address_of” „&”, ennek segítségével adhatjuk értékül változók címét pointereknek:
  - `int i = 10;`
  - `int *pi = &i; // a pi pointer az i változóra vonatkozik`
  - `int j = *pi; // j-t 10-re állítjuk`
- Az '&' operátorral létrehozhatjuk objektumok referenciáit is
  - egy referencia úgy tekinthető mint egy konstans pointer, ami mindig automatikusan dereferenciát hajt végre:
    - `int &r = i; // r és i ugyanarra vonatkozik`
    - `r = 2;  // i=2`
- Az increment, decrement, … veszélyesek lehetnek, vigyázzunk, ne keverjük össze a jelentését!
  - `pi++` a pointert inkrementálja, és a következő memóriacímre fog mutatni, ennek akkor van értelme, ha pi egy tömbre mutat, míg `r++` inkrementálja `i` értékét

# Pointerek

Java

- Nincs hagyományos pointer típus
- A változókban kétféle érték tárolható
  - primitív értékek (egy numerikus típusból vagy egy logikai)
  - referencia értékek
- Az objektumokat (osztályok példányai vagy tömbök) referenciákkal kezeli.
- Ugyanarra az objektumra számos referencia hivatkozhat.
- Objektumok referenciáinak műveletei:
  - mező elérés, metódus hívás, casting, string concatenation, instanceof, '==' '!=' (referencia egyenlőségének vizsgálata) stb.

# Pointerek

Eiffel

- Itt sincsenek hagyományos pointer típusok.
- A változókban kétféle érték tárolható - kiterjesztett értékek és referencia értékek.
- Ugyanarra az objektumra számos referencia hivatkozhat.

# Pointerek

C#

- A referencia típusok objektumait kezelhetjük referenciákkal.
- Egy „unsafe” környezetben egy típus lehet pointer is,  erre számos művelet megengedett (pl. a ++, -- is).

# Adattípusok definiálása

# Hogyan definiálhatunk új adattípusokat?

Példák

- Pascal
  - `type <typn>=<value desc>;`
  - `type myint=integer;`
- C++
  - `typedef <value desc> <typn>;`
  - `typedef int myint;`
- ADA
  - `subtype <typn> is <typ1>;` `type <typn> is new <typ1c>;`
  - `subtype Int is Integer;` `type My_Int is new Integer;`
- Java, Eiffel, C#, Swift, Kotlin, stb.
  - `class`
  - `(struct)`
  - `enum`

# Melyek a megengedett típus-konstrukciók?

Iterált
- egy kiinduló típusból

Direkt szorzat
- több kiinduló típusból

Unió
- több kiinduló típusból

# Tömb típusok

„Egy tömb egy olyan adatszerkezet, amely azonos típusú elemek sorozatait tartalmazza.”

Általában egy tömb egy leképezés egy folytonos diszkrét intervallumról elemek egy halmazára, nem igazi sorozat típus.

- Tömbnév(indexértékek) → elem

A diszkrét intervallum elemeit hívjuk index értékeknek.

Az elemek száma ebben az intervallumban definiálja a tömb *méretét*.

# A legfontosabb kérdések

- Milyen adattípusok lehetnek tömb típusok indextípusai?
- Mi lehet tömb típusok elemtípusa?
- Tartalmazzák-e a tömb típusok az indexhatárokat? És a tömb objektumok?
- Mikor dől el a mérete, a helyfoglalása?
- Van-e indextúlcsordulás-ellenőrzés?
- Van-e többdimenziós tömb?
  - Van-e altömb (szelet) képzés?
  - Van-e teljes tömbre vonatkozó értékadás? (kezdő értékadás?)
  - Van-e tömbkonstans?
- Megváltoztatható-e egy tömb mérete?
- Rögzített méretű sorozat vagy nem?

# Indexelés

Az alapművelet az *indexelés*

- `A[i]`
- Elvárás, hogy az `A` tömb `i`. elemét gyorsan el kell tudni érni!

Általában az elemek ugyanahhoz a típushoz tartoznak

- vagy egy adott típus lehetséges leszármazottai is lehetnek
- nem mindig: Smalltalk esetén az elemek különböző típusúak is lehetnek

# Tömbök

ADA

- Tömb típusok definiálhatók rögzített és megszorítás nélküli indexhatárokkal:
  - `type A is array(Integer range 2 .. 10) of Boolean;`
  - `type Vect is array(Integer range <>) of Integer;`
  - `type Matr is array(Integer range <>,Integer range <>) of Integer;`
- A konkrét indexhatárokat az adott objektum deklarációjánál kell meghatározni:
  - `V : Vect(1 .. 30);`
  - `B : Matr(1 .. 2, 1 .. 4);`
- Gyakran használják alprogramok paramétereként, sablonoknál.

# Tömbök

ADA

- Az index típusa tetszőleges diszkrét típus lehet, az elemek típusa tetszőleges típus
- A definíciókat futási időben értékeli ki, az aktuális indexhatárok nem kell, hogy statikusak legyenek.
- Tömbök szeletei is létrehozhatók:
  - `V(2..12)`
    - de: `V(1..1) <-> V(1)!`
- Megengedett az értékadás azonos típusú tömbök között:
  - `V(1..5):= V(2..6);`
- Lehetséges értékadás tömb „aggregátokkal” is:
  - `V:=(1..3=>1, others=>0);`

# Tömbök

## ADA

- Három előre definiált string típus:
  - ```
    type String is array (Positive range <>)
       of Character;
    ```
  - ```
    type Wide_String is array (Positive range <>)
       of Wide_Character;
    ```
  - ```
    type Wide_Wide_String is array(Positive range <>)
       of Wide_Wide_Character;
    ```
- Vannak speciális attribútumai:
  - `A'First/A'First(N)`
  - `A'Last/A'Last(N)`
  - `A'Range/A'Range(N)`
  - `A'Length/A'Length(N)`
    ```

# Tömbök

```
generic
   type Elem is private;
   type Index is (<>);
   type Vekt is array (Index range <>) of Elem;
   procedure Glinker (V: Vekt; E: Elem;
                      Found: out Boolean; Ind: out Index);



procedure Glinker (V: Vekt; E: Elem;
                   Found: out Boolean; Ind: out Index) is
begin
   Ind:=V'First;
   Found :=V(V'First)=E;
   while not Found and Ind<V'Last loop
      Ind:=Index'Succ(Ind);
      Found:=V(Ind)=E;
   end loop;
end;
```

# Tömbök

C++:

- Egy `T` típusra:
  - `T x[size]` a `T` típusú elemek `size` méretű tömbje.
    Az indexek 0 és size-1 között.
    - `float v[3];  // 3 float tömbje`
    - `int a[2][5]; // 5 int két tömbje`
- Kezdeti érték adható:
  - ```
    char v[2][5] = {{'a','b','c','d','e'},
                    {'0','1','2','3','4'}};
    ```
- A pointerek és tömbök szoros kapcsolatban vannak
  - egy tömbnév mindig a tömb nulladik elemére hivatkozik, így használható a pointeraritmetika tömbökre.
- Nem tudja a méretét!
- STL vector template osztály!
    ```

# Tömbök

Java:

- „Java arrays are objects, are dynamically created and may be assigned to variables of type Object.”
- `int [] ai;  // egészek tömbje`
- `short [][] as1,as2; // as1 és as2 short-ok tömbjének tömbjei`
- ha a '[ ]' –t a változó neve után írjuk, akkor csak ez a változó lesz tömb:
  - `long l, al[]; // l egy long típusú változó,`
    `// al long  típusú elemek tömbje`
- Tömb létrehozása:
  - `a = new int[20];`
- A tömb mérete lekérdezhető: `a.length`
- Kezdeti érték adható:
  - `String []colours = {"red","white","green"};`

# Tömbök

## Java

- Egy tömbök tömbjében (ami a Java esetén gyakran több dimenziós tömbnek hívnak a dokumentációkban) az elemeknek lehet különböző mérete:

```
int[][] m = new int[3][];
for (int i=0; i<m.length; i++ ) {
  m[i] = new int[i+1];
  for (int j=0; j<m[i].length; j++)
    m[i][j] = 0;
}
```

- A szövegek kezelését a `String` és `StringBuffer` (`StringBuilder`) osztályokkal oldották meg.
  - Karakterek egy tömbje nem `String`!

# Tömbök

C#

- A tömb elemeinek számozása 0-val kezdődik
- Kétféleképpen lehet deklarálni
  - adott hosszúságú vagy dinamikus.
- A nyelvben a tömbök objektumok, a deklaráció után szükség van a tömb példányosítására (new)
- Inicializálásra: { }
- Adott méretű tömb deklarálása:

```
int[] Tomb;    Tomb = new int[3];
```

- Ugyanez a tömb inicializálva:

```
Tomb = new int[3]  { 1,2,3 }
```

- Dinamikus tömb létrehozása inicializálással:

```
Tomb = new int[] { 1,2,3 }
```

- A deklarációval egybekötött inicializáció:

```
int[] Tomb = new int[3] { 1,2,3}
```

- Ha egy tömböt nem inicializálunk, akkor a tömb elemei automatikusan inicializálódnak az elem típusának alapértelmezett inicializáló értékére.

# Tömbök

C#

- A tömbök lehetőségei:
  - egydimenziós tömbök,
  - többdimenziósak vagy négyszögszerűek,
  - kesztyűszerűek (tömbök tömbjei)
  - kevert típusúak (az előzőekből)
- Példa egy kesztyűszerű dinamikus tömbre:

```
int[][] numArray = new int[][] { new int[] {1,3,5},
                                 new int[] {2,4,6,8,}};
```

- minden tömb típus a `System.Array` bázistípusból „származik”.
  - Az `Array` osztály egy absztrakt bázisosztály, de a `CreateInstance` metódusa létre tud hozni tömböket.
  - Ez biztosítja a műveleteket a tömbök létrehozásához, módosításához, bennük való kereséshez, illetve rendezésükhöz.

# Tömbök

C#

- Az `Array` osztály tulajdonságait megadó függvények:
  - `IsFixedSize` - rögzített hosszúságú-e
  - `IsReadOnly` - írásvédett-e
  - `IsSynchronized` - a tömb elérése kizárólagos-e (szálbiztos)
  - `Length` - a tömb elemeinek száma
  - `Rank` - a tömb dimenzióinak száma
  - `SyncRoot` - Visszatér egy objektummal, amit a tömb szinkronizált hozzáféréséhez használhatunk
- Az `Array` osztályban még számos szolgáltatás:
  - `BinarySearch` - bináris keresés a tömbön
  - `Clear` - minden elemet  töröl a tömbből és az elemszámot 0-ra állítja
  - `Clone` - másolatot készít
  - `Copy` - egy tömb részét átmásolja egy másik tömbbe, végrehajtja az esedékes típuskényszerítést és csomagolást (boxing).

# Tömbök

C#

- Az `Array` osztályban még számos szolgáltatás:
  - `CopyTo` - átmásolja az elemeket egy egy-dimenziós tömbből egy másik egy-dimenziós tömbbe egy megadott indextől kezdve.
  - `CreateInstance` - Létrehoz egy tömb példányt.
  - `GetEnumerator` - Visszatér egy `IEnumerator-ral` a tömbhöz.
  - `GetLength` - az elemek száma
  - `GetLowerBound` - Megadja a tömb alsó korlátját.
  - `GetUpperBound` - Megadja a tömb felső korlátját.
  - `GetValue` - megadott indexű elem értéke.
  - `IndexOf` - egy-dimenziós tömbben az első érték indexe.
  - `Initialize` - Egy értéktípusú tömbben minden elemre meghívja az elemek alapértelmezett konstruktorát.
  - `LastIndexOf` - Visszaadja az egy-dimenziós tömbben az utolsó érték indexét.
  - `Reverse` - Megfordítja a tömb vagy tömbrészlet bejárási irányát.
  - `SetValue` - A megadott elemet beteszi a megadott helyre a tömbben.
  - `Sort` - A tömbön rendezést hajt végre.

# Tömbök

Eiffel

- Az Eiffel tömbök az `ARRAY[G]` sablon osztály példányai.
- A stringeket a `STRING` osztály objektumai valósítják meg.

# Asszociatív tömbök

Egy asszociatív tömb elemek egy rendezetlen halmaza, amelyet megegyező számú, kulcsnak nevezett értékek indexelnek.

Ezeket a kulcsokat is tárolni kell

- az elemek így (kulcs, érték) párok

# Asszociatív tömbök

## Python

- Beépített „dictionary” típus, kulcs-érték párok tárolására
  - A kulcsnak „hashable” típusnak kell lenniük
  - A kulcsokat használunk az elemek elérésére.
- Példa
  - `tel = {'jack': 4098, 'sape': 4139}`
- Ezekre a változókra
  - a `list()` függvény a kulcsok listáját adja vissza,
  - a `del` utasítással lehet kulcs szerint törölni
    - `del tel['sape']`
  - for ciklussal végig lehet járni a tárolt kulcs-érték párokat
    - `for k, v in tel.items():`
  - az `in` kulcsszó megadja, hogy egy adott kulcs szerepel-e benne
    - `'jack' not in tel`

# Asszociatív tömbök

A Java, a C++, az Eiffel szabványos osztálykönyvtárában is megtalálhatók

A .NET keretrendszer osztálykönyvtárában is

Egyéb nyelvek:

- PHP,
- Perl,
- Ruby,
- Lua,
- Pike,
- stb.

# Rekord típusok

A direkt szorzat és az unió típuskonstrukciókat számos nyelvben az úgynevezett rekord típusok segítségével valósítják meg.

Ha adott két típus, S és T, direkt szorzatukat S × T jelöli.

- $S \times T = \{(x,y) | x \in S; y \in T\}$
- Ez általánosítható több halmazra is:
  $S_1 \times S_2 \times \ldots \times S_n$

A direkt szorzat terminusaival érthetjük meg:

- COBOL, Pascal, Ada stb. rekordok
- Algol68, C, C++, C# stb. struktúrák

# Rekord típusok

```
record
  <name1> : <type1>;
  <name2> : <type2>;
  …
  <namek> : <typek>;
end
```

A komponenseket a rekord (struktúra) mezőinek hívják.

Az alapművelet a komponens kiválasztás.

# Rekord típusok

Lehet-e paramétere a típusnak?

Van-e kezdő értékadás a mezőkre?

Van-e teljes rekordra vonatkozó értékadás?

Van-e rekord konstans?

Hogyan működik a kiválasztás művelet?

# Rekord típusok

Pascal

```
type rektipnev = record
    mnev1 :  tipus1;
    ...
     mnevn :  tipusn;
end;
```

- változó deklarálása
  - `rek: rektipnev;`
- hivatkozás ponttal
  - `rek.mnev1`
- csak mezőnkénti értékadás lehetséges

# Rekord típusok

## Pascal

- Speciális utasítás, aminek a segítségével a rekord mezőire közvetlenül tudunk hivatkozni:

```
type Date= record
    Year : Integer;
    Month : 1..12;
    Day : 1..31;
end;
var R1, R2 : Date;
begin
  R1 := R2; {értékadás megengedett}
  with R1 do begin
    Year := 2011; Month:=9; Day:=29;
    Year := R2.Year;
end;
```

# Rekord típusok

C++

```
struct strnev {
        tipus1 mnev1;
        ...
        típusn mnevn;
};
```

- változó deklarálása
  - `strnev x;`
- hivatkozás ponttal
  - `x.mnev1`

# Rekord típusok

C++

- A tömbökre használt jelölés alkalmazható struktúrákra is. pl.:

```
struct address{
  long number;
  char* street;
  char* town;
  int zip;
}
address a = {1, "Korkeakoulunkatu", "Tampere", 33720}
```

- Az inicializálásra a konstruktorok jobban  használhatóak.
- Értékadás megengedett, de az egyenlőségvizsgálat nem előre definiált.
  - A felhasználó definiálhat operátorokat rá

# Rekord típusok

ADA

```
type Complex is record
          Re: Float;
          Im: Float;
end record;
```

- változó deklarálása
  - `C: Complex;`
- a komponenseire ponttal hivatkozhatunk
  - `C.Re`
    `C.Im`
- megengedett az értékadás is
  - `C1, C2: Complex;`
    `C1 := C2;`

# Rekord típusok

ADA

- A rekord diszkriminánsa(i)
  - a típus paramétere
  - több diszkriminánsa is lehet egy típusnak
  - a rekord diszkrimináns diszkrét típusú

```
type Szöveg(Hossz: Natural) is record
           Érték: String(1 .. Hossz) := (others=>’ ’);
           Pozíció: Natural := 0;
end record;
```

# Rekord típusok

C#:

- a rekord (struct) érték típus
- az osztály (class) referencia típus

Bizonyos programozási nyelvekben nincs rekord típus, a tervezők az osztályok használatát javasolják helyette

- SmallTalk
- Eiffel
- Java
- …

# Uniók és variáns rekordok

Az unió típusértékhalmaza az unió komponensei típusértékhalmazának az uniója.

- Például bútor:
  - szék vagy asztal vagy szekrény
  - színe, anyaga – van mindegyiknek
  - egyéb speciális jellemzők – külön-külön

A variáns rekordok olyan direktszorzatok, ahol a direktszorzat egy – az utolsó – komponense unió.

# „Választó” típusműveletek

A programozási nyelvek a megbízhatóság különböző szintjén támogatják ezt az adatszerkezetet.

Az unió típusnak

- van egy speciális, tag-nek nevezett komponense, és
- egy kiválasztási mechanizmusa, ami megadja a tag különböző értékeinek megfelelő alstruktúrákat

Ha a tag-et tárolja a rekord, és az alkomponensek elérhetősége ennek aktuális értékétől függ,

- akkor ez egy „megkülönböztetett” (discriminated) unió,
- különben ez egy „szabad” (free) unió.

# Unió (Variáns rekord)

Meg lehet-e állapítani, hogy a rekord melyik változat szerint lett kitöltve?

Ki lehet-e olvasni a kitöltéstől eltérő változat szerint?

Szerkezete (gyakran):

```
case <tag-name> : <tag-type> of
   <const1> : ( <fields1>);
   <const2> : ( <fields2>);
   …
   <constv> : ( <fieldsv>);
```

# Unió (Variáns rekord)

A szabad uniók esetében:

- A tag mezők használata opcionális,
- A fordító nem ellenőrzi a kiválasztott mező és a tárolt érték konzisztenciáját.
- Ez a nyelv típusrendszerét megbízhatatlanná teszi.

A megkülönböztetett unió esetében fontos kérdés, hogyan lehet új értéket adni a tag mezőnek.

- A tag értéket beállítja a rekord létrehozásakor.
- Míg a program képes kell legyen új értéket adni a rekord „normális” komponenseinek, a tag megváltoztatása a rekord szerkezet megváltozását vonja maga után!

# Példa

Pascal:

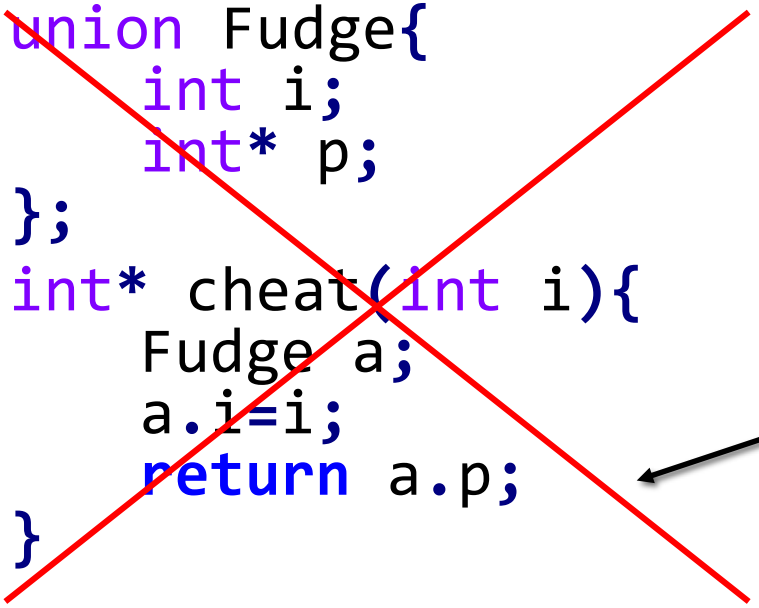
```
type Listptr=^Listnode;
type Listnode=record
       Next: Listptr;
       case Tag: Boolean of
           False: (Data:char);
                  True: (Down: Listptr)
    end;
var p, q: Listptr;
p^.Tag:=true;
p^.Down:=q;
p^.Tag:=false;
writeln(p^.Data);
```

**HIBA**

# Példa

C++:

```
union typname{
    typ1 field1;
    ...
    typm fieldm;
};
union Fudge{
    int i;
    int* p;
};
int* cheat(int i){
    Fudge a;
    a.i=i;
    return a.p;
}
```



**Jó ez??**

# Unió (Variáns rekord)

ADA

```
type Állapot is (Egyedülálló, Házas, Özvegy, Elvált);
subtype Név is String(1..25);
type Nem is (Nö, Férfi);
type Ember (Családi_Áll: Állapot := Egyedülálló) is
  record
     Neve: Név;
     Neme: Nem;
     Születési_Ideje: Dátum;
     Gyermekek_Száma: Natural;
     case Családi_Áll is
       when Házas  => Házastárs_Neve: Név;
       when Özvegy => Házastárs_Halála: Dátum;
       when Elvált => Válás_Dátuma: Dátum; Gyerekek_Gondozója: Boolean;
       when Egyedülálló => null ;
     end case;
  end record;
```

# Unió (Variáns rekord)

ADA

- `Hugó: Ember(Házas);`
  - `Családi_Áll, Neve, Neme, Születési_Ideje, Gyermekek_Száma, Házastárs_Neve`
- `Eleonóra: Ember(Egyedülálló);`
  - `Családi_Áll, Neve, Neme, Születési_Ideje, Gyermekek_Száma`
- `Ödön: Ember(Özvegy);`
  - `Családi_Áll, Neve, Neme, Születési_Ideje, Gyermekek_Száma, Házastárs_Halála`
- `Vendel: Ember(Elvált);`
  - `Családi_Áll, Neve, Neme, Születési_Ideje, Gyermekek_Száma, Válás_Dátuma, Gyerekek_Gondozója`
- `Aladár: Ember;  -- Egyedülálló`
  `Családi_Áll, Neve, Neme, Születési_Ideje, Gyermekek_Száma`
- Helytelen (futási idejű hiba, altípus-megszorítás megsértése):
  - `Hugó.Válás_Dátuma, Aladár.Házastárs_Neve`

# Unió (Variáns rekord)

ADA

- Megszorítatlan altípus használata
  - `Aladár : Ember;      -- alapért. Egyedülálló`
  - `Aladár := (Házas, ....);`
  - `Aladár := Elek;`
  - `Aladár := Hugó;`
- A szerkezetét megváltoztathatjuk, diszkriminánsostul
- Csak úgy, ha az egész rekord értéket kap egy értékadásban

# Unió (Variáns rekord)

Bizonyos programozási nyelvekben nincs unió típus, a tervezők az osztályok és az öröklődés használatát javasolják helyette:

- SmallTalk,
- Eiffel,
- Java,
- C#

# Halmaz

Speciális iterált típus

- Mi lehet az eleme?
- Hány eleme lehet?
- Megvannak-e a „hagyományos” halmazműveletek?

# Halmaz

Pascal

- Alaphalmaz: diszkrét típus
- Elemek száma: max.  256
- Értékek sorszáma csak 0..255 között
  - `type Small_Letters = set of 'a'..'z';`
  - `type Digits = set of '0'..'9';`
  - ```
    type Day = (Monday, Tuesday, Wednesday, Thursday,
                Friday, Saturday, Sunday);
    ```
  - ```
    type Days = set of Day;
    var A, B : Days;
    ```
  - `A:=[Monday, Wednesday] - halmazkonstruktor`
- üres halmaz: [ ]

# Halmaz

## Pascal

- Műveletek:
  - értékadás,
  - halmazműveletek:
    - eleme-teszt (in),
    - az =  , <>,  részhalmaz reláció ('<=', '>=')
    - (a  '<' és '>' nem megengedett!)
    - unió ('+'), differencia ('-').
    - metszet ('*')
- Ez a precedencia-sorrend is.

# Mikor ekvivalens két típus?

```
x, y: array[0..9] of integer;
z: array[0..9] of integer;
```

Strukturális ekvivalencia esetén:

- A rekordok mezőnevei is figyelembe vannak véve, vagy csak a struktúrájuk?
- Számít-e a rekordmezők sorrendje?
- Tömböknél elég-e az indexek számosságának egyenlőnek lenni, vagy az indexhatároknak is egyezniük kell?

# Mikor ekvivalens két típus?

Értékül adhatóak egymásnak? Akarjuk?

```
Dimensions = RECORD
          Breadth: REAL;
          Length: REAL;
    END;


Complex = RECORD
          RealPart: REAL;
          ImPart: REAL;
    END;
Size: Dimensions;
Root: Complex;
```

# Név szerinti ekvivalencia esetén

Deklarálhatók-e egy típushoz típusok, amelyekkel ekvivalens?

Névtelen tömb- illetve rekordtípusok ekvivalensek-e valamivel?

# Típuskonverziók

Van-e, és hogyan működik?

- az automatikus konverzió?
- az identitáskonverzió?
- a bővítő konverzió?
- a szűkítő konverzió?
- a toString konverzió?

# Típuskonverziók

Java

- identitáskonverzió
  - a `boolean` csak ez szabad
- bővítő konverzió
  - `byte` to `short`, `int`, `long`, `float` or `double`
  - `short` to `int`, `long`, `float` or `double`
  - `int` to `long`, `float` or `double`
  - `long` to `float` or `double`
  - `float` to `double`

# Típuskonverziók

## Java

- szűkítő konverzió
  - `byte` to `char`
  - `short` to `byte` or `char`
  - `char` to `byte` or `short` (miért is?)
  - `int` to `byte`, `short` or `char`
  - `long` to `byte`, `short`, `char` or `int`
  - `float` to `byte`, `short`, `char`, `int` or `long`
  - `double` to `byte`, `short`, `char`, `int`, `long` or `float`

# Változók

Változó =  (név, attribútumhalmaz, hely, érték)

Láthatóság, Elérhetőség

Hogyan definiálhatunk változókat és konstansokat?

# Változók

| | Szintaxis | Példa |
|---|---|---|
| Pascal | `var <identif>: <type>;` | `var i : integer;` |
| C++ | `<type> <identif>[= <value>];` | `int i = 0;` |
| Java | `<type> <identif>[= <value>];` | `int i = 0;` |
| ADA | `<identif>: <type>[:= <value>];` | `I : Integer := 0;` |
| CLU | `<identif>: <type>[:= <value>];` | `i : int := 0;` |
| Eiffel | `<identif>: [expanded]<type>;` | `I : INTEGER;` |
| Swift | `var <identif>: [<type>][ = <value>]` | `var x: Integer = 10` |

# Konstansok

| | Szintaxis | Példa |
|---|---|---|
| Pascal | `const <name> = <value>;` | `const val = 3;` |
| C++ | `#define <name> <value>`<br>`const  <type> <name> = <value>;` | `#define val 3`<br>`const int val = 3;` |
| Java | `final <type> <name> = <value>;` | `final int val = 3;` |
| ADA | `<name> :`<br>`constant <type>:= <value>;` | `Val :`<br>`constant Integer := 3;` |
| CLU | `const <type> <name> = <value>;` | `const double PI = 3.14159;` |
| Eiffel | `<name> : <type> is <value>;` | `A: INTEGER is 3;` |
| Swift | `let <identif>: [<type>] = <value>` | `let x: Integer = 10` |

# Kifejezések

Prefix jelölés

- + a b

Postfix jelölés

- a b +

Infix jelölés

- a + b

A műveletek precedencia szintjei fontosak

# Precendica - Pascal

| Leírás | Operátor |
|---|---|
| zárójelezés | |
| függvényhívás | |
| unáris operátor | not @ |
| Multiplikációs operátor | * / div mod and shl shr as << >> |
| Additív operátor | + - or xor |
| Relációs jel | = <> < > <= >= in is |
| Balról jobbra | |

# Precedencia – C++ 1.

| Precedence | Operator | Description |
|---|---|---|
| 1 | :: | Scope resolution |
| 2 | ++ | Suffix increment |
| | -- | Suffix decrement |
| | () | Function call |
| | [] | Array subscripting |
| | . | Element selection by reference |
| | -> | Element selection through pointer |
| | typeid() | |
| | const_cast | |
| | dynamic_cast | |
| | reinterpret_cast | |
| | static_cast | |

# Precedencia – C++ 2.

| Precedence | Operator | Description |
|---|---|---|
| 3 | ++ | Prefix increment |
| | -- | Prefix decrement |
| | + | Unary plus |
| | - | Unary minus |
| | ! | Logical NOT |
| | ~ | Bitwise NOT |
| | (type) | Type cast |
| | * | Indirection (dereference) |
| | & | Address-of |
| | sizeof | Size-of |
| | new, new[] | Dynamic memory allocation |
| | delete, delete[] | Dynamic memory deallocation |
| 4 | .* | Pointer to member |
| | ->* | Pointer to member |
| 5 | * | Multiplication |
| | / | Division |
| | % | Modulo (remainder) |
| 6 | + | Addition |

# Precedencia – C++ 3.

| Precedence | Operator | Description |
|---|---|---|
| 6 | + | Addition |
| | - | Subtraction |
| 7 | << | Bitwise left shift |
| | >> | Bitwise right shift |
| 8 | < | Less than |
| | <= | Less than or equal to |
| | > | Greater than |
| | >= | Greater than or equal to |
| 9 | == | Equal to |
| | != | Not equal to |
| 10 | & | Bitwise AND |
| 11 | ^ | Bitwise XOR (exclusive or) |
| 12 | \| | Bitwise OR (inclusive or) |
| 13 | && | Logical AND |
| 14 | \|\| | Logical OR |
| 15 | ?: | Ternary conditional (see ?:) |

# Precedencia – C++ 4.

| Precedence | Operator | Description |
|---|---|---|
| 6 | + | Addition |
| 16 | = | Direct assignment |
| | += | Assignment by sum |
| | -= | Assignment by difference |
| | *= | Assignment by product |
| | /= | Assignment by quotient |
| | %= | Assignment by remainder |
| | <<= | Assignment by bitwise left shift |
| | >>= | Assignment by bitwise right shift |
| | &= | Assignment by bitwise AND |
| | ^= | Assignment by bitwise XOR |
| | \|= | Assignment by bitwise OR |
| 17 | throw | Throw operator |
| 18 | , | Comma |

# Precedencia – Java

| Operators | Precedence |
|---|---|
| postfix | expr++ expr-- |
| unary | ++expr --expr +expr -expr ~ ! |
| multiplicative | * / % |
| additive | + - |
| shift | << >> >>> |
| relational | < > <= >= instanceof |
| equality | == != |
| bitwise AND | & |
| bitwise exclusive OR | ^ |
| bitwise inclusive OR | \| |
| logical AND | && |
| logical OR | \|\| |
| ternary | ? : |
| assignment | = += -= *= /= %= &= ^= \|= <<= >>= >>>= |

# Érdekesség

Algol 68

- A kétoperandusú operátorok prioritása általában felüldefiniálható a `prio` kulcsszóval:
  - `prio + = 3, * = 2;`
  - `print(6+3*5);  # (6+3)*5=45 lesz az eredmény`

# Vezérlési szerkezetek

# Egyszerű utasítások:

Értékadás – mit jelent?

- „változó” értékadásjel kifejezés

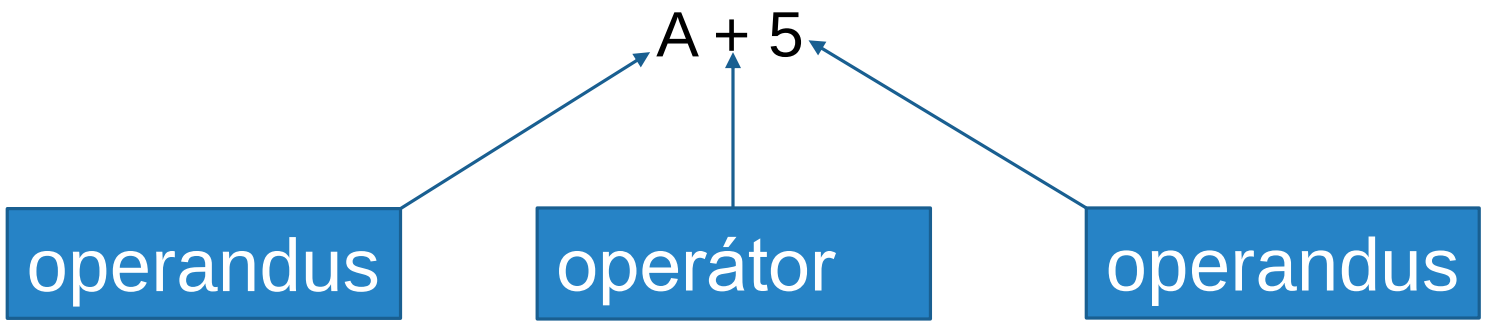
Helyette inkább:

- „balérték” értékadásjel „jobbérték”
  - (vagy fordítva?)
- Mit is jelent ez?
  - Értékadásjel
    - Szokásosan `:=` és `=`
    - De `->` és `<-`
  - Szemantikája
    - „megfelelő” típusú kifejezés kell!

# Kifejezés

operandusokból és operátorokból áll

- minden operandus lehet egy újabb kifejezés
- kiértékelem és megkapom az értékét:



# Kifejezés az értékadás?

Wulf (BLISS)

- „Of course, everything is”

Richie (C)

- „Yes, why not”

Wirth (Pascal)

- „No, only math-like things are expressions”

Két vonulat:

- Pascal, CLU, ADA, Eiffel, …
- C, C++, Java, …

Van-e többszörös értékadás?

# Szemlélet – COBOL

Eredetileg

- `MOVE 23 TO A.`
- `MOVE B TO C.`
- `ADD A TO B GIVING C.`
- `SUBTRACT A FROM B GIVING C.`
- `MULTIPLY A BY B GIVING C.`
- `DIVIDE A BY B GIVING C.`

Újabban

- `COMPUTE A = ( A + B - C / ( A * B ) - A * B )`

# Pascal

„balérték” := kifejezés;

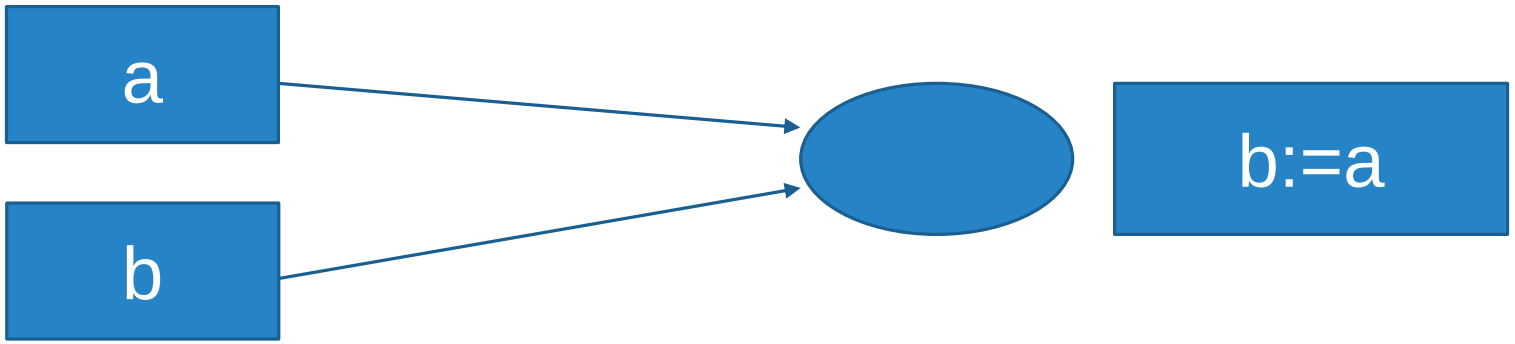
- A „balérték” és a kifejezés típusa megegyező, de legalább kompatibilis kell legyen.
  - `m: integer;`
  - `m:= m+1;`
- vagy:
  - `p := keres(gyoker, x);`

Többszörös értékadás nem megengedett.

# CLU

referenciákat használ

- értékadás hatására két referencia hivatkozhat ugyanarra az objektumra




- Többszörös értékadás megengedett:
  - `x, y := y, x.`

# C++

Számos értékadó operátor jobbról balra feldolgozva

- `A=B=C`
  - jelentése: `A=(B=C);`

Értékadó operátorok

- `= *= /= %= += -= >>= <<= &= ^=`

Például

- `y += g(x);`

# Java

A C++ -hoz hasonló, számos értékadó operátor:

- `= *= /= %= += -= >>= <<= >>>= &= ^=`
- `E1 op= E2`
  - jelentése: `E1 = (T)((E1) op (E2))`,
  - ahol `T` az `E1` típusa.

Összetett értékadó operátoroknál mindkét operandus primitív típusú kell legyen

- kivéve: `+=`, ha a bal operandus `String` típusú),

Implicit cast előfordulhat!

- `short x=3; x+=4.6;` eredménye: `x ==7`!

`final`-nak deklarált változónak nem adható érték.

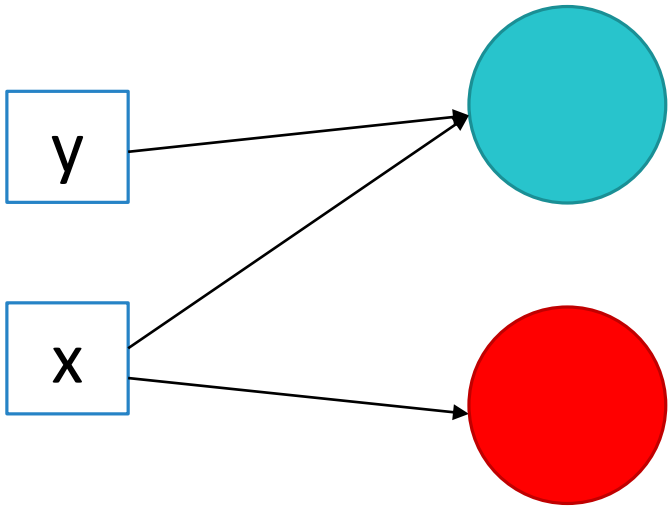
Alapvetően referenciákat használ

# Eiffel

Az értékadás és a paraméterátadás szemantikája megegyezik.

- Ha x:TX, y:TY, akkor az x:=y eredménye TX és TY-tól függ
- Mindkettő lehet referencia vagy kiterjesztett típus.
  - Az Eiffel mechanizmusa a klónozás, referencia másolás szemantikáját segít megérteni

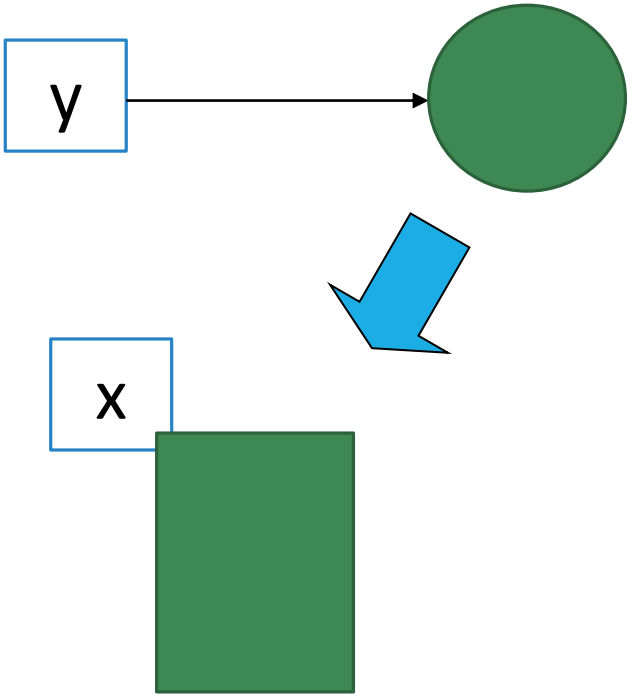
# Mindkettő referencia




*x:=y*

*referencia újrahozzárendelés*

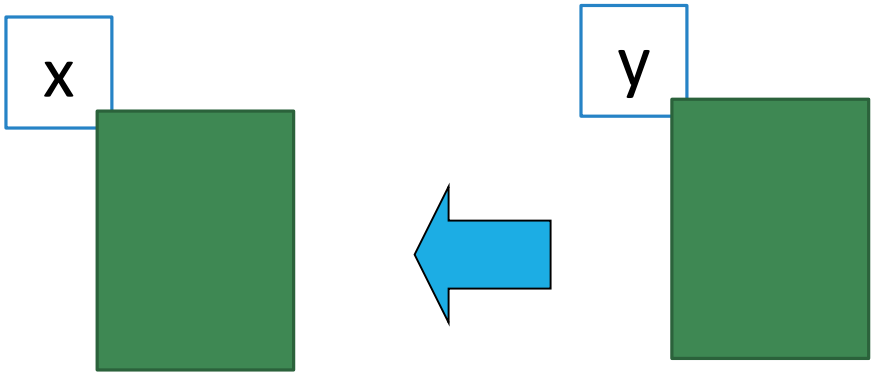
# TX kiterjesztett, TY referencia




*x:=y*

*másolás*

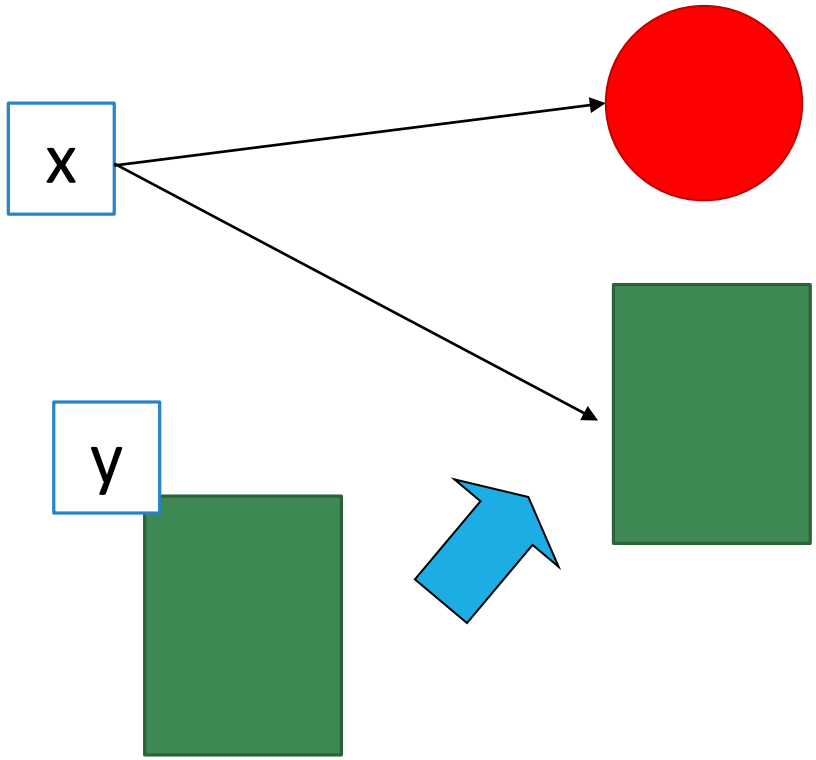
# Mindkettő kiterjesztett




*x:=y*

*másolás*

# TX referencia, TY kiterjesztett




*x:=y*

***klónozás***

# Üres utasítás

Pascal
- megengedett (`case`)

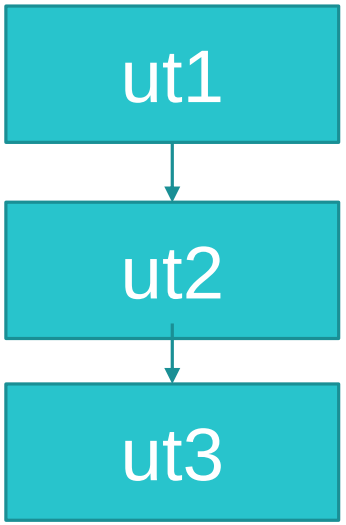
ADA
- null; (`case`)

C++, Java, stb.: „ ; ” használható
- Például ciklusnak lehet üres törzse

Eiffel: „ ; ” használható
- nincs valódi szerepe

# Összetett utasítások

Szekvencia:







ut1; ut2; ut3;

# Szekvencia

Lehet-e üres?

Terminátor vagy utasításelválasztó-e a „ ; ”?

Lehet-e blokkutasítást létrehozni?

Elhelyezhető-e a blokkutasításban deklaráció?

# Szekvencia

Pascal, Delphi:  a „ ; ” elválasztja az utasításokat:

- `begin ut1; ut2; ut3 end`

- üres utasítás is lehet:
  - `begin ut1; ut2; ut3; end`
- Az üres utasítás lehetősége miatt a „ ; ” beírása megváltoztathatja a program jelentését!

ADA: a „ ; ” lezárja az utasítást

C++, Java, Objective-C: a „ ; ” lezárja az utasítást

- de nem mindet, például a blokk utasítást nem

- Illetve , elválaszt kifejezéseket C++ esetén

# Szekvencia

Python: az utasításokat az új sor jelek zárják le

- De a ; megengedett elválasztóként soron belül
  - `x = 5; print(x)`

Swift: a ; elválasztja az utasításokat

- A fordítóprogram elhelyezi

# Blokk utasítások

Utasítások sorozatából alkothatunk egy összetett utasítást, blokkot.

Bizonyos nyelvekben lehet deklarációs része is.

Főleg azokban a nyelvekben fontos, ahol a feltételes és a ciklus utasítások csak egy utasítást tartalmazhatnak

- Pascal, C++,  Java, …

Pascal: `begin utasítássorozat end`

ADA:

- `[declare deklarációk] begin utasítássorozat [kivételkezelő rész] end;`
- Például

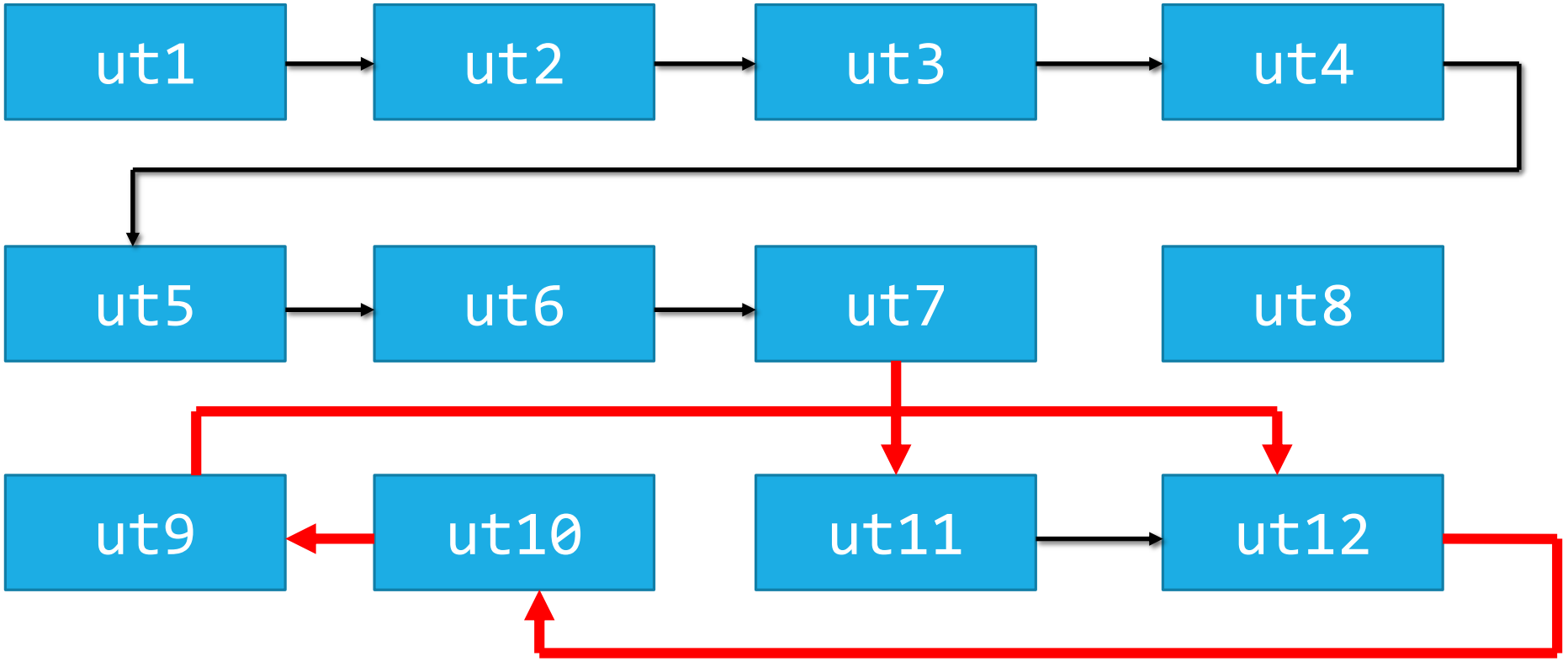
```
Csere:
  declare
  Temp : Integer;
    begin
      Temp := I; I := J; J := Temp;
    end Csere;
```

C++, Java, …

- “{“ és “}” között.

# Feltétel nélküli vezérlésátadás

goto



# Feltétel nélküli vezérlésátadás

javaslat: `goto` nélkül



# Feltétel nélküli vezérlésátadás

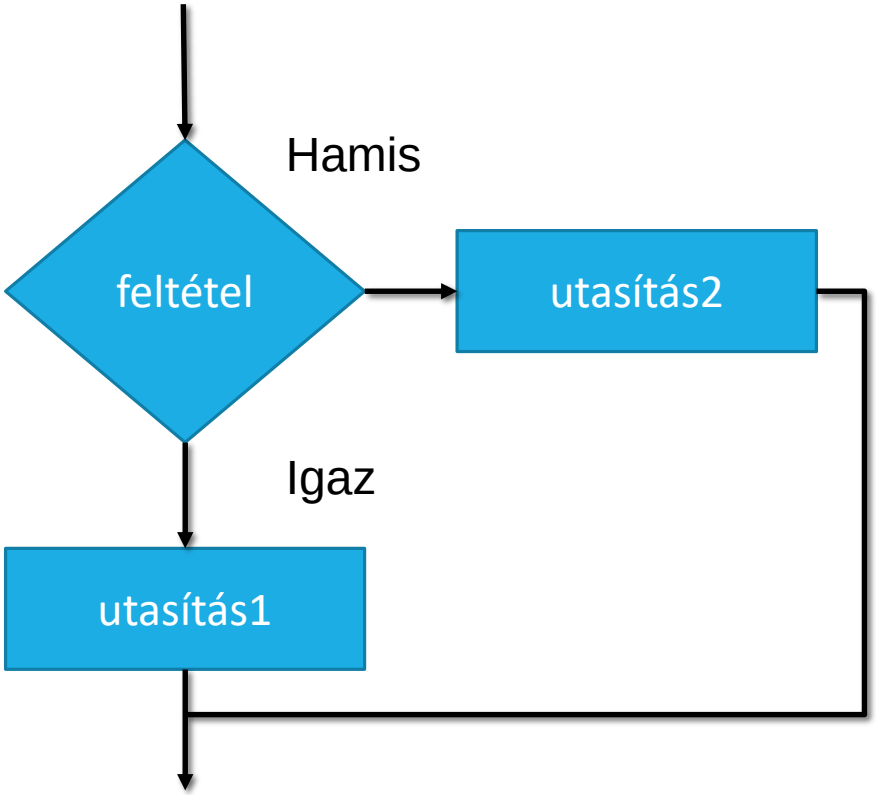
Ha mégis – Pascal, C stb.

```
címke:
   .... utasítások ...
   goto címke;
```

Nincs: Modula-3, Java, …

# Elágazások:

- Feltétel igaz vagy hamis értékétől függően valamilyen utasítás(csoport) végrehajtása






# Elágazások

```
if feltétel then ut1
```

```
if feltétel then ut1 else ut2
```

„csellengő” else kérdése

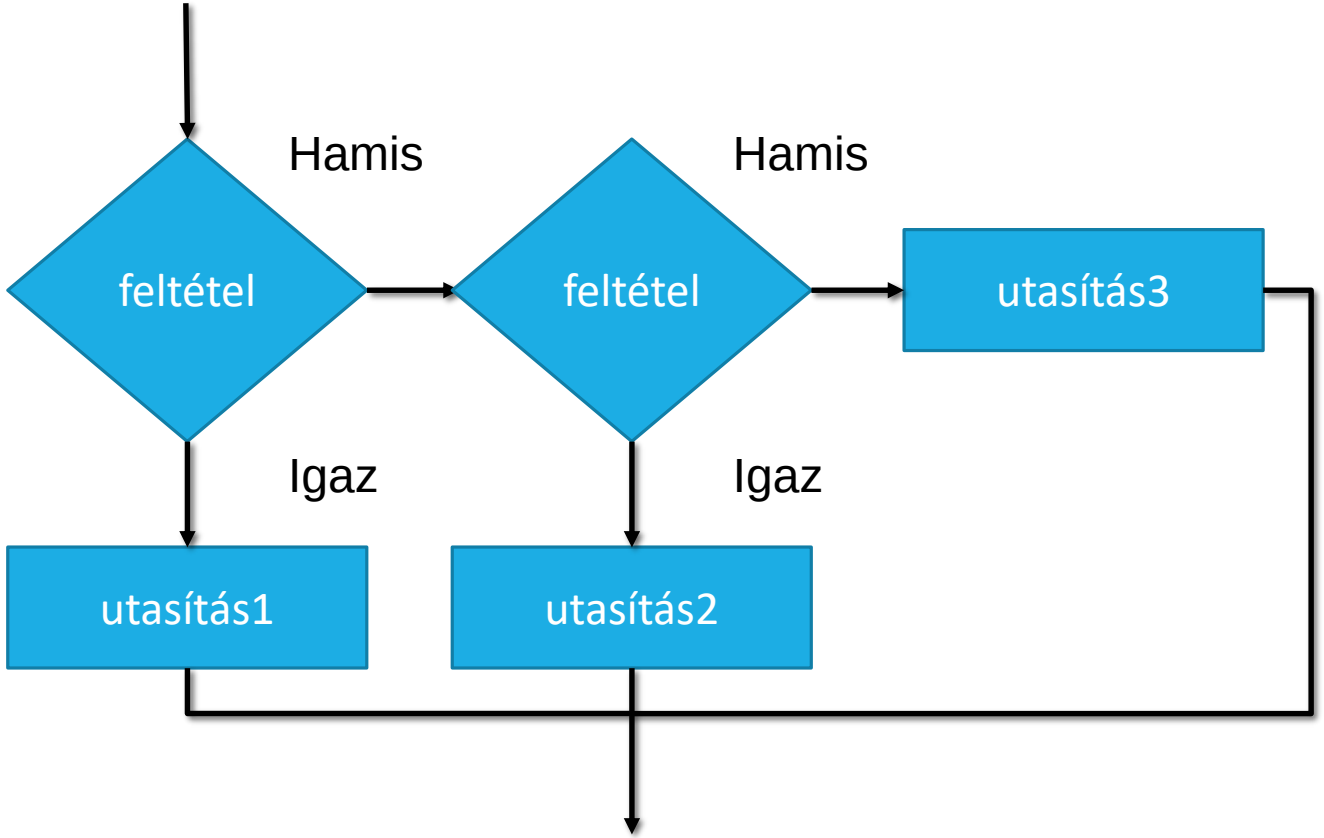
- `if felt1 then if felt2 then ut1 else ut2`
- Mit jelent?
  - `if felt1 then (if felt2 then ut1 else ut2)`
- Vagy?
  - `if felt1 then (if felt2 then ut1) else ut2`

# Elágazások – csellengő else

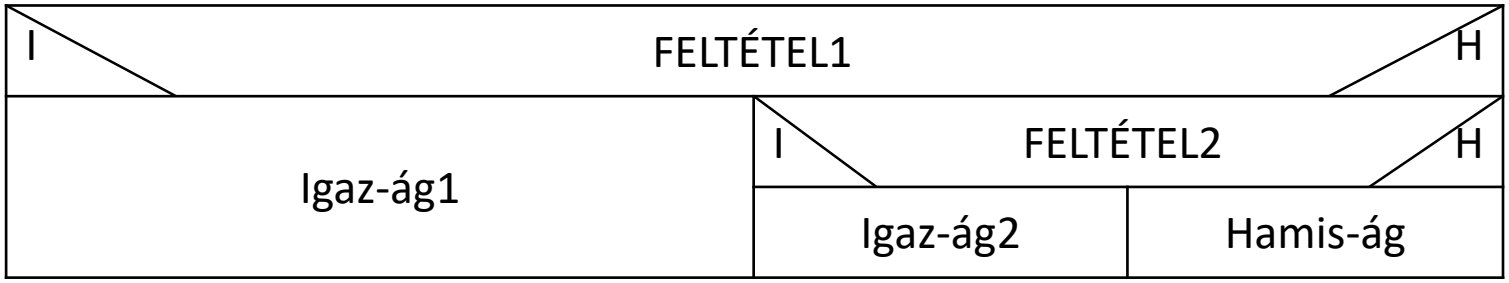
Különböző megoldások:

- blokk utasítás kell a beágyazásnál
- az `else` rész a legbelső `if`-hez tartozik
- explicit `end` konstrukció bevezetése

# Többirányú elágazások



# Többirányú elágazások



# Többirányú elágazások

`elsif` lehetősége – óvatos módosítás kell!

```
if felt1 then s1;
elsif felt2 then s2;
elsif felt3 then s3;
else s4;
end if;
```

# Pascal

```
if felt then S1 [else S2]
```

- több utasítás: blokk kell

„csellengő” else:

- legbelső if :

```
if felt1 then
  if felt2 then S1
           else S2
```

különben blokk:

```
if felt1 then
  begin
    if felt2 then S1
  end
else S2
```

# Pascal

```
if (a>b)
  then writeln("a>b")
else
  if (a<b) then writeln("a<b")
           else writeln("a=b");
```

# Pascal

Vigyázat!

- `if felt1 then ut1 else ut2`
  nem ugyanaz, mint:
- `if felt1 then ut1; else ut2`
  ez ekvivalens:
- `if felt1 then ut1; else ut2`
- ha
  - `if felt1 then ut1 else ut2` -ben
    S1 után beszúrunk egy „ ; ”-t, az hiba!

# ADA

end if a végén, elsif megengedett:

```
if <expr>1 then
   <statm>1;
{elsif (<expr>2)then
   <statm>2;}
[else
   <statm>3; ]
end if;
```

```
if A>B then
   Put_Line("a>b");
elsif A<B then
   Put_Line("a<b");
else
   Put_Line("a=b");
end if;
```

# C++

aritmetikai kifejezés kell, nem nulla: true.

„csellengő” else: legbelső if

- blokk kell különben

```
if (<expr>)
     <statm1>
[else
     <statm2>]
```

```
if (a>b) cout << "a>b";
else
  if (a<b) cout << "a<b";
  else cout << "a=b";
```

aritmetikai if kifejezés pl.:

```
if (a<=b) max=b;
else max=a;
```

inkább:

```
max=(a<=b)?b:a;
```

# Java

Hasonló a C++-hoz

- kivévea kifejezés típusa boolean kell legyen

```
if (<expr>)
   <statm1>
[else
   <statm2>]
```

```
if (a>b)
  System.out.println("a>b");
else
  if (a<b)
    System.out.println("a<b");
  else
    System.out.println("a=b");
```

# Eiffel

end zárja le az utasítást.

Megengedett az elseif használata.

```
if <expr>1 then
   <statm>1
{elseif <expr>2 then
   <statm>2 }
[else
   <statm>3 ]
end
```

```
if (a>b) then
   io.putstring("a>b");
elseif (a < b) then
   io.putstring("a < b")
else
   io.putstring("a = b");
end
```

# Python

Csellengő else: a bekezdés számít!!

```
a, b = 1, 2
if a == 1:
    if b == 1:
        print ("a és b is 1")
else:
    print ("a nem 1")
```

# Python

Csellengő else: a bekezdés számít!!

Így mást jelent:

```
a, b = 1, 2
if a == 1:
    if b == 1:
        print ("a és b is 1")
    else:
        print ("a nem 1")
```

# if és értékadás

Ruby:

- `num += -1 if num < 0`

Scala:

- `val x = if (a > b) a else b`

Jobban olvasható, mint a `? :`

# if és az értékadás

Kotlin

- További utasítások kerülhetnek bele

```
val max = if (a > b) {
    print("Choose a")
    a
} else {
    print("Choose b")
    b
}
```

# Esetkiválasztásos elágazások

Valamilyen kifejezés (szelektor) értékétől függően, általánosan:

| kifejezés | | | | | |
|---|---|---|---|---|---|
| érték1 | érték2 | érték3 | értékek | intervallum | egyéb |
| utasítások | utasítások | utasítások | utasítások | utasítások | utasítások |

# Esetkiválasztásos elágazás

Kérdések

- Mi lehet a szelektor típusa?
- Fel kell-e sorolni a szelektortípus minden lehetséges értékét?
- Mi történik, ha fel nem sorolt értéket vesz fel a szelektor?
- „Rácsorog”-e a vezérlés a következő kiválasztási ágakra is?
- Diszjunktnak kell-e lennie a kiválasztási értékeknek?
- Mi állhat a kiválasztási feltételben?
  - egy érték
  - értékek felsorolása
  - intervallum

# „Case” utasítások

```
case expr of
  const1: st1;
  const2: st2;
 …
  constn: stn
end
```

# „Case” utasítások

Sokszor igaz a case konstansokra:

- Tetszőleges sorrend,
- Nem feltétlenül egymás után,
- Több is vonatkozhat ugyanarra az alutasításra,
- Mind különböző kell legyen - különben, ha $const_i$ és $const_j$ egyenlőek, akkor melyiket választanánk, $st_i$ -t vagy $st_j$-t?
- A kifejezés típusa diszkrét kell legyen (egész vagy felsorolási típ)
- Kell adni egy „others” ágat, hogy minden lehetőséget lefedjünk.

# Pascal

A szelektor típusa lehet: integer, character, boolean vagy tetszőleges felsorolási vagy intervallum típus.

„else” megengedett, de nem kötelező (skip).

```
case var of
  val1: statm1;
  val2.: statm2;
...
  vali…valj : statmi;
[else statm]
end;
```

```
var Age: Byte;
case Age of
  0..13: Write('Child');
 14..23: Write('Young');
 24..65: Write('Adult');
 else    Write('Old');
end
```

# ADA

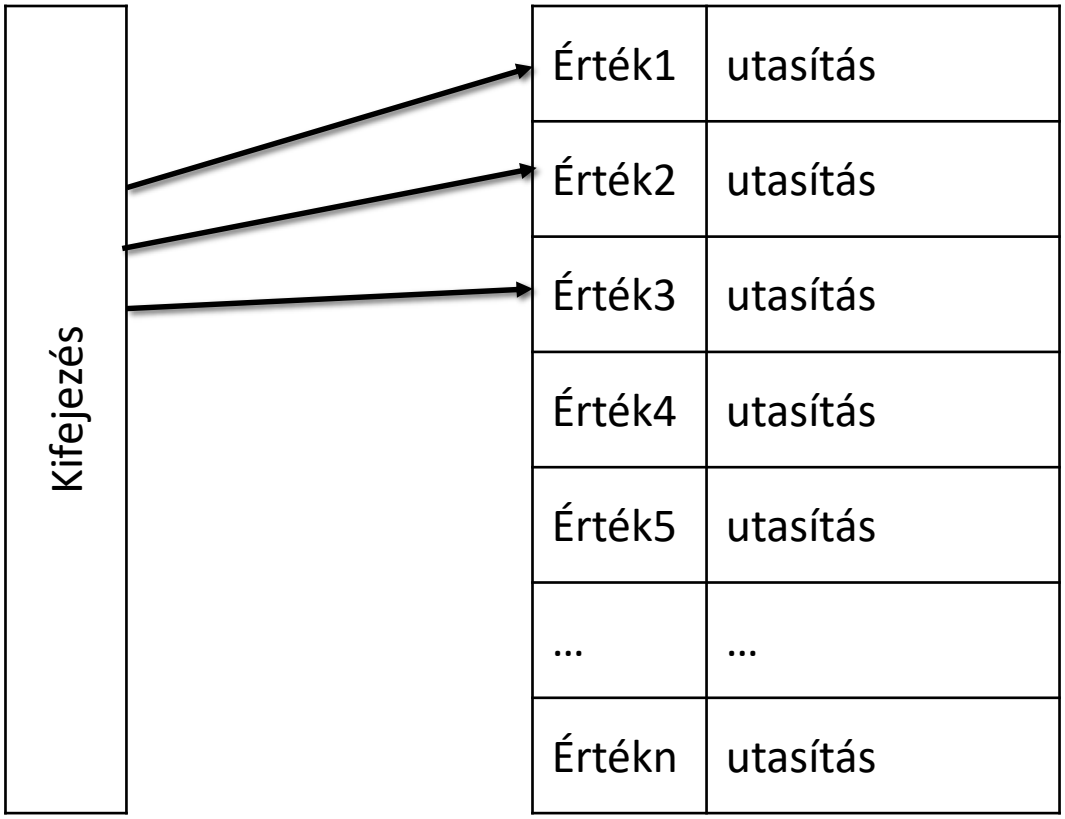
```
case expr is
 when choice1 => statms1;
 when choice2 => statms2; ...
 when others =>  statms;
end case;
```

others kötelező, ha nincs minden lefedve!

.. - intervallum,
| - vagy

```
case Today is
  when Monday => Initial_Balance;
  when Friday => Closing_Balance;
  when Tuesday..Thursday => Report(Today);
  when others => null;
end case;
```

# C++


Kifejezés
Érték1
utasítás
Érték2
utasítás
Érték3
utasítás
Érték4
utasítás
Érték5
utasítás
…
…
Értékn
utasítás

break

# C++

```
switch (intexpr) {
  case label1: statm1; break;
  case label2: statm2; break;
  ...
  default: statm;
}
```

kifejezés „integral” típusú

- „default:” címke lehetősége (nem kötelező).
- itt break kell, ha nem akarom folytatni!

# C++

```
switch (val){
   case 1 : cout<<"case 1\n";
   case 2 : cout<<"case 2\n";
   default: cout<<"default case";
}
```

ha val=1, az eredmény:

```
case 1
case 2
default case
```

# Java

Mint a C++:

```
switch (val){
 case 1 : System.out.println("case 1\n");
 case 2 : System.out.println("case 2\n"); break;
 default: System.out.println("default case");
}
```

ha val=1, az eredmény:

```
case 1
case 2
```

# Java

Értékadás

```
final String exampleString =
 switch (val){
  case 1 : "case 1"; break;
  case 2 : "case 2"; break;
  default: "default"; break;
 }
```

ha val=1, az eredmény:

```
 case 1
```

# Java

Értékadáshoz új szintaxis

```
final String exampleString =
 switch (val){
  case 1 -> "case 1";
  case 2 -> "case 1";
  default -> "default";
 }
```

Fontos változás, hogy itt nincsen átcsorgás!

ha val=1, az eredmény:

```
case 1
```

# Kotlin

Hasonlóan a Java új szintaxishoz, de általánosan:

```
when (x) {
     1 -> print("x == 1")
     2 -> print("x == 2")
     else -> {
          print("x is neither 1 nor 2")
     }
}
```

# Kotlin

Kifejezések és intervallumok használata

```
when (x) {
    parseInt(s) -> print("s encodes x")
    else -> print("s does not encode x")
}

when (x) {
    in 1..10 -> print("x is in the range")
    in validNumbers -> print("x is valid")
    !in 10..20 -> print("x is not in the range")
    else -> print("none of the above")
}
```

# Eiffel

Pascal-szerű szemantika

A változó típusa INTEGER vagy CHARACTER

```
inspect var
  when expr1 then statmblock1
  when expr2 then statmblock2
  ...
  else statmblock
end
```

Exception lép fel, ha nem gondoskodtunk a megfelelő választék-elemről (null utasítás szükséges lehet).

# Eiffel

```
inspect c
  when '0'..'9'                then n:= n + 1;
  when 'a'..'z','A'..'Z' then s:= s + 1;
  else  o:= o + 1;
end
```

# C#

```
switch (kif)
{
    case ertek1 : utasítások… break;
    case ertek2 : utasítások… break;
    case ertek3 : utasítások… break; vagy:
                  goto case KONST/default;
…
}
```

szemantika

Ciklusként is működhet!!

# SWIFT

Lehetséges az értékre rész-megszorítást adni

```
switch vegetable {
  case "celery":
    let vegetableComment = …
  case "cucumber", "watercress":
    let vegetableComment = …"
  case let x where x.hasSuffix("pepper"):
    let vegetableComment = …
  default:
    let vegetableComment = …
}
```

# SWIFT

Tuple használata esetén, lehetséges figyelmen kívül hagyni az értékek egy részét:

```
let somePoint = (1, 1)
switch somePoint {
case (0, 0):
    print("\(somePoint) is at the origin")
case (_, 0):
    print("\(somePoint) is on the x-axis")
case (0, _):
    print("\(somePoint) is on the y-axis")
case (-2...2, -2...2):
    print("\(somePoint) is inside the box")
default:
    print("\(somePoint) is outside of the box")
}
```

# Absztrakció, alprogramok

KÖVETKEZŐ HÉTEN




# Programozási nyelvek és módszerek

## TÍPUSABSZTRAKCIÓ, ALPROGRAMOK

# Vezérlési szerkezetek

# Ciklusok

utasítások ismételt végrehajtása

- valamilyen feltételtől függően



# Ciklusok általános kérdései

Vannak-e nem ismert lépésszámú ciklusok?

- Van-e elöl/hátul tesztelős ciklus?
- A ciklusfeltétel  logikai érték kell legyen, vagy más típusú is lehet?

Kell-e blokkot kijelölni a ciklusmagnak?

# Ciklusok általános kérdései

Van-e előre ismert lépésszámú ciklus?

- A ciklusváltozó mely jellemzője állítható be a következők közül?
  - alsó érték - felső érték - lépésszám
- Mi lehet a ciklusváltozó típusa?
- Biztosított-e a ciklusmagon belül a ciklusváltozó változtathatatlansága?
- Mi a ciklusváltozó hatásköre, definiált-e az értéke kilépéskor?

# Ciklusok általános kérdései

Van-e általános (végtelen) ciklus?

Léteznek-e a következő vezérlésátadó utasítások?

- `break` - `continue`

Van-e ciklusváltozó-iterátor?

# Nem ismert lépésszámú ciklusok

Elöltesztelő „while” ciklusok:

- `while <kif> do <utasítás>`

Pascal: a ciklusmag egyetlen utasítás lehet (de blokk is)

- `while a < b do b := b - a;`

# ADA

end loop zárja, így a ciklusmag utasításssorozat is lehet:

```
while <kif> loop
   <ciklusmag>
end loop
```

kifejezés Boolean típusú

# C++

A kifejezés aritmetikai értéket ad

- nem 0: true, 0: false

A ciklusmag  egyetlen utasítás lehet (de blokk is)

```
while (<expr> ) <statm>
while (a < b)
   b = b - a;
```

# Java

Hasonló a C++-hoz, kivéve: a kifejezés típusa boolean kell legyen.

```
i=0;
while (i<10) {
  System.out.println(i);
  i++;
}
```

# Eiffel

Amíg a ciklusfeltétel hamis!

- end zárja, így a ciklusmag utasításssorozat is lehet.

```
from
   init
[invariant
   loop-inv
   variant
   variant-func]
until
   loop-cond
loop
   loop-body
end
```



# Hátultesztelő „repeat-until” ciklusok

Amíg egy feltétel igaz nem lesz.

Ciklusmag egyszer biztosan lefut

```
repeat
   utasítássorozat
until ciklusfelt
```

nincs szükség blokk utasításra – a ciklusmag vége meghatározott

```
while E do S
```

- Jelentése

```
if E then
   repeat S until not E
```

# Hátultesztelő ciklusok

Pascal:

```
repeat
   b := b - a;
until (b <= a);
```

SWIFT:

```
var m = 2
repeat {
    m = m * 2
} while m < 10
```

# Hátultesztelő ciklusok

C++: a kifejezés aritmetikai érték, nem 0: igaz, amíg értéke 0 (hamis) nem lesz

```
do <statm> while (<expr>);
```

Java: hasonló a C++-hoz, kivéve: a kifejezés típusa boolean kell legyen

Eiffel:

- „until_do” az Iteration könyvtárból.

# Előre ismert lépésszámú ciklusok „For” ciklusok

index változó kezelése – lépésköz - határ

Egyszer, a ciklusba való belépés előtt értékeli ki a lépésközt és a határt, vagy minden végrehajtás után újra?

A határt a ciklusmag végrehajtása előtt vagy után ellenőrzi?

Mi lehet a ciklusváltozó típusa?

Biztosított-e a ciklusmagon belül a ciklusváltozó változtathatatlansága?

Mi a ciklusváltozó hatásköre, definiált-e az értéke kilépéskor?

# Pascal

Jellemzők

- lépésköz és határ kiértékelése egyszer
  - határ ellenőrzése ciklusmag előtt
- ciklusváltozó nem változtatható (ma már)
  - értéke kilépéskor nem definiált
- ciklusváltozó diszkrét típusú lehet

```
for i := 1 to n do sum := sum + i;
for c := 'z' downto 'a' do write(c);
```

# Kotlin

For ciklus

- előre ismert lépésszám

```
for (i in 6 downTo 0 step 2) {
    println(i)
}
```

# Ada

Az index a ciklus lokális konstansa.

- Diszkrét típusú kell legyen.
- A ciklus értékintervallumát csak egyszer, a ciklusba való belépés előtt számítja ki.

```
for <var> in loop-range loop
 <utasítások>;
end loop;
Sum:=0;
for I in 1..N loop
   Sum := Sum + I;
end loop;
```

A „reverse” hatására : fordított sorrend.

# SWIFT I.

Hasonlóan az előzőekhez

- Az ciklusváltozó felveszi a meghatározott intervallum értékét

```
var firstForLoop = 0
for i in 0..<4 {
    firstForLoop += i
}
```

A következő operátort `..<` használatával az `i` `0` és `3` közötti értékeket vesz fel.

- `i in 0...4`

Ha a ciklusváltozóra nincs szükség

```
for _ in 1...max {
  …
}
```

# SWIFT II.

A C++-hoz hasonlóan:

```
var secondForLoop = 0
for var i = 0; i < 4; ++i {
    secondForLoop += i
}
```

For-each

```
for number in numbers {
    if number > largest {
        largest = number
    }
}
```

# C++

a `for (for-init-stm [expr-1]; [expr-2]) stm` jelentése

```
{ for-init-stm
    while (expr-1) {
      stm
      expr-2;
    }
}
```

Kivéve, hogy egy `continue` a `stm`-ben `expr-2`-t hajt végre az `expr-1` kiértékelése előtt.

Hiányzó `expr-1` ekvivalens: `while(1)`

Ha a `for-init-stm` egy deklaráció, akkor a deklarált nevek hatásköre a `for` utasítást tartalmazó blokk

```
sum=0;
for (int i = 1; i < n; i++)
  sum = sum + i;
```

# Java

Eredeti for ciklus: hasonló a  C++-hoz:

```
public class ForDemo {
    public static void main(String[] args) {
        int[] arrayOfInts = { 32, 87, 199, 622, 127, 485 };
        for (int i = 0; i < arrayOfInts.length; i++) {
            System.out.print(arrayOfInts[i] + " "); i++;
        }
        System.out.println();
    }
}
```

Mit csinál?

# Java

for-each ciklus:

```
public class ForEachDemo {
  public static void main(String[] args) {
      int[] arrayOfInts = {32, 87, ...., 622, 127};
      for (int i : arrayOfInts) {

              System.out.print(i + " ");
     }
      System.out.println();
  }
}
```

# Java

Gyűjteményekkel (Collection) és tömbökkel használható

Az iterációk leggyakoribb formájára, amikor az index, illetve az iterátor értéket semmilyen más műveletre nem használják, csak az elemek elérésére.

Még egy példa:

- …. ha van egy Number típusunk:

```
List<Number> szamok = new ArrayList<Number>();
szamok.add(new Integer(42));
szamok.add(new Integer(-30));
szamok.add(new BigDecimal("654.2"));
for ( Number number : szamok ){
  …
}
```

# C#

„hagyományos” for ciklus, hasonlóan a C++-hoz:

```
using System;
public class ForLoopTest
{
   public static void Main()
   {
      for (int i = 1; i <= 5; i++)
         Console.WriteLine(i);
   }
}
```

# C#

foreach utasítás:

```
foreach (type identifier in expression) statement
```

- a ciklusváltozó értékét ne változtassuk, ha ez érték típusú, akkor nem is lehet.
- A kifejezés gyűjtemény vagy tömb lehet, `IEnumerable`-t implementál, vagy egy olyan típust, ami deklarál egy `GetEnumerator` metódust.

# C#

foreach tömbökre:

```
public static void Main() {
        int odd = 0, even = 0;
        int[] arr = new int [] {0,1,2,5,7,8,11};
        foreach (int i in arr) {
            if (i%2 == 0)
                even++;
            else
                odd++;
        }
        …
    }
}
```

# C#

## foreach gyűjteményekre

- a myCollection gyűjteményre a következőknek kell teljesülnie:
  - interface, class, vagy struct típus kell legyen
  - kell legyen egy GetEnumerator nevű példánymetódusa, aminek a visszaadott típusára (pl. Enumerator ) fennáll:
    - Van egy Current nevű property-je, ami ItemType-ot, vagy erre konvertálhatót ad vissza – a gyűjtemény aktuális elemét
    - Van egy MoveNext, bool-t visszaadó metódusa, ami növeli az elemszámlálót, és true-t ad, ha van még elem a gyűjteményben
- Létrehozunk egy gyűjteményt a fenti szabályokkal – ez csak C# programokban használható persze
- Létrehozunk egy gyűjteményt a fenti szabályokkal, és implementáljuk az IEnumerable interfészt
  - ez más nyelvekben is használható lesz, mint pl. a Visual Basic
- Használjuk az előredefiniált gyűjtemény osztályokat

# „Végtelen” ciklusok

ADA

```
loop
   <statm>1; …
      exit when <feltétel>;
   <statm>2; …
end loop;

loop
   get(Current_Char);
   exit when Current_Char =’*’;
end loop;
```

# Vezérlésátadó utasítások

`break`: kiugrás a vezérlési szerkezetből

`continue`: ciklusfeltétel újrakiértékelése

alprogram hívás

`return` alprogramból hívóhoz visszatérés.

`goto` – !!! Veszélyeket rejt

# Példák

C++:

- `break` – ciklusban, vagy switch utasításban
- `continue` – csak ciklusban
- `return`
- `goto`

Java:

- `break`, continue, `return` mint a C++-ban
- NINCS goto!

# Példák

Kotlin

- `break` – ciklusban
- `continue` – ciklusban
- `return` – kilép a legközelebbi függvényből, anonim függvényből
- Címkézés – megadható, hogy hova adódjon át a vezérlés

```
loop@ for (i in 1..100) {
    for (j in 1..100) {
        if (...) break@loop
    }
}
```

# Példák

SWIFT

- Hasonlóan a C++-hoz:
  - `continue`
  - `break` – ciklusban szokásos, switch esetén kilép!
  - `return`
- Switch esetén továbbcsorgat: `fallthrough`

```
let integerToDescribe = 5
var description = "The number \(integerToDescribe) is"
switch integerToDescribe {
case 2, 3, 5, 7, 11, 13, 17, 19:
    description += " a prime number, and also"
    fallthrough
default:
    description += " an integer."
}
print(description)
```

# Python

`for`/`while` ciklus esetén:

- `continue`
- `break`
- **`else`**

Példa

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            break
    else:
        print(n, ' prím’)
```

A `for` ciklushoz tartozik az `else`!

Az `else` ág akkor fut le, ha az összes ciklusiteráció lefut.

- Nincs `break` hívás

# Absztrakció

# Absztrakció – Mi az?

## Gondolkodásmód

- Az általános elvekre koncentrálunk, nem ezeknek a specifikus megjelenési formáira.
  - Filozófia,
  - Matematika,
  - Számítástudomány

## Absztrakció a rendszer analízisben:

- A feladat lényeges aspektusai
  - Eltekintünk a kevésbé lényegestől
- Például: légi közlekedés vezérlése
  - lényeges: helyzete, sebessége stb.
  - lényegtelen: színe, utasok neve…
    - (ez a feladattól függ!)

# Alprogramok mint absztrakciók

Absztrakció a programozásban

- Különböztessük meg a szinteket
  - Mit csinál a program?
  - Hogyan van implementálva?

Minden programozási nyelv a gépi kód absztrakciója.

Magasabb szintű absztrakciók

- Eljárás: Mit csinál az eljárás?
  - Hogyan csak akkor  fontos, amikor implementáljuk.
- Eljárást hívó eljárások:
  - Az absztrakció akárhány szükséges szintje bevezethető.

# Alprogramok mint absztrakciók

## Absztrakciós mechanizmus

- A programozási nyelvi konstrukció, ami megengedi, hogy a programozó megragadja az absztrakciót
  - És a program részeként reprezentálja
    - egyfajta számítási mintaként
- Egyszerű példa
  - Eljárások és függvények
- Egyéb példák
  - Később...

# A legegyszerűbb absztrakciós mechanizmus

## Eljárások és függvények

- Egységek, melyek számításokat tartalmaznak
  - Függvény-absztrakció: egy kiszámítandó kifejezést tartalmaz
  - Eljárás-absztrakció: egy végrehajtandó parancsot tartalmaz.
- A tartalmazott számítás mindig végrehajtásra kerül, amikor egy absztrakciót hívunk.

## Az érdekeltségek szétválasztása:

- A Hívó azzal törődik, mit csinál a számítás.
- Az Implementáló azzal is törődik, hogy hogyan kell a számítást végrehajtani természetesen a „mit” szerint

## A hatékonyságot a paraméterezéssel javítjuk.

# Függvény-absztrakció

Egy kiértékelendő kifejezés – amikor hívjuk, egy értéket ad vissza

Ada példa

```
function Kerulet (R : Float) return Float is
begin
    return 2.0*R*3.14;
end;
```

# ML példa

```
fun power( x: real; n: int) =
    (* felt. hogy  n > 0 *)
    if n = 1 then
        x
    else
        x * power (x, n- 1)
    end
```

# Mi a függvény definíció?

```
function I (FP1; ... ; FPn) return T is
body
```

Egy `I` azonosítót hozzákapcsol egy adott függvény-absztrakcióhoz.

A függvény-absztrakció a megfelelő aktuális paraméterekkel való híváskor eredményt ad vissza.

- A függvényhívás felhasználói szemlélettel egy leképezés az argumentumok és az eredmény között.
- Megvalósítói szemlélet: a függvénytörzs kiértékelése.
  - Az algoritmus változása csak a megvalósítóra tartozik.

# Eljárás-absztrakció

Egy végrehajtandó parancsot testesít meg.

- A felhasználó csak a változók megváltozását érzékeli.

Általános formátum:

```
procedure I ( FP 1 ; ... ; FP n )
body
```

Felhasználói szemlélet:

```
type Dictionary = array [...] of Word;
procedure sort( var words: Dictionary);

sort(a); (* érzékeljük 'a' változását *)
```

Megvalósítói szemlélet: a kódolt algoritmus.

# Absztrakciós elv

Összefoglalás:

- Függvény-absztrakció: egy kifejezés absztrakciója.
  - Egy  függvényhívás egy kifejezés, amikor hívjuk, kiértékeli, és egy értéket ad vissza.
- Eljárás-absztrakció: egy parancs absztrakciója.
  - Egy eljáráshívás egy parancs, ami az eljárástörzs végrehajtása során frissíti/frissítheti a változók értékét.

# Absztrakciós elv

## Általánosítás

- Tetszőleges szintaktikai osztály fölött létrehozhatunk absztrakciókat, feltéve, hogy ez valamifajta számítást specifikál.
  - Absztrakt adattípusok
  - Sablonok
    - Generic, később …

# Alprogramok és paramétereik

Alprogramok használata

- az egyik legelső programozási eszköz.

Charles Babbage – Analytical Engine

- 1840-ben már tervezte, hogy lyukkártyák egy csoportját fogja használni nagyobb számítások gyakrabban ismételt részeinél

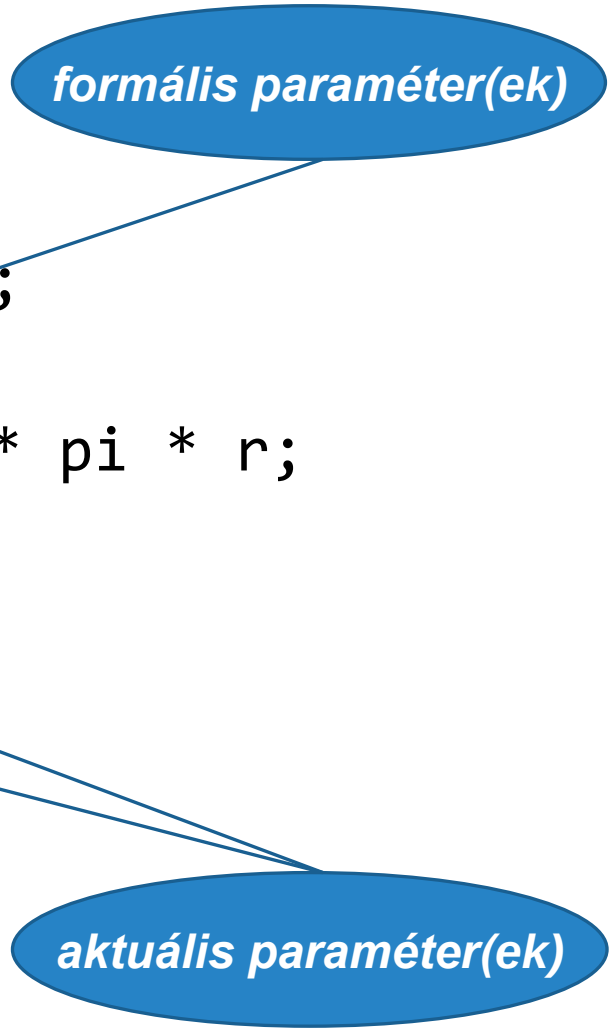
Alprogramok használatával nevet adhatunk egy kódrészletnek és paraméterezhetjük a viselkedését.

# Paraméter átadása

Absztrakció általánosítása

```
val pi = 3.14159;
val r = 1.0;
fun circumference() = 2 * pi * r;

fun circumference (r: real) = 2 * pi * r;
circumference (1.0);
circumference(a + b);
```



# Alprogram

Programegység

Végrehajtás kezdeményezése: meghívással

A program darabolásának eszköze

Különböző számítások elkülönítése egymástól

Újrafelhasználhatóság

# Mire való

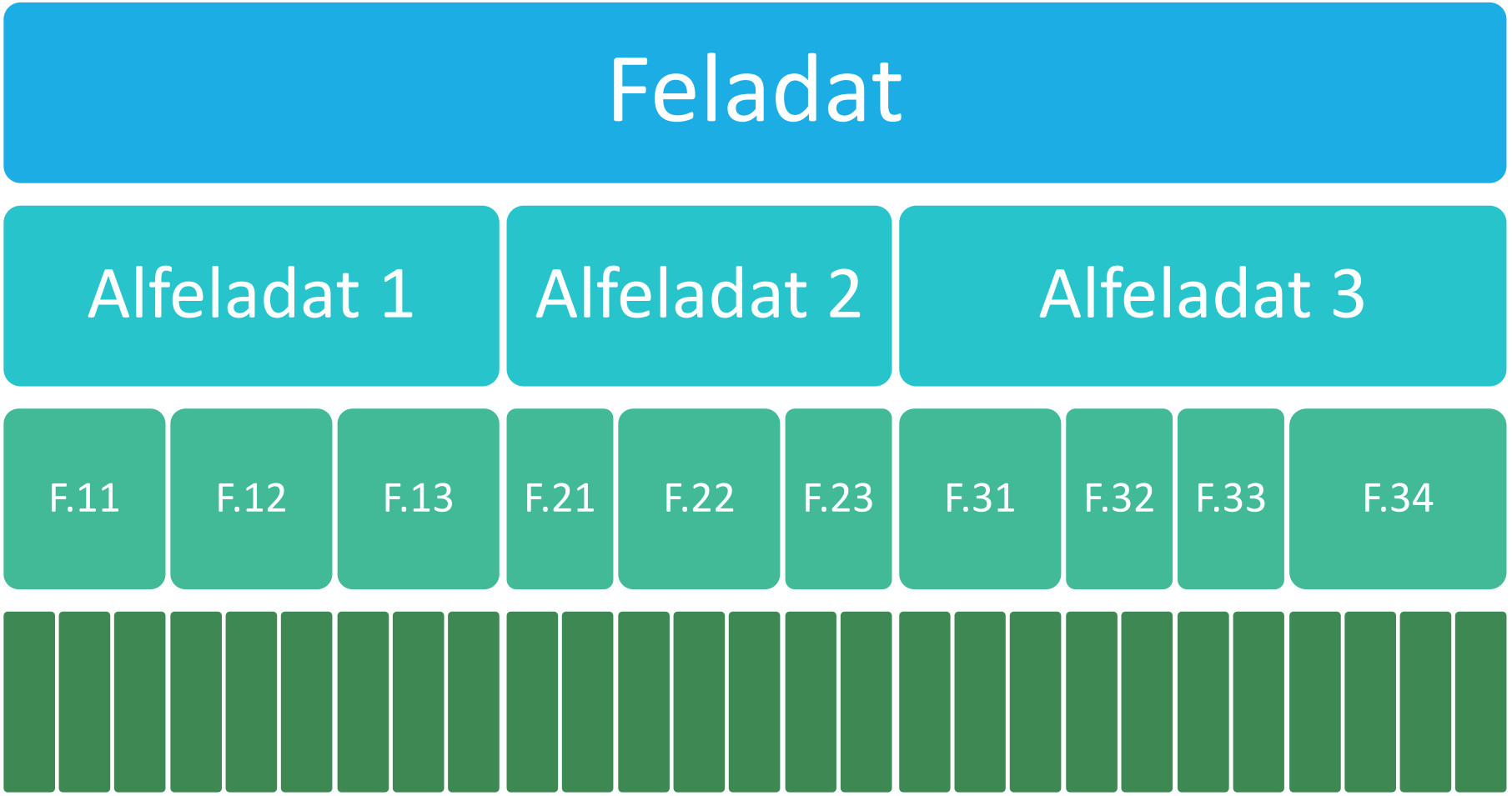
Nagyobb feladat egy részfeladatának megoldása

Algoritmus kódolása

Típusművelet megvalósítása

Matematikai függvény kiszámítása

# Eljárás-absztrakció

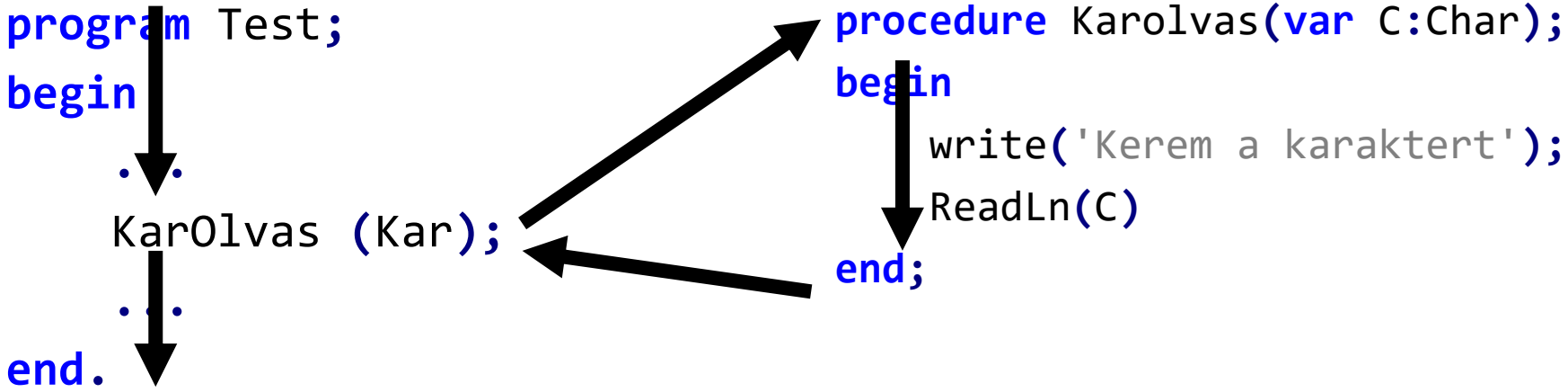


# Alprogram hívása

```
program Test;
procedure Karolvas(var C:Char);
begin
    write('Kerem a karaktert');
    ReadLn(C)
end;

var
    Kar: Char;
begin
    ...
    KarOlvas (Kar);
    ...
end.
```

# Alprogram hívása



# Alprogram

Alprogram=(név, paraméterek, környezet, törzs).

- Az alprogram definíciójakor a formális paraméterekkel írjuk le az adatcsere elemeit
  - A külvilággal való kommunikáció alterének komponenseit
- Az alprogram használatakor pedig az aktuális paraméterek kerülnek ezek helyére
  - Ez a paraméterátadás

# Függvény

```
function Faktorialis ( N: Natural ) return Positive
is
     Fakt: Positive := 1;
begin
     for I in 1..N loop
          Fakt := Fakt * I;
     end loop;
     return Fakt;
end Faktorialis;
```

# Eljárás

```
procedure Cserel ( A, B: in out Integer ) is
    Temp: Integer := A;
begin
    A := B;
    B := Temp;
end Cserel;
```

# Eljárás – függvény

Eljárás végrehajtása: utasítás

Eljárás neve: ige

- `Egyenest_Rajzol(Kezdopont,Vegpont);`

Függvény végrehajtása: kifejezés értékének meghatározása.

Függvény neve: főnév vagy melléknév.

- `if Elemek_Száma(Halmaz) > 0 then ...`

# Paraméterek, visszatérési érték

Információ átadása / átvétele alprogramhívásnál

- paramétereken keresztül
- visszatérési értéken keresztül

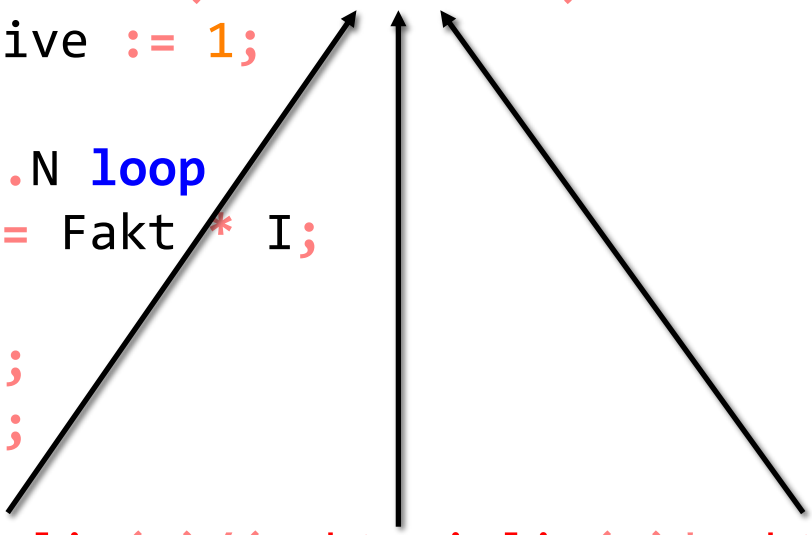
Paramétereknél az információ áramlása

- Merre: a paraméterek módja
- Hogyan: a paraméterátadás technikája (paraméterátadás módja)

Alprogram hívásakor a formális paramétereknek aktuális paramétereket feleltetünk meg.

# Paraméterek

```
function Faktorialis ( N: Natural ) return Positive is
    Fakt: Positive := 1;
begin
    for I in 1..N loop
        Fakt := Fakt * I;
    end loop;
    return Fakt;
end Faktorialis;
```



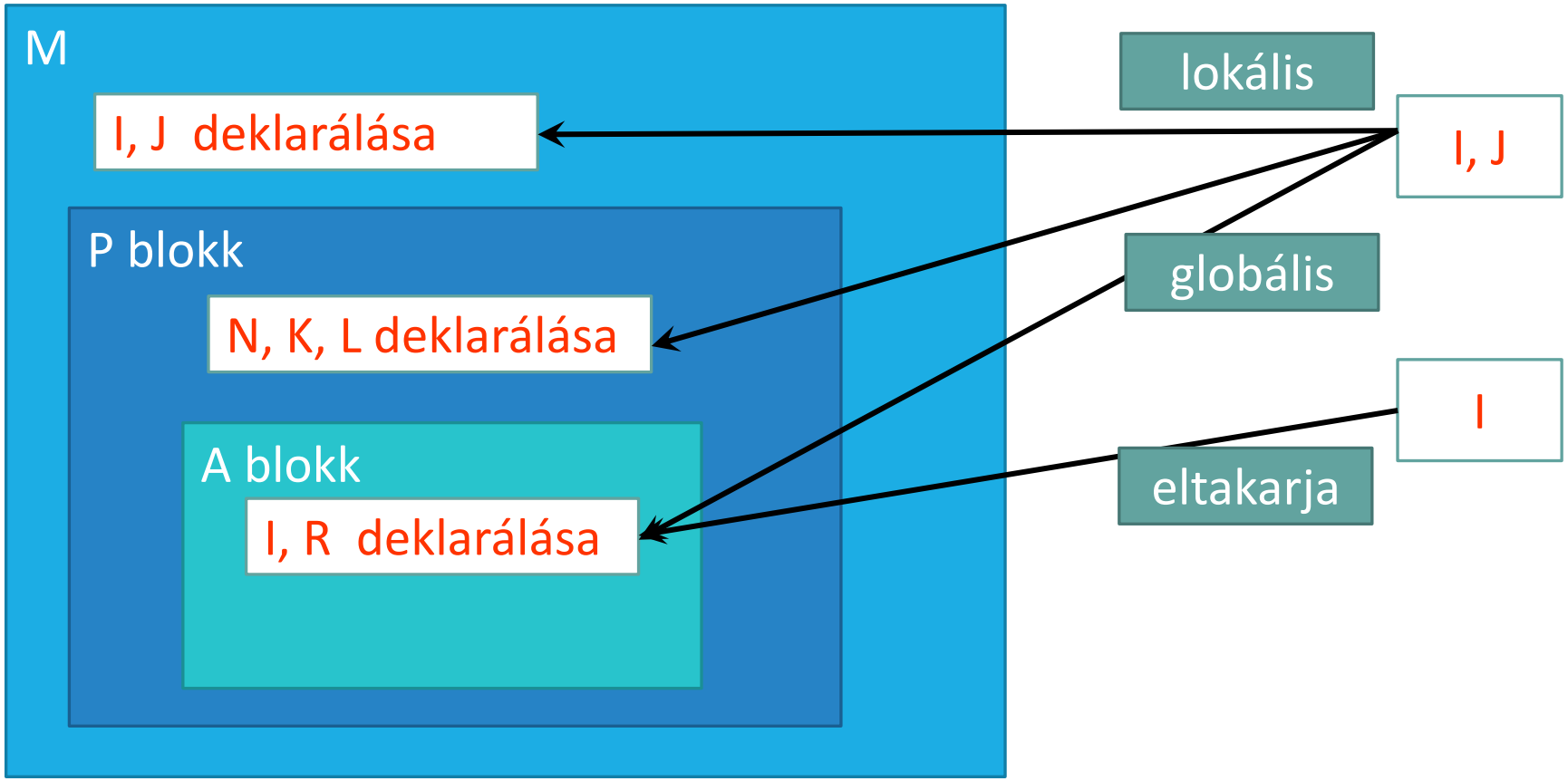
```
L_Al_K:=Faktorialis(L)/(Faktorialis(K)*Faktorialis(L-K));
```

# Alprogram beágyazható

Sok programozási nyelvben:

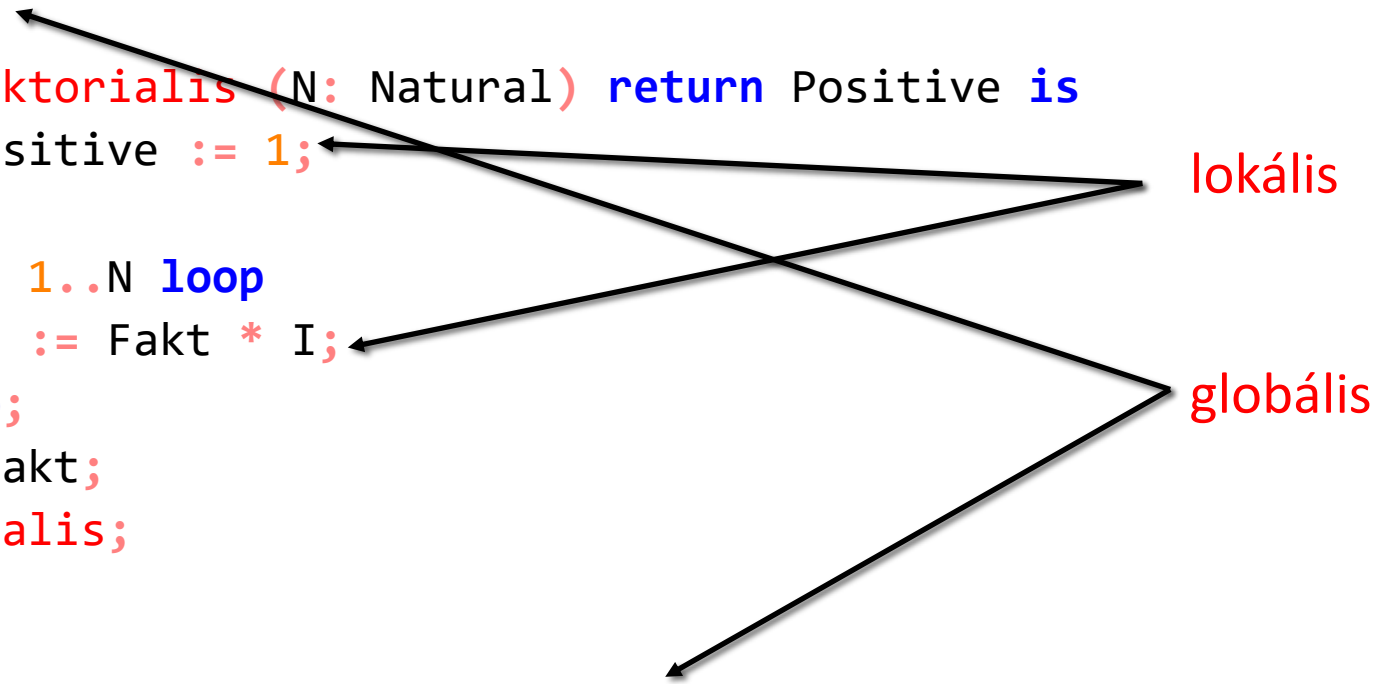
- Blokkszerűen
- Deklarációs részbe
- Lokális - globális deklarációk
- Hatáskör, láthatóság, élettartam

# Változók


M
I, J deklarálása
P blokk
N, K, L deklarálása
A blokk
I, R deklarálása
lokális
I, J
globális
I
eltakarja

# Példa

```
with Ada.Integer_Text_IO, Text_IO;
procedure Faktorialist_Szamit is
   N: Integer;
   function Faktorialis (N: Natural) return Positive is
      Fakt: Positive := 1;
   begin
      for I in 1..N loop
          Fakt := Fakt * I;
      end loop;
      return Fakt;
   end Faktorialis;
begin
   N := 10;
   Ada.Integer_Text_IO.Put(Faktorialis(N));
end Faktorialist_Szamit;
```



# Paraméterek fajtái

Az információáramlás iránya szerint

- Input: hívó $\Rightarrow$ alprogram
- Output: hívó $\Leftarrow$ alprogram
- Update: hívó $\Leftrightarrow$ alprogram

# Paraméterátadási technikák

## A paraméterátadás módja

- Különféle nyelvekben különféle módon adják át a paramétereket
- Legismertebb paraméterátadási módok:
  - érték szerint
  - cím szerint
  - eredmény, érték-eredmény szerint
  - név szerint

# Érték szerinti paraméterátadás

A formális paraméter az alprogram egy lokális változója

- aminek az aktuális paraméter adja a kezdőértéket
- in módú átadásra alkalmas

Az aktuális paraméter

- Tetszőleges kifejezés
- Kiértékelésére egyszer kerül sor
  - Az alprogram végrehajtásának megkezdése előtt

Az alprogram végrehajtása közben

- sem az aktuális paraméter értékének esetleges megváltozása nincs hatással a formális paraméter értékére
- sem a formális paraméter értékének megváltoztatása nincs hatással az aktuális paraméterre

# Érték szerinti paraméterátadás

Példa C-ben

```
int lnko( int a, int b ){
  while( a != b )
    if( a > b )   a -= b;
    else          b -= a;
  return a;
}
```

```
int x,y,z; …
x = 10; y = 5;
z = lnko(x,x+y+1);

a = 10
b = 16
```

# Érték szerinti paraméterátadás

Példa C-ben

```
int lnko( int a, int b ){
  while( a != b )
    if( a > b )   a -= b;
    else          b -= a;
  return a;
}
```

```
int x,y,z; …
x = 10; y = 5;
z = lnko(x,x+y+1);
```

z értéke 2 lesz,
x marad 10

# Érték szerinti paraméterátadás

Csak egyszer értékelődik ki

C példa:

```
int x = 1;
int f( int a ){
  ++x;
  return a+x;
}
int main(){
  int y = f(x);
}
```

a = 1



x = 2

y = 3

# Cím szerinti paraméterátadás

Az aktuális paraméter
- egy változó, vagy
- egy változó komponensét meghatározó kifejezés – balérték - `(x[i+2*j])` lehet.

Az aktuális paraméter kiértékelése
- a hozzá rendelt memóriaterület címének meghatározását jelenti.
- az alprogram végrehajtásának megkezdése előtt kerül sor
  - az aktuális paraméter memóriaterülete rendelődik hozzá.

Az alprogramban
- hivatkozhatunk a formális paraméter értékére,
- adhatunk is neki új értéket
  - update módú átadásra is alkalmas

# Cím szerinti paraméterátadás

Példa Pascalban:

```
procedure Csere
  (var a, b: Integer );
var temp: Integer;
begin
  temp := a;
  a := b;
  b := temp
end;
```

Hívás:

```
…
x: Integer;
y: Integer;
…
x:=3;
y:=6;
Csere(x,y);
```

x=6, y=3

# Cím szerinti paraméterátadás

```
procedure s (var a   : integer);
     // kiírja és lenullázza a paraméterét

procedure p (var a, b: integer);
     // kiírja és lenullázza a paramétereit
begin
   s(a);
   s(b)
end;
```

Mit csinál majd `p(x,x)`?

# Cím szerinti paraméterátadás

„Alias” – amikor a program egy pontján két különböző változó ugyanazt az egyedet jelenti

```
var globalis: Integer;

procedure r (var lokalis: Integer);
begin
    globalis := globalis + 1;
    lokalis := lokalis + globalis
end;

begin
  globalis := 1;
  r(globalis)
end.
```

Mennyi lesz `globalis` értéke?

# Eredmény szerinti paraméterátadás

A kimenő paraméterek megvalósításához.

- Az alprogram formális paraméterében kiszámított eredményt helyezi vissza az aktuális paraméterbe.

A formális paraméter

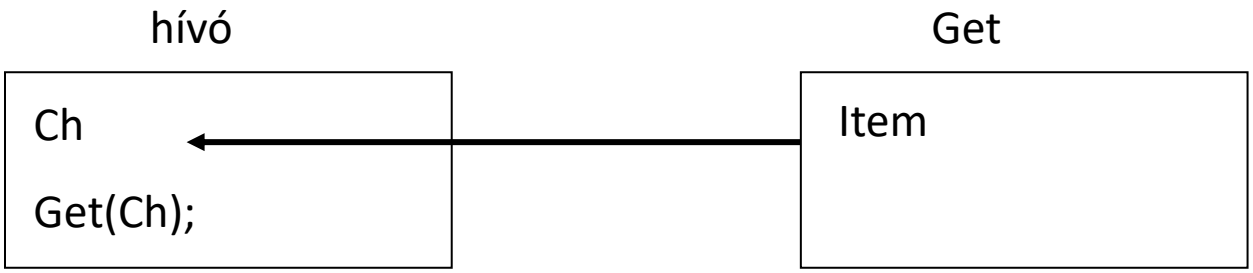
- az alprogram lokális változója,
- értéke az alprogram befejeződésekor kimásolódik az aktuális paraméterbe.
- nem kapja meg az aktuális paraméter értékét az alprogram hívásakor
  - az információáramlás egyirányú (out módú átadásra alkalmas).

Az aktuális paraméter balérték kell legyen.

# Eredmény szerinti paraméterátadás

Ada példa:

```
procedure Get(Item: out Character ) is …
```




Object Pascal példa:

```
procedure GetInfo(out Info: SomeRecordType);
```

# Érték/Eredmény szerinti paraméterátadás

Az alprogram befejezésének pillanatáig megegyezik az érték szerinti paraméterátadással.

Az alprogram végrehajtásának befejezésekor az aktuális paraméter felveszi a formális paraméter pillanatnyi értékét (update módú átadásra alkalmas).

Az aktuális paraméter csak „balérték” lehet

Miben különbözik a cím szerinti átadástól?

# Érték/Eredmény szerinti paraméterátadás

```
procedure Paramproba is
  A,B:Integer;
  procedure Parcsere(X,Y: in out Integer) is
    Temp: Integer;
  begin
    Temp:=X; X:=Y; Y:=Temp;
    A:=12; B:=99;
  end;
begin
  A:=1; B:=2;
  Parcsere(A,B);
end;
```

X=1,Y=2

X=2,Y=1

A=1,B=2

A=2,B=1

# Érték/Eredmény szerinti paraméterátadás

```
procedure s (a    : in out integer) is …
    -- kiírja és lenullázza a paraméterét

procedure p (a, b: in out integer) is
    -- kiírja és lenullázza a paramétereit
begin
   s(a);
   s(b);
end;
```

Mit csinál majd `p(x,x)`?

# Érték/Eredmény szerinti paraméterátadás

```
procedure Program is
 globalis: Integer;
 procedure r (lokalis: in out Integer) is
 begin
  globalis := globalis + 1;
  lokalis := lokalis + globalis;
 end r;
begin
  globalis := 1;
  r(globalis);
end Program;
```

Itt mennyi lesz a `globalis`?

# Név szerinti paraméterátadás

Az aktuális paraméterként leírt teljes kifejezés adódik át

- minden használatkor (dinamikusan!) kiértékelődik;
- például az a[i] hivatkozás változhat, ha menet közben az i értéke változik(!)
- (update módú átadásra is alkalmas)

Lusta kiértékelés…

# Példa

```
program
  I: integer;
  A: integer array;
  SWAP_BY_NAME: procedure(X: name, Y: name)
   TEMP: integer;
   begin
    TEMP := X;  X := Y;   Y := TEMP;
  end;
 begin
  I := 3;
  A[I] := 6;
  output I, A[3];
  SWAP_BY_NAME(I, A[I]);
  output I, A[3];
 end;
```

*Output*

```
I = 3   A[3]  = 6
I = 6   A[3]  = 6 de A[6] = 3 lesz!
```

# Egy híres példa

## Jensen's device  kifejezések kiértékelésére – Algol60

```
real procedure sum (expr, i, low, high);
   value low, high;
   real expr;          -- név szerint az expr és az i,
                       -- ez a default Algolban!
    integer i, low, high;
    begin
     real rtn;
     rtn := 0;
     for i := low step 1 until high do
        rtn := rtn + expr;
     sum := rtn;
end sum
y := sum (3*x*x - 5*x + 2, x, 1, 10);
```

Mit csinál?

# Alprogramok paraméterei

Az alprogramok paramétereinek száma általában kötött, de lehet változó is.

- Bizonyos mértékig szimulálható egy tömb átadásával
- Ezzel nem mindig oldható meg eltérő típusú paraméterek átadása.

# Paraméterek száma

A programnyelvek kezelhetik a túl sok vagy túl kevés aktuális paraméter megadását.

A túl kevés paraméter megadása esetén több módszer alkalmazható:

- alapértelmezett értékek adhatók meg, amelyek a hiányzó aktuális paraméterek helyébe lépnek (pozíció vagy név szerint)
- vesszőket kell tenni a kihagyott paraméterek helyett
- amíg nincsenek alapértelmezett értékek, addig kötelező az aktuális paramétereket megadni, tehát az alapértelmezések csak a paraméterlista végén lehetnek

# Formális-aktuális paraméterek megfeleltetése

```
procedure Get_Line ( File : in  File_Type;
        Item : out String; Last : out Natural )
```

pozícionális formában: az aktuális paramétereket abban a sorrendben kell felsorolni, ahogy a formális paraméterek voltak az alprogram specifikációjában:

```
Get_Line(F, S, N);
```

névvel jelölt formában: a paraméterek sorrendje tetszőleges:

```
Get_Line(File=>F, Last=>L, Item=>S);
```

kevert formában: mindig a pozícionálisan megadott paramétereknek kell elöl állniuk:

```
Get_Line(F, Last=>L, Item=>S);
```

# Paraméterek feltételezett értéke

Az in módú paraméterekhez rendelhetünk feltételezett bemenő értéket.

- Az alprogram specifikációs része tartalmazza ezt a feltételezett értéket
- Ez akkor veszi fel a formális paraméter, ha neki megfelelő aktuális paramétert nem adunk meg.

# Példa

```
procedure New_Line (File : in File_Type;
                    Spacing : in Positive_Count := 1 )
```

Hívhatjuk

- úgy, hogy megadjuk a Spacing értékét, vagy
- úgy, hogy nem ilyenkor a feltételezett értéket használja az eljárás.

```
New_Line(F);        -- 1 sort emel az F fájlban
New_Line(F,1);      -- az előzővel egyenértékű
New_Line(F,42);     -- 42 sort emel
```

# Paraméterek feltételezett értéke

Lehet több alapértelmezett értékkel rendelkező paraméter is.

- Ilyenkor bármelyik aktuális paramétert el lehet hagyni, akár többet is (segít a névvel jelölt hívás).

```
procedure Egyenest_Rajzol (
                Kezdőpont, Vegpont: in Pont;
                Vastagsag: in Positive := 1;
                Szin: in Szinskala := Fekete )
Egyenest_Rajzol(P1,P2);
Egyenest_Rajzol(P1,P2,3);
Egyenest_Rajzol(P1,P2,Szin=>Piros);
```

# Alprogramok túlterhelése (átlapolása)

```
procedure Cserél ( A, B: in out Integer );
procedure Cserél ( A, B: in out Boolean );
```

Azonos név – paraméterek száma és/vagy típusa különböző

A használatból (a hívásból) fog kiderülni, hogy melyikre gondolunk

- Ki kell derülnie!

# Kérdések

Mellékhatás – mi az?

```
z:= sin(x);
y:= sin(x);
z = y?
```

mi az elvárásom?

```
zz:= rnd();
xx:= rnd();
zz = xx
```

és most?

# Kérdések

Mellékhatás 1.: globális változók használata

```
with Text_Io; use Text_Io;
procedure Globprobe is
      I:Integer;
      function Glob(J:Integer) return Integer is
      begin
            I:=I+1;  return I+J;
      end;
begin
      I:=2;
      Put_Line(Integer'Image(Glob(1)));
            …
      Put_Line(Integer'Image(Glob(1)));
end;
```



# Kérdések

Mellékhatás 2.: függvény paraméter is változik

```
int f(int val, int& ref){
  val++;
  ref++;
  return val+ref;
};
void main(){
  int i=1;
  int j=1;
  int k;
  k=f(i,j);
}
```



VIGYÁZAT!!

# Operátorok

Infix alakban is írható műveletek

```
A+42        ”+”(A,42)
```

Operátorok is átlapolhatók:

```
function "+" ( A, B: Matrix ) return Matrix;
```

# Rekurzió

Közvetlenül vagy közvetve önmagát hívó alprogram

```
function Faktorialis ( N: Natural ) return Positive is
begin
    if N > 1 then
            return N * Faktorialis(N-1);
    else return 1;
    end if;
end Faktorialis;
```

# Rekurzió

A rekurzív alprogramok paraméterátadás szempontjából nem különböznek a szokásos alprogramoktól.

Ezért a paramétereket rekurzív alprogramoknak cím szerint átadni csak elővigyázatosan szabad, mert a sorozatos hívások interferálhatnak.

# Fontosabb kérdések

Van-e eljárás?

Van-e függvény?

Mely paraméterátadási módok léteznek?

- érték szerinti
- eredmény szerinti
- érték-eredmény szerinti
- cím szerinti / konstans cím szerinti
- név szerinti

# További fontos kérdések

Meghatározható-e a paraméterek információátadásának iránya?

Adható-e a formális paramétereknek alapértelmezett érték?

Túlterhelhetők-e az alprogramnevek?

Az operátorok átlapolhatók-e?

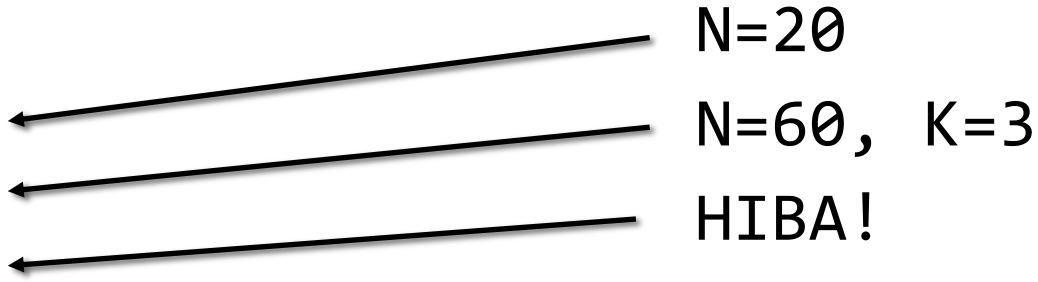
Definiálhatók-e új operátorok?

Típus-e az alprogram? Lehet-e alprogrammal paraméterezni?

# Pascal

A paraméterek alapértelmezésben érték szerint adódnak át, de a VAR kulcsszóval cím szerinti átadás érhető el:

```
program A;
  procedure Megszoroz(var Mit: Integer; Szorzo: Integer);
  begin
     Mit:= Mit * Szorzo;
  end;
var N, K :Integer;
begin
  N:=5; K:=3;
  Megszoroz(N, 4);
  Megszoroz(N, K);
  Megszoroz(5, N)
end.
```



# C/C++

Csak függvény van

- `void` visszatérési érték, ha eljárást szeretnénk

Csak érték szerinti paraméter átvétel

- cím szerinti: mutatók vagy referencia kell.

Lehet a paraméterek számára is kezdőértéket adni.

```
void f(int val, int& ref){
  val++;
  ref++;
}
```

```
void g(){
  int i=1;
  int j=1;
  f(i,j);
}
```

i==1,
j==2 lesz

# C++

const& paraméter – nem változtatható a törzsön belül

```
float fsqrt(const float&);
...
float r=fsqrt(x);
```

Tömb paraméterek

- a tömb első elemére hivatkozó mutató adódik át
- vagyis nem érték szerint adódnak át
- méretük nem adódik át

```
int strlen(const char*);
void f(){
  char v[]='egy tomb';
  int i=strlen(v);
}
```

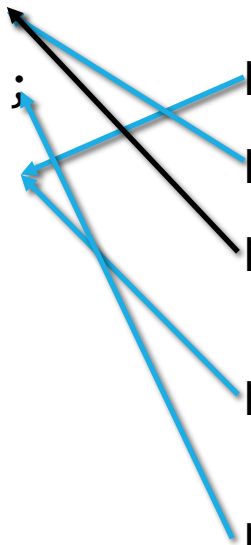


A T[ ] típusú paraméter T* típusúvá lesz átalakítva ekkor.

# C++

## Túlterhelés lehetséges:

```
void print(int);
void print(const char*);
void print(char);
…
```



```
void h(char c, short s, int i){
 print(c); //Pontos illeszkedés
 print(i); //Pontos illeszkedés
 print(s); //Konverzió

 print('a');//Pontos illeszkedés

 print("a");
           //Pontos illeszkedés
}
```

# Java

Csak valamely osztály metódusa lehet

Az aktuális objektum attribútumaira hivatkozik

Paraméterátadás

- primitív típusok érték szerint
- összetett típusú paraméterek referencia szerint
  - mutable-immutable lehetőség!

# Smalltalk

Üzenetküldésekkel érjük el a metódusokat, így a paraméterátadás érték szerintinek tekinthető – referenciák figyelembevételével

- A következő sor az a objektumnak küld üzenetet, két paraméterrel a két paraméter megnevezésével és aktuális értékével
- `a at:3 put:5`

# Ada

`in, out, in out`

Az összetett típusú értékek esetén a fordítóprogram választhat az érték-eredmény, illetve a cím szerinti átadás között.

A függvényeknek csak in paramétereik lehetnek.

Az in paramétereknek lehet alapértelmezett értékük is.

Az alprogramok határozatlan méretű tömb típust is elfogadnak a formális paramétereik között.

# SWIFT

## Lehetőségek

- Konstans paraméter, inout paraméter, kevert formában történő átadás, feltételezett érték létezik, függvényt is át lehet adni paraméterként, változó számú paraméter átadása

## Továbbá:

- Argumentum címke és függvényparaméter megkülönböztetése
- Több visszatérési értéke is lehet

# SWIFT

```
func greet(name s1: String, day s2: String) ->
String {
    return "Hello \(s1), today is \(s2)."
}
greet(name: "Bob", day: "Tuesday")

func calculateStatistics(scores: [Int]) ->
    (min: Int, max: Int, sum: Int ) {
  …
  return (min, max, sum)
}
```

# Kotlin

Infix függvény definiálása

```
infix fun Int.shl(x: Int): Int { ... }
1 shl 2
```

Vezérlési szerkezetekkel összeköthető – paraméterezhető kifejezések:

```
fun hasPrefix(x: Any) = when(x) {
    is String -> x.startsWith("prefix")
    else -> false
}
```

# Procedurális- vagy adatabsztrakció

Egy szoftver rendszer mindig végrehajt

- bizonyos tevékenységeket
- bizonyos adatokon.

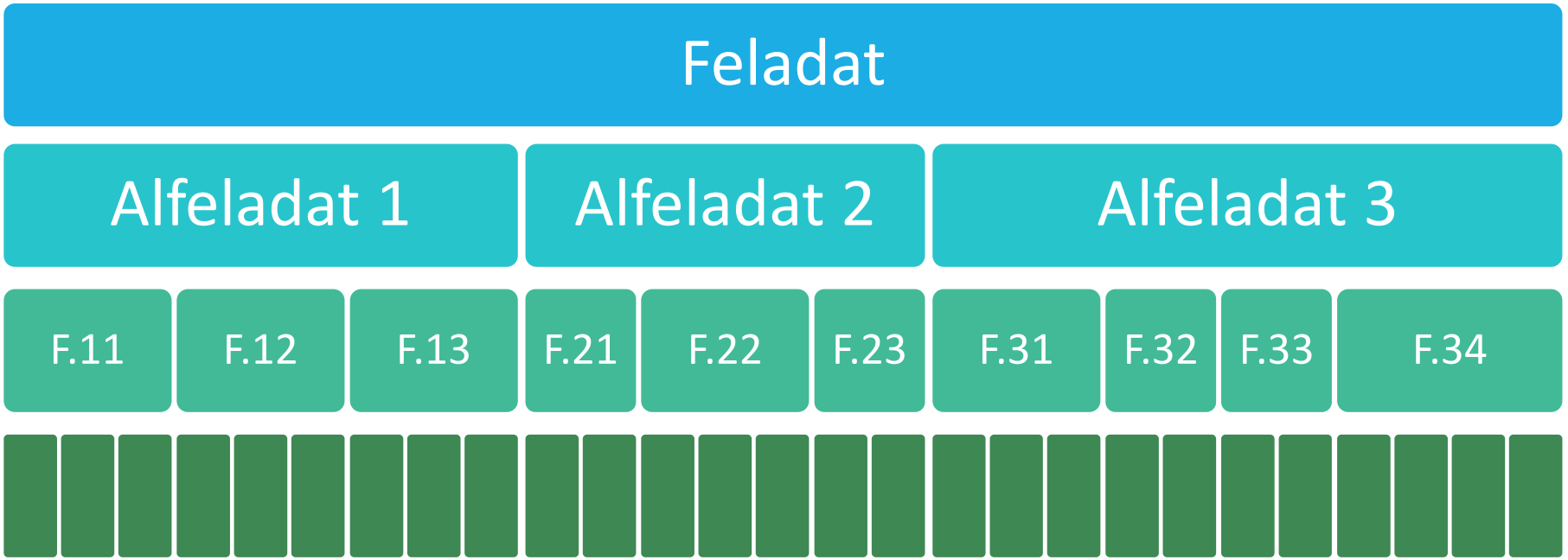
A tervezés központi kérdése, hogy a rendszer felépítését mire alapozzuk

- a végrehajtandó tevékenységekre, vagy
- az adatokra?

# Procedurális absztrakció

Top-down megoldás

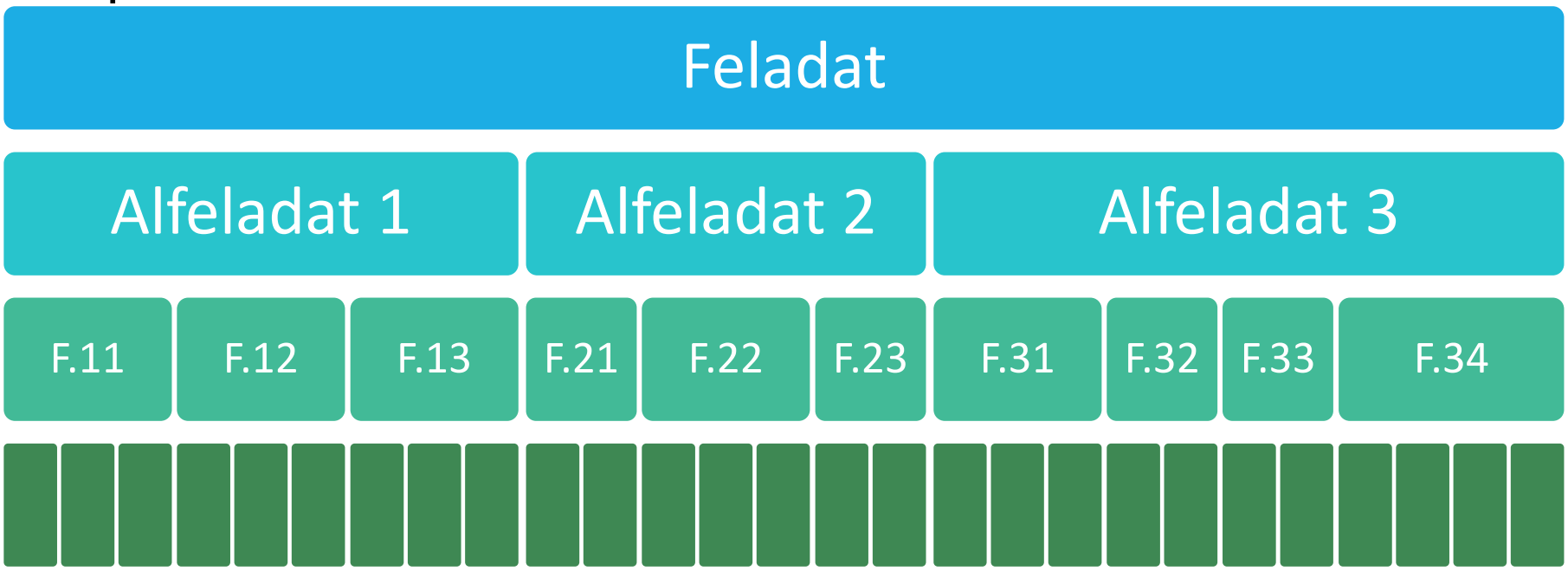
- A megközelítés hátránya: ha változik a specifikáció …



# Procedurális absztrakció

Mivel egy alprogram specifikáció a magasabb szintű specifikációtól függ, így a változás tovább gyűrűzik lefelé, és végül egy egész kis változásnak nagyon nagy hatása (és költsége) lehet

- Specifikáció változásának hatása



# Sablonok Kivételkezelés

KÖVETKEZŐ ALKALOMMAL




# Programozási nyelvek és módszerek

SABLONOK, KIVÉTELKEZELÉS

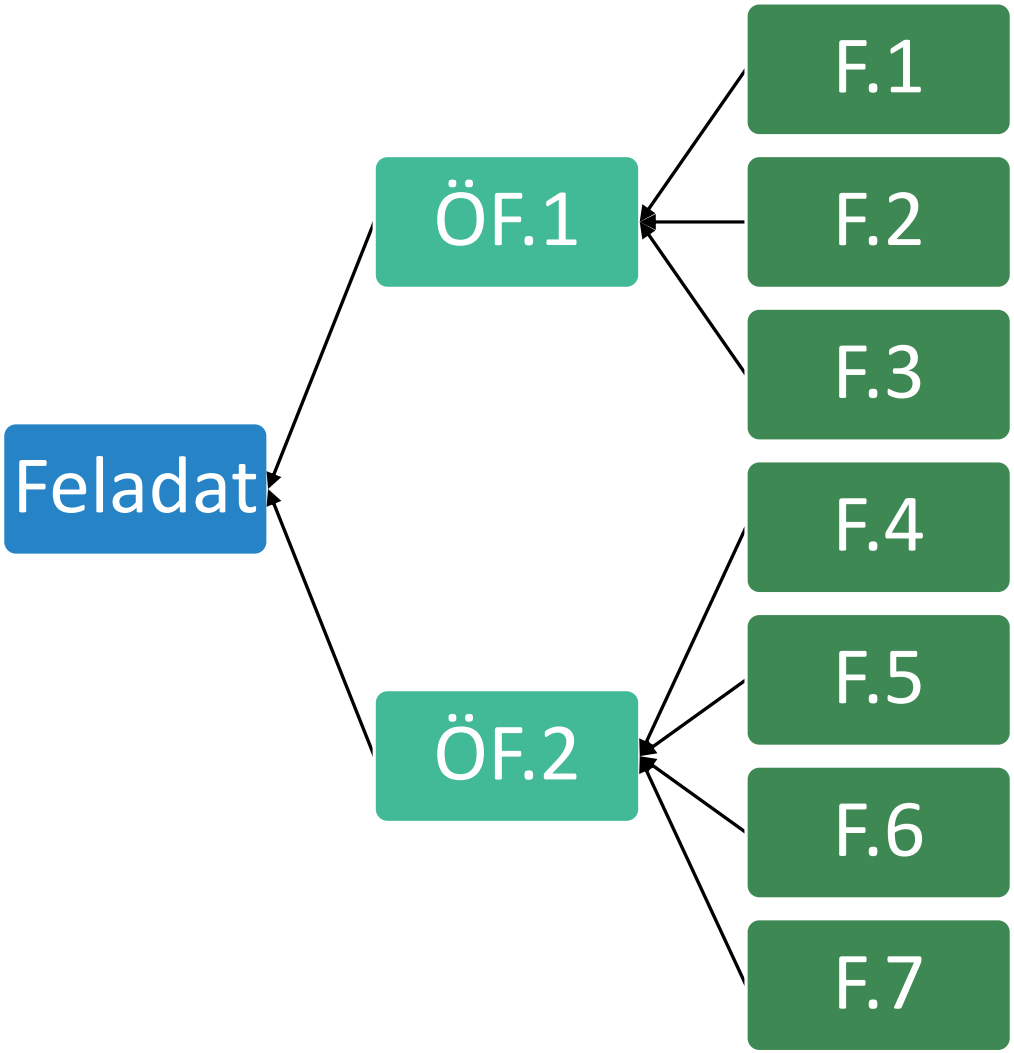
# Adatabsztrakció

# Adatabsztrakció

Ha elég magas szintről analizáljuk a problémát, akkor úgy tűnik, hogy

- a rendszerek jobban jellemezhetőek hosszú távon objektumaikkal, mint a rájuk alkalmazott tevékenységekkel.
- Ezt a megközelítést választva, először megpróbáljuk jellemezni az objektumokat, és az objektumok osztályait (~az adattípusokat) amelyek szerepet játszanak a rendszerben.
- Az új adattípusokat a már meglévők felhasználásával tervezzük – ez a bottom-up tervezés.
- Ez azt jelenti, hogy  az elérhető komponensek szintjéről indulunk, abból építkezünk.
- Ezután oldjuk meg a problémát.

# Alulról felfelé építkezés



# Típus a programozó szemszögéből

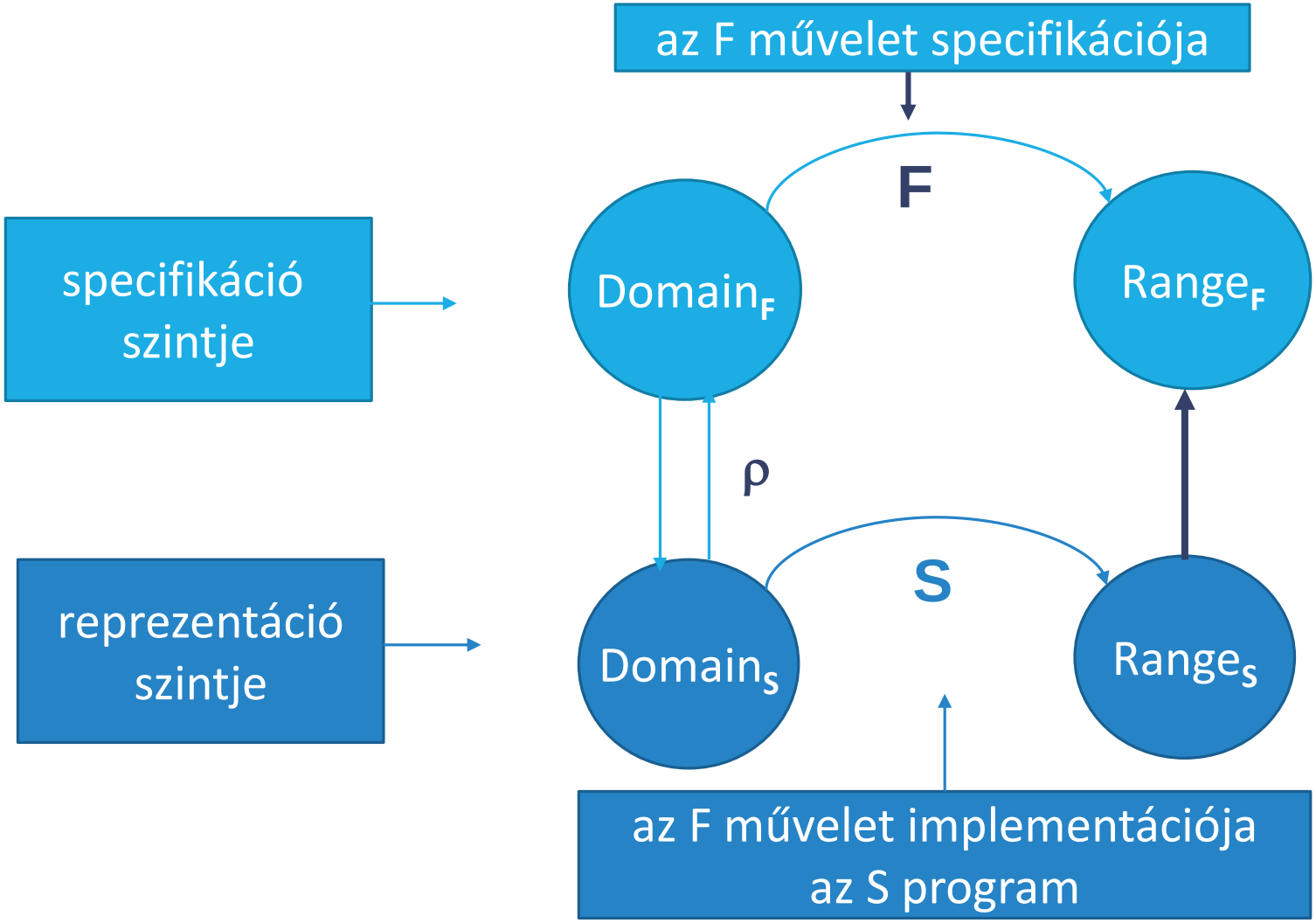
Típusspecifikáció („mit”)

- alaphalmaz: a valós világ minket érdeklő objektumainak halmaza
- specifikációs invariáns ($I_S$) - ezt a halmazt szűkíti
- típusműveletek specifikációja

Típus megvalósítás („hogyan”)

- Reprezentációs függvény
- Típus invariáns
- Típusműveletek megvalósítása

# A típus specifikáció és a típus kapcsolata



# Elvárások és eszközök

Elvárások a programozási nyelvekkel szemben

Absztrakt típusok megvalósításának eszközei

# Elvárások a programozási nyelvekkel szemben

## Modularitás

- Az egyes típusokat önálló fordítási egységekben lehessen megvalósítani!
- Ez biztosítja a típusok újrafelhasználhatóságát valamint a hatékony programfejlesztést
  - az egyes modulok könnyedén átvihetők más programokba,
  - a különböző egységeket más-más programozó is fejlesztheti
    - nem zavarva egymás munkáját

# Absztrakt típusok megvalósításának eszközei

## Modulokra bontás

- A professzionális használatra szánt programozási nyelvek mindegyikében megjelenik a modularizáció támogatása.
- A modularizáció alapját egyre inkább a típusokra bontás jelenti
  - azaz egy modul egy típust implementál
- De
  - sokszor van szükség „alprogram könyvtárak”-ra
  - szükség lehet csak implementációs célokat szolgáló típusok megvalósítására is
    - lehetőség szerint az azt használó adattípus moduljában rejtünk el

# Elvárások a programozási nyelvekkel szemben

## Adatrejtés

- A nyelv támogassa a reprezentáció elrejtését.
- Így a nyelv biztosítja, hogy az adott típus használója csak a specifikációban megadott tulajdonságokat használhassa ki.
- Lehetővé téve a reprezentáció, illetve az implementáció megváltoztatását, a változások a programban felfelé gyűrűzése nélkül.

# Absztrakt típusok megvalósításának eszközei

## A reprezentáció elrejtése

- Az adatrejtés támogatására az összetett típusok komponenseinek láthatóságát szintekre bontásával érhető el gyakran
  - Meghatározva, hogy pontosan kik férhetnek hozzá az adott komponenshez.
- Leggyakrabban három szint:
  - nyilvános (public) – az adott komponens mindenki számára látható
  - védett (protected) – az adott komponens csak a leszármazottak számára látható
  - privát (private) – a reprezentáció teljesen rejtett része, csak a műveletek implementációjában használható komponensek

# Absztrakt típusok megvalósításának eszközei

Adatrejtéshez új láthatósági szintek bevezetése

- Java
  - package-private
- Delphi
  - published (futás idejű típusinformáció)
  - strict private, strict protected
- C++
  - friend
- stb.

# Kérdések

Típus vagy objektum szinten szabályoz-e?
(C++ - SmallTalk)

Vannak olyan nyelvek ahol egyáltalán nincs ilyen jellegű szabályozás

- Python

Egyes nyelvekben a láthatóság még ennél is finomabban szabályozható

- a típus megvalósításakor rendelkezhetünk arról, hogy mely osztály (és leszármazottai) férhessenek hozzá az adott komponenshez.

```
class A
  feature {B, C}
     X: INTEGER;
  feature {ANY}
    make(p_name : STRING ) is
       require
            p_name /= ""
       do
            name := p_name ….
  feature {NONE}
     Y: ARRAY[INTEGER];
end
```



B és C látja (és utódaik)



public



private

# Elvárások a programozási nyelvekkel szemben

## Konzisztens használhatóság

- A felhasználói és a beépített típusok ne különbözzenek a használat szempontjából!
- A típusokat minél inkább a valós világban megszokott, „természetes” módon lehessen kezelni!
- Ugyanúgy lehessen összetett típusokat (tömböket, rekordokat stb.) definiálni a segítségükkel, ugyanúgy lehessen változókat definiálni a saját típusokkal stb.

# Absztrakt típusok megvalósításának eszközei

## Konzisztens használat

- Az új típust be lehessen illeszteni a nyelv logikájába!
  - Ha például az adott nyelvben az a konvenció, hogy egy típust a Read művelet olvas be, akkor fontos, hogy a saját típusokhoz is definiálhasson a programozó egy Read nevű műveletet
  - Azaz legyen lehetőség a Read azonosító túlterhelésére vagy átlapolására.
    - Egy azonosító túlterhelése vagy átlapolása (overloading) azt jelenti, hogy a programszöveg egy adott pontján az azonosítónak több definíciója is érvényben van.
  - Speciálisan az operátor-túlterhelésnek nagy jelentősége van abban, hogy a felhasználói típusokat természetes használatuknak megfelelően használhassuk.

# Elvárások a programozási nyelvekkel szemben

Generikus programsémák támogatása.

- A programozó lehetőleg minél általánosabban írhassa meg programjait!
- A nyelv adjon lehetőséget az ismétlések minimalizálására.
  - Közvetlen kódismétlések elkerülésére
  - Magasabb szintű megoldási struktúrák többszörös megírásának elkerülésére.
- Ez nagyban javítja a kód olvashatóságát és karbantarthatóságát.

# Elvárások a programozási nyelvekkel szemben

Specifikáció és implementáció szétválasztása külön fordítási egységbe

- Ekkor az adott típust használó más modulok a típus specifikáció birtokában elkészíthetők, függetlenül a tényleges implementációtól.

Modulfüggőség kezelése

- A fordító program kezelje maga a modulok közötti relációkat (egyik használja a másikat stb.).

# Absztrakt típusok megvalósításának eszközei

## Specifikáció és implementáció szétválasztása

- Azon nyelvekben, melyek támogatják az „egy modul egy típus” elvet, néha lehetőség van a típusspecifikációnak az implementációtól külön, önálló fordítási egységben történő leírására.
- Ez segíthet abban, hogy az egyes modulok egymástól függetlenül elkészíthetőek legyenek.

# C/C++

A specifikációt fizikailag be kell másolni minden olyan fordítási egységbe, amely az adott típust használni akarja

- Így nincs igazi szétválasztásról.
- A bemásolás megkönnyítésére a specifikáció egy külön, speciális forrásfájlban (header fájl) leírható
  - az előfordító segítségével azt beemelhetjük a megfelelő fordítási egységekbe.
- Ha egy típus specifikációját többször is beírjuk egy fordítási egységbe, az hibát jelent
  - ennek kivédéséről a header fájl megírásakor a programozónak gondoskodnia kell.

# C/C++

```
typedef double Angle;
class Complex {
public:
 Complex(double r=0, double i=0){ R = r; I = i;}
 Complex operator =(Complex z){ R = z.R; I = z.I; return *this; }
 Complex operator +(Complex z) { return  Complex(R+z.R,I+z.I);}
 Complex operator +(double x) { return Complex(R+x,I);}
 Complex operator *(Complex z);
 Complex operator *(double x);
 double Re();
 double Im();
 double Abs();
 Angle  Phi();
private:
   double R;
   double I;
}
```

# C/C++

```
Complex operator + (double x, Complex z) {
    return z+x
}
```

Vagy: **friend**

# C/C++

```
#include "complex.h"
Complex Exp(Complex z, double eps = 0.0001)
{
    Complex zi = Complex(1.0);
    Complex sum;
    double  i  = 1.0;

    while(zi.Abs() >= eps )
    {
        sum = sum+zi;
        i  += 1.0;
        zi  = (zi*z)/i;
    }
    return sum;
}
```

Más nyelvekben a fizikai szétválasztás nem lehetséges

- a fejlesztő eszköz támogatja a kód többszintű nézetét
- a felhasználás szempontjából lényegtelen részletek eltakarhatók.

A Java, C# stb. nyelvekben a specifikáció és implementáció összemosódik.

- Segítséget a dokumentációs lehetőségek jelentenek, amelyek a megfelelően dokumentált programkódból elő tudják állítani a specifikáció szöveges leírását.

# Modulfüggőség kezelése

- Egy program legmagasabb szintű építőkövei a modulok.
- A program működése ezen modulok interakciója.
- Minden modul igényel szolgáltatásokat és segítségükkel más szolgáltatásokat valósít meg, így a modulok között egyfajta függőségi reláció alakul ki, s egy modul megváltozása esetén szükségessé válhat a tőle függő modulok újrafordítása.

Némelyik programozási nyelv (például a C vagy C++) teljes egészében a programozóra bízza ezen függőségek kezelését, minden fordítási egységet önállóan kezel.

- Éppen ezért vannak speciális eszközök (pl. make), amelyek kizárólagos feladata ennek a feladatnak a megkönnyítése.
- Ezek a  megoldások nem tökéletesek, előfordul, hogy túl sokszor fordítanak újra valamit, vagy éppen nem veszik észre az újrafordítás szükségességét stb.

Más nyelvekben a függőségek kezelése a fordító program feladata.

- Az Ada nyelvben például a with utasítással kell megadnunk, hogy milyen más fordítási egységektől függ a szóban forgó modul.
- Ez garantálja azt, hogy a modul fordítása előtt mindazon modulok specifikációs része lefordításra kerül (ha az szükséges), amelyektől az adott modul függ.

# Kérdések

Saját adattípus önálló fordítási egységként megvalósítható?

Reprezentáció elrejtése megvalósítható?

Milyen láthatósági szintek vannak?

Beépített típusokkal megegyező módon használható?

Van-e azonosító túlterhelés/ operátor túlterhelés/ free operátor?

Specifikáció és implementáció különválasztható?

# Sablonok

# Sablonok – bevezetés

Újrafelhasználható, könnyen karbantartható programokra van szükség.

- Ennek a jegyében készít a programozó minél általánosabb típusokat és alprogramokat, hogy azokat mind az éppen aktuális, mind egyéb programjaiban a lehető legszélesebb körben használhassa.
- Maga a programozási módszertan is olyan programozási megoldásokat tanít, amelyek általánosan használhatóak.
- Ezt a törekvést a programozási nyelvek is támogatják kisebb-nagyobb mértékben
  - Most ezeket az eszközöket gyűjtjük össze.
  - Az eszközök némelyike olyan megszokott és hétköznapi, hogy talán nem is vesszük észre, hogy ebbe a kategóriába tartoznak...

# Sablonok – bevezetés

Alprogramok

- Alprogramok paraméterezése

Típusok paraméterezése

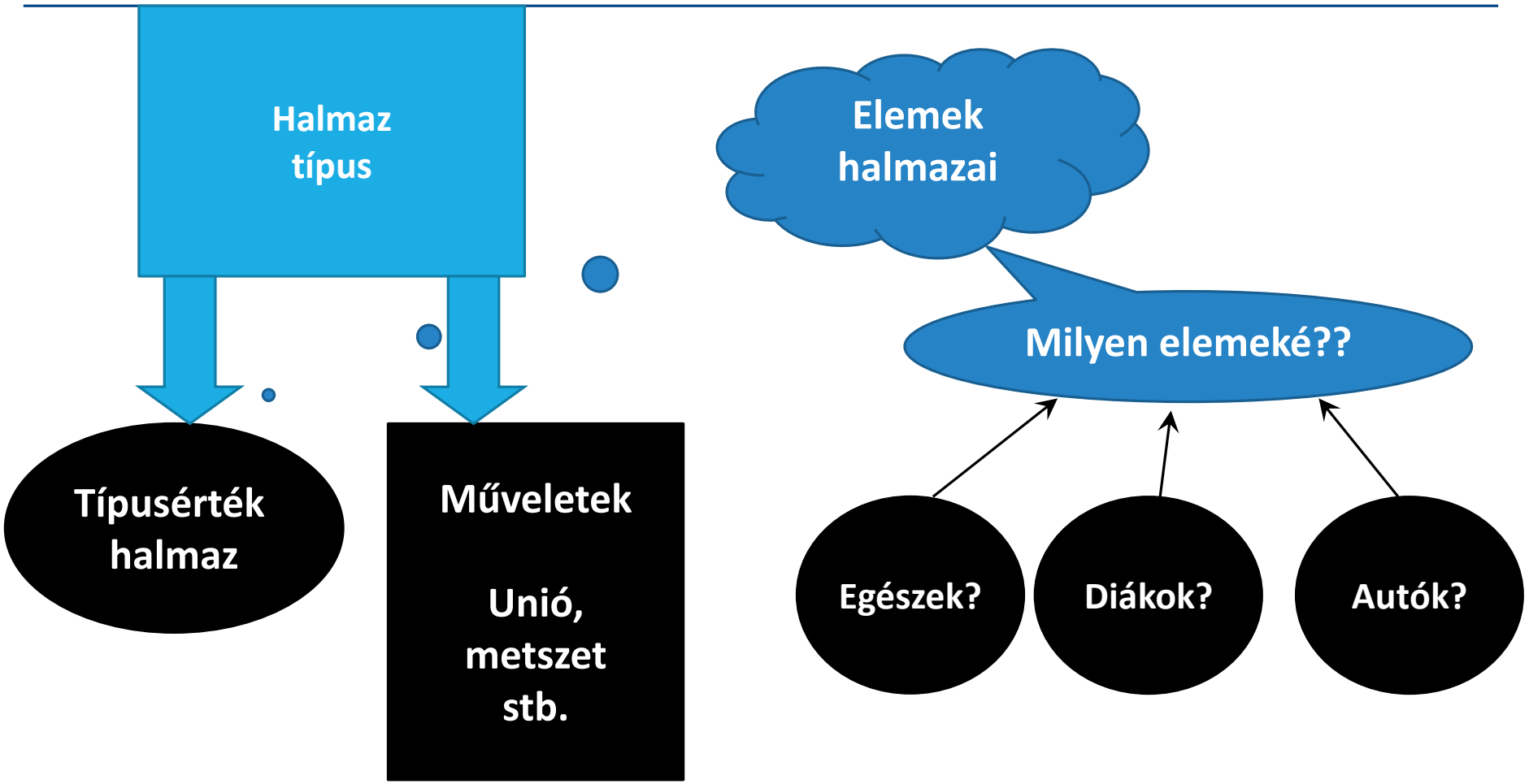
- Például az Ada-ban a diszkriminánsos rekordok, vagy a határozatlan indexhatárú tömb típusok

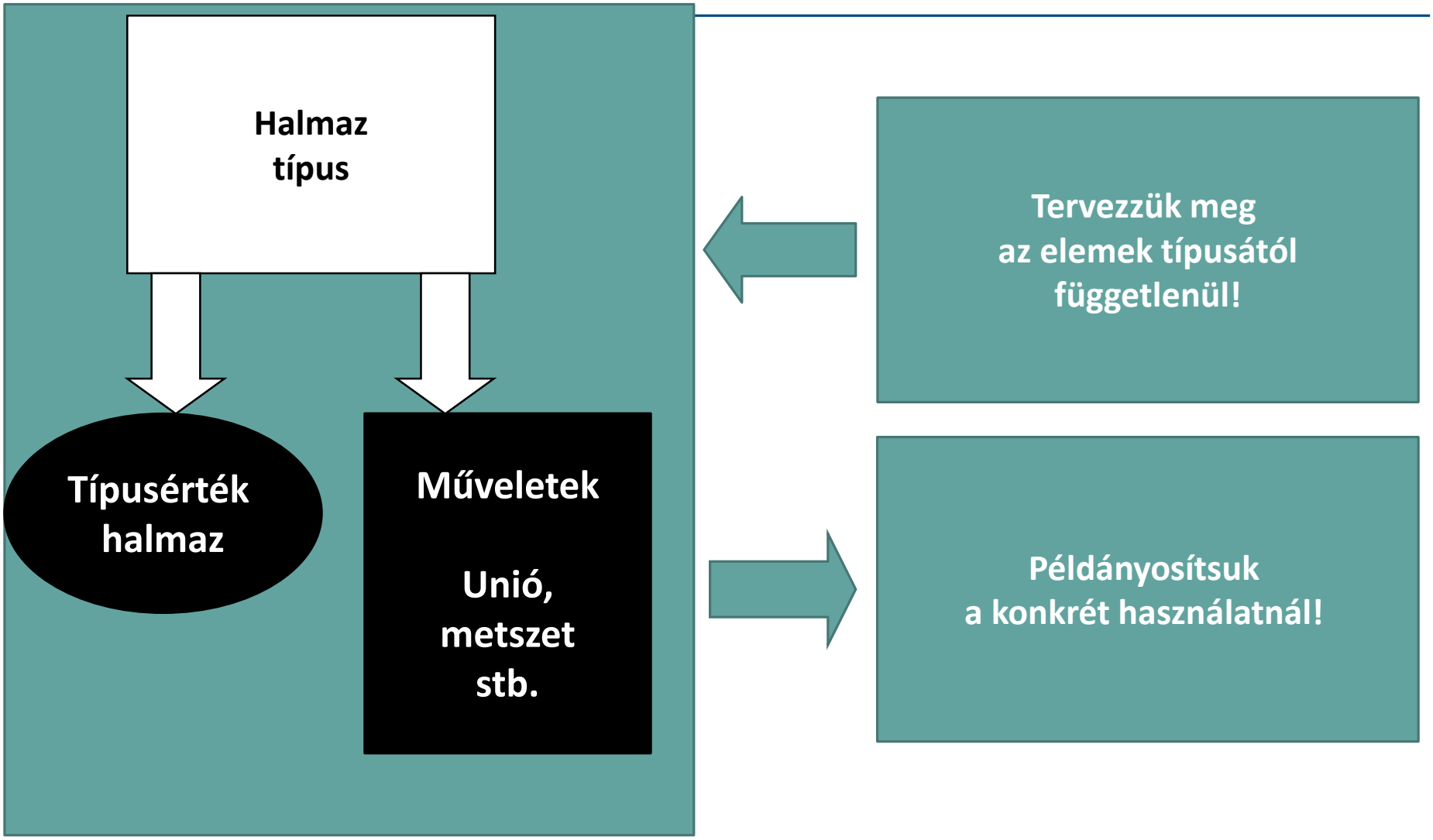
Alprogrammal történő paraméterezés

# Sablonok – cél

Típusokkal történő paraméterezés

- Programozási tételeink, alapvető adatszerkezeteink és adattípusaink is „absztrakt” fogalmakat használnak
  - a számukra feltétlenül szükséges megszorításokat teszik ezekre a fogalmakra.
- Ennek következménye, hogy általában nem egyetlen típus elégíti ki ezen megszorításokat, hanem sok.
- Szeretnénk a fenti tételek, adatszerkezetek, típusok implementációját csak egyszer megadni és azt a leírást általánosan az összes, a feltételeket kielégítő típussal használni.


Halmaz típus
Típusérték halmaz
Műveletek
Unió, metszet stb.
Elemek halmazai
Milyen elemeké??
Egészek?
Diákok?
Autók?


Halmaz
típus
Típusérték
halmaz
Műveletek
Unió,
metszet
stb.
Tervezzük meg
az elemek típusától
függetlenül!
Példányosítsuk
a konkrét használatnál!

# Példák

Keresés vektorban lineárisan

Maximális elem keresése vektorban

Vektor elemeinek rendezése

Verem, Sor

Prioritásos sor

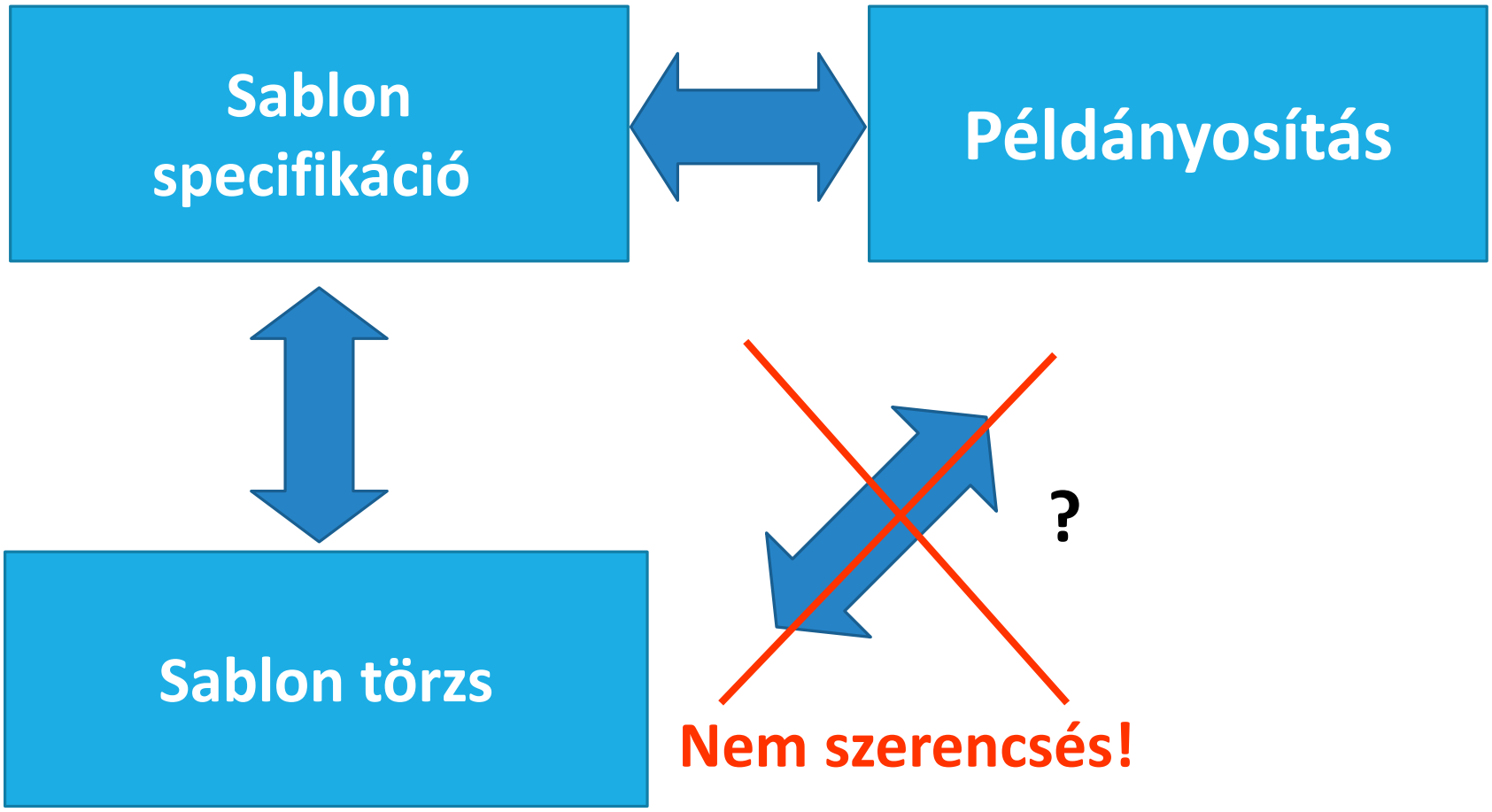
Fa

Keresőfa

Hash-tábla, stb.

Mi a különbség a példák között?

# Példányosítás – összefüggések



Ezen igény kielégítésére többféle eszköz kínálkozik.

- A C nyelvben például nem áll rendelkezésre ilyen eszköz nyelvi szinten
  - hasonló hatások – korlátozott mértékben – elérhetők az előfeldolgozó rendszer használatával.
- Az előfeldolgozó rendszer szerves része a C nyelv specifikációjának
  - Nem tekinthető igazán nyelvi eszköznek, mivel a forrásnyelvű programot manipulálja
  - Közelebb áll a megíráshoz használt szövegszerkesztőhöz.

```
typedef struct cell (ELEMENT data; struct cell *next) CELL;
typedef CELL *LIST;

int find_item(ELEMENT find, LIST head, LIST *scan, LIST *prior)
{
        scan=head;
        prior=NULL;

        while (GREATHAN(find, (*scan)->data)

        {

                *prior=*scan;

                *scan=(*scan)->next;

        };

        return EQUALS(find, (*scan)->data);

}
```

Ez a megoldás független az aktuális ELEMENT típustól

El kell készíteni a megfelelő makrókat, és működik

```
/* alfabetikus eset */
#ifdef ALPHA
#define ELEMENT char*
#define LESSTHAN(x,y) (strcmp((x),(y))>0)
#define GREATHAN(x,y) (strcmp((x),(y))<0)
#define EQUALS(x,y) (strcmp((x),(y))==0)
#define ASSIGN(x,y) (strcpy((x),(y)))
#define MAXVALUE '\377'

/* short integer használata*/
#elif SHORT
#define ELEMENT short int
#define LESSTHAN(x,y) ((x)<(y))
#define GREATHAN(x,y) ((x)>(y))
#define EQUALS(x,y) ((x)==(y))
#define ASSIGN(x,y) ((x)=(y))
#define MAXVALUE 0x7fff
```

# CLU

```
generic_proc_name = proc [ t : type ] (...)
                                returns (...) signals (...)
        where t has op1 : proctype (...) returns (...)
        where t has op2 : proctype (...) returns (...)
          ...
        where t has opk : proctype (...) returns (...)

sort = proc[ t : type ] ( A : array[ t ] )
                                returns ( array[ t ] )
   where t has lt : proctype (t, t) returns (bool)
```

# OOP megoldás

Objektumorientált programozási nyelvekben a hatás elvileg elérhető kizárólag az öröklődés és a polimorfizmus használatával

- A szükséges közös tulajdonságokat össze lehetne fogni egyetlen absztrakt osztályba – vagy interfészbe –
- Ez közös őse lehetne az összes, a feltételeknek eleget tevő osztálynak.

A megoldás problémája, hogy

- a nyelv alapvető osztályhierarchiájának kialakításakor ismerni kellene az összes számításba jövő feltételkombinációt

# Típussal paraméterezés

A típussal paraméterezést támogató nyelvek általában valamilyen önálló konstrukciót vezetnek be.

Ezekben a struktúrákban megadhatunk formális paraméterként típusokat, amelyeket aztán más típusokhoz hasonlóan használhatunk a struktúra belsejében.

Amikor a formális paramétereknek aktuális értéket adva létrehozzuk a struktúra megadott típusokhoz tartozó változatát, akkor a struktúra példányosításáról beszélünk.

# C++

```
#define MAX(a,b) a > b ? a : b
MAX( x, y )*2 --> x > y ? x : y*2
#define MAX(a,b) ((a) > (b) ? (a) : (b))
MAX( ++x, y ) --> ++x > y ? ++x : y

void swap( double& x, double& y ) {
  double temp = x;
        // !! Hogy határozzuk meg x típusát?
  x = y;
  y = temp;
}
```

# C++

A C++ nyelvben a típussal paraméterezés lehetőségét a template-ek biztosítják.

- Több, mint az előző makróhelyettesítés, ahol a fordító a megadott aktuális típussal helyettesíti a hozzá tartozó formális paraméter minden előfordulását.
- Bizonyos minimális szintaxisellenőrzést végez a fordító, de a formális paraméternél csak annyit tudok meghatározni, hogy az adott paraméter egy típust jelöl.

# C++

Nem tudom előírni a megvalósítandó alprogram vagy típus számára fontos tulajdonságok meglétét

- Bizonyos, szükséges műveletek
- Az ebből származó hibák így a példányosításkor jelentkeznek.
- Előnye a nyelvnek, hogy a template példányosítása teljesen automatikusan történik.

# C++

```
template <typename T>
void swap( T& x, T& y) {
   T temp = x;
   x = y;
   y = temp;
}
```

# C++

```
template <class T>
T max(T x, T y){
     return (x>y) ? x : y;
}
```

Ennek a használata:

```
int i;
Own_typ a,b;
int j=max(0,i);       // T –t helyettesíti "int"-tel
Own_typ m=max(a,b); // Own_typ -pal
```

# C++

```
double x, y = 5.1, z = 3.14;
x = max(y, z);
int i = 3, j = 4, k;
double x = 3.14, y = 4.14, z;
const int ci = 6;
k = max(i, j);  // max(int, int)
z = max(x, y);  // max(double, double)
k = max(i, ci); // max(int, int) - triviális
z = max(i, x);  // Nem egyértelmű!
```

# C++

```
template <class T, class S>
T max( T a, S b) {
  if ( a > b ) return a;
  else return b;
}
int i = 3;
double x = 3.14;
double z = max( i, x);   // Ez ok, megtörténik
z == 3.0;              // A konverziók miatt a 3.14
                       //  az 3.0 lesz
```

# Explicit specializáció

```
template <class R, class T, class S>
R max(T a, S b) {
  if ( a > b ) return a;
  else return b;
}
z = max<double>(i, x);   // ok, 3.14
k = max<long, long, int>(i, x);
                // Konvertál long és int -re
k = max<int, int, int>(i, j);
                // Fölösleges
```

# C++

```
#include <iostream.h>
template <class T> class Vector {
     T* data;
     int size;
public:
     Vector(int);
     ~Vector() { delete[] data; };
     T& operator[](int i) { return data[i]; }
};
template <class T> Vector<T>::Vector(int n) {
     data=new T[n];
     size=n;
};
```

# C++

```
main(){
   Vector <int> x(5);
   for (int i=0; i<5;++i) {
       x[i]=i;
   }
   for (i=0; i<5;++i) {
      cout  << x[i] << ' ';
   }
   cout << '\n';
}
```

# C++ concept

A concept-ek lehetősége

- megfogalmazhatjuk elvárásainkat egy típussal szemben
- ezt még példányosítás előtt tud ellenőrizni a fordítóprogram
- meg tudja nevezni a hiba okát olvasható formátumban

C++20-ban bevezetésre került!

# C++ concept

```
template<typename T>
concept EqualityComparable =
  requires(T a, T b) {
    { a == b } -> bool;
    { a != b } -> bool;
};
```

Megjegyzés

- draft cpp szabvány esetén (8.x gcc-vel pl) a szintaxis eltérő lehet több esetben
- Itt: `concept bool EqualityComparable =`

# C++ concept

```
template<EqualitiyComparable T>
void equals(T x, T y) {
    if (x==y)
        std::cout << "Azonosak";
    else
        std::cout << "Nem azonosak";
}

equals<int>(42, 42); // "Azonosak"

equals<int>(42, 10); // "Nem azonosak"
```

# C++ concept

```
class ownClass {
public:
    int member;
    ownClass(int init): member(init) {};
};

ownClass o1(10), o2(20);
equals<ownClass>(o1, o2); // HIBA!!!
```

No matching function for call to 'equals' candidate template ignored: constraints not satisfied [with T = ownClass] because 'ownClass' does not satisfy 'EqualitiyComparable' because 'a == b' would be invalid: invalid operands to binary expression ('ownClass' and 'ownClass')

# C++ concept

```
class ownClass {
public:
    int member;
    ownClass(int init): member(init) {};

    bool operator == (ownClass const &other) {
        return member == other.member;
    }

    bool operator != (ownClass const &other) {
        return member != other.member;
    }
};
```

# C++ concept

További megadási lehetőség

```
template<typename T> requires EqualitiyComparable<T>
void equals(T x, T y) …
```

Sőt, akár

```
template<typename T> requires !EqualitiyComparable<T>
```

Követelmények együttes megadása is lehetséges

```
template<typename T>
requires floating_point<T> && integral<T>;


template<typename T>
requires floating_point<T> || integral<T>;
```

# C++ concept

Előzőekhez az összetevők:

```
template < class T >
concept integral = std::is_integral_v<T>;

template < class T >
concept floating_point = std::is_floating_point_v<T>;
```

Egymásba ágyazással:

```
template<typename T>
concept MyConcept = requires(T a) {
    requires floating_point<T>;
    requires integral<T>;
};
```

# Eiffel: generic

Objektum orientált megközelítés.

A formális paraméterei osztályok lehetnek ahol a szükséges speciális tulajdonságokat azáltal lehet meghatározni, hogy megadható, mely osztályból kell származnia az aktuális paraméternek.

- Előnye: A generic belsejében használt műveletek a megadott ősosztály műveletei lehetnek.
  - A használat helyessége a generic megírásakor ellenőrizhető.
- Hátránya:  a szükséges közös tulajdonságokat jóval előre tudni kell, hogy a megfelelő közös ős definiálható legyen.
  - Ekkor viszont a problémák jórészt megoldhatóak az öröklődés és a polimorfizmus eszközeivel.

# Eiffel

Megszorítás nélküli generic

```
deferred class TREE [G] ...
class LINKED_LIST [G] ...
class ARRAY [G] ...
```

Az osztályoknak akárhány formális generic paramétere lehet.

Típus létrehozásához egy generic osztályból:

- Minden formális generic paraméterhez kell egy típus, az aktuális generic paraméter.
- Ez egy generikusan származtatott (példányosított) típust eredményez.

# Eiffel

Generic származtatások - példányosítás

```
TREE [INTEGER]
TREE [PARAGRAPH]
LINKED_LIST [PARAGRAPH]
TREE [TREE [TREE [PARAGRAPH]]]
ARRAY [LINKED LIST [TREE [LINKED LIST
[PARAGRAPH]]]
```

# Eiffel

## Korlátozott generic

- `class HASH TABLE [G, KEY -> HASHABLE] ...`
  - KEY –t megszorítjuk a HASHABLE osztállyal.
  - Minden T aktuális generic paraméter bázisosztálya, amit a KEY-hez használunk a HASHABLE megszorító osztály leszármazottja kell legyen.
- `HASH TABLE [PARAGRAPH, STRING]`
- A Korlátozott formális generic paraméterek általános szintaxisa:
  - `T -> Class-type`

# Ada

A sablonok paraméterezhetősége sokkal szélesebb körű és rugalmasabb

- Lehet:
  - megadott őstípusból származó típussal paraméterezni,
  - kiköthető, hogy az aktuális paraméter valamilyen típusosztályba (diszkrét típus, felsorolási típus, tetszőleges típus stb.) tartozzon.
  - megadhatók a használni kívánt további műveletek, amelyek segítségével kikerülhető a kötelező közös ős megléte.
  - az explicit megadott műveletek segítségével a generic még rugalmasabb használatára nyílik lehetőség, megtehető például, hogy egy rendezési relációt használó sablont az egész számokkal példányosítva nem a szokásos rendezési relációt adjuk meg, hanem például az oszthatóság parciális rendezési relációját.

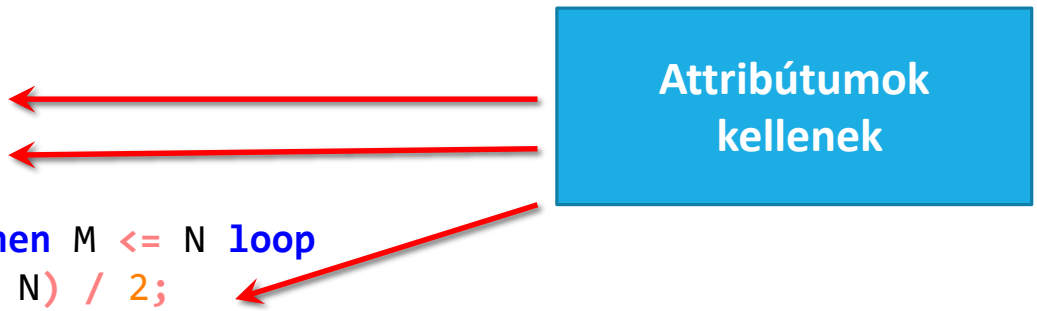
# Ada

```
generic
      type Item is private;
      type Index is (<>);
      type Vector is array (Index range <>) of Item;
      with function "<"(X, Y : Item) return Boolean is <> ;

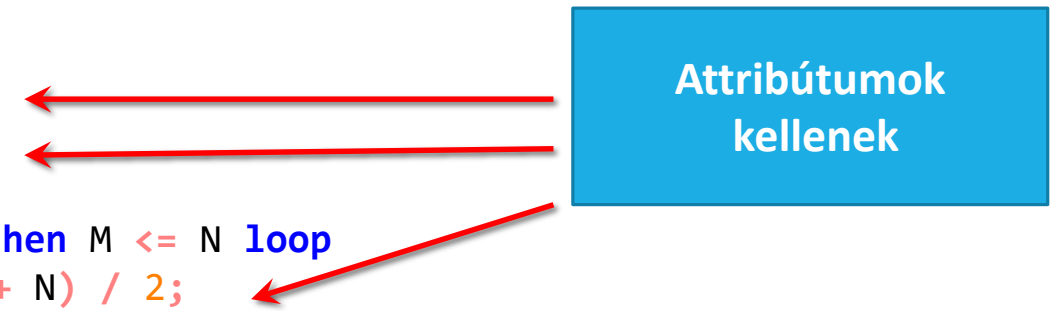
procedure Log_Search(V: in Vector; X: in Item;
                     Found: out Boolean; Ind: out Index);
```

```
procedure Log_Search(V:in Vector; X:in Item Found: out Boolean; Ind:out Index) is
    M, N, K : Integer;
    begin
        M := Index'Pos(V'First);
        N := Index'Pos(V'Last);
        Found := False;
            while not  Found and then M <= N loop
                        K := (M + N) / 2;
                        if  X < V(Index'Val(K)) then
                                N := K - 1;
                        elsif X = V(Index'Val(K)) then
                                Ind := Index'Val(K);
                                Found := True;
                        else
                                M := K + 1;
                        end if;
                end loop;
    end Log_Search;
```

```
procedure Log_Search(V:in Vector; X:in Item Found: out Boolean; Ind:out Index) is
    M, N, K : Integer;
    begin
        M := Index'Pos(V'First);
        N := Index'Pos(V'Last);
        Found := False;
            while not  Found and then M <= N loop
                        K := (M + N) / 2;
                        if  X < V(Index'Val(K)) then
                                N := K - 1;
                        elsif X = V(Index'Val(K)) then
                                Ind := Index'Val(K);
                                Found := True;
                        else
                                M := K + 1;
                        end if;
                end loop;
    end Log_Search;
```



```
procedure Log_Search(V:in Vector; X:in Item Found: out Boolean; Ind:out Index) is
    M, N, K : Integer;
    begin
        M := Index'Pos(V'First);
        N := Index'Pos(V'Last);
        Found := False;
            while not  Found and then M <= N loop
                        K := (M + N) / 2;
                        if  X < V(Index'Val(K)) then
                                N := K - 1;
                        elsif X = V(Index'Val(K)) then
                                Ind := Index'Val(K);
                                Found := True;
                        else
                                M := K + 1;
                        end if;
                end loop;
    end Log_Search;
```



# Java

## Generic bevezetése

```
List<Integer> myIntList =
                              new LinkedList<Integer>();
myIntList.add(new Integer(0));
Integer x = myIntList.iterator().next();
```

# Java

Ehhez a java.util-ban példa

```
public interface List <E> extends Collection<E>
{
    void add(E x);
    Iterator<E> iterator();
}

public interface Iterator<E>{
    E next();
    boolean hasNext();
}
```

# Java

```
public class Box<T> {
     private T t;                          // T: Típus
     public void add(T t) { this.t = t; }
     public T get() { return t; }
}
Box<Integer> integerBox;
integerBox = new Box<Integer>();
```

Java 7-től:

```
Box<Integer> integerBox = new Box<>();
```

- is engedélyezett, ha a fordító ki tudja következtetni

# Java

A használata:

```
public class BoxDemo3 {
  public static void main(String[] args) {
    Box<Integer> integerBox = new Box<Integer>();
    integerBox.add(new Integer(10));
    Integer someInteger = integerBox.get();
    System.out.println(someInteger);
  }
}
```

# Java

generic metódusok lehetősége is megvan:

```
public class Box<T> {
    private T t;
    public void add(T t) { this.t = t; }
    public T get() { return t; }
    public <U> void inspect(U u){
        System.out.println("T: " + t.getClass().getName());
        System.out.println("U: " + u.getClass().getName());
    }

    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();
        integerBox.add(new Integer(10));
        integerBox.inspect("some text");
    }
}
```

# Java

Az eredmény:

- `T: java.lang.Integer`
- `U: java.lang.String`

# Java

Típustörlés – nyers (raw) típusok

- `Box rawBox = new Box();`
  - ez is lehetséges – ezzel a generic előtti viselkedést kapjuk…
- `Box` a nyers típusa a `Box<T>`-nek

A korábbi programok így gond nélkül működhetnek

Futási időben minden generic `Object`-tel paraméterezett lesz

- Az ellenőrzés fordításkor történik

# Java

Megszorított típusparaméter lehetősége:

- `<U extends Valami>`
  és akkor az aktuális típusparaméter U-ra csak a Valami leszármazottja lehet
- vagy, ha interfész megvalósítását is kérjük:
  - `<U extends Valami & MyInterface>`
- A  generic törzsében számíthatunk a Valami és a MyInterface műveleteire!
- Mindig először kell az osztály, azután az interfészek

Öröklődéssel való kapcsolata később, az OOP-nél!

# Java

Példa:

```
public class Gen1<A extends Number> {
  public Gen1(A aa) { a = aa; }
  public A getElem() { return a; }
  public void setElem(A aa) { a=aa; }
  public int egesz() { return a.intValue(); }
  protected A a;
}
Gen1<Double> g2 = new Gen1 <Double>(2.3);
g2.egesz();
```

# Java – Típushelyettesítő

Tételezzük fel, hogy

```
public class Gen<T> {
  public Gen(T st) {storage = st; }
  public T getElem() { return storage; }
  public void setElem(T st) { storage = st; }
  protected T storage;
}
```

És készítünk egy f fv-t:

```
static <A,B> Gen f(Gen<A> ga, Gen<B> gb) {
  Object o1 = ga.getElem();
  Object o2 = gb.getElem();
  if (o1.toString().length() < o2.toString().length() )
     return ga;
  else
     return gb;
}
```

Mi legyen a visszatérési típusa?

# Java – Típushelyettesítő

## Probléma:

```
Gen<> g = f(new Gen<Integer>(1), new Gen<Double>(2.0) );
```

- ez fordítási hibás!

```
Gen g = f( new Gen<Integer>(1), new Gen<Double>(2.0) );
```

- Ok, de ez „nyers” (raw) típus…

```
Gen<Number> gi = f(new Gen<Integer>(1),
                                        new Gen<Double>(2.0) );
```

- ez rendben, de warning

## Típushelyettesítő

```
Gen<?> g1 = f( new Gen<Integer>(1), new Gen<Double>(2.0));
```

# Java – Típushelyettesítő

Még jobb, ha már a definíciójába tesszük:

```
static Gen<? extends Number> max(
                                Gen<? extends Number> ga,
                                Gen<? extends Number> gb) {
  Number n1 = ga.getValue();
  Number n2 = gb.getValue();
  if ( n1.doubleValue() < n2.doubleValue() )
    return ga;
  else
    return gb;
}
```

# Java további lehetőségek

Nemcsak felső megszorítás adható

Vegyük az alábbi függvényt

```
public void append(
            Collection<Integer> integers, int n) {
    for (int i = 1; i <= n; i++) {
        integers.add(i);
    }
}
```

A következő listát hogyan tudjuk paraméterként átadni?

```
List<Number> numbers = new ArrayList<Number>();
```

# Java további lehetőségek

A felső megszorítás értelemszerűen ekkor nem jó.

```
public void append(
    Collection<? super Integer> integers, int n) {
    for (int i = 1; i <= n; i++) {
        integers.add(i);
    }
}
```

```
List<Number> numbers = new ArrayList<Number>();
append(numbers, 5);
```

# Java generic és a new

A kapott típus szerinti objektumok létrehozása – probléma

```
class Dummy<T> {
    public T getInstance() throws Exception {
        new T();
        T.newInstance();
        T.class.newInstance();
    }
}
```

Egyik sem lehetséges

- Az megfejtés abban rejlik, hogy a típustörlést követően Object került a T helyére és nem az aktuális típusparaméter
  - Tehát a formális ellenőrzésen átmegy, de a törlést követően más a viselkedése

# Java generic és a new

Generic tömbök létrehozása – nem lehetséges

```
T[] ts = new T[n]
```

Ennek a megfejtése más

- A Java tömbök kovariánsak (erről még lesz szó). Lehetséges:

```
Integer[] myInts = {1,2,3,4};
Number[] myNumber = myInts;
```

  - A tömbökhöz futásidőben elérhető típusinformáció tartozik, a fenti esetben a `myNumber` változó mögötti tömbbe nem lehetséges a `4.5` érték berakása. Ezt futásidőben ellenőrzi és megakadályozza a VM. (Ez rendben is van, a tömbök definícióját teljesíti.)
  - Generikus tömb esetén azonban a típusinformáció a típustörlés miatt nem áll rendelkezésre, azaz `Object` lesz minden. Így a fenti típusellenőrzés nem lehetséges futásidőben, ami baj.

# C#

Sok szempontból hasonlít a Java megoldásaira

- Szintaktika
- Megszorítások

Különbség azonban, hogy C# esetén nincsen típustörlés

- E tekintetben a C++ template-hez hasonló a működés
- Minden típusparaméter-értékhez „natív” kódot generál

# C#

```
public class LinkedList<TKey, TData> where TKey : IComparable
{
    …
        public bool Find(TKey key)   {
            Node<TKey,TData> current = _head;
            while (current.NextNode != null)
            {
                if (current.Key.CompareTo(key) != 0)
                    current = current.NextNode;
                else
                    break;
            }
            return (current.Key.CompareTo(key) == 0);
        }
}
```

# Kérdések:

Milyen nyelvi elemekből tudunk sablonokat létrehozni?

- Új adattípusokból és/vagy alprogramokból?

Milyen paraméterei lehetnek ezeknek a mintáknak?

- Típusok, objektumok, alprogramok?
- Létrehozható paraméter nélküli sablon?

Milyenfajta típusok lehetnek aktuális típus-paraméterek?

- Csak beépített típusok és altípusaik, vagy felhasználó által definiált típusok is?

# Kérdések

Adhatunk megszorításokat a formális generic paramétereknek?

Egymásba ágyazható ez a konstrukció?

A példányosítás fordítási (statikusan) vagy futási időben (dinamikusan) történik?

# Kivételkezelés

# Program készítésének célja

A célunk jó minőségű program készítése

Alapvető elvárás

- helyes – a specifikációnak  megfelelően működik
- robosztus – nem várt helyzetekben sem történik katasztrófa

# Hibalehetőségek

A programok futása közben hibák léphetnek fel.

Hardverhibák

- elromlott winchester miatti leállások,
- elfogy a memória, stb.

# Hibalehetőségek

A programok futása közben hibák léphetnek fel.

Szoftverhibák

- a futtatott programban,
  - kísérlet nullával való osztásra
  - aritmetikai túlcsordulás, alulcsordulás
  - üres pointer/referencia dereferencia kísérlete
  - tömb hibás indexelése
  - hibás típus konverzió kísérlete
  - hibás input stb.
- az operációs rendszerben meglévő programozói hibák.

A program

megbízhatóságának növelése

érdekében

figyelni kell a hibákra

és meg kell próbálni

megelőzni,

vagy ha már bekövetkeztek,

kezelni

őket!

# Hibakezelés históriája

Hogyan kezelték régebben?

- Abortált a program
- Minden alprogram valamilyen jelzést (hibaértéket) ad vissza
  - A lefutás OK
  - A lefutás nem sikerült

# Példa

Tegyük fel, hogy szeretnénk beolvasni egy fájlból

```
ReadFile(f: in File; n1: out byte, n2: out integer,
                     n3: out longint, n4: out real);
begin
    ReadByte(f,n1);
    ReadInt(f,n2);
    ReadLongInt(f,n3);
    ReadReal(f,n4);
end ReadFile;
```

# Ellenőrzés lépésenként

## Ha ellenőrizni akarjuk:

```
ReadFile(f: in File; n1: out byte, n2: out integer,
                     n3: out longint, n4: out real): boolean;
success:boolean;
begin

  success:=ReadByte(f,n1);
    if success then success:=ReadInt(f,n2);
      if success then success:=ReadLongInt(f,n3);
         if success then success:=ReadReal(f,n4);

            if not success then
              // ReadReal hiba kezelése,
            end if;

         else … ReadLongInt hiba kezelése end if ;
      else … ReadInt hiba kezelése end if ;
    else … ReadByte hiba kezelése end if ;

  return success;

end ReadFile;
```

# Meggondolások

Nem biztos, hogy függvényt lehet/érdemes csinálni az eljárásból

- Lehet, hogy nem visszatérési érték, hanem egy paraméter jelzi a sikert.

A kód sokkal bonyolultabb, az eredeti cél szinte elvész

- A későbbi karbantartás sokkal nehezebb!

Nem biztos, hogy a hívó figyeli a sikert jelző értéket!

# Meggondolások

Lehetne, hogy globális változót használunk a lefutás helyességének figyelésére

- Osztott környezetben beláthatatlan következmények!

Hiba esetén meghívunk egy hibakezelő alprogramot

- Mi történik, ha nem tesszük meg?

# Elvárások

A hibák kezelése kapcsán a következő elvárásokat lehet megfogalmazni

1. meg tudjuk különböztetni a hibákat,
2. a hibát kezelő kód különüljön el a tényleges kódtól,
3. megfelelően tudjuk kezelni - a hiba könnyen jusson el arra a helyre, ahol azt kezelni kell,
4. kötelező legyen kezelni a hibákat

**KIVÉTELKEZELÉS**

# Megoldás elemzése

Az első kikötés, hogy meg tudjuk különböztetni a hibákat.

- Ez általában a fellépés helyén történik.

A második kikötés azt szeretné elérni, hogy a programot először látó ember is azonnal el tudja különíteni a hibakezelő és a működésért felelős részeket.

- Ez könnyebben továbbfejleszthető, karbantartható programokhoz vezet.

# Példa

```
gyumolcs {
    alma();
    //...
    korte();
    //...
    barack();
}
```

# Példa hagyományos módon

```
gyumolcs {
   ret1=alma();
   if (ret1==ok) {
      //...
      ret2=korte();
      if (ret2==ok) {
         //...
         ret3=barack();
         if (ret3==ok) {
            return ok;
         }
         else { /* 3. hiba kezelése */}
      }
      else { /* 2. hiba kezelése */ }
   }
   else {/* 1. hiba kezelése */ }
}
```

# A példa kivételkezeléssel

```
class AlmaException extends Exception {}
class KorteException extends Exception {}
class BarackException extends Exception {}


gyumolcs {
   try {
      alma(); /*küldhet AlmaException-t*/
      korte();
      barack();
   }
   catch(AlmaException hiba1) { /* 1. hiba kezelése */ }
   catch(KorteException hiba2) { /* 2. hiba kezelése */ }
   catch(BarackException hiba3) { /* 3. hiba kezelése */ }
}
```

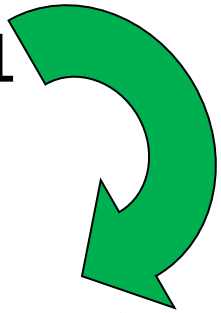
# A hiba jusson el oda, ahol kezelni kell

A harmadik kikötés arra vonatkozik, hogy egy alacsonyabb szinten jelentkező hibát nem biztos, hogy az adott szint le tud kezelni, ezért a megfelelő helyre kell eljuttatni.

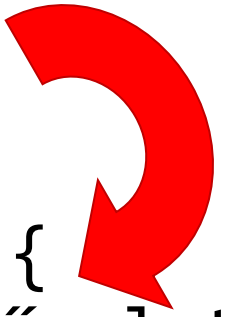
- A programok többszintűek
- Példa
  - egy fájlt nem tudunk megnyitni
  - egy üres veremből akar valaki kiolvasni

# Példa

```
gyumolcs {
     alma();
     //folytatás1
}
```



```
          alma {
               jonatan();
               //folytatás2
          }
```



```
                    jonatan {
                         //műveletek
                    }
```

# Hiba terjesztése hagyományosan

Tegyük fel, hogy a `jonatan()`-ban fellépő hibát a `gyumolcs()` kell lekezelje:

```
gyumolcs {
    ret=alma();
    if(ret!=ok) {
        //hibakezelés
    }
    else {
        //folytatás1
    }
}
```

```
alma {
    ret=jonatan();
    if(ret!=ok) {
        return hiba;
    }
    else {
        //folytatás2
    }
}
jonatan {
    //műveletek
    if( művelethiba ) {
        return hiba;
    }
}
```

# Hiba terjesztése kivételkezeléssel

Kivételkezeléssel a felfelé terjesztést egyszerűen és kevesebb módosítással lehet megoldani:

```
gyumolcs {
    try {
        alma();
        //folytatás
    }
    catch(VmiExc) {
        //hibakezelés
    }
}
```

```
jonatan throws VmiExc {
    //műveletek
}
```

```
alma throws VmiExc {
    jonatan();
    //folytatás
}
```

# Kivétel és nem hiba

```
type Napok is (Vasarnap, Hetfo, Kedd, … Szombat);

function Holnap(Ma: Napok) return Napok is
begin
   return Napok'Succ(Ma);
exception
   when Constraint_Error => return Napok'First;
end Holnap;
```

# Kérdések

Hiba- vagy kivételkezelést ad-e a nyelv?

Hogyan kell kivételeket definiálni-kiváltani-kezelni?

Milyen a kivételkezelés szintaktikai formája

Mihez kapcsolódik a kezelés: utasítás-, blokk- vagy eljárás/függvényszintű?

A kivételekből lehet-e kivételcsoportokat, kivételosztályokat képezni, amelyeket a kivételkezelőben egységesen lehet kezelni?

# Kérdések

Paraméterezhető-e a kivétel információ továbbadás céljából?

Támogatja-e a nyelv explicit módon valamilyen végső tevékenység („finally”) megadását?

- Ha nem, akkor tudjuk-e szimulálni azt?
- Milyen megszorításokkal?

A nyelv biztosít-e olyan nyelvi konstrukciót, amelynek nem kell explicit módon megadni, hogy mely kivételt kell továbbadni?

# Kérdések

Újrakezdhető-e az alprogram (blokk) a hiba kezelése után?

Megadható-e /meg kell-e adni, hogy egy alprogram milyen lekezeletlen  kivételeket küldhet?

Párhuzamos környezetben vannak-e speciális kivételek?

Mi történik a váratlan kivételekkel?

Kiváltható-e kivétel kivételkezelőben?

Melyik kivételkezelő kezeli a kivételt?

# C++

Kivétel lehet egy  objektum, aminek tetszőleges a típusa.

Hasznos, ha definiálunk osztályokat a felhasználó kivételeihez

Például

- `class MyException {};`

# C++

Néhány predefinit kivétel:

- `bad_cast`: ha a dynamic_cast operator nem használható egy referenciára,
- `bad_typeid`: ha a typeid operátora egy null pointer dereferenciája.
  (mindkettő a typeinfo.h-ban deklarálva).

# C++

## Kivétel kiváltása

- `throw [expression] ";"`
  A kifejezés egy időszakos objektumba kerül.
- A normál program lefutás megszakad, a vezérlés a legközelebbi megfelelő kezelőhöz kerül
- Ha nincs kifejezés: csak kivételkezelőben, illetve innen hívott függvényben lehet, az aktuális kivétel újrakiváltása.
- A fordító nem biztos, hogy ellenőrzi, ha `throw` utasítás kivétel-objektum nélkül: `terminate` függvény hívása.

# C++

Példák

- `throw 5;`
- `throw "An exception has occurred";`
- `throw MyException();`
- `throw;`

További információk: a kivételosztályban lehetnek nyilvános attribútumok, konstruktor beállíthatja stb.

```
class ExceptionWithParameters {
 public:
  int a,b;
  ExceptionWithParameters(int a0,int b0)
                                  { a=a0; b=b0; }
  };

  throw ExceptionWithParameters(0,1);
```

# C++

Kivételek kezelése - try blokkal:

- `try compound_statement handler_list`
  `ahol`
  `handler_list = handler{handler}`
- `handler = catch "(" exception_declaration ")"`
  `compound_statement`
  `exception_declaration = type [ident] |"..."`

# C++

A dobott kivételeknek megfelel egy `catch` ág, ha a következő feltételek közül valamelyik teljesül:

- A két típus pont ugyanaz.
- A `catch` ág típusa publikus bázisosztálya az eldobott objektumnak.
- A `catch` ág típusa mutató, és az eldobott objektum olyan mutató, melyet valamely standard mutató konverzióval át lehet konvertálni a `catch` ág típusára.

A kezeletlen kivételek továbbgyűrűződnek a hívó `try` blokkban, majd az azt hívóban.

- A legkülső `try` blokk után a *terminate* függvény kerül meghívásra.

# C++

Például

```
class A {};
class B: public A {};
class C: public B {};
class D: public A {};
class X {};
class AX: public A, public X {};

catch (A)          // A, B, C, D, AX típusúakat kap el
catch (A*)         // A*,B*,C*,D*,AX*
catch (X&)         // X és AX
catch (const B&) // B és C
catch (char*)      // char*
catch (void*)      // tetszőleges pointer típusút
```

# C++

Mindig az első kezelőt választja ki

- ha rossz sorrendet írunk, nem hiba!

```
try { // ...
  }catch (A) {/* ... */}
   catch (B) {/* ... */}//soha nem jön ide!

try {// ...
  }catch (B) { /* ... */ }
   catch (A) { /* ... */ } // így OK.
```

# C++

Lehet a kivételobjektumra is hivatkozni:

```
catch (ExceptionWithParameters e) {
    cout<< e.a; // kiírja
}
```

Lehet ...

- ez bárminek megfelel – utolsó legyen a `try` blokkban

A `terminate` hívása általában abort-ot jelent

- A felhasználó a `set_terminate`-tel megadhat mást is
- ez is le kell állítsa a programot.

Miközben a vezérlés átadódik a kivételkezelőnek, destruktort hív minden automatikus objektumra, ami a `try` blokkba történő belépés óta keletkezett.

# C++

Kivétel specifikációk

- `throw "(" [type {"," type}] ")"`

Például: `void f(int x) throw (A,B,C);`

Az f függvény csak A,B és C típusú kivételt generál, vagy azok leszármazottait.

- `int f2(int x) throw ()` - f2 nem válthat ki kivételt!
- `void f3(int x)` - f3 bármit kiválthat!

Ha specifikáltunk lehetséges kivételtípusokat, akkor minden más esetben a rendszer meghívja az `unexpected()` függvényt

- Az alapértelmezett viselkedése a `terminate()` függvény meghívása
- Ez a `set_unexpected` segítségével szintén átállítható.

# C++

Példák:

```
class X {};
class Y: public X {};
class Z {};
void A() throw(X,Z) // A X,Y,Z-t dobhat
{ /* ... */ }
void B() throw(Y,Z) {
   A();      // unexpected, ha A X-t dob, ok Y, Z típusúra
}
void C();    // semmit sem tudunk C-ről
void D() throw() {
   C();      // ha C bármit kivált, unexpected -t hív
   throw 7; // unexpected-t hív
}
```

# Java

A C++ szintaxistól nem sokban különbözik.

- Eltérések a szemantikában.

```
try {
     throw new EgyException ("parameter");
}
catch (Type variable) {

}
finally {

}
```

# Java

A továbbiakban csak az eltérések:

- **`finally`** nyelvi konstrukció, ezzel a Java megbízhatóbb programok írását segíti elő.

Egy függvény által kiváltható kivételek specifikálása:

- `void f (int x) throws AException, BException;`

Egy szálon futó program esetén ha a virtuális gép nem talál a kivétel kezelésére alkalmas kódot, akkor a VM és a program is terminál.

- Ha többszálú a program, akkor egy `uncaughtException()` metódus fut le.

# Java

Minden kivétel a `java.lang.Throwable` leszármazottja.

Ha olyan kivételt szeretnénk dobni, amely nem a `Throwable` leszármazottja, akkor az fordítási hibát okoz.

A kivételek két nagy csoportba sorolhatóak

- ellenőrzöttek: `Exception` leszármazottjai
- nem-ellenőrzöttek: `Error` leszármazottjai

# Java

Számos olyan kivétel van, ami

- előre nem látható
- És fölösleges lenne mindenhol lekezelni őket.
- Ezeket a kivételeket nevezzük nem-ellenőrzött kivételeknek.
  - Például nem ellenőrizzük le minden utasítás végrehajtása előtt, hogy van-e elég memóriánk stb.

Ez az oka a kivételek két csoportjának.

- A csoportosítás nem konzisztens
  - Az `Exception` egyik leszármazott osztálya, a `RuntimeException` és leszármazottjai sem ellenőrzöttek.

Az ellenőrzött kivételek esetén fordítási hiba lép fel, ha nincsenek specifikálva vagy elkapva.

- Továbbá ha olyan ellenőrzött kivételt kívánunk elkapni, amely hatókörön kívül van.

# Java

Néhány predefinit nem ellenőrzött kivétel

A `RuntimeException` leszármazottjai

- `ArithmeticException`
- `ClassCastException`
- `IndexOutOfBoundsException`
  - `ArrayIndexOutOfBoundsException`
  - `StringIndexOutOfBoundsException`
- `NullPointerException`.

Az `Error` leszármazottjai

- `OutOfMemoryError`
- `StackOverflowError`
- stb.

# Java példa

```
class VeremException extends Exception {}
class VeremMegteltException extends VeremException
{
      private int utolso;
      public VeremMegteltException (int i) {
            utolso = i;
      }
      public int miVolt () {
            return utolso;
      }
}
```

# Java példa

```
class Verem {
  final static public int MERET = 10;
  private int tarolo [] = new int[MERET];
  private int mutato = 0;
  public void betesz (int i) {
    try {
      if (mutato < MERET)
        tarolo [mutato++] = i;
      else
        throw new VeremMegteltException (i);
    } catch (VeremMegteltException e) {
      System.out.println( e.miVolt () + " nem fert be");
      throw;
    }
    finally {
      System.out.println ("finally: mindig lefutok");
    }
  }
}
```

Mi a gond ezzel?

# Java

Predefint kivételek előfordulása:

```
class A { }
class B extends A { }
class C {
   void X() {
      A a = new A;
      B b = (B)a; // ClassCastException
   }
   void Y() {
     int ia[] = new int[10];
     for (int i=1; i<=10; i++)
       ia[i]=0;  // amikor i==10 lesz: ArrayIndexOutOfBoundsException
   }
   void Z() {
     C c=null;
     c.X();       // NullPointerException
   }
}
```

# Java

Kivételek kezelése

```
try {
  // utasítások
}
catch (MyException e) {
  // utasítások MyException kezelésére
}
catch (AnotherException e) {
  // utasítások AnotherException kezelésére
}
finally {
  // mindig végrehajtódó utasítások
}
```

# Java

```
class A extends Exception {}
class B extends A {}

try {
    // ...
}
catch (A a) { /* ... */ }
catch (B b) { /* ... */ }

void MyMethod() throws MyException {
    // utasítások
    throw new MyException();
    // utasítások
}
```

# Java

```
class ExcA extends Exception {}
class ExcD extends RuntimeException {}
class ExcF extends ExcA {}

void A() throws ExcA {
    // ...
    throw new ExcA();
    // ...
}
```

# Java

```
void B1() {
    A(); // Hiba: ExcA nincs lekezelve B1-ben
}
void B2() {
  try {
     A(); // OK
  } catch (ExcA e) {
     // exception kezelése
  }
 }
void B3() throws ExcA {
     A(); // OK
}
```

# Java

```
void D() throws ExcD {
    // ...
}
void E() {
    D(); // ok, ExcD leszármazottja
         // RuntimeException-nak,
         // így nem kell jelezni
}
void F() throws ExcA {
      throw new ExcF();    // OK
}
```

# Java try-with-resources

További hasznos lehetőség a finally használata helyett:

```
try (BufferedReader br =
        new BufferedReader(new FileReader(path)))
{
    return br.readLine();
}
```

Ehhez a `Closeable` vagy `AutoCloseable` interfészt kell megvalósítania a használandó osztálynak

# OOP

## KÖVETKEZŐ HÉTEN




# Programozási nyelvek és módszerek

OOP – I. (ISMÉTLÉS)

# Kivételkezelés

# C# – .NET

Közös elv alapján a .NET-ben

Nagyon hasonlít a Javához, de van különbség is

Hasonlóság:

- `try` – `catch` blokk, (ellenőrzi a jó sorrendet)
- `finally` lehetősége
- közös ősosztály (`System.Exception`)

Különbség:

- Nincs exception-specifikáció

# C# – .NET

## Miért nincs a C#-ban ellenőrzött kivétel?

- Verziókezelés
  - Ha egy következő verzióban egy metódus új kivételt dob, akkor az őt hívó metódust is változtatni kell
  - Ez máshol is probléma, ha le akarjuk kezelni az új kivételt, de legtöbbször nem kezelik
  - 10 az 1-hez a `try`–`finally` és a `try`–`catch` aránya
- Méretezhetőség:
  - Nagy projektekben, négy-öt alrendszerrel a sok kivétel már kezelhetetlenné válik
  - Ilyenkor keletkezik sok `catch { }` üresen
- Nem szigorú szabályok kellenek a kezeletlen kivételek ellen, hanem elemző eszközök, amelyek felkutatják a gyanús kódokat, lehetséges réseket

# C# – .NET

```
using System;
class ExceptionTestClass {
  public static void Main() {
    int x = 0;
    try {
      int y = 100/x;
    }
    catch (ArithmeticException e) {
        Console.WriteLine("ArithmeticException: {0}", e.ToString());
    }
    catch (Exception e) {
        Console.WriteLine("Generic Exception: {0}", e.ToString());
    }
    finally {
        Console.WriteLine("It is always executed.");
    }
  }
}
```

# C# – .NET

Saját exception-osztály:

```
using System;
public class EmployeeListNotFoundException: ApplicationException {
  public EmployeeListNotFoundException() { }
  public EmployeeListNotFoundException(
           string message) : base(message) { }
  public EmployeeListNotFoundException(
           string message, Exception inner) :
           base(message, inner) { }
}
```

# Delphi

Eltérő szintaktikára példa

```
begin
  Try
    // The code we want to execute
    ...
  Except
    // Exception handling
    ...
  Finally
    // Finally block
    ...
  end;
end;
```

# Delphi

## Több kivételtípus kezelése

```
except
    // IO error
    On E : EInOutError do
      ShowMessage('IO error : '+E.Message);
    // Dibision by zero
    On E : EDivByZero do
      ShowMessage('Div by zero error : '+E.Message);
    // Catch other errors
    else
      ShowMessage('Unknown error');
 end;
```

# Delphi

## Kivétel kiváltása

- `raise object at address;`
- Példa

```
raise EMathError.Create;
raise Exception.Create at @MyFunction;
```

# Python

Előzőekhez hasonlóan

```
def divide(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("division by zero!")
    else:
        print("result is", result)
    finally:
        print("executing finally clause")
```

- Az **else** jelentősége, hogy lokális változókhoz hozzáfér, de a **try** által védett kódot már nem tartalmaz

# Python

Előredefiniált clean-up kódok lefuttatása

- A következő kód helyett

```
file = open('file_path', 'w')
try:
    file.write('hello world')
finally:
    file.close()
```

- Írható ez is (a try-with-resources mintájára)

```
with open('file_path', 'w') as file:
    file.write('hello world !')
```

# Python

A **with** használatára alkalmas osztály készíteni az __enter__ és __exit__ függvények definiálásával lehet

```
class MessageWriter(object):
    def __init__(self, file_name):
        self.file_name = file_name

    def __enter__(self):
        self.file = open(self.file_name, 'w')
        return self.file

    def __exit__(self):
        self.file.close()
```

# SWIFT

A kivételeket az `Error` protokollt megvalósító felsorolási típusok segítségével reprezentálja

- SWIFT esetén a protokoll megközelítőleg a Java interfész fogalomnak felel meg

Például

```
enum VendingMachineError: Error {
    case invalidSelection
    case insufficientFunds(coinsNeeded: Int)
    case outOfStock
}
```

# SWIFT

Az egyes esetek paraméterezhetők, amivel a kivétel specifikusabban megadható.

Kiváltani a **`throw`** utasítással lehet

- `throw VendingMachineError.insufficientFunds(coinsNeeded: 5)`

# SWIFT

## Kivételek kezelése

- Egyrészt, a Java-hoz hasonlóan egy függvénynek jeleznie kell, hogy kivételt dobhat
  - `func canThrowErrors() throws -> String`
  - `func cannotThrowErrors() -> String`
- Ha kritikus függvényt hívunk, akkor **`try`** kulcsszóval kell tenni
  - `try canThrowErrors()`

# SWIFT

A kapott kivételt vagy kezelni kell, vagy jelölni, hogy tovább tud dobódni.

Kivétel kezelés:

```
do {
    try buyFavoriteSnack()
} catch VendingMachineError.invalidSelection {

} catch VendingMachineError.outOfStock {

} catch VendingMachineError.insufficientFunds(let c) {

}
```

# SWIFT

## További lehetőségek

- Értékadásnál, ha nem sikerült, akkor null értéket ad az új változónak
- Ezt később lehet kezelni
  - Optional – formájában

```
let x = try? someThrowingFunction()
```

  - Összeköthető feltétel vizsgálattal is

```
func fetchData() -> Data? {
    if let data = try? fetchDataFromDisk()   { return data }
    if let data = try? fetchDataFromServer() { return data }
    return nil
}
```

# SWIFT

## További lehetőségek

- Amennyiben biztosak vagyunk, hogy a hívott függvény valóságban nem dobhat kivételt akkor

```
let photo = try! loadImage( )
```

- Ebben az esetben, ha mégis keletkezik kivétel, akkor a futás megszakad és egy futás idejű hiba keletkezik

# SWIFT

## Hibától függetlenül lefutó kód

```
func processFile(filename: String) throws {
    if exists(filename) {
        let file = open(filename)
        defer {
            close(file)
        }
        while let line = try file.readline() {
           // line
        }
        // close(file) hívása itt történik a scope végén
    }
}
```

- Tehát a **`defer`** részben megadott hívás a blokk befejeződése előtt lefut

# Eiffel

## Újrakezdés

- A komponens írásakor egy kivétel lehetőségét előre lehet látni és egy alternatív megoldást találni a szerződés betartatására.
- Ekkor a végrehajtás megpróbálja ezt az alternatívát.

## Szervezett pánik

- Ha nincs rá mód, hogy teljesítsük a szerződést, akkor az objektumokat egy elfogadható állapotba kell hozni (típusinvariáns helyreállítása), és a felhasználónak jelezni kell a kudarcot.

# Eiffel – Újrakezdés

```
try_once_or_twice is
  local
    already_tried : BOOLEAN
  do
    if not already_tried then
      method_1
    else
      method_2
    end
  rescue
    if not already_tried then
      already_tried := true;
      retry
    end
 end -- try_once_or_twice
```

# Példa

```
transmit: (p: PACKET)
-- Transmit packet `p'
require
      packet_not_void: p /= Void
local
      current_retries: INTEGER
      r: RANDOM_NUMBER_GENERATOR
do
      line.send (p)
rescue
      if current_retries < max_retries then
            r.next
            wait_millisecs (r.value_between (20, 500))
      current_retries := current_retries + 1
            retry
      end
end
```

# Eiffel – Szervezett pánik

```
attempt_transaction(arg : CONTEXT) is
  -- megpróbáljuk a transaction-t arg argumentummal;
  -- ha nem sikerül, az akt. obj. invariánsát visszaállítjuk
  require
     ...
  do
     ...
  ensure
     ...
  rescue
     reset(arg)
 end -- attempt_transaction
```

# Eiffel

```
default_rescue is
do
end --default_rescue

class C creation
  make ...
inherit
  ANY
    redefine default_rescue end
  ...
feature
  make, default_rescue is
     -- nincs előfeltétel
    do ...
    end;
  ...
end -- class C
```

# Kivételkezelés – érdekesség

A C++ nyelvet számos kritika éri, többek között a kivételkezelés kapcsán.

Eredményképpen számos Code-Style tiltja

- Google
- LLVM
- Qt

# Objektum Orientált Programozás

# Az OO paradigma

Mitől OO egy program?

Objektum

Osztály

Öröklődés

# A valós világ modellezése

## Az ember a világ megértéséhez modelleket épít

## Modellezési alapelvek

- Absztrakció
  - Szemléletmód, amelynek segítségével a valós világot leegyszerűsítjük
    - Csak a lényegre, a cél elérése érdekében feltétlenül szükséges részekre összpontosítunk
  - Elvonatkoztatunk a számunkra pillanatnyilag nem fontos, közömbös információktól
    - Kiemeljük az elengedhetetlen fontosságú részleteket.
- Megkülönböztetés
  - Az objektumok a modellezendő valós világ egy-egy önálló egységét jelölik.
  - Az objektumokat a számunkra lényeges tulajdonságaik, viselkedési módjuk alapján megkülönböztetjük.

# A valós világ modellezése

## Modellezési alapelvek

- Osztályozás
  - Az objektumokat kategóriákba, osztályokba
    - a hasonló tulajdonságokkal rendelkező objektumok egy osztályba kerülnek.
    - a különböző vagy eltérő tulajdonságokkal rendelkező objektumok külön osztályokba kerülnek.
  - Az objektum-osztályok hordozzák a hozzájuk tartozó objektumok jellemzőit, objektumok mintáinak tekinthetők.
- Általánosítás, specializálás
  - Az objektumok között állandóan hasonlóságokat vagy különbségeket keresünk
    - Ezáltal bővebb vagy szűkebb kategóriákba, osztályokba soroljuk őket.

# OOP – alapelvek

OOP Alapelvek (Benjamin C. Pierce)

- Dynamic binding (dinamikus kötés)
  - Egy objektum esetén dinamikusan, futási időben dől el, hogy egy metódus melyik implementációja kerül futtatásra
- Encapsulation (egységbe zárás)
  - Adatok és rajtuk végrehajtható műveletek egységet alkotnak
  - Praktikusan ez a modern típus-definícióval konzisztens
- Subtype polymorphism (altípusos polimorfizmus)
  - Egy rögzített típusú változó több a típus altípusának példányára is hivatkozhat
  - Altípus
    - Az eredeti típus megszorításával létrehozott új típus
- Inheritance, vagy delegation (öröklődés, delegáció)
  - Egy adott osztályból lehetőség van képezni egy másik osztályt
    - Ez az ős tulajdonságait megtartja
    - Azonban módosíthatja, bővítheti
- Open recursion (nyílt rekurzió)
  - Speciális változó, amely egy metódus esetén lehetővé teszi az aktuális példány elérését

# Objektum

Belső állapota van
- Ebben információt tárol
- Változókkal valósítjuk meg
  - Ezek az adattagok, vagy tulajdonságok.

Kérésre feladatokat hajt végre
- Metódusok – melyek hatására állapota megváltozhat

Üzeneteken keresztül lehet megszólítani
- Kommunikál más objektumokkal

Minden objektum egyértelműen azonosítható

# Osztály, példány

Osztály (class)

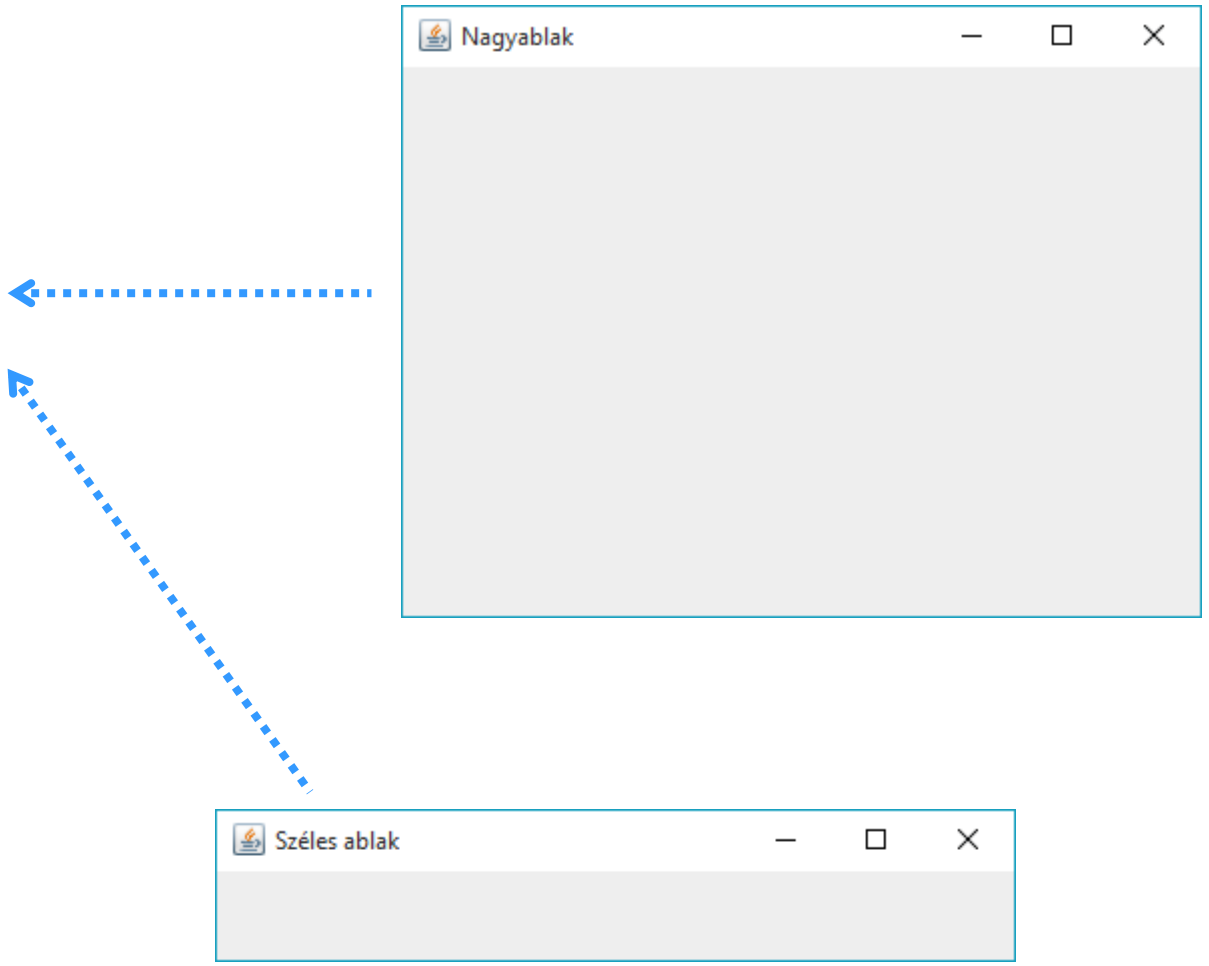
- Olyan objektumminta vagy típus
- Ez alapján példányokat (objektumokat) hozhatunk létre

Példány (instance)

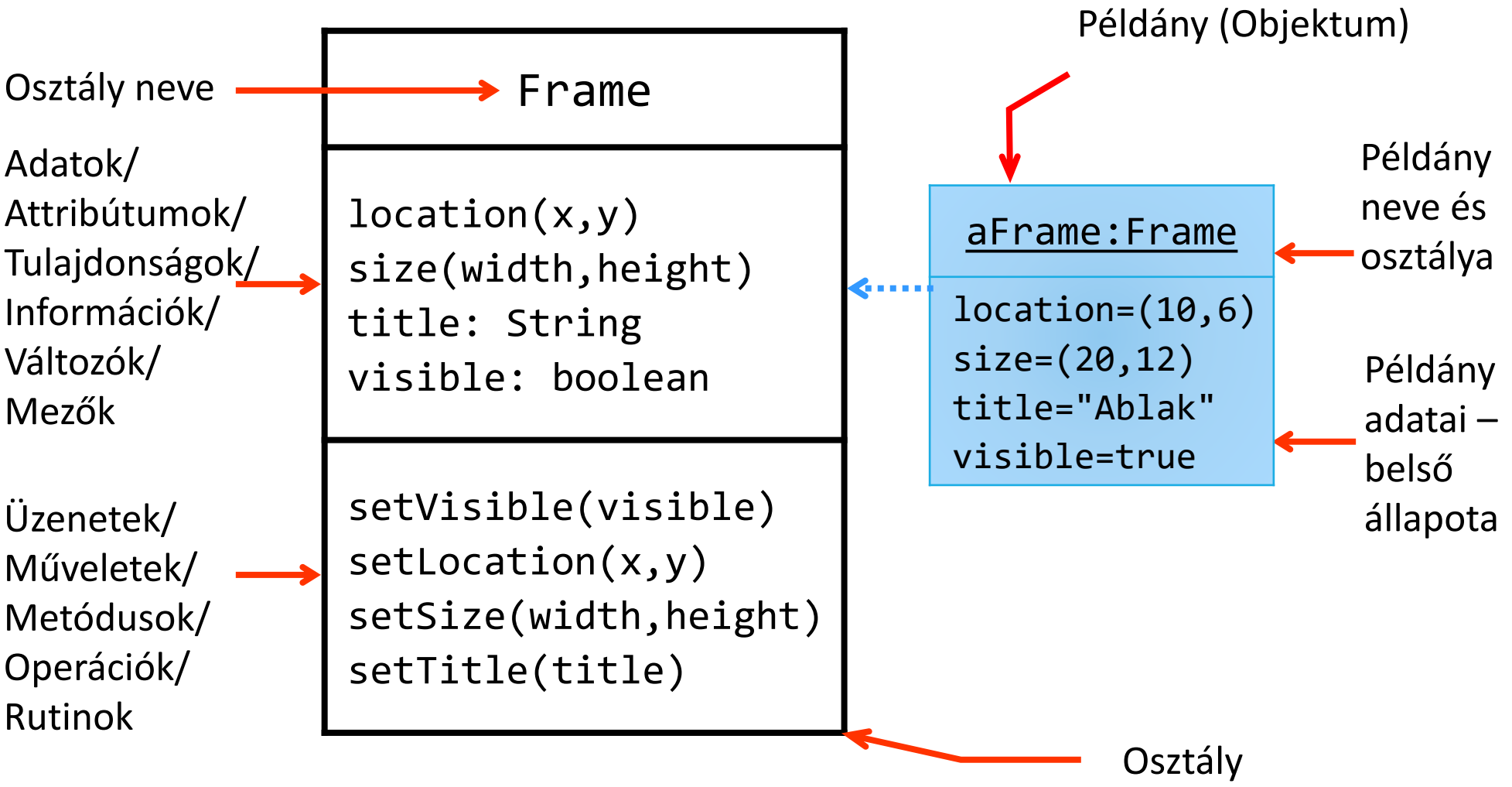
- Egy osztály (minta) alapján létrejött konkrét példány
- Minden objektum születésétől kezdve egy osztályhoz tartozik

# Frame osztály és példányai

| Frame |
| --- |
| location(x,y)<br>size(width,height)<br>title: String<br>visible: boolean |
| setVisible(visible)<br>setLocation(x,y)<br>setSize(width,height)<br>setTitle(title) |



# Osztály és példány az UML-ben



# Objektum életciklusa

Objektum életciklusa

- Létrejön – „megszületik”
- Állapotváltozásokon esik keresztül – „él”
- Megszűnik – „meghal”

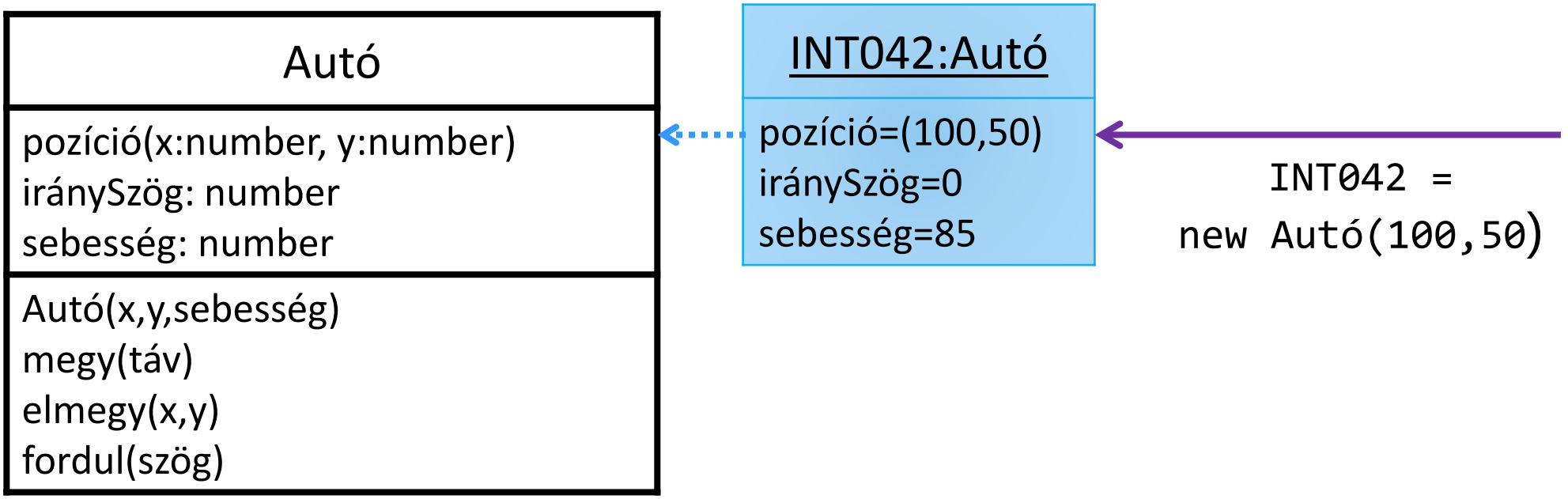
# Objektum létrehozása, inicializálása

Az objektumot létre kell hozni és inicializálni kell!

Objektum inicializálása

- Konstruktor (constructor) végzi
- Adatok kezdőértékadása
- Objektum működéséhez szükséges tevékenységek végrehajtása
- Típusinvariáns beállítása

# Objektum létrehozása, inicializálása



# Objektum állapotváltozása

Az objektum kérésre feladatokat hajt végre

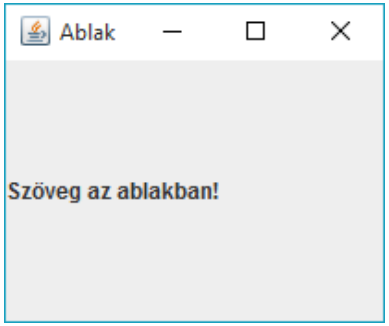
Életciklusa során üzeneteket fogad és feldolgozza

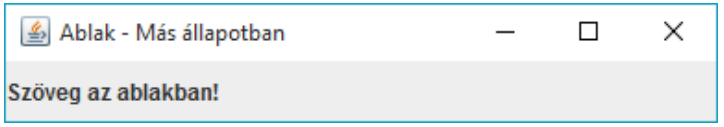
- Hatására az állapot megváltozik
- Ezek során üzeneteket küldhet más objektumoknak

# Állapotváltozás

Üzenetek

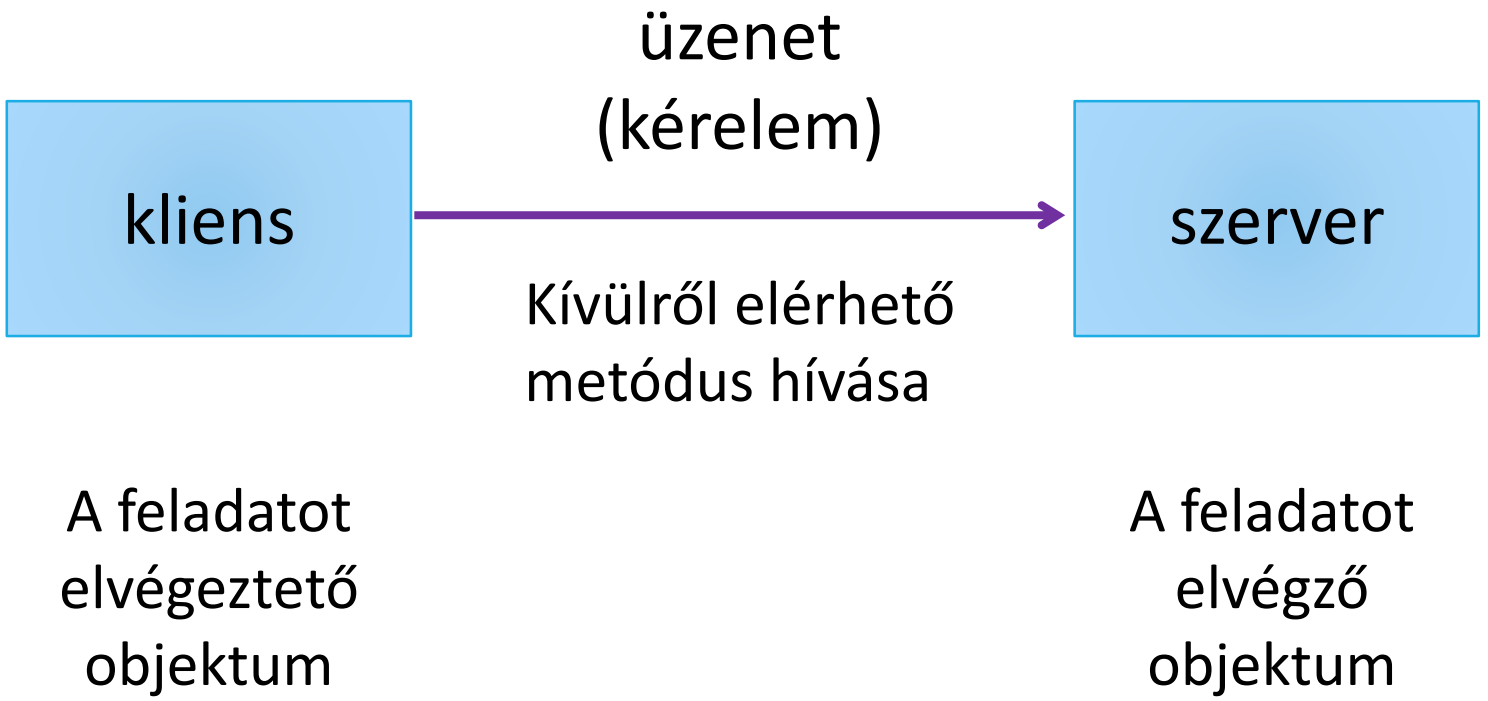
```
setVisible(true)
setLocation(40,8)
   setSize(20,16)
setTitle("Ablak")
```




| aFrame:Frame | |
|---|---|
| (20,16) | location(x,y) |
| (100,80) | size(width,height) |
| "Ablak" | title |
| true | visible |

# Üzenetküldési modell



# Üzenetküldési modell

Kliens

- Aktív objektum, másik objektumon végez műveleteket, de rajta nem végeznek
- Nincs export felülete
- Például óra (órajel)
  - Meghatározott időközönként művelet egy regiszteren

Szerver

- Passzív objektum
- Csak export felülete van
- Másoktól érkező üzenetekre vár, mások szolgáltatását nem igényli

Ágens

- Általános objektum, van export és import felülete

# Objektum műveletei

## Export műveletek

- Amelyeket más objektumok hívhatnak
- Például verem
  - `push`, `pop`, `top` stb.

## Import műveletek

- Amelyeket az objektum igényel ahhoz, hogy az export szolgáltatásait nyújtani tudja
- Például verem, ha fix méretű (vektoros) reprezentáció
  - Vektorműveletek

# Objektum műveletei

Export műveletek csoportosítása:

- Létrehozó (konstruktor)
  az objektum létrehozására, felépítésére
  - Példa veremnél
    - `create`: $\rightarrow$ Verem
- Állapot megváltoztató
  - Példa verem esetén
    - pop: Verem $\rightarrow$ Verem $\times$ Elem
    - `push`: Verem $\times$ Elem $\rightarrow$ Verem

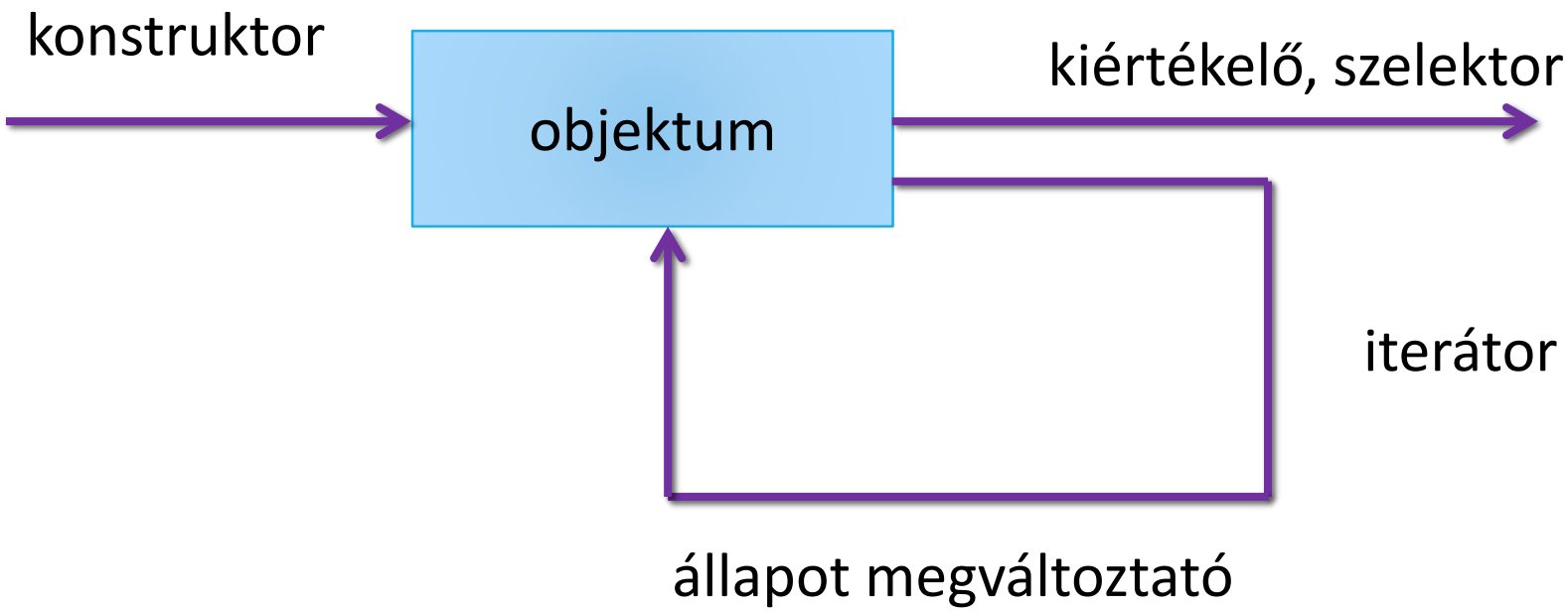
# Objektum műveletei

Export műveletek csoportosítása:

- Szelektor – kiemeli az objektum bizonyos részét
  - Például
    - vektor adott indexű elemét
    - `access`: $\text{Vektor} \times \text{Index} \rightarrow \text{Elem}$
- Kiértékelő – objektum jellemzőit lekérdező műveletek (`size`, `has`, stb.)
- Iterátor – bejáráshoz

# Objektum műveletei

Export műveletek csoportosítása:



# Objektum megszűnése

Az objektum életciklusának végén:

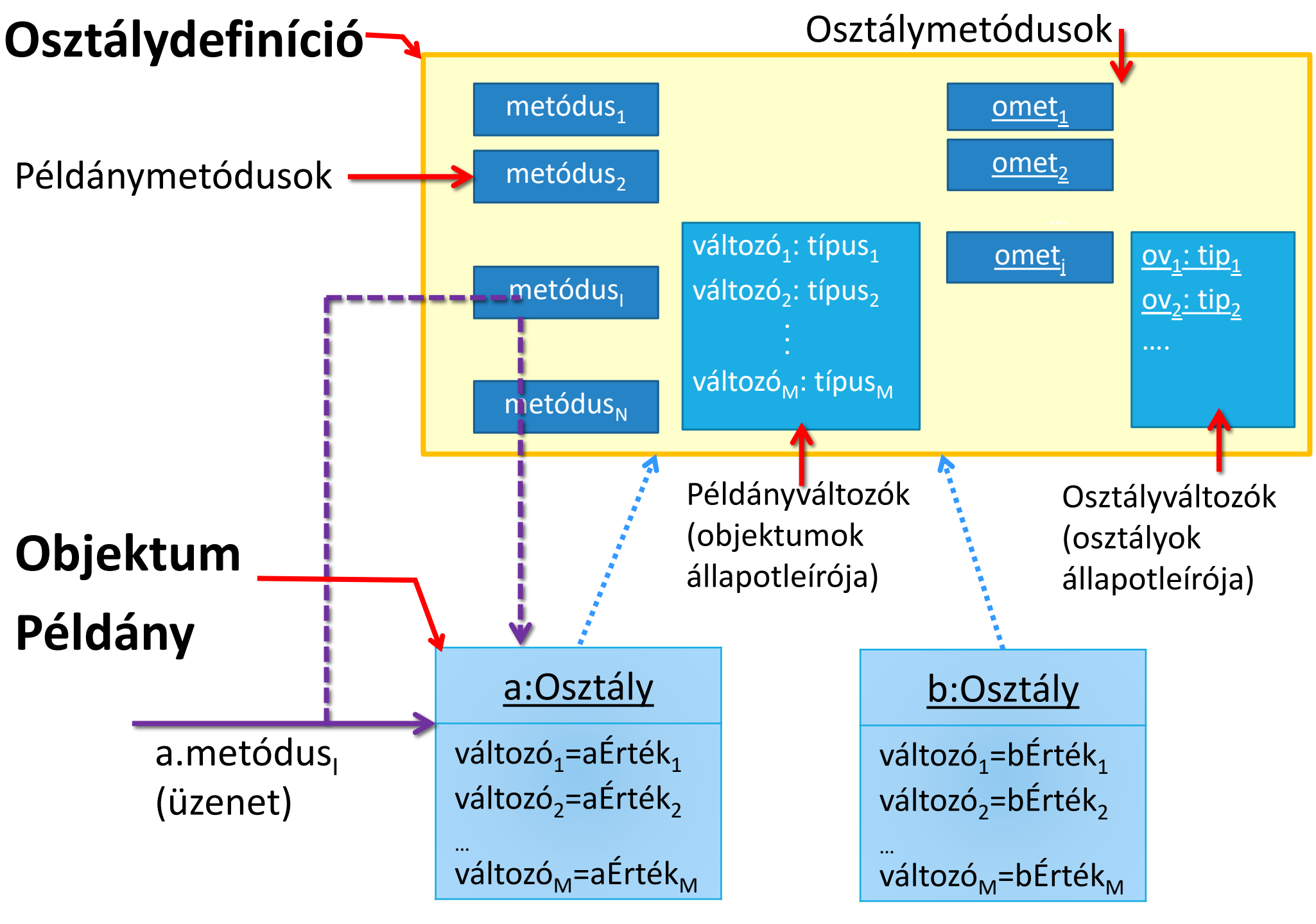
- Erőforrások (memória) felszabadítása
- Adatok, állapot mentése

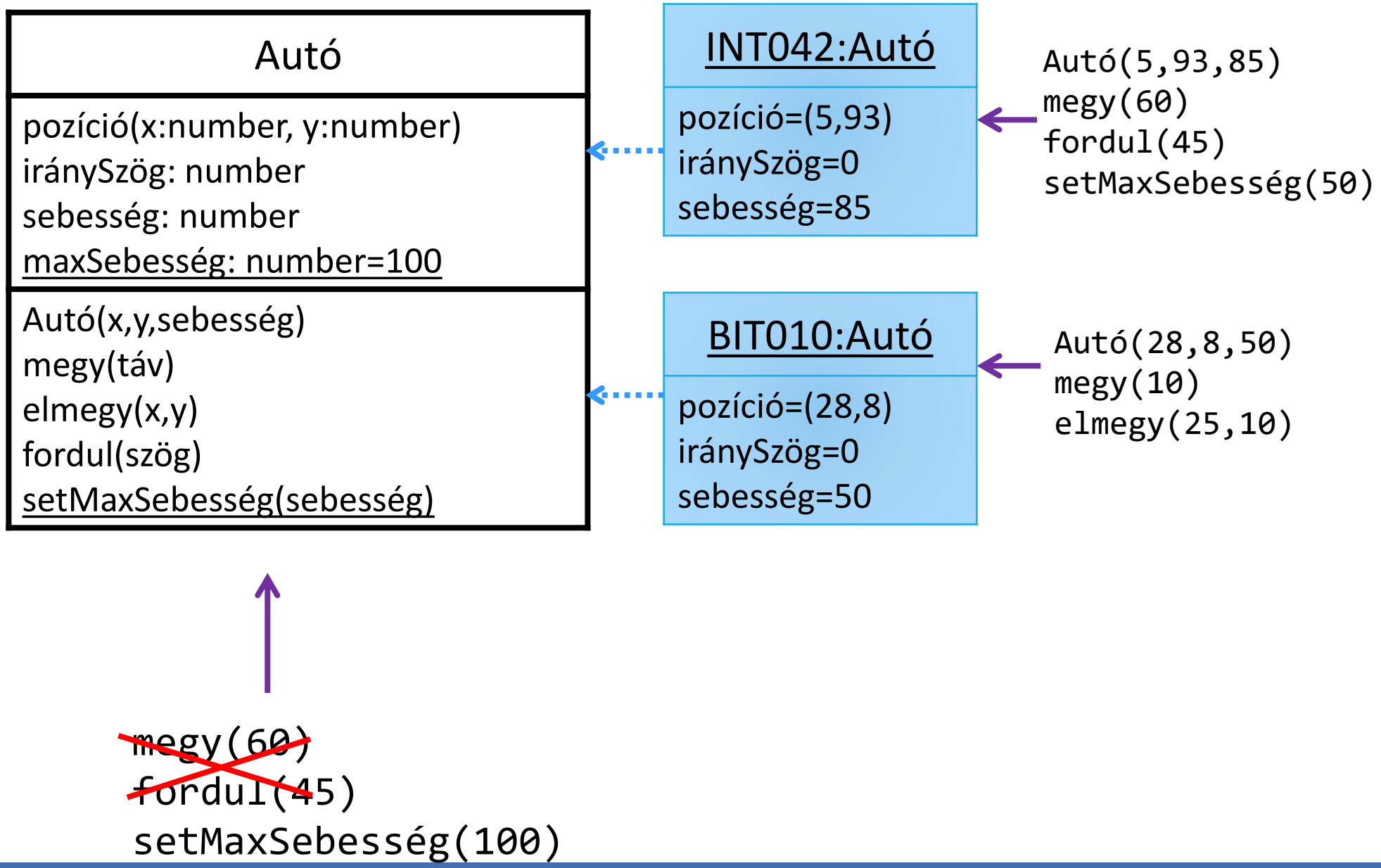
Ha a nyelv támogatja, akkor a destruktor (destructor) hajtja végre.

# Osztály, példány

Osztálydefiníció:

- Példányváltozó
  - Példányonként helyet foglaló változó
- Példánymetódus
  - Példányokon dolgozó metódus
- Osztályváltozó
  - Osztályonként helyet foglaló változó
- Osztálymetódus
  - Osztályokon dolgozó metódus


Osztálydefiníció
Osztálymetódusok
Példánymetódusok
metódus1
metódus2
metódusI
metódusN
omet1
omet2
...
ometj
változó1: típus1
változó2: típus2
változóM: típusM
ov1: tip1
ov2: tip2
....
Példányváltozók
(objektumok
állapotleírója)
Osztályváltozók
(osztályok
állapotleírója)
Objektum
Példány
a.metódusI
(üzenet)
a:Osztály
változó1=aÉrték1
változó2=aÉrték2
...
változóM=aÉrtékM
b:Osztály
változó1=bÉrték1
változó2=bÉrték2
...
változóM=bÉrtékM


Autó
pozíció(x:number, y:number)
iránySzög: number
sebesség: number
maxSebesség: number=100
Autó(x,y,sebesség)
megy(táv)
elmegy(x,y)
fordul(szög)
setMaxSebesség(sebesség)
INT042:Autó
pozíció=(5,93)
iránySzög=0
sebesség=85
Autó(5,93,85)
megy(60)
fordul(45)
setMaxSebesség(50)
BIT010:Autó
pozíció=(28,8)
iránySzög=0
sebesség=50
Autó(28,8,50)
megy(10)
elmegy(25,10)
megy(60)
fordul(45)
setMaxSebesség(100)

# A „this”

Adott osztályhoz tartozó példányok esetén

- Honnan tudjuk, hogy éppen melyik objektum hívta meg a megfelelő metódust
- Melyik objektum adataival fog dolgozni a metódus?

Szükségünk van egy olyan mutatóra, amely mindig a metódust meghívó objektumpéldányra mutat.

- Ezt szolgálja a „`this`” pszeudóváltozó
  - Néha explicit formális paraméter
- Ez metódushíváskor egyértelműen rámutat azokra az adatokra, amelyekkel a metódusnak dolgoznia kell.

Az objektum a saját magának küldött üzenet esetén a `this.Üzenet(Paraméterek)` formát kell, hogy használja.

- A metódustörzsekben az adott példányra mindig a `this` segítségével hivatkozhatunk.
  - Ez számos nyelvben alapértelmezett.

# OOP elvárások

## Bezárás (encapsulation)

- Adatok és metódusok összezárása
- Egybezárás, egységbezárás – osztály (class)

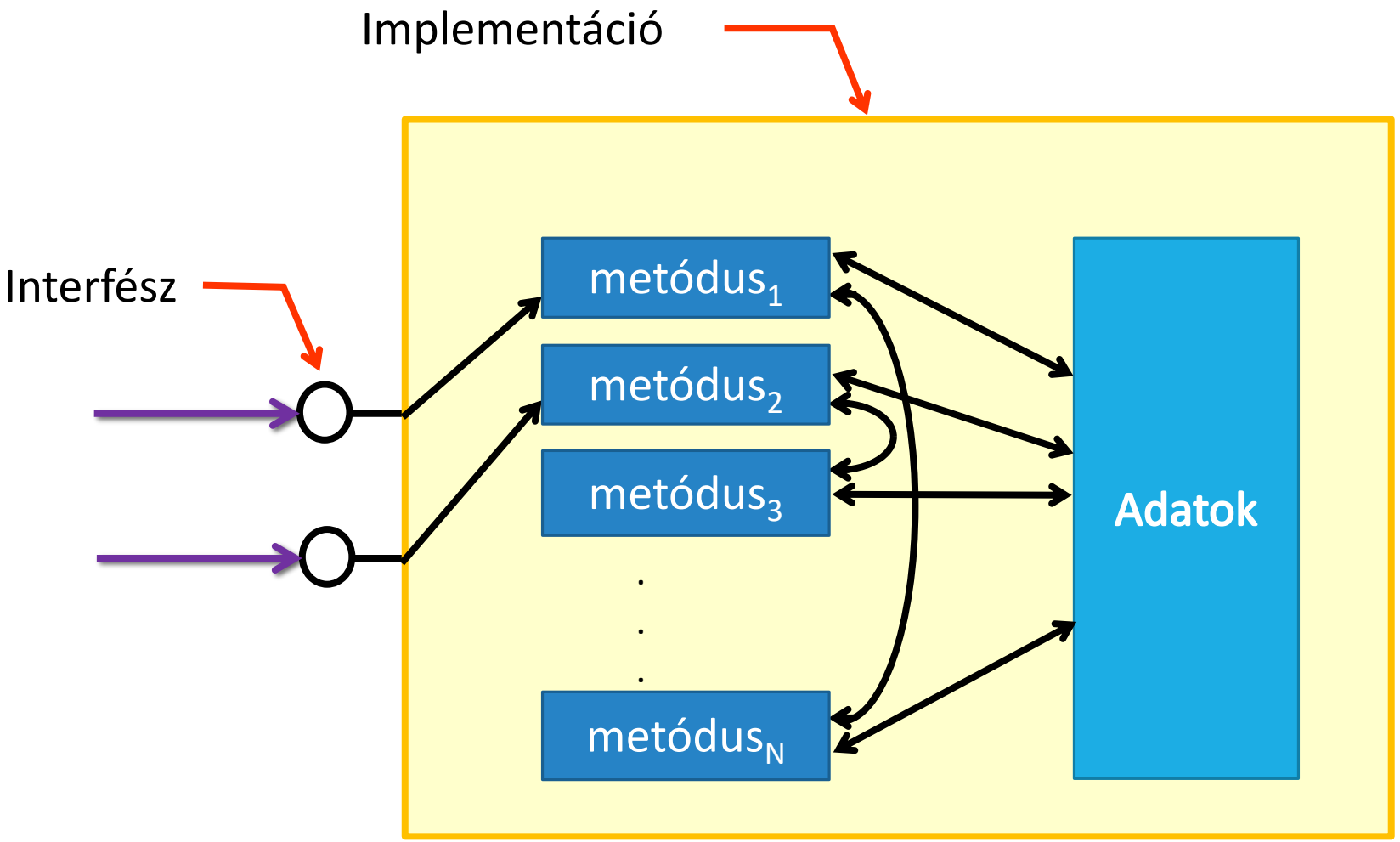
## Információ elrejtése (information hiding)

- Az objektum „belügyeit” csak az interfészen keresztül lehet megközelíteni (láthatóságok!)

## Kód újrafelhasználása (code reuse)

- Megírt kód felhasználása példány létrehozásával vagy osztály továbbfejlesztésével

# Információ elrejtése



# Információ elrejtése

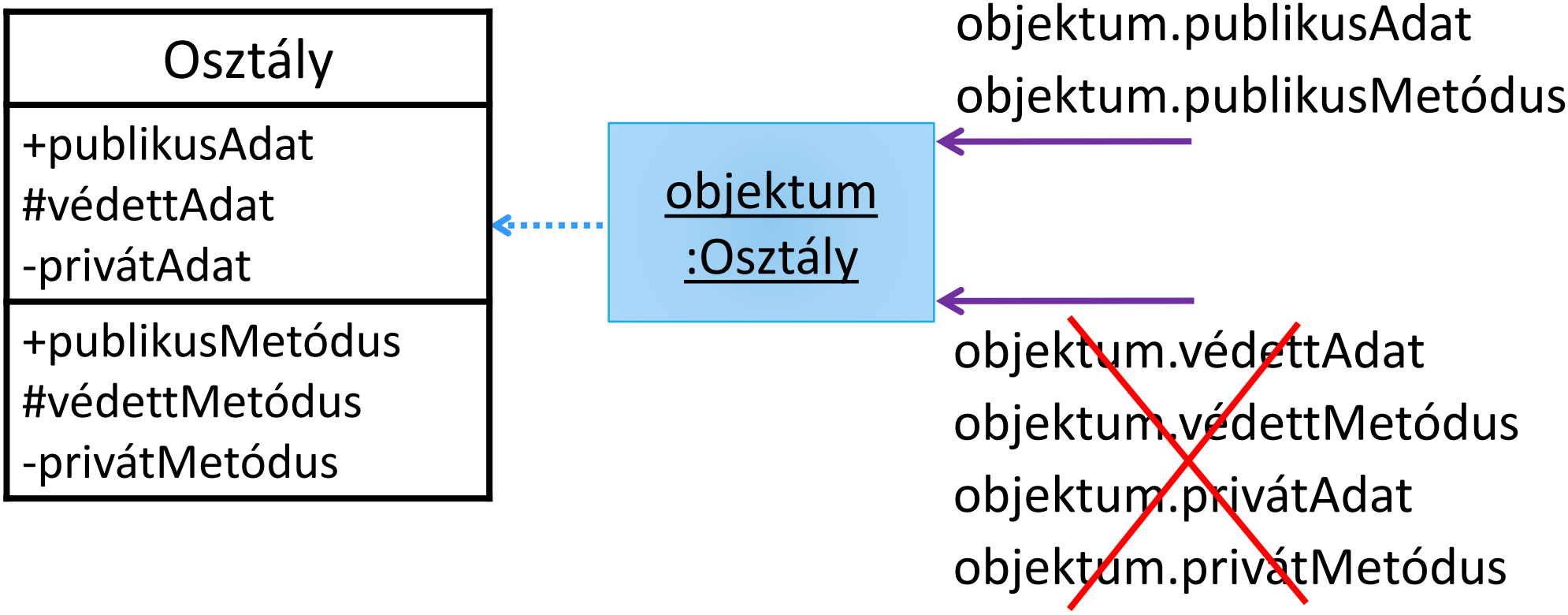
Az objektum adatait csak interfészen keresztül lehet elérni

- Az elv szigorú követése esetén, minden adattag elrejtésre kerül, közvetlenül nem hozzáférhető
  - Dedikált függvények használatával lehetséges a hozzáférés.

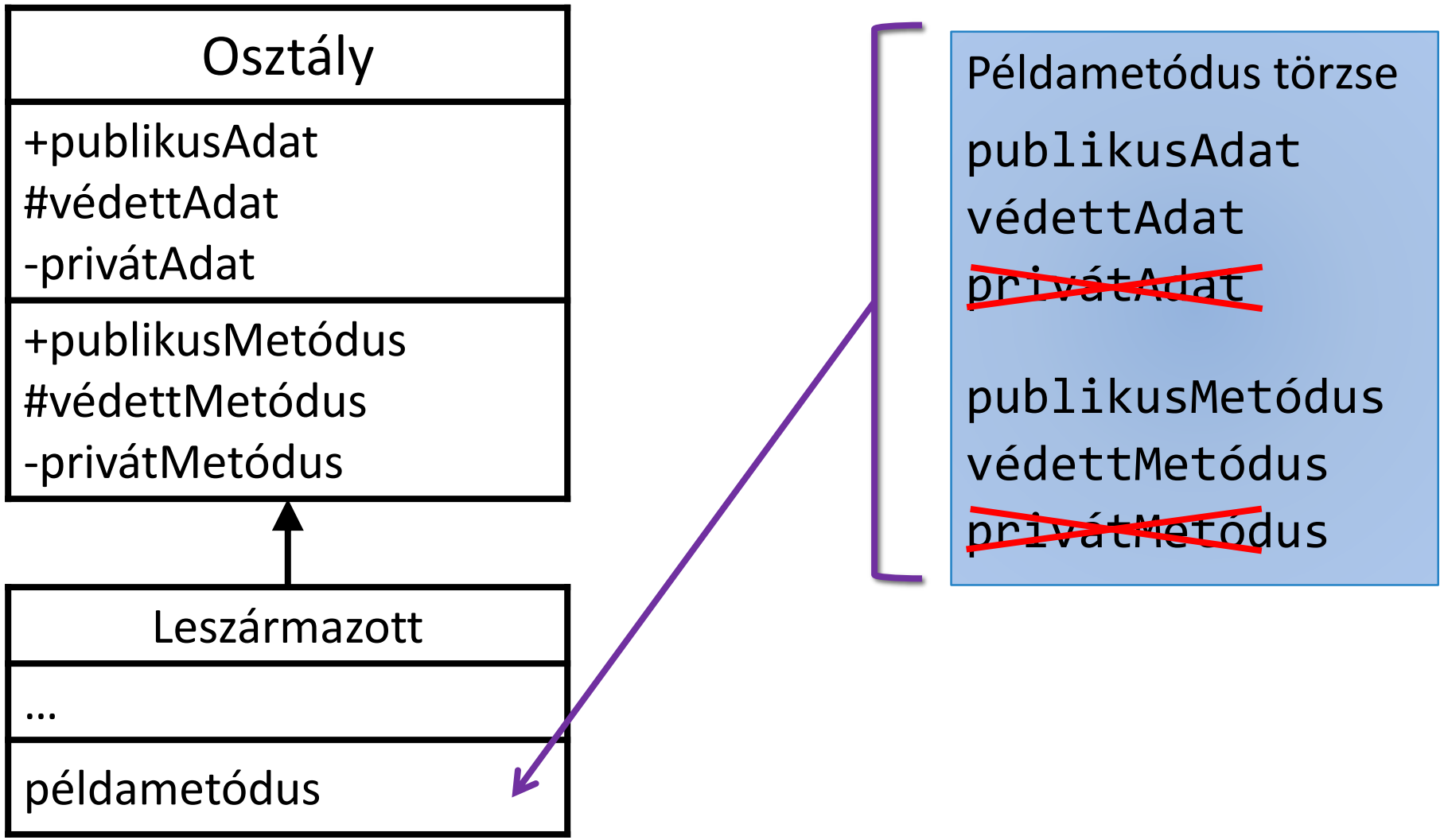
A láthatóság minősítője leggyakrabban

- public
  - a külső felhasználók elérik
- protected
  - csak a leszármazottak érhetik el
- private
  - csak az adott osztály számára elérhető

# Láthatóság – Objektum védelme



# Láthatóság – Objektum védelme



# Öröklés

Az alapgondolat: a gyerekek öröklik őseik metódusait és változóit

Az örököl azt jelenti, hogy az ősosztály minden metódusa és adattagja a gyerekosztálynak is metódusa és adattagja lesz

- A gyerek minden új művelete vagy adattagja egyszerűen hozzáadódik az örökölt metódusokhoz és adattagokhoz
- Minden metódus, amit átdefiniálunk a gyerekben, a hierarchiában felülbírálja az örökölt metódust

Lehet egyszeres, vagy többszörös

- Ez a közvetlen ősosztályok számát jelenti

# Öröklés

Mit örököl a leszármazott?

- Tulajdonságokat (Adattagokat)
- Metódusokat (Üzeneteket)

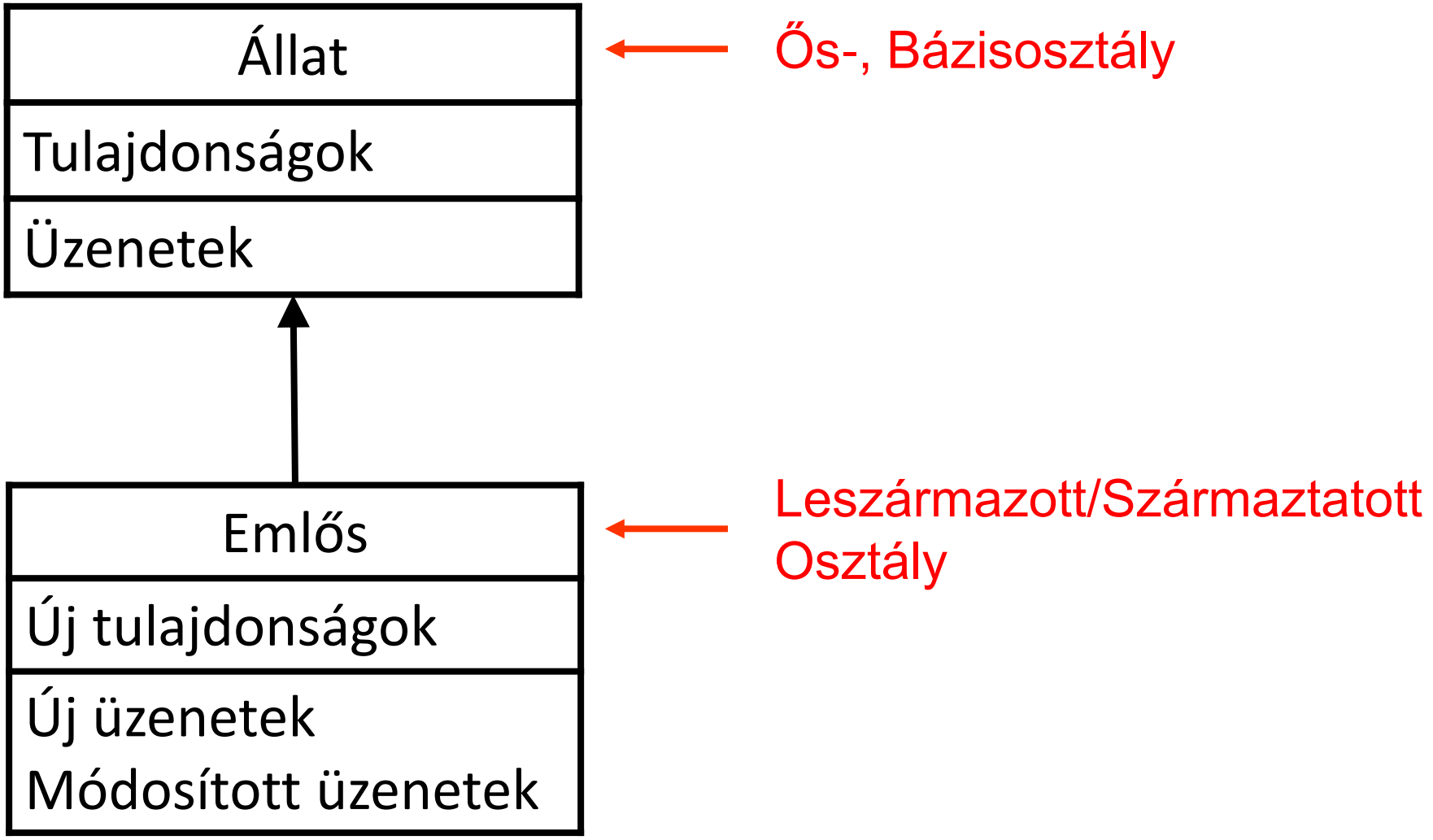
Mit nem örököl a leszármazott?

- Ősosztály konstruktorait, destruktorát
  - Ám tudja használni / meghívni / delegálni
- Ősosztály értékadás operátorát

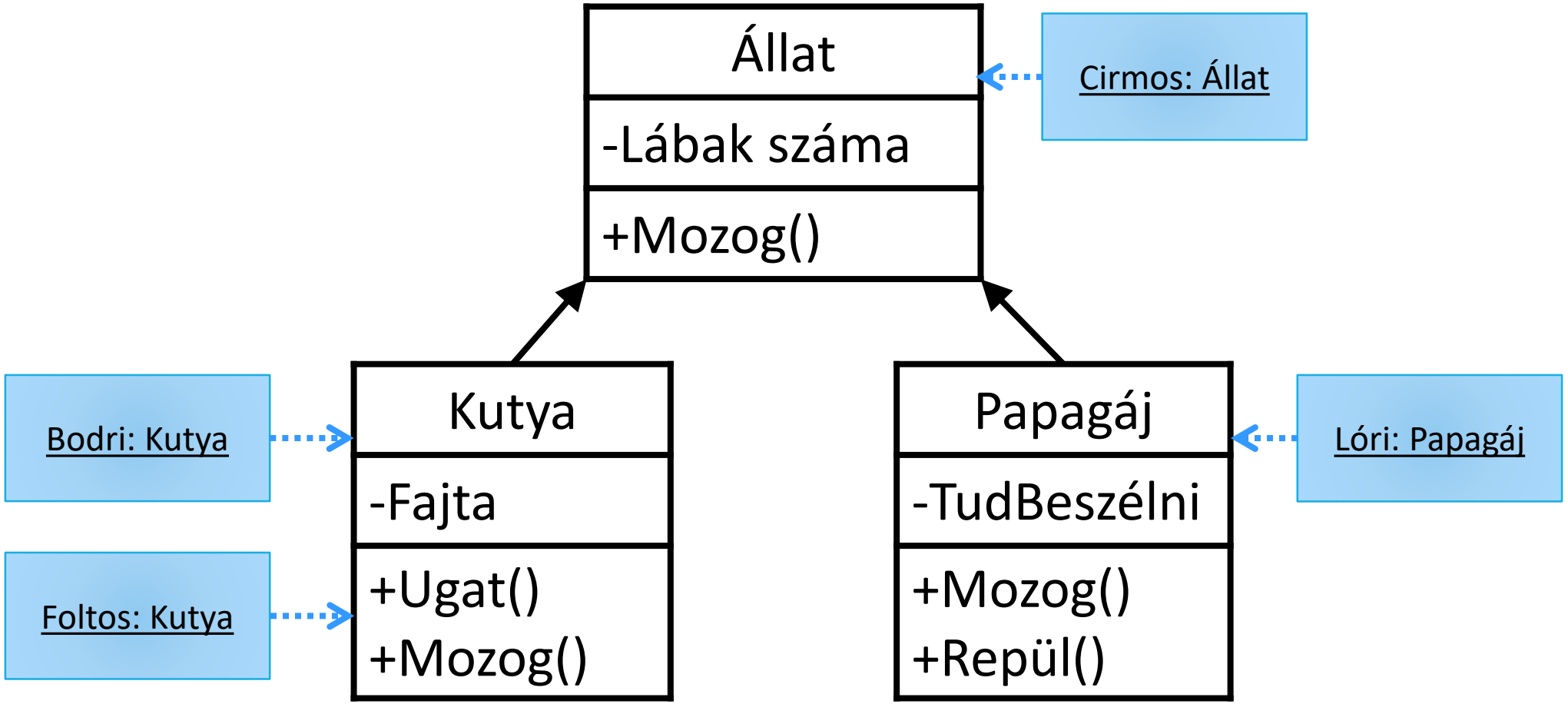
Mit tehet a leszármazott osztály?

- Új tulajdonságokat vezethet be
- Új metódusokat vezethet be
- Felüldefiniálhat, vagy elfedhet már meglévőket
- Új konstruktorokat és destruktort

# Öröklés – IS-A reláció



# Öröklés – IS-A reláció



# Implementáció újrahasznosítás

## Alosztályképzés

- Használd egy típus  implementációját egy másik típus implementálására!
- Gyakran használjuk az őstípus implementációját az altípus implementálására
- A gyakran használt OO programozási nyelvek keverik az altípus és alosztály fogalmakat.

# Felüldefiniálás/Elfedés

## Két különböző jelenség

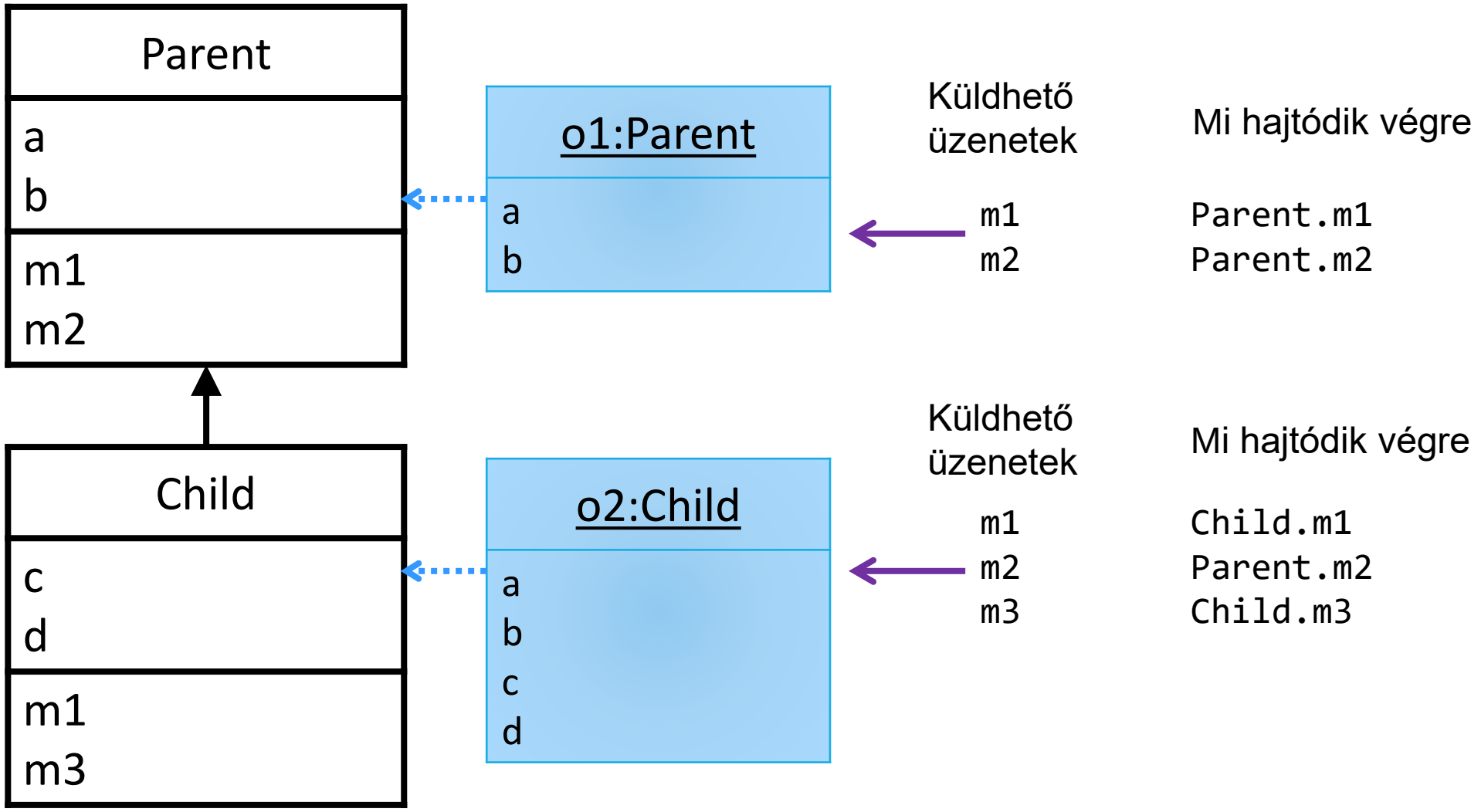
Ami közös a felüldefiniálás (átdefiniálás) illetve elfedés során:

- Az ősosztályban már meglevő, bevezetett függvénynek/üzenetnek ad új definíciót
  - A szignatúra azonos

Ami különböző

- A metóduselérés hogyan történik polimorfizmus esetén
- A két működést kulcsszavakkal lehet szabályozni általában

# Küldhető üzenetek/Elérhető adatok



# Polimorfizmus

Polimorfizmus (többalakúság)

- Az a jelenség, hogy egy azonosító (név: változó, függvény) nem csak egyfajta „objektumra” hivatkozhat.

A polimorfizmusnak több formája is van

- Altípusos
  - A (változó)név egy szupertípusra és leszármazottjaira hivatkozhat
- Nyers – Duck typing
- Parametrikus
  - Lényegében a sablonok
- Ad hoc
  - Lényegében a függvények túlterhelése

# Altípusos polimorfizmus

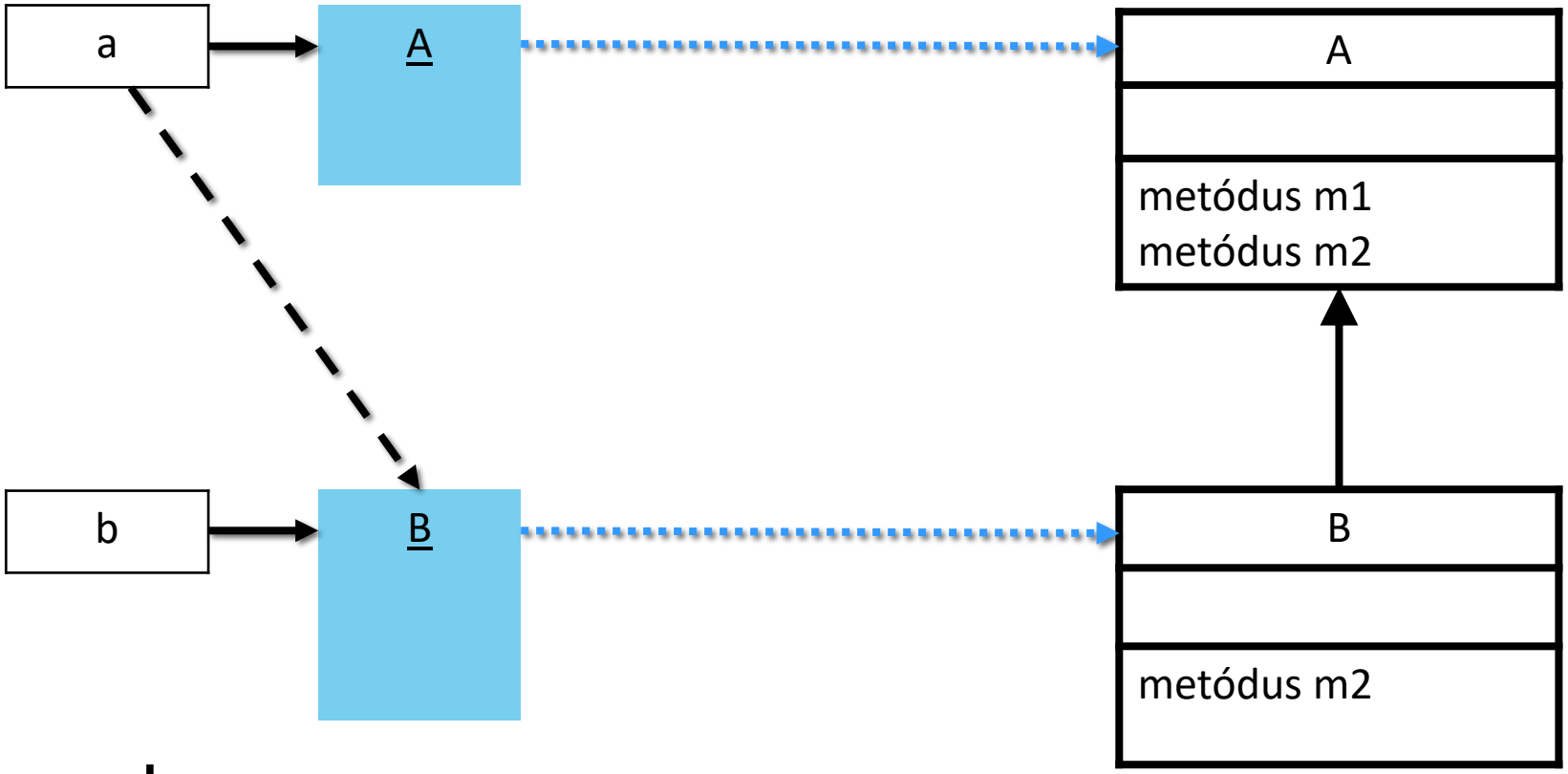
Változó nem csak egyfajta típusú objektumra hivatkozhat, hanem annak altípusaira is

- Statikus típus: a deklaráció során kapja.
- Dinamikus típus: futásidőben, éppen milyen típusú objektumra hivatkozik
  - a statikus típus, vagy annak leszármazottja

`A` típus altípusa `B`, ha a `B` minden olyan helyzetben használható, ahol az `A` is.

- A `Kutya` egyben `Állat` is (IS-A).
- A `Háromszög` az egy `Alakzat`.

# Altípusos polimorfizmus




a:=b

# Altípusos polimorfizmus

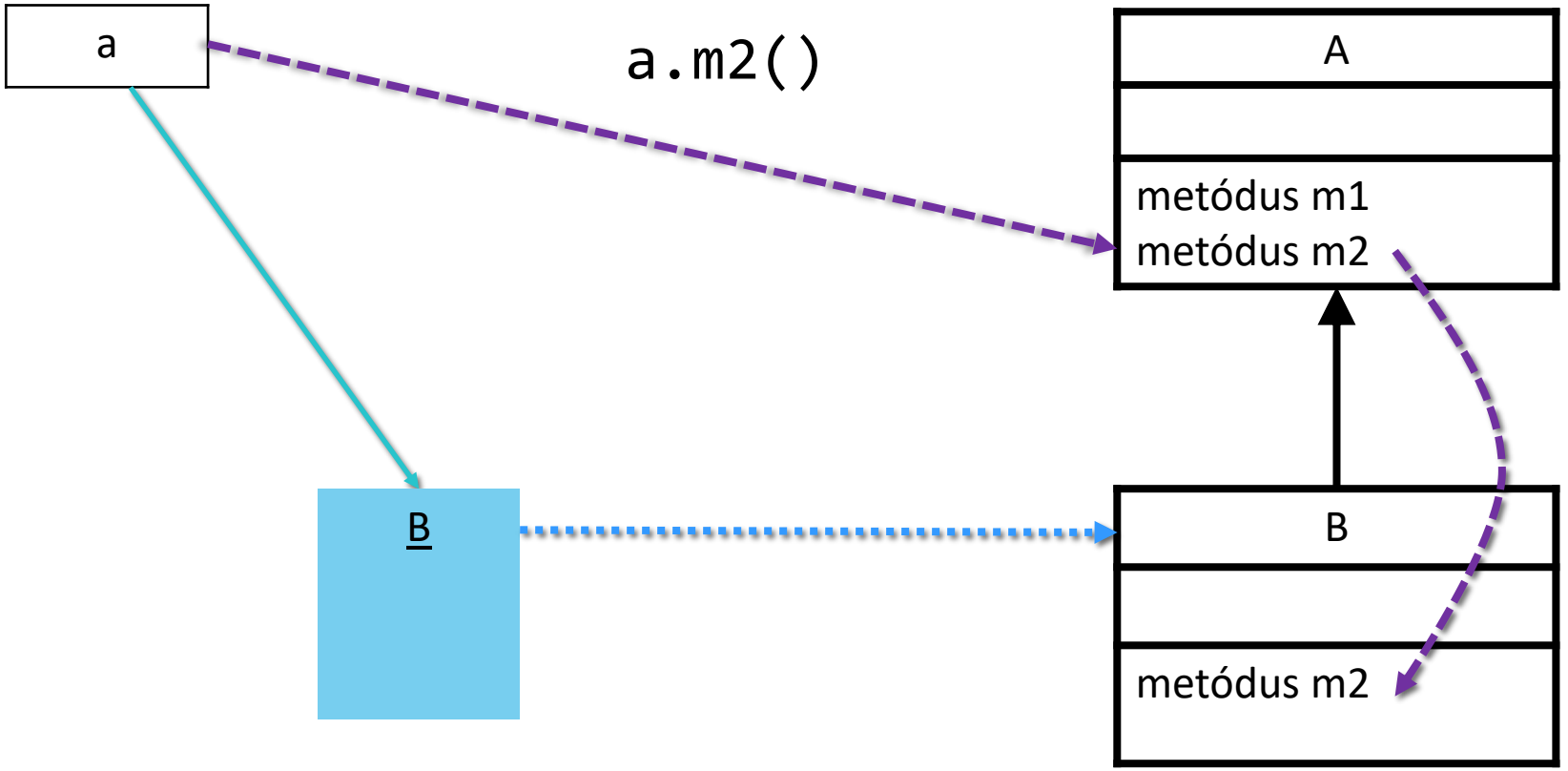
Az előző példában a `a` változó

- Statikus típusa: `A`
- Értékadást követően a dinamikus típusa `B`

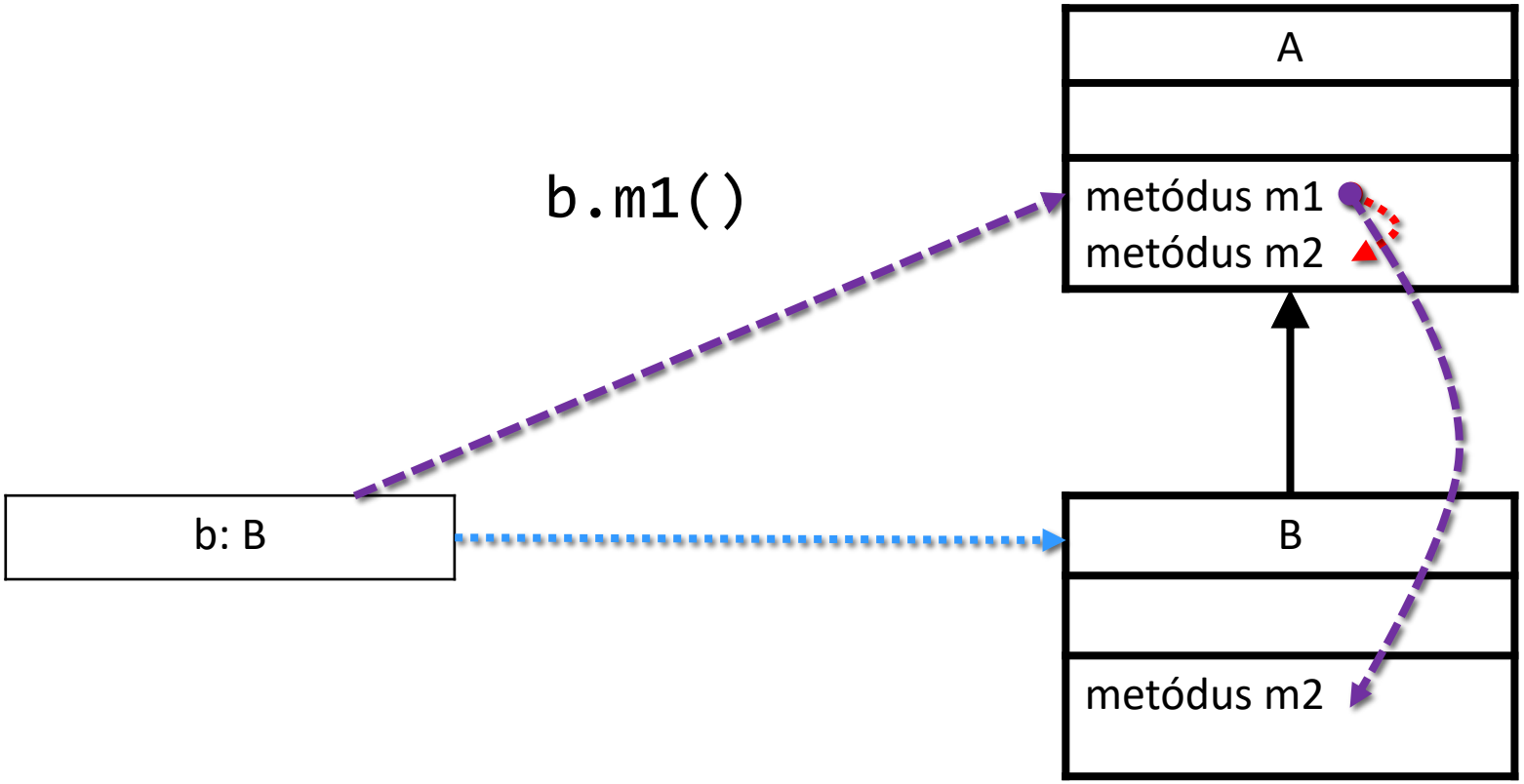
Mi történjen, ha a `B` osztály által felüldefiniált `m2()` függvényt hívjuk az `A` statikus típusú változót használva?

- Egy `B` objektum viselkedése az objektum típusának feleljen meg!
- Ezt a dinamikus összekapcsolás valósítja meg

# Dinamikus összekapcsolás



# Dinamikus összekapcsolás



# Dinamikus összekapcsolás

Run-time fogalom

- Az a jelenség, hogy a változó éppen aktuális dinamikus típusának megfelelő metódus implementáció hajtódik végre
- Felüldefiniált függvények/metódusok esetén történik meg
  - Elfedés esetén nem!

# Altípus

Szabályok:

- Reflexív
  - $T \subseteq T$
- Tranzitív
  - Ha: $T1 \subseteq T2$; $T2 \subseteq T3$
  - Akkor: $T1 \subseteq T3$

# Altípus – alprogramok esetén

Tegyük fel, hogy Triangle $\subseteq$ Shape

- `fss = proc (Shape) returns (Shape)`
- `fts = proc (Triangle) returns (Shape)`
- `fst = proc (Shape) returns (Triangle)`
- `ftt = proc (Triangle) returns (Triangle)`

- `fts` $\subseteq$ `fss` ?
- `fst` $\subseteq$ `fss` ?
- `ftt` $\subseteq$ `fss` ?
- `fss` $\subseteq$ `fts` ?

# Hogyan döntsünk?

Egy függvényt A → B alakban írunk, ha A típusú paramétere van és B típusú eredményt ad.

Ha (A’ → B’) <: (A → B), - altípus

- akkor képesek kell legyünk arra, hogy az első függvénytípus egy elemét használhassuk minden olyan kontextusban, ahol a másodikat.

Tegyük fel, hogy van egy f: A→B típusú függvényünk.

- Ha egy f’ : A’ → B‘ függvényt az f helyén akarunk használni, akkor A-beli argumentumot kell fogadnia és B-beli eredményt adnia.

# Hogyan döntsünk?

Mivel az f' értelmezési tartománya A', így akkor alkalmazható A-beli elemekre, ha A<:A', vagyis

- ha a : A tekinthető mint A'-beli, és f'(a) típus-helyes.

Másrészt, ha az f' eredménye B'-beli, akkor B'<:B garantálja, hogy az eredmény B-beliként kezelhető.

Összegezve:
(A' → B') <: (A → B), ha A<:A' és B'<:B

# Altípus – Megoldás

- `fss = proc (Shape)    returns (Shape)`
- `fts = proc (Triangle) returns (Shape)`
- `fst = proc (Shape)    returns (Triangle)`
- `ftt = proc (Triangle) returns (Triangle)`

~~`fts ⊆ fss`~~ `fst ⊆ fss` ← Az eredmény lehet speciálisabb (monoton = kovariáns)

~~`ftt ⊆ fss`~~ `fss ⊆ fts`

A paraméterek kevésbé speciálisak lehetnek (anti-monoton = kontravariáns)

# Mikor biztonságos az S $\subseteq$ T altípus reláció?

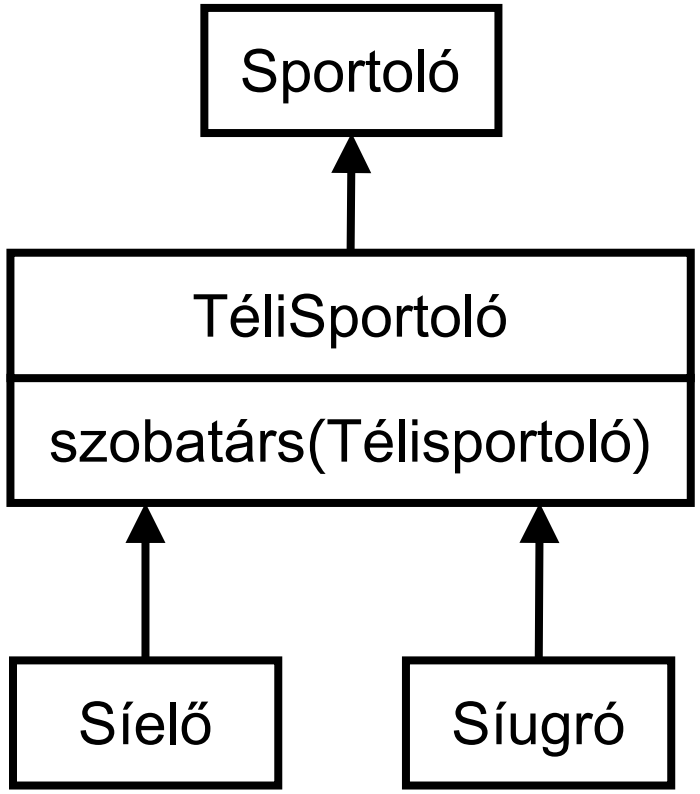
Az altípus metódusai megőrzik az őstípus viselkedését

- Szignatúra
  - Az argumentumok kontravarianciája, az eredmény kovarianciája
  - Az $m_S$ kivételei benne vannak az $m_T$ kivételei között
- Metódusok szabálya
  - Előfeltétel: „kontravariancia – altípus gyengébb”
  - Utófeltétel: „kovariancia – altípus erősebb”

Az altípusok megőrzik a szupertípusok tulajdonságait – típusinvariánsukra:

- „kovariancia – altípus erősebb”

# Kontra/Ko-Variancia




Hogyan tudja `Síugró` felüldefiniálni `szobatárs`-at?

Kovariancia esetén
- `Síelő.szobatárs (Síelő)`
- `Síugró.szobatárs (Síugró)`

Kontravariancia esetén
- `szobatárs (Sportoló)`

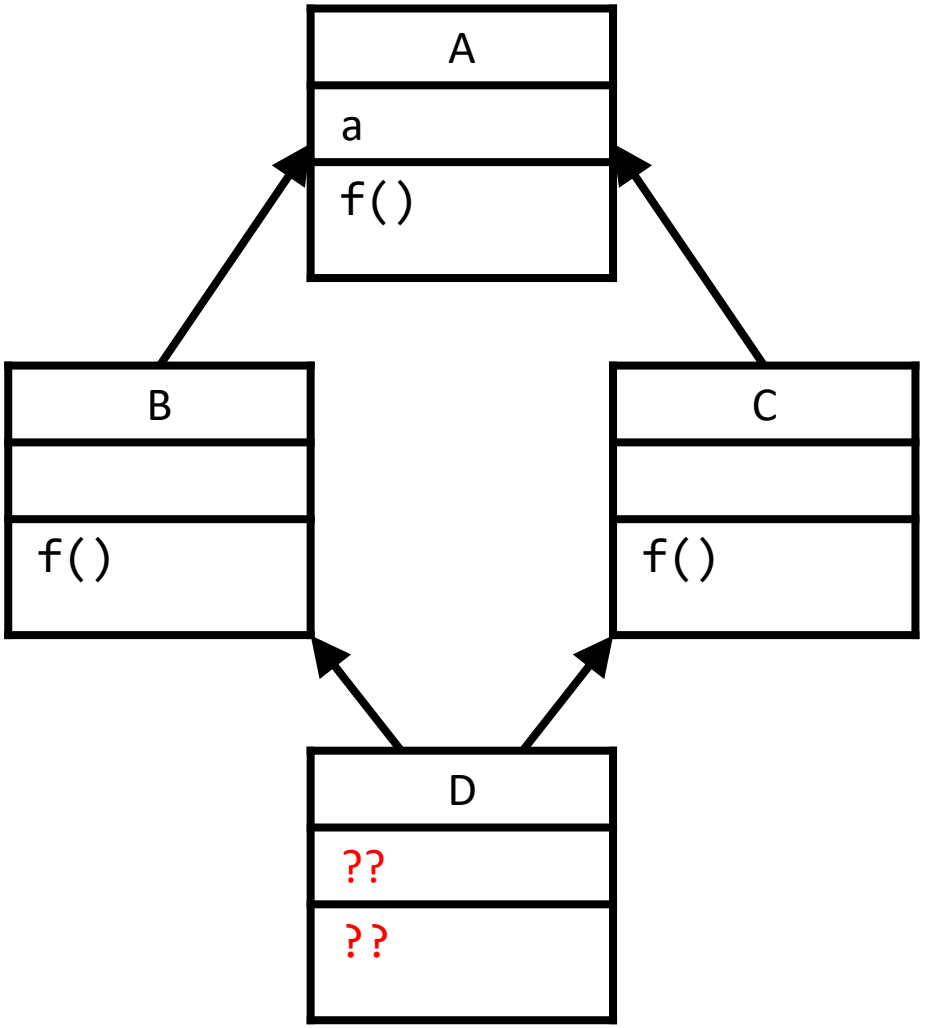
Novariancia esetén
- `szobatárs (TéliSportoló)`

Probléma (kovariáns esetben)
- `t:  TeliSportolo`
- `si: Sielo`
- `sg: Siugro`
- `t := si`
- `t.szobatars(sg)`

# A többszörös öröklődés problémái

Diamond problem – A bázisosztály definiál f függvényt és a attribútumot.

- Ezt B és C osztályok függetlenül származtatják – felüldefiniálják
- D osztály B és C osztályokból közvetlenül származik többszörös örökléssel
- Ekkor
  1. A D-beli f melyiket jelentse?
  2. Az „a” attribútum hány példányban jelenjen meg D-ben?



# A többszörös öröklődés problémái

A két kérdés lényegében ugyanazt a problémát veti fel: ha kétértelműség van, hogyan válasszunk?

- A legtöbb esetben az ilyen kódot nem lehet lefordítani
  - A fordító, vagy a futtató környezet kétértelműségre (*ambiguous*) hivatkozva hibajelzéssel leáll.
- Megoldás
  - Az ősosztály mondja meg, hogy mit szeretne tenni ilyen esetben.
  - A származtatott osztály mondja meg, hogy melyiket szeretné használni.

# Absztrakt osztály

Tervezés eszköze

Egy felső szinten összefogja a közös tulajdonságokat

A metódusok között van olyan, aminek csak specifikációja van, törzse nincs

Nem hozható létre példánya.

A leszármazott teszi konkréttá.

# Interfész

Típusspecifikáció támogatása

„Szolgáltatások”

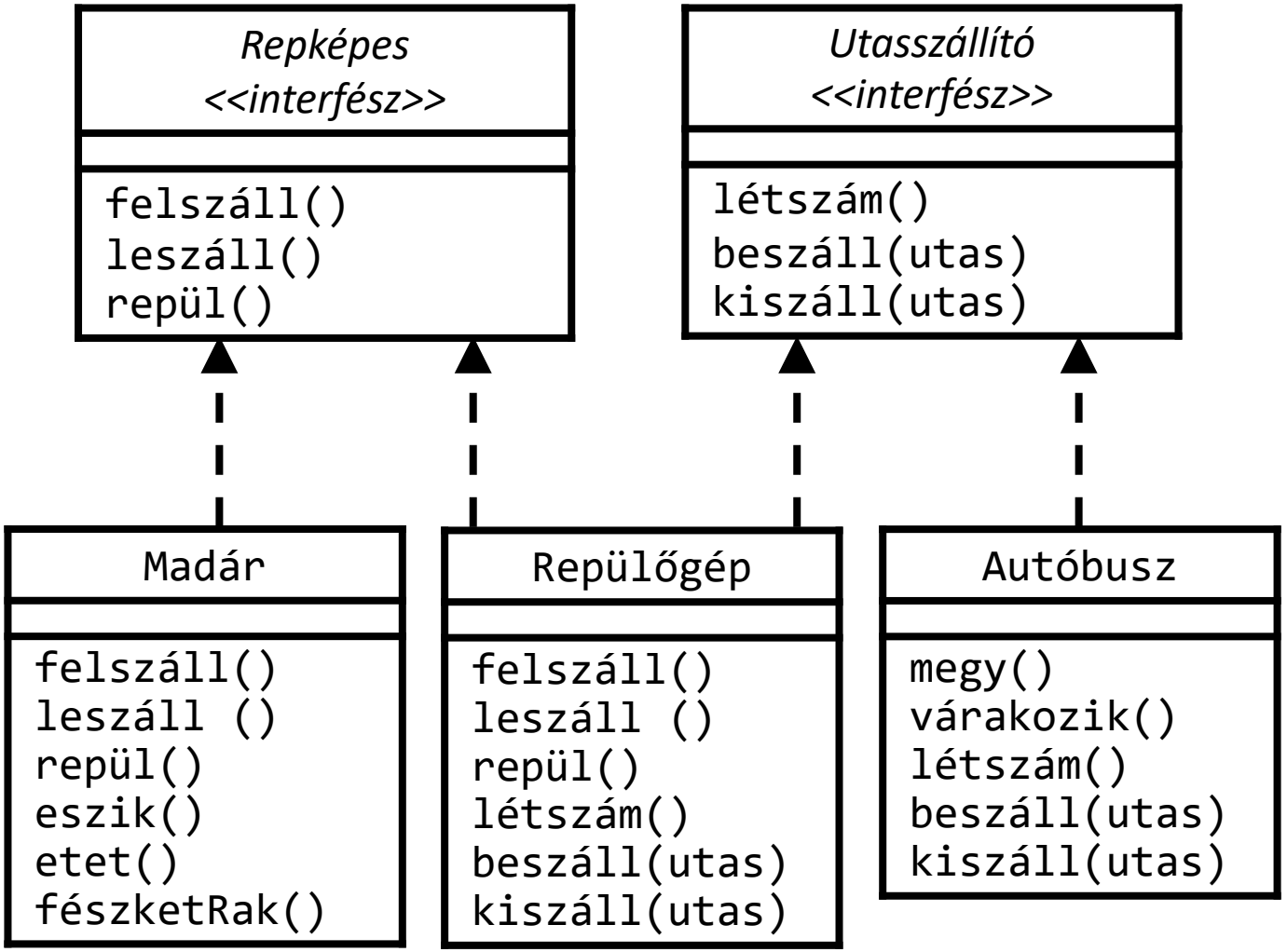
Meg kell valósítani a szolgáltatásait

Többszörös öröklődéssel nincs (annyi) probléma

- összefogja a közös jellemzőket

Interfész ≠ Absztrakt osztály!

# Interfész




**Interfészek**

**Implementáló osztályok**

# Mi az objektumorientált programozás?

A programozó definiálhat altípus kapcsolatokat

A típusszabályok megengedik, hogy az altípus használható legyen a szupertípus helyén

- altípusos polimorfizmus

Típus-vezérelt metódus elérés

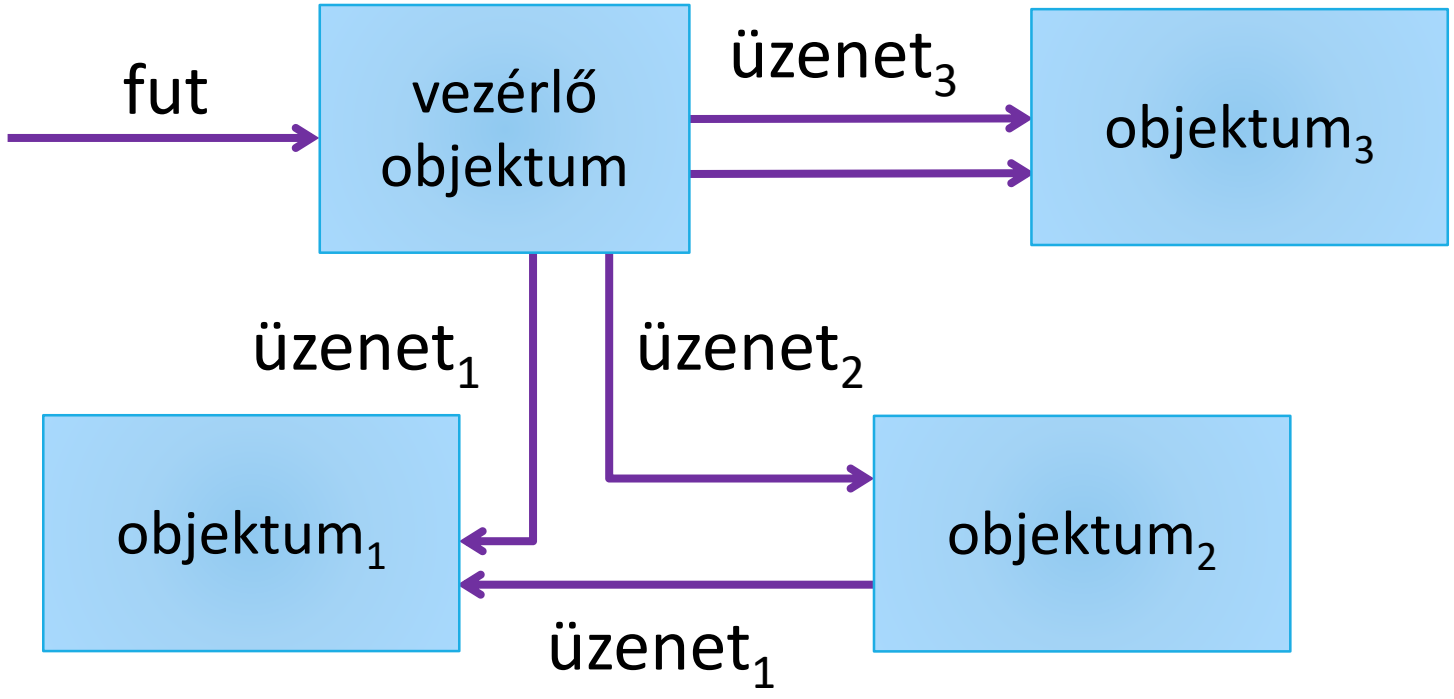
- dinamikus kötés

Implementáció megosztása

- öröklődés

# OO program

Egy objektumorientált program egymással kommunikáló objektumok összessége, melyben minden objektumnak megvan a feladatköre



# Osztály, példány

Minden objektum?

Akkor az osztályok is …

- Lehet belső állapotuk,
- Küldhetünk üzeneteket neki …
- Minek az objektuma?
  - Metaosztály
    - Singleton objektum
    - És a metaosztály is objektum?

# Kérdések

Van-e öröklődés?

Van-e többszörös öröklődés?

Adattagok és metódusok elrejtése hogy van megoldva?

Támogatja-e a polimorfizmust és dinamikus összekapcsolást?

- Esetleg csak mutatók használatával?

Van-e alapértelmezett ősosztály?

- Object?

# Kérdések

Van-e absztrakt osztály?

- Nem példányosítható közvetlenül?

Konstruktor

Destruktor?

- Garbage Collector?

Standard objektum könyvtárak?

Osztályszintű attribútumok és metódusok?

# C++

# Osztálydefiníció

A C++ **`struct`** és **`class`** sok szempontból hasonló

- OOP elvárások tekintetében a **`class`** használata szükséges

Példa egy egyszerű osztály definíciójára

- Osztálydefiníciók egymásba ágyazhatók

```
class Rectangle {
    double a_side, b_side;
public:
    void print() const {  }
};
```

Példányosítás

```
Rectangle rect(10, 2);
Rectangle *rectpt = &rect;
```

# Konstruktorok

Konstruktorok: „nincs objektum konstruktor nélkül”, ha kell, implicit hívódnak

- Neve megegyezik az osztály nevével
- Ha már megadtunk egy konstruktort, akkor default konstruktor nem definiálódik
- A default konstruktor meghívja az attribútumok konstruktorát, de a beépített típusokat nem inicializálja (konzisztensen a C-vel)
- A konstruktornak nem lehet visszatérési értéke
- C++11 óta lehetséges
  - Mezők inicializálása
  - Konstruktor delegáció
  - Szülő osztály konstruktorának örökítése

# Konstruktorok

```
class Rectangle {
    double a_side, b_side;
public:
    Rectangle(double a, double b): a_side(a), b_side(b) {};
    Rectangle() : Rectangle(1.0, 1.0) {};

    void print() const {
        std::cout << "Rectangle " << a_side << " " <<
                                     b_side << std::endl;
    }

    double calculateArea() {
        return a_side * b_side;
    }
};
```

# Destruktor

Téglalap esetén nincs kifejezett haszna, de egy általános sokszögnél (illetve példa az inicializációra)

```
class Polygon {
    double *sides {nullptr};
    int nsides {0};

public:
    Polygon(double *data, int ndata): nsides(ndata) {
        sides=new double[nsides];
        std::memcpy(sides, data, sizeof sides);
    }
    ~Polygon() { delete [] sides; }

    void print() const {
        std::cout << "Polygon, number of sides " << nsides
                                                << std::endl;
    }
};
```

# Osztályváltozó, osztálymetódus

```
class Circle {
    double radius;
    static double PI;
public:
    Circle(double r): radius(r) {};
    Circle() : Circle(1.0) {};

    void print() const {
        std::cout << "Circle " << radius << std::endl;
    }
};

double Circle::PI = 3.1415926535;
```

Az osztályon kívül definiálni kell!

# Osztályváltozó, osztálymetódus

```
class Circle {
 //...
 double calculateArea() {
        return radius * radius * PI;
    }
 //...
};
```

Az osztályon belül elérhető közvetlenül, itt a külső használathoz publikus metódus kell!

```
static double getPi() { return Circle::PI; }
```

# Osztályváltozó, osztálymetódus

Használat:

```
Circle::PI; // HIBA
Circle::getPi();
Circle c1(10);
c1.getPi();
```

Szemantikailag helytelen példányon keresztül elérni, azaz:

- ~~`c1.getPi();`~~
- ~~`return radius * radius * PI;`~~

Helyette

- `return radius * radius * Circle::PI;`

# Láthatóság

Az adattagok és metódusok elrejtése megoldott

A láthatóság minősítője C++-ban

- **`public`**
  - a külső felhasználók elérik
- **`protected`**
  - csak a leszármazottak érhetik el
- **`private`**
  - csak az adott osztály és a barátok (**`friend`**) számára elérhető – alapértelmezés

# Öröklődés

A téglalap egy specializációja a négyzet:

```
class Square: public Rectangle {
public:
    Square(double a): Rectangle(a, a) { };
    Square(): Square(1) { };

    void print() const {
        std::cout << "Square " << a_side << std::endl;
    }
};
```

Azonban ez nem működik, mert ugyan van saját `a_side` attribútuma a négyzetnek, de azt nem éri el közvetlenül, mivel az ősben ez **`private`**.

# Öröklődés

Módosítsuk!

```
class Rectangle {
protected:
    double a_side, b_side;
 //...
};
```

Ekkor

```
rect.print();    // Rectangle 10 2
Square s(5);
s.print();       // Square 5
```

# Konstruktor öröklés

Lehetséges C++11 óta

```
class BaseClass {
public:
    BaseClass(int value);
};

class DerivedClass : public BaseClass {
public:
    using BaseClass::BaseClass;
};
```

# Altípusos polimorfizmus

Tekintsük az alábbi kódot

```
Rectangle rect(10, 2);
Square s(5);
rect = s;
```

Mi történik?

```
rect.print();
```

- Fontos, hogy itt még nincsen dinamikus kötés!

# Altípusos polimorfizmus

Tekintsük az alábbi kódot

```
Rectangle rect(10, 2);
Square s(5);
rect = s;
```

Mi történik?

```
rect.print();
```

- Fontos, hogy itt még nincsen dinamikus kötés!

# Altípusos polimorfizmus

Ha B (pl. `Square`) altípusa az A (pl. `Rectangle`) típusnak, akkor B objektumainak referenciái értékül adhatók az A típus referenciáinak.

Azaz:

```
Rectangle* prect = new Rectangle(4, 3);
Square* psqu = new Square(2);

prect->print();
psqu->print();
prect = psqu;
```

Mi történik?

```
prect->print();
```

# Dinamikus összekapcsolás

C++ nyelven a dinamikus kötéshez:

- Az **őstípus** `virtual`-nak deklarálja a metódust, amire megengedi a felüldefiniálást
- Ellenkező esetben elfedés lesz!
- Tipikusan az ősbeli **`virtual`-**t a leszármazottban is **`virtual`-**nak jelöljük.

# Dinamikus összekapcsolás

Módosítás a `Rectangle` osztályban

```
virtual void print() const {
    std::cout << "Rectangle " << a_side << " "
                              << b_side << std::endl;
}
```

Ekkor az előző példa

```
prect->print();
// Square 2
```

# Felüldefiniálás jelzése

Expliciten jelezhetjük, hogy felüldefiniálást szeretnénk

```
class Square: public Rectangle {
    //...
    virtual void print() const override {//...
};
```

- Ellenőrizni fogja a fordító ekkor
- Hibát kapunk ha felüldefiniálandó függvény
  - nem **`virtual`**
  - **`final`**

Felüldefinálás (és leszármazás) megakadályozása

```
virtual void print() const final {//...
```

# Virtuális destruktor

A destruktor esetén is fontos a **`virtual`** használata

- Amennyiben az altípusos polimorfizmus használjuk.

Nézzük az alakzatot

```
class Shape {
public:
    virtual void print() const {
        std::cout << "Generic Shape" << std::endl;
    }
};
```

# Virtuális destruktor

Ha a Polygon ebből származik le

```
class Polygon: public Shape {
  //...
  ~Polygon() {
    delete [] sides;
    std::cout<< "Deleted" << std::endl;
  }
};
```

Akkor a következő eset memóriaszivárgás

```
Shape *sh = new Polygon(new double[2]{10.0, 10.3}, 2);
delete sh;
std::cout << "^^ Memory leak ^^ " << std::endl;
```

# Virtuális destruktor

Helyesen

```
class Shape {
public:
    virtual ~Shape() {};
    virtual void print() const {
        std::cout << "Generic Shape" << std::endl;
    }
};
```

Így már rendben lesz

```
Shape *sh = new Polygon(new double[2]{10.0, 10.3}, 2);
delete sh;
std::cout << "^^ No memory leak ^^ " << std::endl;
```

# Automatikusan létrehozott függvények

A fordító több tagfüggvényt létrehoz, ha nem definiáljuk azokat. Dönthetünk az automatikusan létrejövő tagfüggvények

- létrehozásáról (**`default`**)
- elutasításáról (**`delete`**)

```
class Square {
public:
    Square() = default;
    Square(const Square&) = default;
    Square& operator=(const Car&) = delete;
    virtual ~Square() = default;
};
```

Az utolsó sornál fontos a **`virtual`**.

- Ezzel jelezzük, hogy automatikusan létrejövő destruktor virtuális is legyen!

# Absztrakt osztály

Pure virtual függvények bevezetése

```
class Shape {
 public:
      virtual ~Shape() {};
      virtual void print() = 0;
      virtual double calculateArea() = 0;
 };
```

# Absztrakt osztály

Ilyenkor a leszármazott adja meg a definíciót:

```
class Circle : public Shape {
    double radius;
public:
    Circle(double r): radius(r) {};

    virtual void print() const override {
        std::cout << "Circle radius: " << radius << std::endl;
    }
    virtual double calculateArea() const override {
        return radius * radius * PI;
    }
};
```

# Absztrakt osztály típusként

Természetesen nem példányosítható:

```
Shape s;
```

De statikus típusa egy változónak lehet:

```
Circle c;
Shape * pa = &c;
```

De statikus típusa egy változónak lehet és dinamikus kötés működik

```
pa->calculateArea(); // Circle::calculateArea()
```

# Többszörös öröklés

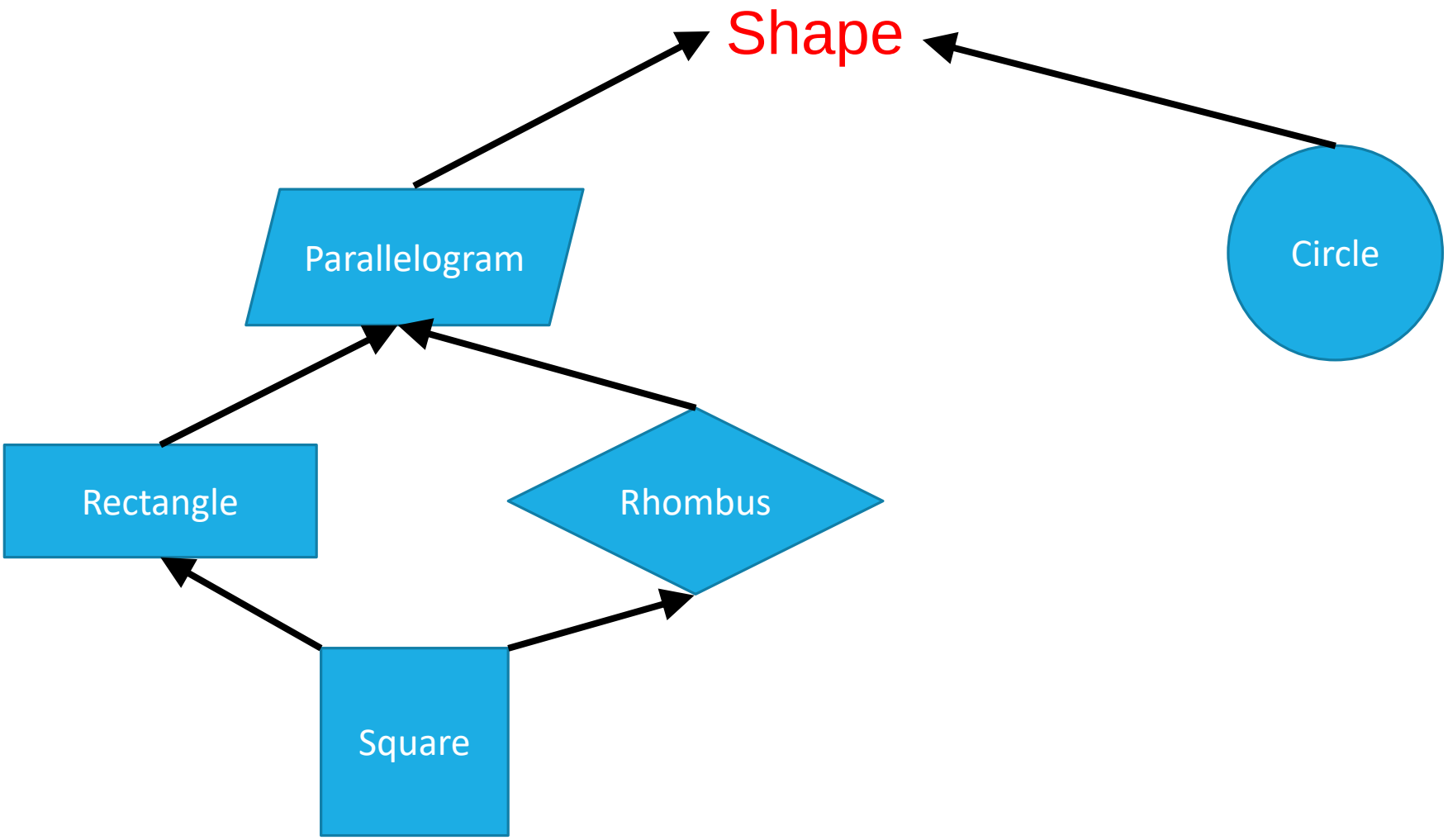
Az osztályhierarchia lehet fa, de lehet általánosabb körmentes gráf is:

- `class Shape`
- `class Circle: public Shape`
- `class Parallelogram: public Shape`
- `class Rhombus: public Parallelogram`
- `class Rectangle: public Parallelogram`
- `class Square: public Parallelogram, public Rhombus`

Problémák:

- Adattagok hányszor?
- Melyik metódus?

# Többszörös öröklés



# Többszörös öröklés

```
double PI = 3.1415926535;

class Shape {
public:
    virtual ~Shape() {};
    virtual void print() const = 0;
    virtual double calculateArea() const = 0;
};

class Circle : public Shape {
    double radius;
public:
    Circle(double r): radius(r) {};

    virtual void print() const override {
        std::cout << "Circle radius: " << radius << std::endl;
    }
    virtual double calculateArea() const override {
        return radius * radius * PI;
    }
};
```

# Többszörös öröklés

```
class Parallelogram : public Shape {
protected:
    double _side_a;
    double _side_b;
    double _delta;
    double _height_a;
public:
    virtual double get_a_side() const final { return _side_a; }
    virtual double get_b_side() const final { return _side_b; }
    virtual double get_angle()  const { return _delta;  }
    virtual bool is_equilateral() const { return _side_a == _side_b; }
    virtual bool is_right() const { return _delta == 90; }

    Parallelogram(double a, double b, double angle): _side_a(a), _side_b(b),
                                                     _delta(angle) {
        _height_a = b * std::sin(_delta*PI / 180.0);
    }

    virtual void print() const override {
        std::cout << "Parallelogram a, b, delta: " << _side_a << ", "
                  << _side_b << ", "
                  << _delta << std::endl;
    }

    virtual double calculateArea() const override {
        return _side_a * _height_a;
    }
};
```

# Többszörös öröklés

```
class Rhombus : public Parallelogram {
public:
    Rhombus(double a, double angle): Parallelogram(a, a, angle) {  };

    virtual void print() const override {
        std::cout << "Rhombus side, angle " << _side_a << ", " << _delta << std::endl;
    }

    virtual double calculateArea() const override {
        return _side_a * _side_a * std::sin(_delta*PI / 180.0);
    }

    virtual bool is_equilateral() const override final { return true; }
};

class Rectangle : public Parallelogram {
public:
    Rectangle(double a, double b): Parallelogram(a, b, 90) { _height_a = b; };

    virtual void print() const override {
        std::cout << "Rectangle " << _side_a << ", " << _side_b << std::endl;
    }

    virtual bool is_right() const override final { return true; }
};
```

# Többszörös öröklés

```
class Square: public Rectangle, public Rhombus {
public:
    Square(double a): Rectangle(a,a), Rhombus(a, 90) { };

    virtual void print() const override final {
        std::cout << "Square " << _side_a << std::endl;
    }

    virtual double calculateArea() const override final {
        return _side_a * _side_a;
    }
};

int main() {
    Rectangle *s = new Square(10);
    std::cout << s->calculateArea() << std::endl;
    s->print();

    return 0;
}
```

# Többszörös öröklés

Ebben a példában a `Square` esetén a `_side_a` mezőre hibát jelez a fordító

- A mezőt két ágon örökli, el kell dönteni, hogy hogyan legyen

1. Lehetséges megadni, például

```
virtual double calculateArea() const override final {
    return Rectangle::_side_a * Rectangle::_side_a;
}
```

- Ez azonban nem oldja meg a következőt

```
Parallelogram *s = new Square(10);
```

# Többszörös öröklés

Ebben a példában a `Square` esetén a `_side_a` mezőre hibát jelez a fordító

- A mezőt két ágon örökli, el kell dönteni, hogy hogyan legyen

2. Virtuális öröklés

- Ilyenkor a virtuális ősosztály nem két irányban lesz ős, hanem egy közös ős
- Gondoskodni kell a virtuális ősök inicializációjáról is

```
class Rhombus: public virtual Parallelogram
class Rectangle: public virtual Parallelogram
Square(double a): Parallelogram(a, a, 90),
                  Rhombus(a, 90), Rectangle(a,a) { };
```

- Így már jó

```
Parallelogram *s = new Square(10);
```

# Többszörös öröklés függvény példa

```
class A{
public:
    virtual void f() { std::cout << "A.f()" << std::endl; };
};

class B : public A {
public:
    void f() { std::cout << "B.f()" << std::endl; };
};

class C : public A{
public:
    void f() { std::cout << "C.f()" << std::endl; };
};

class D : public B, public C {
public:
    void g() { std::cout << "D.g()" << std::endl; };
};
```

# Többszörös öröklés függvény példa

Használatkor

- `D d;`
  `d.f();`
- Hibaüzenet: `request for member „f” is ambiguous`

Itt is döntés kell

- `using C::f;`
- Azonban ilyenkor!
- `B *b = new D();`
  `b->f();  // B.f()`

# Friend

A barát hozzáfér a privát adatokhoz, függvényekhez

- Tipikusan olyan operátorok készítésére használjuk, amelyek a modellezés szerint nem részei a típusnak, de szorosan kötődnek hozzá
  - Kiírás

```
class Point {
    friend ostream &operator<<( ostream &, const Point &);
public:
    Point( int = 0, int = 0 );
    void setPoint( int, int );
    int getX() const { return x; }
    int getY() const { return y; }
protected:
    int x, y;
};
```

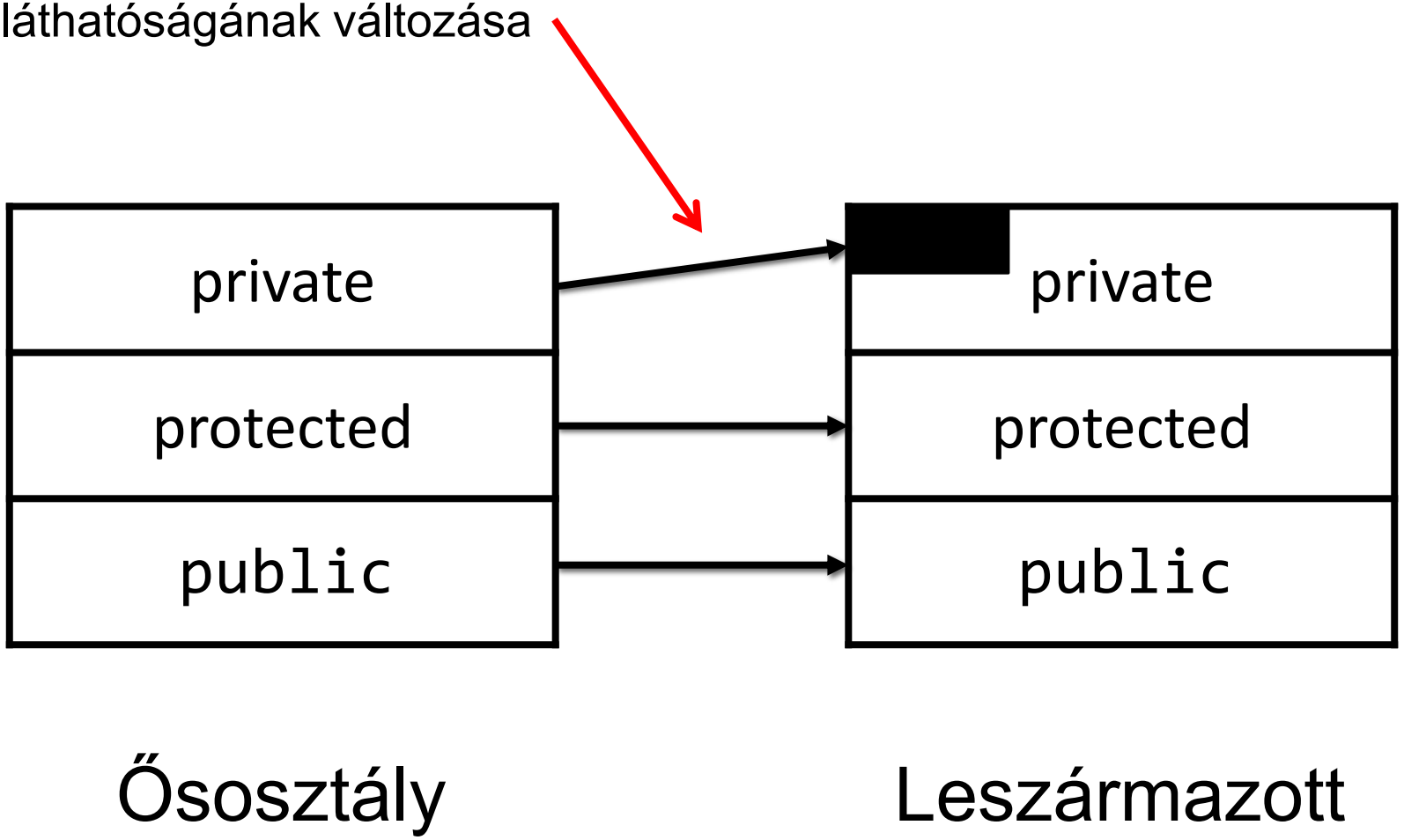
# Alosztályképzés

C++ - implementációs öröklés altípus létrehozása nélkül

- **`private`**, **`protected`** öröklés

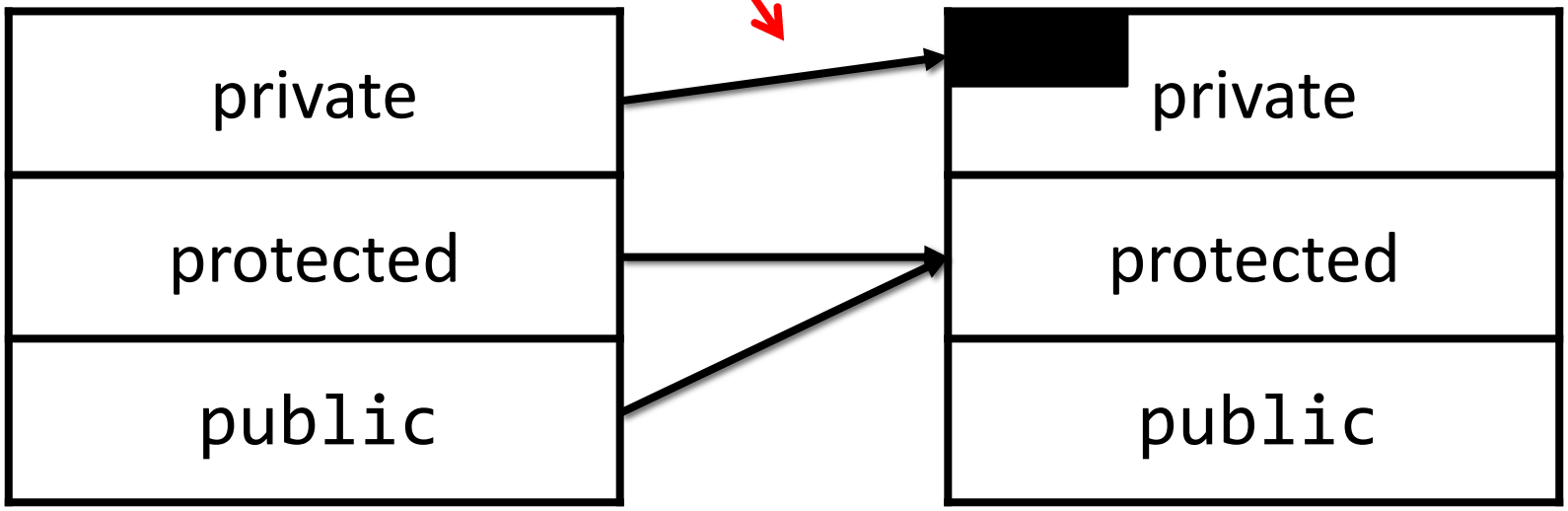
# Public öröklés

Tagok láthatóságának változása



# Protected öröklés

Tagok láthatóságának változása

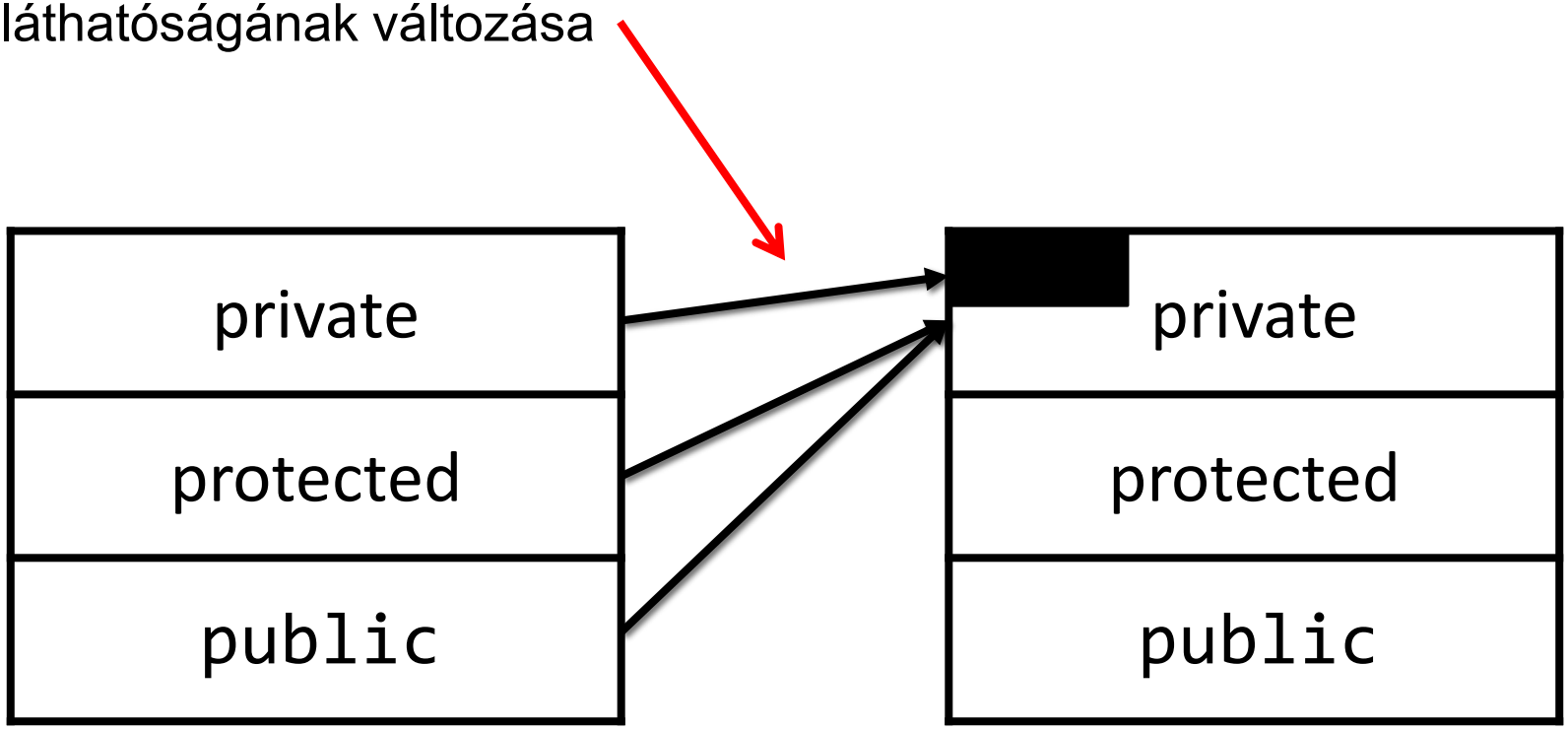



Bázisosztály

Származtatott osztály

# Private öröklés

Tagok láthatóságának változása




Bázisosztály

Származtatott osztály

# Kovariáns metódusok

Lehet bevezetni kovariáns és kontravariáns metódusokat az altípusban

- ezek elfedik (túlterhelik) az eredeti metódust, nem felüldefiniálják

```
class A {
public:
    int data;
};

class B : public A {
public:
    virtual void f(B *p)  { std::cout << "B.f(B) " << this->data << " " << p->data << std::endl; };
};

class C : public B {
public:
    // Kontravariáns
    // void f(A *p)  { std::cout << "C.f(A) " << this->data << " " << p->data << std::endl; };

    // Novariáns
    // void f(B *p) override { std::cout << "C.f(B) " << this->data << " " << p->data << std::endl;
};

    // Kovariáns
    void f(C *p)  { std::cout << "C.f(C) " << this->data << " " << p->data << std::endl; };
};
```

# Kovariáns metódusok

Az osztálydefiníciókkal a következő kód nem fordul

- ```
  A *a = new A(); a->data=1;
  B *b = new B(); b->data=10;
  C *c = new C(); c->data=100;
  C *test = c;
  test->f(b);
  ```

A problémát az okozza, hogy a C osztályban van egy f() implementáció, amely C paramétert vár.

- Ezt a fordító megtalálja, ám a szignatúra nem megfelelő.
- Nem keres tovább, mert az elfedés miatt
- Megoldás: `using B::f;`
  ```

# Kis kitérő – `struct`

Struct esetén minden tag **`public`** alapértelmezés szerint.

- Ez az információ elrejtést nem segíti, úgy mint az osztályoknál
- Azonban célszerű „sima” adatrekordok tárolására (rekord típus)
  - Plain Old Data Structure (POD)
  - C-vel kompatibilis

# C++ – Összefoglalás

Mit örököl a leszármazott?

- Adattagokat
- Metódusokat

Mit nem örököl a leszármazott?

- Ősosztály konstruktorait, destruktorát
  - Konstruktor örökíthető
  - Leszármazottban delegálható rá, használható
- Ősosztály értékadás operátorát
- Ősosztály barátait

# C++ – Összefoglalás

Mit vezethet be a leszármazott osztály?

- Új adattagokat
- Új metódusokat
- Felüldefiniálhat már meglévőket (virtual)
- Új konstruktorokat és destruktort
- Új barátokat

# C++

Egy általános metódus deklarációja a következőket jelenti:

1. A metódus elérheti a privát mezőket is
2. Az osztály scope-ját használja
3. A metódus egy konkrét objektumra hívódik meg, ezért birtokolja a „this” pointert

Statikus metódus csak az 1, 2 -vel rendelkezik,

Ha egy függvényt friend-nek deklarálunk, akkor csak az 1. jogunk lesz (friend mechanizmus)

# OOP folytatás

KÖVETKEZŐ ALKALOMMAL




# Programozási nyelvek és módszerek

## BEVEZETÉS

# Fontos

## Oktató

- Tornai Kálmán
- 232-es szoba
- tornai.kalman@itk.ppke.hu

## Tárgyadatok

- Programozási nyelvek és módszerek
- P-ITSZT-0007
- P-ITSZT-0045
- Hétfő: 10.15-13.00
  - Neumann ea.
  - Zárva van órákon kívül, kint kell várakozni.

## Követelmények

- Jegy: Szóbeli vizsga
  - Vizsga előfeltétele az érvényes (>1) gyakorlati jegy a Java programozási nyelv tárgyból!
- Az előadások látogatása legalább 80%-ban kötelező
  - Maximum két hiányzás!
- A félév során két hétfő oktatási szünet
  - Tavaszi szünet

# Irodalom

## Kötelező irodalom (részek)

- Sebesta, Robert W., Concepts of programming languages, Edinburgh, Pearson, 2013, ISBN 978-0-273-76910-1
- Nyékyné Gaizler Judit et al., (szerk.), Programozási nyelvek, Budapest, Kiskapu, 2003

## Ajánlott irodalom

- SCOTT, Michael L., Programming Language Pragmatics, Amsterdam, Morgan Kaufmann, 2009
- Comparative programming languages, Harlow, Addison-Wesley, 2001

# Tematika

- Bevezetés
- A programozási nyelvek elemei
- Beépített adattípusok, változók, kifejezések
- Utasítások
- Alprogramok és paramétereik
- Absztrakt adattípusok
- Sablonok
- Kivételek
- Objektum-orientált programozás
- Helyesség
- Párhuzamosság
- A könyvtártervezés elvei
- A funkcionális programozás alapjai

# A félév célja

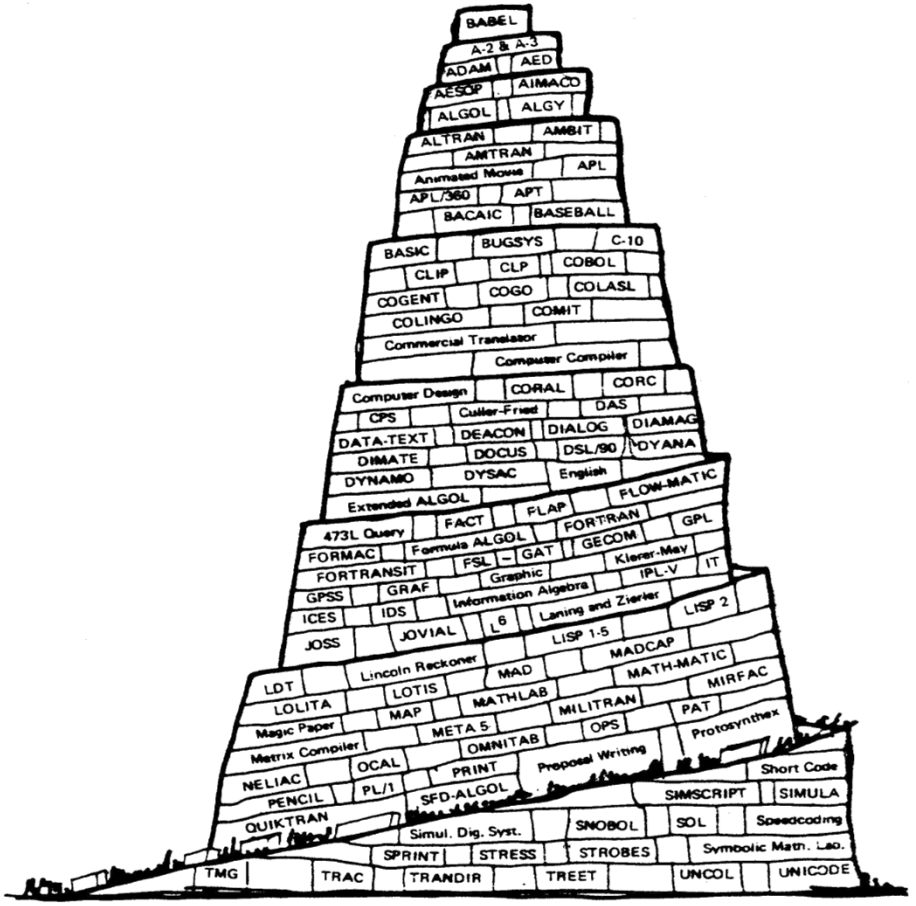
Programozási módszertan alapjai

Programozási nyelvek eszközkészlete

- Melyik nyelvet válasszuk egy feladatra?
- Milyen lehetőségek vannak?
- Milyen nyelvi stílusok vannak?
  - Ne ragadjunk bele a C++/Java logikájába
- Alapok későbbi programozási paradigmák, nyelvek elsajátítására
  - Mik a népszerű nyelvek most?
  - Mik lesznek?
    - Van még bőven idő a nyugdíjig …

# Ráhangolódás

# A nyelvek – Bábel tornya




Tower of Babel

# Mother Tongues

## Tracing the roots of computer languages through the ages

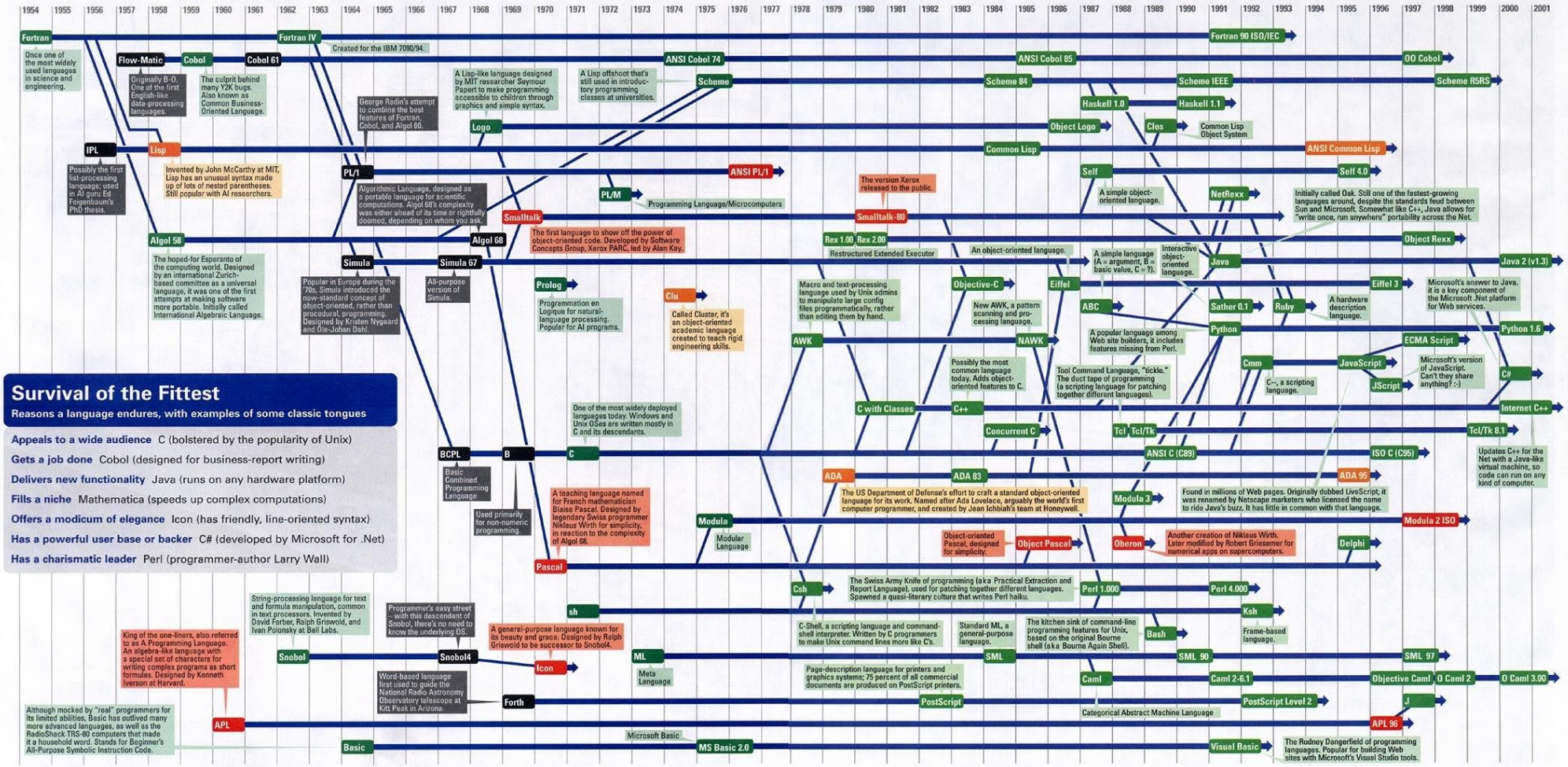
**Just like** half of the world's spoken tongues, most of the 2,300-plus computer programming languages are either endangered or extinct. As powerhouses C/C++, Visual Basic, Cobol, Java, and other modern source codes dominate our systems, hundreds of older languages are running out of life.

An ad hoc collection of engineers – electronic lexicographers, if you will – aim to save, or at least document, the lingo of classic software. They're combing the globe's 9 million developers in search of coders still fluent in these nearly forgotten lingua francas. Among the most endangered are Ada, APL, B (the predecessor of C), Lisp, Oberon, Smalltalk, and Simula.

Code-raker Grady Booch, Rational Software's chief scientist, is working with the Computer History Museum in Silicon Valley to record and, in some cases, maintain languages by writing new compilers so our ever-changing hardware can grok the code. Why bother? "They tell us about the state of software practice, the minds of their inventors, and the technical, social, and economic forces that shaped history at the time," Booch explains. "They'll provide the raw material for software archaeologists, historians, and developers to learn what worked, what was brilliant, and what was an utter failure." Here's a peek at the strongest branches of programming's family tree. For a nearly exhaustive rundown, check out the Language List at *www.informatik.uni-freiburg.de/Java/misc/lang_list.html*. **– Michael Menduno**

**Key**

- 1954 **Year Introduced**
- **Active:** thousands of users
- **Protected:** taught at universities; compilers available
- **Endangered:** usage dropping off
- **Extinct:** no known active users or up-to-date compilers
- **Lineage continues**



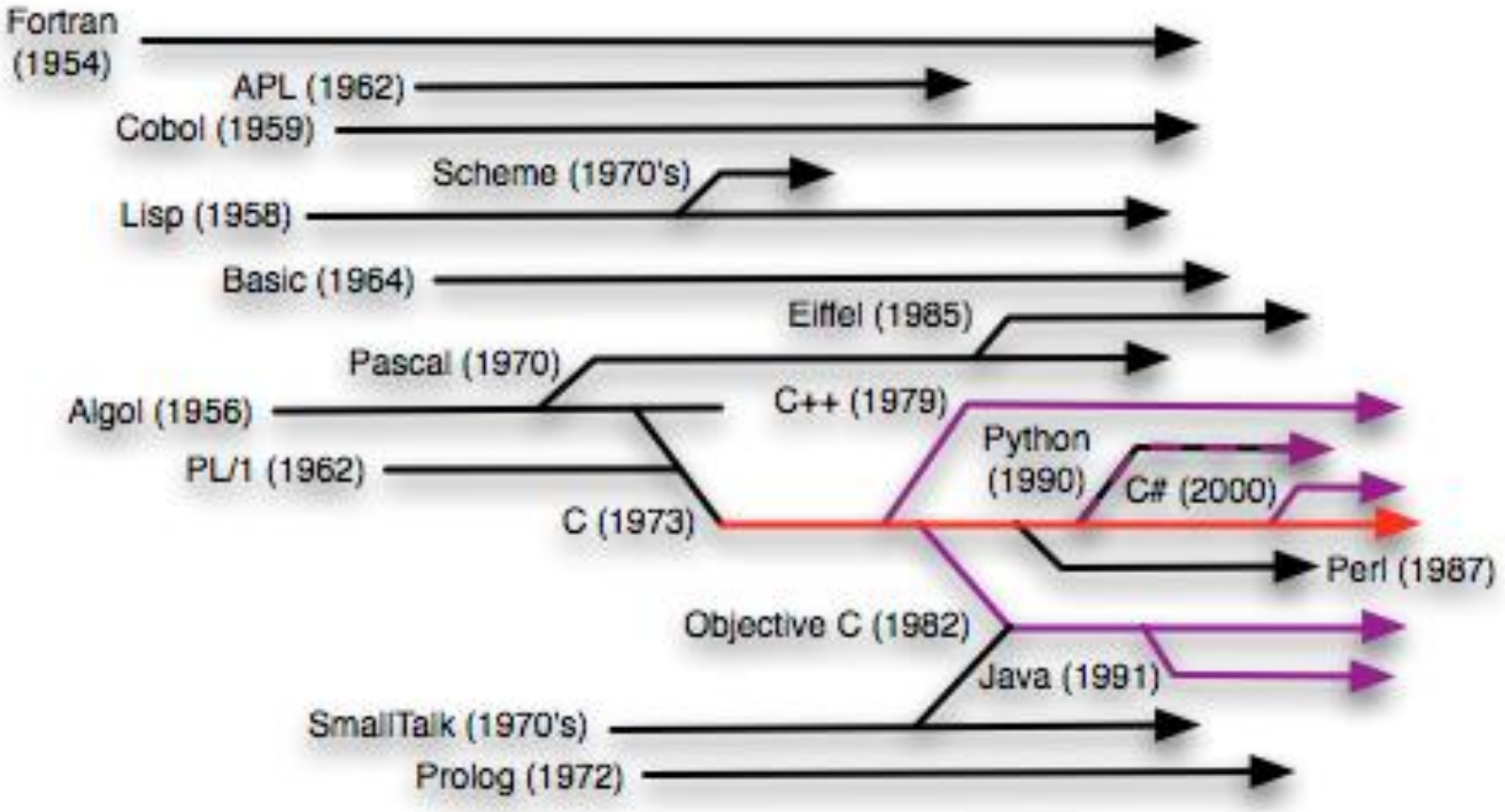

## Survival of the Fittest

**Reasons a language endures, with examples of some classic tongues**

- **Appeals to a wide audience** C (bolstered by the popularity of Unix)
- **Gets a job done** Cobol (designed for business-report writing)
- **Delivers new functionality** Java (runs on any hardware platform)
- **Fills a niche** Mathematica (speeds up complex computations)
- **Offers a modicum of elegance** Icon (has friendly, line-oriented syntax)
- **Has a powerful user base or backer** C# (developed by Microsoft for .Net)
- **Has a charismatic leader** Perl (programmer-author Larry Wall)

Sources: Paul Boutin; Brent Hailpern, associate director of computer science at IBM Research; The Retrocomputing Museum; Todd Proebsting, senior researcher at Microsoft; Gio Wiederhold, computer scientist, Stanford University

# „Fontosabb” nyelvek családfája



# Miért van ennyi?

Számos ok

- Evolúció
  - Újabb nyelvi konstrukciók, megközelítések
  - Pl: OOP, szkriptnyelvek
- Speciális célok
  - C: alacsonyszintű programozás
  - Awk: karakter alapú feldolgozás
  - Python: könnyű prototipizálás, számítások
- Személyes preferenciák
  - Rekurziós algoritmusok <-> Iteratív algoritmusok
  - Pointerek használata <-> automatikus (implicit) referencing

# Mitől sikeres egy nyelv?

Több ok

- Kifejező erő
  - Gyakori érv bizonyos nyelvek mellett / ellen, hogy „erős”
  - Matematikai értelemben nem igaz, mindegyik nyelv azonos kifejezőerővel bír, minden algoritmus leírható benne
    - Lásd később
    - A különbség a szintaktikai támogatottság
      - Le lehet írni, más kérdés, hogy mennyire bonyolultan
- Kezdők számára könnyű használat
- Könnyű implementálhatóság
  - Fordítóprogram, környezet
- Szabványosság
  - „Nyelvjárásoktól mentes”

# Mitől sikeres egy nyelv?

Több ok

- Nyílt forrás
  - Manapság szinte bármely nyelvre igaz, hogy van nyílt forrású implementáció
- Jó fordítóprogram
  - Hibamentes
  - Jó optimalizálás
    - Gyors kódot állít elő
  - Könnyű használhatóság
    - Toolchain

# Történelmi mérföldkövek

FORTRAN (1950-es évek közepe-vége)

- hangsúly a tudományos programozáson

LISP (1950-es évek vége)

- rekurzió, dinamikus helyfoglalás és felszabadítás

ALGOL 60 (1960)

- az adattípus fogalma
- érték- és név szerinti paraméterátadás

COBOL (1960)

- angol-szerű szintaxis

# Történelmi mérföldkövek

BASIC (1960-as évek közepe)
- Interpretált

PL/I (1960-as évek közepe)
- Általános célú
- Párhuzamosan végrehajtódó taszkok
- Kezeli a futási idejű kivételeket
- Pointerek mint adattípusok

APL (1960 körül)
- Tömb és mátrix kezelés

SNOBOL (1960-as évek közepe)
- Szövegkezelés, szövegminták hasonlítására

# Történelmi mérföldkövek

SIMULA 67 (1967)
- Osztály fogalmának bevezetése – egységbezárás
- Öröklődés

ALGOL 68 (1968)
- A legtöbb ezt követő programozás nyelv merített a tervezéséből

Pascal (1970-es évek eleje)
- Programozás oktatásra

C (1970-es évek eleje)
- rendszerprogramozásra

Prolog (1970-es évek közepe)
- Logikai nyelv

# Történelmi mérföldkövek

Ada (1970-es évek vége)

- A Pentagon megbízásából
- Kivételkezelés
- Sablonok
- Párhuzamosság támogatása (taszkok, randevúk)
- Helyességellenőrzés (2012-től)

Smalltalk (1980)

- Objektumorientált – máshogy

C++ (1980-as évek eleje óta)

- SIMULA 67 és a Smalltalk + C

# Történelmi mérföldkövek

Objective C (1980-as évek eleje)
- C és Smalltalk

Eiffel (1980-as évek eleje)
- Objektumorientált
- Tervezés szerződéssel a függvényhívó és a függvény között

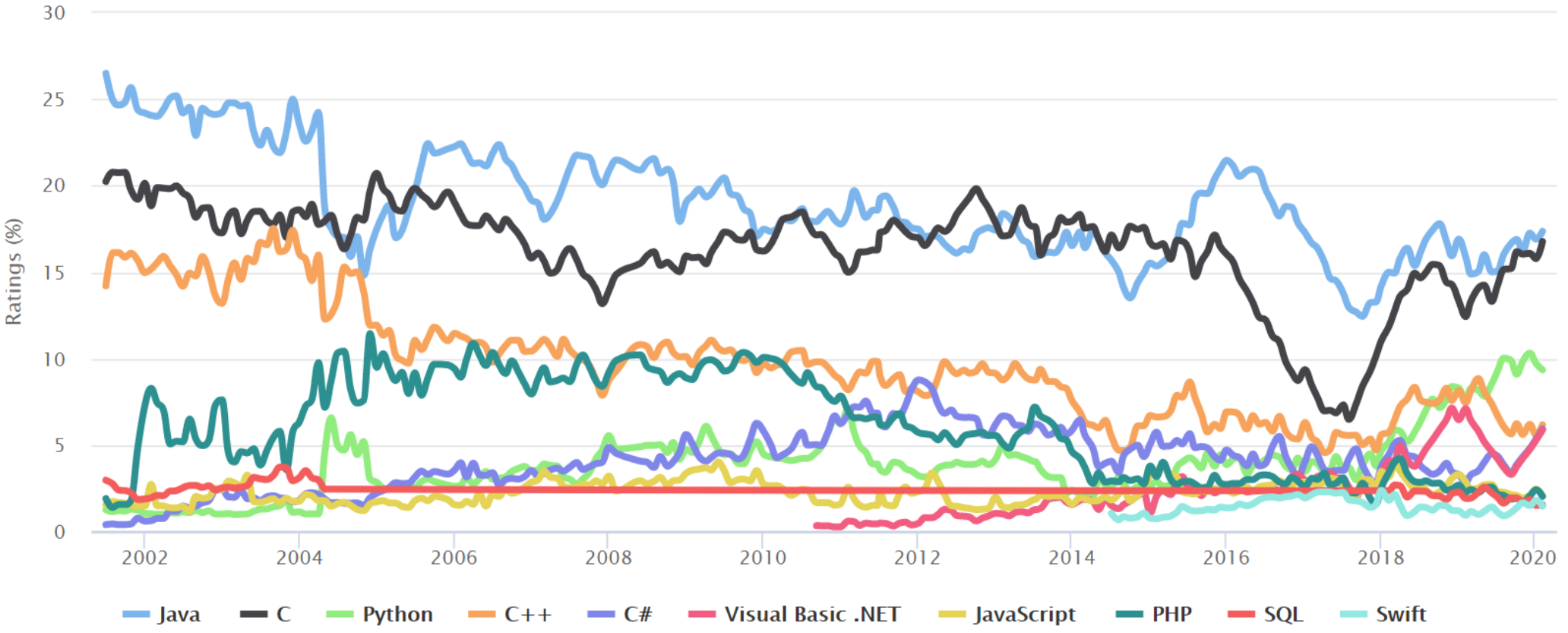
Java (1990-es évek közepe)
- OOP
- Párhuzamosság támogatása
- GC
- Kivételkezelés

C#
- „Microsoft Java-ja”
- .NET alapú fejlesztés támogatása

# Mitől fontos egy programozási nyelv?

HTTPS://WWW.TIOBE.COM/TIOBE-INDEX/
HTTPS://SPECTRUM.IEEE.ORG/COMPUTING/SOFTWARE/THE-TOP-PROGRAMMING-LANGUAGES-2019

# TIOBE Index


Ratings (%)
30
25
20
15
10
5
0
2002
2004
2006
2008
2010
2012
2014
2016
2018
2020
Java
C
Python
C++
C#
Visual Basic .NET
JavaScript
PHP
SQL
Swift

# TIOBE Index

| 2020 | 2019 | 2018 | 2017 | 2016 | 2015 | 2014 | 2013 | 2012 | 2011 | 2010 | 2009 | 2008 | Nyelv |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 2 | 2 | 1 | 2 | 1 | 1 | 1 | 1 | Java |
| 2 | 2 | 2 | 2 | 2 | 1 | 1 | 2 | 1 | 2 | 2 | 2 | 2 | C |
| 3 | 3 | 4 | 5 | 5 | 8 | 8 | 7 | 8 | 8 | 7 | 8 | 6 | Python |
| 4 | 4 | 3 | 3 | 3 | 3 | 4 | 4 | 4 | 3 | 3 | 4 | 3 | C++ |
| 5 | 7 | 5 | 4 | 4 | 5 | 5 | 5 | 5 | 4 | 6 | 7 | 8 | C# |
| 6 | 5 | 7 | 6 | 7 | 9 | 10 | 12 | 15 | 39 | | | | Visual Basic .NET |
| 7 | 6 | 6 | 7 | 9 | 6 | 9 | 11 | 11 | 10 | 11 | 9 | 9 | JavaScript |
| 8 | 8 | 9 | 10 | 6 | 7 | 6 | 6 | 6 | 5 | 4 | 3 | 5 | PHP |
| 9 | 9 | | | | | | | | | | | | SQL |
| 10 | 15 | 12 | 14 | 16 | | | | | | | | | SWIFT |

# TIOBE Index

| 2020 | 2019 | 2018 | 2017 | 2016 | 2015 | 2014 | 2013 | 2012 | 2011 | 2010 | 2009 | 2008 | Nyelv |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 11 | 16 | 19 | 13 | | | | | | | | | | Go |
| 12 | 14 | 15 | 9 | 13 | | | | | | | | | Assembly |
| 13 | 12 | 8 | 16 | 17 | 18 | 44 | | | | | | | R |
| 14 | 23 | | | | | | | | | | 12 | | D |
| 15 | 18 | 11 | 12 | 11 | 20 | 12 | 10 | 10 | 11 | 10 | 10 | 11 | Ruby |
| 16 | 11 | 18 | 19 | 18 | 17 | 14 | 18 | 20 | 22 | | | | MATLAB |
| 17 | 19 | 20 | 20 | 19 | 13 | 15 | 22 | 19 | 17 | 20 | 14 | 13 | PL/SQL |
| 18 | 17 | 13 | 11 | 10 | 11 | 20 | 15 | 12 | 12 | 12 | 11 | 10 | Delphi / Object Pascal |
| 19 | 13 | 10 | 8 | 11 | 12 | 13 | 9 | 9 | 9 | 9 | 6 | 7 | Perl |
| 20 | 10 | 16 | 18 | 14 | 4 | 3 | 3 | 3 | 6 | 8 | 18 | 42 | Objective-C |

# TIOBE hosszútávú összehasonlítás

| Nyelv | 2020 | 2015 | 2010 | 2005 | 2000 | 1995 | 1990 | 1985 |
|---|---|---|---|---|---|---|---|---|
| Java | 1 | 2 | 1 | 2 | 3 | - | - | - |
| C | 2 | 1 | 2 | 1 | 1 | 2 | 1 | 1 |
| Python | 3 | 7 | 6 | 6 | 22 | 21 | - | - |
| C++ | 4 | 4 | 4 | 3 | 2 | 1 | 2 | 12 |
| C# | 5 | 5 | 5 | 8 | 8 | - | - | - |
| Visual Basic .NET | 6 | 10 | - | - | - | - | - | - |
| JavaScript | 7 | 8 | 8 | 9 | 6 | - | - | - |
| PHP | 8 | 6 | 3 | 4 | 27 | - | - | - |
| SQL | 9 | - | - | 97 | - | - | - | - |
| Objective-C | 10 | 3 | 21 | 37 | - | - | - | - |
| Lisp | 31 | 18 | 16 | 13 | 14 | 5 | 3 | 2 |
| Ada | 35 | 29 | 24 | 15 | 15 | 6 | 4 | 3 |
| Pascal | 229 | 16 | 13 | 65 | 11 | 3 | 15 | 5 |

# IEEE Rank – 2019

| Rank | Language | Type | Score |
|---|---|---|---|
| 1 | Python | web, desktop, embedded | 100.0 |
| 2 | Java | web, mobile, desktop | 96.3 |
| 3 | C | mobile, desktop, embedded | 94.4 |
| 4 | C++ | mobile, desktop, embedded | 87.5 |
| 5 | R | desktop | 81.5 |
| 6 | JavaScript | web | 79.4 |
| 7 | C# | web, mobile, desktop, embedded | 74.5 |
| 8 | Matlab | desktop | 70.6 |
| 9 | Swift | mobile, desktop | 69.1 |
| 10 | Go | web, desktop | 68.0 |

# Miért tanuljunk (tovább) programozási nyelveket?

Egyben a tárgy célja is

- Megérteni a nyelvekben rejlő további funkcionalitást
  - C++ union típuskonstrukció
  - Változó számú argumentumok
- Alternatív módon kifejezni megoldásokat
  - Választott nyelvnek megfelelően
    - Matlab: mátrix műveletek
    - Python: list comprehensing
      - `x = [2 ** x for x in range(10) if x > 5]`
- Hibakeresők, assemblerek, …
  - Használatuk, céljuk megértése

# Miért tanuljunk (tovább) programozási nyelveket?

Egyben a tárgy célja is

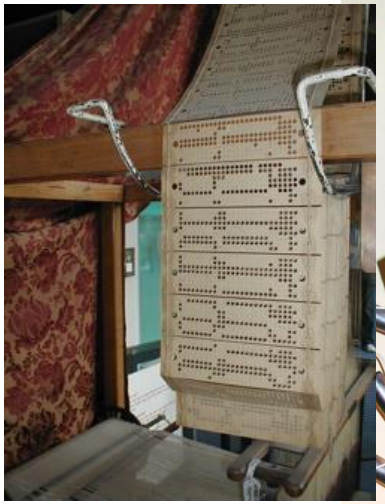
- Hatékony, hasznos nyelvi eszközök „átültetése”
  - Megvalósítás / Szimulálás más nyelven
- Szélesíteni a látókört a programnyelvek nyelvi eszközkészletét illetően
  - További konstrukciók
  - Megfelelő eszköz kiválasztása
  - Nem megrettenni egy új nyelvtől
    - Stratégia a megtanulásra

# Szoftverminőség

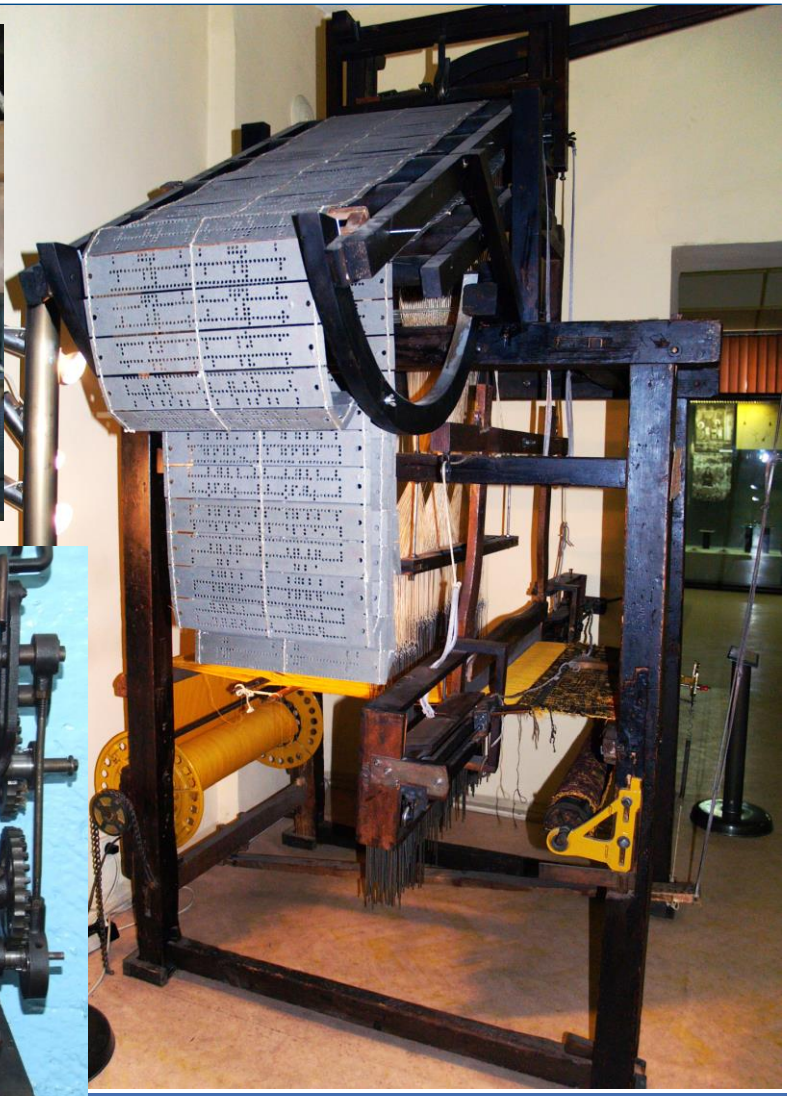
# Miért készítünk programokat?



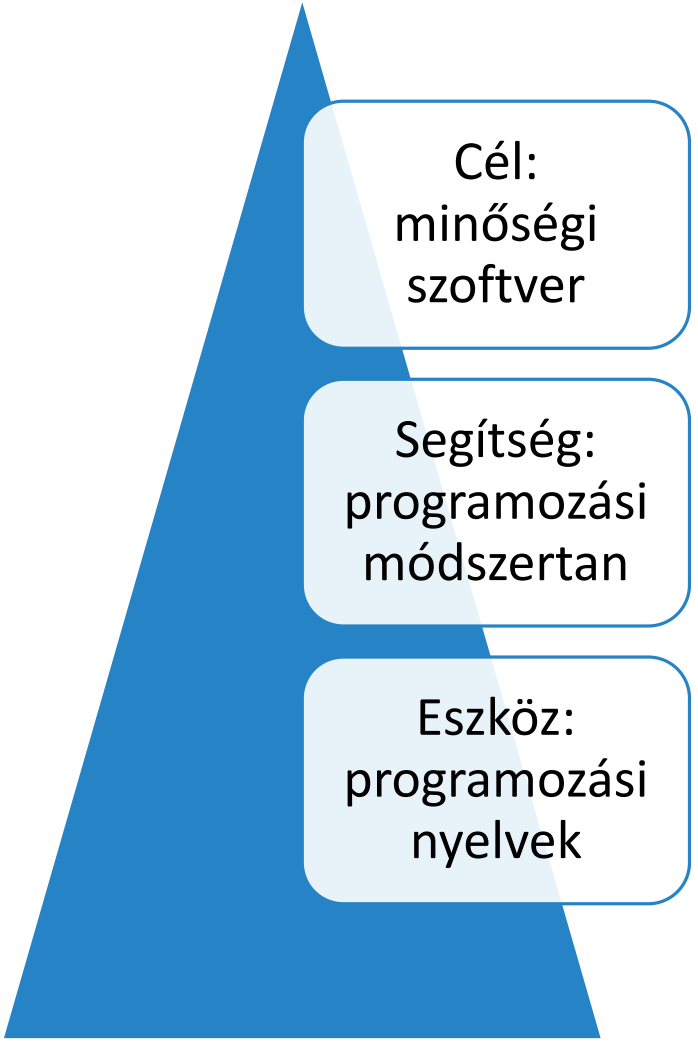
Joseph Marie Jacquard
(szövőgép)







# Minőségi szoftver




Szempontok

- Helyesség
- Megbízhatóság
- Karbantarthatóság
- Újrafelhasználhatóság
- Kompatibilitás
- Hordozhatóság
- Hatékonyság
- stb.

# Minőségi szoftver

## Helyesség

- A program helyes, ha pontosan megoldja a feladatot, megfelel a kívánt specifikációnak.
  - Ehhez szükséges a pontos és minél teljesebb specifikáció.

## Megbízhatóság

- A program megbízható, ha
  - *Helyes*, és
  - Abnormális helyzetekben sem történik katasztrófa
    - Valamilyen „ésszerű" működést produkál
    - Abnormális: a specifikációban nem leírt

# Minőségi szoftver

## Karbantarthatóság

- A karbantarthatóság annak mérőszáma, hogy milyen könnyű a programterméket a *specifikáció változtatásához* adaptálni.
  - Egyes felmérések szerint a szoftver költségek 70 %-át szoftver karbantartásra fordítják!
- Növelésének két alapelve
  - Tervezési egyszerűség
  - Decentralizáció – minél önállóbb modulok létrehozása

## Újrafelhasználhatóság

- A szoftver termék, vagy komponens egy másik, új alkalmazásban felhasználható.
  - Ez vonatkozik a szoftvertermék
    - Egészére
    - Részére

# Minőségi szoftver

## Kompatibilitás

- A kompatibilitás megmutatja, hogy mennyire könnyű a szoftver termékeket egymással kombinálni.
  - Nem légüres térben fejlesztünk!

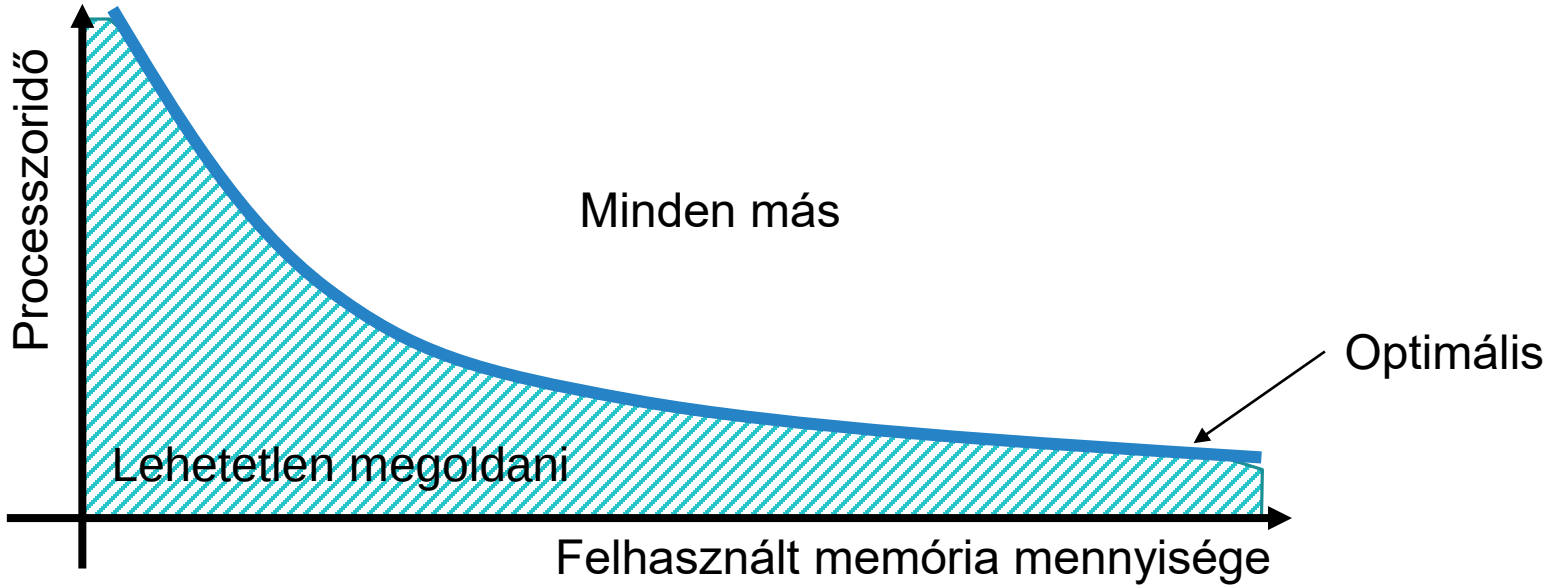
## Hordozhatóság

- A program hordozhatóságának mértéke mutatja meg, mennyire könnyű a programot átalakítani
  - Más géphez
  - Más konfigurációhoz
  - Más operációs rendszerhez
  - Általában más környezethez

# Minőségi szoftver

## Hatékonyság

- A program hatékonysága a futási idővel és a felhasznált memória méretével, illetve *további erőforrások* használatával arányos.
  - Egyszerűsítve: egy program hatékony, ha gyorsan lefut és kevés memóriát használ.



# Minőségi szoftver

## Barátságosság

- A program emberközelisége, barátságossága a felhasználó számára rendkívül fontos
  - Megköveteli, hogy legyen
    - az input logikus és egyszerű
    - az eredmények formája áttekinthető

## Tesztelhetőség

- A tesztelhetőség, áttekinthetőség a program karbantartói, fejlesztői számára fontos.

# Modularitás

# Programozási módszertanok

## Moduláris tervezés

- Programjainkat egymástól minél inkább független, de jól definiált kapcsolatban álló programegységek segítségével tervezzük.

## Paradigmák

- Procedurális
- Objektumorientált
- Funkcionális
- Logikai
- Szimbolikus
- …

# Moduláris dekompozíció

Jelentése

- A feladat több egyszerűbb részfeladatra bontása
- A részfeladatok megoldása egymástól függetlenül elvégezhető
- Ennek segítségével csökkentjük a feladat bonyolultságát.

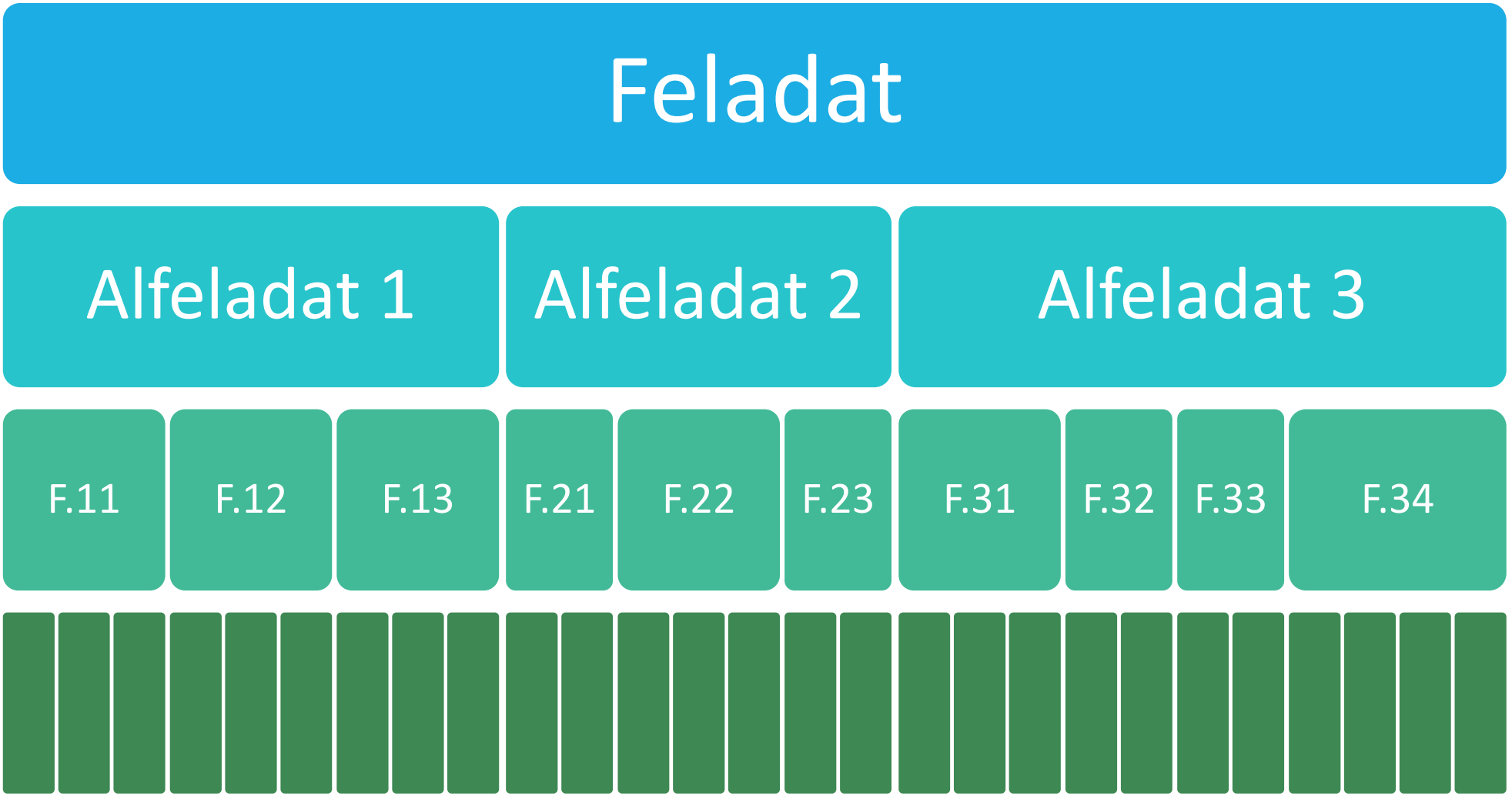
Általában a módszert ismételten alkalmazzuk

- Azaz az alrendszereket magukat is dekomponáljuk.
- Ez lehetővé teszi, hogy a feladat megoldásán egyszerre többen dolgozzanak

A módszer általi felbontás egy fával megadható.

- A fa csomópontjai az egyes dekompozíciós lépések

# Moduláris dekompozíció

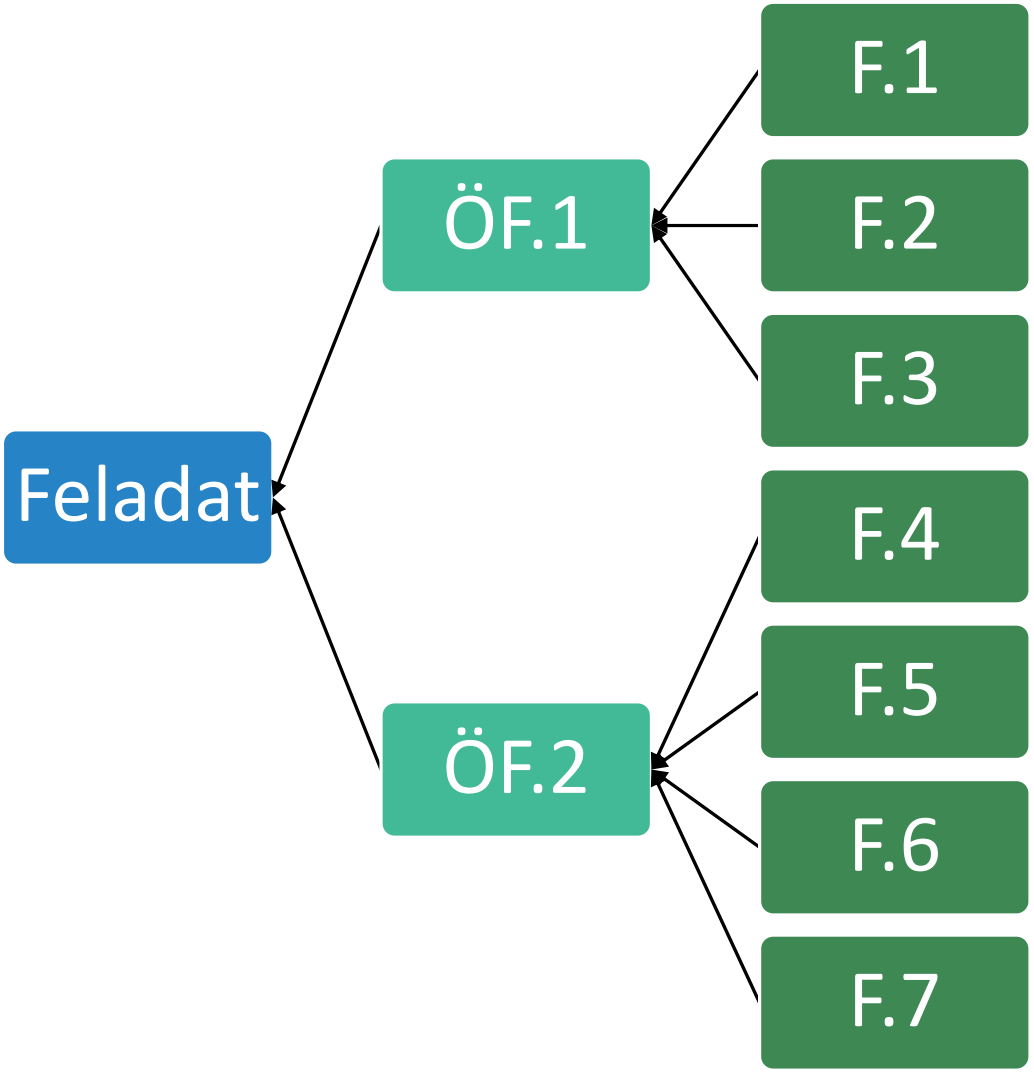


# Moduláris kompozíció

Olyan szoftver elemek létrehozását támogatja, amelyek szabadon kombinálhatók egymással.

- A programok lehetőleg a már meglévő programegységekből, mint építőkövekből épülnek fel.

# Moduláris kompozíció



# Modularitás

## Érthetőség

- A modulok önállóan is egy-egy értelmes egységet alkotnak
  - Megértésükhez minél kevesebb „szomszédos” modulra van szükség

## Folytonosság

- A specifikáció „kis” változtatása
  → a programban is csak „kis” változtatásra van szükség

## Védelem

- Célunk a program egészének a védelme az abnormális helyzetek hatásaitól.
  - Egy hiba hatása egy – esetleg néhány – modulra korlátozódjon

# A modularitás alapelvei

A modulokat nyelvi egységek támogassák

- A modulok illeszkedjenek a használt programozási nyelv szintaktikai egységeihez.

Kevés kapcsolat legyen

- Minden modul minél kevesebb másik modullal kommunikáljon!

Gyenge legyen a kapcsolat

- A modulok olyan kevés információt cseréljenek, amennyi csak lehetséges!

Explicit interface kell

- Ha két modul kommunikál egymással, akkor annak ki kell derülnie legalább az egyikük szövegéből.

# A modularitás alapelvei

## Információ elrejtés

- Minden információ egy modulról rejtett kell legyen, kivéve, amit explicit módon nyilvánosnak deklaráltunk.

## Nyitott és zárt modulok

- Zárt modul: más modulok számára egy jól definiált felületen keresztül elérhető, a többi modul változatlan formában felhasználhatja.
- Nyitott modul: kiterjeszthető
  - az általa nyújtott szolgáltatások bővíthetők vagy
  - hozzávehetünk további mezőket a benne levő adatszerkezetekhez, s ennek megfelelően módosíthatjuk eddigi szolgáltatásait.

# Újrafelhasználhatóság

A típusok változatossága
- A modulok többféle típusra is működjenek.

Adatstruktúrák és algoritmusok változatossága

Egy típus – egy modul
- Egy típus műveletei kerüljenek egy modulba.

Reprezentáció-függetlenség

# Programkönyvtárak

# Újrafelhasználható programkönyvtárak

## Tervezési igények

- Az alap-követelmények ugyanazok, mint  más szoftver tervezésénél
  - Helyesség
  - Megbízhatóság
  - Karbantarthatóság
  - Újrafelhasználhatóság
  - Kompatibilitás
  - Hordozhatóság
  - Hatékonyság
  - stb.
- A programkönyvtár felé az igények erőteljesebbek!

# Újrafelhasználható programkönyvtárak

Osztályokkal kapcsolatos további elvárások:

- Könnyű és intuitív használat
  - Könnyen megérthető legyen, néhány használat után jól megjegyezhető
- Azonnali, széleskörű felhasználhatóság
  - Szolgáltatásokat nyújt
- A lehető leghatékonyabb megvalósítás – state-of-the-art
- A tesztelés körültekintősége és az osztály megbízhatósága
- Magas szintű dokumentáció
  - Pontos, jól szervezett, hogy a felhasználó könnyen eligazodjon
- Platform-függetlenség
- Sokszor nyelv-függetlenség
- Meg kell hagyni a későbbi fejlesztés lehetőségét

# Újrafelhasználható programkönyvtárak

Néhány következmény

- Konzisztencia
  - a könyvtár minden komponense egy átfogó, koherens tervezésnek felel meg
  - számos szisztematikus, explicit és egységes konvenciót követ
- Jó minőségű, homogén komponensek kellenek
- OOP megközelítés az adatabsztrakció támogatása miatt különösen is alkalmas a könyvtárak készítésére

Ezek egyidejű teljesítése nehéz

- Lehetséges, hogy a legáltalánosabb algoritmus nem a leghatékonyabb sok speciális, de gyakran előforduló esetben.

# Újrafelhasználható programkönyvtárak

A könnyű felhasználhatóságnak különböző aspektusai vannak

- Az egyik ezek közül az osztályok elkülöníthetősége
  - Ha nagyon sok hasonló osztály található egy könyvtárban, akkor a felhasználó számára igen nehéz a pontosan megfelelő kiválasztása.
- Általában előnyösebb a kevés számú, majdnem teljesen ortogonális osztály definiálása
  - Minden esetben egy jól meghatározott szerep betöltésére
    - Az optimális megvalósítást követve

# Fejlesztő tulajdonságai

## A jó könyvtárfejlesztő tulajdonságai

- Van absztrakciós készsége
  - Képes arra, hogy az egyedi jelenségekből általánosítson
- Van érzéke a részletekhez
  - Egy könyvtárban minden kis tulajdonság számít.
  - Itt nincs olyan, hogy 'elég jó’
    - Teljes kell legyen a könyvtár
- Rend-mániákus
  - Képes az osztályozásra, hogy mindennek megtalálja a helyét.

# Fejlesztő tulajdonságai

Van irodalmi érzéke

- A leírásainak, a specifikációs részeket magyarázó megjegyzéseknek legyen stílusa és eleganciája

Elsőrendű programozó és tervező kell legyen

- Tudja átlátni a könyvtár későbbi használójának igényeit

„Ego-less” legyen

- Nem szabad, hogy egyéni stílus érvényesüljön
  - A cél a konzisztens stílus használatának lehetősége
- A kreativitása a könyvtár által nyújtott szolgáltatásokban jöjjön elő!

# Programkönyvtárak jellemzése

Egy programkönyvtár osztályok gyűjteménye.

- Egy osztályt az általa nyújtott szolgáltatások jellemeznek.

Ezen szolgáltatásokat a következő két módon lehet csoportosítani:

- I.
  - Számított (rutin)
    - Függvény
    - Parancs
  - Tárolt
    - Attribútum

# Programkönyvtárak jellemzése

Egy programkönyvtár osztályok gyűjteménye.

- Egy osztályt az általa nyújtott szolgáltatások jellemeznek.

Ezen szolgáltatásokat a következő két módon lehet csoportosítani:

- II.
  - Értéket visszaadó
    - Számított – függvény
    - Tárolt – attribútum
  - Parancs

# Programkönyvtárak jellemzése

Osztályok csoportosítási lehetősége

- csomagok (Java)
- clusterek (Eiffel)

használata.

A könyvtár és a csomag fogalmak kapcsolata lehet

- 1-sok
- sok-1

A csomagok gyakran a fizikai fájlrendszer struktúráját követik.

# Osztályhierarchia

OO esetben egy programkönyvtár elkészítése egy osztályhierarchia kialakítását is jelenti.

- A használhatóság szempontjából nem lényegtelen ezen hierarchia bonyolultsága vagy egyszerűsége.

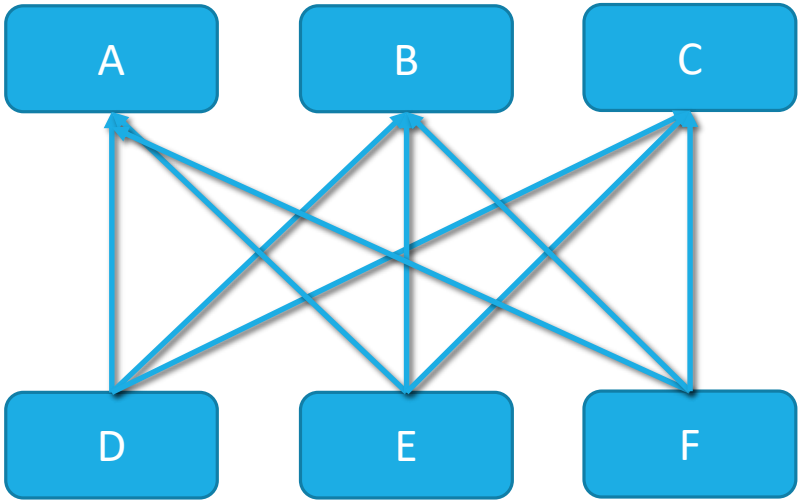
Az objektumorientált szemléletnek megfelelően

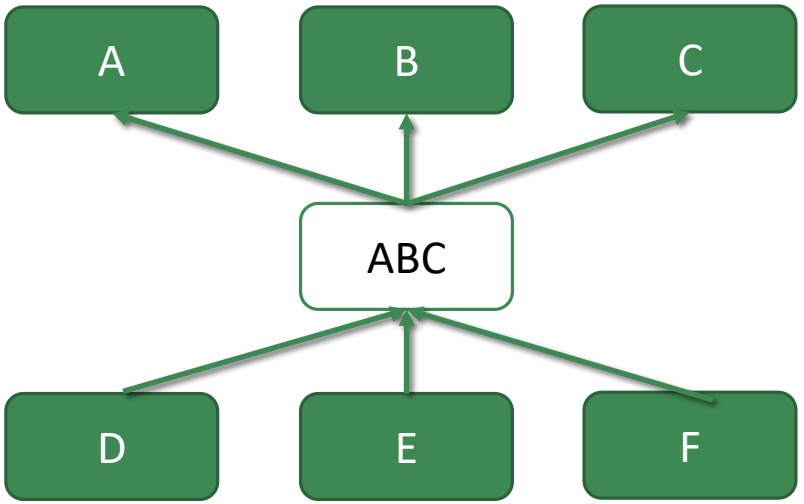
- igyekezzünk minél általánosabb osztályokat bevezetni (generalizáció)
- az öröklődést felhasználva hozzuk létre a szükséges osztályokat (specializálódás).

Ezáltal az osztályhierarchia karbantartása is leegyszerűsödik.

# Osztályhierarchia

Új osztály bevezetésének másik oka az osztályhierarchia egyszerűsítése is lehet:



# Biztonság

Egy sokak által használt programkönyvtár esetén fontos szempont a megbízhatóság és a biztonság.

- elő- és utófeltételek megadása
- invariánsok adásával

„Szerződés” létrejötte

- az előfeltétel a felhasználó kötelezettsége és a programozó biztosítéka
- az utófeltétel a programozó kötelezettsége és a használó biztosítéka
- Nem kell „defenzíven” programozni
  - minden lehetséges hibalehetőségre és bemenetre felkészülve
  - a kód hatékonyabb és egyszerűbb lesz

# Biztonság

## Két megközelítési mód

- Toleráns
  - A programkönyvtári rutinoknak nincs (vagy csak gyenge) előfeltételük
  - Minden lehetséges bemenetre valamilyen módon reagálnak
- Követelőző
  - Minden rutin szigorú előfeltételekkel rendelkezik
  - Ezek biztosítása a felhasználó felelőssége

## Az ajánlott megközelítés:

- Csak az absztrakt művelet logikai helyes elvégzéséhez szükséges előfeltételeket ellenőrizzük.
- Csak azt ellenőrizzük, ami, ha nem teljesülne, súlyosan befolyásolná a hatékonyságot.

# Biztonság

Az így létrejött szerződés bármilyen megsértése programhibát jelent

- Az előfeltétel megsérülése felhasználó oldali hiba
- Az utófeltétel vagy az invariáns sérülése programkönyvtár hiba

A programkönyvtár fejlesztése után az előfeltétel teljesülés vizsgálatát kell elvégezni

- A terjesztett változatban a hatékonyság érdekében.

# Dokumentáció

Egy programkönyvtár több fejlesztő munkája.

A felhasználhatóság szempontjából ezért igen fontos a jó dokumentáció.

Lehetséges megközelítési módok:

- „Forrásszöveg maga a dokumentáció, mi kell még?”
- Különálló dokumentáció írása

# Példák

```
/**
 * Always returns true.
 */
public boolean isAvailable() {
    return false;
}
```

```
for (j=0; j<arra_len; j+ =8) {
  total += array[j+0];
  total += array[j+1];
  total += array[j+2]; /* comment
  total += array[j+3];  * out
  total += array[j+4];  * some
  total += array[j+5];  * lines
  total += array[j+6];  */
  total += array[j+7];
}
```

```
#define TRUE FALSE
```

```
a=a++
```

```
*++b ? (*++b + *(b-1)) 0
```

- Nem definiált viselkedés, a precedencia miatt

# Híres példa – $1/\sqrt{(x)}$

```
float Q_rsqrt( float number )
{
    long i;
    float x2, y;
    const float threehalfs = 1.5F;

    x2 = number * 0.5F;
    y = number;
    i = * ( long * ) &y; // evil floating point bit level hacking
    i = 0x5f3759df - ( i >> 1 ); // WTF?
    y = * ( float * ) &i;
    y = y * ( threehalfs - ( x2 * y * y ) );
                        // 1st iteration
// y = y * ( threehalfs - ( x2 * y * y ) );
                        // 2nd iteration, this can be removed
    return y;
}
```

# Dokumentáció

A forrásszöveg nem elég absztrakt

- A felhasználó számára fontos információkon kívül az alacsonyszintű implementációt is tartalmazza
- A használata túl sok információ átfésülését jelentené
- Arra sarkallná a programozókat, hogy nem publikusnak szánt implementációs lehetőségeket is kihasználjanak
- Előnye, hogy mindig aktuális.

Különálló dokumentáció esetén nem biztosított a szoftver és a dokumentáció konzisztenciája.

- Előnye, hogy csak a felhasználó számára fontos dolgokat tartalmazza.

# Dokumentáció

Belső dokumentáció elve: a szoftver dokumentációja a programszövegbe legyen beágyazva.

Ennek előnyei:

- Forrásszöveg és dokumentáció mindig konzisztens marad.
  - Előbbi példa …
- Dokumentáció kinyerése automatizálható
  - javadoc

Eiffel flat-short forma

- az osztálynak egy olyan verziója, ami tartalmazza az öröklött és az itt bevezetett jellemzőket
  - figyelembe véve minden átnevezést és átdefiniálást
  - felépítve a teljes elő- és utófeltételt és a típusinvariánst

# Karbantartás

Egy programkönyvtár állandó fejlesztés alatt áll,

- Lehetőséget kell biztosítani újabb verziók problémamentes beépítésére.

A következő változtatások nem jelentenek problémát:

- Osztály bővítése új szolgáltatással.
- Egy szolgáltatás implementációjának lecserélése
- Előfeltétel gyengítése, illetve utófeltétel szigorítása

# Karbantartás

Szükség lehet egy szolgáltatás új verziójának bevezetésére, ha

- A szolgáltatás neve megváltozik
  - Például egységes névkonvenció kialakítása miatt
- Jobb szignatúra vagy specifikáció megadása
- Rutin elavulttá nyilvánítása.
  - Ez történhet nyelvi kulcsszavak használatával
    - Eiffel: obsolete
    - Java: deprecated
  - Vagy a régi változat átírásával
    - Az egyből az új változatot hívja meg (hívásátirányítás)
    - Esetleges figyelmeztető üzenet kiírása mellett

# Osztályok mérete

Egy osztály méretét a következő módokon lehet meghatározni:

- Sima (teljes) méret = szolgáltatások száma (öröklött + itt bevezetett).
- Közvetlen méret = közvetlen (nem absztrakt, illetve újradefiniált) szolgáltatások száma.
- Növekményes méret = közvetlen és újradefiniált szolgáltatások száma.

Másik szempont lehet, hogy tartalmazza-e a nem exportált jellemzőket:

- Külső méret
- Belső méret

# Osztályok mérete

A teljes méretre nyilván nincs felső határ

- A közvetlen, és a növekményes méret az, ami szerepel az osztály szövegében.
- De egy adott méreten felül az osztály kezelhetetlen.

Két megközelítési lehetőség

- Minimalista nézőpont
  - egy programkönyvtár osztálya a megvalósított típusnak csak a legalapvetőbb (atomi) szolgáltatásait tartalmazza,
  - redundáns (azaz ami atomi szolgáltatások használatával is megvalósítható) metódusokat pedig nem.
  - Ez persze a programkönyvtár írójának kedvez, hiszen kevesebb munkát és egyszerűbb karbantartást jelent.

# Osztályok mérete

- Bevásárlólista nézőpont: minden olyan szolgáltatást vegyünk bele az osztályunkba, ami az alábbi szempontoknak megfelel:
  - A szolgáltatás beleillik az absztrakt adattípus megvalósítási sémájába.
  - Nem rontja el az osztály helyességét, azaz az osztályinvariánst.
  - Hasznos funkcionalitást valósít meg.
  - Nem duplikál valamely már meglévő szolgáltatást.
  - Megfelel a könnyen kezelhetőség kritériumainak (lásd később).

# Osztályok mérete

A gyakorlatban általában

- a programozási nyelvtől elvárt szempont a minimalista nézőpont
  - azaz hogy műveletet csak minél kevesebb módon lehessen hatékonyan elvégezni,
- míg a programkönyvtárak tervezésekor a bevásárlólista nézőpont érvényesül
  - a felhasználás megkönnyítése érdekében

# Szolgáltatások száma

Túl sok szolgáltatás a programkönyvtár használatának megtanulhatóságát is kedvezőtlenül befolyásolja

Ezért

- Minden szolgáltatás egy jól meghatározott szerződéssel definiált
  - csak a specifikációjuk, nem pedig az implementációjuk alapján kerülnek felhasználásra.
- Az osztálydokumentáció egységes és könnyen áttekinthető formátumú legyen
  - feltüntetve minden szolgáltatás szerződését és elrejtve az implementációs részleteket
- A szolgáltatások nevei szigorú elnevezési konvenciót kövessenek.
- A szolgáltatásokat csoportosítsuk pontosan definiált kategóriák szerint, ezen kategóriák minden osztálynál egyezzenek meg
  - sorrendjük a dokumentációban legyen azonos
  - kategórián belül a szolgáltatások névsorrendben következzenek

# Példaadatok

Általában kijelenthető, hogy 80 szolgáltatásnál többet tartalmazó osztály esetén már meggondolandó, hogy nem lehetne-e az osztályhierarchiát tovább bontani.

# Szolgáltatások mérete

Fontos, hogy a használó pontosan ismerje egy szolgáltatás paramétereinek számát, típusát és sorrendjét.

- Ezért a felhasználás szempontjából egy szolgáltatás méretét
  - Tulajdonképpen a megadandó paraméterek száma
- Ez a méret erősen függ a programkönyvtár feladatától is,
  - Egy grafikus felületet megvalósító, vagy statisztikai számításokat végző osztály metódusai rengeteg argumentumot igényelhetnek.

# Szolgáltatások mérete

Nagyszámú argumentum esetén érdemes azokat két csoportra osztani:

- Paraméter: olyan argumentum, amelynek értékével valamilyen műveletet kell elvégezni.
- Opció: olyan argumentum, amely akár el is hagyható, ugyanis rendelhető hozzá egy alapértelmezett érték.
  - Csak a paraméterek feldolgozását befolyásolja.

Egy szolgáltatás fejlesztése során a paraméterlista lehetőleg ne változzon

- Új opciókat nyugodtan fel lehet venni.

# Szolgáltatások mérete

A szolgáltatás méretének csökkentése érdekében csak paramétereket vegyünk fel az argumentumlistába.

Az opciókat pedig külön attribútummal reprezentáljuk

- Adjunk hozzá új beállító/lekérdező szolgáltatásokat.
  - Opció beállítási technika.
- Növeli az osztály méretét
  - Esetleges új, globális opció-attribútum

# Példaadatok

Általában kijelenthető, hogy az a jó, ha a szolgáltatások átlagos paraméterszáma 0.5

- 5-nél több paraméter esetén meggondolandó a továbbontás

A szolgáltatások lekérdezés – parancs javasolt aránya

- 60 - 40%

# Elnevezés

Egy programkönyvtár használatát nagyban elősegíti az egységes elnevezési konvenció.

- Ez konzisztens névhasználatot jelent
- Kialakítása a megvalósított adattípusok közös tulajdonságain alapul

Például

- egy adott elem elérésére ( get )
- egy elem megváltoztatására ( put )
- egy elem törlésére ( remove )

Azonos neveket használunk különböző osztályokon belül.

- Mégsem vezet félreértéshez
  - a típusok különbözősége miatt (általában) a szignatúrák nem egyeznek meg

# Elnevezés

Standard elnevezések:

- `capacity` - adott struktúra kapacitását adja vissza.
- `count` - adott struktúra elemszámát adja vissza.
- `empty` - megadja, hogy az adott adatstruktúra üres-e.
- `full` - megadja, hogy az adott adatstruktúra tele van-e.
- `get` - struktúra adott elemét adja vissza.
- `has` - lekérdezi, hogy adott elem a struktúra eleme-e.
- `put` - struktúra adott elemét állítja be.
- `remove` - struktúra adott elemét kitörli.

# Elnevezés

Irányelvek:

- A név legyen rövid (rendszerint egy szó), de beszédes.
- Szolgáltatás neve ne tartalmazzon utalást a szolgáltatást tartalmazó osztályra
  - kivéve többszörös öröklődés esetén, ha átnevezés kell
- Osztály neve mindig főnév(i szerkezet) legyen.
- Parancsok (eljárások) nevei legyenek (felszólító módú) igék
  - Esetleges főnévi vagy értelmező jelzői kiegészítővel
- Nem logikai lekérdezések nevei legyenek főnevek vagy főnévi szerkezetek.

# Elnevezés

- Logikai lekérdezés neve legyen olyan melléknév, amely igen/nem választ sugall
  - Vagy használjuk az is_ (-e) szerkezetet
- A szolgáltatás nevét a célobjektum szemszögéből lehessen értelmezni
  - A szolgáltatás hívásakor mindig létezik egy objektum (a célobjektum) aminek a szolgáltatásáról szó van.
  - has
- Összetartozó művelet és lekérdezés párok elnevezéseinek szótöve legyen azonos
  - extendible - extend

# Osztályok fajtái

Konkrét típusok

- Minden programkönyvtár hierarchiában találunk olyan típusokat, melyek alapvető adatstruktúrákat reprezentálnak
- lista, vektor, string, dátum, komplex szám, …

# Osztályok fajtái

Konkrét típusok – tulajdonságok

- Szoros illeszkedés egy konkrét fogalomhoz és a megvalósítás módjához.
- Érthetőség, önálló használhatóság.
- Erősen implementációfüggők
  - Ezért hatékony a megvalósításuk
  - Bármilyen módosításukkor minden felhasználói kódot újra kell fordítani.
- Minimális mértékben függenek más osztályoktól.
- Különállóan (izoláltan) is fordíthatóak és használhatóa
- Nem, illetve ritkán lehet tőlük örökölni.

# Osztályok fajtái

## Absztrakt típusok

- Szerepük rendszerint egy adott specifikáció (akár interfész is lehetne) bevezetése. Pl. halmaz
- Az absztrakt típusokat konkrét típusok segítségével lehet implementálni.

## Absztrakt típusok – tulajdonságok

- Azonos elérési módhoz többfajta implementáció is létezhet.
- Hatékony tárkihasználás és elfogadható futási idő a virtuális metódusok segítségével.
- Az implementáció minimális mértékben függ más osztályoktól.
- Különállóan is lefordíthatóak – érthetőek önmagukban.

# Osztályok fajtái

## Csomópont típusok

- Az öröklési hierarchia belső pontjait alkotják.

## Jellemzőik

- Az ősosztályok szolgáltatásait implementálja,
  - Interfészét kibővíti virtuális metódusokkal is, melyeket maga is implementál.
- Függ az őseitől.
- Lehet belőle örökölni.
- Példányosítható.

# Osztályok fajtái

## Felületosztályok

- Egy felület igazítása úgy, hogy jobban illeszkedjen a felhasználó elvárásaihoz.
  - Például vektor ne 0-tól legyen indexelve.
- Ellenőrzött vagy korlátozott felületek biztosítása.

# Osztályok fajtái

„Kövér interfészek”

- Ez egy olyan típus, amely nem függ más osztályoktól, rengeteg szolgáltatást vezet be
  - ezek közül csak a legalapvetőbbekhez ad implementációt, míg a speciálisabb szolgáltatásokat virtuálisként deklarálja
  - Lehet példányosítani, ad hozzá egy üres implementációt, ami rendszerint egy hibaüzenet kiírását jelenti
    - Jelzi, hogy az adott szolgáltatás nem implementált.
    - Példa: általános tároló (container) osztály.

# Osztályok fajtái

Keret osztályok (Framework)

- Ezen absztrakt osztályok tulajdonképp egy mini alkalmazást valósítanak meg.
- Maga a programlogika van csak az osztályon belül implementálva
  - minden egyéb paraméter-bekérő és alapműveletet elvégző metódus virtuális
- Példa: szűrő osztály, amely egy bemeneti adatfolyam minden elemén végiglépkedve adott feltétel teljesülésekor bizonyos műveleteket elvégez.
- A bemeneti adatfolyamon történő végigléptetés, adott tulajdonság fennállásának ellenőriztetése,
  - A kívánt művelet meghívása és esetleges hibakezelés ezen osztályban van implementálva.

# Osztályok fajtái

## Leíró osztályok

- Felmerülhet az igény adott interfészen keresztül változó méretű osztályok objektumainak kezelésére is.
  - Erre ad megoldást a leíró osztályok használata.
- Ekkor ugyanis az implementáció két részre bomlik
  - az aktuális reprezentációra
  - a reprezentáció elérését biztosító leíró (handle) objektumra
  - Ez tulajdonképp a mutató típus objektumorientált megvalósításának tekinthető.
- Példa: fájlleíró osztályok

# Memóriakezelés

A programkönyvtár tervezés egyik kulcsfontosságú kérdése.

Két alapvető probléma:

- Memóriafoglalás új objektumok létrehozásakor.
- Memória felszabadítása.

Míg az első az operációs rendszer feladata, addig a második problémára két megoldási irányzat is született:

- Automatikus szemétgyűjtés (pl. Java).
  - Ez rendszerint referenciák bevezetését és a mutató típus megszüntetését jelenti.
- Objektumok kézzel történő felszabadítása.
  - Hatékonyabb, de sokkal veszélyesebb.

# Algoritmus, grammatika, fordítóprogramok

JÖVŐ HÉTEN




# Programozási nyelvek és módszerek

# Algoritmus

# Algoritmus

Abu Abdallah Muḥammad ibn Mūsā al-Khwārizmī

- 780 – ?845
- arabul alkotó perzsa tudós
- matematikus

Kidolgozta a matematikai algoritmusok fogalmát

- néhányan a számítástechnika nagyapjának

Az algoritmus szó a nevének félrefordított latin változatából ered



# Algoritmus

„Az algoritmus egyértelműen előírt módon  és sorrendben végrehajtandó tevékenységek véges sorozata egy feladat megoldására, ami véges idő alatt befejeződik.”

Hétköznapi életben is: egy feladat megoldási lépései teljes részletességgel

- Feladat: Süss egy gyümölcstortát!
  - Hogy írjuk le a feladatot?
  - Mi az input?
  - Mi az output?
- Itt az algoritmus: a recept

Nem minden feladatra adható algoritmus!

# Algoritmus

Megadható:

- Szövegesen, természetes nyelven
- Pszeudokóddal
- Grafikusan
  - Folyamatábrával
  - Struktogrammal
  - …
- Automatákkal vagy Turing-gép segítségével
- Táblázattal
- Programozási nyelv segítségével

# Algoritmus

Komponensei:

- Szekvencia  - parancsok egymásutánja
- Elágazás – valamilyen feltételtől függően alternatívák választása
- Ciklus – utasítások ismételt végrehajtása

Fontos kérdések

- Biztosan megoldja a feladatot?
  - Helyes – specifikációnak megfelel?
  - Mit jelent az, hogy megoldás?
    - Részletesen még jön!

# Feladat specifikációja

## Specifikáció megadása

- Szövegesen
  - Példa: számítsd ki **a** és **b** legnagyobb közös osztóját!
  - Számos információt „beleértünk”….
  - Néha félreértjük – véletlenül vagy szándékosan
    - „Találkozzunk a Keletinél a metró lépcsőjénél”
      - Pontosabb az, hogy „a Keletinél a metró új kijáratánál”?
    - „A királynőt megölni nem kell félnetek jó lesz”

# Feladat specifikációja

## Specifikáció megadása

- Formálisan
  - $I$ – a lehetséges bemenetek,
    $O$ – a lehetséges kimenetek
    végesen leírható halmaza
  - $F \subseteq I \times O$ reláció, ahol $i \in I$ és $o \in O$ esetén $(i,o) \in F$ azt jelenti, hogy $F$ az $i$ bemenethez hozzárendeli az $o$ kimenetet.
  - Például:
    - $I = N \times N$
    - $O = N$
    - $F = \{((a,b),c)$, ahol $a, b, c \in N \wedge c \mid a \wedge c \mid b \wedge \forall x : x \mid a \wedge x \mid b \rightarrow c \geq x\}$
    - $c \mid a$ jelentése: $a$ osztható $c$-vel.

# Feladat specifikációja

*Szerződés* a felhasználó és az implementáló között

- Előfeltétel: állítás, ami leírja azt a feltételt, ami szükséges a feladat helyes működéséhez
- Utófeltétel: állítás, ami leírja azt a feltételt, amit a függvény teljesít a helyes végrehajtás után

Helyesség a specifikáció figyelembevételével

- A program felhasználója teljesíti az előfeltételt
- A program lefut
- A program végeztével az utófeltétel igaz lesz.

Mit kell az implementációnak teljesíteni, ha a felhasználó megsérti az előfeltételt?

# Feladat specifikációja

Elő- és utófeltételekkel

- Állapottér: $\mathbb{N}\times\mathbb{N}\times\mathbb{N}$
  $a \quad b \quad c$
- Előfeltétel (Ef)
  - $a=a' \wedge b=b'$
- Utófeltétel (Uf):
  - $\text{Ef} \wedge c \mid a \wedge c \mid b \wedge \forall x : x \mid a \wedge x \mid b \rightarrow c \geq x$

# Programozási nyelv

# Mi a programozási nyelv?

Egy jelölés …

Eszköz

- a számítógép vezérlésére
- programozók közötti kommunikációra
- algoritmusok leírására
- magas szintű tervezésre
- a feladat bonyolultságának kezelésére

# Programozási nyelvek kialakulása

Gépi kód

Assembly nyelv

- Mnemonikok, címkék használata
- Assembler

Magas szintű nyelv

- Utasításkészlete független egy adott számítógép-architektúra utasításkészletétől
  - Végrehajtásuk előtt egy fordítóprogramnak kell fordítani.
    - Assembly kód -> assembler.
    - Közvetlenül gépi kód.

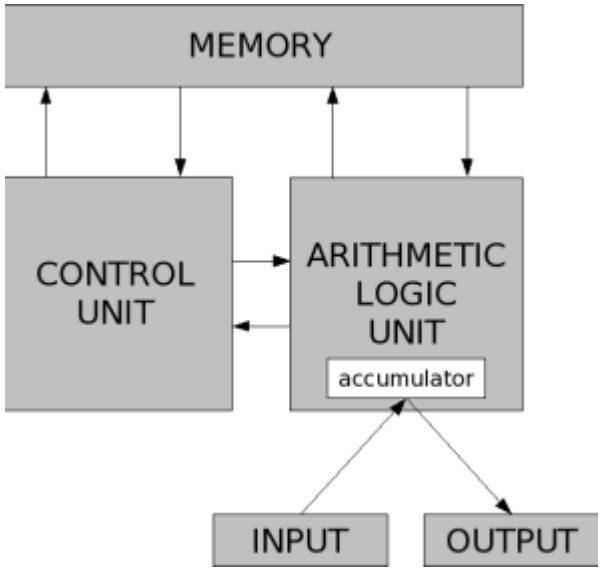
# Neumann János (1903 – 1957)



Neumann–elvek a számítógépről:

- Legyen univerzális
  - bármilyen feladat megoldására használható.
- Legyen soros működésű
  - az utasításokat sorban egymás után hajtsa végre
- Legyen teljesen elektronikus működésű
- Az egyes utasítások és adatok leírásához és feldolgozásához a kettes számrendszert használja.
- Legyen belső memóriája
- Tárolt program elven működjön
  - Egy program indításakor a programot alkotó utasítások és a működéshez szükséges adatok is kerüljenek be a belső memóriába
  - Az utasításokat a memóriából kiolvasva, sorban egymás után hajtsa végre a berendezés.

# Neumann János – 1946




A számítógép a következő egységekből álljon:

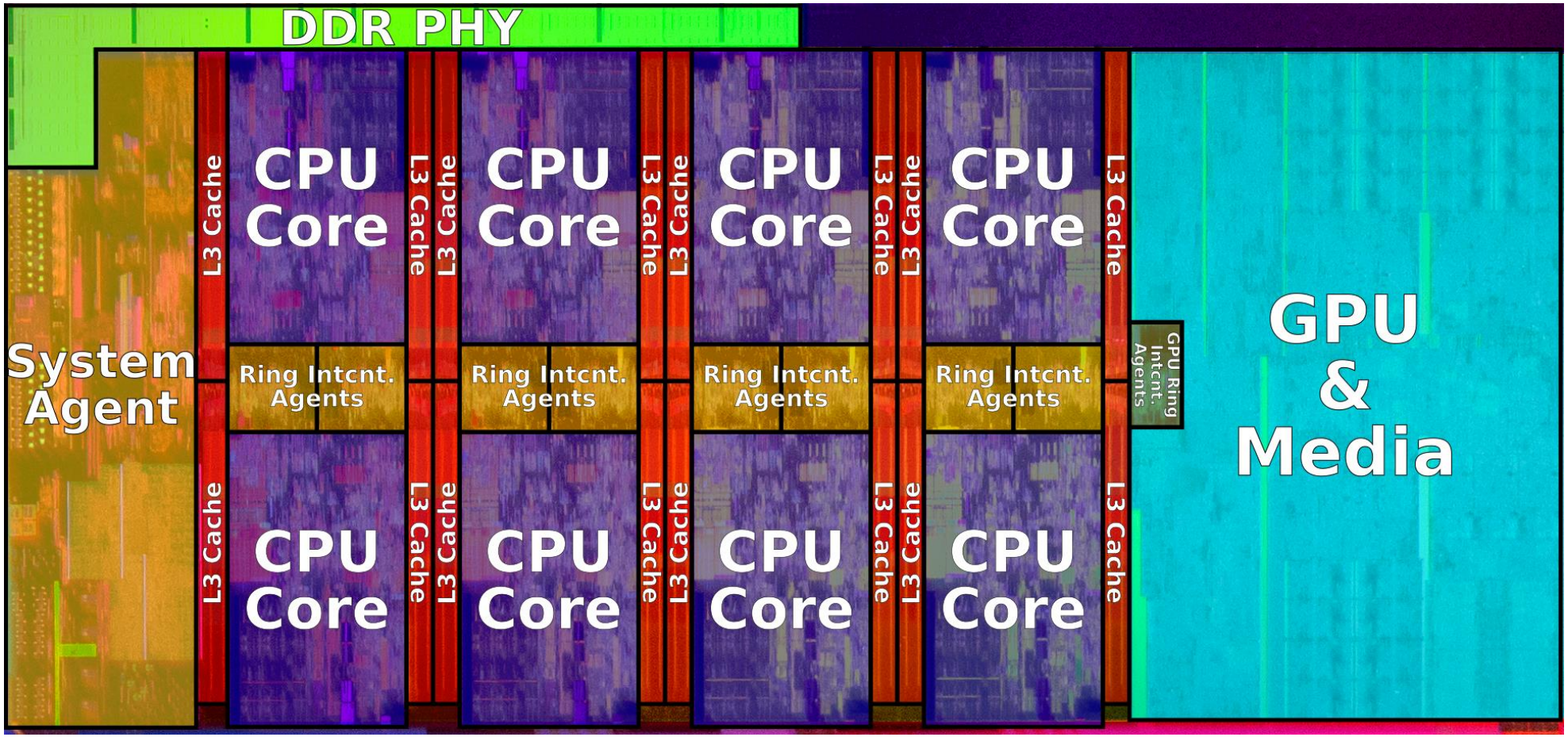
- Központi egység:
  - vezérlés, műveletvégzés, adatmozgatás
- Memória:
  - az adatok és a programok tárolása a program végrehajtása közben
- Bemeneti-kimeneti egység:
  - kommunikáció a külvilággal, az adatbevitel- és kivitel megvalósítása

# Egyszerűsített példa

HTTPS://GODBOLT.ORG/

# CPU – Core i7 9700K



# Program végrehajtása

Gépi kód: CPU / Vezérlő által értelmezett utasítások

```
55
48 89 e5
89 7d ec
89 75 e8
82 7d ec 00
79 03
f7 5d ec
83 7d e8 00
79 03
f7 5d e8
83 7d e8 00
7e 18
8b 45 e8
89 45 fc
8b 45 ec
99
f7 7d e8
89 55 e8
8b 45 fc
89 45 ec
eb e2
8b 45 ec
5d
c3
```

```
55
48 89 e5
b0 0a 00 00 00
bf 22 00 00 00
e8 ae ff ff ff
b8 00 00 00 00
5d
c3
0f 1f 44 00 00
```

Ember számára nehezen olvasható.
Számítógép közvetlenül végre tudja hajtani.

# Alacsonyszintű nyelv

Assemby – Ember számára is olvasható kód.

```
_Z4lnkoii:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    %edi, -20(%rbp)
        movl    %esi, -24(%rbp)
        cmpl    $0, -20(%rbp)
        jns     .L2
        negl    -20(%rbp)
.L2:
        cmpl    $0, -24(%rbp)
        jns     .L5
        negl    -24(%rbp)
.L5:
        cmpl    $0, -24(%rbp)
        jle     .L4
        movl    -24(%rbp), %eax
        movl    %eax, -4(%rbp)
        movl    -20(%rbp), %eax
        cltd
        idivl   -24(%rbp)
        movl    %edx, -24(%rbp)
        movl    -4(%rbp), %eax
        movl    %eax, -20(%rbp)
        jmp     .L5
.L4:
        movl    -20(%rbp), %eax
        popq    %rbp
        ret
```

```
main:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    $10, %esi
        movl    $34, %edi
        call    _Z4lnkoii
        movl    $0, %eax
        popq    %rbp
        ret
```

Ember számára olvashatóbb.
Közvetlen fordítható gépi kódra.

# Magasszintű nyelv

C++ – Ember számára olvasható kód.

```
int lnko(int a, int b) {
   int tmp;
   if (a<0) a= -a;
   if (b<0) b= -b;
   while(b > 0) {
      tmp = b;
      b = a % b;
      a = tmp;
   }
   return a;
}
```

```
int main()
{
    lnko(34, 10);
    return 0;
}
```

Ezt szoktuk meg.
A nyelv szabályait definiálni kell.
Fordítóprogramok kellenek.

# Modell

HTTP://WWW.PETERHIGGINSON.CO.UK/LMC/

# A „Little Man Computer”

Neumann elvek az oktatásban

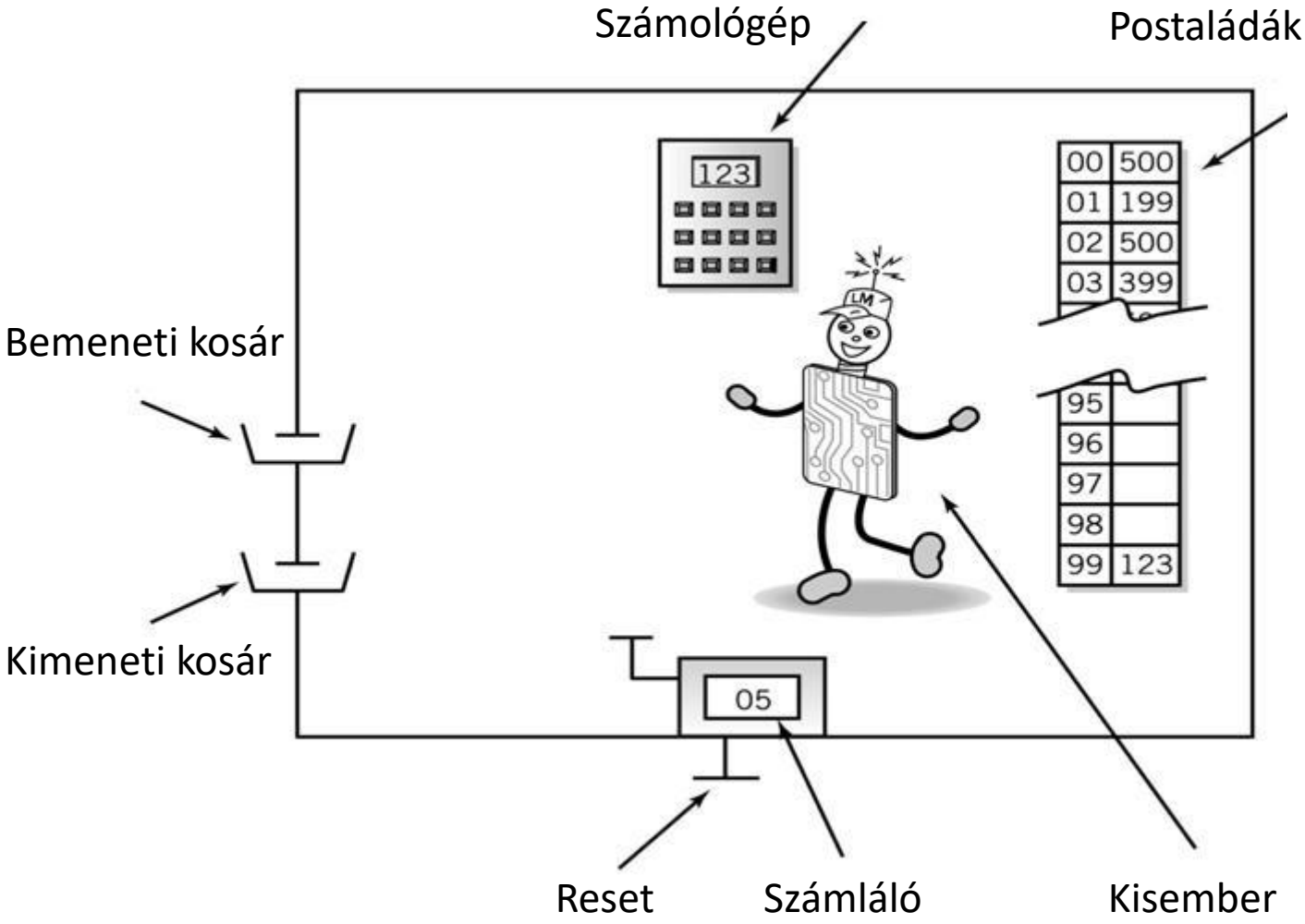
Stuart Madnick, MIT

Szoba, benne:

- Kisember
- 100 postaláda (00-99)
- Számláló (2)
- Bemenő és kimenő kosár
- Számológép

Nem foglalkozunk azzal, hogyan

- töltjük fel a postaládákat
- tegyünk adatot a bemenő kosárba



# LMC elemei

Bemeneti és kimeneti kosár:

- Kisember kap/beletesz egy 3 jegyű számot tartalmazó cédulát

Számláló:

- 0 és 99 között tud számolni
- Pedál megnyomására a számlálóban tárolt érték eggyel megnő
- Értékét nullára állíthatjuk egy külső „reset” gombbal

Számológép:

- Háromjegyű decimális számokat tud kezelni.
- Képes
  - Kivonni
  - Összeadni
  - A begépelt vagy a számítás eredményeként kapott értéket eltárolni.

# Postaládák

A postaládákban számokat tárolunk

- elérésük a postaláda címe (sorszáma) alapján

A címek folyamatosan egymás után jövő értékek

- 00 .. 99

A tartalom lehet

- Adat vagy
- Utasítás

| Cím | Tartalom |
|---|---|
| 00 | 522 |
| 01 | 123 |

# Utasítás esetén

Műveleti kód

Operandus

- A művelet operandusa: adat vagy az adat címe

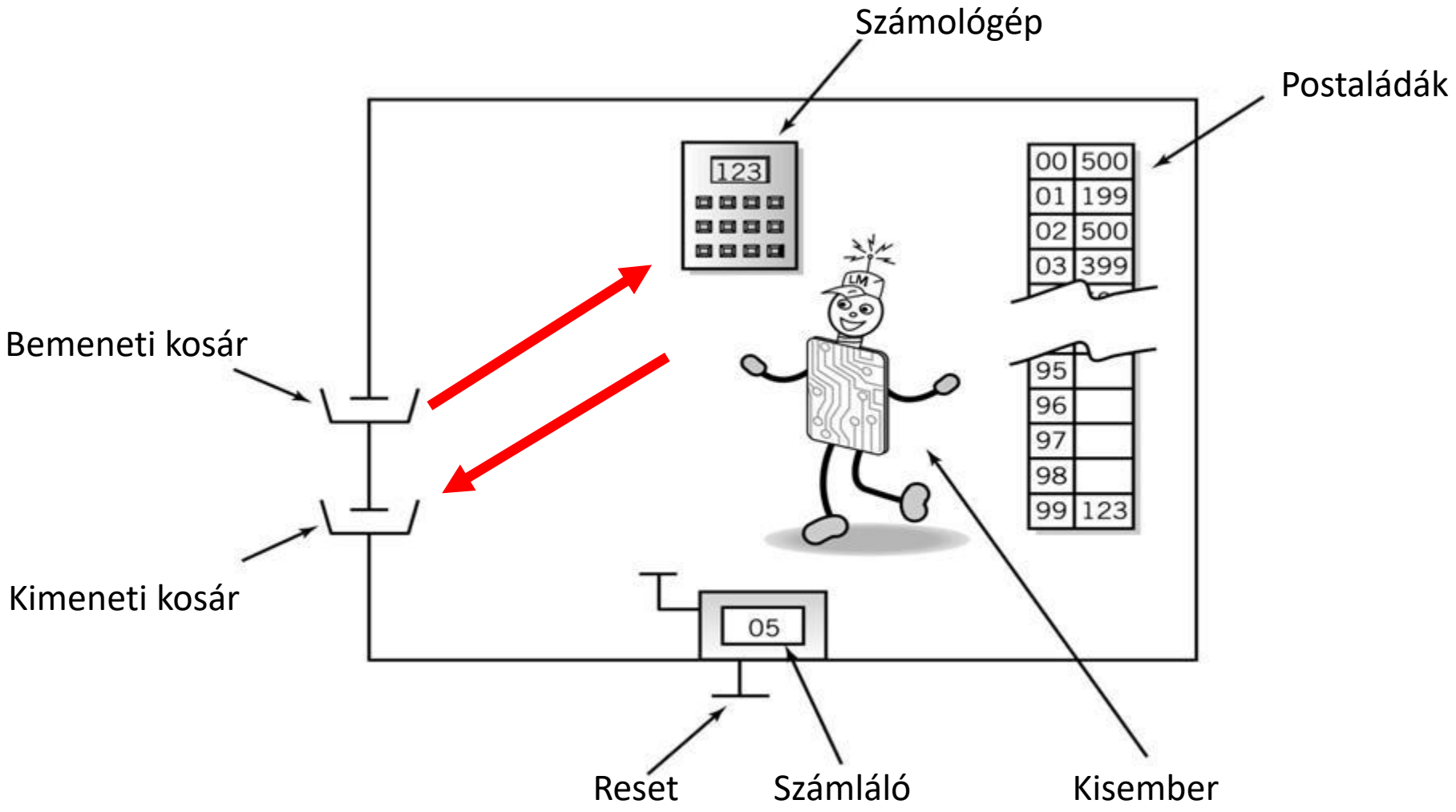
| Cím | Tartalom | |
|---|---|---|
| | Műv. kód | Operandus |
| | | |

# Utasításkészlet

Lehetséges utasításkészlet:

| | | |
|---|---|---|
| Aritmetikai | 1xx | Összeadás |
| | 2xx | Kivonás |
| Adat mozgató | 3xx | Tárolás |
| | 5xx | Betöltés |
| Input/Output | 901 | Beolvasás |
| | 902 | Kiírás |
| Vezérlés | 000 | Leállás |

# Input/Output

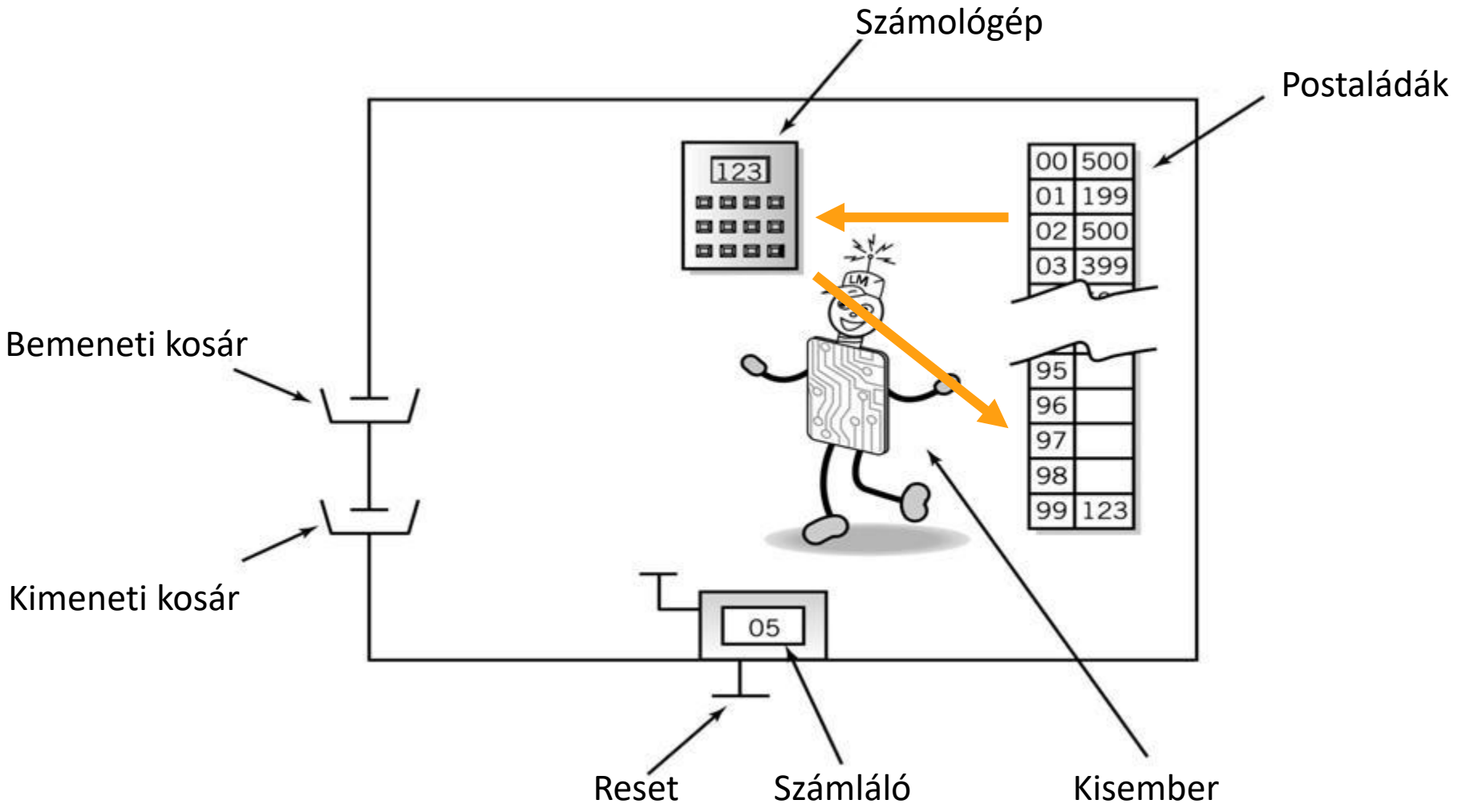


# Input/Output

Adatok mozgatása a számológép és a bemeneti/kimeneti kosarak között

| | Tartalom | |
|---|---|---|
| | Műveleti kód | Operandus (cím) |
| Beolvasás | 9 | 01 |
| Kiírás | 9 | 02 |

# LMC belső adatmozgatás



# Belső adatmozgatás

A postaláda és a számológép között

| | Tartalom | |
|---|---|---|
| | Műveleti kód | Operandus (cím) |
| Tárolás | 3 | xx |
| Betöltés | 5 | xx |

# Adatok tárolása

Az utasításokat és az adatokat tároló postaláda fiókok fizikailag azonosak

- Ezt valós esetekben elkülönítve tároljuk a memóriában

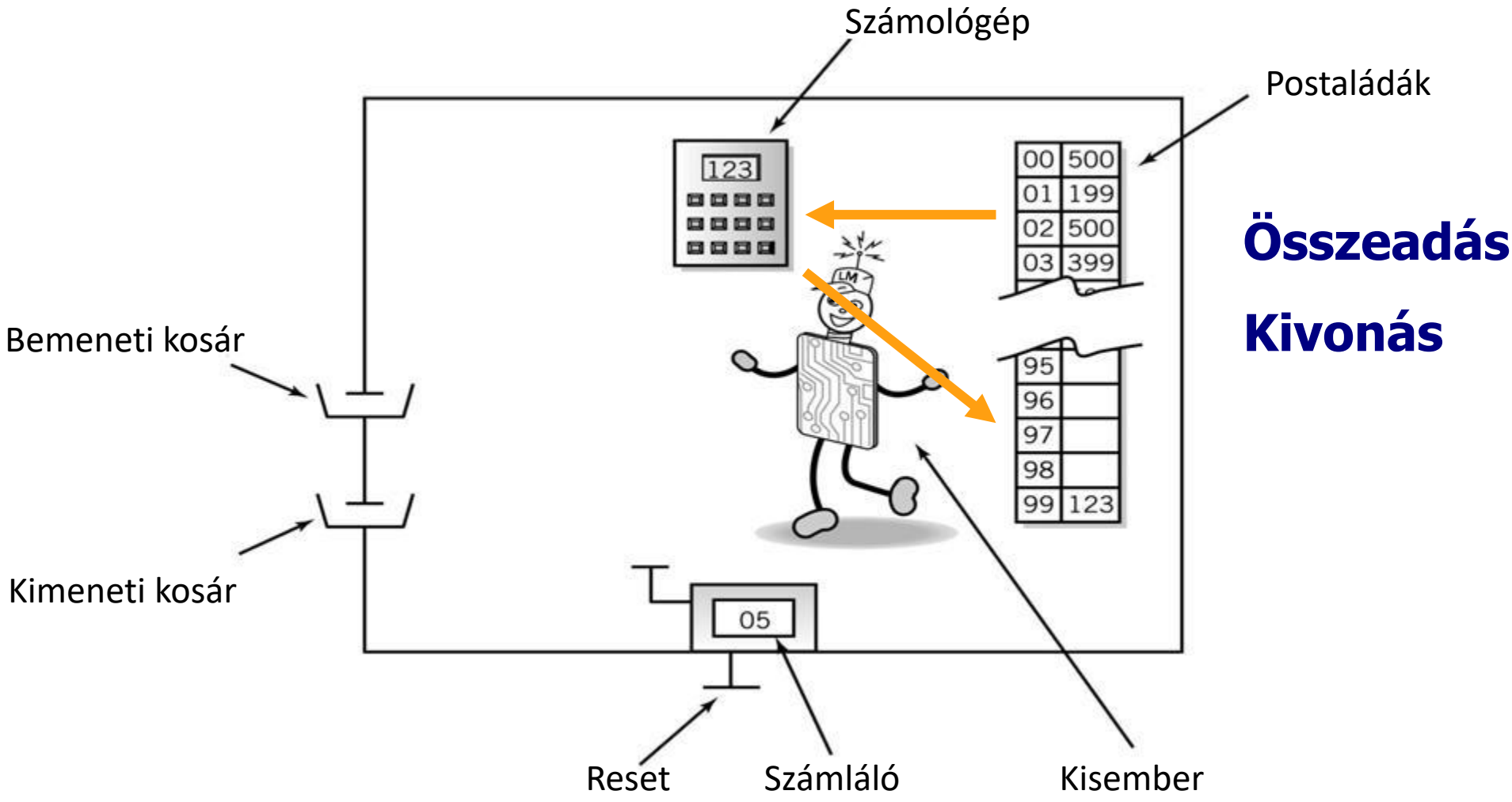
# Aritmetikai utasítások

Olvassuk el az operandus által meghatározott levelesládában lévő értéket

Hajtsuk végre a műveletet a számológéppel

| | Tartalom | |
|---|---|---|
| | Műveleti kód | Operandus (cím) |
| Összeadás | 1 | xx |
| Kivonás | 2 | xx |

# LMC Aritmetikai utasítások



# Utasítás-végrehajtási ciklus

Az utasítások végrehajtása

- Kikeresés (Fetch)
  - Kisember kikeresi, hogy melyik utasítást kell végrehajtani
- Értelmezi az utasítást (Decode)
  - Az utasítás kódja alapján eldönti, hogy mit kell pontosan csinálni
- Végrehajtás (Execute)
  - Kisember elvégzi a munkát.

A számítógép által egy-egy utasítás végrehajtásakor elvégzett tevékenység-sorozatot utasítás-végrehajtási ciklusnak nevezzük

- Mivel ezek ciklikusan ismétlődnek.

# Kikeresés és dekódolás

1. Kisember kiolvassa az utasításszámlálóból annak a postaládának a sorszámát (címét), ami az utasítást tartalmazza
2. Átsétál ahhoz a postaládához, ami megfelel a számláló értékének.
3. Kiveszi belőle a cédulát, elolvassa és dekódolja a számot, ami rajta van
   - A cetlit visszateszi, arra az esetre, ha esetleg később szükség lenne rá, hogy újra elolvassa

# Végrehajtás

1. Kisember odamegy a postaládához, aminek a sorszámát az éppen kikeresett utasítás tartalmazza.
2. Kiveszi belőle a cédulát és elolvassa a rajta lévő számot
   - nem felejti el a cédulát visszatenni, mert később is szükség lehet rá
3. A kiolvasott utasítás kódjának megfelelően elvégzi a teendőt az adattal
4. Odasétál a számlálóhoz és továbblöki, így folytathatja a következő utasítás kikeresésével.

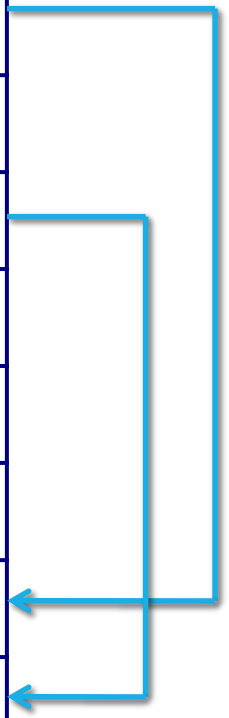
# Példaprogram

Adjuk meg két szám különbségét

- Olvassuk be az első számot
- Tároljuk el
- Olvassuk be a második számot
- Tároljuk el
- Vonjuk ki belőle az elsőt
- Írjuk ki az eredményt
- Fejezzük be a munkát

# LMC gépi kód

| Postaláda | Kód |
|---|---|
| 00 | 901 |
| 01 | 307 |
| 02 | 901 |
| 03 | 308 |
| 04 | 207 |
| 05 | 902 |
| 06 | 000 |
| 07 | 000 |
| 08 | 000 |



# LMC gépi kód

| Postaláda | Kód | Utasítás leírása |
|---|---|---|
| 00 | 901 | Első szám beolvasása |
| 01 | 307 | Szám tárolása |
| 02 | 901 | Második szám beolvasása |
| 03 | 308 | Szám tárolása |
| 04 | 207 | Az első kivonása a 2.-ból |
| 05 | 902 | Eredmény kiírása |
| 06 | 000 | Stop |
| 07 | 000 | Adatrekesz |
| 08 | 000 | Adatrekesz |

# Számítsuk ki két szám különbségének abszolút értékét!

1. Olvassuk be az első számot
2. Tároljuk el
3. Olvassuk be a másik számot
4. Tároljuk el
5. Vonjuk ki belőle az elsőt
6. Ha pozitív az eredmény, folytassuk a 9.–nél
7. (Különben) töltsük be az első számot
8. Vonjuk ki belőle a másodikat
9. Írjuk ki az eredményt
10. Fejezzük be a munkát

# Program vezérlés

## Elágazás utasítások

- Az utasítások egymásutánjából való kiugrás lehetősége
- Megváltoztatja a címet az utasítás számlálóban

## Leállás

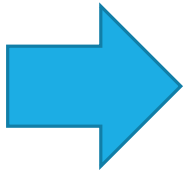
| | Tartalom | |
|---|---|---|
| | Műv. kód | Operandus (cím) |
| Ugrás az xx számú postafiókra | 6 | xx |
| Ha a számológép tartalma 0, ugrás az xx számú postafiókra | 7 | xx |
| Ha a számológép tartalma >0, ugrás az xx számú postafiókra | 8 | xx |

# LMC utasításkészlete

| | |
|---|---|
| Hozzáadás | 1xx |
| Kivonás | 2xx |
| Tárolás | 3xx |
| Betöltés | 5xx |
| Ugrás | 6xx |
| Ugrás 0-nál | 7xx |
| Ugrás pozitívnál | 8xx |
| Input | 901 |
| Output | 902 |
| Stop | 000 |

# Számítsuk ki két szám különbségének abszolút értékét!

| Postaláda | Kód |
|---|---|
| 00 | 901 |
| 01 | 307 |
| 02 | 901 |
| 03 | 308 |
| 04 | 207 |
| 05 | 902 |
| 06 | 000 |
| 07 | 000 |
| 08 | 000 |



| Postaláda | Kód |
|---|---|
| 00 | 901 |
| 01 | 310 |
| 02 | 901 |
| 03 | 311 |
| 04 | 210 |
| 05 | 808 |
| 06 | 510 |
| 07 | 211 |
| 08 | 902 |
| 09 | 000 |
| 10 | 000 |
| 11 | 000 |

# Számítsuk ki két szám különbségének abszolút értékét!

| | | |
|---|---|---|
| 00 | 901 | Első szám beolvasása |
| 01 | 310 | Szám tárolása |
| 02 | 901 | Második szám beolvasása |
| 03 | 311 | Szám tárolása |
| 04 | 210 | Első számot kivonjuk |
| 05 | 808 | Ha pozitív, ugorj a 08-ra (a kiírásra) |
| 06 | 510 | (különben) töltsd be az elsőt |
| 07 | 211 | Vond ki belőle a másodikat |
| 08 | 902 | Írd ki az eredményt |
| 09 | 000 | Stop |
| 10 | 000 | Az  első adat helye |
| 11 | 000 | A második adat helye |

# Nevezzük el az utasításkódokat!

| | | |
|---|---|---|
| Hozzáadás | 1xx | ADD xx |
| Kivonás | 2xx | SUB xx |
| Tárolás | 3xx | STO xx |
| Betöltés | 5xx | LDA xx |
| Ugrás | 6xx | BR xx |
| Ugrás 0-nál | 7xx | BRZ xx |
| Ugrás pozitívnál | 8xx | BRP xx |
| Input | 901 | INP |
| Output | 902 | OUT |
| Stop (Halt) | 000 | HLT |
| Adattároló | | DAT |

# Számítsuk ki két szám különbségének abszolút értékét!

| | | | |
|---|---|---|---|
| 00 | INP | 901 | Első szám beolvasása |
| 01 | STO 10 | 310 | Szám tárolása |
| 02 | INP | 901 | Második szám beolvasása |
| 03 | STO 11 | 311 | Szám tárolása |
| 04 | SUB 10 | 210 | Első számot kivonjuk |
| 05 | BRP 08 | 808 | Ha pozitív, ugorj a 08-ra (a kiírásra) |
| 06 | LDA 10 | 510 | (különben) töltsd be az elsőt |
| 07 | SUB 11 | 211 | Vond ki belőle a másodikat |
| 08 | OUT | 902 | Írd ki az eredményt |
| 09 | HLT | 000 | Stop |
| 10 | DAT 00 | 000 | Az  első adat helye |
| 11 | DAT 00 | 000 | A második adat helye |

# Adjunk nevet a változóknak és az utasításoknak!

| | | | | |
|---|---|---|---|---|
| | 00 | INP | 901 | Első szám beolvasása |
| | 01 | STO ELS | 310 | Szám tárolása |
| | 02 | INP | 901 | Második szám beolvasása |
| | 03 | STO MAS | 311 | Szám tárolása |
| | 04 | SUB ELS | 210 | Első számot kivonjuk |
| | 05 | BRP KI | 808 | Ha pozitív, ugorj a 08-ra (a kiírásra) |
| | 06 | LDA ELS | 510 | (különben) töltsd be az elsőt |
| | 07 | SUB MAS | 211 | Vond ki belőle a másodikat |
| KI | 08 | OUT | 902 | Írd ki az eredményt |
| | 09 | HLT | 000 | Stop |
| ELS | 10 | DAT 00 | 000 | Az első adat helye |
| MAS | 11 | DAT 00 | 000 | A második adat helye |

# Példa

Mit csinál?

```
     INP
     STO ELS
     INP
     ADD ELS
     OUT
     HLT
ELS  DAT 00
```

# Másik példa

És ez mit csinál?

```
      INP
LOOP  SUB  ONE
      OUT
      BRZ  QUIT
      BR   LOOP
QUIT  HLT
ONE   DAT  1
```

# Assembly nyelv

Jellemzői:

- CPU specifikus
- 1 - 1 megfelelés az assembly nyelvű utasítás és a gépi nyelvű utasítás között
  - makró assembly később alakult ki
- Mnemonikok reprezentálják az utasításokat
- Változók deklarálhatók
- Utasítások címkézhetők

Assembler lefordítja

- inputja az assembly nyelvű program
- outputja a gépi nyelvű

# Végeztünk?

# Assembly

Láthattuk, hogy bár az assembly mindenre jó, mégsem célszerű.

Szembetűnően hiányzik:

- Átláthatóság
  - Absztrakció hiánya
- Hordozhatóság
  - CPU, környezet specifikus
- Újrafelhasználhatóság
  - Magasabb szintű, nemcsak hívható alprogramok
- …

# Magasabb szintű nyelvek



# Magasabb szintű nyelvek

Láttuk, hogy a gépi kód egyértelmű, utasításokat adunk a CPU-nak

- Az Assembly ennek az olvasmányosabb változata
  - További funkciókkal
- A magasabb szintű nyelvek nyelvi elemei nem feleltethetők meg egy-egy módon ennek, fordítás vagy értelmezés szükséges
- **Szabályok kellenek, amik leírják az adott nyelvet:**
  - Mik a nyelvi elemek
  - Hogyan kell használni
  - Mi az értelmezése ezeknek

Szintaktika és szemantika

# Avram Noam Chomsky (1928-)



Amerikai nyelvész,

A generatív nyelvtan elméletének megalkotója

# Chomsky – Generatív grammatikák

Chomsky nyelvészként a természetes nyelveket vizsgálta

- Leírást adni a természetes nyelvek szabályaira
- Számos elmélet, cikk szerzője

Alapgondolat tömören

- Gyerek – véges számú mondatot hall
  - Képes tetszőleges nyelvtanilag helyes mondat megalkotására
  - Kell legyen egy szabályrendszer minden nyelvben!

A mesterséges nyelvek esetén nagyobb a siker …

# Generatív grammatika – Példa

Egyszerű példa, illusztrációra – kis nyelvtani emlékeztető.

- „ A haszontalan kisfiú ledobta a tányért.”

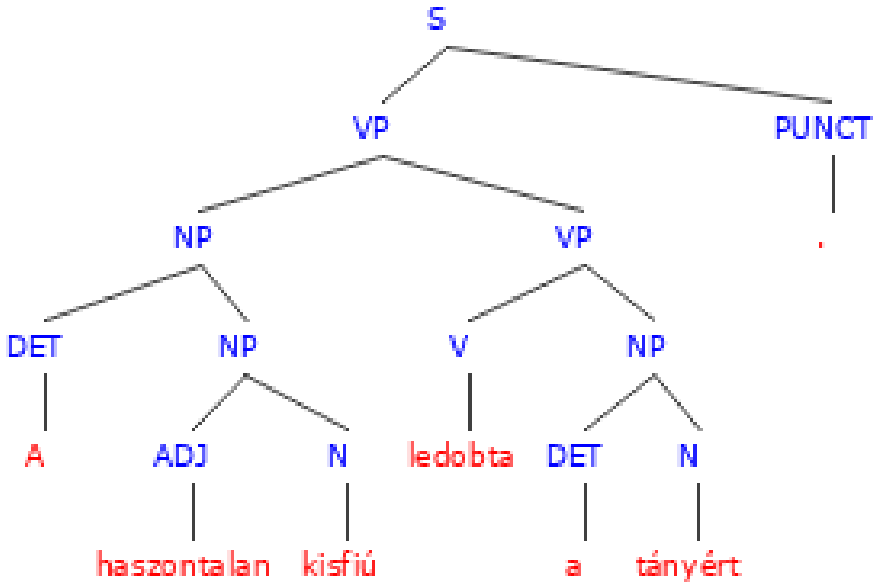
Ez egy mondat, ami felbontható

- Van alanyi és állítmányi részre
  - Az alanyi rész névelő, jelző és alany sorozata lehet
  - Az állítmányi rész az állítmány és a tárgy egymásutánja, stb.

A struktúra leírásakor „grammatikai jeleket” is használhatunk a mondatrészek, illetve szófajok jelzésére

- ezeket < > zárójelek közé tesszük

# Elemzési fa




| | |
|---|---|
| ADJ | melléknév (adjective) |
| DET | névelő (determiner) |
| N | főnév (noun) |
| NP | névszói csoport (noun phrase) |
| PRON | névmás (pronoun) |
| PUNCT | írásjel (punctuation) |
| S | mondat (sentence) |
| V | ige (verb) |
| VP | igei csoport (verbal phrase) |

https://www.webforditas.hu/elemzo

# Példa egyszerűsített szabályokra

| | |
|---|---|
| < S > | → < VP > < PUNCT > |
| < VP > | → < VP > < NP > |
| < VP > | → < V > < NP > |
| < NP > | → < DET > < NP> |
| < NP > | → < DET > < N > |
| < NP > | → < ADJ > < N > |
| | |
| < DET > | → a |
| < ADJ > | → haszontalan |
| < N > | → kisfiú |
| < N > | → tányér(t) |
| < V > | → ledobta |
| < PUNCT > | → . |

| | |
|---|---|
| ADJ | melléknév (adjective) |
| DET | névelő (determiner) |
| N | főnév (noun) |
| NP | névszói csoport (noun phrase) |
| PRON | névmás (pronoun) |
| PUNCT | írásjel (punctuation) |
| S | mondat (sentence) |
| V | ige (verb) |
| VP | igei csoport (verbal phrase) |

# Legbaloldalibb levezetés

Ha adott a fenti módon egy szabályhalmaz, akkor azt használhatjuk mondatok képzésére a következő módon:

- A < S > elnevezésű bal oldalon lévő grammatikai elemből kiindulva behelyettesíthetjük a szabály jobb oldalán álló jelsorozatot
- Majd ennek első elemét vesszük sorra, s írjuk át olyan szabály felhasználásával, ahol megfelelő részt bal oldalon találjuk.
- Ezt az eljárást folytatjuk, amíg még találunk mondatrészt a levezetett jelsorozatban, azaz, amíg át nem térünk a grammatikai egységekről a nyelv szavaira.
- A levezetés során az egyes grammatikai elemeket tetszés szerinti sorrendben helyettesíthetjük.

A következőkben a fenti grammatikai szabályok segítségével oly módon jutunk el az előző példamondatunkhoz, hogy a levezetés során mindig a lehetséges legbaloldalibb helyettesítést választjuk.

# Legbaloldalibb levezetés

< S > → < VP > < PUNCT >

< VP > < PUNCT > → < NP > < VP > < PUNCT >

< NP > < VP > < PUNCT > → < DET > < NP > < VP > < PUNCT >

< DET > < NP > < VP > < PUNCT > → A < NP > < VP > < PUNCT >

A < NP > < VP > < PUNCT > → A < ADJ > < N > < VP > < PUNCT >

A < ADJ > < N > < VP > < PUNCT > → A haszontalan < N > < VP > < PUNCT >

A haszontalan < N > < VP > < PUNCT > → A haszontalan kisfiú < VP > < PUNCT >

A haszontalan kisfiú < VP > < PUNCT > → A haszontalan kisfiú < V > < NP > < PUNCT >

A haszontalan kisfiú < V > < NP > < PUNCT > → A haszontalan kisfiú ledobta < NP > < PUNCT >

A haszontalan kisfiú ledobta < NP > < PUNCT > → A haszontalan kisfiú ledobta < DET > < N > < PUNCT >

A haszontalan kisfiú ledobta < DET > < N > < PUNCT > → A haszontalan kisfiú ledobta a < N > < PUNCT >

A haszontalan kisfiú ledobta a < N > < PUNCT > → A haszontalan kisfiú ledobta a tányért < PUNCT >

A haszontalan kisfiú ledobta a tányért < PUNCT > → A haszontalan kisfiú ledobta a tányért .

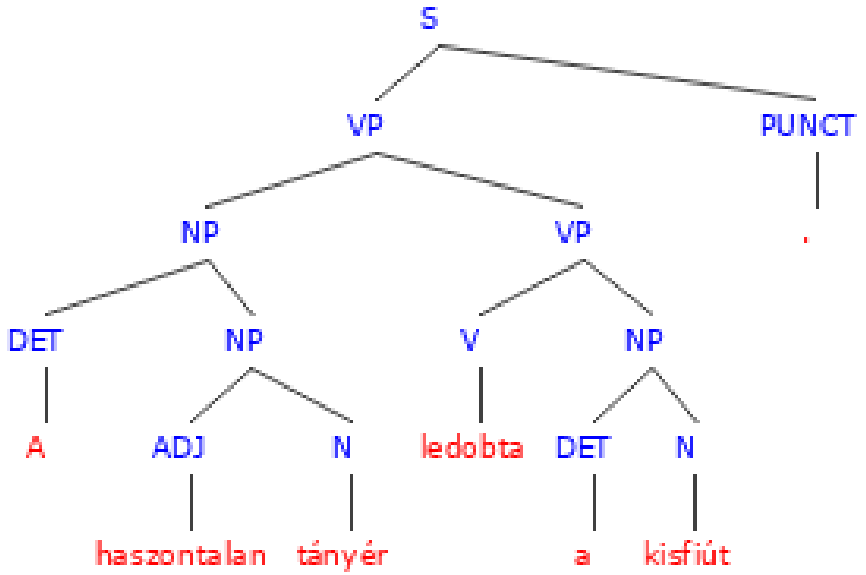
# Legbaloldalibb levezetés

A szabályok szerint lehetséges még:

- A haszontalan tányér ledobta a kisfiút.
- A haszontalan kisfiú ledobta a kisfiút.
- A haszontalan tányér ledobta a tányért.

Nem triviális.

- A továbbiakban szorítkozzunk a mesterséges nyelvekre.



# Mesterséges nyelvek

Magas szintű programozási nyelv esetén szükség van a forrásprogram „fordítására”, vagy „értelmezésére”

- Assemblyre
- Gépi kódra

Ehhez szükséges egy

- Fordítóprogram (compiler), vagy
- Értelmezőprogram (interpreter)

# A programok végrehajtása

Az interpreter egy utasítás lefordítása után azonnal végrehajtja azt

A fordítóprogram átalakítja a programot egy vele ekvivalens formára

- ez lehet a számítógép által (majdnem) közvetlenül végrehajtható forma, vagy
- lehet egy másik programozási nyelv

# Fordítás

A lefordítandó program a forrásprogram.

- Source program / sorce code

A fordítás eredményeként kapott program a tárgyprogram.

- Object program / object code

Az az idő, amikor az utasítások fordítása folyik, a fordítási idő.

- Compilation time

Az az idő, amikor az utasítások végrehajtása folyik, a végrehajtási idő.

- Execution time / runtime

# Szabályok

Szabályok kellenek, amik leírják

- Mik a nyelvben használható jelek (karakterek)
- Milyen nyelvi elemek vannak (formailag helyes karaktersorozatok)
- Jelentés szempontjából mik a helyes karaktersorozatok

A forráskódot író és az azt fordítóprogram ugyanazt kell, hogy értse a leírt kódon

- A gép a szabályok szerint értelmezett parancsainkat teljesíti, nem a kívánságainkat.

# Általános fogalmak

A nyelv szintaxisa azoknak a szabályoknak az összessége, amelyek az adott nyelven írható összes lehetséges, formailag helyes programot (jelsorozatot) definiálják.

- Reguláris kifejezések, BNF forma

Az adott nyelv programjainak jelentését leíró szabályok összessége a nyelv szemantikája.

# Szintaxis és szemantika

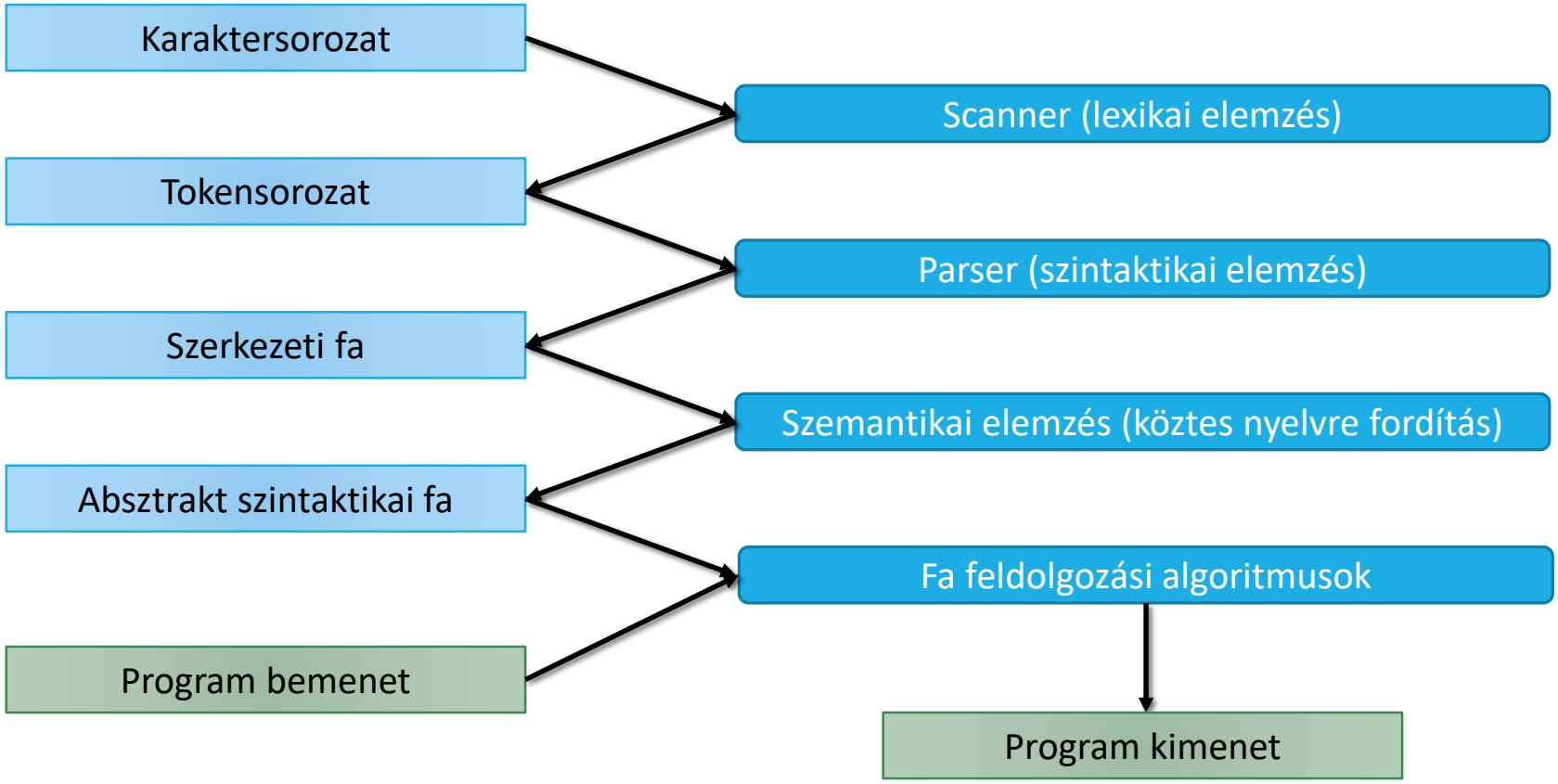
Példa:

- DD / DD / DDDD
- 02 / 11 / 2019

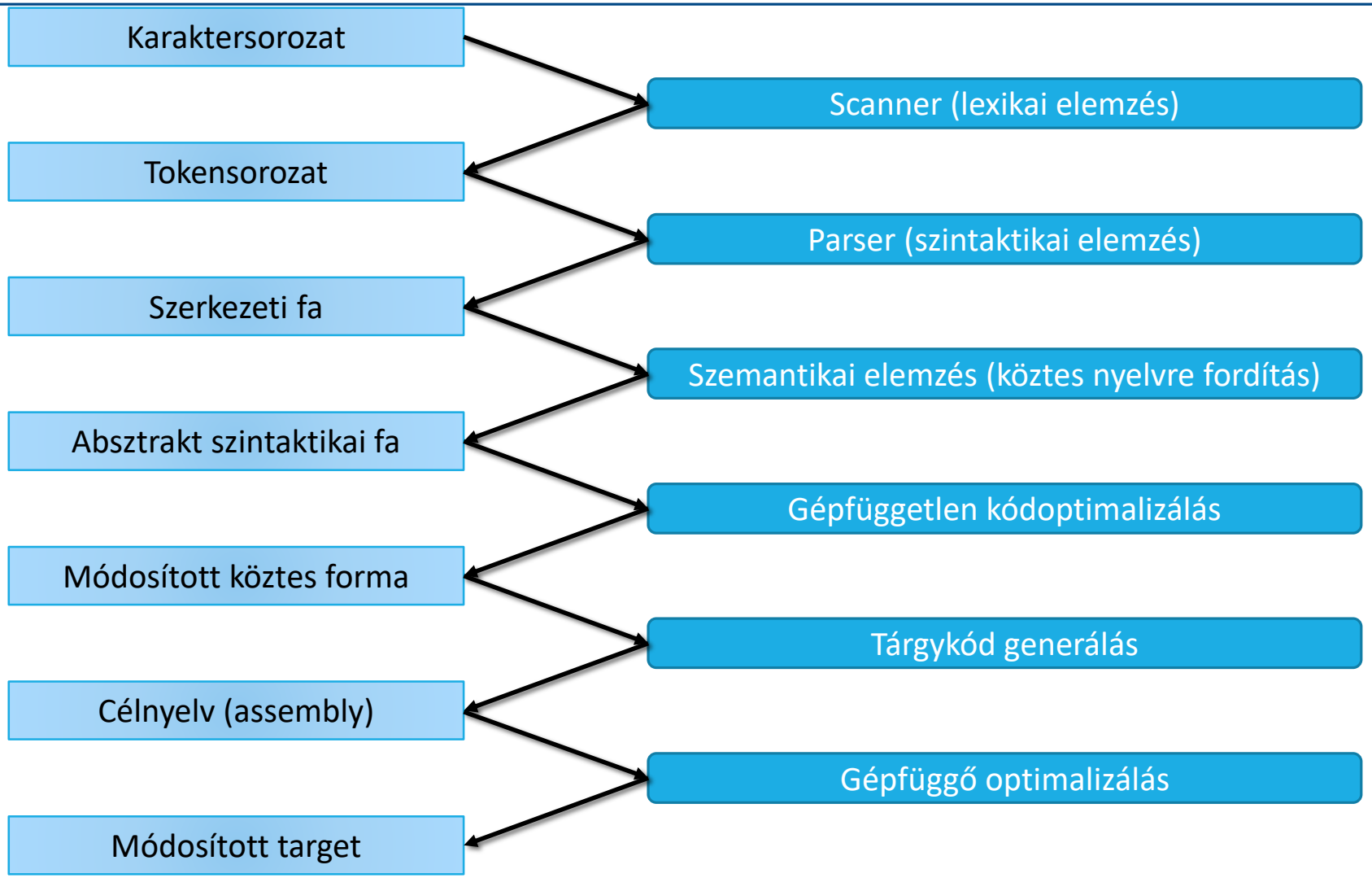
2019. február 11. vagy 2019. november 2.?

A szintaxis befolyásolja a programok megbízhatóságát

# Interpretálás



# Fordítás


Karaktersorozat
Scanner (lexikai elemzés)
Tokensorozat
Parser (szintaktikai elemzés)
Szerkezeti fa
Szemantikai elemzés (köztes nyelvre fordítás)
Absztrakt szintaktikai fa
Gépfüggetlen kódoptimalizálás
Módosított köztes forma
Tárgykód generálás
Célnyelv (assembly)
Gépfüggő optimalizálás
Módosított target

# Nyelv leírása szabályokkal

# Nyelv leírása a fordítónak

Formális nyelv

- Grammatika – generálja a nyelvet
  - A szabályok alapján a nyelv szerinti „mondatok” hozhatók létre
- Automaták – felismerik
  - A generált „mondatok” elemzése, ellenőrzése, feldolgozása

# Formális nyelvek – definíciók

**Σ:** az **ábécé** tetszőleges szimbólumok véges, nem üres halmaza

A **szó** az ábécé elemeiből képezett $a_1a_2...a_k$ alakú sorozat, ahol $k \geq 0$, $a_1, a_2, ..., a_k \in \Sigma$

A **Σ*** a $\Sigma$ ábécé elemeiből képezhető összes szavak halmaza:

- $\Sigma^* = \{a_1a_2...a_k \mid k \geq 0, a_1, a_2, ..., a_k \in \Sigma\}$
- Ha $k=0$, akkor a szó üres szó, jele $\varepsilon$ vagy $\lambda$

$\Sigma^+$ a nem üres szavak halmaza:

- $\Sigma^+ = \Sigma^* \setminus \{\lambda\}$

# Példa

$$\Sigma = \{a, b\}$$

$$\Sigma^* = \{\lambda, a, b, aa, ab, ba, bb, aaa, aab, \dots\}$$

$$\Sigma^+ = \Sigma^* - \lambda$$

# Grammatikák – definíciók

**Generatív nyelvtan (G)**

$G=(N, \Sigma, S, P)$, ahol

$N$ a nemterminálisok (grammatikai jelek) ábécéje

$\Sigma$ a terminálisok abc-je, $N \cap \Sigma = \varnothing$

$S \in N$ a kezdőszimbólum

$P$: $\alpha \rightarrow \beta$ alakú átírási szabályok véges halmaza

$\alpha$ a szabály baloldala, $\beta$ a szabály jobboldala

$\alpha, \beta \in (N \cup \Sigma)^*$

$\alpha$-ban van legalább egy nemterminális szimbólum

# Magyarázat

## Nemterminális

- Nem „célnyelvi” szavak, hanem a levezetési szabályok még behelyettesíthető szimbólumai
  - < ADJ > < NP > < PUNCT>

## Terminális

- „Célnyelvi” szavak, nem helyettesíthetők be tovább
  - Kisfiú tárnyért

## Kezdőszimbólum

- A startszimbólum, ezzel kezdjük meg a levezetést

# Grammatikák – definíciók

**Közvetlen levezetés (közvetlen levezetési reláció, $\Rightarrow G$)**

Legyen $\gamma, \delta \in (N \cup \Sigma)^*$

$\gamma \Rightarrow G\ \delta$, ha van olyan $\alpha \rightarrow \beta \in P$ szabály és

vannak olyan $\alpha', \beta' \in (N \cup \Sigma)^*$ szavak, hogy

$\gamma = \alpha'\alpha\beta'$ és $\delta = \alpha'\beta\beta'$

- $\Rightarrow G^n$ - n lépéses levezetés
- $\Rightarrow G^+$ - legalább egy lépéses levezetés
- $\Rightarrow G^*$ - levezethetőség (esetleg nulla lépésben is)

# Grammatikák – definíciók

## G grammatika által generált nyelv L(G)

- Legyen G=(N, $\Sigma$, S, P) generatív nyelvtan.
- L(G)={u$\in\Sigma^*$ | S $\Rightarrow_G^*$ u}
- L(G) = a G generatív grammatika által generált nyelv
  - összes lehetséges szó

## Ekvivalens nyelvtanok

- Legyenek G=(N, $\Sigma$, S, P) és G'=(N', $\Sigma$', S', P') generatív nyelvtanok.
  - G ekvivalens G'-vel, ha L(G)=L'(G').
  - Ekvivalens két nyelvtan, ha ugyanazt a nyelvet generálja

# Chomsky-féle nyelvosztályok

## 0. típusú nyelvtan (kifejezés struktúrájú)

- ha semmilyen korlátozás nincs
- Turing gép tudja feldolgozni

## 1. típusú nyelvtan (környezetfüggő)

- ha P-ben minden szabály $\alpha A\beta \rightarrow \alpha\delta\beta$ alakú
  - Itt $\delta \neq \lambda$, kivéve esetleg az $S \rightarrow \lambda$ szabályt, mely esetben $\alpha=\beta=\delta=\lambda$, de ekkor S nem szerepelhet egyetlen szabálynak sem a jobboldalán.
  - $(\alpha,\beta,\gamma,\delta \in (N \cup \Sigma)^*)$
- Turing gép tudja feldolgozni
- *Környezettfüggőség: a behelyettesítésnél figyelembe kell venni a behelyettesítendő szimbólum környezetében (szomszédságában) levő további szimbólumokat*

# Chomsky nyelvosztályok (folyt.)

## 2. típusú nyelvtan (környezetfüggetlen)

- ha P-ben minden szabály $A \rightarrow \alpha$ alakú
- ($A \in N$, $\alpha \in (N \cup \Sigma)^*$)
- Nemdeterminisztikus veremautomata tudja feldolgozni
- *Környezettfüggőség: a behelyettesítésnél nincs meghatározva megkötés a behelyettesítendő szimbólum környezetében levő további szimbólumokra*

## 3. típusú nyelvtan (reguláris, jobblineáris)

- ha P-ben minden szabály $A \rightarrow xB$, vagy $A \rightarrow x$ alakú.
- ($A, B \in N$, $x \in \Sigma^*$)
- Véges determinisztikus automata tudja feldolgozni

# Chomsky nyelvosztályok

**Nyelv típusa**

Egy $L \subseteq \Sigma^*$ nyelv i típusú (i=0,1,2,3),
ha van olyan i típusú nyelvtan, ami éppen L-et generálja.

$$\mathcal{L}_0 \supset \mathcal{L}_1 \supset \mathcal{L}_2 \supset \mathcal{L}_3$$

# Példák

Reguláris grammatika: $G = (\{S\}, \{a,b\}, S, P)$

P: $S \rightarrow abS$

$S \rightarrow a$

$S \Rightarrow a$

$S \Rightarrow abS \Rightarrow aba$

$S \Rightarrow abS \Rightarrow ababS \Rightarrow ababa = (ab)^2a$

$S \Rightarrow abS \Rightarrow ababS \Rightarrow abababS \Rightarrow abababa = (ab)^3a$

$S \Rightarrow abS \Rightarrow ababS \Rightarrow abababS \Rightarrow abababababS \Rightarrow abababababa = (ab)^4a$

$L(G) = \{(ab)^n a \mid \text{ahol } n \geq 0\}$

$L(G) = (ab)^*a$

# Példák

Környezetfüggetlen grammatika: $G = (\{S\}, \{a,b\}, S, P)$

P: $S \rightarrow aSb$

$S \rightarrow \lambda$

$S \Rightarrow \lambda$

$S \Rightarrow aSb \Rightarrow ab$

$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb = a^2b^2$

$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaabbb = a^3b^3$

....

$L(G) = \{a^n b^n \mid \text{ahol } n \geq 0\}$

# Példák és problémák

# Reguláris kifejezések

Definíció: Egy $S$ abc feletti reguláris kifejezések:

1. $S$ minden eleme.
2. $\lambda$ (vagy $e$)
3. $\emptyset$
4. Ha $a$ és $b$ reguláris kifejezések, akkor $(ab)$ is az.
5. Ha $a$ és $b$ reguláris kifejezések, akkor $(a \cup b)$ is az.
6. Ha a reguláris kifejezés, akkor $a^*$ is az.

Minden reguláris kifejezés reprezentál egy nyelvet.
Példa reguláris kifejezésre és nyelvre:

$(a^* bc)\ ^*ab$ $(\{a\}^*\{b\}\{c\}\ )^*\{a\}\{b\}$

# Reguláris kifejezések

Minták megadása – „hagyományos” jelölések a metakarakterekre:

- Karakter megfeleltetés
  - . (pont) Bármilyen karakter
    - Pl. .a. ⇔ ‘pap’, ‘rak’, ‘lap’, ‘kap’, stb.
  - [karakterek] A szögletes zárójelek között felsorolt karakterek valamelyikével megegyező karakter:
    - Pl. [abc] ⇔ ‘a’ vagy ‘b’ vagy ‘c’
    - A - (mínusz) jellel tartományt is megadhatunk.
    - Például [0-9] megfelel bármely számjegynek vagy [a-zA-Z] bármely kis vagy nagybetűnek.
  - [^karakterek] az előző operátor tagadása
    - Pl. [^a-d] : bármely karakter, ami nem az ‘a’, ‘b’, ‘c’ vagy ‘d’

# Reguláris kifejezések

- \d rövidítés: számjegyek 0-9
- \w rövidítés: betűk, számjegyek, ‘_’
- \s rövidítés: whitespace (space,tab,CR,LF)
- \D, \W, \S rövidítések: a fentiek negációi

Többszörözés - ismétlő operátorok a minták után:

- ? : 0 vagy 1 egyezés - pl. \d? ⇔ ∅, vagy egy számjegy
- * : 0 vagy több egyezés - pl. ab*a ⇔ aa, aba, abba, …
- + : 1 vagy több egyezés - pl. [0-9]+ ⇔ 1, 45, 460, ...
- {n} : n egyezés (n >= 1) - pl. .{3} ⇔ aaa, 888, ...
- {n,m} : n és m közötti egyezés (n>=1, n<=m) –
  pl. a{2,4} ⇔ ‘aaaa’, ‘aaa’ vagy ‘aa’
- {n,} : legalább n egyezés (n>=1) -pl. xa{2,}. ⇔ xaaf, xaaa, ...
- {,m}:  legfeljebb m egyezés

# Reguláris kifejezések

Diszjunkció

- „|” (pipe) szimbólum két minta között: „vagy”

  pl. a|A ⇔ ‘a’, ‘A’

- lehet sorozatban több is:

  pl. a|b|c ugyanaz, mint [a-c]

- a „|” -jel precedenciája a legalacsonyabb az operátorok között. Ezért:

  te|i ⇔ ‘te’, ‘i’

  t(e|i) ⇔ ‘te’, ‘ti’

# Reguláris kifejezések

## Horgonyok

- ^ : az elejére illeszkedik
  pl. ^M : <‘M’ a kezdete>
- $ : a  végére illeszkedik:
  pl. \.$ : <pont a végén>
- \b : szóhatárra illeszkedik (\w és \W karakterek közötti pozícióra, vagy \w és sztring kezdete/vége közé)
  pl. c\b : ‘abc’-re illeszkedik ‘abc  def’-ben
- \B: \b negáltja (karakter „szó” közepén)

## Speciális karakterek csak ‘\’ segítségével használhatók

- Pl. \. vagy \-

# Reguláris kifejezések

Csoportok

- A zárójel operátorokkal tetszőleges számú csoport kijelölhető a mintában:

  pl. a((bla)ko(k))(a)t
- A csoportokra alkalmazhatók az operátorok:

  pl. a(lm|kn)a

# Használat

Bármilyen keresésnél, formai ellenőrzésnél:

- Google-ben, ha nem tudjuk biztosan, vagy több írásmód is létezik:
  - "Handel", "Händel”  vagy "Haendel” ?
  - H(ä|ae?)ndel
- IP-cím formátum:
  - <3-jegyű_szám>.<3-jegyű_szám>.<3-jegyű_szám>.<3-jegyű_szám>
  - RegKif: \b\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}\b

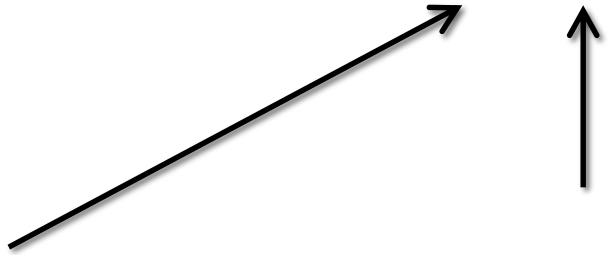
Fordítóprogramokban a lexikális elemző (lexer)

- Eldönthető, hogy érvényes szimbólumokból állnak-e a szavak
- változónév: [a-zA-Z][a-zA-Z0-9]*
- egész szám: (\+|\-)?[0-9]+
- valós szám: [0-9]+\.[0-9]*

# Context-Free Grammatika

$G=(N, \Sigma, S, P)$

ha P-ben minden szabály $A \rightarrow \alpha$ alakú.



Grammatikai jel

Grammatikai jelek és terminálisok sorozata

$(A \in N,\ \alpha \in (N \cup \Sigma)^*)$

# A grammatika által generált nyelv

$G$ grammatika

$S$ kezdőszimbólum

$$L(G) = \{w : \quad S \overset{*}{\Rightarrow} w, \quad w \in \Sigma^*\}$$

Terminálisok sorozata vagy a λ

# Context-Free nyelv

Egy L nyelv  környezetfüggetlen (CF) ha van egy G context-free grammatika, ahol

$$L = L(G)$$

# Levezetési fa

Definíció: Egy G CF grammatika szerinti levezetés egy fával ábrázolható

- A gyökere a kezdőszimbólum
- A levelek balról jobbra a levezetett szó betűi
- További csomópontjai nemterminálisok
- Az elágazások egy-egy alkalmazott szabálynak felelnek meg

Ez a levezetési fa.

# Példa

Context-free grammatika: $G$

$$S \to aSa \mid bSb \mid \lambda$$

Példa levezetések:

$$S \Rightarrow aSa \Rightarrow abSba \Rightarrow abba$$

$$S \Rightarrow aSa \Rightarrow abSba \Rightarrow abaSaba \Rightarrow abaaba$$

---

$$L(G) = \{ww^R : \quad w \in \{a,b\}^*\}$$

Páros hosszú palindromák

# Másik példa

Context-free grammatika : $G$

$$S \to aSb \mid SS \mid \lambda$$

Példa levezetések:

$$S \Rightarrow SS \Rightarrow aSbS \Rightarrow abS \Rightarrow ab$$

$$S \Rightarrow SS \Rightarrow aSbS \Rightarrow abS \Rightarrow abaSb \Rightarrow abab$$

---

$$L(G) = \{w \ : n_a(w) = n_b(w),$$

$$\text{és } \ n_a(v) \geq n_b(v)$$

$$\text{a } \ w \ \textit{bármely} \ v \ \textit{prefixében}\}$$

Leírja a jó zárójelezést: () ((( ))) (( ))

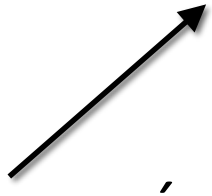
$a = ($, b $= )$

# A matematikai kifejezések grammatikája

$$E \rightarrow E + E \mid E * E \mid (E) \mid a$$

Példa sorozatok:

$$(a + a) * a + (a + a * (a + a))$$



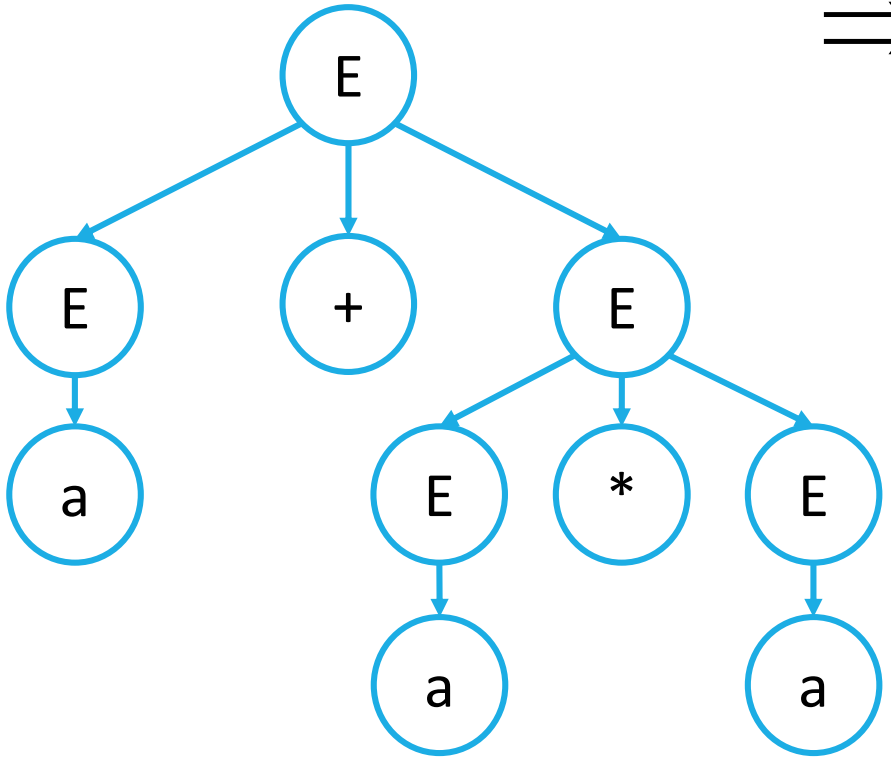
Tetszőleges szám

$$E \rightarrow E + E \mid E * E \mid (E) \mid a$$

$$E \Rightarrow E + E \Rightarrow a + E \Rightarrow a + E * E$$

$$\Rightarrow a + a * E \Rightarrow a + a * a$$

$$a + a * a$$




egy legbaloldalibb levezetése

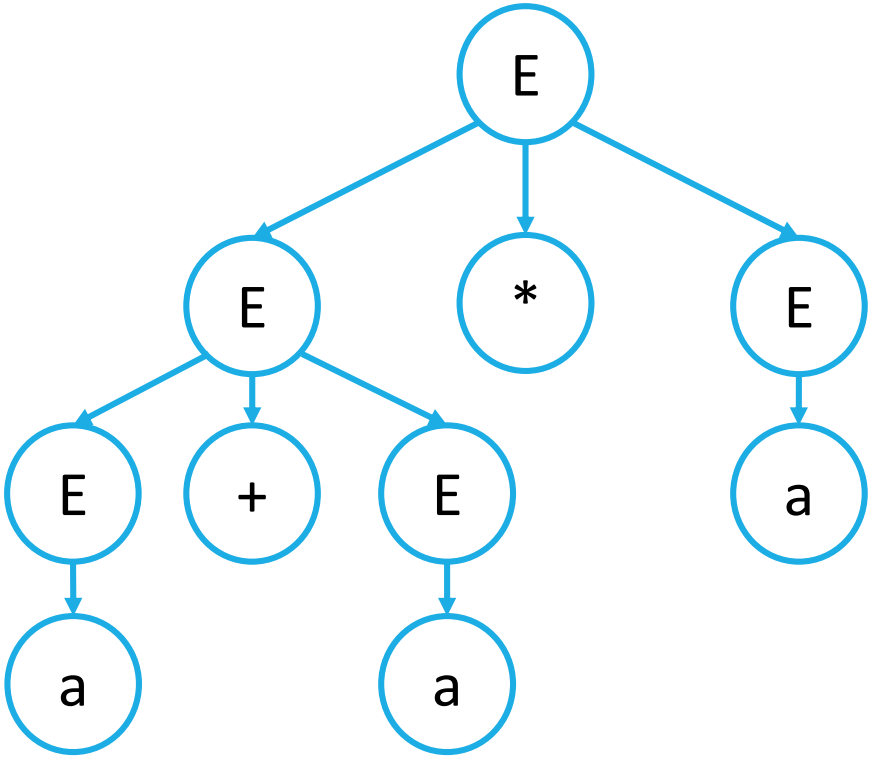
$$E \rightarrow E + E \mid E * E \mid (E) \mid a$$

$$E \Rightarrow E * E \Rightarrow E + E * E \Rightarrow a + E * E$$

$$\Rightarrow a + a * E \Rightarrow a + a * a$$

$$a + a * a$$

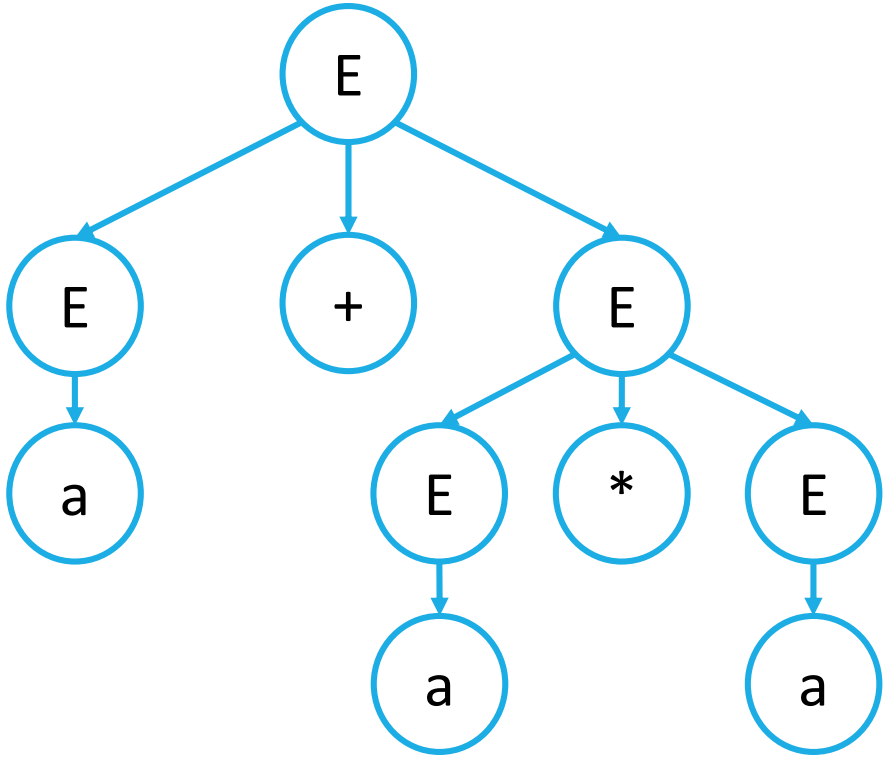
Egy másik legbaloldalibb levezetése

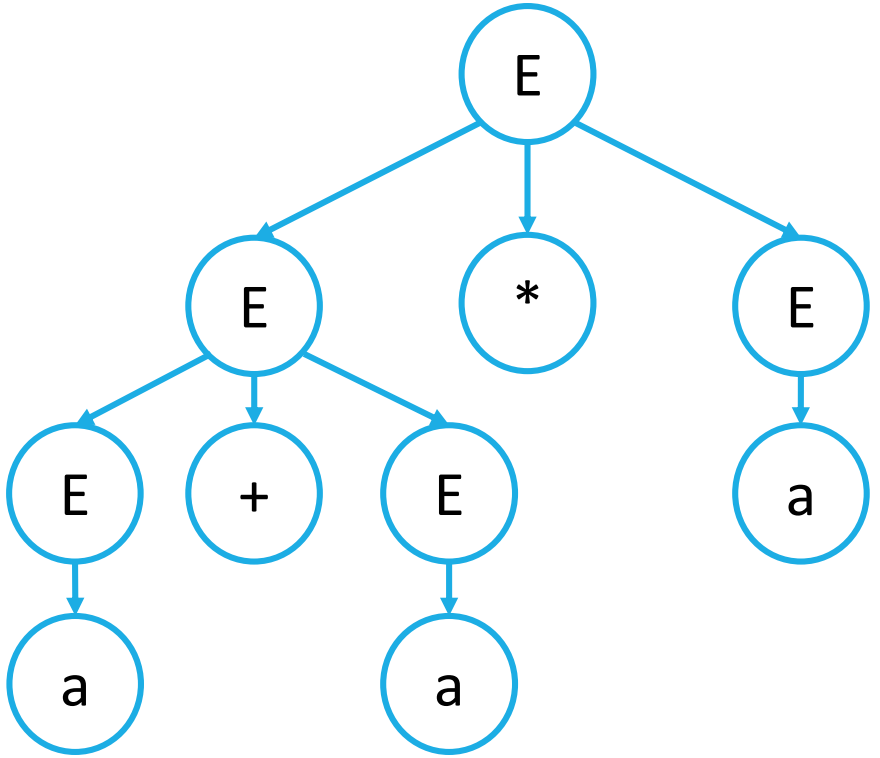


$$E \rightarrow E + E \mid E * E \mid (E) \mid a$$

$$a + a * a$$

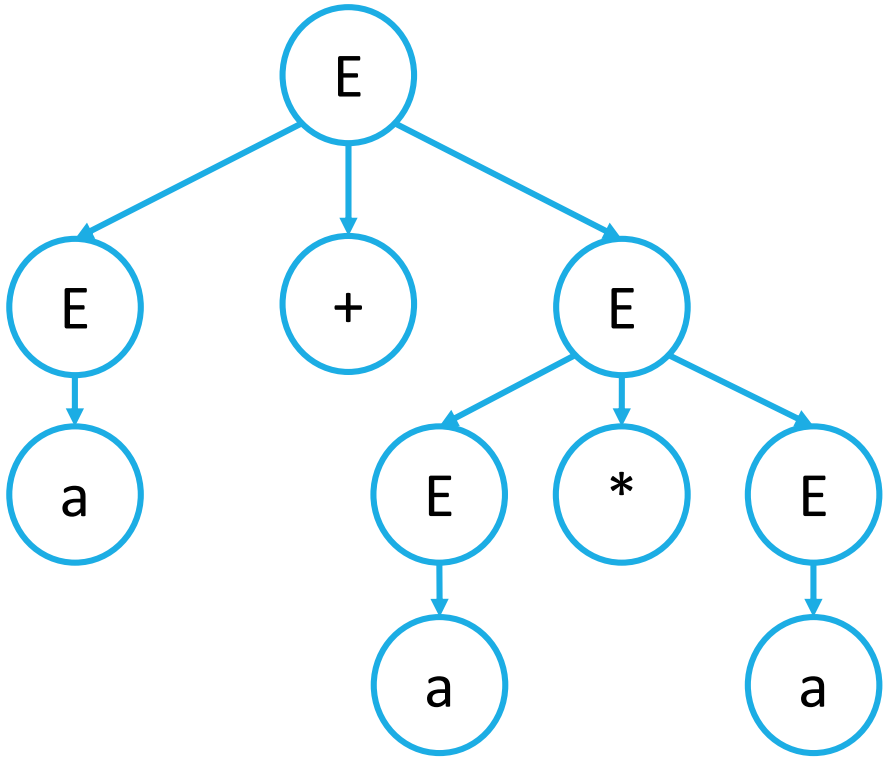
Két levezetési fa!

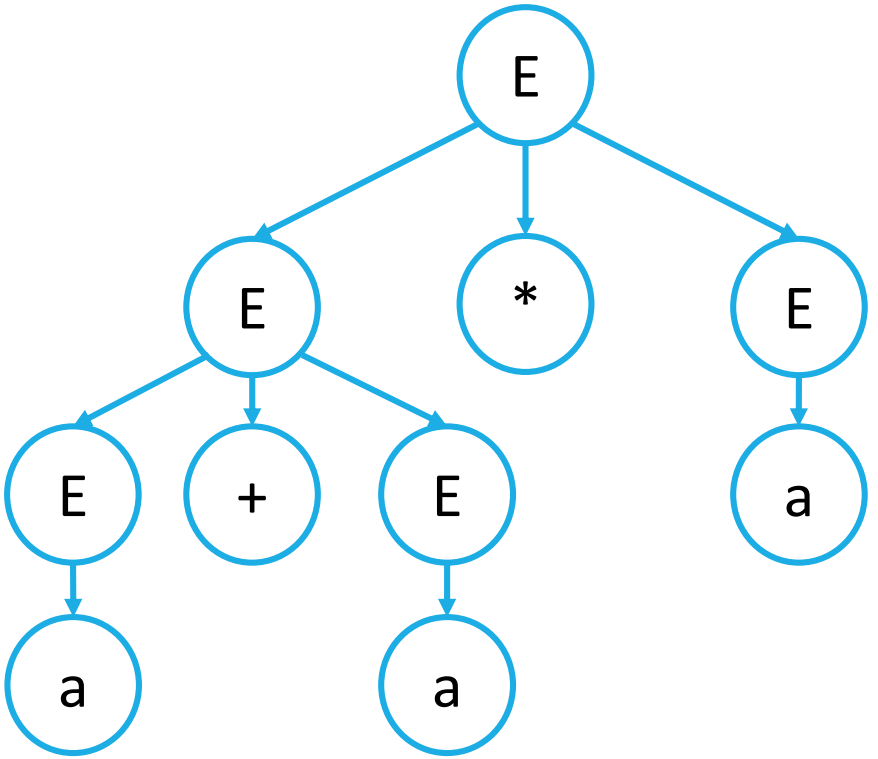


# Legyen a = 2

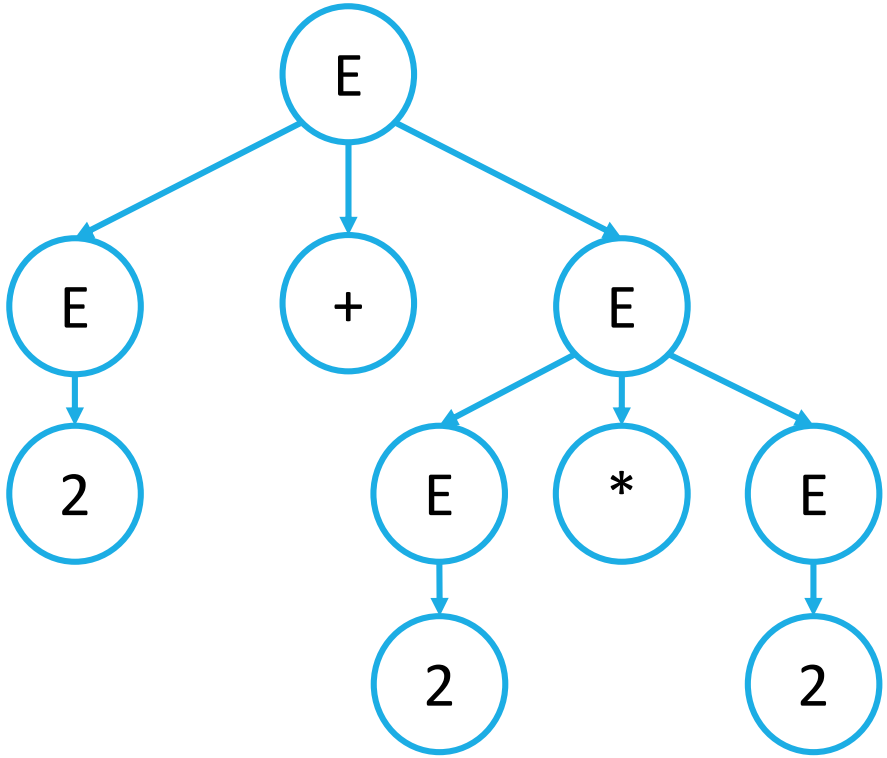
$$a + a * a = 2 + 2 * 2$$



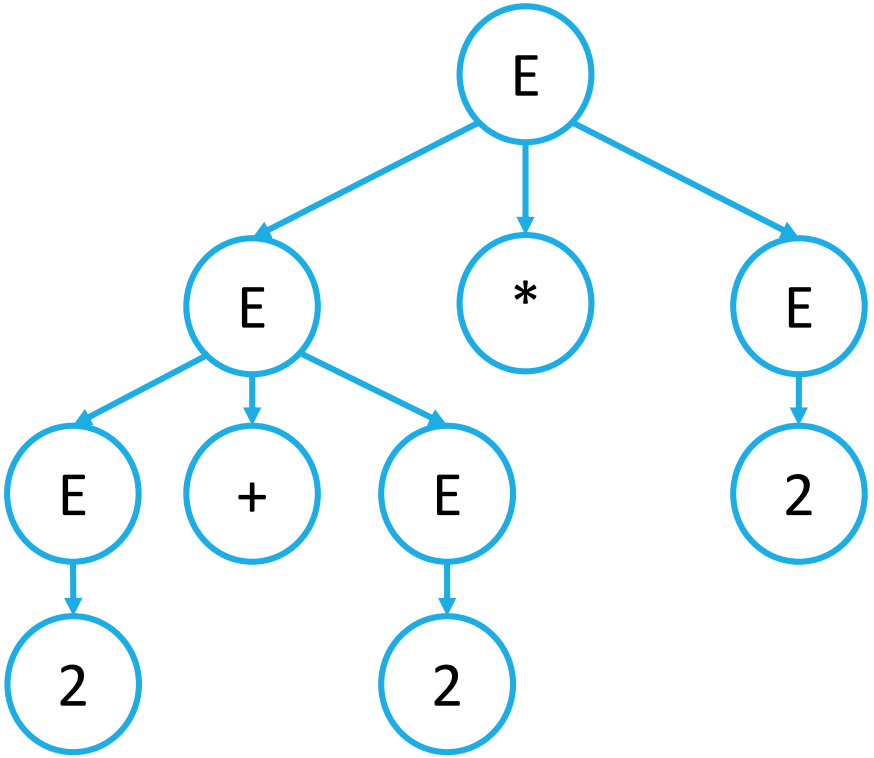
# Legyen a = 2

$2 + 2 * 2 = 6$




Számítsuk ki a kifejezés eredményét a fát használva

$2 + 2 * 2 = 8$



# Többértelmű grammatika

Egy G környezetfüggetlen grammatika **többértelmű**

ha van olyan $w \in L(G)$ amelyiknek:

- két különböző levezetési fája van, vagy
- két különböző legbaloldalibb levezetése van

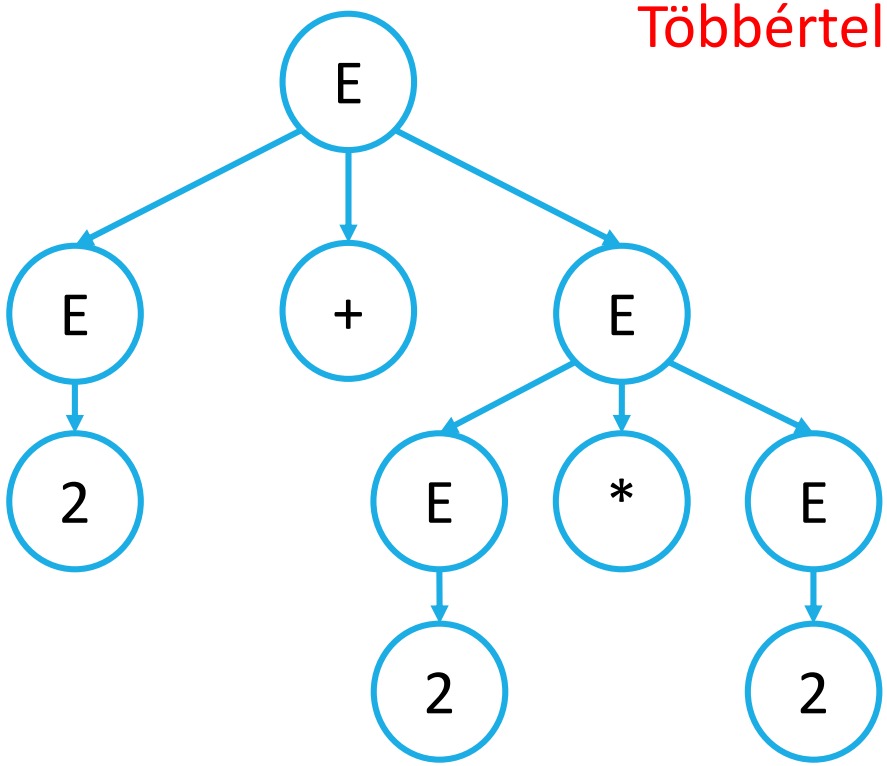
Két különböző levezetési fa problémákat okozhat

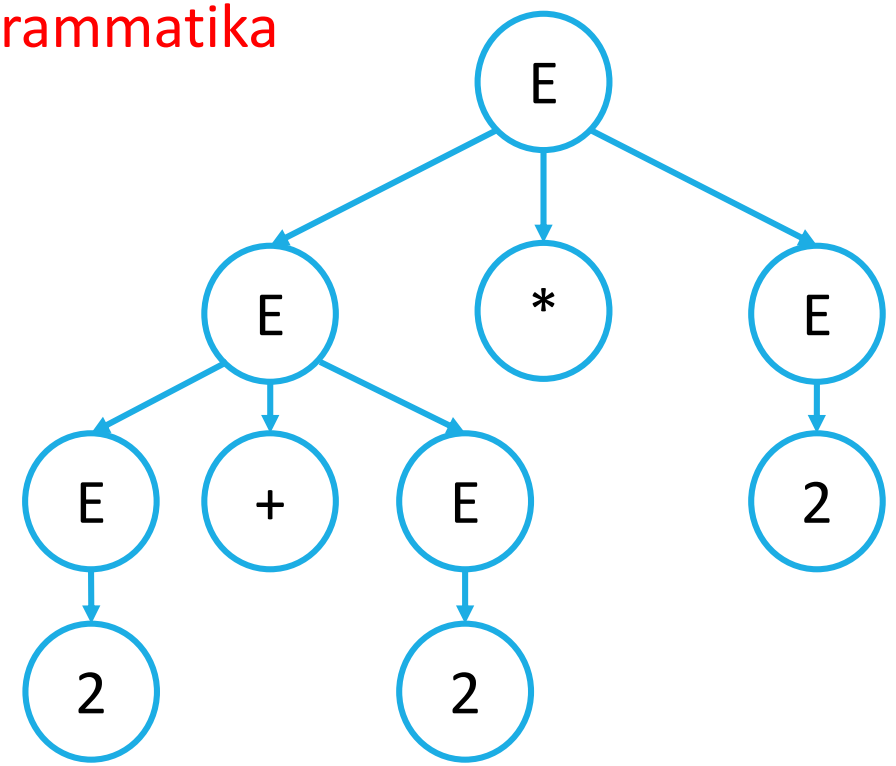
- kifejezések kiértékelésénél
- Általában - programozási nyelvek fordítóprogramjainál!

$$E \rightarrow E + E \mid E * E \mid (E) \mid a$$

$$a + a * a$$

Többértelmű a grammatika

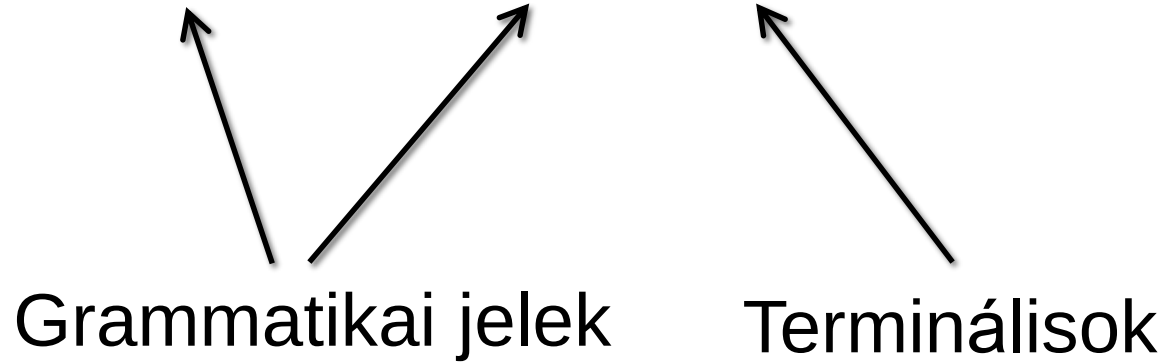


# Egy másik többértelmű grammatika

IF_STMT → **if** EXPR **then** STMT

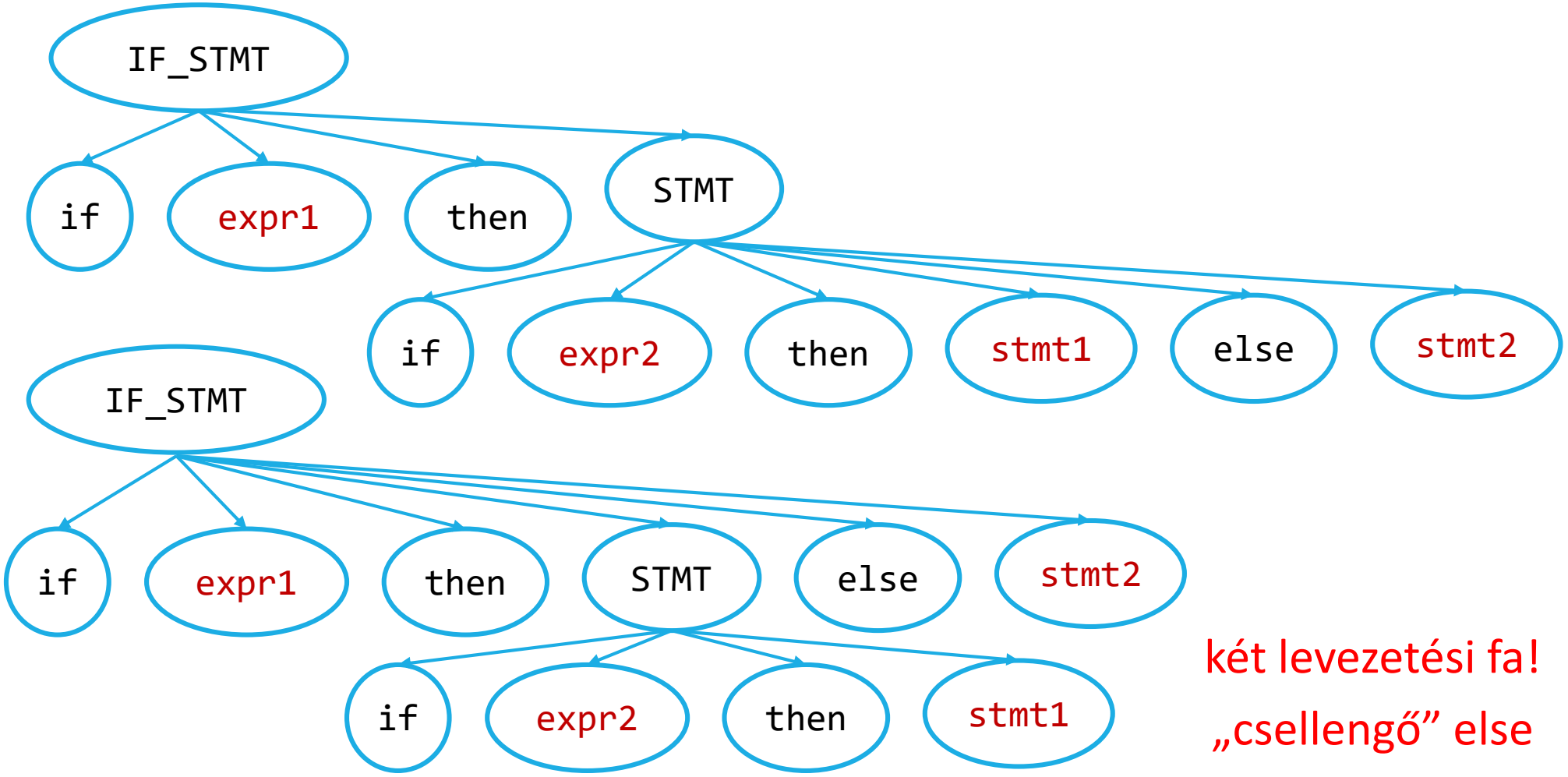
| **if** EXPR **then** STMT **else** STMT




A programozási nyelvek grammatikájának jellemző része

# Egy másik többértelmű grammatika

```
if expr1 then if expr2 then stmt1 else stmt2
```




két levezetési fa!

„csellengő” else

# Többértelműség

Általában a többértelműség rossz, és szeretnénk megszüntetni…

- Néha sikerül találni egyértelmű  (nem többértelmű) grammatikát a nyelvhez
- De általánosságban ezt nehéz elérni...

# Egy sikeres példa

Többértelmű grammatika:

$$E \to E + E$$
$$E \to E * E$$
$$E \to (E)$$
$$E \to a$$

Ekvivalens

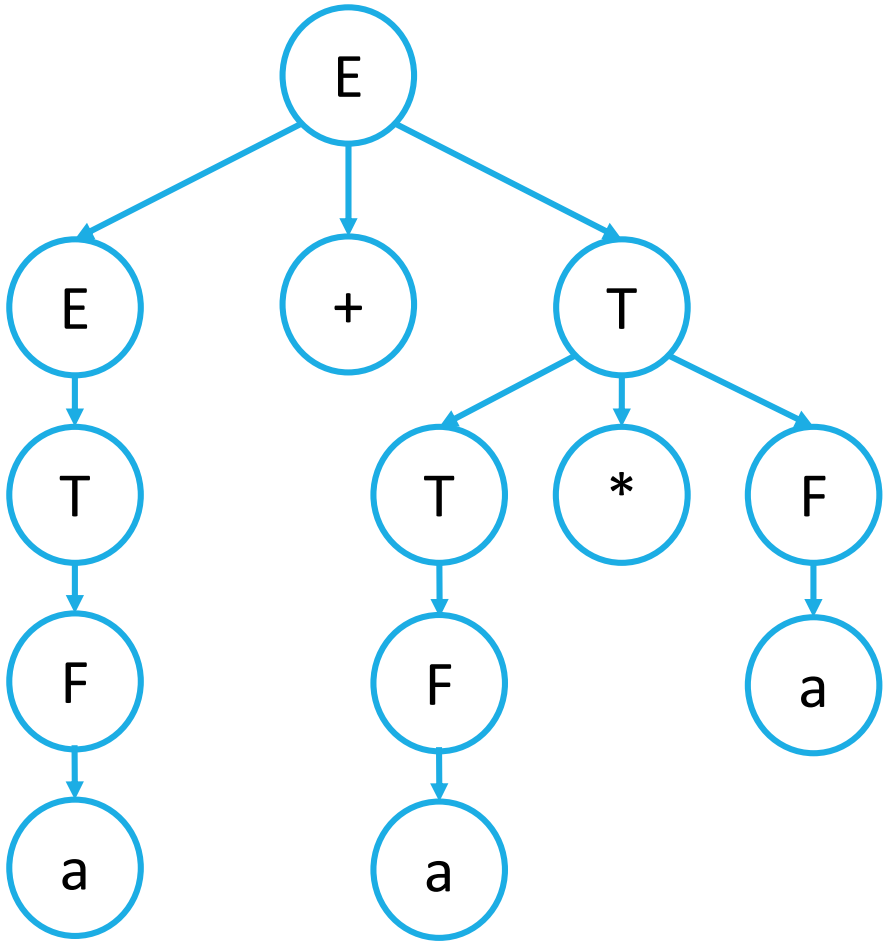
Nem-többértelmű grammatika

Ugyanazt a nyelvet generálja!

$$E \to E + T \mid T$$
$$T \to T * F \mid F$$
$$F \to (E) \mid a$$

$$E \Rightarrow E+T \Rightarrow T+T \Rightarrow F+T \Rightarrow a+T \Rightarrow a+T*F$$

$$\Rightarrow a+F*F \Rightarrow a+a*F \Rightarrow a+a*a$$




$$E \rightarrow E+T \mid T$$

$$T \rightarrow T*F \mid F$$

$$F \rightarrow (E) \mid a$$

$$a+a*a$$

Egyértelmű a levezetési fa

# Programozási nyelv megadása

# Programozási nyelv megadása

Formális szintaxis leírás

- ezt használja a fordítóprogram a nyelvi helyesség ellenőrzésére

Formális szemantika leírás

- a jelentés ellentmondásmentességének ellenőrzésére

# John Backus (1924-2007)



FORTRAN tervezője

"Munkám túlnyomó része egyszerűen a lustaságomból eredt. Amikor a rakéták röppályáját kiszámító programokon dolgoztam, hozzáfogtam egy programozási rendszer kidolgozásához, amelynek elsődleges feladata a munka megkönnyítése, leegyszerűsítése volt”

# Backus-Naur Form (BNF)

Nyelv a programozási nyelvek definiálására

- meta-nyelv, eredetileg 59-60-ból, az Algol és Fortran definiálására
- változatok ma is programozási nyelvek megadására

BNF formában adott szintaxishoz léteznek eszközök:

- automatikusan generálnak egy elemzőt
  - a szintaktikus helyesség ellenőrzésére
  - levezetési fa készítésére
  - adott esetben a program lefordítására

# Backus-Naur Form (BNF)

Példa

```
<loop statement> ::= <while loop> | <for loop>

<while loop> ::= while ( <condition> ) <statement>

<for loop> ::= for ( <expression> ; <expression>;
    <expression> ) <statement>

<assignment statement> ::=
    <variable> = <expression>
```

# BNF Linkek

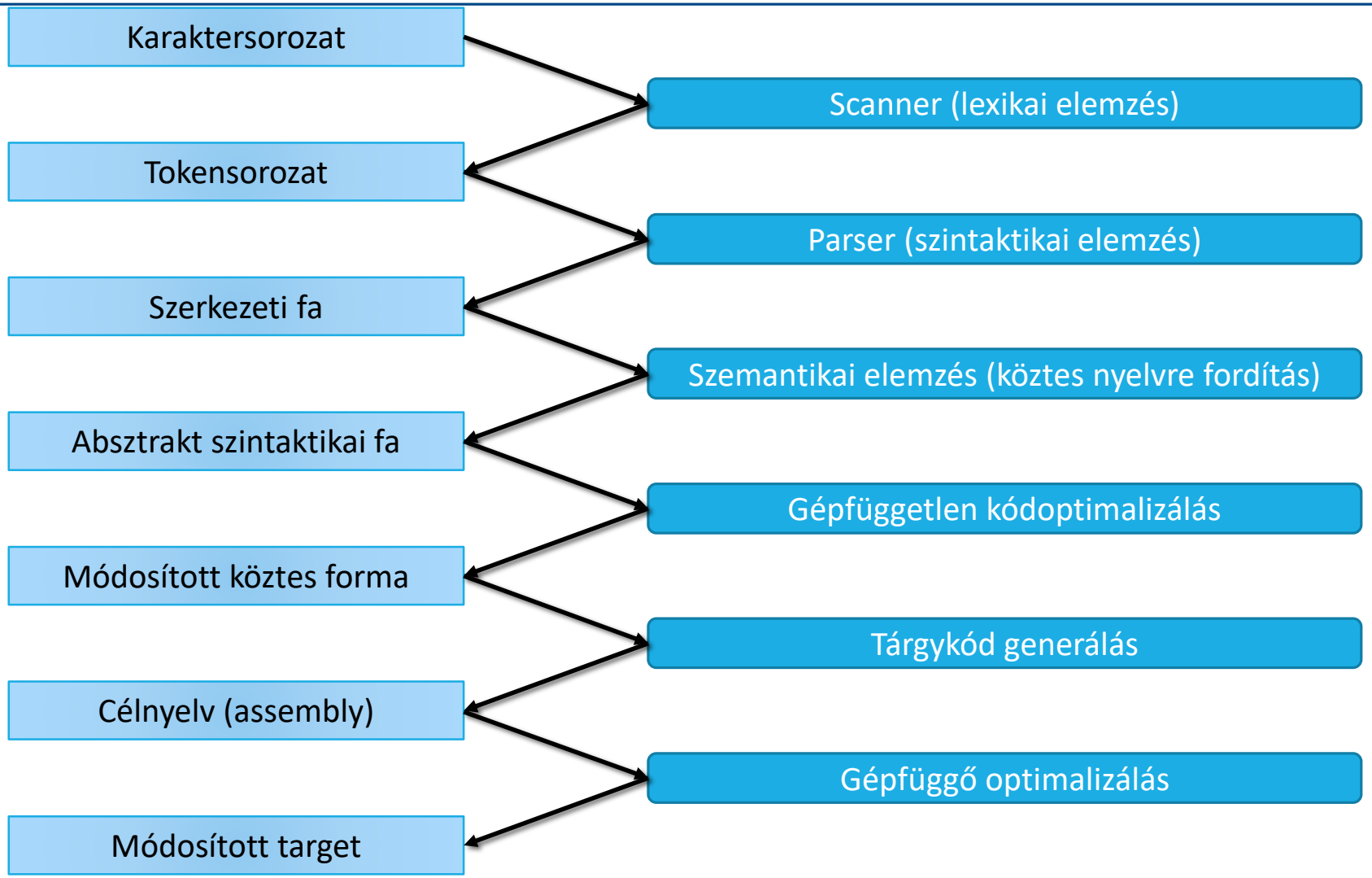
Java BNF

- https://docs.oracle.com/javase/specs/jls/se13/html/jls-19.html

Python BNF

- https://docs.python.org/3/reference/grammar.html

# Fordítás


Karaktersorozat
Scanner (lexikai elemzés)
Tokensorozat
Parser (szintaktikai elemzés)
Szerkezeti fa
Szemantikai elemzés (köztes nyelvre fordítás)
Absztrakt szintaktikai fa
Gépfüggetlen kódoptimalizálás
Módosított köztes forma
Tárgykód generálás
Célnyelv (assembly)
Gépfüggő optimalizálás
Módosított target

# Fordítóprogram

3. Lexikális elemzés
   - megadható reguláris nyelvtannal
2. Szintaktikus elemzés
   - megadható környezetfüggetlen nyelvtannal
1. Szemantikus elemzés
   - környezetfüggő
     - egy változó használatának helyessége függhet a deklarációjától, azaz a környezettől

A három lépés szétválasztásának oka
   - Egyszerű feladathoz ne használjunk bonyolult eszközöket.

# Programszerkezet

## Fordítási egység vagy modul

- A nyelvnek az az egysége, ami a fordítóprogram egyszeri lefuttatásával, a program többi részétől elkülönülten lefordítható
- Több modulból álló programok esetében a fordítás és a végrehajtás között: a program összeszerkesztése



# Lexikai elemek

JÖVŐ HÉTEN




# Programozási nyelvek és módszerek

## LEXIKAI ELEMEK, BEÉPÍTETT TÍPUSOK

# Programegység, fordítási egység

A programegység egy részfeladatot megoldó, tehát funkcionálisan összefüggő programrész.

- Részei
  - A specifikációs rész írja le az egységnek a más egységből elérhető “kapcsolódási pontjait".
  - A törzs tartalmazza az egység funkcióját megvalósító programot.

A fordítási egység programegységek halmaza.

# Ada példa

```
procedure Hello;
```



specifikáció

```
with Text_IO; use Text_IO;

procedure Hello is
   S: String(1..40):=(others=>' ');
   Last: Natural;
begin
   Put("Hogy hívnak?");
   Get_Line(S, Last);
   New_Line;
   Put("Hello "&S(1..Last));
end Hello;
```



törzs

# A törzs felépítése

A deklarációs részben (ha létezik külön)

- azonosítók deklarációit,
- konstansok, típusok, objektumok és programegységek definícióit

Az utasítássorozat részben

- utasításokat, amelyek végrehajtása nyomán a programegység kifejti a hatását.

# Deklarációk

A deklarációs rész feldolgozását, a definíciók értelmezését a deklarációs rész kiértékelésének mondjuk.

A deklarációs rész definícióinak kiértékelése történhet:

- Fordítási (pl, C, C++) vagy
  - statikus definíció-kiértékelés
- Futási (pl. PL/I, Ada) időben
  - dinamikus definíció-kiértékelés

# Deklarációk

Változódeklaráció – sokszor definíció is

- állapottér megadásához
- Formája általában
  - `típus azonosító`
  - `int i;`
- Vagy
  - `azonosító: típus`
  - `N: Natural;`

# Hatáskör, láthatóság

A programszövegnek azt a szakaszát, amelyen belül az adott deklaráció érvényben van, a deklaráció hatáskörének nevezzük.

A programszövegnek azt a szakaszát, ahol a deklarált azonosítóval az adott deklarációra hivatkozni lehet, a deklaráció láthatósági körének nevezzük.

# Blokkstruktúra

A blokkstruktúra a programegységek egymásba ágyazásával előálló hierarchikus struktúra.

Blokkstruktúrában egy programegység deklarációinak hatásköre kiterjed az összes tartalmazott programegységre is.

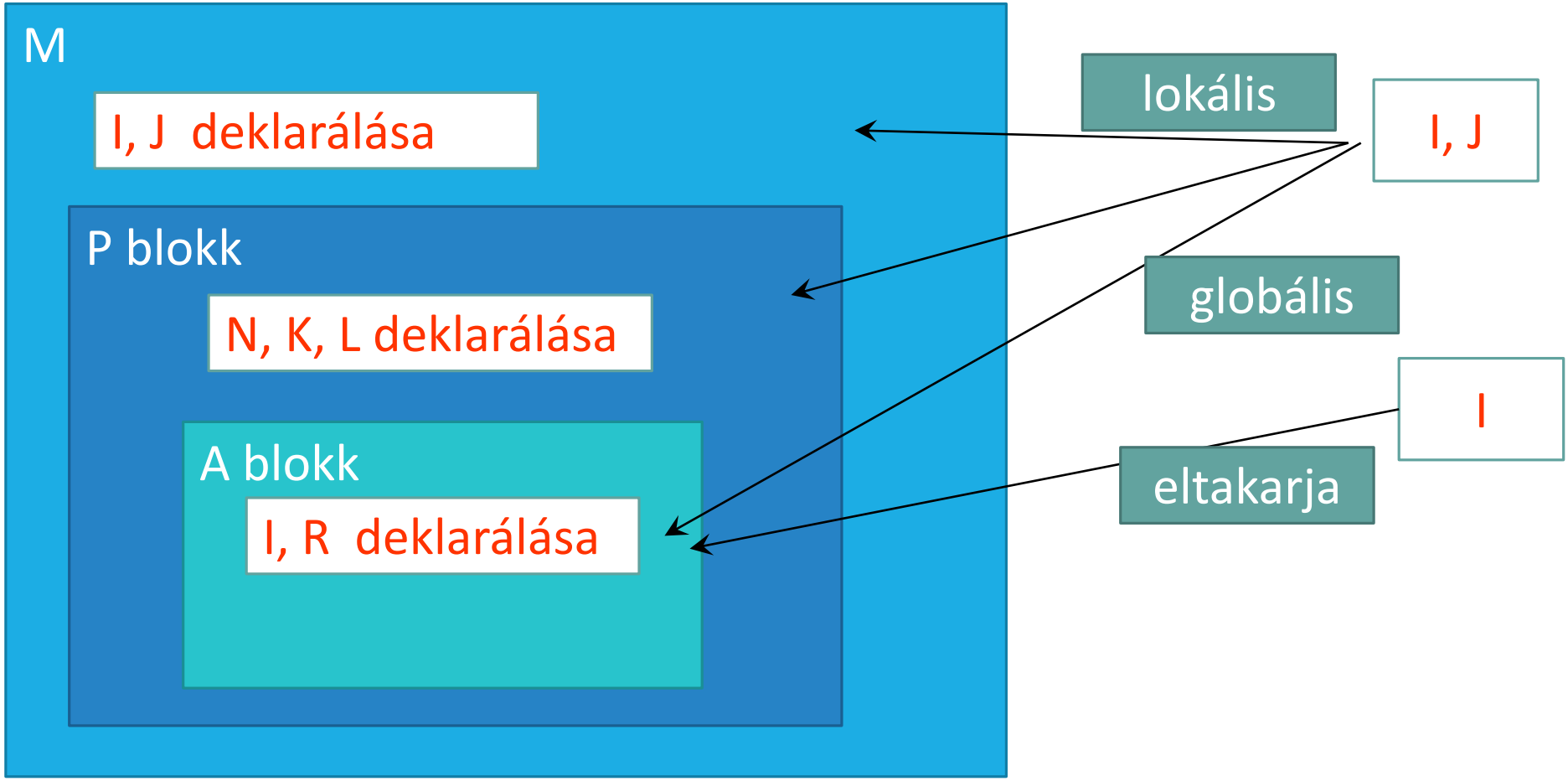
# Globális – lokális

Egy programegységben a tartalmazó programegységek által deklarált azonosítókat (a tartalmazott programegység szemszögéből) globális azonosítóknak nevezzük.

A programegységben deklarált azonosítókat magában a programegységben lokális azonosítóknak nevezzük.

Blokkstruktúra esetén egy programegységben egy globális azonosító újradeklarálható, ekkor a lokális deklaráció eltakarja a globális deklarációt.

# Példa


M
I, J deklarálása
P blokk
N, K, L deklarálása
A blokk
I, R deklarálása
lokális
I, J
globális
I
eltakarja

# Példa

```
procedure M is
  I, J:Integer;
  procedure P is
    N, K, L:Integer;
    procedure A is
      I, R:Integer;
      begin
         I:=3; J:=9;          -- A-beli I, M-beli J
         N:=1;                -- P-beli N
         M.I:=4;              -- minősített hivatkozás!
                              -- ...
      end;
  begin
    I:=5;                     -- M-beli I
    A;                        -- hívja A-t
                              -- ....
  end;

begin
  I:=3;
  P;                          -- hívja P-t ...
end;
```

# Azonosító túlterhelése

Egy azonosítónak ugyanabban a szövegrészben több definíciója is érvényben lehet

A túlterhelést a leggyakrabban alprogramok nevei esetében teszik lehetővé

- általában a paraméterek segítségével tudjuk megkülönböztetni
  - `write(2)`
  - `write("ez valami más")`
  - `2 + 3`
  - `2.1 + 3.0`

# Erősen típusos nyelv

Egy nyelv erősen típusos, ha minden érték, objektum, formális paraméter és függvény típusa fordítási időben egyértelműen meghatározható.

Statikus típusosság, ha a típusok ellenőrzése fordítási időben történik, míg a dinamikus típusosság esetén a típusellenőrzések (műveletekhez) futás időben történnek.

Az erősen típusos nyelv és a statikus típusosság összefonódása ebből következik.

# Memóriakezelés

A változóhoz a memóriaterület hozzárendelése (allokálása) történhet:

- automatikusan, a definíció kiértékelésekor, vagy
- a programozó rendelkezik róla

a változó által lefoglalt terület felszabadítása lehet:

- automatikus, vagy
- a programozó hatáskörébe tartozik.

# Élettartam

A változókhoz a szükséges memóriaterület lefoglalása (allokálása) és annak felszabadítása közötti időt a változó élettartamának nevezzük.

# Statikus változó

élettartama a program egész működése idejére kiterjed

elhelyezése mindig az úgynevezett statikus (main) memória-területre történik

a statikus terület egyszer, a program betöltésekor kerül lefoglalásra

# Dinamikus változó

a program explicit módon foglal le területet, a dinamikus (heap) memóriaterületen, amire a címével, úgynevezett pointerrel lehet hivatkozni.

egyes nyelvek tartalmaznak utasítást a dinamikus területek felszabadítására is.

blokkstruktúrában a programegységek (nem statikus) lokális változói az úgynevezett végrehajtási veremben helyezkednek el.

# Paraméterátadás

A programegységek legelterjedtebb fajtája az alprogram.

- A nyelvek kétféle alprogramot használnak, ezek az eljárások és a függvények.

Az alprogramoknál a hívóval való kommunikációt tervezéskor formális paraméterekkel írjuk le,

- Az alprogram aktivizálásakor (hívásakor) aktuális paramétereket kell megfeleltetni.

# Lexikális elemek

# A fordítási egység

A programok fordítási egységei lexikális elemek sorozatai.

- A lexikális elemek karaktersorozatok, határoló jelekkel elválasztva.

A nyelvekben általában megkülönböztetjük

- A grafikus karaktereket
  - a Latin-1 vagy az angol abc betűi, számjegyek és speciális karakterek
  - Például: `" # & ' ( ) , * - + / : ; < = _ [ ] { } | . -`
- Formátum vezérlő karaktereket
  - Például az ISO 6429 szabványban:
    - character tabulation (HT)
    - line tabulation (VT)
    - carriage return (CR)
    - line feed (LF)
    - and form feed (FF)
- Az egyéb vezérlő karaktereket
  - Ez implementáció függő

# Lexikális elemek

Lexikális elem

- Azonosító
- Numerikus literál
  - Numerikus érték leírására
- Karakter literál
  - ...
- String literál
- Határoló
- Megjegyzés

# A jelkészlet

A karakterkészlet  különböző karakterek egy halmaza.

- Nincs feltételezésünk a belső ábrázolásáról, rendezettségéről.
- Megadjuk a karakterek neveit és a karakter megjelenítését látható formában.
  - Tartalmazhat olyan karaktereket, amelyek bizonyos megjelenítéseknél ugyanúgy néznek ki, mégis logikailag különböző.
    - Három különböző karakter, ami nagyon hasonló megjelenésű:
      - a latin nagy „A”
      - a cirill nagy „A”
      - görög nagybetűs Alfa „A”
  - Példa
    - EXCLAMATION !
    - QUESTION_MARK ?
    - SEMICOLON ;

# Karakterkódok – táblázatok

A karakterkód egy leképezés:

- Kölcsönösen egyértelmű megfeleltetést ad a karakterkészlet karakterei és a nemnegatív egészek egy halmaza között
- Egy egyedi számkódot, egy kódpozíciót rendel a karakterkészlet minden karakteréhez.
  - Gyakran táblázat
  - http://unicode-table.com

# ASCII táblázat

| Decimal | Hex | Char | Decimal | Hex | Char | Decimal | Hex | Char | Decimal | Hex | Char |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | [NULL] | 32 | 20 | [SPACE] | 64 | 40 | @ | 96 | 60 | ` |
| 1 | 1 | [START OF HEADING] | 33 | 21 | ! | 65 | 41 | A | 97 | 61 | a |
| 2 | 2 | [START OF TEXT] | 34 | 22 | " | 66 | 42 | B | 98 | 62 | b |
| 3 | 3 | [END OF TEXT] | 35 | 23 | # | 67 | 43 | C | 99 | 63 | c |
| 4 | 4 | [END OF TRANSMISSION] | 36 | 24 | $ | 68 | 44 | D | 100 | 64 | d |
| 5 | 5 | [ENQUIRY] | 37 | 25 | % | 69 | 45 | E | 101 | 65 | e |
| 6 | 6 | [ACKNOWLEDGE] | 38 | 26 | & | 70 | 46 | F | 102 | 66 | f |
| 7 | 7 | [BELL] | 39 | 27 | ' | 71 | 47 | G | 103 | 67 | g |
| 8 | 8 | [BACKSPACE] | 40 | 28 | ( | 72 | 48 | H | 104 | 68 | h |
| 9 | 9 | [HORIZONTAL TAB] | 41 | 29 | ) | 73 | 49 | I | 105 | 69 | i |
| 10 | A | [LINE FEED] | 42 | 2A | * | 74 | 4A | J | 106 | 6A | j |
| 11 | B | [VERTICAL TAB] | 43 | 2B | + | 75 | 4B | K | 107 | 6B | k |
| 12 | C | [FORM FEED] | 44 | 2C | , | 76 | 4C | L | 108 | 6C | l |
| 13 | D | [CARRIAGE RETURN] | 45 | 2D | - | 77 | 4D | M | 109 | 6D | m |
| 14 | E | [SHIFT OUT] | 46 | 2E | . | 78 | 4E | N | 110 | 6E | n |
| 15 | F | [SHIFT IN] | 47 | 2F | / | 79 | 4F | O | 111 | 6F | o |
| 16 | 10 | [DATA LINK ESCAPE] | 48 | 30 | 0 | 80 | 50 | P | 112 | 70 | p |
| 17 | 11 | [DEVICE CONTROL 1] | 49 | 31 | 1 | 81 | 51 | Q | 113 | 71 | q |
| 18 | 12 | [DEVICE CONTROL 2] | 50 | 32 | 2 | 82 | 52 | R | 114 | 72 | r |
| 19 | 13 | [DEVICE CONTROL 3] | 51 | 33 | 3 | 83 | 53 | S | 115 | 73 | s |
| 20 | 14 | [DEVICE CONTROL 4] | 52 | 34 | 4 | 84 | 54 | T | 116 | 74 | t |
| 21 | 15 | [NEGATIVE ACKNOWLEDGE] | 53 | 35 | 5 | 85 | 55 | U | 117 | 75 | u |
| 22 | 16 | [SYNCHRONOUS IDLE] | 54 | 36 | 6 | 86 | 56 | V | 118 | 76 | v |
| 23 | 17 | [ENG OF TRANS. BLOCK] | 55 | 37 | 7 | 87 | 57 | W | 119 | 77 | w |
| 24 | 18 | [CANCEL] | 56 | 38 | 8 | 88 | 58 | X | 120 | 78 | x |
| 25 | 19 | [END OF MEDIUM] | 57 | 39 | 9 | 89 | 59 | Y | 121 | 79 | y |
| 26 | 1A | [SUBSTITUTE] | 58 | 3A | : | 90 | 5A | Z | 122 | 7A | z |
| 27 | 1B | [ESCAPE] | 59 | 3B | ; | 91 | 5B | [ | 123 | 7B | { |
| 28 | 1C | [FILE SEPARATOR] | 60 | 3C | < | 92 | 5C | \ | 124 | 7C | \| |
| 29 | 1D | [GROUP SEPARATOR] | 61 | 3D | = | 93 | 5D | ] | 125 | 7D | } |
| 30 | 1E | [RECORD SEPARATOR] | 62 | 3E | > | 94 | 5E | ^ | 126 | 7E | ~ |
| 31 | 1F | [UNIT SEPARATOR] | 63 | 3F | ? | 95 | 5F | _ | 127 | 7F | [DEL] |

# Karakterkódolás

A karakterkódolás egy *algoritmus* karakterek digitális formában történő megadására.

- Ez a karakterek kódszámainak sorozatait „oktetek” sorozatára képezi le.
  - Például az ISO 10646 karakterkódban az 'a'  az 'ä' és az ‰ karaktereknek megfelelő számkódok a 97, a 228 és a 8240.

A karakterkódolás adja meg, hogy a karakterkódokat *hogyan* képezzük le oktetek sorozatára.

- Az ISO 10646 egy lehetséges kódolásában minden karakter kódolásához két oktetet használva a kódolási algoritmus ezeknek a (0, 97), a (0, 228) és a (32,48) oktetpárokat felelteti meg.

# Milyen karakterek használhatók egy nyelvben?

## Példák

- Pascal (ISO 7185:1990):
  - Minimum követelmények
    - '0'..'9' rendezett és folytonos kell legyen
    - 'A'..'Z' rendezett, de nem feltétlenül folytonos
    - 'a'..'z' nem kötelező, de ha van, akkor mint a nagybetűk.
  - Nincs előírva a lehetséges karakterek halmaza
    - A különböző implementációk különböző kódolást használhatnak(!)
    - pl. EBCDIC vagy ASCII => különböző implementációk másképpen viselkedhetnek
      - '}' $\in$ 'A'..'Z'  EBCDIC-ben és '}' $\notin$ 'A'..'Z' ASCII-ben
      - ' A' < '1'  EBCDIC-ben és '1' < 'A' ASCII-ben
  - **Ez nem tesz jót a programok hordozhatóságának**

# Milyen karakterek használhatók egy nyelvben?

Példák:

- C++ - ISO/IEC 14882:2017
  - Az alap karakterkészlet 96 elemű
    - Space, horizontal tab, vertical tab, new-line, form feed
    - 91 grafikus karakter:
      - 'a'..'z' 'A'..'Z' '0'..'9'
      - _ { } [ ] # ( ) < > % : ; . ? * + - / ^ & | ~ ! = , \ " ’
  - Univerzális karakterek (UCS)
    - ISO 10646 szerint kódolva
    - Lényegében Unicode
  - **Ez precízebb régebbi verziókhoz viszonyítva:**
    - **Ahol nem volt a kód előírva**
    - **Nem lehetett UCS karaktereket használni**

# Milyen karakterek használhatók egy nyelvben?

Példák:

- CLU, Eiffel:
  - ASCII kód a megengedett.
- ADA (ISO/IEC 8652:2012(E))
  - A teljes ISO/IEC 10646:2011
- Java:
  - "Java programs are written using the Unicode character set”
- C#:
  - „A source file is an ordered sequence of Unicode characters.”
- Python:
  - „Python reads program text as Unicode code points; …”

# Elhatároló jelek

A legtöbb programozási nyelvben a helyköz (space), a tab, a sorvége elhatároló jel.

- Van sok egy karakterből álló elhatároló jel
  - Pascal: & ' ( ) + - * / : . ; < = >
  - C++: & % ^ ' ( ) + - { } | ~ [ ] \ " * / : . ; , < = > ! ?
  - Java: ( ) { } [ ] ; , . + - * / : < = > ! ~ ? : & | ^ %
  - ADA: & ' ( ) * + , - . / : ; < = > |
- Van számos több karakterből álló elhatároló jel is
  - Pascal: := >= <= <> << ** ..
  - C++: -> ++ -- .* ->* == <= >= && || << >> <<= >>= != .. += -= &= ^= |= :: *= /=
  - Java: == <= >= << != && || ++ -- >> >>> += -= *= /= &= |= ^= %= <<= >>= >>>=
  - ADA: => .. ** := /= >= <= << >> <> --

# Megkülönbözteti-e a nyelv a kis- és nagybetűket?

`Kutya = kutya = KUTYA = KuTya = …?`

Alapvetően két megközelítés

- Pascal, Fortran, CLU, ADA, … esetén ekvivalensek
- C, C++, Modula, Perl, Java, C#, … esetén minden betű különböző

De!

- Eiffel: a kis- és a nagybetűk ekvivalensek, de használhatjuk ugyanazt a nevet egy objektumra és a típusára
  - `a: A`
  - (itt az `A` az típus és `a` az objektum)

# Azonosítók

Milyen karakterek használhatóak az azonosítók leírására?

Mi az azonosító megengedett szintaxisa?

Van-e hosszúsági megkötés az azonosítókra?

Vannak-e kötött szavak?

Van-e különbség a kulcsszavak és az előre definiált szavak között?

# Milyen karakterek használhatóak azonosítókban?

Pascal (ISO 7185:1990):

- betűk ('A'..'Z', 'a'..'z') és számjegyek ('0'..'9')

C++ (ISO/IEC 14882:2017), CLU, Eiffel:

- betűk ('A'..'Z', 'a'..'z'), számjegyek ('0'..'9') és ‚_'

Perl:

- az előzőeken túl $, @ és % jellel is kezdődhetnek
  - $ a skalárokat, @ számmal indexelt tömböt, % asszociatív tömböt jelöl.

ADA

- Ada2012 szabvány : minden, aminek a nevében szerepel, hogy betű vagy számjegy! (Πλάτων, Чайковский stb. is)

# Milyen karakterek használhatóak azonosítókban?

Java:

- A Unicode betűi és számjegyei, a '$' és ‚_'

SWIFT:

- Identifiers begin with
  - an uppercase or lowercase letter A through Z,
  - an underscore (_),
  - a noncombining alphanumeric Unicode character in the Basic Multilingual Plane,
  - or a character outside the Basic Multilingual Plane that isn't in a Private Use Area.
- After the first character, digits and combining Unicode characters are also allowed.
  - `let π = 3.14159`
  - `let 你好 = "你好世界"`
  - `let 🐶🐮 = "dogcow"`

# Milyen karakterek használhatóak azonosítókban?

C#:

- Az azonosítókat a Unicode szabvány ajánlásának megfelelően kell írni.
- Érdekesség, hogy a kulcsszavak a @ bevezető jellel használhatóak azonosítóként is (például: `@bool`)
  - Ez a jel más azonosítót nem vezethet be.
- Ezt a lehetőséget a programok hordozhatósága illetve „átjárhatósága” miatt vezették be.
- A @ prefixű szimbólumok neve valójában @ nélkül kerül fordításra.
  - Így attól függetlenül, hogy egy másik nyelv nem tartalmazza például a „`sealed`” kulcsszót, és egy programban azonosítóként használják, C#-ban lehetőség van az azonosító közvetlen használatára.

# Mi az azonosítók megengedett szintaxisa?

A leggyakrabban:

- letter { letter | digit }
- néha az '_' beszúrása is megengedett.

Pascal: letter{letter|digit}

ADA: letter{[_]letter|[_]digit},

- itt a betűk halmaza nagyobb, minden, aminek a nevében szerepel az, hogy „letter”

Java: Java_letter{[_]Java_letter|[_]digit},

- αβγδε vagy El_Ňiňo

# Van-e hosszúsági megkötés az azonosítókra?

Érdekesség

- Fortran66, Fortran77: maximum. 6 jel
- Fortran90: maximum. 31 jel

Általában tetszőleges hosszúság de, vannak kivételek

- ADA: be kell férjen egy sorba

# Vannak-e „fenntartott” szavak?

Pascal, C, C++, Java, Perl, CLU, ADA, …

- a fenntartott szavak (kulcsszavak) nem használhatók azonosítóként
- Példa C++ további megkötésekre:
  - Each identifier that contains a double underscore __ or begins with an underscore followed by an uppercase letter is reserved to the implementation for any use
  - Each identifier that begins with an underscore is reserved to the implementation for use as a name in the global namespace

# Vannak-e „fenntartott” szavak?

ADA

- különbséget tesz a kulcsszavak és az előredefiniált szavak között, mint például: Integer, True, etc..
- Az előredefiniált szavak átdefiniálhatóak:
  - type Integer is range -999_999..+999_999;
  - így a program implementáció-független
  - de: True: Integer … -- nincs értelme…

Fortran, PL/1

- megengedett a kulcsszavakat azonosítóként használni!

# Literálok

Milyen numerikus literálok vannak?

Milyen más alapok megengedettek a 10-esen kívül?

Mi az egész, illetve valós literálok szintaxisa?

Többsoros string literálok megengedettek-e?

Vezérlőkarakterek string literálokban megengedettek-e, és hogyan írjuk le őket?

# Numerikus literálok

A gyakorlatban csak a '0'..'9’ és az 'A'..'F'  karakterek használhatók

- További lehetőségek:
  - '_' : 456_789
    - ADA, Perl, Eiffel
  - Exponenciális alak: 1E6
    - Ada
  - 'L’ ‘U’ a típus megadására (long, unsigned): 4L
    - C, C++, Java, C#

Mi a számok szerkezete?

- Egészek: decimális:  [-]digit{digit}
  - -123, 456789

# Numerikus literálok

Mi a számok szerkezete?

- Racionális: [-]digit{digit}.digit{digit}[exponent]
  - -123.456E+3
- Majdnem minden nyelv ad ehhez valami specialitást:
  - Pascal
    - [-]digit{digit}[exponent] 12E+3
  - ADA
    - ‘_’ is lehet: -1_234.0
  - Eiffel
    - [-]{digit}.{digit}[exponent] -1.
  - Java
    - „float_suffix” is lehet a végén

# Milyen alapok megengedettek?

Általában: decimális számok, néha más is

- Például 2, 8 16

Példák

- Pascal, Modula, Oberon: csak decimális számok
- C, C++, Java…: Oktális, decimális és hexadecimális
  - oktális: 0734
  - hexadecimális:  0xFF, 0xC0B0L.
- ADA: minden alap 2 és 16 között:
  - base#mantissa#[exponent]: 16#FFF.F#E1
- Mathematica: minden alap 2 és 36 között!
  - Minden számjegy és angol betű
  - basis^^mantissa : 30^^Mathematica = 13 207 019 439 499 57010

# Példa a szabályokra (Java)

IntegerLiteral:
- DecimalIntegerLiteral
- HexIntegerLiteral
- OctalIntegerLiteral
- BinaryIntegerLiteral

DecimalIntegerLiteral:
- DecimalNumeral [IntegerTypeSuffix]

HexIntegerLiteral:
- HexNumeral [IntegerTypeSuffix]

OctalIntegerLiteral:
- OctalNumeral [IntegerTypeSuffix]

BinaryIntegerLiteral:
- BinaryNumeral [IntegerTypeSuffix]

IntegerTypeSuffix:
- (one of)
- l L

DecimalNumeral:
- 0
- NonZeroDigit [Digits]
- NonZeroDigit Underscores Digits

NonZeroDigit:
- (one of)
- 1 2 3 4 5 6 7 8 9

Digits:
- Digit
- Digit [DigitsAndUnderscores] Digit

Digit:
- 0
- NonZeroDigit

DigitsAndUnderscores:
- DigitOrUnderscore {DigitOrUnderscore}

DigitOrUnderscore:
- Digit
- _

Underscores:
- _ {_}

# Karakterek és karaktersorozatok

Karakterek:

- 'A' vagy „escape szekvenciák”: '\n’, '\'’, '\ooo' stb.

Stringek "A”

Tárolásuk különböző lehet.

# Megjegyzések

Szerkezete befolyásolja a program megbízhatóságát

Szokásos lehetőségek:

- egy speciális oszloptól (jeltől) a sor végéig
- speciális jel(ek) elején – végén
- speciális jel(ek) elején – vége a sor vége

# Megjegyzések

## Példák

- Pascal:
  - (* és *)
  - { és }
- CLU, Matlab
  - %-tól sor végéig
- Python
  - #-tól a sor végéig
  - ’’ és ’’ között

## Példák

- C:
  - /* és  */
- C++: mint C-ben, és még
  - // -tól sor végéig
- Java: mint C++-ban és még
  - /** documentation */
- C#: mint C++-ban és még
  - /// egysoros dokumentációs megjegyzés
- ADA, Eiffel:
  - -- -tól sor végéig

# Beépített adattípusok, változók, kifejezések

# Mit jelent a nyelvben az adattípus?

Egy értékhalmaz és egy művelethalmaz

Specifikáció – Reprezentáció – Implementáció

Példa

- Specifikáció
  - egészek -∞ -től +∞ -ig, és a megengedett műveletek +, - *, stb.
- Reprezentáció
  - 8, 16, 32, 64-bit, előjeles kettes komplemens kóddal (megszorítások)
- Implementáció
  - a műveletek megvalósítása

# Típus a programozó szemszögéből

Típusspecifikáció („mit”)

- alaphalmaz: a valós világ minket érdeklő objektumainak halmaza
- specifikációs invariáns ($I_S$) – ezt a halmazt tovább szűkíti
- típusműveletek specifikációja – csak a „mit”!

# Típus a programozó szemszögéből

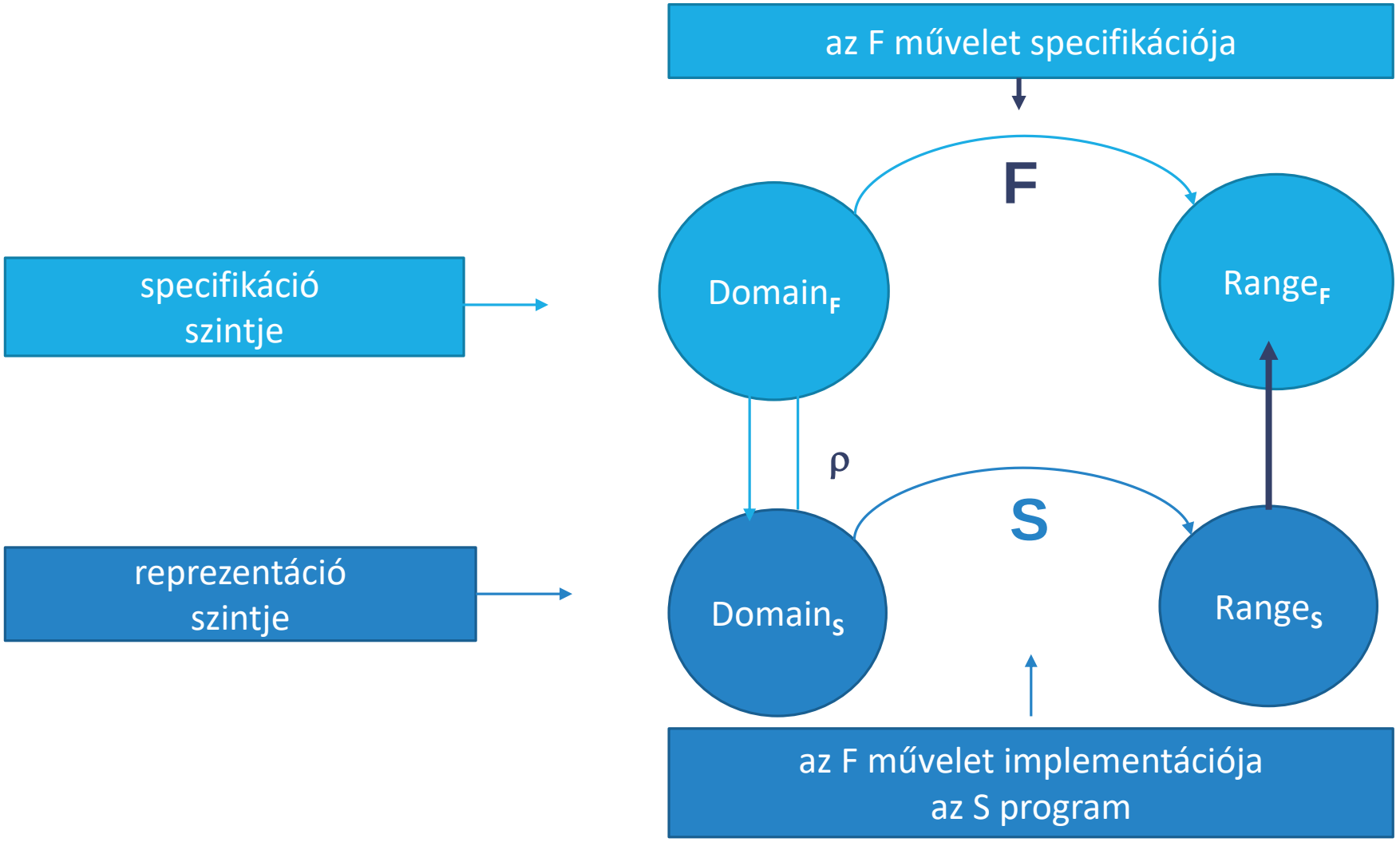
Típus megvalósítás („hogyan”)

- Reprezentációs függvény
- Típus invariáns
- Típusműveletek megvalósítása

Például

- Komplex számok
- Verem
- stb.

# A típus specifikáció és a típus kapcsolata



# Típusok támogatása

A legtöbb programozási nyelvben van valamilyen adattípusra támogatás.

Vannak „beépített” („built-in”) típusok, itt

- a specifikáció – a programozási nyelv referencia kézikönyvében
- a reprezentáció és az implementáció a fordítóprogrammal jön.
- mennyire szabványosak a beépített típusok?

Vannak megengedett típuskonstrukciók

- tömb, rekord, stb.

A programozási nyelvek gyakran lehetővé teszik új adattípusok tervezésekor, hogy önálló modulokban elrejtsük a reprezentációt és az implementációt

# A programozási nyelvek típusszerkezete

Hogyan osztályozhatjuk a típusokat?

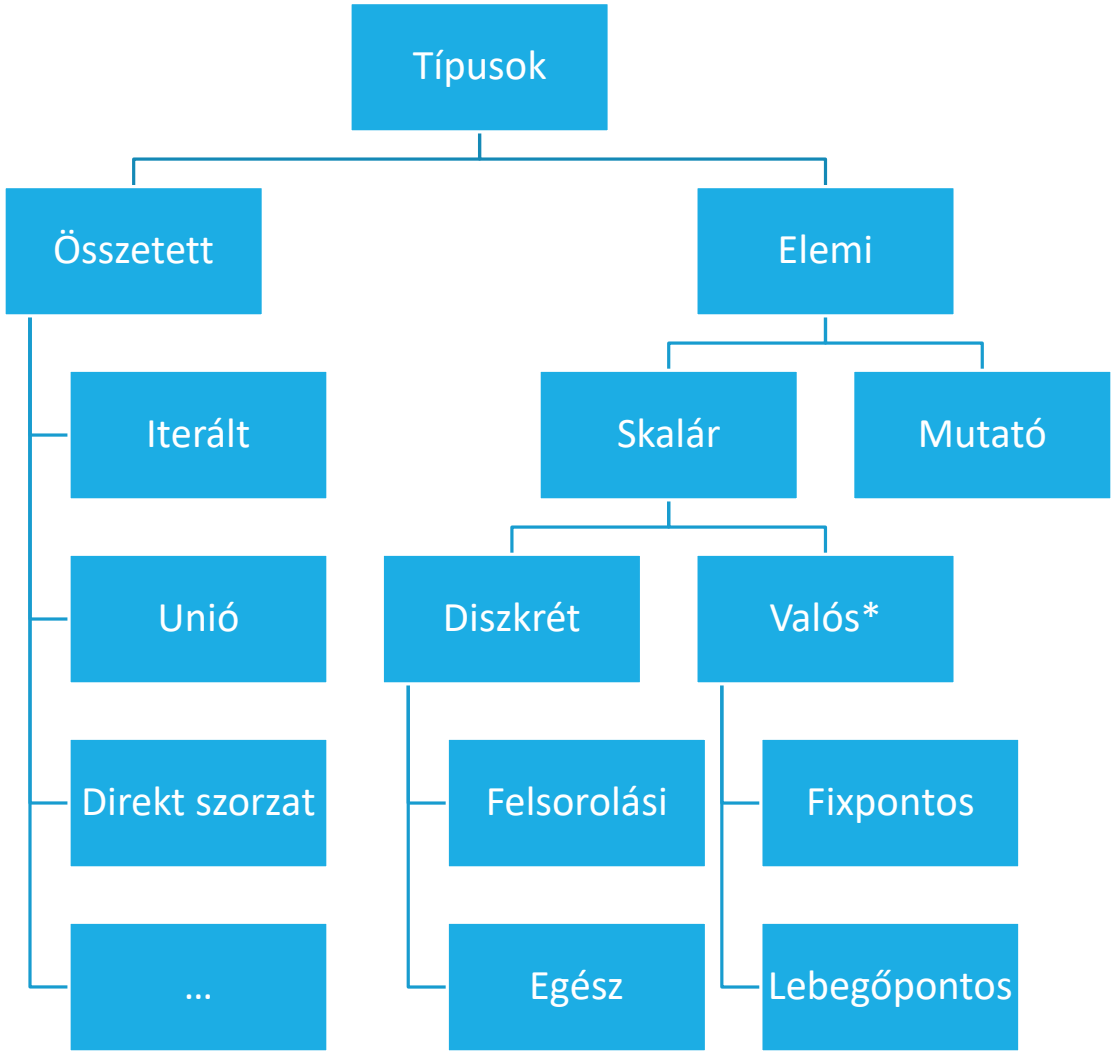
Vannak típusosztályok a nyelvben?

- Vannak típusosztályokra vonatkozó műveletek?

Elemi és összetett típus

- Elemi típusoknak nevezzük azokat a típusokat, amelyek logikailag felbonthatatlanok
- Az összetett típusokat már meglévő típusokból, mint komponensekből hozzuk létre

# Típusosztályok



# Példák

# Típusosztály műveletekre példa

Ada – felsorolási típus

- `type Hónapok is (Január, Február, Március,…., December);`
- automatikusan keletkezik:
  - `Hónapok'First, Hónapok'Last,`
  - `Hónapok'Range,`
  - `Hónapok'Min(x,y), Hónapok'Max(x,y)`
  - `Hónapok'Succ(x), Hónapok'Pred(x)`
  - `Hónapok'Image(x)`
  - `Hónapok'Value(x)`
  - `Hónapok'Val(x)`
  - `Hónapok'Pos(x)`
  - …

# Van-e speciális tulajdonsága a nyelv típusrendszerének?

## ADA

- Különbség az altípus és a származtatott típus között:
  - Egy típus altípusa a típus értékhalmazának részhalmazát jelöli.
  - Az altípusra alkalmazható a típus összes művelete.
- Például:
  - Dátum – a hónap napjai az 1 .. 31 intervallumba esnek.
  - Ehhez deklarálhatunk egy altípust:
    - `subtype Napok is Integer range 1 .. 31;`

# Van-e speciális tulajdonsága a nyelv típusrendszerének?

## ADA

- Új típus létrehozása már létező típusból: származtatás
  - itt kicsit mást jelent, mint az OOP-ben
  - `type Hosszúság is new Float;`
  - A származtatott típus átveszi az eredeti típus struktúráját, értékhalmazát és műveleteit, átveszi a kezdeti értékeit is.
    - De a két típus nem ekvivalens!
  - `type Terület is new Float;`
    - `H: Hosszúság; T: Terület;`
    - `H := T;` hibás! (már szintaktikailag!)
  - Konverzió lehetősége szükséges:
    - `function "*" (x, y: Hosszúság) return Terület is`
    - `  begin return Terület (Float(x) * Float(y)); end;`
    - `T:=H*H;` írható, de `T:=H;` vagy `T:=H/H;` továbbra sem!

# Van-e speciális tulajdonsága a nyelv típusrendszerének?

Java:

- Kétféle típus létezik: a primitív típusok (numerikus és logikai), és a referencia típusok.
- A felhasználó által definiált típusok az osztályok.
- A primitív típusoknak van „csomagoló” osztálya, boxing, unboxing lehetőségekkel

Eiffel:

- alapértelmezés szerint az értékek objektumokra vonatkozó referenciák, de definiálhatunk kiterjesztett (expanded) típusokat is, ezek változói maguk az objektumok.
- Az alap típusok mindig kiterjesztettek.
- Új típus = új osztály létrehozása.

C#: érték és referencia típusok.

- Érték típusok: egyszerű típusok (pl. char, int, float), felsorolási és struct típusok.
- Referencia típusok - osztály (class), interface típusok,  delegate és tömb típusok.
- Boxing – unboxing lehetőségek vannak itt is.

# Mik a beépített adattípusok?

Az alaptípusok általában, a szokásos műveletekkel

- az egészek
- a karakterek
- a logikai típus
- a valósak

# A skalár típusok osztálya

A skalár típusok a diszkrét és a valós típusok.

- Értékei rendezettek, így a relációs operátorok (<, <=, =, >=, >) előredefiniáltak ezekre a típusokra.
- ADA–ban számos attribútum:
  - `S'First, S'Last, S'Range`
  - `S'Min, S'Max`
  - `S'Succ, S'Pred (Constraint_Error)`
  - `S'Image , S'Width, S'Value stb.`
  - `Hónapok'First = Január`
  - `Hónapok'Succ(Március)= Április`

# A skalár típusok osztálya

Eiffelben ezek a típusok mindig kiterjesztettek

C++ – ezek az „integral” típusok

Perl

- Skalár adattípus
  - a neve kötelezően $ jellel kezdődik
  - számok, stringek (!) és referenciák tárolására, a típusok közötti (numerikus, string,...) konverzió automatikus
  - `$skalar1 = 'string';  # karakterlanc`
  - `$skalar2 = 1234;      # integer`
  - `$skalar3 = 4.5;       # float`
  - `$skalar4 = $skalar3;  # ertekadas`
  - `$skalar5 = \$skalar2; # referencia`

# A diszkrét típusok osztálya

A diszkrét típusok a

- felsorolási és az
- egész típusok

# Felsorolási típusok

A típusértékhalmaz megadható egy explicit felsorolással

- `type Days is (Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday);`

Gyakran a karakter és a logikai típus is előredefiniált felsorolási típus.

- A logikai értékek kezelésére szolgáló típust sok programozási nyelvben `Boolean`–nak hívják
  - két lehetséges értéke van, a `True` és a `False`,
  - a szokásos műveletek a `not`, `and`, `or` és a `xor`.
- De vannak kivételek
  - Például `Yes`, `No` Objective-C esetén

# Felsorolási típusok

Object Pascal

- Egy felsorolási típus definíciója:
- `type Flower = (Rose, Tulip, Violet, Geranium);`
- Az azonosítóknak különbözőeknek, és a programban egyedieknek kell lenniük.
- Standard függvények
  - `Pred(X)`
  - `Succ(X)`
  - `Dec(X)`
  - `Inc(X)`
  - `<`
  - …

# Felsorolási típusok

## Object Pascal

- Logikai
  - `Boolean`, `ByteBool` 1 byte
  - `WordBool` 2 byte
  - `LongBool` 4 byte
  - `ByteBool`, `WordBool`, `LongBool` : numerikus `boolean` típusok, (minden numerikus értéket logikaiként értelmez) a nemnulla a `True`
    - `-1 true`, `0 false`
- Karakter típusok
  - `ANSIChar` (1 byte),
    `WideChar` (2 byte, Unicode)

# Felsorolási típusok

ADA

- Minden felsorolási literál a felsorolási típus egy önálló típusértéke, és van egy pozíció száma.
  - Az első felsorolási literál pozíció száma 0.
  - A rendezési relációt a felsorolás sorrendje adja. Egy felsorolási típus értékei lehetnek azonosítók vagy karakterek.
  - `type Color is (Red, Blue, Green);`
  - `type Roman_Digit is ('I','V','X','L','C','D','M');`
- Megengedett a felsorolási nevek átlapolása, és ha szükséges, a típusnév segít a megkülönböztetésben: `Color'(Red)`.

# Felsorolási típusok

## ADA

- Három előredefiniált  karakter típus van
  - `Character` – ennek az értékei az ISO 10646 Basic Multilingual Plane (BMP) Row 00 256 kódpozíciójának (Latin-1)
  - `Wide_Character` típus értékei pedig az ISO 10646 Basic Multilingual Plane (BMP) 65536 kódpozíciójának
  - `Wide_Wide_Character` típus értékei pedig az ISO/IEC 10646:2011 2 147 483 648 kódpozíciójának felelnek meg.
- Logikai
  - `type Boolean is (False, True);`
    - speciális előredefiniált típus a szokásos műveleteken kívül az
    - "`and then`" és az
    - "`or else`" műveletekkel  (lusta kiértékelés).

# Felsorolási típusok

C++

- Az „enum” kulcsszót használják a felsorolási típusok, a megfelelő egész érték 0-val kezd, egyesével nő, kivéve, ha „= expr” szerepel valahol
  - `enum color {red, blue, green=20, yellow);`
- Az előredefiniált `char`, `signed char` és `unsigned char` típusok írják le a lehetséges karakter típusokat, sok probléma származik abból, hogy ezeket nem szabványosították igazán.
- Előredefiniált logikai típus a `bool` a `false` és `true` értékekkel, kifejezésekben a 0 értéket is hamis-ként kezeli
  - Minden nem 0-t igaznak.

# Felsorolási típusok

Java

- Nem volt felsorolási típus – a Java 5.0-ig!
- Az előredefiniált `char` típus a 16 bites Unicode character-készletet támogatja '\u0000' –tól '\uffff'-ig,
  azaz 0-tól 65535-ig.
- Előredefiniált logikai típus, a `boolean` a `false` és `true` értékekkel
- A műveletek
  - Relációs: `==` `!=`
  - not: `!`
  - and, or: `&&` `||`
  - and, or, xor (bitwise): `&` `|` `^`
  - feltételes kifejezés: `?` `:`

# Felsorolási típusok

Java

- `enum Season { WINTER, SPRING, SUMMER, FALL }`
- Lehet állapota is és műveletei is
  - https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html
  - Minden enum típusnak ez a (rejtett) közös őse
  - ```
    public abstract class Enum<E extends Enum<E>>
                  extends Object
                  implements Comparable<E>, Serializable
    ```
    ```

# Példa

```
import java.util.*;
public class Card {
      public enum Rank { DEUCE, THREE, FOUR, FIVE, SIX,
          SEVEN, EIGHT, NINE, TEN, JACK, QUEEN, KING, ACE }

      public enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES }

      private final Rank rank;
      private final Suit suit;
      private Card(Rank rank, Suit suit) {
          this.rank = rank;
          this.suit = suit;
      }

      public Rank rank() { return rank; }
      public Suit suit() { return suit; }
      public String toString() { return rank + " of " + suit; }

      private static final List<Card> protoDeck = new ArrayList<Card>();

       // Initialize prototype deck
      static {
          for (Suit suit : Suit.values())
             for (Rank rank : Rank.values())
                 protoDeck.add(new Card(rank, suit));
      }
```

Használja a Rank és a Suit toString-jét

Használja a Rank és a Suit values műveletét

# Tulajdonságok és viselkedés

```
public enum Planet {
 MERCURY (3.303e+23, 2.4397e6),
 VENUS (4.869e+24, 6.0518e6),
 EARTH (5.976e+24, 6.37814e6),
 MARS (6.421e+23, 3.3972e6),
 JUPITER (1.9e+27, 7.1492e7),
 SATURN (5.688e+26, 6.0268e7),
 URANUS (8.686e+25, 2.5559e7),
 NEPTUNE (1.024e+26, 2.4746e7);

 private final double mass; // in kilograms
 private final double radius; // in meters

 Planet(double mass, double radius) {
  this.mass = mass;
  this.radius = radius;
 }
 private double mass() { return mass; }
 private double radius() { return radius; }
```

# Tulajdonságok és viselkedés

```
// universal gravitational constant  (m3 kg-1 s-2)
public static final double G = 6.67300E-11;

double surfaceGravity() {
 return G * mass / (radius * radius);
}

double surfaceWeight(double otherMass) {
 return otherMass * surfaceGravity();
}
```

Próba

```
public static void main(String[] args) {
   double earthWeight = Double.parseDouble(20.0);
   double mass = earthWeight/EARTH.surfaceGravity();
   for (Planet p : Planet.values())
     System.out.printf("Your weight on %s is %f%n", p,
                                                  p.surfaceWeight(mass));
}
```

# Tulajdonságok és viselkedés – Absztrakt művelet

```
public enum Operation {
  PLUS   { double eval(double x, double y) { return x + y; } },
  MINUS  { double eval(double x, double y) { return x - y; } },
  TIMES  { double eval(double x, double y) { return x * y; } },
  DIVIDE { double eval(double x, double y) { return x / y; } };
   // Do arithmetic op represented by this constant
  abstract double eval(double x, double y);
}
```

Próba

```
public static void main(String args[]) {
  double x = Double.parseDouble(args[0]);
  double y = Double.parseDouble(args[1]);
  for (Operation op : Operation.values())
    System.out.printf("%f %s %f = %f%n", x, op, y, op.eval(x, y));

}
```

# Felsorolási típusok

## C# - Enum

- Rendelkezik alaptípussal – ez tetszőleges integral típus lehet
  - `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`
- Deklaráció példa:

```
enum Color: long {
    Red,
    Green,
    Blue
}
```

- Alapértelmezésben int
  - hozzárendelt egész-értékek 0-val kezdődnek, ez átállítható
- `System.Enum` a közös ősosztály

# Felsorolási típusok

C# logikai típus:

- `bool` a `true` és `false` értékekkel, szokásos műveletek
- Nincs szabványos konverziós lehetőség!

C# karakterek:

- char típus unsigned 16-bit egészek 0 és 65535 között.
- A Unicode karakter-halmaznak felel meg.
- Bár ugyanaz a reprezentációja, mint az ushort-nak, nem ugyanaz a művelethalmaz!
- A konstansokat kétféle módon lehet írni: karakter-literálként vagy egész literálként, explicit típuskényszerítéssel: `(char)10` ugyanaz, mint `'\x000A'`.

# Felsorolási típusok

SWIFT

- Alaptípus megadása
- Hozzárendelt reprezentációs érték megadható
  - Értékek lekérdezhetők
- Példa

```
enum Rank: Int {
    case Ace = 1
    case Two, Three, Four, Five, Six, Seven, Eight, Nine, Ten
    case Jack, Queen, King
}
let ace = Rank.Ace
let aceRawValue = ace.rawValue
```

# Egész típusok

Az egész típusok nagyon közel vannak a számítógépes reprezentációhoz.

A műveletek általában a szokásosak, néhol gond van az osztással és a hatványozással

# Egész típusok

Object Pascal lehetőségek:

- `ShortInt` signed8 bit
- `SmallInt` signed 16 bit (csak Delphi 2, 3)
- `Integer` signed system-dep. Delphi 1 – 16 bit
  Delphi 2-től – 32 bit
- `LongInt` signed32 bit
- `Byte` unsigned 8 bit
- `Word` unsigned 16 bit
- `Cardinal` unsigned system-dep. Delphi 1 – 16 bit
  Delphi 2-től – 32 bit

# Egész típusok

ADA:

- Előjeles egészek:
  - `type Page_Num is range 1 .. 2_000;`
  - `type Line_Size is range 1 .. Max_Line_Size;`
- Maradékosztályok:
  - `type Byte is mod 256; -- egy előjel nélküli byte`
  - `type Hash_Index is mod 97;`
- Minden egész típust úgy tekintenek, mint ami a névtelen, előredefiniált `root_integer` típusból lett származtatva.
- Egész literálok ennek az `universal_integer` osztályába tartoznak.
  - Ez a szigorú típusosság miatt fontos.

# Egész típusok

ADA:

- Az előredefiniált egész típus az Integer, van két előredefiniált altípusa:
  - `subtype Natural is Integer range 0 .. Integer'Last;`
  - `subtype Positive is Integer range 1 .. Integer'Last;`
- Az értékintervallumnak egy tetszőleges implementáció esetén tartalmaznia kell a –2**15+1 .. +2**15–1 –t.
- Megengedett, de nincs előírva:
  - `Short_Integer`
  - `Long_Integer`
  - `Short_Short_Integer`
  - `Long_Long_Integer`
  - …

# Egész típusok

ADA:

- A műveletek a szokásosak
  - minden attribútum,
  - relációs operátorok,
  - +, -,(unáris, bináris) *, / (csonkít),
  - rem (maradék),
  - mod (modulus),
  - abs,
  - ** (hatványozás Natural kitevőre)
- A modulo típusok műveletei a maradékosztályokon

# Egész típusok

C++:

- Az előredefiniált egész típusoknak 4 mérete lehet: `short int, int, long int, long long int`.
- „Longer integers provide no less storage than shorter ones.”
- Az `unsigned int`, `unsigned long int`, `unsigned short int` típusokat modulo $2^n$ aritmetikával használja (n a reprezentációban a bitek száma).
- A szabvány definiálja a minimum értékintervallumokat
- C++11 fix méretű típusok
  - `int8_t, int16_t, int32_t, int64_t`

# Egész típusok

C++

- A műveletek:

| | |
|---|---|
| Relációs | == , != < <= > >= |
| Unáris | * + - & |
| Multiplikatív | * / % |
| Additív | + - |
| incr. prefix postfix | ++ |
| decr. prefix postfix | -- |
| shift előjeles | << >> |
| Komplemens bitenként | ~ |
| Feltételes op. | ? : |
| Sizeof | |
| pointer_to_member | ->* .* |
| Értékadó op. | = *= /= %= += -= >>= <<= &= ^= != |

# Egész típusok

Java

- Az előredefiniált egész típusoknak előírt specifikációja van:

| | |
|---|---|
| byte | -128..127 |
| short | -32768..32767 |
| int | -2147483648.. 2147483647 |
| long | -9223372036854775808 .. 9223372036854775807 |
| char | '\u0000'..'\uffff', vagyis: 0..65535 |

# Egész típusok

Java

- Operátorok

| | |
|---|---|
| Relációs | == , != < <= > >= |
| Unáris | +- |
| Multiplikatív | * / % |
| Additív | + - |
| incr. Prefix postfix | ++ |
| decr. Prefix postfix | -- |
| shift előjeles, előjeln. | << >> >>> |
| komplemens bitenként | ~ |
| feltételes op. | ? : |

- További hasznos előredefiniált műveletek a csomagoló osztályokban

# Egész típusok

C#

- Előírt specifikáció van itt is
  - `sbyte` signed 8-bit $-128 \dots 127$
  - `byte` unsigned 8-bit $0 \dots 255$
  - `short` signed 16-bit $-32768 \dots 32767$
  - `ushort` unsigned 16-bit $0 \dots 65535$
  - `int` signed 32-bit $-2147483648 \dots 2147483647$
  - `uint` unsigned 32-bit $0 \dots 4294967295$
  - `long` signed 64-bit $-9223372036854775808 \dots 9223372036854775807$
  - `ulong` unsigned 64-bit $0 \dots 18446744073709551615.$
- Műveleteknél előírták a konverziókat is.

# Egész típusok

Eiffel:

- Az `INTEGER` kiterjesztett, a `COMPARABLE` és a `NUMERIC` osztályok leszármazottja.
- A reprezentáció legalább `Integer_bits` bitet használ, ez egy platform-függő konstans, amelyet a PLATFORM osztály definiál.
- A műveletek:
  - `< <= > >=` a `COMPARABLE`-ból,
  - `+ - * /` a `NUMERIC`-ból,
  - és az újak:
    - hatványozás `^`
    - egész osztás `//`
    - maradék `\\`

# Valós típusok

A valós típusok a valós számok közelítései.

- a lebegőpontos típusok - relatív pontosság
- fixpontos típusok - abszolút pontosság

A legtöbb programozási nyelv támogatja az előjeles lebegőpontos típusokat, ahol 1 bit az előjel, és a szám formátuma:




- mantissa * $10^{exponent}$, ahol  0<=abs(mantissa)<10.

A kitevőnek is lehet előjele

- A kitevőt általában az 'E' betű jelöli.

# Valós típusok

## Pascal

- A nyelv lebegőpontos valósakat használ.

| Típus: | Reprezentáció: |
|---|---|
| Single | 32 bit (1+23+8)<br>1.401E-45 .. 3.402E38 |
| Real | 48 bit (1+39+8)<br>2.9E-39 .. 1.7E38 |
| Double | 64 bit (1+52+11)<br>4.941E-324 .. 1.797E308 |
| Extended | 80 bit (1+64+15)<br>3.363E-4932 ..1.189E4932 |
| Comp | 64 bit (1+63)<br>-9.2E-18 .. 9.2E18 |

# Valós típusok

Object Pascal:

- `Single`, `Double`, `Extended`
  - IEEE nemzetközi szabvány szerint.
- Fixpontos valós típus 4 számjegy pontossággal:

| | | |
|---|---|---|
| Currency | -922337203685477.5808 .. 922337203685477.5807 | 8 byte |

  - Műveletek a szokásosak

# Valós típusok

ADA:

- A nyelv ad lehetőséget a lebegőpontos és a fixpontos típusok kezelésére, de csak egy előredefiniált lebegőpontos típus van, a `Float`.
- A lebegőpontos típusoknál a relatív pontosság, míg a fixpontos típusoknál az abszolút pontosság megadására van lehetőség. Megadható egy értékintervallum is:

```
type Real is digits 8;
type Coefficient is digits 10 range -1.0 .. 1.0;
type Mass is digits 7 range 0.0 .. 1.0E35;
```

# Valós típusok

ADA:

- Minden valós típust úgy tekintenek, mint egy előre definiált `root_real` típusból származtatott típust. A valós literálok ennek az osztályába tartoznak, így `universal_real` típusúak.
- Ha egy implementációban a lebegőpontos típusok pontossága legalább 6 számjegy, akkor a `Float` típus pontossága is legalább ennyi kell legyen.
- Megengedett, hogy egy implementáció támogasson további előre definiált lebegőpontos típusokat, pl.: `Short_Float`, `Long_Float`, `Short_Short_Float`, `Long_Long_Float`...
- A szokásos műveletek, minden attribútum, relációs operátorok, `+`, `-`,(unary, binary) `*`, `/` `abs`, `**` (hatványozás).

# Valós típusok

C++:

- Az előre definiált valós típusok a `float`, `double` és a `long double`.
- „The type double provides no less precision then float, and the type long double provides no less precision then double.” …

# Valós típusok

C++:

- Műveletek

| | |
|---|---|
| egyenlő, nem egyenlő összehasonlítás | == , != < <= > >= |
| indirekció, előjel, cím | * + - & |
| szorzás, osztás, moduló | * / % |
| összeadás, kivonás | + - |
| növelés prefix/postfix | ++ |
| csökk. prefix/postfix | -- |
| léptetés balra/jobbra | << >> |
| bitenkénti komplemens | ~ |
| feltételes kifejezés | ? : |
| obj/típus mérete | sizeof |
| tagkiválasztás | ->* .* |
| értékadások | = *= /= %= += -= >>= <<= &= ^= != |

# Valós típusok

Java:

- Az előredefiniált lebegőpontos típusoknak előírt pontossága van

| | |
|---|---|
| float | 32 bit IEEE 754 |
| double | 64 bit IEEE 754 |

- A `float` típus $s*m*2^e$ alakú, ahol
  - $s = +1$ vagy $-1$,
  - $m$ egy pozitív egész, kisebb, mint $224$
  - $e$ egy egész a $-149..104$ intervallumból.
- A `double` típus $s*m*2^e$ alakú, ahol
  - $s = +1$ vagy $-1$,
  - $m$ egy pozitív egész, kisebb, mint $253$
  - $e$ egy egész a $-1075..970$ intervallumból

# Valós típusok

Java:

- Érdekesség:
  - a pozitív és negatív végtelenek (`POSITIVE_INFINITY`, `NEGATIVE_INFINITY`)
  - speciális Not-a-Number (`NaN`) érték is szabvány.
    - Ha x a `NaN` mi az értéke az x != x -nek?
- Műveletek a `NaN`–ra nem értelmezettek
  - További hasznos műveletek a `Float`, `Double` és `Math` osztályokban.

# Valós típusok

Java:

- Műveletek a `NaN`–ra nem értelmezettek
  - További hasznos műveletek a `Float`, `Double` és `Math` osztályokban.

| | |
|---|---|
| Relációs | == , != < <= > >= |
| Unáris | +- |
| Multiplikatív | * / % |
| Additív | + - |
| Incr. prefix, postfix | ++ |
| Decr. prefix, postfix | -- |
| Shift előjeles és előjel nélküli | << >> >>> |
| Komplemens bitenként | ~ |
| Feltételes op. | ? : |

# Valós típusok

C#:

- Javához hasonló - 2 lebegőpontos típus: `float` és `double`

| | |
|---|---|
| float | 32 bit IEEE 754 |
| double | 64 bit IEEE 754 |

- Pozitív és negatív 0
  - legtöbbször ugyanaz, de osztásnál más lehet.
- Pozitív és negatív  végtelen (∞)
  - 1.0 / 0.0 a pozitív végtelen
  - –1.0 / 0.0 a negatív.
- A Not-a-Number érték (NaN).
  - 0.0/0.0

# Valós típusok

C#

- Decimal típus a pénzügyi számításokhoz:
  - 128-bites adattípus,
    értékei: $1.0 \times 10^{-28}$ … $7.9 \times 10^{+28}$,
    28-29 szignifikáns számjeggyel.
  - Lényeg, hogy a tízes számrendszerbeli törtértékeket is pontosan ábrázolja, műveleteknél szokásos módon kerekít.

# Valós típusok

Eiffel

- A REAL osztály kiterjesztett, a `COMPARABLE` és a `NUMERIC` osztályok leszármazottja.
- A reprezentáció legalább `Real_bits` bitet használ, ez egy platform-függő konstans, amelyet a `PLATFORM` osztály definiál.
- A műveletek
  - `< <= > >=` a `COMPARABLE`-ból,
  - `+ - * /` a `NUMERIC`-ból,
  - és az újak
    - hatványozás ^
    - stb.

# Típusok Vezérlési szerkezetek

KÖVETKEZŐ HÉTEN




# Programozási nyelvek és módszerek

# Pointerek

# Pointer és referencia típusok

Egy pointer (és egy referencia) egy olyan objektum, amely megadja egy másik objektum címét a memóriában

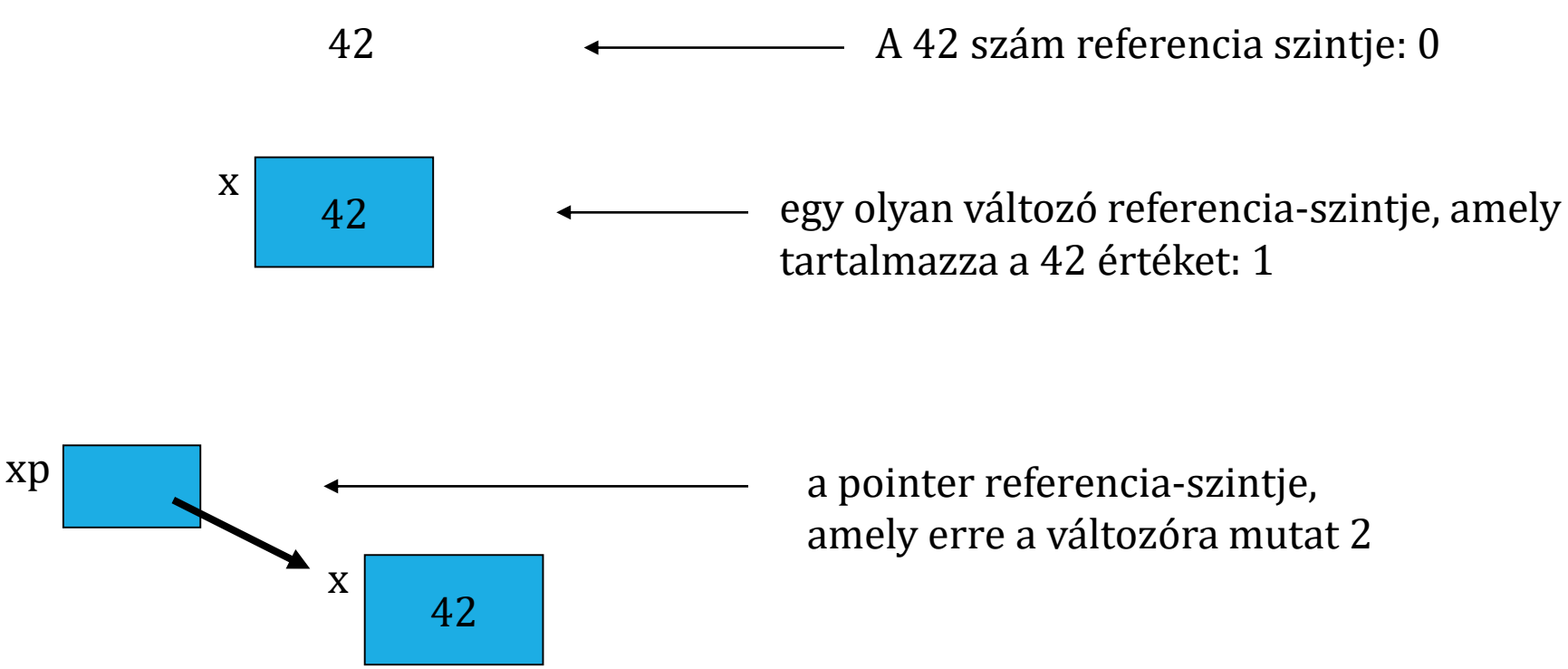
Egy pointer értéke egy memóriacím

- gépi nyelvekben az indirekt címzés lehetősége motiválta a pointerek létrehozását.

Vannak típusos és típus nélküli pointerek.

# Referencia szint

Pointerek segítségével magasabb referencia-szinten hivatkozhatunk objektumainkra.



# Mire kellenek a mutatók?

Hatékonyság

- ahelyett, hogy nagy adatszerkezeteket mozgatnánk a memóriában, sokkal hatékonyabb, ha az erre mutató pointert másoljuk, mozgatjuk.

x

| 23 | 10 | 11 | 17 | 0 | 34 | 28 | 22 | 55 | 88 |
|---|---|---|---|---|---|---|---|---|---|
| 4 | 7 | 0 | 0 | 6 | 8 | 1 | 9 | 10 | 3 |

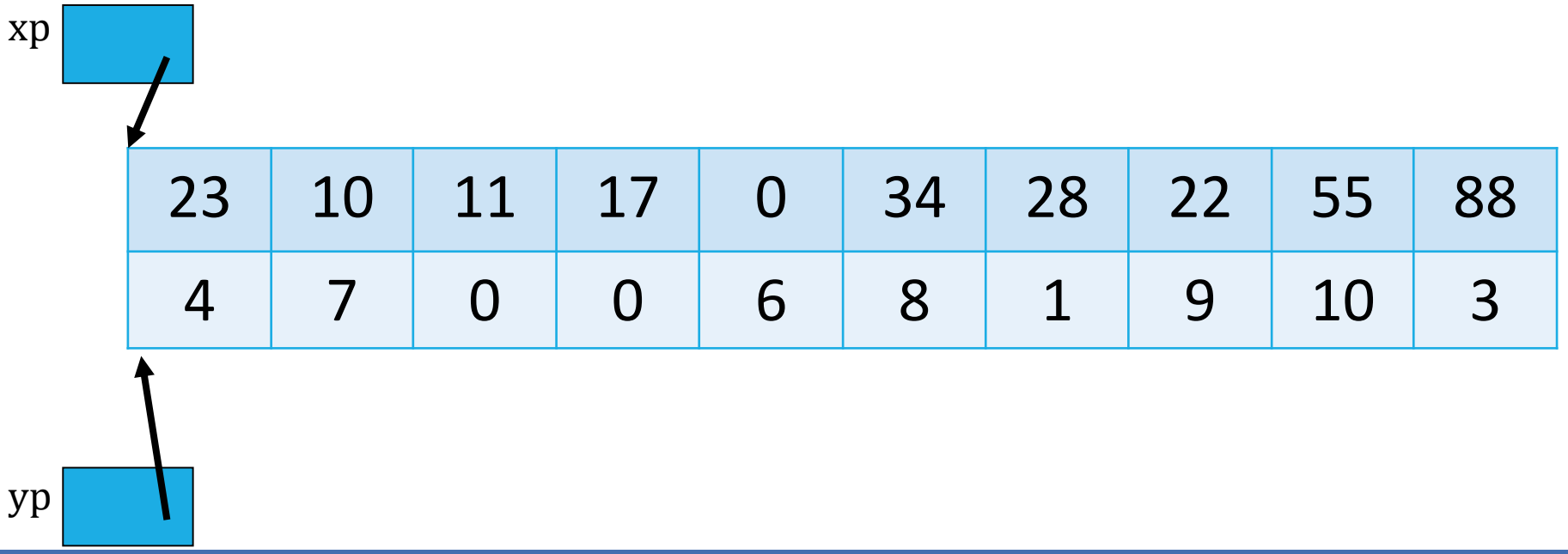
y

| 23 | 10 | 11 | 17 | 0 | 34 | 28 | 22 | 55 | 88 |
|---|---|---|---|---|---|---|---|---|---|
| 4 | 7 | 0 | 0 | 6 | 8 | 1 | 9 | 10 | 3 |

# Mire kellenek a mutatók?

Hatékonyság

- ahelyett, hogy nagy adatszerkezeteket mozgatnánk a memóriában, sokkal hatékonyabb, ha az erre mutató pointert másoljuk, mozgatjuk.
  - itt  vigyázni kell az osztott használatra! – jó, ha van read-only elérési lehetőség is



# Mire kellenek a mutatók?

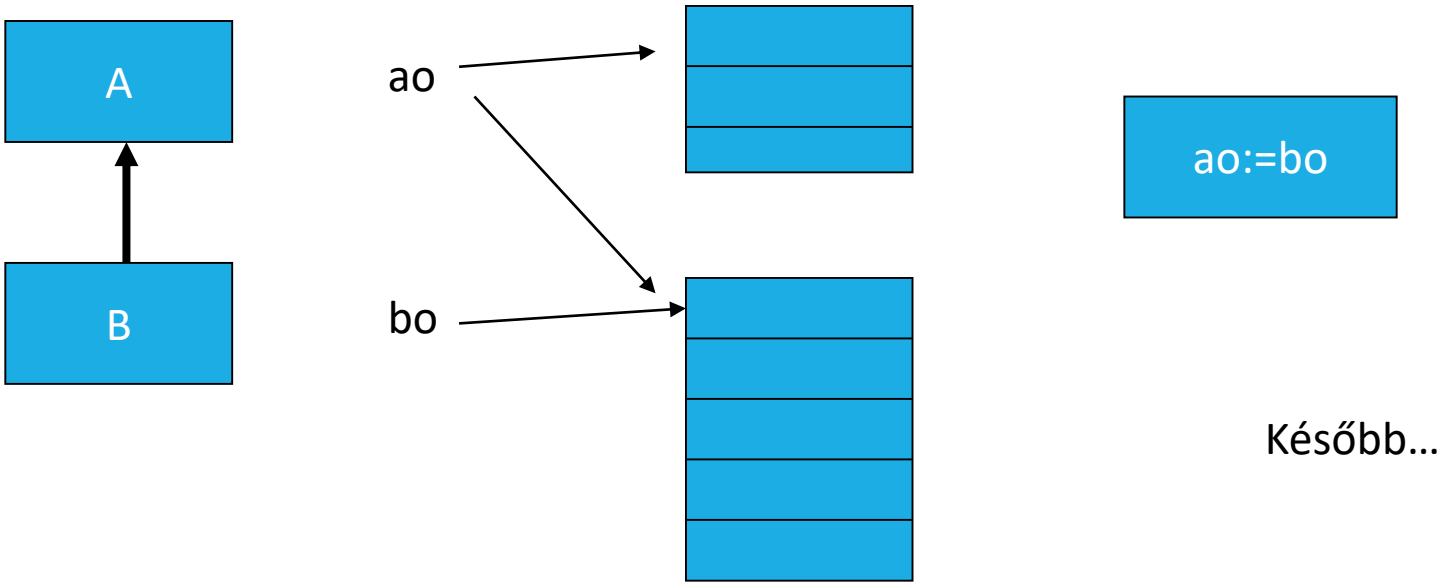
Dinamikus adatszerkezetek építéséhez



# Mire kellenek a mutatók/referenciák?

Objektumorientált funkciókhoz

- a programozási nyelvekben a polimorfizmust akkor tudjuk támogatni, ha a változók objektumokra való referenciákat tartalmaznak.



# A szokásos műveletek

Értékadás - pointerek között, hivatkozott objektumok között (copy, clone, deep-copy, deep-clone!)

Egyenlőség vizsgálat - ha két ugyanolyan típusú pointer ugyanarra az adatszerkezetre mutat (ez is több szinten lehet)

Dereferencing - a *mutatott* objektum részére vagy egészére való hivatkozás

Referencing - egy objektum címe

Új objektum dinamikus allokálása

Egy objektum deallokálása - explicit művelettel vagy implicit módon egy garbage collector-ral

Néha (pl. C, C++) összeadás, kivonás is megengedett

# „Csellengő” pointerek

„Csellengő” pointer: kísérlet olyan változó elérésére, ami már nem létezik.

```
#include <iostream>
int *r;
double *r2;
void f() {int v; r=&v; }
void g() {double v; v=2.1; r2=&v; }
int main() {
  f(); g(); *r=3; *r2=1.2;
  cout << "dangling *r="<<*r;
  cout << "\n *r2 " <<*r2 << ".\n";
}
```

**Compiler, builder:**
**0 error(s), 0 warning(s)**
**Az eredmény megjósolhatatlan!**

# „Csellengő” pointerek 2.:

```
int main() {
   int *j,*i;
   double *d;
   j=new int;
   *j=3;
   i=j;
   delete j;
   d=new double;
   *d=4.2;
   cout << *i;
}
```

**Compiler, builder:**
**0 error(s), 0 warning(s)**
**Az eredmény megjósolhatatlan!**

# Pointerek az egyes nyelvekben

A programozási nyelvek között a lehetséges különbségek:

- Csak konkrét típusra mutató pointerek megengedettek, vagy vannak típus nélküli pointerek is?
- Csak dinamikusan allokált objektumokra mutathat pointer, vagy „normál” változókra is?
- Lehetnek-e alprogramra mutató pointerek is?
- Milyen fajta konstans pointerek megengedettek?
  - Egy tömbnév C-ben egy konstans pointer a tömb objektum 0. elemére.
- Kötelező a pointer típusoknak önálló nevet adni, vagy csak a mutatott típust kell megadni?
  - `type PInteger is access to Integer;` (ADA-ban)
  - `int * x;` (C-ben)

# Pointerek az egyes nyelvekben

Milyen biztonságosan kezelhető a „csellengő” pointerek problémája?

Mi a megengedett műveletek halmaza?

Kapnak a pointer változók kezdeti (üres) értéket a deklarációnál?

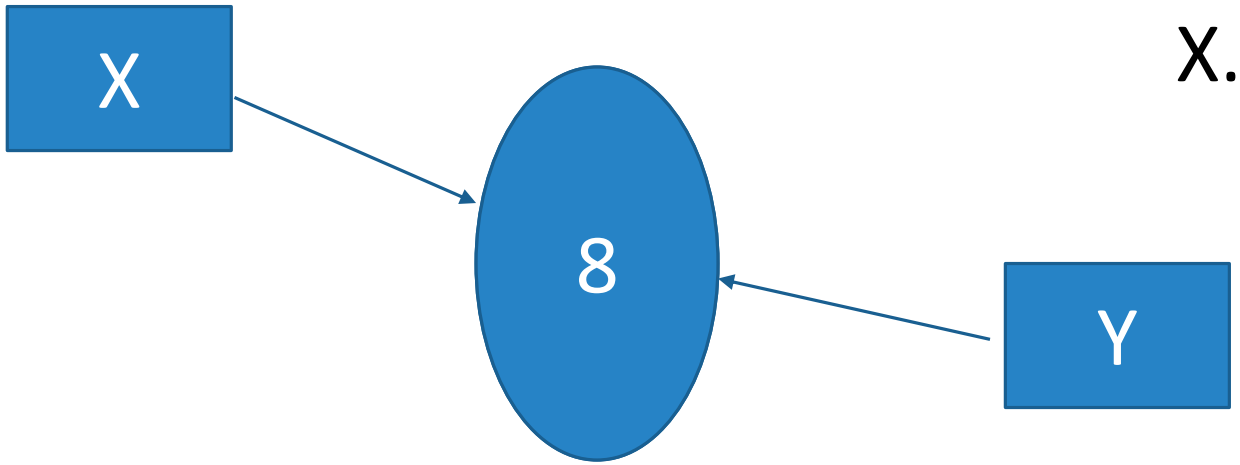
Lehetséges-e ugyanazt az adatot két (vagy több) pointeren keresztül is változtatni/elérni?

# Pointerek

CLU

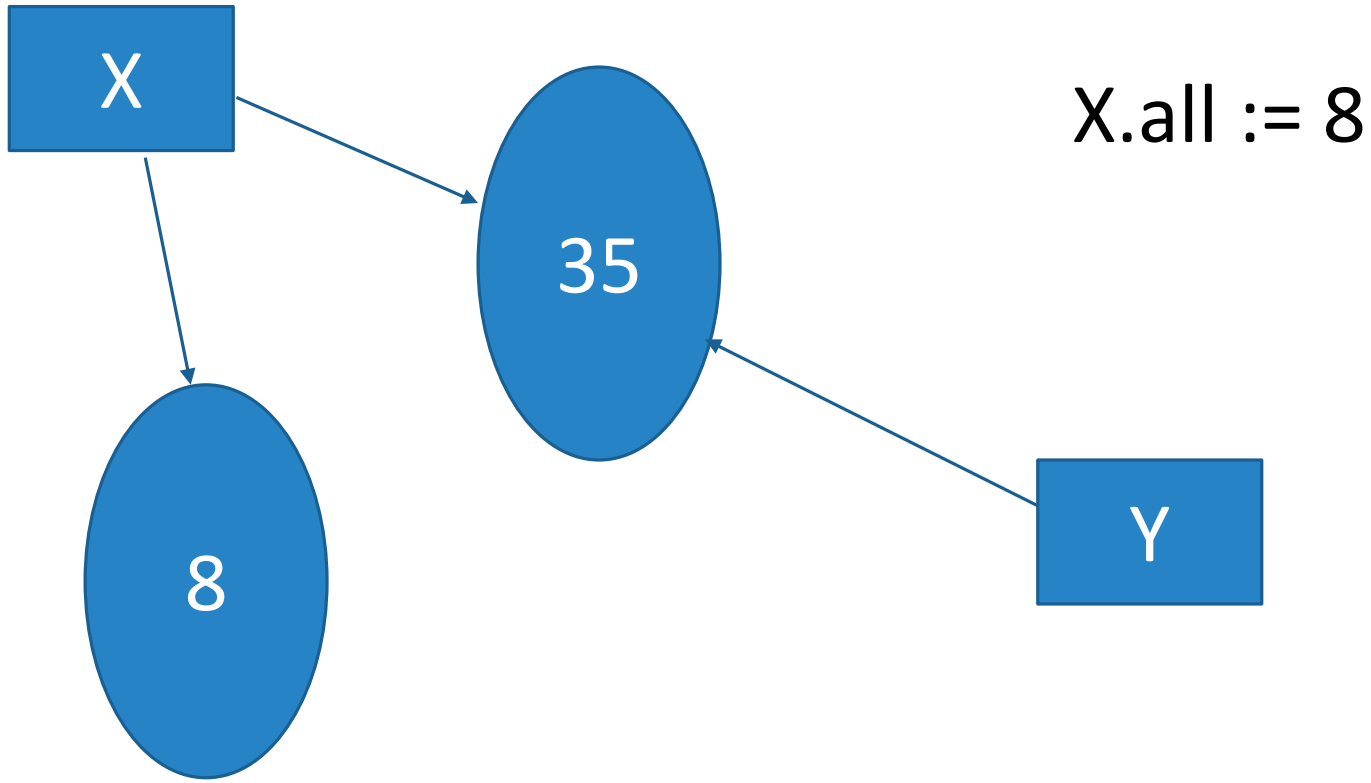
- A CLU-ban nincs hagyományos pointer típus.
- A program végrehajt műveleteket objektumokon
- Az objektumok mint egy univerzum részei léteznek, a program változói hivatkoznak ezekre az objektumokra
  - Garbage collection a felszabadításra
- A programban kétféle objektum lehet:
  - mindig ugyanaz az értéke (immutable
  - változhat az értéke (mutable)

# Mutable – változtatható




X.all := 8

# Immutable – nem megváltoztatható



# Pointerek

C++

- A legtöbb `T` típusra, `T*` a megfelelő pointer típus:
  - `int *p;`
- A tömbökre és függvényekre mutató pointereknek kicsit bonyolultabb jelölése van
  - `int (*vp)[10]; // pointer 10 int tömbjére`
  - `int (*fp) (char, char*);`
    - függvényre mutató pointer, amelynek
    - (char, char*) argumentumai vannak és egy egészet (int) ad vissza

# Pointerek

C++

- A megengedett műveletek:
  - Dereferencing: prefix *
  - Dinamikus allokálás `new`
  - Deallokálás `delete`
  - értékadás: `=`
  - egyenlőség: `==`
  - additív műveletek `+ -`
  - increment, decrement `++ --`
  - member ref. `.* ->*`

# Pointerek

C++

- van egy speciális operátor, az „address_of” „&”, ennek segítségével adhatjuk értékül változók címét pointereknek:
  - `int i = 10;`
  - `int *pi = &i; // a pi pointer az i változóra vonatkozik`
  - `int j = *pi; // j-t 10-re állítjuk`
- Az '&' operátorral létrehozhatjuk objektumok referenciáit is
  - egy referencia úgy tekinthető mint egy konstans pointer, ami mindig automatikusan dereferenciát hajt végre:
    - `int &r = i; // r és i ugyanarra vonatkozik`
    - `r = 2;  // i=2`
- Az increment, decrement, … veszélyesek lehetnek, vigyázzunk, ne keverjük össze a jelentését!
  - `pi++` a pointert inkrementálja, és a következő memóriacímre fog mutatni, ennek akkor van értelme, ha pi egy tömbre mutat, míg `r++` inkrementálja `i` értékét

# Pointerek

Java

- Nincs hagyományos pointer típus
- A változókban kétféle érték tárolható
  - primitív értékek (egy numerikus típusból vagy egy logikai)
  - referencia értékek
- Az objektumokat (osztályok példányai vagy tömbök) referenciákkal kezeli.
- Ugyanarra az objektumra számos referencia hivatkozhat.
- Objektumok referenciáinak műveletei:
  - mező elérés, metódus hívás, casting, string concatenation, instanceof, '==' '!=' (referencia egyenlőségének vizsgálata) stb.

# Pointerek

Eiffel

- Itt sincsenek hagyományos pointer típusok.
- A változókban kétféle érték tárolható - kiterjesztett értékek és referencia értékek.
- Ugyanarra az objektumra számos referencia hivatkozhat.

# Pointerek

C#

- A referencia típusok objektumait kezelhetjük referenciákkal.
- Egy „unsafe” környezetben egy típus lehet pointer is,  erre számos művelet megengedett (pl. a ++, -- is).

# Adattípusok definiálása

# Hogyan definiálhatunk új adattípusokat?

Példák

- Pascal
  - `type <typn>=<value desc>;`
  - `type myint=integer;`
- C++
  - `typedef <value desc> <typn>;`
  - `typedef int myint;`
- ADA
  - `subtype <typn> is <typ1>;` `type <typn> is new <typ1c>;`
  - `subtype Int is Integer;` `type My_Int is new Integer;`
- Java, Eiffel, C#, Swift, Kotlin, stb.
  - `class`
  - `(struct)`
  - `enum`

# Melyek a megengedett típus-konstrukciók?

Iterált
- egy kiinduló típusból

Direkt szorzat
- több kiinduló típusból

Unió
- több kiinduló típusból

# Tömb típusok

„Egy tömb egy olyan adatszerkezet, amely azonos típusú elemek sorozatait tartalmazza.”

Általában egy tömb egy leképezés egy folytonos diszkrét intervallumról elemek egy halmazára, nem igazi sorozat típus.

- Tömbnév(indexértékek) → elem

A diszkrét intervallum elemeit hívjuk index értékeknek.

Az elemek száma ebben az intervallumban definiálja a tömb *méretét*.

# A legfontosabb kérdések

- Milyen adattípusok lehetnek tömb típusok indextípusai?
- Mi lehet tömb típusok elemtípusa?
- Tartalmazzák-e a tömb típusok az indexhatárokat? És a tömb objektumok?
- Mikor dől el a mérete, a helyfoglalása?
- Van-e indextúlcsordulás-ellenőrzés?
- Van-e többdimenziós tömb?
  - Van-e altömb (szelet) képzés?
  - Van-e teljes tömbre vonatkozó értékadás? (kezdő értékadás?)
  - Van-e tömbkonstans?
- Megváltoztatható-e egy tömb mérete?
- Rögzített méretű sorozat vagy nem?

# Indexelés

Az alapművelet az *indexelés*

- `A[i]`
- Elvárás, hogy az `A` tömb `i`. elemét gyorsan el kell tudni érni!

Általában az elemek ugyanahhoz a típushoz tartoznak

- vagy egy adott típus lehetséges leszármazottai is lehetnek
- nem mindig: Smalltalk esetén az elemek különböző típusúak is lehetnek

# Tömbök

ADA

- Tömb típusok definiálhatók rögzített és megszorítás nélküli indexhatárokkal:
  - `type A is array(Integer range 2 .. 10) of Boolean;`
  - `type Vect is array(Integer range <>) of Integer;`
  - `type Matr is array(Integer range <>,Integer range <>) of Integer;`
- A konkrét indexhatárokat az adott objektum deklarációjánál kell meghatározni:
  - `V : Vect(1 .. 30);`
  - `B : Matr(1 .. 2, 1 .. 4);`
- Gyakran használják alprogramok paramétereként, sablonoknál.

# Tömbök

ADA

- Az index típusa tetszőleges diszkrét típus lehet, az elemek típusa tetszőleges típus
- A definíciókat futási időben értékeli ki, az aktuális indexhatárok nem kell, hogy statikusak legyenek.
- Tömbök szeletei is létrehozhatók:
  - `V(2..12)`
    - de: `V(1..1) <-> V(1)!`
- Megengedett az értékadás azonos típusú tömbök között:
  - `V(1..5):= V(2..6);`
- Lehetséges értékadás tömb „aggregátokkal” is:
  - `V:=(1..3=>1, others=>0);`

# Tömbök

## ADA

- Három előre definiált string típus:
  - ```
    type String is array (Positive range <>)
       of Character;
    ```
  - ```
    type Wide_String is array (Positive range <>)
       of Wide_Character;
    ```
  - ```
    type Wide_Wide_String is array(Positive range <>)
       of Wide_Wide_Character;
    ```
- Vannak speciális attribútumai:
  - `A'First/A'First(N)`
  - `A'Last/A'Last(N)`
  - `A'Range/A'Range(N)`
  - `A'Length/A'Length(N)`
    ```

# Tömbök

```
generic
   type Elem is private;
   type Index is (<>);
   type Vekt is array (Index range <>) of Elem;
   procedure Glinker (V: Vekt; E: Elem;
                      Found: out Boolean; Ind: out Index);



procedure Glinker (V: Vekt; E: Elem;
                   Found: out Boolean; Ind: out Index) is
begin
   Ind:=V'First;
   Found :=V(V'First)=E;
   while not Found and Ind<V'Last loop
      Ind:=Index'Succ(Ind);
      Found:=V(Ind)=E;
   end loop;
end;
```

# Tömbök

C++:

- Egy `T` típusra:
  - `T x[size]` a `T` típusú elemek `size` méretű tömbje.
    Az indexek 0 és size-1 között.
    - `float v[3];  // 3 float tömbje`
    - `int a[2][5]; // 5 int két tömbje`
- Kezdeti érték adható:
  - ```
    char v[2][5] = {{'a','b','c','d','e'},
                    {'0','1','2','3','4'}};
    ```
- A pointerek és tömbök szoros kapcsolatban vannak
  - egy tömbnév mindig a tömb nulladik elemére hivatkozik, így használható a pointeraritmetika tömbökre.
- Nem tudja a méretét!
- STL vector template osztály!
    ```

# Tömbök

Java:

- „Java arrays are objects, are dynamically created and may be assigned to variables of type Object.”
- `int [] ai;  // egészek tömbje`
- `short [][] as1,as2; // as1 és as2 short-ok tömbjének tömbjei`
- ha a '[ ]' –t a változó neve után írjuk, akkor csak ez a változó lesz tömb:
  - `long l, al[]; // l egy long típusú változó,`
    `// al long  típusú elemek tömbje`
- Tömb létrehozása:
  - `a = new int[20];`
- A tömb mérete lekérdezhető: `a.length`
- Kezdeti érték adható:
  - `String []colours = {"red","white","green"};`

# Tömbök

## Java

- Egy tömbök tömbjében (ami a Java esetén gyakran több dimenziós tömbnek hívnak a dokumentációkban) az elemeknek lehet különböző mérete:

```
int[][] m = new int[3][];
for (int i=0; i<m.length; i++ ) {
  m[i] = new int[i+1];
  for (int j=0; j<m[i].length; j++)
    m[i][j] = 0;
}
```

- A szövegek kezelését a `String` és `StringBuffer` (`StringBuilder`) osztályokkal oldották meg.
  - Karakterek egy tömbje nem `String`!

# Tömbök

C#

- A tömb elemeinek számozása 0-val kezdődik
- Kétféleképpen lehet deklarálni
  - adott hosszúságú vagy dinamikus.
- A nyelvben a tömbök objektumok, a deklaráció után szükség van a tömb példányosítására (new)
- Inicializálásra: { }
- Adott méretű tömb deklarálása:

```
int[] Tomb;    Tomb = new int[3];
```

- Ugyanez a tömb inicializálva:

```
Tomb = new int[3]  { 1,2,3 }
```

- Dinamikus tömb létrehozása inicializálással:

```
Tomb = new int[] { 1,2,3 }
```

- A deklarációval egybekötött inicializáció:

```
int[] Tomb = new int[3] { 1,2,3}
```

- Ha egy tömböt nem inicializálunk, akkor a tömb elemei automatikusan inicializálódnak az elem típusának alapértelmezett inicializáló értékére.

# Tömbök

C#

- A tömbök lehetőségei:
  - egydimenziós tömbök,
  - többdimenziósak vagy négyszögszerűek,
  - kesztyűszerűek (tömbök tömbjei)
  - kevert típusúak (az előzőekből)
- Példa egy kesztyűszerű dinamikus tömbre:

```
int[][] numArray = new int[][] { new int[] {1,3,5},
                                 new int[] {2,4,6,8,}};
```

- minden tömb típus a `System.Array` bázistípusból „származik”.
  - Az `Array` osztály egy absztrakt bázisosztály, de a `CreateInstance` metódusa létre tud hozni tömböket.
  - Ez biztosítja a műveleteket a tömbök létrehozásához, módosításához, bennük való kereséshez, illetve rendezésükhöz.

# Tömbök

C#

- Az `Array` osztály tulajdonságait megadó függvények:
  - `IsFixedSize` - rögzített hosszúságú-e
  - `IsReadOnly` - írásvédett-e
  - `IsSynchronized` - a tömb elérése kizárólagos-e (szálbiztos)
  - `Length` - a tömb elemeinek száma
  - `Rank` - a tömb dimenzióinak száma
  - `SyncRoot` - Visszatér egy objektummal, amit a tömb szinkronizált hozzáféréséhez használhatunk
- Az `Array` osztályban még számos szolgáltatás:
  - `BinarySearch` - bináris keresés a tömbön
  - `Clear` - minden elemet  töröl a tömbből és az elemszámot 0-ra állítja
  - `Clone` - másolatot készít
  - `Copy` - egy tömb részét átmásolja egy másik tömbbe, végrehajtja az esedékes típuskényszerítést és csomagolást (boxing).

# Tömbök

C#

- Az `Array` osztályban még számos szolgáltatás:
  - `CopyTo` - átmásolja az elemeket egy egy-dimenziós tömbből egy másik egy-dimenziós tömbbe egy megadott indextől kezdve.
  - `CreateInstance` - Létrehoz egy tömb példányt.
  - `GetEnumerator` - Visszatér egy `IEnumerator-ral` a tömbhöz.
  - `GetLength` - az elemek száma
  - `GetLowerBound` - Megadja a tömb alsó korlátját.
  - `GetUpperBound` - Megadja a tömb felső korlátját.
  - `GetValue` - megadott indexű elem értéke.
  - `IndexOf` - egy-dimenziós tömbben az első érték indexe.
  - `Initialize` - Egy értéktípusú tömbben minden elemre meghívja az elemek alapértelmezett konstruktorát.
  - `LastIndexOf` - Visszaadja az egy-dimenziós tömbben az utolsó érték indexét.
  - `Reverse` - Megfordítja a tömb vagy tömbrészlet bejárási irányát.
  - `SetValue` - A megadott elemet beteszi a megadott helyre a tömbben.
  - `Sort` - A tömbön rendezést hajt végre.

# Tömbök

Eiffel

- Az Eiffel tömbök az `ARRAY[G]` sablon osztály példányai.
- A stringeket a `STRING` osztály objektumai valósítják meg.

# Asszociatív tömbök

Egy asszociatív tömb elemek egy rendezetlen halmaza, amelyet megegyező számú, kulcsnak nevezett értékek indexelnek.

Ezeket a kulcsokat is tárolni kell

- az elemek így (kulcs, érték) párok

# Asszociatív tömbök

## Python

- Beépített „dictionary” típus, kulcs-érték párok tárolására
  - A kulcsnak „hashable” típusnak kell lenniük
  - A kulcsokat használunk az elemek elérésére.
- Példa
  - `tel = {'jack': 4098, 'sape': 4139}`
- Ezekre a változókra
  - a `list()` függvény a kulcsok listáját adja vissza,
  - a `del` utasítással lehet kulcs szerint törölni
    - `del tel['sape']`
  - for ciklussal végig lehet járni a tárolt kulcs-érték párokat
    - `for k, v in tel.items():`
  - az `in` kulcsszó megadja, hogy egy adott kulcs szerepel-e benne
    - `'jack' not in tel`

# Asszociatív tömbök

A Java, a C++, az Eiffel szabványos osztálykönyvtárában is megtalálhatók

A .NET keretrendszer osztálykönyvtárában is

Egyéb nyelvek:

- PHP,
- Perl,
- Ruby,
- Lua,
- Pike,
- stb.

# Rekord típusok

A direkt szorzat és az unió típuskonstrukciókat számos nyelvben az úgynevezett rekord típusok segítségével valósítják meg.

Ha adott két típus, S és T, direkt szorzatukat S × T jelöli.

- $S \times T = \{(x,y) | x \in S; y \in T\}$
- Ez általánosítható több halmazra is:
  $S_1 \times S_2 \times \ldots \times S_n$

A direkt szorzat terminusaival érthetjük meg:

- COBOL, Pascal, Ada stb. rekordok
- Algol68, C, C++, C# stb. struktúrák

# Rekord típusok

```
record
  <name1> : <type1>;
  <name2> : <type2>;
  …
  <namek> : <typek>;
end
```

A komponenseket a rekord (struktúra) mezőinek hívják.

Az alapművelet a komponens kiválasztás.

# Rekord típusok

Lehet-e paramétere a típusnak?

Van-e kezdő értékadás a mezőkre?

Van-e teljes rekordra vonatkozó értékadás?

Van-e rekord konstans?

Hogyan működik a kiválasztás művelet?

# Rekord típusok

Pascal

```
type rektipnev = record
    mnev1 :  tipus1;
    ...
     mnevn :  tipusn;
end;
```

- változó deklarálása
  - `rek: rektipnev;`
- hivatkozás ponttal
  - `rek.mnev1`
- csak mezőnkénti értékadás lehetséges

# Rekord típusok

## Pascal

- Speciális utasítás, aminek a segítségével a rekord mezőire közvetlenül tudunk hivatkozni:

```
type Date= record
    Year : Integer;
    Month : 1..12;
    Day : 1..31;
end;
var R1, R2 : Date;
begin
  R1 := R2; {értékadás megengedett}
  with R1 do begin
    Year := 2011; Month:=9; Day:=29;
    Year := R2.Year;
end;
```

# Rekord típusok

C++

```
struct strnev {
        tipus1 mnev1;
        ...
        típusn mnevn;
};
```

- változó deklarálása
  - `strnev x;`
- hivatkozás ponttal
  - `x.mnev1`

# Rekord típusok

C++

- A tömbökre használt jelölés alkalmazható struktúrákra is. pl.:

```
struct address{
  long number;
  char* street;
  char* town;
  int zip;
}
address a = {1, "Korkeakoulunkatu", "Tampere", 33720}
```

- Az inicializálásra a konstruktorok jobban  használhatóak.
- Értékadás megengedett, de az egyenlőségvizsgálat nem előre definiált.
  - A felhasználó definiálhat operátorokat rá

# Rekord típusok

ADA

```
type Complex is record
          Re: Float;
          Im: Float;
end record;
```

- változó deklarálása
  - `C: Complex;`
- a komponenseire ponttal hivatkozhatunk
  - `C.Re`
    `C.Im`
- megengedett az értékadás is
  - `C1, C2: Complex;`
    `C1 := C2;`

# Rekord típusok

ADA

- A rekord diszkriminánsa(i)
  - a típus paramétere
  - több diszkriminánsa is lehet egy típusnak
  - a rekord diszkrimináns diszkrét típusú

```
type Szöveg(Hossz: Natural) is record
           Érték: String(1 .. Hossz) := (others=>’ ’);
           Pozíció: Natural := 0;
end record;
```

# Rekord típusok

C#:

- a rekord (struct) érték típus
- az osztály (class) referencia típus

Bizonyos programozási nyelvekben nincs rekord típus, a tervezők az osztályok használatát javasolják helyette

- SmallTalk
- Eiffel
- Java
- …

# Uniók és variáns rekordok

Az unió típusértékhalmaza az unió komponensei típusértékhalmazának az uniója.

- Például bútor:
  - szék vagy asztal vagy szekrény
  - színe, anyaga – van mindegyiknek
  - egyéb speciális jellemzők – külön-külön

A variáns rekordok olyan direktszorzatok, ahol a direktszorzat egy – az utolsó – komponense unió.

# „Választó” típusműveletek

A programozási nyelvek a megbízhatóság különböző szintjén támogatják ezt az adatszerkezetet.

Az unió típusnak

- van egy speciális, tag-nek nevezett komponense, és
- egy kiválasztási mechanizmusa, ami megadja a tag különböző értékeinek megfelelő alstruktúrákat

Ha a tag-et tárolja a rekord, és az alkomponensek elérhetősége ennek aktuális értékétől függ,

- akkor ez egy „megkülönböztetett” (discriminated) unió,
- különben ez egy „szabad” (free) unió.

# Unió (Variáns rekord)

Meg lehet-e állapítani, hogy a rekord melyik változat szerint lett kitöltve?

Ki lehet-e olvasni a kitöltéstől eltérő változat szerint?

Szerkezete (gyakran):

```
case <tag-name> : <tag-type> of
   <const1> : ( <fields1>);
   <const2> : ( <fields2>);
   …
   <constv> : ( <fieldsv>);
```

# Unió (Variáns rekord)

A szabad uniók esetében:

- A tag mezők használata opcionális,
- A fordító nem ellenőrzi a kiválasztott mező és a tárolt érték konzisztenciáját.
- Ez a nyelv típusrendszerét megbízhatatlanná teszi.

A megkülönböztetett unió esetében fontos kérdés, hogyan lehet új értéket adni a tag mezőnek.

- A tag értéket beállítja a rekord létrehozásakor.
- Míg a program képes kell legyen új értéket adni a rekord „normális” komponenseinek, a tag megváltoztatása a rekord szerkezet megváltozását vonja maga után!

# Példa

Pascal:

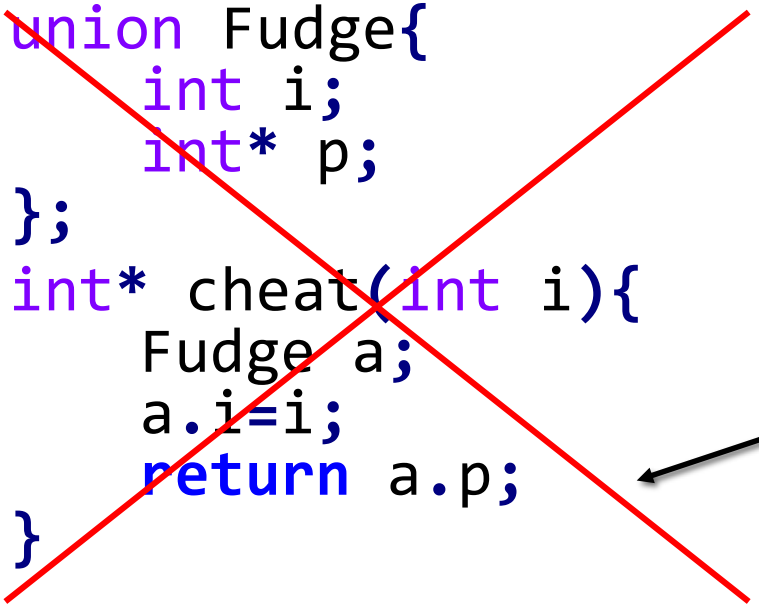
```
type Listptr=^Listnode;
type Listnode=record
       Next: Listptr;
       case Tag: Boolean of
           False: (Data:char);
                  True: (Down: Listptr)
    end;
var p, q: Listptr;
p^.Tag:=true;
p^.Down:=q;
p^.Tag:=false;
writeln(p^.Data);
```

**HIBA**

# Példa

C++:

```
union typname{
    typ1 field1;
    ...
    typm fieldm;
};
union Fudge{
    int i;
    int* p;
};
int* cheat(int i){
    Fudge a;
    a.i=i;
    return a.p;
}
```



**Jó ez??**

# Unió (Variáns rekord)

ADA

```
type Állapot is (Egyedülálló, Házas, Özvegy, Elvált);
subtype Név is String(1..25);
type Nem is (Nö, Férfi);
type Ember (Családi_Áll: Állapot := Egyedülálló) is
  record
     Neve: Név;
     Neme: Nem;
     Születési_Ideje: Dátum;
     Gyermekek_Száma: Natural;
     case Családi_Áll is
       when Házas  => Házastárs_Neve: Név;
       when Özvegy => Házastárs_Halála: Dátum;
       when Elvált => Válás_Dátuma: Dátum; Gyerekek_Gondozója: Boolean;
       when Egyedülálló => null ;
     end case;
  end record;
```

# Unió (Variáns rekord)

ADA

- `Hugó: Ember(Házas);`
  - `Családi_Áll, Neve, Neme, Születési_Ideje, Gyermekek_Száma, Házastárs_Neve`
- `Eleonóra: Ember(Egyedülálló);`
  - `Családi_Áll, Neve, Neme, Születési_Ideje, Gyermekek_Száma`
- `Ödön: Ember(Özvegy);`
  - `Családi_Áll, Neve, Neme, Születési_Ideje, Gyermekek_Száma, Házastárs_Halála`
- `Vendel: Ember(Elvált);`
  - `Családi_Áll, Neve, Neme, Születési_Ideje, Gyermekek_Száma, Válás_Dátuma, Gyerekek_Gondozója`
- `Aladár: Ember;  -- Egyedülálló`
  `Családi_Áll, Neve, Neme, Születési_Ideje, Gyermekek_Száma`
- Helytelen (futási idejű hiba, altípus-megszorítás megsértése):
  - `Hugó.Válás_Dátuma, Aladár.Házastárs_Neve`

# Unió (Variáns rekord)

ADA

- Megszorítatlan altípus használata
  - `Aladár : Ember;      -- alapért. Egyedülálló`
  - `Aladár := (Házas, ....);`
  - `Aladár := Elek;`
  - `Aladár := Hugó;`
- A szerkezetét megváltoztathatjuk, diszkriminánsostul
- Csak úgy, ha az egész rekord értéket kap egy értékadásban

# Unió (Variáns rekord)

Bizonyos programozási nyelvekben nincs unió típus, a tervezők az osztályok és az öröklődés használatát javasolják helyette:

- SmallTalk,
- Eiffel,
- Java,
- C#

# Halmaz

Speciális iterált típus

- Mi lehet az eleme?
- Hány eleme lehet?
- Megvannak-e a „hagyományos” halmazműveletek?

# Halmaz

Pascal

- Alaphalmaz: diszkrét típus
- Elemek száma: max.  256
- Értékek sorszáma csak 0..255 között
  - `type Small_Letters = set of 'a'..'z';`
  - `type Digits = set of '0'..'9';`
  - ```
    type Day = (Monday, Tuesday, Wednesday, Thursday,
                Friday, Saturday, Sunday);
    ```
  - ```
    type Days = set of Day;
    var A, B : Days;
    ```
  - `A:=[Monday, Wednesday] - halmazkonstruktor`
- üres halmaz: [ ]

# Halmaz

## Pascal

- Műveletek:
  - értékadás,
  - halmazműveletek:
    - eleme-teszt (in),
    - az =  , <>,  részhalmaz reláció ('<=', '>=')
    - (a  '<' és '>' nem megengedett!)
    - unió ('+'), differencia ('-').
    - metszet ('*')
- Ez a precedencia-sorrend is.

# Mikor ekvivalens két típus?

```
x, y: array[0..9] of integer;
z: array[0..9] of integer;
```

Strukturális ekvivalencia esetén:

- A rekordok mezőnevei is figyelembe vannak véve, vagy csak a struktúrájuk?
- Számít-e a rekordmezők sorrendje?
- Tömböknél elég-e az indexek számosságának egyenlőnek lenni, vagy az indexhatároknak is egyezniük kell?

# Mikor ekvivalens két típus?

Értékül adhatóak egymásnak? Akarjuk?

```
Dimensions = RECORD
          Breadth: REAL;
          Length: REAL;
    END;


Complex = RECORD
          RealPart: REAL;
          ImPart: REAL;
    END;
Size: Dimensions;
Root: Complex;
```

# Név szerinti ekvivalencia esetén

Deklarálhatók-e egy típushoz típusok, amelyekkel ekvivalens?

Névtelen tömb- illetve rekordtípusok ekvivalensek-e valamivel?

# Típuskonverziók

Van-e, és hogyan működik?

- az automatikus konverzió?
- az identitáskonverzió?
- a bővítő konverzió?
- a szűkítő konverzió?
- a toString konverzió?

# Típuskonverziók

Java

- identitáskonverzió
  - a `boolean` csak ez szabad
- bővítő konverzió
  - `byte` to `short`, `int`, `long`, `float` or `double`
  - `short` to `int`, `long`, `float` or `double`
  - `int` to `long`, `float` or `double`
  - `long` to `float` or `double`
  - `float` to `double`

# Típuskonverziók

## Java

- szűkítő konverzió
  - `byte` to `char`
  - `short` to `byte` or `char`
  - `char` to `byte` or `short` (miért is?)
  - `int` to `byte`, `short` or `char`
  - `long` to `byte`, `short`, `char` or `int`
  - `float` to `byte`, `short`, `char`, `int` or `long`
  - `double` to `byte`, `short`, `char`, `int`, `long` or `float`

# Változók

Változó =  (név, attribútumhalmaz, hely, érték)

Láthatóság, Elérhetőség

Hogyan definiálhatunk változókat és konstansokat?

# Változók

| | Szintaxis | Példa |
|---|---|---|
| Pascal | `var <identif>: <type>;` | `var i : integer;` |
| C++ | `<type> <identif>[= <value>];` | `int i = 0;` |
| Java | `<type> <identif>[= <value>];` | `int i = 0;` |
| ADA | `<identif>: <type>[:= <value>];` | `I : Integer := 0;` |
| CLU | `<identif>: <type>[:= <value>];` | `i : int := 0;` |
| Eiffel | `<identif>: [expanded]<type>;` | `I : INTEGER;` |
| Swift | `var <identif>: [<type>][ = <value>]` | `var x: Integer = 10` |

# Konstansok

| | Szintaxis | Példa |
|---|---|---|
| Pascal | `const <name> = <value>;` | `const val = 3;` |
| C++ | `#define <name> <value>`<br>`const  <type> <name> = <value>;` | `#define val 3`<br>`const int val = 3;` |
| Java | `final <type> <name> = <value>;` | `final int val = 3;` |
| ADA | `<name> :`<br>`constant <type>:= <value>;` | `Val :`<br>`constant Integer := 3;` |
| CLU | `const <type> <name> = <value>;` | `const double PI = 3.14159;` |
| Eiffel | `<name> : <type> is <value>;` | `A: INTEGER is 3;` |
| Swift | `let <identif>: [<type>] = <value>` | `let x: Integer = 10` |

# Kifejezések

Prefix jelölés

- + a b

Postfix jelölés

- a b +

Infix jelölés

- a + b

A műveletek precedencia szintjei fontosak

# Precendica - Pascal

| Leírás | Operátor |
|---|---|
| zárójelezés | |
| függvényhívás | |
| unáris operátor | not @ |
| Multiplikációs operátor | * / div mod and shl shr as << >> |
| Additív operátor | + - or xor |
| Relációs jel | = <> < > <= >= in is |
| Balról jobbra | |

# Precedencia – C++ 1.

| Precedence | Operator | Description |
|---|---|---|
| 1 | :: | Scope resolution |
| 2 | ++ | Suffix increment |
| | -- | Suffix decrement |
| | () | Function call |
| | [] | Array subscripting |
| | . | Element selection by reference |
| | -> | Element selection through pointer |
| | typeid() | |
| | const_cast | |
| | dynamic_cast | |
| | reinterpret_cast | |
| | static_cast | |

# Precedencia – C++ 2.

| Precedence | Operator | Description |
|---|---|---|
| 3 | ++ | Prefix increment |
| | -- | Prefix decrement |
| | + | Unary plus |
| | - | Unary minus |
| | ! | Logical NOT |
| | ~ | Bitwise NOT |
| | (type) | Type cast |
| | * | Indirection (dereference) |
| | & | Address-of |
| | sizeof | Size-of |
| | new, new[] | Dynamic memory allocation |
| | delete, delete[] | Dynamic memory deallocation |
| 4 | .* | Pointer to member |
| | ->* | Pointer to member |
| 5 | * | Multiplication |
| | / | Division |
| | % | Modulo (remainder) |
| 6 | + | Addition |

# Precedencia – C++ 3.

| Precedence | Operator | Description |
|---|---|---|
| 6 | + | Addition |
| | - | Subtraction |
| 7 | << | Bitwise left shift |
| | >> | Bitwise right shift |
| 8 | < | Less than |
| | <= | Less than or equal to |
| | > | Greater than |
| | >= | Greater than or equal to |
| 9 | == | Equal to |
| | != | Not equal to |
| 10 | & | Bitwise AND |
| 11 | ^ | Bitwise XOR (exclusive or) |
| 12 | \| | Bitwise OR (inclusive or) |
| 13 | && | Logical AND |
| 14 | \|\| | Logical OR |
| 15 | ?: | Ternary conditional (see ?:) |

# Precedencia – C++ 4.

| Precedence | Operator | Description |
|---|---|---|
| 6 | + | Addition |
| 16 | = | Direct assignment |
| | += | Assignment by sum |
| | -= | Assignment by difference |
| | *= | Assignment by product |
| | /= | Assignment by quotient |
| | %= | Assignment by remainder |
| | <<= | Assignment by bitwise left shift |
| | >>= | Assignment by bitwise right shift |
| | &= | Assignment by bitwise AND |
| | ^= | Assignment by bitwise XOR |
| | \|= | Assignment by bitwise OR |
| 17 | throw | Throw operator |
| 18 | , | Comma |

# Precedencia – Java

| Operators | Precedence |
|---|---|
| postfix | expr++ expr-- |
| unary | ++expr --expr +expr -expr ~ ! |
| multiplicative | * / % |
| additive | + - |
| shift | << >> >>> |
| relational | < > <= >= instanceof |
| equality | == != |
| bitwise AND | & |
| bitwise exclusive OR | ^ |
| bitwise inclusive OR | \| |
| logical AND | && |
| logical OR | \|\| |
| ternary | ? : |
| assignment | = += -= *= /= %= &= ^= \|= <<= >>= >>>= |

# Érdekesség

Algol 68

- A kétoperandusú operátorok prioritása általában felüldefiniálható a `prio` kulcsszóval:
  - `prio + = 3, * = 2;`
  - `print(6+3*5);  # (6+3)*5=45 lesz az eredmény`

# Vezérlési szerkezetek

# Egyszerű utasítások:

Értékadás – mit jelent?

- „változó” értékadásjel kifejezés

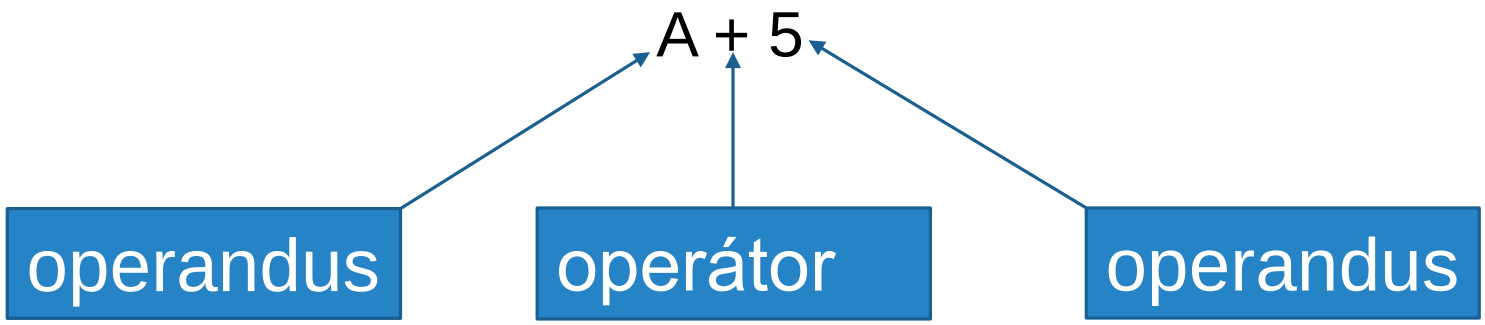
Helyette inkább:

- „balérték” értékadásjel „jobbérték”
  - (vagy fordítva?)
- Mit is jelent ez?
  - Értékadásjel
    - Szokásosan `:=` és `=`
    - De `->` és `<-`
  - Szemantikája
    - „megfelelő” típusú kifejezés kell!

# Kifejezés

operandusokból és operátorokból áll

- minden operandus lehet egy újabb kifejezés
- kiértékelem és megkapom az értékét:



# Kifejezés az értékadás?

Wulf (BLISS)

- „Of course, everything is”

Richie (C)

- „Yes, why not”

Wirth (Pascal)

- „No, only math-like things are expressions”

Két vonulat:

- Pascal, CLU, ADA, Eiffel, …
- C, C++, Java, …

Van-e többszörös értékadás?

# Szemlélet – COBOL

Eredetileg

- `MOVE 23 TO A.`
- `MOVE B TO C.`
- `ADD A TO B GIVING C.`
- `SUBTRACT A FROM B GIVING C.`
- `MULTIPLY A BY B GIVING C.`
- `DIVIDE A BY B GIVING C.`

Újabban

- `COMPUTE A = ( A + B - C / ( A * B ) - A * B )`

# Pascal

„balérték” := kifejezés;

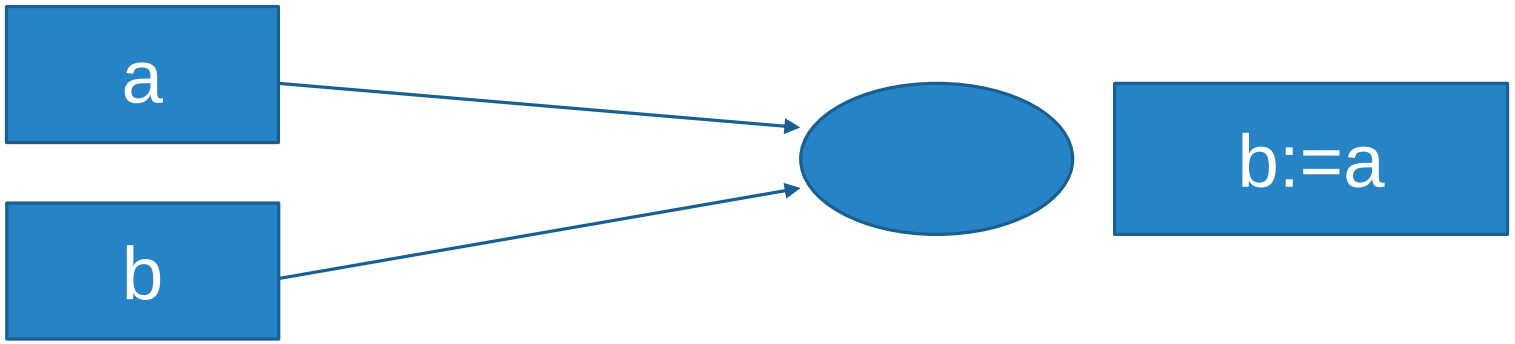
- A „balérték” és a kifejezés típusa megegyező, de legalább kompatibilis kell legyen.
  - `m: integer;`
  - `m:= m+1;`
- vagy:
  - `p := keres(gyoker, x);`

Többszörös értékadás nem megengedett.

# CLU

referenciákat használ

- értékadás hatására két referencia hivatkozhat ugyanarra az objektumra




- Többszörös értékadás megengedett:
  - `x, y := y, x.`

# C++

Számos értékadó operátor jobbról balra feldolgozva

- `A=B=C`
  - jelentése: `A=(B=C);`

Értékadó operátorok

- `= *= /= %= += -= >>= <<= &= ^=`

Például

- `y += g(x);`

# Java

A C++ -hoz hasonló, számos értékadó operátor:

- `= *= /= %= += -= >>= <<= >>>= &= ^=`
- `E1 op= E2`
  - jelentése: `E1 = (T)((E1) op (E2))`,
  - ahol `T` az `E1` típusa.

Összetett értékadó operátoroknál mindkét operandus primitív típusú kell legyen

- kivéve: `+=`, ha a bal operandus `String` típusú),

Implicit cast előfordulhat!

- `short x=3; x+=4.6;` eredménye: `x ==7`!

`final`-nak deklarált változónak nem adható érték.

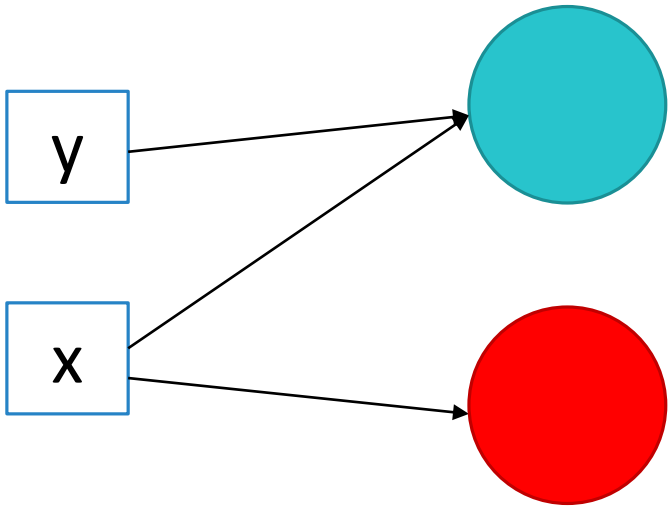
Alapvetően referenciákat használ

# Eiffel

Az értékadás és a paraméterátadás szemantikája megegyezik.

- Ha x:TX, y:TY, akkor az x:=y eredménye TX és TY-tól függ
- Mindkettő lehet referencia vagy kiterjesztett típus.
  - Az Eiffel mechanizmusa a klónozás, referencia másolás szemantikáját segít megérteni

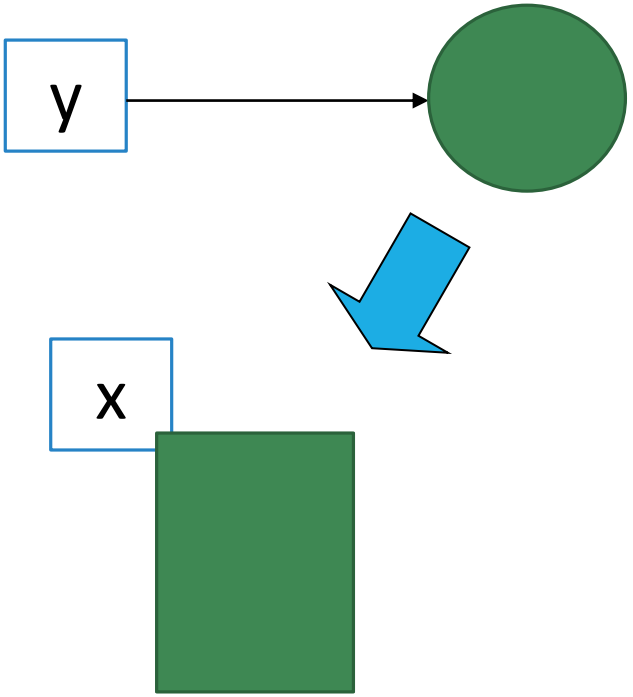
# Mindkettő referencia




*x:=y*

*referencia újrahozzárendelés*

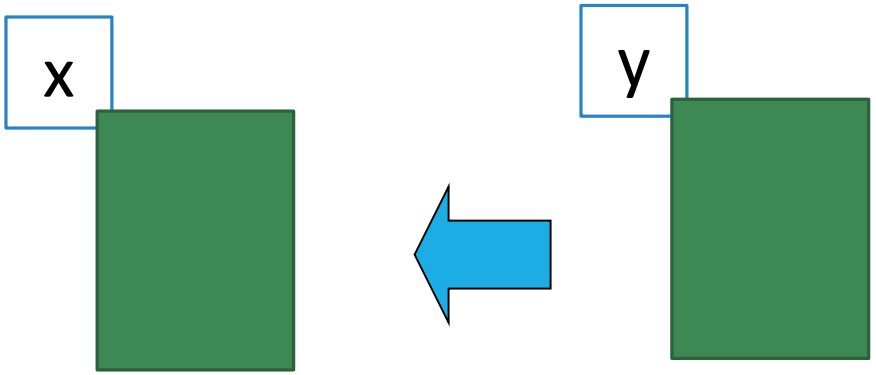
# TX kiterjesztett, TY referencia




*x:=y*

*másolás*

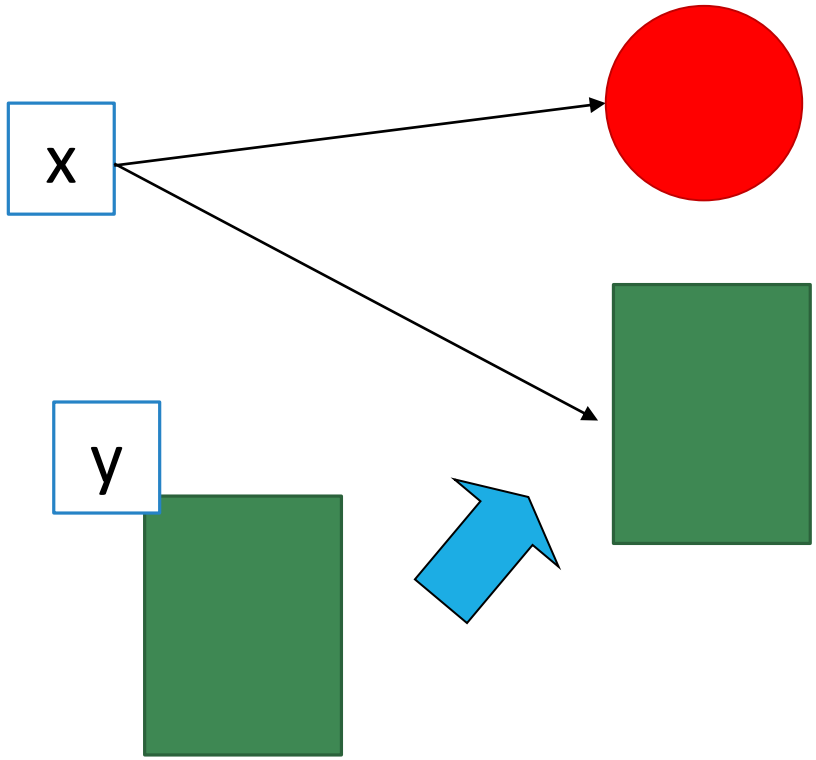
# Mindkettő kiterjesztett




*x:=y*

*másolás*

# TX referencia, TY kiterjesztett




*x:=y*

***klónozás***

# Üres utasítás

Pascal
- megengedett (`case`)

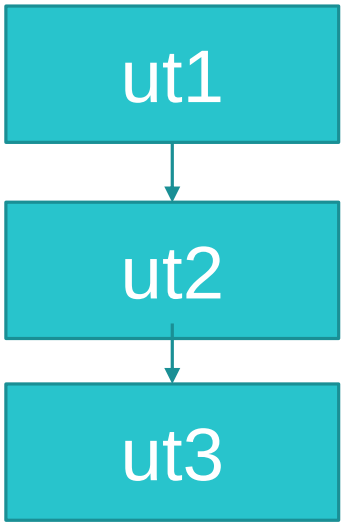
ADA
- null; (`case`)

C++, Java, stb.: „ ; ” használható
- Például ciklusnak lehet üres törzse

Eiffel: „ ; ” használható
- nincs valódi szerepe

# Összetett utasítások

Szekvencia:







ut1; ut2; ut3;

# Szekvencia

Lehet-e üres?

Terminátor vagy utasításelválasztó-e a „ ; ”?

Lehet-e blokkutasítást létrehozni?

Elhelyezhető-e a blokkutasításban deklaráció?

# Szekvencia

Pascal, Delphi:  a „ ; ” elválasztja az utasításokat:

- `begin ut1; ut2; ut3 end`

- üres utasítás is lehet:
  - `begin ut1; ut2; ut3; end`
- Az üres utasítás lehetősége miatt a „ ; ” beírása megváltoztathatja a program jelentését!

ADA: a „ ; ” lezárja az utasítást

C++, Java, Objective-C: a „ ; ” lezárja az utasítást

- de nem mindet, például a blokk utasítást nem

- Illetve , elválaszt kifejezéseket C++ esetén

# Szekvencia

Python: az utasításokat az új sor jelek zárják le

- De a ; megengedett elválasztóként soron belül
  - `x = 5; print(x)`

Swift: a ; elválasztja az utasításokat

- A fordítóprogram elhelyezi

# Blokk utasítások

Utasítások sorozatából alkothatunk egy összetett utasítást, blokkot.

Bizonyos nyelvekben lehet deklarációs része is.

Főleg azokban a nyelvekben fontos, ahol a feltételes és a ciklus utasítások csak egy utasítást tartalmazhatnak

- Pascal, C++,  Java, …

Pascal: `begin utasítássorozat end`

ADA:

- `[declare deklarációk] begin utasítássorozat [kivételkezelő rész] end;`
- Például

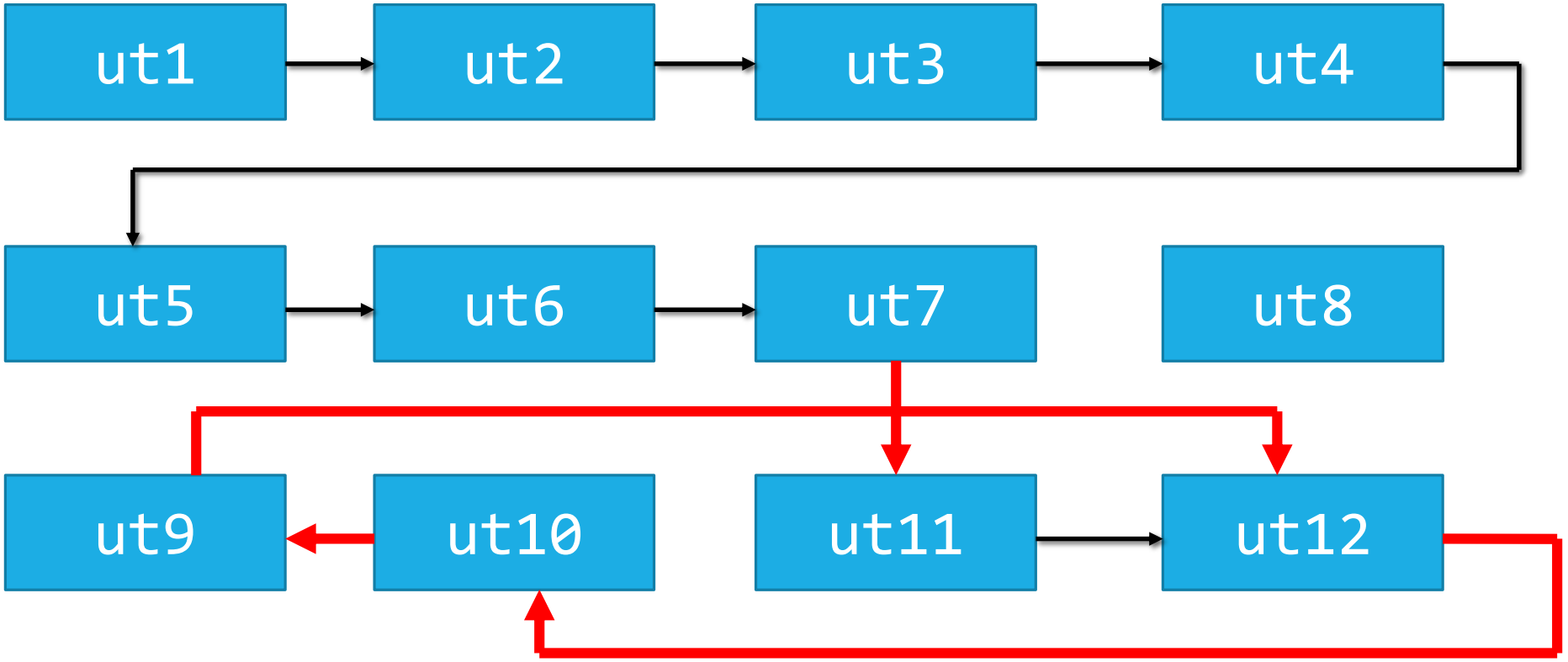
```
Csere:
  declare
  Temp : Integer;
    begin
      Temp := I; I := J; J := Temp;
    end Csere;
```

C++, Java, …

- “{“ és “}” között.

# Feltétel nélküli vezérlésátadás

goto



# Feltétel nélküli vezérlésátadás

javaslat: `goto` nélkül



# Feltétel nélküli vezérlésátadás

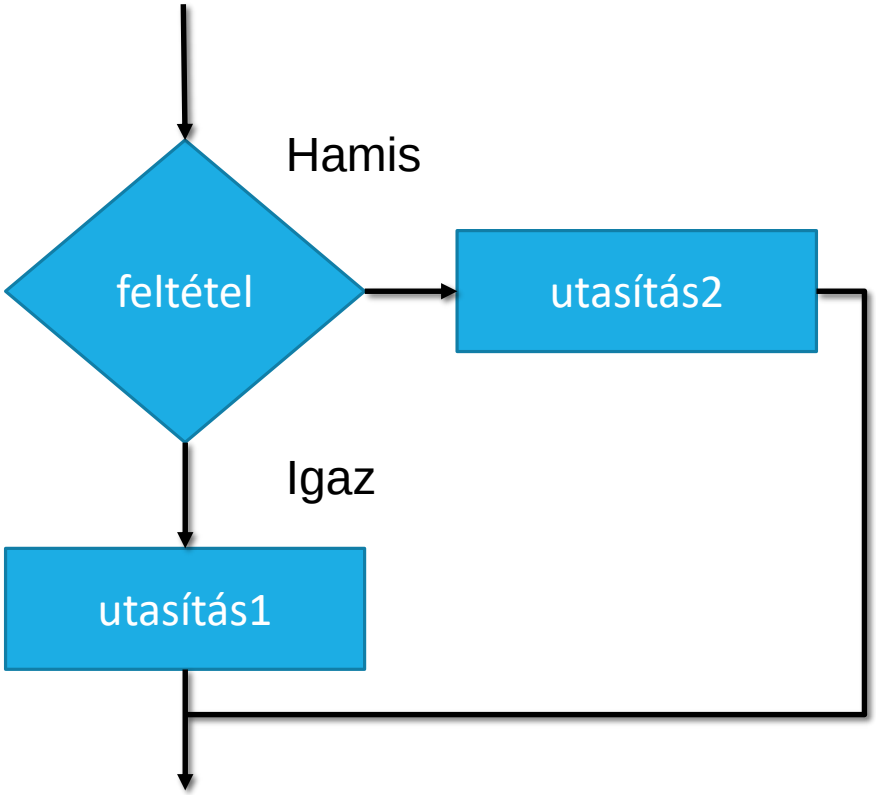
Ha mégis – Pascal, C stb.

```
címke:
   .... utasítások ...
   goto címke;
```

Nincs: Modula-3, Java, …

# Elágazások:

- Feltétel igaz vagy hamis értékétől függően valamilyen utasítás(csoport) végrehajtása






# Elágazások

```
if feltétel then ut1
```

```
if feltétel then ut1 else ut2
```

„csellengő” else kérdése

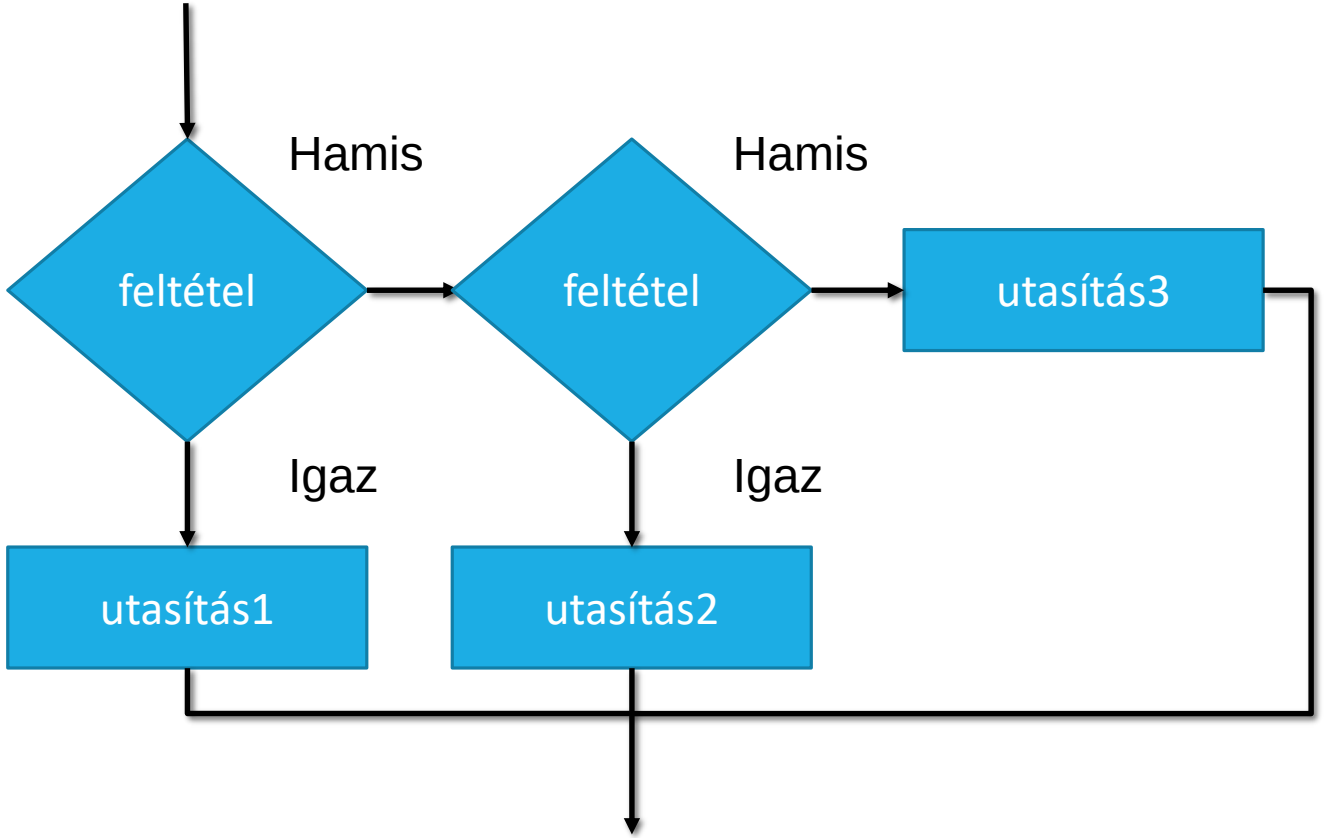
- `if felt1 then if felt2 then ut1 else ut2`
- Mit jelent?
  - `if felt1 then (if felt2 then ut1 else ut2)`
- Vagy?
  - `if felt1 then (if felt2 then ut1) else ut2`

# Elágazások – csellengő else

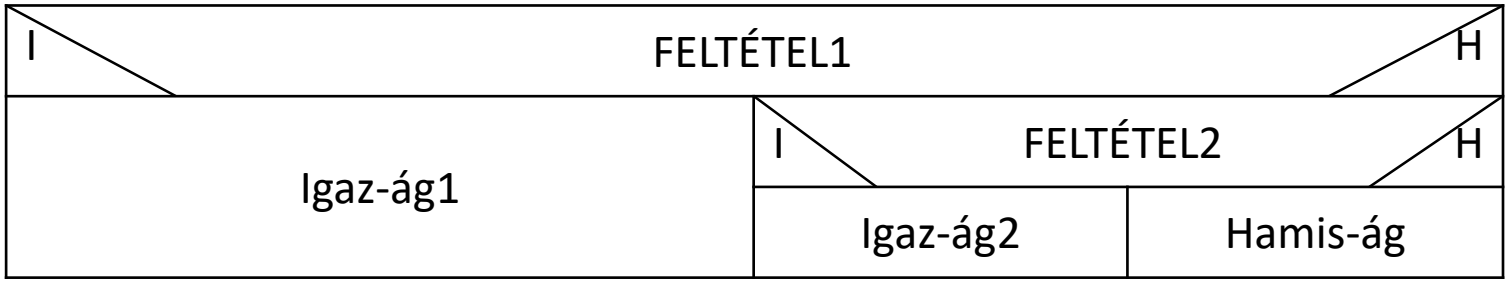
Különböző megoldások:

- blokk utasítás kell a beágyazásnál
- az `else` rész a legbelső `if`-hez tartozik
- explicit `end` konstrukció bevezetése

# Többirányú elágazások



# Többirányú elágazások



# Többirányú elágazások

`elsif` lehetősége – óvatos módosítás kell!

```
if felt1 then s1;
elsif felt2 then s2;
elsif felt3 then s3;
else s4;
end if;
```

# Pascal

```
if felt then S1 [else S2]
```

- több utasítás: blokk kell

„csellengő” else:

- legbelső if :

```
if felt1 then
  if felt2 then S1
           else S2
```

különben blokk:

```
if felt1 then
  begin
    if felt2 then S1
  end
else S2
```

# Pascal

```
if (a>b)
  then writeln("a>b")
else
  if (a<b) then writeln("a<b")
           else writeln("a=b");
```

# Pascal

Vigyázat!

- `if felt1 then ut1 else ut2`
  nem ugyanaz, mint:
- `if felt1 then ut1; else ut2`
  ez ekvivalens:
- `if felt1 then ut1; else ut2`
- ha
  - `if felt1 then ut1 else ut2` -ben
    S1 után beszúrunk egy „ ; ”-t, az hiba!

# ADA

end if a végén, elsif megengedett:

```
if <expr>1 then
   <statm>1;
{elsif (<expr>2)then
   <statm>2;}
[else
   <statm>3; ]
end if;
```

```
if A>B then
   Put_Line("a>b");
elsif A<B then
   Put_Line("a<b");
else
   Put_Line("a=b");
end if;
```

# C++

aritmetikai kifejezés kell, nem nulla: true.

„csellengő” else: legbelső if

- blokk kell különben

```
if (<expr>)
     <statm1>
[else
     <statm2>]
```

```
if (a>b) cout << "a>b";
else
  if (a<b) cout << "a<b";
  else cout << "a=b";
```

aritmetikai if kifejezés pl.:

```
if (a<=b) max=b;
else max=a;
```

inkább:

```
max=(a<=b)?b:a;
```

# Java

Hasonló a C++-hoz

- kivévea kifejezés típusa boolean kell legyen

```
if (<expr>)
   <statm1>
[else
   <statm2>]
```

```
if (a>b)
  System.out.println("a>b");
else
  if (a<b)
    System.out.println("a<b");
  else
    System.out.println("a=b");
```

# Eiffel

end zárja le az utasítást.

Megengedett az elseif használata.

```
if <expr>1 then
   <statm>1
{elseif <expr>2 then
   <statm>2 }
[else
   <statm>3 ]
end
```

```
if (a>b) then
   io.putstring("a>b");
elseif (a < b) then
   io.putstring("a < b")
else
   io.putstring("a = b");
end
```

# Python

Csellengő else: a bekezdés számít!!

```
a, b = 1, 2
if a == 1:
    if b == 1:
        print ("a és b is 1")
else:
    print ("a nem 1")
```

# Python

Csellengő else: a bekezdés számít!!

Így mást jelent:

```
a, b = 1, 2
if a == 1:
    if b == 1:
        print ("a és b is 1")
    else:
        print ("a nem 1")
```

# if és értékadás

Ruby:

- `num += -1 if num < 0`

Scala:

- `val x = if (a > b) a else b`

Jobban olvasható, mint a `? :`

# if és az értékadás

Kotlin

- További utasítások kerülhetnek bele

```
val max = if (a > b) {
    print("Choose a")
    a
} else {
    print("Choose b")
    b
}
```

# Esetkiválasztásos elágazások

Valamilyen kifejezés (szelektor) értékétől függően, általánosan:

| kifejezés | | | | | |
|---|---|---|---|---|---|
| érték1 | érték2 | érték3 | értékek | intervallum | egyéb |
| utasítások | utasítások | utasítások | utasítások | utasítások | utasítások |

# Esetkiválasztásos elágazás

Kérdések

- Mi lehet a szelektor típusa?
- Fel kell-e sorolni a szelektortípus minden lehetséges értékét?
- Mi történik, ha fel nem sorolt értéket vesz fel a szelektor?
- „Rácsorog”-e a vezérlés a következő kiválasztási ágakra is?
- Diszjunktnak kell-e lennie a kiválasztási értékeknek?
- Mi állhat a kiválasztási feltételben?
  - egy érték
  - értékek felsorolása
  - intervallum

# „Case” utasítások

```
case expr of
  const1: st1;
  const2: st2;
 …
  constn: stn
end
```

# „Case” utasítások

Sokszor igaz a case konstansokra:

- Tetszőleges sorrend,
- Nem feltétlenül egymás után,
- Több is vonatkozhat ugyanarra az alutasításra,
- Mind különböző kell legyen - különben, ha $const_i$ és $const_j$ egyenlőek, akkor melyiket választanánk, $st_i$ -t vagy $st_j$-t?
- A kifejezés típusa diszkrét kell legyen (egész vagy felsorolási típ)
- Kell adni egy „others” ágat, hogy minden lehetőséget lefedjünk.

# Pascal

A szelektor típusa lehet: integer, character, boolean vagy tetszőleges felsorolási vagy intervallum típus.

„else” megengedett, de nem kötelező (skip).

```
case var of
  val1: statm1;
  val2.: statm2;
...
  vali…valj : statmi;
[else statm]
end;
```

```
var Age: Byte;
case Age of
  0..13: Write('Child');
 14..23: Write('Young');
 24..65: Write('Adult');
 else    Write('Old');
end
```

# ADA

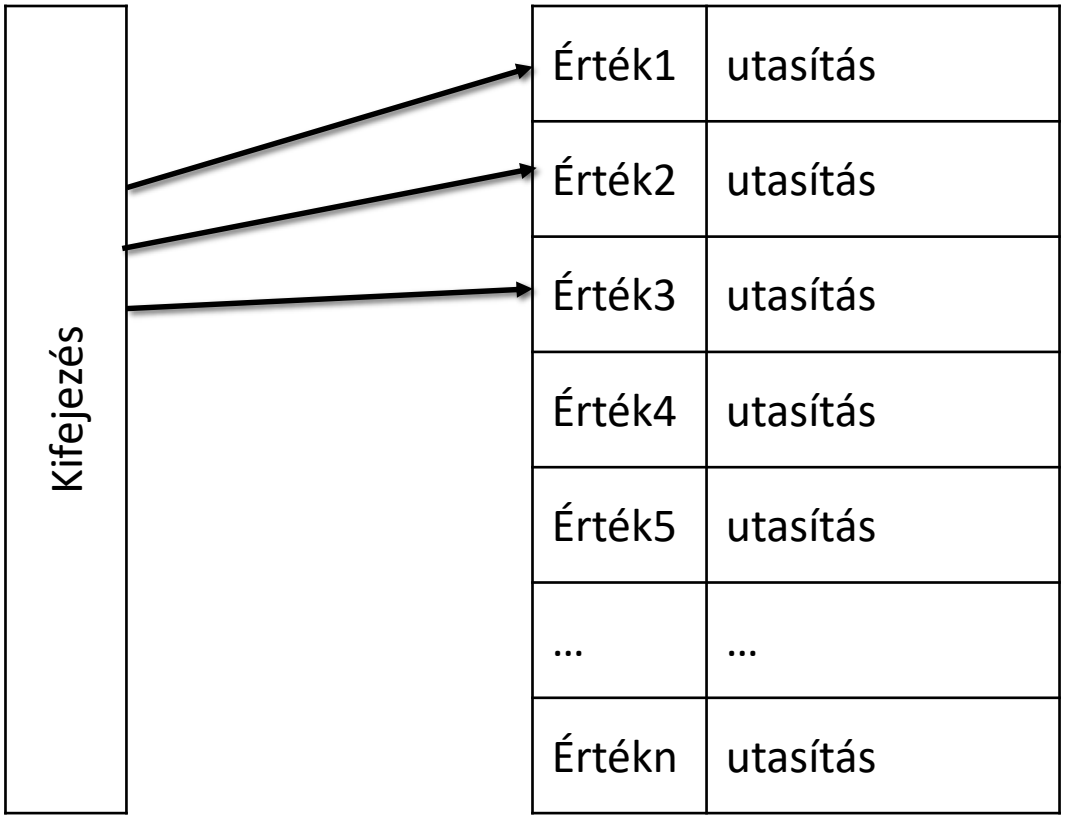
```
case expr is
 when choice1 => statms1;
 when choice2 => statms2; ...
 when others =>  statms;
end case;
```

others kötelező, ha nincs minden lefedve!

.. - intervallum,
| - vagy

```
case Today is
  when Monday => Initial_Balance;
  when Friday => Closing_Balance;
  when Tuesday..Thursday => Report(Today);
  when others => null;
end case;
```

# C++


Kifejezés
Érték1
utasítás
Érték2
utasítás
Érték3
utasítás
Érték4
utasítás
Érték5
utasítás
…
…
Értékn
utasítás

break

# C++

```
switch (intexpr) {
  case label1: statm1; break;
  case label2: statm2; break;
  ...
  default: statm;
}
```

kifejezés „integral” típusú

- „default:” címke lehetősége (nem kötelező).
- itt break kell, ha nem akarom folytatni!

# C++

```
switch (val){
   case 1 : cout<<"case 1\n";
   case 2 : cout<<"case 2\n";
   default: cout<<"default case";
}
```

ha val=1, az eredmény:

```
case 1
case 2
default case
```

# Java

Mint a C++:

```
switch (val){
 case 1 : System.out.println("case 1\n");
 case 2 : System.out.println("case 2\n"); break;
 default: System.out.println("default case");
}
```

ha val=1, az eredmény:

```
case 1
case 2
```

# Java

Értékadás

```
final String exampleString =
 switch (val){
  case 1 : "case 1"; break;
  case 2 : "case 2"; break;
  default: "default"; break;
 }
```

ha val=1, az eredmény:

```
 case 1
```

# Java

Értékadáshoz új szintaxis

```
final String exampleString =
 switch (val){
  case 1 -> "case 1";
  case 2 -> "case 1";
  default -> "default";
 }
```

Fontos változás, hogy itt nincsen átcsorgás!

ha val=1, az eredmény:

```
case 1
```

# Kotlin

Hasonlóan a Java új szintaxishoz, de általánosan:

```
when (x) {
     1 -> print("x == 1")
     2 -> print("x == 2")
     else -> {
          print("x is neither 1 nor 2")
     }
}
```

# Kotlin

Kifejezések és intervallumok használata

```
when (x) {
    parseInt(s) -> print("s encodes x")
    else -> print("s does not encode x")
}

when (x) {
    in 1..10 -> print("x is in the range")
    in validNumbers -> print("x is valid")
    !in 10..20 -> print("x is not in the range")
    else -> print("none of the above")
}
```

# Eiffel

Pascal-szerű szemantika

A változó típusa INTEGER vagy CHARACTER

```
inspect var
  when expr1 then statmblock1
  when expr2 then statmblock2
  ...
  else statmblock
end
```

Exception lép fel, ha nem gondoskodtunk a megfelelő választék-elemről (null utasítás szükséges lehet).

# Eiffel

```
inspect c
  when '0'..'9'                then n:= n + 1;
  when 'a'..'z','A'..'Z' then s:= s + 1;
  else  o:= o + 1;
end
```

# C#

```
switch (kif)
{
    case ertek1 : utasítások… break;
    case ertek2 : utasítások… break;
    case ertek3 : utasítások… break; vagy:
                  goto case KONST/default;
…
}
```

szemantika

Ciklusként is működhet!!

# SWIFT

Lehetséges az értékre rész-megszorítást adni

```
switch vegetable {
  case "celery":
    let vegetableComment = …
  case "cucumber", "watercress":
    let vegetableComment = …"
  case let x where x.hasSuffix("pepper"):
    let vegetableComment = …
  default:
    let vegetableComment = …
}
```

# SWIFT

Tuple használata esetén, lehetséges figyelmen kívül hagyni az értékek egy részét:

```
let somePoint = (1, 1)
switch somePoint {
case (0, 0):
    print("\(somePoint) is at the origin")
case (_, 0):
    print("\(somePoint) is on the x-axis")
case (0, _):
    print("\(somePoint) is on the y-axis")
case (-2...2, -2...2):
    print("\(somePoint) is inside the box")
default:
    print("\(somePoint) is outside of the box")
}
```

# Absztrakció, alprogramok

KÖVETKEZŐ HÉTEN




# Programozási nyelvek és módszerek

## TÍPUSABSZTRAKCIÓ, ALPROGRAMOK

# Vezérlési szerkezetek

# Ciklusok

utasítások ismételt végrehajtása

- valamilyen feltételtől függően



# Ciklusok általános kérdései

Vannak-e nem ismert lépésszámú ciklusok?

- Van-e elöl/hátul tesztelős ciklus?
- A ciklusfeltétel  logikai érték kell legyen, vagy más típusú is lehet?

Kell-e blokkot kijelölni a ciklusmagnak?

# Ciklusok általános kérdései

Van-e előre ismert lépésszámú ciklus?

- A ciklusváltozó mely jellemzője állítható be a következők közül?
  - alsó érték - felső érték - lépésszám
- Mi lehet a ciklusváltozó típusa?
- Biztosított-e a ciklusmagon belül a ciklusváltozó változtathatatlansága?
- Mi a ciklusváltozó hatásköre, definiált-e az értéke kilépéskor?

# Ciklusok általános kérdései

Van-e általános (végtelen) ciklus?

Léteznek-e a következő vezérlésátadó utasítások?

- `break` - `continue`

Van-e ciklusváltozó-iterátor?

# Nem ismert lépésszámú ciklusok

Elöltesztelő „while” ciklusok:

- `while <kif> do <utasítás>`

Pascal: a ciklusmag egyetlen utasítás lehet (de blokk is)

- `while a < b do b := b - a;`

# ADA

end loop zárja, így a ciklusmag utasításssorozat is lehet:

```
while <kif> loop
   <ciklusmag>
end loop
```

kifejezés Boolean típusú

# C++

A kifejezés aritmetikai értéket ad

- nem 0: true, 0: false

A ciklusmag  egyetlen utasítás lehet (de blokk is)

```
while (<expr> ) <statm>
while (a < b)
   b = b - a;
```

# Java

Hasonló a C++-hoz, kivéve: a kifejezés típusa boolean kell legyen.

```
i=0;
while (i<10) {
  System.out.println(i);
  i++;
}
```

# Eiffel

Amíg a ciklusfeltétel hamis!

- end zárja, így a ciklusmag utasításssorozat is lehet.

```
from
   init
[invariant
   loop-inv
   variant
   variant-func]
until
   loop-cond
loop
   loop-body
end
```



# Hátultesztelő „repeat-until” ciklusok

Amíg egy feltétel igaz nem lesz.

Ciklusmag egyszer biztosan lefut

```
repeat
   utasítássorozat
until ciklusfelt
```

nincs szükség blokk utasításra – a ciklusmag vége meghatározott

```
while E do S
```

- Jelentése

```
if E then
   repeat S until not E
```

# Hátultesztelő ciklusok

Pascal:

```
repeat
   b := b - a;
until (b <= a);
```

SWIFT:

```
var m = 2
repeat {
    m = m * 2
} while m < 10
```

# Hátultesztelő ciklusok

C++: a kifejezés aritmetikai érték, nem 0: igaz, amíg értéke 0 (hamis) nem lesz

```
do <statm> while (<expr>);
```

Java: hasonló a C++-hoz, kivéve: a kifejezés típusa boolean kell legyen

Eiffel:

- „until_do” az Iteration könyvtárból.

# Előre ismert lépésszámú ciklusok „For” ciklusok

index változó kezelése – lépésköz - határ

Egyszer, a ciklusba való belépés előtt értékeli ki a lépésközt és a határt, vagy minden végrehajtás után újra?

A határt a ciklusmag végrehajtása előtt vagy után ellenőrzi?

Mi lehet a ciklusváltozó típusa?

Biztosított-e a ciklusmagon belül a ciklusváltozó változtathatatlansága?

Mi a ciklusváltozó hatásköre, definiált-e az értéke kilépéskor?

# Pascal

Jellemzők

- lépésköz és határ kiértékelése egyszer
  - határ ellenőrzése ciklusmag előtt
- ciklusváltozó nem változtatható (ma már)
  - értéke kilépéskor nem definiált
- ciklusváltozó diszkrét típusú lehet

```
for i := 1 to n do sum := sum + i;
for c := 'z' downto 'a' do write(c);
```

# Kotlin

For ciklus

- előre ismert lépésszám

```
for (i in 6 downTo 0 step 2) {
    println(i)
}
```

# Ada

Az index a ciklus lokális konstansa.

- Diszkrét típusú kell legyen.
- A ciklus értékintervallumát csak egyszer, a ciklusba való belépés előtt számítja ki.

```
for <var> in loop-range loop
 <utasítások>;
end loop;
Sum:=0;
for I in 1..N loop
   Sum := Sum + I;
end loop;
```

A „reverse” hatására : fordított sorrend.

# SWIFT I.

Hasonlóan az előzőekhez

- Az ciklusváltozó felveszi a meghatározott intervallum értékét

```
var firstForLoop = 0
for i in 0..<4 {
    firstForLoop += i
}
```

A következő operátort `..<` használatával az `i` `0` és `3` közötti értékeket vesz fel.

- `i in 0...4`

Ha a ciklusváltozóra nincs szükség

```
for _ in 1...max {
  …
}
```

# SWIFT II.

A C++-hoz hasonlóan:

```
var secondForLoop = 0
for var i = 0; i < 4; ++i {
    secondForLoop += i
}
```

For-each

```
for number in numbers {
    if number > largest {
        largest = number
    }
}
```

# C++

a `for (for-init-stm [expr-1]; [expr-2]) stm` jelentése

```
{ for-init-stm
    while (expr-1) {
      stm
      expr-2;
    }
}
```

Kivéve, hogy egy `continue` a `stm`-ben `expr-2`-t hajt végre az `expr-1` kiértékelése előtt.

Hiányzó `expr-1` ekvivalens: `while(1)`

Ha a `for-init-stm` egy deklaráció, akkor a deklarált nevek hatásköre a `for` utasítást tartalmazó blokk

```
sum=0;
for (int i = 1; i < n; i++)
  sum = sum + i;
```

# Java

Eredeti for ciklus: hasonló a  C++-hoz:

```
public class ForDemo {
    public static void main(String[] args) {
        int[] arrayOfInts = { 32, 87, 199, 622, 127, 485 };
        for (int i = 0; i < arrayOfInts.length; i++) {
            System.out.print(arrayOfInts[i] + " "); i++;
        }
        System.out.println();
    }
}
```

Mit csinál?

# Java

for-each ciklus:

```
public class ForEachDemo {
  public static void main(String[] args) {
      int[] arrayOfInts = {32, 87, ...., 622, 127};
      for (int i : arrayOfInts) {

              System.out.print(i + " ");
     }
      System.out.println();
  }
}
```

# Java

Gyűjteményekkel (Collection) és tömbökkel használható

Az iterációk leggyakoribb formájára, amikor az index, illetve az iterátor értéket semmilyen más műveletre nem használják, csak az elemek elérésére.

Még egy példa:

- …. ha van egy Number típusunk:

```
List<Number> szamok = new ArrayList<Number>();
szamok.add(new Integer(42));
szamok.add(new Integer(-30));
szamok.add(new BigDecimal("654.2"));
for ( Number number : szamok ){
  …
}
```

# C#

„hagyományos” for ciklus, hasonlóan a C++-hoz:

```
using System;
public class ForLoopTest
{
   public static void Main()
   {
      for (int i = 1; i <= 5; i++)
         Console.WriteLine(i);
   }
}
```

# C#

foreach utasítás:

```
foreach (type identifier in expression) statement
```

- a ciklusváltozó értékét ne változtassuk, ha ez érték típusú, akkor nem is lehet.
- A kifejezés gyűjtemény vagy tömb lehet, `IEnumerable`-t implementál, vagy egy olyan típust, ami deklarál egy `GetEnumerator` metódust.

# C#

foreach tömbökre:

```
public static void Main() {
        int odd = 0, even = 0;
        int[] arr = new int [] {0,1,2,5,7,8,11};
        foreach (int i in arr) {
            if (i%2 == 0)
                even++;
            else
                odd++;
        }
        …
    }
}
```

# C#

## foreach gyűjteményekre

- a myCollection gyűjteményre a következőknek kell teljesülnie:
  - interface, class, vagy struct típus kell legyen
  - kell legyen egy GetEnumerator nevű példánymetódusa, aminek a visszaadott típusára (pl. Enumerator ) fennáll:
    - Van egy Current nevű property-je, ami ItemType-ot, vagy erre konvertálhatót ad vissza – a gyűjtemény aktuális elemét
    - Van egy MoveNext, bool-t visszaadó metódusa, ami növeli az elemszámlálót, és true-t ad, ha van még elem a gyűjteményben
- Létrehozunk egy gyűjteményt a fenti szabályokkal – ez csak C# programokban használható persze
- Létrehozunk egy gyűjteményt a fenti szabályokkal, és implementáljuk az IEnumerable interfészt
  - ez más nyelvekben is használható lesz, mint pl. a Visual Basic
- Használjuk az előredefiniált gyűjtemény osztályokat

# „Végtelen” ciklusok

ADA

```
loop
   <statm>1; …
      exit when <feltétel>;
   <statm>2; …
end loop;

loop
   get(Current_Char);
   exit when Current_Char =’*’;
end loop;
```

# Vezérlésátadó utasítások

`break`: kiugrás a vezérlési szerkezetből

`continue`: ciklusfeltétel újrakiértékelése

alprogram hívás

`return` alprogramból hívóhoz visszatérés.

`goto` – !!! Veszélyeket rejt

# Példák

C++:

- `break` – ciklusban, vagy switch utasításban
- `continue` – csak ciklusban
- `return`
- `goto`

Java:

- `break`, continue, `return` mint a C++-ban
- NINCS goto!

# Példák

Kotlin

- `break` – ciklusban
- `continue` – ciklusban
- `return` – kilép a legközelebbi függvényből, anonim függvényből
- Címkézés – megadható, hogy hova adódjon át a vezérlés

```
loop@ for (i in 1..100) {
    for (j in 1..100) {
        if (...) break@loop
    }
}
```

# Példák

SWIFT

- Hasonlóan a C++-hoz:
  - `continue`
  - `break` – ciklusban szokásos, switch esetén kilép!
  - `return`
- Switch esetén továbbcsorgat: `fallthrough`

```
let integerToDescribe = 5
var description = "The number \(integerToDescribe) is"
switch integerToDescribe {
case 2, 3, 5, 7, 11, 13, 17, 19:
    description += " a prime number, and also"
    fallthrough
default:
    description += " an integer."
}
print(description)
```

# Python

`for`/`while` ciklus esetén:

- `continue`
- `break`
- **`else`**

Példa

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            break
    else:
        print(n, ' prím’)
```

A `for` ciklushoz tartozik az `else`!

Az `else` ág akkor fut le, ha az összes ciklusiteráció lefut.

- Nincs `break` hívás

# Absztrakció

# Absztrakció – Mi az?

## Gondolkodásmód

- Az általános elvekre koncentrálunk, nem ezeknek a specifikus megjelenési formáira.
  - Filozófia,
  - Matematika,
  - Számítástudomány

## Absztrakció a rendszer analízisben:

- A feladat lényeges aspektusai
  - Eltekintünk a kevésbé lényegestől
- Például: légi közlekedés vezérlése
  - lényeges: helyzete, sebessége stb.
  - lényegtelen: színe, utasok neve…
    - (ez a feladattól függ!)

# Alprogramok mint absztrakciók

Absztrakció a programozásban

- Különböztessük meg a szinteket
  - Mit csinál a program?
  - Hogyan van implementálva?

Minden programozási nyelv a gépi kód absztrakciója.

Magasabb szintű absztrakciók

- Eljárás: Mit csinál az eljárás?
  - Hogyan csak akkor  fontos, amikor implementáljuk.
- Eljárást hívó eljárások:
  - Az absztrakció akárhány szükséges szintje bevezethető.

# Alprogramok mint absztrakciók

## Absztrakciós mechanizmus

- A programozási nyelvi konstrukció, ami megengedi, hogy a programozó megragadja az absztrakciót
  - És a program részeként reprezentálja
    - egyfajta számítási mintaként
- Egyszerű példa
  - Eljárások és függvények
- Egyéb példák
  - Később...

# A legegyszerűbb absztrakciós mechanizmus

## Eljárások és függvények

- Egységek, melyek számításokat tartalmaznak
  - Függvény-absztrakció: egy kiszámítandó kifejezést tartalmaz
  - Eljárás-absztrakció: egy végrehajtandó parancsot tartalmaz.
- A tartalmazott számítás mindig végrehajtásra kerül, amikor egy absztrakciót hívunk.

## Az érdekeltségek szétválasztása:

- A Hívó azzal törődik, mit csinál a számítás.
- Az Implementáló azzal is törődik, hogy hogyan kell a számítást végrehajtani természetesen a „mit” szerint

## A hatékonyságot a paraméterezéssel javítjuk.

# Függvény-absztrakció

Egy kiértékelendő kifejezés – amikor hívjuk, egy értéket ad vissza

Ada példa

```
function Kerulet (R : Float) return Float is
begin
    return 2.0*R*3.14;
end;
```

# ML példa

```
fun power( x: real; n: int) =
    (* felt. hogy  n > 0 *)
    if n = 1 then
        x
    else
        x * power (x, n- 1)
    end
```

# Mi a függvény definíció?

```
function I (FP1; ... ; FPn) return T is
body
```

Egy `I` azonosítót hozzákapcsol egy adott függvény-absztrakcióhoz.

A függvény-absztrakció a megfelelő aktuális paraméterekkel való híváskor eredményt ad vissza.

- A függvényhívás felhasználói szemlélettel egy leképezés az argumentumok és az eredmény között.
- Megvalósítói szemlélet: a függvénytörzs kiértékelése.
  - Az algoritmus változása csak a megvalósítóra tartozik.

# Eljárás-absztrakció

Egy végrehajtandó parancsot testesít meg.

- A felhasználó csak a változók megváltozását érzékeli.

Általános formátum:

```
procedure I ( FP 1 ; ... ; FP n )
body
```

Felhasználói szemlélet:

```
type Dictionary = array [...] of Word;
procedure sort( var words: Dictionary);

sort(a); (* érzékeljük 'a' változását *)
```

Megvalósítói szemlélet: a kódolt algoritmus.

# Absztrakciós elv

Összefoglalás:

- Függvény-absztrakció: egy kifejezés absztrakciója.
  - Egy  függvényhívás egy kifejezés, amikor hívjuk, kiértékeli, és egy értéket ad vissza.
- Eljárás-absztrakció: egy parancs absztrakciója.
  - Egy eljáráshívás egy parancs, ami az eljárástörzs végrehajtása során frissíti/frissítheti a változók értékét.

# Absztrakciós elv

## Általánosítás

- Tetszőleges szintaktikai osztály fölött létrehozhatunk absztrakciókat, feltéve, hogy ez valamifajta számítást specifikál.
  - Absztrakt adattípusok
  - Sablonok
    - Generic, később …

# Alprogramok és paramétereik

Alprogramok használata

- az egyik legelső programozási eszköz.

Charles Babbage – Analytical Engine

- 1840-ben már tervezte, hogy lyukkártyák egy csoportját fogja használni nagyobb számítások gyakrabban ismételt részeinél

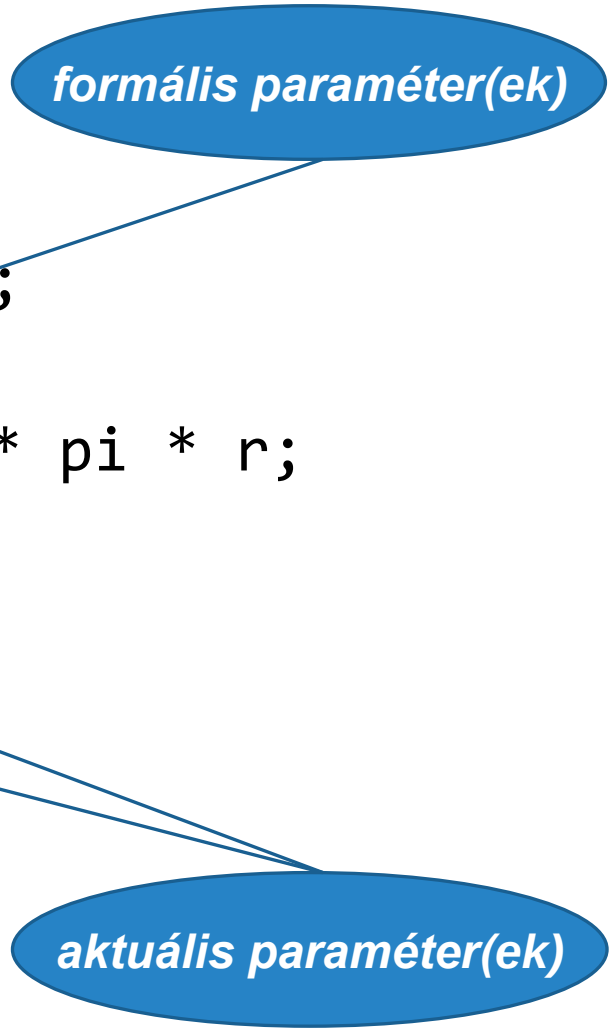
Alprogramok használatával nevet adhatunk egy kódrészletnek és paraméterezhetjük a viselkedését.

# Paraméter átadása

Absztrakció általánosítása

```
val pi = 3.14159;
val r = 1.0;
fun circumference() = 2 * pi * r;

fun circumference (r: real) = 2 * pi * r;
circumference (1.0);
circumference(a + b);
```



# Alprogram

Programegység

Végrehajtás kezdeményezése: meghívással

A program darabolásának eszköze

Különböző számítások elkülönítése egymástól

Újrafelhasználhatóság

# Mire való

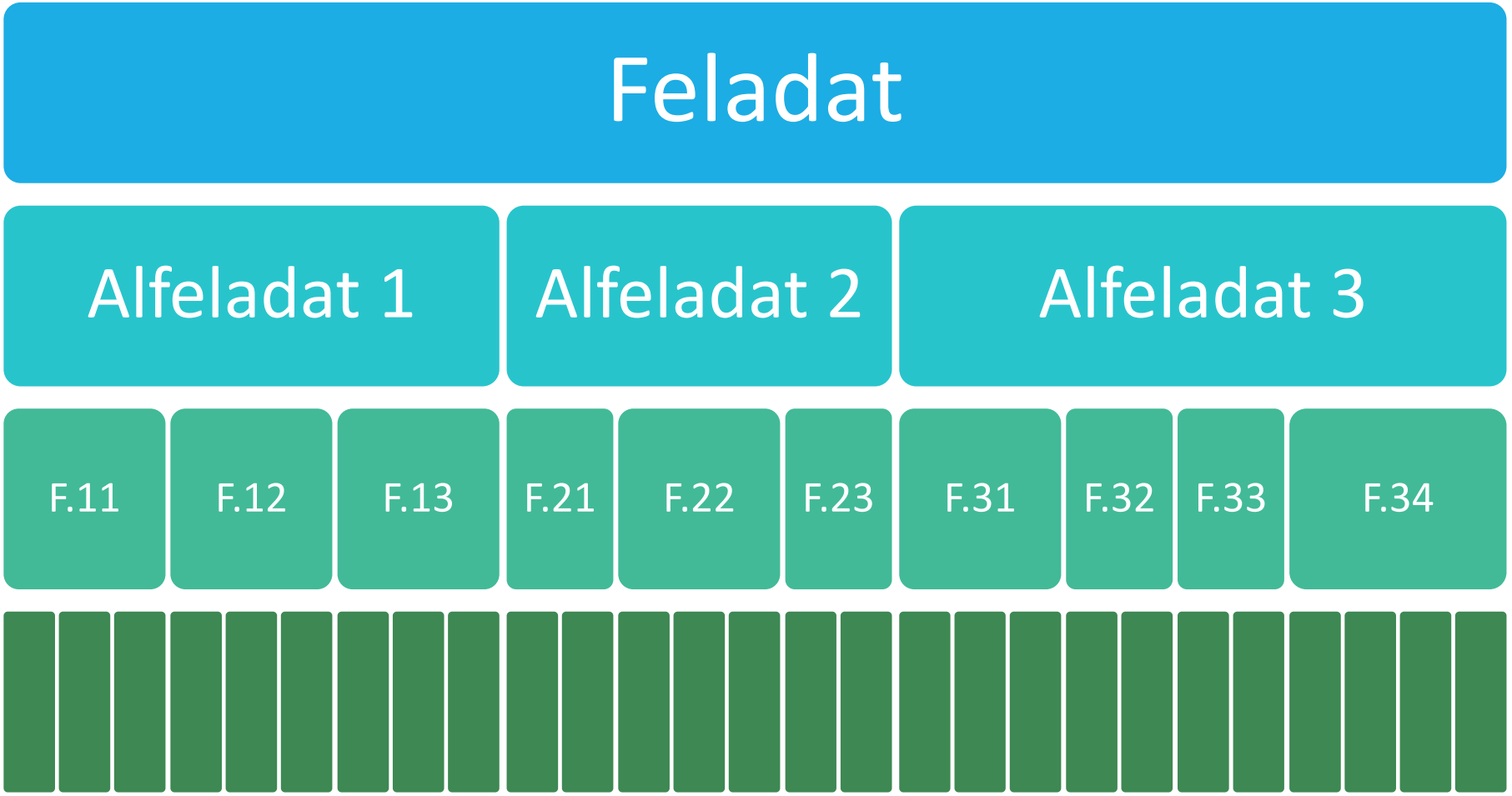
Nagyobb feladat egy részfeladatának megoldása

Algoritmus kódolása

Típusművelet megvalósítása

Matematikai függvény kiszámítása

# Eljárás-absztrakció

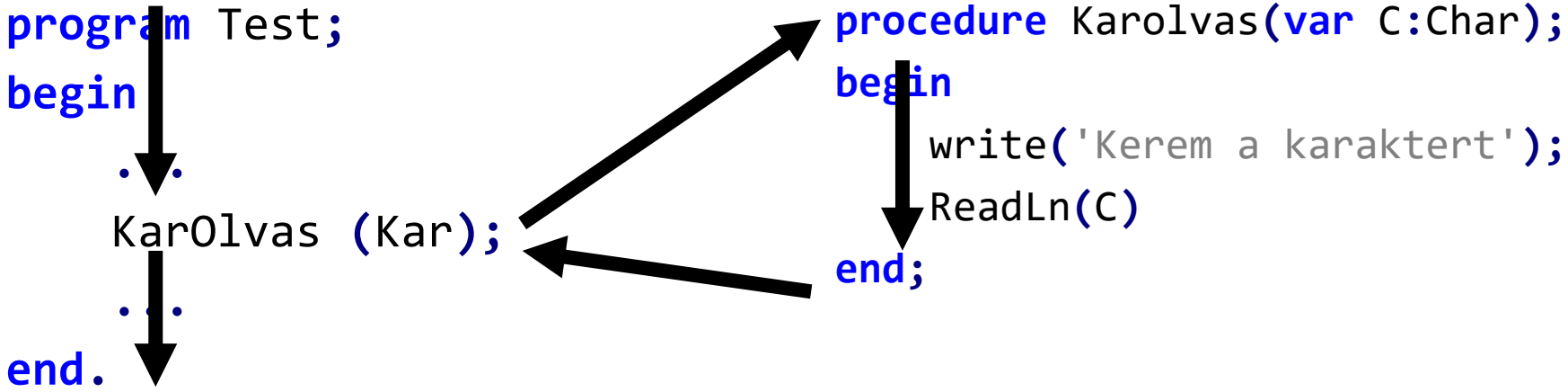


# Alprogram hívása

```
program Test;
procedure Karolvas(var C:Char);
begin
    write('Kerem a karaktert');
    ReadLn(C)
end;

var
    Kar: Char;
begin
    ...
    KarOlvas (Kar);
    ...
end.
```

# Alprogram hívása



# Alprogram

Alprogram=(név, paraméterek, környezet, törzs).

- Az alprogram definíciójakor a formális paraméterekkel írjuk le az adatcsere elemeit
  - A külvilággal való kommunikáció alterének komponenseit
- Az alprogram használatakor pedig az aktuális paraméterek kerülnek ezek helyére
  - Ez a paraméterátadás

# Függvény

```
function Faktorialis ( N: Natural ) return Positive
is
     Fakt: Positive := 1;
begin
     for I in 1..N loop
          Fakt := Fakt * I;
     end loop;
     return Fakt;
end Faktorialis;
```

# Eljárás

```
procedure Cserel ( A, B: in out Integer ) is
    Temp: Integer := A;
begin
    A := B;
    B := Temp;
end Cserel;
```

# Eljárás – függvény

Eljárás végrehajtása: utasítás

Eljárás neve: ige

- `Egyenest_Rajzol(Kezdopont,Vegpont);`

Függvény végrehajtása: kifejezés értékének meghatározása.

Függvény neve: főnév vagy melléknév.

- `if Elemek_Száma(Halmaz) > 0 then ...`

# Paraméterek, visszatérési érték

Információ átadása / átvétele alprogramhívásnál

- paramétereken keresztül
- visszatérési értéken keresztül

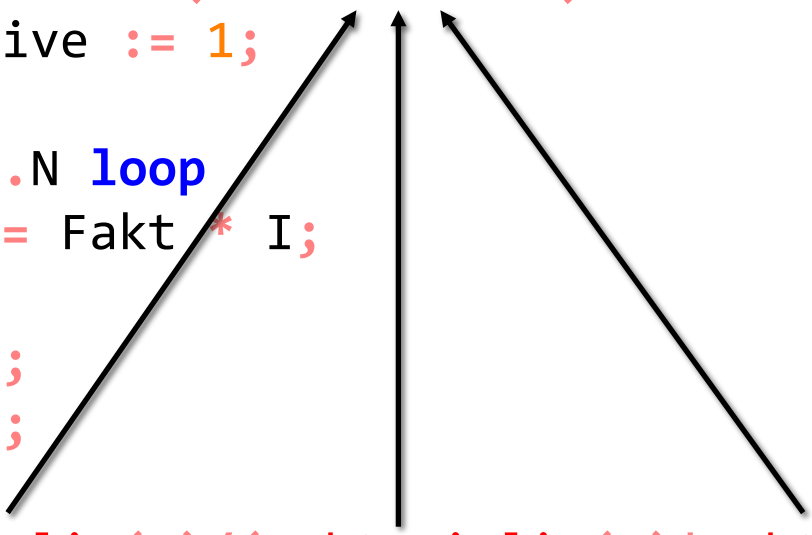
Paramétereknél az információ áramlása

- Merre: a paraméterek módja
- Hogyan: a paraméterátadás technikája (paraméterátadás módja)

Alprogram hívásakor a formális paramétereknek aktuális paramétereket feleltetünk meg.

# Paraméterek

```
function Faktorialis ( N: Natural ) return Positive is
    Fakt: Positive := 1;
begin
    for I in 1..N loop
        Fakt := Fakt * I;
    end loop;
    return Fakt;
end Faktorialis;
```



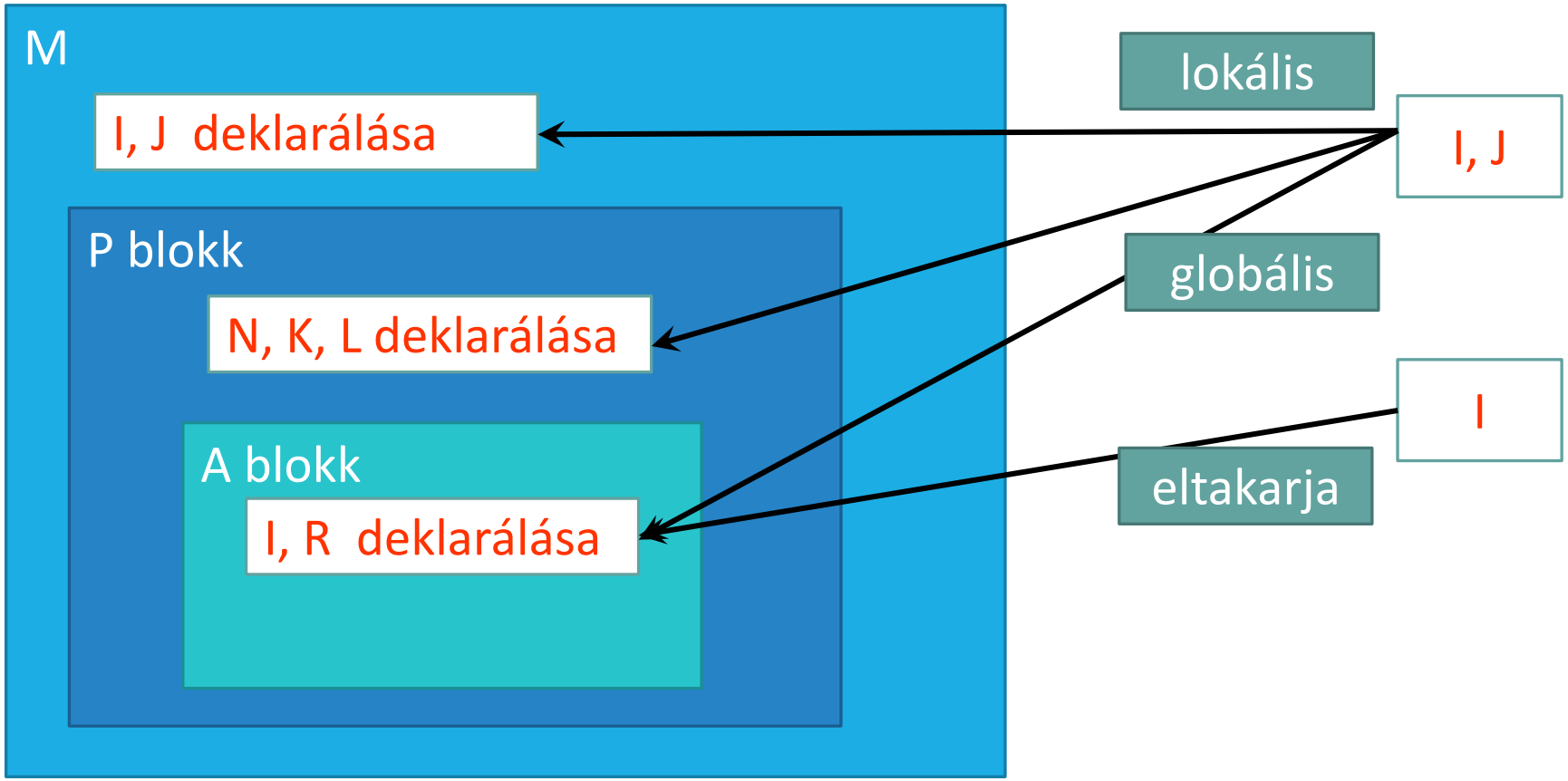
```
L_Al_K:=Faktorialis(L)/(Faktorialis(K)*Faktorialis(L-K));
```

# Alprogram beágyazható

Sok programozási nyelvben:

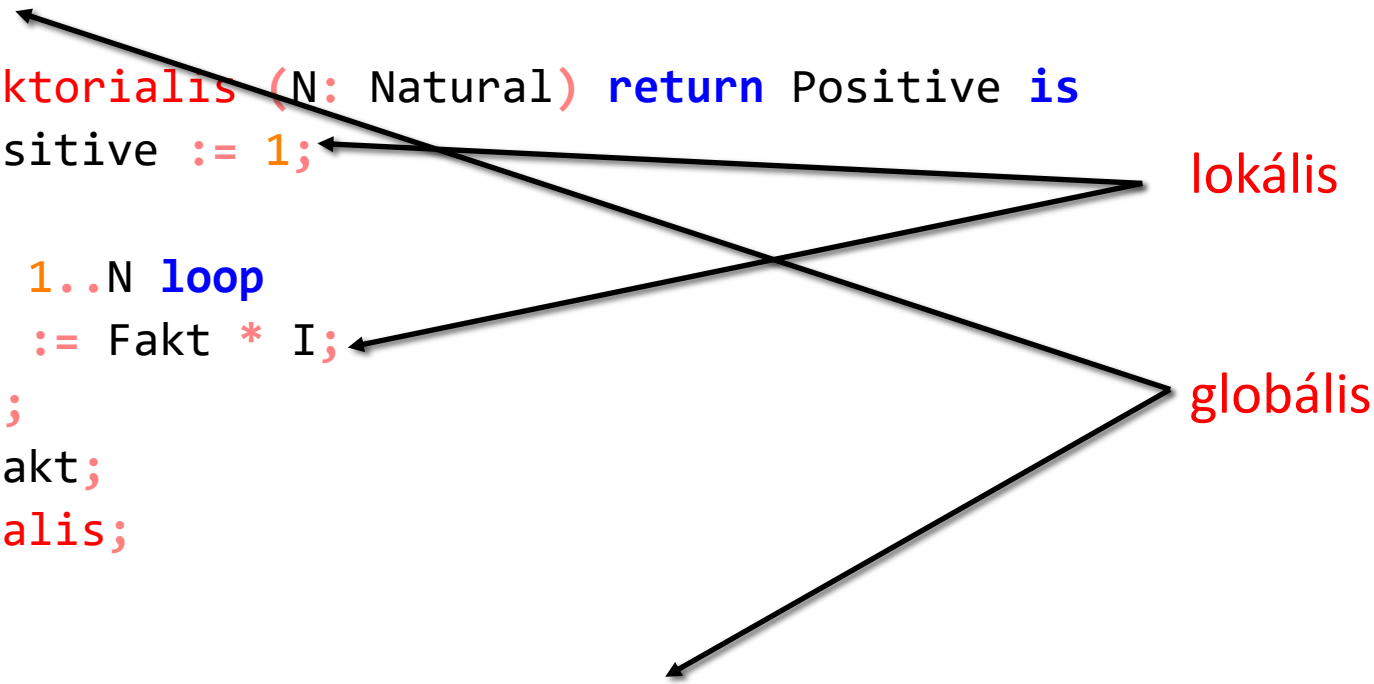
- Blokkszerűen
- Deklarációs részbe
- Lokális - globális deklarációk
- Hatáskör, láthatóság, élettartam

# Változók


M
I, J deklarálása
P blokk
N, K, L deklarálása
A blokk
I, R deklarálása
lokális
I, J
globális
I
eltakarja

# Példa

```
with Ada.Integer_Text_IO, Text_IO;
procedure Faktorialist_Szamit is
   N: Integer;
   function Faktorialis (N: Natural) return Positive is
      Fakt: Positive := 1;
   begin
      for I in 1..N loop
          Fakt := Fakt * I;
      end loop;
      return Fakt;
   end Faktorialis;
begin
   N := 10;
   Ada.Integer_Text_IO.Put(Faktorialis(N));
end Faktorialist_Szamit;
```



# Paraméterek fajtái

Az információáramlás iránya szerint

- Input: hívó $\Rightarrow$ alprogram
- Output: hívó $\Leftarrow$ alprogram
- Update: hívó $\Leftrightarrow$ alprogram

# Paraméterátadási technikák

## A paraméterátadás módja

- Különféle nyelvekben különféle módon adják át a paramétereket
- Legismertebb paraméterátadási módok:
  - érték szerint
  - cím szerint
  - eredmény, érték-eredmény szerint
  - név szerint

# Érték szerinti paraméterátadás

A formális paraméter az alprogram egy lokális változója

- aminek az aktuális paraméter adja a kezdőértéket
- in módú átadásra alkalmas

Az aktuális paraméter

- Tetszőleges kifejezés
- Kiértékelésére egyszer kerül sor
  - Az alprogram végrehajtásának megkezdése előtt

Az alprogram végrehajtása közben

- sem az aktuális paraméter értékének esetleges megváltozása nincs hatással a formális paraméter értékére
- sem a formális paraméter értékének megváltoztatása nincs hatással az aktuális paraméterre

# Érték szerinti paraméterátadás

Példa C-ben

```
int lnko( int a, int b ){
  while( a != b )
    if( a > b )   a -= b;
    else          b -= a;
  return a;
}
```

```
int x,y,z; …
x = 10; y = 5;
z = lnko(x,x+y+1);

a = 10
b = 16
```

# Érték szerinti paraméterátadás

Példa C-ben

```
int lnko( int a, int b ){
  while( a != b )
    if( a > b )   a -= b;
    else          b -= a;
  return a;
}
```

```
int x,y,z; …
x = 10; y = 5;
z = lnko(x,x+y+1);
```

z értéke 2 lesz,
x marad 10

# Érték szerinti paraméterátadás

Csak egyszer értékelődik ki

C példa:

```
int x = 1;
int f( int a ){
  ++x;
  return a+x;
}
int main(){
  int y = f(x);
}
```

a = 1



x = 2

y = 3

# Cím szerinti paraméterátadás

Az aktuális paraméter
- egy változó, vagy
- egy változó komponensét meghatározó kifejezés – balérték - `(x[i+2*j])` lehet.

Az aktuális paraméter kiértékelése
- a hozzá rendelt memóriaterület címének meghatározását jelenti.
- az alprogram végrehajtásának megkezdése előtt kerül sor
  - az aktuális paraméter memóriaterülete rendelődik hozzá.

Az alprogramban
- hivatkozhatunk a formális paraméter értékére,
- adhatunk is neki új értéket
  - update módú átadásra is alkalmas

# Cím szerinti paraméterátadás

Példa Pascalban:

```
procedure Csere
  (var a, b: Integer );
var temp: Integer;
begin
  temp := a;
  a := b;
  b := temp
end;
```

Hívás:

```
…
x: Integer;
y: Integer;
…
x:=3;
y:=6;
Csere(x,y);
```

x=6, y=3

# Cím szerinti paraméterátadás

```
procedure s (var a   : integer);
     // kiírja és lenullázza a paraméterét

procedure p (var a, b: integer);
     // kiírja és lenullázza a paramétereit
begin
   s(a);
   s(b)
end;
```

Mit csinál majd `p(x,x)`?

# Cím szerinti paraméterátadás

„Alias” – amikor a program egy pontján két különböző változó ugyanazt az egyedet jelenti

```
var globalis: Integer;

procedure r (var lokalis: Integer);
begin
    globalis := globalis + 1;
    lokalis := lokalis + globalis
end;

begin
  globalis := 1;
  r(globalis)
end.
```

Mennyi lesz `globalis` értéke?

# Eredmény szerinti paraméterátadás

A kimenő paraméterek megvalósításához.

- Az alprogram formális paraméterében kiszámított eredményt helyezi vissza az aktuális paraméterbe.

A formális paraméter

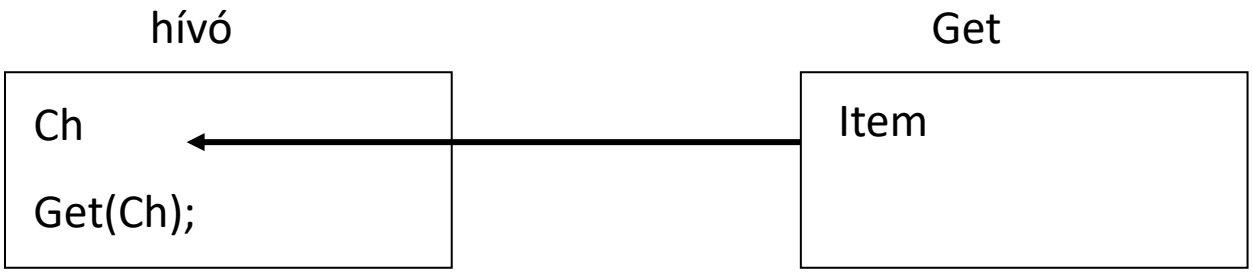
- az alprogram lokális változója,
- értéke az alprogram befejeződésekor kimásolódik az aktuális paraméterbe.
- nem kapja meg az aktuális paraméter értékét az alprogram hívásakor
  - az információáramlás egyirányú (out módú átadásra alkalmas).

Az aktuális paraméter balérték kell legyen.

# Eredmény szerinti paraméterátadás

Ada példa:

```
procedure Get(Item: out Character ) is …
```




Object Pascal példa:

```
procedure GetInfo(out Info: SomeRecordType);
```

# Érték/Eredmény szerinti paraméterátadás

Az alprogram befejezésének pillanatáig megegyezik az érték szerinti paraméterátadással.

Az alprogram végrehajtásának befejezésekor az aktuális paraméter felveszi a formális paraméter pillanatnyi értékét (update módú átadásra alkalmas).

Az aktuális paraméter csak „balérték” lehet

Miben különbözik a cím szerinti átadástól?

# Érték/Eredmény szerinti paraméterátadás

```
procedure Paramproba is
  A,B:Integer;
  procedure Parcsere(X,Y: in out Integer) is
    Temp: Integer;
  begin
    Temp:=X; X:=Y; Y:=Temp;
    A:=12; B:=99;
  end;
begin
  A:=1; B:=2;
  Parcsere(A,B);
end;
```

X=1,Y=2

X=2,Y=1

A=1,B=2

A=2,B=1

# Érték/Eredmény szerinti paraméterátadás

```
procedure s (a    : in out integer) is …
    -- kiírja és lenullázza a paraméterét

procedure p (a, b: in out integer) is
    -- kiírja és lenullázza a paramétereit
begin
   s(a);
   s(b);
end;
```

Mit csinál majd `p(x,x)`?

# Érték/Eredmény szerinti paraméterátadás

```
procedure Program is
 globalis: Integer;
 procedure r (lokalis: in out Integer) is
 begin
  globalis := globalis + 1;
  lokalis := lokalis + globalis;
 end r;
begin
  globalis := 1;
  r(globalis);
end Program;
```

Itt mennyi lesz a `globalis`?

# Név szerinti paraméterátadás

Az aktuális paraméterként leírt teljes kifejezés adódik át

- minden használatkor (dinamikusan!) kiértékelődik;
- például az a[i] hivatkozás változhat, ha menet közben az i értéke változik(!)
- (update módú átadásra is alkalmas)

Lusta kiértékelés…

# Példa

```
program
  I: integer;
  A: integer array;
  SWAP_BY_NAME: procedure(X: name, Y: name)
   TEMP: integer;
   begin
    TEMP := X;  X := Y;   Y := TEMP;
  end;
 begin
  I := 3;
  A[I] := 6;
  output I, A[3];
  SWAP_BY_NAME(I, A[I]);
  output I, A[3];
 end;
```

*Output*

```
I = 3   A[3]  = 6
I = 6   A[3]  = 6 de A[6] = 3 lesz!
```

# Egy híres példa

## Jensen's device  kifejezések kiértékelésére – Algol60

```
real procedure sum (expr, i, low, high);
   value low, high;
   real expr;          -- név szerint az expr és az i,
                       -- ez a default Algolban!
    integer i, low, high;
    begin
     real rtn;
     rtn := 0;
     for i := low step 1 until high do
        rtn := rtn + expr;
     sum := rtn;
end sum
y := sum (3*x*x - 5*x + 2, x, 1, 10);
```

Mit csinál?

# Alprogramok paraméterei

Az alprogramok paramétereinek száma általában kötött, de lehet változó is.

- Bizonyos mértékig szimulálható egy tömb átadásával
- Ezzel nem mindig oldható meg eltérő típusú paraméterek átadása.

# Paraméterek száma

A programnyelvek kezelhetik a túl sok vagy túl kevés aktuális paraméter megadását.

A túl kevés paraméter megadása esetén több módszer alkalmazható:

- alapértelmezett értékek adhatók meg, amelyek a hiányzó aktuális paraméterek helyébe lépnek (pozíció vagy név szerint)
- vesszőket kell tenni a kihagyott paraméterek helyett
- amíg nincsenek alapértelmezett értékek, addig kötelező az aktuális paramétereket megadni, tehát az alapértelmezések csak a paraméterlista végén lehetnek

# Formális-aktuális paraméterek megfeleltetése

```
procedure Get_Line ( File : in  File_Type;
        Item : out String; Last : out Natural )
```

pozícionális formában: az aktuális paramétereket abban a sorrendben kell felsorolni, ahogy a formális paraméterek voltak az alprogram specifikációjában:

```
Get_Line(F, S, N);
```

névvel jelölt formában: a paraméterek sorrendje tetszőleges:

```
Get_Line(File=>F, Last=>L, Item=>S);
```

kevert formában: mindig a pozícionálisan megadott paramétereknek kell elöl állniuk:

```
Get_Line(F, Last=>L, Item=>S);
```

# Paraméterek feltételezett értéke

Az in módú paraméterekhez rendelhetünk feltételezett bemenő értéket.

- Az alprogram specifikációs része tartalmazza ezt a feltételezett értéket
- Ez akkor veszi fel a formális paraméter, ha neki megfelelő aktuális paramétert nem adunk meg.

# Példa

```
procedure New_Line (File : in File_Type;
                    Spacing : in Positive_Count := 1 )
```

Hívhatjuk

- úgy, hogy megadjuk a Spacing értékét, vagy
- úgy, hogy nem ilyenkor a feltételezett értéket használja az eljárás.

```
New_Line(F);        -- 1 sort emel az F fájlban
New_Line(F,1);      -- az előzővel egyenértékű
New_Line(F,42);     -- 42 sort emel
```

# Paraméterek feltételezett értéke

Lehet több alapértelmezett értékkel rendelkező paraméter is.

- Ilyenkor bármelyik aktuális paramétert el lehet hagyni, akár többet is (segít a névvel jelölt hívás).

```
procedure Egyenest_Rajzol (
                Kezdőpont, Vegpont: in Pont;
                Vastagsag: in Positive := 1;
                Szin: in Szinskala := Fekete )
Egyenest_Rajzol(P1,P2);
Egyenest_Rajzol(P1,P2,3);
Egyenest_Rajzol(P1,P2,Szin=>Piros);
```

# Alprogramok túlterhelése (átlapolása)

```
procedure Cserél ( A, B: in out Integer );
procedure Cserél ( A, B: in out Boolean );
```

Azonos név – paraméterek száma és/vagy típusa különböző

A használatból (a hívásból) fog kiderülni, hogy melyikre gondolunk

- Ki kell derülnie!

# Kérdések

Mellékhatás – mi az?

```
z:= sin(x);
y:= sin(x);
z = y?
```

mi az elvárásom?

```
zz:= rnd();
xx:= rnd();
zz = xx
```

és most?

# Kérdések

Mellékhatás 1.: globális változók használata

```
with Text_Io; use Text_Io;
procedure Globprobe is
      I:Integer;
      function Glob(J:Integer) return Integer is
      begin
            I:=I+1;  return I+J;
      end;
begin
      I:=2;
      Put_Line(Integer'Image(Glob(1)));
            …
      Put_Line(Integer'Image(Glob(1)));
end;
```



# Kérdések

Mellékhatás 2.: függvény paraméter is változik

```
int f(int val, int& ref){
  val++;
  ref++;
  return val+ref;
};
void main(){
  int i=1;
  int j=1;
  int k;
  k=f(i,j);
}
```



VIGYÁZAT!!

# Operátorok

Infix alakban is írható műveletek

```
A+42        ”+”(A,42)
```

Operátorok is átlapolhatók:

```
function "+" ( A, B: Matrix ) return Matrix;
```

# Rekurzió

Közvetlenül vagy közvetve önmagát hívó alprogram

```
function Faktorialis ( N: Natural ) return Positive is
begin
    if N > 1 then
            return N * Faktorialis(N-1);
    else return 1;
    end if;
end Faktorialis;
```

# Rekurzió

A rekurzív alprogramok paraméterátadás szempontjából nem különböznek a szokásos alprogramoktól.

Ezért a paramétereket rekurzív alprogramoknak cím szerint átadni csak elővigyázatosan szabad, mert a sorozatos hívások interferálhatnak.

# Fontosabb kérdések

Van-e eljárás?

Van-e függvény?

Mely paraméterátadási módok léteznek?

- érték szerinti
- eredmény szerinti
- érték-eredmény szerinti
- cím szerinti / konstans cím szerinti
- név szerinti

# További fontos kérdések

Meghatározható-e a paraméterek információátadásának iránya?

Adható-e a formális paramétereknek alapértelmezett érték?

Túlterhelhetők-e az alprogramnevek?

Az operátorok átlapolhatók-e?

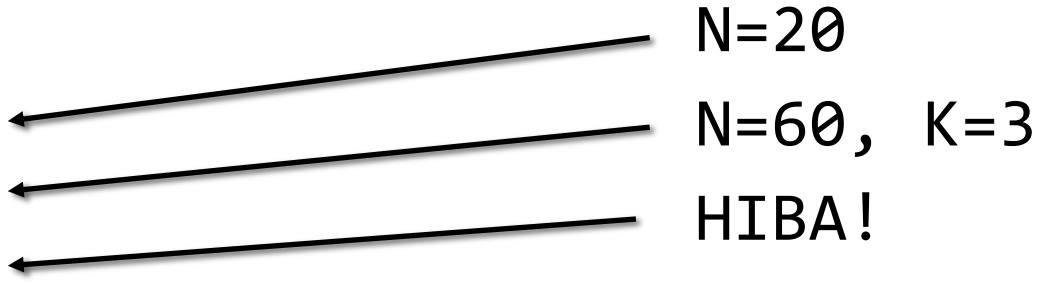
Definiálhatók-e új operátorok?

Típus-e az alprogram? Lehet-e alprogrammal paraméterezni?

# Pascal

A paraméterek alapértelmezésben érték szerint adódnak át, de a VAR kulcsszóval cím szerinti átadás érhető el:

```
program A;
  procedure Megszoroz(var Mit: Integer; Szorzo: Integer);
  begin
     Mit:= Mit * Szorzo;
  end;
var N, K :Integer;
begin
  N:=5; K:=3;
  Megszoroz(N, 4);
  Megszoroz(N, K);
  Megszoroz(5, N)
end.
```



# C/C++

Csak függvény van

- `void` visszatérési érték, ha eljárást szeretnénk

Csak érték szerinti paraméter átvétel

- cím szerinti: mutatók vagy referencia kell.

Lehet a paraméterek számára is kezdőértéket adni.

```
void f(int val, int& ref){
  val++;
  ref++;
}
```

```
void g(){
  int i=1;
  int j=1;
  f(i,j);
}
```

i==1,
j==2 lesz

# C++

const& paraméter – nem változtatható a törzsön belül

```
float fsqrt(const float&);
...
float r=fsqrt(x);
```

Tömb paraméterek

- a tömb első elemére hivatkozó mutató adódik át
- vagyis nem érték szerint adódnak át
- méretük nem adódik át

```
int strlen(const char*);
void f(){
  char v[]='egy tomb';
  int i=strlen(v);
}
```

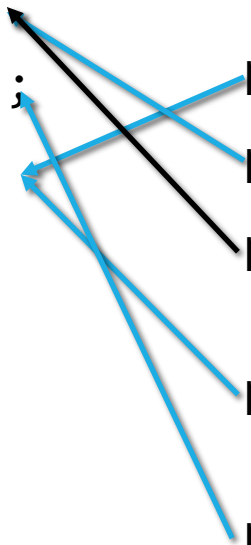


A T[ ] típusú paraméter T* típusúvá lesz átalakítva ekkor.

# C++

## Túlterhelés lehetséges:

```
void print(int);
void print(const char*);
void print(char);
…
```



```
void h(char c, short s, int i){
 print(c); //Pontos illeszkedés
 print(i); //Pontos illeszkedés
 print(s); //Konverzió

 print('a');//Pontos illeszkedés

 print("a");
           //Pontos illeszkedés
}
```

# Java

Csak valamely osztály metódusa lehet

Az aktuális objektum attribútumaira hivatkozik

Paraméterátadás

- primitív típusok érték szerint
- összetett típusú paraméterek referencia szerint
  - mutable-immutable lehetőség!

# Smalltalk

Üzenetküldésekkel érjük el a metódusokat, így a paraméterátadás érték szerintinek tekinthető – referenciák figyelembevételével

- A következő sor az a objektumnak küld üzenetet, két paraméterrel a két paraméter megnevezésével és aktuális értékével
- `a at:3 put:5`

# Ada

`in, out, in out`

Az összetett típusú értékek esetén a fordítóprogram választhat az érték-eredmény, illetve a cím szerinti átadás között.

A függvényeknek csak in paramétereik lehetnek.

Az in paramétereknek lehet alapértelmezett értékük is.

Az alprogramok határozatlan méretű tömb típust is elfogadnak a formális paramétereik között.

# SWIFT

## Lehetőségek

- Konstans paraméter, inout paraméter, kevert formában történő átadás, feltételezett érték létezik, függvényt is át lehet adni paraméterként, változó számú paraméter átadása

## Továbbá:

- Argumentum címke és függvényparaméter megkülönböztetése
- Több visszatérési értéke is lehet

# SWIFT

```
func greet(name s1: String, day s2: String) ->
String {
    return "Hello \(s1), today is \(s2)."
}
greet(name: "Bob", day: "Tuesday")

func calculateStatistics(scores: [Int]) ->
    (min: Int, max: Int, sum: Int ) {
  …
  return (min, max, sum)
}
```

# Kotlin

Infix függvény definiálása

```
infix fun Int.shl(x: Int): Int { ... }
1 shl 2
```

Vezérlési szerkezetekkel összeköthető – paraméterezhető kifejezések:

```
fun hasPrefix(x: Any) = when(x) {
    is String -> x.startsWith("prefix")
    else -> false
}
```

# Procedurális- vagy adatabsztrakció

Egy szoftver rendszer mindig végrehajt

- bizonyos tevékenységeket
- bizonyos adatokon.

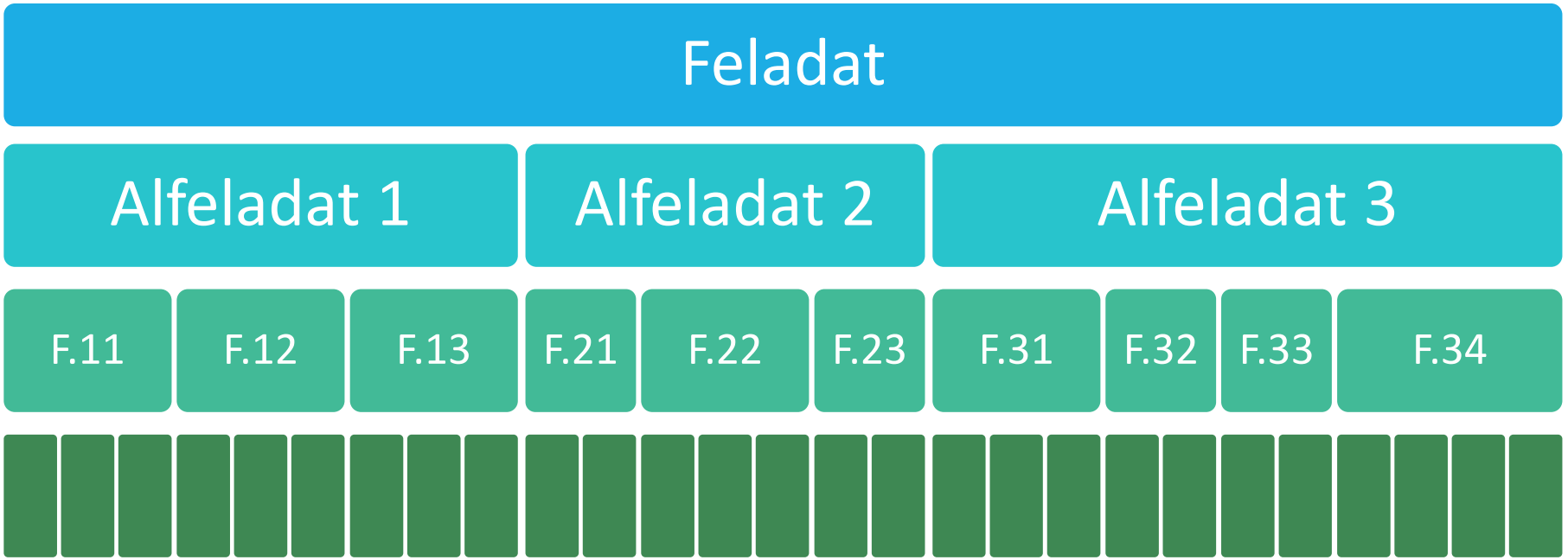
A tervezés központi kérdése, hogy a rendszer felépítését mire alapozzuk

- a végrehajtandó tevékenységekre, vagy
- az adatokra?

# Procedurális absztrakció

Top-down megoldás

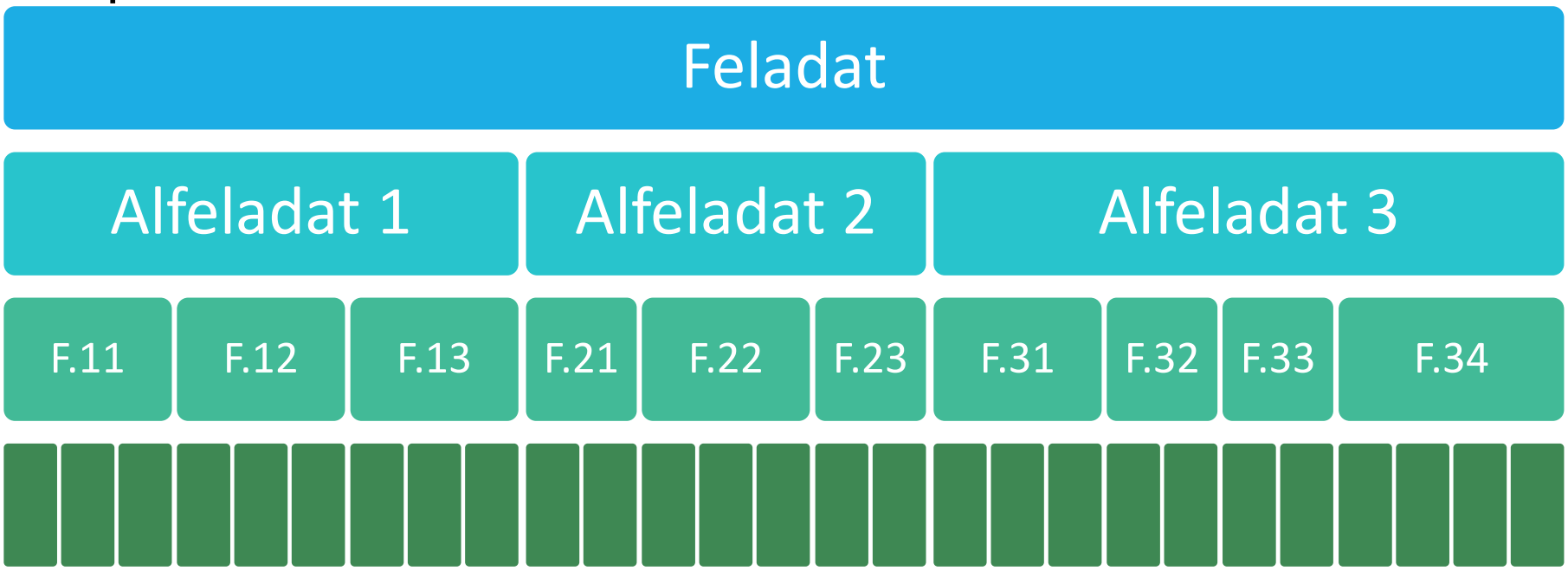
- A megközelítés hátránya: ha változik a specifikáció …



# Procedurális absztrakció

Mivel egy alprogram specifikáció a magasabb szintű specifikációtól függ, így a változás tovább gyűrűzik lefelé, és végül egy egész kis változásnak nagyon nagy hatása (és költsége) lehet

- Specifikáció változásának hatása



# Sablonok Kivételkezelés

KÖVETKEZŐ ALKALOMMAL




# Programozási nyelvek és módszerek

SABLONOK, KIVÉTELKEZELÉS

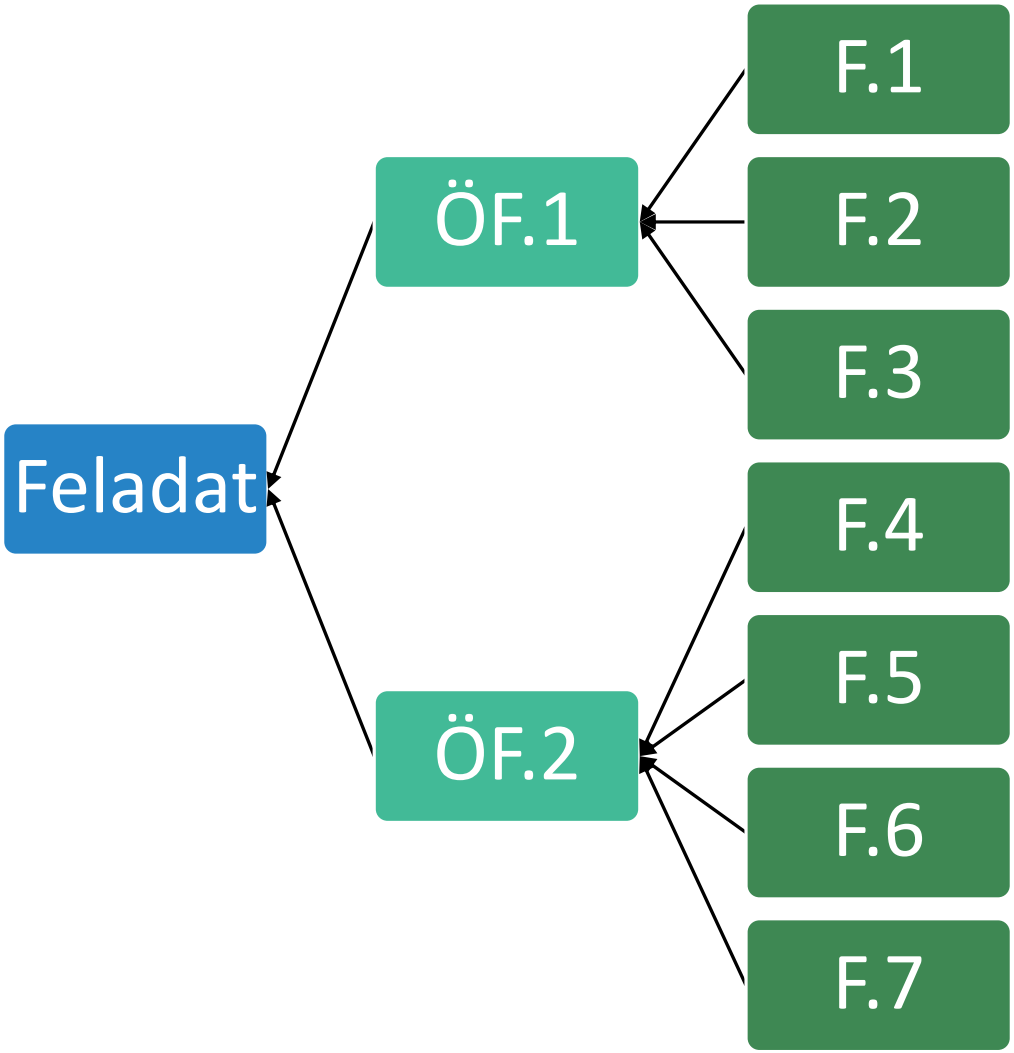
# Adatabsztrakció

# Adatabsztrakció

Ha elég magas szintről analizáljuk a problémát, akkor úgy tűnik, hogy

- a rendszerek jobban jellemezhetőek hosszú távon objektumaikkal, mint a rájuk alkalmazott tevékenységekkel.
- Ezt a megközelítést választva, először megpróbáljuk jellemezni az objektumokat, és az objektumok osztályait (~az adattípusokat) amelyek szerepet játszanak a rendszerben.
- Az új adattípusokat a már meglévők felhasználásával tervezzük – ez a bottom-up tervezés.
- Ez azt jelenti, hogy  az elérhető komponensek szintjéről indulunk, abból építkezünk.
- Ezután oldjuk meg a problémát.

# Alulról felfelé építkezés



# Típus a programozó szemszögéből

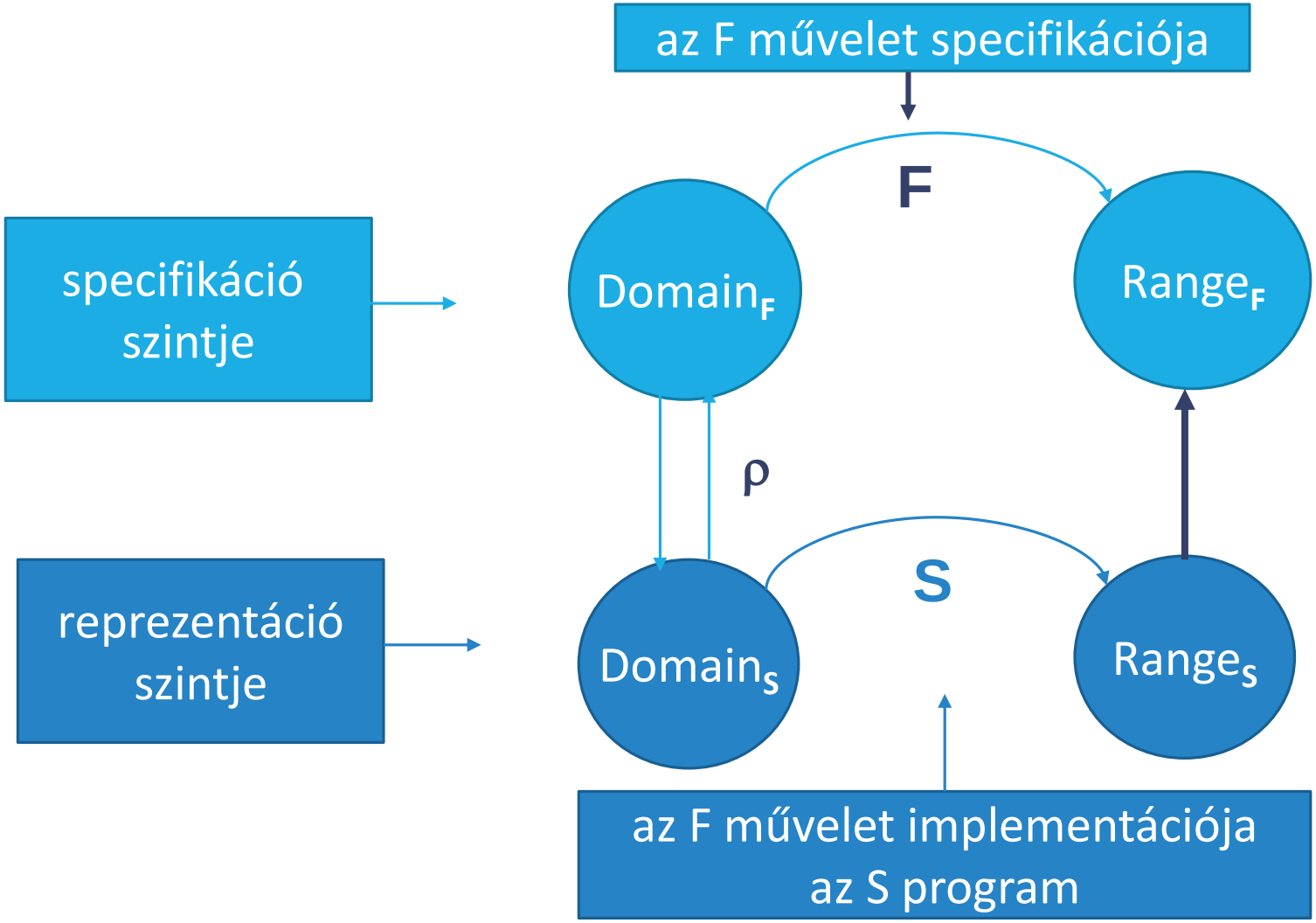
Típusspecifikáció („mit”)

- alaphalmaz: a valós világ minket érdeklő objektumainak halmaza
- specifikációs invariáns ($I_S$) - ezt a halmazt szűkíti
- típusműveletek specifikációja

Típus megvalósítás („hogyan”)

- Reprezentációs függvény
- Típus invariáns
- Típusműveletek megvalósítása

# A típus specifikáció és a típus kapcsolata



# Elvárások és eszközök

Elvárások a programozási nyelvekkel szemben

Absztrakt típusok megvalósításának eszközei

# Elvárások a programozási nyelvekkel szemben

## Modularitás

- Az egyes típusokat önálló fordítási egységekben lehessen megvalósítani!
- Ez biztosítja a típusok újrafelhasználhatóságát valamint a hatékony programfejlesztést
  - az egyes modulok könnyedén átvihetők más programokba,
  - a különböző egységeket más-más programozó is fejlesztheti
    - nem zavarva egymás munkáját

# Absztrakt típusok megvalósításának eszközei

## Modulokra bontás

- A professzionális használatra szánt programozási nyelvek mindegyikében megjelenik a modularizáció támogatása.
- A modularizáció alapját egyre inkább a típusokra bontás jelenti
  - azaz egy modul egy típust implementál
- De
  - sokszor van szükség „alprogram könyvtárak”-ra
  - szükség lehet csak implementációs célokat szolgáló típusok megvalósítására is
    - lehetőség szerint az azt használó adattípus moduljában rejtünk el

# Elvárások a programozási nyelvekkel szemben

## Adatrejtés

- A nyelv támogassa a reprezentáció elrejtését.
- Így a nyelv biztosítja, hogy az adott típus használója csak a specifikációban megadott tulajdonságokat használhassa ki.
- Lehetővé téve a reprezentáció, illetve az implementáció megváltoztatását, a változások a programban felfelé gyűrűzése nélkül.

# Absztrakt típusok megvalósításának eszközei

## A reprezentáció elrejtése

- Az adatrejtés támogatására az összetett típusok komponenseinek láthatóságát szintekre bontásával érhető el gyakran
  - Meghatározva, hogy pontosan kik férhetnek hozzá az adott komponenshez.
- Leggyakrabban három szint:
  - nyilvános (public) – az adott komponens mindenki számára látható
  - védett (protected) – az adott komponens csak a leszármazottak számára látható
  - privát (private) – a reprezentáció teljesen rejtett része, csak a műveletek implementációjában használható komponensek

# Absztrakt típusok megvalósításának eszközei

Adatrejtéshez új láthatósági szintek bevezetése

- Java
  - package-private
- Delphi
  - published (futás idejű típusinformáció)
  - strict private, strict protected
- C++
  - friend
- stb.

# Kérdések

Típus vagy objektum szinten szabályoz-e?
(C++ - SmallTalk)

Vannak olyan nyelvek ahol egyáltalán nincs ilyen jellegű szabályozás

- Python

Egyes nyelvekben a láthatóság még ennél is finomabban szabályozható

- a típus megvalósításakor rendelkezhetünk arról, hogy mely osztály (és leszármazottai) férhessenek hozzá az adott komponenshez.

```
class A
  feature {B, C}
     X: INTEGER;
  feature {ANY}
    make(p_name : STRING ) is
       require
            p_name /= ""
       do
            name := p_name ….
  feature {NONE}
     Y: ARRAY[INTEGER];
end
```



B és C látja (és utódaik)



public



private

# Elvárások a programozási nyelvekkel szemben

## Konzisztens használhatóság

- A felhasználói és a beépített típusok ne különbözzenek a használat szempontjából!
- A típusokat minél inkább a valós világban megszokott, „természetes” módon lehessen kezelni!
- Ugyanúgy lehessen összetett típusokat (tömböket, rekordokat stb.) definiálni a segítségükkel, ugyanúgy lehessen változókat definiálni a saját típusokkal stb.

# Absztrakt típusok megvalósításának eszközei

## Konzisztens használat

- Az új típust be lehessen illeszteni a nyelv logikájába!
  - Ha például az adott nyelvben az a konvenció, hogy egy típust a Read művelet olvas be, akkor fontos, hogy a saját típusokhoz is definiálhasson a programozó egy Read nevű műveletet
  - Azaz legyen lehetőség a Read azonosító túlterhelésére vagy átlapolására.
    - Egy azonosító túlterhelése vagy átlapolása (overloading) azt jelenti, hogy a programszöveg egy adott pontján az azonosítónak több definíciója is érvényben van.
  - Speciálisan az operátor-túlterhelésnek nagy jelentősége van abban, hogy a felhasználói típusokat természetes használatuknak megfelelően használhassuk.

# Elvárások a programozási nyelvekkel szemben

Generikus programsémák támogatása.

- A programozó lehetőleg minél általánosabban írhassa meg programjait!
- A nyelv adjon lehetőséget az ismétlések minimalizálására.
  - Közvetlen kódismétlések elkerülésére
  - Magasabb szintű megoldási struktúrák többszörös megírásának elkerülésére.
- Ez nagyban javítja a kód olvashatóságát és karbantarthatóságát.

# Elvárások a programozási nyelvekkel szemben

Specifikáció és implementáció szétválasztása külön fordítási egységbe

- Ekkor az adott típust használó más modulok a típus specifikáció birtokában elkészíthetők, függetlenül a tényleges implementációtól.

Modulfüggőség kezelése

- A fordító program kezelje maga a modulok közötti relációkat (egyik használja a másikat stb.).

# Absztrakt típusok megvalósításának eszközei

## Specifikáció és implementáció szétválasztása

- Azon nyelvekben, melyek támogatják az „egy modul egy típus” elvet, néha lehetőség van a típusspecifikációnak az implementációtól külön, önálló fordítási egységben történő leírására.
- Ez segíthet abban, hogy az egyes modulok egymástól függetlenül elkészíthetőek legyenek.

# C/C++

A specifikációt fizikailag be kell másolni minden olyan fordítási egységbe, amely az adott típust használni akarja

- Így nincs igazi szétválasztásról.
- A bemásolás megkönnyítésére a specifikáció egy külön, speciális forrásfájlban (header fájl) leírható
  - az előfordító segítségével azt beemelhetjük a megfelelő fordítási egységekbe.
- Ha egy típus specifikációját többször is beírjuk egy fordítási egységbe, az hibát jelent
  - ennek kivédéséről a header fájl megírásakor a programozónak gondoskodnia kell.

# C/C++

```
typedef double Angle;
class Complex {
public:
 Complex(double r=0, double i=0){ R = r; I = i;}
 Complex operator =(Complex z){ R = z.R; I = z.I; return *this; }
 Complex operator +(Complex z) { return  Complex(R+z.R,I+z.I);}
 Complex operator +(double x) { return Complex(R+x,I);}
 Complex operator *(Complex z);
 Complex operator *(double x);
 double Re();
 double Im();
 double Abs();
 Angle  Phi();
private:
   double R;
   double I;
}
```

# C/C++

```
Complex operator + (double x, Complex z) {
    return z+x
}
```

Vagy: **friend**

# C/C++

```
#include "complex.h"
Complex Exp(Complex z, double eps = 0.0001)
{
    Complex zi = Complex(1.0);
    Complex sum;
    double  i  = 1.0;

    while(zi.Abs() >= eps )
    {
        sum = sum+zi;
        i  += 1.0;
        zi  = (zi*z)/i;
    }
    return sum;
}
```

Más nyelvekben a fizikai szétválasztás nem lehetséges

- a fejlesztő eszköz támogatja a kód többszintű nézetét
- a felhasználás szempontjából lényegtelen részletek eltakarhatók.

A Java, C# stb. nyelvekben a specifikáció és implementáció összemosódik.

- Segítséget a dokumentációs lehetőségek jelentenek, amelyek a megfelelően dokumentált programkódból elő tudják állítani a specifikáció szöveges leírását.

# Modulfüggőség kezelése

- Egy program legmagasabb szintű építőkövei a modulok.
- A program működése ezen modulok interakciója.
- Minden modul igényel szolgáltatásokat és segítségükkel más szolgáltatásokat valósít meg, így a modulok között egyfajta függőségi reláció alakul ki, s egy modul megváltozása esetén szükségessé válhat a tőle függő modulok újrafordítása.

Némelyik programozási nyelv (például a C vagy C++) teljes egészében a programozóra bízza ezen függőségek kezelését, minden fordítási egységet önállóan kezel.

- Éppen ezért vannak speciális eszközök (pl. make), amelyek kizárólagos feladata ennek a feladatnak a megkönnyítése.
- Ezek a  megoldások nem tökéletesek, előfordul, hogy túl sokszor fordítanak újra valamit, vagy éppen nem veszik észre az újrafordítás szükségességét stb.

Más nyelvekben a függőségek kezelése a fordító program feladata.

- Az Ada nyelvben például a with utasítással kell megadnunk, hogy milyen más fordítási egységektől függ a szóban forgó modul.
- Ez garantálja azt, hogy a modul fordítása előtt mindazon modulok specifikációs része lefordításra kerül (ha az szükséges), amelyektől az adott modul függ.

# Kérdések

Saját adattípus önálló fordítási egységként megvalósítható?

Reprezentáció elrejtése megvalósítható?

Milyen láthatósági szintek vannak?

Beépített típusokkal megegyező módon használható?

Van-e azonosító túlterhelés/ operátor túlterhelés/ free operátor?

Specifikáció és implementáció különválasztható?

# Sablonok

# Sablonok – bevezetés

Újrafelhasználható, könnyen karbantartható programokra van szükség.

- Ennek a jegyében készít a programozó minél általánosabb típusokat és alprogramokat, hogy azokat mind az éppen aktuális, mind egyéb programjaiban a lehető legszélesebb körben használhassa.
- Maga a programozási módszertan is olyan programozási megoldásokat tanít, amelyek általánosan használhatóak.
- Ezt a törekvést a programozási nyelvek is támogatják kisebb-nagyobb mértékben
  - Most ezeket az eszközöket gyűjtjük össze.
  - Az eszközök némelyike olyan megszokott és hétköznapi, hogy talán nem is vesszük észre, hogy ebbe a kategóriába tartoznak...

# Sablonok – bevezetés

Alprogramok

- Alprogramok paraméterezése

Típusok paraméterezése

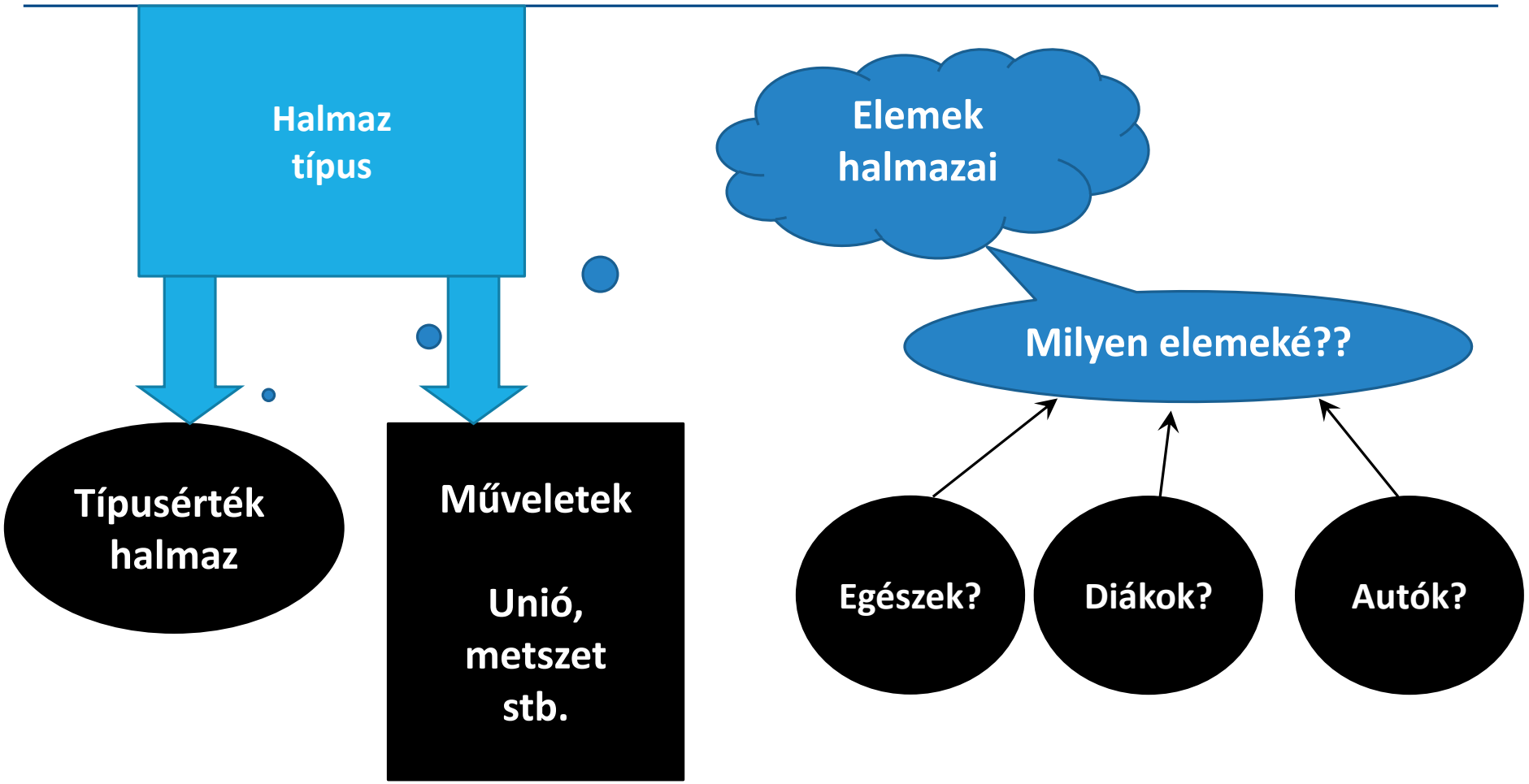
- Például az Ada-ban a diszkriminánsos rekordok, vagy a határozatlan indexhatárú tömb típusok

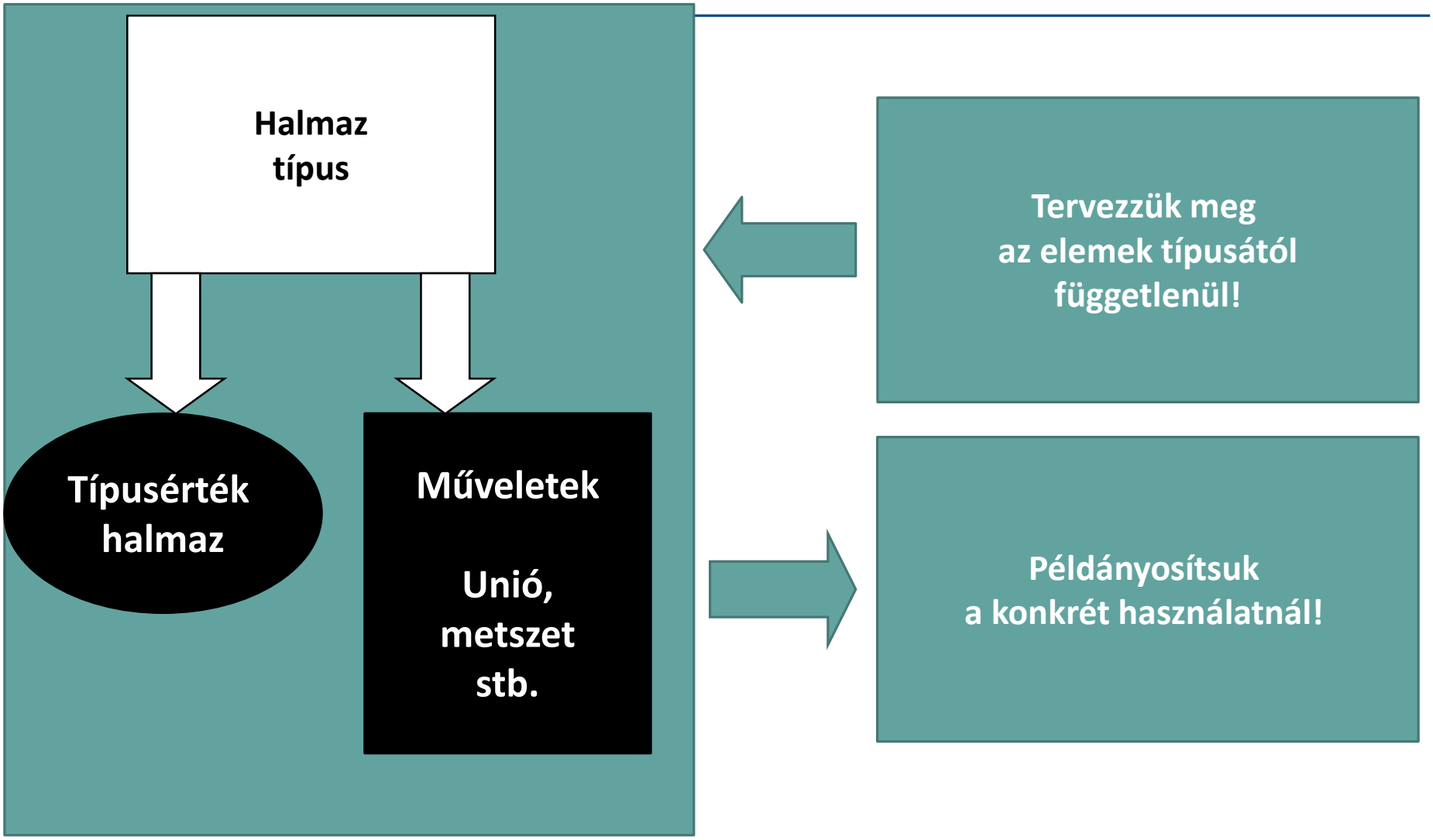
Alprogrammal történő paraméterezés

# Sablonok – cél

Típusokkal történő paraméterezés

- Programozási tételeink, alapvető adatszerkezeteink és adattípusaink is „absztrakt” fogalmakat használnak
  - a számukra feltétlenül szükséges megszorításokat teszik ezekre a fogalmakra.
- Ennek következménye, hogy általában nem egyetlen típus elégíti ki ezen megszorításokat, hanem sok.
- Szeretnénk a fenti tételek, adatszerkezetek, típusok implementációját csak egyszer megadni és azt a leírást általánosan az összes, a feltételeket kielégítő típussal használni.


Halmaz típus
Típusérték halmaz
Műveletek
Unió, metszet stb.
Elemek halmazai
Milyen elemeké??
Egészek?
Diákok?
Autók?


Halmaz
típus
Típusérték
halmaz
Műveletek
Unió,
metszet
stb.
Tervezzük meg
az elemek típusától
függetlenül!
Példányosítsuk
a konkrét használatnál!

# Példák

Keresés vektorban lineárisan

Maximális elem keresése vektorban

Vektor elemeinek rendezése

Verem, Sor

Prioritásos sor

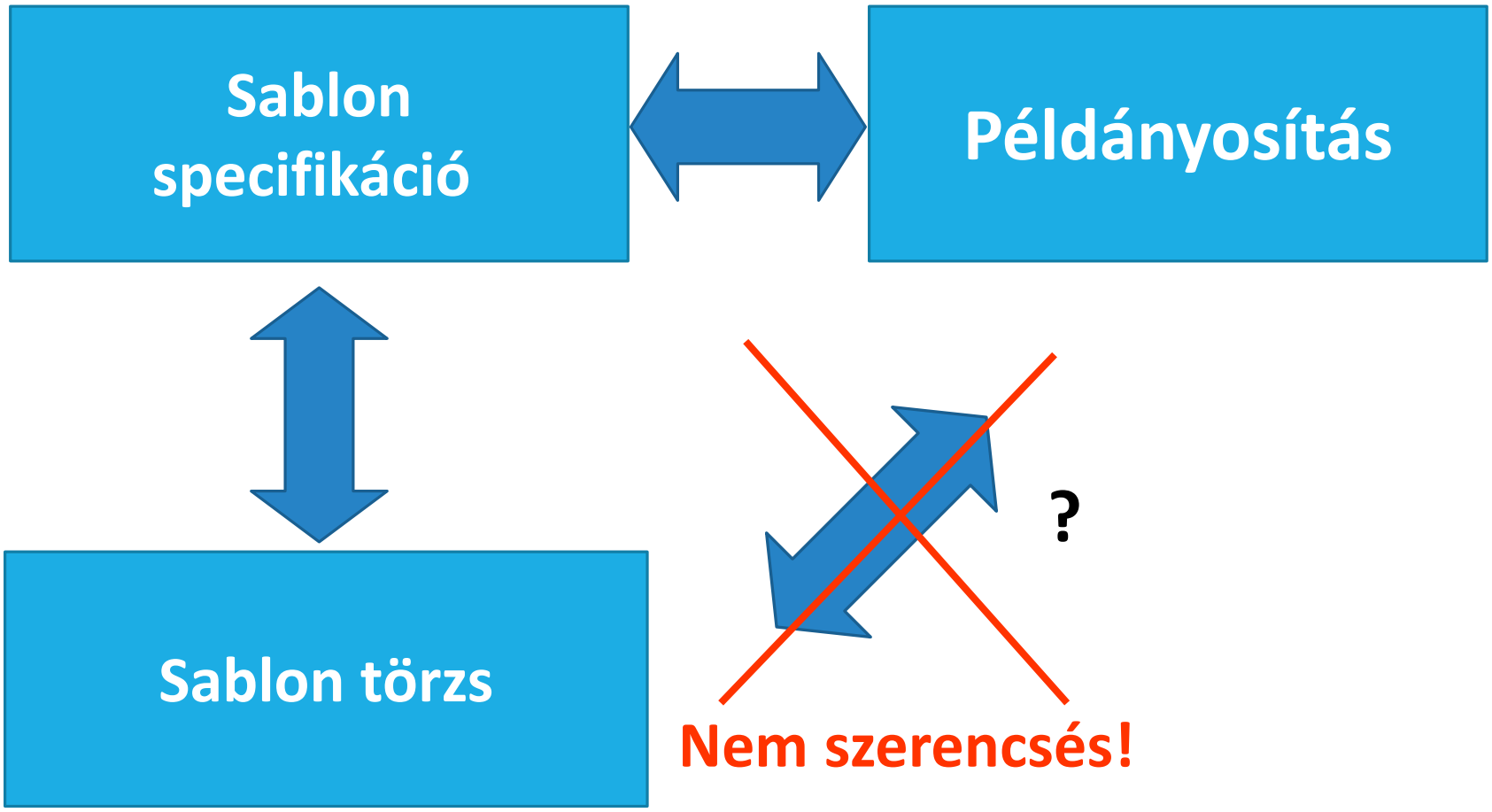
Fa

Keresőfa

Hash-tábla, stb.

Mi a különbség a példák között?

# Példányosítás – összefüggések



Ezen igény kielégítésére többféle eszköz kínálkozik.

- A C nyelvben például nem áll rendelkezésre ilyen eszköz nyelvi szinten
  - hasonló hatások – korlátozott mértékben – elérhetők az előfeldolgozó rendszer használatával.
- Az előfeldolgozó rendszer szerves része a C nyelv specifikációjának
  - Nem tekinthető igazán nyelvi eszköznek, mivel a forrásnyelvű programot manipulálja
  - Közelebb áll a megíráshoz használt szövegszerkesztőhöz.

```
typedef struct cell (ELEMENT data; struct cell *next) CELL;
typedef CELL *LIST;

int find_item(ELEMENT find, LIST head, LIST *scan, LIST *prior)
{
        scan=head;
        prior=NULL;

        while (GREATHAN(find, (*scan)->data)

        {

                *prior=*scan;

                *scan=(*scan)->next;

        };

        return EQUALS(find, (*scan)->data);

}
```

Ez a megoldás független az aktuális ELEMENT típustól

El kell készíteni a megfelelő makrókat, és működik

```
/* alfabetikus eset */
#ifdef ALPHA
#define ELEMENT char*
#define LESSTHAN(x,y) (strcmp((x),(y))>0)
#define GREATHAN(x,y) (strcmp((x),(y))<0)
#define EQUALS(x,y) (strcmp((x),(y))==0)
#define ASSIGN(x,y) (strcpy((x),(y)))
#define MAXVALUE '\377'

/* short integer használata*/
#elif SHORT
#define ELEMENT short int
#define LESSTHAN(x,y) ((x)<(y))
#define GREATHAN(x,y) ((x)>(y))
#define EQUALS(x,y) ((x)==(y))
#define ASSIGN(x,y) ((x)=(y))
#define MAXVALUE 0x7fff
```

# CLU

```
generic_proc_name = proc [ t : type ] (...)
                                returns (...) signals (...)
        where t has op1 : proctype (...) returns (...)
        where t has op2 : proctype (...) returns (...)
          ...
        where t has opk : proctype (...) returns (...)

sort = proc[ t : type ] ( A : array[ t ] )
                                returns ( array[ t ] )
   where t has lt : proctype (t, t) returns (bool)
```

# OOP megoldás

Objektumorientált programozási nyelvekben a hatás elvileg elérhető kizárólag az öröklődés és a polimorfizmus használatával

- A szükséges közös tulajdonságokat össze lehetne fogni egyetlen absztrakt osztályba – vagy interfészbe –
- Ez közös őse lehetne az összes, a feltételeknek eleget tevő osztálynak.

A megoldás problémája, hogy

- a nyelv alapvető osztályhierarchiájának kialakításakor ismerni kellene az összes számításba jövő feltételkombinációt

# Típussal paraméterezés

A típussal paraméterezést támogató nyelvek általában valamilyen önálló konstrukciót vezetnek be.

Ezekben a struktúrákban megadhatunk formális paraméterként típusokat, amelyeket aztán más típusokhoz hasonlóan használhatunk a struktúra belsejében.

Amikor a formális paramétereknek aktuális értéket adva létrehozzuk a struktúra megadott típusokhoz tartozó változatát, akkor a struktúra példányosításáról beszélünk.

# C++

```
#define MAX(a,b) a > b ? a : b
MAX( x, y )*2 --> x > y ? x : y*2
#define MAX(a,b) ((a) > (b) ? (a) : (b))
MAX( ++x, y ) --> ++x > y ? ++x : y

void swap( double& x, double& y ) {
  double temp = x;
        // !! Hogy határozzuk meg x típusát?
  x = y;
  y = temp;
}
```

# C++

A C++ nyelvben a típussal paraméterezés lehetőségét a template-ek biztosítják.

- Több, mint az előző makróhelyettesítés, ahol a fordító a megadott aktuális típussal helyettesíti a hozzá tartozó formális paraméter minden előfordulását.
- Bizonyos minimális szintaxisellenőrzést végez a fordító, de a formális paraméternél csak annyit tudok meghatározni, hogy az adott paraméter egy típust jelöl.

# C++

Nem tudom előírni a megvalósítandó alprogram vagy típus számára fontos tulajdonságok meglétét

- Bizonyos, szükséges műveletek
- Az ebből származó hibák így a példányosításkor jelentkeznek.
- Előnye a nyelvnek, hogy a template példányosítása teljesen automatikusan történik.

# C++

```
template <typename T>
void swap( T& x, T& y) {
   T temp = x;
   x = y;
   y = temp;
}
```

# C++

```
template <class T>
T max(T x, T y){
     return (x>y) ? x : y;
}
```

Ennek a használata:

```
int i;
Own_typ a,b;
int j=max(0,i);       // T –t helyettesíti "int"-tel
Own_typ m=max(a,b); // Own_typ -pal
```

# C++

```
double x, y = 5.1, z = 3.14;
x = max(y, z);
int i = 3, j = 4, k;
double x = 3.14, y = 4.14, z;
const int ci = 6;
k = max(i, j);  // max(int, int)
z = max(x, y);  // max(double, double)
k = max(i, ci); // max(int, int) - triviális
z = max(i, x);  // Nem egyértelmű!
```

# C++

```
template <class T, class S>
T max( T a, S b) {
  if ( a > b ) return a;
  else return b;
}
int i = 3;
double x = 3.14;
double z = max( i, x);   // Ez ok, megtörténik
z == 3.0;              // A konverziók miatt a 3.14
                       //  az 3.0 lesz
```

# Explicit specializáció

```
template <class R, class T, class S>
R max(T a, S b) {
  if ( a > b ) return a;
  else return b;
}
z = max<double>(i, x);   // ok, 3.14
k = max<long, long, int>(i, x);
                // Konvertál long és int -re
k = max<int, int, int>(i, j);
                // Fölösleges
```

# C++

```
#include <iostream.h>
template <class T> class Vector {
     T* data;
     int size;
public:
     Vector(int);
     ~Vector() { delete[] data; };
     T& operator[](int i) { return data[i]; }
};
template <class T> Vector<T>::Vector(int n) {
     data=new T[n];
     size=n;
};
```

# C++

```
main(){
   Vector <int> x(5);
   for (int i=0; i<5;++i) {
       x[i]=i;
   }
   for (i=0; i<5;++i) {
      cout  << x[i] << ' ';
   }
   cout << '\n';
}
```

# C++ concept

A concept-ek lehetősége

- megfogalmazhatjuk elvárásainkat egy típussal szemben
- ezt még példányosítás előtt tud ellenőrizni a fordítóprogram
- meg tudja nevezni a hiba okát olvasható formátumban

C++20-ban bevezetésre került!

# C++ concept

```
template<typename T>
concept EqualityComparable =
  requires(T a, T b) {
    { a == b } -> bool;
    { a != b } -> bool;
};
```

Megjegyzés

- draft cpp szabvány esetén (8.x gcc-vel pl) a szintaxis eltérő lehet több esetben
- Itt: `concept bool EqualityComparable =`

# C++ concept

```
template<EqualitiyComparable T>
void equals(T x, T y) {
    if (x==y)
        std::cout << "Azonosak";
    else
        std::cout << "Nem azonosak";
}

equals<int>(42, 42); // "Azonosak"

equals<int>(42, 10); // "Nem azonosak"
```

# C++ concept

```
class ownClass {
public:
    int member;
    ownClass(int init): member(init) {};
};

ownClass o1(10), o2(20);
equals<ownClass>(o1, o2); // HIBA!!!
```

No matching function for call to 'equals' candidate template ignored: constraints not satisfied [with T = ownClass] because 'ownClass' does not satisfy 'EqualitiyComparable' because 'a == b' would be invalid: invalid operands to binary expression ('ownClass' and 'ownClass')

# C++ concept

```
class ownClass {
public:
    int member;
    ownClass(int init): member(init) {};

    bool operator == (ownClass const &other) {
        return member == other.member;
    }

    bool operator != (ownClass const &other) {
        return member != other.member;
    }
};
```

# C++ concept

További megadási lehetőség

```
template<typename T> requires EqualitiyComparable<T>
void equals(T x, T y) …
```

Sőt, akár

```
template<typename T> requires !EqualitiyComparable<T>
```

Követelmények együttes megadása is lehetséges

```
template<typename T>
requires floating_point<T> && integral<T>;


template<typename T>
requires floating_point<T> || integral<T>;
```

# C++ concept

Előzőekhez az összetevők:

```
template < class T >
concept integral = std::is_integral_v<T>;

template < class T >
concept floating_point = std::is_floating_point_v<T>;
```

Egymásba ágyazással:

```
template<typename T>
concept MyConcept = requires(T a) {
    requires floating_point<T>;
    requires integral<T>;
};
```

# Eiffel: generic

Objektum orientált megközelítés.

A formális paraméterei osztályok lehetnek ahol a szükséges speciális tulajdonságokat azáltal lehet meghatározni, hogy megadható, mely osztályból kell származnia az aktuális paraméternek.

- Előnye: A generic belsejében használt műveletek a megadott ősosztály műveletei lehetnek.
  - A használat helyessége a generic megírásakor ellenőrizhető.
- Hátránya:  a szükséges közös tulajdonságokat jóval előre tudni kell, hogy a megfelelő közös ős definiálható legyen.
  - Ekkor viszont a problémák jórészt megoldhatóak az öröklődés és a polimorfizmus eszközeivel.

# Eiffel

Megszorítás nélküli generic

```
deferred class TREE [G] ...
class LINKED_LIST [G] ...
class ARRAY [G] ...
```

Az osztályoknak akárhány formális generic paramétere lehet.

Típus létrehozásához egy generic osztályból:

- Minden formális generic paraméterhez kell egy típus, az aktuális generic paraméter.
- Ez egy generikusan származtatott (példányosított) típust eredményez.

# Eiffel

Generic származtatások - példányosítás

```
TREE [INTEGER]
TREE [PARAGRAPH]
LINKED_LIST [PARAGRAPH]
TREE [TREE [TREE [PARAGRAPH]]]
ARRAY [LINKED LIST [TREE [LINKED LIST
[PARAGRAPH]]]
```

# Eiffel

## Korlátozott generic

- `class HASH TABLE [G, KEY -> HASHABLE] ...`
  - KEY –t megszorítjuk a HASHABLE osztállyal.
  - Minden T aktuális generic paraméter bázisosztálya, amit a KEY-hez használunk a HASHABLE megszorító osztály leszármazottja kell legyen.
- `HASH TABLE [PARAGRAPH, STRING]`
- A Korlátozott formális generic paraméterek általános szintaxisa:
  - `T -> Class-type`

# Ada

A sablonok paraméterezhetősége sokkal szélesebb körű és rugalmasabb

- Lehet:
  - megadott őstípusból származó típussal paraméterezni,
  - kiköthető, hogy az aktuális paraméter valamilyen típusosztályba (diszkrét típus, felsorolási típus, tetszőleges típus stb.) tartozzon.
  - megadhatók a használni kívánt további műveletek, amelyek segítségével kikerülhető a kötelező közös ős megléte.
  - az explicit megadott műveletek segítségével a generic még rugalmasabb használatára nyílik lehetőség, megtehető például, hogy egy rendezési relációt használó sablont az egész számokkal példányosítva nem a szokásos rendezési relációt adjuk meg, hanem például az oszthatóság parciális rendezési relációját.

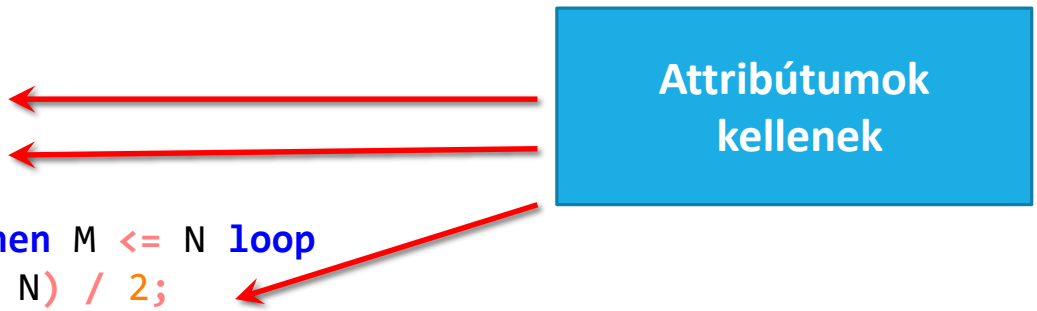
# Ada

```
generic
      type Item is private;
      type Index is (<>);
      type Vector is array (Index range <>) of Item;
      with function "<"(X, Y : Item) return Boolean is <> ;

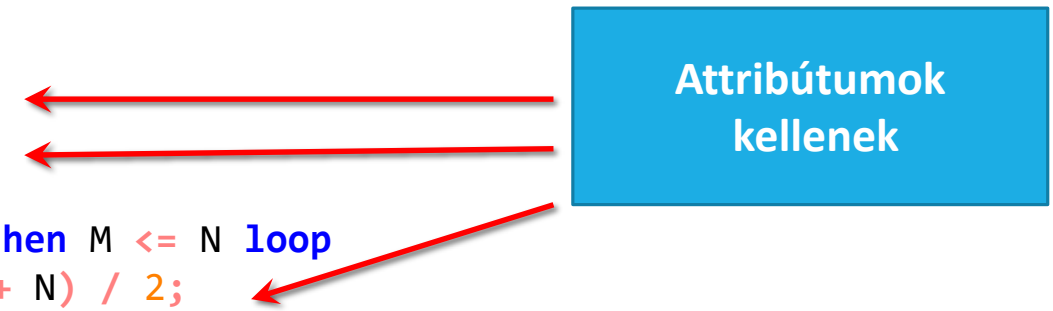
procedure Log_Search(V: in Vector; X: in Item;
                     Found: out Boolean; Ind: out Index);
```

```
procedure Log_Search(V:in Vector; X:in Item Found: out Boolean; Ind:out Index) is
    M, N, K : Integer;
    begin
        M := Index'Pos(V'First);
        N := Index'Pos(V'Last);
        Found := False;
            while not  Found and then M <= N loop
                        K := (M + N) / 2;
                        if  X < V(Index'Val(K)) then
                                N := K - 1;
                        elsif X = V(Index'Val(K)) then
                                Ind := Index'Val(K);
                                Found := True;
                        else
                                M := K + 1;
                        end if;
                end loop;
    end Log_Search;
```

```
procedure Log_Search(V:in Vector; X:in Item Found: out Boolean; Ind:out Index) is
    M, N, K : Integer;
    begin
        M := Index'Pos(V'First);
        N := Index'Pos(V'Last);
        Found := False;
            while not  Found and then M <= N loop
                        K := (M + N) / 2;
                        if  X < V(Index'Val(K)) then
                                N := K - 1;
                        elsif X = V(Index'Val(K)) then
                                Ind := Index'Val(K);
                                Found := True;
                        else
                                M := K + 1;
                        end if;
                end loop;
    end Log_Search;
```



```
procedure Log_Search(V:in Vector; X:in Item Found: out Boolean; Ind:out Index) is
    M, N, K : Integer;
    begin
        M := Index'Pos(V'First);
        N := Index'Pos(V'Last);
        Found := False;
            while not  Found and then M <= N loop
                        K := (M + N) / 2;
                        if  X < V(Index'Val(K)) then
                                N := K - 1;
                        elsif X = V(Index'Val(K)) then
                                Ind := Index'Val(K);
                                Found := True;
                        else
                                M := K + 1;
                        end if;
                end loop;
    end Log_Search;
```



# Java

## Generic bevezetése

```
List<Integer> myIntList =
                              new LinkedList<Integer>();
myIntList.add(new Integer(0));
Integer x = myIntList.iterator().next();
```

# Java

Ehhez a java.util-ban példa

```
public interface List <E> extends Collection<E>
{
    void add(E x);
    Iterator<E> iterator();
}

public interface Iterator<E>{
    E next();
    boolean hasNext();
}
```

# Java

```
public class Box<T> {
     private T t;                          // T: Típus
     public void add(T t) { this.t = t; }
     public T get() { return t; }
}
Box<Integer> integerBox;
integerBox = new Box<Integer>();
```

Java 7-től:

```
Box<Integer> integerBox = new Box<>();
```

- is engedélyezett, ha a fordító ki tudja következtetni

# Java

A használata:

```
public class BoxDemo3 {
  public static void main(String[] args) {
    Box<Integer> integerBox = new Box<Integer>();
    integerBox.add(new Integer(10));
    Integer someInteger = integerBox.get();
    System.out.println(someInteger);
  }
}
```

# Java

generic metódusok lehetősége is megvan:

```
public class Box<T> {
    private T t;
    public void add(T t) { this.t = t; }
    public T get() { return t; }
    public <U> void inspect(U u){
        System.out.println("T: " + t.getClass().getName());
        System.out.println("U: " + u.getClass().getName());
    }

    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();
        integerBox.add(new Integer(10));
        integerBox.inspect("some text");
    }
}
```

# Java

Az eredmény:

- `T: java.lang.Integer`
- `U: java.lang.String`

# Java

Típustörlés – nyers (raw) típusok

- `Box rawBox = new Box();`
  - ez is lehetséges – ezzel a generic előtti viselkedést kapjuk…
- `Box` a nyers típusa a `Box<T>`-nek

A korábbi programok így gond nélkül működhetnek

Futási időben minden generic `Object`-tel paraméterezett lesz

- Az ellenőrzés fordításkor történik

# Java

Megszorított típusparaméter lehetősége:

- `<U extends Valami>`
  és akkor az aktuális típusparaméter U-ra csak a Valami leszármazottja lehet
- vagy, ha interfész megvalósítását is kérjük:
  - `<U extends Valami & MyInterface>`
- A  generic törzsében számíthatunk a Valami és a MyInterface műveleteire!
- Mindig először kell az osztály, azután az interfészek

Öröklődéssel való kapcsolata később, az OOP-nél!

# Java

Példa:

```
public class Gen1<A extends Number> {
  public Gen1(A aa) { a = aa; }
  public A getElem() { return a; }
  public void setElem(A aa) { a=aa; }
  public int egesz() { return a.intValue(); }
  protected A a;
}
Gen1<Double> g2 = new Gen1 <Double>(2.3);
g2.egesz();
```

# Java – Típushelyettesítő

Tételezzük fel, hogy

```
public class Gen<T> {
  public Gen(T st) {storage = st; }
  public T getElem() { return storage; }
  public void setElem(T st) { storage = st; }
  protected T storage;
}
```

És készítünk egy f fv-t:

```
static <A,B> Gen f(Gen<A> ga, Gen<B> gb) {
  Object o1 = ga.getElem();
  Object o2 = gb.getElem();
  if (o1.toString().length() < o2.toString().length() )
     return ga;
  else
     return gb;
}
```

Mi legyen a visszatérési típusa?

# Java – Típushelyettesítő

## Probléma:

```
Gen<> g = f(new Gen<Integer>(1), new Gen<Double>(2.0) );
```

- ez fordítási hibás!

```
Gen g = f( new Gen<Integer>(1), new Gen<Double>(2.0) );
```

- Ok, de ez „nyers” (raw) típus…

```
Gen<Number> gi = f(new Gen<Integer>(1),
                                        new Gen<Double>(2.0) );
```

- ez rendben, de warning

## Típushelyettesítő

```
Gen<?> g1 = f( new Gen<Integer>(1), new Gen<Double>(2.0));
```

# Java – Típushelyettesítő

Még jobb, ha már a definíciójába tesszük:

```
static Gen<? extends Number> max(
                                Gen<? extends Number> ga,
                                Gen<? extends Number> gb) {
  Number n1 = ga.getValue();
  Number n2 = gb.getValue();
  if ( n1.doubleValue() < n2.doubleValue() )
    return ga;
  else
    return gb;
}
```

# Java további lehetőségek

Nemcsak felső megszorítás adható

Vegyük az alábbi függvényt

```
public void append(
            Collection<Integer> integers, int n) {
    for (int i = 1; i <= n; i++) {
        integers.add(i);
    }
}
```

A következő listát hogyan tudjuk paraméterként átadni?

```
List<Number> numbers = new ArrayList<Number>();
```

# Java további lehetőségek

A felső megszorítás értelemszerűen ekkor nem jó.

```
public void append(
    Collection<? super Integer> integers, int n) {
    for (int i = 1; i <= n; i++) {
        integers.add(i);
    }
}
```

```
List<Number> numbers = new ArrayList<Number>();
append(numbers, 5);
```

# Java generic és a new

A kapott típus szerinti objektumok létrehozása – probléma

```
class Dummy<T> {
    public T getInstance() throws Exception {
        new T();
        T.newInstance();
        T.class.newInstance();
    }
}
```

Egyik sem lehetséges

- Az megfejtés abban rejlik, hogy a típustörlést követően Object került a T helyére és nem az aktuális típusparaméter
  - Tehát a formális ellenőrzésen átmegy, de a törlést követően más a viselkedése

# Java generic és a new

Generic tömbök létrehozása – nem lehetséges

```
T[] ts = new T[n]
```

Ennek a megfejtése más

- A Java tömbök kovariánsak (erről még lesz szó). Lehetséges:

```
Integer[] myInts = {1,2,3,4};
Number[] myNumber = myInts;
```

  - A tömbökhöz futásidőben elérhető típusinformáció tartozik, a fenti esetben a `myNumber` változó mögötti tömbbe nem lehetséges a `4.5` érték berakása. Ezt futásidőben ellenőrzi és megakadályozza a VM. (Ez rendben is van, a tömbök definícióját teljesíti.)
  - Generikus tömb esetén azonban a típusinformáció a típustörlés miatt nem áll rendelkezésre, azaz `Object` lesz minden. Így a fenti típusellenőrzés nem lehetséges futásidőben, ami baj.

# C#

Sok szempontból hasonlít a Java megoldásaira

- Szintaktika
- Megszorítások

Különbség azonban, hogy C# esetén nincsen típustörlés

- E tekintetben a C++ template-hez hasonló a működés
- Minden típusparaméter-értékhez „natív” kódot generál

# C#

```
public class LinkedList<TKey, TData> where TKey : IComparable
{
    …
        public bool Find(TKey key)   {
            Node<TKey,TData> current = _head;
            while (current.NextNode != null)
            {
                if (current.Key.CompareTo(key) != 0)
                    current = current.NextNode;
                else
                    break;
            }
            return (current.Key.CompareTo(key) == 0);
        }
}
```

# Kérdések:

Milyen nyelvi elemekből tudunk sablonokat létrehozni?

- Új adattípusokból és/vagy alprogramokból?

Milyen paraméterei lehetnek ezeknek a mintáknak?

- Típusok, objektumok, alprogramok?
- Létrehozható paraméter nélküli sablon?

Milyenfajta típusok lehetnek aktuális típus-paraméterek?

- Csak beépített típusok és altípusaik, vagy felhasználó által definiált típusok is?

# Kérdések

Adhatunk megszorításokat a formális generic paramétereknek?

Egymásba ágyazható ez a konstrukció?

A példányosítás fordítási (statikusan) vagy futási időben (dinamikusan) történik?

# Kivételkezelés

# Program készítésének célja

A célunk jó minőségű program készítése

Alapvető elvárás

- helyes – a specifikációnak  megfelelően működik
- robosztus – nem várt helyzetekben sem történik katasztrófa

# Hibalehetőségek

A programok futása közben hibák léphetnek fel.

Hardverhibák

- elromlott winchester miatti leállások,
- elfogy a memória, stb.

# Hibalehetőségek

A programok futása közben hibák léphetnek fel.

Szoftverhibák

- a futtatott programban,
  - kísérlet nullával való osztásra
  - aritmetikai túlcsordulás, alulcsordulás
  - üres pointer/referencia dereferencia kísérlete
  - tömb hibás indexelése
  - hibás típus konverzió kísérlete
  - hibás input stb.
- az operációs rendszerben meglévő programozói hibák.

A program

megbízhatóságának növelése

érdekében

figyelni kell a hibákra

és meg kell próbálni

megelőzni,

vagy ha már bekövetkeztek,

kezelni

őket!

# Hibakezelés históriája

Hogyan kezelték régebben?

- Abortált a program
- Minden alprogram valamilyen jelzést (hibaértéket) ad vissza
  - A lefutás OK
  - A lefutás nem sikerült

# Példa

Tegyük fel, hogy szeretnénk beolvasni egy fájlból

```
ReadFile(f: in File; n1: out byte, n2: out integer,
                     n3: out longint, n4: out real);
begin
    ReadByte(f,n1);
    ReadInt(f,n2);
    ReadLongInt(f,n3);
    ReadReal(f,n4);
end ReadFile;
```

# Ellenőrzés lépésenként

## Ha ellenőrizni akarjuk:

```
ReadFile(f: in File; n1: out byte, n2: out integer,
                     n3: out longint, n4: out real): boolean;
success:boolean;
begin

  success:=ReadByte(f,n1);
    if success then success:=ReadInt(f,n2);
      if success then success:=ReadLongInt(f,n3);
         if success then success:=ReadReal(f,n4);

            if not success then
              // ReadReal hiba kezelése,
            end if;

         else … ReadLongInt hiba kezelése end if ;
      else … ReadInt hiba kezelése end if ;
    else … ReadByte hiba kezelése end if ;

  return success;

end ReadFile;
```

# Meggondolások

Nem biztos, hogy függvényt lehet/érdemes csinálni az eljárásból

- Lehet, hogy nem visszatérési érték, hanem egy paraméter jelzi a sikert.

A kód sokkal bonyolultabb, az eredeti cél szinte elvész

- A későbbi karbantartás sokkal nehezebb!

Nem biztos, hogy a hívó figyeli a sikert jelző értéket!

# Meggondolások

Lehetne, hogy globális változót használunk a lefutás helyességének figyelésére

- Osztott környezetben beláthatatlan következmények!

Hiba esetén meghívunk egy hibakezelő alprogramot

- Mi történik, ha nem tesszük meg?

# Elvárások

A hibák kezelése kapcsán a következő elvárásokat lehet megfogalmazni

1. meg tudjuk különböztetni a hibákat,
2. a hibát kezelő kód különüljön el a tényleges kódtól,
3. megfelelően tudjuk kezelni - a hiba könnyen jusson el arra a helyre, ahol azt kezelni kell,
4. kötelező legyen kezelni a hibákat

**KIVÉTELKEZELÉS**

# Megoldás elemzése

Az első kikötés, hogy meg tudjuk különböztetni a hibákat.

- Ez általában a fellépés helyén történik.

A második kikötés azt szeretné elérni, hogy a programot először látó ember is azonnal el tudja különíteni a hibakezelő és a működésért felelős részeket.

- Ez könnyebben továbbfejleszthető, karbantartható programokhoz vezet.

# Példa

```
gyumolcs {
    alma();
    //...
    korte();
    //...
    barack();
}
```

# Példa hagyományos módon

```
gyumolcs {
   ret1=alma();
   if (ret1==ok) {
      //...
      ret2=korte();
      if (ret2==ok) {
         //...
         ret3=barack();
         if (ret3==ok) {
            return ok;
         }
         else { /* 3. hiba kezelése */}
      }
      else { /* 2. hiba kezelése */ }
   }
   else {/* 1. hiba kezelése */ }
}
```

# A példa kivételkezeléssel

```
class AlmaException extends Exception {}
class KorteException extends Exception {}
class BarackException extends Exception {}


gyumolcs {
   try {
      alma(); /*küldhet AlmaException-t*/
      korte();
      barack();
   }
   catch(AlmaException hiba1) { /* 1. hiba kezelése */ }
   catch(KorteException hiba2) { /* 2. hiba kezelése */ }
   catch(BarackException hiba3) { /* 3. hiba kezelése */ }
}
```

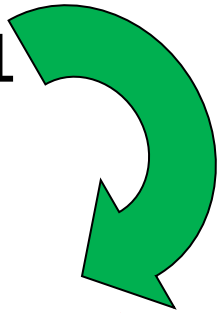
# A hiba jusson el oda, ahol kezelni kell

A harmadik kikötés arra vonatkozik, hogy egy alacsonyabb szinten jelentkező hibát nem biztos, hogy az adott szint le tud kezelni, ezért a megfelelő helyre kell eljuttatni.

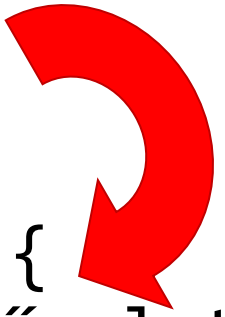
- A programok többszintűek
- Példa
  - egy fájlt nem tudunk megnyitni
  - egy üres veremből akar valaki kiolvasni

# Példa

```
gyumolcs {
     alma();
     //folytatás1
}
```



```
          alma {
               jonatan();
               //folytatás2
          }
```



```
                    jonatan {
                         //műveletek
                    }
```

# Hiba terjesztése hagyományosan

Tegyük fel, hogy a `jonatan()`-ban fellépő hibát a `gyumolcs()` kell lekezelje:

```
gyumolcs {
    ret=alma();
    if(ret!=ok) {
        //hibakezelés
    }
    else {
        //folytatás1
    }
}
```

```
alma {
    ret=jonatan();
    if(ret!=ok) {
        return hiba;
    }
    else {
        //folytatás2
    }
}
jonatan {
    //műveletek
    if( művelethiba ) {
        return hiba;
    }
}
```

# Hiba terjesztése kivételkezeléssel

Kivételkezeléssel a felfelé terjesztést egyszerűen és kevesebb módosítással lehet megoldani:

```
gyumolcs {
    try {
        alma();
        //folytatás
    }
    catch(VmiExc) {
        //hibakezelés
    }
}
```

```
jonatan throws VmiExc {
    //műveletek
}
```

```
alma throws VmiExc {
    jonatan();
    //folytatás
}
```

# Kivétel és nem hiba

```
type Napok is (Vasarnap, Hetfo, Kedd, … Szombat);

function Holnap(Ma: Napok) return Napok is
begin
   return Napok'Succ(Ma);
exception
   when Constraint_Error => return Napok'First;
end Holnap;
```

# Kérdések

Hiba- vagy kivételkezelést ad-e a nyelv?

Hogyan kell kivételeket definiálni-kiváltani-kezelni?

Milyen a kivételkezelés szintaktikai formája

Mihez kapcsolódik a kezelés: utasítás-, blokk- vagy eljárás/függvényszintű?

A kivételekből lehet-e kivételcsoportokat, kivételosztályokat képezni, amelyeket a kivételkezelőben egységesen lehet kezelni?

# Kérdések

Paraméterezhető-e a kivétel információ továbbadás céljából?

Támogatja-e a nyelv explicit módon valamilyen végső tevékenység („finally”) megadását?

- Ha nem, akkor tudjuk-e szimulálni azt?
- Milyen megszorításokkal?

A nyelv biztosít-e olyan nyelvi konstrukciót, amelynek nem kell explicit módon megadni, hogy mely kivételt kell továbbadni?

# Kérdések

Újrakezdhető-e az alprogram (blokk) a hiba kezelése után?

Megadható-e /meg kell-e adni, hogy egy alprogram milyen lekezeletlen  kivételeket küldhet?

Párhuzamos környezetben vannak-e speciális kivételek?

Mi történik a váratlan kivételekkel?

Kiváltható-e kivétel kivételkezelőben?

Melyik kivételkezelő kezeli a kivételt?

# C++

Kivétel lehet egy  objektum, aminek tetszőleges a típusa.

Hasznos, ha definiálunk osztályokat a felhasználó kivételeihez

Például

- `class MyException {};`

# C++

Néhány predefinit kivétel:

- `bad_cast`: ha a dynamic_cast operator nem használható egy referenciára,
- `bad_typeid`: ha a typeid operátora egy null pointer dereferenciája.
  (mindkettő a typeinfo.h-ban deklarálva).

# C++

## Kivétel kiváltása

- `throw [expression] ";"`
  A kifejezés egy időszakos objektumba kerül.
- A normál program lefutás megszakad, a vezérlés a legközelebbi megfelelő kezelőhöz kerül
- Ha nincs kifejezés: csak kivételkezelőben, illetve innen hívott függvényben lehet, az aktuális kivétel újrakiváltása.
- A fordító nem biztos, hogy ellenőrzi, ha `throw` utasítás kivétel-objektum nélkül: `terminate` függvény hívása.

# C++

Példák

- `throw 5;`
- `throw "An exception has occurred";`
- `throw MyException();`
- `throw;`

További információk: a kivételosztályban lehetnek nyilvános attribútumok, konstruktor beállíthatja stb.

```
class ExceptionWithParameters {
 public:
  int a,b;
  ExceptionWithParameters(int a0,int b0)
                                  { a=a0; b=b0; }
  };

  throw ExceptionWithParameters(0,1);
```

# C++

Kivételek kezelése - try blokkal:

- `try compound_statement handler_list`
  `ahol`
  `handler_list = handler{handler}`
- `handler = catch "(" exception_declaration ")"`
  `compound_statement`
  `exception_declaration = type [ident] |"..."`

# C++

A dobott kivételeknek megfelel egy `catch` ág, ha a következő feltételek közül valamelyik teljesül:

- A két típus pont ugyanaz.
- A `catch` ág típusa publikus bázisosztálya az eldobott objektumnak.
- A `catch` ág típusa mutató, és az eldobott objektum olyan mutató, melyet valamely standard mutató konverzióval át lehet konvertálni a `catch` ág típusára.

A kezeletlen kivételek továbbgyűrűződnek a hívó `try` blokkban, majd az azt hívóban.

- A legkülső `try` blokk után a *terminate* függvény kerül meghívásra.

# C++

Például

```
class A {};
class B: public A {};
class C: public B {};
class D: public A {};
class X {};
class AX: public A, public X {};

catch (A)          // A, B, C, D, AX típusúakat kap el
catch (A*)         // A*,B*,C*,D*,AX*
catch (X&)         // X és AX
catch (const B&) // B és C
catch (char*)      // char*
catch (void*)      // tetszőleges pointer típusút
```

# C++

Mindig az első kezelőt választja ki

- ha rossz sorrendet írunk, nem hiba!

```
try { // ...
  }catch (A) {/* ... */}
   catch (B) {/* ... */}//soha nem jön ide!

try {// ...
  }catch (B) { /* ... */ }
   catch (A) { /* ... */ } // így OK.
```

# C++

Lehet a kivételobjektumra is hivatkozni:

```
catch (ExceptionWithParameters e) {
    cout<< e.a; // kiírja
}
```

Lehet ...

- ez bárminek megfelel – utolsó legyen a `try` blokkban

A `terminate` hívása általában abort-ot jelent

- A felhasználó a `set_terminate`-tel megadhat mást is
- ez is le kell állítsa a programot.

Miközben a vezérlés átadódik a kivételkezelőnek, destruktort hív minden automatikus objektumra, ami a `try` blokkba történő belépés óta keletkezett.

# C++

Kivétel specifikációk

- `throw "(" [type {"," type}] ")"`

Például: `void f(int x) throw (A,B,C);`

Az f függvény csak A,B és C típusú kivételt generál, vagy azok leszármazottait.

- `int f2(int x) throw ()` - f2 nem válthat ki kivételt!
- `void f3(int x)` - f3 bármit kiválthat!

Ha specifikáltunk lehetséges kivételtípusokat, akkor minden más esetben a rendszer meghívja az `unexpected()` függvényt

- Az alapértelmezett viselkedése a `terminate()` függvény meghívása
- Ez a `set_unexpected` segítségével szintén átállítható.

# C++

Példák:

```
class X {};
class Y: public X {};
class Z {};
void A() throw(X,Z) // A X,Y,Z-t dobhat
{ /* ... */ }
void B() throw(Y,Z) {
   A();      // unexpected, ha A X-t dob, ok Y, Z típusúra
}
void C();    // semmit sem tudunk C-ről
void D() throw() {
   C();      // ha C bármit kivált, unexpected -t hív
   throw 7; // unexpected-t hív
}
```

# Java

A C++ szintaxistól nem sokban különbözik.

- Eltérések a szemantikában.

```
try {
     throw new EgyException ("parameter");
}
catch (Type variable) {

}
finally {

}
```

# Java

A továbbiakban csak az eltérések:

- **`finally`** nyelvi konstrukció, ezzel a Java megbízhatóbb programok írását segíti elő.

Egy függvény által kiváltható kivételek specifikálása:

- `void f (int x) throws AException, BException;`

Egy szálon futó program esetén ha a virtuális gép nem talál a kivétel kezelésére alkalmas kódot, akkor a VM és a program is terminál.

- Ha többszálú a program, akkor egy `uncaughtException()` metódus fut le.

# Java

Minden kivétel a `java.lang.Throwable` leszármazottja.

Ha olyan kivételt szeretnénk dobni, amely nem a `Throwable` leszármazottja, akkor az fordítási hibát okoz.

A kivételek két nagy csoportba sorolhatóak

- ellenőrzöttek: `Exception` leszármazottjai
- nem-ellenőrzöttek: `Error` leszármazottjai

# Java

Számos olyan kivétel van, ami

- előre nem látható
- És fölösleges lenne mindenhol lekezelni őket.
- Ezeket a kivételeket nevezzük nem-ellenőrzött kivételeknek.
  - Például nem ellenőrizzük le minden utasítás végrehajtása előtt, hogy van-e elég memóriánk stb.

Ez az oka a kivételek két csoportjának.

- A csoportosítás nem konzisztens
  - Az `Exception` egyik leszármazott osztálya, a `RuntimeException` és leszármazottjai sem ellenőrzöttek.

Az ellenőrzött kivételek esetén fordítási hiba lép fel, ha nincsenek specifikálva vagy elkapva.

- Továbbá ha olyan ellenőrzött kivételt kívánunk elkapni, amely hatókörön kívül van.

# Java

Néhány predefinit nem ellenőrzött kivétel

A `RuntimeException` leszármazottjai

- `ArithmeticException`
- `ClassCastException`
- `IndexOutOfBoundsException`
  - `ArrayIndexOutOfBoundsException`
  - `StringIndexOutOfBoundsException`
- `NullPointerException`.

Az `Error` leszármazottjai

- `OutOfMemoryError`
- `StackOverflowError`
- stb.

# Java példa

```
class VeremException extends Exception {}
class VeremMegteltException extends VeremException
{
      private int utolso;
      public VeremMegteltException (int i) {
            utolso = i;
      }
      public int miVolt () {
            return utolso;
      }
}
```

# Java példa

```
class Verem {
  final static public int MERET = 10;
  private int tarolo [] = new int[MERET];
  private int mutato = 0;
  public void betesz (int i) {
    try {
      if (mutato < MERET)
        tarolo [mutato++] = i;
      else
        throw new VeremMegteltException (i);
    } catch (VeremMegteltException e) {
      System.out.println( e.miVolt () + " nem fert be");
      throw;
    }
    finally {
      System.out.println ("finally: mindig lefutok");
    }
  }
}
```

Mi a gond ezzel?

# Java

Predefint kivételek előfordulása:

```
class A { }
class B extends A { }
class C {
   void X() {
      A a = new A;
      B b = (B)a; // ClassCastException
   }
   void Y() {
     int ia[] = new int[10];
     for (int i=1; i<=10; i++)
       ia[i]=0;  // amikor i==10 lesz: ArrayIndexOutOfBoundsException
   }
   void Z() {
     C c=null;
     c.X();       // NullPointerException
   }
}
```

# Java

Kivételek kezelése

```
try {
  // utasítások
}
catch (MyException e) {
  // utasítások MyException kezelésére
}
catch (AnotherException e) {
  // utasítások AnotherException kezelésére
}
finally {
  // mindig végrehajtódó utasítások
}
```

# Java

```
class A extends Exception {}
class B extends A {}

try {
    // ...
}
catch (A a) { /* ... */ }
catch (B b) { /* ... */ }

void MyMethod() throws MyException {
    // utasítások
    throw new MyException();
    // utasítások
}
```

# Java

```
class ExcA extends Exception {}
class ExcD extends RuntimeException {}
class ExcF extends ExcA {}

void A() throws ExcA {
    // ...
    throw new ExcA();
    // ...
}
```

# Java

```
void B1() {
    A(); // Hiba: ExcA nincs lekezelve B1-ben
}
void B2() {
  try {
     A(); // OK
  } catch (ExcA e) {
     // exception kezelése
  }
 }
void B3() throws ExcA {
     A(); // OK
}
```

# Java

```
void D() throws ExcD {
    // ...
}
void E() {
    D(); // ok, ExcD leszármazottja
         // RuntimeException-nak,
         // így nem kell jelezni
}
void F() throws ExcA {
      throw new ExcF();    // OK
}
```

# Java try-with-resources

További hasznos lehetőség a finally használata helyett:

```
try (BufferedReader br =
        new BufferedReader(new FileReader(path)))
{
    return br.readLine();
}
```

Ehhez a `Closeable` vagy `AutoCloseable` interfészt kell megvalósítania a használandó osztálynak

# OOP

## KÖVETKEZŐ HÉTEN




# Programozási nyelvek és módszerek

OOP – I. (ISMÉTLÉS)

# Kivételkezelés

# C# – .NET

Közös elv alapján a .NET-ben

Nagyon hasonlít a Javához, de van különbség is

Hasonlóság:

- `try` – `catch` blokk, (ellenőrzi a jó sorrendet)
- `finally` lehetősége
- közös ősosztály (`System.Exception`)

Különbség:

- Nincs exception-specifikáció

# C# – .NET

## Miért nincs a C#-ban ellenőrzött kivétel?

- Verziókezelés
  - Ha egy következő verzióban egy metódus új kivételt dob, akkor az őt hívó metódust is változtatni kell
  - Ez máshol is probléma, ha le akarjuk kezelni az új kivételt, de legtöbbször nem kezelik
  - 10 az 1-hez a `try`–`finally` és a `try`–`catch` aránya
- Méretezhetőség:
  - Nagy projektekben, négy-öt alrendszerrel a sok kivétel már kezelhetetlenné válik
  - Ilyenkor keletkezik sok `catch { }` üresen
- Nem szigorú szabályok kellenek a kezeletlen kivételek ellen, hanem elemző eszközök, amelyek felkutatják a gyanús kódokat, lehetséges réseket

# C# – .NET

```
using System;
class ExceptionTestClass {
  public static void Main() {
    int x = 0;
    try {
      int y = 100/x;
    }
    catch (ArithmeticException e) {
        Console.WriteLine("ArithmeticException: {0}", e.ToString());
    }
    catch (Exception e) {
        Console.WriteLine("Generic Exception: {0}", e.ToString());
    }
    finally {
        Console.WriteLine("It is always executed.");
    }
  }
}
```

# C# – .NET

Saját exception-osztály:

```
using System;
public class EmployeeListNotFoundException: ApplicationException {
  public EmployeeListNotFoundException() { }
  public EmployeeListNotFoundException(
           string message) : base(message) { }
  public EmployeeListNotFoundException(
           string message, Exception inner) :
           base(message, inner) { }
}
```

# Delphi

Eltérő szintaktikára példa

```
begin
  Try
    // The code we want to execute
    ...
  Except
    // Exception handling
    ...
  Finally
    // Finally block
    ...
  end;
end;
```

# Delphi

## Több kivételtípus kezelése

```
except
    // IO error
    On E : EInOutError do
      ShowMessage('IO error : '+E.Message);
    // Dibision by zero
    On E : EDivByZero do
      ShowMessage('Div by zero error : '+E.Message);
    // Catch other errors
    else
      ShowMessage('Unknown error');
 end;
```

# Delphi

## Kivétel kiváltása

- `raise object at address;`
- Példa

```
raise EMathError.Create;
raise Exception.Create at @MyFunction;
```

# Python

Előzőekhez hasonlóan

```
def divide(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("division by zero!")
    else:
        print("result is", result)
    finally:
        print("executing finally clause")
```

- Az **else** jelentősége, hogy lokális változókhoz hozzáfér, de a **try** által védett kódot már nem tartalmaz

# Python

Előredefiniált clean-up kódok lefuttatása

- A következő kód helyett

```
file = open('file_path', 'w')
try:
    file.write('hello world')
finally:
    file.close()
```

- Írható ez is (a try-with-resources mintájára)

```
with open('file_path', 'w') as file:
    file.write('hello world !')
```

# Python

A **with** használatára alkalmas osztály készíteni az __enter__ és __exit__ függvények definiálásával lehet

```
class MessageWriter(object):
    def __init__(self, file_name):
        self.file_name = file_name

    def __enter__(self):
        self.file = open(self.file_name, 'w')
        return self.file

    def __exit__(self):
        self.file.close()
```

# SWIFT

A kivételeket az `Error` protokollt megvalósító felsorolási típusok segítségével reprezentálja

- SWIFT esetén a protokoll megközelítőleg a Java interfész fogalomnak felel meg

Például

```
enum VendingMachineError: Error {
    case invalidSelection
    case insufficientFunds(coinsNeeded: Int)
    case outOfStock
}
```

# SWIFT

Az egyes esetek paraméterezhetők, amivel a kivétel specifikusabban megadható.

Kiváltani a **`throw`** utasítással lehet

- `throw VendingMachineError.insufficientFunds(coinsNeeded: 5)`

# SWIFT

## Kivételek kezelése

- Egyrészt, a Java-hoz hasonlóan egy függvénynek jeleznie kell, hogy kivételt dobhat
  - `func canThrowErrors() throws -> String`
  - `func cannotThrowErrors() -> String`
- Ha kritikus függvényt hívunk, akkor **`try`** kulcsszóval kell tenni
  - `try canThrowErrors()`

# SWIFT

A kapott kivételt vagy kezelni kell, vagy jelölni, hogy tovább tud dobódni.

Kivétel kezelés:

```
do {
    try buyFavoriteSnack()
} catch VendingMachineError.invalidSelection {

} catch VendingMachineError.outOfStock {

} catch VendingMachineError.insufficientFunds(let c) {

}
```

# SWIFT

## További lehetőségek

- Értékadásnál, ha nem sikerült, akkor null értéket ad az új változónak
- Ezt később lehet kezelni
  - Optional – formájában

```
let x = try? someThrowingFunction()
```

  - Összeköthető feltétel vizsgálattal is

```
func fetchData() -> Data? {
    if let data = try? fetchDataFromDisk()   { return data }
    if let data = try? fetchDataFromServer() { return data }
    return nil
}
```

# SWIFT

## További lehetőségek

- Amennyiben biztosak vagyunk, hogy a hívott függvény valóságban nem dobhat kivételt akkor

```
let photo = try! loadImage( )
```

- Ebben az esetben, ha mégis keletkezik kivétel, akkor a futás megszakad és egy futás idejű hiba keletkezik

# SWIFT

## Hibától függetlenül lefutó kód

```
func processFile(filename: String) throws {
    if exists(filename) {
        let file = open(filename)
        defer {
            close(file)
        }
        while let line = try file.readline() {
           // line
        }
        // close(file) hívása itt történik a scope végén
    }
}
```

- Tehát a **`defer`** részben megadott hívás a blokk befejeződése előtt lefut

# Eiffel

## Újrakezdés

- A komponens írásakor egy kivétel lehetőségét előre lehet látni és egy alternatív megoldást találni a szerződés betartatására.
- Ekkor a végrehajtás megpróbálja ezt az alternatívát.

## Szervezett pánik

- Ha nincs rá mód, hogy teljesítsük a szerződést, akkor az objektumokat egy elfogadható állapotba kell hozni (típusinvariáns helyreállítása), és a felhasználónak jelezni kell a kudarcot.

# Eiffel – Újrakezdés

```
try_once_or_twice is
  local
    already_tried : BOOLEAN
  do
    if not already_tried then
      method_1
    else
      method_2
    end
  rescue
    if not already_tried then
      already_tried := true;
      retry
    end
 end -- try_once_or_twice
```

# Példa

```
transmit: (p: PACKET)
-- Transmit packet `p'
require
      packet_not_void: p /= Void
local
      current_retries: INTEGER
      r: RANDOM_NUMBER_GENERATOR
do
      line.send (p)
rescue
      if current_retries < max_retries then
            r.next
            wait_millisecs (r.value_between (20, 500))
      current_retries := current_retries + 1
            retry
      end
end
```

# Eiffel – Szervezett pánik

```
attempt_transaction(arg : CONTEXT) is
  -- megpróbáljuk a transaction-t arg argumentummal;
  -- ha nem sikerül, az akt. obj. invariánsát visszaállítjuk
  require
     ...
  do
     ...
  ensure
     ...
  rescue
     reset(arg)
 end -- attempt_transaction
```

# Eiffel

```
default_rescue is
do
end --default_rescue

class C creation
  make ...
inherit
  ANY
    redefine default_rescue end
  ...
feature
  make, default_rescue is
     -- nincs előfeltétel
    do ...
    end;
  ...
end -- class C
```

# Kivételkezelés – érdekesség

A C++ nyelvet számos kritika éri, többek között a kivételkezelés kapcsán.

Eredményképpen számos Code-Style tiltja

- Google
- LLVM
- Qt

# Objektum Orientált Programozás

# Az OO paradigma

Mitől OO egy program?

Objektum

Osztály

Öröklődés

# A valós világ modellezése

## Az ember a világ megértéséhez modelleket épít

## Modellezési alapelvek

- Absztrakció
  - Szemléletmód, amelynek segítségével a valós világot leegyszerűsítjük
    - Csak a lényegre, a cél elérése érdekében feltétlenül szükséges részekre összpontosítunk
  - Elvonatkoztatunk a számunkra pillanatnyilag nem fontos, közömbös információktól
    - Kiemeljük az elengedhetetlen fontosságú részleteket.
- Megkülönböztetés
  - Az objektumok a modellezendő valós világ egy-egy önálló egységét jelölik.
  - Az objektumokat a számunkra lényeges tulajdonságaik, viselkedési módjuk alapján megkülönböztetjük.

# A valós világ modellezése

## Modellezési alapelvek

- Osztályozás
  - Az objektumokat kategóriákba, osztályokba
    - a hasonló tulajdonságokkal rendelkező objektumok egy osztályba kerülnek.
    - a különböző vagy eltérő tulajdonságokkal rendelkező objektumok külön osztályokba kerülnek.
  - Az objektum-osztályok hordozzák a hozzájuk tartozó objektumok jellemzőit, objektumok mintáinak tekinthetők.
- Általánosítás, specializálás
  - Az objektumok között állandóan hasonlóságokat vagy különbségeket keresünk
    - Ezáltal bővebb vagy szűkebb kategóriákba, osztályokba soroljuk őket.

# OOP – alapelvek

OOP Alapelvek (Benjamin C. Pierce)

- Dynamic binding (dinamikus kötés)
  - Egy objektum esetén dinamikusan, futási időben dől el, hogy egy metódus melyik implementációja kerül futtatásra
- Encapsulation (egységbe zárás)
  - Adatok és rajtuk végrehajtható műveletek egységet alkotnak
  - Praktikusan ez a modern típus-definícióval konzisztens
- Subtype polymorphism (altípusos polimorfizmus)
  - Egy rögzített típusú változó több a típus altípusának példányára is hivatkozhat
  - Altípus
    - Az eredeti típus megszorításával létrehozott új típus
- Inheritance, vagy delegation (öröklődés, delegáció)
  - Egy adott osztályból lehetőség van képezni egy másik osztályt
    - Ez az ős tulajdonságait megtartja
    - Azonban módosíthatja, bővítheti
- Open recursion (nyílt rekurzió)
  - Speciális változó, amely egy metódus esetén lehetővé teszi az aktuális példány elérését

# Objektum

Belső állapota van
- Ebben információt tárol
- Változókkal valósítjuk meg
  - Ezek az adattagok, vagy tulajdonságok.

Kérésre feladatokat hajt végre
- Metódusok – melyek hatására állapota megváltozhat

Üzeneteken keresztül lehet megszólítani
- Kommunikál más objektumokkal

Minden objektum egyértelműen azonosítható

# Osztály, példány

Osztály (class)

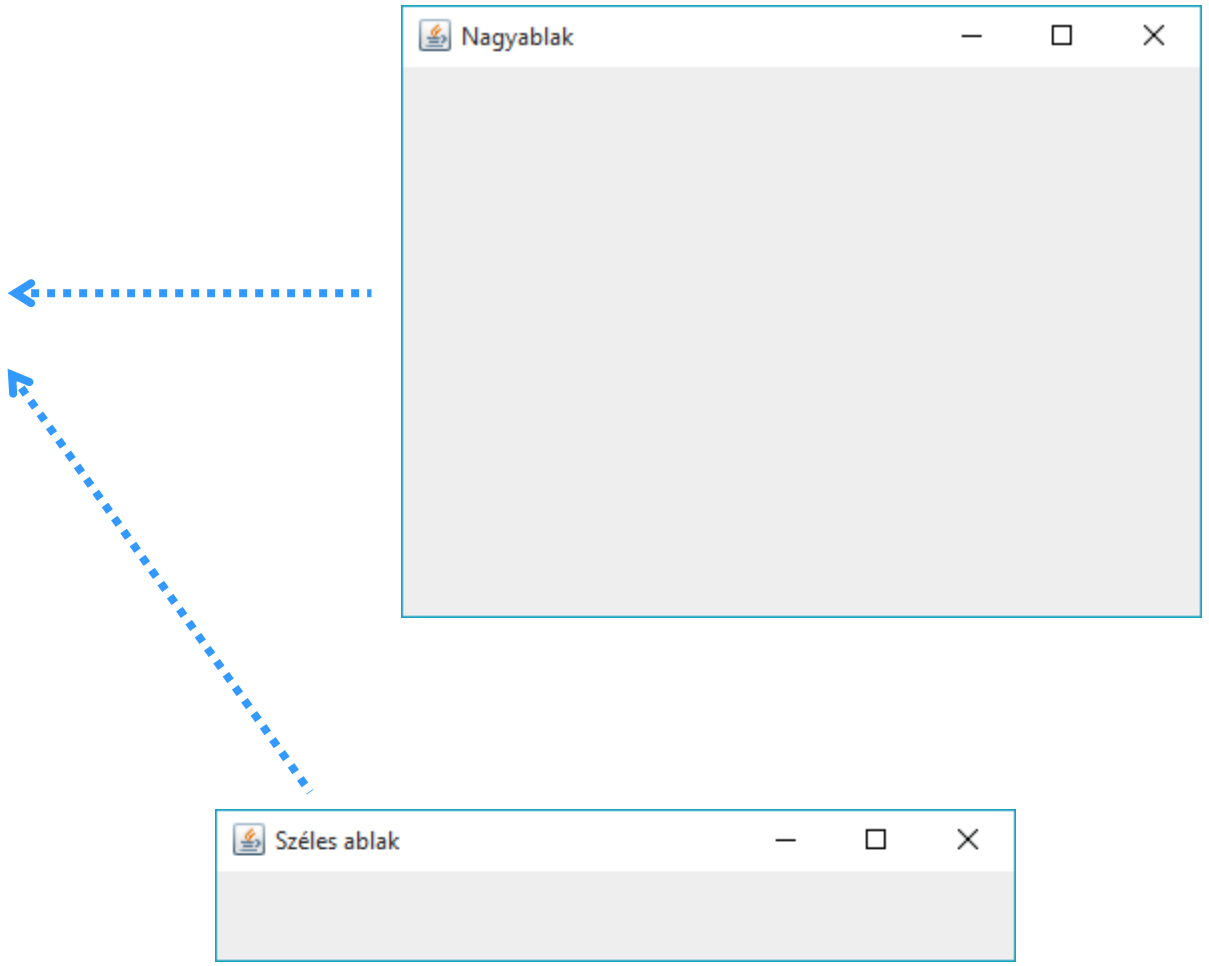
- Olyan objektumminta vagy típus
- Ez alapján példányokat (objektumokat) hozhatunk létre

Példány (instance)

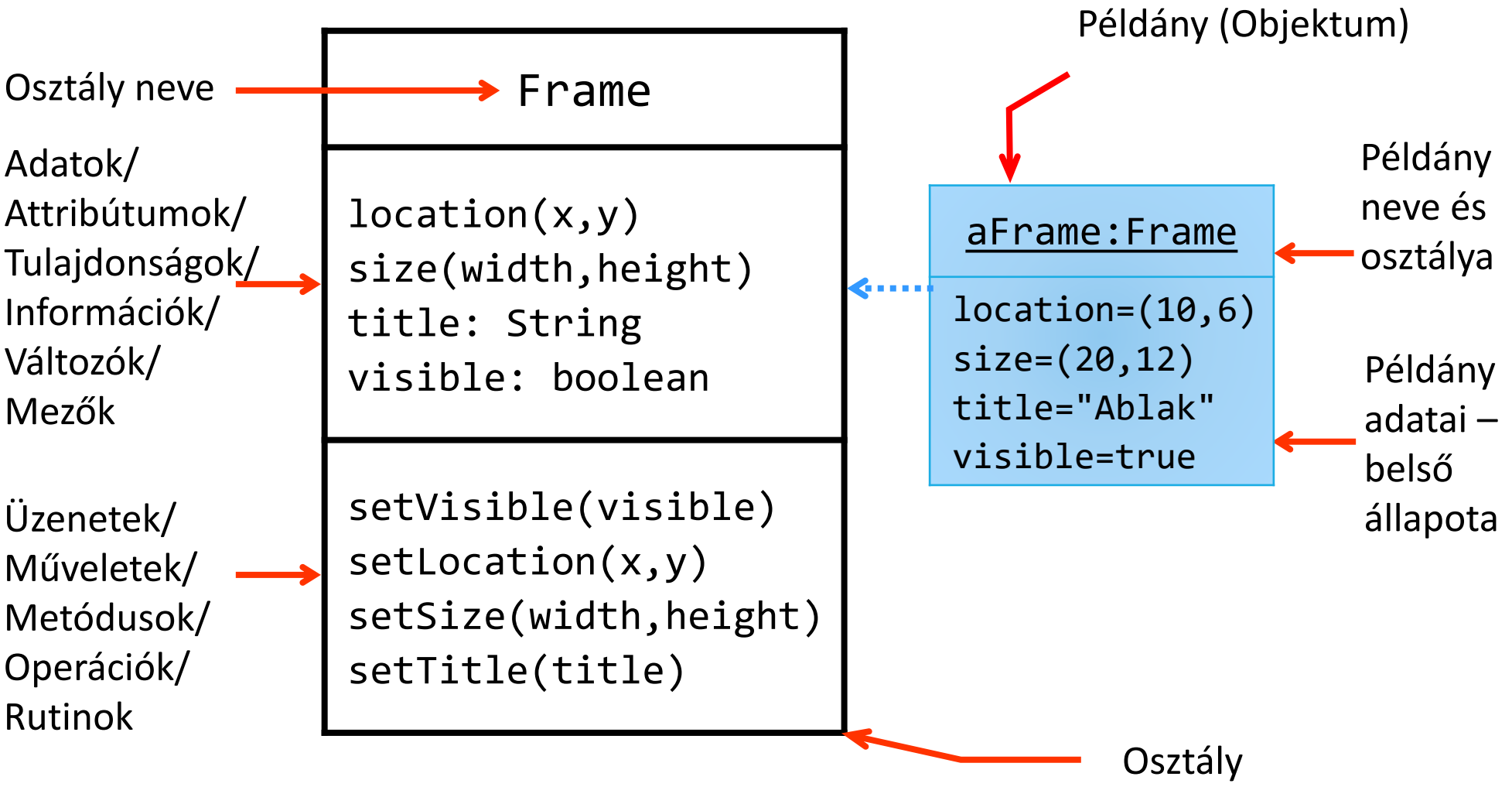
- Egy osztály (minta) alapján létrejött konkrét példány
- Minden objektum születésétől kezdve egy osztályhoz tartozik

# Frame osztály és példányai

| Frame |
| --- |
| location(x,y)<br>size(width,height)<br>title: String<br>visible: boolean |
| setVisible(visible)<br>setLocation(x,y)<br>setSize(width,height)<br>setTitle(title) |



# Osztály és példány az UML-ben



# Objektum életciklusa

Objektum életciklusa

- Létrejön – „megszületik”
- Állapotváltozásokon esik keresztül – „él”
- Megszűnik – „meghal”

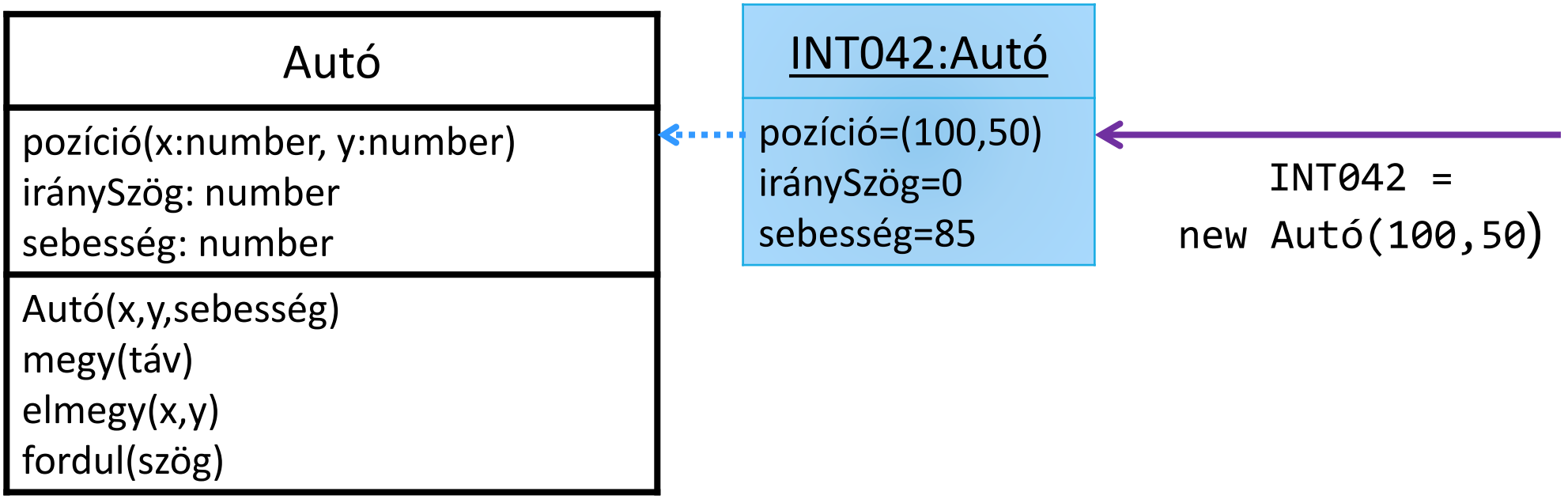
# Objektum létrehozása, inicializálása

Az objektumot létre kell hozni és inicializálni kell!

Objektum inicializálása

- Konstruktor (constructor) végzi
- Adatok kezdőértékadása
- Objektum működéséhez szükséges tevékenységek végrehajtása
- Típusinvariáns beállítása

# Objektum létrehozása, inicializálása



# Objektum állapotváltozása

Az objektum kérésre feladatokat hajt végre

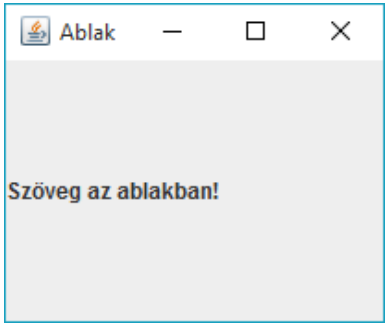
Életciklusa során üzeneteket fogad és feldolgozza

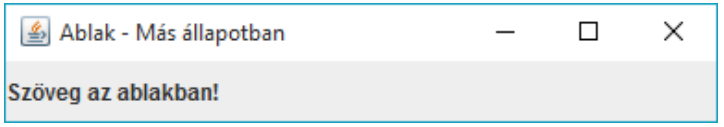
- Hatására az állapot megváltozik
- Ezek során üzeneteket küldhet más objektumoknak

# Állapotváltozás

Üzenetek

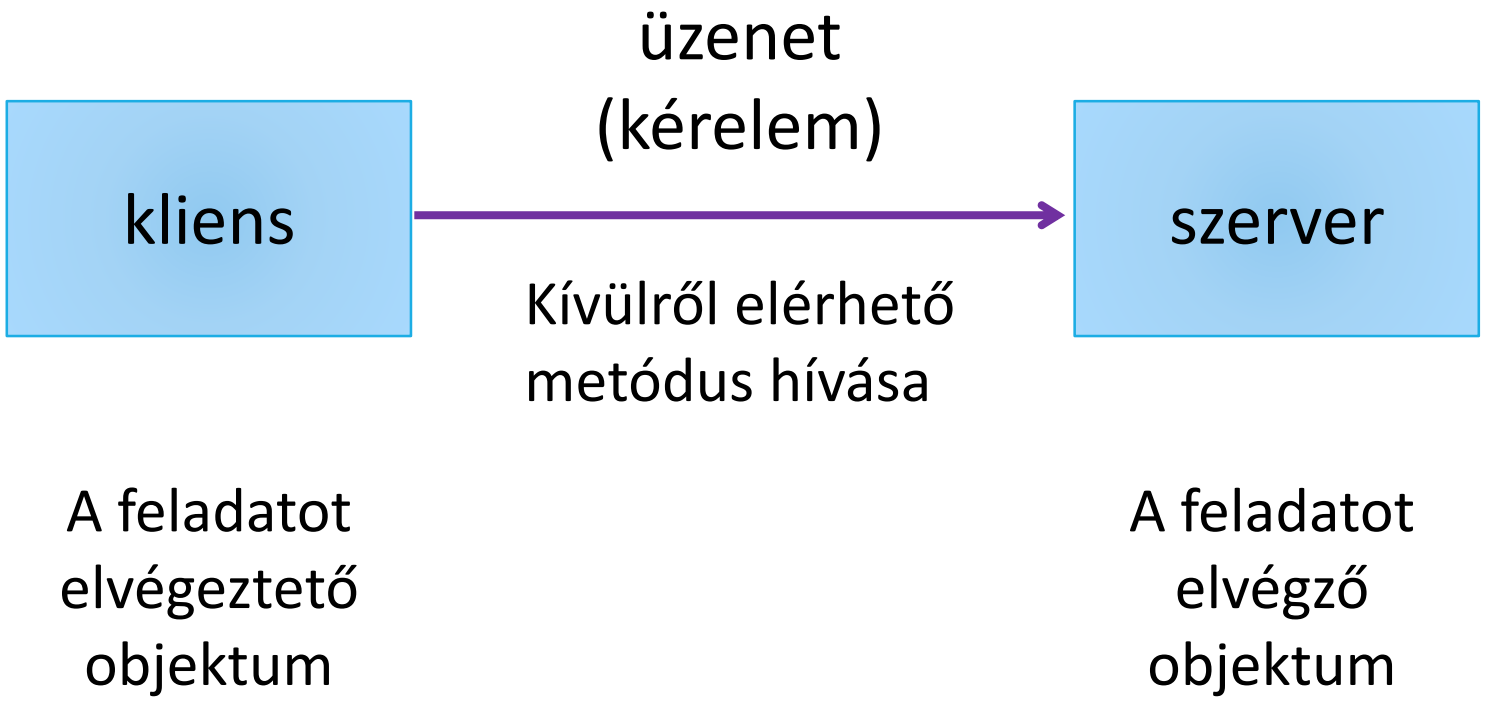
```
setVisible(true)
setLocation(40,8)
   setSize(20,16)
setTitle("Ablak")
```




| aFrame:Frame | |
|---|---|
| (20,16) | location(x,y) |
| (100,80) | size(width,height) |
| "Ablak" | title |
| true | visible |

# Üzenetküldési modell



# Üzenetküldési modell

Kliens

- Aktív objektum, másik objektumon végez műveleteket, de rajta nem végeznek
- Nincs export felülete
- Például óra (órajel)
  - Meghatározott időközönként művelet egy regiszteren

Szerver

- Passzív objektum
- Csak export felülete van
- Másoktól érkező üzenetekre vár, mások szolgáltatását nem igényli

Ágens

- Általános objektum, van export és import felülete

# Objektum műveletei

## Export műveletek

- Amelyeket más objektumok hívhatnak
- Például verem
  - `push`, `pop`, `top` stb.

## Import műveletek

- Amelyeket az objektum igényel ahhoz, hogy az export szolgáltatásait nyújtani tudja
- Például verem, ha fix méretű (vektoros) reprezentáció
  - Vektorműveletek

# Objektum műveletei

Export műveletek csoportosítása:

- Létrehozó (konstruktor)
  az objektum létrehozására, felépítésére
  - Példa veremnél
    - `create`: $\rightarrow$ Verem
- Állapot megváltoztató
  - Példa verem esetén
    - pop: Verem $\rightarrow$ Verem $\times$ Elem
    - `push`: Verem $\times$ Elem $\rightarrow$ Verem

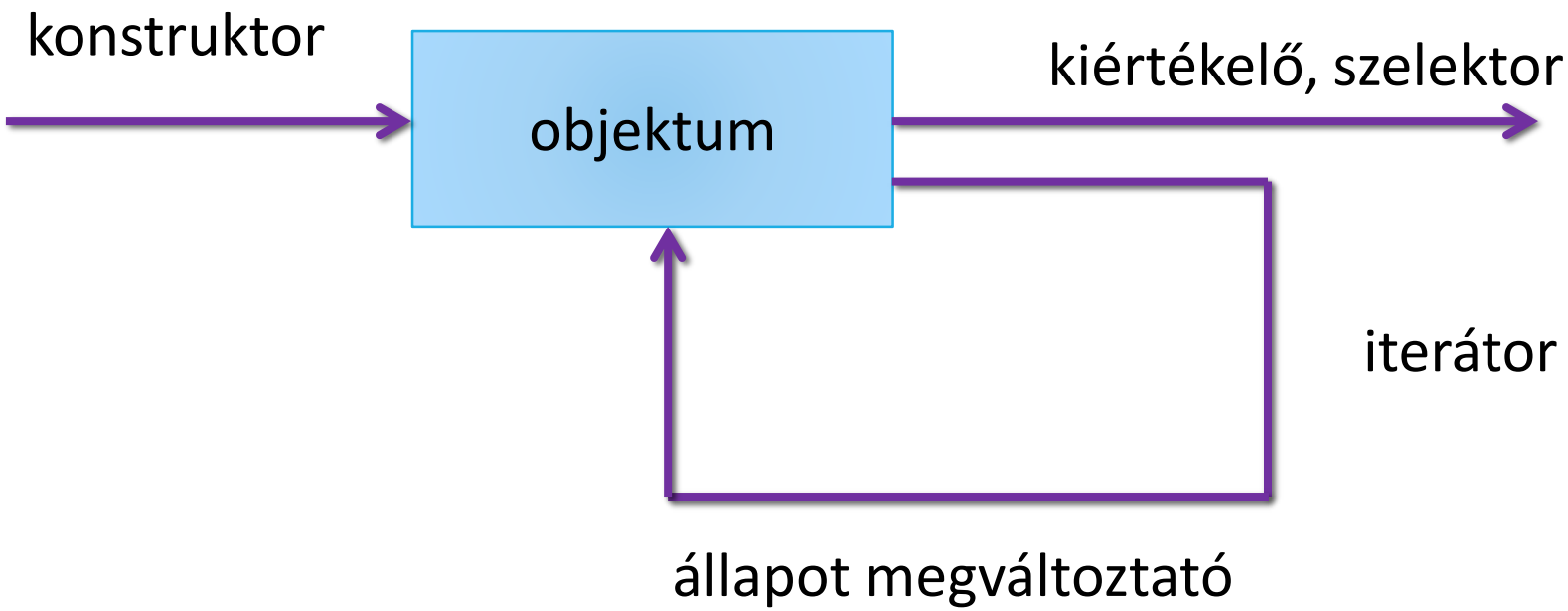
# Objektum műveletei

Export műveletek csoportosítása:

- Szelektor – kiemeli az objektum bizonyos részét
  - Például
    - vektor adott indexű elemét
    - `access`: $\text{Vektor} \times \text{Index} \rightarrow \text{Elem}$
- Kiértékelő – objektum jellemzőit lekérdező műveletek (`size`, `has`, stb.)
- Iterátor – bejáráshoz

# Objektum műveletei

Export műveletek csoportosítása:



# Objektum megszűnése

Az objektum életciklusának végén:

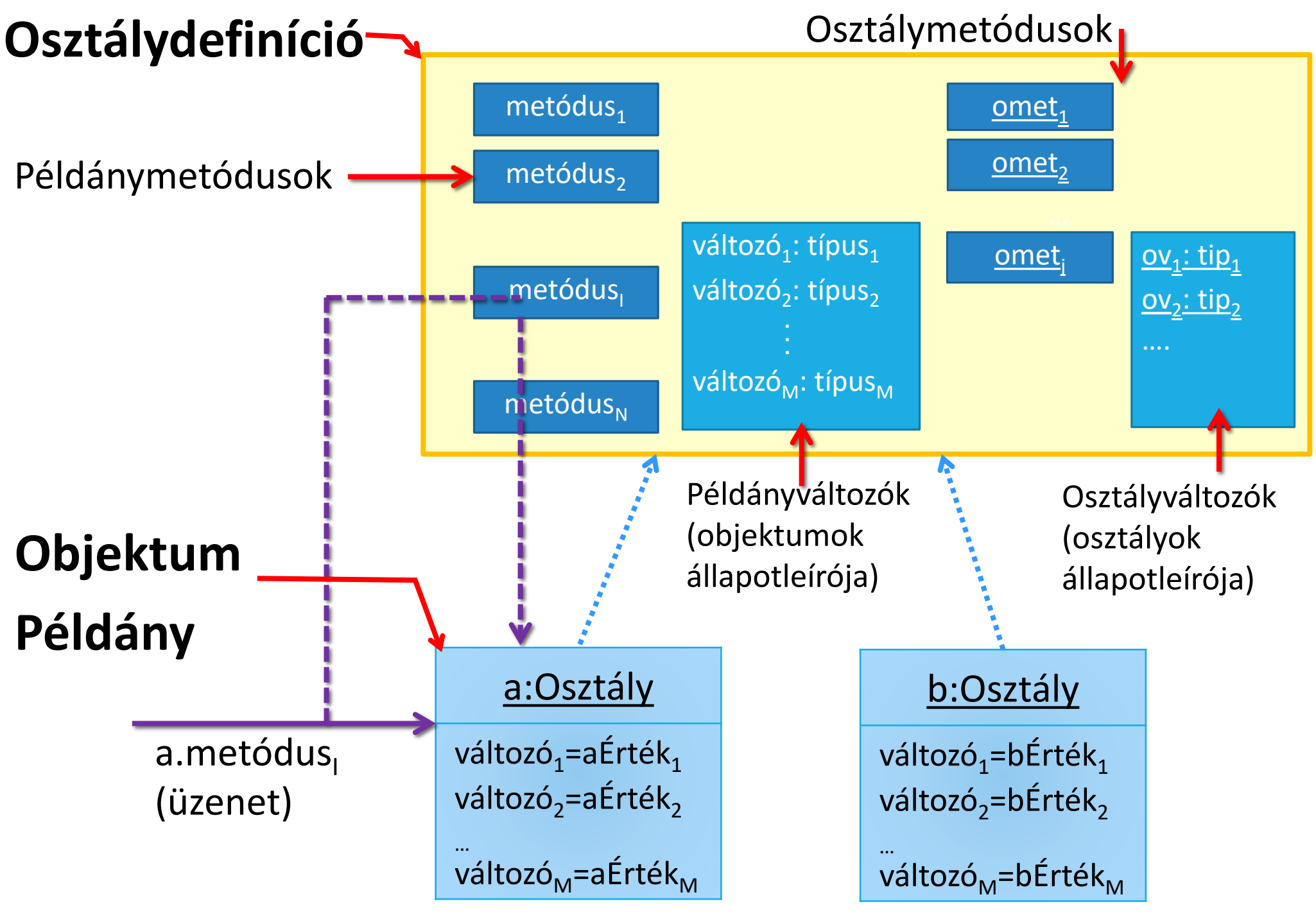
- Erőforrások (memória) felszabadítása
- Adatok, állapot mentése

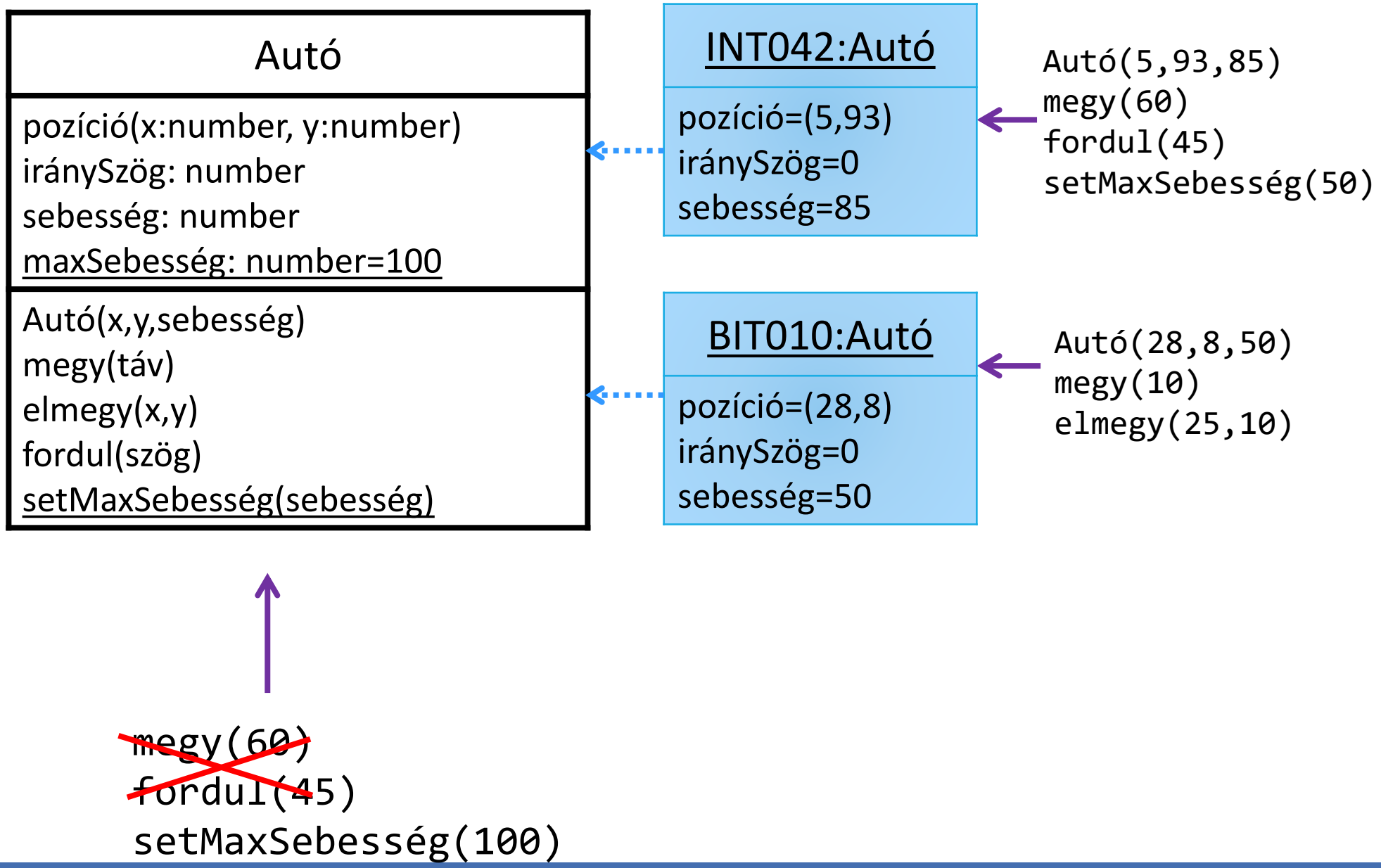
Ha a nyelv támogatja, akkor a destruktor (destructor) hajtja végre.

# Osztály, példány

Osztálydefiníció:

- Példányváltozó
  - Példányonként helyet foglaló változó
- Példánymetódus
  - Példányokon dolgozó metódus
- Osztályváltozó
  - Osztályonként helyet foglaló változó
- Osztálymetódus
  - Osztályokon dolgozó metódus


Osztálydefiníció
Osztálymetódusok
Példánymetódusok
metódus1
metódus2
metódusI
metódusN
omet1
omet2
...
ometj
változó1: típus1
változó2: típus2
változóM: típusM
ov1: tip1
ov2: tip2
....
Példányváltozók
(objektumok
állapotleírója)
Osztályváltozók
(osztályok
állapotleírója)
Objektum
Példány
a.metódusI
(üzenet)
a:Osztály
változó1=aÉrték1
változó2=aÉrték2
...
változóM=aÉrtékM
b:Osztály
változó1=bÉrték1
változó2=bÉrték2
...
változóM=bÉrtékM


Autó
pozíció(x:number, y:number)
iránySzög: number
sebesség: number
maxSebesség: number=100
Autó(x,y,sebesség)
megy(táv)
elmegy(x,y)
fordul(szög)
setMaxSebesség(sebesség)
INT042:Autó
pozíció=(5,93)
iránySzög=0
sebesség=85
Autó(5,93,85)
megy(60)
fordul(45)
setMaxSebesség(50)
BIT010:Autó
pozíció=(28,8)
iránySzög=0
sebesség=50
Autó(28,8,50)
megy(10)
elmegy(25,10)
megy(60)
fordul(45)
setMaxSebesség(100)

# A „this”

Adott osztályhoz tartozó példányok esetén

- Honnan tudjuk, hogy éppen melyik objektum hívta meg a megfelelő metódust
- Melyik objektum adataival fog dolgozni a metódus?

Szükségünk van egy olyan mutatóra, amely mindig a metódust meghívó objektumpéldányra mutat.

- Ezt szolgálja a „`this`” pszeudóváltozó
  - Néha explicit formális paraméter
- Ez metódushíváskor egyértelműen rámutat azokra az adatokra, amelyekkel a metódusnak dolgoznia kell.

Az objektum a saját magának küldött üzenet esetén a `this.Üzenet(Paraméterek)` formát kell, hogy használja.

- A metódustörzsekben az adott példányra mindig a `this` segítségével hivatkozhatunk.
  - Ez számos nyelvben alapértelmezett.

# OOP elvárások

## Bezárás (encapsulation)

- Adatok és metódusok összezárása
- Egybezárás, egységbezárás – osztály (class)

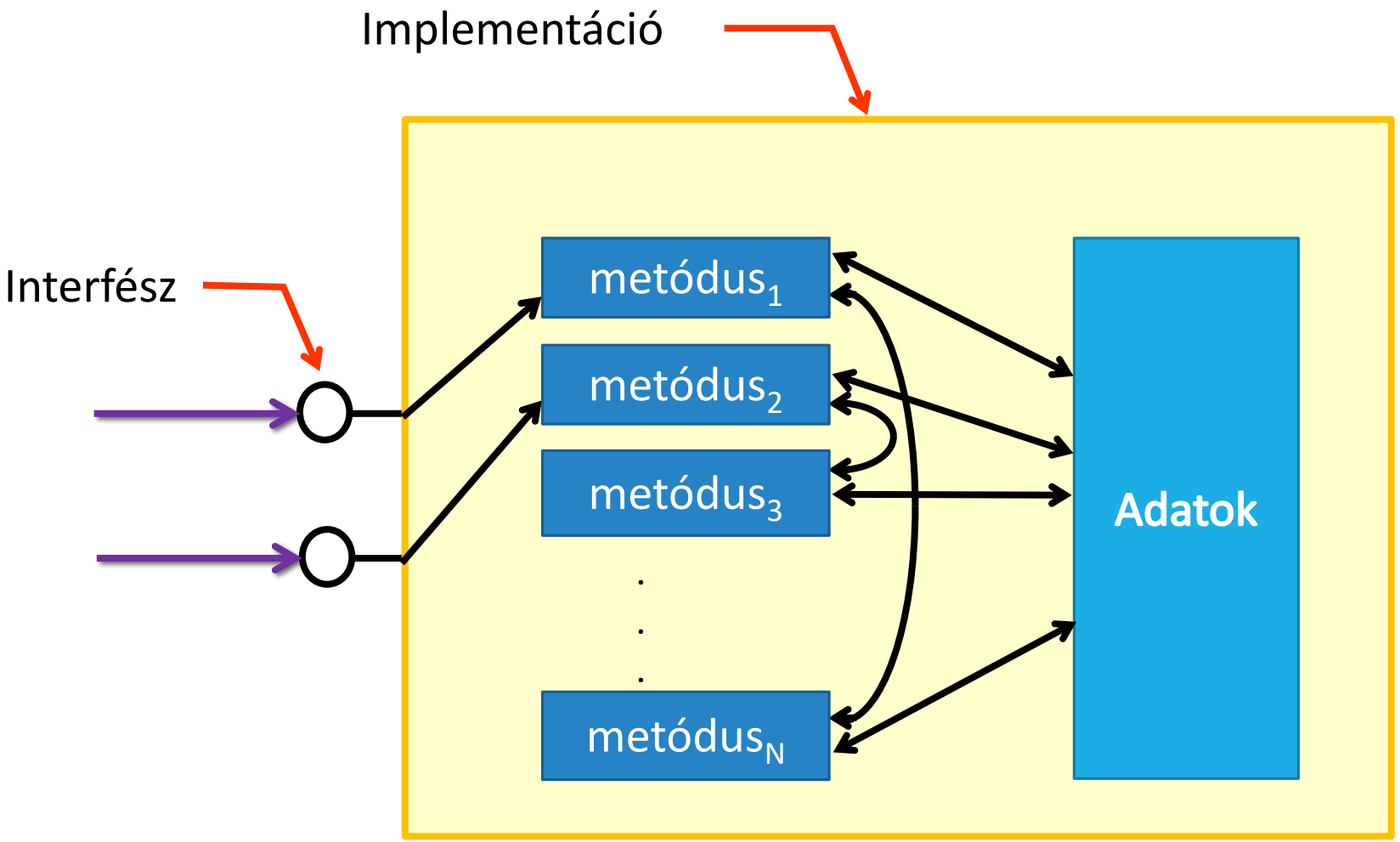
## Információ elrejtése (information hiding)

- Az objektum „belügyeit” csak az interfészen keresztül lehet megközelíteni (láthatóságok!)

## Kód újrafelhasználása (code reuse)

- Megírt kód felhasználása példány létrehozásával vagy osztály továbbfejlesztésével

# Információ elrejtése



# Információ elrejtése

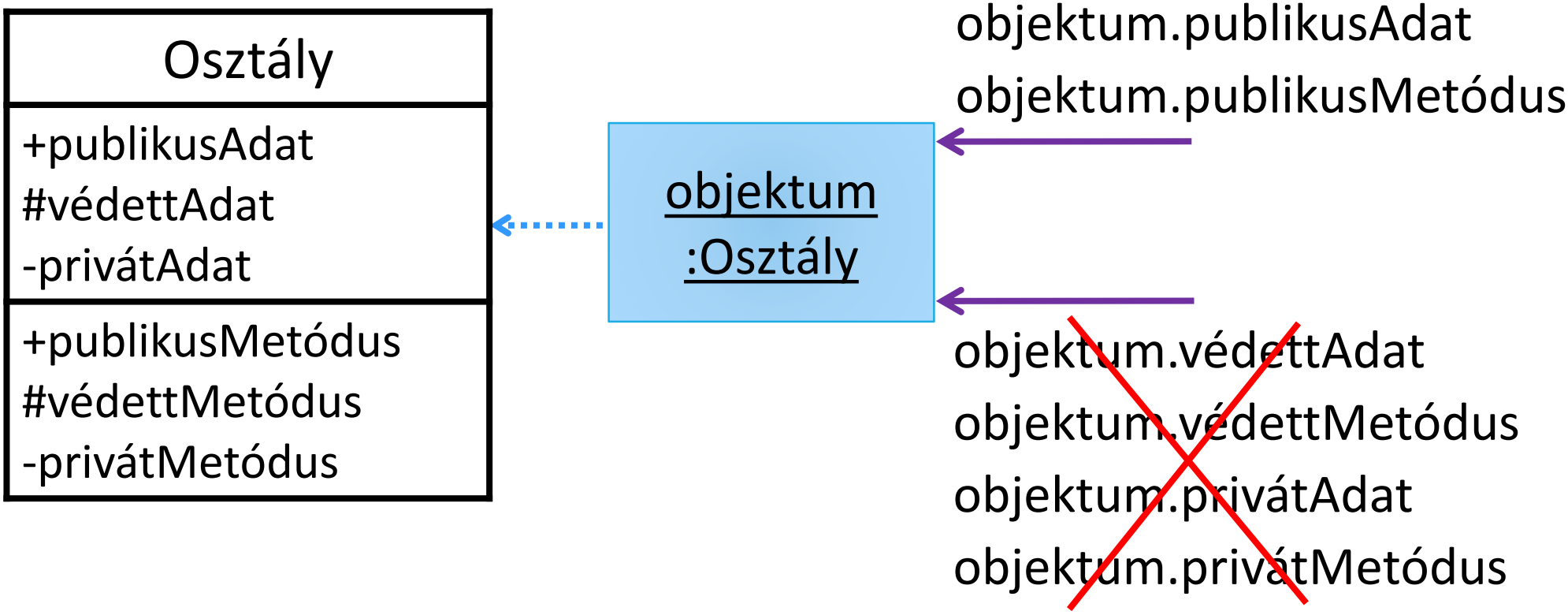
Az objektum adatait csak interfészen keresztül lehet elérni

- Az elv szigorú követése esetén, minden adattag elrejtésre kerül, közvetlenül nem hozzáférhető
  - Dedikált függvények használatával lehetséges a hozzáférés.

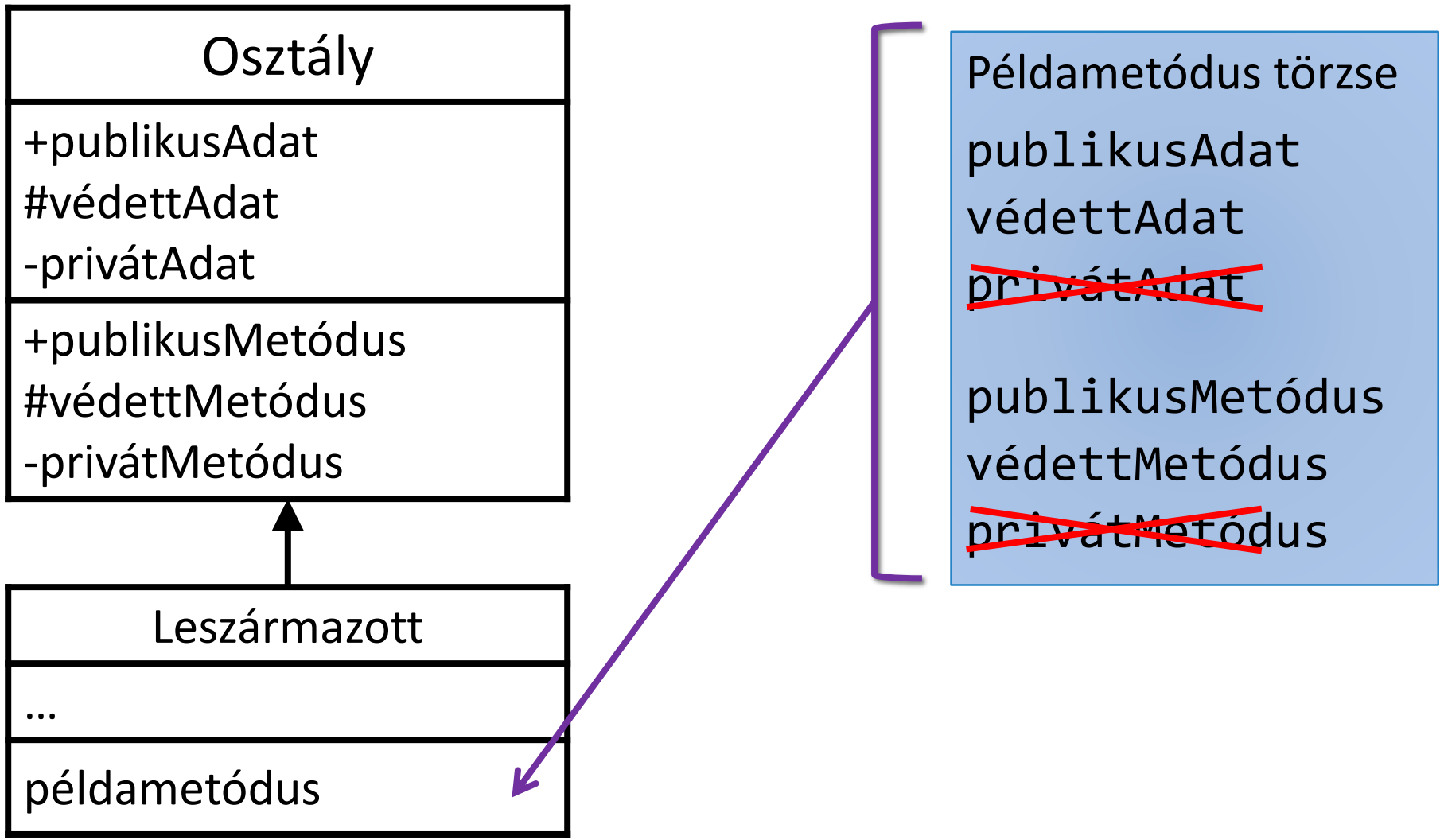
A láthatóság minősítője leggyakrabban

- public
  - a külső felhasználók elérik
- protected
  - csak a leszármazottak érhetik el
- private
  - csak az adott osztály számára elérhető

# Láthatóság – Objektum védelme



# Láthatóság – Objektum védelme



# Öröklés

Az alapgondolat: a gyerekek öröklik őseik metódusait és változóit

Az örököl azt jelenti, hogy az ősosztály minden metódusa és adattagja a gyerekosztálynak is metódusa és adattagja lesz

- A gyerek minden új művelete vagy adattagja egyszerűen hozzáadódik az örökölt metódusokhoz és adattagokhoz
- Minden metódus, amit átdefiniálunk a gyerekben, a hierarchiában felülbírálja az örökölt metódust

Lehet egyszeres, vagy többszörös

- Ez a közvetlen ősosztályok számát jelenti

# Öröklés

Mit örököl a leszármazott?

- Tulajdonságokat (Adattagokat)
- Metódusokat (Üzeneteket)

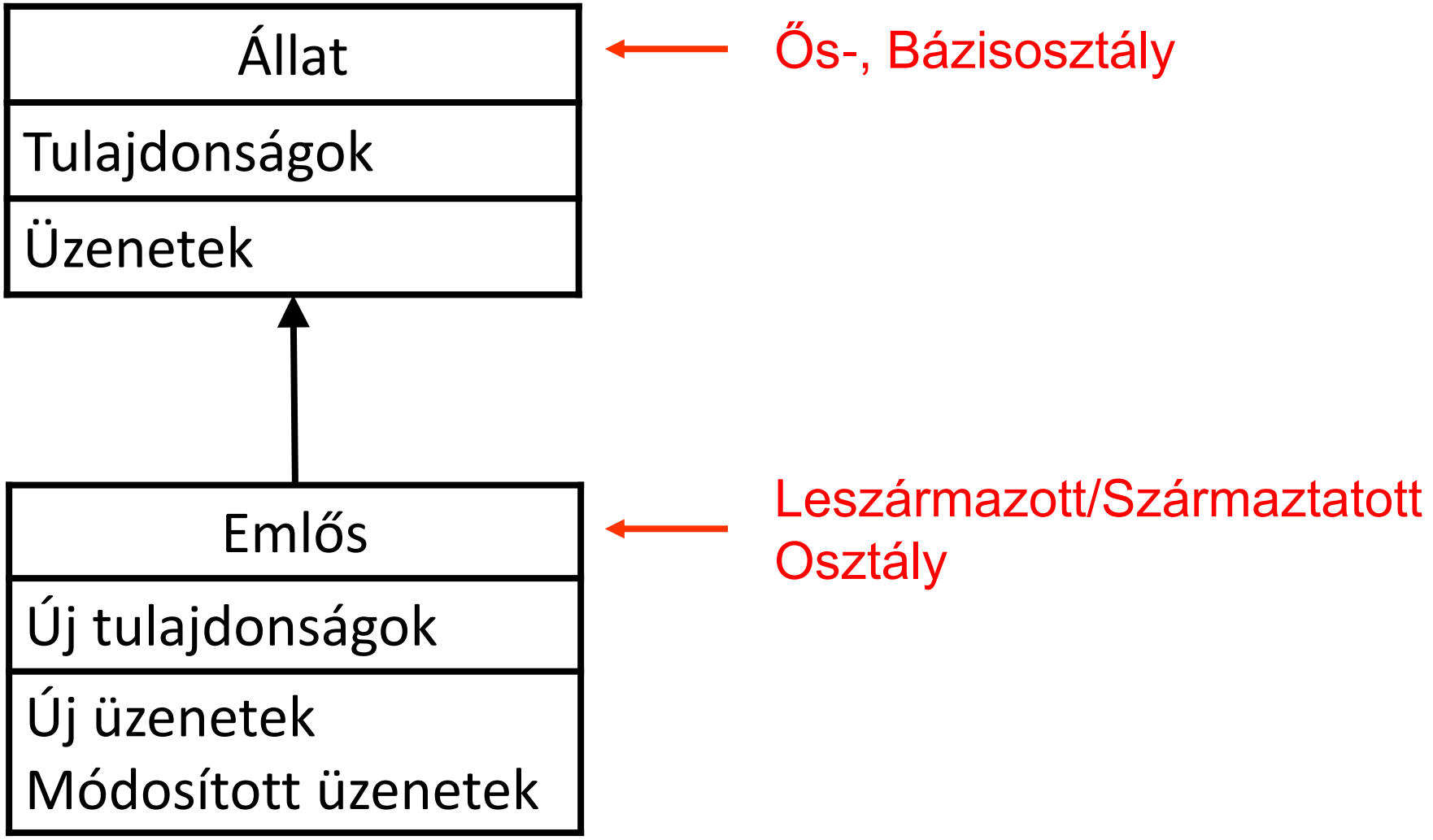
Mit nem örököl a leszármazott?

- Ősosztály konstruktorait, destruktorát
  - Ám tudja használni / meghívni / delegálni
- Ősosztály értékadás operátorát

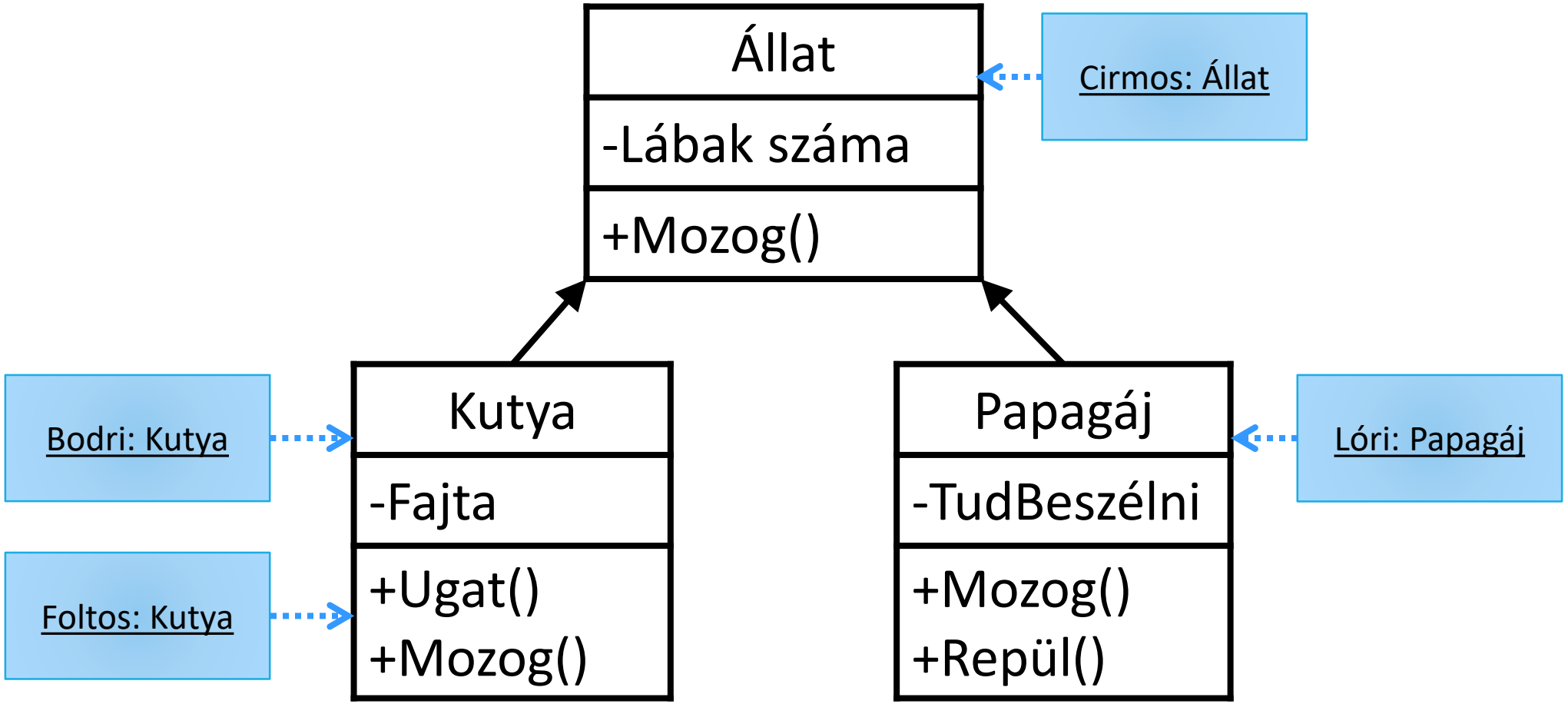
Mit tehet a leszármazott osztály?

- Új tulajdonságokat vezethet be
- Új metódusokat vezethet be
- Felüldefiniálhat, vagy elfedhet már meglévőket
- Új konstruktorokat és destruktort

# Öröklés – IS-A reláció



# Öröklés – IS-A reláció



# Implementáció újrahasznosítás

## Alosztályképzés

- Használd egy típus  implementációját egy másik típus implementálására!
- Gyakran használjuk az őstípus implementációját az altípus implementálására
- A gyakran használt OO programozási nyelvek keverik az altípus és alosztály fogalmakat.

# Felüldefiniálás/Elfedés

## Két különböző jelenség

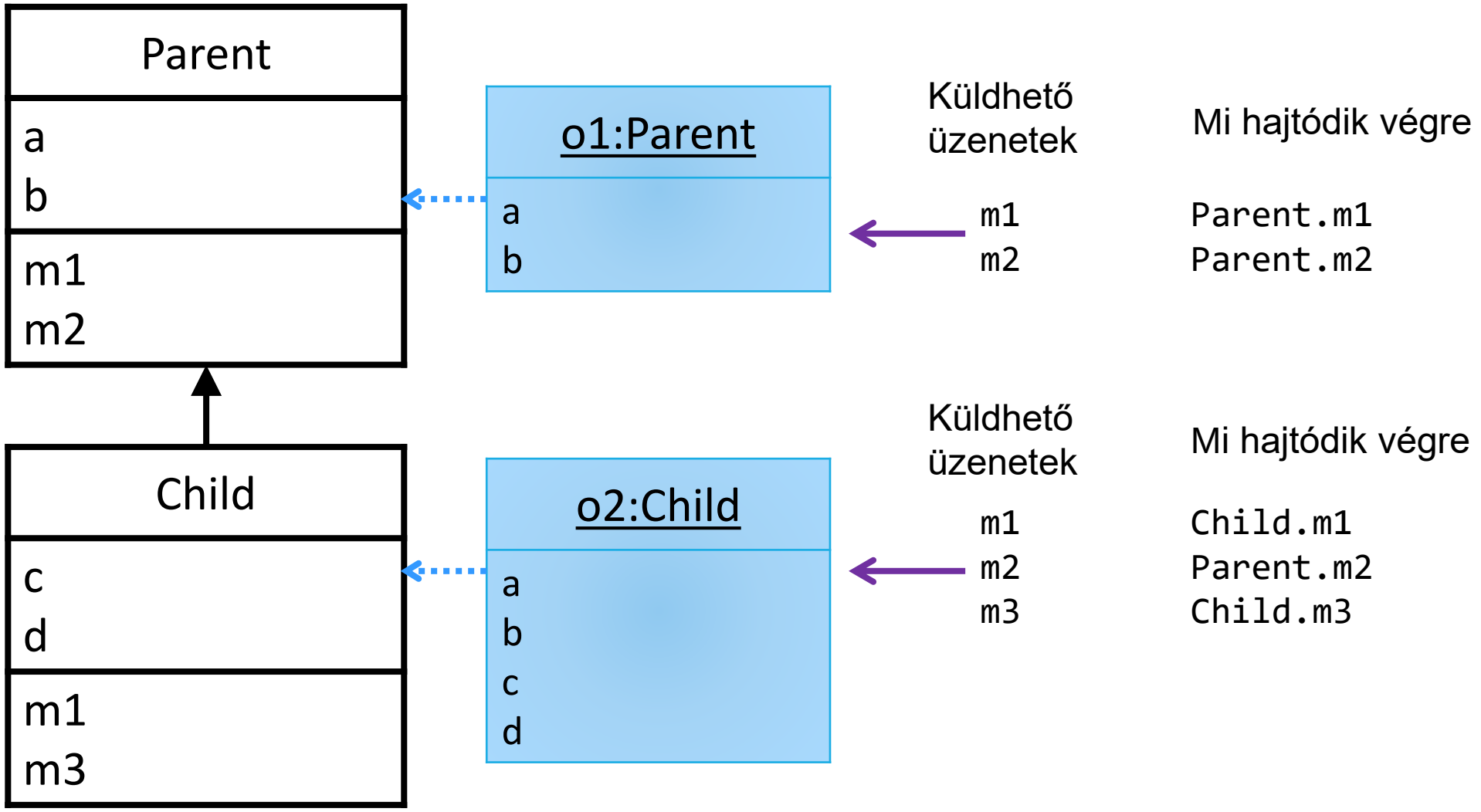
Ami közös a felüldefiniálás (átdefiniálás) illetve elfedés során:

- Az ősosztályban már meglevő, bevezetett függvénynek/üzenetnek ad új definíciót
  - A szignatúra azonos

Ami különböző

- A metóduselérés hogyan történik polimorfizmus esetén
- A két működést kulcsszavakkal lehet szabályozni általában

# Küldhető üzenetek/Elérhető adatok



# Polimorfizmus

Polimorfizmus (többalakúság)

- Az a jelenség, hogy egy azonosító (név: változó, függvény) nem csak egyfajta „objektumra” hivatkozhat.

A polimorfizmusnak több formája is van

- Altípusos
  - A (változó)név egy szupertípusra és leszármazottjaira hivatkozhat
- Nyers – Duck typing
- Parametrikus
  - Lényegében a sablonok
- Ad hoc
  - Lényegében a függvények túlterhelése

# Altípusos polimorfizmus

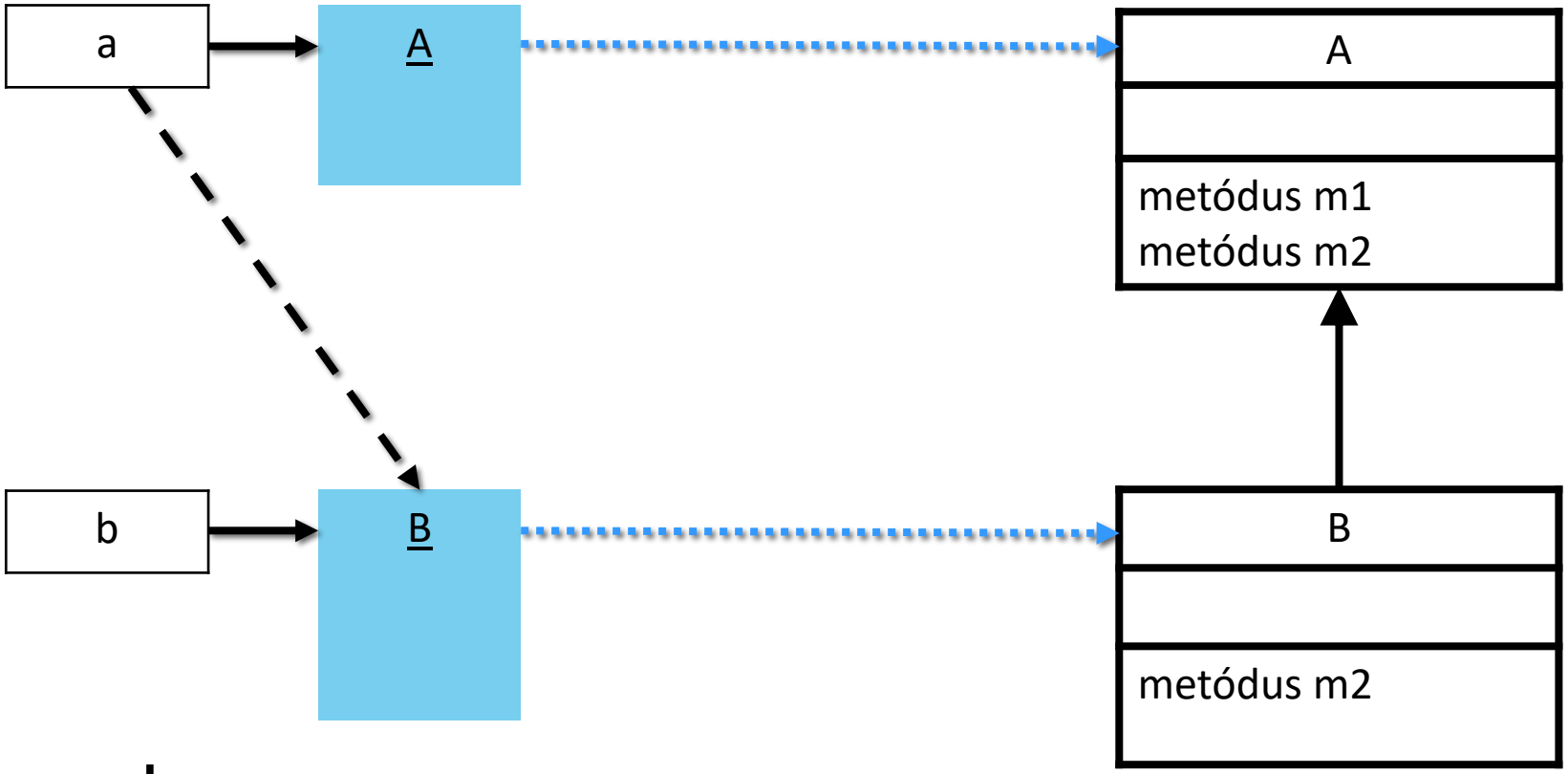
Változó nem csak egyfajta típusú objektumra hivatkozhat, hanem annak altípusaira is

- Statikus típus: a deklaráció során kapja.
- Dinamikus típus: futásidőben, éppen milyen típusú objektumra hivatkozik
  - a statikus típus, vagy annak leszármazottja

`A` típus altípusa `B`, ha a `B` minden olyan helyzetben használható, ahol az `A` is.

- A `Kutya` egyben `Állat` is (IS-A).
- A `Háromszög` az egy `Alakzat`.

# Altípusos polimorfizmus




a:=b

# Altípusos polimorfizmus

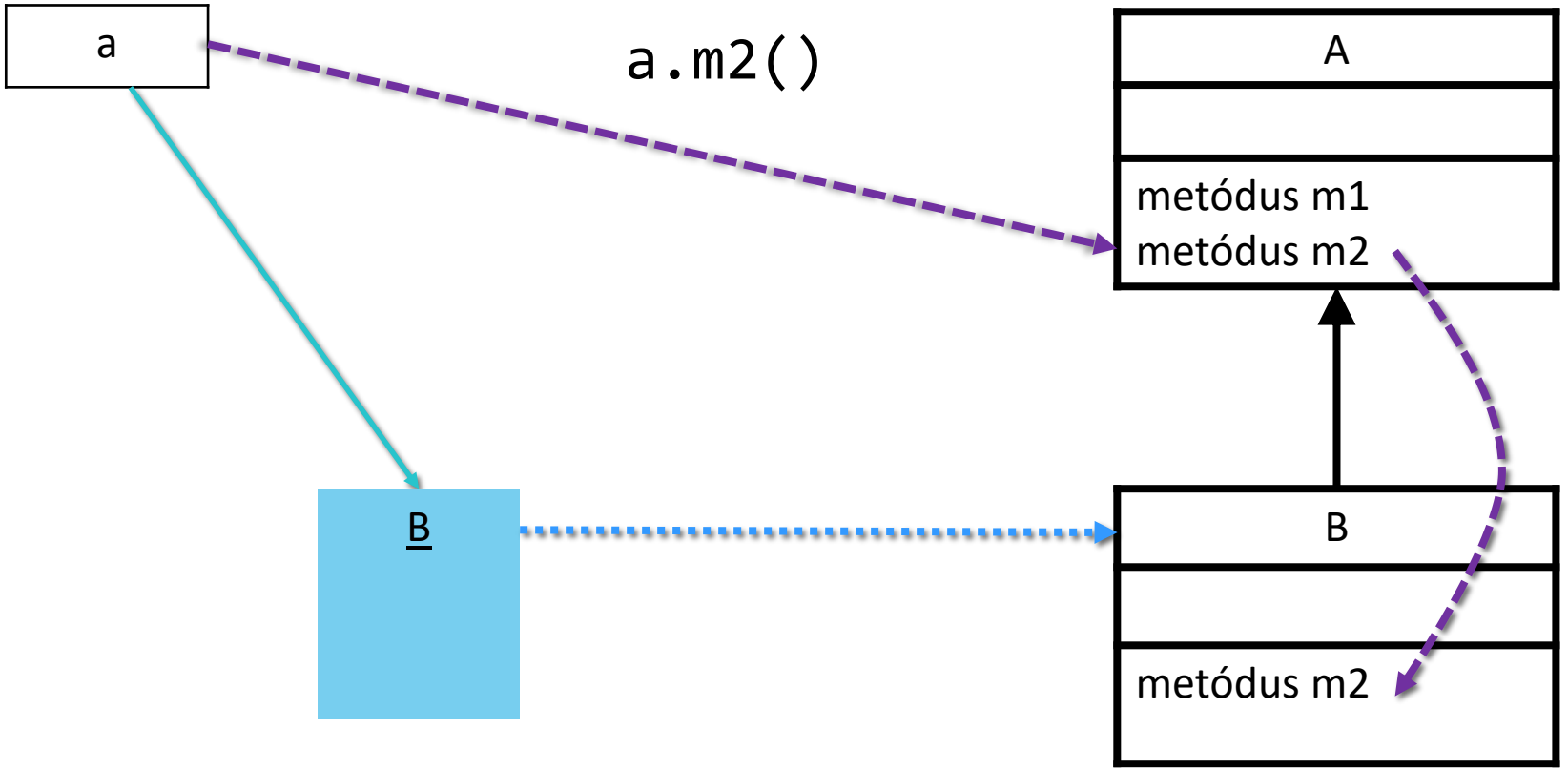
Az előző példában a `a` változó

- Statikus típusa: `A`
- Értékadást követően a dinamikus típusa `B`

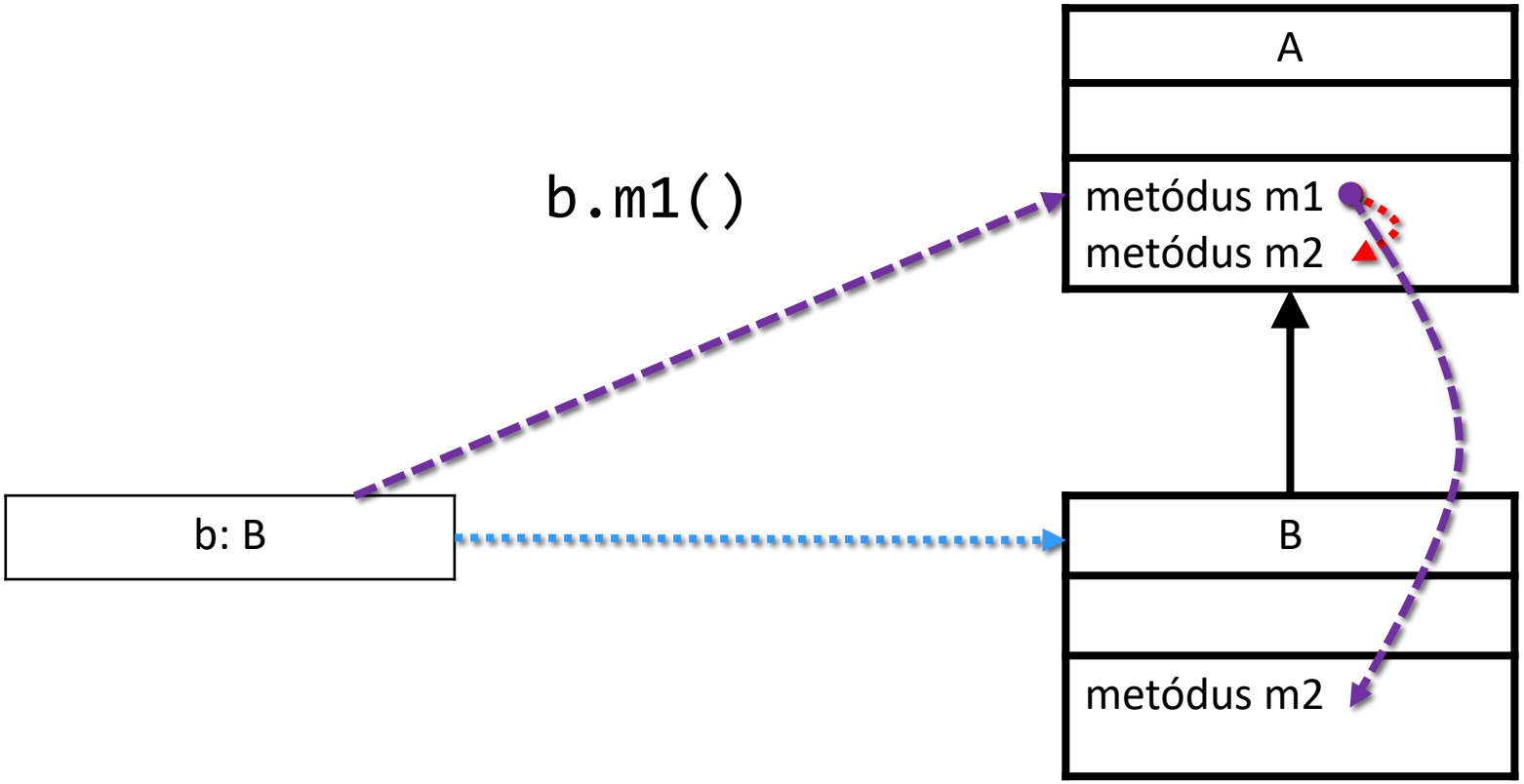
Mi történjen, ha a `B` osztály által felüldefiniált `m2()` függvényt hívjuk az `A` statikus típusú változót használva?

- Egy `B` objektum viselkedése az objektum típusának feleljen meg!
- Ezt a dinamikus összekapcsolás valósítja meg

# Dinamikus összekapcsolás



# Dinamikus összekapcsolás



# Dinamikus összekapcsolás

Run-time fogalom

- Az a jelenség, hogy a változó éppen aktuális dinamikus típusának megfelelő metódus implementáció hajtódik végre
- Felüldefiniált függvények/metódusok esetén történik meg
  - Elfedés esetén nem!

# Altípus

Szabályok:

- Reflexív
  - $T \subseteq T$
- Tranzitív
  - Ha: $T1 \subseteq T2$; $T2 \subseteq T3$
  - Akkor: $T1 \subseteq T3$

# Altípus – alprogramok esetén

Tegyük fel, hogy Triangle $\subseteq$ Shape

- `fss = proc (Shape) returns (Shape)`
- `fts = proc (Triangle) returns (Shape)`
- `fst = proc (Shape) returns (Triangle)`
- `ftt = proc (Triangle) returns (Triangle)`

- `fts` $\subseteq$ `fss` ?
- `fst` $\subseteq$ `fss` ?
- `ftt` $\subseteq$ `fss` ?
- `fss` $\subseteq$ `fts` ?

# Hogyan döntsünk?

Egy függvényt A → B alakban írunk, ha A típusú paramétere van és B típusú eredményt ad.

Ha (A’ → B’) <: (A → B), - altípus

- akkor képesek kell legyünk arra, hogy az első függvénytípus egy elemét használhassuk minden olyan kontextusban, ahol a másodikat.

Tegyük fel, hogy van egy f: A→B típusú függvényünk.

- Ha egy f’ : A’ → B‘ függvényt az f helyén akarunk használni, akkor A-beli argumentumot kell fogadnia és B-beli eredményt adnia.

# Hogyan döntsünk?

Mivel az f' értelmezési tartománya A', így akkor alkalmazható A-beli elemekre, ha A<:A', vagyis

- ha a : A tekinthető mint A'-beli, és f'(a) típus-helyes.

Másrészt, ha az f' eredménye B'-beli, akkor B'<:B garantálja, hogy az eredmény B-beliként kezelhető.

Összegezve:
(A' → B') <: (A → B), ha A<:A' és B'<:B

# Altípus – Megoldás

- `fss = proc (Shape)    returns (Shape)`
- `fts = proc (Triangle) returns (Shape)`
- `fst = proc (Shape)    returns (Triangle)`
- `ftt = proc (Triangle) returns (Triangle)`

~~`fts ⊆ fss`~~ `fst ⊆ fss` ← Az eredmény lehet speciálisabb (monoton = kovariáns)

~~`ftt ⊆ fss`~~ `fss ⊆ fts`

A paraméterek kevésbé speciálisak lehetnek (anti-monoton = kontravariáns)

# Mikor biztonságos az S $\subseteq$ T altípus reláció?

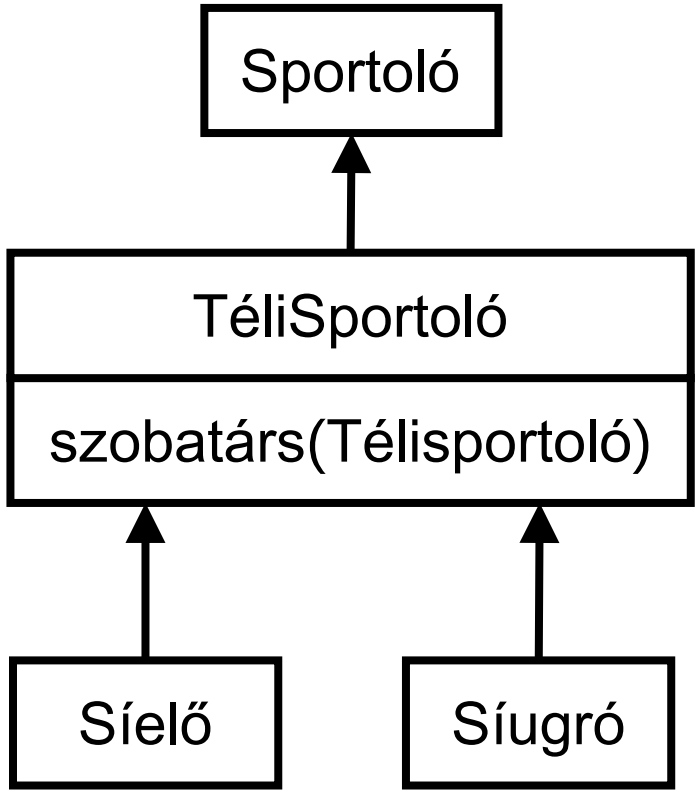
Az altípus metódusai megőrzik az őstípus viselkedését

- Szignatúra
  - Az argumentumok kontravarianciája, az eredmény kovarianciája
  - Az $m_S$ kivételei benne vannak az $m_T$ kivételei között
- Metódusok szabálya
  - Előfeltétel: „kontravariancia – altípus gyengébb”
  - Utófeltétel: „kovariancia – altípus erősebb”

Az altípusok megőrzik a szupertípusok tulajdonságait – típusinvariánsukra:

- „kovariancia – altípus erősebb”

# Kontra/Ko-Variancia




Hogyan tudja `Síugró` felüldefiniálni `szobatárs`-at?

Kovariancia esetén
- `Síelő.szobatárs (Síelő)`
- `Síugró.szobatárs (Síugró)`

Kontravariancia esetén
- `szobatárs (Sportoló)`

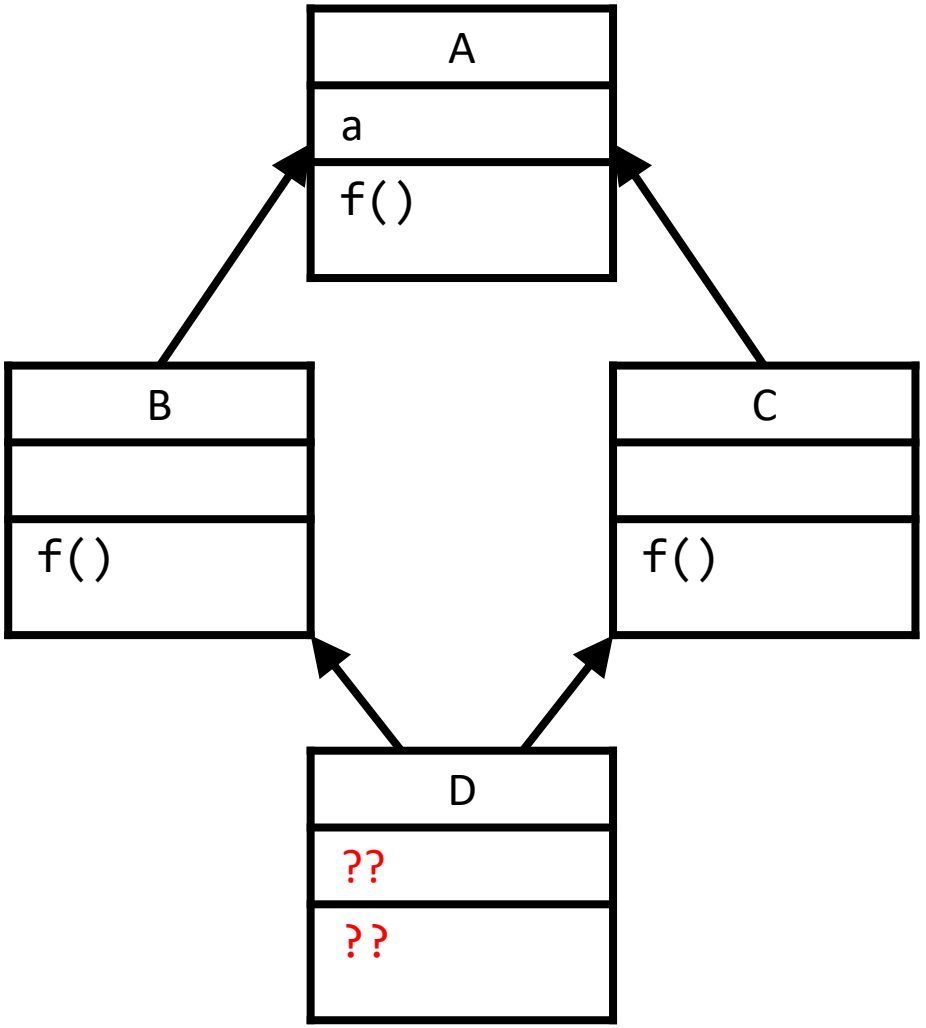
Novariancia esetén
- `szobatárs (TéliSportoló)`

Probléma (kovariáns esetben)
- `t:  TeliSportolo`
- `si: Sielo`
- `sg: Siugro`
- `t := si`
- `t.szobatars(sg)`

# A többszörös öröklődés problémái

Diamond problem – A bázisosztály definiál f függvényt és a attribútumot.

- Ezt B és C osztályok függetlenül származtatják – felüldefiniálják
- D osztály B és C osztályokból közvetlenül származik többszörös örökléssel
- Ekkor
  1. A D-beli f melyiket jelentse?
  2. Az „a” attribútum hány példányban jelenjen meg D-ben?



# A többszörös öröklődés problémái

A két kérdés lényegében ugyanazt a problémát veti fel: ha kétértelműség van, hogyan válasszunk?

- A legtöbb esetben az ilyen kódot nem lehet lefordítani
  - A fordító, vagy a futtató környezet kétértelműségre (*ambiguous*) hivatkozva hibajelzéssel leáll.
- Megoldás
  - Az ősosztály mondja meg, hogy mit szeretne tenni ilyen esetben.
  - A származtatott osztály mondja meg, hogy melyiket szeretné használni.

# Absztrakt osztály

Tervezés eszköze

Egy felső szinten összefogja a közös tulajdonságokat

A metódusok között van olyan, aminek csak specifikációja van, törzse nincs

Nem hozható létre példánya.

A leszármazott teszi konkréttá.

# Interfész

Típusspecifikáció támogatása

„Szolgáltatások”

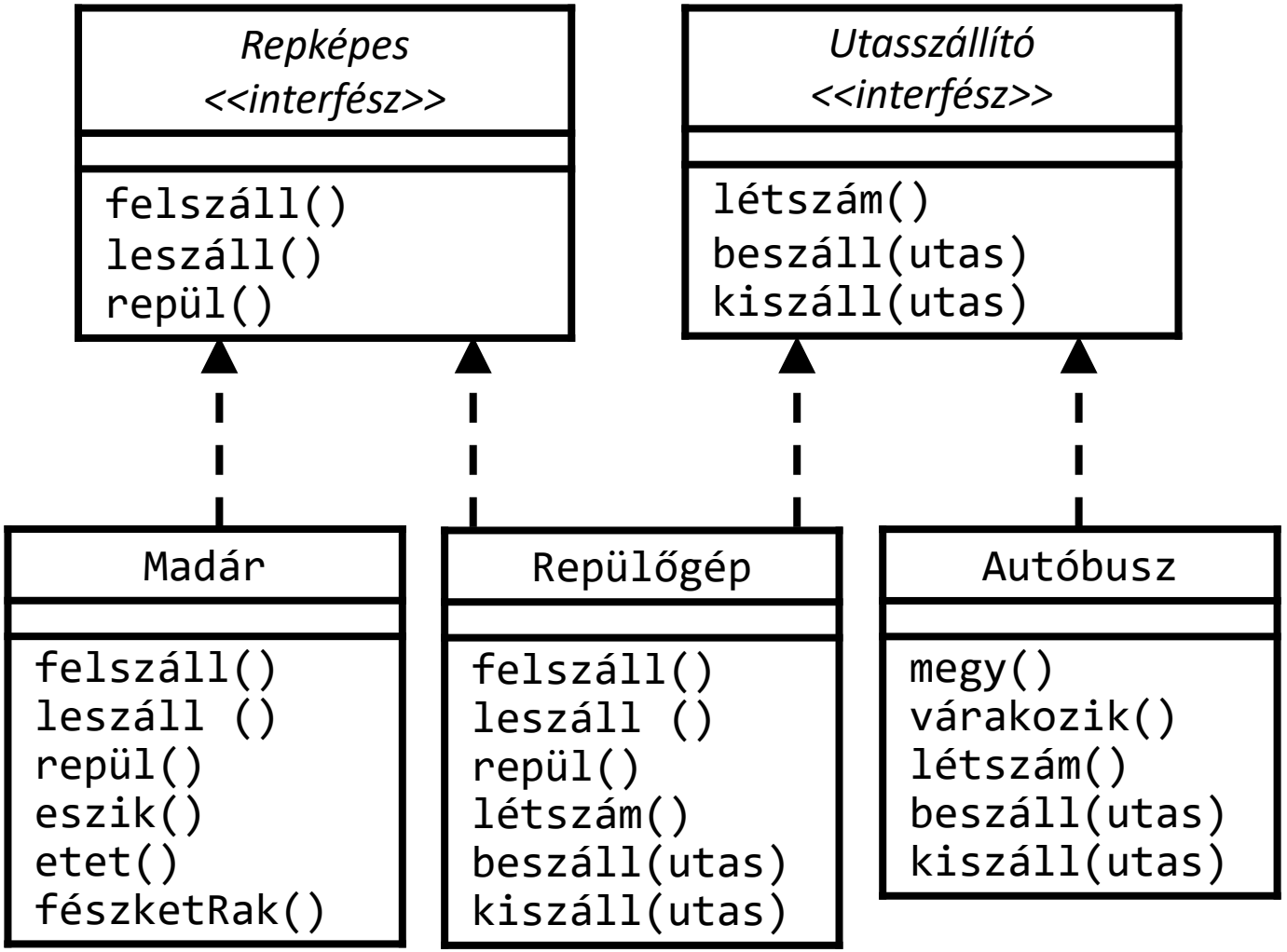
Meg kell valósítani a szolgáltatásait

Többszörös öröklődéssel nincs (annyi) probléma

- összefogja a közös jellemzőket

Interfész ≠ Absztrakt osztály!

# Interfész




**Interfészek**

**Implementáló osztályok**

# Mi az objektumorientált programozás?

A programozó definiálhat altípus kapcsolatokat

A típusszabályok megengedik, hogy az altípus használható legyen a szupertípus helyén

- altípusos polimorfizmus

Típus-vezérelt metódus elérés

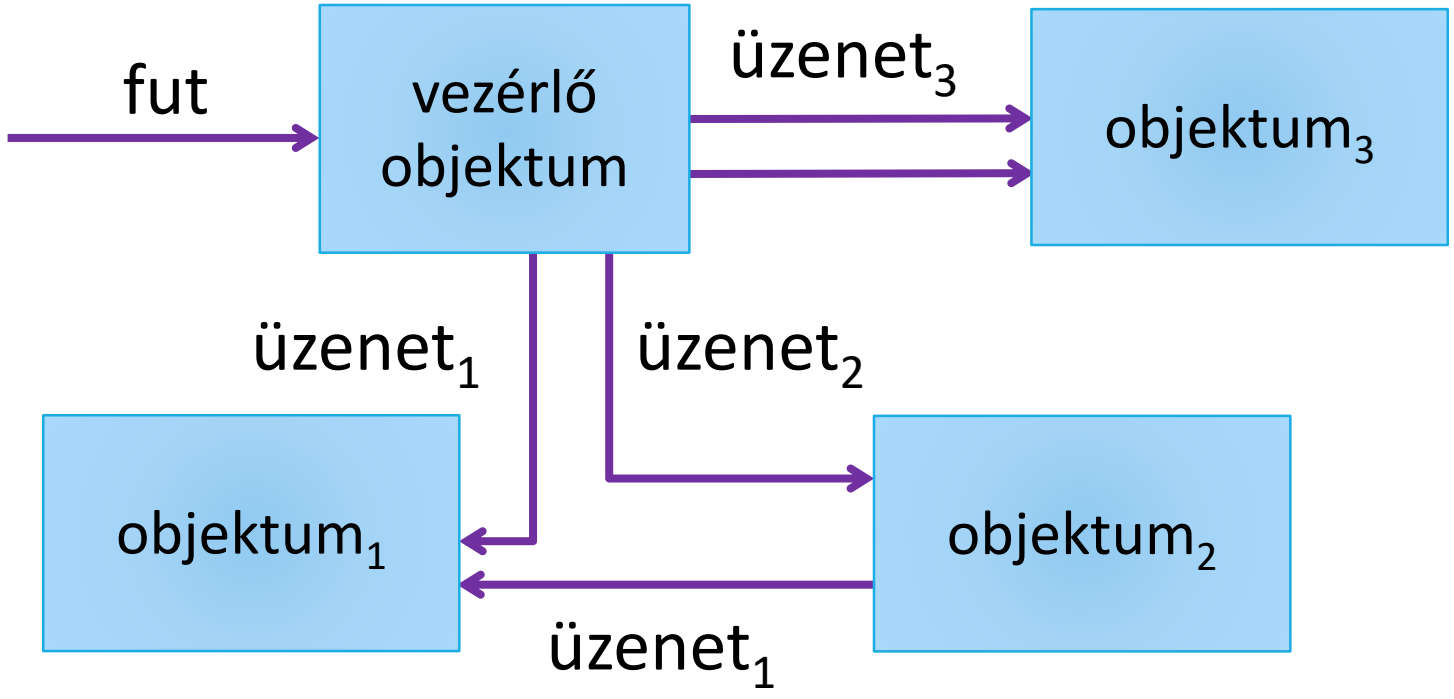
- dinamikus kötés

Implementáció megosztása

- öröklődés

# OO program

Egy objektumorientált program egymással kommunikáló objektumok összessége, melyben minden objektumnak megvan a feladatköre



# Osztály, példány

Minden objektum?

Akkor az osztályok is …

- Lehet belső állapotuk,
- Küldhetünk üzeneteket neki …
- Minek az objektuma?
  - Metaosztály
    - Singleton objektum
    - És a metaosztály is objektum?

# Kérdések

Van-e öröklődés?

Van-e többszörös öröklődés?

Adattagok és metódusok elrejtése hogy van megoldva?

Támogatja-e a polimorfizmust és dinamikus összekapcsolást?

- Esetleg csak mutatók használatával?

Van-e alapértelmezett ősosztály?

- Object?

# Kérdések

Van-e absztrakt osztály?

- Nem példányosítható közvetlenül?

Konstruktor

Destruktor?

- Garbage Collector?

Standard objektum könyvtárak?

Osztályszintű attribútumok és metódusok?

# C++

# Osztálydefiníció

A C++ **`struct`** és **`class`** sok szempontból hasonló

- OOP elvárások tekintetében a **`class`** használata szükséges

Példa egy egyszerű osztály definíciójára

- Osztálydefiníciók egymásba ágyazhatók

```
class Rectangle {
    double a_side, b_side;
public:
    void print() const {  }
};
```

Példányosítás

```
Rectangle rect(10, 2);
Rectangle *rectpt = &rect;
```

# Konstruktorok

Konstruktorok: „nincs objektum konstruktor nélkül”, ha kell, implicit hívódnak

- Neve megegyezik az osztály nevével
- Ha már megadtunk egy konstruktort, akkor default konstruktor nem definiálódik
- A default konstruktor meghívja az attribútumok konstruktorát, de a beépített típusokat nem inicializálja (konzisztensen a C-vel)
- A konstruktornak nem lehet visszatérési értéke
- C++11 óta lehetséges
  - Mezők inicializálása
  - Konstruktor delegáció
  - Szülő osztály konstruktorának örökítése

# Konstruktorok

```
class Rectangle {
    double a_side, b_side;
public:
    Rectangle(double a, double b): a_side(a), b_side(b) {};
    Rectangle() : Rectangle(1.0, 1.0) {};

    void print() const {
        std::cout << "Rectangle " << a_side << " " <<
                                     b_side << std::endl;
    }

    double calculateArea() {
        return a_side * b_side;
    }
};
```

# Destruktor

Téglalap esetén nincs kifejezett haszna, de egy általános sokszögnél (illetve példa az inicializációra)

```
class Polygon {
    double *sides {nullptr};
    int nsides {0};

public:
    Polygon(double *data, int ndata): nsides(ndata) {
        sides=new double[nsides];
        std::memcpy(sides, data, sizeof sides);
    }
    ~Polygon() { delete [] sides; }

    void print() const {
        std::cout << "Polygon, number of sides " << nsides
                                                << std::endl;
    }
};
```

# Osztályváltozó, osztálymetódus

```
class Circle {
    double radius;
    static double PI;
public:
    Circle(double r): radius(r) {};
    Circle() : Circle(1.0) {};

    void print() const {
        std::cout << "Circle " << radius << std::endl;
    }
};

double Circle::PI = 3.1415926535;
```

Az osztályon kívül definiálni kell!

# Osztályváltozó, osztálymetódus

```
class Circle {
 //...
 double calculateArea() {
        return radius * radius * PI;
    }
 //...
};
```

Az osztályon belül elérhető közvetlenül, itt a külső használathoz publikus metódus kell!

```
static double getPi() { return Circle::PI; }
```

# Osztályváltozó, osztálymetódus

Használat:

```
Circle::PI; // HIBA
Circle::getPi();
Circle c1(10);
c1.getPi();
```

Szemantikailag helytelen példányon keresztül elérni, azaz:

- ~~`c1.getPi();`~~
- ~~`return radius * radius * PI;`~~

Helyette

- `return radius * radius * Circle::PI;`

# Láthatóság

Az adattagok és metódusok elrejtése megoldott

A láthatóság minősítője C++-ban

- **`public`**
  - a külső felhasználók elérik
- **`protected`**
  - csak a leszármazottak érhetik el
- **`private`**
  - csak az adott osztály és a barátok (**`friend`**) számára elérhető – alapértelmezés

# Öröklődés

A téglalap egy specializációja a négyzet:

```
class Square: public Rectangle {
public:
    Square(double a): Rectangle(a, a) { };
    Square(): Square(1) { };

    void print() const {
        std::cout << "Square " << a_side << std::endl;
    }
};
```

Azonban ez nem működik, mert ugyan van saját `a_side` attribútuma a négyzetnek, de azt nem éri el közvetlenül, mivel az ősben ez **`private`**.

# Öröklődés

Módosítsuk!

```
class Rectangle {
protected:
    double a_side, b_side;
 //...
};
```

Ekkor

```
rect.print();    // Rectangle 10 2
Square s(5);
s.print();       // Square 5
```

# Konstruktor öröklés

Lehetséges C++11 óta

```
class BaseClass {
public:
    BaseClass(int value);
};

class DerivedClass : public BaseClass {
public:
    using BaseClass::BaseClass;
};
```

# Altípusos polimorfizmus

Tekintsük az alábbi kódot

```
Rectangle rect(10, 2);
Square s(5);
rect = s;
```

Mi történik?

```
rect.print();
```

- Fontos, hogy itt még nincsen dinamikus kötés!

# Altípusos polimorfizmus

Tekintsük az alábbi kódot

```
Rectangle rect(10, 2);
Square s(5);
rect = s;
```

Mi történik?

```
rect.print();
```

- Fontos, hogy itt még nincsen dinamikus kötés!

# Altípusos polimorfizmus

Ha B (pl. `Square`) altípusa az A (pl. `Rectangle`) típusnak, akkor B objektumainak referenciái értékül adhatók az A típus referenciáinak.

Azaz:

```
Rectangle* prect = new Rectangle(4, 3);
Square* psqu = new Square(2);

prect->print();
psqu->print();
prect = psqu;
```

Mi történik?

```
prect->print();
```

# Dinamikus összekapcsolás

C++ nyelven a dinamikus kötéshez:

- Az **őstípus** `virtual`-nak deklarálja a metódust, amire megengedi a felüldefiniálást
- Ellenkező esetben elfedés lesz!
- Tipikusan az ősbeli **`virtual`-**t a leszármazottban is **`virtual`-**nak jelöljük.

# Dinamikus összekapcsolás

Módosítás a `Rectangle` osztályban

```
virtual void print() const {
    std::cout << "Rectangle " << a_side << " "
                              << b_side << std::endl;
}
```

Ekkor az előző példa

```
prect->print();
// Square 2
```

# Felüldefiniálás jelzése

Expliciten jelezhetjük, hogy felüldefiniálást szeretnénk

```
class Square: public Rectangle {
    //...
    virtual void print() const override {//...
};
```

- Ellenőrizni fogja a fordító ekkor
- Hibát kapunk ha felüldefiniálandó függvény
  - nem **`virtual`**
  - **`final`**

Felüldefinálás (és leszármazás) megakadályozása

```
virtual void print() const final {//...
```

# Virtuális destruktor

A destruktor esetén is fontos a **`virtual`** használata

- Amennyiben az altípusos polimorfizmus használjuk.

Nézzük az alakzatot

```
class Shape {
public:
    virtual void print() const {
        std::cout << "Generic Shape" << std::endl;
    }
};
```

# Virtuális destruktor

Ha a Polygon ebből származik le

```
class Polygon: public Shape {
  //...
  ~Polygon() {
    delete [] sides;
    std::cout<< "Deleted" << std::endl;
  }
};
```

Akkor a következő eset memóriaszivárgás

```
Shape *sh = new Polygon(new double[2]{10.0, 10.3}, 2);
delete sh;
std::cout << "^^ Memory leak ^^ " << std::endl;
```

# Virtuális destruktor

Helyesen

```
class Shape {
public:
    virtual ~Shape() {};
    virtual void print() const {
        std::cout << "Generic Shape" << std::endl;
    }
};
```

Így már rendben lesz

```
Shape *sh = new Polygon(new double[2]{10.0, 10.3}, 2);
delete sh;
std::cout << "^^ No memory leak ^^ " << std::endl;
```

# Automatikusan létrehozott függvények

A fordító több tagfüggvényt létrehoz, ha nem definiáljuk azokat. Dönthetünk az automatikusan létrejövő tagfüggvények

- létrehozásáról (**`default`**)
- elutasításáról (**`delete`**)

```
class Square {
public:
    Square() = default;
    Square(const Square&) = default;
    Square& operator=(const Car&) = delete;
    virtual ~Square() = default;
};
```

Az utolsó sornál fontos a **`virtual`**.

- Ezzel jelezzük, hogy automatikusan létrejövő destruktor virtuális is legyen!

# Absztrakt osztály

Pure virtual függvények bevezetése

```
class Shape {
 public:
      virtual ~Shape() {};
      virtual void print() = 0;
      virtual double calculateArea() = 0;
 };
```

# Absztrakt osztály

Ilyenkor a leszármazott adja meg a definíciót:

```
class Circle : public Shape {
    double radius;
public:
    Circle(double r): radius(r) {};

    virtual void print() const override {
        std::cout << "Circle radius: " << radius << std::endl;
    }
    virtual double calculateArea() const override {
        return radius * radius * PI;
    }
};
```

# Absztrakt osztály típusként

Természetesen nem példányosítható:

```
Shape s;
```

De statikus típusa egy változónak lehet:

```
Circle c;
Shape * pa = &c;
```

De statikus típusa egy változónak lehet és dinamikus kötés működik

```
pa->calculateArea(); // Circle::calculateArea()
```

# Többszörös öröklés

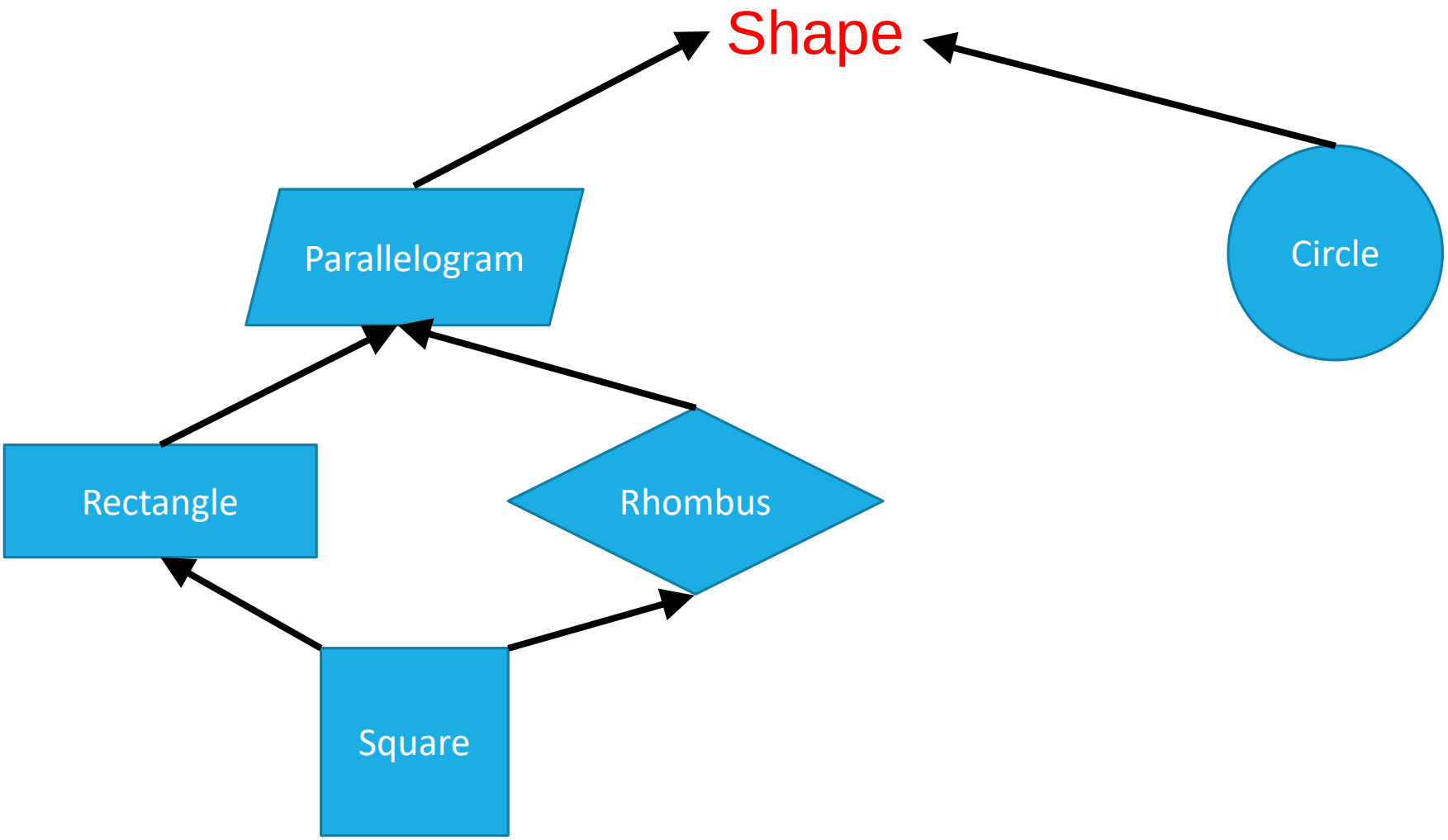
Az osztályhierarchia lehet fa, de lehet általánosabb körmentes gráf is:

- `class Shape`
- `class Circle: public Shape`
- `class Parallelogram: public Shape`
- `class Rhombus: public Parallelogram`
- `class Rectangle: public Parallelogram`
- `class Square: public Parallelogram, public Rhombus`

Problémák:

- Adattagok hányszor?
- Melyik metódus?

# Többszörös öröklés



# Többszörös öröklés

```
double PI = 3.1415926535;

class Shape {
public:
    virtual ~Shape() {};
    virtual void print() const = 0;
    virtual double calculateArea() const = 0;
};

class Circle : public Shape {
    double radius;
public:
    Circle(double r): radius(r) {};

    virtual void print() const override {
        std::cout << "Circle radius: " << radius << std::endl;
    }
    virtual double calculateArea() const override {
        return radius * radius * PI;
    }
};
```

# Többszörös öröklés

```
class Parallelogram : public Shape {
protected:
    double _side_a;
    double _side_b;
    double _delta;
    double _height_a;
public:
    virtual double get_a_side() const final { return _side_a; }
    virtual double get_b_side() const final { return _side_b; }
    virtual double get_angle()  const { return _delta;  }
    virtual bool is_equilateral() const { return _side_a == _side_b; }
    virtual bool is_right() const { return _delta == 90; }

    Parallelogram(double a, double b, double angle): _side_a(a), _side_b(b),
                                                     _delta(angle) {
        _height_a = b * std::sin(_delta*PI / 180.0);
    }

    virtual void print() const override {
        std::cout << "Parallelogram a, b, delta: " << _side_a << ", "
                  << _side_b << ", "
                  << _delta << std::endl;
    }

    virtual double calculateArea() const override {
        return _side_a * _height_a;
    }
};
```

# Többszörös öröklés

```
class Rhombus : public Parallelogram {
public:
    Rhombus(double a, double angle): Parallelogram(a, a, angle) {  };

    virtual void print() const override {
        std::cout << "Rhombus side, angle " << _side_a << ", " << _delta << std::endl;
    }

    virtual double calculateArea() const override {
        return _side_a * _side_a * std::sin(_delta*PI / 180.0);
    }

    virtual bool is_equilateral() const override final { return true; }
};

class Rectangle : public Parallelogram {
public:
    Rectangle(double a, double b): Parallelogram(a, b, 90) { _height_a = b; };

    virtual void print() const override {
        std::cout << "Rectangle " << _side_a << ", " << _side_b << std::endl;
    }

    virtual bool is_right() const override final { return true; }
};
```

# Többszörös öröklés

```
class Square: public Rectangle, public Rhombus {
public:
    Square(double a): Rectangle(a,a), Rhombus(a, 90) { };

    virtual void print() const override final {
        std::cout << "Square " << _side_a << std::endl;
    }

    virtual double calculateArea() const override final {
        return _side_a * _side_a;
    }
};

int main() {
    Rectangle *s = new Square(10);
    std::cout << s->calculateArea() << std::endl;
    s->print();

    return 0;
}
```

# Többszörös öröklés

Ebben a példában a `Square` esetén a `_side_a` mezőre hibát jelez a fordító

- A mezőt két ágon örökli, el kell dönteni, hogy hogyan legyen

1. Lehetséges megadni, például

```
virtual double calculateArea() const override final {
    return Rectangle::_side_a * Rectangle::_side_a;
}
```

- Ez azonban nem oldja meg a következőt

```
Parallelogram *s = new Square(10);
```

# Többszörös öröklés

Ebben a példában a `Square` esetén a `_side_a` mezőre hibát jelez a fordító

- A mezőt két ágon örökli, el kell dönteni, hogy hogyan legyen

2. Virtuális öröklés

- Ilyenkor a virtuális ősosztály nem két irányban lesz ős, hanem egy közös ős
- Gondoskodni kell a virtuális ősök inicializációjáról is

```
class Rhombus: public virtual Parallelogram
class Rectangle: public virtual Parallelogram
Square(double a): Parallelogram(a, a, 90),
                  Rhombus(a, 90), Rectangle(a,a) { };
```

- Így már jó

```
Parallelogram *s = new Square(10);
```

# Többszörös öröklés függvény példa

```
class A{
public:
    virtual void f() { std::cout << "A.f()" << std::endl; };
};

class B : public A {
public:
    void f() { std::cout << "B.f()" << std::endl; };
};

class C : public A{
public:
    void f() { std::cout << "C.f()" << std::endl; };
};

class D : public B, public C {
public:
    void g() { std::cout << "D.g()" << std::endl; };
};
```

# Többszörös öröklés függvény példa

Használatkor

- `D d;`
  `d.f();`
- Hibaüzenet: `request for member „f” is ambiguous`

Itt is döntés kell

- `using C::f;`
- Azonban ilyenkor!
- `B *b = new D();`
  `b->f();  // B.f()`

# Friend

A barát hozzáfér a privát adatokhoz, függvényekhez

- Tipikusan olyan operátorok készítésére használjuk, amelyek a modellezés szerint nem részei a típusnak, de szorosan kötődnek hozzá
  - Kiírás

```
class Point {
    friend ostream &operator<<( ostream &, const Point &);
public:
    Point( int = 0, int = 0 );
    void setPoint( int, int );
    int getX() const { return x; }
    int getY() const { return y; }
protected:
    int x, y;
};
```

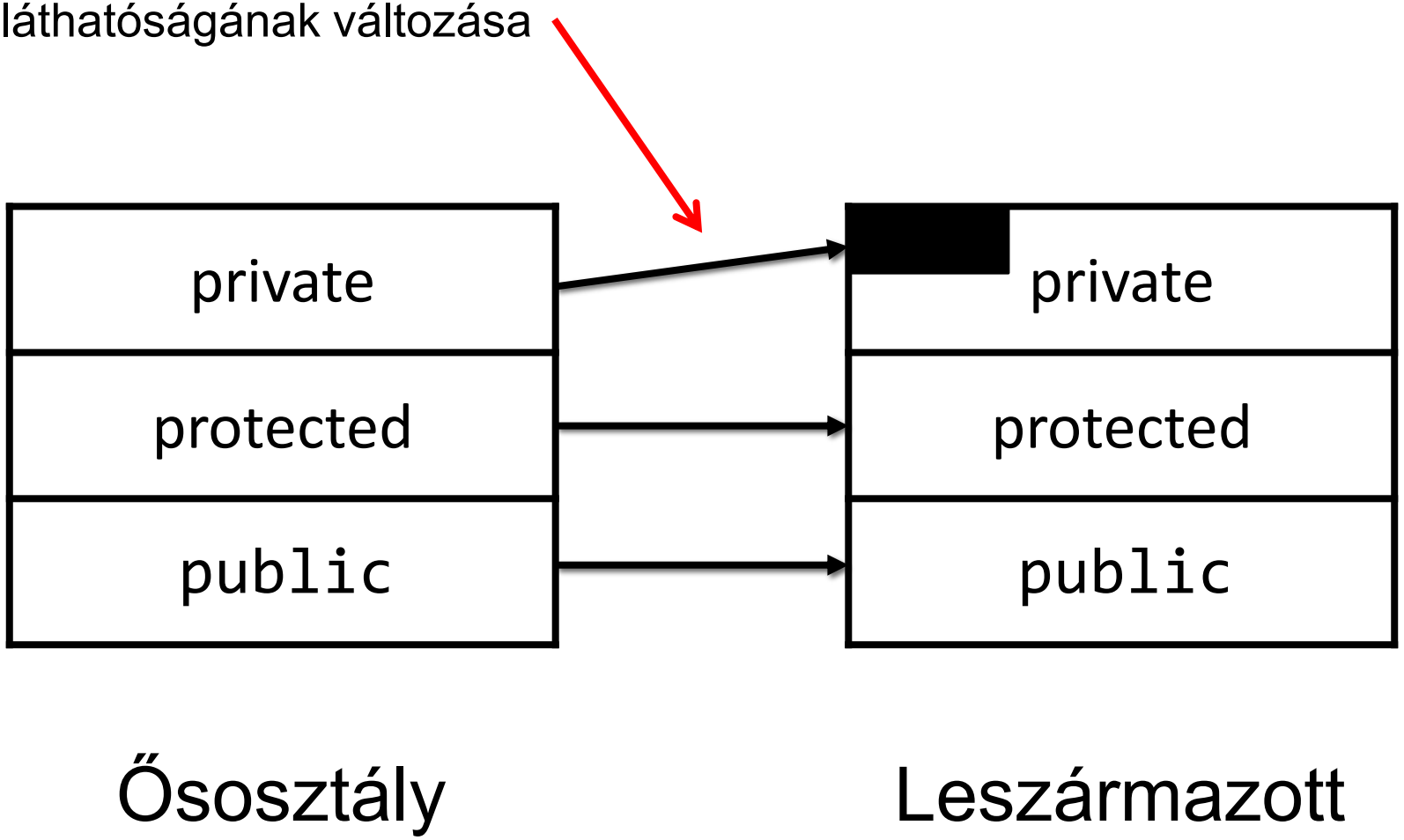
# Alosztályképzés

C++ - implementációs öröklés altípus létrehozása nélkül

- **`private`**, **`protected`** öröklés

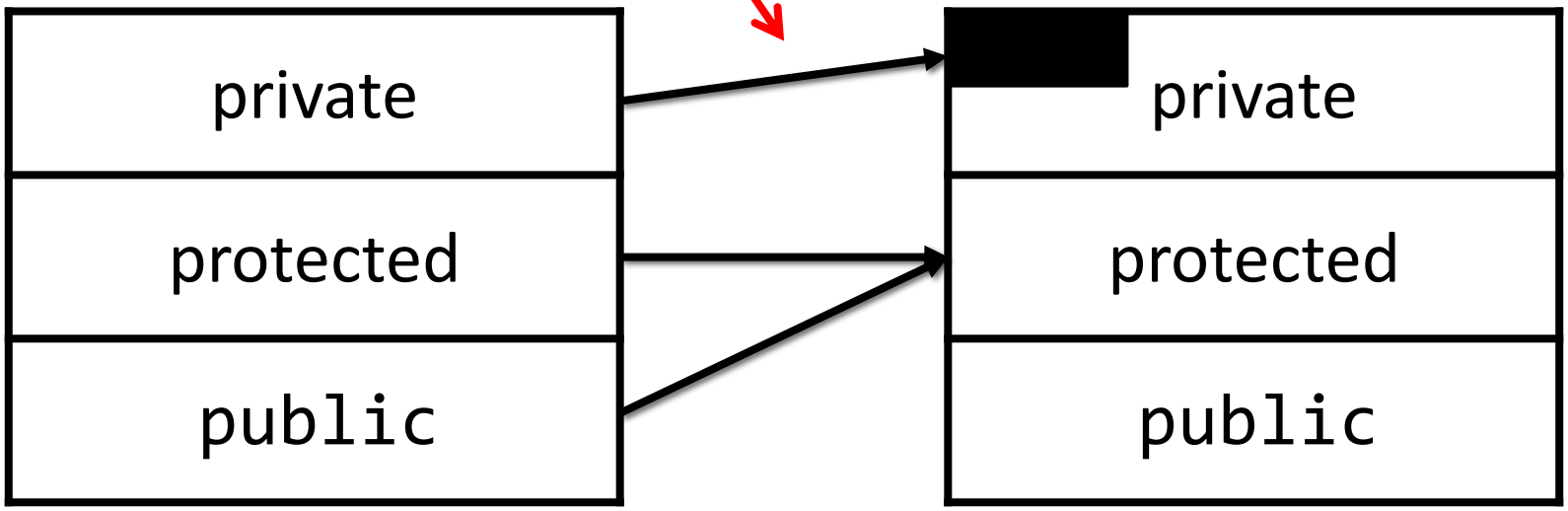
# Public öröklés

Tagok láthatóságának változása



# Protected öröklés

Tagok láthatóságának változása

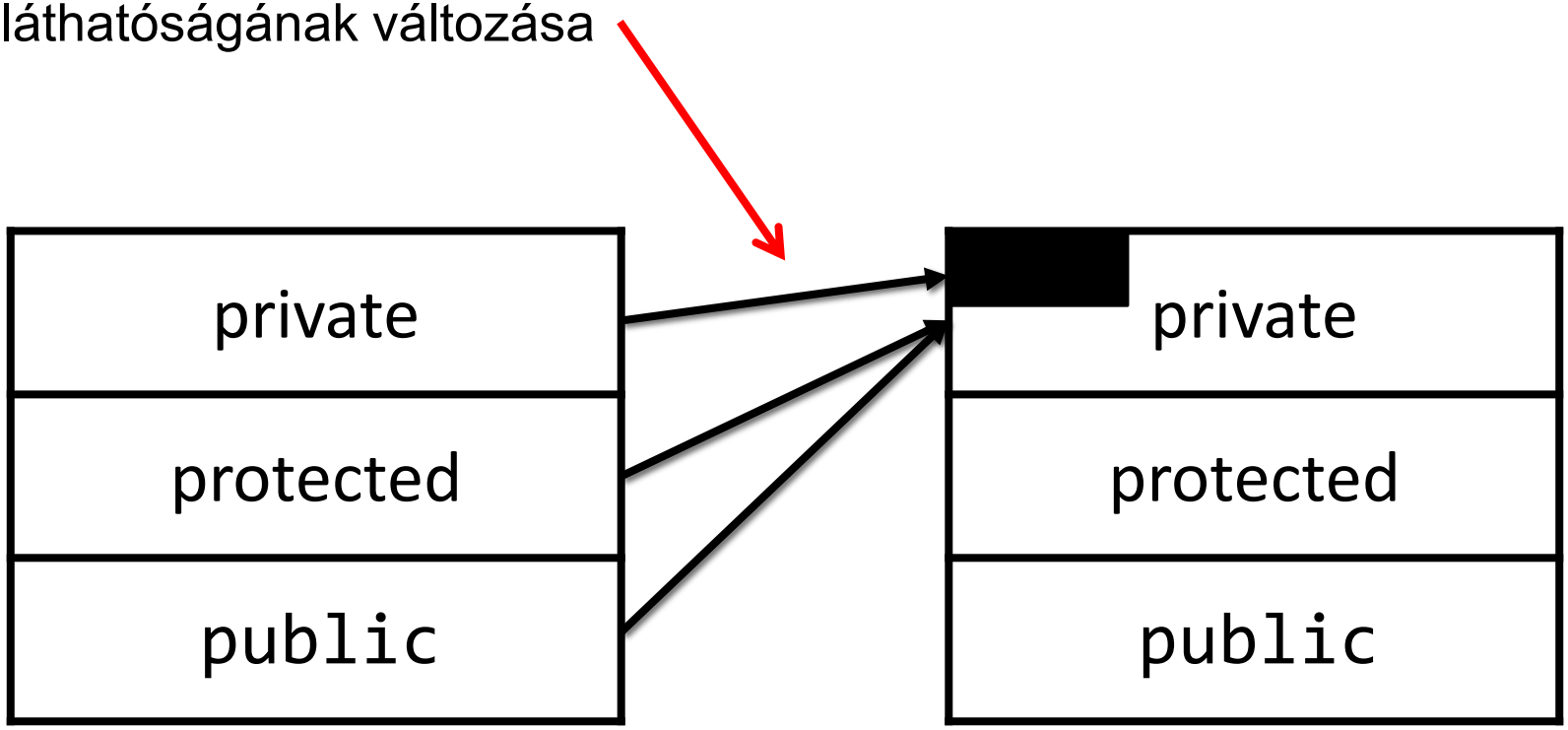



Bázisosztály

Származtatott osztály

# Private öröklés

Tagok láthatóságának változása




Bázisosztály

Származtatott osztály

# Kovariáns metódusok

Lehet bevezetni kovariáns és kontravariáns metódusokat az altípusban

- ezek elfedik (túlterhelik) az eredeti metódust, nem felüldefiniálják

```
class A {
public:
    int data;
};

class B : public A {
public:
    virtual void f(B *p)  { std::cout << "B.f(B) " << this->data << " " << p->data << std::endl; };
};

class C : public B {
public:
    // Kontravariáns
    // void f(A *p)  { std::cout << "C.f(A) " << this->data << " " << p->data << std::endl; };

    // Novariáns
    // void f(B *p) override { std::cout << "C.f(B) " << this->data << " " << p->data << std::endl;
};

    // Kovariáns
    void f(C *p)  { std::cout << "C.f(C) " << this->data << " " << p->data << std::endl; };
};
```

# Kovariáns metódusok

Az osztálydefiníciókkal a következő kód nem fordul

- ```
  A *a = new A(); a->data=1;
  B *b = new B(); b->data=10;
  C *c = new C(); c->data=100;
  C *test = c;
  test->f(b);
  ```

A problémát az okozza, hogy a C osztályban van egy f() implementáció, amely C paramétert vár.

- Ezt a fordító megtalálja, ám a szignatúra nem megfelelő.
- Nem keres tovább, mert az elfedés miatt
- Megoldás: `using B::f;`
  ```

# Kis kitérő – `struct`

Struct esetén minden tag **`public`** alapértelmezés szerint.

- Ez az információ elrejtést nem segíti, úgy mint az osztályoknál
- Azonban célszerű „sima” adatrekordok tárolására (rekord típus)
  - Plain Old Data Structure (POD)
  - C-vel kompatibilis

# C++ – Összefoglalás

Mit örököl a leszármazott?

- Adattagokat
- Metódusokat

Mit nem örököl a leszármazott?

- Ősosztály konstruktorait, destruktorát
  - Konstruktor örökíthető
  - Leszármazottban delegálható rá, használható
- Ősosztály értékadás operátorát
- Ősosztály barátait

# C++ – Összefoglalás

Mit vezethet be a leszármazott osztály?

- Új adattagokat
- Új metódusokat
- Felüldefiniálhat már meglévőket (virtual)
- Új konstruktorokat és destruktort
- Új barátokat

# C++

Egy általános metódus deklarációja a következőket jelenti:

1. A metódus elérheti a privát mezőket is
2. Az osztály scope-ját használja
3. A metódus egy konkrét objektumra hívódik meg, ezért birtokolja a „this” pointert

Statikus metódus csak az 1, 2 -vel rendelkezik,

Ha egy függvényt friend-nek deklarálunk, akkor csak az 1. jogunk lesz (friend mechanizmus)

# OOP folytatás

KÖVETKEZŐ ALKALOMMAL




# Programozási nyelvek és módszerek

OOP II.

# OOP fő elvei

OO moduláris struktúra

Adatabsztrakció

Osztályok

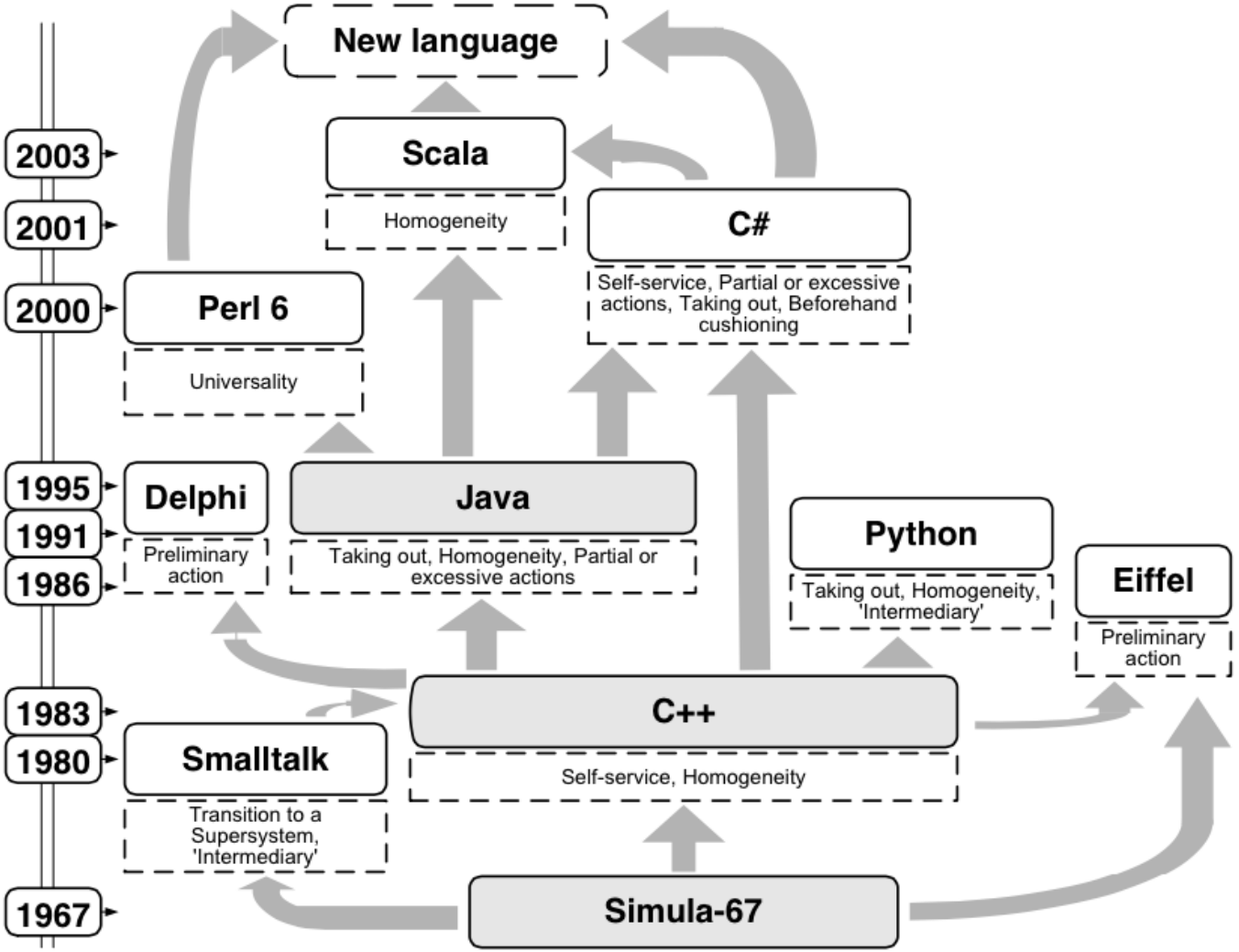
Öröklődés

Polimorfizmus és dinamikus összekapcsolás

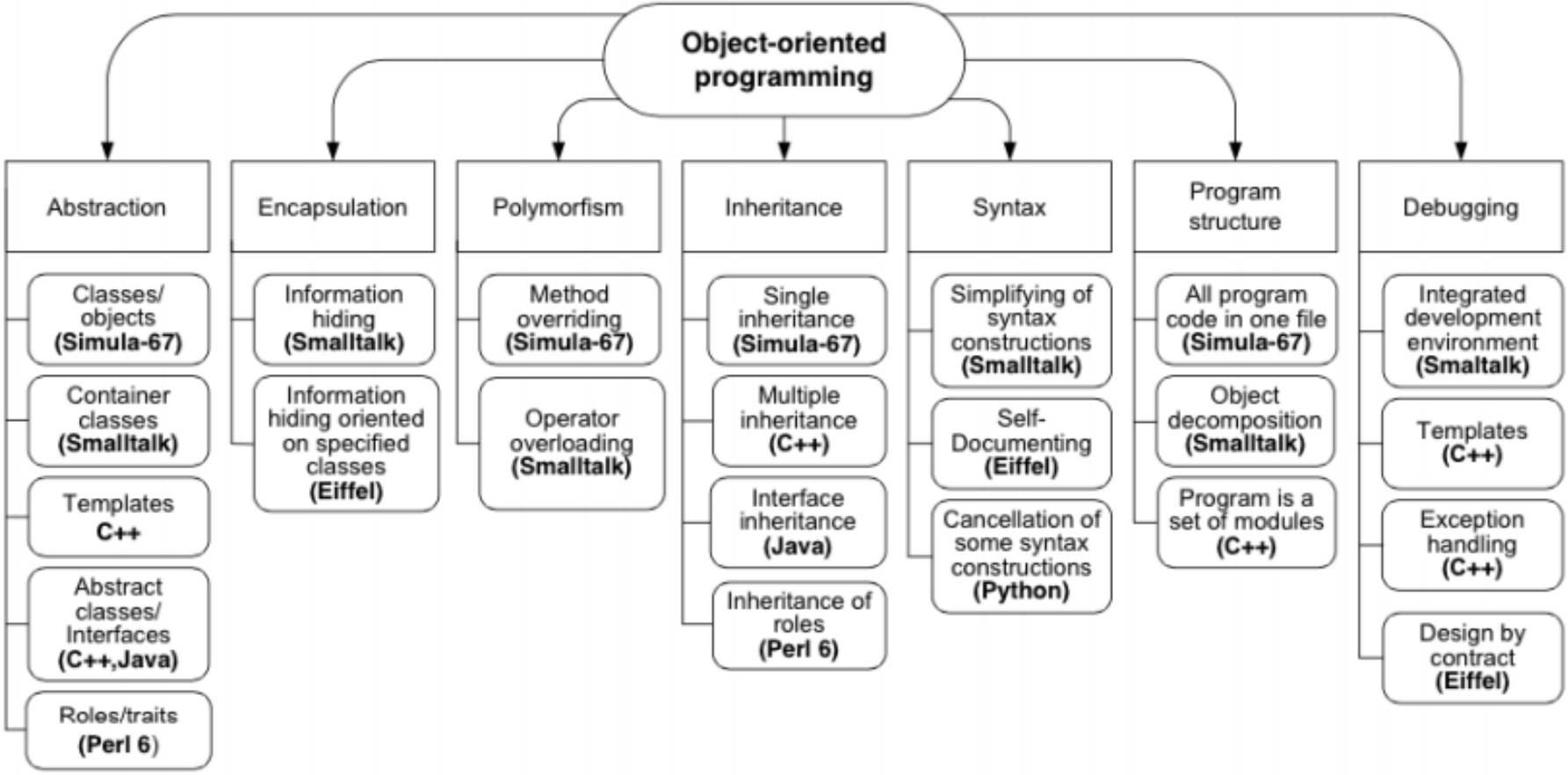
Többszörös és ismételt öröklés

Automatikus memóriakezelés

# OOP nyelvek – hatások



# OOP nyelvek – jellemzők



# Simula-67

Simula

- https://www.tutorialspoint.com/compile_simula_online.php

Itt jelenik meg először:

- Osztály (class)
- Öröklődés
- Osztályhierarchia
- A teljesen önálló típusfogalom
  - objektumhivatkozás: `ref (< osztályazonosító >)`
  - Bármely A osztály egyértelműen meghatároz egy `ref(A)` hivatkozási típust
    - ez minden objektumhoz létezik

# Simula-67

Szintaktika: a blokkok prefixelése

- Ennek hatására a blokk és az osztály fogalma összeolvad
- A blokkban az osztályban definiált publikus fogalmakat használhatjuk (SIMULATION)
- Például

```
prefix_obj begin
...
end
```

# Simula-67

```
Begin
    class Rectangle(SideA, SideB);
    real sideA;
    real sideB;
    Begin
        integer Procedure GetArea;
        Begin
            GetArea:=sideA * sideB;
        End;
    End of Square;

    Ref (Rectangle) r;
    r :- new Rectangle(10, 20);
    OutFix(r.GetArea, 2, 6);
    OutImage;
End;
```

# Simula-67

Öröklés (prefixeléssel)

```
Rectangle class ColorRectangle(Color);
text Color;
Begin
    Procedure Print;
    Begin
        OutText("Rectangle: ");
        OutFix(SideA, 2, 6);
        OutFix(SideB, 2, 6);
        OutText(" "); OutText(Color);
        OutImage;
    End;
End of ColorRectangle;

Ref (ColorRectangle) r;
r :- new ColorRectangle(20, 10, "Zöld");
r.Print;
```

# Simula-67

Tulajdonságok

- Van GC
- Nincs többszörös öröklődés
- Van `this` pointer
- `virtual`
  - Függvények felüldefiniálásához
  - Absztrakttá teszi az osztályt, ha nem adunk meg törzset
- `inner`
  - Leszármazott osztály inicializálásakor lefutó kód
- Láthatósági védelem
  - hidden és protected prefixek az attribútumok előtt

# Simula-67 Példakód

```
Begin
    class Shape;
    Virtual:
        Procedure Print is Procedure Print;;
        Procedure GetArea is integer Procedure GetArea;;
        ! Virtuális függvények;
    Begin
        OutText("Shape - Normal"); OutImage;
        ! Ez a Shape inicializálásakor fut le
    Inner;
        OutText("Shape - Inner"); OutImage;
        ! Ezek a leszármazottakhoz tartozó utasítások;
    End of Shape;
```

# Simula-67 Példakód

```
Shape class Rectangle(SideA, SideB)
real sideA;
real sideB;
Begin
    integer Procedure GetArea;
    Begin
        GetArea:=sideA * sideB;
    End;
    Procedure Print;
    Begin
        OutText("Rectangle: ");
        OutFix(SideA, 2, 6); OutFix(SideB, 2, 6);
        OutImage;
    End;
    OutText("Rectangle - Normal"); OutImage;
    ! Ez a Rectangle inicializálásakor fut le;
End of Rectangle;
```

# Simula-67 Példakód

```
    Rectangle class ColorRectangle(Color);
    text Color;
    Begin
        Procedure Print;
        Begin
            OutText("CRectangle: ");
            OutFix(SideA, 2, 6); OutFix(SideB, 2, 6);
            OutText(" "); OutText(Color); OutImage;
        End;
    End of ColorRectangle;

    Ref (Shape) r;
    r :- new ColorRectangle(20, 10, "Zöld");
    r.Print;
    OutFix(r.GetArea, 2, 6);
    OutImage;
End;
```

# Simula-67

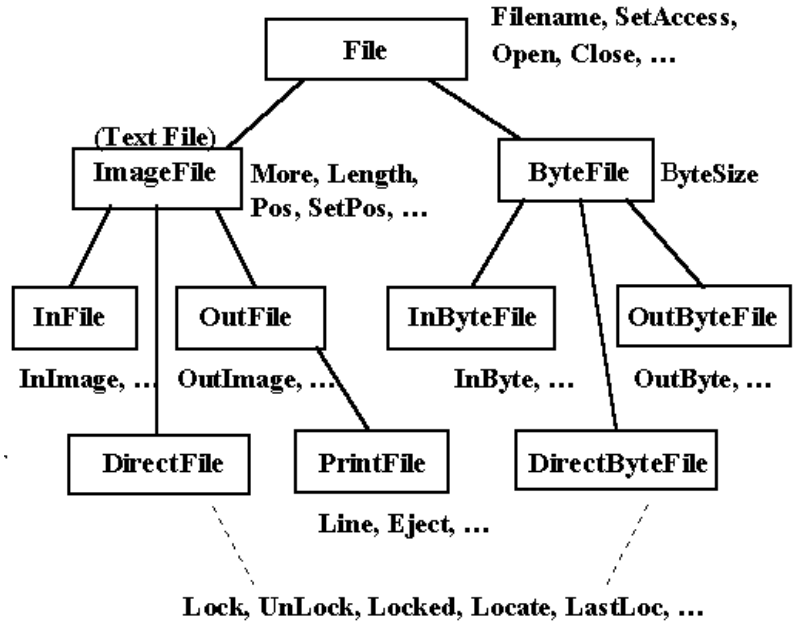
Az `inspect` utasítás a dinamikus összekapcsolás megvalósítása mellett:

```
X in osztály_1

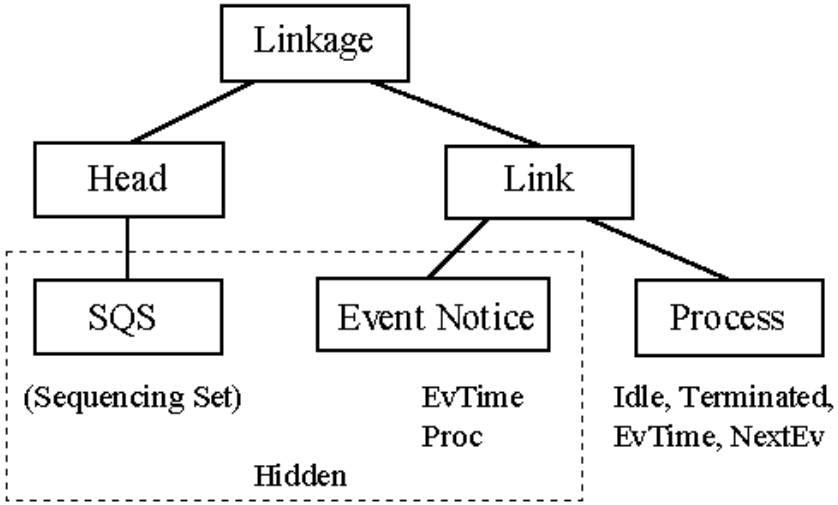
inspect X
  when osztály_1 do utasítás_1
  when osztály_2 do utasítás_2
  ...
  when osztály_n do utasítás_n
  otherwise utasítás_0
```

# Simula-67 Standard könyvtárak

BASICIO, FILE




SIMULATION, PROCESS



# SmallTalk

SmallTalk
- https://www.tutorialspoint.com/execute_smalltalk_online.php

Minden objektum
- Integer,…Character
- Az üzenetek, amelyeket az objektumoknak küldünk
- Az IDE is: ClassBrowser, Workspace, Debugger

A ST szintaxisa 1 nap alatt megtanulható
- Jó részét egy perc alatt:
  - `object message`
  - Példák
    - `myWindow drawCircle,`
    - `myAge + 1,`
    - `myWindow drawCircleofRadius: afloat, color: Red`

# SmallTalk

## Tulajdonságok

- RTTI (run-time type information)
  - Egy változóról nem tudjuk, hogy milyen típusú objektumra mutat, a változók típusnélküliek
- Szabványos objektumkönyvtárak
  - Array
  - String
  - File Stream
  - …
- VM
- GC

# SmallTalk

Legnagyobb közös osztó keresése:

```
| a b |
a := 10.
b := 12.

[ a ~= b ] whileTrue:
    [ a < b ifTrue:
            [ b := b - a ]
        ifFalse:
            [ a := a - b ]
    ].

Transcript show: (a
printString).
^a
```

Magyarázat

ciklus: logikai értéket visszaadó blokk objektumnak küldjük

- a `whileTrue:` üzenetet, aminek az argumentuma is egy blokk

elágazás: logikai értéknek küldjük

- az `ifTrue: ifFalse:` üzenetet, blokkok az argumentumok

# SmallTalk

Minden objektum

- Minden objektum egy osztály példánya

Az osztály is objektum

- Kérdés: ez minek a példánya?
- Válasz: egy metaosztály példánya
  - (metaosztály hierarchia)

# SmallTalk

Az objektumok osztályát a class üzenettel kapjuk vissza

Osztályok definiálása a szülőnek küldött üzenettel történik (subclass)

Nincs többszörös öröklődés

- nincs anomália

Láthatóság

- nincsenek láthatósági predikátumok
- a belső változók minden alosztály számára láthatóak
- De kívülre nem, viszont a metódusok globálisak
  - konvenció: a megjegyzésbe private-t írhatunk

Van osztályszintű attribútum és metódus

# SmallTalk

Közös ősosztály
- Object
  - objektum kiíratása
  - objektum osztályának meghatározása
  - copy
  - klónozás

Konstruktor
- osztálynak küldött üzenet

Destruktor nincs
- szemétgyűjtéssel oldja meg

Nincs absztrakt osztály, azaz a metódusokat definiálni kell, nem csak deklarálni

# SmallTalk

Osztály létrehozása

```
Object subclass: #Square
    instanceVariableNames: 'sidea'
    classVariableNames: 'count'
    poolDictionaries: ''
    category: nil.
```

# SmallTalk

Osztálymetódus – példányosításhoz

```
Square class extend [
    new [
        <category: 'instance creation'>
        | r |
        r := super new.
        r init.
        ^r
    ]
]
```

# SmallTalk

Példánymetódusok (inicializáláshoz)

```
Square extend [
    init [
        <category: 'initialization'>
        sidea := 0.
    ]

    init: side [
        <category: 'initialization'>
        sidea := side.
    ]
```

# SmallTalk

Példánymetódusok (alap funkciók)

```
    printOn: stream [
        <category: 'printing'>
        super printOn: stream.
        stream nextPutAll: ' with side A = '.
        sidea printOn: stream
    ]

    getArea [
        ^ sidea*sidea.
    ]
]
```

# SmallTalk

Próba

```
S := Square new.
S init: 10.
S printOn: stdout.
Transcript cr.
S getArea printOn: stdout.
```

# Object Pascal

Többféle következetlenség

- `object – class`
- `virtual` (sebesség) - `dynamic` (méret)
- Láthatósági megkötések alkalmazhatóak – de fájlon belül nem érvényesek!
  - De: strict private, strict protected lehetőség is van

Új láthatóság: published (publikált)

- ugyanazon elérési szabályok, mint a public részekre, de a fordító futási idejű típus-információkat is kapcsol hozzá

Objektumok attribútuma

- `property p: word read olvas write ír default 5;`

# class kulcsszó

## Automatikus memóriakezelés

- A Delphi referencia modellt használ az osztályok esetén
  - Az ilyen objektum mindig dinamikus, implicit referencia.

## Közös ős

- Minden osztály a `TObject` ősosztály leszármazottja.

## Absztrakt metódusok

- Ha a metódus deklarációjában az `abstract` kulcsszót használjuk, nem kell törzset megadni – de az ilyen osztálynak is lehet példánya!

# Object Pascal

```
type
  TShape = class
  public
     constructor Create;
     function Print: string; virtual; abstract;
     function GetArea: Single; virtual; abstract;
  end;

type TRectangle = class (TShape)
  private
     sideA: Single;
     sideB: Single;
  public
    constructor Create;
    constructor Create(side_A: Single; side_B: Single);
    function Print: string; override;
    function GetArea: Single; override;
  end;
```

# Object Pascal

```
constructor TShape.Create;
begin
    WriteLn('Shape Create')
end;
constructor TRectangle.Create;
begin
     self.Create(0.0, 0.0)
end;
```

# Object Pascal

```
constructor TRectangle.Create(side_A: Single; side_B: Single);
begin
     self.sideA := side_A;
     self.sideB := side_B;
end;
function TRectangle.Print: string;
begin
    Result := 'Rectangle: ' + FloatToStr(self.sideA) + ' ' +
FloatToStr(self.sideB);
end;
function TRectangle.GetArea: Single;
begin
    Result := self.sideA * self.sideB;
end;
```

# Object Pascal

```
var
  s: TShape;
  r: TRectangle;

begin
  s := TRectangle.Create(10, 20);
  WriteLn(s.GetArea);

  ReadLn();
end.
```

# Object Pascal – Ős konstruktora

```
type TOs = class
    a, b: Integer;
    constructor Letrehoz(x, y: Integer);
end;
type TUj = class(Tos)
    c: Integer;
    constructor Letrehoz(x, y, z: Integer);
end;
constructor TOs.Letrehoz(x, y: Integer);
begin
    a:=x; b:=y;
end;
constructor TUj.Letrehoz(x, y, z: Integer);
begin
    inherited Letrehoz(x, y); c:=z;
end;
```

# Object Pascal – virtuális konstruktor

```
TFoo = class
 public
  constructor Create; virtual;
 end;
TBar = class(TFoo);
 public
  constructor Create; override;
 end;
```

```
TFooClass = class of TFoo;
 // class reference type
 var
  fc: TFooClass;
  afoo: TFoo;
 begin
  fc := TFoo;
  afoo := fc.Create;
  // returns a TFoo
  fc := TBar;
  afoo := fc.Create;
  // returns a TBar
```

# Object Pascal – sealed osztály

**sealed** – nem származtatható belőle

```
type
  TMyClass = class sealed
    procedure SomeProcedure;
end;
```

**final** – ha metódus

```
procedure MyMethod(const AMyParameter: Integer);
override; final;
```

# Object Pascal – interfészek

```
type
IDrawable = interface
    procedure draw();
end;
```

Az interface-ben:

- csak a metódusok fejrésze van leírva
- tartalmazhat property-ket is, de ezek szintén nincsenek kidolgozva, csak a property neve van megadva, típusa, és az, hogy írható vagy olvasható,
- mezőket, konstruktort, destruktort nem tartalmazhat
- minden rész az interface-ben automatikusan publikus,
- a metódusokat nem lehet megjelölni virtual, dynamic, abstract, override kulcsszavakkal,
- az interface-eknek lehetnek ősei, melyek szintén interface-ek.

# Objective-C

Objektumorientált nyelv

- a C programozási nyelvet egészíti ki
- a Smalltalk üzenetküldési lehetőségeivel.

1986-ban jelent meg, Brad Cox és Tom Love tervezték

Apple által továbbfejlesztett és támogatott

- Mostanra megvan az utódja is …

# Objective-C

## Osztályok

- Kötelező szétválasztani az interfészt és az implementációt.
- Az interfész megadja a specifikációt
  - Deklarálja az osztály adattagjait, metódusait
  - Megnevezi az ősosztályát.
- Az implementációban pedig definiáljuk a metódusokat
  - Azaz az osztályt magát adjuk meg.

# Objective-C

Az osztály interfész része

```
@interface ClassName : ItsSuperclass
{
      float width;
      float height;
      BOOL filled;
      NSColor *fillColor;
      //...
}   //itt az adattagok
      + alloc;          // osztálymetódus előtt +
      - (void)display; // példánymetódus előtt –
      - (void)setWidth:(float)width height:(float)height;
      - makeGroup:group, ...;
@end
```

# Objective-C

Az osztály implementáció része

```
#import "ClassName.h„
@implementation ClassName
    + alloc
    {           //...
    }
    - (void)display
    {           //...
    }
    - (void)setWidth:(float)width height:(float)height
    {           //...
    }
    - makeGroup:group, ...
    {
       va_list ap;
       va_start(ap, group);
       //...
    }
@end
```

# Objective-C – üzenetküldés

Itt a metódusok/függvények hívása helyett az objektumok közötti üzenetküldés kerül előtérbe

- `[foo bar:parameter];`

Az hogy az üzenetet fel tudja-e dolgozni az objektum, az futásidőben dől el

- Ahogyan a típusellenőrzés sem történik meg, csak futásidőben

Az üzenetküldés során a paraméterek a függvény nevével adhatók meg

```
-(type)method:(type)param1 andParam2:(típus)param2;
```

# Objective-C

Láthatóság szabályozása a következő direktívákkal lehetséges

- `@public`
- `@private`
- `@protected`
  - Alapértelmezett
- `@package`

# Objective-C

## Öröklődés

- Egyszeres öröklődés
- Nincs absztrakt osztály
- `NSObject` a legfelső szinten

## Interface

- `@protocol`

## Kategóriák

- Metódusok egy halmaza, amelyet „csomagban” lehet egy osztályhoz hozzáadni.
- Nem azonos az interfésszel, vagy öröklődéssel.

## polimorfizmus, dinamikus kötés – automatikusan

# Objective-C

```
@protocol Archiving
- read: (FILE *) f;
- write: (FILE *) f;
@end

@protocol ReferenceCounting
- incrementCount;
- decrementCount;
- (unsigned) refCount;
@end

@interface Shape : NSObject <Archiving, ReferenceCounting>
```

# Objective-C

## Kategóriák

- A kategória segítségével már meglévő osztályokhoz adhatunk hozzá metódusokat
  - akkor is, ha nincs meg az osztály forrása
- Ez egy rendkívül erős eszköz, amellyel öröklés nélkül terjeszthetjük ki az osztályok funkcionalitását
  - akár beépített osztályokét is
- Az új függvények az osztály meglévő interfészét egészítik ki
  - mixing -> mixin-s

# Objective-C

```
#import "ClassName.h"
@interface ClassName ( CategoryName )
  // method declarations
@end

#import "ClassName+CategoryName.h"
   // ilyen neve legyen!
@implementation ClassName ( CategoryName )
  // method definitions
@end
```

# Objective-C

Példa – NSString minden objektumának legyen isURL metódusa:

```
@interface NSString (Utilities)
- (BOOL) isURL;
@end

#import "NSString+Utilities.h"
@implementation NSString (Utilities)
- (BOOL) isURL
{
      if ( [self hasPrefix:@"http://"] )
            return YES;
      else
            return NO;
}
@end
```

# Objective-C

Most már bármelyik NSString-re alkalmazhatjuk ezt a metódust0

```
NSString* string1 = @"http://pixar.com/";
NSString* string2 = @"Pixar";

if ( [string1 isURL] )
  NSLog (@"a string1 egy URL");
if ( [string2 isURL] )
  NSLog (@"a string2 egy URL");
```

# Java

Smalltalk-kal rokonság

GC (VM dolga)

Objektumelvűség

- A Javában gyakorlatilag minden objektum (osztály)
- Primitív típusokhoz csomagoló („wrapper”) osztály tartozik
  - **`boolean`**, **`char`**, **`byte`**, **`short`**, **`int`**, **`long`**, **`float`**, **`double`**
  - Felelős a konverzióért, kiíratásért, stb.
    - immutable osztályok
    - **`boolean`** -> `Boolean` (nagybetű a különbség)
    - **`char`** -> `Character`
    - **`int`** -> `Integer`
  - Automatikus konverzió: boxing-unboxing

# Java

Nincs osztályon kívüli (globális) változó

Öröklődési fa gyökere az `Object` osztály

Nincs többszörös öröklődés

- Interfészekkel részben helyettesíthető
  - Minden függvénye publikus és absztrakt
  - Konstans változókat deklarálhatunk benne
  - Többszörös interface öröklés engedélyezett
- **`extends`** – **`implements`**

**`instanceof`**

- új operátor a C++ -hoz képest

# Java

## Módosító kulcsszavak

- **`final`**
  - Osztálynév előtt: tiltja a további származtatást
  - Metódus előtt: tiltja a felüldefiniálást
  - Változó előtt: tiltja a kezdeti értékadás után az érték megváltoztatását
    - Objektum esetén referencia
- **`abstract`**
  - Metódus előtt: a metódus deklarálható, de nem definiálható az adott osztályban, csak a származtatottban
  - Osztály előtt: az osztálynak van **`abstract`** metódusa (nem kötelező kiírni, de az osztály akkor is **`abstract`** lesz implicite)
- **`static`**
  - Osztálymetódus, illetve osztályváltozó jelzésére
- Egy osztály nem lehet egyszerre **`final`** és **`abstract`**

# Java láthatóság

| Hol | | Kulcsszó | | | |
|---|---|---|---|---|---|
| Package | Osztály | public | protected | no modifier default | private |
| Azonos csomag | Az osztály maga | X | X | X | X |
| | Belső osztály | X | X | X | X |
| | Leszármazott | X | X | X | |
| | Bárki | X | X | X | |
| Másik csomag | Leszármazott | X | X | | |
| | Bárki | X | | | |

# Java – inicializálás

Nincs operátor túlterhelés

- De függvény túlterhelés van

Ha egy osztálynak nincs konstruktora, a fordító a következőt generálja neki:

```
class MyClass extends SuperClass {
      MyClass() {  super();  }
}
```

Az ősosztály konstruktora a leszármazott lefutása előtt történik

- Nem kell expliciten megadni, ilyenkor az azonos paraméterezésűt futtatja

# Java – inicializálás

Lehetséges inicializációkat is beszúrni a kódba

- A mezőknél kezdeti érték adásával
- Blokkokkal

Lefutási sorrend

- Ősosztály statikus inicializációja
- Leszármazott osztály statikus inicializációja
- Ősosztály inicializációs kódja(i)
- Ősosztály konstruktora
- Leszármazott osztály inicializációs kódja(i)
- Leszármazott osztály konstruktora

# Java – finalize()

Destruktorok: a nyelvben nem szerepelnek

- helyette a `finalize()`
- a GC hívja felszámolás előtt

# Java

```
public abstract class Shape {
    public abstract double calculateArea();
    public abstract void print();
}

public interface Polygon {
    double[] getEdges();
}
```

# Java

```java
public class Circle extends Shape{
    public static final double PI = 3.1415926535;

    private double radius;

    public Circle(double r) {
        this.radius = r;
    }

    @Override
    public void print() {
        System.out.println("Circle, radius: " + this.radius);
    }

    @Override
    public double calculateArea() {
        return this.radius * this.radius * Circle.PI;
    }
}
```

# Java

```
public class Parallelogram extends Shape implements Polygon {
    protected double _side_a;
    protected double _side_b;
    protected double _delta;
    protected double _height_a;

    public double getSideA() { return this._side_a; }
    public double getSideB() { return this._side_b; }
    public double getAngle() { return this._delta;  }

    public boolean isEquilateral() { return
                                     this._side_a==this._side_b; }
    public boolean isRight()  { return this._delta == 90; }

    public Parallelogram(double a, double b, double angle) {
        this._side_a = a;
        this._side_b = b;
        this._delta = angle;
        this._height_a = b *
                         Math.sin(this._delta * Math.PI / 180.0);
    }
```

# Java

```
    @Override
    public void print() {
        System.out.println("Parallelogram a, b, delta: "
                + this._side_a + ", "
                + this._side_b + ", "
                + this._delta);
    }

    @Override
    public double calculateArea() {
        return this._side_a * this._height_a;
    }

    @Override
    public double[] getEdges() {
        return new double[] {_side_a, _side_b, _side_a, _side_b};
    }
}
```

# Java

```
public class Main {

    public static void main(String[] args) {
        Rectangle r = new Rectangle(10, 30);
        Shape s = r;
        s.print();
        System.out.println(s.calculateArea());

        Polygon p = r;
        System.out.println(p.getEdges().length);
    }
}
```

# Java – default

```
public interface Addressable
{
    String getStreet();
    String getCity();

    default String getFullAddress()
    {
        return getStreet()+", "+getCity();
    }
}
```

# Java – default

```
public class Letter implements Addressable
{
    private String street;
    private String city;
    public Letter(String street, String city)
    {
        this.street = street;
        this.city = city;
    }
    @Override
    public String getCity() { return city; }
    @Override
    public String getStreet() { return street; }

    public static void main(String[] args)
    {
        Letter l = new Letter("123 AnyStreet", "AnyCity");
        System.out.println(l.getFullAddress());
    }
}
```

# Java – többszörös implementáció

```
public interface Interface2 {

    default void doIt(){
        System.out.println("I2.doIt");
    }
}
public interface Interface1 {

    default void doIt(){
        System.out.println("I1.doIt");
    }
}
```

# Java – többszörös implementáció

```
public class TheClass
        implements Interface1, Interface2 {

    public void doIt()
    {
        Interface2.super.doIt();
        Interface1.super.doIt();
    }
}
```

# Java – Annotációk

Metaadatokat lehet a kódhoz fűzni

- Információ a fordítónak
  - Például felüldefiniálás esetén
    - `@Override`
  - Figyelmeztetések ignorálására
  - …
- Deployment-nél felhasználható adatok
- Futásidőben használható adatok, ellenőrzésre
  - `myString = (@NonNull String) str;`

# C#

OOP támogatása nagyon hasonlít a Javáéhoz

- Egyszeres öröklődés
- Interfészek (itt lehet többszörös öröklődés)

Eltérések, újdonságok

- statikusan és dinamikusan kötött metódusok is megadhatók
  - `virtual` kulcsszóval meg kell jelölni a felüldefiniálható függvényt, különben elfedi a leszármazott
  - Felüldefiniált metódust jelölni kell: `override`
- A közvetlen ősosztályra a `base` kulcsszóval hivatkozhatunk
- A `sealed` kulcsszóval megakadályozhatjuk, hogy
  - egy osztályból származtassanak
  - egy metódust felüldefiniáljon a származtatott osztály

# C#

## Eltérések, újdonságok

- Egy metódus lehet `sealed override`
  - Ekkor a leszármazott nem definiálhatja felül
  - De elfedheti, ezt jelölnie kell a `new` kulcsszóval
    - Polimorfizmus esetén vigyázni kell!
- Az `external` módosítóval rendelkező metódusok valamilyen más nyelven vannak implementálva.
  - Éppen ezért a metódus törzsében csak egy pontosvessző áll.
  - Az ilyen metódus nem lehet `abstract`.

# C#

## Konstruktorok

- Közvetlenül a konstruktor törzsének végrehajtása előtt automatikusan történik egy másik konstruktor hívás is
  - Ezt nevezik konstruktor inicializálásnak is.
- Két lehetőségünk van
  - Az adott osztály egy másik konstruktorát hívjuk meg először.
  - Meghívjuk az ős valamelyik példány konstruktorát
    - Ha nem használjuk egyiket sem, akkor az ős alapértelmezett konstruktora hívódik meg.
- Azaz a következő két deklaráció ekvivalens egymással:

```
C( ... ) {...}
C( ... ): base() {...}
```

# C#

## Konstruktorok

- Ha egy osztálynak nem deklarálunk semmilyen példány konstruktort, akkor egy default konstruktor jön létre
  - E az alapértelmezett konstruktor inicializátorral fog rendelkezni.
- Készíthetünk `private` konstruktorokat is
  - Ebben az esetben az osztály nem példányosítható és nem lehet örököltetni sem belőle.
  - Akkor célszerű ezt használni, ha például csak statikus elemeket tartalmaz az osztályunk.
- Statikus konstruktorok
  - A konstruktor neve előtt a `static` kulcsszót kell használni.
  - A statikus konstruktorok az osztály inicializálásakor futnak le, amikor az osztály betöltődik.
  - Ezekre a konstruktorokra nem lehet hivatkozni és nem is öröklődnek.

# C#

## Destruktorok

- Az objektumok megsemmisülésekor hívódnak meg.
  - Automatikusan hívódnak meg, a GC miatt
  - Az öröklődési láncon végighaladva egymás után hívódnak meg.
    - A szemétgyűjtő nyilvántartja azon objektumok listáját, amelyek osztályában definiáltunk destruktort.

```
~MyClass(){}
```

  - A fordító ezt generálja belőle

```
protected override void Finalize()
{
  try {
   …
  }
  finally {
    base.Finalize();
  }
}
```

# C#

Szemétgyűjtés felülbírálható az `IDisposable` interfész megvalósításával

- Ekkor kell egy `Dispose()` metódus.
- Nem lehetünk biztosak benne, hogy a felhasználó megbízhatóan hívja…
- A using utasítást vezették be, hogy biztosan lefusson.

# C#

## Property

- Definiálhatók elérők (accessor), melyek tulajdonságként viselkednek, de az írás (set) illetve olvasás (get) számára külön eljárások írhatók.
- Például, ha egy elérőre csak a get eljárást definiáljuk, akkor az elérő egy csak olvasható attribútumként fog viselkedni.

# C#

## Absztrakt osztályok

- Az `abstract` kulcsszó használatával azt jelezhetjük, hogy az adott osztály még nincs teljesen megvalósítva.
- Egy absztrakt osztály nem lehet `sealed`.
- Egy absztrakt metódus
  - Automatikusan `virtual`
  - Nem lehet `static`
- A származtatott osztály tartalmazza az absztrakt metódusok implementációt
  - Nem absztrakt ősmetódusra tud hivatkozni a `base` kulcsszóval

# C#

## Az öröklődés

- `class SubClass: BaseClass {..}`
- Utód osztályban nem lehet szűkebb a hozzáférés, mint ős osztályban
- A `base` kulcsszóval a közvetlen ősosztály metódusa elérhető
- A `virtual` függvényeknél a `new` kulcsszó használatával elfedhetjük az ős `virtual` függvényét
  - Ekkor egyidejűleg nem lehetséges az ős függvény elfedése (`override`)

# C#

## Interfészek

- Nincs többszörös öröklődés C#-ban, helyette interface-k vannak.
  - Az interfészek között lehetséges a többszörös öröklődés.
- Az interfész definiálhat: (De nem valósíthatja meg)
  - Metódusokat
  - Indexelőket
  - property-ket
  - eseményeket
- Egy interfésznek több más interfész is lehet őse, egy osztály pedig több interfészt is megvalósíthat.
- A interfész tagjaira az `abstract`, `virtual`, `static`, `override` módosítók nem használhatók.

# C# – Példa

```
abstract class Shape {
    public abstract double CalculateArea();
    public abstract void Print();
}


interface IPolygon {
    double[] GetEdges();
}
```

# C# – Példa

```
class Circle : Shape {
    static double PI = 3.1415926535;
    // const alapértelmezetten

    private double radius;

    public Circle(double r) {
        radius = r;
    }

    public override void Print() {
        Console.WriteLine("Circle, radius: " + radius);
    }

    public override double CalculateArea() {
        return radius * radius * Circle.PI;
    }
}
```

# C# – Példa

```
class Parallelogram : Shape, IPolygon
{
    protected double _side_a;
    protected double _side_b;
    protected double _delta;
    protected double _height_a;

    private void UpdateHeightA()
    {
        _height_a = _side_b * Math.Sin(_delta * Math.PI / 180.0);
    }

    public double SideA
    {
        get => _side_a;
    }
```

# C# – Példa

```
public double SideB
{
    get { return _side_b; }
    set
    {
        _side_b = value;
        UpdateHeightA();
    }
}

public double Angle
{
    get => _delta;
    set
    {
        _delta = value;
        UpdateHeightA();
    }
}

public virtual bool isEquilateral() { return _side_a==_side_b; }
public virtual bool isRight()  { return _delta == 90; }
```

# C# – Példa

```
    public Parallelogram(double a, double b, double angle) {
        _side_a = a;
        SideB = b;
        Angle = angle;
        UpdateHeightA();
    }

    public override void Print() {
        Console.WriteLine("Parallelogram a, b, delta: "
                          + _side_a + ", "
                          + _side_b + ", "
                          + _delta);
    }

    public override double CalculateArea() {
        return_side_a *_height_a;
    }

    public double[] GetEdges() {
        return new double[] {_side_a, _side_b, _side_a, _side_b};
    }
}
```

# C# – Példa

```
sealed class Rectangle : Parallelogram
{
    public Rectangle(double a, double b) : base(a, b, 90) {

    }

    public override void Print() {
        Console.WriteLine("Rectangle a, b: "
                          + _side_a + ", "
                          + _side_b);
    }

    public override bool isRight() {
        return true;
    }
}
```

# C# – Példa

```
class Program
{
    static void Main(string[] args)
    {
        Parallelogram pa = new Parallelogram(10, 5, 90);
        Console.WriteLine(pa.CalculateArea());
        pa.Angle = 45;
        Console.WriteLine(pa.CalculateArea());

        Rectangle r = new Rectangle(10, 30);
        Shape s = r;
        s.Print();
        Console.WriteLine(s.CalculateArea());

        IPolygon p = r;
        Console.WriteLine(p.GetEdges().Length);
    }
}
```

# C# – Függvények öröklésnél

```
public class CA {
    public virtual void f() { }
}

public class CB : CA {
    public override void f() { }    //felüldefiniálja
}

public class CC : CA {
    private new void f() { }        //elrejti, nem virtuális függvénnyel
}

public class CD : CA {
    public new virtual void f() { } // új virtuális függvény!
}

public class CE : CA {
    public sealed override void f() { } //nem lehet többször átdefiniálni
}
```

# C# – Interfészek megvalósítás

```
interface I1
{
    void MyFunction();
}

interface I2
{
    void MyFunction();
}

class Test : I1, I2
{
    public void MyFunction()
    {
        Console.WriteLine("Which interface?");
    }
}
```

# C#

Tetszőleges statikus típus esetén az egyetlen implementáció fut le:

```
Test t1 = new Test();
t1.MyFunction();
I1 i1 = t1;
i1.MyFunction();
I2 i2 = t1;
i2.MyFunction();
```

Ám adható a két interfész megvalósításához különböző implementációs kód …

# C#

```
class Test2 : I1,I2
{
    void I1.MyFunction()
    {
        Console.WriteLine("I1 implemented!");
    }

    void I2.MyFunction()
    {
        Console.WriteLine("I2 implemented!");
    }
}
```

Ekkor a következővel probléma van, mivel nincs rá implementáció:

```
Test2 t1 = new Test2();
t1.MyFunction();
```

# C#

Lehetőség van operátor-túlterhelésre is

- csak **public static** lehet

```
public static Digit operator+(Digit a, Digit b)
{
      return new Digit(a.value + b.value);
}
```

# C#

## Indexelők:

- Definiálhatók indexelők (indexer) is, melyekkel szögletes zárójellel indexelhető osztályok hozhatók létre.
  - Az indexelő paramétere bármilyen típus lehet.
- Bármilyen objektum úgy indexelhető, mint egy tömb.
- Az indexelő deklarálásakor meg kell adni
  - Az elemek típusát
  - A this-t a formális paraméterlistával.
  - Ez a paraméterlista határozza meg az indexelés szintaxisát.

# C#

```
class Grid {
  const int NumRows = 26;
  const int NumCols = 10;
  int[,] cells = new int[NumRows, NumCols];
  public int this[char c,int colm]  {
    get
    {
      c = Char.ToUpper(c);
      if (c < 'A' || c > 'Z') throw new ArgumentException();
      if (colm < 0 || colm > = NumCols) throw new IndexOutOfRangeException();
      return cells[c - 'A', colm];
    }
    set
    {
      c = Char.ToUpper(c);
      if (c < 'A' || c > 'Z') throw new ArgumentException();
      if (colm < 0 || colm > = NumCols) throw new IndexOutOfRangeException();
      cells[c - 'A', colm] = value;
    }
  }
}
```

# Eiffel

Lehetőségek

- Többszörös öröklődés
- Ismételt öröklődés
- A metódusok alapértelmezetten virtuálisak
- Van absztrakt osztály és metódus

https://www.eiffel.org/codeboard

# Eiffel

Osztályhierarchia:

GENERAL

↑

PLATFORM

↑

ANY

↑

+ NONE osztály

# Eiffel – Láthatóságszabályozás

```
class A
  feature {B, C}
     X: INTEGER;
  feature {ANY}
    make(p_name : STRING ) is
      require
           p_name /= ""
      do
           name := p_name ….
  feature {NONE}
     Y: ARRAY[INTEGER];
end
```

B és C látja (és utódaik)

public

private

# Eiffel

rename

- átnevezés lehetősége

```
class D inherit
        C
           rename a as ac
            end
        B
  feature
  …
end
```

# Eiffel

Példa:

```
class ACCOUNT
creation
    make
feature
    balance: INTEGER
    owner: STRING
    number: STRING

    deposit( sum: INTEGER )
    require
          sum > 0
    do
          balance := balance + sum
    ensure
          balance = old balance + sum
    end -- deposit
```

# Eiffel

```
withdraw( sum: INTEGER )
require
      sum > 0
      sum <= balance
do
      balance := balance - sum
ensure
      balance = old balance - sum
end – withdraw
```

# Eiffel

```
feature
 make( owner_, number_: STRING )
    require
          valid_owner:   owner_ /= Void
          valid_number: number_ /= Void
    do
          owner  := owner_
          number := number_
    ensure
          owner = owner_ and number = number_
          balance = 0
    end -- make

invariant
    balance >= 0
    owner /= Void
    number /= Void
end --class ACCOUNT
```

# Eiffel

```
class SAVINGS_ACCOUNT
inherit
    ACCOUNT
        rename
            make as account_make
        end
creation
    make
feature
    interest: INTEGER
    set_interest( interest_: INTEGER )
      require
          interest_ >= 0
      do
          interest := interest_
      ensure
          interest = interest_
    end -- set_interest
```

# Eiffel

```
pay_interest
        do
                deposit(balance*interest//100)
        ensure
                balance = old (balance + balance*interest//100)
        end
```

# Eiffel

```
feature
    make( owner_, number_: STRING; interest_: INTEGER )
    require
            valid_owner:   owner_ /= Void
            valid_number: number_ /= Void
            valid_interest: interest_ >= 0
    do
            account_make(owner, number_)
            set_interest(interest_)
    ensure
            owner = owner_ and number = number_
            balance = 0
            interest = interest_
    end -- make
invariant
    interest >= 0
end --class SAVINGS_ACCOUNT
```

# Eiffel – többszörös öröklődés

A leszármazott mondja meg, hogy hogyan szeretné az örökölt attribútumokat és metódusokat felhasználni (anomália megoldása)

Ez lehet:

- átnevezés
  - `rename` kulcsszó
- export státusz megváltoztatása a származtatottban
- metódus felüldefiniálása
  - pl. csak az előfeltételt, vagy az implementációt
- műveletek absztrakttá tétele
  - `deferred` - késleltetett
- ismételt öröklődés
  - ismételt öröklődésnél csak egyet örököl, vagy megmondhatja, hogy legyen mindkettő, ha akarja

# Eiffel

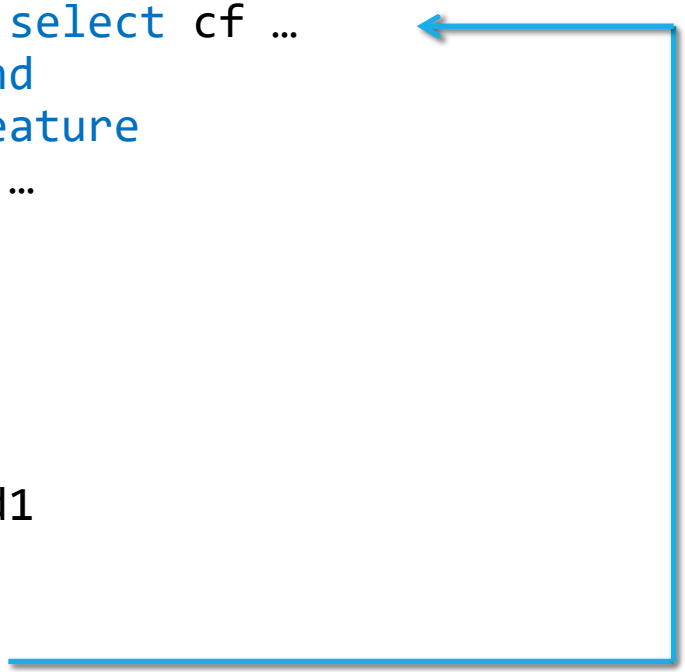
```
class A …
  feature
    f …
end
```

```
class B inherit A
   redefine
        f …
   end
   feature
        f …
   …
end
```

```
class C inherit A
   redefine
        f …
   end
   feature
        f …
   …
end
```

# Eiffel (select)

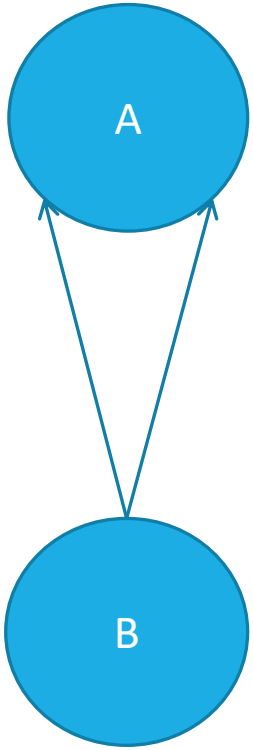
```
class D inherit
      B
        rename f as bf …
      end
      C
        rename f as cf
        select cf …
      end
      feature
        …
end
…
a1:A;
d1:D
…
create d1
d1.bf
a1:=d1
a1.f
```



# Eiffel – ismételt öröklődés

```
class A
 feature
   f …
end -- class A


class B
  inherit A
    redefine f
      select f
  end
  inherit  A
    rename f as old_f
  end
  feature
    f …
    do
      old_f …
    end - f
 end -- class B
```



# Eiffel – Példa

```
deferred class
    Shape

    feature
        calculateArea: INTEGER
            deferred
            end

        printName
            deferred
            end
end
```

# Eiffel – Példa

```
class
    Square
    inherit
        Shape
    create
        make
    feature {NONE}
        sideA: INTEGER
    feature
            make (a: INTEGER)
            do
                sideA := a
            end
            calculateArea: INTEGER
              do
                  Result := sideA * sideA
              end
             printName
              do
                  print("Square ")
                  print(sideA)
                  print("%N")
              end
end
```

# Eiffel – Példa

```
class
    APPLICATION
inherit
    ARGUMENTS_32
create
    make
feature {NONE} -- Initialization
    make
        local
            s : Shape
            sq: Square
        do
            create sq.make(5)
            s := sq
            s.printName
            print(s.calculateArea)
        end
end
```

# Eiffel

Lehet bevezetni kovariáns metódusokat az altípusban

Példa:

```
class
      TeliSportolo
      feature
         szobatars (s: TeliSportolo)
         do
             print("TeliSportolo szobatars%N")
             s.printname
         end
         printname
         do
             print("TeliSportolo vagyok%N")
         end
end
```

# Eiffel

```
class
      Sielo
         inherit TeliSportolo
         redefine
            szobatars, printname
         end
      feature
         szobatars (s: Sielo)
         do
             print("Sielo szobatars%N")
             s.printname
         end
         printname
         do
             print("Sielo vagyok%N")
         end
end
```

# Eiffel

Főprogram

```
ts: TeliSportolo
tt: TeliSportolo
si: Sielo

create ts
ts.szobatars(ts)
create si
si.szobatars(si)
-- nem lehetséges, a formális paraméter miatt
-- si.szobatars(ts)

tt := si
-- problémás! (Nem típusbiztos!)
tt.szobatars(ts)
```

# Eiffel

Gyakran a kovariancia kódismétlést okoz

- Ezt Anchorokkal lehet kiküszöbölni, az előzőekkel azonos eredményt lehet elérni:

```
class
      TeliSportolo
         feature szobatars (s: like Current)
         do
              print("TeliSportolo szobatars%N")
              s.printname
         end
        …
end
class
      Sielo
         inherit TeliSportolo
         redefine
              printname
         end
      …
end
```

# Python

Szkriptnyelv, interpretált.

- Elsősorban a kód olvashatóságára koncentrál, valamint a programozó koncepciójának rövid megfogalmazására
- Az OOP-t támogatja – szükséges mértékben

Tulajdonságok

- Az osztályok is objektumok
- Osztályhoz tartozó nevet bármikor és bárhol be lehet vezetni
- Absztrakt függvény nincs
  - Dekorátorral jelezhető
- Láthatósági szabályozás nincs
  - Névkonvenciók
- Többszörös öröklődés

# Python

## Tulajdonságok

- Új adattagot bármikor bevezethetünk és törölhetünk
  - `self.adattag = érték`
  - `del self.adattag`
- Property-k
  - Decorator-okkal megadhatók propertyk
  - get és set függvények külön

# Python

## Tulajdonságok

- Konstruktor
  - ha létezik egy `__init__()` inicializáló metódusa az osztálynak, akkor példányosításkor az objektum létrehozása után meghívódik
- __del__
  - Az objektumhoz rendelt memória terület felszabadításakor lefutó függvény
- Osztályváltozó
  - Osztály definíciójában (és nem függvényében) deklarált változó
- Osztálymetódus
  - `@classmethod` annotációval ellátott metódus, első paramétere az osztály `cls`
- Dinamikus típusosság
  - Polimorfizmusról a megszokott formában nincsen szó
  - Dinamikus kötésről sem, de névfeloldás futásidőben

# Python

## Öröklődés

- Többszörös öröklődés lehetséges
- Névfeloldásra két módszer

## Python2

- Ha egy hivatkozást nem talál az aktuális osztályban, akkor első közvetlen ősben keresi
  - Ha ott sem, akkor az első ős ősében
- Ezután ha még mindig nem találta, akkor a második közvetlen ősben
  - Stb.
- Lényegében mélységi keresés

# Python

## Öröklődés

- Többszörös öröklődés lehetséges
- Névfeloldásra két módszer

## Python3

- Ha egy hivatkozást nem talál az aktuális osztályban, akkor közvetlen ősökben keresi
  - Az ősosztályok definiálási sorrendjében
- Ha nem találta, akkor az ősök közvetlen őseiben folytatja
  - Stb.
- Lényegében szélességi keresés

# Python – Példa

```
class Shape:

    def print(self):
        raise NotImplementedError

    def calculateArea(self):
        raise NotImplementedError


class Circle(Shape):
    PI = 3.1415926535

    def __init__(self, radius):
        self._radius = radius

    def print(self):
        print("Circle r: " + self._radius)

    def calculateArea(self):
        return self._radius * self._radius * Circle.PI
```

# Python – Példa

```
def polygon_function(self):
    return self._sideA, self._sideB, self._sideA, self._sideB


class Parallelogram(Shape):

    def __init__(self, a, b, deg):
        self._sideA = a
        self._sideB = b
        self._delta = deg
        self._update_height()

    def _update_height(self):
        self._height_a = self._sideA * math.sin(self._delta * math.pi / 180.0)

    @property
    def sideA(self):
        return self._sideA

    @property
    def sideB(self):
        return self._sideB

    @sideB.setter
    def sideB(self, value):
        self._sideB = value
        self._update_height()
```

# Python – Példa

```
    @property
    def angle(self):
        return self._delta

    @angle.setter
    def angle(self, value):
        self._delta = value
        self._update_height()

    get_edges = polygon_function

    def is_equilateral(self):
        return self._sideA == self._sideB

    def is_right(self):
        return self._delta == 90

    def print(self):
        print("Parallelogram a, b, delta: " ,
                    self.sideA , " " , self.sideB , " " , self.angle)

    def calculateArea(self):
        return self.sideA * self._height_a
```

# Python – Példa

```
class Rhombus (Parallelogram):
    def __init__(self, a, deg):
        Parallelogram.__init__(self, a, a, deg)

    def print(self):
        print("Rhombus a, delta: " , self.sideA , " " , self.angle)

    def calculateArea(self):
        return self.sideA * self.sideA * math.sin(self.angle * math.pi/180.0)

    def is_equilateral(self):
        return True

class Rectangle (Parallelogram):
    def __init__(self, a, b):
        Parallelogram.__init__(self, a, b, 90)

    def print(self):
        print("Rectangle a, b: " , self.sideA , " " , self.sideB)

    def is_right(self):
        return True
```

# Python – Példa

```
class Square (Rectangle, Rhombus):

    def __init__(self, a):
        Rectangle.__init__(self, a, a)

    def print(self):
        print("Square a: " , self.sideA)

    def calculateArea(self):
        return self.sideA * self.sideA
```

# Python – Példa

```
n1 = Square(30)
n1.print()
print(n1.calculateArea())

n2 = Rhombus(30, 45)
n2.print()
n2.angle = 90
print(n2.is_right())
print(n2.calculateArea())

n2 = n1
print(n2.get_edges())
```

# Python

```
class Grand:
    def call(self):
        print("Grandparent")

class Parent1(Grand):
    pass

class Parent2(Grand):
    def call(self):
        print("Parent 2")

class Child(Parent1, Parent2):
    def test(self):
        self.call()

c = Child()
c.test()
```

# Python

```
$ python2 demo.py
>> Grandparent

$ python3 demo.py
>> Parent 2
```

De lehetséges expliciten megadni

```
class Child(Parent1, Parent2):
    def test(self):
        Grand.call(self)
```

# Ruby https://repl.it/languages/ruby

Teljesen objektumorientált
- Smalltalk, Python hatások, hasonlóságok
- Minden típus osztály, így minden változó objektum
  - Erősen típusos nyelv!
- Az osztályok is objektumok
- Az `Object` osztály minden más típusnak őse
  - Numerikus típusok hierarchiába szervezve
  - Logikai típus nincs
    - A true és a false a TrueClass / FalseClass egyke (Singleton) osztályok egyetlen példánya
- GC
  - Nincs destruktor
- Egyszeres öröklődés
- Dinamikus öröklődés

# Ruby

Jelölési konvenciók

- A nagybetűvel kezdődő változók konstansnak számítanak
  - osztálynév nagybetűvel kezdődik
- A kisbetűvel kezdődő nevű változók lokálisak
- A tagváltozók @-cal kezdődnek.
- A osztályszintű (statikus) változók @@-cal kezdődnek.
- A globális változók $ jellel kezdődnek.

```
class Vector
 def initialize( x, y )
  @x = x
  @y = y
 end
end
```

# Ruby

A konstruktor az initialize nevű függvény.

- A Rubyban nincs függvénytúlterhelés
  - Egy osztálynak csak egy konstruktora lehet
  - Ha több konstruktorra van szükségünk, akkor ezeket más nevű függvényekként kell megvalósítanunk, és explicite kell őket meghívnunk az objektumra.
- Az initialize függvény paraméterezése nincs előre megszabva
  - annyi paramétert kaphat, amennyit megszabunk

Egy függvény automatikusan visszaadja a legutoljára kiértékelt kifejezést

```
def add a, b
  a + b
end
```

# Ruby

Getter függvények

```
class Vector
 def initialize( x, y )
  @x = x
  @y = y
 end
 def x
  @x
 end
 def y
  @y
 end
end
```

Setter függvény

```
 def x=(x)
    @x = x
 end
```

Példányosítás és lekérdezés

```
v = Vector.new(2, 3)
puts v.x, v.y
```

# Ruby

Egyszeres öröklődés

- Minden tagfüggvény virtuális.
- A tagváltozók privát hozzáférhetőségűek.
- A tagfüggvények hozzáférhetőségét befolyásolhatjuk a private, public és protected minősítőkkel.
  - Az alapértelmezés a publikus.
- Származtatásnál csak a metódusok öröklődnek
  - Az adattagok nem, azok mindig objektumokhoz vannak kötve.

# Ruby

```ruby
class Person
  def initialize(nev)
    @nev = nev
  end
end

class Employee < Person
  def initialize(nev, fizetes)
    super(nev)
    @fizetes = fizetes
  end
end
```

# Ruby

Lehet példányból dinamikusan örököltetni, és módosítani

- Az örököltetéskor létrejön az ős egy alosztálya
- A példányra mutató változó dinamikusan megváltoztatja a típusát erre az osztályra

# Ruby

```ruby
class Foo
     def bar
          puts "Foo::bar"
     end
end
f = Foo.new
g = Foo.new

class << f
  def only_f
    puts "f::only_f"
  end
end
f.only_f
g.only_f
```

Alternatíva

```ruby
def f.only_f
     # …
end
```

# Ruby

## Modulok: a többszörös öröklődés helyett

- Modul neve nagybetűvel kezdődik
- Nem példányosítható, az osztályok modulokat foglalhatnak magukba
- A befogadott modul metódusai az osztály példánymetódusai lesznek
- Mixineknek nevezik
- A modulok tartalmazhatnak metódusokat, állandókat, más modulokat, vagy osztályokat
- Örökölhetnek (include) más moduloktól, de osztályoktól nem

# Ruby

```ruby
module A
  def a
    puts "A1"
  end
end
module B
  def b
    puts "B1"
  end
end
class Test
  include A
  include B
end
obj = Test.new
obj.a     # => A1
obj.b     # => B1
```

# PHP

PHP 5-től van OOP támogatás

- Egyszeres öröklődés
- Interfészek
  - Többet is implementálhat
  - Többszörös öröklődés lehetséges
- Láthatóság
  - public, protected, private
  - Alapértelmezett a public
- Statikus esetben nincs $this, helyette self és parent
- Absztrakt osztályok szokásos módon
- Trait-ek

# PHP

```php
abstract class Shape
{
    abstract public function print();
    abstract public function calculateArea();
}

interface Polygon
{
    public function getEdges();
}
```

# PHP

```php
class Circle extends Shape
{
    public static $PI = 3.1415926535;

    private $radius;

    public function __construct($r) {
        $this->radius = $r;
    }

    public function print() {
        print("Circle r: " . $this->radius . "\n" );
    }

    public function calculateArea() {
        return $this->radius * $this->radius * Circle::$PI;
    }
}
```

# PHP

```php
class Parallelogram extends Shape implements Polygon
{
    protected $sideA;
    protected $sideB;
    protected $delta;
    protected $heightA;

    public function getSideA() { return $this->sideA; }
    public function getSideB() { return $this->sideB; }
    public function getAngle() { return $this->delta; }

    public function isEquilateral()
         { return $this->sideA===$this->sideB; }
    public function isRight()
         { return $this->delta == 90; }
```

# PHP

```
    public function __construct($a, $b, $angle) {
        $this->sideA = $a;
        $this->sideB = $b;
        $this->delta = $angle;
        $this->heightA = $b *
                         sin($this->delta * pi() / 180.0);
    }

    public function print() {
        print("Parallelogram a, b, deg: " .
                  $this->sideA . " " . $this->sideB . " " .
                  $this->delta . "\n" );
    }

    public function calculateArea() {
        return $this->sideA * $this->heightA;
    }

    public function getEdges() {
        return [$this->sideA, $this->sideB,
                $this->sideA, $this->sideB];
    }
}
```

# PHP

```php
class Rectangle extends Parallelogram
{
    public function __construct($a, $b) {
        parent::__construct($a, $b, 90);
    }

    public function print() {
        print("Rectangle a, b: " . $this->sideA . " " .
                $this->sideB . "\n" );
    }

    public function calculateArea() {
        return $this->sideA * $this->sideB;
    }
}

Shape: $r = new Rectangle(10, 20);
$r->print();
print($r->calculateArea() . "\n");
```

# PHP

## Trait

- Kód újrafelhasználás a cél
- Metódusok egybefoglalása, akár törzzsel is…
- Nem lehet önállóan példányosítani.

```
trait TrA{
    function a1() { print("TrA->a1()\n"); }
    function a2() { print("TrA->a2()\n"); }
}

class ClB {
    use TrA;
}

ClB: $v = new ClB();
$v->a1(); // TrA->a1()
```

# PHP

## Trait precedencia

- Az osztályban definiált metódus felülírja a trait-ből kapottat
  - A trait nevével lehet hivatkozni a trait-belire
- A trait felülírja az őstől örökölt metódust
  - A „parent”-tel a lehet hivatkozni az ősre
  - (Abban az esetben, ha egy osztályon belül van egy olyan származtatott tagunk, aminek a neve megegyezik egy, az osztályon belül használt traiten belüli tag nevével, akkor a traiten belüli tag felülírja az osztály származtatott tagját.)
- Az as kulcsszó
  - A metódus láthatóságát megváltoztatja
  - Vagy alias nevet lehet létrehozni

# PHP

```php
trait TrA{
    function a1() { print("TrA->a1()\n"); }
    function a2() { print("TrA->a2()\n"); }
}

class ClB {
    use TrA;

    public function a2() {
        print("ClB->a2()\n");
        TrA::a2();
    }
}

ClB: $v = new ClB();
$v->a2(); // ClB->a2() TrA->a2()
```

# PHP

```php
trait TrA{
    function a1() { print("TrA->a1()\n"); }
    function a2() { print("TrA->a2()\n"); }
}

class ClB {
    use TrA { a2 as hidden; }

    public function a2() {
        print("ClB->a2()\n");
    }
}

ClB: $v = new ClB();
$v->a2();      // ClB->a2()
$v->hidden(); // TrA->a2()
```

# PHP

```php
trait TrA{
    function a1() { print("TrA->a1()\n"); }
    function a2() { print("TrA->a2()\n"); }
}

class Base {
    public function a2() {
        print("Base->a2()\n");
    }
}

class ClB extends Base {
    use TrA;
}

ClB: $v = new ClB();
$v->a2();      // TrA->a2()
```

# PHP

Traitek esetén névütközés előfordulhat

```
trait TrA{
    function a1() { print("TrA->a1()\n"); }
    function a2() { print("TrA->a2()\n"); }
}

trait TrB{
    function a1() { print("TrB->a1()\n"); }
    function a2() { print("TrB->a2()\n"); }
    function a3() { print("TrB->a3()\n"); }
}
```

# PHP

```php
class ClC {
    use TrA, TrB {
        TrA::a1 insteadof TrB;
        TrB::a2 insteadof TrA;
        TrB::a1 as private hidden;
    }
}
```

# PHP

Traitekben lehet még

- Absztrakt metódusok
- Statikus adattagok és statikus metódus
  - Statikus adattag csak metóduson belül, statikus metódus osztálymetódusként viselkedik

```
trait TrA{
    function a1() { print("TrA->a1()\n"); }
    function a2() { print("TrA->a2()\n"); }
    abstract function b1();
    static function s1() { print("TrA->static\n"); }
}
```

# PHP

Használatuk

```
class ClStat {
    use TrA;
    function b1() {
        print("ClStat->b1() implemented\n");
    }
}

ClStat: $v = new ClStat();
$v->b1();
ClStat::s1();
```

# Kotlin

Java OOP képességein alapul

- Kiegészítésekkel
- Következetesebb, rövidebb szintaxissal

Tulajdonságok

- Az osztályok alapértelmezés szerint nem származtathatók tovább
  - Open
- Elsődleges és másodlagos konstruktorok
  - Elsődleges célja a rövidebb kód, a mezők közvetlen felsorolásával és inicializálsával
  - Másodlagos, minden további esetre
  - Elsődlegesből legfeljebb egy lehet, másodlagosból több is
  - Konstruktor delegáció

# Kotlin

Tulajdonságok

- Közös ős: Any
- Felüldefiniálást jelezni kell
  - `override`
- Felüldefiniálás megakadályozható
  - `final`
- Tulajdonságok és getter, setter függvények definiálhatók
  - Mögöttes változót automatikusan – szükség esetén – hozza létre
- Egyszeres öröklődés
  - Interfészek között többszörös öröklődés
  - Interfészben lehet tulajdonságot megadni
  - Tulajdonságok is felüldefiniálhatók
- Absztrakt osztályok

# Kotlin

Tulajdonságok

- Adatosztályok
  - `data class`
- Singleton
  - `object` a `class` helyett
- Static adatok
  - `companion object`

# Kotlin

```
abstract class Shape {
    abstract fun calculateArea(): Double
    abstract fun print()
}
interface Polygon {
    fun getEdges(): DoubleArray?
}
```

# Kotlin

```
class Circle(var radius: Double): Shape() {

    override fun print() {
        println("Circle, radius: $radius")
    }
    override fun calculateArea(): Double =
                    radius * radius * Circle.PI

    companion object {
        const val PI: Double = 3.1415926535
    }
}
```

# Kotlin

```
open class Parallelogram constructor(
    protected var _sideA: Double,
    protected var _sideB: Double,
    private var _delta: Double) : Shape(), Polygon {
    private var _heightA: Double = updateHeight()

    var sideA: Double
        get() = _sideA
        set(value) { _sideA = value }

    var sideB: Double
        get() = _sideB
        set(value) { _sideB = value;
                     _heightA = updateHeight()}

    var angle: Double
        get() = _delta
        set(value) { _delta = value;
                     _heightA = updateHeight()}
```

# Kotlin

```
    private fun updateHeight(): Double =
        _sideB * kotlin.math.sin(this._delta *
                                 kotlin.math.PI / 180.0)

    open fun isEquilateral(): Boolean = (sideA == _sideB)

    open fun isRight(): Boolean = (_delta == 90.0)

    override fun print() = println("Parallelogram a, b,
                        delta: $_sideA, $_sideB, $_delta")

    override fun calculateArea(): Double =
                                        _sideA * _heightA

    override fun getEdges(): DoubleArray? {
        return doubleArrayOf(_sideA, _sideB,
                             _sideA, _sideB)
    }
}
```

# Kotlin

```
class Rectangle(
        sideA: Double,
        sideB: Double):
    Parallelogram(sideA, sideB, 90.0) {

    override fun print() {
        println("Rectangle a, b:
                    $sideA, $sideB")
    }

    override fun isRight(): Boolean {
        return true
    }
}
```

# Kotlin

```
fun main(args: Array<String>) {
    var s: Shape = Rectangle(10.0, 20.0)
    s.print()

    var p: Parallelogram =
                Parallelogram(10.0,20.0,45.0)
    p.angle = 90.0
    p.print()
}
```

# Kotlin

Kiterjesztés adható egy osztályhoz öröklés nélkül

```
fun MutableList<Int>.swap(index1: Int,
                          index2: Int) {
    val tmp =       this[index1] // the list
    this[index1] = this[index2]
    this[index2] = tmp
}

val l = mutableListOf(1, 2, 3)
l.swap(0, 2)
```

# Kotlin

Hivatkozás ősre (interfész esetén is)

```
interface A {
    fun foo() { print("A") }
    fun bar()
}
interface B {
    fun foo() { print("B") }
    fun bar() { print("bar") }
}
class C : A {
    override fun bar() { print("bar") }
}
```

# Kotlin

Hivatkozás ősre (interfész esetén is)

```
class D : A, B {
    override fun foo() {
        super<A>.foo()
        super<B>.foo()
    }
    override fun bar() {
        super<B>.bar()
    }
}
```

# Funkcionális és logikai programozás

KÖVETKEZŐ ALKALOMMAL



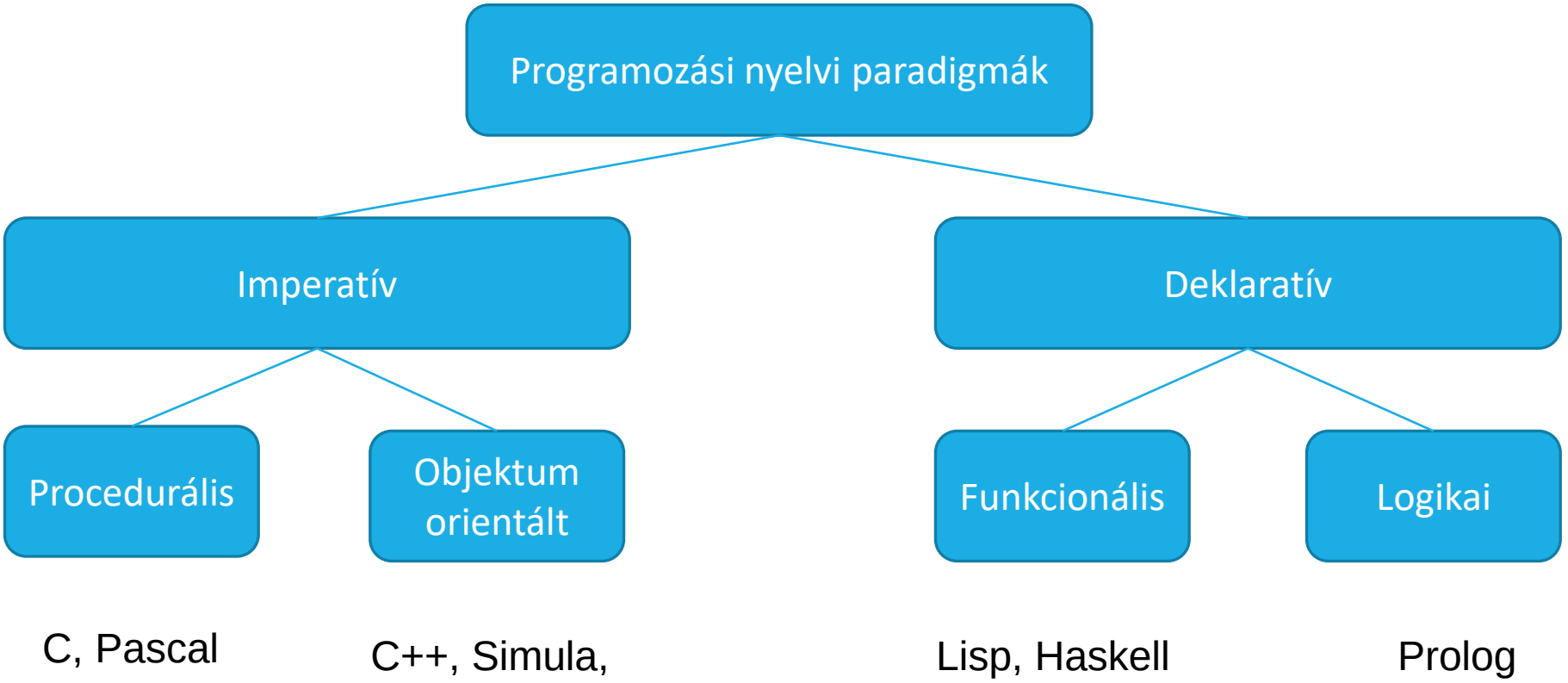

# Programozási nyelvek és módszerek

## FUNKCIONÁLIS ÉS LOGIKAI PROGRAMOZÁS

# Bevezetés

- Programozási nyelvi paradigmák
- Lambda kalkulus
- Funkcionális programozás
- Logikai programozás

# Programozási nyelvi paradigmák



# Paradigmák

Az **imperatív** programozás sajátosságai

- felszólító mód
  - parancsok, utasítások
- a lényeg az algoritmus megtalálása
  - hogyan oldjuk meg a feladatot
  - pontosan követhetőek a végrehajtott lépések
- a változó egy adott memória helyen tárolt aktuális érték, mely ismételten újabb és újabb értékeket vesz fel

# Paradigmák

A **deklaratív** programozás sajátosságai

- kijelentő mód
  - állítások, egyenletek
- a lényeg
  - a specifikáció leírása
  - mit kell megoldanunk
- a végrehajtás a nyelv értelmező, fordító programjától függ
  - nem mindig követhetőek a végrehajtott lépések
- a változó a matematikából ismert fogalomnak felel meg, egy konkrét érték lehet, mely a program írásakor még ismeretlen
- Alkalmazási területek
  - Mesterséges intelligencia
  - Telekommunikáció
  - …

# Funkcionális programozás

# Paradigmák

**Funkcionális programozási** eszközökkel minden olyan feladat megoldható, amelyik megoldható **imperatív** nyelven és viszont.

Funkcionális programozási stílus

- A megoldó programokat mellékhatásoktól mentes összetevőkből, szabályos programkonstrukciókkal építjük fel.

„A  funkcionális nyelvek matematikai számítási modellje a lambda-kalkulus.”

Mi az a lambda kalkulus?

- Egy formális rendszer, amit matematikai függvények vizsgálatára  vezettek be.
- Church 1936-ban ennek segítségével bizonyította be, hogy nem létezik megoldás a Russel-paradoxonra.

# Russel paradoxon

A múlt század elején tudatosodott, hogy a matematika addig használt halmazelméleti és logikai alapozása a Russel paradoxont is tartalmazza, belső ellentmondásai vannak.

Naiv halmazelméleti formában:

- Legyen
  - $R = \{X \mid X \text{ nem eleme } X\text{-nek}\}$, akkor $R \in R \Leftrightarrow (R \notin R)$

# Példák

## Borbély paradoxon

- Tegyük fel, hogy a laktanya katonai borbélya a szolgálati szabályzatnak megfelelően csak azokat a katonákat borotválja, akik maguk nem borotválkoznak, de nem borotválhatja azokat, akik maguk borotválkoznak.
- Kérdés: magát megborotválhatja-e?
  - Ha megborotválja magát, akkor olyan katonának számít, aki maga borotválja magát, ergo a szolgálati szabályzat megtiltja, hogy megborotválkozzon.
  - Ha ennek megfelelően, nem borotválkozik, akkor a szolgálati szabályzat értelmében, olyan katonának számít, akit borotválnia kell.

# Példák

Katalógus paradoxon

Karinthy-paradoxon

- „Ohó álljunk csak meg. Ön azt mondja, a rögeszmém, hogy őrült vagyok. De hiszen tényleg az vagyok, az imént mondta. De hiszen akkor ez nem rögeszme, akkor az egy logikus gondolat. Tehát nincs rögeszmém. Tehát mégse vagyok őrült. Tehát csak rögeszme, hogy őrült vagyok, tehát rögeszmém van, tehát őrült vagyok, tehát igazam van, tehát nem vagyok őrült. Mégiscsak gyönyörű dolog a tudomány!”

# Lambda kalkulus

A lambda kalkulusban egyfajta érték van – ez a függvény (ezt hívják lambda absztrakciónak is)

- és egyfajta művelet – a függvény alkalmazás, ez a változó behelyettesítés

Érdekesség

- Bizonyítható, hogy minden algoritmus, ami Turing gépen megvalósítható, megvalósítható tisztán lambda kalkulusban is.

# Lambda kalkulus

Vannak függvényeink:

- $x+5$
- $x^2+b*x+c$
- $4$

Minek a függvénye a kifejezés értéke?

Jelölés:

- $x \rightarrow x+5$ $\quad \lambda x.(x+5)$ - itt az x változó „kötött”
- $x \rightarrow x^2+b*x+c,$ $\quad \lambda x.(x^2+b*x+c)$
- $x \rightarrow 4$ $\quad \lambda x.4$
- $4$ $\quad \lambda x.4$

# Érdekesség

Honnan jött a jelölés?

Jelölni akarták, ha egy változó „kötött” .

- Russell és Whitehead  kezdetben a „kalap” jelölést használta
  - t[â].
- Church módosította a jelölést
  - â.t[a]
  - de ezt ∧a.t[a] formában nyomtatták ki
- Végül – azt mondják, nyomdai tévedésből – kialakult a végleges forma
  - λa.t[a].

# Művelet lambda kalkulusban

## Művelet: alkalmazás

- Azaz a változó behelyettesítése

## Példák

- λx.(2*x+3) 7
  - Azt jelenti, hogy x minden előfordulásába behelyettesítjük a 7-t
  - 2*7+3
  - 17
- λx.(2*x+3) (y-5)
  - Azt jelenti, hogy x minden előfordulásába behelyettesítjük az (y-5)-t
  - 2*(y-5)+3

# Művelet lambda kalkulusban

Többváltozós függvények

- Egyváltozós függvények egymás utáni alkalmazásával

Példa

- $f(x,y) = x+y$
- $\lambda x.( \lambda y.(x+y) )$

Ha alkalmazni akarjuk pl. $x=3$, $y=5$-re:

- $\lambda x.( \lambda y.(x+y) )\ 3\ 5$
- $\lambda y.(3+y)\ 5$
- $3+5$
- $8$

# Magasabbrendű függvények

A függvények alkalmazhatóak más függvényekre is

- $\lambda f.(\ \lambda z.(\ f\ (f\ z)\ )\ )$
- Az f-t kétszer alkalmazza egymás után

Ha ezt alkalmazzuk a $\lambda x.(\ 3*x + 5)$ kifejezésre, akkor a következőt kapjuk

- $\lambda f.(\ \lambda z.(\ f\ (f\ z)\ )\ )\ \lambda x.(3*x + 5)$

Behelyettesítés után

- $\lambda z.(\ \lambda x.(3*x + 5)\ (\lambda x.(\ 3*x + 5\ )\ z)\ )$

Ha ezt alkalmazzuk 2-re, akkor az helyébe 2-t írunk

# Magasabbrendű függvények

$\lambda z.(\lambda x.(3*x + 5)\ (\lambda x.(3*x + 5)\ z))\ 2 \Rightarrow$
$\lambda x.(3*x + 5)\ (\lambda x.(3*x + 5)\ 2) \Rightarrow$
$\lambda x.(3*x + 5)\ (3*2 + 5) \Rightarrow$
$\lambda x.(3*x + 5)\ (6 + 5) \Rightarrow$
$\lambda x.(3*x + 5)\ 11 \Rightarrow$
$3*11 + 5 \Rightarrow$
$33 + 5 \Rightarrow$
$38$

(Ez valóban az `f(f(2))`

Ez a redukálás normál formához vezetett, ami már tovább nem redukálható.

# Szabad és kötött változók

Informálisan, egy változó akkor kötött egy λ-kifejezésben, ha tőle balra található egy λ amelyik „kiterjed rá.”

- Például a λx.x kifejezésben nincs szabad változó, míg a λx.y kifejezésben az y szabad változó.

Csak azt a λ-absztrakciót tekintjük függvénynek, amely nem tartalmaz szabad változókat.

Azt a λ-kifejezést amelyik nem tartalmaz szabad változókat zártnak nevezzük.

- (Formális definíciók nélkül)

# Béta konverzió

A β-konverzió a függvény-alkalmazás elnevezése a λ-kalkulusban.

Amikor egy λ-absztrakciót alkalmazunk egy λ-kifejezésre, akkor a λ-kifejezést behelyettesítjük a λ-val kötött változó szabad előfordulásainak helyeire.

Speciálisan, ha mindig a „nyíl irányában” haladunk, azaz végrehajtjuk a kijelölt függvény-alkalmazásokat, így redukálva a λ-kifejezést, akkor azt mondjuk, hogy β-redukciót hajtunk végre.

Legfontosabb szerepe a λ-kifejezések szisztematikus kiértékelésében van.

# Béta konverzió

A β-redukció esetén egy λ-kifejezésben egyszerre több kifejezés is várhat kiértékelésre.

- Ezeket redukálható részkifejezéseknek, röviden „redex”-nek (Reducible Expression) nevezzük.

A β-redukció definíciójában nincs megkötés arra, hogy ezeket milyen sorrendben kell elvégezni, de a választott sorrendnek jelentősége van.

- A választott kiértékelési sorrendet kiértékelési stratégiának nevezzük.

# Példa

Tekintsük az alábbi λ-kifejezést például

- Két redexet is tartalmaz
- $(\lambda x.(x + 5))\ \ ((\lambda y.y)\ 2)$

Hogy értékeljük ki? Hol kezdjük?

- $((\lambda y.y)\ 2) + 5 \rightarrow\ 2+5 \rightarrow 7$ (lusta)
- $(\lambda x.(x + 5))\ 2 \rightarrow 2+5 \rightarrow 7$ (mohó)

Az a λ-kifejezés amelyben nincs egyetlen redukálható részkifejezés sem, normál formában van.

A λ-kifejezések kiértékelésének célja tehát a normál forma elérése, egymás utáni β-redukciókkal

# Példa

Normál formában van

- $\lambda y.y$
- $42$

Még redukálható

- $(\lambda x.(\lambda y.x + y - 18))\ 2\ 9$

Nincs normál formája

- $(\lambda x.x\ x)(\lambda x.x\ x)$ $\lambda$-kifejezésnek nincs $\beta$-normál formája.

Bizonyítható, hogy ha egy $\lambda$-kifejezésnek van normál formája, akkor ahhoz véges redukciós lépések sorozatával eljuthatunk, ha mindig a legbaloldalibb redukálható részkifejezést redukáljuk.

# Lambda kifejezések

# Lambda kifejezések

Egyes programnyelvek lehetőséget biztosítanak lambda kifejezések írására

- Ennek legegyszerűbb formája lehet például névtelen objektumok/osztályok létrehozása, névtelen függvények készítése
- Kiegészíti a procedurális, vagy OOP megközelítés lehetőségeit
- Átláthatóbb kód írható, a műveletekre koncentrálva a szintaxis helyett
  - Redundancia csökkentése

# Java

Az Anonymus belső osztályok, illetve a funkcionális interfészek használatakor írandó kód rövidítése

```
JButton testButton = new JButton("Test Button");
testButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent ae) {
        System.out.println("Click Detected by Anon Class");
    }
});

public interface ActionListener extends EventListener {
    public void actionPerformed(ActionEvent e);
}
```

# Java

## Szintaktika

| Argumentum | Nyíl | Törzs |
| --- | --- | --- |
| `(int x, int y)` | `->` | `x + y` |

## Példák

```
(int x, int y) -> { return x + y; };
() -> 42;
(String s) -> { System.out.println(s); };

Runnable r = () -> System.out.println("Hello");
r.run();
```

# Java

## Lambda és Collections

```
public static void main(String[] args) {
    List<Person> pl = Person.createShortList();
    pl.forEach(p -> p.printName() );
    pl.forEach(Person::printName);
    // Létező metódusra visszavezeti
    pl.forEach(p -> { System.out.println(…); });
}
```

# C++

## Anonim függvények, lambda kifejezések formájában

```
[capture](parameters) -> return_type { function_body }
```

## Példa

```
[](int x, int y) -> int { return x + y; }
```

# C++ - capturing

```
[]      //no variables defined.
[x, &y] //x érték szerint, y referencia szerint
[&]     //minden külső változó referencia szerint
[=]     //minden külső változó érték szerint
[&, x]  //x érték szerint. Minden más referencia szerint
[=, &z] //z referencia szerint. Minden más érték szerint
[this]  //this, pointer az objektumra
```

# C++

Az **auto** kulcsszó használatával lambda kifejezések típusát a fordító határozza meg

```
auto my_lambda_func = [&](int x) { /*...*/ };
```

Ugyanez dinamikusan létrehozva a heap-en

```
auto my_lambda = new auto([=](int x) { /*...*/ });
```

# C++ Template / auto

```
template<typename T>

…
[](T a, T b) -> T { return a + b; };
```

Vagy

```
[](auto a, auto b) { return a + b; };
```

# Python 3

Lambda kifejezés létrehozása – kulcsszóval

- `lambda arguments : expression`

Például

- ```
  x = lambda a : a + 10
  print(x(5))
  ```

Használhatók gyűteményekben

- Elemenként

```
a = [1, 2, 3, 4, 5, 6]
print(list(map(lambda x: x*x, a)))
```

- Inkább javasolt

```
print([x*x for x in a])
```
```

# Mathematica

Pontosabbal Wolfram nyelv

- #0 a függvényre magára hivatkozik
- #1 … #n az egyes argumentumokra hivatkozik
- & lezárja a kifejezést

Példa

- `f:= #1^2&`
- `f[8]`

Rekurzív anonim függvény

- `If[#1 == 1, 1, #1 * #0[#1-1]]&`

Példa

- `If[#1 == 1, 1, #1 * #0[#1-1]]&[6]`

# Funkcionális programok

# Funkcionális programozás

Egy funkcionális program felépítése:

- függvény definíciók, kezdeti kifejezés, operátorok, adatszerkezetek, …
- típus definíciók

Például – néhány egyszerű függvénydefiníció:

- `nulla x = 0`
- `id  x = x`
- `inc  x = x + 1`
- `square  x = x * x`
- `squareinc  x = square  ( inc x )` - kompozíció
- kezdeti kifejezés: `squareinc  7`

# Funkcionális programozás

A funkcionális program végrehajtása:

- a kezdeti kifejezés értékének meghatározása, kiértékelése, redukálása
- a normál forma meghatározása, - egyértelmű, nincs lehetőség további redukciós lépésre

A program végrehajtása a kezdeti kifejezésből kiinduló redukciós vagy más néven átírási lépések sorozata.

# Funkcionális programozás

Néhány kezdeti kifejezés

- Cleanben
  - `Start = sqrt 5.0;`
- Haskellben
  - `main = product $ [1..10]`
- A normál formák
  - `2.236068`
  - `3628800`
  - `64`

# Funkcionális programozás

Egy-egy redukciós lépésben egy redukálható részkifejezésben, a redex-ben szereplő függvényhívás helyettesítődik a függvény törzsében megadott kifejezéssel a formális és aktuális paraméterek megfeleltetése mellett.

- Kiértékelési stratégia szerint

Normál formájú egy kifejezés, ha további redukcióra nincs lehetőség.

- Ez az átírási lépéssorozat végeredménye.

# Kiértékelési stratégiák

## Lusta kiértékelés

- A kifejezések kiértékelésekor először a legbaloldalibb legkülső redexet helyettesíti
  - Ha a kifejezés függvény megadással kezdődik, előbb a függvény definíció lesz alkalmazva
  - Az argumentumok kiértékelését csak szükség esetén végzi el.
- A lusta kiértékelés normalizáló kiértékelési módszer
  - Mindig megtalálja a normál formát, ha az létezik
    - Pl. Clean, Haskell, Miranda

# Kiértékelési stratégiák

## Mohó kiértékelés

- A legbaloldalibb, legbelső redex, az argumentumok helyettesítése történik meg először
  - hatékonyabb, mint a lusta rendszer
  - de: nem mindig ér véget a kiértékelési folyamat
    - Lisp, SML, Hope

## A két stratégia kombinálható

# Példák

Mohó kiértékelés:

- `squareinc  7`
- `-> square  ( inc 7 )`
- `-> square  ( 7 + 1 )  - először az argumentum(ok)`
- `-> square 8`
- `-> 8*8`
- `-> 64`

Lusta kiértékelés:

- `squareinc  7`
- `-> square  ( inc 7 )`
- `-> ( inc 7 ) * ( inc 7 )   - először a függvény`
- `-> (7 + 1) * ( 7 + 1 )`
- `-> 8*8`
- `-> 64`

# Függvény definíciók:

Nem mindegy, hogy milyen a kiértékelés, pl.:

- `f x = f x`
- `g x y = y` – ezek függvény definíciók

Példa

- `g 1 2` ⇒ `2`
- `g (f 3) 5` ⇒ 5

Milyen kiértékelés jó?

csak a lusta!

# A modern funkcionális nyelvek fő jellemzői

## Hivatkozási átláthatóság (referential transparency)

- A kifejezések értéke hivatkozásuk előfordulási helyétől független
  - ugyanaz a kifejezés a program szövegében mindenhol ugyanazt az értéket jelöli.
- A függvények kiértékelésének nincs mellékhatása
  - egy függvény kiértékelése nem változtathatja meg egy kifejezés értékét!
- Példa:
  - $f(x) + 2*f(x) = 3* f(x)$ – biztosan!

# Hatásra ellenpélda

Emlékeztető példa, hogy ez nem mindig igaz:

```
int x = 1;
int f( int a ){
  ++x;
  return a+x;
}
int main(){
  int y = f(x) + 2*f(x);
  int z = 3*f(x);
}
```

$y \neq z$

# A modern funkcionális nyelvek fő jellemzői

## Feltételek megadási lehetősége (definition by cases)

- A függvény értékét az első, igaz egyenlőséghez tartozó kifejezés adja, mely kifejezést az aktuális paraméter értéke alapján kapjuk
- Példa: az argumentum előjelének a meghatározása:

```
elojel:: Int -> Int
elojel x| x < 0 = -1
elojel x| x > 0 = 1
elojel x| x == 0 = 0
main = print (elojel (-10))
```

# A modern funkcionális nyelvek fő jellemzői

Rekurzivitás lehetősége (definition by recursion):

- A függvények hivatkozhatnak önmagukra és kölcsönösen egymásra
- Példa: két egész szám  legnagyobb közös osztója:

```
lnko :: Int -> Int -> Int
lnko a b | (a == b) = a
lnko a b | (a > b) = lnko (a-b) b
lnko a b | (a < b) = lnko a (b-a)
main = print $ lnko 18 56
```

# A modern funkcionális nyelvek fő jellemzői

A függvények ugyanolyan értékek, mint az elemi típusértékhalmazok elemei.

Magasabbrendű függvénynek nevezzük azokat a függvényeket, amelyeknek valamelyik argumentuma vagy értéke maga is függvény.

- Példa

```
Twice f x = f ( f  x )
```

# A modern funkcionális nyelvek fő jellemzői

## Mintaillesztés

- Azt vizsgálja a kiértékelő rendszer, hogy az aktuális paraméterek értéke vagy alakja megfelel-e valamelyik egyenlőség bal oldalán megadott mintának.
  - A minták sorrendje meghatározza az eredményt
  - Az első illeszkedést keresi
- Példa: faktoriális (mintaillesztés és feltétel, rekurzióval):

```
fakt  0         =  1
fakt  n | n>0   =  n * fakt (n-1)
```

# A modern funkcionális nyelvek fő jellemzői

## Szigorú, statikus típusosság

- Típusdeklarációk megadása nem kötelező, de megköveteljük, hogy a kifejezések típusa a típuslevezetési szabályok által meghatározott legyen.
- Ez azt jelenti, hogy egy adott kifejezés legáltalánosabb típusát a fordítóprogram általában még akkor is meg tudja határozni a benne szereplő részkifejezések típusa alapján, ha a programszöveg készítője nem deklarálta a kifejezés típusát.

# A modern funkcionális nyelvek fő jellemzői

Iteratív adatszerkezet elemeinek és azok sorrendjének megadására alkalmas

- a matematikában halmazok megadásánál alkalmazott jelölésrendszernek megfelelő nyelvi eszköz a Zermelo-Fraenkel halmazkifejezés.

A végtelen adatszerkezetek kiértékelése lusta kiértékelési módszerrel történik.

- listák, halmazok, sorozatok, vektorok

Például:

```
negyzetek n = [x^2 | x <- [1..n]]
```

# A modern funkcionális nyelvek fő jellemzői

A függvények típusát értelmezési tartományuk és értékkészletük megadásával határozzuk meg.

- Például

```
fakt: int -> int
```

Többváltozós függvények

- minden függvénynek egyetlen argumentuma lehet, de ez esetleg egy függvény!

## Egyszerű típuskonstrukciók

- Rendezett n-esek
- Iterált – véges vagy végtelen sorozatok

## Sorozatok

- Konstruktor – egy elemből és egy sorozatból új, az adott sorozatot balról kiterjesztő sorozatot készít – mintában is használhatóak
- Első elem (fejelem, head), lista fejelem nélküli maradéka (tail),…

## Például: sorozat elemeinek összege:

```
ossz :: [Int] ->Int
ossz [] = 0
ossz (x:xs) = x + ossz xs
```

Sorozatok generálhatók is

Például: Párosak valameddig:

```
paros n= [x | x <- [1..n], even x]
```

- (az x <- [ 1 .. n] generátorral előállítjuk az összes értéket 1-től n-ig, majd az even x szűrővel kiválogatjuk a megfelelőeket)

# Példa

Példák magasabb rendű függvényekre

- Filter – adott tulajdonságot teljesítő elemek leválogatása

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs)
  | p x = x : filter p xs
  | otherwise = filter p xs

even x =  x 'mod' 2 == 0
evens = filter even [0..]
filter even [3,2,2,1] -> [2,2]
```

# Példa

map: Elemenként alkalmazza a paraméterül kapott függvényt

Példa:

```
map inc [2,1,7] -> [3;2;8]
```

Definíció:

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs)= f x : map f xs
```

Példa:

```
odds = map (+1) evens
```

# Példa

Példa:

```
qs :: [Int]->[Int]
qs [] = []
qs (x:xs) = qs smaller ++ [x] ++ (qs larger)
  where
    smaller = filter (<x) xs
    larger  = filter (>x) xs


main = do
  print (qs [10, 20, 40, 0, 15, 25, -10] )
```

# Logikai programozás

# Logikai programozás

Ha adott egy rendszer logikai formulákkal definiált modellje

- kiegészítendő vagy eldöntendő kérdéseket fogalmazhatunk meg bizonyítandó állítások formájában.

Kiegészítendő kérdés esetén a megfelelő tulajdonságú objektumok, tervek stb. meghatározása számítási folyamatnak tekinthető.

Ez a logikai programozás (LP) alapgondolata.

# Logikai programozás

A logikai program egy modellre vonatkozó állítások (axiómák) egy halmaza.

- Az állítások a modell objektumainak tulajdonságait és kapcsolatait, (relációit) írják le.

Ha például adottak állítások, amelyek arra vonatkoznak, hogy ki kinek az apja, ezek együtt a apja nevű, kettő aritású relációt írják le.

Ha megmondjuk, hogy kik a férfiak, vagy kik a nők, akkor az erre vonatkozó állítások együtt a férfi, illetve nő nevű, egy aritású relációt írják le.

# Példa

Tények: (a legegyszerűbb logikai programok – atomi formulák)

```
apja('Ábrahám','Izsák').
apja('Ábrahám','Ismáel').
apja('Ábrahám','Ismeretlen').
apja('Izsák','Jákob').
apja('Izsák','Ézsau').

anyja('Sára','Izsák').
anyja('Hágár','Ismáel').
anyja('Rebeka','Jákob').
anyja('Rebeka','Ézsau').

férfi('Ábrahám').
férfi('Izsák').
férfi('Ismáel').
férfi('Jákob').
férfi('Ézsau').

nő('Sára').
nő('Hágár').
nő('Rebeka').
nő('Ismeretlen').
```

# Logikai programozás

Az állítások egy adott relációt meghatározó részhalmazát predikátumnak nevezzük.

- A program futtatása minden esetben egy, az állításokból következő tétel konstruktív bizonyítása, azaz – a logikai programozásban szokásos szóhasználattal – a programnak feltett kérdés vagy más néven cél megválaszolása.
- Ennek során a predikátumok eljárásokként működnek.

A ma használatos LP nyelvekben minden, a predikátumokat alkotó állítás tény vagy szabály lehet.

Prolog: Programming in logic (1972-től)

Kérdéseket tehetek fel:

```
|?- apja('Ábrahám','Izsák').
yes
|?- anyja('Sára','Jákob').
no
- Van-e olyan X, akinek az apja Izsák?
|?- apja('Izsák',X).
X = 'Jákob'  ;    X = 'Ézsau' ;
```

Összetett célok:

- `|?- apja('Ábrahám',X), nő(X).`
- `X = 'Ismeretlen' ;`

Ilyenkor a célsorozat megoldásai a részcélok közös megoldásai.

# Horn Klózok

A logikai programok  állításait Horn klózoknak is nevezik.

Egy Horn klóz azt fejezi ki, hogy egy adott állítás igaz, ha nulla vagy több másik állítás igaz. Pl.:

- ha (if) egy személy öreg és bölcs,
  akkor (then) ez a személy boldog
- ha (if) X az apja Y-nak és  Y az apja Z-nek,
  akkor (then) X a nagyapja Z-nek

Nulla vagy több állítás lehet az if-részben, és pontosan egy következtetés a then-részben.

Egy feltétel nélküli Horn klóz egy tényt ír le.

# Horn Klózok

Egy Horn klóz a következő formában írható:

- $G_0 \leftarrow G_1, G_2, \ldots, G_n$.

Ennek a jelentése:

- Ha (If) $G_1$ ... $G_n$ mind igazak, akkor (then) $G_0$ is igaz.

A szabály baloldala a feje (head); a jobboldala a törzse (body)

- $G_0$... $G_n$ a célok

# Horn Klózok

A törzsben lévő célokat alcéloknak (subgoals) hívjuk

- A vesszőket logikai AND szimbólumként kell olvasni
- Megjegyzés: a $\leftarrow$ csak azt jelenti, hogy „ha” (“if”) - nem pedig azt, hogy „akkor és csak akkor” (“if and only if”) – lehetnek más szabályok is $G_0$ –ra!
  - (a $\leftarrow$ jelölése :- )
  - Speciális eset, ha $n=0$, ekkor $G_0$ egy egyszerű tény, ami mindig igaz.

# Logikai programozás

A logikai programozás lehetővé teszi, hogy

- a programozó megadja tények és if-then szabályok egy listáját
- azután a rendszer automatikusan el tudja dönteni, hogy egy állítás igaz-e, vagy meg tudja mondani, mire igaz, stb.

Szabályok:

- `'Ábrahám lánya'(X) :- apja('Ábrahám',X), nő(X).`
- `szülője(X,Y) :- anyja(X,Y).`
- `szülője(X,Y) :- apja(X,Y).`

# Logikai programozás

A szülője reláció az állítások által definiált relációk uniója.

```
fia(X,Y) :- szülője(Y,X), férfi(X).
lánya(X,Y) :- szülője(Y,X), nő(X).
```

A fia reláció a szülője és a férfi relációk metszete, és hasonlóan adódik a lánya reláció is.

# Logikai programozás

```
nagyszülője(X,Y) :- szülője(X,Z), szülője(Z,Y).
```

Ez eltér az előzőektől abban, hogy a jobb oldalán új változó is előfordul. Így kétféle olvasata is van:

- Minden X,Y,Z-re nagyszülője(X,Y), ha szülője(X,Z) és szülője(Z,Y).
- Minden X,Y-ra nagyszülője(X,Y), ha van olyan Z, hogy szülője(X,Z) és szülője(Z,Y).
- Ezek a deklaratív olvasatai a szabálynak.

Ezzel szemben áll a procedurális olvasat, ami a logikai programok futtatásához, vagyis a konstruktív bizonyítási eljáráshoz kapcsolódik.

- Minden esetben felteszünk egy kérdést.
  - Ez a bizonyítandó cél(sorozat).
  - Ezután a részcélok bizonyítása, és így kiejtése a feladat. Felülről lefelé érdemes …

# Keresési fa a részcélokhoz

```
?- nagyszülője('Ábrahám',X).
     szülője('Ábrahám',Z),szülője(Z,X).
        anyja('Ábrahám',Z),szülője(Z,X).  % meghiúsul
        apja('Ábrahám',Z),szülője(Z,X).
          { Z <- 'Izsák' }
            szülője('Izsák',X).
              anyja('Izsák',X).           % meghiúsul
              apja('Izsák',X).
                { X <- 'Jákób' }          % 1. megoldás
                { X <- 'Ézsau' }          % 2. megoldás
          { Z <- 'Ismáel' }
            szülője('Ismáel',X).
              anyja('Ismáel',X).          % meghiúsul
              apja('Ismáel',X).           % meghiúsul
          { Z <- 'Ismeretlen' }
            szülője('Ismeretlen',X).
              anyja('Ismeretlen',X).      % meghiúsul
              apja('Ismeretlen',X).       % meghiúsul
```

# Rekurzív szabályok

Tegyük fel, hogy azt szeretnénk leírni, X mikor őse Y-nak.

- Világos, hogy akkor őse, ha szülője, vagy valamelyik ősének a szülője.

Látható, hogy a relációnak két esete van, és az egyik eset rekurzív.

- Ezt ennek megfelelően egy nem rekurzív és egy rekurzív szabállyal fejezhetjük ki:
  - `őse(X,Y) :- szülője(X,Y).`
  - `őse(X,Y) :- szülője(X,Z), őse(Z,Y).`
  - kérdés: `?-őse('Ábrahám',X)`

A keresőfa előállításához célszerű a legbal részcélkiválasztási módszert alkalmazni

- Általában is érdemes a nem rekurzívval kezdeni…

# Mire alkalmas?

Felhasználási területek

- mesterséges intelligencia
- számítógépes nyelvészet
- adatbáziskezelés …

# Párhuzamos programozás

KÖVETKEZŐ ALKALOMMAL




# Programozási nyelvek és módszerek

## PÁRHUZAMOSSÁG

# Bemelegítés

# Párhuzamosság

Puzzle analógia – Henry Neeman @ Oklahoma University

Soros programozás: egyvalaki szeretné megcsinálni

- 1000 darabos puzzle, kb. 1 óra



Tegyük fel, hogy egy barát leül és segít kirakni

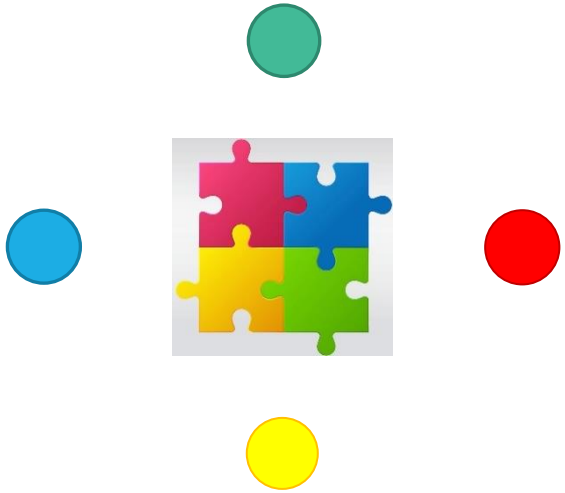
- Ekkor kommunikálni is kell.
  - Néha ugyanazon a részen dolgoznak, és zavarják egymást.
- 35 perc alatt oldható meg
  - A várt 30 helyett.



# Párhuzamosság

Minél többen annál jobb?

- Ha már négyen vagyunk, 20 perc kell (15 helyett), mert egyre többet kell koordinálni egymással



Csökkenő hozadék törvénye

- Ahogy többen próbálnak meg közreműködni, egyre többet kell kommunikálni és egymásra várni vagy egymást kikerülni.
- Ugyanazt a problémát egyre több párhuzamos ágenssel megoldani a csökkenő hozadékok törvényéhez vezet.

# Elosztott párhuzamosság

Osszuk szét a puzzle darabokat két kupacba

- Mindenki dolgozik a saját felén teljesen függetlenül,
- A végén össze kell illeszteni a két felet



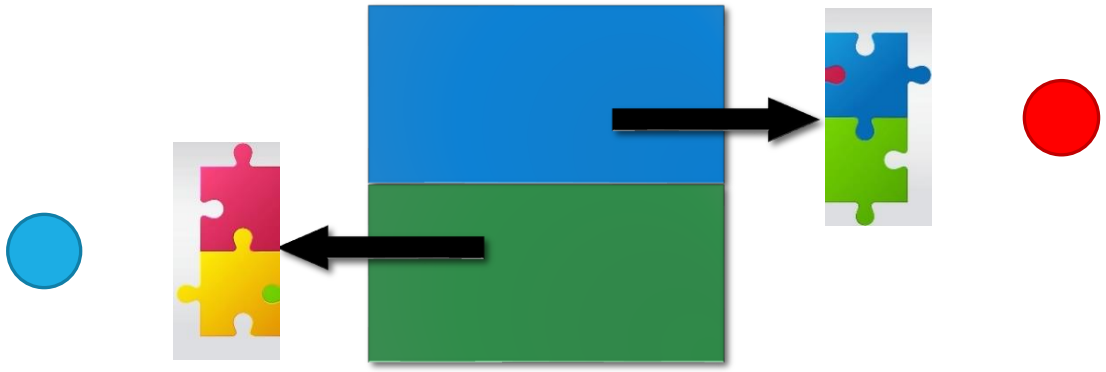
Szét is kell osztani a darabokat az elején

- Attól függően hogy mennyi töltünk egy jó elosztással, több vagy kevesebb kommunikációra van szükség a végén az összeillesztésekor
  - Tökéletes felosztás a végső kép két felére: sok munka az elején, a végén triviális.
  - Véletlen leosztás: triviális az elején, sok munka a végén

# Elosztott párhuzamosság

A kezdeti elosztás határozza meg a két fél kiegyensúlyozottságát is.

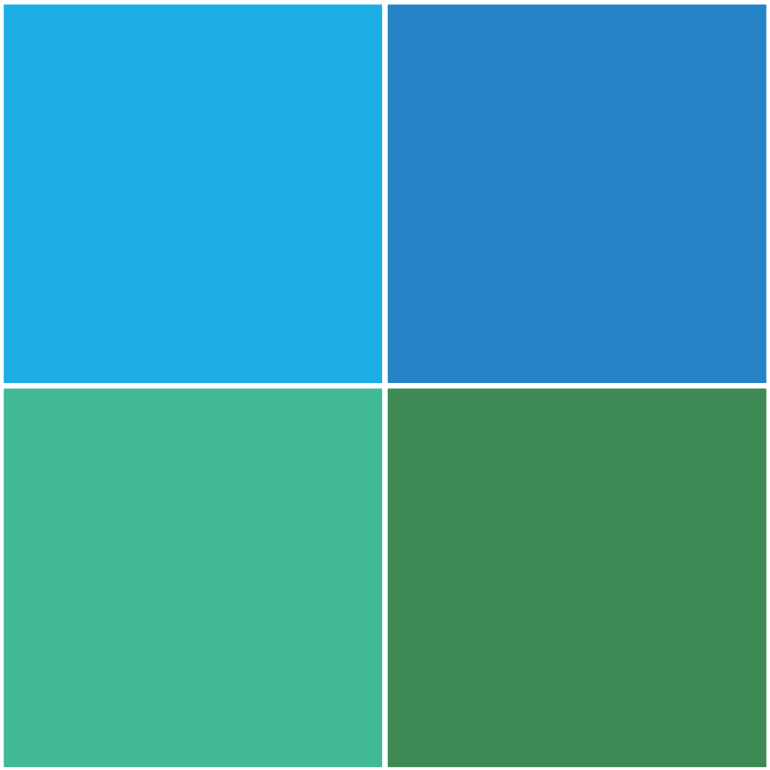
- Ha a kép fele ég a másik föld, könnyű a leosztás, a végén csak a perem mentén kell „kommunikálni”

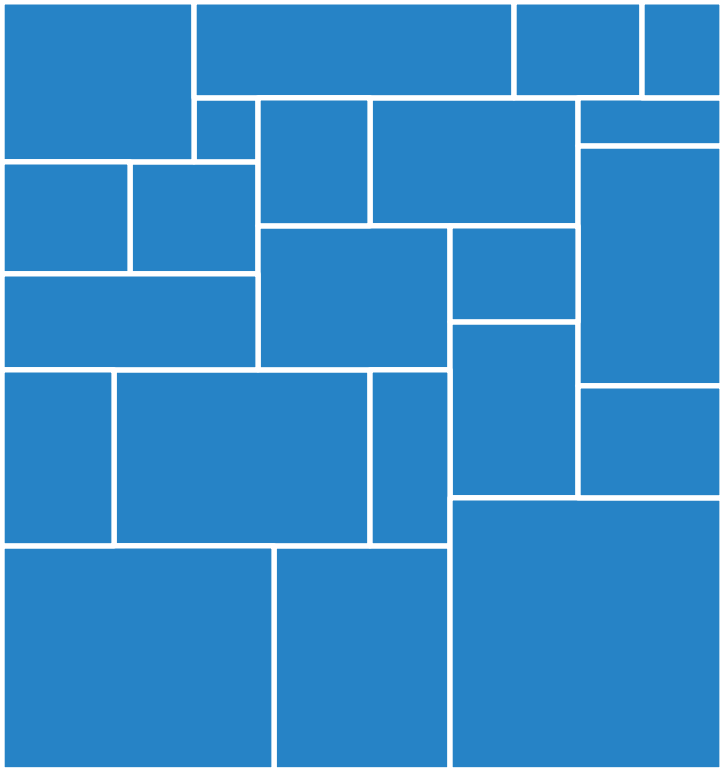


Bizonyos problémák esetén ez a felosztás könnyebb, mint más problémákon

# Elosztott párhuzamosság

A munka kiegyensúlyozása nehéz, főleg ha az egyik résztvevő gyorsabb





# Párhuzamosság

FOGALMAK

# Alapelvek

A hagyományos számítógépeken

- Egy adott időben egy program futhat
- A számítógép architektúrája pedig egy egyprocesszoros gép
  - Csak egy program lehet aktív, egy adott pillanatban egy utasítás kerül végrehajtásra.
  - A különböző programok végrehajtása egymás után történik

Igény a valódi párhuzamosításra

- Hardver szintjén
- Szoftver szintjén

# Párhuzamosság

- Van egy rendszer és egy időben több komponense működik
- Előnyei a szekvenciális programozással szemben
  - Gyorsabb programvégrehajtás
  - Hatékonyabb programvégrehajtás
    - Megfelelő hardver esetén

Természetesebb kifejezésmód lehetősége a nagyobb számú elemi művelet segítségével (szoftver szempont)

Összehangolás kérdései:

- Kommunikáció
- Szinkronizáció

Gondok

- Tesztelés
- Holtpont
- Kiéheztetés

Az elosztott (distributed) jelző – többféle jelentés

- Elosztott – a „párhuzamos” speciális esete, amikor a rendszer komponensei térben elkülönülten helyezkednek el
- Elosztott memória használata

Megosztott (shared) jelző

- Erőforrások együttes használata – tipikusan memória

A konkurrens (concurrent) szó gyakran azt hangsúlyozza, hogy a rendszer komponensei vetélkednek egymással az erőforrásokért

# Párhuzamos hardverek

A párhuzamos hardver rendszerek sokféleképpen osztályozhatók

Egy ilyen a máig is használt Flynn-féle osztályozás

- Komponensei
  - A processzor
  - A memória és
  - A vezérlő (controller)

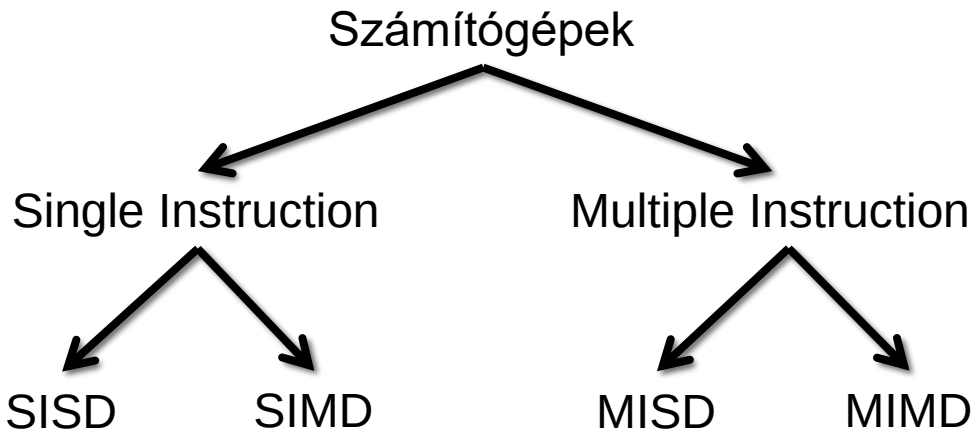
A processzor(ok) a vezérlőtől kapják a végrehajtandó utasítássorozatot (instruction stream)

- A memóriá(k)ból pedig a feldolgozandó adatokat (data stream)

# Flynn-féle modell

Csoportok aszerint, hogy hány utasításfolyam (Instruction stream), illetve adatfolyam (Data stream) különíthető el a rendszerben

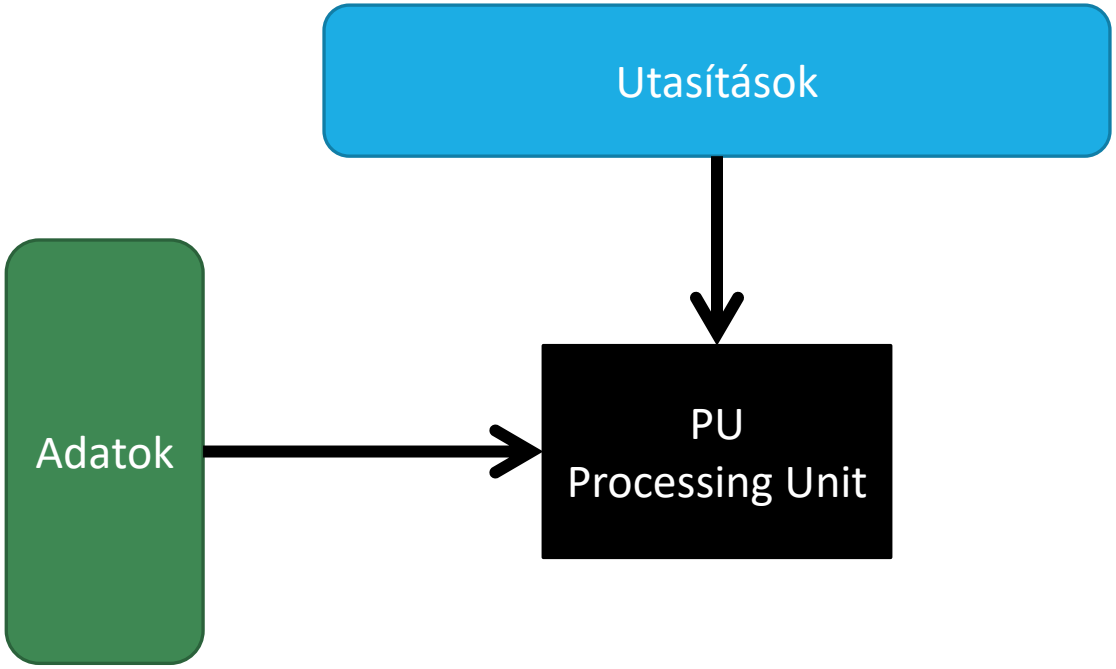
Mindkettő egyszeres (Single), vagy többszörös (Multiple) lehet



# SISD (Single Instruction, Single Data)

„Klasszikus” számítógépek

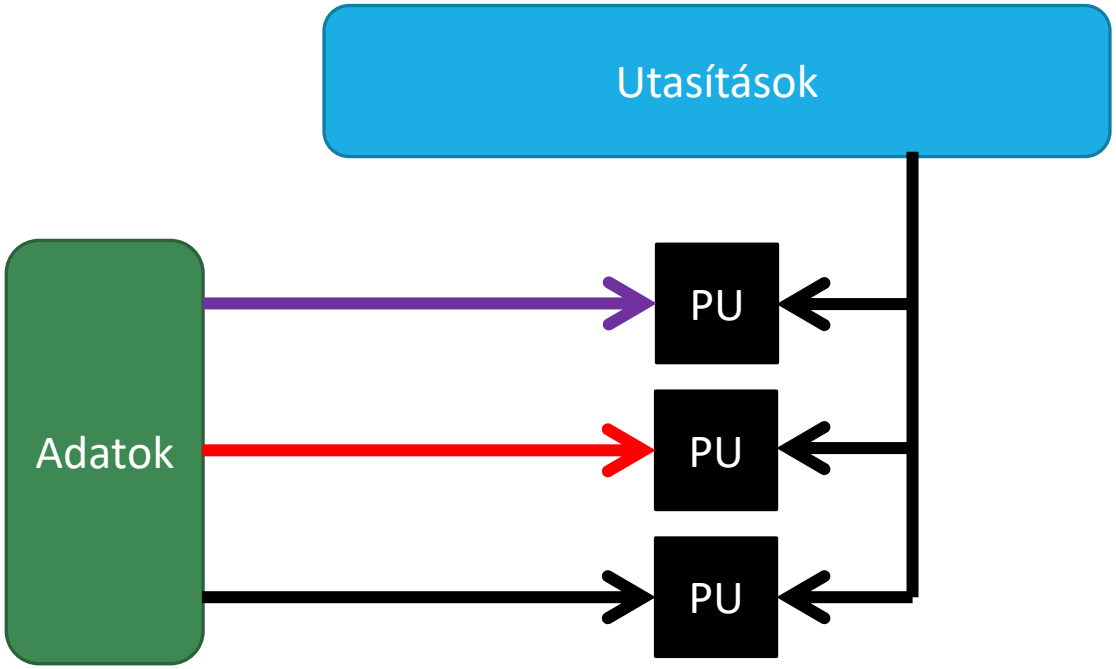
- Egy processzor, egy mag
- Napjainkban: számológép



# SIMD (Single Instruction, Multiple Data)

Nagyon sok egyszerű processzorból álló gépek

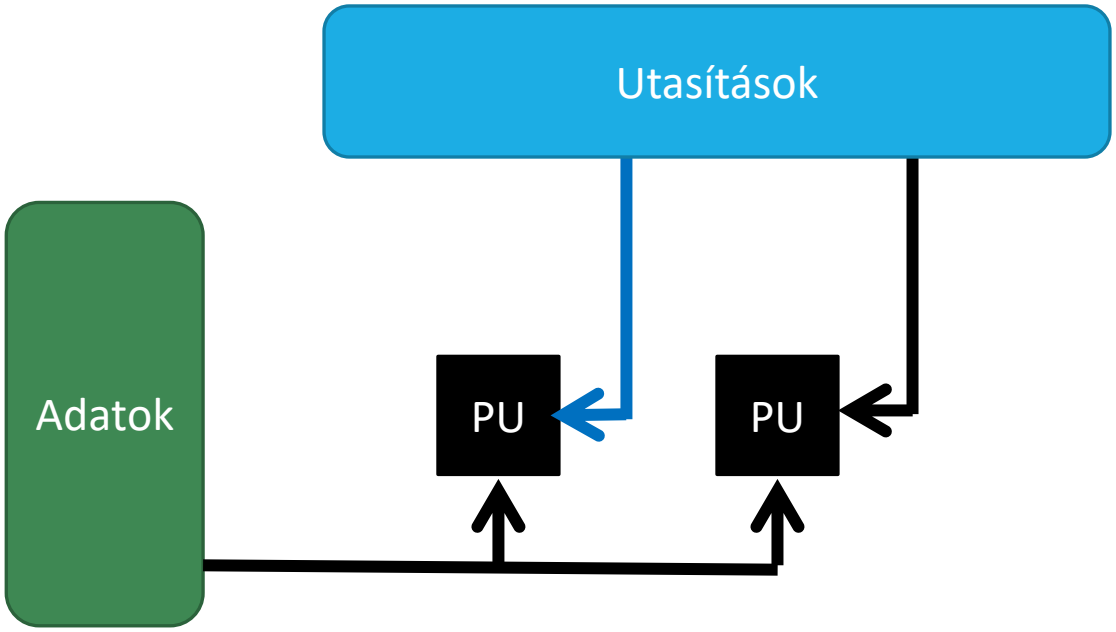
- Mindegyike rendelkezik saját memóriával
- Mindegyiken azonos program fut
- GPGPU



# MISD (Multiple Instruction, Single Data)

Nem gyakori az ilyen gép

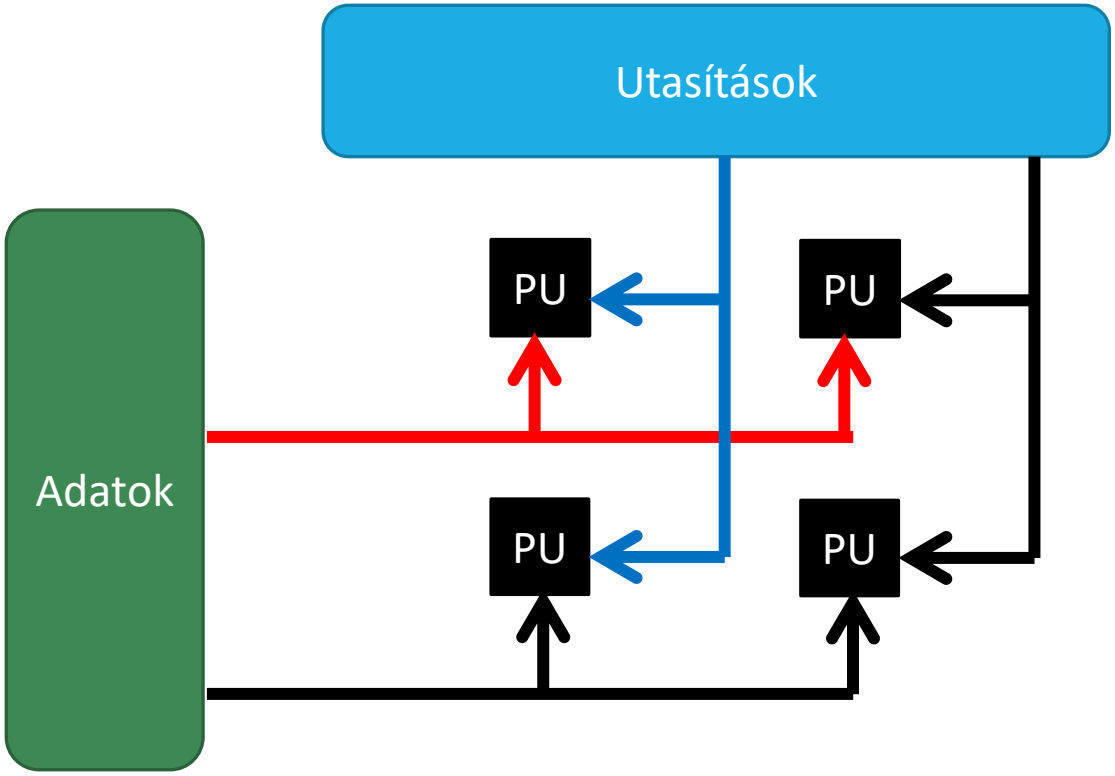
- Hibatűrő rendszerek esetén használatos
- Ugyanazt a feladatot több algoritmus oldja meg
  - Például háromból két azonos eredményt tekintünk jó megoldásnak



# MIMD (Multiple Instruction, Multiple Data)

Számunkra most a legfontosabb csoport

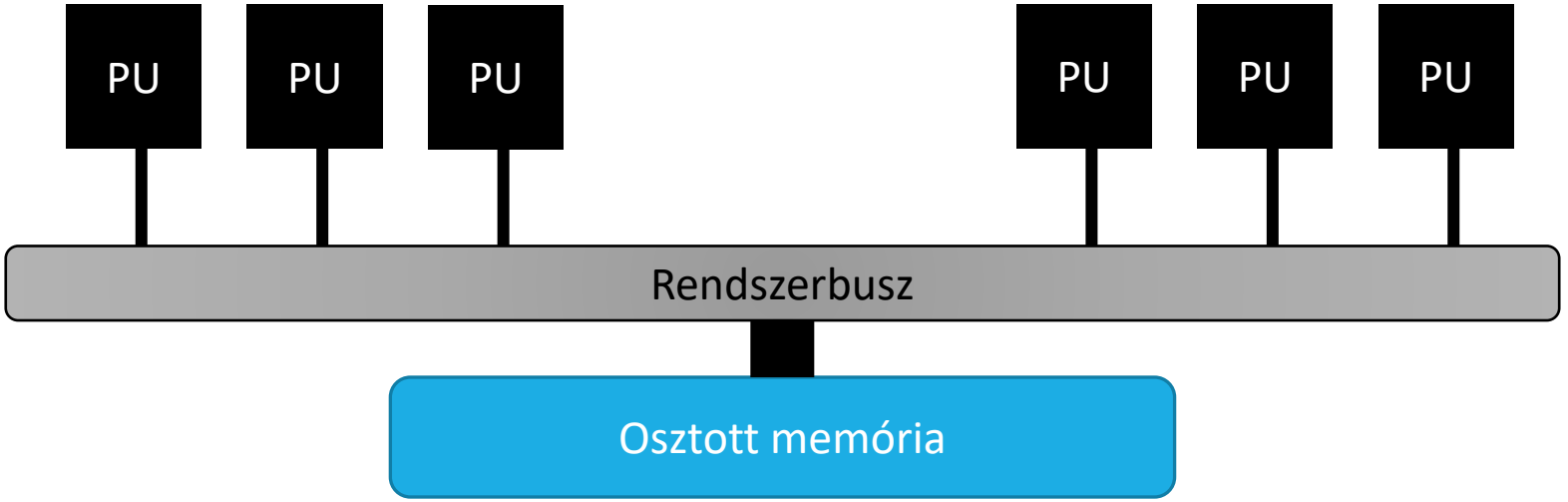
- Többféle elrendezésben
- Egy lehetséges elrendezés:



# MIMD (Multiple Instruction, Multiple Data)

MIMD

- SMP (Symmetric multiprocessing) rendszerek
- SM-MIMD (shared-memory MIMD) rendszerek
  - a rendszerbuszra egyetlen fizikai memória csatlakozik, amelyet minden processzor lát. (Pl. többprocesszoros PC)



# SM-MIMD architektúra

Előnye

- az osztott memórián keresztül nagyon egyszerű a programok közötti kommunikáció, illetve szinkronizáció megoldása,
- hagyományos programok nagyon könnyen átültethetőek ilyen környezetbe.

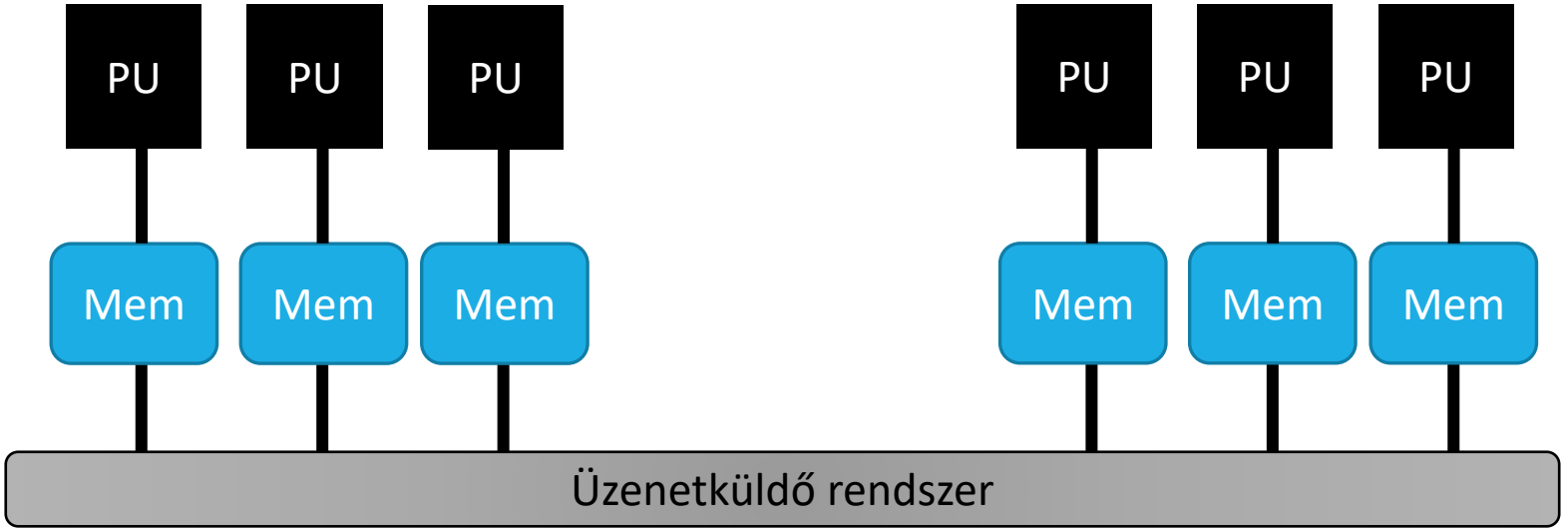
Hátránya

- a memória elérésének fizikai korlátai korlátokat szabnak a lehetséges processzorok számának.

# MPP (Massively parallel processing)

Vagy DM-MIMD (distributed memory MIMD) architektúrák

- Önálló memóriával rendelkező processzorokból felépített nagy számítógépek – üzenetküldéssel kommunikálnak

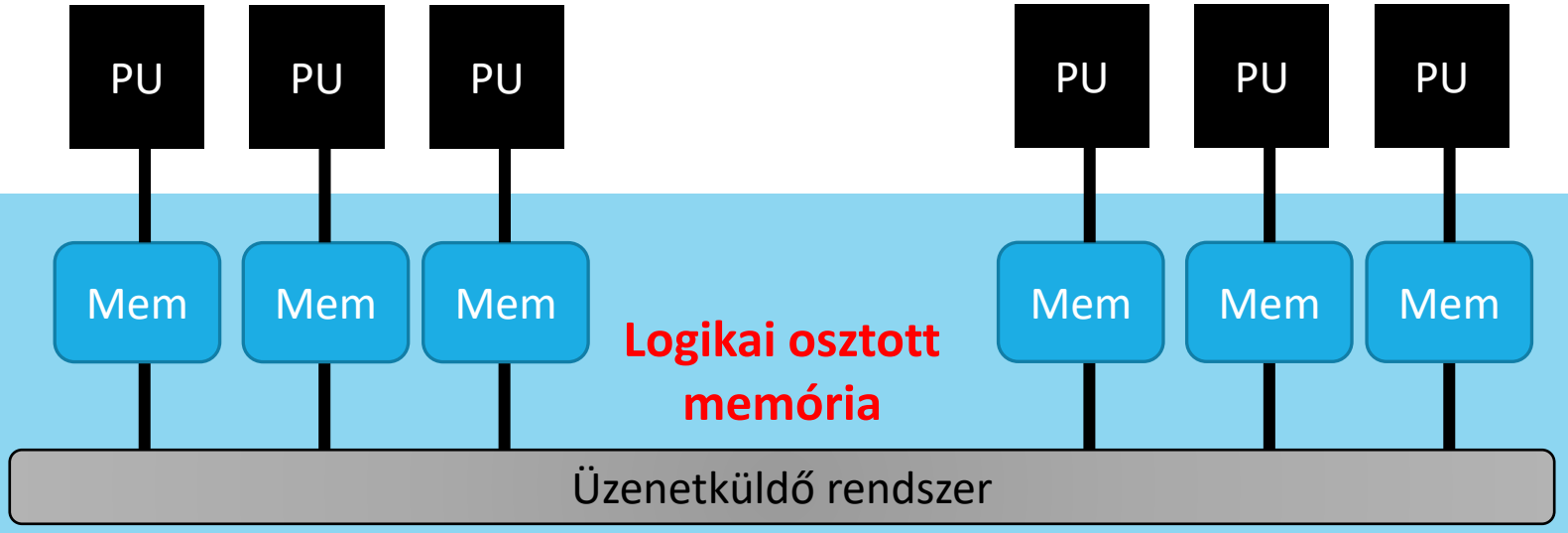


# SPP (Scalable parallel processing)

Olyan DM-MIMD rendszer amelyben a processzorok közötti kommunikációs eszköz szimulálni tudja az egyetlen osztott memóriát.

A processzorok számára nem saját memóriájuk elérése csak időbeni különbséget jelent.

Ötvözi az SM-MIMD és DM-MIMD rendszerek előnyeit – sokkal bonyolultabb hardware megoldásokra van szükség.



# A szoftveres párhuzamosság

Az első próbálkozás aszinkron működésre az input-output műveleteknek a központi egység műveleteivel egy időben való végzése volt.

- Ezeket már alapjában véve konkurens műveleteknek nevezték.

A következő nagyobb lépést az operációs rendszer különböző műveleteinek párhuzamos végzése jelentette.

- Ezáltal megjelent a multiprogramozás fogalma.
- A cél a gép erőforrásainak és aszinkron lehetőségeinek kihasználása volt.

Különböző feladatok a készültség különböző állapotaiban párhuzamosan létezhetnek.

- Az ilyen rendszerekben a  CPU folyamatosan váltogatja az egyes programokat, melyeket csak néhány századmásodpercig futtat
- Így a CPU egyszerre csak egyetlen programot hajt végre, viszont adott idő intervallumban akár több száz programot is kiszolgálhat a párhuzamos futás illúzióját keltve.

# Folyamatok (process)

A processz egy szekvenciálisan végrehajtott program szerinti műveletsor.

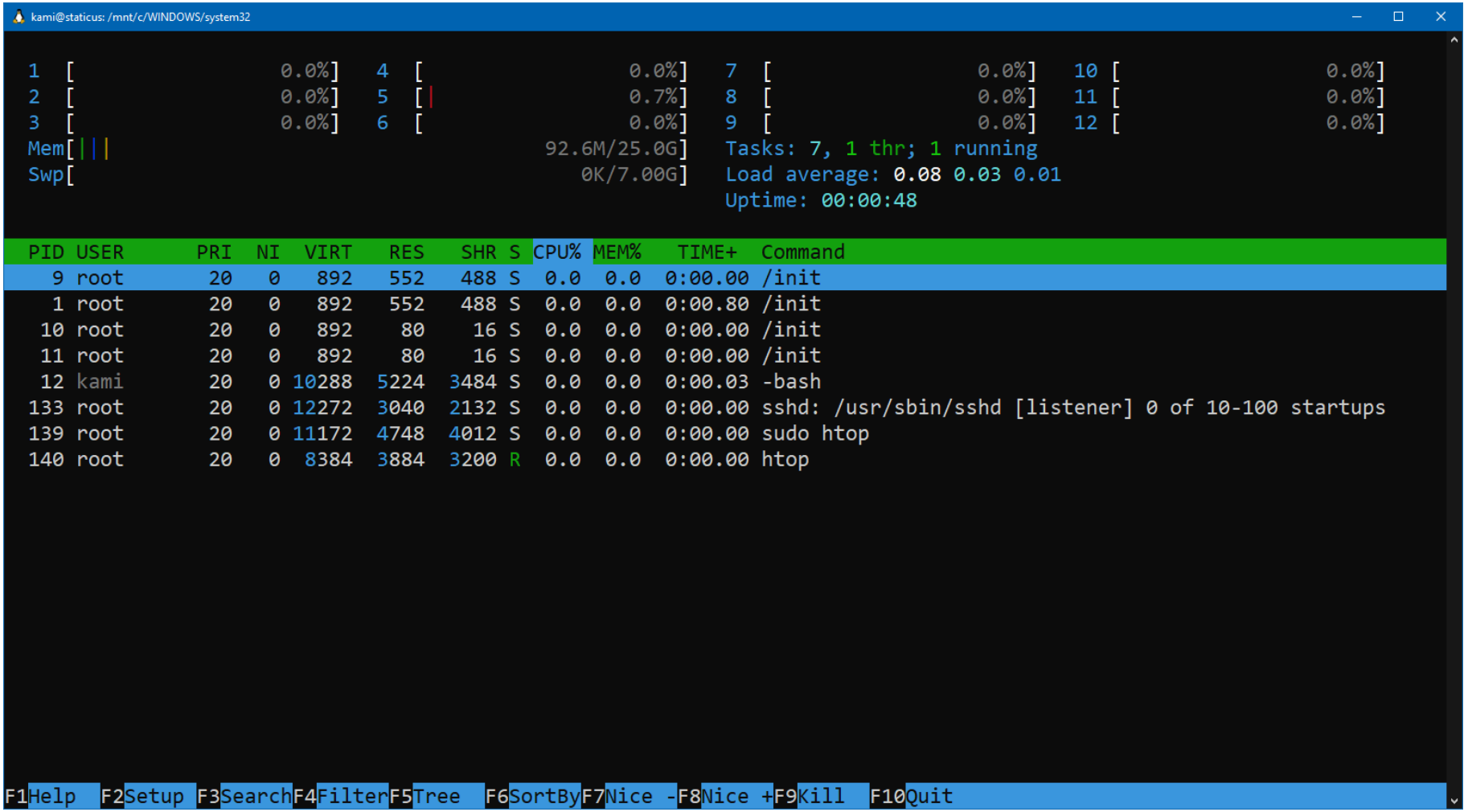
- Részei
  - A műveletsor (program)
  - Kimenő és bemenő adatok
  - Folyamat állapota
- Alapvetően egy folyamatnak öt állapota lehet
  - Futó: a CPU éppen ezt a folyamatot futtatja
  - Kész (ready): futtatható, éppen áll, hogy egy másik folyamat futhasson
  - Blokkolt: vár egy külső esemény bekövetkezésére (pl. I/O)
  - Megszakított: a folyamatoknak jeleket (signal) küldhetünk, melyekre megállnak
  - Halott: befejeződött a folyamat

# Folyamatok (process)

Tulajdonságok

- A folyamat kernelszintű fogalom, ezért minden egyede speciális adatokkal rendelkezik:
  - UID (User IDentification number)
  - GID (Group ID)
  - PID (Process ID)
  - Prioritás
  - Virtuális memória
  - Fájlleíró
  - …
- A folyamatleíró struktúra minden része a kernelben van
  - A felhasználói program nem tudja elérni azt.
- Más folyamatok csak rendszerhívásokon keresztül férnek hozzá a folyamat adataihoz, állapotához.
- A folyamat eljárásai, függvényei viszont a felhasználói területen vannak, így a folyamaton belül elérhetőek.

# Folyamatok (process)



# Folyamatok állapotai

## Halott

- Amint a folyamat utasításai elfogynak, akkor befejezi futását. Ez után már nem létezik, mint folyamat többé.

## Futó / kész

- A két állapot nagyon hasonlít egymásra, mindkettő esetén a processz alkalmas a futásra
- A lényeges különbség, hogy a futó folyamat éppen használja a CPU-t, míg a kész számára a CPU nem elérhető.
  - Egyprocesszoros/egymagos környezetben a CPU több folyamat között van megosztva
  - Ütemező algoritmus szabályozza, hogy mikor kell megállítani egy folyamat feldolgozását, hogy egy másik folyamatot szolgálhasson ki.

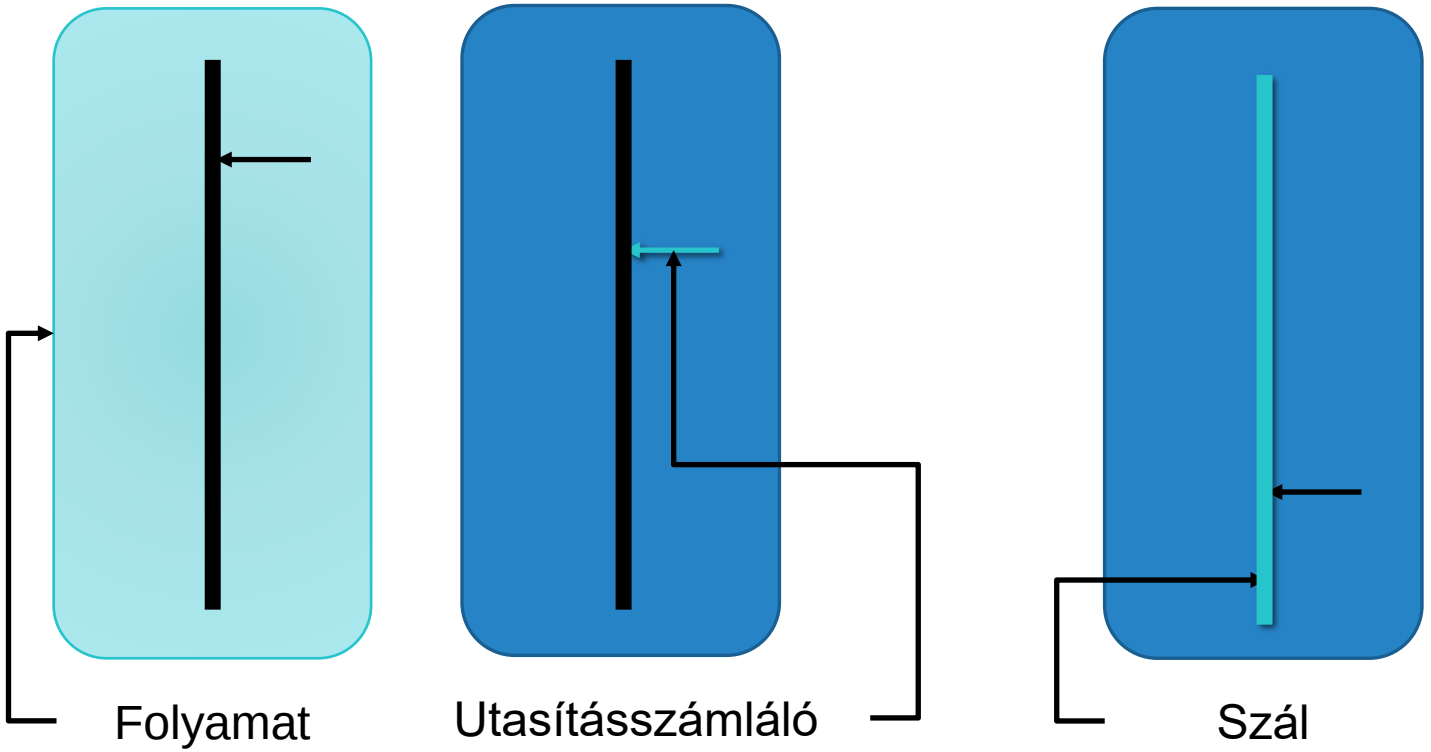
# Folyamatok állapotai

Blokkolt

- A folyamat nem futhat tovább (nem tud)
  - Még akkor sem, ha a CPU-nak nincs más feladata.
- A folyamat vár egy esemény bekövetkeztére
  - Egy lemez blokk beolvasására, egy karakter leütésére.
- Amint bekövetkezik az esemény processz kész állapotba kerül
  - Ezzel ütemezhetővé válik.

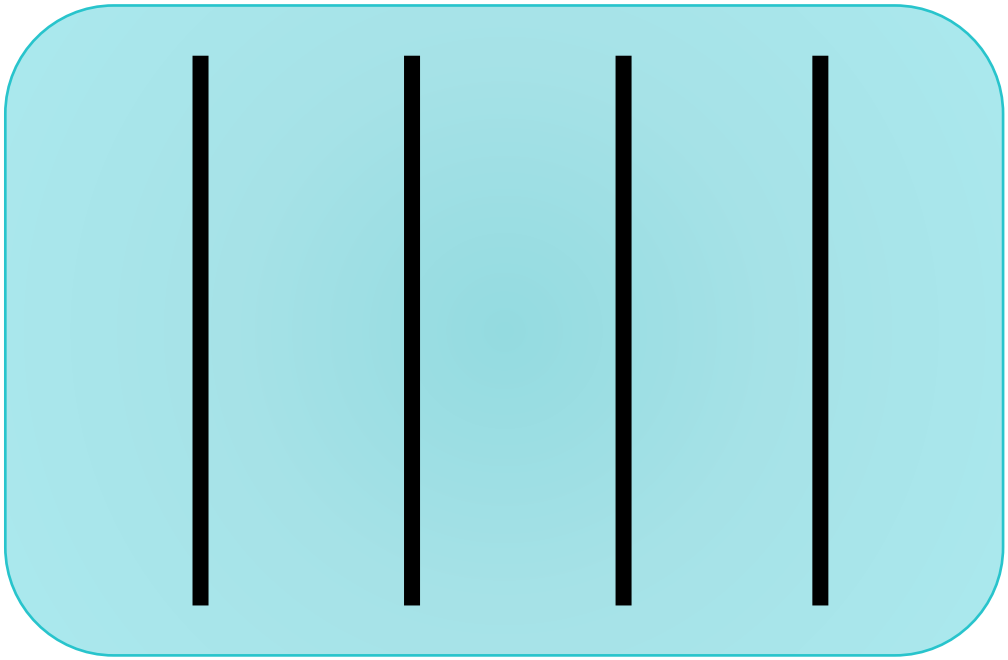
# Folyamatok részei

Egy hagyományos folyamatban egy vezérlő szál és egy utasításmutató található:



# Több szál egy folyamaton belül

A mai operációs rendszerek képesek már több vezérlő szál kezelésére egy folyamatban is.



Egy folyamat több szállal

# Szálak (thread)

Egy szál koncepcionálisan hasonlít egy folyamathoz

A szál felhasználói szintű fogalom

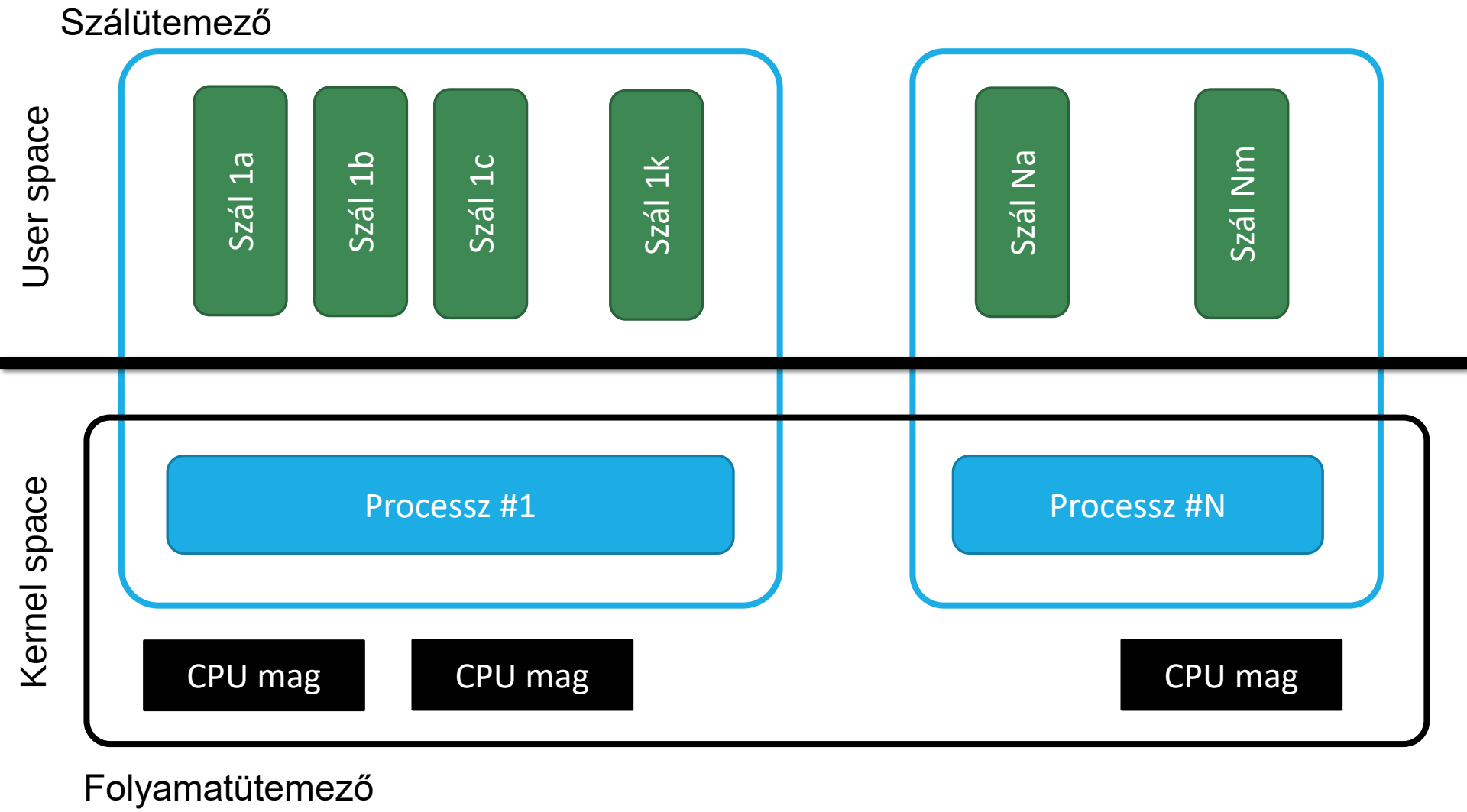
- (Nem kernel szintű)
- A szálleíróstruktúra is a felhasználói területen van, és ezért közvetlenül elérhető.
- A szálak, mint programokat futtató taszkok, a folyamatokhoz hasonlóan rendelkeznek regiszter adatokkal
  - utasításszámláló – program counter
  - veremmutató – stack pointer
    - Vagyis minden szálnak saját veremterülete van.
  - állapotleíró adatok
  - stb.

# Szálak (thread)

A szál felhasználói szintű fogalom

- A szálaknak ugyanúgy lehet futó, kész és blokkolt állapota.
- Egy folyamat minden szála osztozik az erőforrásokon
  - ugyanazon memória területet látja
  - ugyanazokat  a műveleteket és adatokat használja mindegyik szál.
- Például
  - Ha egy szál megváltoztat egy folyamat szintű adatot, akkor a folyamat összes szála érzékelni fogja ezt az adat legközelebbi hozzáférésekor.
  - Ha egy szál megnyit egy fájlt olvasásra, akkor a többi szál is olvashatja ugyanazt a fájlt.

# Folyamatok és szálak



# Kapcsolatok processzek között

## Kommunikáció

- Ez a processzek közötti adatcserét jelenti
- A kommunikáció során egy folyamat hozzáfér egy másik, vele párhuzamosan futó folyamat által szolgáltatott információhoz.
- Történhet
  - Osztott változók használatával – a program változóinak egy része (esetleg minden változó) hozzáférhető egyszerre több folyamat számára.
    - Ekkor az egyik beírja az átadni kívánt információt egy változóba, a másik folyamat pedig kiolvassa onnan.
    - Fontos a folyamatok összehangolása
      - Ha két folyamat például egyidőben ír ugyanarra a memóriaterületre (ugyanabba a változóba) a kapott eredmény határozatlan érték lesz…

# Kapcsolatok processzek között

## Kommunikáció

- Történhet
  - Explicit üzenetküldéssel
    - Az egyik folyamat üzenetet küld a másik folyamatnak.
    - Aszinkron kommunikáció
      - Amikor egy folyamat üzenetet küld, akkor nem várja meg, hogy a másik folyamat fogadja az adott üzenetet
      - A küldő folyamat végrehajtása csak az üzenet elküldésének idejére kerül felfüggesztésre
    - Szinkron kommunikáció (randevú)
      - Amikor egy folyamat kommunikálni akar egy másik folyamattal, akkor végrehajtása felfüggesztésre kerül addig, amíg a megszólított folyamat alkalmas nem lesz az üzenetvételre.
      - A két folyamat csak a kommunikáció befejeztével futhat tovább.

# Kapcsolatok

## Szinkronizáció

- Az egyik processz vezérlését kapcsolatba hozza a másik processz vezérlésével
- p a P processz egy végrehajtási pontja és q a Q processzé, akkor
- A szinkronizációt használjuk arra, hogy megszorításokat tehessünk arra, hogyan éri el P p-t és Q q-t.
- Több probléma is felmerülhet, amelyek elkerülésére a konkrét megvalósításkor figyelnünk kell.
  - Például deadlock:
  - A holtpont – leegyszerűsítve – olyan állapot, melynél a folyamatok körkörösen egymásra várakoznak, és egyik sem tud tovább haladni.

# Példa

```
u1: x:=12;
u2: y:=x/4;
u3: a:=x+y;
u4: b:=x-y;
u5: c:=a*b;
u6: d:=x+1;
u7: e:=c+d;
```

Párhuzamosíthatók?

# Példa

```
u1: x:=12;
u2: y:=x/4;
u3: a:=x+y;
u4: b:=x-y;
u5: c:=a*b;
u6: d:=x+1;
u7: e:=c+d;
```

Bizonyos utasítások csak egymás után hajthatók végre, szekvenciálisan

- Például u1 és u2, hiszen u2 használja u1 eredményét

# Példa

```
u1: x:=12;
u2: y:=x/4;
u3: a:=x+y;
u4: b:=x-y;
u5: c:=a*b;
u6: d:=x+1;
u7: e:=c+d;
```

Nem mindegyik

- Az u3 és u4 egyszerre, egymással párhuzamosan is végrehajtható
- Nem függenek egymástól.

# Precedenciagráf

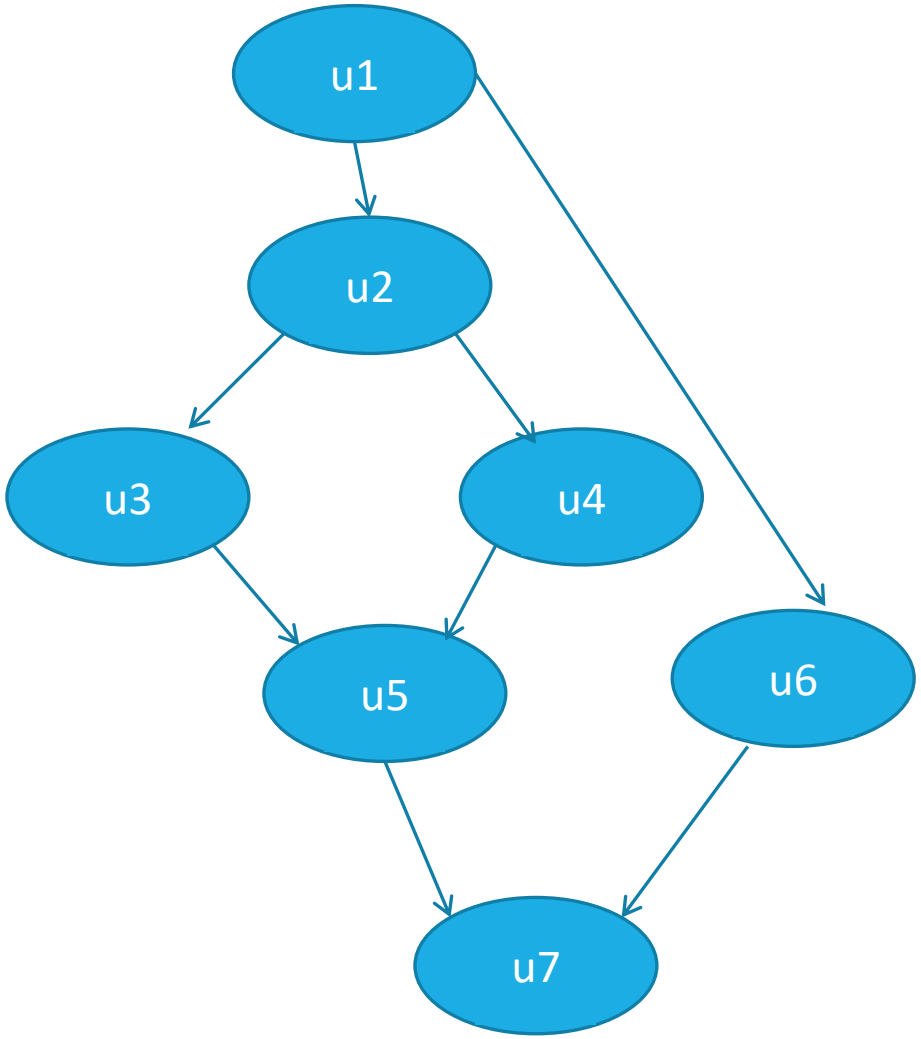
A lehetséges párhuzamosítást mutatja

- A csomópontok reprezentálják az utasításokat
- Két csomópont közötti (A és B) irányított él megadja a két utasítás sorrendjét.
  - A -> B esetén az A utasítás a végrehajtás során meg kell, hogy előzze a B utasítást.

# Példa

```
u1: x:=12;
u2: y:=x/4;
u3: a:=x+y;
u4: b:=x-y;
u5: c:=a*b;
u6: d:=x+1;
u7: e:=c+d;
```

Ha több él találkozik egy csomópontban: szinkronizálás!



# Precedenciagráf

A precedenciagráfok szemléletesen megadják a párhuzamosítási lehetőségeket

- Programozásra közvetlenül nem használhatók

Lehetőségek

- fork / join utasításpár
- parbegin / parend utasításpár

# Fork-join pár

fork utasítás

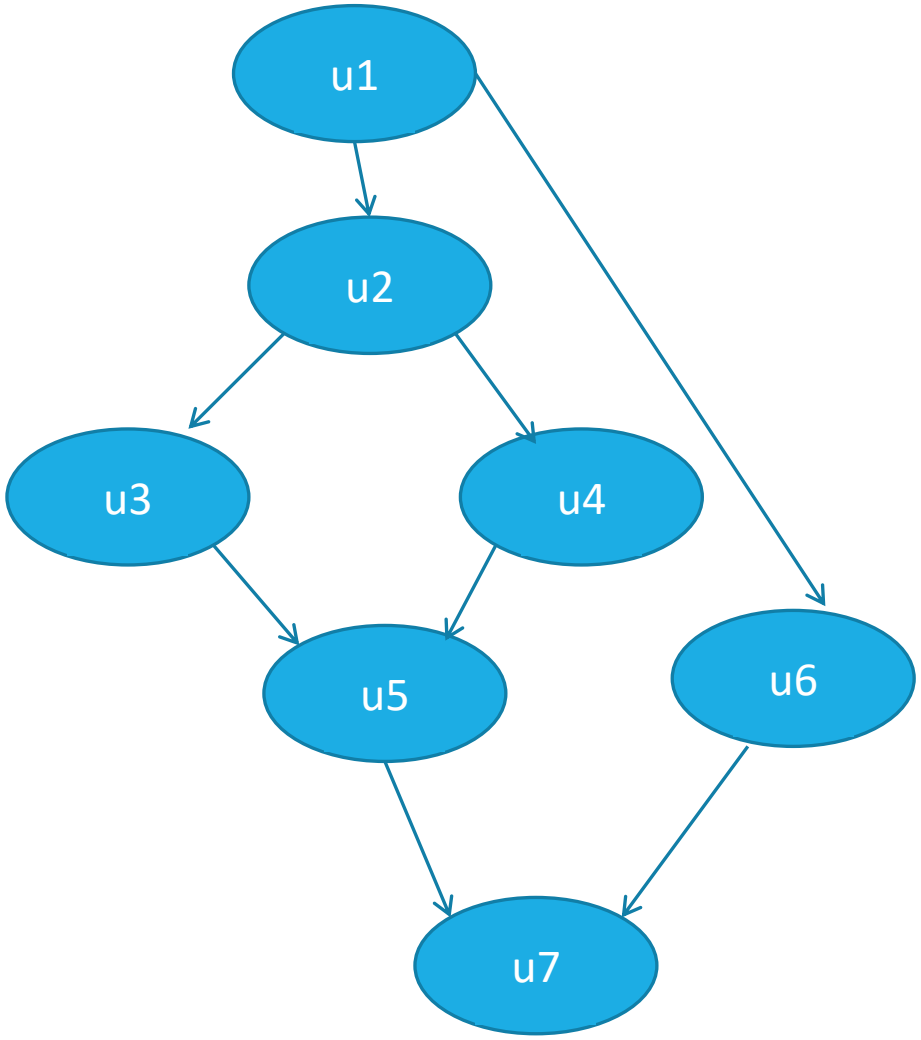
- A végrehajtás két egymással párhuzamosan végrehajtható ágra szakad
  - Az egyik ág közvetlenül a fork utasítás után folytatódik
  - A másik ág a megadott címkénél található

join utasítás

- A két vagy több ág egyesítése
- az ágak „bevárják egymást” (szinkronizáció) és csak a legutoljára „érkező” folytatódik, a többi „meghal”
- az egyesítendő  ágak számát egy számláló mutatja

# Fork-join pár

```
    száml2 := 2;
    száml1 := 2;
    u1;
    fork L1;
    u2;
    fork L2;
    u3; goto L3;
L2: u4;
L3: join száml1;
    u5; goto L4;
L1: u6;
L4: join száml2;
    u7;
```



# Parbegin-parend pár

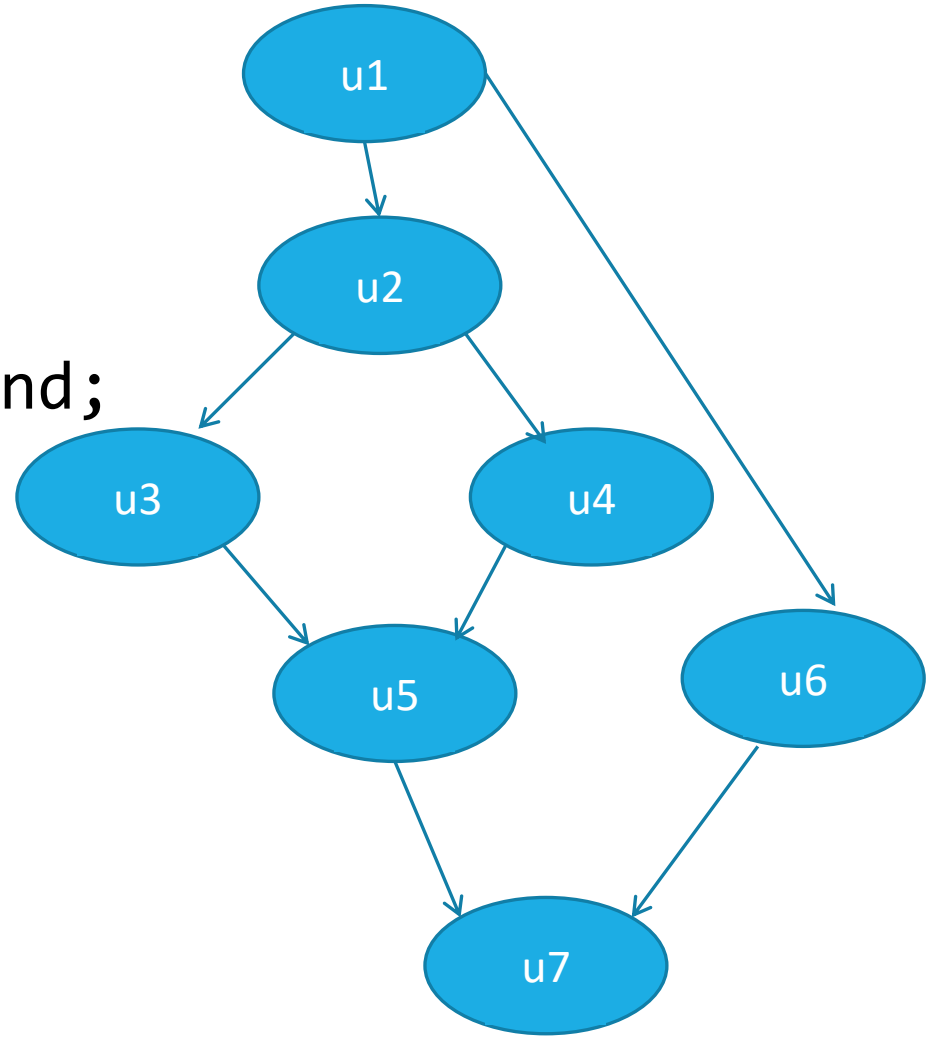
```
u1;

parbegin
u6;

begin
   u2;
   parbegin u3; u4; parend;
   u5;
end;

parend;

u7;
```




Ez nem jó minden precedenciagráfra

- ki kell kiegészíteni ilyenkor
  - Például szemaforokkal

# Viselkedés

A processzek viselkedésük alapján lehetnek

- Egymástól függetlenek (independent)
- Egymással versengők (competing)
  - Többen szeretnének megszerezni egy adott erőforrást, amit csak kizárólagos módon lehet használni
    - Repülőgépen egy adott járat adott ülőhelye
    - Adott nyomtató
  - Kölcsönös kizárás kell, szinkronizáció
- Egymással együttműködők (cooperating)
  - Amikor ugyanazon probléma különböző részein dolgoznak
    - Puzzle

# Függetlenség?

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Ketszal is
     task Egyik;
     task body Egyik is
     begin
          for i in 1..1000 loop
               Put_Line("Szia!");
          end loop;
     end Egyik;
begin
     for i in 1..1000 loop
          Put_Line("Viszlát!");
     end loop;
end Ketszal;
```

# Lehetséges kimenetek

Példaprogramok és eredményfájlok put-ra és print-re

- Ketszalu_put.adb – ketszal_put.txt
- Ketszalu_print.adb – ketszal_print.txt
- Ketszalu_put_print.adb – ketszal_put_print.txt
- Ketszalu_put_line.adb – ketszal_put_line.txt

Tapasztalatok

- Sokszor az egyik (például Viszlát!)
- Utána sokszor a másik
- Nem ugyanannyiszor
- Változik futás közben, hogy melyikből mennyi
- Futásról futásra is változik

# Ütemezés jelentősége

A futtató rendszer ütemezi a szálakat

- Operációs rendszer
- Futtató környezet (Előző példában Ada; Java)
- Programozó (Co-rutionk)

Például az Ada nyelv nem tesz megkötést az ütemező algoritmusra

- Egy implementációban bármilyen ütemező lehetséges

Jó program: bármely ütemezésre jó

# Időosztásos ütemezés

Jellemzők

- Nem valódi párhuzamosság
- Egy processzor: egyszerre csak egyet
- Időszeletek (time slicing)
- Gyorsan váltogatunk, párhuzamosság látszata
- Egyenlő időszeletek: pártatlanság
- Kooperatív viselkedés

# Kompetitív viselkedés

Jellemzők

- „Fusson, amíg van mit csinálnia”
- Amíg számol, addig övé a vezérlés
- Vezérlés elvétele: csak speciális pontokon
  - blokkolódás, szinkronizáció
- Hatékonyabb, mint az időszeletes ütemezés
  - Kevesebb kontextusváltás (context switch)
- Nem pártatlan:
  - Egy taszk kisajátíthatja az erőforrásokat

# Pártatlanság, kooperáció

Egy taszk lemondhat a vezérlésről

delay utasítás:

```
loop
    Put_Line("Szia!");
    delay 0.0;
end loop;
```

Jellemző kimenet: felváltva írnak ki, de ez sem mindig

- Ketszalu_put_delay.adb – ketszal_put_delay.txt

# Multiprogramozott rendszer

A multiprogramozott rendszer

- Szekvenciális programok halmazának tekinthető
- A programok üzenetek és szinkronizációs jelek segítségével együttműködnek egymással, és ugyanazon korlátozott számú erőforrásért versenyeznek.

A multiprocesszoros gépek megjelenése több kérdést vetett fel.

- Ezek közül az egyik a közös tár használata, amikor is a párhuzamos folyamatok a közös (osztott) változókon keresztül kommunikálnak.
- Az olyan rendszert, amelyik a konkurens programozást így valósítja meg, osztott rendszernek nevezzük.

# Multiprogramozott rendszer

Kritikus szakasz:

- egy multiprogramozott rendszerben különböző programok közös adatterületet használhatnak.
- A végrehajtás alatt vannak olyan szakaszok, amikor módosítják vagy vizsgálják a közös adatterület elemét.
- Ezek a szakaszok a „kritikus szakasz”-ok
- Megköveteljük, hogy
  - a kritikus szakasz végrehajtása véges idő alatt befejeződjön,
  - egy program, ha akar, mindig véges idő alatt beléphessen a kritikus szakaszába,
  - ha egy processz kritikus szakaszon kívül leáll, az ne befolyásolja a többi processzt.

# Multiprogramozott rendszer

Kölcsönös kizárás: annak biztosítása, hogy egyidejűleg csak egy program lehessen kritikus szakaszban.

Erőforrás: A rendszer valamely alkotórésze, amelyből véges mennyiségű van csak, és több processz akarhatja egyidejűleg használni.

# Kommunikációs modellek

# Szinkron kommunikáció

Amikor egy folyamat kommunikálni akar egy másik folyamattal, akkor a végrehajtása felfüggesztésre kerül addig, amíg a megszólított folyamat alkalmas nem lesz az üzenetvételre.

- A két folyamat csak a kommunikáció befejeztével futhat tovább.
- Előnye az aszinkron kommunikációval szemben, hogy az információ áramlása kétirányú lehet.
- Ezt a kommunikációs modellt használja pl. az Ada nyelv.

# Ada – szinkron kommunikáció

Randvú – Taszkok szinkronizációjához és kommunikációjához használható

Belépési pont (entry) definiálható az egyik taszkban, amihez az accept utasítással kódot rendelünk

A másik taszkban meghívhatjuk a belépési pontot

A belépési pontok is „callable unit”-ok, mint az alprogramok

# Példa

```
task HF is
     entry Bead;
end HF;

task body HF is
begin
     accept Bead;
end HF;
```

```
task Diak;

task body Diak is
begin
     HF.Bead;
end Diak;
```

# Randevú az Adában

Tulajdonságok

- Aszimmetrikus
  - Megkülönböztetjük a hívót és a hívottat
    - diák és HF
- Másként működik a két fél
  - Szintaxisban is kimutatható a különbség
- A hívó ismeri a hívottat, de fordítva nem
- A „hívó” és a „hívott” csak szerepek, amik egy randevúra vonatkoznak, nem egy egész taszkra
  - ugyanaz a taszk az egyik randevúban lehet hívó és a másikban hívott

# Randevú az Adában

## Tulajdonságok

- Szinkron kommunikáció
  - A randevú akkor jön létre, amikor mindketten akarják
  - Bevárják egymást a folyamatok az információcseréhez
  - Az aszinkron kommunikációt le kell programozni, ha szükség van rá
  - Pont-pont kapcsolat
  - Egy randevúban mindig két taszk vesz részt

## Az accept törzse:

- A randevú az accept utasítás végrehajtásából áll
- Ez alatt a két taszk együtt van
- A fogadó taszk hajtja végre az accept-et
- Az accept-nek törzse is lehet, egy utasítássorozat

# Példa

```
task HF is
     entry Bead;
end HF;
task body HF is
begin
     accept Bead do ... end Bead;
end HF;
```

```
task Diak;

task body Diak is
begin
    HF.Bead;
end Diak;
```

# Információcsere

Kommunikáció

- A randevúban információt cserélhetnek a taszkok
- Paramétereket használunk erre a célra
- Az entry specifikációja formális paramétereket is tartalmazhat
- A kommunikáció kétirányú lehet,
  - Adában a paraméterek módja szerint (in, out, in out)
- Az alprogramok hívására hasonlít a mechanizmus

# Példa

```
task Tarolo is
      entry Betesz( C: in Character );
      entry Kivesz( C: out Character );
end Tarolo;

task body Tarolo is
      Ch: Character;
begin
      loop
            accept Betesz( C: in Character ) do
                  Ch := C;
            end Betesz;
            accept Kivesz( C: out Character ) do
                  C := Ch;
            end Kivesz;
      end loop;
end Tarolo;
```

# Példa – próba

```
Ch: Character;
begin
    ...
    Get(Ch);
    Tarolo.Betesz(Ch);
    ...
    Tarolo.Kivesz(Ch);
    Put(Ch);
    ...
end;
```

# Aszinkron kommunikáció

## Aszinkron kommunikáció üzenetekkel

- Ha egy folyamat üzenetet küld, akkor az üzenet a fogadó folyamatnál egy üzenetsorba kerül.
- A küldő folyamat végrehajtása csak az üzenet elküldésének idejére függesztődik fel.
- Az üzenet feldolgozásához a fogadó folyamatnak ki kell venni az üzenetsorából az üzenetet.
- Ilyen kommunikációs modellt valósít meg például a PVM-rendszer.

# Aszinkron kommunikáció Adában

```
task A;
task body A is
     Ch: Character;
begin
     ...
     Get(Ch);
     Tarolo.Betesz(Ch);
     ...
end;
```

```
task B;
task body B is
     Ch: Character;
begin
     ...
     Tarolo.Kivesz(Ch);
     Put(Ch);
     ...
end;
```

Az A és a B taszk a Tároló-n keresztül aszinkron módon kommunikálnak.

# Osztott memória használata

Közös változók

- A folyamatok ugyanazt a memóriarészt látják.
- Probléma adódhat abból, hogy két folyamat egy időben írhatja az adott erőforrást (pl. változót), ilyenkor az eredmény határozatlan érték lesz.
- Hasonló probléma merülhet fel egy írási és egy olvasási művelet összeakadásánál.

# Nyelvi elemek

Milyen nyelvi elemekre van szüksége egy programozási nyelvnek a párhuzamosság támogatására?

- Elsősorban a párhuzamosan elindított kód részeit leíró nyelvi elemekre (task, szál, folyamat, process) és ezeknek a végrehajtását szolgáló leírásra, módszerre.
- Egy folyamat most éppen végrehajtható-e?

# Nyelvi elemek

Kommunikációhoz

- Osztott adatok esetén kölcsönös kizárást garantáló eszközök.

Két fő irányzat:

1. nyelvi mechanizmus az adatok védésére (pl. szemaforok, monitorok).
2. a folyamatok üzenetek révén kommunikálnak az üzenetek eljáráshívásnak felelnek meg, a megosztott adatok pedig az átadott paraméterek

Konkurrens részek szinkronizálására megfelelő eszközök

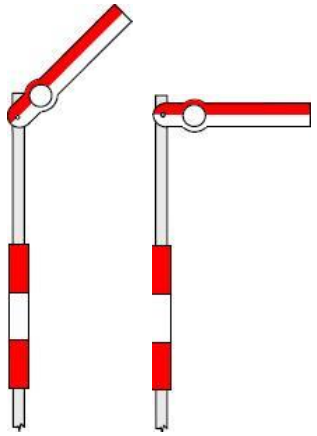
A prioritások meghatározása

Késleltetések, időzítések kezelése

# Nyelvi elemek

## Szemafor

- A szemafort, mint típust, Dijkstra vezette be egy 1968-as művében
- Egy egész értékű számláló
  - Illetve egy hozzá tartozó várakozási sor.
- Egy inicializált szemafornak két megengedett művelete van:
  - P = passeren (áthaladni)
  - V = vrijmaken (szabaddá tenni)
- A p - wait, a v - signal.



# Nyelvi elemek

## Szemafor

- A műveletek szemantikája az alábbi módon van definiálva:
- S: semaphore;
  - wait(S):
    - Ha S>0, akkor S:=S-1
    - Különben a folyamat blokkolódik, és a szemaforhoz tartozó várakozási sorba kerül mindaddig, amíg valaki fel nem ébreszti
      - azaz rá vagyunk utalva a többi folyamatra
  - signal(S):
    - Ha van várakozó folyamat az S-hez tartozó várakozási sorban, akkor felébreszti
    - Különben S:=S+1

# Lehetséges megvalósítás Adában

```
task A;                              -- kritikus szakasz előtti
task Szemafor is                     --   utasítások
     entry P;                        Szemafor.P;  -- megvárjuk,
     entry V;                                     -- amíg beenged
end Szemafor;                        -- kritikus szakasz utasításai
                                     Szemafor.V;
task body Szemafor is                -- kritikus szakaszok közötti
begin                                --  utasítások
     loop                            Szemafor.P;
          accept P;                  -- kritikus szakasz utasításai
          accept V;                  Szemafor.V;
     end loop;                       -- kritikus szakasz utáni
end Szemafor;                        --  utasítások
```

# Általánosított szemafor

```
task type Szemafor ( Max: Positive := 1 ) is
     entry P;
     entry V;
end Szemafor;
```

Legfeljebb Max számú folyamat tartózkodhat egy kritikus szakaszában

# Általánosított szemafor megvalósítása

```
task body Szemafor is
    N: Natural := Max;
begin
    loop
        select
        when N > 0 => accept P; N := N-1;
            or
                accept V; N := N+1;
            or
                terminate;
        end select;
    end loop;
end Szemafor;
```

# Probléma a szemaforral

A szemafor nagyon alacsony absztrakciós szintű eszköz

Könnyű elrontani a használatát

Akár a P, akár a V művelet meghívását felejtjük el, a program megbolondul

Nem lokalizált a kód, sok helyen kell odafigyeléssel használni

# Kölcsönös kizárás

A szemafor segítségével kritikus szakaszokat tudunk leprogramozni

Csak egy folyamat tartózkodhat a kritikus szakaszában

A kritikus szakaszra (a kritikus erőforráshoz való hozzáférésre) kölcsönös kizárást (mutual exclusion) biztosítottunk

# Monitor – Hoare, 1974

A monitor a folyamatok szinkronizálására használt eszköz és az osztott adatokról ad információt.

- Szerkezete:

```
monitor_név
 begin
  lokális adatok;
  eljárások;
  lokális adatok értékadásai;
end név.
```

- A monitor biztosítja, hogy csak egy folyamat hajthat végre egy monitor-eljárást egy adott időben.
- Mielőtt egy másik folyamat belép a monitorba, az előzőnek véget kell érnie.
- Sok nyelvben megtalálható a monitor, mint könyvtári osztály.

# Az étkező filozófusok problémája

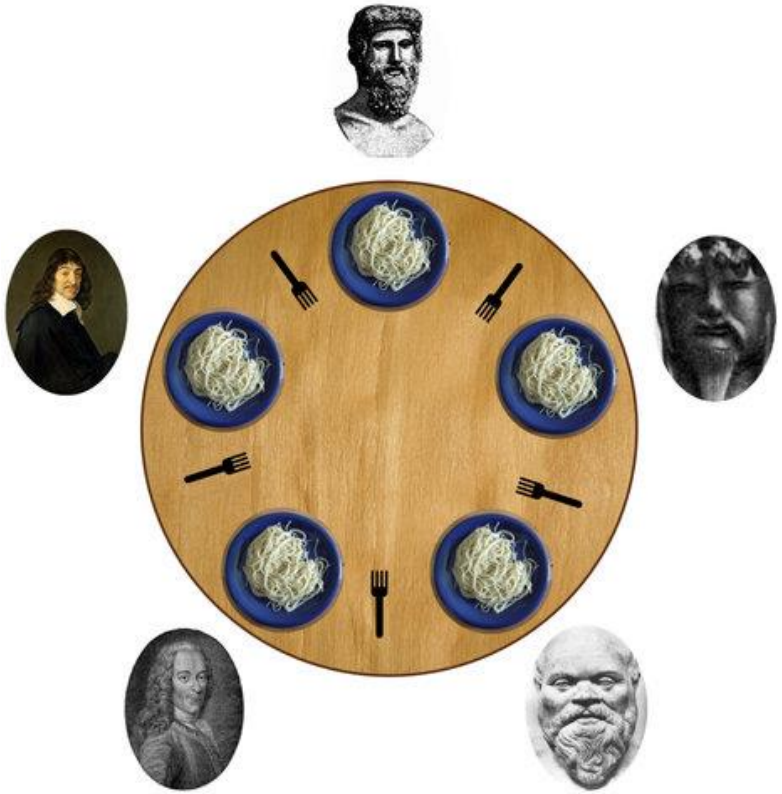
Dijkstra: dining philosophers

Erőforráshasználat modellezése

- Erőforrások: villák
- Folyamatok: filozófusok

Holtpont kialakulása

Éheztetés veszélye



# Holtpont

## Deadlock

- Ha a folyamatok egymásra várnak
- Ha egyszerre éheznek meg a filozófusok

## Elkerülés

- Például a szimmetria megtörésével:
  - Erőforrások lekötése rendezés szerint
  - Véletlenszerű időkorlátok
- Ajtó bevezetésével

# Kiéheztetés

Livelock, starvation

- Nincs holtpont (deadlock)

Fut a rendszer

- De: egy folyamat mindig rosszul jár
- Nem képes elvégezni a feladatát
- Példa: ügyetlen „író-olvasó” megvalósításnál
  - Az olvasók könnyen kiéheztethetik az írókat
- Példa: ügyetlen „evő filozófusok” megvalósításnál
  - Szimmetria megtörése a filozófusok beszámozásával
  - A kisebb sorszámú előnyt élvez a nagyobb sorszámúval szemben

# Író-olvasó feladat

Van egy több folyamat által használt erőforrás

Lehet változtatni („írni”) és lekérdezni („olvasni”)

Az olvasások mehetnek egyidőben

Az írás kizárólagos

A monitornál megengedőbb

# Egy task írhat egyszerre

```
task Kizaro is
   entry Kiir( Str: String );
end Kizaro;

task body Kizaro is
begin
   loop
    accept Kiir(Str: String) do
          Text_IO.Put_Line(Str);
    end Kiir;
   end loop;
end Kizaro;
```

# Egy task írhat egyszerre

```
protected Vedett is
      procedure Kiir ( ... );
end Vedett;

protected body Vedett is
      procedure Kiir ( ... ) is ... end;
end Vedett;

Vedett.Kiir(...);
```

# További lehetőségek

Lehetséges nyelvi elemek még

- A fork egy párhuzamos folyamat elindítását váltja ki.
- A join újra egyesíti a párhuzamos folyamatokat.
- A cobegin-coend a párhuzamos folyamat kezdeti, illetve végpontját jelöli.
  - Ugyanilyen nyelvi elem a parbegin-parend is.
- A select feltételhez kötött párhuzamos indítást valósít meg.
- A wait, delay késleltet egy folyamatot.
- A quit befejez egy processzt.

# Nyelvek

# Nyelvek osztályozása

## Vezérlési modellek szerint

- „Control driven” típusú vezérlés esetén az elindított folyamatok ellenőrzése fork, join, wait parancsokkal történik.
  - Ez egy központosított ellenőrzés, mivel a folyamatok közös memóriaésszel dolgoznak.
- A „data driven computation” modellben a folyamatok közvetlen módon kommunikálnak, közös memória használata nélkül.
  - A végrehajtási sorrend az adatok közötti összefüggésen alapul.
- A „demand driven computation” modellben a folyamatok akkor kerülnek végrehajtásra, amikor ezt egy másik folyamat kifejezetten kéri, a vezérlést mindig a kérések határozzák meg.
  - Például: Parlog, Concurrent Prolog, GHC.

# Nyelvek osztályozása

Ezeknek a modelleknek megfelel egy-egy programozási nyelvosztály.

Az ellenőrzéssel vezérelt nyelvek csoportja lehet: szinkron vagy aszinkron programozási nyelv.

A szinkron nyelvek esetében a folyamatok ugyanolyan típusú műveleteket hajtanak végre ugyanolyan típusú adatcsomagokra.

- Ezek többprocesszoros gépek programozási nyelvei.

# Nyelvek osztályozása

Az aszinkron nyelveket konkurens és osztott programozásra használják.

Ezeket három nagy csoportba sorolhatjuk:

- Eljárás orientált nyelvek: a folyamatok közötti kommunikáció az osztott változók segítségével történik.
  - Például: Concurrent Pascal, Pascal Pluss, Modula-2, Ada.
- Üzenetküldés orientált nyelvek: a kommunikálás send-receive típusú utasításokkal történik.
  - Például: Occam, Ada, CSP.
- Művelet orientált nyelvek: az előbbiek kombinációi.
  - Például: StarMod, Ada.

# Osztott változókkal rendelkező nyelvek

## Concurrent Pascal

- A processz, monitor és osztály fogalmaival dolgozik, valamint a delay és continue utasításokkal és a queue várakozási sor típussal.
- A processz szekvenciális program, mely egy időben futtatható más processzel.
- A monitor a processzek között osztott változók egységbezárása, az osztály pedig absztrakt adattípus.
- Az osztály példánya egy processzhez vagy monitorhoz csatolható.
- A monitor biztosítja adatainak a védelmét, a monitor eljárásai pedig egy várakozási sorban állnak, egy adott pillanatban csak 1 eljárás futtatható.
- A monitor egy eljárása felfüggeszthető a delay-vel vagy folytatható a continue-val, az inicializálás pedig init-tel történik.

# Osztott változókkal rendelkező nyelvek

## Modula-2

- A processzek fogalmára épít
  - A processzek közötti kommunikáció kétféleképpen valósul meg: osztott változókkal, melyeket a monitorok tartalmaznak és jelek segítségével.
- A jelek a processz modulokból vannak exportálva.
  - Ezeket szinkronizálásra használják, ezek nem tartalmaznak adatot.
  - Egy processz küldhet jeleket (send) vagy pedig várhat jeleket (wait) egy másik processztől.
  - A jelek abban különböznek a szemaforoktól, hogy egy olyan jel, amelyet egy processz sem vár, nulla műveletnek számít.
- A korutinok a kvázipárhuzamosság szimulálására alkalmasak.
  - A korutinok párhuzamosan futó folyamatok, ezek egymást aktiválják.
  - Amikor az egyik folyamat újból aktív lesz, akkor onnan folytatódik ahol az előző vezérléscserénél megszakadt.

# Osztott változókkal rendelkező nyelvek

## Modula-3

- A Modula-3 -ban a folyamatok szálak, amelyek közös memóriaterületen kommunikálnak.
- A párhuzamosság eszközei a Thread modulban találhatók (Fork, Join).
- A kölcsönös kizárást a LOCK parancs valósítja meg.

# Üzenetküldéssel, osztott memóriával rendelkező nyelvek

Ezeknél a nyelveknél a következőket vizsgáljuk:

- Hogyan határozza meg a nyelv a processzeket, a szinkronizálást?
- Milyen az üzenetek szerkezete és küldése?
- Hogyan kezeli az esetleges kommunikációs hibákat?

# Simula-67

A Simula-67 nyelvben a korutin valamilyen osztályba tartozó objektum, amely azonos vagy más osztályba tartozó objektumokkal működik együtt.

- A létrehozott objektum (korutin) alárendeltségi viszonyban van azzal az objektummal, ami létrehozta.
- A létrehozott objektum a detach utasítással adhatja vissza a vezérlést a létrehozó objektumnak.
- A létrehozó objektum a call(X) utasítással aktivizálhatja újra a leválasztott objektumot.
- Az egyik korutinból (objektumból) a vezérlést a resume(X) utasítás segítségével adhatjuk át az X korutinnak (objektumnak).
- A vezérlés visszakapásakor az objektumok végrehajtása ott folytatódik, ahol a legutóbbi vezérlésátadáskor megszakadt.

# Simula-67

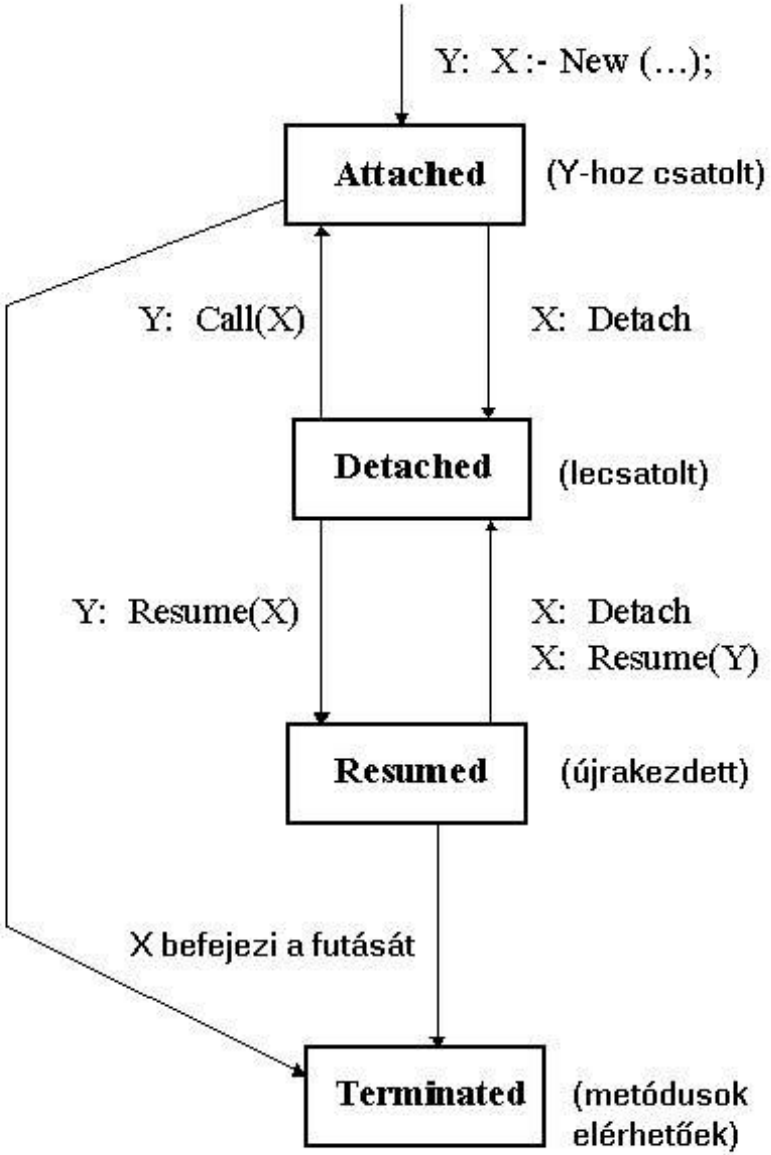
Egy korutin objektum élettartalma a következő:

- A generálódásakor az objektum elkezdi végrehajtani a törzsében található utasításokat.
  - Az objektum ilyenkor működő állapotban van, és a generátorhoz (az objektumhoz, ami létrehozta) hozzárendeltnek nevezzük.
- A korutin egy detach utasítással lemondhat a futásról a generátor javára.
  - Ilyenkor az objektumot leválasztottnak, de még nem lezártnak nevezzük.

# Simula-67

Egy korutin objektum élettartalma a következő:

- Az objektum egy resume(X) utasítással egy másik korutin javára mond le a vezérlésről.
- Az objektum újra vezérlést kaphat egy rá vonatkozó call(X) (a generátortól) vagy egy resume(X)
  - egy „testvér-folyamattól” utasítás hatására.
- Az objektum futása befejeződik, ha a törzsének végrehajtása elérte a befejező end kulcsszót.
  - Ilyenkor az objektum futását sem call sem resume utasítással nem lehet felújítani.
  - Az objektum azonban továbbra is létezik, tagváltozói elérhetőek, műveleteit meg lehet hívni.


Y: X :- New (…);
Attached
(Y-hoz csatolt)
Y: Call(X)
X: Detach
Detached
(lecsatolt)
Y: Resume(X)
X: Detach
X: Resume(Y)
Resumed
(újrakezdett)
X befejezi a futását
Terminated
(metódusok elérhetőek)

# Ada

Egy Ada programban egy folyamatot egy task-objektum reprezentál.

A taskok rendelkezhetnek belépési (entry) pontokkal.

- Ezek hívhatóak egy másik taszkból
- Így kommunikálhatnak a taszkok.

A taszknak van egy törzse, ez írja le azt a tevékenységet, amelyet a folyamat végez.

A törzsben minden entry-hez tartozik egy accept utasítás, amikor itt tart a törzs végrehajtása, akkor hajlandó elfogadni egy másik taszk hívását erre a randevúra.

A kommunikáció szinkron, a hívó folyamat a randevú elfogadásáig és az accept utasítás végéig felfüggesztődik.

Az entrynek lehetnek paraméterei (in, out vagy in out típusúak is).

Ha egy task meghívja egy másik taszk entry-pontját, akkor futása felfüggesztődik a randevú létrejöttéig és lekezeléséig.

# Ada – Randevú

```
task Lany is
    entry Randi;
end Lany;

task body Lany is
begin
    …
    accept Randi;
    …
end Lany;
```

```
task Fiu;

task body Fiu is
begin
    …
    Lany.Randi;
    …
end Fiu;
```

# Ada – Randevú

## Pont-pont kapcsolat

- Egy randevúban mindig két taszk vesz részt
  - Egy fiú és egy lány
- Szerepek
  - A taszkok szerepi felcserélődhetnek
- Nincs „broadcast” jellegű randevú

## Aszimmetrikus

- Megkülönböztetjük a hívót és a hívottat
  - Fiú és lány
- Másként működik a két fél
  - A szintaxisban is kimutatható a különbség
  - A hívó ismeri a hívottat, de fordítva nem

# Ada – Randevú

## Az accept törzse

- A randevú az accept utasítás  végrehajtásából áll
- Ez alatt a két taszk „találkozik”
- A fogadó taszk hajtja végre az accept-et
- Az accept-nek törzse is lehet, egy utasítássorozat

## Szinkronitás

- A randevú akkor jön létre, amikor mindketten akarják
- Bevárják egymást a folyamatok az információcseréhez
- Az aszinkron kommunikációt le kell programozni, ha szükség van rá

# Ada – Randevú

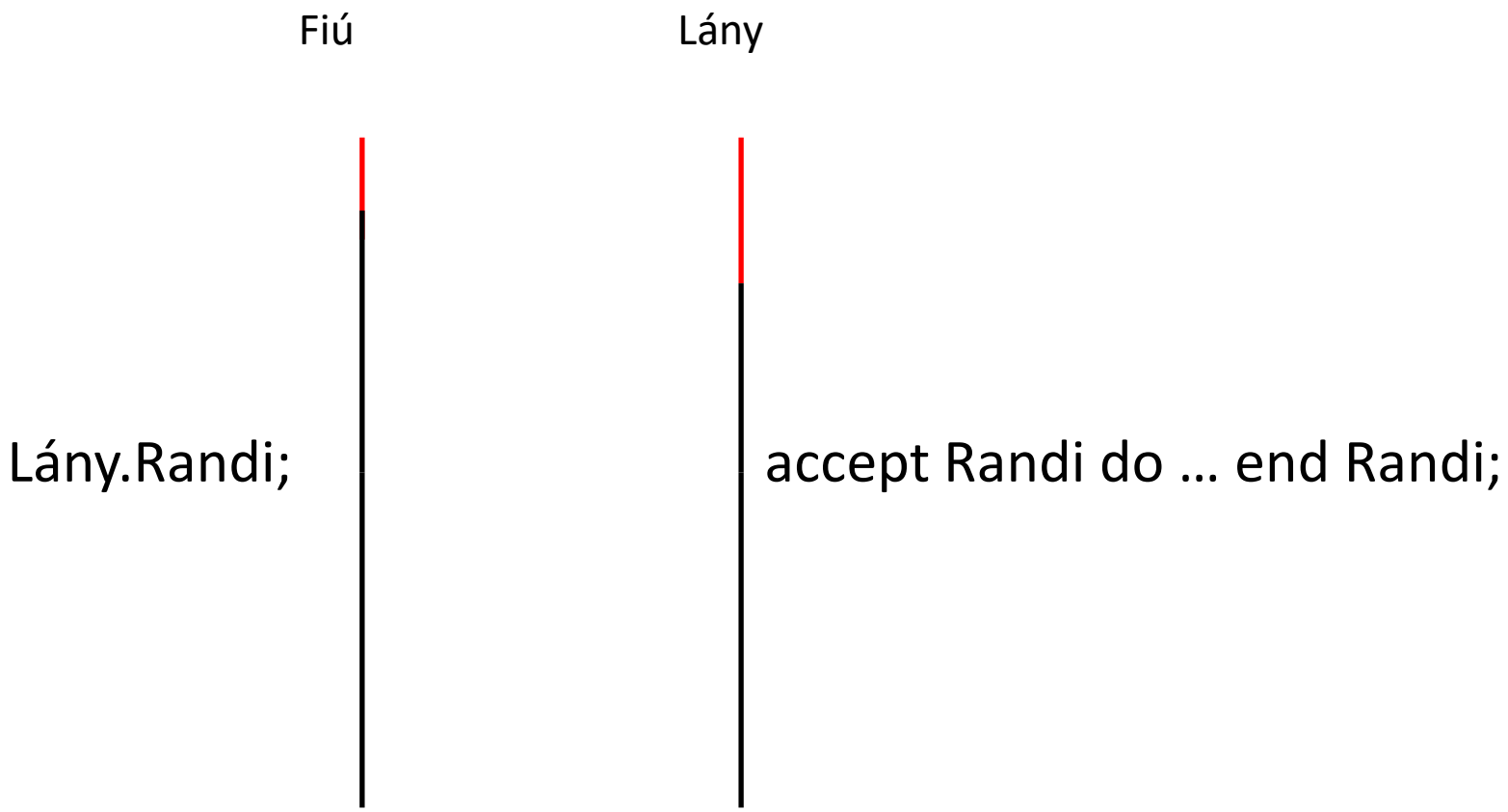
```
task Lany is
     entry Randi;
end Lany;

task body Lany is
begin
     …
     accept Randi do
     end;
     …
end Lany;
```

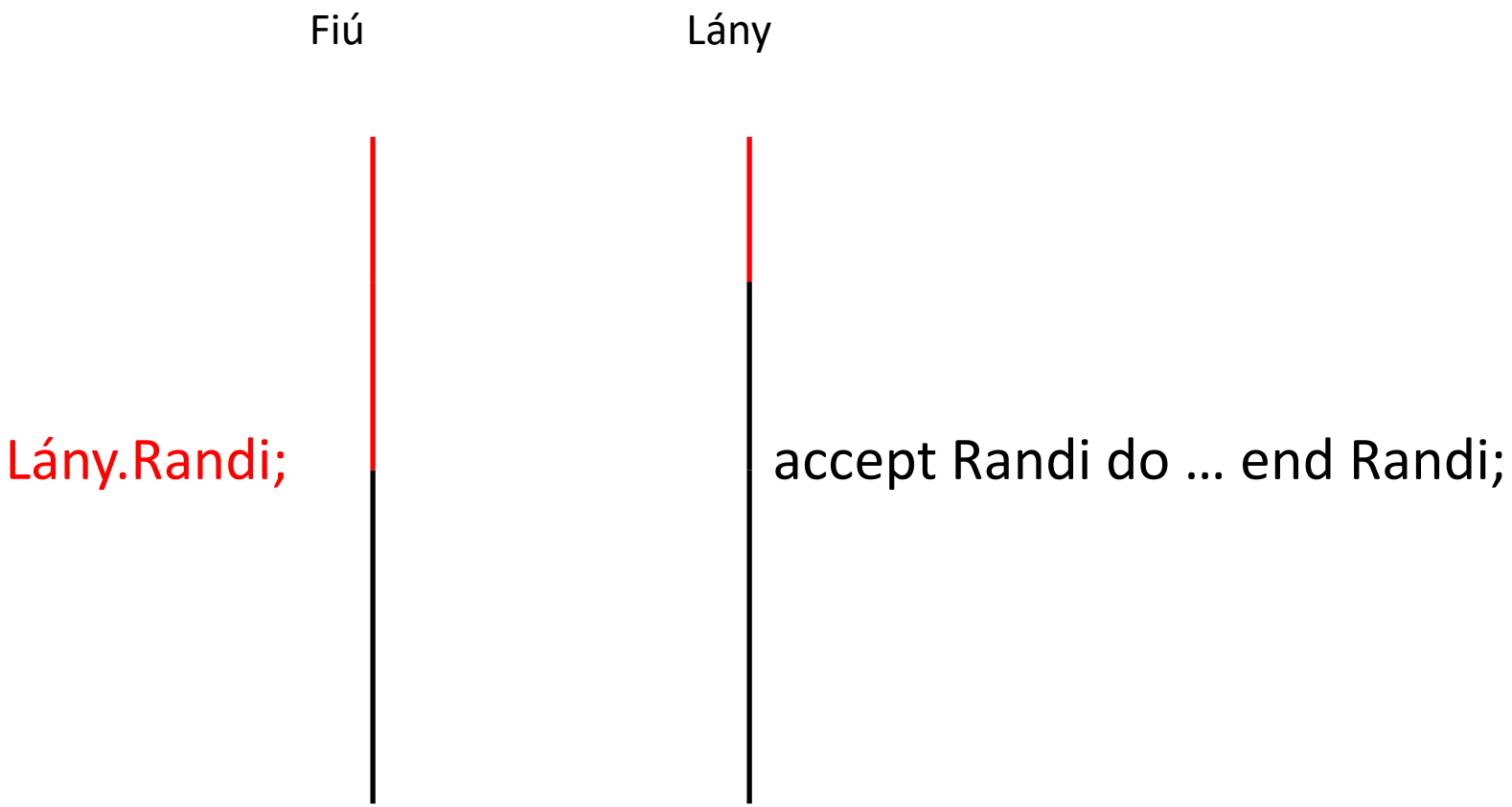
```
task Fiu;

task body Fiu is
begin
     …
     Lany.Randi;
     …
end Fiu;
```

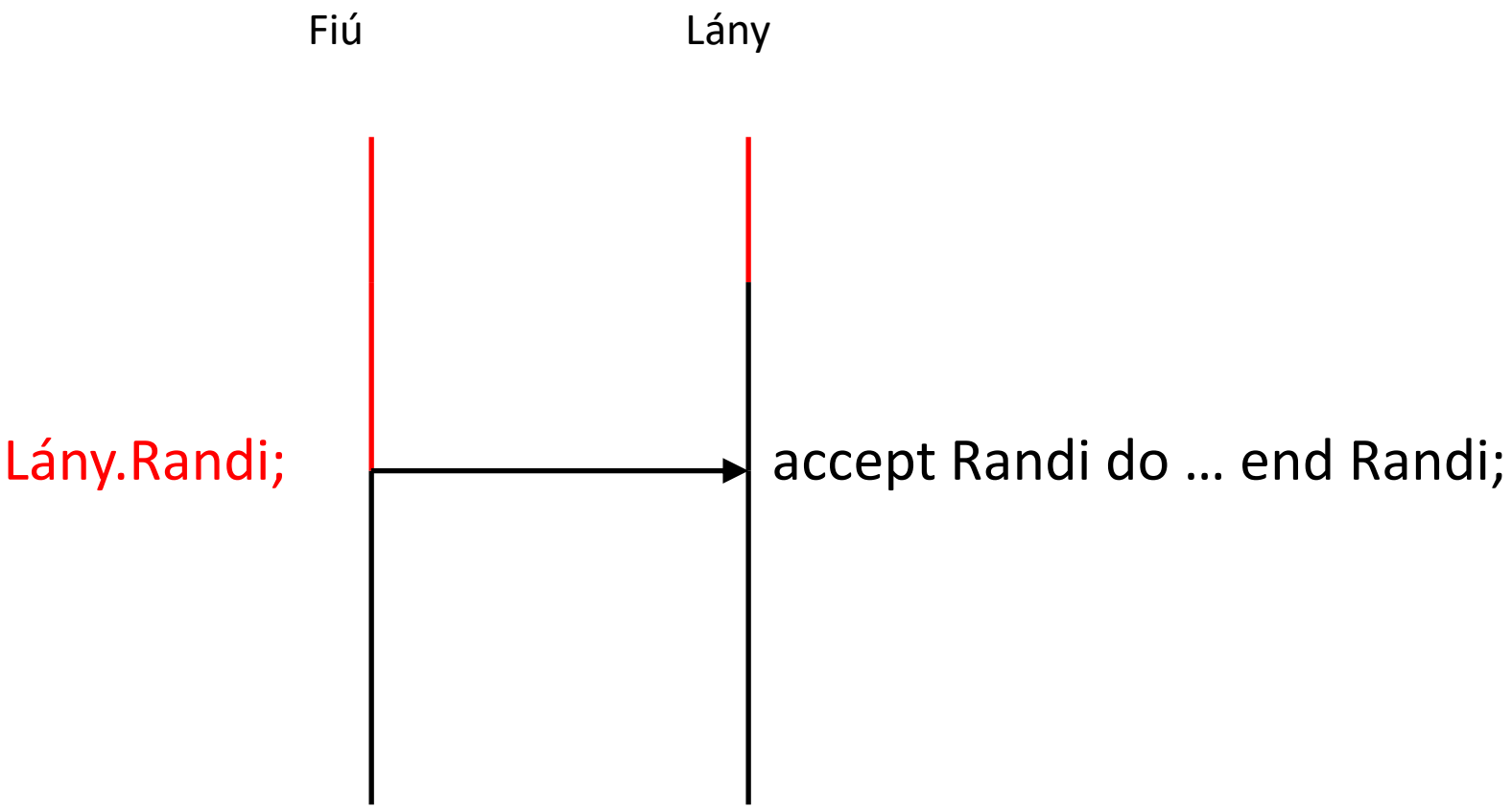
# Ada – Szinkronitás



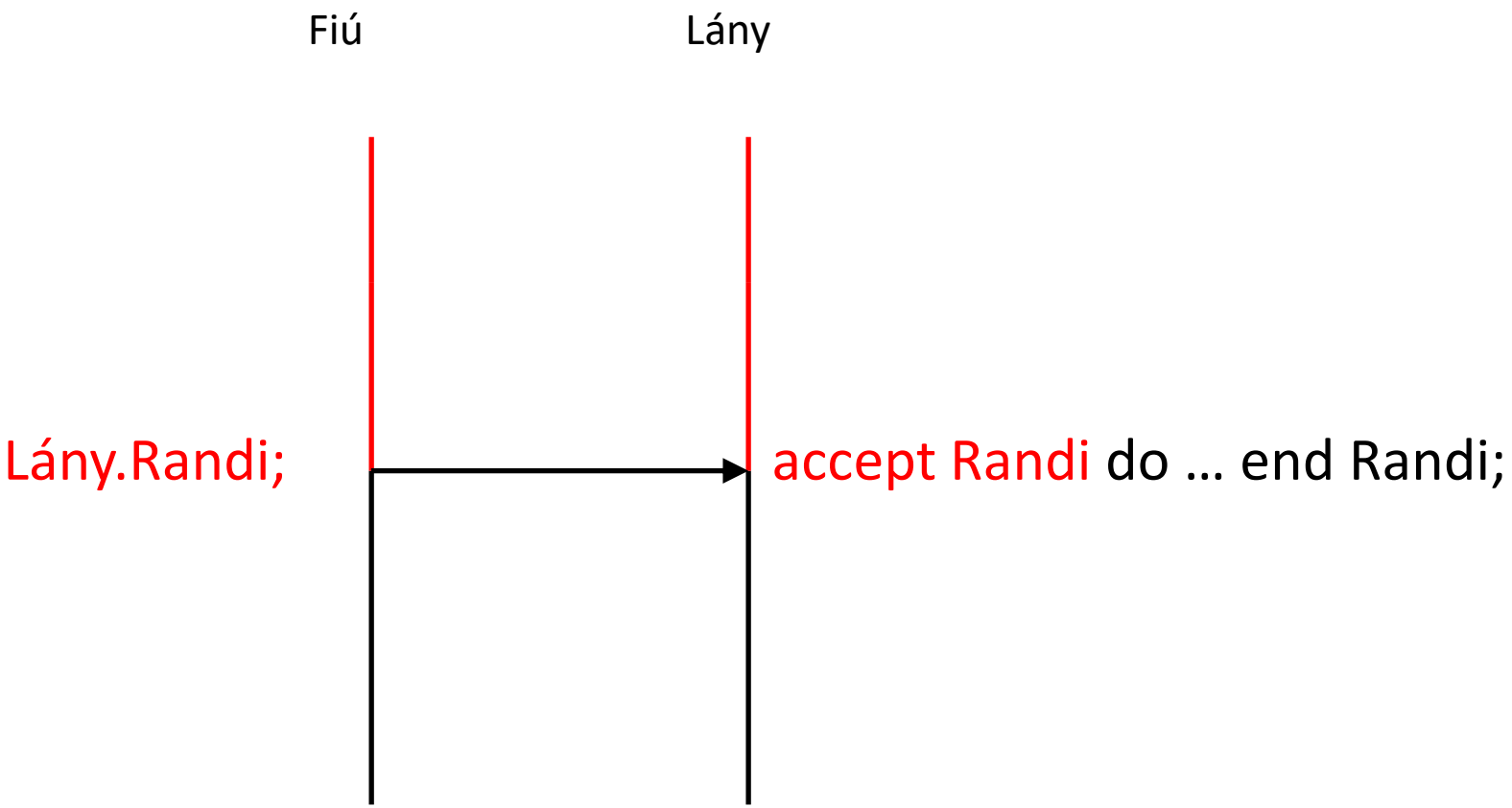
# Ada – Szinkronitás



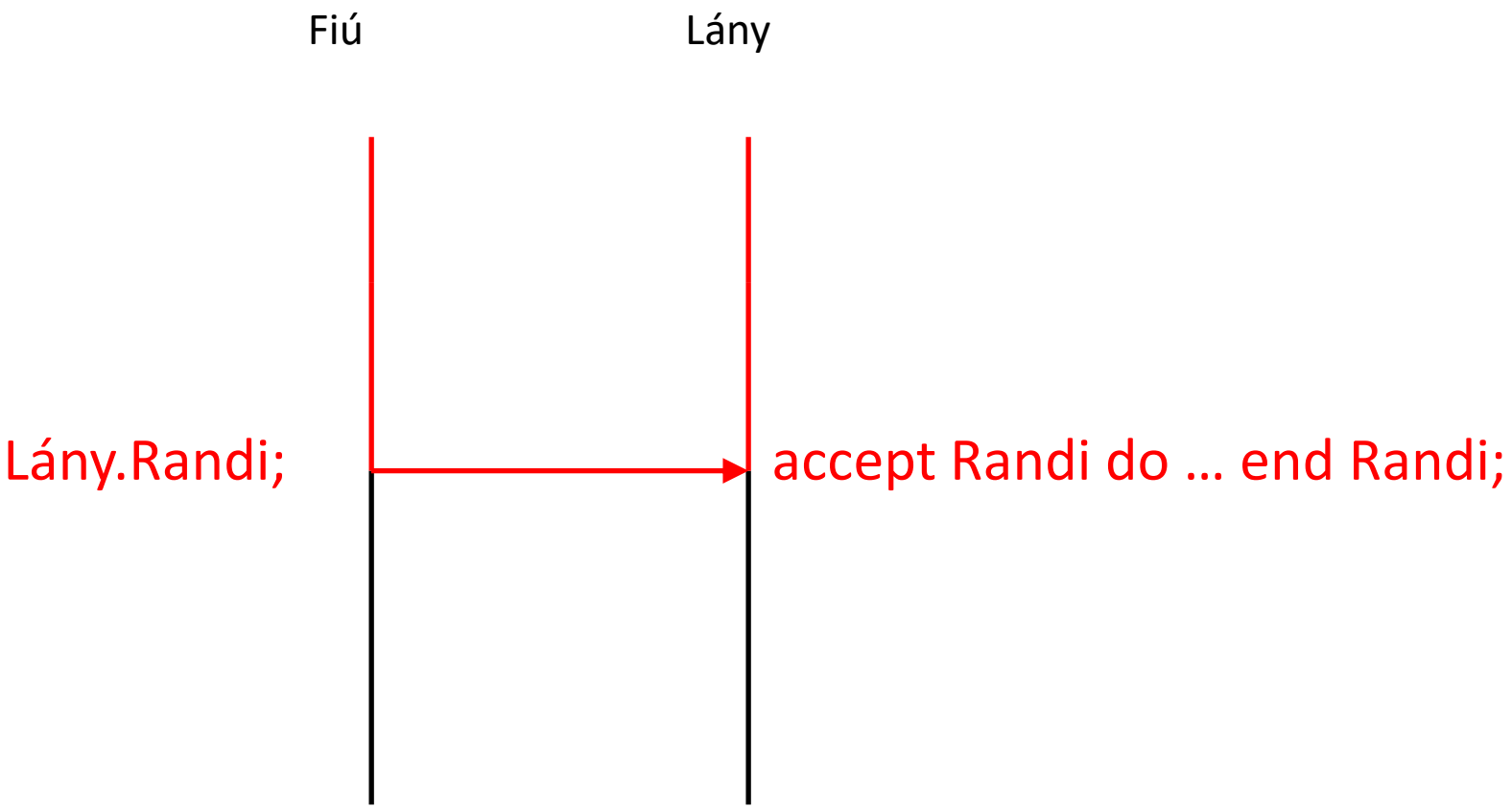
# Ada – Szinkronitás



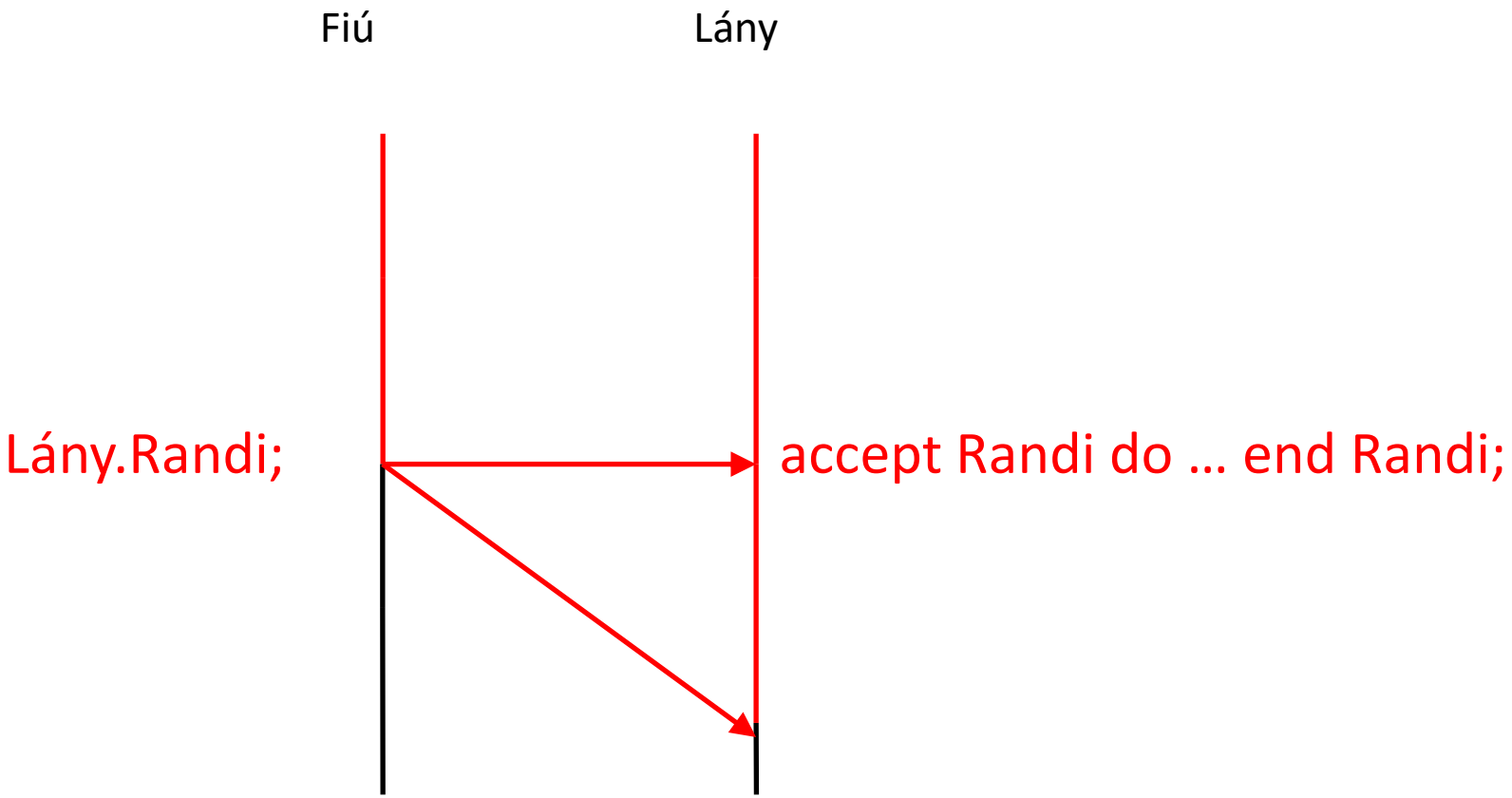
# Ada – Szinkronitás



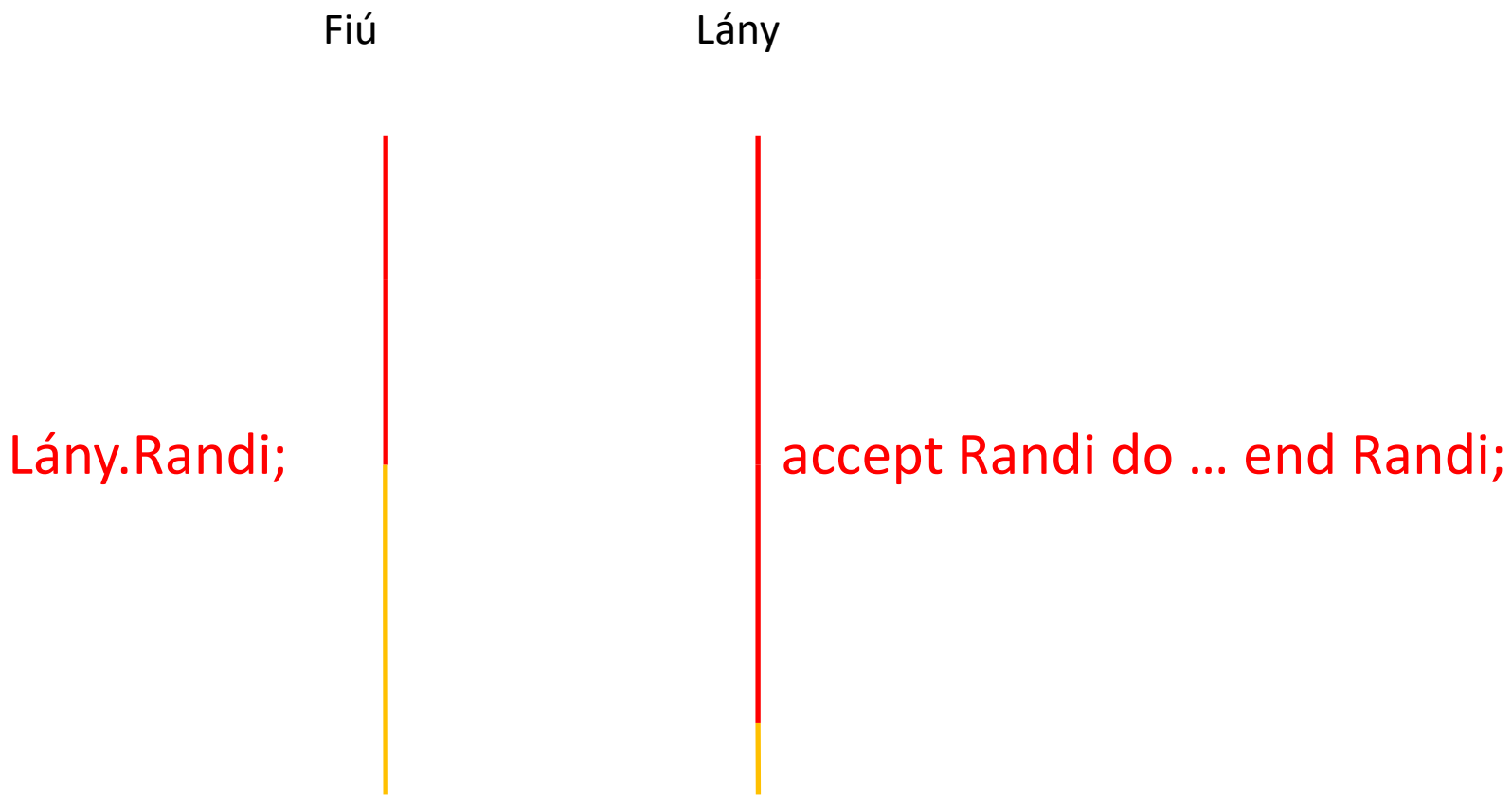
# Ada – Szinkronitás



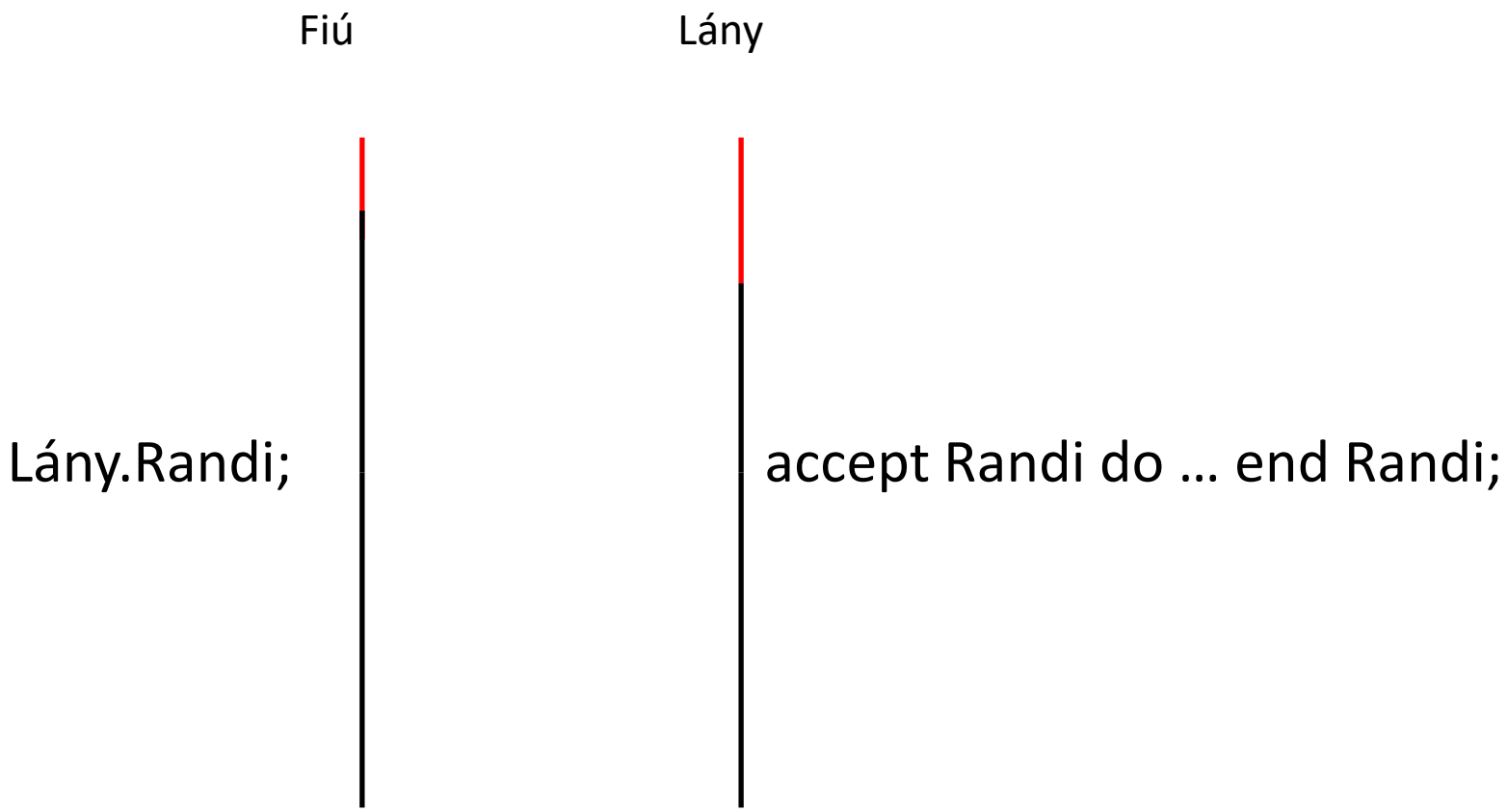
# Ada – Szinkronitás



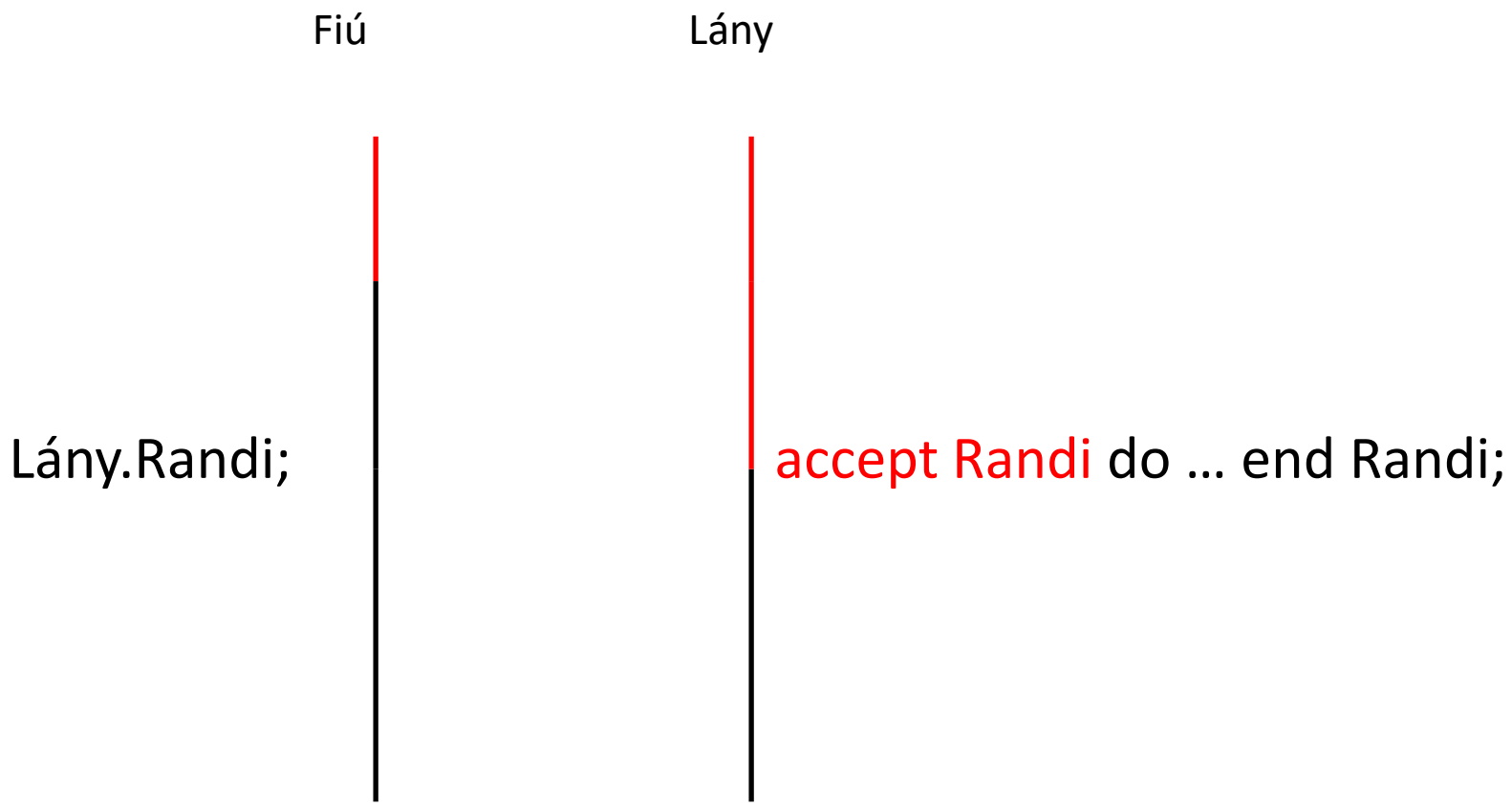
# Ada – Szinkronitás



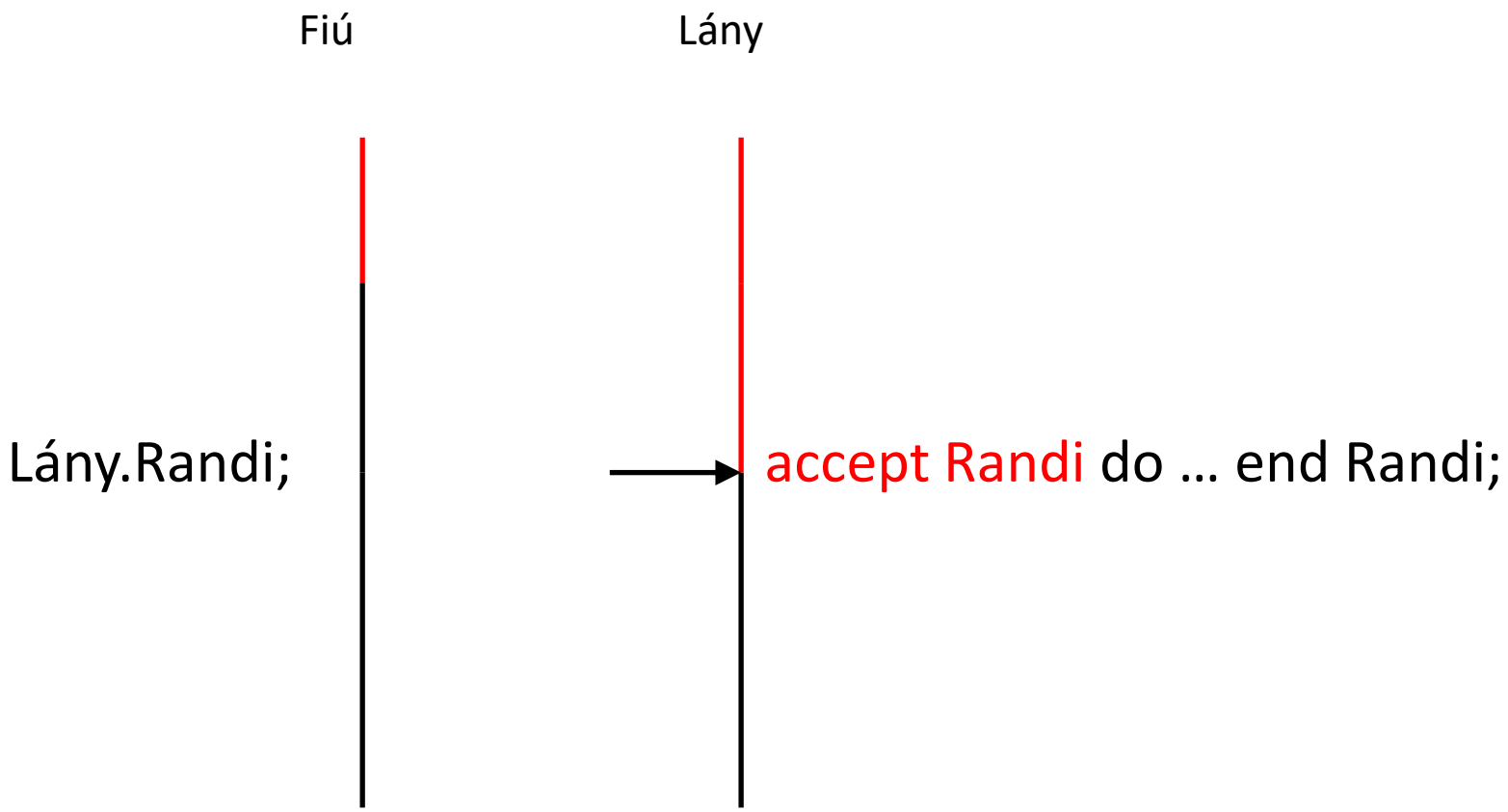
# Ada – Szinkronitás



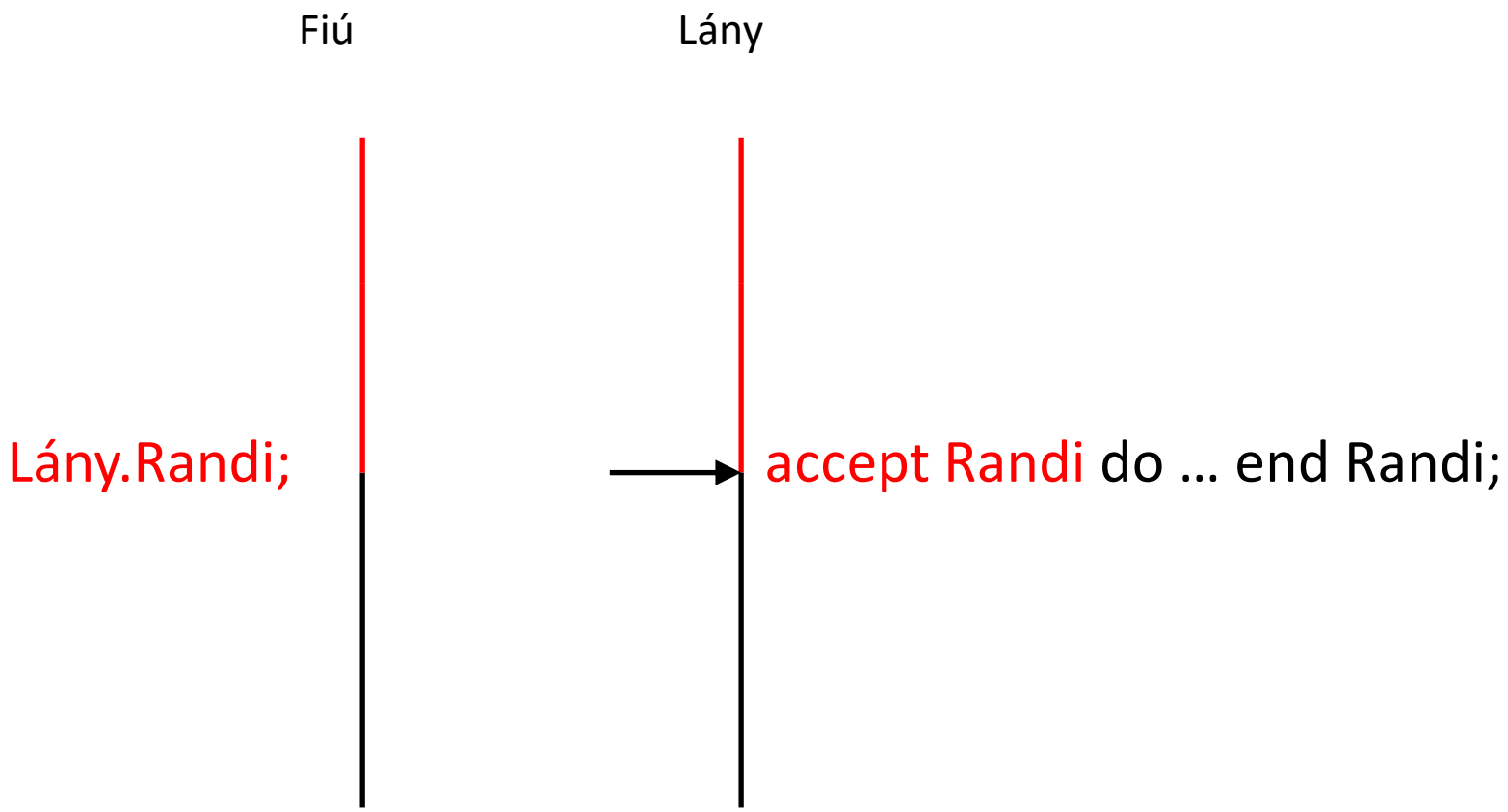
# Ada – Szinkronitás



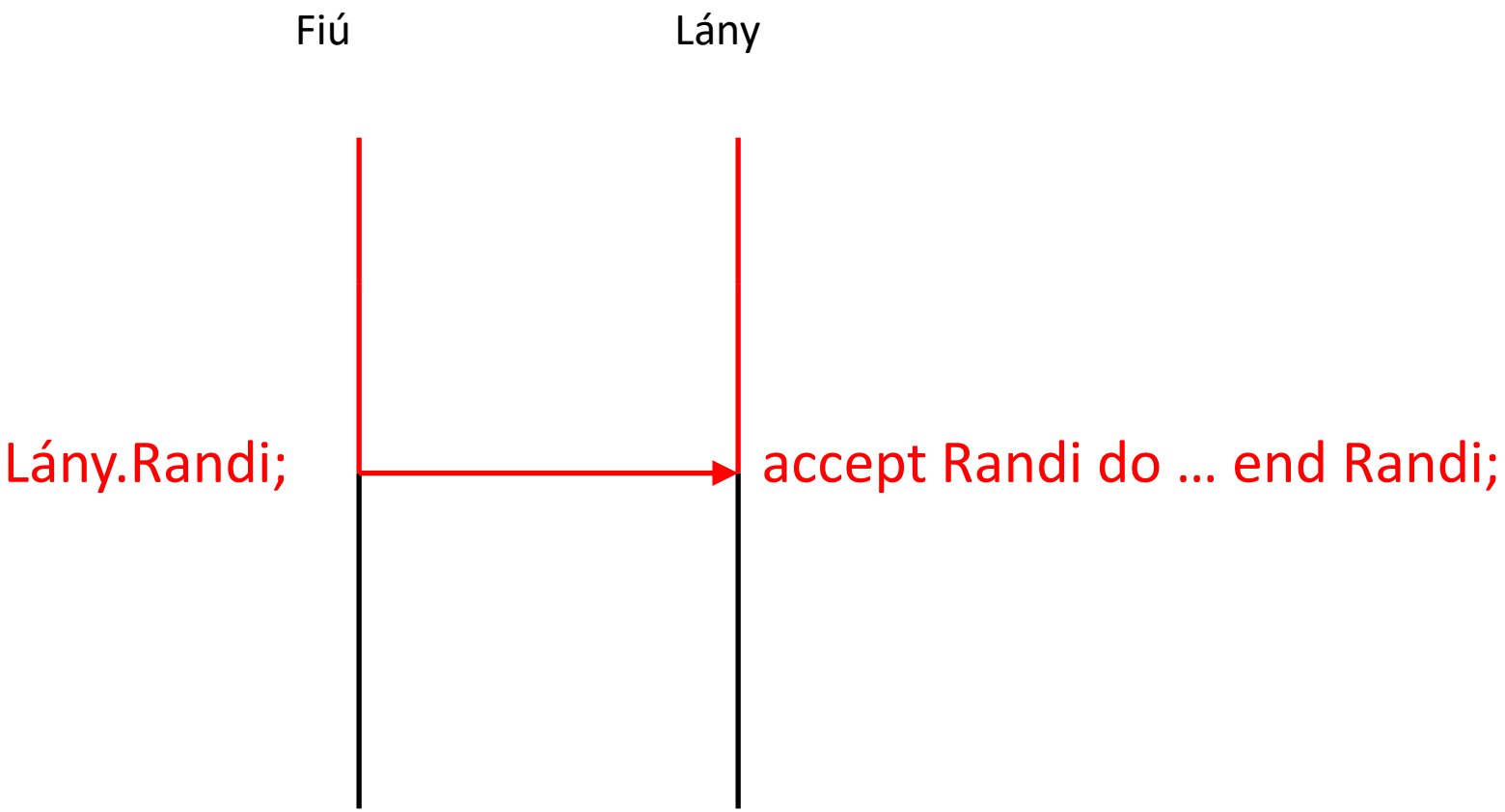
# Ada – Szinkronitás



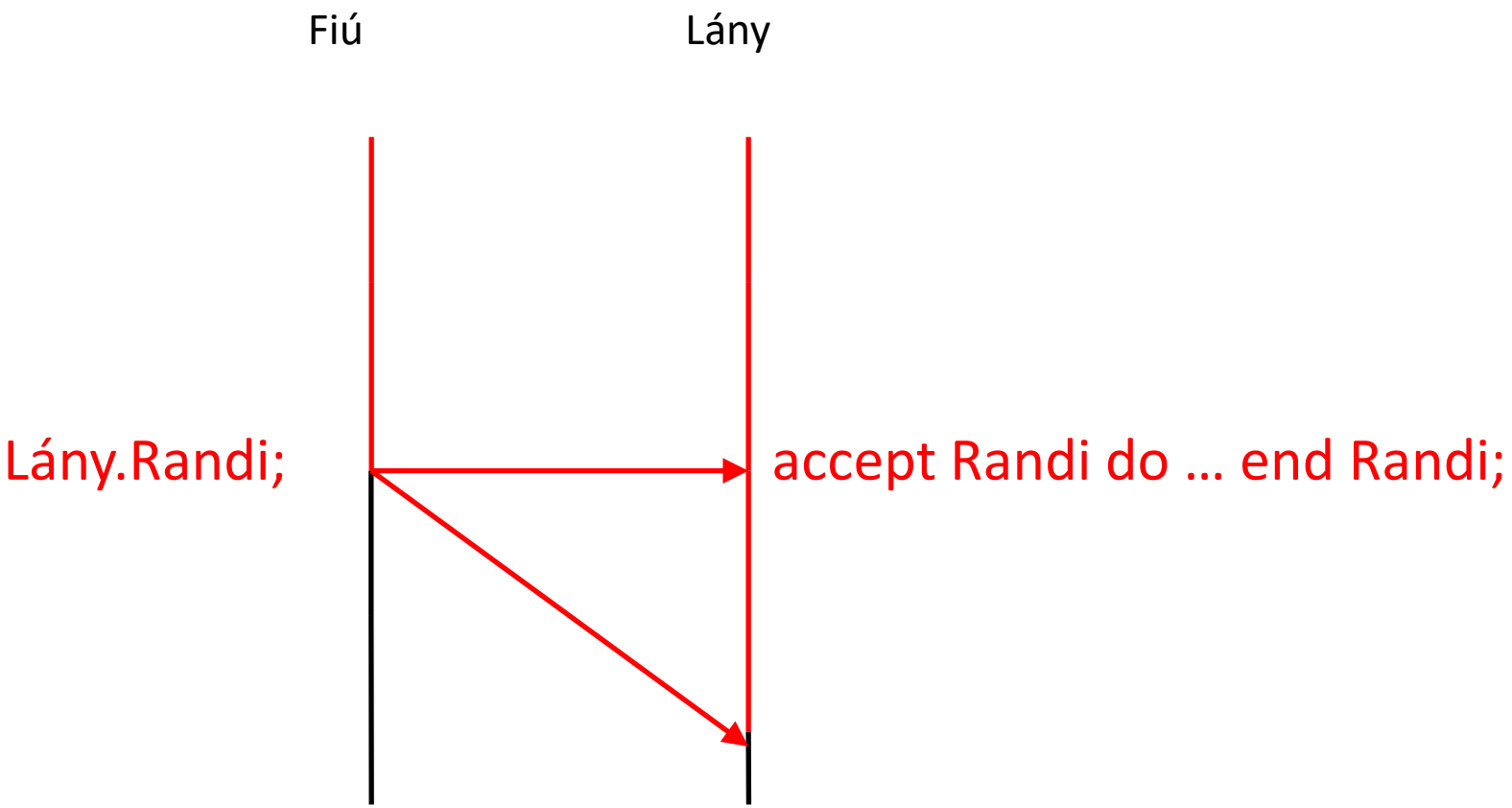
# Ada – Szinkronitás



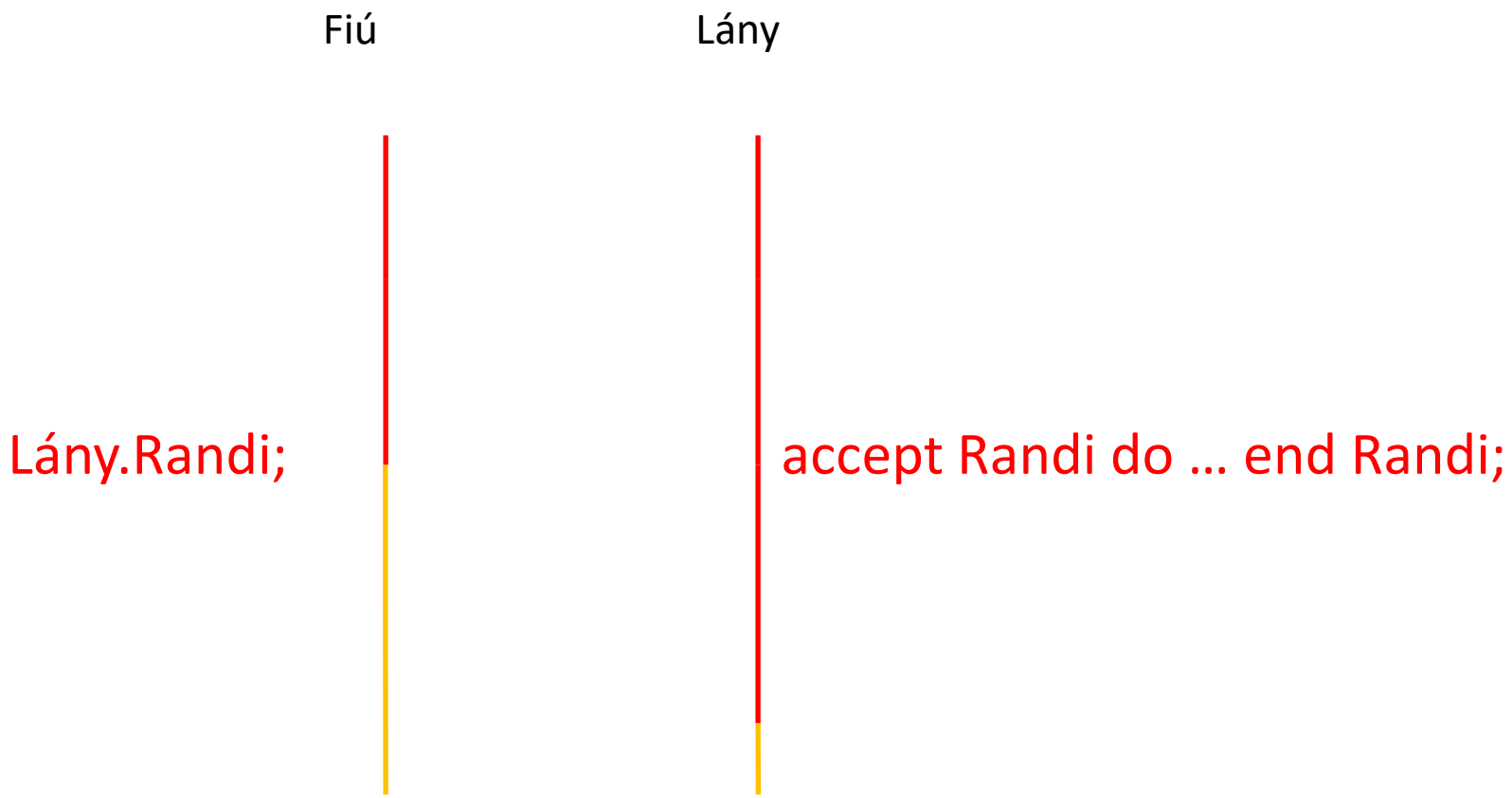
# Ada – Szinkronitás



# Ada – Szinkronitás



# Ada – Szinkronitás



# Ada – Várakozás

Egy időre felfüggeszthető a taszk a delay utasítással:

- `delay 2.4;`

A delay után valós típusú érték van, a másodperc megfelelője

Legalább ennyi ideig nem fut a taszk

Szimulációkban gyakran használjuk

Használjuk szinkronizációs utasításokban is

# Ada

```
procedure P is
          task T;
          task body T is … begin  … end T;
begin
          -- Itt a T is indul
end P;
-- P bevárja T-t
```

# Ada

## Szülő egység

- Az a taszk / alprogram / könyvtári csomag / blokk, amelyben deklaráltuk
- Elindulás: a szülő deklarációs részének kiértékelése után, a szülő első utasítása előtt
- A szülő nem ér véget, amíg a gyerek véget nem ér
  - Függőségi kapcsolatok

## Befejezés

- Bonyolult szabályok
  - Például
    - elfogytak az utasításai (akár kivétel miatt)
    - és a tőle függő taszkok már terminálták
- Fogalmak: komplett, abortált, terminált
- `T'Callable T'Terminated`

# Ada – Több szál

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Haromszalu is
  task Egyik;
  task body Egyik is
  begin
    for i in 1..1000 loop Put_Line("Egyik!"); end loop;
  end Egyik;

  task Masik;
  task body Masik is
  begin
  for i in 1..1000 loop Put_Line("Másik!"); end loop;
  end Masik;

begin
  for i in 1..1000 loop Put_Line("Viszlát!"); end loop;
end Haromszalu;
```

# Ada – Taszk típussal

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Haromszalu is

   task type Udv;
   task body Udv is
   begin
     loop Put_Line("Szia!");end loop;
   end Udv;
   Egyik, Masik: Udv;

begin
   loop Put_Line("Viszlát!"); end loop;
end Haromszalu;
```

# Ada – Taszk típus

Taszkok létrehozásához

- Ugyanazt csinálják, ugyanolyan utasítássorozatot hajtanak végre
- Saját példánnyal rendelkeznek a lokális változókból

Korlátozott (limited) típusok

Cím szerint átadandó, ha alprogram-paraméter

Programegységek

- Mint a taszk programegységek
- Specifikáció és törzs különválik
- Nem lehet könyvtári egység
  (be kell ágyazni más programegységbe)

# Ada – Taszk típus

```
procedure Haromszal is
  task type Szalak(I : Integer); -- a szál azonosítója
  task body Szalak is
  begin
      for j in 1..1000 loop
              put(I); new_line;
      end loop;
  end Szalak;
  X:Szalak(1);
  Y:Szalak(2);
begin
   for i in 1..100 loop
         put_line("foprogram");
   end loop;
end Haromszal;
```

# Ada – Szinkron kommunikáció

Hívott (szolgáltató) részéről

- Meddig várjon?
- Több igényt is kiszolgálhat
- Feltétel lehetősége jó lenne
- Mindörökké fusson?

Hívó részéről

- Meddig várjon?

Igény a programnyelvi támogatásra!

# Ada – Szinkron kommunikáció

A select utasítás

- A hívóban is és a hívottban is szerepelhet – lehetséges magatartásformák támogatása
- A randevúk szervezéséhez segítség
  - Hívott
    - Többágú hívásfogadás
    - Feltételhez kötött hívásfogadás
    - Időhöz kötött hívásfogadás
    - Feltételes hívásfogadás
    - Termináltatás
  - Hívó
    - Időhöz kötött hívás
    - Feltételes hívás

# Ada – többágú hívásfogadás

```
select
     accept E1 do ... end E1;
or
     accept E2;
or
     accept E3( P: Integer ) do ... end E3;
end select;
```

A hívott választ egy hívót valamelyik várakozási sorból

Ha mindegyik üres, vár az első hívóra

- Bármelyik hívást feldolgozza

# Ada – Többműveletes monitor

```
task body Monitor is
   Adat: Tipus; ...
begin
   loop
    select
         accept Muvelet_1( ... ) do ... end;
    or
         accept Muvelet_2( ... ) do ... end;
    ...
    end select;
  end loop;
end;
```

# Ada – Végtelen taszk

Az előző taszk végtelen ciklusban szolgálja ki az igényeket

- Sosem ér véget
- A szülője sem ér véget soha
- Az egész program „végtelen sokáig” fut

Ha már senki sem akarja használni a monitort, akkor leállhat:

- A select egyik ága lehet terminate utasítás
- Ez azt jelenti, amit az előbb mondtunk
  - ha már senki nem akarja használni, akkor terminál jon
- Akkor „választódik ki” a terminate ág, ha már soha többet nem jöhet hívás az alternatívákra

# Ada – terminate

```
task body Monitor is
   Adat: Tipus; ...
begin
   loop
     select
           accept Muvelet_1( ... ) do ... end;
     or
           accept Muvelet_2( ... ) do ... end;
     ...
     or
           terminate;
     end select;
  end loop;
end;
```

# Ada

Feltételhez kötött hívásfogadás

- A select egyes ágaihoz feltétel is rendelhető
- Az az ág csak akkor választódhat ki, ha a feltétel igaz

```
select
     accept E1 do ... end;
or
     when Feltetel => accept E2 do ... end;
...
end select;
```

# Ada – feltételes fogadás

A select utasítás végrehajtása az ágak feltételeinek kiértékelésével kezdődik

- Zárt egy ág, ha a feltétele hamisnak bizonyul ebben a lépésben
- A többi ágat nyitottnak nevezzük
  - A nyitott ágak közül nem-determinisztikusan választunk egy olyat, amihez van várakozó hívó
  - Ha ilyen nincs, akkor várunk arra, hogy valamelyik nyitott ágra beérkezzen egy hívás
  - Ha már nem fog hívás beérkezni, és van terminate ág, akkor lehet terminálni (terminálási szabályok)

A feltételek csak egyszer értékelődnek ki

# Ada – időhöz kötött fogadás

Ha nem vár rám hívó egyik nyitott ágon sem, és nem is érkezik hívó egy megadott időkorláton belül, akkor továbblép.

Tipikus alkalmazási területek

- timeout-ok beépítése (pl. holtpont elkerülésére)
- ha rendszeres időközönként kell csinálni valamit, de két ilyen között figyelni kell a többi taszkra is

# Ada – időhöz kötött fogadás

```
select
     accept E1;
or
     when Feltetel => accept E2 do ... end;
or
     delay 3.0;
end select;

loop
     select
          when Feltetel => accept E do ... end;
     or
          delay 3.0;
     end select;
end loop;
```

# Ada – Feltételes hívásfogadás

Ha most azonnal nem akar velem egy hívó sem randevúzni, akkor nem randevúzom.

- Nem várok hívóra, futok tovább...

Tipikus alkalmazási területek

- valamilyen számítás közben egy jelzés megérkezését ellenőrzöm
- rendszeres időközönként randevúzni vagyok hajlandó, de egyébként valami mást csinálok
- holtpont elkerülése (kiéheztetés veszélye mellett)

# Ada

```
select
     accept E1;
or
     when Feltetel => accept E2 do ... end;
else
     Csinaljunk_Valami_Mast;
end select;
```

Ha minden nyitott ág várakozási sora üres, akkor csináljunk valami mást

# Ada

```
select
     accept E1;
or
     when Feltetel => accept E2 do ... end;
else
     delay 3.0;
end select;
```

Nem ugyanaz, mint az időhöz kötött hívásfogadás!

# Ada – busy waiting

```
loop
      select
            accept E1;
      else
            null;
            -- busy-waiting
      end select;
end loop;
```

Amíg a szükséges helyzet elő nem áll, a folyamat dinamikusan várakozik

- Folyamatosan vizsgálja, hogy a kívánt feltételek teljesülnek-e.
- Hátránya, hogy a folyamat várakozás közben foglalja a processzoridőt.

# Ada – Termelő-fogyasztó

Egy folyamat adatokat állít elő, egy másik adatokat dolgoz fel

Kommunikáció: egy adatokat tartalmazó sor

- Termelő => sor => fogyasztó
- Levelesláda, futószalag; a Tároló általánosítása
- Aszinkron kommunikáció

Korlátos vagy korlátlan sor (buffer)

Általánosítás: több termelő, több fogyasztó

# Ada – Termelő-fogyasztó

```
task type Termelo ( Sor: Sor_Access );
task body Termelo is
  Adat : Tipus;
begin
  loop
    Eloallit( Adat ); Betesz( Sor.all, Adat );
  end loop;
end Termelo;

task type Fogyaszto ( Sor: Sor_Access );
task body Fogyaszto is
  Adat: Tipus;
begin
  loop
    Kivesz( Sor.all, Adat ); Feldolgoz( Adat );
  end loop;
end Fogyaszto;
```

# Ada – Taszkok elrejtése

```
generic
      type Elem is private;
package Osztott_Sorok is
      type Sor(Max_Meret: Positive) is limited private;
      procedure Betesz( S: in out Sor; E: in Elem );
      procedure Kivesz( S: in out Sor; E: out Elem );
      -- üres, tele
private
      task type Sor( Max_Meret: Positive) is
            entry Betesz( E: in Elem );
            entry Kivesz( E: out Elem );
      end Sor;
end Osztott_Sorok;
```

Megvalósítás: monitorral

# Ada – Taszkok elrejtése

```
with Sorok;
package body Osztott_Sorok is
    procedure Betesz( S: in out Sor; E: in Elem ) is
    begin S.Betesz(E); end;
    procedure Kivesz( S: in out Sor; E: out Elem ) is
    begin S.Kivesz(E); end;
    task body Sor is … end Sor;
end Osztott_Sorok;

task body Sor is
     package Elem_Sorok is new Sorok(Elem);
     use Elem_Sorok;
     S: Elem_Sorok.Sor(Max_Meret);
begin
     ……
end Sor;
```

# Ada – Taszkok elrejtése

```
loop
   select
      when Üres(S) => accept Betesz( E: in Elem ) do
                                 Betesz(S,E);
                              end Betesz;
   or
      when Tele(S) => accept Kivesz( E: out Elem ) do
                                 Kivesz(S,E);
                              end Kivesz;
   or
      terminate;
   end select;
end loop;
```

# Ada – védett, korlátos pufferrel

```
protected type Korlatos_Buffer ( Max: Positive ) is
   entry Betesz( X: in Elem );
   entry Kivesz( X: out Elem );
private
   Adatok: Vektor(1..Max);
   Be, Ki: Positive := 1;
end Korlatos_Buffer;

protected body Korlatos_Buffer is
    entry Betesz( X: in Elem ) when Be-Ki < Max is
    begin Adatok(Be) := X;  Be := Be + 1; end Betesz;
    entry Kivesz( X: out Elem ) when Be-Ki > 0 is
    begin
       X := Adatok(Ki); Ki := Ki + 1;
       if Ki > Max then Ki := Ki - Max; Be := Be - Max; end if;
    end Kivesz;
end Korlatos_Buffer;
```

# Algol-68

Az Algol-68-ban a vesszővel elválasztott és begin - end közé írt utasítások párhuzamosan futtathatók.

A begin-end addig nem fejeződik be, amíg minden vessző közötti egység - párhuzamos klóz- véget nem ér.

A szinkronizálásra Dijkstra szemaforokat vezettek be (sema).

A szemaforok párhuzamos klózai elé ki kell tenni a par kulcsszót.

# Algol-68

```
begin
    sema mutex := level 1;
    bool not finished := true;
       # közösen használt puffer deklarációja #
    proc producer = void:
       while not finished
       do
       down mutex
       # pufferbe egy elemet #
       up mutex
       od;
    proc consumer = void:
       while not finished
       do
       down mutex
       # pufferből egy elemet #
       up mutex
       od;
    par (producer, consumer) ;
end;
```

# Delphi

A `TThread` osztályból képzett alosztályok párhuzamosan végrehajthatók.

- Az osztály `Execute()` metódusát kell felüldefiniálni.
  - (virtuális) absztrakt művelet
  - Ez tartalmazza a szál fő kódját.
- Create konstruktor meghívásával hozható létre
  - Egy Boolean paramétere: (CreateSuspended)
    - Ha igaz, akkor felfüggesztve indul, különben azonnal

Szinkronizáció és kommunikáció

- A főosztályban létre kell hozni egy eljárást, amely a vezérlésért felelős
- A synchronize(eljárás_név) segítségével a párhuzamos alosztályok kommunikálását ellenőrizhetjük.

# Delphi

```
type TMyThread = class(TThread) protected
          procedure Execute; override;
end;

thread:=TMyThread.Create(false);
// el is indítjuk egyben
```

Meg kell adni az Execute törzsét …

# Delphi

Szemafor:

```
var MySemaphore: TSemaphore;
    // szemafor változó deklarálása
begin
  MySemaphore := TSemaphore.Create(nil, 5, 5, '');
    // 5 szál számára engedélyezzük az egyidejű használatot
  MySemaphore.Acquire;
    // várakozás egy szemaforra (lefoglalás)
    // itt dolgozhatunk az osztott erőforráson
  MySemaphore.Release;    // szemafor elengedése
  MySemaphore.Free;       // szemafor megsemmisítése
end;
```

# Delphi

A szinkronizációs esemény egy olyan objektum, amelyet egy szál kiválthat

- Ezzel jelezi egy másik szálnak, hogy folytathatja a tevékenységét.

# Delphi

```
type
  TMyThread1 = class(TThread)
    protected procedure Execute; override;
  end;
  TMyThread2 = class(TThread)
    protected procedure Execute; override;
  end;
var MyEvent: TEvent;          // ez lesz az esemény
    Thread1: TMyThread1; Thread2: TMyThread2;
procedure TMyThread1.Execute;
begin
  MyEvent.SetEvent;           // kiváltjuk az eseményt
end;
procedure TMyThread2.Execute;
begin
  MyEvent.WaitFor(INFINITE); // várunk az eseményre
end;
```

# Java

A Java nyelvben a folyamatok (szálak) leírására a Thread osztályt használják

- Az osztály egy objektuma reprezentál egy folyamatot.
- A folyamat törzse a run metódusban leírt kód.
- Runnable interfész megvalósítása kell, ha nem tud örökölni a Threadtől!

Egy programban tetszőleges számú szálat indíthatunk.

- Ezeknek megadhatjuk a prioritását, felfüggeszthetjük, újraindíthatjuk, megállíthatjuk.
- A szálakat szinkronizálhatjuk a join segítségével.
- A kommunikáció osztott változókon keresztül történik
- A kölcsönös kizárás biztosítására a synchronized kulcsszó szolgál.
- Ha egy blokk vagy metódus synchronized akkor a hozzá csatolt monitor biztosítja a kölcsönös kizárást.

# Java

```
class MyThread extends Thread {
 public void run() {
  System.out.println(getName() + " Thread" );
 }
}
public class ExtendedThread {
 static public void main(String args[]) {
    MyThread a, b;
            a = new MyThread();
            b = new MyThread();
     a.start();
     b.start();
  }
}
```

# Java

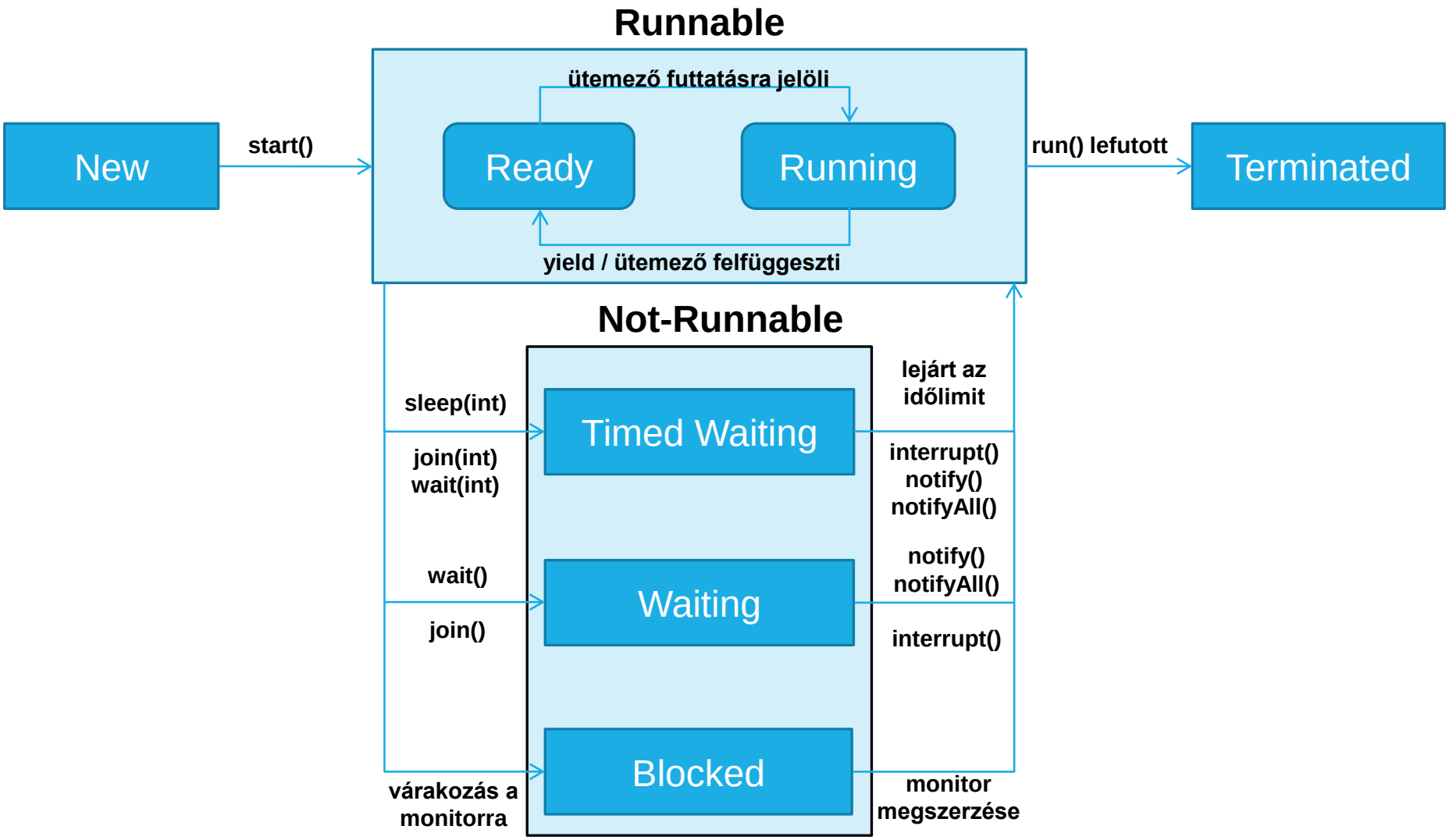
```
class MyThread implements Runnable {
  public void run() {
   System.out.println(Thread.currentThread().getName);
  }
 }
public class RunnableThread {
 static public void main(String s[]) {
   MyThread work2do;
   Thread a, b;
   work2do = new MyThread();
   a = new Thread(work2do);
   b = new Thread(work2do);
   a.start();
   b.start();
   }
 }
```

# Java

Szálak állapotai:

- Futó: a VM éppen ezt a szálat futtatja
- Kész (ready): futtatható, éppen áll, hogy egy másik szál futhasson
- Blokkolt: vár egy külső esemény bekövetkezésére (pl. I/O)
- Megszakított: a folyamatoknak jeleket küldhetünk, melyekre megállnak
- Halott: befejeződött a folyamat

# Szálak állapotai



# Java

## Szálcsoportok is létrehozhatók

- A ThreadGroup segítségével csoportszinten is kezelhetők a szálak

## Futásvezérlés

- Felfüggesztés – újraindítás
  - suspend() - resume()
    - Nem ajánlják – holtpontveszély
      - pl. egy szál zárol egy erőforrást, és felfüggesztik …
    - Javaslat: segédváltozó és a wait() – notify()

# Java

- Megszakítás
  - interrupt()
- Leállítás
  - volt: stop() – nem biztonságos
  - most: volatile Thread referencia, ami a futó szálra mutat, ha a szálat leállítják, ez null lesz, amit a run() vizsgál
- Szálak összekapcsolása
  - join()
  - Szinkronizálásra, megvárja, amíg a másik befejeződik
    - vagy egy adott ideig vár)
- Prioritások kezelése
  - 10 szint, állítható

# Java

## Versengés

- Több szál szimultán szeretne ugyanahhoz az erőforráshoz hozzájutni
  - synchronized blokk
    - objektum- és osztályszinten
  - Problémák
    - író/olvasó esetben egyszerre csak egy írhat és olvashat, bár elvben több is olvashatna

# Java – synchronized

Metódusok elé írhatjuk (de pl. interfészekben nem!)

Kölcsönös kizárás arra a metódusra

Kulcs (lock) + várakozási sor

- A kulcs azé az objektumé, amelyiké a metódus
- Ugyanaz a kulcs az összes szinkronizált metódusához

1. Mielőtt egy szál beléphetne egy szinkronizált metódusba, meg kell szereznie a kulcsot
2. Vár rá a várakozási sorban
3. Kilépéskor visszaadja a kulcsot

# Java – synchronized

A synchronized kulcsszó védhet blokk utasítást is

- Ilyenkor meg kell adni, hogy melyik objektum kulcsán szinkronizáljon
  - `synchronized(obj){...}`

Sokszor úgy használjuk, hogy a monitor szemléletet megtörjük

- Nem az adat műveleteire biztosítjuk a kölcsönös kizárást, hanem az adathoz hozzáférni igyekvő kódba tesszük…
- Létjogosultság: ha nem egy objektumban vannak azok az adatok, amelyekhez szerializált hozzáférést akarunk garantálni

# Java – wait-notify

Szignálokra hasonlít

Egy feltétel teljesüléséig blokkolhatja magát egy szál

- A feltétel (potenciális) bekövetkezését jelezheti egy másik szál

Alapfeladat

- termelő - fogyasztó
  (korlátos) bufferen keresztül kommunikálnak
  - egy termelő, egy fogyasztó
  - több termelő, több fogyasztó

# Java – wait-notify

Minden objektumhoz tartozik a sima kulcshoz tartozó várakozási soron kívül egy másik wait-várakozási sor

- A `wait()` hatására a szál bekerül ebbe
- A `notify()` hatására az egyik várakozó kikerül belőle
- A `wait()` és `notify()` hívások csak olyan kódrészben szerepelhetnek, amelyek ugyanazon az objektumon szinkronizáltak
- `synchronized(obj){ ... obj.wait(); ... }`

# Java – wait-notify

A szál megszerzi az objektum kulcsát, ehhez, ha kell, sorban áll egy ideig (**`synchronized`**)

- A `wait()` hatására elengedi a kulcsot, és bekerül a wait-várakozási sorba
- Egy másik szál megkaparinthatja a kulcsot (kezdődik a **`synchronized`**)
- A `notify()` vagy `notifyAll()` metódussal felébreszthet egy vagy az összes `wait`-es alvót, aki bekerül a kulcsos várakozási sorba
- A **`synchronized`** végén elengedi a kulcsot
- A felébredt alvónak (is) lehetősége van megszerezni a kulcsot és továbbmenni
- A **`synchronized`** blokkja végén ő is elengedi a kulcsot…

# C#

A párhuzamos végrehajtás során az alábbi lépéseket kell megtenni

- Definiálunk egy függvényt, aminek nincsenek paraméterei és a visszatérési típusa `void`.
  - Ez több rendszerben kötelezően `run` névre hallgat.
- Ennek a függvénynek a segítségével egy függvénytípust, delegáltat definiálunk.
  - Könyvtári szolgáltatásként a `ThreadStart` ilyen, használhatjuk ezt is.
- Az így kapott delegáltat felhasználva készítünk egy `Thread` objektumot.
- Meghívjuk az így kapott objektum Start függvényét.

A párhuzamos végrehajtás támogatását biztosító könyvtári osztályok a `System.Threading` névtérben találhatók.

# C#

```
using System;
using System.Threading;
class program {
  public static void szal() {
    Console.WriteLine("Ez a szálprogram!");
  }
  public static void Main() {
    Console.WriteLine("A főprogram itt indul!");
    ThreadStart szalmutato=new ThreadStart(program.szal);
    // létrehoztuk a függvénymutatót, ami a szal()
    //  függvényre mutat
    Thread fonal=new Thread(szalmutato);
    // létrehoztuk a párhuzamos végrehajtást végző objektumot
    //  paraméterül kapta azt a delegáltat (függvényt) amit
    //   majd végre kell hajtani
    fonal.Start();
    // elindítottuk a fonalat, valójában a szal() függvényt
    //  a fonal végrehajtása akkor fejeződik be amikor a
    //  szal függvényhívás befejeződik
    Console.WriteLine("Program vége!");
  }
}
```

# C#

## Szálak vezérlése

- `Sleep (millisec)`
  - statikus függvény, az aktuális szál végrehajtása várakozik a paraméterben megadott ideig.
- `Join()`
  - az aktuális szál befejezését várjuk meg
- `Interrupt()`
  - az aktuális szál megszakítása.
  - A szál objektum `interrupt` hívása `ThreadInterruptedException` eseményt okoz a szál végrehajtásában.

# C#

## Szálak vezérlése

- `Abort()`
  - Az aktuális szál befejezése, valójában a szálban egy AbortThreadException kivétel dobását okozza
  - Ezzel befejeződik a szál végrehajtása.
  - Ha egy szálat abortáltunk, nem tudjuk a Start függvénnyel újraindítani
    - `ThreadStateException` kivételt dob.
    - Ezt a kivételt elkaphatjuk, sőt visszavonható az abortálás a `Thread.ResetAbort()` függvényhívással
  - IsAlive: tulajdonság, megmondja, hogy a szál élő-e
- `Suspend()`
  - A szál végrehajtásának felfüggesztése
- `Resume()`
  - A felfüggesztés befejezése, a szál továbbindul

# C#

```
class adatok {
      public void mentes(string s) {
            Console.WriteLine("Adatmentés elindul!");
            for(int i=0;i<50;i++) {
                Thread.Sleep(1);
                Console.Write(s);
                }
            Console.WriteLine("");
            Console.WriteLine("Adatmentés befejeződött!");
      }
}
```

# C#

```
class program {
  public static adatok a=new adatok();

  public static void szal1() {
    Console.WriteLine("Ez a szál1 program indulás!");
    Console.WriteLine("Adatmentés meghívása szál1-ből!");
    a.mentes("+");
    Console.WriteLine("Szál1 vége!");
  }

  public static void szal2() {
    Console.WriteLine("Ez a szál2 programindulás!");
    Console.WriteLine("Adatmentés meghívása szal2-ből!");
    a.mentes("-");
    Console.WriteLine("Szál2 vége!");
  }
```

# C#

```
public static void Main() {
  Console.WriteLine("A főprogram itt indul!");
  ThreadStart szalmutato1 = new ThreadStart(program.szal1);
  ThreadStart szalmutato2 = new ThreadStart(program.szal2);
  Thread fonal1=new Thread(szalmutato1);
  Thread fonal2=new Thread(szalmutato2);
  fonal1.Start();
  fonal2.Start();
  Console.WriteLine("Program vége!");
 }
}
```

Felváltva ír ki +-t és - -t a képernyőre!!

# C#

## Erőforráskezelés

- A .NET keretrendszer számos osztályt és adattípust kínál számunkra, hogy a közös erőforrásokat kezelhessük.
- A CLR (Common Language Runtime) háromféle elérési módot biztosít globális változók, metódusok, osztályszintű metódusok, objektumok és blokkok számára:
  - Szinkronizált kódterület
    - Szinkronizálható bármely osztályszintű és példányszintű metódus, vagy annak egy része Monitor használatával.
    - Statikus adattagokat nem lehet szinkrozinálni
  - Klasszikus kézi szinkronizáció
    - Számos szinkronizációs osztály használható arra, hogy összhangot teremtsünk a saját elvárásainknak megfelelően.
  - Szinkronizált környezet
    - A `SynchronizationAttribute` használata egyszerű, automatikus szinkronizációt valósít meg `ContextBoundObject` objektumokon.
    - Minden objektum közös környezetben osztozik a záron.

# C#

A `Monitor` osztály használatával elérhetjük, hogy egy kódrészletet egy adott időpontban csak egy szál használhasson.

- A `Monitor` osztály minden metódusa statikus, így használatához nem kell példányosítani az osztályt.
- A monitor indítása a `Monitor.Enter(object)` vagy a `Monitor.TryEnter(object)` metódushívással történhet.
- Feloldása a `Monitor.Exit(object)` vagy a `Monitor.Wait(object)` hívással történhet.
- Ha a monitor feloldása megtörtént, akkor a `Monitor.Pulse` vagy a `Monitor.PulseAll` hívás üzenetet küld a hozzáférési sorban álló szálaknak, hogy szabad az elérés.

# C#

## Monitor

- Amikor egy szál a lefoglalt kódrészletében `Wait` -et hív
  - Megszakad a hozzáférése, és bekerül egy várakozó sorba
  - A sor első elemét reprezentáló szál meghívja a `Pulse` vagy `PulseAll` metódust és lép egyet a hozzáférési sorban.
- Az Enter metódus atomi, így ha két szál egyszerre hívja ugyanarra a kódrészletre, akkor is csak az egyik kapja meg a jogot a futtatásra.
- Külső objektum zára deadlock-hoz vezethet, ezért érdemes belső objektumon használni

## Szinkronizáció:

1. A `Synchronization()` attribútummal kell jelölni az osztályt.
2. Az osztályt a `ContextBoundObject` osztályból kell származtatni.

# C# – Példák

```
[Syncronization()]
class szinkronosztály: ContextBoundObject {
  int i=5;                    // egy időben csak egy szál fér hozzá
  public void Novel() {
    // ezt a függvényt csak egy szál tudja egyszerre végrehajtani
    i++;
  }
}

public void mentes(string s) {
  Monitor.Enter(this);
  Console.WriteLine("Adatmentés elindul!");
  for(int i=0;i<50;i++) {
    Thread.Sleep(1); Console.Write(s);
  }
  Console.WriteLine("");
  Console.WriteLine("Adatmentés befejeződött!");
  Monitor.Exit(this);
}
```

# C# – További lehetőségek

Számos további osztály

- `WaitHandle`
- `Mutex, AutoResetEvent, ManualResetEvent`
- `InterLocked`
- `ThreadPool`
- `Timer`
- `Backgroundworker`
- …

# C++

`std::thread` szabványos könyvtár

- `thread` osztály
- Szálkezelő függvények
- Kölcsönös kizárást biztosító osztályok:
  - `timed_mutex`
  - `recursive_mutex`
  - `recursive_timed_mutex`
  - `shared_timed_mutex`

# C++ példa

```
void f1()
{
    cout << "Fuggvenydemo 1.\n";
}
void f2(int x)
{
    cout << "Fuggvenydemo 2. x=" << x << "\n";
}
```

# C++ példa

```
thread elso(f1);
 // uj szalat indit, ami az egyik-et hivja
thread masodik(f2, 0);
 // uj szalat indit, ami a masik(0)-t hivja
cout << "Harom szal fut parhuzamosan\n";

// szalak szinkronizalasa:
elso.join();    // var, amig elso befejezodik
masodik.join(); // var, amig masodik befejezodik

cout << "elso es masodik lefutott.\n";
```

# C++ példa

```
mutex mtx;     // mutex a kritikus szakaszra

void f2_mutex(int x) {
     mtx.lock();
     cout << "Fuggvenydemo 2. x=" << x << "\n";
     mtx.unlock();
}
```

Az előző indítás helyett:

- `thread masodik(f2_mutex, 0);`

# C++ példa

```
#include <iostream> // std::cout
#include <thread>   // std::thread
#include <mutex>    // std::mutex
std::mutex mtx;     // mutex a kritikus szakaszhoz

void print_block (int n, char c) {
  mtx.lock();
  for (int i=0; i<n; ++i) { std::cout << c; }
  std::cout << '\n';
  mtx.unlock();
}
int main () {
  std::thread th1 (print_block, 50,'*');
  std::thread th2 (print_block, 50,'$');
  th1.join(); th2.join();
  return 0;
}
```

# Go



## Google nyelve

- 2007 – ben kezdték el fejleszteni
- Jelenlegi verzió: 1.14.2 (2020. május)

## Tulajdonságok

- Fordított, konkurens, imperatív programozási nyelv
- C-szerű szintaxis
- Statikus (erősen) típusos nyelv
- Szemétgyűjtés

# Go – gorutinok

Osztott memória helyett csatornákon való kommunikáció

goroutine:

- Konkurens folyamat
- Közös címterületen, saját veremmel, ami dinamikusan nő
- Saját szemétgyűjtő
- A tényleges szálak kezelését elrejti a programozó elől
- A go kulcsszóval hívott függvény új goroutine-ban indul

# Go – példa

```
package main;
import (
       "fmt"
       "time"
)
func IsReady(what string, minutes int64) {
    time.Sleep(time.Duration(minutes));
    fmt.Println(what, "is ready")
}
func main() {
    go IsReady("tea", 4);
    go IsReady("coffee", 2);
       // nem var a befejezesre, ha o
       // befejezte, akkor kesz, ezert kell ide is egy Sleep...
    time.Sleep(time.Duration(6));
    fmt.Println("I'm ready...");
}
```

# Go

A Go a párhuzamosságot osztott memória helyett adatcsatornákon történő üzenetküldésekkel támogatja.

- `chan // kétirányú csatorna`
- `<-chan // fogadó csatorna`
- `chan <- // küldő csatorna`

Az elemek típusa tetszőleges

Az inicializálatlan csatorna értéke nil

Új csatornát a make() függvénnyel hozhatunk létre

- `make(típus, kapacitás)`
- `make(chan int, 100)`

Csatorna típusnak kell lennie

Ha a buffer méret 0, akkor blokkol, amíg a fogadó goroutine nem áll készen fogadni, egyébként aszinkron

# PVM – Parallel Virtual Machine

Szoftver rendszer, mellyel hálózatba kapcsolt számítógépeket egyetlen nagy párhuzamos számítógépként lehet látni és kezelni.

- Kezdet: 1989 - Oak Ridge National Laboratory
- A párhuzamos programok írásának egyik szabványává vált.
- A publikus változata ingyen elérhető.
- Sok hardver gyártó biztosítja a saját gépére optimalizált, gyorsabb változatát is.

# PVM

PVM rendszerbe kapcsolhatunk több számítógépet

- Akár különböző típusúakat is

Tulajdonságok

- A rendszer a rajta futó programok szempontjából ezek után egy nagy, elosztott memóriájú virtuális számítógépnek látszik.
- Különböző processzorokon, különböző programokat indíthatunk el.
- A szinkronizációt és a kommunikációt a PVM könyvtári függvényeivel oldhatjuk meg.

# PVM

## Működése

- Számítógépek hálózatba szervezve
- A felhasználó jogosult bármelyik gépre bejelentkezni
- Minden gépen futtat egy pvm „démon”-t
- És futtatja a taszkokat, melyek a démonokon keresztül lépnek egymással kapcsolatba
  - Egy gépen több taszk is futhat

# PVM

Egy nagy virtuális címtér

- PVM
- Partitioned Global Address Space (PGAS) nyelvek és rendszerek

Üzenetküldés, elosztott memória

- Communicating Sequential Processes (CSP)
- Egymás memóriáját nem érik el, csak üzenetekkel kommunikálhatnak
- Message Passing Interface standard – legelterjedtebb
- Több millió folyamat (process) kezelésére alkalmas

# Helyesség

KÖVETKEZŐ ALKALOMMAL




# Programozási nyelvek és módszerek

## PROGRAMOK HELYESSÉGE

# Programok helyessége

Feladat

- Algoritmust adok a megoldására
- Megoldja?
  - Teszteljük
    - Fekete doboz
    - Fehér doboz
  - Nem biztos a megoldás

Formális matematikai megközelítés szükséges!

# Sir Charles Antony Richard Hoare

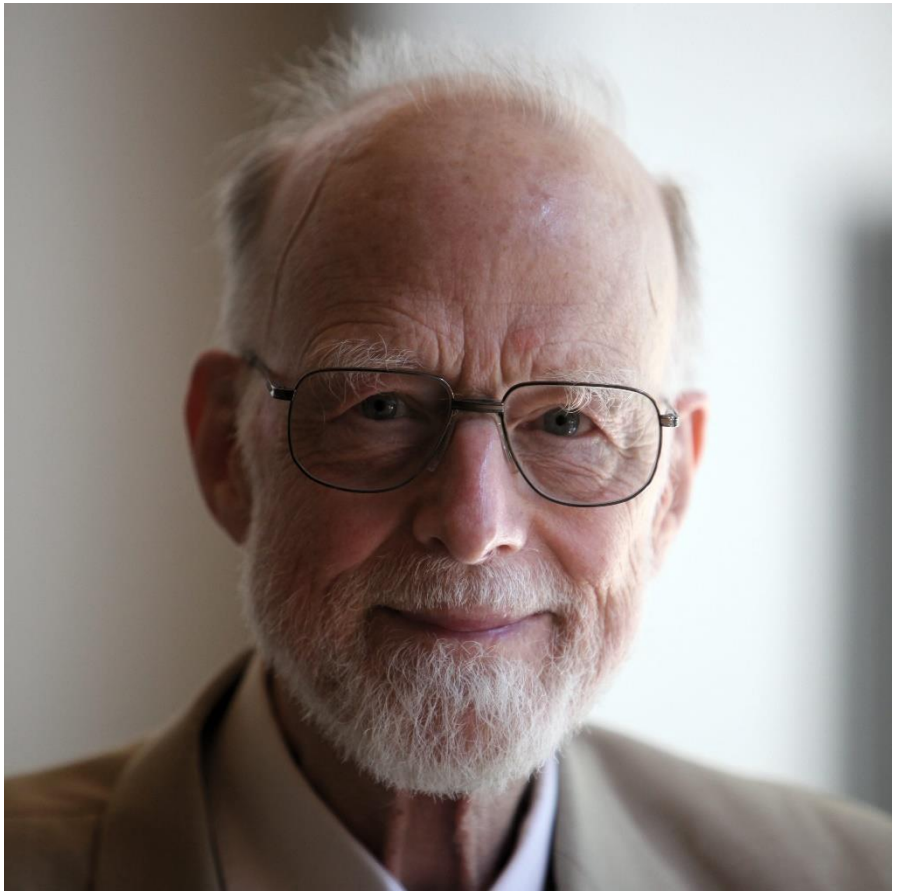
Brit tudós

Fontosabb eredmények

- Quicksort algoritmus
- „Hoare logic”
- A programhelyesség igazolására
- Strukturált programozás
- Párhuzamos programozás
- Null referencia
  - Ezért 2009-ben bocsánatot is kért

Hol

- Oxford
- Microsoft Research Lab. (Cambridge)



# Feladat specifikációja

Szerződés a felhasználó és az implementáló között

- előfeltétel: egy állítás, ami leírja azt a feltételt, ami szükséges a feladat helyes működéséhez
- utófeltétel: egy állítás, ami leírja azt a feltételt, amit a függvény teljesít a helyes végrehajtás után

Helyesség a specifikáció figyelembevételével:

- ha a függvény felhasználója teljesíti az előfeltételt, a függvény elkezd futni, s amikor befejezi, akkor az utófeltétel igaz lesz.

(Mit kell az implementációnak teljesíteni, ha a felhasználó megsérti az előfeltételt?)

# Feladat specifikációja

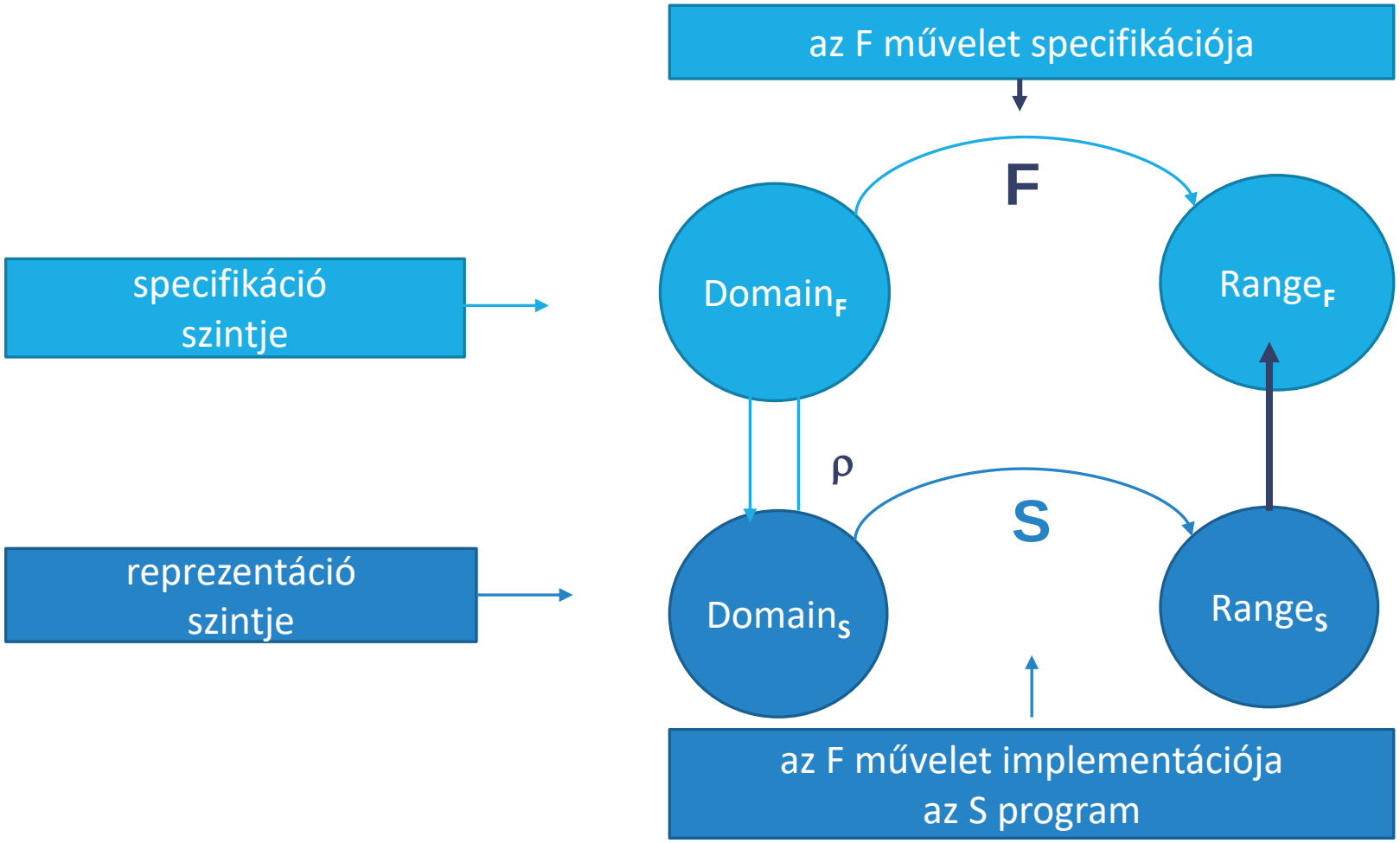
A strukturált és az objektumorientált programozási nyelvek központi fogalmává vált az absztrakció és az absztrakt adattípus.

- Ez a megközelítés azt jelenti, hogy a típust a típusspecifikációval, a típusimplementációval és a közöttük levő kapcsolatot megadó reprezentációs függvénnyel definiáljuk.
- A helyességbizonyításhoz tehát olyan specifikációs módszerekre van szükségünk, amelyekkel a típusspecifikációt és a típusimplementációt adhatjuk meg.
  - Ezt nevezzük kettős specifikációnak.
  - Ebben az esetben a helyességbizonyítás során azt kell ellenőrizni, hogy az implementáció megfelel-e a specifikációnak.

A  Hoare-féle specifikációban:

- A típusműveletekhez elő- és utófeltételeket rendelünk.
- A típusértékhalmazok leírása  invariánsokat is tartalmazhat

# A típus specifikáció és a típus kapcsolata



# Lehetőségek

Az elő- és utófeltételes specifikációt alkalmazza számos programozási nyelv

- Közvetlenül pl.
  - az Alphard,
  - az Eiffel,
  - a D,
  - az Oxygene (korábban Chrome) (Object Pascal alapú)
  - a Cobra
  - …
- Közvetve (valamilyen eszköz segítségével)
  - C#, VB a Code Contracts használatával (ez egy Microsoft Research project, a .Net Framework -be integrálva),
  - Java számos eszközzel, pl. iContract2, Contract4J, jContractor, Jcontract, Java Modeling Language (JML),
  - …

# Hoare - módszer

A feladat:

- adjuk össze egy valós elemeket tartalmazó vektor elemeit!

Kiinduló adatok:

- egy n hosszú, valós elemeket tartalmazó v vektor

Eredmény:

- s valós érték, ami tartalmazza a vektor elemeinek összegét!

Írjunk rá megoldó algoritmust!

# Hoare - módszer

```
float sum(float *v, int n) {
  float s = 0.0;
  int i = 0;
  while (i < n) {
    s = s + v[i];
    i = i + 1;
  }
  return s;
}
```

Hogyan indokoljuk, hogy ez a program helyes (megoldja  a feladatot)?

# Feladat specifikációja

Szerződés a felhasználó és az implementáló között

- előfeltétel: egy állítás, ami leírja azt a feltételt, ami szükséges a feladat helyes működéséhez
- utófeltétel: egy állítás, ami leírja azt a feltételt, amit a függvény teljesít a helyes végrehajtás után

Helyesség a specifikáció figyelembevételével:

- ha a függvény felhasználója teljesíti az előfeltételt, a függvény elkezd futni, s amikor befejezi, akkor az utófeltétel igaz lesz.

(Mit kell az implementációnak teljesíteni, ha a felhasználó megsérti az előfeltételt?)

# Feladat specifikációja

Állítás: egy logikai függvény a program állapotával kapcsolatosan

Például:

- $x=3$
- $y > x$
- $(x \neq 0) \Rightarrow (y+z = w)$
- $s = \Sigma_{(i \in 1..n)}\ v[i]$
- $\forall i \in 1..n : v[i] > v[i-1]$
- $\text{true}$

# Feladat specifikációja

Állapottér

- Jelölje $\mathcal{V}$ az $n$ hosszú, valós elemekből álló vektorok típusát! ($0$-tól $n-1$-ig indexelve)
- $\mathcal{V} \times \mathcal{R}$
- Változók: $v \quad s$
- Előfeltétel: $v = v' \text{ és } n > 0$
- Utófeltétel: $s = (\Sigma i \mid 0 \leq i < n : v[i])$

# Hoare – hármas

A programhelyesség formális indoklásához, elő- és utófeltételek használatával

Szintaxis: {Q} S {R}

- Q és R állítások
- S program

Jelentése: Ha kiinduláskor igaz a Q és végrehajtjuk S-t, akkor S egy olyan állapotban terminál, ahol R igaz.

# Példák Hoare – hármasokra

$\{ \text{true} \}\ x := 5\ \{ x=5 \}$

$\{ x = y \}\ x := x + 3\ \{ x = y + 3 \}$

$\{ x > 1 \}\ x := x * 2\ \{ x > 2 \}$

$\{ x=a \}$ if $(x < 0)$ then $x := -x\ \{ x=|a| \}$

$\{ \text{false} \}\ x := 3\ \{ x = 8 \}$

# Legerősebb utófeltétel

Néhány érvényes Hoare-hármas:

- $\{x = 5\}\ x := x * 2\ \{\ \text{true}\ \}$
- $\{x = 5\}\ x := x * 2\ \{\ x > 0\ \}$
- $\{x = 5\}\ x := x * 2\ \{\ x = 10 \mid\mid x = 5\ \}$
- $\{x = 5\}\ x := x * 2\ \{\ x = 10\ \}$

Mind igaz, de az utolsó a leghasznosabb:

- x=10  a legerősebb utófeltétel

# Legerősebb utófeltétel

Definíció: Ha $\{Q\}\ S\ \{R\}$ és minden olyan $R'$-re, amire $\{Q\}\ S\ \{R'\}$, $R \Rightarrow R'$, akkor $R$ az $S$ $Q$-ra vonatkozó legerősebb utófeltétele

- ellenőrizzük: x = 10 ⇒ true
- ellenőrizzük : x = 10 ⇒ x > 0
- ellenőrizzük : x = 10 ⇒ x = 10 || x = 5
- ellenőrizzük : x = 10 ⇒ x = 10

# Leggyengébb előfeltétel

Néhány érvényes Hoare-hármas:

- $\{x = 5 \;\&\&\; y = 10\}\; z := x / y \;\{ z < 1 \}$
- $\{x < y \;\&\&\; y > 0\}\;\; z := x / y \;\{ z < 1 \}$
- $\{y \neq 0 \;\&\&\; x / y < 1\}\; z := x / y \;\{ z < 1 \}$

Mind igaz, de az utolsó a leghasznosabb, mert ez az, ami a legáltalánosabb feltételekkel engedi hívni a programot:

- $y \neq 0 \;\&\&\; x / y < 1$ a leggyengébb előfeltétel

Definíció: Ha $\{Q\}\; S \;\{R\}$ és $\forall$ Q' –re, ahol $Q' \Rightarrow Q$ igaz, hogy $\{Q'\}\; S \;\{R\}$, akkor Q az S R-re vonatkozó leggyengébb előfeltétele (weakest precondition).

- Jelölje ezt *wp*(S,R) .

# A wp általános tulajdonságai

1. A csoda kizárásának törvénye
   wp(S,false) = false
2. Monotonitási tulajdonság
   ha P ⇒ R akkor wp(S,P) ⇒ wp(S,R)
3. wp(S,P and R) = wp(S,P) and wp(S,R)
4. wp(S,P) or wp(S,R) ⇒ wp(S,P or R)

(Bizonyítás az érdeklődőknek az irodalomban.)

# Levezetési szabályok

Értékadás

- $\{ Q \}\ x := 3\ \{ x+y > 0 \}$
- Mi a Q leggyengébb előfeltétel?
- Hogy kapjuk meg?
- Mi az a legáltalánosabb értéke az y-nak, amire $3 + y > 0$ ?
- $y > -3$

# Levezetési szabályok

Értékadás

- $\{ Q \}\ x := 3*y + z\ \{ x * y - z > 0 \}$
- Mi a Q leggyengébb előfeltétel?
- Hogy kapjuk meg?

- Hogy csináltuk az előbb?

# Levezetési szabályok

Értékadás

- $\{ Q \}\, x := 3 \,\{ x+y > 0 \}$
- Mi a Q leggyengébb előfeltétel?
- Értékadási szabály:

$$wp(x := E, V) = V^{x \leftarrow E}$$

$$\{V^{x \leftarrow E}\}\, x := E \,\{ V \}$$

$$(x + y > 0)^{\,x \leftarrow 3}$$
$$= (3) + y > 0$$
$$= y > -3$$

# Levezetési szabályok

Értékadás

- $\{ Q \}\ x := 3*y + z\ \{ x * y - z > 0 \}$
- Mi a Q leggyengébb előfeltétel?
- Értékadási szabály:

$$wp(x := E, V) = V^{x \leftarrow E}$$

$$\{V^{x \leftarrow E}\}\ x := E\ \{ V \}$$

$$(x * y - z > 0)\ x \leftarrow 3*y+z$$
$$= (3*y+z) * y - z > 0$$
$$= 3*y^2 + z*y - z > 0$$

# Levezetési szabályok

Szekvencia

- $\{ Q \}\ x := x + 1; y := x + y \{ y > 5 \}$

- Mi a Q leggyengébb előfeltétel?
- Hogy kapjuk meg?

# Levezetési szabályok

Szekvencia

- $\{ Q \}\ x := x + 1; y := x + y \{ y > 5 \}$
- Mi a Q leggyengébb előfeltétel?
- Szekvencia szabálya:

$$wp(S;T, R) = wp(S, wp(T, R))$$

$$wp(x{:=}x{+}1; y{:=}x{+}y, y{>}5)$$
$$= wp(x{:=}x{+}1, wp(y{:=}x{+}y, y{>}5))$$
$$= wp(x{:=}x{+}1, x{+}y{>}5)$$
$$= x{+}1{+}y{>}5$$
$$= x{+}y{>}4$$

# Levezetési szabályok

## Elágazás

- $\{ Q \}$ if $x > 0$ then $y := x$ else $y := -x$ $\{ y > 5 \}$
- Mi a Q leggyengébb előfeltétel?
- Intuitív megközelítés

ha a feltétel igaz: $\{Q1\}\ y := x\ \{ y > 5\}$

$$Q1 = x>5$$

else ág: $\{Q2\}\ y := -x\ \{ y > 5\}$

$$Q2 = -x > 5$$

$$Q2 = x < -5$$

$$Q = x > 5 \ || \ x < -5$$

# Levezetési szabályok

## Elágazás

- $\{ Q \}$ if $x > 0$ then $y := x$ else $y := -x$ $\{ y > 5 \}$
- Mi a Q leggyengébb előfeltétel?
- Elágazás szabálya:

$wp(\text{if } B \text{ then } S \text{ else } T, R)$
$= B \Rightarrow wp(S,R) \;\&\&\; \neg B \Rightarrow wp(T,R)$

$wp(\text{if } x>0 \text{ then } y:=x \text{ else } y:=-x, y>5)$
$= x>0 \Rightarrow wp(y:=x,y>5) \;\&\&\; x \leq 0 \Rightarrow wp(y:=-x,y>5)$
$= x>0 \Rightarrow x>5 \;\&\&\; x \leq 0 \Rightarrow -x > 5$
$= x>0 \Rightarrow x>5 \;\&\&\; x \leq 0 \Rightarrow x < -5$
$= x > 5 \;||\; x < -5$

# Levezetési szabályok

Ciklusok

- { Q } while (felt) törzs { R }
- Mi a Q leggyengébb előfeltétel?

# Ciklushelyesség bizonyítása

Tekintsük először  a parciális helyességet

- Nem biztos, hogy terminál, de ha igen, az utófeltétel igaz lesz
  - {Q} while B do S {R}

Keressünk egy Inv invariánst, amire

- $Q \Rightarrow Inv$
  - az invariáns kezdetben igaz
- $\{ Inv \;\&\&\; B \}\; S \;\{Inv\}$
  - a ciklusmag minden végrehajtása  megőrzi az invariánst
- $(Inv \;\&\&\; \neg\, B) \Rightarrow R$
  - az invariánsból és a ciklus kilépési feltételéből következik az utófeltétel

# Ciklus példa

Bizonyítsuk, hogy  a vektor összegzés helyes:

$\{ n \geq 0 \}$

$j := 0;$

$s := 0;$

$\text{while } (j < n) \text{ do}$

$\quad s := s + v[j];$

$\quad j := j + 1;$

$\text{end}$

$\{ s = (\Sigma i \mid 0 \leq i < n : v[i]) \}$

# Ciklus példa

Bizonyítsuk, hogy  a vektor összegzés helyes:

$\{ n \geq 0 \}$

$j := 0;$

$s := 0;$

$\{ Inv \}$

$\text{while } (j < n) \text{ do}$

$\{ Inv \;\&\&\; j < n \}$

$\quad s := s + v[j];$

$\quad j := j + 1;$

$\{ Inv \}$

$\text{end}$

$\{ s = (\Sigma i \mid 0 \leq i < n : v[i]) \}$

# Ciklus invariáns keresése

Általában az utófeltétel valamilyen  gyengítése

- $s = (\Sigma i \mid 0 \leq i < n : v[i])$

Függ valamilyen módon a ciklusváltozótól

- Tudjuk, hogy $j$ kezdetben $0$, és inkrementáljuk, amíg el nem éri $n$-t
  - Így  $0 \leq j \leq n$ valószínűleg az invariáns része

Ciklus kilépési feltétel && invariáns ⇒ utófeltétel

- Ciklus kilépési feltétel : $j = n$

Jó ötlet: helyettesítsük az utófeltételben $n$-t $j$-vel:

- $(\Sigma i \mid 0 \leq i < j : v[i])$

Invariáns

- $0 \leq j \leq n \;\&\&\; s = (\Sigma i \mid 0 \leq i < j : v[i])$

# Ciklus példa

Bizonyítsuk, hogy  a vektor összegzés helyes:

$\{ n \geq 0 \}$

$j := 0;$

$s := 0;$

$\{ 0 \leq j \leq n \;\&\&\; s = (\Sigma i \mid 0 \leq i < j : v[i]) \}$

while $(j < n)$ do

$\{0 \leq j \leq n \;\&\&\; s = (\Sigma i \mid 0 \leq i < j : a[i]) \;\&\&\; j < n\}$

$s := s + v[j];$

$j := j + 1;$

$\{0 \leq j \leq n \;\&\&\; s = (\Sigma i \mid 0 \leq i < j : v[i]) \}$

end

$\{ s = (\Sigma i \mid 0 \leq i < n : v[i]) \}$

# Mit kell belátni?

1. $Q \Rightarrow Inv$
   - az invariáns kezdetben igaz
2. $\{ Inv \;\&\&\; B \}\; S \;\{Inv\}$
   - a ciklusmag minden végrehajtása  megőrzi az invariánst
3. $(Inv \;\&\&\; \neg\, B) \Rightarrow R$
   - az invariánsból és a ciklus kilépési feltételéből következik az utófeltétel

# Alphard

Az Alphardot a ’70-es évek második felében fejlesztették ki.

- A cél a Hoare-féle helyességbizonyításhoz egy megfelelő specifikációs eszköz kidolgozása volt, ezért sokan a tökéletes megoldást látták benne.
- Végül azonban az implementálásig soha nem jutott el.

Mégis számos helyen oktatják a típusspecifikációs módszerek között, mert a specifikációs lehetőségei és a helyességbizonyításra való alkalmassága a legtöbb módszerhez képest sokkal kifinomultabbak.

# Alphard

Az Alphard a típusokat formoknak nevezi. Egy típust a következő módon kell megadni:

- ```
  form típusnév (formális paraméterek) =
  beginform
  specifications
  ...
  representation
  ...
  implementation
  ...
  endform;
  ```
  ```

# Alphard

```
form istack (n: integer) =
beginform
  specifications
    requires n > 0;
    let istack=<...,xi,...> where xi is integer;
       -- egészekből álló sorozat
    invariant 0 <= length (istack) <= n;
    initially istack = nullseq;
       -- az absztrakt objektum kezdeti tulajdonságai
```

# Alphard

```
function push (s: istack, x: integer)
                pre 0 <= length (s) < n
                post s = s' ~ x,  -- konkatenáció
          pop (s: istack)
                pre 0 < length (s) <= n
                post s = leader (s'),
          top (s: istack) returns x: integer
                pre 0 < length (s) <= n
                post x = last (s),
          isempty (s: istack) returns b: boolean
                post b = (s = nullseq);
representation
          ...
implementation
          …
endform;
```

# Alphard

A `requires` a formális paraméterekre vonatkozó megszorításokat tartalmazza.

A `let` kulcsszó után a típus leírására használt absztrakt adattípust adjuk meg (jelen esetben az integereket tartalmazó sorozatot).

Az `invariant` a specifikáció szerinti típusinvariánst írja le.

- A példányosítás majd a konkrét térben fog megtörténni.

A reprezentációs függvény, amelyet a `representation`-ben kell megadni, fog visszaképezni erre az absztrakt adattípusra.

# Alphard

Minden konkrét térben inicializált objektumra alkalmazzuk a reprezentációs függvényt és az így kapott „absztrakt objektumoknak” ki kell elégíteniük az `initially`ban megadott feltételt.

Az `initially` az absztrakt kezdeti objektum tulajdonságait írja le.

A `function` záradékban az absztrakt adattípus műveleteit specifikáljuk elő-utófeltételekkel.

# Alphard

```
form istack (n: integer) =
beginform
    specifications
    ...
    representation
    unique  v: vector (integer, 1, n)
            sp: integer init sp <- 0;
    rep (v, sp) = seq (v, 1, sp);
    invariant 0 <= sp <= n;
    states
          empty when sp = 0,
          normal when 0 < sp < n,
          full when sp = n,
          error otherwise;
```

# Alphard

Implementation

```
body push out (s.sp=s.sp’+1 and s.v=alpha(s.v’,s.sp,x)) =
    empty, normal:: (s.sp<-s.sp+1; s.v[s.sp] <- x);
    otherwise:: FAIL;
body pop out (s.sp = s.sp’ – 1) =
    normal, full:: s.sp <- s.sp – 1;
    otherwise:: FAIL;
body top out (x = s.v [s.sp]) =
    normal, full:: x <- s.v [s.sp];
    otherwise:: FAIL;
body isempty out (b = (sp = 0)) =
    normal, full:: b <- false;
    empty:: b <- true;
    otherwise:: FAIL;
endform;
```

# Alphard

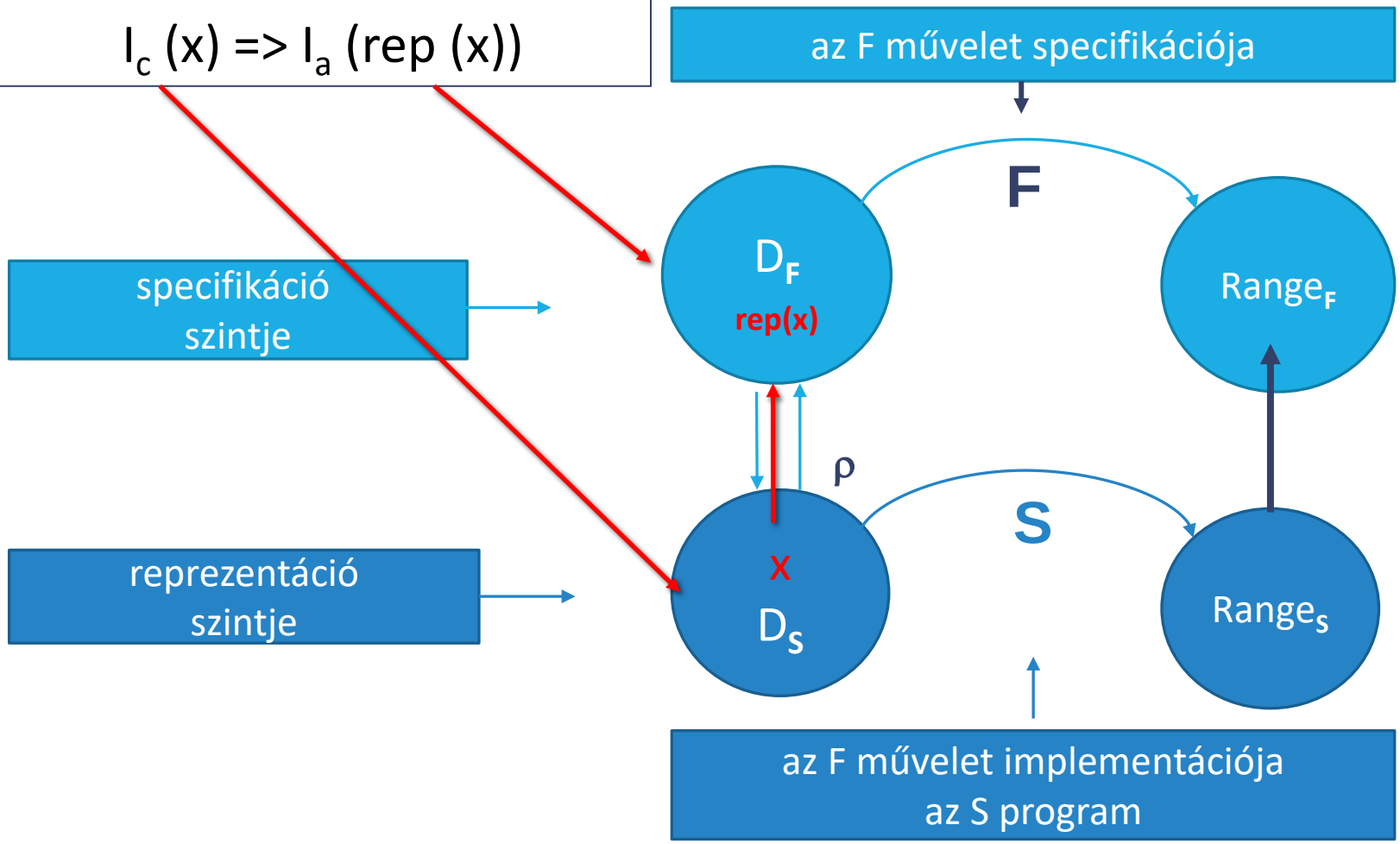
Hogyan láthatjuk be az így specifikált adattípus (form) helyességét?

Jelöljük

- a specifications záradék invariánsát $I_a$-val,
- a representation záradékbeli invariánst $I_c$-vel.
- a specifications záradékban levő requirest $\beta_{req}$-val
- az initiallyt pedig $\beta_{init}$-tel.
- az inicializáló műveletet $f_{init}$ –tel,
- a többit műveletet f-fel
- egy művelet végrehajtását { } jelekkel
- az f absztrakt műveletekhez tartozó elő- és utófeltételeket $pre_a^f$ –vel és $post_a^f$ –vel
- az f konkrét művelethez elő- és utófeltételeit $pre_c^f$ –vel és $post_c^f$ –vel

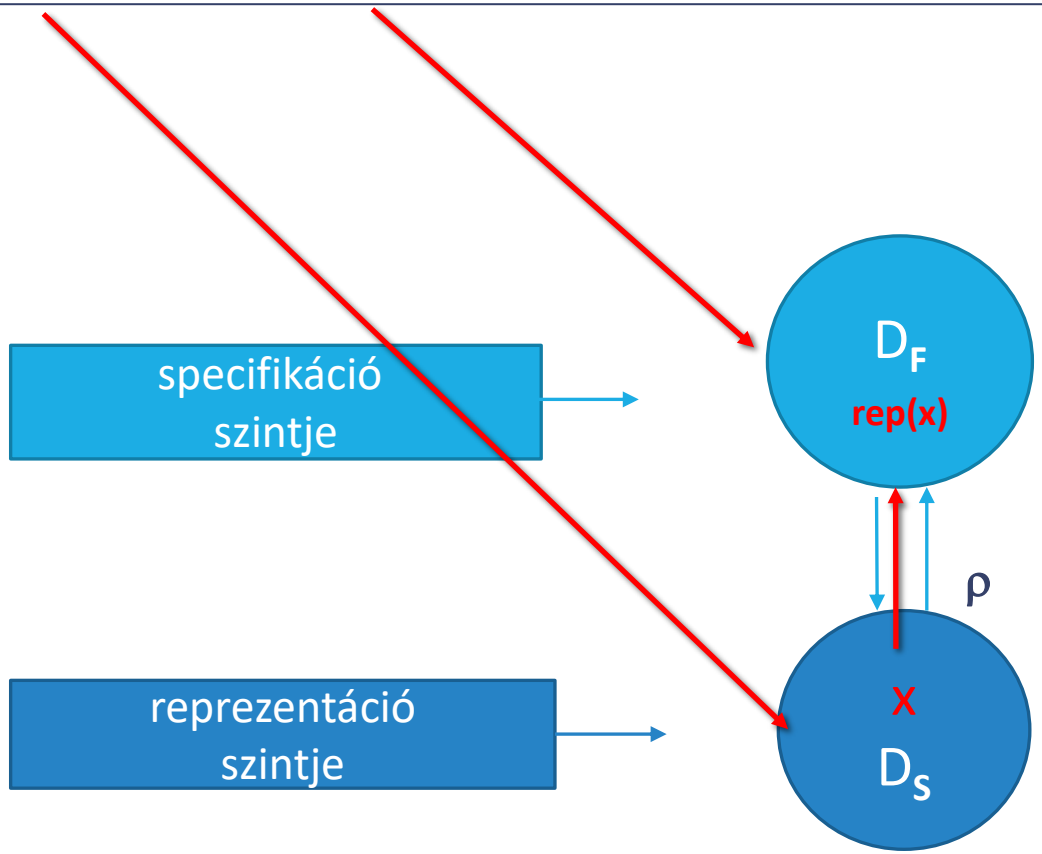
1. A reprezentáció helyességének ellenőrzése:

$$I_c(x) \Rightarrow I_a(rep(x))$$



2. Az objektum inicializálásának ellenőrzése:

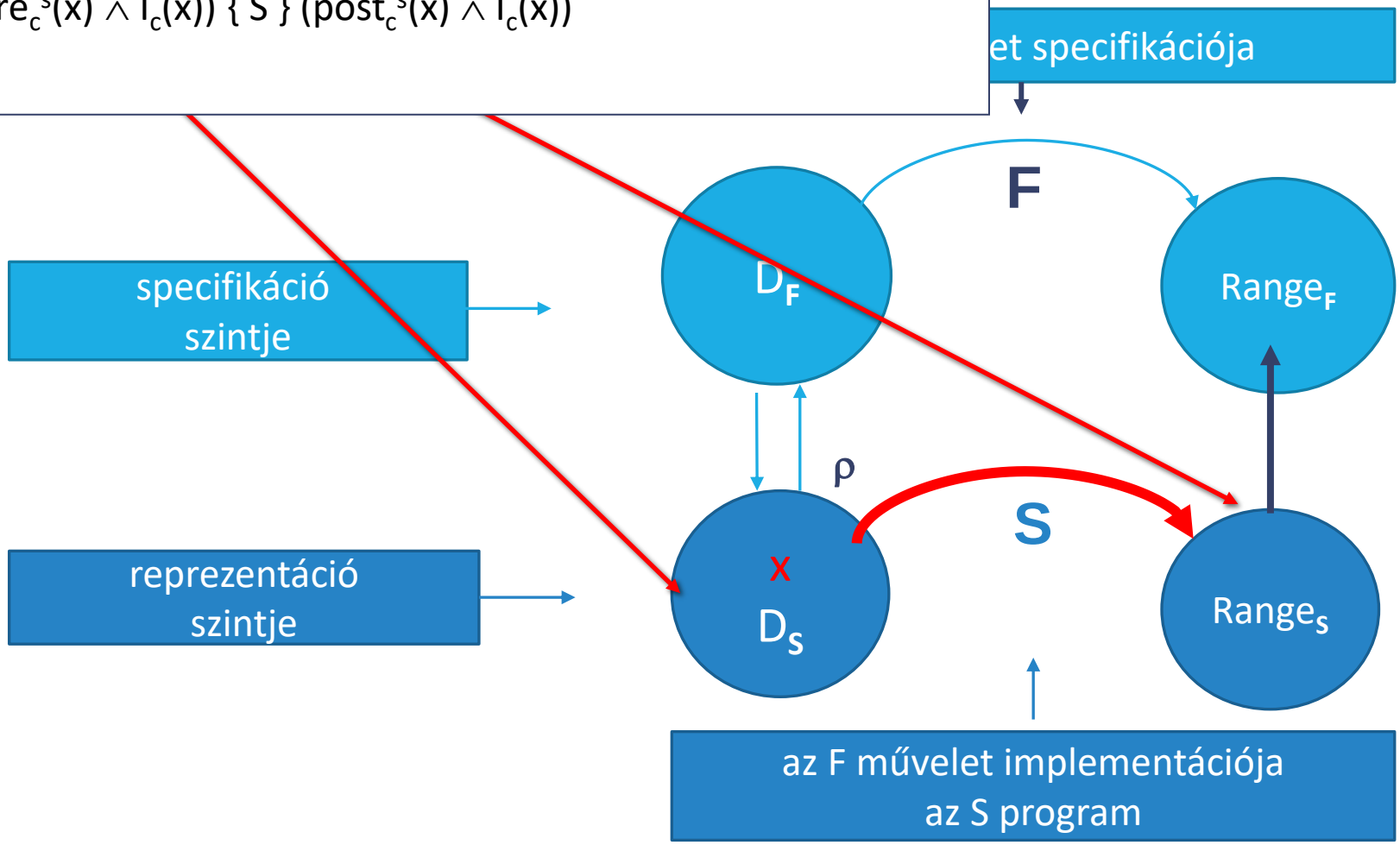
$\beta_{req}\ \{f_{init}\}\ (\beta_{init}\ (rep\ (x)) \wedge I_c\ (x))$



3. **Minden** f műveletre:

a. A konkrét művelet (S) helyességének ellenőrzése:

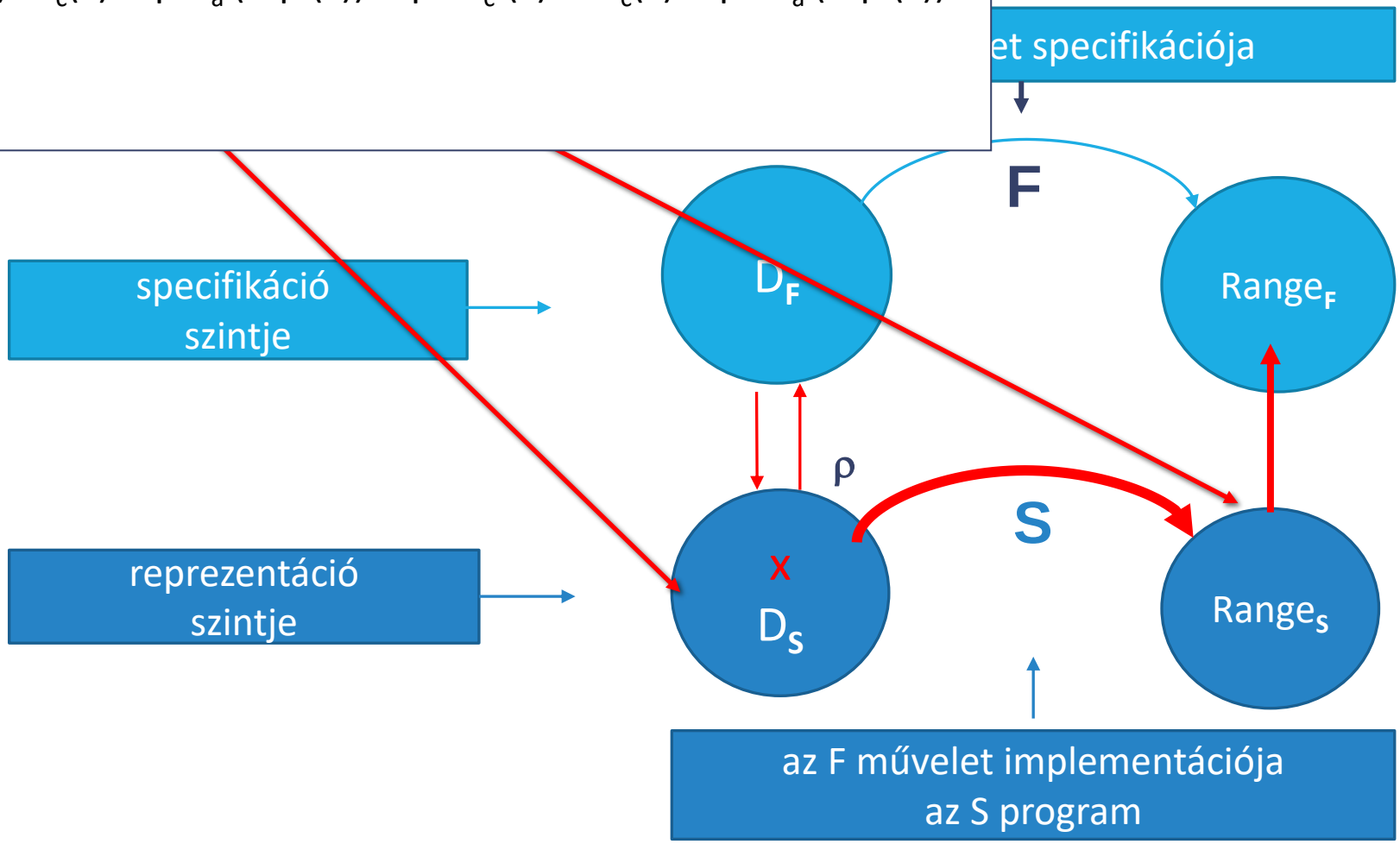
$(pre_c^S(x) \wedge I_c(x))\ \{ S \}\ (post_c^S(x) \wedge I_c(x))$



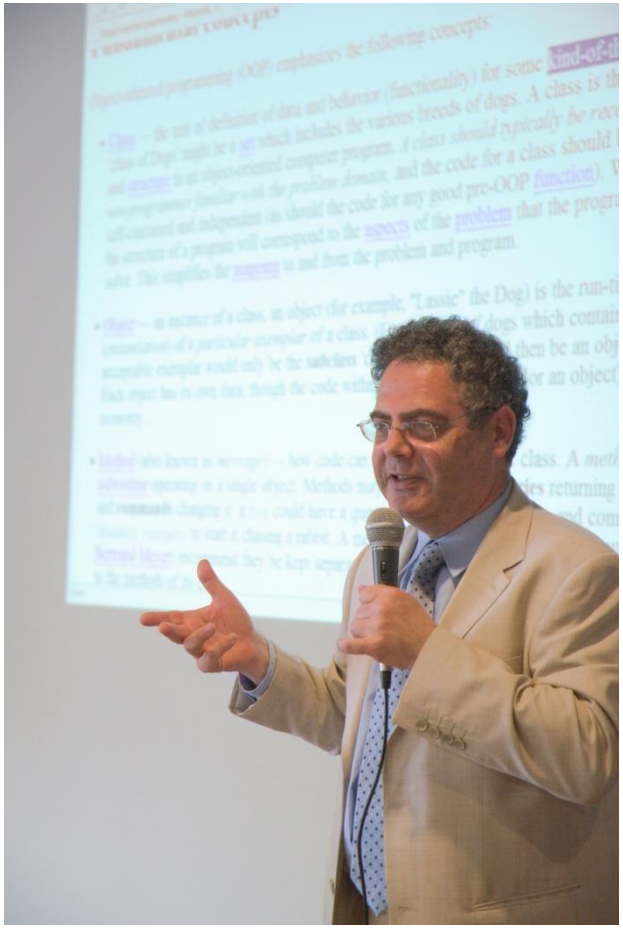
b. A konkrét és az absztrakt specifikáció közötti kapcsolat ellenőrzése:

(1) $I_c(x) \wedge pre_a^f(rep(x)) \Rightarrow pre_c^s(x)$

(2) $I_c(x) \wedge pre_a^f(rep(x)) \wedge post_c^s(x) \Rightarrow I_c(x) \wedge post_a^f(rep(x))$



# Bertrand Meyer (1950- )



francia tudós

Munkássága

- „Design by Contract”
- „Object-Oriented Software Construction”
- Eiffel
- ETH Zürich

# „Design by Contract”

Programozás szerződéssel

- Bertrand Meyer „Object-Oriented Software Construction”
- 1986 óta

Jogok és kötelezettségek

- Az előfeltétel a megrendelő kötelezettsége
- Az utófeltétel a teljesítést vállalóé

# Eiffel

Az Eiffel beépített nyelvi eszközökkel rendelkezik formális specifikációk megadására

- Az Eiffel futtató rendszer ellenőrizni is tudja, hogy az egyes programegységek nem sértik-e meg a specifikációt.

A típusok műveleteinek elő-utófeltételes specifikációja fontos szerepet játszik az Eiffel módszertanában:

- felhasználják a program tervezése során a követelmények pontosabb megfogalmazására, de teszteléskor és dokumentációs célokra is.

A nyelv tervezésekor ugyanakkor nem volt cél, hogy a fordítóprogram bizonyítsa a program helyességét, mert ez a gyakorlatban nem oldható meg.

# Elő- utófeltételek

```
forth is
-- Move forward one position.
require
 not_after: not after
 do
   ... Some appropriate implementation
ensure
 position = old position + 1
 end – forth
 put_i_th (v: like first; i: INTEGER) is
  -- Put item v at i-th position.
require
 index_large enough: i >= 1;
 index_small_enough: i <= count;
 deferred
ensure
 not empty
end – put_i_th
```

# Osztályinvariáns

```
deferred class CHAIN feature
…
invariant
 -- Definitions:
 empty = (count = 0);
 off = ((position = 0) or (position = count + 1 ));
 isfirst = (position = 1 );
 islast = (not empty and (position = count));
 -- Axioms:
 count >= 0;
 position >= 0; position <= count + 1;
 empty => (position = 0);
 (not off) => (item = i_th (position));
 -- Theorems:
 (isfirst or islast) implies not empty
end -- class CHAIN
```

# Az osztály konzisztens

Az osztály minden p konstruktor eljárására:

- $\{\mathrm{pre}_p\}\ \mathrm{do}_p\ \{\mathrm{INV}_C\}$

Az osztály minden r metódusára, amit általában, vagy szelektíven exportál:

- $\{\mathrm{pre}_r \wedge \mathrm{INV}_C\}\ \mathrm{do}_r\ \{\mathrm{post}_r \wedge \mathrm{INV}_C\}$

# Megadás

Kvantorok megadására nincs lehetőség, de használhatjuk az `old` operátort:

- Az `old` operátorral egy objektum végrehajtandó metódusába való belépése előtti állapotát jelöljük;
  Utófeltételben: `x = old x + 1`

# Ciklushelyesség:

```
from –INIT
   go_before
invariant -- INV
  0 <= child_position; child_position <= arity + 1
variant   -- VAR
 arity - child_position + 1
 until    -- EXIT
 child after or else (j = i)
 loop     -- BODY
    child_forth;
 if (sought = child) then
   j:=j+1
 end
end
```

Ciklusinvariáns és terminátor függvény

# Ciklushelyesség

A C osztály r metódusa akkor és csak akkor ciklushelyes, ha minden ciklusra:

1. {True} INIT {INV}
2. {True } INIT {VAR>=0 }
3. {INV and then not EXIT} BODY { INV }
4. {INV and then not EXIT and then (VAR=v)} BODY { 0<=VAR<v }

# Exception helyesség

Egy C osztály egy r rutinja exception-helyes akkor és csak akkor, ha a rescue blokkjának minden ‘b’ ágára a következő igaz:

- Ha b egy retry-jal végződik:
  - $\{ \text{true} \}\ b\ \{ \text{INV}_C \text{ and } \text{pre}_r \}$
- Ha b nem retry-jal végződik:
  - $\{ \text{true} \}\ b\ \{ \text{INV}_C \}$

# Öröklődés és helyesség

Az öröklődés során az elő-utófeltételek újradeklarálhatóak, amelyre az Eiffel a `require else` ... alternatív előfeltételt és az `ensure then` ... extra utófeltételt adja.

- Legyenek $pre_1$, ....$pre_n$ az ősök előfeltételei, $post_1$,...$post_n$ az ősök utófeltételei
- Minegyike `or else` kapcsolatban lesznek az előfeltételnél és `and then` kapcsolatban az utófeltételnél
- (az esetek többségében n=1).
- Ekkor a metódus előfeltétele:
  - alternatív előfeltétel `or else` $pre_1$ `or else ... or else` $pre_n$ utófeltétele:
  - alternatív utófeltétel `and then` $post_1$ `and then ... and then` $post_n$

# Újradefiniálás

A DbC szerint az újradefiniálálás

- alszerződést kötök a leszármazottal, amire a kliensek az ősével már kötöttek.
- Egy becsületes alvállalkozó legalább olyan jól teljesíti a követelményeket, mint az eredeti.
- Ez azt jelenti:
  - megtartja, vagy gyengíti az előfeltételt, vagyis nincs új követelménye a kliens felé,
  - megtartja vagy szűkíti az utófeltételt, vagyis az eredmény legalább olyan jó, mint az eredeti szerződésben volt.

# Ellenőrzés

A program tetszőleges pontján elhelyezhetünk egy állítást, amelyet a futtató rendszer ellenőriz.

- Erre a check utasítás szolgál.

A környezet számos opciója segíti, hogy melyik állítást értékeljük ki. A lehetséges szintek:

- no - nem ellenőrzi az állításokat
- require - mindig az előfeltételeket ellenőrzi (alapértelmezett)
- ensure - előfeltételek és utófeltételek ellenőrzése
- invariant - az előzőeken túl az osztályinvariáns is
- loop - az előzőeken túl a ciklusok ellenőrzése
- all - az előzőeken túl a check utasítások ellenőrzése

Az egyes szintek jelentősége főleg a tesztelésnél van.

# D nyelv

Assert a legegyszerűbb feltétel

- Ha nem teljesül, AssertException-t dob

```
int i;
Symbol s = null;
for (i = 0; i < 100 ; i++)
{
      s = search(array[i]);
      if (s != null) break;
}
assert (s != null); //megtaláltuk-e a keresett elemet
```

# D nyelv

## Elő és utófeltételek

```
in { … } out { … } body {…}
```

- `in` és `out` elhagyható
- Ha csak `body` van, akkor a „`body`” tag is elhagyható
- `in`: a hívás előtt fut le
- `out`: a blokk elhagyásakor fut le, bárhogyan is lépünk ki belőle
- `body`: a függvény / blokk törzse
- Szabály: in és out bármilyen utasítást tartalmazhat, de a környezetet nem változtathatja meg
  - tipikusan assertekkel van tele

# D példa elő és utófeltételekre

```
long square_root(long x)
in
{
    assert(x >= 0);
}

out (result)
{
   assert((result *result)<=x &&
            (result+1) * (result+1) >x);
}

body
{
   return cast(long)std.math.sqrt(cast(real)x);
}
```

# D – osztály invariáns

Speciális tagfüggvény: invariant()

- Az osztály adattagjait nem módosíthatja
- A konstruktor elhagyásakor, a destruktor előtt, illetve minden egyes tagfüggvény hívása előtt és után lefut
- Az objektum helyességét ellenőrzi

```
class Date {
 int day;
 int hour;
 invariant()
 {
   assert(1 <= day && day <= 31);
   assert(0 <= hour && hour < 24);
 }
}
```

# Oxygene

Object Pascal alapján

- Eiffel, C#, Java, F# hatások.

Elő és utófeltételek

- Az előfeltételt a `require` kulcsszó után kell megadni. Igaznak kell lennie a metódusba való belépéskor.
  - Például itt megkötéseket tehetünk a metódus paramétereire.
- Az utófeltételnél megadott feltételek az előtt értékelődnek ki, mielőtt a metódus befejeződne és a vezérlés az őt hívó blokkba térne vissza.
  - Az `ensure` kulcsszó után tehetjük
- Az elő és utófeltételeket egy osztály metódusánál lehet alkalmazni.

# Oxygene – invariáns

`invariants` kulcsszó után lehet megadni.

- `public invariants`
  - Az összes publikus metódus végén leellenőrzi (az utófeltétel után).
- `private invariants`
  - Az összes metódus végén leellenőrzi

# Oxygene – invariáns

```
type
   MyClass = class;
      public
        … some methods or properties
             public invariants
               fField1 > 35;
               SomeProperty = 0;
               SomeBoolMethod() and not (fField2 = 5);
             private invariants
               fField > 0;
             end;
```

# Cobra

2009. február 29-én, MIT licensz alatt publikált általános célú, open source programozási nyelv, még jelentős fejlesztés alatt áll.

Chuck Esterbrook tervezte, a Python nyelvhez kapcsolódó fejlesztései miatt ismert.

Ebből következik, hogy a Cobra jelentősen épít a Python alapjaira

- A két nyelv szintaktikája majdnem teljesen megegyező
- Egyelőre halott …

# Cobra

A Cobra a contractok fogalmát átörökítette az Eiffel nyelvből.

- Megkövetelhetjük, hogy bizonyos elő-, utófeltételek és invariáns tulajdonságok teljesüljenek egy metódus végrehajtásakor.
  - require - előfeltételek
  - ensure  - utófeltételek
- A feltételeket egymás alatt felsorolva, logikai kifejezések listájaként adhatjuk meg
  - invariant - osztályinvariáns
  - assert használható
- öröklődnek, a „szokásos” szabályok szerint

# Cobra

```
class Person
    def drive(v as Vehicle)
        require
            not v.hasDriver
            v.isOperable
        ensure
            v.miles > old v.miles
        body
            …
```

# Cobra

```
class ContiguousList<of T>
  implements IList<of T>
    def insert(index as int, item as T)
    require
       index >=0 and index < count
    ensure
      .count = old .count + 1
       this[index] is item
    body ...
```

# Cobra – elő-utófeltétel öröklődés

```
class NonContiguousList<of T>
  inherits ContiguousList<of T>
  """
  Allows insertions past the end of the list.
  """
    def insert(index as int, item as T)
      or require index >= 0
    body ...
```

- utófeltételnél: `and ensure`

# Cobra - invariáns

```
class Player

invariant
  .name.length
  .score >= 0
  .isAlive implies .health > 0
```

# Java – assert

`assert` utasítás

Az utasítással különböző elő- és utófeltételeket, invariánsokat fogalmazhatunk meg.

Előnye a feltételes utasításokkal szemben: használata ki- és bekapcsolható.

Az utasításnak két alakja van:

- `assert logikai_kifejezés;`
- `assert logikai_kifejezés : üzenet;`

# Java – assert

Például két változóról azt feltételezzük, hogy az egyik kisebb, mint a másik.

- Ekkor az utasítás így nézhet ki:
  - `assert x < y;`

Ha a feltétel nem teljesül, akkor az utasításnak ez a formája `AssertionError` hibát dob.

- Ha az üzenetet is megadjuk, akkor azt is kiírja.

Fontos szabály, hogy az `assert` utasítás feltételének nem szabad befolyásolnia az alkalmazás működését, ugyanis az előfeltevések használata kikapcsolható.

Nem érdemes például metódushívást szerepeltetni benne.

- Egyébként minden feltétel szóba jöhet.

# Java – assert példa

Ellenőrzi, hogy a beolvasott érték 0 és 10 között van-e:

```
import java.util.Scanner;
 public class AssertTest {
   public static void main( String args[] ) {
      Scanner input = new Scanner( System.in );
      System.out.print( "Enter a number between 0 and 10: " );
      int number = input.nextInt();
       // assert that the value is >= 0 and <=10
       assert ( number >= 0 && number <= 10 ):
                              "bad number: " + number;
      System.out.printf( "You entered %d\n", number );
    } // end main
} // end class AssertTest

Enter a number between 0 and 10: 5
You entered 5
```

# Java – JML

Java Modelling Language (JML)

- A JML egy formális viselkedési interfész-specifikációs nyelv.
  - A Java osztályok interfészének szokásos szintaktikus leírásán túl az egyes függvények viselkedését is specifikáljuk a rendszerben.
- A JML nyelvben a szerződéseket megfogalmazó feltételeket annotációs megjegyzések formájában helyezhetjük el a programkódban.
  - Ez lehet egy //*-gal kezdődő sor vagy egy /*@ ... @*/ blokk.
- A kifejezésekben a Java jelölésrendszerét használhatjuk, kibővítve néhány speciális elemmel.
- Az elkészült kódot a JML saját fordítójával, a jmlc-vel kell lefordítanunk,
  - A kód fordítható marad az eredeti Java fordítóval is, mivel a JML kommentben szerepel
- A kontraktusok ellenőrzése kikapcsolható.

# Java – JML

```
public class IMath {
  /*@ requires (* x is positive *);
    @  ensures \result >= 0 &&
    @  (* \result is an approximation to square root of x *)
    @ */
  public static int isqrt(int x) { … }
}
```

| Szintaxis | Jelentés |
|---|---|
| \result | Eredmény |
| A==>B | A–ból következik B |
| A<==B | B–ből következik B |
| A<==>B | A iff B |
| A<=!=>B | !(A<==>B) |
| \old(E) | E eredeti értéke |

## Minősítők

- Univerzális és egzisztenciális
  - `\forall \exists`
- Általános
  - `\sum, \product, \min, \max`
- Numerikus
  - `\num_of`

# Java – JML

```
/*@ requires len >= 0
  @ ensures \result ==
  @         (\sum int j; 0 <= j && j < len; v[j])
  @*/
float sum (int v[], int len) {
    float s = 0.0;
    int i = 0;
    while (i < len) {
        s = s + v[i];
        i = i + 1;
    }
    return s;
}
```

# Java – JML

```
byte[] a; /* The array a is sorted */
/*@ invariant
  (\forall int i;
           0 <= i && i < a.length-1;
           a[i] < a[i+1]);
  @ */
```

# Java – JML

Invariáns:

```
public class BankAccount {
  …
  final static int MAX_BAL = 1000;
  int balance;
  /*@ invariant 0 <= balance &&
              balance <= MAX_BAL;
    @*/
  …
}
```

# Java – JML

```
public class BankAccount {
  final static int MAX_BALANCE = 1000;
  int balance;

  // Elő és utófeltétel
  /*@ requires amount >= 0;
    ensures balance == \old(balance)+amount &&
            \result == balance;
    @*/
  public int debit(int amount) {

  }
}
```

# Java – cofoja

A Google –nál fejlesztették 2011-ben

Annotációkat lehet használni

- Típus  invariáns – Invariant kulcsszó – minden publikus és csomagszintű láthatóságú metódus  be- és kilépésnél, és a konstruktorok végén ellenőrzi, öröklődésnél and kapcsolat
- Előfeltételek – Requires kulcsszó - metódus  belépésnél ellenőrzi, öröklődésnél or kapcsolat
- Utófeltételek – Ensures kulcsszó - metódus  normál kilépésnél  ellenőrzi, öröklődésnél and kapcsolat
- Kivételes utófeltételek - ThrowEnsures Checked  - ha exception lépett fel, ezt ellenőrzi
- old és result lehetősége

# Java – cofoja

```
@Invariant("size() >= 0")
interface Stack<T> {
  public int size();

  @Requires("size() >= 1")
  @Ensures({ "size() == old(size()) - 1",
             "result == old(peek())" })
  public T pop();
}
```

# Java – valid4j

Külső könyvtár Java-hoz, amely a kódba beágyazva teszi lehetővé ellenőrzéseket

- Code by contract támogatása
- Előfeltételek definiálása
- Utófeltételek definiálása
- Kivételeket vált ki a feltételek megsértésekor

```
// Initialize members with valid arguments
this.message = require(message, containsString("Greetings"));
validate(v, containsString("Greetings"),
             otherwiseThrowing(MissingGreatException.class));
// Return valid results
return ensure(list, hasSize(greaterThan(1)));
```

# .NET (C# és VB)

Code Contracts Library  a .NET-ben

- http://research.microsoft.com/en-us/projects/contracts/

Előfeltételek: `Contract.Requires(…)`

- általában paraméterek ellenőrzésére
- az előfeltételben szereplő összes tagnak elérhetőnek kell lennie az adott helyen (különben az előfeltételt nem tudja értelmezni a hívó metódus )
- a megadott feltételeknek nem lehet mellékhatásuk.

# .NET

## Előfeltételek: `Contract.Requires(…)`

Példák:

- az x paraméter nem lehet null:

```
Contract.Requires ( x ! = null );
```

- ha az adott feltétel nem teljesül, milyen kivétel váltódjék ki:

```
Contract.Requires<ArgumentNullException>(x != null);
```

# .NET

„Örökölt” követelmények:

- A legtöbb kódban a paraméterek ellenőrzésére if-then-throw szerkezet, ha ezek a metódus elején, át lehet alakítani előfeltételekké, de egy explicit contract metódushívás kell utána
  - pl.: Requires, Ensures, EnsuresOnThrow vagy EndContractBlock

```
if ( x == null ) throw new ...
Contract.EndContractBlock();
// minden megelőző if-et előfeltételnek tekint
```

# .NET

## Utófeltételek

- Ellenőrzése az adott metódus végrehajtása után
- Közönséges utófeltétel

```
Contract.Ensures()
Contract.Ensures( this .F > 0 );
```

- Kivételes utófeltételek:
  - Ha a metódus végrehajtása során kivétel váltódik ki, akkor is lehetőség van az utófeltétel ellenőrzésére.
  - Ekkor a kivétel típusától (T) függően is lehet megadni feltételt:

```
Contract.EnsuresOnThrow<T>( this.F > 0 );
```

# .NET

Speciális metódusok az utófeltételben:

- csak utófeltételek belsejében lehet használni
- `Contract.Result<T>()`
  - hivatkozik a T típusú visszatérési értékre (void típusú függvényeknél nem használható)
- `Contract.Ensures(0 < Contract.Result<int>());`

# .NET

- `Contract.OldValue<T>(exp)`
  az exp kifejezés metódus hívása előtt értékét adja vissza.
  Megszorítások:
  - az exp kifejezés nem tartalmazhat másik régi értéket lekérdező függvényt
  - csak olyan kifejezésre hivatkozhat, aminek létezett értéke a metódus meghívása előtt.
- Példák lehetséges hibákra:
  - A metódus visszatérési értékének a régi értékére nem lehet hivatkozni.

```
Contract.OldValue(Contract.Result<int>() + x) // ERROR
```

  - A régi érték nem függhet a visszatérési értéktől.

```
Contract.ForAll(0,Contract.Result<int>(),
          i => Contract.OldValue(xs[i]) > 3 ); // ERROR
```

# .NET

`Contract.ValueAtReturn<T>(out T t)`

Az out paraméter értékének ellenőrzésére az utófeltételben.

```
public void OutParam(out int x){
    Contract.Ensures(Contract.ValueAtReturn(out x) == 3);
    x = 3;
}
```

# .NET

## Invariánsok (Invariants)

- Az összes invariáns `void` típusú függvény kell legyen, ha az adott osztályból lehet származtatni, akkor a láthatóságnak protected-et kell megadni.
- A `[ContractInvariantMethod]` attribútummal jelöljük, hogy az adott metódus egy invariáns
- Az invariáns ellenőrzése az összes publikus metódus végrehajtása után megtörténik.
- Ha az invariánson belül hivatkozunk az osztály egy másik publikus metódusára, akkor csak a legkülső függvénynél történik ellenőrzés.

# .NET

```
[ContractInvariantMethod]
    protected void ObjectInvariant()
    {
        Contract.Invariant( this.y >= 0 );
        Contract.Invariant( this.x > this.y );
        ...
    }
```

# .NET

## Contract.Assert

- a program egy bizonyos pontján tudunk ellenőrizni:

```
Contract.Assert( this.privateField > 0 );
Contract.Assert( this.x == 3,
            "Why isn't the value of x 3?" );
```

## Contract.Assume

- Egy feltevést ír le, működése megegyezik az Assert-tel futási időben, de itt fordítási idejű ellenőrzés is van!

```
Contract.Assume( this.privateField > 0 );
Contract.Assume( this.x == 3,
             "Static checker assumed this");
```

# .NET

## Contract.EndContractBlock

- ha az előfeltételek if-then-throw formában vannak leírva, akkor ez jelzi, hogy ezek előfeltételek, itt van az ellenőrző blokk vége:

```
if ( x == null ) throw
        new ArgumentNullException("x");
    if ( y < 0 ) throw
        new ArgumentOutOfRangeException(...);
    Contract.EndContractBlock( );
```

# .NET

## Contract.ForAll

- Ellenőrző ciklus, contract-on belül használható
- Két paraméteres változat:

```
public int Foo<T>(IEnumerable<T> xs){
    Contract.Requires(
       Contract.ForAll( xs, (T x) => x != null) );
```

- Első paraméter egy kollekció
- A második egy predikátum
  - ha ez igaz a gyűjtemény minden elemére, akkor igazat ad vissza
  - ha van olyan elem, amire hamis, akkor megáll, és hamisat ad vissza.

# .NET

- Három paraméteres változat:

```
public int[] Bar(){
  Contract.Ensures(
  Contract.ForAll(0, Contract.Result<int[]>().Length,
        index => Contract.Result<int[]>()[index] > 0));
}
```

- Első paraméter az alsó határ
- A második a felső határ
- A harmadik a predikátum, aminek van egy index argumentuma
  - A két határ között bejárja a predikátumban adott kollekció elemeit, ellenőrzi, hogy igaz-e mindegyikre.

# .NET

`Contract.Exists`

- Ugyanolyan paraméterei vannak, mint a ForAll-nak
- Akkor tér vissza igaz értékkel, ha a kollekció legalább egy elemére teljesül az adott predikátum és hamis értékkel tér vissza, ha egyre sem.

# .NET

## Interfész contract-ok

- Itt nem írhatunk függvény törzset, ezért egy külön contract osztály kell, amit az interfésszel attribútumok kapcsolnak össze.

```
[ContractClass(typeof(IFooContract))]
interface IFoo {
      int Count { get; }
            void Put(int value );
}
```

# .NET

## Interfész contract-ok

```
[ContractClassFor(typeof(IFoo))]
 abstract class IFooContract : IFoo {
   int IFoo.Count {
     get {
      Contract.Ensures( 0 <= Contract.Result<int>() );
      return default( int ); // dummy return
      // lehet egy exc.-t is dobni itt
     }
   }
   void IFoo.Put(int value){
     Contract.Requires( 0 <= value );
   }
 }
```

# .NET

## Absztrakt metódus szerződések

- itt se lehet függvény törzset írni, ezért itt is egy külön contract osztály kell, amit az absztrakt osztállyal  attribútumok kapcsolnak össze:

```
[ContractClass(typeof(FooContract))]
  abstract class Foo {
    public abstract int Count { get; }
    public abstract void Put(int value );
  }
```

# .NET

Absztrakt metódus szerződések (folyt.):

```
[ContractClassFor(typeof(Foo))]
  abstract class FooContract : Foo {
    public override int Count {
      get {
       Contract.Ensures( 0 <= Contract.Result<int>() );
                return default( int ); // dummy return
     }
    }
    public override void Put(int value){
        Contract.Requires( 0 <= value );
    }
  }
```

# .NET

## Contract metódusok túlterhelése:

- Mindegyik metódusnak lehet egy string típusú paramétere is.
- Ez kiíródik, ha a feltétel nem teljesül.
  - A string értékének fordítási időben ismertnek kell lennie.

```
Contract.Requires(x ! = null,
    "If x is null, then the missiles are fired!");
```

## Contract öröklődés

- Egy szerződés a típus altípusára is öröklődik.
- Az altípus előfeltételének gyengébbnek kell lennie ezért, ha az őstípusnál nincs megadva semmilyen előfeltétel, akkor az azt jelent, hogy az azonosan igaz az előfeltétele ezért az altípusokhoz nem lehet előfeltételt írni.
- Az utófeltételnél erősebbet kell / lehet megadni.

# Ada 2012

Elő- és utófeltételek megadhatók

```
procedure Push(S: in out Stack; X: in Item)
  with
    Pre => not Is_Full(S),
    Post => not Is_Empty(S);
```

lehet az eredeti értékre is hivatkozni az utófeltételben:

- `Post => I = I'Old`
- `A(I)'Old`
- `A(I'Old)`

# Ada 2012

Típusinvariáns

```
type Disc_Pt is private
     with Type_Invariant => Check_In(Disc_Pt);

type Stack is private
   with Type_Invariant => Is_Unduplicated(Stack);
```

# Python – pycontracts

Python esetén több külső programkönyvtár is elérhető

- icontract
- PyContracts
- Decontractors
- dpcontracts
- zope.interface
- PyDBC
- Contracts for Python
- …

Tervezték a „natív” támogatást is

- Egyelőre elhalasztották

# Python – pycontracts

Példák megadásra:

```
@contract(a='int,>0', b='list[N],N>0', returns='list[N]')
def my_function(a, b):

@contract
def my_function(a : 'int,>0', b : 'list[N],N>0') -> 'list[N]':
     # Requires b to be a nonempty list, and the return
     # value to have the same length.

@contract
def my_function(a, b):
    """ Function description.
        :type a: int,>0
        :type b: list[N],N>0
        :rtype: list[N]
    """
```

# Python – pycontracts

Több feltétel

```
@contract(prob='>=0,<=1')
def f(prob):

@contract(x='<0|>1')
def f(x):

@contract(x=int)
def f(x):

@contract(x='(int|float),>=0')
def f(x):
```

# Python – pycontracts

Saját „contract” definiálása

- Dekorátor létrehozása

```
@new_contract
def even(x):
  if x % 2 != 0:
    msg = 'The number %s is not even.' % x
    raise ValueError(msg)
```

# Kérdések

Az elő- és utófeltételes specifikációt alkalmazó programozási nyelvekkel kapcsolatos kérdések

- Alprogramoknak megadhatunk-e, és ha igen, milyen formában elő- és utófeltételeket?
- A típushelyesség ellenőrzéséhez megadhatunk-e specifikációs- és típusinvariánst?
- Kezeli-e a specifikációs elő- és utófeltételek és az implementált program elő - és utófeltételeinek különbségét, és így ellenőrzi-e a megfelelés helyességét?
- Vannak-e eszközei a ciklushelyesség ellenőrzéséhez?
- Exception-helyességet támogatja-e?
  - Exception kezelésnél legalább a típusinvariánst helyre kell állítani.
- Milyen az öröklődés és az elő- és utófeltételek, valamint a típusinvariáns kapcsolata?

# „Programming methodology” kurzus MSc-n

FOLYTATÁSBAN