# **Tartalomjegyzék**

[**Tartalomjegyzék**](#_u40c61nfkve9) **1**

[**Ellenőrző kérdések - 01 - Áttekintés**](#_diqcnlqz2p3h) **6**

[✓ F01-01 Sorolj fel 5-10 fogalmat, témakört, problémát az alábbi paradigmák mindegyikéhez.](#_7copqba75cu) 6

[**Ellenőrző kérdések - 02 - Imperatív programozás**](#_2p64fzkh3ul9) **7**

[✓ F02-01 Hol tárolódnak egy program változói? Egy Eiffel program változóiról mit mondhatunk? Mik az expanded osztályok?](#_2fdsfh9kvtsb) 8

[✓ F02-02 Milyen szerepekben használható a create kulcsszó? Mit jelent ezekben a szerepekben?](#_poz6flz8cswd) 8

[✓ F02-03 Milyen utasítások, vezérlési szerkezetek találhatók Eiffelben? Mi a pontosvessző szerepe?](#_w229sv122wy9) 8

[✓ F02-04 Mire való, hogyan használható a debug utasítás az Eiffelben?](#_doflficu6t11) 9

[✓ F02-05 Milyen változók írhatók (azaz milyen változóknak adhatunk értéket) az Eiffelben? Mi az assign procedure szerepe?](#_ayj5smn7b1t) 9

[**Ellenőrző kérdések - 03 - Szerződésalapú programozás**](#_8akcx0d58hqv) **10**

[F03-01 Mutasd be a check utasítás szerepét és hasznát, használatának módját! Más nyelvekben milyen hasonló lehetőséget láthatunk?](#_w9wm5xuhwmhm) 10

[✓ F03-02 Hogyan néz ki egy until-ciklus az Eiffelben? Hogyan adhatjuk meg a szerződését? Hogyan viszonyul ez a ciklus levezetési szabályához?](#_anmnr7mi7cxx) 10

[F03-03 Mik a Hoare-hármasok? Mire használjuk őket?](#_gw525y8wrr2v) 11

[F03-04 Ismétlés más tárgyakból: Hogyan definiáljuk a leggyengébb előfeltételt? Hogyan számíthatjuk ki egy értékadó utasításra?](#_11hmdesbcq87) 12

[F03-05 Mi egy rutin szerződése az Eiffelben? Hogyan adjuk meg? Hogyan írjuk le a jelentését Hoare-hármassal? Hogyan viszonyul ez az elő- és utófeltételes specifikációkhoz?](#_u8x031dhpjy3) 12

[**Ellenőrző kérdések - 04 - Objektumelvű programozás**](#_6eph61dos6zl) **13**

[F04-01 Hogyan védekezik az Ada nyelv az üres referenciák által okozott problémákkal szemben?](#_o4mtstj8w3yh) 13

[F04-02 Mik az annotációk a Javában? Mire való a javax.annotation.NonNull annotáció?](#_qfj4u8ajpiw) 13

[F04-03 Mi a Maybe (avagy Option avagy Optional) típus? Hogyan segít az üres referenciák okozta problémák leküzdésében? Például: hogyan emelhető be egy számítás az Option monádba a Scalában?](#_rd8z7r5uem54) 14

[F04-04 Mit értünk aliasing alatt? Mire jó? Miért okoz problémákat? Hogyan segít a szemétgyűjtés?](#_27x9374ptil0) 14

[F04-05 Mik az attached és a detachable típusok az Eiffelben? Hogyan jelöljük őket? Mi a viszonyuk egymáshoz?](#_mewrantfmtqz) 15

[F04-06 Hogyan kell egy detachable típusú kifejezést értékül adni egy attached változónak?](#_fm73qxb2qkyv) 15

[F04-07 Mit értünk dinamikus szemantikai hiba alatt? Miben más ez, mint a statikus szemantikai hiba?](#_vve8vawoqcjx) 16

[✓ F04-08 Mik a dokumentációs megjegyzések? Mire valók? Hogyan írunk ilyet Eiffelben?](#_sih8jj5fb0i5) 16

[F04-09 Mire való a safe navigation operator és a null-coalescing operator (időnként Elvis-operator)? Mutasd be példákon!](#_c61iblsgmrjl) 16

[✓ F04-10 Mi a különbség a közönséges és az expanded osztályok között?](#_uasj5o80um3m) 17

[F04-11 Sorold fel és illusztráld példákon, milyen fajta feature-ök alkotják az osztály definíciókat Eiffelben! Melyik miben tér el a másiktól? Mik azok a lekérdezések?](#_kn8ma3fm5rbl) 18

[F04-12 Adj áttekintést arról, milyen lehetőségek vannak az üres referenciák által okozott problémák kezelésére a különböző nyelvekben!](#_rticnk7fha69) 19

[✓ F04-13 Mik az üres (Eiffelben: Void) referenciák? Milyen problémát okoznak?](#_ltpa70h39swv) 19

[✓ F04-14 Mi az osztályinvariáns? Magyarázd el szóban is és formálisan megfogalmazott szerződésekkel is a szerepét! Mi a szintaxisa az Eiffelben?](#_d27ytg5aouof) 19

[F04-15 Mit értünk statikus típusozás alatt? Mutass példát ezt követő, illetve nem követő nyelvekre! Térj ki a típusellenőrzés mellett a típuskikövetkeztetésre is!](#_yv401raszx7g) 20

[F04-16 Mit nevezünk szemétgyűjtésnek (garbage collection)?](#_y28t1bibbgq8) 21

[**Ellenőrző kérdések - 05 - Procedurális programozás**](#_alojupjozmkp) **21**

[F05-01 Mik a mohó és a lusta operátorok? Mutass példát Eiffelben!](#_dxtlax3hnge4) 21

[F05-02 Mi a különbség az imperatív és a funkcionális stílusú rutin között? Mutasd be a divide\_by és a divided\_by példáján!](#_vz8enb35y6cl) 21

[✓ F05-03 Hogyan fejezhetjük ki a nemváltozás tényét Eiffelben?](#_sfd0g012t8k1) 21

[✓ F05-04 Mire való az old operátor? Hogyan fejezünk ki vele változást és nemváltozást?](#_otf6itkr34h8) 22

[✓ F05-05 Hogyan kell operátorokat definiálni az Eiffelben?](#_v1axpajkfx0) 22

[✓ F05-06 Milyen operátorok vannak az Eiffel nyelvben?](#_ujcoj1aps9ee) 23

[F05-07 Hogyan működik a paraméterátadás Eiffelben? (2-fajta...)](#_ngt95093w63s) 23

[F05-08 Milyen problémát vet fel az aliasing a referencia szerinti paraméterátadásnál? Mutasd be egy példa segítségével!](#_wwmnjtahjr8q) 23

[**Ellenőrző kérdések - 06 - Generikus programozás**](#_h56tlt1jkd4m) **24**

[✓ F06-01 Sorold fel az ARRAY[T] osztály legfontosabb műveleteit! Mi helyettesíti az elavult make creation procedure-t?](#_17n3d0np2w9w) 24

[✓ F06-02 Mit szokás generikus definíciónak nevezni egy programozási nyelvben? Eiffelben milyen generikus definíciót lehet írni?](#_x8j1tdeht153) 24

[✓ F06-03 Mi az Eiffel-terminológia jelentése az osztály, generikus osztály és a típus kapcsán?](#_jer3tox46a80) 24

[✓ F06-04 Mutasd be egy példán, hogyan kell Eiffelben egy generikus osztályt definiálni, illetve használni!](#_nhplkgtabrj) 24

[**Ellenőrző kérdések - 07 - Öröklődés**](#_j2niqhin8itj) **25**

[✓ F07-01 Mondd ki a Liskov-féle helyettesítési elvet!](#_gfqbid8q4sxv) 25

[✓ F07-02 Hogyan hívjuk meg a régi törzset egy felüldefiniált feature-ben?](#_p3lrgg8wloxz) 25

[✓ F07-03 Mit értünk a S.O.L.I.D. elvek alatt?](#_j10n29mh0i05) 25

[✓ F07-04 Hogyan adjuk meg a dinamikus típust egy create utasításban, illetve egy create kifejezésben?](#_feec57l3we8r) 25

[✓ F07-05 Mi a default\_create?](#_k3n6jfe6ji1v) 26

[✓ F07-06 Hogyan fejezzük ki a dinamikus típusellenőrzést az Eiffelben?](#_7kq2yk9eimou) 26

[✓ F07-07 Hogyan módosítható egy rutin szerződése a felüldefiniálás során? Mi ennek a formája?](#_myrasswgz1vl) 26

[✓ F07-08 Mit jelent az, ha egy osztályba nem írunk invariáns klózt, illetve, ha egy feature-be nem írunk elő- és utófeltételt?](#_vrgaonbvumty) 27

[F07-09 Hogyan fejezzük ki az öröklődést Eiffelben? Mi az alapvetés?](#_mmnq9a2ltqcp) 27

[✓ F07-10 Hogyan fejezzük ki a feature-ök láthatóságát Eiffelben? Mik azok a nyilvános és titkos feature-ök?](#_6x0jlo588z0r) 27

[✓ F07-11 Hogyan kell felüldefiniálni egy feature-t az Eiffelben?](#_vik097l540sz) 27

[F07-12 Mire való az átnevezés az Eiffelben? Mutass példát a használatára!](#_wqhq24s694ol) 28

[F07-13 Mi a kapcsolat az öröklődés és a creation procedure-ök között?](#_kfpcsvkm4x35) 28

[✓ F07-14 Mit értünk egy osztály invariánsa alatt? (Vedd figyelembe az öröklődést is!)](#_52y2n2exyd7x) 28

[**Ellenőrző kérdések - 08 - Absztrakt osztályok**](#_5k24ezmclprj) **29**

[✓ F08-01 Mit értünk absztrakt típus alatt az OOP-nyelvekben? Mi a szerepe? Mi ennek a megfelelője az Eiffelben?](#_x1nb2zez9njd) 29

[✓ F08-02 Mit értünk az alatt, hogy egy absztrakt típus megvalósítása? Mutass példát az Eiffelben (is)! Mit lehet tudni ez ügyben az adattagokról (attribútumokról)?](#_5wtes3m27us3) 29

[✓ F08-03 Mit jelent a kovariáns visszatérési típus? Mutass rá példát! Miért jó?](#_709xdnko0dx0) 30

[✓ F08-04 Lehet deferred feature-nek szerződése? Magyarázd el!](#_yhezfmh1lkx6) 30

[✓ F08-05 Mit jelent a frozen kulcsszó egy művelet deklarációjában? És egy osztály deklarációjában?](#_cjdu5ov8x1tz) 30

[✓ F08-06 Mi a módja az Eiffelben annak, hogy egy modul használ egy másik modult?](#_tcu8yc8sjolc) 31

[✓ F08-07 Mik a once-rutinok az Eiffelben? Hogyan működnek? Milyen veszéllyel járnak?](#_fdbigul1civf) 31

[**Ellenőrző kérdések - 09 - Variancia**](#_wyeylko2pl7e) **31**

[✓ F09-01 Milyen problémát vet fel a kovariáns paraméter?](#_ttqqsqdt8oyd) 31

[✓ F09-02 Milyen alakúak, milyen tulajdonságúak az A -> B függvények típusának altípusai?](#_5s75swfvjek5) 31

[✓ F09-03 Mit értünk binary method alatt? Mutass példát is rá!](#_mshnpfyt0w9t) 32

[✓ F09-04 Mit jelent a kovariáns visszatérési érték (covariant return) kifejezés? Mutass rá példát!](#_bbdxgzmooswy) 32

[✓ F09-05 Magyarázd el, mit jelent a kapcsolt típus (anchored type) az Eiffelben! Mire jó? Mi az előnye?](#_qklyg1in43pz) 32

[✓ F09-06 Mit jelent a kontravariáns típus kifejezés? Mutass példát pszeudokóddal!](#_3wt389kbj685) 33

[✓ F09-07 Mit jelent a kovariáns típusú paraméter? Mutass példát, amely ennek az előnyét tárja fel!](#_k8txgj613q05) 33

[✓ F09-08 Mit jelent polymorphic CAT-call kifejezés? Miért érdekes ez a fogalom?](#_czlt97v7v1ei) 34

[✓ F09-09 Miben hasonlít a függvénytípusok altípusaira vonatkozó szabály az Eiffel feature-ök szerződéseinek öröklődéssel kapcsolatos mechanizmusára?](#_93fnltl8ox9w) 34

[✓ F09-10 Hogyan néz ki az Eiffelben az ANY osztály klónozásra bevezetett twin művelete?](#_gefs42e0b1l2) 34

[**Ellenőrző kérdések - 10 - Egyenlőségvizsgálat**](#_xiehcvsz8qh4) **34**

[✓ F10-01 Mit jelent a = és a ~ operátor az Eiffelben?](#_fyob8uqwlezz) 34

[F10-02 Milyen tulajdonságokkal kell rendelkezzen egy ekvivalenciareláció, és milyenekkel egy parciális rendezés?](#_vr710qi4d4b) 35

[✓ F10-03 Hogyan definiálhatunk saját logikájú (custom) tartalmi összehasonlítást egy Eiffel osztályban?](#_1e0jf57ofdrn) 35

[✓ F10-04 Mi a különbség a referencia-, illetve a kifejtett típusokon definiált egyenlőségvizsgálatok között?](#_fyqz1uo1xkf8) 35

[✓ F10-05 Hogyan függnek össze, hogyan alkotnak rendszert az egyenlőségvizsgálatra és az objektummásolásra szolgáló feature-ök az Eiffelben?](#_jqfo4sb4qam6) 35

[✓ F10-06 Hogyan viszonyulnak egymáshoz a standard\_equal, az equal és a deep\_equals relációk?](#_hr9bfhsq5t4d) 35

[✓ F10-07 Hogyan függ az equal az is\_equal-tól?](#_9o71hl89ze2k) 36

[✓ F10-08 Mi a különbség az object identity és az object equality között?](#_bs7ziufxqwy8) 36

[✓ F10-09 Mi az is\_equal szignatúrája az Eiffel ANY osztályában?](#_oontbfxlaiw) 36

[✓ F10-10 Mi a szerződése az is\_equal-nak? Mi az alapértelmezett implementációja?](#_fdoqjpni0ih8) 36

[✓ F10-11 Mit illik tudni a Java egyenlőségvizsgálatot végző equals metódusának paramétertípusáról? Hogyan kapcsolódik ez a felüldefiniálás és a túlterhelés kérdéséhez?](#_4mlzdm2rv4ra) 36

[✓ F10-12 Hogyan viszonyulhat az altípusosság az egyenlőségvizsgálathoz? Mi a két fő lehetőség?](#_8bgr79i93j6u) 36

[✓ F10-13 Mi a különbség a sekély és a mély összehasonlítás között?](#_i7qwljmoyhv2) 37

[✓ F10-14 Mit értünk az alatt, hogy öröklődés kiváltása kompozícióval?](#_8o1hnmhrfdcg) 37

[**Ellenőrző kérdések - 11 - Típusmegfelelőség**](#_a2lcuiiwxy4t) **37**

[✓ F11-01 A strukturális típusosságnál miben tér el az értékekre, illetve a változókra vonatkozó altípusfogalom? Azaz mi a fő különbség a funkcionális és az imperatív paradigma között ebben a tekintetben?](#_wfmwlxk483i5) 37

[F11-02 Mutass példát a conversion function működésére, használatára az Eiffelben!](#_ath021wz6rgc) 38

[F11-03 Mutass példát a conversion procedure működésére, használatára az Eiffelben!](#_ap2yq9uk173l) 38

[✓ F11-04 Mit nevezünk curryingnek?](#_u1j3zpnog0yr) 39

[✓ F11-05 Mit értünk downcast alatt? Hogyan fordul elő ilyen az Eiffelben? Mutass példát!](#_6d4o9vb08rk5) 39

[✓ F11-06 Mit értünk duck-typing alatt? Mutass példát!](#_agr8utkweodb) 40

[✓ F11-07 Milyen altípus-relációban vehetnek részt a kifejtett (expanded) típusok?](#_s9f6i256fzud) 40

[✓ F11-08 Mi a nominális és a strukturális típusekvivalencia, illetve altípusosság?](#_hp6k31t4qhty) 40

[✓ F11-09 Mit értünk privát öröklődés alatt? Milyen tulajdonságai vannak? Hogyan csinálunk ilyet az Eiffelben?](#_sv2y0ydk07cd) 40

[✓ F11-10 Milyenek a prototípus alapú objektumelvű nyelvek? Mi jellemzi őket?](#_9dicem8hel51) 40

[✓ F11-11 Hogyan definiálhatjuk a strukturális altípusosságot az összetett típusú módosítható változókra?](#_j3z0z7pus04) 40

[✓ F11-12 Hogyan definiálhatjuk a strukturális altípusosságot az összetett típusú értékekre?](#_tgw0ydo5f6it) 41

[✓ F11-13 Mit értünk upcast alatt? Hogyan fordul elő ilyen az Eiffelben? Mutass példát!](#_95ila4ybl339) 41

[**Ellenőrző kérdések - 12 - Többszörös öröklődés**](#_7hhwe7r56uks) **41**

[✓ F12-01 Sorolj fel nyelveket, melyek csak egyszeres, illetve többszörös öröklődést is támogatnak!](#_3xofh29vutp1) 41

[✓ F12-02 Mit értünk ismételt öröklődés (repeated inheritance) alatt?](#_er6culxtda4z) 42

[✓ F12-03 Hogyan oldja meg az Eiffel a többszörös öröklődés szokásos problémáit?](#_5vbaj3yvjtyn) 42

[✓ F12-04 Mire jó a select kulcsszó az Eiffelben?](#_k56mf0l4jrgf) 42

[F12-05 Milyen különböző eseteket különböztethetünk meg a többszörös öröklődésben a szülőtípusokhoz való konformitás jegyében? Mutass példákat!](#_newizho4g3ps) 43

[F12-06 Milyen problémákat okozhat a többszörös öröklődés pl. a C++ nyelvben?](#_otku8bm6aeh9) 43

[✓ F12-07 Mire jó az undefine kulcsszó az Eiffelben?](#_n137zry0bd6p) 43

[**Ellenőrző kérdések - 13 - Láthatóság**](#_22ejflczzugb) **44**

[✓ F13-01 Adható-e új érték egy attribútumnak? Mi az assign procedure?](#_kktda3318txt) 44

[✓ F13-02 Mit jelent a CAT az Eiffelben? Magyarázd el!](#_rkiewreockc) 44

[✓ F13-03 Megváltoztatható egy feature láthatósága az öröklődés során? Hogyan? Milyen problémát vet ez fel?](#_nno4wi4ovc1w) 44

[F14-04 Milyen lehetőségek vannak az Eiffelben a láthatóság megadására? Hol lehet ezeket a lehetőségeket használni, mire lehet megadni a láthatóságot?](#_z9gyix385jet) 44

[✓ F14-05 Milyen problémát okoznak az Eiffelben a módosítható (mutable) adattagok az információ elrejtésében?](#_2pwsp2oyprei) 44

[✓ F14-06 Mit tekinthetünk mutable, illetve immutable objektumnak?](#_7ynrxhq90fwa) 45

[✓ F14-07 Mik a jellemző láthatósági kategóriák az OOP-nyelvekben? Hogyan feleltethető meg ennek az Eiffel módszere?](#_fqvy3u4si7e) 45

[✓ F14-08 Mit jelent a szelektív láthatóság az Eiffelben? Hogyan viszonyul az altípusossághoz?](#_urrovvlhh043) 45

[F14-09 Mit jelent az Eiffelben a titkos feature fogalma? Mennyire titkos?](#_skp92uy69u5o) 45

[**Egyéb: generic paraméterek, virtuális típusok, agent**](#_qdcfsnvwna1v) **46**

[**Kivételkezelés**](#_qvff3ewffknl) **50**

[**Helyesség**](#_j43ypw91csmv) **55**

# **Ellenőrző kérdések - 01 - Áttekintés**

## ✓ F01-01 Sorolj fel 5-10 fogalmat, témakört, problémát az alábbi paradigmák mindegyikéhez.

* Imperatív programozás
* Procedurális programozás
* Funkcionális programozás
* Moduláris programozás
* Objektumelvű programozás
* Generikus programozás
* Konkurens és párhuzamos programozás

Imperatív programozás:

* alapvető absztrakciós eszköz: utasítás
* összetett utasítások: vezérlési szerkezetek: szekvencia, ciklus, elágazás, switch, goto, break
* utasítások sorrendben történő végrehajtása
* a memóriát manipuláljuk utasításokkal
* utasítás kiolvas valamit a memóriából, ír a memóriába
* mellékhatásos kifejezések
* input, output, megjegyzések

Procedurális programozás

* alprogram (számítási absztrakció): procedúra, függvény, eljárás
* alprogram többször meghívható (újrafelhasználás)
* alprogram paraméterezhető
* alprogram általában rekurzívan hívható
* hatókör
* kivételkezelés
* blokkszerkezetesség: alprogramon belül alprogram definiálása
* túlterhelés (overloading)

Funkcionális programozás

* legszebb dolog a világon
* rekurzió
* magasabb rendű függvények
* pure / impure (tiszta / nem tiszta)
* lusta / mohó kiértékelés
* részleges alkalmazás
* currying
* mintaillesztés
* Eiffel: agent (ágens)

Moduláris programozás

* modul: logikailag összetartozó kódok, egy nagyobb feladat különböző részeinek megvalósítására
* modul lehet: modul, package, osztály, nyelvtől függően
* egységbezárás
* információs elrejtés
* modulok szűk interfésszel
* modulokon belül nagy a belső kohézió
* könnyen módosítható implementáció (az előző 2 miatt)
* modulok helyettesíthetősége (interfész egyezés esetén)
* külön fordítás lehetősége

Objektumelvű programozás

* adatközpontú dekompozíció: a megoldandó feladatot a felhasználandó adatokból felépítve építem fel
* típusmegvalósítás
* öröklődés, többszörös öröklődés
* felüldefiniálás
* dinamikus kötés
* altípusos polimorfizmus
* az adatok és az adatokon értelmezett műveletek szoros összekapcsolása

Generikus programozás

* típusfüggetlen kód: adatszerkezet, algoritmus
* parametrikus polimorfizmus: típussal való paraméterezés
* korlátozott parametrikus polimorfizmus: megkötés (altípus) a típusparaméterre

Konkurens és párhuzamos programozás

* folyamatok
* ütemezés: vagy a prog. nyelv végzi vagy az OPrendszer
* szinkronizáció, kommunikáció
* nemdeterminisztikusság
* interferencia, haladási problémák
* Eiffel: SCOOP kulcsszó
* Bertrand Meyer elképzelése egészen más a világról, mint a világ többi részének 🙂

# **Ellenőrző kérdések - 02 - Imperatív programozás**

## ✓ F02-01 Hol tárolódnak egy program változói? Egy Eiffel program változóiról mit mondhatunk? Mik az *expanded* osztályok?

Vannak referencián keresztül elérhető változók, ezek a heapen tárolódnak, a legtöbb objektum ilyen. Vannak olyan változók amelyek a stacken is képesen létrejönni, az Eiffel expandált objektumnak hívja ezeket, pl: INTEGER.  
Változó lehet: adattag, lokális változó, formális paraméter. Az Eiffelben minden objektum.

## ✓ F02-02 Milyen szerepekben használható a *create* kulcsszó? Mit jelent ezekben a szerepekben?

create kulcsszó:

* objektum létrehozáshoz create utasításban: a korábban deklarált, általában még Void objektum inicializálására, pl: lokális változó inicializálása

john: PERSON

create john.set\_name(“john”)

* objektum létrehozáshoz create kifejezésben: típus alapján új objektum létrehozására, mint más nyelvekben a “new”, pl új objektumot paraméterként átadni

… create {PERSON}.set\_name(“Peter”)

* a classok klózai között a creation procedure megjelölésére
* megkötéses típus paraméterek: VECTOR[G -> ADDABLE create make end]

## ✓ F02-03 Milyen utasítások, vezérlési szerkezetek találhatók Eiffelben? Mi a pontosvessző szerepe?

Csak strukturált utasítások vannak, nincs például break, return.

utasítások, vezérlési szerkezetek

* értékadás: kettőspont egyenlő “:=”
* elágazás: if then elseif else end
* elágazás: inspect, sokágú elágazás
* ciklus: from until loop (while)
* ciklus: across, adatszerkezet bejárása
* check
* debug
* create
* procedure call
* retry

A pontosvessző opcionális.

## ✓ F02-04 Mire való, hogyan használható a *debug* utasítás az Eiffelben?

A szoftverfejlesztés támogatására került a nyelvbe. A debug(Debug\_key) … end közötti utasítások csak akkor hajtódnak végre, ha a debug be van kapcsolva és az indításkor az adott Debug\_key meg van adva.

## ✓ F02-05 Milyen változók írhatók (azaz milyen változóknak adhatunk értéket) az Eiffelben? Mi az *assign procedure* szerepe?

Minden objektum csak a saját mezőit és lokális változóit írhatja, más objektum változóinak és a formális paramétereknek nem lehet értéket adni. Más objektum belső állapotát csak annak műveletei segítségével lehet megváltoztatni.

Az assign procedure egy syntax sugar, amivel úgy tűnik, hogy közvetlenül lehet írni egy másik objektum attribútumát, valójában egy eljárást hívok meg rajta.

name: STRING assign set\_name  
set\_name(new\_name: STRING)  
…  
friend.name := “new name” – itt a set\_name hívódik meg, nem a name-et írjuk

# 

# **Ellenőrző kérdések - 03 - Szerződésalapú programozás**

## F03-01 Mutasd be a check utasítás szerepét és hasznát, használatának módját! Más nyelvekben milyen hasonló lehetőséget láthatunk?

Formája:

check Assertion [ then Compound ]

end

Assertion is a sequence of zero or more assertions, it will have no effect unless assertion monitoring is turned on at the Check level or higher. If so it will evaluate all the assertions listed, having no further effect if they are all satisfied; if any one of them does not hold, the instruction will trigger an exception. This instruction serves to state properties that are expected to be satisfied at some stages of the computation -- other than the specific stages, such as routine entry and exit, already covered by the other assertion mechanisms such as preconditions, postconditions and invariants. A recommended use of check involves calling a routine with a precondition, where the call, for good reason, does not explicitly test for the precondition.

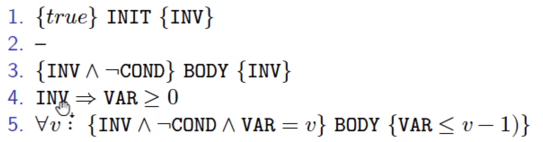
C++: assert()

## ✓ F03-02 Hogyan néz ki egy *until*-ciklus az Eiffelben? Hogyan adhatjuk meg a szerződését? Hogyan viszonyul ez a ciklus levezetési szabályához?

| from -- inicializálás  i := 0  invariant -- opcionális, ciklusba belépés előtt,  -- ciklusmag végrehajtás előtt  -- és ciklusmag végrehajtás után kell fennállnia  n > 0  variant -- opcionális, haladási feltétel, csökkennie kell  n - i + 1  until -- kilépési feltétel  i = n+1  loop -- ciklusmag  print (i)  print ("%N")  i := i+1  end |
| --- |

A ciklus szerződése az invariánsa és a variánsa együtt. Az invariáns egy biztonsági feltétel (mi az ami nem romlik el), a ciklusba belépéskor teljesülnie kell (inicializáció után), a ciklusmag végrehajtása előtt és után is, a ciklusmag végrehajtása közben nem szükséges fennállnia. Az invariánsban logikai feltételeket adhatunk meg. A variáns (merre haladunk) a haladási feltétel, ennek csökkennie kell a ciklusmag végrehajtásai között. A program végrehajtási menetébe vannak beépítve ellenőrzési pontok és ezek a program végrehajtása során bekövetkező vizsgálatok mentén próbálják kimutatni a szerződésszegést. Ez gyengébb mint a bizonyítás, jobb mint a tesztelés.

Hogyan viszonyul ez a ciklus levezetési szabályához? Röviden: az Eiffel ciklus megfelel a ciklus levezetési szabályoknak, kivéve: nincs elő és utófeltétel.



1. előfeltételre nem adunk megkötést {true}, a from részben olyan állapotot állítunk be, ami megfelel az invariánsnak
2. az utófeltételre nem adunk megkötést
3. az invariant és nem until-ból a loop-ot végrehajtva az invariant-ba jutunk
4. az invariant-ból következik, hogy a variant >= 0
5. a variant értéke csökken a loop végrehajtásával

## F03-03 Mik a Hoare-hármasok? Mire használjuk őket?

Hoare logikája lehetővé teszi annak megmondását, hogy a kód egy része hogyan változtatja meg a rendszer állapotát. A Hoare-hármasokat a számítástechnikai programozásban és a formális módszereket használó területeken, például a számítógépes biztonsági elemzésben és a formális verifikációban használják.

A Hoare-hármasokat általában a következő formában írják le:

{P} C {Q}

Ebben a leírásban:

* P az úgynevezett "előfeltétel" (precondition), amely meghatározza azokat az állapotokat, amelyekben a parancs (C) érvényesíthető.
* C a parancs, amelyet a Hoare-hármasban leírunk.
* Q az úgynevezett "utófeltétel" (postcondition), amely meghatározza azokat az állapotokat, amelyeket a parancs (C) végrehajtása után elérünk.

Tehát a Hoare-hármasok segítségével leírhatjuk, hogy egy adott parancs milyen hatással van a program állapotára, és milyen feltételeknek kell teljesülniük ahhoz, hogy a parancs érvényesíthető legyen. A Hoare-hármasok használatával lehetővé válik a programok bizonyos tulajdonságainak formális igazolása, például a helyességük vagy a teljesítményük.

## F03-04 Ismétlés más tárgyakból: Hogyan definiáljuk a leggyengébb előfeltételt? Hogyan számíthatjuk ki egy értékadó utasításra?

A leggyengébb előfeltétel az a feltétel, amely a legkevésbé korlátozza a parancs végrehajtását.

Az értékadó utasítások esetében a leggyengébb előfeltétel az a feltétel, amely biztosítja, hogy az értékadás érvényes legyen. Az értékadás érvényes, ha az adatok szerint a célmezőben tárolt érték a megfelelő típusú és az értékadás előtt a célmezőben tárolt értéknek megegyező típusú.

Például, ha az x értékadó utasítással y-t akarjuk értékadni, a leggyengébb előfeltétel az lehet, hogy y típusa megegyezik az x típusával. Ha például x egy egész szám, y is egész számnak kell lennie ahhoz, hogy az értékadás érvényes legyen.

* A leggyengébb előfeltétel meghatározásához általában az a legjobb módszer, hogy a parancs végrehajtásához szükséges feltételeket és korlátozásokat figyelembe vesszük.
* Ha például egy parancs végrehajtásához szükséges, hogy az adatok bizonyos formában legyenek eltárolva, akkor az előfeltételnek tartalmaznia kell a megfelelő adattárolási formát.
* Ha a parancs végrehajtásához szükséges, hogy az adatok bizonyos értékeket tartalmazzanak, akkor az előfeltételnek tartalmaznia kell a megfelelő értékeket.
* Ha a parancs végrehajtásához szükséges, hogy az adatok bizonyos módon rendezettek legyenek, akkor az előfeltételnek tartalmaznia kell a megfelelő rendezési feltételeket.
* És így tovább.

## F03-05 Mi egy rutin szerződése az Eiffelben? Hogyan adjuk meg? Hogyan írjuk le a jelentését Hoare-hármassal? Hogyan viszonyul ez az elő- és utófeltételes specifikációkhoz?

A rutin szerződése a paraméterei, a visszatérési értéke, a require (előfeltétel a paraméterekre és a belső állapotra) és az ensure (utófeltétel a visszatérési értékre és belső állapotra). Az require megadja, hogy milyen körülmények kellenek ahhoz, hogy meghívható legyen a rutin. Az ensure meghatározza, hogy mit várunk el a rutintól, milyen utófeltételt fog biztosítani. Bizonyos feltételeket nem lehet megfogalmazni az utófeltételben, például: minden egész számra igaz, hogy …

| require  a > 0; b > 0 ... ensure  Result > 0 |
| --- |

Hoare-hármassal megadva: {REQ} rutin {ENS}. A Hoare-hármassal megadható parciális (ha terminál, akkor az előfeltételből az utófeltételbe jutunk) és teljes helyesség (előző + garantáltan terminál is). A Hoare teljes helyessége párhuzamba állítható a leggyengébb előfeltétel fogalommal.

# **Ellenőrző kérdések - 04 - Objektumelvű programozás**

## F04-01 Hogyan védekezik az Ada nyelv az üres referenciák által okozott problémákkal szemben?

3. előadás 1:08:56-tól

Lehet subtype-ot létrehozni is not null-al, futás idejű ellenőrzést ad. + typeot a fordításkor ellenőrzi, a subtype futási időben, is not null := null - futási idejű hiba + warning fordításkor.

## F04-02 Mik az annotációk a Javában? Mire való a javax.annotation.NonNull annotáció?

3. előadás 1:07:30-tól

In Java, annotations are a way to provide metadata about a program, which can be used by the compiler, build tools, or other tools to process the code. Annotations are similar to comments, but they are designed to be read by the compiler or other tools rather than by humans.

Annotations are written using the "@" symbol, followed by the name of the annotation. They can be placed before a class, method, field, or other elements of a Java program, and they can be used to provide information about how the element should be treated by the compiler or other tools.

javax.annotation.NonNull is a Java annotation that indicates that a reference should not be null when it is used. This annotation can be used to help prevent null pointer exceptions in Java programs by providing a way to specify that a reference must not be null at certain points in the code.

If a null reference is passed to either of these methods, the program will throw a NullPointerException.

## F04-03 Mi a *Maybe* (avagy *Option* avagy *Optional*) típus? Hogyan segít az üres referenciák okozta problémák leküzdésében? Például: hogyan emelhető be egy számítás az *Option* monádba a Scalában?

4. előadás 00:10:44-től

visszatérési érték nem biztos, hogy van egy függvényben, beszédesebb hibákat eredményez

addig hajtjuk végre a számítást míg nem kapunk null értéket, getOrElse()-vel csomagoljuk ki

Monadic style is based on the idea of "chaining" together small, independent computations to create larger, more complex programs. This is done by using monadic functions, which are functions that take a monad as an input and return a monad as an output.

## F04-04 Mit értünk *aliasing* alatt? Mire jó? Miért okoz problémákat? Hogyan segít a szemétgyűjtés?

In object-oriented programming (OOP), aliasing refers to the ability to have multiple references to the same object. This can happen when an object is assigned to more than one variable, or when an object is passed as an argument to a function and a reference to it is stored in a local variable.

Aliasing can have unintended consequences, because changes made to the object through one reference will be visible through all other references to the same object. This can make it difficult to understand the behavior of a program, and it can lead to errors if the program relies on the object having a specific state.

To avoid these problems, it is generally recommended to minimize aliasing in OOP code. This can be achieved by using copy constructors or copy assignment operators to create new objects rather than assigning or passing references to existing objects, and by using immutable objects whenever possible.  
  
In object-oriented programming (OOP), garbage collection is a mechanism that automatically reclaims the memory used by objects that are no longer needed. This can help avoid problems with aliasing, because it ensures that objects are not left hanging around in memory after they are no longer being used.

## F04-05 Mik az attached és a detachable típusok az Eiffelben? Hogyan jelöljük őket? Mi a viszonyuk egymáshoz?

acc: attached ACCOUNT - nem lehet void, elhagyható (vizsgán nem)

maybe\_acc: detachable ACCOUNT -lehet Void

expanded csak attached lehet

detachable általánosabb mint az attached

## F04-06 Hogyan kell egy *detachable* típusú kifejezést értékül adni egy *attached* változónak?

Kidolgozza:

ellenőrizni!

1. if változat: megkeresni: 05 előadásban azt mondja múlt héten, tehát 04-ben
2. check változat

| if attached detached\_var as safe\_var then  attached\_var := safe\_var  end |
| --- |
|  |

| check attached values as attached\_value then attached\_variable := attached\_value |
| --- |

Ha a check nem sikerül, futás idejű kivétel lesz, ez nem elágazás (olyan mint egy cast). Itt előfeltételel és típusinvariánssal lehet biztosítani, hogy a check sose okozzon kivételt: **invariant has\_value implies attached value … require has\_value**

## F04-07 Mit értünk dinamikus szemantikai hiba alatt? Miben más ez, mint a statikus szemantikai hiba?

Dynamic semantic errors are errors that occur during the execution of a program, while static semantic errors are errors that are detected by the compiler during the compilation process.

Dynamic semantic errors are often caused by problems with the runtime environment, such as the values of variables or the state of the program at the time the error occurs. These errors can be difficult to detect and fix, because they may not be immediately obvious from the code and may only occur under certain circumstances.

Static semantic errors, on the other hand, are errors that are related to the meaning of the code itself, rather than the runtime environment. These errors are detected by the compiler when it checks the code for correctness before it is executed. Examples of static semantic errors include syntax errors, type errors, and undeclared variables.

## ✓ F04-08 Mik a dokumentációs megjegyzések? Mire valók? Hogyan írunk ilyet Eiffelben?

Speciális megjegyzések, amelyek a kód dokumentálására szolgálnak, nem csak a kód olvasója számára, hanem az szoftver eszközök számára is elérhetőek. Fejlesztés során segéd információkat adhatnak a hívott kódról, ha annak vannak dokumentációs megjegyzései, továbbá automatikus kód dokumentációt lehet belőlük készíteni. Eiffelben nem speciális karaktersorozattal jelöljük, hanem a megfelelő helyre kell írni őket, például rutin esetén a név(paraméterlista) utáni sorba kell írni. Eiffel Studió a fájlok elején “note, description… “ dok. megj. szerű jelölést használ.

## F04-09 Mire való a *safe navigation operator* és a *null-coalescing operator* (időnként *Elvis-operator*)? Mutasd be példákon!

val a = "Kotlin"

val b: String? = null

println(b?.length)

Safe calls are useful in chains. For example, Bob is an employee who may be assigned to a department (or not). That department may in turn have another employee as a department head. To obtain the name of Bob's department head (if there is one), you write the following:

bob?.department?.head?.name

Such a chain returns null if any of the properties in it is null.

When you have a nullable reference, b, you can say "if b is not null, use it, otherwise use some non-null value":

val l: Int = if (b != null) b.length else -1

Instead of writing the complete if expression, you can also express this with the Elvis operator ?::

val l = b?.length ?: -1

If the expression to the left of ?: is not null, the Elvis operator returns it, otherwise it returns the expression to the right. Note that the expression on the right-hand side is evaluated only if the left-hand side is null.

## ✓ F04-10 Mi a különbség a közönséges és az *expanded* osztályok között?

A közönséges osztályok referencián keresztül érhetőek el, a heap-en tárolódnak. Az expanded osztályok nem referencián keresztül, hanem közvetlenül elérhetőek, a stack-en jönnek létre. Az osztály definíciójában lehet megadni az “expanded class …” kifejezéssel. Hatékonyság szempontjából érdemes használni bizonyos esetekben. Mivel az expandált típusok esetén nincs referencia, ezért a referencián keresztüli polimorfizmust nem lehet rájuk alkalmazni.

|  | Közönséges class | expanded class |
| --- | --- | --- |
| Helye | heap | stack |
| Hozzáférés | indirekt hozzáférés | közvetlen hozzáférés |
| ? | Dinamikus memória | ? |
| Referencián keresztüli polimorfizmus | van | nincs |
| Aliasing | van | nincs |
| Szemétgyűjtés | kell hozzá szemétgyűjtés | nem kell |
| Üres referencia | lehet | nincs referencia, így nem lehet üres |
| Öröklődés |  |  |

## F04-11 Sorold fel és illusztráld példákon, milyen fajta feature-ök alkotják az osztály definíciókat Eiffelben! Melyik miben tér el a másiktól? Mik azok a *lekérdezés*ek?

Kidolgozza:

Feature-ök:

attribútum

konstans

függvény

Eljárás: hasonlít a függvényhez, de nincs visszatérési értéke. Az eljárásokon keresztül lehet kívülről módosítani az objektum állapotát. Speciális esete a “creation procedure”, ez hívható normál eljárásként is, de az objektum inicializálására is szolgálhat.

| feature myproc do print ("myproc called") end ... local p: PROCEDURE\_EXAMPLE do create p; p.myproc end |
| --- |

Eltérnek:

Lekérdezés: az Eiffel manualban Query-ként szerepel, logikai csoportosítása Eiffel feature-öknek. Arra valók, hogy információkat tudjunk meg egy objektumról. Ide tartoznak az attribútumok, konstansok, és a függvények (az eljárások nem!).

## F04-12 Adj áttekintést arról, milyen lehetőségek vannak az üres referenciák által okozott problémák kezelésére a különböző nyelvekben!

Kidolgozza:

## ✓ F04-13 Mik az *üres* (Eiffelben: *Void*) referenciák? Milyen problémát okoznak?

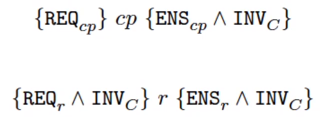
Ha egy változó típusa egy referencia típus, akkor a változó egy referencián keresztül mutat az objektumra. Ha a változó deklarálva lett, de az objektum még nem lett inicializálva, akkor egy üres referenciánk van. Az Eiffel ez Void referenciának hívja. A probléma ezzel, hogy futás időben derül ki, hogy a referencia üres, a program kivétel állapotba kerül. Az Eiffel statikus ellenőrzéssel védekezik a Void referenciák ellen, a változók lehetnek “attached” (not null) (ez a default) és “detached” (maybe null). Az attached változók esetén a típusrendszer garantálja, hogy nem vehetnek fel null értéket. A detached változókat futás idejű ellenőrzésekkel kell védeni az üres reference dereferálása ellen.

Ellenőrzés: if attached myvariable as myattachedvariable then

## ✓ F04-14 Mi az osztályinvariáns? Magyarázd el szóban is és formálisan megfogalmazott szerződésekkel is a szerepét! Mi a szintaxisa az Eiffelben?

Az osztálydefiníció része, tulajdonképpen egy típusinvariáns. Az objektumoknak meg kell őrizni ezeket a tulajdonságokat, de nem kell mindig fennállniuk. A objektum létrejötte után, műveletek előtt és műveletek után kell fennállnia. Művelet végrehajtása közben “elromolhat”.

| class INVARIANT\_EXAMPLE invariant  **name\_of\_invariant**: True -- igaz  **name\_of\_invariant2**: 1 = 1 -- igaz  **name\_of\_invariant3**: 2 = 3 -- hamis lesz |
| --- |

Formálisan: cp=creation procedure, C:osztály, r: rutin: tehát a creation procedure után igaz az invariáns, rutin előtt és után igaz az invariáns  


## F04-15 Mit értünk statikus típusozás alatt? Mutass példát ezt követő, illetve nem követő nyelvekre! Térj ki a típusellenőrzés mellett a típuskikövetkeztetésre is!

Static typing is a type of type checking in which the type of a variable is checked at compile-time, rather than at runtime. In a statically-typed language, variables must be declared with a specific type, and the type of the value assigned to the variable must match the declared type.

követi: Java, C++, C#

nem követi: Python, Haskell

Type checking is the process of verifying that the type of a value or expression in a program is correct. This can be done at various stages of the development process, including at compile-time (static type checking) or at runtime (dynamic type checking).

In dynamically-typed languages, type checking is done at runtime by the interpreter, which checks the types of values and expressions as the code is executed. If the types do not match, the interpreter will throw a runtime error.

Type inference is the process of automatically determining the type of a value or expression in a program based on context. It is a feature of many programming languages that allows developers to omit explicit type declarations in certain cases, and it can help make code more concise and readable.

Type inference is typically used in statically-typed languages, which require variables and expressions to have a specific type. In these languages, type inference algorithms are used to analyze the context in which a value is used and to determine the most likely type for that value.

## F04-16 Mit nevezünk szemétgyűjtésnek (garbage collection)?

When an object is no longer needed, the garbage collector will automatically deallocate the memory that was used by that object, making it available for use by other parts of the program. This helps ensure that the program has enough memory available to run efficiently, and it helps prevent the program from consuming too much memory over time.

# **Ellenőrző kérdések - 05 - Procedurális programozás**

## F05-01 Mik a mohó és a lusta operátorok? Mutass példát Eiffelben!

4. EA 50:46

Kidolgozza: and - and then, or - or else

mohó - mindig azonnal kiértékelődik

lusta - csak addig/akkor értékelődik ki, amikor szükség van rá

## F05-02 Mi a különbség az imperatív és a funkcionális stílusú rutin között? Mutasd be a *divide\_by* és a *divided\_by* példáján!

4. EA 40:20

imperatív: az objektumon végzi a műveletet (eljárás):

divide\_by(other: attached FRACTION)

funkcionális: egy új objektumot ad vissza, az lesz az eredmény:

divided\_by(other:attached FRACTION): attached FRACTION

## ✓ F05-03 Hogyan fejezhetjük ki a nemváltozás tényét Eiffelben?

4. EA 1:00:25

Másnéven Frame problem. Az old kulcsszóval elérhető egy adatunk paramétertérbeli értéke, így utófeltételben tudunk vizsgálatot írni az aktuális és a kezdeti érték egyenlőségére (vagy egyenlőtlenségére), ezzel kifejezve a változást / nemváltozást: **ensure frame: adat = old adat**.

Ha sok a paraméter, akkor a tételes felsorolásuk a nemváltozás kifejezéséhez macerás az old használata, továbbá a későbbi módosítás során hibaforrás is lehet, ezért másik megközelítés, hogy nem azt is kifejezhetjük, hogy mi nem változhat, hanem azt, hogy mi az ami változhat és semmi más nem! Ez az **only** kulcsszó: **only balance**, csak a balance értéke változhat meg, semmi más (Eiffel studio ezt nem támogatja, csak az ECMA szabvány része). Ennek egy korábbi szintaktikája: **strip(balance) ~ old strip(balance)**, ami azt fejezi ki, hogy az objektum, a balance-n kívüli értékei és az objektum kezdeti állapotának balance-n kívüli értékei megegyeznek.

## ✓ F05-04 Mire való az old operátor? Hogyan fejezünk ki vele változást és nemváltozást?

4. EA 51:50

Az Eiffelben az old olyan, mint Fóthi tanár úr jelölésében a vessző `. Ezzel hivatkozunk az adatunk korábbi, paramétertér szerinti értékére. Csak az utófeltételben használhatjuk. Az tudjuk vele mondani hogy az adat = old adat, amivel azt fejezzük ki, hogy nem változik meg az értéke, itt az old adat az adat kezdeti (paramétertérbeli) értéke. A változást az adat /= old adat tudjuk kifejezni.

## ✓ F05-05 Hogyan kell operátorokat definiálni az Eiffelben?

A függvények esetén lehet definiálni operátort az “alias” kulcsszó segítségével. A saját operátor lehet infix bináris (egy paramétere van) vagy prefix unáris (nincs paramétere), kivéve az mixfix indexelés operátor [,], aminek több argumentuma is lehet . A meglévő operátorokat is túl lehet terhelni, például a “/”-t lehet definiálni egy tetszőleges classra is.

Bináris operátor

| multiply\_by alias "\*" (with: INTEGER) : attached OVERLOADING do create Result.make(value\*with) end |
| --- |

Unáris operátor

| dobule alias "!" : attached OVERLOADING do create Result.make(value\*2) end |
| --- |

## ✓ F05-06 Milyen operátorok vannak az Eiffel nyelvben?

Az Eiffel programozási nyelvben számos operátor áll rendelkezésre, amelyeket a programokban használhatunk. Az Eiffel operátorokat a következő csoportokba oszthatjuk:

* **Aritmetikai operátorok:** Ezek az operátorok aritmetikai műveleteket végeznek, például összeadás, kivonás, szorzás, osztás és módulo. Például az +, -, \*, / és \\ operátorok aritmetikai operátorok.
* **Relációs operátorok:** Ezek az operátorok relációkat (összehasonlításokat) végeznek. Például az =, /=, <, <=, > és >= operátorok relációs operátorok.
* **Logikai operátorok:** Ezek az operátorok logikai műveleteket végeznek, például AND, OR, NOT. Például az and , or és not operátorok logikai operátorok.
* **Bitsorozat-operátorok:** Ezek az operátorok bitenkénti műveleteket végeznek, például AND, OR, XOR, NOT. Például az and, or, xor és not operátorok bitsorozat-operátorok.
* **Rendezési operátorok:** Ezek az operátorok rendezési műveleteket végeznek. Például az *ascending* és *descending* operátorok rendezési operátorok.
* **Indexelési operátorok:** Ezek az operátorok segítségével lehet elérni egy tömb elemeit vagy egy szöveg karaktereit. Például az [] operátor indexelési operátor.
* **Dereferencia-operátorok:** Ezek az operátorok segítségével lehet elérni egy változó értékét a memóriában. Például az ^ operátor dereferencia-operátor.
* **Kiterjesztési operátorok:** Ezek az operátorok segítségével lehet kiterjeszteni egy osztályt új feature-ökkel. Például az *extend* operátor kiterjesztési operátor.
* **Egyéb operátorok**: navigációs ., intervallum képző operátor .., old operátor

## F05-07 Hogyan működik a paraméterátadás Eiffelben? (2-fajta...)

* Kifejtett tipusú: érték szerinti (call-by-value):
  + denominator: INTEGER attribute Result := 1 end
* Referencia tipusú: megosztás szerinti (call-by-sharing)

A paraméter típusa határozza meg, nem az átadott objektum típusa.

## F05-08 Milyen problémát vet fel az *aliasing* a referencia szerinti paraméterátadásnál? Mutasd be egy példa segítségével!

4. EA 33:51

Csinálunk egy FRACTION törtszám osztályt, és definiáljuk a divide\_by(other: attached FRACTION) eljárását. Ekkor ha f.divide\_by(f) eljáráshívás történik, akkor a numerator := numerator \* other.denominator értékadás elvégzésénél már megváltoztatjuk az other numenatorját is, így a denominatort majd rosszul számoljuk ki, hiszen a Current és az other ugyanazokra az objektumokra hivatkoznak.

# **Ellenőrző kérdések - 06 - Generikus programozás**

## ✓ F06-01 Sorold fel az ARRAY[T] osztály legfontosabb műveleteit! Mi helyettesíti az elavult make *creation procedure*-t?

make\_empty, make\_filled(val:T, min\_index, max\_index ), make(min\_index, max\_index), item, put, lekérdezései: lower, upper, count, capacity

?A make helyett a make\_filled -et ajánlott használni.

## ✓ F06-02 Mit szokás generikus definíciónak nevezni egy programozási nyelvben? Eiffelben milyen generikus definíciót lehet írni?

A generikus definíció az típussal paraméterezhető definíció. Olyan általánosan van megírva, hogy különböző típusokon is lehet használni.  
Eiffelben az osztályok lehetnek generikusak (a rutinok nem!).

## ✓ F06-03 Mi az Eiffel-terminológia jelentése az osztály, generikus osztály és a típus kapcsán?

4. EA 1:24:00

Osztály, amit definiálunk, lehet generikus. Generikus osztály: típusparaméterrel rendelkező osztály. Típus: amivel típusozunk, lehet osztály, lehet felparaméterezett generikus osztály, paraméterezetlen generikus osztály nem!

## ✓ F06-04 Mutasd be egy példán, hogyan kell Eiffelben egy generikus osztályt definiálni, illetve használni!

A következő osztály egy generikus értéket tud tárolni, azt le lehet belőle kérdezni:

| class GENERICCLASS[T] create make feature  value:T  make(v:T) do value := v end end ... local p: GENERICCLASS[INTEGER] do create p.make(5); print (p.value); end |
| --- |

# **Ellenőrző kérdések - 07 - Öröklődés**

## ✓ F07-01 Mondd ki a Liskov-féle helyettesítési elvet!

A Liskov-féle viselkedési altípus fogalma meghatározza az objektumok helyettesíthetőségének fogalmát; vagyis ha S a T altípusa, akkor egy programban a T típusú objektumok helyettesíthetők S típusú objektumokkal anélkül, hogy a program kívánatos tulajdonságai (pl. a helyesség) megváltoznának.

## ✓ F07-02 Hogyan hívjuk meg a régi törzset egy felüldefiniált feature-ben?

A Precursor kulcsszó segítségével.

## ✓ F07-03 Mit értünk a S.O.L.I.D. elvek alatt?

Single Responsibility Principle - Egyetlen felelősség elve: Egy osztály vagy modul egy, és csak egy felelősséggel rendelkezzen (azaz: egy oka legyen a változásra).

Open/Closed Principle - Nyílt/zárt elv: Egy osztály vagy modul legyen nyílt a kiterjesztésre, de zárt a módosításra.

Liskov substitution principle - Liskov helyettesítési elv: Minden osztály legyen helyettesíthető a leszármazott osztályával anélkül, hogy a program helyes működése megváltozna.

Interface segregation principle - Interfész elválasztási elv: Az interfészek (kapcsolódási felületek) szétválasztásának elve: egyetlen kliens se legyen rákényszerítve arra, hogy olyan eljárásoktól függjön, amelyeket nem is használ.

Dependency inversion principle - Függőség megfordítási elv: A magas szintű modulok ne függjenek az alacsony szintű moduloktól. Mindkettő absztrakcióktól függjön.

## ✓ F07-04 Hogyan adjuk meg a dinamikus típust egy create utasításban, illetve egy create kifejezésben?

| local  a: ACCOUNT  b: POINT do  create a.make(19878238)  create {SAVINGS\_ACCOUNT} a.make(19878238) -- utasítás  !SAVINGS\_ACCOUNT! a.make(19878238) -- Régi szintaxis  b := create {POINT} -- kifejezés  b := create {POINT}.default\_create |
| --- |

## ✓ F07-05 Mi a default\_create?

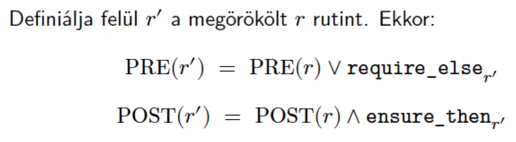
Ha egy osztály nem ad meg creation proceduret, akkor az ANY osztályból örökölt üres törzsű az ANY-ben default\_create nevű lesz a creation procedureje. Gyakorlatilag egy üres alapértelmezett konstruktor.

## ✓ F07-06 Hogyan fejezzük ki a dinamikus típusellenőrzést az Eiffelben?

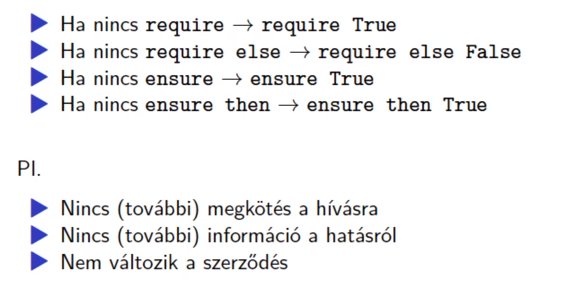
| local  a: attached ACCOUNT do  create {SAVINGS\_ACCOUNT} a.make(19878238)  a.deposit(1000)  if attached {SAVINGS\_ACCOUNT} a as sa then  sa.set\_interest(10)  end |
| --- |

## ✓ F07-07 Hogyan módosítható egy rutin szerződése a felüldefiniálás során? Mi ennek a formája?

Felüldefiniálás során egy rutin szerződését úgy lehet módosítani, hogy az a kliensek számára ne “rontsa el” a szerződést, tehát az előfeltétel gyengíthető, az utófeltétel szigorítható. A szerződés módosításhoz használt kulcsszavak: require else, ensure then.



## ✓ F07-08 Mit jelent az, ha egy osztályba nem írunk invariáns klózt, illetve, ha egy feature-be nem írunk elő- és utófeltételt?



## F07-09 Hogyan fejezzük ki az öröklődést Eiffelben? Mi az alapvetés?

Inherit ANY

Alapvetés, hogy a bázistípus szerződését meg kell őrizni: az előfeltétel gyengíthető, az utófeltétel szigorítható. Többszörös öröklődés támogatott.

## ✓ F07-10 Hogyan fejezzük ki a feature-ök láthatóságát Eiffelben? Mik azok a nyilvános és titkos feature-ök?

Az Eiffel szelektív láthatóságot alkalmaz. Az adott feature klóz után kapcsos zárójelben adhatjuk meg az osztályt, az osztály és annak leszármazottai érhetik el az adott featureöket. A **feature** és a **feature {ANY}** mindenki számára elérhető, a **feature {NONE}** (régi szintaxisban **feature {}**) pedig senkinek nem elérhető. A **feature {MYCLASS}** a MYCLASS és leszármazottai számára elérhető.

## ✓ F07-11 Hogyan kell felüldefiniálni egy feature-t az Eiffelben?

Az öröklődés megadásánál a renamehez hasonlóan használható a redefine kulcsszó, amivel megadhatjuk, hogy mely featureöket akarjuk újra definiálni.

| class CIRCLE  inherit ANY  redefine default\_create  end  feature  x, y, r: REAL  default\_create do r := 1.0 end end |
| --- |

## F07-12 Mire való az átnevezés az Eiffelben? Mutass példát a használatára!

FOLYTATNI

Az öröklődés során a szülő osztály featurejeit át lehet nevezni másra. Ezzel többszörös öröklődés esetén elkerülhetőek a névütközések,

| class SAVINGS\_ACCOUNT inherit ACCOUNT  rename make as make\_account  end feature  make ... |
| --- |

Ebben a példában az accountból megörökölt make-et átnevezzük make\_accountra és saját make-et definiálunk.

## F07-13 Mi a kapcsolat az öröklődés és a creation procedure-ök között?

Minden osztálynak kell hogy legyen egy creation procedure-ja. Ha nem hozunk létre egyett akkor az ANY osztály default\_create üres törzsű creation procedureját fogja megörökölni. Ez azért lehetséges mert minden osztály az ANY osztály örököse. Ezt a default\_create creation procedure felül tudjuk irni és át is tudjuk nevezni.

Meghivása: az alábbi kettő ugyanaz, nem kell kiirni a default\_create-t

Create {POINT}

create {POINT}.default\_create

## ✓ F07-14 Mit értünk egy osztály invariánsa alatt? (Vedd figyelembe az öröklődést is!)

Az invariant klóz alatt megadott feltételeknek az osztály creation procedure lefutása után fenn kell álljanak, valamint minden művelet előtt és után is igaznak kell legyenek. Az osztály invariánsa és az összes szülő invariánsa ÉS-el összekapcsolva értendő.

# **Ellenőrző kérdések - 08 - Absztrakt osztályok**

## ✓ F08-01 Mit értünk absztrakt típus alatt az OOP-nyelvekben? Mi a szerepe? Mi ennek a megfelelője az Eiffelben?

Absztrakt típus: nem teljesen definiált típus. Implementálatlan viselkedéseket akarunk bennük elhelyezni. Teljesen absztrakt típus az interface, amiben nincs viselkedés definiálva, az OOP nyelvekben az absztrakt osztályokban lehetőség van implementáció megadására. Ez tulajdonképpen az OOP megfelelője a magasabb rendű függvényeknek, itt függvénnyel paraméterezünk. Eiffel a **deferred** kulcsszót használja az absztrakt típusokra és featureökre is.

## ✓ F08-02 Mit értünk az alatt, hogy egy absztrakt típus megvalósítása? Mutass példát az Eiffelben (is)! Mit lehet tudni ez ügyben az adattagokról (attribútumokról)?

Az Eiffel terminológiában ez a “making effective”. Az absztrakt típus megvalósítása, amikor tényleges implementációt adunk meg az absztrakt tagokhoz. Ha az leszármazott osztályban nem valósítjuk meg az összes deferred featuret, akkor az az osztály is deferred kell, legyen. Egy absztrakt feature lehet, hogy függvénynek tűnően deklarálunk az ősosztályban, a megvalósító osztályban azonban ezt megvalósíthatjuk akár egy attribútummal is.

| deferred class ANIMAL feature  talk: STRING deferred end and  class CAT  inherit ANIMAL feature  talk: STRING attribute Result := "Miaow" end end |
| --- |

C# példa

| abstract class Person {  public abstract string Greet(); }  class Student : Person {  public override string Greet()  {  return "Hello!";  } } |
| --- |

## ✓ F08-03 Mit jelent a kovariáns visszatérési típus? Mutass rá példát! Miért jó?

Örökléskor (specializáláskor) egy felüldefiniált függvény visszatérési típusát szűkítjük (specializáljuk). Ez olyan, mint az utófeltétel szigorítása.

| deferred class FOOD end class MILK inherit FOOD  ...  class ANIMAL  feature prefers: detachable FOOD do end class CAT inherit ANIMAL redefine prefers end feature  prefers: attached MILK do create Result end ... |
| --- |

Itt a CAT prefers-ben a FOOD speciálisabb típusát, a MILK-et adjuk vissza, ez a kovariáns visszatérési érték (és a detachable-ből attached lett, ami szintén kovariancia).

## ✓ F08-04 Lehet deferred feature-nek szerződése? Magyarázd el!

Igen. require és ensuret lehet definiálni, attól függetlenül, hogy a törzs deferred. Pont erre való a deferred feature, hogy ismerjük az elvárt viselkedését a featurenek (elő és utófeltétel), de nem akarunk hozzá implementációt rendelni. A szerződést a felüldefiniálásnál ismert módon lehet a megvalósítás során módosítani.

## ✓ F08-05 Mit jelent a frozen kulcsszó egy művelet deklarációjában? És egy osztály deklarációjában?

Frozen: nem lehet felüldefiniálni. A featureök esetén nem lehet felüldefiniálni az adott featuret. Osztály esetén nem lehet belőle leszármaztatni, az az lezárjuk az öröklődést. Frozen nem lehet deferred (se class se feature).

## ✓ F08-06 Mi a módja az Eiffelben annak, hogy egy modul használ egy másik modult?

Az Eiffelben minden OOP-vel van megoldva, így a más modulok használata is, öröklődést kell használni. Például a print művelet is így érhető el, amit az ANY-ből öröklünk.

## ✓ F08-07 Mik a once-rutinok az Eiffelben? Hogyan működnek? Milyen veszéllyel járnak?

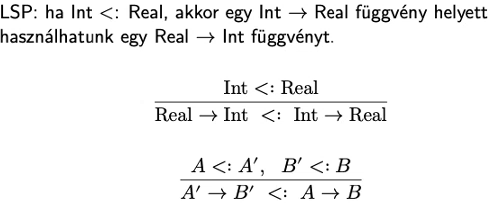
Olyan függvény, amelynek a visszatérési értékét az első híváskor történő kiszámítás után eltárolódik és a további hívások során a már korábban kiszámított érték kerül visszaadásra. Ez kb a lazy initialization megfelelője, nyelvi szintre emelve. A függvény törzsét nem do-val adjuk meg, hanem once -al. Probléma a paraméterek használatával adódhat, amikor X paraméterrel meghívjuk a függvényt, ezzel kiszámolódik az értéke, majd Y paraméterrel hívva is az X szerinti értéket kapjuk vissza.

# **Ellenőrző kérdések - 09 - Variancia**

## ✓ F09-01 Milyen problémát vet fel a kovariáns paraméter?

Matematikailag helytelen (kontravarianciát várnánk el), sérti az LSP-t, de gyakorlatban hasznos lehet, technikailag problémás, csökken vele a statikus típusbiztonság. Könnyű hibázni, amikor a függvények szerződésükben csak a paraméterük altípusában térnek el. A hiányában nehezebb jól megvalósítani például az equals típusú eljárásokat: sok boilerplate kód kell és könnyű összetéveszteni a felüldefiniálás-túlterhelés kérdés és ezután a hívási kontextustól függően rossz rutint hívni.

## ✓ F09-02 Milyen alakúak, milyen tulajdonságúak az A -> B függvények típusának altípusai?

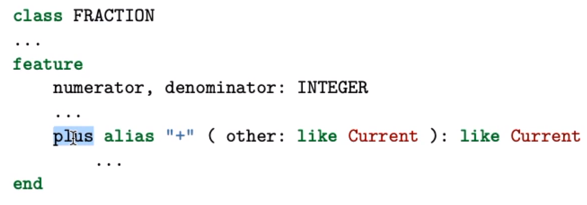


kontravariáns paramétertípus

kovariáns visszatérési típus

## ✓ F09-03 Mit értünk *binary method* alatt? Mutass példát is rá!

Két azonos típusú paraméterrel rendelkező műveletek. Például az aritmetikai műveletek.



## ✓ F09-04 Mit jelent a *kovariáns visszatérési érték* (*covariant return*) kifejezés? Mutass rá példát!

Örökléskor (specializáláskor) egy felüldefiniált függvény visszatérési típusát szűkítjük (specializáljuk). Ez olyan, mint az utófeltétel szigorítása.

| deferred class FOOD end class MILK inherit FOOD  ...  class ANIMAL  feature prefers: detachable FOOD do end class CAT inherit ANIMAL redefine prefers end feature  prefers: attached MILK do create Result end ... |
| --- |

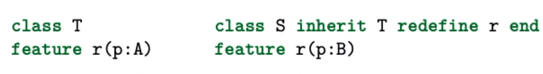
Itt a CAT prefers-ben a FOOD speciálisabb típusát, a MILK-et adjuk vissza, ez a kovariáns visszatérési érték (és a detachable-ből attached lett, ami szintén kovariancia).

## ✓ F09-05 Magyarázd el, mit jelent a *kapcsolt típus* (*anchored type*) az Eiffelben! Mire jó? Mi az előnye?

A like kulcsszóval tudjuk használni. A típus definíciójában nem a konkrét típust adjuk meg, hanem azt mondjuk, hogy az a típus mint az aktuális objektum. Előnye például, hogy a kovariánst visszatérési típus megvalósításához megspóroljuk az újradefiniálást (twin feature), adaptív a típus.

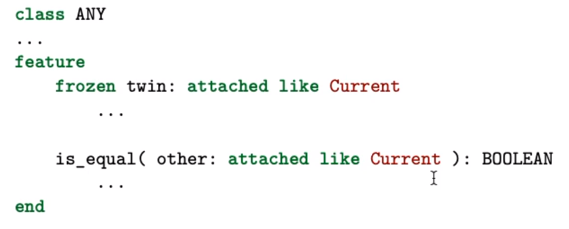
## ✓ F09-06 Mit jelent a *kontravariáns típus* kifejezés? Mutass példát pszeudokóddal!

Specializáció során általánosabb lesz. A szülő osztályban használt T típus helyett a gyerek osztályban Q típust használok, ahol T a Q-nak altípusa. Ez nem sérti az LSP-t, a programozási nyelvek általában nem támogatják. Függvények általában kontravariánsak a paraméter típusukban, és kovariánsak az eredmény típusukban.



## ✓ F09-07 Mit jelent a kovariáns típusú paraméter? Mutass példát, amely ennek az előnyét tárja fel!

Leszármazott típusban speciálisabb paramétert várunk. Matematikailag, LSP szerint helytelen. Az Eiffelben rutinok paraméterei és a generikus típusok paraméterei is lehetnek kovariánsak.



Az is\_equal kovariáns paraméterrel rendelkezik és ezzel elkerüli a más programozási nyelvekben ismert futásidejű típusellenőrzés szükségességét az egyenlőségvizsgálatoknál.

## ✓ F09-08 Mit jelent polymorphic CAT-call kifejezés? Miért érdekes ez a fogalom?

Changed Availability or Type. Ha leszármaztatásnál örökölt featurenek csökken a láthatósága, vagy mező vagy rutin paraméternek szűkítem a típusát. Paraméter szűkítés például, ha egy altípusú objektumot bázis típusú referencián keresztül érünk el, de az altípusban egy paraméterének a típusát kovariáns módot módosítottuk, viszont a referencián keresztül az eredeti típus szerinti paramétert adjuk át neki, ekkor futás időben derül ki, hogy nem megfelelő típust kap az objektum.

CAT <: ANIMAL, MILK <: FOOD, ANIMAL.EAT(FOOT), CAT.EAT(MILK)

AnimalVariable = Cat  
AnimalVariable.Eat(Food) - futás idejű hiba

## ✓ F09-09 Miben hasonlít a függvénytípusok altípusaira vonatkozó szabály az Eiffel feature-ök szerződéseinek öröklődéssel kapcsolatos mechanizmusára?

A függvénytípus paramétere és visszatérési értéke áll párhuzamban a szerződés elő és utófeltételével. Az altípusban az előfeltétel gyengíthető, az utófeltétel szigorítható (a függvénytípusban a paraméter bővíthető, a visszatérés szűkíthető).

## ✓ F09-10 Hogyan néz ki az Eiffelben az ANY osztály klónozásra bevezetett twin művelete?

| frozen twin: attached like Current do [magic shallow copy] end |
| --- |

# **Ellenőrző kérdések - 10 - Egyenlőségvizsgálat**

## ✓ F10-01 Mit jelent a = és a ~ operátor az Eiffelben?

Az = referencia típusokon referencia egyenlőséget vizsgál, kifejtett típusokon tartalmi egyenlőséget, a ~ minden esetben tartalmi egyenlőséget vizsgál.

## F10-02 Milyen tulajdonságokkal kell rendelkezzen egy ekvivalenciareláció, és milyenekkel egy parciális rendezés?

Ekvivalenciareláció: RST, Reflexive, Symmetric, Transitive  
Parciális rendezés: RAT, Reflexive, Asymetric, Transitive

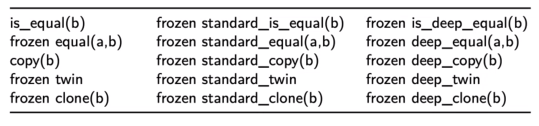
## ✓ F10-03 Hogyan definiálhatunk saját logikájú (custom) tartalmi összehasonlítást egy Eiffel osztályban?

Az is\_equal rutin felüldefiniálásával.

## ✓ F10-04 Mi a különbség a referencia-, illetve a kifejtett típusokon definiált egyenlőségvizsgálatok között?

A referencia típusokon tudunk referencia és tartalmi egyenlőséget is vizsgálni, a kifejtett típusokon csak tartalmi egyenlőséget.

## ✓ F10-05 Hogyan függnek össze, hogyan alkotnak rendszert az egyenlőségvizsgálatra és az objektummásolásra szolgáló *feature*-ök az Eiffelben?



Előre definiált rendszer, amiben van shallow (standard), deep, custom, a fejlesztő a customot tudja bővítésre felhasználni, a shallow és deep frozen. A standard\_copy a használata után a standard\_equal fog fennállni.

## ✓ F10-06 Hogyan viszonyulnak egymáshoz a standard\_equal, az equal és a deep\_equals relációk?

Mindegyik egyenlőségvizsgálatra való, más szemantikával.  
standard\_equal a sekély egyenlőségvizsgálat, objektumok mezőinek =-vel történő összehasonlítása, nem felüldefiniálható,   
equal, bővíthető egyenlőségvizsgálat, alapból standard\_equal  
deep\_equals a mély (rekurzív) egyenlőségvizsgálat, objektumok mezőinek rekurzív tartalmi egyenlőségvizsgálata, nem felüldefiniálható.

## ✓ F10-07 Hogyan függ az equal az is\_equal-tól?

Az equal implementációjában meghív(hat)ja az első paraméter objektum is\_equal rutinját. Az equal-t mint egy globális statikus függvényt használjuk, az is\_equal-t pedig instance methodként.

## ✓ F10-08 Mi a különbség az *object identity* és az *object equality* között?

Az object identity a referenciák egyenlőségét jelenti és csak referencia típusokon értelmezhető. Az object equality az objektumok tartalmi egyenlőségét jelenti és több típusa van: shallow, deep, custom.

## ✓ F10-09 Mi az is\_equal szignatúrája az Eiffel ANY osztályában?

is\_equal( other: attached like Current ): BOOLEAN

## ✓ F10-10 Mi a szerződése az is\_equal-nak? Mi az alapértelmezett implementációja?

A paraméter nem lehet Void, a típusa meg kell egyezzen a Current-tel. Az alapértelmezett implementáció megegyezik a standard\_is\_equal-al

## ✓ F10-11 Mit illik tudni a Java egyenlőségvizsgálatot végző equals metódusának paramétertípusáról? Hogyan kapcsolódik ez a felüldefiniálás és a túlterhelés kérdéséhez?

Javaban lehet null Eiffelben nem, Mivel az object-ből öröklődik és a Java az öröklés mentén a paramétertípusokra invarianciát követel meg, ezért object-nek kell lennie a paraméternek az örökölt osztályokban is. Ha módosítjuk a paraméter típusát valami másra, akkor nem felüldefiniáljuk az equals hanem túlterheljük, amivel könnyen rossz függvényhívást lehet elvégezni (bug).

## ✓ F10-12 Hogyan viszonyulhat az altípusosság az egyenlőségvizsgálathoz? Mi a két fő lehetőség?

Az intuíció azt mondja, hogy az egyenlőségvizsgálathoz azonos típusok kellenek. Az altípusossággal a típusok nem egyeznek, ezért koncepcionálisan nem illeszkedik jól egymáshoz ez a két kérdéskör. Az egyenlőségvizsgálatnál van monomorf (pontos típusegyezés) és polymorf (altípus egyezés) megközelítés. A gyakorlatban nem mindig szükséges a teljes típus egyezés, esetenként megfelelő az altípusosság is, sőt, van amikor teljesen különböző típusokon is értelmezhetjük az egyenlőséget.

## ✓ F10-13 Mi a különbség a sekély és a mély összehasonlítás között?

A sekély összehasonlítás az objektum referencia típusú mezőinek referencia szerinti, kifejtett típusú mezőinek tartalmi összehasonlítása. A mély összehasonlítás minden mezőnek a tartalmi összehasonlítása.

## ✓ F10-14 Mit értünk az alatt, hogy *öröklődés kiváltása kompozícióval*?

A program modellezésében nem minden koncepciót kell öröklődéssel megvalósítani, használhatjuk helyette a kompozíció technikát is. Gyakorlatban azt jelenti, hogy ha használni akarjuk a kódját egy osztálynak, akkor nem örökléssel osztjuk meg a kódot, hanem egy adattagon keresztül használjuk az adott osztály kódját.

# **Ellenőrző kérdések - 11 - Típusmegfelelőség**

## ✓ F11-01 A strukturális típusosságnál miben tér el az értékekre, illetve a változókra vonatkozó altípusfogalom? Azaz mi a fő különbség a funkcionális és az imperatív paradigma között ebben a tekintetben?

A strukturális tipizálás szempontjából az érték és a változó közötti fő különbség az, hogy az érték egy rögzített, megváltoztathatatlan adattípus, míg a változó egy változó értékre való hivatkozás, amely idővel változhat.

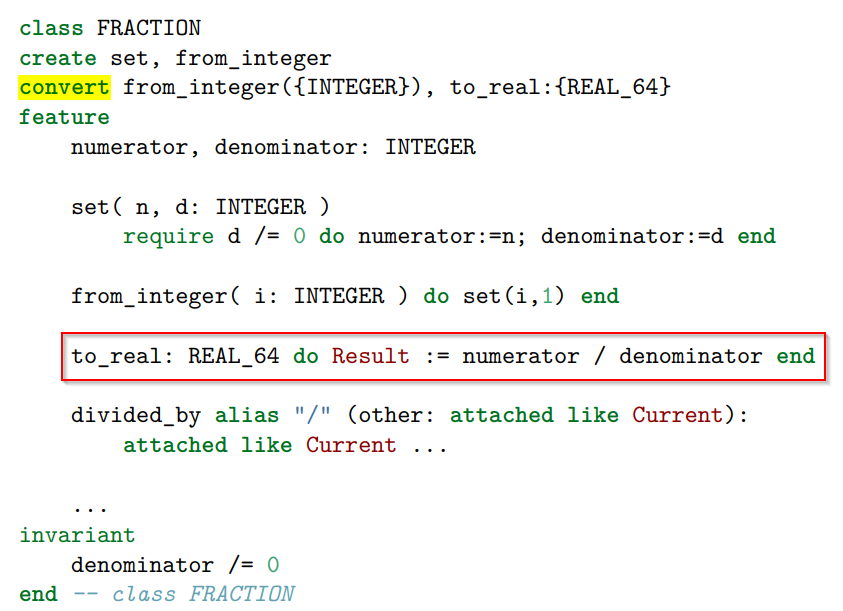
A funkcionális programozási paradigmában az értékeket részesítik előnyben a változókkal szemben, mivel könnyebben értelmezhetőek, és kevésbé hajlamosak a hibákra. Az értékeken működő függvények tiszták, vagyis nincs mellékhatásuk, és mindig ugyanazt a kimenetet adják vissza egy adott bemenethez. Ez megkönnyíti a funkcionális kód tesztelését és hibakeresését, mert biztos lehet benne, hogy egy függvény mindig ugyanazt az eredményt adja ugyanazzal a bemenettel.

Ezzel szemben az imperatív programozási paradigma inkább a program állapotának megváltoztatására összpontosít változók használatával. Az imperatív nyelvek függvényeit gyakran eljárásoknak nevezik, és mellékhatásaik lehetnek, például a globális állapot módosítása vagy a külső erőforrásokkal való interakció. Ez megnehezíti az imperatív kód viselkedésének érvelését, mivel egy függvény kimenete függhet a program állapotától a meghívásakor.

Összességében a fő különbség a funkcionális és az imperatív paradigmák között a szerkezeti tipizálás szempontjából az, hogy a funkcionális nyelvek nagyobb hangsúlyt fektetnek az értékekre, míg az imperatív nyelvek inkább változókra támaszkodnak a program állapotának megváltoztatásához.

## F11-02 Mutass példát a *conversion function* működésére, használatára az Eiffelben!

A convert klózban megadott függvény definiálásával elérhető, hogy pl egy eiffeles értékadás utasítás esetében, amennyiben a kifejezés bal oldalán egy REAL\_64 típus van, jobb oldalán pedig egy FRACTION típus, akkor a FRACTION implicite automatikusan konvertálódjon REAL\_64 típussá



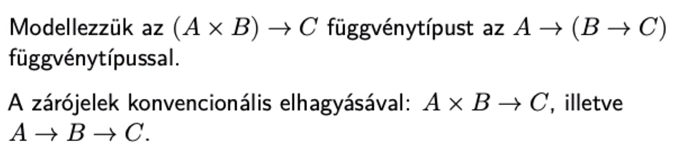
## F11-03 Mutass példát a *conversion procedure* működésére, használatára az Eiffelben!

Convert klózban lehet megadni egy rutint, ami a kapott paraméterek alapján beállítja az objektum állapotát a megfelelő értékekre.



## ✓ F11-04 Mit nevezünk *currying*nek?

A többparaméteres függvényeket visszavezetjük kevesebb paraméteres függvényekre.



## ✓ F11-05 Mit értünk *downcast* alatt? Hogyan fordul elő ilyen az Eiffelben? Mutass példát!

Explicit újraértelmezés. Bázistípusú objektumot belekényszerítünk egy altípus szerint referenciába.

| if attached {POLAR\_CDMPLEX} c as p then  pc := p end |
| --- |

## ✓ F11-06 Mit értünk *duck-typing* alatt? Mutass példát!

A típushelyességet a szerint vizsgálom, hogy rendelkezik-e azokkal az elvárásokkal a szóban forgó objektum, amiket én elvárok tőle. A felhasználás helyén vizsgálni meg, hogy megfelelő formájú-e az objektum. Hasonlít a strukturális altípusságra, de ezt a dinamikus típusozású nyelveknél használjuk.

## ✓ F11-07 Milyen altípus-relációban vehetnek részt a kifejtett (expanded) típusok?

A kifejtett típusból örökléskor nem jön létre altípus reláció, de referencia típusból örökléskor létrejön az altípusosság a kifejtett típusok esetében is, például ezek is az ANY-ből származnak.

## ✓ F11-08 Mi a *nominális* és a *strukturális* típusekvivalencia, illetve altípusosság?

Nominális altípusosság, amikor az altípus relációt a programozó expliciten megadja, például az Eiffelben az inherits. A strukturális altípusosság, amikor két objektum definíciójából következik (azonos adattagok, egyik ugyanaz mint a másik csak bővebb) az altípus reláció. Típusok felépítése szerinti induktív definíció.

## ✓ F11-09 Mit értünk privát öröklődés alatt? Milyen tulajdonságai vannak? Hogyan csinálunk ilyet az Eiffelben?

Olyan öröklődés, amikor a kódot megörökli az alosztály, de a típust nem, azaz nem lesz altípusa a bázistípusnak. Eiffelben az inherit {NONE} segítségével tudunk privát öröklést végrehajtani. Kifejtett típusoknál az öröklés hatásában olyan mint a privát öröklés, nem alakít ki altípus relációt.

## ✓ F11-10 Milyenek a prototípus alapú objektumelvű nyelvek? Mi jellemzi őket?

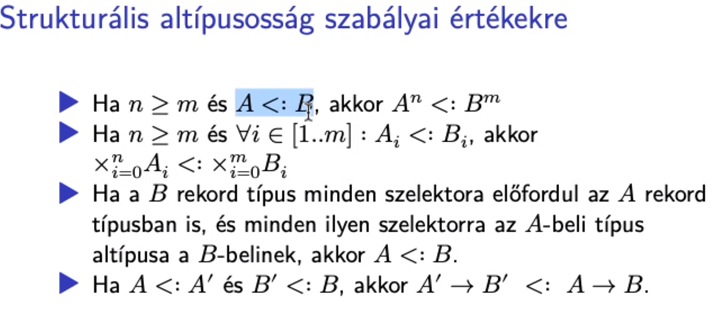
Objektumok klónozásával hozunk létre új objektumokat (nincs osztály). Az objektumok a prototípusukon keresztül dinamikusan bővíthetőek (nem kell öröklődést). Tipikusan az interpretált nyelvek ilyenek.

## ✓ F11-11 Hogyan definiálhatjuk a strukturális altípusosságot az összetett típusú módosítható változókra?

A T típusú változót modellezünk egy setter(T) és egy getter:T segítségével. Ekkor visszavezethetjük arra, amit korábban a rutinoknál tárgyaltunk: paraméter esetén kontravariancia, visszatérés esetén kovariancia. Tehát módosítható adatokra az invariancia a szabály.

Általánosan v:T változó modellezhető T x ( T -> Unit ) párként  
Altípusosság: A x ( A -> Unit ) <: B x ( B -> Unit ) feltétele: A <: B ∧ B <: A (azaz A ≡ B, invariancia)

## ✓ F11-12 Hogyan definiálhatjuk a strukturális altípusosságot az összetett típusú értékekre?



## ✓ F11-13 Mit értünk *upcast* alatt? Hogyan fordul elő ilyen az Eiffelben? Mutass példát!

Implicit újraértelmezés. Egy altípusba tartozó érték bázistípusként használatát jelenti. Ha egy altípus objektumot értékül adok egy bázis típusú változónak.

class Parent, class Child inherit Parent, p:Parent; p:= create {Child}

# **Ellenőrző kérdések - 12 - Többszörös öröklődés**

## ✓ F12-01 Sorolj fel nyelveket, melyek csak egyszeres, illetve többszörös öröklődést is támogatnak!

Csak egyszeres: Simula, Smalltalk

Többszörös interface, egy szülőosztály: C#, Java

Többszörös szülőosztály: C++, Eiffel, CLOS

## ✓ F12-02 Mit értünk *ismételt öröklődés* (repeated inheritance) alatt?

Többszörös öröklődés használatával egy osztály leszármazottja lehet egy másiknak több különböző leszármazási láncon keresztül is. Legegyszerűbb példa a “diamond inheritance”

## ✓ F12-03 Hogyan oldja meg az Eiffel a többszörös öröklődés szokásos problémáit?

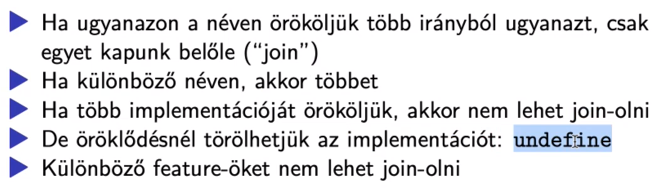
Több lehetőséget is nyújt az Eiffel nyelv a többszörös öröklődés problémáinak megoldására:

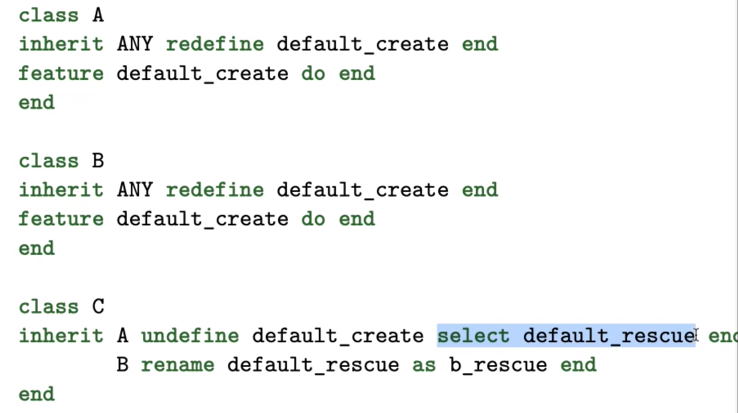
* csak egy osztályból öröklünk publikusan, a többiből privát így csak egyszeres altípus kapcsolat alakul ki
* többszörös öröklés esetén a névütközések feloldására az átnevezést vezeti be (, azzal együtt, hogy a név csak a lokális kontextusban azonosít, globálisan különböző nevei is lehetnek ugyanannak a rutinnak, a dinamikus kötések esetén a compiler a statikus típus szerinti implementációt választja ki)
* ugyanannak a featurenek több irányból öröklése működik, ha az implementáció megegyezik (Eiffel terminológia: join). Ha az implementáció különbözik, akkor használhatjuk az **undefine** kulcsszót valamelyik implementáció eldobására

## ✓ F12-04 Mire jó a select kulcsszó az Eiffelben?

Legyen egy alosztály, ami ismételten örökölt egy bázisosztályból. Ha polimorfizmust szeretnénk használni, és értékül adni egy alosztály példányt egy bázisosztályú változónak, akkor a futási időnek tudnia kell, hogy metódus hívás esetén ténylegesen melyik metódust kell meghívni a több változat közül. A select ezt oldja meg, mert meg tudja határozni, hogy melyik őstől származó metódust használjuk az alosztályban.

## F12-05 Milyen különböző eseteket különböztethetünk meg a többszörös öröklődésben a szülőtípusokhoz való konformitás jegyében? Mutass példákat!





## F12-06 Milyen problémákat okozhat a többszörös öröklődés pl. a C++ nyelvben?

Kidolgozza:

## ✓ F12-07 Mire jó az undefine kulcsszó az Eiffelben?

Többszörös öröklődés esetén ha egy featurehöz különböző implementációk társulnak, akkor implementációkat vethetünk el, ezzel feloldva az ütközést.

# **Ellenőrző kérdések - 13 - Láthatóság**

## ✓ F13-01 Adható-e új érték egy attribútumnak? Mi az *assign procedure*?

Igen, de csak az objektum saját maga írhatja az attribútumait, más objektumok nem. Egy kívülről readonly attribútum írására bevezetünk egy setter eljárást, az attribútum definíciójában pedig assign procedure-ként hivatkozunk a setterre, ekkor kívülről az attribútum írhatónak tűnik, de tulajdonképpen csak a setter eljárás hívódik meg (mint C# property setter).

Ez az attribútumok referenciájára vagy a kifejtett típusok értékére vonatkozik. Referencia típusú attribútumok mutálására lehetőség van a típus definíciója szerint.

## ✓ F13-02 Mit jelent a CAT az Eiffelben? Magyarázd el!

Changed Availability or Type. Öröklődés során előfordulhat, hogy egy leszármazott osztály megváltoztatta egy feature láthatóságát, vagy kovariánsan paraméter típussal írt felül egy procedúrát/függvényt. Ez futási idejű hibákhoz vezet, pl ha a leszármazott osztályon megpróbálunk meghívni egy priváttá tett függvényt. Ez tulajdonképpen az LSP-nek való nem megfelelés.

## ✓ F13-03 Megváltoztatható egy feature láthatósága az öröklődés során? Hogyan? Milyen problémát vet ez fel?

Igen. Bárhogyan, szűkíthető (akár teljesen) és bővíthető is. Az öröklés során új láthatósági szabályt adunk {OSZTÁLY\_NEVE} formában az adott featurenél. A probléma ezzel az, hogy nem felel meg az LSP-nek, így ha B alosztályban csökkentettük egy A bázisosztálybeli C feature láthatóságát, akkor futás időben kivételt okoz (CAT CALL), ha az A referencia mögött lévő B objektumon próbáljuk meghívni a C featuret.

## F14-04 Milyen lehetőségek vannak az Eiffelben a láthatóság megadására? Hol lehet ezeket a lehetőségeket használni, mire lehet megadni a láthatóságot?

Kidolgozza:

## ✓ F14-05 Milyen problémát okoznak az Eiffelben a módosítható (mutable) adattagok az információ elrejtésében?

Eiffelben egy objektum attribútumait csak saját maga változtathatja meg, így az információ elrejtésre nincs a módosíthatóság szempontjából szükség, mint más nyelvekben. Ha az attribútum mögött egy mutable objektum van, annak a belső állapotát a referenciáján keresztül módosítani tudjuk, például egy STRING esetében is. Ezzel kiszökik a belső állapot, ezért fokozott óvatossággal kell bánni az ilyen objektumokkal.

## ✓ F14-06 Mit tekinthetünk mutable, illetve immutable objektumnak?

Immutable objektum, aminek a belső állapota nem tud változni a kezdeti értékbeállítás után, a mutable objektumoknak változhat a belső állapotuk.

## ✓ F14-07 Mik a jellemző láthatósági kategóriák az OOP-nyelvekben? Hogyan feleltethető meg ennek az Eiffel módszere?

Azt mondhatjuk, hogy a klasszikus láthatósági kategóriák az Eiffel által támogatott lehetőségek speciális esetei.

Public, Eiffel: class A feature {ANY} (vagy elhagyva) …

Protected, Eiffel: class B feature {B} …

Private, Eiffel: class B feature {NONE} …

## ✓ F14-08 Mit jelent a *szelektív láthatóság* az Eiffelben? Hogyan viszonyul az altípusossághoz?

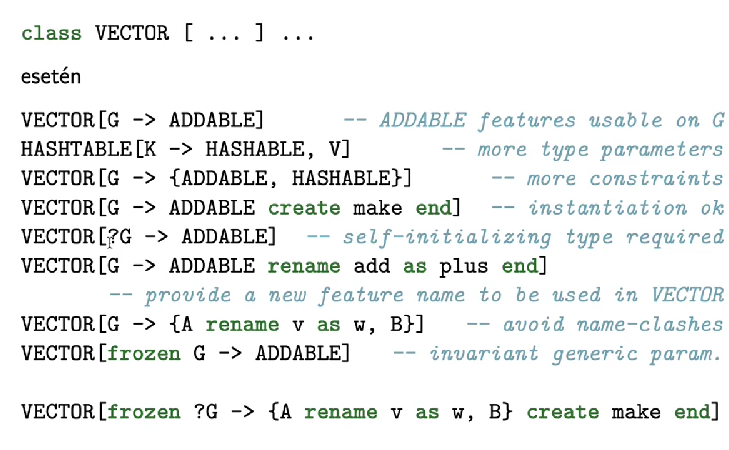
A create, feature, export (más?) klózok esetén fel tudom sorolni azokat a típusokat, amelyek -és azok leszármazottai- elérhetik az adott elemeket. Ez az általános public-protected-private láthatóságokhoz képest finomabb lehetőséget ad a láthatóság szabályozására.

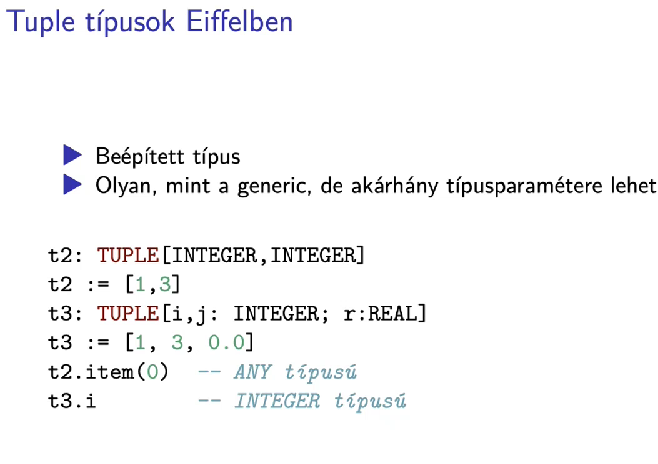
## F14-09 Mit jelent az Eiffelben a titkos feature fogalma? Mennyire titkos?

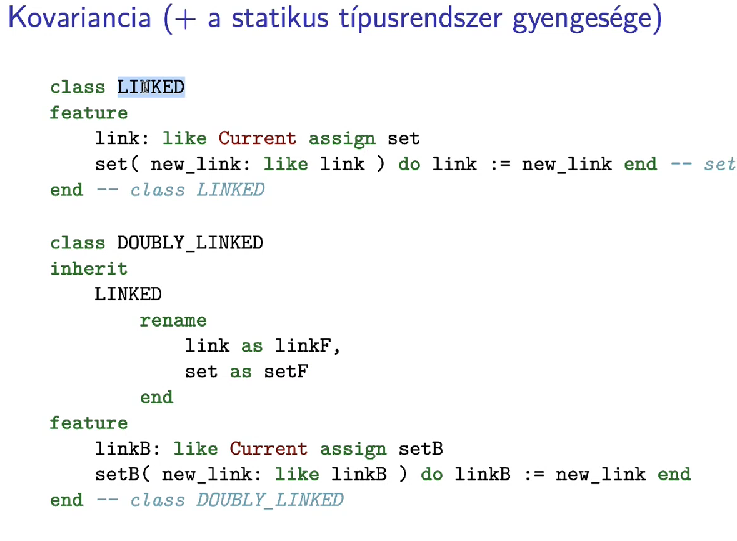
Csak az osztály és annak alosztályaiból elérhető featureök.

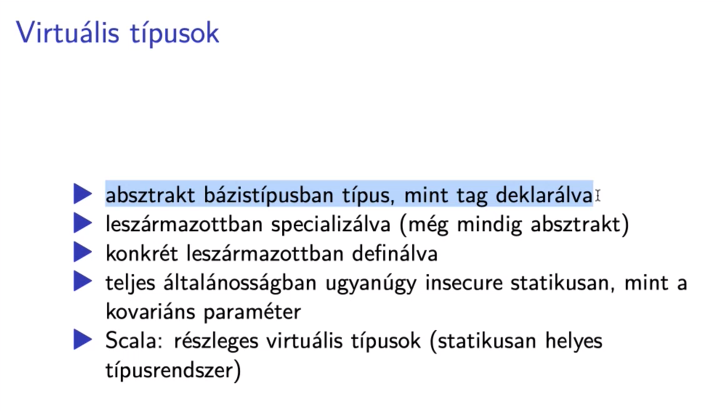
class B feature {B}

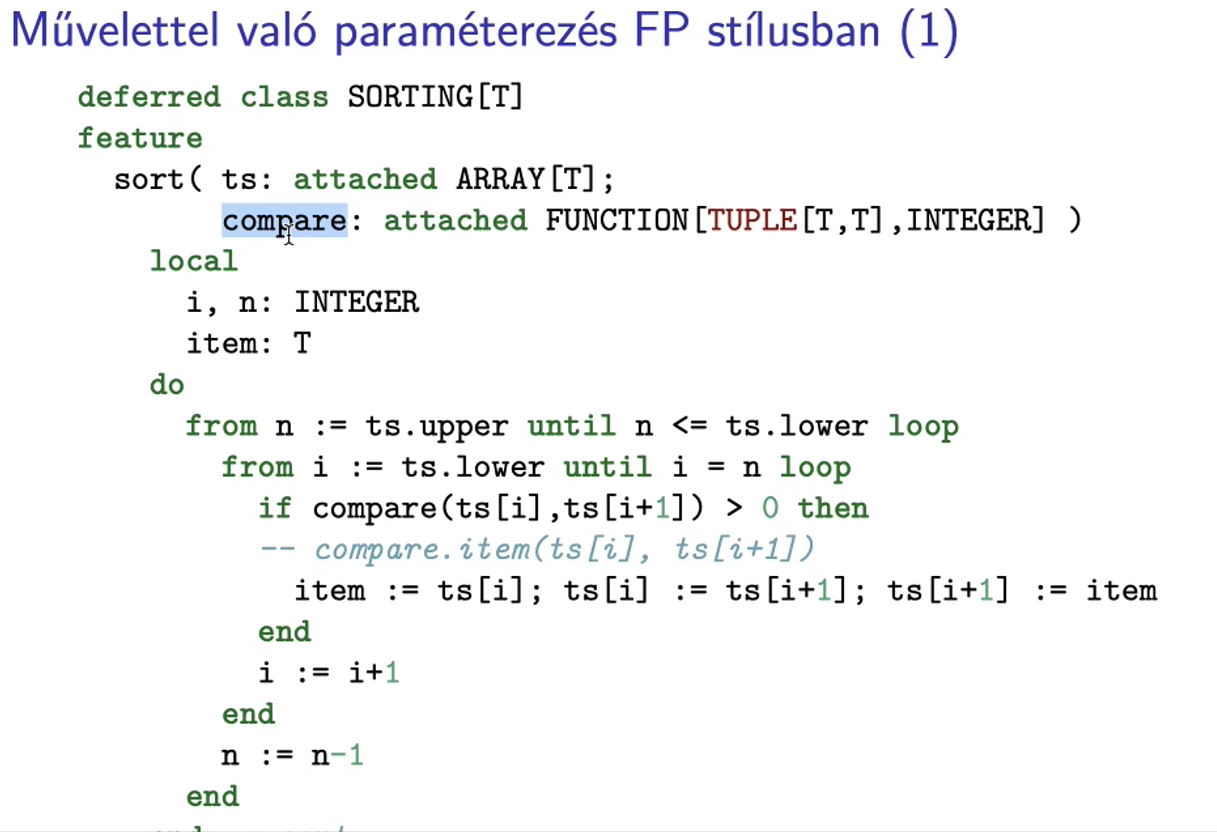
# Egyéb: generic paraméterek, virtuális típusok, agent

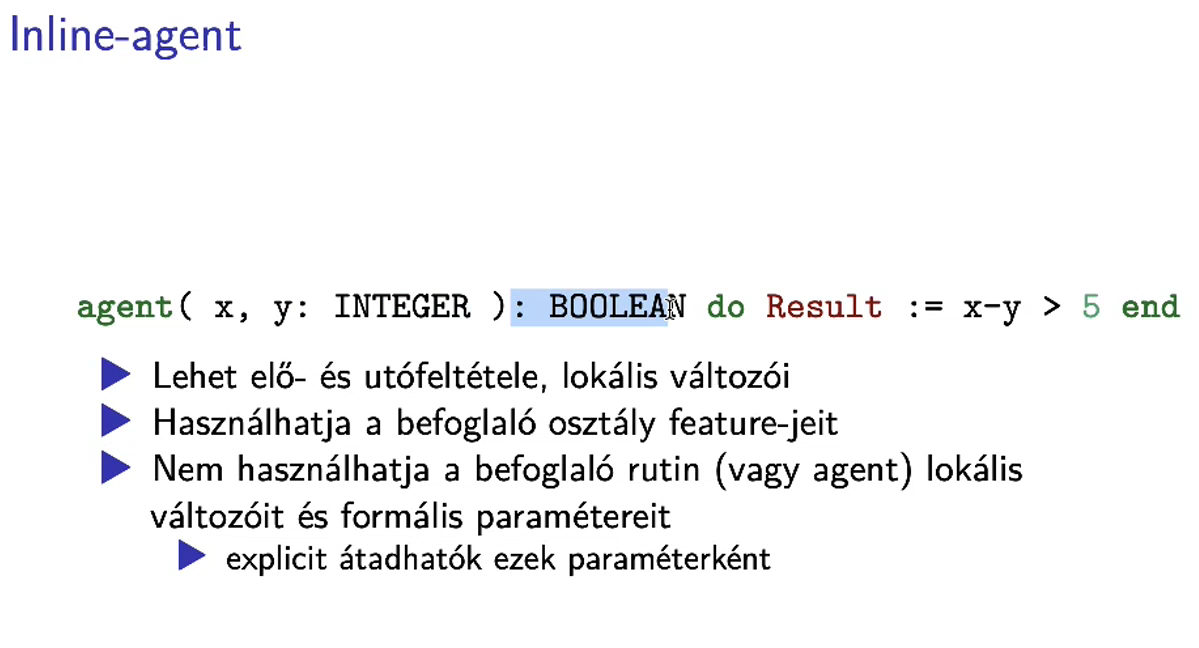


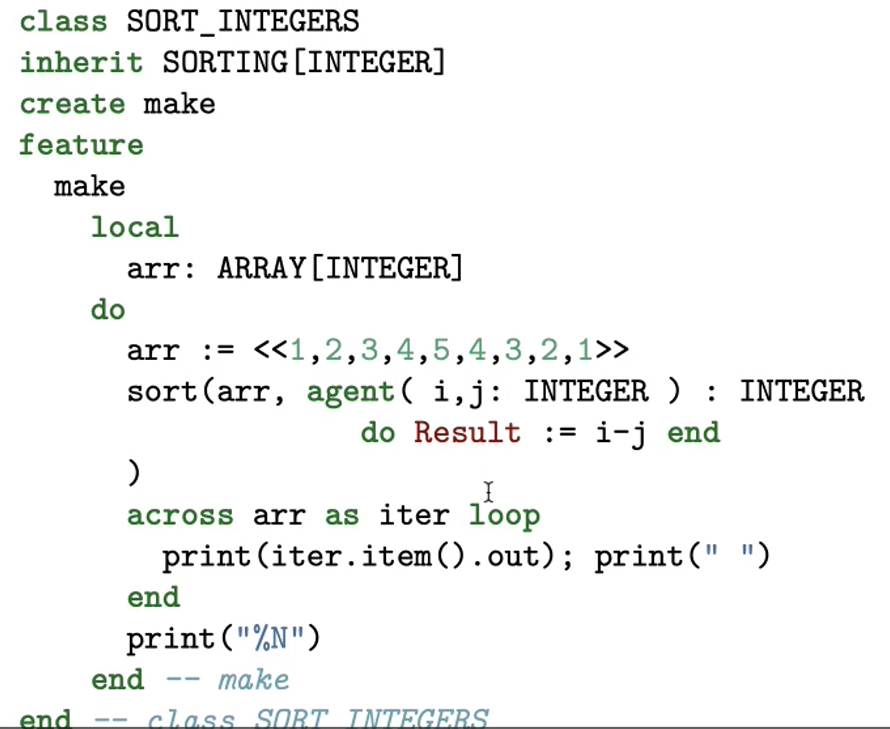




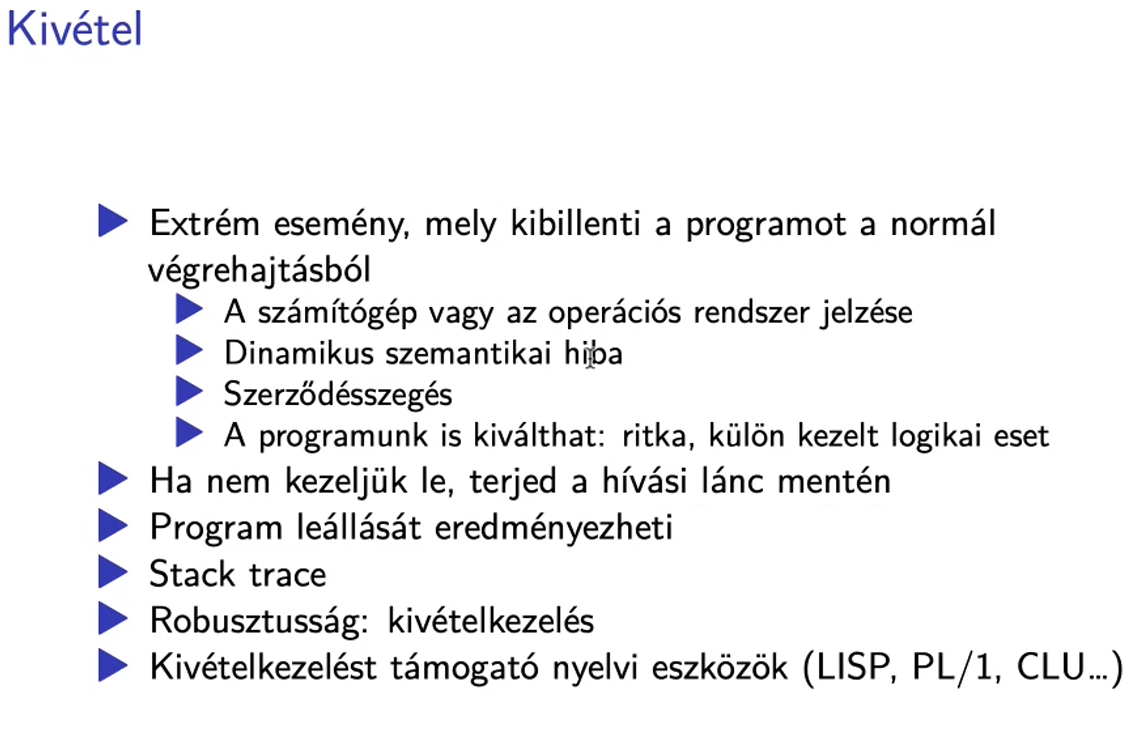


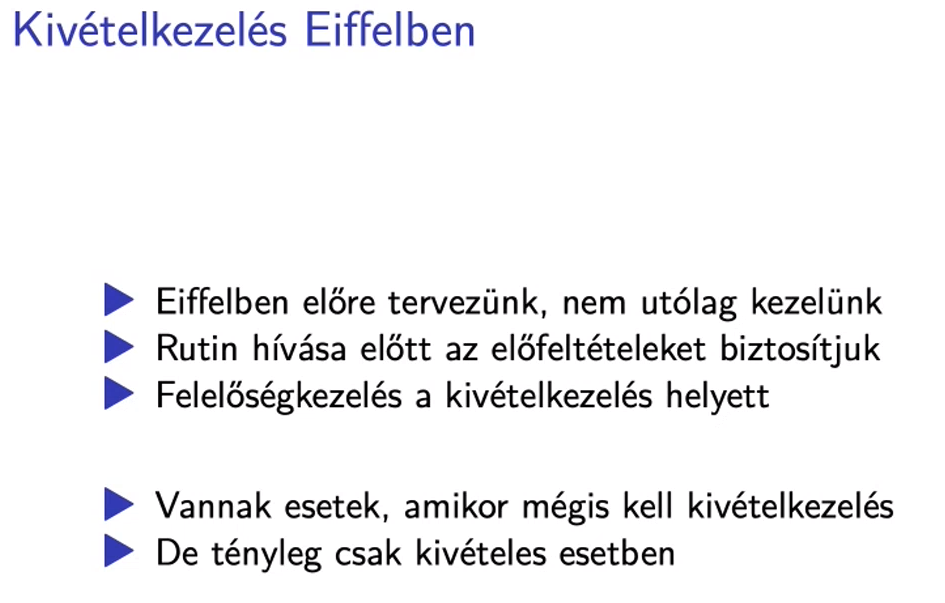




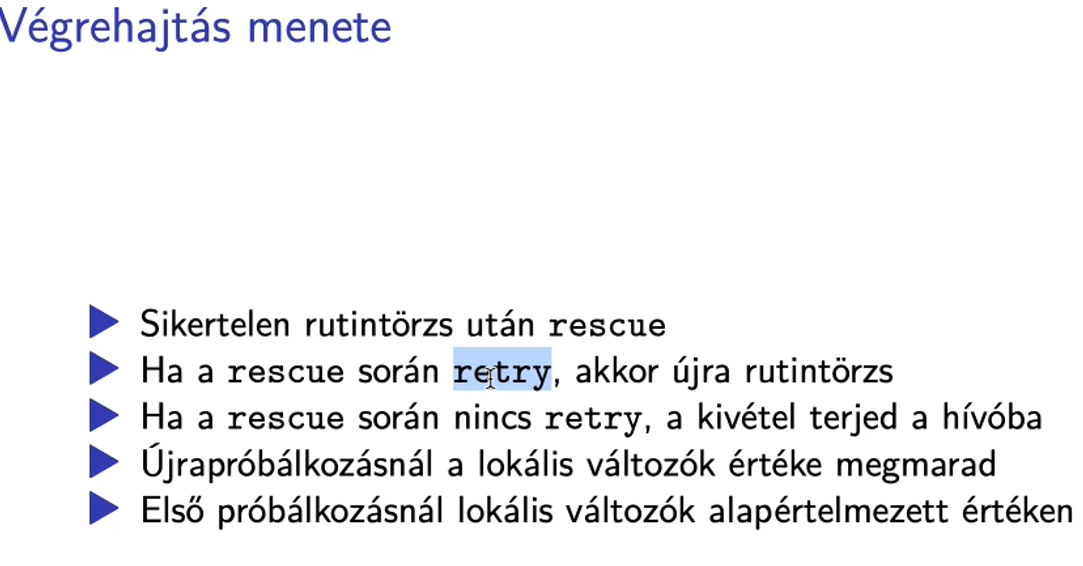


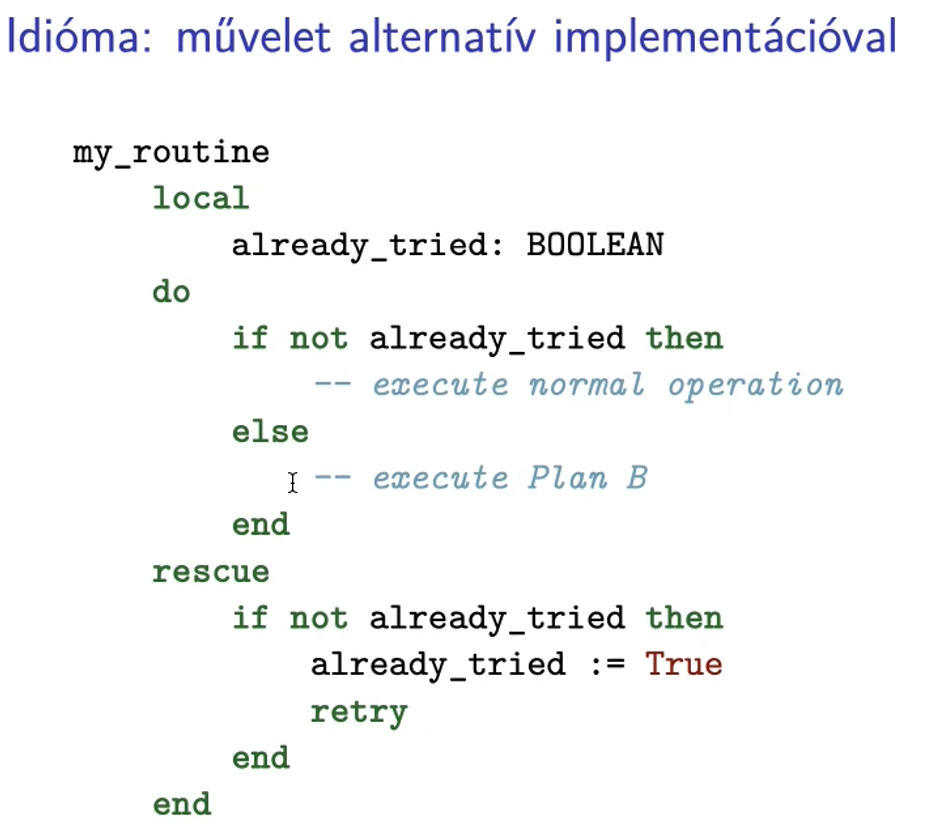
# Kivételkezelés

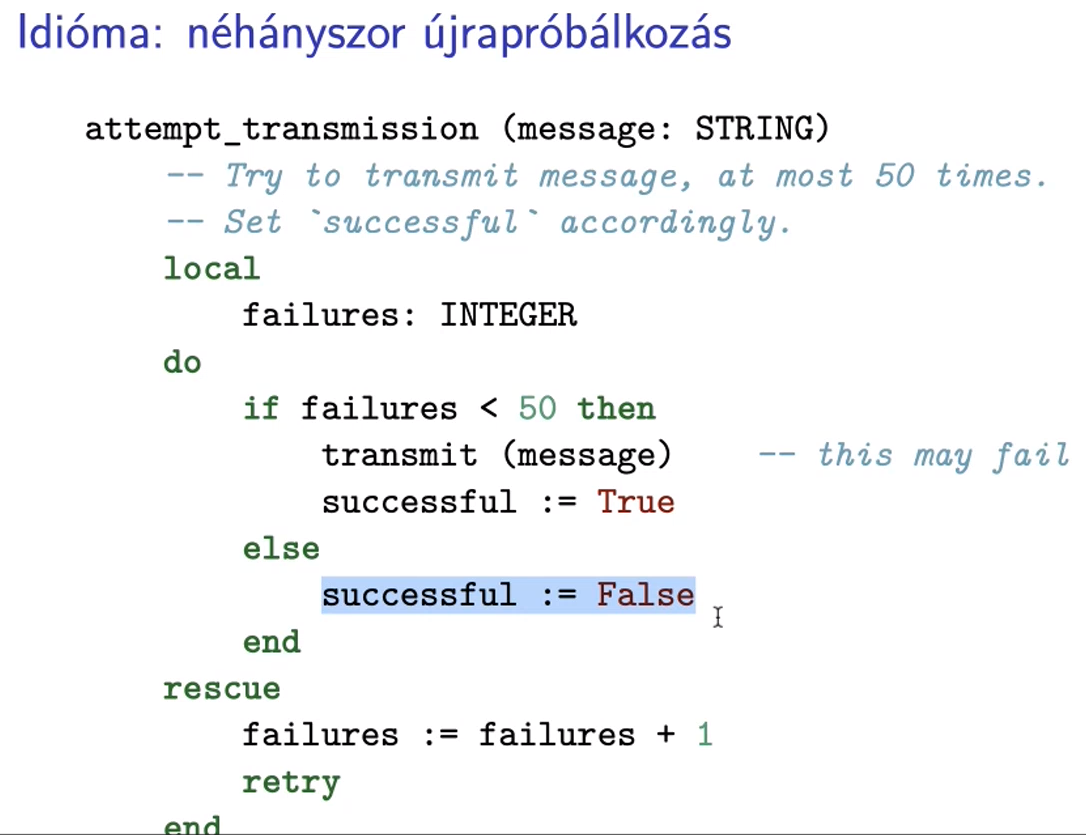


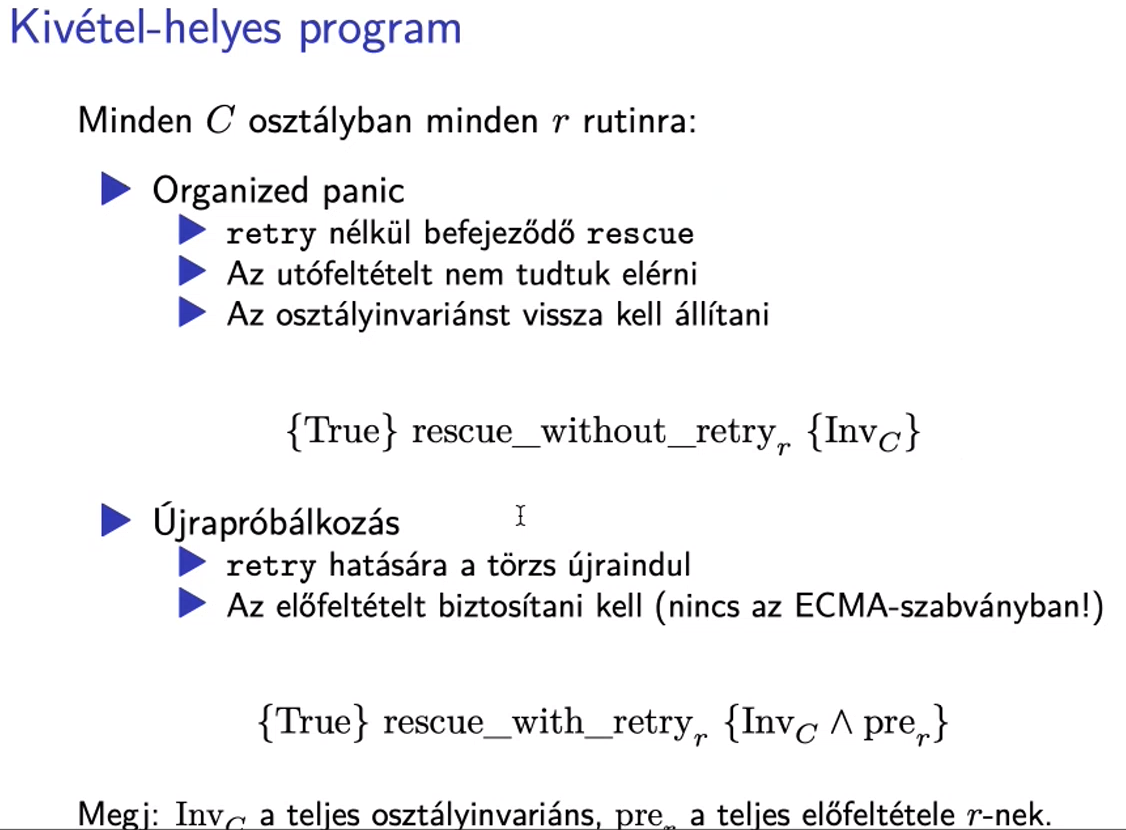


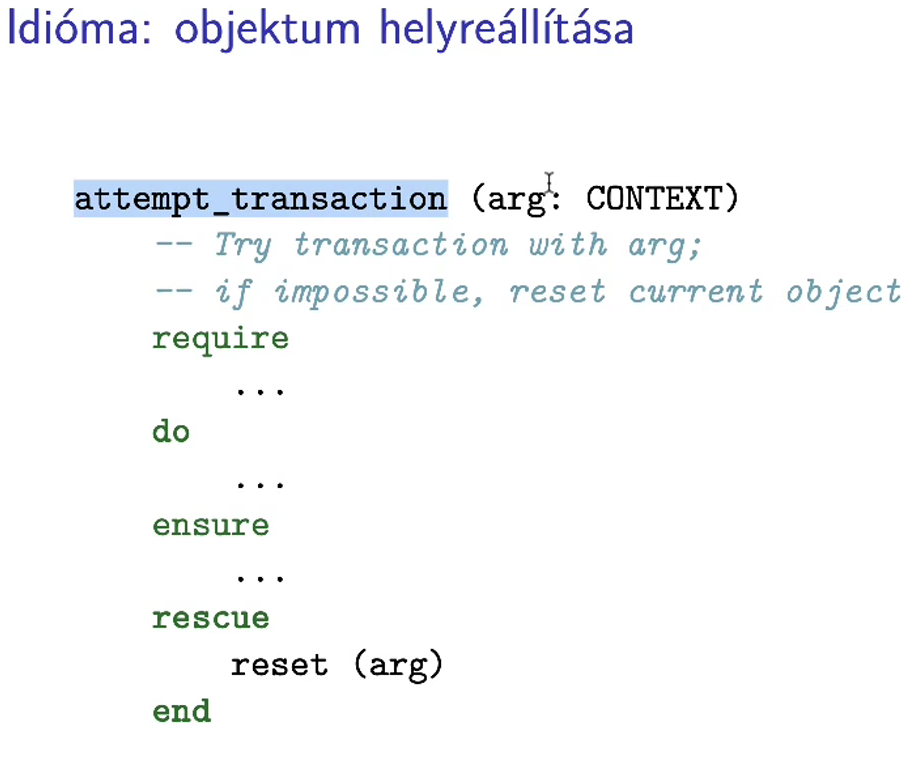


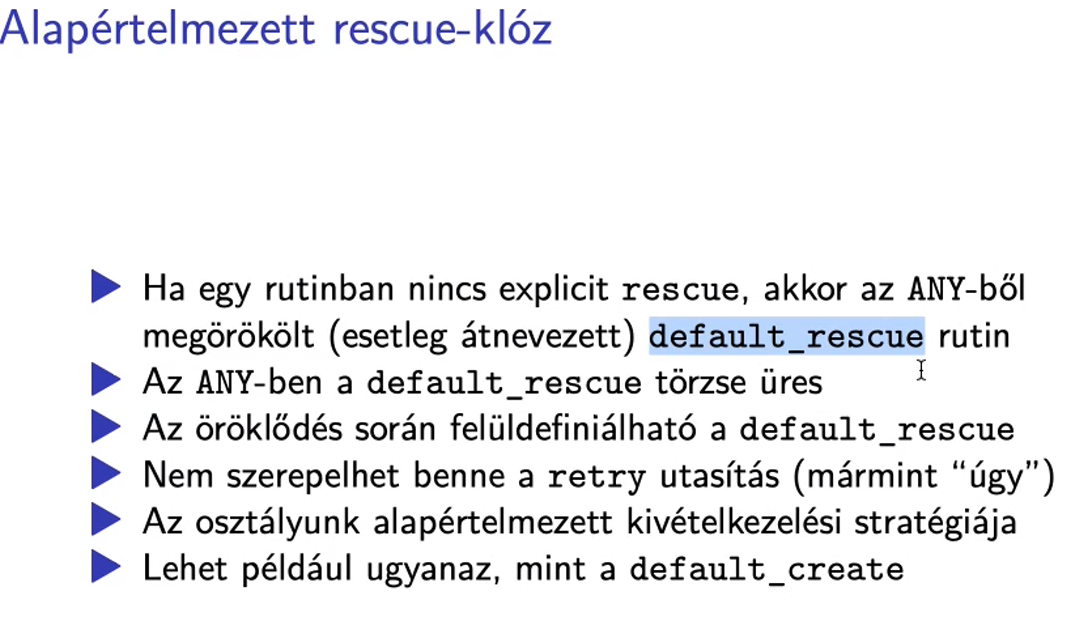












# Helyesség

