# PNYP: tételek kidolgozás

## Contents

# Minta válaszok

## Mit jelent a rescue klóz? Mi a szerződése (Hoare-triple), ha nincs benne retry?

Egy Eiffel rutin végén állhat egy rescue klóz, mely akkor fut le, amikor a rutin törzsében kivétel lép fel. A rescue klóz célja, hogy a rutin úgy érhessen véget, hogy legalább az osztályinvariáns teljesüljön, ha már az utófeltételt nem sikerült elérni. (A retry a rescue klózban a rutin újrakezdését kezdeményezi.) A szerződést (utófeltételt) teljesíteni nem tudó, és ezért kivételt kiváltó törzset követve lefut a rescue, majd a fellépett kivételt propagálja a hívó felé, azaz a hívóban a hívás helyszínén fellép a szóban forgó kivétel. A hívó a szerződést teljesíteni nem tudó objektumot konzisztens állapotban látja (az osztályinvariáns teljesül rá).

Az $r$ rutin $retry_r$ retry-klózának szerződése tehát: $\{true\}\ retry_r\ \{inv_C\}$ ahol $inv_C$ az $r$ osztályának, azaz a $C$-nek az egyesített invariánsa (a bázistípusokban és a $C$-ben deklarált invariánsok konjunkciója).

Amennyiben egy rutin nem tartalmaz explicit retry-klózt, az ANY-ből megörökölt (és esetleg átnevezett) `default_retry` eljárás lesz implicit módon a rutin retryklóza, melynek ANY-beli implementációja üres (SKIP utasítás). Egy lehetőség a retry-klózra egy creation-procedure végrehajtása (például a `default_retry` és a `default_create` ugyanolyan módon történő implementálása.)

## Mit értünk multimethodon?

Bizonyos objektumorientált nyelvekben a dinamikus kötés nem egyetlen kitüntetett paraméter dinamikus típusán működik, hanem az összes paraméter dinamikus típusát figyelembe veszi. Például az Eiffelben az `a.b(c)` rutinhívásnál az `a` és a `c` változók statikus típusa alapján dől el, hogy a `b` rutin hívása értelmes-e, és az a változó dinamikus típusa alapján dől el, hogy a művelet mely osztályban definiált implementációja hajtódik végre. Egy multimethodokat (azaz multiple dispatchet) támogató nyelvben az `a` és a `c` dinamikus típusa alapján választódik ki a végrehajtandó kód (azaz az a statikus típusában, illetve leszármazottaiban több olyan `b` művelet is lehet, amelynek a paramétere a `c` statikus típusának valamilyen altípusába tartozik).

A multimethodokat támogató nyelvekben az úgynevezett bináris műveletek (azaz amikor a művelet paraméterének típusa a fogadó objektum típusával kell megegyezzen, mint mondjuk egyenlőségvizsgálatnál) megvalósítása megoldott, hiszen minden osztályban megírhatjuk az arra az osztályra jellemző (és a fogadó és paraméter objektumot ugyanazzal

típusozó) implementációt, és a dispatch során a legjobban illeszkedő implementáció hívódik meg. A szimmetriáról a multiple dispatch gondoskodik

### Milyen lehetőségei vannak a constrained genericitynek Eiffelben?

Az Eiffelben a generikus típusoknak több (fix számú) típusparamétere lehet, és mindre tehetünk megszorításokat (az aktuális típusparamétereknek meg kell felelniük a formális típusparaméterekre tett megszorításoknak). Természetesen a generic típusdefiníción belül a típusparamétert olyan típusként használhatjuk, amely következik a rá adott megszorításokból. (Például olyan műveleteket használhatunk rá. . . )

Ha nem teszünk megszorítást, mint például a `class G[T]` esetben, az ekvivalens azzal, mintha azt írtuk volna, hogy class `G[T->ANY]`. Természetesen az `ANY` helyett más felső korlátot is adhatunk a típusparaméterre, pl: `class G[T -> HASHABLE]`.

Még érdekesebb, amikor egy típusparaméterre több megszorítást adunk: ebben az esetben az aktuális paramétertípusnak az összes feltüntetett típus altípusa kell lenni. Például `class G[ T -> {HASHABLE, NUMERIC} ]`.

Ha több típust is meghatározunk egy típusparaméter elvárt bázisaként, akkor ezeket szükséges lehet összehangolni, hogy konzisztensen lehessen használni a típusparamétert és a rá értelmezett műveleteket a generikus definícióban. Ez az összehangolás azt jelenti, hogy a különböző bázistípusokban lévő műveleteket átnevezhetjük, hogy a névütközést elkerüljük. Például class `G[T -> {B rename f as g end, C}]`. Ezek az átnevezések nem jelentik azt, hogy az aktuális típusparaméterben ilyen névvel kell legyenek deklarálva ezek a műveletek.

A típusparaméterre megadható az is, hogy egyedeket abból a típusból milyen creation procedure-ök segítségével lehet létrehozni. Ha például a `B` osztálynak van egy `make` nevű, paraméter nélküli művelete, akkor a `class G[T -> B create make end]` azt jelenti, hogy a `T`-nek megfeleltetett típus csak a `B` leszármazottja lehet, és ebben a típusban a `B`-ból megörökölt (és esetleg átnevezett) `make` művelet egy creation procedure.

Lehet olyan megkötést is adni egy típusparaméterre, hogy az aktuális csak olyan attached típus lehessen, amely öninicializáló, azaz amelynek az `ANY`-ból megörökölt (és esetleg átnevezett) `default_create` egy creation procedureje. Például `class G[?T]`. Lehet olyan megkötést tenni, hogy a típusparaméter invariáns: `class G[frozen T]`. Ellenkező esetben a típusparaméter kovariáns.

Az Eiffelben nincs F-bounded polymorphism, azaz egy `T` típusparaméterre adott megszorításban nem hivatkozhatunk a `T`-re.

### Strukturális altípusosság esetén milyen altípusrelációk kell, hogy teljesüljenek az $A$, $A'$, $B$, $B'$, $C$, $C'$ típusokra, hogy matematikailag helyes legyen, hogy az $(A \to B) \to C$ altípusa az $(A' \to B') \to C'$ típusnak?

A paraméter kontravariáns, az eredmény kovariáns kell legyen, ezért $(A' \to B') <: (A \to B)$ és $C <: C'$ kell, továbbá az elsőhöz az kell, hogy $A <: A'$ és $B' <: B$.

## `elmeleti_kerdesek.pdf`

### Írd fel egy exportált rutin és egy exportált creation procedure szerződését Hoare-hármasok segítségével, és magyarázd el a különbséget!

Rutin: $\{INV \wedge PRE\} rutin \{INV \wedge POST\}$

Creation procedure: $\{PRE\} creation \{INV \wedge POST\}$

Ahol $INV$ az osztály invariánst, $PRE$ és $POST$ a rutin/creation elő- és utófeltételét jelenti.

Különbség: egy creation procedure egy osztálypéldány elkészítéséért felelős, így az osztály invariáns a végrehajtás kezdetén még (általában) nem teljesül, viszont a creation procedure lefutása után (illetve bármely egyéb rutin lefutása után) igaznak kell lennie.

## Mi a szerepe a like kulcsszónak az Eiffelben?

Kapcsolt típusok hozhatók a `like` kulcsszó segítségével létre, az Eiffel "kovariáns" típusrendszerének fontos részét alkotja. Segítségével könnyen definiálhatóak olyan rutinok, amelyek kovariáns módon altípusokban is helyesek: nincs szükség őket újradefiniálni (pl. `is_equal`).

Használat: `like <változó>`, például `like Current` vagy `like item` (ha `item` egy attribútum).

A `like Current` az mindig olyan típusra fog utalni, ami a pillanatnyi osztály példány dinamikus típusa. Így például az alábbi osztályban a `next` attribútum olyan típusú értéket tárol, mint ami az osztálypéldány típusa. Ez azt jelenti, hogy amennyiben pl. a `LINKED_LIST` leszármazik a `LINKED` osztályból, akkor a `next` attribútum `attached LINKED_LIST` típusú lesz egy `LINKED_LIST` példányban.

```
class LINKED
feature
  next: attached like Current
```

## Mi a mixin-inheritance lényege?

Többszörös öröklődésre mutat alternatívát, lineárisan öröklődési láncokat képez a típusparaméterből való öröklődés engedélyezésével. Ennek köszönhetően nem lép fel a diamond inheritance probléma.

Eiffelben nincsen erre lehetőség. Az alábbi példa pszeudo-kód.

```
--Rational egy láncon keresztül örököl mind Numeric-ből, mind Comparable-ből
class Numeric[T] inherit T ... end --T nincs semmi másra használva
class Comparable[T] inherit T ... end --T nincs semmi másra használva
class Rational inherit Comparable[Numeric[Any]] ... end
--Rational örököl Comparable-ből, ami örököl Numeric-ből, ami örköl Any-ből
```

## Milyen szerződések vonatkoznak az Eiffel rutinokra? Hogyan befolyásolja ezt az öröklődés? Hogyan a felüldefiniálás? Magyarázd el a kapcsolódó fogalmakat és szabályokat. Fogalmazd meg a szerződéseket Hoare-hármasok segítségével!

A rutinok szerződése elő- és utófeltételből, illetve a befoglaló osztályuk invariánsából áll. Öröklődés, felüldefiniálás során, a Liskov helyettesítési elvet betartva, az előfeltétel gyengíthető (`require else`, kontravariancia), az utófeltétel pedig szigorítható (`ensure then`, kovariancia).

A szerződés Hoare-hármassal megadva: $\{INV \wedge PRE\} rutin \{INV \wedge POST\}$

Ahol $INV$ az osztály és a szülőosztályok invariánsainak konjukciója, a $PRE$ a rutin és, amennyiben van(nak), a felüldefiniált rutinok előfeltételeinek diszjunkciója, a $POST$ pedig az utófeltételek konjukciója.

Az osztály invariáns az `invariant`, az előfeltétel a `require`, az utófeltétel pedig az `ensure` klózzal adható meg.

Amennyiben nincsenek ezek a klózok kiírva, akkor alapértelmezett működésük az alábbival ekvivalens:

```
invariant True
require True
require else False
ensure True
ensure then True
```

A creation procedure lefutása kezdetén az osztály invariáns még nem feltétlenül teljesül (invariánstól függően, például a triviális `invariant True` már akkor is teljesül). A creation procedure-ök ilyen tekintetben különlegesek a szerződések szempontjából, de rájuk is vonatkozik minden más előbb felsorolt megkötés.

## Milyen viszonyban kell álljanak az $A_i$ és a $B_j$ típusok ahhoz, hogy fennálljon az alábbi altípus reláció (matematikailag helyesen)? Indokold válaszod! `(A1 -> A2) -> A3 -> A4 <: (B1 -> B2) -> B3 -> B4`

`(A1 -> A2) -> A3 -> A4 <: (B1 -> B2) -> B3 -> B4`

A függvénytípusok altípusossága a Liskov helyettesítési elvhez hasonlóan működik: az altípusban a paraméterek kontravariáns, az eredménytípus kovariáns módon változhat.

Azaz $A_1 \to A_2 <: B_1 \to B_2$ akkor teljesül, ha $A_2 <: B_2$ és $B_1 <: A_1$.

```
A4 <: B4
B3 <: A3
B2 <: A2
A1 <: B1
```

## A strukturális altípusosságot hogyan szokás definiálni direktszorzat (tuple) típusokra?

Az Eiffel programozási nyelv `TUPLE` típusán "helyesen" van értelmezve az altípusosság, amennyiben funkcionális módon vannak a tuple-k használva: nem módosítjuk a bennük tárolt értékeket. Ilyenkor a generikus paraméter(ek)ben kovarianciát engedünk meg.

`TUPLE[X] <: Tuple[Y]` amennyiben `X <: Y`

Ezen felül egy tuple több adattaggal altípusa egy kevesebb adattag rendelkező tuple-nek (a korábbi altípusossági szabályok betartása esetén).

`TUPLE[X,A] <: TUPLE[Y] <: TUPLE` amennyiben `X <: Y`

Formálisan: $n \geq m \wedge \forall i \in [1..m] : A_i <: B_i \implies \times_{i=1}^{n} A_i <: \times_{i=1}^{m} B_i$

## A strukturális altípusosságot hogyan szokás definiálni rekord típusokra?

Lásd tuple kérdés. Különbség: nevesítve vannak az adattagok, nem pedig az indexük azonosítja őket.

Formálisan: $B$ rekord típus minden szelektora szerepel az rekord típusban is, és minden ilyen szelektorra az $A$-beli típus altípusa a $B$-beli típusnak, akkor $A <: B$

## Mire való az old kulcsszó az Eiffelben? Magyarázd el, fogalmazd meg a rá vonatkozó szabályokat, illusztráld használatát példán!

Az `old` kulcsszó segítségével egy rutin utófeltételében (`ensure`) a pillanatnyi osztálypéldány azon állapotát tudjuk elérni, amelyet a rutin futása legelején ("eredetileg") vett fel.

El tudjuk érni például egyetlen attribútum eredeti értékét, de a például a `strip` kulcsszóval is kombinálható az `old` kulcsszó, hogy egy vagy több attribútum kivételével hivatkozzunk az osszes attribútumra.

```
feature
  balance: INTEGER
  add_balance(amount: INTEGER)
    do
      balance := balance + amount
    ensure
      balance_added: balance = old balance + amount
      frame: strip(balance) ~ old strip(balance)
    end
```

## Hogyan működik, miért és mire jó az átnevezés az öröklődés során?

Öröklődés során a `rename` kulcsszó segítségével lehet rutinokat, attribútumokat átnevezni. Ilyenkor az adott feature ugyan az marad, csupán egy másik karaktersorozaton keresztül lehet majd rá hivatkozni a továbbiakban.

```
--használat
class TEST
inherit ANY rename default_create as make end
create make
end
```

Az átnevezés segítségével feloldhatóak például a többszörös öröklődés során felmerülő névütközések: amennyiben több osztályból öröklünk és több osztályban is szerepel ugyan olyan nevű feature, akkor átnevezéssel megtartható mindegyik ilyen feature, nincs szükség azok join-olására vagy undefine-olására.

## Mit jelent a mixin inheritence? Mi az előnye? Minek az alternatívája?

Lásd Mi a mixin-inheritance lényege?

## Milyen szerződésnek kell megfelelnie a retry kulcsszóval végződő kivételkezelő ágaknak? Fogalmazd meg Hoare-hármas segítségével!

A `retry` kulcsszó hatására a rutin törzse újra le fog futni, ezáltal (bár a szabvány nem követeli meg) az előfeltételt (a rutin előfeltételét és az osztály invariánsát) helyre kell állítani. A kivételkezelő ágnak nincsen saját előfeltétele: nem feltétlezhetjük, hogy valamilyen jól definiált állapotból lépünk a kivételkezelő ágra.

`{True} rescue_with_retry {INV and PRE}`

## Milyen szerződésnek kell megfelelnie egy ciklusnak? Fogalmazd meg Hoare-hármasok segítségével!

```
from INIT -- Inicializáció; utasítás(ok)
invariant INV -- Végig igaz állítás; A -> BOOLEAN
until COND -- Terminálási feltétel; A -> BOOLEAN
loop BODY -- Ciklustörzs; utasítás(ok)
variant VAR -- Iterációnként csökkenő érték; A -> INTEGER
end
--"A" jelentése: állapottér, azaz paraméterek, lokális változók, adattagok értékei
```

Egy ciklus megfelel a szerződésnek, ha ezek teljesülnek:

- $\{True\}\ INIT\ \{INV\}$
- $\{INV \wedge \neg COND\}\ BODY\ \{INV\}$
- $INV \implies VAR \geq 0$
- $\forall v : \{INV \wedge \neg COND \wedge VAR = v\}\ BODY\ \{VAR < v\}$

## Mire való az “old”? Mikor, hogyan, mire használjuk? Mutass példát is!

Lásd Mire való az old kulcsszó az Eiffelben? Magyarázd el, fogalmazd meg a rá vonatkozó szabályokat, illusztráld használatát példán!

## Mit szokás bináris műveletnek nevezni az objektumorientált programozásban?

Egy kétparaméteres metódust. Az egyik paraméter lehet a kiemelt paraméter, a pillanatnyi osztálypéldány (`Current`).

Általában a két paraméter típusa megegyezik, és bináris operátorok esetében (pl. összeadás) az eredmény típusa is megegyezik a paraméterek típusával. Tehát ilyenkor a bináris művelet egy $T \times T \to T$ függvény, ahol $T$ egy típust jelöl.

```
feature
  added alias "+" (other: like Current): like Current
    do
      create Result.make(value + other.value)
    end
```

Nem feltétlenül egyezik meg az eredmény és a paraméterek típusa. Példa az `ANY` osztály `is_equal` művelete: `like Current` típusú paramétert vár, de `BOOLEAN` értékkel tér vissza.

## Mi a különbség az =, a ~, az is_equal, az equals, a standard_is_equal és az is_deep_equal között? Mi közük van egymáshoz?

Az Eiffel programozási nyelv 3-4 féle egyenlőségvizsgálatot különböztet meg.

Referencia típusok esetén a `=` operátor a referenciákat hasonlítja össze, azaz ez az azonosságot vizsgálja, nem az egyenlőséget. Pontosan akkor ad vissza igazat, ha két oldalán lévő osztálypéldányok ugyan azok (egymás alias-ai). Kifejtett (`expanded`) típusok esetén a `=` és a `~` operátorok azonos módon működnek.

A három féle egyenlőségvizsgálat: egyedi (`is_equal`), sekély (`standard_is_equal`), mély (`is_deep_equal`). Amennyiben nincsen az egyedi egyenlőségvizsgálat felüldefiniálva, akkor azonos módon működik a sekély egyenlőségvizsgálattal.

A `~`, `is_equal` és `equal` ugyan azt csinálják, csupán különböző módon adnak hozzáférést az implementációhoz:

```
local
  a,b: attached ANY
do --Az alábbiak ekvivalens megfogalmazások
  a ~ b
  a.is_equal(b)
  equal(a,b)
end
```

Az egyedi egyenlőségvizsgálat az `is_equal` rutin felüldefiniálásával adható meg. Egy ekvivalenciarelációnak kell lennie, amely a `copy` metódussal összhangban van: `copy` után a két osztálypéldánynak meg kell egyeznie `is_equal` szerint.

A sekély egyenlőségvizsgálat osztálypéldányok mezőit `=`-vel hasonlítja össze. A mély egyenlőségvizsgálat a mezőket `is_deep_equal`-lal hasonlítja össze.

Az egyenlőségvizsgálat kifejtett és referencia típusokon eltérő módon működik.

- Referencia típus esetén:
  - Azonosság (referencia összehasonlítás): `a = b`
  - Tartalmi egyenlőség, programozó által definiálva: `a ~ b`
  - $= \subset$ standard_equal $\subseteq$ equal $\equiv \sim$
  - $= \subset$ standard_equal $\subseteq$ deep_equal
- Kifejtett (`expanded`) típus esetén:
  - Tartalmi egyenlőség, programozó által definiálva: `a ~ b` vagy `a = b` (kettő megegyezik)
  - standard_equal $\subseteq = \equiv$ equal $\equiv \sim$
  - standard_equal $\subseteq$ deep_equal

## Hogyan szabályozza az Eiffel a láthatóságot? Miben különbözik ez a más nyelvekben megszokottól?

Eiffelben megadhatjuk, hogy mely osztályok (és azok leszármazottai) számára legyen egy-egy attribútum vagy rutin elérhető.

Publikus láthatóság: `{ANY}`, pl. `feature {ANY} set_balance`

Protected féle láthatóság a `MYCLASS` osztályban (saját és alosztályok számára elérhető, de független osztályok számára nem): `{MYCLASS}`

Valójában a két eset hasonlóan működik: mindig a kapcsos zárójelek között jelzett osztály leszármazottai számára lesz elérhető a feature, viszont minden osztály (közvetve vagy közvetlen) leszármazik az `ANY` osztályból.

Privát láthatóság: csak a pillanatnyi osztálypéldány számára érhető el (akár alosztályban, akár a pillanatnyi osztályban): `{NONE}`
Fontos kiemelni, hogy itt az osztálypéldány számít: egy másik osztálypéldány `{NONE}` láthatóságú feature-jeit nem tudjuk elérni, csak a `Current` példányét.

Eiffelben öröklődés során van lehetőség mind a láthatóság bővítésére, mind a láthatóság szűkítésére. A legtöbb programozási nyelvben, a Liskov helyettesítési elvet követve, csak a láthatóság bővítésére van lehetőség.

## Milyen szerződés vonatkozik egy retry nélküli kivételkezelőre? Írd le Hoare-hármassal!

Amennyiben `retry` nélkül fejeződik be egy kivételkezelő ág, akkor a kivétel tovább terjed (felfelé) a stacken (a hívóhoz), így feltehetően az utófeltétel sem lett kielégítve. Ilyenkor a kivételkezelő ágnak az osztály invariánsát helyre kell állítania. A rutin előfeltételét nem kell helyreállítani, hiszen a rutin nem fog újból lefutni, mint `retry` alkalmazása esetén. A kivételkezelő ágnak nincsen saját előfeltétele: nem feltétlezhetjük, hogy valamilyen jól definiált állapotból lépünk a kivételkezelő ágra.

`{True} rescue_without_retry {INV}`

## Milyen szerződés köti az alosztályban felüldefiniált műveletet? Írd le Hoare-hármassal!

Lásd Milyen szerződések vonatkoznak az Eiffel rutinokra? Hogyan befolyásolja ezt az öröklődés? Hogyan a felüldefiniálás? Magyarázd el a kapcsolódó fogalmakat és szabályokat. Fogalmazd meg a szerződéseket Hoare-hármasok segítségével!

## Mi a family polimorfizmus alapproblémája?

Olyan (altípusossági) kapcsolatok alakulnak ki öröklődés miatt osztályok között, amelyek nem kívánatosak.

Egy megoldási módszer a kódismétlés (öröklődésre alternatíva).

Egy másik megoldási módszer például a Scala programozási nyelvben az osztálycsaládok létrehozása virtuális típusok segítségével.

## Milyen esetekben áll fenn altípus reláció paraméterezett típusok között az Eiffelben? Mit jelent itt a frozen?

Normál esetben a típusparaméter kovarianciája meg van engedve Eiffelben: egy `STACK[T]` osztály esetén `STACK[INTEGER] <: STACK[ANY]`-nek, hiszen `INTEGER <: ANY`.

Több típusparaméterrel is működik a kovariancia, illetve amennyiben `DEQUEUE <: STACK`, akkor `DEQUEUE[INTEGER] <: STACK[ANY]`, stb.

`frozen` típusparaméter esetén (pl. `STACK[frozen T]`) invariancia áll fenn a típusparaméterre nézve: `STACK[INTEGER]` és `STACK[ANY]` között nincs altípus reláció.

## Mit jelent az attached (az Eiffel szabványban "!") és a detachable (a szabványban "?")?

`attached T`: egy olyan `T` típus, ami nem lehet `Void` értékű.

`detachable T`: egy olyan `T` típus, ami lehet `Void` értékű.

Jelenleg Eiffelben ha nem adjuk meg, hogy egy típus `attached` vagy `detachable`, akkor automatikusan `attached`-ként lesz kezelve.

A kifejtett (`expanded`) osztályok alapértelmezetten `attached`-ek, hiszen ezek nem referenciákon/pointereken keresztül vannak használva, hanem kifejtésre kerülnek az őket tartalmazó egységbe (objektumba/stack-re).

Használatra példa: `feature name: attached STRING`

Az `attached T` típus altípusa a `detachable T` típusnak, hiszen egy specializációról beszélünk: a `Void` érték kizárásáról.

Eiffelben több mód is van annak ellenőrzésére, hogy egy adott referencia egyenlő-e `Void`-dal (nullpointer-rel):

```
local
  ref: detachable T
do
  if ref = Void then ... end
```

```
if attached ref as a_ref then ...
                          else ... end
check attached ref as a_ref then ... end
```

### Milyen módon kezeli a kivételeket egy rutin, ha nincs benne rescue?

Eiffelben előre tervezünk (az előfeltétel biztosítása a rutin meghívása előtt a hívó felelőssége), nem pedig utólag kezelünk, tehát kivételeket tényleg csak nagyon kivételes esetekben használunk. A többi nyelvben megszokott `try-catch` utasítások helyett Eiffelben van lehetőség minden rutinhoz megadni egy `rescue` klóz, ami akkor fut le, ha a rutin végrehajtása során bárhol egy kivétel lépett fel.

Ha egy rutinhoz nem tartozik `rescue` klóz, akkor az a megörökölt rutin fog lefutni, ami az `ANY` osztályban `default_rescue` néven volt deklarálva. Ez a rutin azóta lehet, hogy átnevezésre, illetve felüldefiniálásra került.

Alapértelmezetten ez a rutin nem csinál semmit: üres `do end` implementációval rendelkezik. Azonban felüldefiniálás esetén lehetőséget biztosít például különböző debug információk logolására, vagy tetszőleges logika futtatására.

A `default_rescue` lefutását követően a kivétel propagálódik a stack-en felfelé, azaz a rutin hívójához kerül lekezelésre (ha van `rescue`) vagy tovább propagálódik.

A le nem kezelt kivétel eredményeként előfordulhat, hogy az osztály invariánsa is megsértésre kerül, hiszen nincsen egy `rescue` klóz, ami helyre tudná állítani az invariánst. Ideális esetben a `default_rescue` ezt megoldja: helyreállítja az osztály invariánst, például egy creation procedure-ként is használható feature lefuttatásával.

### Mi a default_create?

A `default_create` egy `ANY` osztályban definiált rutin. Alosztályokban automatikusan egy creation procedure lesz belőle, amennyiben az alosztály nem deklarál explicit legalább egy creation procedure-t. Természetesen a `default_create` is deklarálható, mint creation procedure.

A `default_create` különlegessége még továbbá, hogy amennyiben egy osztálypéldány létrehozásakor nincsen megnevesítve, hogy melyik creation procedure fusson le, akkor a `default_create` fog lefutni (amennyiben az egy creation procedure az adott osztályra): `create {PERSON}`

A `default_create` átnevezhető és felüldefiniálható leszármazás során, ez a fent leírt funkcionalitáson nem változtat, csupán más néven keresztül lehet majd hivatkozni rá.

Alapértelmezetten a `default_create` rutin üres törzzsel rendelkezik, nem csinál semmit, azaz az osztály attribútumait azok nullaszerű értékeire inicializálja (referencia típusok esetén ez a `Void`, számok esetén ez a 0).

### Hogyan szól a Liskov-féle helyettesítési elv (substitution principle)?

A Liskov-féle helyettesítési elv szerint egy szülőosztály példánya helyett szabadon használható annak egy alosztály példánya, anélkül, hogy a megvalósított funkcionalitás elromlana.

Ennek elérése érdekében például az alábbi tulajdonságokkal kell rendelkeznie az öröklődésnek:

- Kovariáns módon változhat:
  - Visszatérési érték típusa
  - Felléphető kivételek listája (szűkülhet)
  - Utófeltétel (erősödhet)
- Kontravariáns módon változhat:
  - Paraméterek típusa
  - Láthatóság (bővülhet)
  - Előfeltétel (gyengülhet)
- Egy módosíthatatlan típusnak nem lehet módosítható altípusa

### Mi a kapcsolt (anchored) típus az Eiffelben? Mutass példát arra, hogy miért jó!

Lásd Mi a szerepe a like kulcsszónak az Eiffelben?

## Milyen szerződésnek kell eleget tegyen egy creation procedure? Írd le Hoare-hármas segítségével!

A creation procedure egy osztálypéldány létrehozásakor fut le, az osztálypéldány inicializációjáért, az osztályinvariáns beállításáért felel. Az invariáns a creation procedure lefutása előtt még nem (feltétlenül) áll fenn.

$\{PRE\}creation\{INV \wedge POST\}$

Ahol $PRE$ és $POST$ az elő- és utófeltétele a creation procedure-nek, az $INV$ pedig az osztály invariánsa (beleértve ezekbe az öröklődés során megörökölt megkötéseket is).

A creation procedure-rel ellentétben a további művelet szerződése az alábbi:

$\{INV \wedge PRE\}routine\{INV \wedge POST\}$

## Mire való az only az Eiffelben (illetve a strip a jelenlegi implementációkban)?

Elsősorban úgynevezett "frame condition" utófeltételek készítésében hasznos a kulcsszó: le tudjuk a segítségével írni, hogy egy (vagy több) attribútum kivételével az osztály állapota változatlan maradt.

A `strip`-re úgy gondolhatunk, mint egy műveletre, ami visszaad egy rekordot, ami minden attribútumát tartalmazza a pillanatnyi osztálypéldánynak, kivéve a `strip` paraméterlistájában felsorolt attribútumokat. A visszakapott rekordot össze tudjuk egy másik rekorddal hasonlítani az utófeltétel során, elsősorban az `old` kulcsszóval kombinálva.

```
feature
  set_balance(amount: INTEGER)
  do ...
  ensure
    balance_set: balance = amount
    frame_rule: strip(balance) ~ old strip(balance)
  end
```

A motiváció (frame problem) az, hogy bár az utófeltételben le tudjuk írni, hogy mi változott, valahogy azt is le akarjuk írni, hogy nem változott meg minden más is. Azaz például attól még, hogy megváltozott egy személy lakcíme, attól még a személy neve, születési dátuma, stb. változatlan marad. Erre ad nekünk egyszerű lehetőséget a `strip` kulcsszó.

## Mi a különbség a strukturális és a nominális altípusosság között?

Strukturális altípusosság esetén `C <: P` akkor áll fenn, ha `C`-ben minden rutin, attribútum, stb. megtalálható, amely `P`-ben is (azaz ha szerkezetileg `P` részhalmaza `C`-nek).

Ezzel elletétben nominális altípusosság esetén csak akkor áll fenn az altípus reláció (`C <: P`), amennyiben a `C` osztály expliciten a `P` osztály altípusának van megjelölve. Jellemzően ez az öröklődéssel együtt történik: egy osztályból való öröklődés vezeti be az altípusosságot. Ilyen például az Eiffel nyelv: az `inherits` kulcsszóval lehet altípusosságot bevezetni.

Azaz bár strukturális altípusosság esetében nem szükséges, hogy a két típusnak köze legyen egymáshoz (elég, ha az egyik tartalma részhalmaza a másik tartalmának), nominális altípusosság esetében az altípus reláció csak explicit módon tud létrejönni.

## Mit jelent a polymorphic CAT-call?

A "CAT" akronim "Changed Availability or Type"-ot jelöl.

A polymorphic szó jelen esetben az altípusosságra, a leszármazásra utal, ami az Eiffel programozási nyelvben a polimorfizmus egyik lehetséges megtestesülése.

Akkor beszélünk CAT-ről, amikor egy megörökölt metódus láthatóságát, paramétererének típusát vagy visszatérési értékének típusát megváltoztatjuk. Eiffelben van lehetőség a láthatóság bővítésére, illetve szűkítésére is, illetve kovariáns módon a paraméterek és a visszatérési értékek típusinak megváltoztatására is.

A CAT-call alatt a problémás esetet értjük általában, azaz amikor egy bázistípusú referencia egy altípus példányára mutat, ahol a meghívás alatt álló rutin az altípusban megváltozott láthatósággal vagy típusokkal rendelkezik: a láthatósága vagy csökkent, vagy a kovariáns módon változott egy/több paraméterének a típusa. Ilyenkor a metódus meghívása nem feltétlenül lesz sikeres, annak ellenére, hogy statikusan a bázis osztály által deklarált láthatóság/paraméter típus megfelelő lenne. Mivel az altípusban a Liskov helyettesítési elvet megsértő változtatásokat vezettünk be, ezért bár a bázistípusú referencián keresztül a fordító szerint meg tudjuk hívni a metódust, valójában a metódus szignatúrája nem felel meg az aktuális paramétereknek vagy a szükséges láthatóságnak, így egy hiba lép fel.

Példa:

```
class FOOD end
class MILK inherit FOOD end

class ANIMAL
feature eat(p: attached FOOD) do ... end
end

class CAT
inherit ANIMAL
feature eat(p: attached MILK) do ... end
end
```

Ilyen esetben az alábbi függvényhívás egy hibát okozó CAT-call:

```
local
  f: attached FOOD
  a: attached ANIMAL
do
  create f
  create {CAT} a
  a.eat(f) --sikeres fordítás :(
```

## Mit értünk az alatt, ha egy objektumművelet esetén a paraméterek nonvariánsak? Milyen problémát vet fel?

A paraméterek nonvarianciája azt jelenti, hogy (elsősorban leszármazáskor) a paraméterek típusa nem változtatható meg az alosztályban.

Az Eiffel programozási nyelv nem ilyen: ott kovariánsak a paraméterek.

Invariáns paraméterek esetén egyes műveletek megvalósítása nehezebb, mintha a kovarincia engedélyezve lenne. Viszont a kovariáns paraméterek problémákat okozhatnak, hiszen megsértik a Liskov helyettesítési elvet, lásd CAT-call. Tehát a Liskov helyettesítési elv szerint a kontravariáns paraméterek lennének a helyesek, de az invariáns paraméterek is biztonságosak, azonban egyes esetekben hasznos a paraméterek esetén is a kovariancia.

Például egy `is_equal` metódus, amely eldönti, hogy a pillanatnyi osztálypéldány egyezik-e a paraméterként kapott példánnyal, hasznos, ha olyan típusú paramétert fogad, mint amilyen a pillanatnyi osztálypéldány.

Hasonlóan, néha az alosztályokban csak speciálisabb értékeket szeretnénk paraméterként elfogadni, hiszen maguk az alosztályok is az általános osztály, a bázisosztály speciális változatai. Például invariáns paraméterek esetén nem tudjuk azt leírni, hogy az `ANIMAL` osztály `eat` metódusa `FOOD` típusú értéket kaphat, de a `CAT <: ANIMAL` osztály `eat` metódusa csak `MILK` paramétert fogad el.

## Hogyan, milyen módokon örökölheti meg az A osztály a B osztály egy f feature-ét, ha C és D a B gyermeke, valamint A a C és a D gyermeke? Milyen eseteket különböztethetünk meg?

Ezt a problémát diamond inheritance-nek is szokás nevezni, az öröklődési gráf lerajzolt alakja miatt.

Az $A$ osztály a $B$ osztályt két irányból is ($C$-n és $D$-n keresztül is) megörökli. Attól függően különböztetünk eseteket, hogy $C$ és $D$ az $f$ feature átnevezte-e, felüldefiniálta-e.

- $f$ át lett nevezve
  - $C$-ben és $D$-ben eltérő nevekre: $A$-ban az $f$ feature kétszer, két különböző néven fog szerepelni. A `select` kulcsszó használata kötelező, hogy dinamikus kötés esetén el tudja dönteni a futási környezet, hogy a két implementáció ($C$ és $D$ beli) közül melyik fusson le, amennyiben az $f$ feature egy $D$ statikus típusú referencián keresztül van meghívva
  - $C$-ben és $D$-ben azonos névre: olyan, mintha nem lett volna átnevezve
- $f$ nem lett átnevezve
  - sem $C$, sem $D$ nem definiálta felül az $f$ rutint
    - Ilyenkor automatikus join történik: csak egy $f$ rutin lesz $A$-ban, ami azonos módon működik, mint $D$-ben
  - $C$ vagy $D$ felüldefiniálta az $f$ rutint
    - Ilyenkor nem lehetséges az automatikus join, hiszen nem tudja a fordítóprogram, hogy két implementáció közül melyik tartsa meg.
    - Egyik lehetőség: eldobjuk az egyik implementációt (`undefine`), ilyenkor a másik marad meg
    - Másik lehetőség: átnevezzük az egyiket, így két feature-ünk lesz, nem fognak ütközni egymással. Ilyenkor szintén `select`-tel jelezni kell, hogy dinamikus kötés során melyik implementáció fusson le

## Milyen lehetőségeket biztosít egy kovariáns sablonparaméter?

Eifellben van lehetőségünk kovariáns típusparaméterek deklarálására (pl. `STACK[T]`). Opcionálisan, a `frozen` kulcsszó segítségével meg lehet jelölni egy típusparamétert invariánsként (pl. `STACK[frozen T]`).

A kovariáns típusparaméter a statikus típusrendszert bár el tudja rontani, de bizonyos esetekben (amikor pl. funkcionális módon használjuk a generikus osztályt, azaz csak kiolvasunk belőle adatokat és nem módosítjuk azt) hasznos tud lenni.

A kovariáns sablonparaméternek köszönhetően egy `STACK[T]` osztály esetén `STACK[INTEGER] <: STACK[ANY]`-nek, hiszen `INTEGER <: ANY`.

Több típusparaméterrel is működik a kovariancia, illetve amennyiben `DEQUEUE <: STACK`, akkor `DEQUEUE[INTEGER] <: STACK[ANY]`, stb.

`frozen` típusparaméter esetén (pl. `STACK[frozen T]`) invariancia áll fenn a típusparaméterre nézve: `STACK[INTEGER]` és `STACK[ANY]` között nincs altípus reláció.

Használat példa: legyen `LIST[T]` egy kovariáns típusparaméterrel rendelkező osztály és legyen egy szekvenciát feldolgozó osztály, amely `LIST[ANY]` típusú paraméterrel rendelkezik. Ennek az osztálynak átadhatunk egy `LIST[INTEGER]` példányt, hogy azzal dolgozzunk: nem kell explicit egy `LIST[ANY]` példányt létrehoznunk.

## Mi a név szerepe, és miért fontos az átnevezés az Eiffelben?

A név egy rutint, attribútumot, osztályt, paramétert, lokális változót, stb. azonosító karaktersorozat. Egy osztályban minden ilyen elemnek egyedi névvel kell rendelkeznie.

Az Eiffel programozási nyelv támogatja a többszörös öröklődést. Amennyiben több, eltérő (vagy csak eltérő implementációval rendelkező) feature-t probálunk megörökölni, akkor probléma ütközünk, ha ezek nevei megegyeznek. Az átnevezés nyújt erre egy megoldási lehetőséget: így megtarthatjuk a több örökölt feature-t, nem kell az egyik implementációt eldobnunk, csupán a továbbiakban az egyik implementációt egy másik név segítségével tudjuk majd elérni.

## Mi a különbség a nyilvános és a privát öröklődés között?

Nyilvános öröklődés során kialakul a bázistípus és az altípus között egy altípus reláció, míg privát öröklődés során csak a bázistípus strukturális elemei (rutinok, azok implementációi, attribútumok, stb.) kerülnek bemásolásra az öröklést végző osztályba, anélkül, hogy egy altípus reláció jönne létre.

Ennek következtében például az Eiffel `once` rutinjai le fognak tudni futni egyszer az eredeti osztályban, egyszer peddig abban az osztályban, ami megörökölte a `once` rutint, azaz összesen többször is le fognak tudni ezek futni.

Míg a C++ programozási nyelv esetén a privát öröklődés során a megörökölt metódusok láthatósági is megváltozik (privát lesz, ha korábban például publikus volt), az Eiffel programozási nyelvben a privát öröklődés nem változtatja

meg a láthatóságot: `{ANY}` láthatóságú feature-ök `{ANY}` láthatóságúak maradnak.

## Milyen problémákat vet fel a többszörös öröklődés?

Lásd Hogyan, milyen módokon örökölheti meg az A osztály a B osztály egy f feature-ét, ha C és D a B gyermeke, valamint A a C és a D gyermeke? Milyen eseteket különböztethetünk meg?

Hozzáfűzés:

A többszörös öröklődés nem csak diamond inheritance esetén okoz problémákat. Névütközések (egymástól független, csak véletlenül azonos névvel rendelkező) feature-ok között is lehetnek: több osztály egymástól függetlenül deklarálhat egy azonos nevű feature-t. A diamond inheritance problémához hasonlóan itt is átnevezéssel vagy `undefine` segítségével oldható meg a probléma, a különbség annyi, hogy itt nincs szükség `select`-re, hiszen nincsen egy közös bázistípus, amiben a névütközés alatt álló feature deklarálva van.

## Mik az expandált típusok?

Más néven kifejtett típusok, ezek a referencia típusokon kívül a másik típus fajta az Eiffel programozási nyelvben.

A referencia típusok példányai a heap memórián helyezkednek el, és átadáskor vagy változóban való tároláskor csak egy pointer (referenciát) tárolunk, ami a példányhoz tartozó memóriaterületre mutat. Ezzel szemben a kifejtett típusok kifejtésre kerülnek az őket tároló egységbe (például az őket tartalmazó osztályba, vagy a stack-be). Így a kifejtett típusok is elhelyezkedhetnek a heap memóriában, amennyiben egy olyan osztálypéldány részét alkotják, amely a heap-en található. Amikor viszont egy lokális változót készítünk, amelynek kifejtett típusa van, akkor ez a lokális változó közvetlenül a stack-en fog elhelyezkedni: nem egy mutató lesz a heap-re.

Amíg a referencia típusok "call-by-sharing" módon működnek, azaz például paraméterátadásnál, vagy egy függvény visszatérési értékénél egy referencia (pointer) kerül átadásra (amin keresztül el tudjuk érni az adatokat), addig a kifejtett típusok esetében az egész osztálypéldány (annak összes adattagja) átadásra kerül (call-by-value, érték szerinti átadás) és az adatok közvetlenül, egy pointer követése nélkül is elérhetőek. Így például egy 64 bites pointer helyett lehet, hogy 128 bit memóriát kell átadni, amennyiben a kifejtett osztály 4db 32 bites integer változót tartalmaz.

Az egyenlőségvizsgálat is eltérő módon működik referencia, illetve kifejtett típusok esetén. Míg referencia típusokon az `=` operátor azonosságot vizsgál, azaz azt, hogy a két osztálypéldány azonos memóriaterületet foglal-e el, addig az `=` operátor kifejtett típusok esetén a `~` operátorral azonos módon működik, az "egyedi" egyenlőségvizsgálatot (avagy alapértelmezett implementáció esetén a sekély egyenlőségvizsgálatot) végzi el.

Sekély másolás esetén a másolandó osztálypéldány referencia típusú adattagjainak a referenciája (memóriaterület mutatója) kerül lemásolásra, nem pedig az összes adattagjuk (a tartalmuk). Azaz aliasing alakul ki, több mutató fog azonos memóriaterületre mutatni. Ezzel ellentétben kifejtett típusok esetén az adatok kerülnek lemásolásra, mint mély másolás esetén.

Továbbá kifejtett típusok bár leszármazhatnak más osztályokból (altípusok reláció kialakulásával együtt vagy akár nélkül privát öröklődés esetén), addig kifejtett típusokból nem lehet olyan módon leszármazni, hogy az altípus reláció kialakuljon: csak nem konform leszármazásra van lehetőség egy kifejtett típusból.

Kifejtett típusok mivel nem referenciák/pointerek segítségével működnek, így a `Void` értéket sem vehetik fel: olyanok, mintha `attached`-ek lennének.

## Hogyan számoljuk ki egy osztály invariánsát? (. . . öröklődés. . . )

Lásd Milyen szerződések vonatkoznak az Eiffel rutinokra? Hogyan befolyásolja ezt az öröklődés? Hogyan a felüldefiniálás? Magyarázd el a kapcsolódó fogalmakat és szabályokat. Fogalmazd meg a szerződéseket Hoare-hármasok segítségével!

## Mit értünk sekély, illetve mély másoláson?

Egy osztálypéldány sekély másolása esetén az osztálypéldány adattagjait egy másik osztálypéldány adattagjainak adjuk értékül "közvetlenül":

- a referencia típusok esetében a mutatók kerülnek lemásolásra, aliasing alakaul ki: több referencia fog azonos objektumra, azonos memóriaterületre mutatni
- kifejtett típusok érték szerint kerülnek lemásolásra: a kifejtett típus értékei kerülnek átadásra, hiszen itt nincsenek pointerek, amikkel aliasing alakulhatna ki. Amennyiben a kifejtett típus egy referencia típusú adattagal rendelkezik, akkor a fent említett módon aliasing alakul ki, hiszen a referencia típusnak csak a referenciája (mutatója) kerül lemásolásra.

Egy osztálypéldány mély másolása esetén viszont nem alakul ki aliasing: minden referencia típusú adattag mély másoláson megy keresztül, az adattagjai kerülnek lemásolásra, nem pedig a referencia. Így a referencia típusok is kifejtett típusokhoz hasonlóan teljesen duplikálva lesznek, aliasing nélkül.

A másolásokra vonatkozik egy megkötés: az egyedi, sekély, illetve mély másolás lefutása után a két osztálypéldánynak egyedi, sekély, illetve mély egyenlőségvizsgálat szerint meg kell egyeznie. Ez a beépített sekély és mély másolás esetén természetesen mindig teljesül, hisz ezek előre (helyesen) megvalósított műveletek.

## Milyen lehetőségek vannak a referenciák nem-ürességének leírására az Eiffelben?

Lásd Mit jelent az attached (az Eiffel szabványban "!") és a detachable (a szabványban "?")?

## Mi az F-bounded polimorfizmus? Mutass rá példát Javában!

F-bounded polimofizmusnak hívjuk azt, amikor a típusparaméter meg van szorítva a típusparaméter által.

Általában olyan esetben jön elő, ahol egy osztály/interfész konkrét implementációja mint egy típusparaméter van bekérve egy bázisosztályban/interfészben, hogy elérhető legyen ebben a szülőosztályban. Így különböző műveleteket tudunk deklarálni, amik majd a konkrét implementációban helyesen lesznek típusozva (pl. `Comparable` esetén a `compareTo` a megfelelő típusú paramétert várja ennek köszönhetően).

Példa: `<T extends Comparable<T>> T max(List<T> list)`
Vagy általánosabban: `<T extends Comparable<? super T>> T max(Collection<? extends T> input)`
Ez azt jelenti, hogy a típusparaméternek meg kell valósítania a `Comparable` interfészt olyan módon, hogy a `Comparable` interfész típusparamétere a `T`-vel egyenlő típust kap. Tehát értelmezve kell lennie az összehasonlításnak a `T` típuson, `T` típuspéldányok között.

Másik példa: ~~~java class Linked<T extends Linked> {...} class LinkedClass extends Linked {...}

class DoublyLinked<T extends DoublyLinked> extends Linked {...} class DoublyLinkedClass extends DoublyLinked {...} ~~~

## Mit jelent a "frame condition" kifejezés? Hogyan írunk ilyet az Eiffelben?

Lásd Mire való az only az Eiffelben (illetve a strip a jelenlegi implementációkban)?

## Milyen kifejezések szerepelhetnek értékadás baloldalán az Eiffelben?

- Lokális változók
  - A (rutin) paraméterek nem számítanak lokális változóknak; a paraméterek felülírására nincsen lehetőség (lokális változó bevezetése szükséges, amennyiben ilyenre van igény).
- Attribútumok (adattagok)
  - Fontos megjegyezni, hogy az Eiffel programozási nyelvben attól még, hogy láthatunk egy attribútumot és ki tudjuk olvasni annak az értékét, nem feltétlenül írhatjuk is azt: egy attribútum írására (értékének módosítására) csak annak az osztálypéldánynak van jogosultsága, amelyikhez az attribútum tartozik. Azaz attribútumok írásakor csak a `Current.xyz` attribútumok írhatóak, ahol `xyz` az attribútum neve (viszont a `Current.` kiírása nem szükséges).
- `Result` kulcsszó (visszatérési érték megadásához, egy függvényen belül)

## Mire jó a debug utasítás az Eiffelben?

A `debug` utasítás segítségével feltételesen futtathatunk le egy blokkot (utasítássorozatot): amennyiben a `debug` kulcsszó után megadott "debug key" be van kapcsolva a fordításkor, akkor (és csak akkor) kerül be a programba az

utasítássorozat. Így a hibakeresési funkciók (például extra kiírások, extra logolás) könnyen ki- és bekapcsolható a forrásfájlok módosítása nélkül, csupán a bekapcsolt "debug kulcsokat" kell módosítani.

```
debug (Debug_key, ...)
  ...  -- Lefut, ha a Debug_key be van kapcsolva
end
```

## Hogyan történik a paraméterátadás az Eiffelben? Térj ki az expandált típusokra is!

A referencia típusok call-by-sharing módon kerülnek átadásra, azaz egy pointert adunk át, amely a megfelelő memóriaterületre mutat. Aliasing alakul ki, több referencia is mutat így ugyan arra a memóriaterületre. Előnye, hogy nem másolódik le a teljes objektum, csupán egy 32-64 bites pointer kerül átadásra.

Ezzel ellentétben a kifejtett típusok call-by-value módon kerülnek átadásra, azaz az értékeik (adattagjaik értékei) lemásolásra kerülnek (sekély módon) és ezek az értékek kerülnek átadásra. Ennek következtében előfordulhat, hogy többszáz bájtnyi adatot kell átmásolni kifejtett típusú paraméterek használata esetén. Sekély másolás miatt itt is történhet aliasing: amennyiben a kifejtett típusnak van egy referencia típusú adattagja, akkor az adattag által mutatott memóriaterület nem kerül lemásolásra, csupán a maga a pointer, azaz aliasing alakul ki: több referencia fog az adott memóriaterületre mutatni.

Fontos megjegyezni, hogy Eiffelben a formális paraméterek nem írhatóak, csak olvashatóak. Amennyiben új értéket szeretnénk nekik adni, akkor egy lokális változót kell inkább használnunk.

Referencia típusú paraméterek esetén vigyázni kell, mert előfordulhat, hogy több paraméter is azonos memóriaterületre mutat, azaz aliasing lép fel. Előfordulhat például, hogy a kitüntett paraméter (`Current`) és az egyik formális paraméter megegyező osztálypéldányra mutat, ami esetlegesen nem kívánt működéshez vezethet, amennyiben nem készülünk fel rá (például egy osztás művelet megvalósításakor).

## Milyen elvárások fogalmazhatók meg egy egyenlőségvizsgálat műveletre?

Az egyenlőségvizsgálat műveletnek egy ekvivalenciarelációt kell definiálnia: reflexív, tranzitív, szimmetrikus.

Továbbá konzisztenciát is elvárunk az egyenlőségvizsgálattól: ha nem változtatjuk sem `x`-et, sem `y`-t, akkor az `x ~ y` művelet eredményének sem kéne változnia.

Az altípusosság egy részbenrendezés, így az egyenlőségvizsgálattal sokszor nehezen hozható összhangba.

Monomorf megközelítés: egy osztálypéldány csak olyan példánnyal tekinthető egyenlőnek, amelyik pontosan ugyan ahhoz az osztályhoz tartozik. Az adott osztály altípusának egy példánya nem tekinthető egyenlőnek a bázistípus egy példányával. Ez a megközelítés általában adathordozó objektumok esetén hasznos, a szimmetria így triviálisan teljesíthető. Ilyenkor általában nem is kívánatos az altípus reláció megjelenése és hasznosabb lehet a privát öröklődés vagy a kompozíció használata öröklődés helyett.

Polimorf megközelítés: egy bázisosztályban megadhatjuk az egyenlőségvizsgálat működését, amelyet aztán az alosztályok nem változtatnak meg (maximum hatékonyabbá tesznek egyes speciális esetekben). Ez olyankor hasznos, amikor a bázisosztály valamilyen általános koncepciót ábrázol, aminek több lehetséges megvalósítása lehet: például lehet a `List` a bázisosztály és tekinthetünk egy `LinkedList` és egy `ArrayList` példányt egyenlőnek, ha azonos indexen egyenlőnek tekinthető elemek helyezkednek el.

## Mi a különbség egy exportált és egy nem exportált feature szerződésében? Mutasd be Hoare-hármas segítségével!

Amennyiben az exportáltság arra utol, hogy mások (másik osztályok) számára látható az adott feature, akkor a szerződésükben nincs különbség, csupán a szerződés megváltoztatása során vannak eltérések: míg egy nem exportált feature szerződésének megváltozása esetén csupán az osztályon belül kell ellenőrizni, hogy továbbra is helyesen használjuk-e az adott feature-t, addig egy exportált feature-t bármelyik másik osztály is használhatja, így sokkal több helyen kell ellenőrizni azt, hogy a változtatás után is helyes-e a programunk. Ez az ellenőrzés nem feltétlenül kivitelezhető, például ha egy könyvtárat írunk, amely más, nem általunk fejlesztett programokban van felhasználva.

Ez valójában nem teljesen igaz, hiszen egy nem exportált feature öröklődés során exportálható.

## Mit értünk multimethodon?

Lásd minta válasz: Mit értünk multimethodon?

Multiplemethod, más néven multiple dispatch alatt a dynamic dispatch olyan változatát értjük, ahol nem csak egy kitüntett paraméter dinamikus típusa dönti el, hogy melyik rutin implementáció fusson le, hanem több ilyen kitüntett paraméter is van (akár az összes paraméter is lehet kitüntett paraméter).

A legtöbb nyelv nem támogatja a multimethod-ot, azaz egy `a.b(c)` metódushívás esetén a `c` paraméter statikus típusa, illetve az `a` paraméter dinamikus típusa dönti el, hogy a melyik implementáció fusson le.

A legtöbb nyelv nem támogatja a multimethod-ot, azaz egy `a.b(c)` metódushívás esetén az `a` és `c` paraméter statikus típusa dönti el, hogy melyik metódus fusson le, az `a` (kitüntett) paraméter dinamkus típusa pedig azt dönti el, hogy a metódusnak melyik implementációja fusson le.

Ezzel ellentétben multiple dispatch esetén `c`-nek nem a statikus, hanem a dinamikus típusa alapján van kiválasztva, hogy melyik implementáció fusson le, azaz mind `a` és `c` dinamikus típusa számít, ezek a kitüntett paraméterek.

A multiple dispatch például a bináris műveletek létrehozásakor tud nagyon hasznos lenne: segítségükkel könnyen megvalósítható egy egyenlőségvizsgálat művelet: minden osztálynak csupán definiálni kell egy megfelelő (az osztály típusával megegyező) típusú paraméterrel ellátott egyenlőségvizsgálat műveletet. Ennek a műveletnek a használata során a multiple dispatch mindig a legjobban illeszkedő implementációt fogja lefuttatni (közben gondoskodva a szimmetriáról).

## Mik a virtuális típusok? Mire jók?

A legtöbb programozási nyelv nem rendelkezik virtuális típusokkal, azonban a Scala nyelvben a virtuális típusok részlegesen meg vannak valósítva (a teljes körű megvalósítás a statikus típusrendszer elrontását eredményezné, azonban a részleges mevalósítás nem).

Virtuális típusok segítségével a típusparamétereknél bővebb lehetőségeink vannak a típusok testre szabására. Egy (absztrakt) bázisosztályban deklarálunk (de nem definiálunk) egy típust, egy adattaghoz hasonlóan. Ezt a típust a leszármazott osztályokban tovább specializálhatjuk, finomíthatjuk, végül pedig az implementáló osztályban definiálhatjuk: megadjuk, hogy pontosan melyik típusnak legyen megfeleltetve a korábban deklarált típus.

```
//Pair osztály típusparaméterrel
class Pair[L,R]( left: L, right: R )
val v = new Pair[Int,String](1,"hi")
v.right

//Pair osztály virtuális típussal
abstract class Pair {
    type L
    type R
    val left: L
    val right: R
}
//Ez itt egy leszármazás és konstruktor hívás egyben
val v = new Pair{ type L=Int; type R=String; val left=1; val right="hi" }
v.right

abstract class Linked {
    type MyType <: Linked
    var next: MyType = null.asInstanceOf[MyType]
}
class List extends Linked { type MyType = List }

abstract class DoublyLinked extends Linked {
    var prev: MyType = null.asInstanceOf[MyType]
}
class List2 extends DoublyLinked { type MyType = List2 }
```

A virtuális típusok az F-bounded polimorfizmushoz hasonló kifejezőerővel bírnak, azonban annál általánosabbak, könnyebben kezelhetőek és érthetőek.

## Mit nevezünk parametrikus polimorfizmusnak?

Parametrikus polimorfizmusnak nevezzük azt, amikor egy osztályt típusparaméterrel lehet testre szabni. Például a `STACK[T]` osztály deklarációban a `T` egy típusparaméter, ami helyére konkrét típusok megadhatóak az osztályból való leszármazás, vagy az osztály példányosítása során. Így egyetlen `STACK[T]` osztály deklarációjával megvalósítható mind egy `INTEGER`, mind egy `STRING` példányok tárolására képes verem.

A típusparaméterek Eiffelben kovariánsak, azaz amennyiben `INTEGER <: ANY`, akkor `STACK[INTEGER] <: STACK[ANY]`. A `frozen` kulcsszó segítségével viszont lehetőségünk van invariáns típusparamétert deklarálni: `STACK[frozen T]`.

A típusparaméterre számos megszorítás adható. Megadhatjuk például, hogy a típusparaméter helyére behelyettesített konkrét típusnak milyen szülőosztállyal kell rendelkeznie: `STACK[T -> HASHABLE]`. Ilyenkor a `T` típusú változókon használhatjuk a `HASHABLE` osztályban elérhető attribútumokat, rutinokat, hiszen ezeknek a `T` helyére behelyettesített konkrét típuson is értelmezhetőnek kell lennie az altípus reláció miatt.

Megadható több úgynevezett felső korlát is: `STACK[T -> {A,B}]`, ilyenkor a `T` helyére behelyettesített típunak mind az `A`, mind a `B` osztályból le kell származnia.

A lehetséges névütközések feloldása érdekében lehetőségünk van feautre-ök átnevezésére a megszorítások megadásakor: `STACK[T -> {A rename f as g end, B}]`

Továbbá, hogy a típusparaméternek megfeleltett osztálynak egy példányát létre tudjuk hozni a generikus osztályon belül, van lehetőségünk annak deklarálására, hogy elvárjuk, hogy egy adott nevű creation procedure létezzen az osztályon: `STACK[T -> A create make end]`

## Mit nevezünk altípusos polimorfizmusnak?

Az altípusos polimorfizmus az objektum orientált programozásnak bár nem kötelező eleme, szinte minden objektumorientált nyelv támogatja azt. Ilyenkor osztályokból le lehet származni, ezzel altípus relációt hozva létre. A leszármazás során megörököljük a bázisosztály metódusait, adattagjait, azokat esetleg specializálhatjuk, illetve a metódusok implementációját felülírhatjuk.

Az altípusos polimorfizmus lényege, hogy egy `Base` típusú referencia lehet, hogy `Sub` dinamikus típusú osztálypéldányra mutat, amennyiben `Sub <: Base`-nek.

A leszármazást, illetve a metódus felüldefiniálást sokszor valamilyen kulcsszó segítségével le tudjuk tiltani (pl. `final`, `frozen`).

Az altípusok képzésekor fontos szem előtt tartani a Liskov-féle helyettesítése elvet, miszerint a bázistípus példányoknak lecserélhetőnek kell lennie altípus példányokra anélkül, hogy a program viselkedése elromlana. Ezt úgy tudjuk elérni, ha a felüldefiniálás során a láthatóság, paraméterek és az előfeltétel kontravariáns, a visszatérési érték és az utófeltétel kovariáns módon változik csak.