Jegyzet 2 - Elméleti anyag.

Magas szintű programozási nyelvek 2 (Debreceni Egyetem)

Az objektumorientált paradigma (OOP)
Az objektumorientált (OO) paradigma középpontjában a programozási nyelvek absztrakciós szintjének növelése áll. Ezáltal egyszerűbbé, könnyebbé válik a modellezés, a valós világ jobban leírható, a valós problémák hatékonyabban oldhatók meg. Az OO szemlélet szerint az adatmodell és a funkcionális modell egymástól elválaszthatatlan, külön nem kezelhető. Alapját az absztrakt adattípusok jelentik. (osztályok fogják megvalósítani az absztrakt adattípust az oo nyelvekben)

Az egységbezárás elve
A valós világot egyetlen modellben kell leírni és együtt kell kezelni a statikus és dinamikus jellemzőket.
- Statikus jellemzők: adat
- Dinamikus jellemzők: viselkedés

Osztály
Az OO nyelvek legfontosabb alapeszköze, az absztrakt adattípust megvalósító osztály.
Az osztály egy absztrakt nyelvi eszköz. Rendelkezik attribútumokkal és metódusokkal.
- Attribútumok: statikus jellemzőket definiálnak. Segítségével írjuk le az osztályokhoz kötődő adatokat, adatmodelleket, megvalósítása tetszőleges bonyolultságú adatstruktúrák lehetnek.
- Metódusok: dinamikus jellemzőket definiálnak (eljárás-orientált nyelvekben ~ alprogramok). Eljárás metódusok; függvény metódusok

Objektum
Másik alapeszköz az oo nyelveknek. Konkrét nyelvi eszköz, amely mindig egy osztály példányaként jön létre. Önmagukban nem léteznek. Az objektum létrejöttét ***példányosítás***nak nevezzük. Egy osztálynak egyszerre több példánya is lehet, amelyek az osztályhoz hasonló adatstruktúrával és viselkedésmóddal rendelkeznek. A memóriában helyezkednek el, ezért van memóriabeli címük, ahol tárolódik az objektum adatstruktúrája.

Bezárás,láthatóság
Az osztályok eszközeinek a láthatóságát szabályozza. Az eddigi fogalmaink szerint eszköznek tekintjük a az osztályban deklarált attribútumokat és metódusokat.
Alapvető három szint:
- publikus (public)
- védett (protected)
- privát (private) – láthatatlan öröklődés

Publikus: Az adott eszközt a definiáló osztályomon a belőle leszármazott/származtatott osztályokban és a kliens osztályokban is lehet látni és használni.
Védett: Az eszköz az adott osztályban, illetve, a belőle leszármazott/származtatott osztályokban érhető el.
Privát: az így definiált eszközök, csak azon helyen használhatók, ahol definiálva voltak.

Elérési szintek (láthatóság)				
Módosító	**Osztály**	**Csomag**	**Leszármazott**	**Összes osztály**
public	**X**	**X**	**X**	**X**
protected	**X**	**X**	**X**	**-**
módosító nélküli	**X**	**X**	**?**	**-**
private	**X**	**-**	**-**	**-**

Absztrakt adattípus
Olyan nyelvi egység, amely megvalósítja az egységbe zárás elvét, elrejti a típus reprezentációját. Használatával a programozó lehetőséget kap arra, hogy magasabb szinten gondolkodjon a programról, ne az alacsony szintű megvalósítási kérdések legyenek a meghatározók.

Objektum állapota
Az objektumhoz tartozó memóriaterületen tárolt bitsorozat.
Az objektum a *kezdőállapot*át példányosításkor kapja meg.

Öntudat
Minden objektum csakis önmagával azonos, minden más objektumtól különbözik.

Objektumazonosító (OI)
Az objektumokat meg lehet egymástól különböztetni, tehát rendelkeznek objektumazonosítóval. A megkülönböztetést a nyelvekbe beépített mechanizmusok végzik.

Üzenetküldés
Az objektumok kommunikációja üzenetküldés formájában történik.
Az objektumok között interfészek segítségével párhuzamos üzenetküldésre van lehetőség. Az üzenetküldés megváltoztathatja az objektum állapotát. Üzenetküldés során egy adott objektum egy számára látható metódust meghív és megcímzi a másik objektumot. A fogadó objektum visszatérési értékkel vagy output értékkel válaszol.

Interfész
Azt írja le, hogy egy osztály példányainak mely attribútum és metódus jellemzői látszanak más objektumok számára. Az interfészt maga az osztály definiálja.

Osztály szintű és példány szintű jellemzők.
Attribútumok és metódusok is lehetnek osztályszintűek vagy példányszintűek.
<u>Példány szintű attribútum:</u>
Azok az attribútumok, amelyek minden példányosításkor elhelyezésre kerülnek a memóriában. Ők írják le a példánynak, az objektumnak az állapotát. Ezért ha egy osztály tartalmaz példány szintű attribútumot, minden egyes objektumnál a memóriában tárolásra kerülnek.
<u>Osztály szintű attribútumok:</u>
Nem a példányokhoz, hanem az osztályhoz kötődnek. Minden osztályhoz egyszer. Osztályonként egy darab van. Tipikus osztály szintű attribútum, az osztály kiterjedése, amely azt mondja meg, hogy az osztálynak az adott pillanatban hány példánya van.
<u>Példány szintű metódusok:</u>
A példány viselkedését határozzák meg. Használatuknál azt kell figyelni, hogy mindig meg kell adnunk azt a példányt, amelynek a metódusát használni szeretnénk.
Aktuális példány az az objektum, ahol a metódus éppen működik.
(this, self (aktuális példány jelölésére))
<u>Osztály szintű metódusok:</u>
Nincsen aktuális példány. Ezeket a metódusokat az osztály szintű attribútumok manipulálására használjuk.

Kommunikáció menetrendje: egyik objektum meghívja a másik objektum számára látható metódusát. A meghívott valamit csinál rá, válaszol az üzenetre.
Függvény esetében visszatérési értéket tekinthetjük válasznak, eljárás esetén válasznak tekintjük, hogy a meghívott objektum megváltoztatja az állapotát.

Üzenetváltáskor példányszintű metódust csak objektummal tudunk meghívni.

A példány szintű metódusok mindig valamilyen objektumon végeznek valamilyen műveletet. Ezt az objektumot ***aktuális példány***nak nevezzük (this, self szavakkal lehet rájuk hivatkozni).
- Osztályszintű metódusoknál nincs aktuális példány. Osztályszintű attribútumok manipulálására használják.

Metódusok csoportosítása.
Beállító és lekérdező metódusok
A példány szintű beállító metódusok használatakor az aktuális példány állapotot vált.
Legtöbbször egy ilyen beállító metódus egy attribútum értéket tud beállítani.
Lekérdező metódusok a példánynak az állapotáról adnak információt. Lekérdezik valamelyik, vagy akár több példány szintű attribútum értékét.
Beállító metódusok eljárás szerűek és paraméterként kell megadni nekik a beállítandó értékeket.
A lekérdező metódusok pedig függvényszerűek, és ott visszatérési értékként kapjuk meg a példányszintű attribútum értékét.
Beállító metódusok léteznek osztályszintű és példányszintűen is.

Példányosítás, objektum élettartalma.
Példányosításkor lefoglalódik a memóriában egy terület az objektum számára, ott elhelyezésre kerülnek a példányszintű attribútumok, és ezzel meghatározódik az objektum kezdőállapota. Az objektum ettől kezdve él, van öntudata és tudja melyik osztály példányaként jött létre.

Objektumok kezdőállapotának beállítására az objektum orientált nyelvek egy speciális metódust szoktak használni. Ezt a metódust nevezik úgy, hogy **konstruktor.**

Öröklődés
Matematikai kapcsolat. Újrafelhasználhatóság eszköze. Osztályok közötti asszimetrikus viszony.
- Szuperosztály, alaposztály, szülőosztály
- Alosztály, származtatott osztály, gyermekosztály

Az alosztály örökli a szuperosztályának minden a bezárás által megengedett attribútumát és metódusát. Ezeket az örökölt attribútumokat és metódusokat az alosztály egyből fel is tudja használni.
Az alosztály új attribútumokat, metódusokat is definiálhat, átvett eszközöket átnevezheti, az átvett/örökölt neveket újra deklarálhatja, megváltoztathatja a láthatósági viszonyokat, metódusokat újra implementálhatja. Az öröklődés lehet egyszeres és többszörös....

Öröklődés
Az OO nyelvekben az osztályok között létező aszimmetrikus kapcsolat, az újrafelhasználhatóság eszköze. Az öröklődési viszonynál egy már létező osztályhoz kapcsolódóan hozunk létre egy új osztályt.
- Szülőosztály: más néven szuperosztály vagy alaposztály.
- Gyermekosztály: más néven alosztály vagy származtatott osztály.

Öröklődés során az alosztály átveszi (örökli) a szuperosztályának minden (bezárás által megengedett) attribútumát és metódusát, amelyeket azonnal fel is tud használni. Az alosztály ezen felül új attribútumokat és metódusokat definiálhat, az átvett neveket átnevezheti, újradeklarálhatja, vagy megváltoztathatja a láthatóságaikat és újraimplementálhatja a metódusokat.

Az imperatív objektum orientált nyelvek szétválasztják a fordítási és futtatási időt. A deklarációt fordítási időben végzik el, az öröklődés is itt történik. Egy új osztály tehát a fordítási időben jön létre és emiatt az öröklődés is itt történik -> az öröklődés statikus. A Smalltalk nyelvekben mindez a futási időben történik, azaz az öröklődés dinamikus.

Egyszeres és többszörös öröklődés
Egyszeres öröklődés során egy osztálynak pontosan egy szuperosztálya lehet, míg többszörös öröklődés esetén több is. Bármely osztálynak tetszőleges számú alosztálya lehet. Természetesen egy alosztály lehet egy másik osztály szuperosztálya, így egy ***osztályhierarchia*** jön létre. Az osztályhierarchia egyszeres öröklődés esetén fa, többszörös öröklődés esetén hálós hierarchia (aciklikus gráf). A JAVA az egyszeres öröklődést vallja. Az osztályhierarchiában kiemelt elem a **gyökérosztály**, melynek nincs szuperosztálya (Objectnek nevezzük Javaban és C#- ban). Meg kell említeni még a **levélosztály**, melynek nincs alosztálya. (Java-ban ezeket final, C#- ban sealed osztályoknak nevezzük)
Névütközés
Akkor fordul elő, ha a szuperosztályok között van 2 azonos nevű, amely miatt az azonosítás nem lehetséges egyértelműen.

Helyettesíthetőség elve, Liskov-elv:
Öröklődéshez kapcsolódóan az újrafelhasználhatóságnak egy jellegzetes megnyilvánulási formája. Egy leszármaztatott példánya a program szövegében minden olyan helyen megjelenhet, ahol az elől osztály egy példánya áll.

Példány: Csak egy osztályhoz köthető
Objektum: A példányosító osztályhoz, és az ő összes elődosztályaként tekinthetünk rá.

Polimorf (többalakú) metódusok –method overriding:
A polimorf metódusok fogalma az öröklődéshez kapcsolódik. Örökölt metódusok újraimplementálását jelenti. Az ilyen újraimplementált metódusokat egymás polimorf metódusának nevezzük.

Ha egy objektumnak egy polimorf metódusát hívjuk meg, akkor melyik implementáció fog érvénybe lépni?
Nyelvi mechanizmus mondja meg:

Kötés: Az a mechanizmus, amely megszólítás esetén megmondja, hogy az adott metódus melyik osztályban van implementálva.
- *Statikus kötés:* már a fordításkor eldől a kérdés, a helyettesíthetőség nem játszik szerepet. Megnézi, hogy azt az objektumot hogyan deklarálták -> a megadott objektum deklaráló osztályának metódusa fog végrehajtódni.
- *Dinamikus kötés:* a kérdés csak futási időben dől el. A megoldás a helyettesíthetőségen alapul.

A JAVA- ban és a C#- ban mindkettő benne van.

Két féle lehetőség:
- statikus: A kérdés már fordítási időben eldől, és ekkor a helyettesíthetőség már nem játszik szerepet. Az objektumoknak a deklarációját tudja figyelembe venni.
- dinamikus: A kérdés csak futási időben dől el. Itt a helyettesíthetőség is szerepet játszik. A futtató rendszer megnézi, hogy mely osztály példányaként hoztuk létre és megkeresi, hogy az a példányosító osztály mely implementációját ismeri.
  Két féle:
  - az öröklött nem implementálta újra a metódust
  - öröklött metódust újraimplementálta a metódust

  Azt a metódust fogja végrehajtani, amit a példányosító osztály ismer.

Az objektum orientált nyelvek egy része csak az egyik kötést alkalmazza. Más részt mind a kettő benne van. Egyik alapértelmezett, másik programozható. Különböző helyzetekben egyértelműen

megmondja, hogy most statikus, vagy dinamikus kötést kell alkalmazni. Osztályszintű (statikus kötés) és példányszintű (dinamikus kötés) metódusok.

Metódusok túlterhelése- method overloading:
Akkor beszélünk ilyenről, ha egy osztályon belül azonos névvel rendelkezik 2 vagy több metódus. Kell hozzá egy osztály, és osztályon belül azonos nevű metódusok. Ha a nevük megegyezik, akkor hogyan különböztetjük meg egymástól. Nem csak névvel rendelkeznek, hanem specifikációval is. A specifikáció alapján tudjuk megkülönböztetni. Eltérő specifikációval kell rendelkezniük.
Három féle módon különböztethetjük meg:

- formális paraméter száma
- számuk megegyezik, de más típusai vannak
- bár azonos típusai vannak, de eltérő sorrendben.

Ha nem tudjuk megkülönböztetni, akkor fordítási hiba.

Absztrakt osztály (abstract class):
Absztrakt osztályt nem lehet példányosítani. Tartalmazhat implementáció nélküli metódusspecifikációkat (viselkedési minták) .
Más osztályok ősosztályaként szolgálhatnak. Azzal hogy implementáció nélküli metódusspecifikációval rendelkeznek, a származtatott osztálynak implementálni kell ezt a viselkedési mintát. Megszólítás annyi, hogy a konkrét osztály nem tartalmazhat implementáció nélküli viselkedés mintákat. Ha egy osztályban létezik implementáció nélküli metódusspecifikáció, akkor az osztály csak absztrakt osztály lehet.

Az implementáció nélküli metódusspecifikációk lesznek az absztrakt metódusok.
Nem szoktak a hierarchia levél elemei lenni. Az osztályhierarchia felső részén szerepelnek.
Konkrét osztály:
Akkor konrkét,ha nem absztrakt ,illetve nem lehetnek benne absztrakt metódusok.
A hierarchia levél osztályai:
A levélosztályok nem szoktak absztraktak lenni. A nyelvekben van lehetőség arra, hogy az osztályhierarchia egyes osztályait levélosztálynak definiáljuk. Ebben az esetben a levél osztály tovább már nem örököltethető.
Egyes nyelvekben különböző néven:

- levélosztályok: final, sealed. (java, c#)

UML – Unified Modelling Language (Egységesített modellező nyelv)
Osztály diagramm: Az osztálydiagram egy statikus modell. A rendszerben használt osztályokat mutatja azok attribútumaival együtt. Az osztálydiagram tartalmazza továbbá az osztály szintű kapcsolatokat.
Téglalapok:

- felső rész: tartalmazza a nevét és valamilyen jellemzőjét.
- középső rész: attribútumok
- alsó rész: metódusok
-

Különböző előtagok:
+: publikus láthatóság
-: privát láthatóság
#: védett láthatóság
~: csomagszintű láthatóság

Mezőknek és típusoknak lehet jelölni a típusát.
Dőltbetűs az absztrakt (osztályok, metódusok)
Aláhúzott a static (attribútumok, metódusok)

Az UML diagrammban látható nyilak a kapcsolatok jelzésére szolgálnak.

1. **Függőség:** két elem között az egyiknek hatása van a másik elemben. Az egyik elemet függetlennek, a másik elemet függőnek nevezzük.
   Szaggatott nyíl: a függőtől a független elem felé mutat.
   Ezen belül 4 felé bontás:
   - Finomítás, részletezés:
   - Nyomkövetés:
   - Tartalmazás:
   - Kiterjesztés:

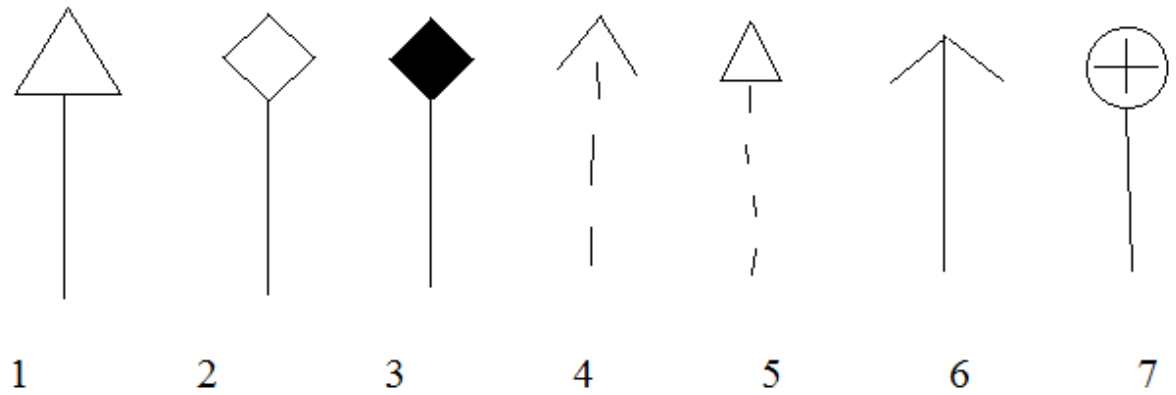
2. **Társítás:** elemek valamiféle kapcsolatát fogja meghatározni.
   2 fajta:
   - Ismertség: folytonos nyíl. nyíl irányába van az ismeretség. ha nincs nyíl, akkor kétirányú. 6-os nyíl (füzet)
   - Tartalmazás / gyenge és erős: 2 alcsoport:
     - aggregáció: rész az egészhez tartozik, de önmagában is létező entitás. pl.: kerék-autó viszony.
       2-es nyíl.
       nyíl végét a teljeshez kötjük. Gyenge tartalmazás
     - kompozíció: rész és egész között, amikor a rész önmagában nem létezhet. 3-as nyíl.
       Nyíl végét a teljeshez kötjük. Erős tartalmazás

3. **Általánosítás**
   - specializáció, öröklés: Aszimmetrikus viszony. alá-fölérendeltség
     alárendelt:?? fölérendelt:??
     UML diagramon osztályszerű megjelenés
     1-es nyíl.

4. **Megvalósítás:**
   - egy dolog meg akar valósítani egy másik dolgot
     Általánosításnak és függőségnek egyfajta keveréke. Logikai viszony két elem között. csak osztályszerű elemek között lehetséges. azt akarja kifejezni, hogy az egyik osztály biztosítson arról, hogy elvégez számára valamilyen feladatot.
     Interfész implementációk jellemzésére. 4-es és 5-ös nyíl.

+ -os nyíl (7.) : Osztályok egymásba ágyazhatóságát jelenti.

Az UML-ben tetszőleges szöveg (megjegyzés) rendelhető tetszőleges elemhez.



Karakterkészlet:
Java: UNICODE 16 bittes
C#: UTF-8
Karakterek csoportosítása: betűk, számok, egyéb karakterek.
Azonosítók.

Típusok:

- Java:
  - primitív típusok:
    - char: 16 biten ábrázolt adat típus, ami az értékét 0 és 65536 között
    - byte: egész szám típusok: 1 bájt -128-tól 127-ig
    - short: egész szám típusok: 2 bájt -32768-tól 32767-ig
    - int: egész szám típus: 4 bájt –(2^32)-1–től 2^32 -ig
    - long: egész szám típus: 8 bájt –(2^64)-1–től 2^64 -ig
    - float:
    - double:
    - boolean: logikai értékek típusa. két érték: true és false
  - referencia típusok:
    - felsorolás
    - osztály
    - tömb
    - interfész

Literálok
A literál olyan programozási eszköz, amelynek segítségével fix, explicit értékek építhetők be a program szövegébe. *Két komponense van:* típus és érték.
egész, lebegőpontos, logikai, karakter, sztring, null literál.
- Egész: Az egész literál felírató decimális, hexadecimális, oktális és bináris alakban. Egy egész literál long típusú, ha az ASCII L vagy l betű az utótagja, egyébként int típusú. Az aláhúzás karakterek megengedettek az egész szám számjegyei között elválasztó (csoportosító) karakterekként.
- Lebegőpontos: részei: egész rész, tizedespont, törtrész, kitevő rész, típus jelző utótag. Egy lebegőpontos literál 10- esen kívül felírható 16- os számrendszerben is. A decimális lebegőpontos számokban legalább egy számjegynek és vagy egy tizedespontnak vagy egy kitevőnek vagy egy típusjelző utótagnak szerepelnie kell. A hexadecimális számokban kötelező a kitevő! Float típusú, ha f van a végén, amúgy double. Ezek olyan értékeket vehetnek fel, amelyek megfelelnek az IEEE 754- es szabványnak. A float 32 biten tárol egyszeres pontossággal, a double 64 biten kétszeressel.
- Logikai: A logikai típusnak két értéke van. Az ASCII karakterekből álló 'true' és a 'false'.
- Karakter literálok: A karakter literálok egy karakter egy escape szekvencia aposztrófok közé zárva. A karakter literál mindig char típusú.
- Sztring literál: A sztring literál tetszőleges sok (nulla vagy több) karakter idézőjelek közé zárva.
- null literál: pl.: null szemantikája: Egy referencia típusú eszköz, amely nem hivatkozik semmire.

Elválasztó karakterek
12 db elválasztó karakter van ( ) { } [ ] ; , . … @ :: Ezek mindig más nyelvi eszközöket fognak elválasztani egymástól.

- C#:
  - primitív típusok:
    - byte
    - short
    - integer
    - long
    - single (float, 4 byte)
    - double (float, 8 byte)
    - boolean

C# típusrendszere:

- referencia típusok:
  osztályok, többek közt a System.Object, a System.ValueType és a System.Enum
- érték típusok:
  struktúrák és enumerációk (felsorolások)

Java Utasítások:

Deklarációs utasítások:

Mind a két nyelv ismeri a deklarációs utasításokat. Programozónak minden programozási eszközt deklarálni kell. Épp úgy tartozik a változókra,mint egyéb eszközökre. Deklaráció általános alakja:

[módosítók] típus azonosító [az eszközt jellemző paraméterek]

Jellemző paraméterek lehetnek:

- metódusok esetén formális paraméter lista
- változó esetén kezdőértékadás

Kifejezések

A kifejezések szintaktikai eszközök, arra valók, hogy a program egy adott pontján ismert értékekből új értékeket határozzunk meg. *Két komponense van:* típus és érték. Alapvetően kétféle feladata van: végrehajtani a számításokat, és visszaadni a számítás végeredményét.

Összetevők:

· Operandusok: Az értéket képviselik. Lehet literál, nevesített konstans, változó vagy függvényhívás az operandus.
· Operátorok: Műveleti jelek (pl. *, /, +, -).
· Kerek zárójelek: A műveletek végrehajtási sorrendjét befolyásolják.

Példa a kifejezésre: a + sin(0) Operandos operátor Operandus

Attól függően, hogy egy operátor hány operandussal végez műveletet, beszélhetünk:
· egyoperandusú (unáris pl. (!a)),
· kétoperandusú (bináris pl. (2+3)) vagy
· háromoperandusú (ternáris pl. (kif1 ? kif2 : kif3)) operátorokról.

Implicit konverzió

Amikor egy aritmetikai operátor egyik operandusa egész a másik pedig lebegőpontos, akkor az eredmény is lebegőpontos lesz. Az egész érték implicit módon lebegőpontossá konvertálódik, mielőtt a művelet végrehajtódna.
Példa: Egy egész számot osztunk egy valós számmal, a végeredmény automatikusan valós lesz.

A konverziót ki is "kényszeríthetjük" explicit konverzióval. A programozó ebben az esetben a kifejezés értékére "ráerőltet" egy típust. Ez a **castolás**.

Kifejezés kiértékelése

Azt a folyamatot, amikor a kifejezés értéke és típusa meghatározódik, a kifejezés kiértékelésének nevezzük. A kiértékelés során adott sorrendben elvégezzük a műveleteket, előáll az érték, és hozzárendelődik a típus.

Konstans kifejezés

Azt a kifejezést, amelynek értéke fordítási időben eldől konstans kifejezésnek nevezzük. Létrehozására a final módosítót használjuk, így érjük el, hogy az adattag értékét nem változtathatja meg senki.

1) Elágaztató utasítások

- if (feltétel) utasítás [else utasítás]

A feltétel nem lehet numerikus, csak **logikai típusú**. Nem lehet elhagyni a feltételt! Szintaktikája hasonló a C-hez. Rövid alakja csak az if-es ágat tartalmazza, hosszabb alakban szerepel az else

kulcsszó is. Az if-hez tartozó utasítás, vagy az utasítás helyén álló blokk akkor hajtódik végre, ha a kerek zárójelben található logikai kifejezés igaz értékű. Az else ág utasítás helyén álló blokk akkor hajtódik végre, ha a kerek zárójelben található kifejezés hamis, és van else ág.

- Switch:

switch (egész_kifejezés) {case egész_literál: utasítások [case egész_literál: utasítások] …
[default: utasítások]}

Szerkezetileg olyan, mint a C-beli. A kifejezésnek egésznek, vagy egésszé konvertálható kifejezésnek kell lennie, kerülhet ide még sztring típusú kifejezés is.
A () –t követi a {} –el határolt switch blokk. Vannak benne címkézett utasítások, a case, illetve a default. A case címkéknél a kulcsszó mögött álló literálnak egész típusúnak vagy sztring típusú konstans kifejezésnek kell lennie. Bármely utasítás címkézhető, egy címkéhez több utasítás is rendelhető. Tetszőleges sok case címkével ellátható. A default címke csak egyszer használható a switch blokkban, viszont a helye az nem kötött. Azonos értékű case cimkék nem szerepelhetnek a switch blokkban.

C#:
switch ugyan úgy néz ki mint a java-ban.

```
switch(kifejezés){
        case címke
        case címke:
                utasítás;
                utasítás;       →switch-szelekció
                …
        case címke:
        …
        case címke:
                utasítás;
                utasítás;       →switch-szelekció
                …
        default:
                utasítás;
}
```

Ezt a switch utasítást úgy kell megírnunk, hogy a switch szelekciós vége elérhetetlenek legyenek, tehát ne kerüljön oda vezérlés.
Switch-szekció végének az elérhetetlenségét biztosítja továbbá:

- goto case_címke;
- goto default;
- végtelen ciklus

2) Ciklusfajták

- while (feltétel) utasítás;

A while, do…while ciklusok megegyeznek a C–belivel. Szemantikájuk annyiban tér el, hogy a () zárójelek közzé **csak logikai kifejezés**t írhatunk, ami igaz vagy hamis értékként értékelődik ki.

- For –ból több féle is létezik:
  1) for ([P1]; [P2]; [P3]) utasítás;
  2) for (típus változó : kollekció) utasítás; (~ foreach)

1) 3 kifejezéses:
Először kiértékelődik az első kifejezést, utána kiértékelődik a 2. kifejezés, amennyiben igaz, akkor végrehajtódik a ciklus magja, majd utána kiértékelődik a 3. kifejezés, majd utána a kif2, majd kif3… A **kettes kifejezés mindig logikai**. Az első kifejezés lehet deklarációs utasítás alakú is. A {}-ben deklarált változó a ciklus fejében és törzsében lesz lokális változó, máshonnan nem tudjuk hivatkozni. Emiatt kell neki adni kezdőértéket, és csak egyféle típust tudunk megadni.

2) foreach:
for(típus változó :objektumreferencia) // a típus és a változó együtt lokális változó
    utasítás
Pl.: for(int item : a) // ahol a egy int tömb
    sout (item)
Ezt a ciklust Java- ban foreach ciklusnak nevezzük. Ennek a referenciának egy olyan objektumra kell hivatkoznia, amely bejárható. Tipikusan ilyenek a tömbök és a kollekciók. A változó alapvetően azonosító. A lokális változó típusa vagy a bejárandó objektum deklaráló típusa, vagy annak valamilyen ősosztálya lehet. A bejárandó objektum minden egyes elemét pontosan egyszer érinteni fogja.

C# foreach ciklus
foreach(típus változó in objektum_referencia)
    utasítás;
Az objektum_referenciának egy olyan objektumra kell hivatkoznia, amely nmegvalósíta a System.Collections.IEnumerable vagy System.Collections.Generic.Ienumerable<T> interface-t bejáró ciklus. Választ magának egy gyűjteményt vagy kollekciót, ahol különféle objektumok vannak tárolva. erre a kollekcióra hivatkozik a ciklus fejében az obejktum referencia.
A ciklus felveszi a bejárandó objektumon az értékeket, és minden egyes értéken végig megy.
pl.:tömbnél az indexek növekvő sorrendje határozza meg a bejárást.

C#:
```
int[] fib = new int [] {0,1,1,2,3,5,8,13,21}

foreach(int element in fib){
    System.Console.WriteLine(element);
}

for(int i = 0;i<fib.Lenght;i++){
    System.Console.WriteLn(fib[i])
}

for(kif1;kif2;kif3){
    …
    …
    continue; itt ér véget az utasítás, és a kif3-ra megy a vezérlés.
    …
    …
}
```
3) Vezérlésátadó utasítások
- break [címke] (címke nélkül a legbelső blokkra hivatkozik):
  A címke egy utasítás címkéje, annak a ciklusnak a futását szakítja meg, amelyhez ez a címke tartozik.
- continue:
  Ciklusutasításokban használható. Befejezi az adott iterációját a ciklusnak, és a következő iterációnak adja át a vezérlést.
- continue [címke] (a megadott címkére vonatkozik):
  A címkének olyan ciklusutasítást kell címeznie, amelyhez tartozik continue, vagy tartalmazza azt. A continue a címke utasítás iterációját fejezi be.
- return: Metódusok befejeztetésére szolgál, void típusúakra.
- return kifejezés: A nem void típusú metódusok befejeztetésére szolgál.

C#:

- goto címke;
- goto default;
- goto case címke;
- yield return [kifejezés];
- yield break;

Yield: feladattunk hogy az elemeket sorban dolgozzuk fel, de egy megfelelő feltételnek ne az összes elemet dolgozzuk fel. Jegyezze meg azt a pozíciót, ahol végrehajtottuk a return vagy break utasításokat. Ettől a pozíciótól folytassa a munkát.

Kivételkezeléssel kapcsolatos utasítások:
try-catch, try-catch-finally, try-finally
erőforráskezelő try utasítás (try with resources)
throw

Checked és unchecked utasítás (C#)
Kifejezés kiértékelésekor túlcsordulás fordulhat elő
Program a túlcsordult értékkel számoljon-e tovább vagy pedig ne azzal.

```
int i = 10;
Consolse.WriteLine(2147483647 + ten)

checked{
        int i = 214748367 + ten;
        Console.WriteLine(i);
```

Szinkronizációs utasítások:

- await (C#)
- fixed (C#)
- lock (C#)

Osztályok Java-ban:

Az osztályok a referencia típusok közé tartoznak. Az osztály szolgál az absztrakt adattípus megvalósítására.

<u>Osztály definíciója:</u>
[módosító]… class név
[extends szülőosztály_név]
[implements interfésznév_lista]
törzs

extends:
Arra szolgál, hogy az osztályt elhelyezzek a Java osztály hierarchiájában. Itt kell megadni az osztály szülőosztályának nevét. Örökli azokat az eszközöket, amiket tud, amik meg vannak engedve számára. Ha nem adjuk meg, akkor alapértelmezés szerint a java.lang.Object osztálya lesz a szülőosztálya.

implements:
interfészek: viselkedés minta gyűjtemények. azért csak minták, mert nem szerepel bennük implementáció. Az osztályunk akar valamilyen viselkedés mintát használni, és ezeket kell felsorolni a listában.
Szülőosztály név csak 1 lehet, implements-be pedig többet is.

módosítók:
Funkciójukat tekintve csoportokra lehet osztani.
Osztályra vonatkozó láthatósági vagy bezárási módosítók:

- public
- protected
- private
- (semmi)

Külsőszinten csak két féle módosító: public vagy (semmi), beágyazva pedig mind a négyet.
public:
Összes kliens osztály látja és tudja használni.

(semmi):
Csomag szintű láthatósági osztály. Csomag egy valamilyen logikai csoportosítása az osztályainknak. Csak a vele egy csomagban lévő osztályokból hivatkozható.

protected:
Leszármazási hierarchiából hivatkozható, másrészt az egy csomagban lévőkre is.

private:
Az osztály csak azon a típuson belül hivatkozható, amelyben definiálták. Legerősebb adatrejtés.

Egyéb módosítók:

- abstract: lehet belső, és külső osztály egyaránt. Az osztályt a tervezője abstractnak definiálta. Nem lehet példányosítani, viszont más osztályok szülőosztálya lehet.
- final: az osztály nem örököltethető. nem lehet más osztály szülőosztálya. az osztály hierarchiának a levél osztályai lesznek.
- static: Az osztály statikus osztály lesz. A static kulcsszó csak beágyazott osztályoknál jöhet elő. A beágyazó osztályhoz fog tartozni.

Struktúrák C#-ban:

[[attribútum]…]
[módosító]… struct név [: interfész_lista]
törzs

módosítók: szintén csoportokba sorolhatók
Struktúrákra és osztályra vonatkozó láthatósági vagy bezárási módosítók:

- public
- protected internal
- internal
- protected
- private

Külső szinten: public és az internal
public: teljes elérhetőséget jelenti
private: adott típuson belül érhető el.
protected: „elvárt jelenség”, adott típuson belül, és az össze leszármazott osztályon használható

internal: - az egy assembly-be kerülő típusok

A különbség, hogy a java csomagokat használ, a C# assembly-ket.

C#-ban ha elhagyjuk a módosítót, akkor az alapértelmezés private lesz.

Osztályok a C#-ban:

[[attribútum]…]
[módosító]… class név [: [szülőosztálynév], interfész_lista]
törzs

egyéb módosítók:
- abstract: nem lehet példányosítani, de lehet örököltetni
- sealed: final megfelelője. levél elemek.
- static: vannak. bármely szinten szerepelhet. csak statikus tagokat tartalmazhatnak.
- partial: részleges osztály definíció. grafikus alkalmazásokban.

Amennyiben nem adjuk meg a szülőosztályt, akkor System.Object lesz a gyökérosztálya.

Attribútumok (tagok) a Java osztályokban:
- tagváltozók (member variables): osztályok törzsében definiált változók
  [módosító]… típus név [= kezdőérték];
  módosító: a láthatósági módosítók.
  Ezeken kívül a final és a static is szóba jöhet.
    - final: a tagváltozó csak egyszer kaphat értéket, és amikor ezt az értéket megkapja, akkor nem kaphat új értéket. ezzel nevesített konstansokat tudunk definiálni.
    - static: az adott tagváltozó osztály szintű, tehát csak egy példányban jön létre és az osztályhoz kapcsolódik. Ha nem szerepel a static, akkor példány szintű, ha igen, akkor static tagváltozó.

  Adható kezdőérték, de ha nem adjuk meg akkor létezik alapértelmezés, és ez típustól függ. referencia típusoknál a „Null”. egész típus esetén „0”, valós típus esetén: „0,0”, karakter típusként a nulladik ascii karakter és a boolean típusnál a „false” érték.

- lokális változók: blokkon belül definiált
- metódus deklarációban használt paraméterek nevei

Metódusok:

[módosító]… típus név ([f.p.l]) [throws kivételosztálynév_lista]
Törzs

Típus helyett megadható a void kulcsszó is, tehát a metódusnak nincs visszatérési értéke. Visszatérési típussal nem rendelkezik. A void típusú metódusokat utasításként vehetjük. Ahol void típus van, azt eljárás vesszük. Ahol típus szerepel az általában függvény.
Tipikus metódusnevek: runTest, getBeckground, getFinalData

Formális paraméter lista: [[módosító] típus név [,[módosító] típus név]…]
A módosítok, a működésbeli funkciókra vannak hatással.
Módosító leggyakrabban a final lehet.

Metódusok előtti módosítók:

- láthatóságig módosítók: public, protected, (semmi), private
- static: esetén osztály metódus, ha nem írjuk ki, akkor példány metódus
- abstract: absztrakt metódus esetén nem lesz implementációjuk, tehát a törzsük normális blokk helyett csak egy pontosvessző. Az osztály, amely absztrakt metódust tartalmaz, annak is absztraktnak kell lennie. Fordított irányban nem igaz.
- final

Metódusok túlterhelése (overloading)

Java nyelvben a metódusok túlterhelhetők. Ugyan abban az egy Java osztályban több ugyan olyan nevű metódusok vannak, akkor a formális paraméter listája különbözteti meg egymástól. 3 dologban tud különbözni:

- darabszám
- típus
- típusok sorrendje

Statikus metódusok hívása:

- osztálynév.metódusnév(a.p.l)
- objektumreferencia.metódusnév(a.p.l)

Pl.:Math.abs(3.0)

Nem statikus (példányszintű) metódusok hívása:

- objektumreferencianév.metódusnév(a.p.l)

Konstruktorok (consturctors):

[módosító]… név([f.p.l]) [throws kivételosztálynév_lista]
Törzs

- A konstruktorok is egyfajta metódusok, de annyiban különböznek, hogy ők a Java nyelv szabálya szerint nem öröklődnek.
- Típusjelzés nélküli metódus
- Névre vonatkozóan van egy megszorítás, miszerint a névnek meg kell egyeznie az osztály nevével.
- Módosító mind a 4 lehet (public, protected, private, (semmi)). Viszont nem lehet a többi (statick, abstract, final).
- Törzse egy valódi blokk.

Konstruktor működése:

- Minden osztály rendelkezik konstruktorral.
- Ha a programozó nem definiál konstruktort, akkor a fordító generál egy alapértelmezett konstruktort (default constructor)
- Publikus konstruktor, aminek 0 paramétere van, és működésképp annyit csinál, hogy meghívja a szülő osztályának a 0 paraméteres konstruktorát.
- A példányszintű tagváltozók beállítására szolgál.

Példányosítás után a példány alapállapotát be kell állítani, amelyet megtehetünk paraméterek segítségével, de ezt a célt szolgálják a konstruktorok is, amelyek típus nélküli – az osztály nevével megegyező nevű – módszerek. Ezen módszereknek csak a láthatóságát szabályozhatjuk. A konstruktorok a példányosításnál hívódnak meg és inicializálják a példányt.

Máshol közvetlenül nem hívhatóak meg. A programozó egy osztályhoz akárhány konstruktort írhat, ezek neve mindig túlterhelt, ezért a paramétereinek száma vagy típusa különböző kell, hogy legyen, különben fordítási hibát eredményezünk. A törzs általában blokk, végrehajtható és deklarációs utasításokból áll. A konstruktor törzse minden esetben egy másik konstruktor meghívásával kezdődik. A másik konstruktor a szülőosztály paraméter nélküli konstruktora lesz (ha a programozó mást nem mond). Az object osztály konstruktora már nem hív meg több konstruktort. A meghívott konstruktor lehet a szülő osztály konstruktora, vagy az adott osztály egy másik konstruktora. A konstruktort példányosításkor úgy használjuk, hogy a new operátor után a paramétereket felsoroljuk a paraméterlistában.
Például: Osztály példány = new Osztály(”paraméter”,”paraméter”);
A megfelelő konstruktort a fordító **paraméterlista illesztés**sel választja ki. Ha a programozó nem ad meg konstruktort, akkor a rendszer automatikusan felépít egyet, amely paraméter nélküli, törzse pedig üres. Ezt hívjuk **alapértelmezett konstruktor**nak. Ez a konstruktor alapértelmezetten kinullázza az adattagokat. Ha explicit módon adunk meg konstruktort, akkor a fordító azt fordítja bele a kódba, nem foglalkozik az alapértelmezéssel.

Inicializáló blokk:
Az osztály törzsén belül van, formális alakja egy blokk.

```
blokk
{
   [utasítás]…
}
```

Statikus inicializáló blokk:
Az osztály első hivatkozásakor lefut, és csak utána foglalkozik az osztály többi részével. Csak egyszer fut le.

```
static blokk

static {
   [utasítás]…
}
```

Paraméterátadás
A Javaban csak **értékszerinti** paraméterátadás van. Az információ áramlás iránya a hívótól a hívott felé zajlik olyan módon, hogy a hívó programegységben a paraméter rendelkezik értékkomponenssel, a hívott programegység pedig rendelkezik címkomponenssel, ahová a hívás folyamán bemásolódik. Primitív típusú értékek esetén a primitív érték másolódik át, referencia esetén a referencia érték másolódik át. A paraméterek száma lehet fix, vagy változó. Változó esetén a paraméterlistán a típusnév mögött 3 db pont van. Lehet kombinálni a fix és a változó paraméterlistát, még pedig úgy, hogy elől van a fix paraméter/ paraméterek, utána pedig egyetlen egy változó paraméter. pl.: system.out.printf(„%s: %d, %s%n”, name, age, adress);
A paraméterek kis kezdőbetűvel kezdődnek, szóösszetételek határán az első betű nagybetű. Paraméterek nevének a hatáskörén belül egyedinek kell lennie. A hatásköre a törzs és a specifikáció.

Paraméterkiértékelés
Paraméterkiértékelésen azt a folyamatot értjük, amikor egy metódus hívásánál egymáshoz rendelődnek a formális és aktuális paraméterek, és meghatározódnak azok az információk, amelyek a paraméterátadásnál a kommunikációt szolgáltatják. Paraméterkiértékelésnél a formális paraméter lista az elsődleges, ezt a metódus specifikációja tartalmazza, egy darab van belőle, aktuálisból viszont annyi van, ahányszor meghívjuk a metódust. Egymáshoz rendelésnél

az aktuálist rendeljük a formálishoz.

A Java nyelv az alábbi módon történik a paraméteregyeztetés: sorrendi kötés, számbeli egyeztetés, típusbeli egyeztetés.

Öröklődés (osztályok)

Osztályok között egyszeres öröklődést enged a JAVA nyelv.

	szuperosztály osztálymetódusát	szuperosztály példánymetódusát
alosztály osztálymetódusa	elrejti (hides)	fordítási hiba
alosztály példánymetódusa	fordítási hiba	felülírja (overrides)

Amikor 2 vagy több egymástól függetlenül definiált default metódus összeütközésbe kerül, vagy egy default metódus absztrakt metódussal ütközik, akkor a Java fordító fordítási hibát generál. Ilyen helyzetekben explicit módon felül kell írnunk a szuper metódusok metódusait.

C# osztályok tagjai (members):

- mezők (fields)
- konstansok (constants)
- tulajdonságok (properties)
- metódusok (methods)
- események (events)
- operátorok (operators)
- indexelők (indexers)
- konstruktorok (constructors)
- destruktorok (destructors)
- beágyazott típusok (nested types)

A tagok láthatósági szintjei:

- public
- protected internal
- protected
- internal
- private

Ha egyet sem adunk meg, akkor alapértelmezett a „private”.

Mezők (fields):

Egy osztály vagy egy struktúra létezhet osztályszintű vagy statikus mezőkkel egyaránt. Ezeket a mezőket deklarálni kell. Tartalmazhat módosítókat, típusnak és névnek kötelező lennie.

- public, protected internal, protected, internal, private
- static
- readonly: futási időben rendel hozzá értéket

Az osztály példányosítása nélkül is használható, ha ott van a static kulcsszó előtte.
Readonly módosító mezőkhöz csak az inicializáláskor vagy konstruktorral adható érték.
Futási időben rendel hozzá értéket a readonly szemben a konstanshoz, ami fordítási időben.

Konstansok:
Fordítási időben ismertek, futási időben nem változik az értékük. Onnan lehet felismerni, hogy a deklarációjukban szerepel a „const” kulcsszó a módosítók között. A felhasználó által definiált típusok (osztályok, struktúrák, tömbök) nem lehetnek konstansok. Ha saját konstanst

szeretnénk használni akkor azt readonly módosítóval kell ellátni. A C# nem támogatja a konstansok létrehozását. Külön típus a felsorolásos típus (enum).
Tulajdonságok
Lehetőséget biztosít privát mezők írására, olvasására, privát értékekből származtatott értékek kiszámítására.

Hozzáférők: lekérdező és beállító
get blokk
set blokk

Mindegyik mögött egy metódus áll.
T típusú tulajdonság esetén: T get_Prop(), void set_Prop(T value)

Automatikusan definiált/implementált tulajdonságok (auto-implemented properties)

[módosító]… típus név {[private] get; [private] set;}

Indexelők (indexers)

[módosító] … típus this[[f.p.l]] { hozzáférők }

Részleges osztályok (partial class) és részleges metódusok (partial methods)

Statikus konstruktorok (Static constructors)

- f.p.l. mindig üres
- nem lehet meghívni
- statikus mezők kapnak értéket(statikus inicializáló blokk)

Példánykonstruktorok (Instance constructors)
base -> szülőosztály konstruktora
this -> a fölötte lévő konstruktort hívja meg

Véglegesítők (destruktorok)
Olyan eszközök leprogramozása, ami memória felszabadulásával jár. Meghívódik a finalize metódus, ami végrehajtódik, ezáltal az objektum törlődik a memóriából. Amikor már kevés memória áll rendelkezésünkre, a futtató rendszer szabadítja fel a memória ezen részét, ahol a már nem használt objektumok elhelyezkednek: először a származtatott (gyerek) osztályokat, utána a szülő(base) osztályokat.

Metódusok

[módosítók] … típus név ([f.p.l.]) törzs

Paraméterátadási módok:

- értékszerinti (alapértelmezett, nincs külön jelölése)
- cím szerinti (ref kulcsszóval)
- eredmény szerintei (out kulcsszó)

Változó számú paraméterek kezelése:

- params kulcsszóval, a params utáni típus határozza meg, hogy milyen típusú paraméterek következnek.

Operátorok
public static típus operátor.(típus lhs, típus rhs) törzs

. helyére : - egyoperandusú: + - ! ~ ++ -- true false
- kétoperandusú: + - * / % & | ^ << >> == != <> <= >=

```
pl.: static public Complex operator+(Complex lhs, Complex rhs)
{
	return lhs+rhs;
}
```

Ha alá van húzva az operátor, akkor kétoperandusú.
Konverziós operátorok : public static {implicit|explicit} operator típus(típusnév) törzs

C# delegált
A delegált egy metódust címző referencia típus, amely a **System.Delegate** osztályból származik.
A delegált mindig egy osztály leszármazottja.

Minden delegált példány egy **hívási listá**t tartalmaz, tehát metódusokat, mint meghívható entitásokat (~ **mutatók**). Egy delegált példány szintű metódus számára egy ilyen meghívható entitás egy példányból és egy adott példányon operáló metódusból áll.

Statikus metódusok számára az entitás csak egy metódust jelent, a példány az <<null>>.
A delegált nem ismeri a hozzá tartozó metódusok osztályát, egyetlen követelmény a delegált és a metódus típusának **kompatibilitás**a. Tehát a delegáltak esetében **anonim** a **metódushívás**.

Egy delegált és a metódus típusa akkor kompatibilis, ha visszatérési típusok azonos és paraméterlistájukban a paraméterek száma, sorrendje, típusa és módosítóik azonosak.

Delegáltat hozzáadni és elvenni az eseményekhez hasonlóan a +, -, += és -= operátorok segítségével lehet.

Egy delegált példány a következők valamelyikét hivatkozhatja:
- statikus metódust,
- célobjektumot és a célmetódusát,
- egy másik delegáltat.

A delegált meghívása egy delegált és egy metódus esetén **egyszer**ű. Lefut a metódus és a delegált visszatér az eredményével. Ha közben kivétel következik be, amelyet a metódus nem kezel, akkor a megfelelő kivételkezelő keresésével a delegáltat meghívó metódusban folytatódik tovább, mintha a metódust közvetlenül hívta volna meg a delegált.

Többszörös delegált példánynál minden metódus lefut a felsorolás sorrendjében ugyanazon paramétereken. Ekkor a delegált értéke az utolsó metódus visszatérési értéke lesz, ha van neki. Kivétel bekövetkezés esetén a kivétel átadódik a delegáltat hívó metódusnak és a többi metódus már nem fut le.

Enumeráció, felsorolás
Az enum típusok **nevesített konstansok**, csak úgy mint a Java nyelvben.
Az enum, mint típus a **System.Enum** absztrakt osztályból származik.

Alapértelmezetten az első elem értéke 0, de lehetnek azonos értékek is.

Alakja:
[*attribútumok*] [*módosítók*] **enum** [**:** egész.típus] { [*attribútumok*] azonosító [= *konstans.kifejezés*] … }

Beágyazott osztály
Osztályokat más osztályokon és módszereken belül is lehet definiálni, amelyeket **beágyazott osztály**oknak hívunk. Egy osztály **tagosztály**ának nevezzük azt az osztályt, amelyet az osztály tagjaként definiáltunk. Az ilyen osztály ugyanúgy örökölhető és elfedhető.

Egy statikus tagosztály törzsében csak az aktuális példány tagjaira és a tartalmazó osztály statikus tagjaira hivatkozhatunk **minősítés nélkül**.
A statikus tagosztályokat és a nem beágyazott osztályokat **külső osztály**oknak hívjuk.

A példány és a környezet közötti kapcsolat a példányosítás pillanatában jön létre, az osztály definíciójának helyén éppen érvényben lévő aktuális példány és változóállapotok alapján.
Belső osztályok **nem tartalmazhatnak statikus tagokat**.

Egy belső tagosztály példányosítása esetén minden példányhoz tartozik egy tartalmazó példány, amely rögzített. Úgy viselkedik, mint egy második aktuális példány.
A beágyazás tetszőleges mértékű lehet. Az aktuális példányok elérése érdekében a **this** pszeudováltozót az osztály nevével helyettesítjük.

A this a legmélyebben lévő osztály aktuális példányát hivatkozza, ezért a tagosztály neve nem fedheti el a tartalmazó alosztály nevét. Ha a példányosítás során a tartalmazó osztálynak van aktuális példánya, akkor az lesz az aktuális példány. Ha nincs, akkor a this változó azon osztály aktuális példányát fogja hivatkozni, amelynek a beágyazásból kifelé haladva van aktuális példánya.

Árnyékolás (shadowing):
Egy belső osztály képes elfedni az őt tartalmazó osztály változóit.
Pl:

```
public class Shadowing {
  public int x = 0;

  class FirstLevel{
    public int x = 3;

    void methodInFirstLevel(){
      System.out.println("x = " + x);
      System.out.println("this.x = " + this.x);
      System.out.println("Shadowing.this.x = " + Shadowing.this.x);
    }
  }

 public static void main(String[] args) {
    Shadowing s = new Shadowing();
    Shadowing.FirstLevel f = s.new FirstLevel();
    f.methodInFirstLevel();
  }
}
```

Output:
x = 3

this.x = 3
Shadowing.this.x = 0

Lokális és névtelen osztályok:

A **lokális osztály**ok láthatóságát a blokk határozza meg, törzsében minden tagja minősítés nélkül hivatkozható. A lokális osztály példányai túlélhetik a blokkot és ekkor megőrzik a lokális változók értékeit.

- egy lokális osztály hozzáférhet az őt tartalmazó osztály tagjaihoz
- továbbá más lokális változókhoz (final-ökhöz is)
- mikor hozzáfér egy lokális változóhoz vagy a tartalmazó blokk valamely paraméteréhez, akkor a lokális osztály elkapja azt a bizonyos paramétert
- Java 8-tól kezdve olyan változókhoz és eszközökhöz is hozzáférhet, amelyek elől hiányzik a final kulcsszó, de gyakorlatilag final
- gyakorlatban final = inicializálása után sosem változtatja meg az értékét
- lokális osztály nem tartalmazhat statikus tagokat
- ha van static, akkor csak a tartalmazó osztály statikus metódusaihoz fér hozzá

LOKÁLIS INTERFÉSZ NEM LÉTEZIK!!!

Névtelen osztályt úgy hozhatunk létre, hogy nem adunk neki nevet. Így kizárólag az adott helyen tudjuk felhasználni osztályt, amelynek pontosan egy példánya létezik. Névtelen osztály példányosításkor is megkonstruálható. Ezek az osztályok tulajdonképpen statikusak, nincs specifikációjuk és nem tartalmaznak konstruktort (inicializáló blokkot viszont igen). A példányosítás azonban paraméterezhető a szülőosztály konstruktorával.

- lehetővé teszik, hogy egy időben egyszerre deklaráljuk és példányosítsunk egy osztályt
- név operátor
- interfész vagy osztálynév
- kerek zárójelek között formális paraméterlista (átadódnak a konstruktornak)
- törzs, az osztályban deklarált és definiált eszközök
- képesek elkapni változókat, ezen kívül a névtelen osztály hozzáfér a tartalmazó osztály tagjaihoz
- nem férnek hozzá a NEM final és gyakorlatilag NEM final tagokhoz
- tudnak árnyékolni
- nem tartalmazhatnak statikus tagokat, kivéve, ha azok konstans változók
- nem lehet bennük taginterfész és statikus inicializáló blokk
- lehet bennük mezőket definiálni,lehetnek bennük metódusok
- lehet bennük példány inicializáló blokk
- lehet bennük lokális osztály

Kivételkezelés:

1) **Java:**

A kivétel egy olyan esemény, amely a program futása során következik be, és arra utal, hogy a program nem tudta normálisan végrehajtani azokat az utasításokat, amelyeket benne kódoltunk.

Ilyenkor létrejön egy kivétel objektum. Ez az objektum átadódik a futtató rendszernek, hogy kezdjen vele, amit akar, amit tud. Kivétel létrejöttekor a program futása megszakad, és onnantól kezdve a futtató rendszer próbál valamit csinálni. Alapját a metódusok hívási lánca fogja jelenteni. Előfordul, hogy a hívási lánc valamelyik metódusában (az aktív metódus) kiváltódik

egy kivétel, ott megszakad a program futása, és átadódik a futtató rendszernek. A virtuális gép érvényes kivételkezelőt fog keresni. A kivételkezelőket a hívási lánc metódusaiban fogja keresni, a kiváltódó metódustól visszafele a main metódusig.

Ez a visszalépkedés addig tart ameddig meg nem találtuk az alkalmas és megfelelő kivételkezelőt, vagy el nem jutottunk a main metódusig. Ha megvan az alkalmas kivételkezelő, akkor elkaptuk a kivételt, különben a program futási hibával kiakad. Ha megtalálta, lefut a kivételkezelő kódja, majd a kivételkezelő kódját követő utasítással folytatódik tovább a program.

A JVM-ben vannak **ellenőrzött** és **nem ellenőrzött kivétel**ek.

- Az ellenőrzött kivételeket mindig specifikálni kell és mindig el kell őket kapni. Ha nem kapjuk el őket, akkor a fordító hibát jelez.
- A nem ellenőrzött kivételeket, ha nem kapjuk el, akkor a program leáll.

A fordító elengedi az ellenőrzést olyan eseményeknél, amelyek bárhol előfordulhatnak és ellenőrzésük kényelmetlen lenne (kódtömeg), ezért meg kell adni azokat a kivételeket, melyeket a fordító nem kezel, de futása közben bekövetkezhetnek. Ezeket hívjuk **eldobott kivételek**nek.

Eldobandó kivételek megadása: **throws** *kivételnév.lista**;***

A Java nyelvben csak a **Throwable** osztály objektumai dobhatóak el. Ezen osztálynak alosztálya az **Error** (például OutOfMemoryError) és az **Exception** (például RunTimeException) osztály. Az Error osztály minden objektumára igaz, hogy nem ellenőrzött, az Exception osztálynak azonban csak a RunTimeException objektuma nem ellenőrzött.

Kivételt a **throw** paranccsal dobhatunk el, amelyet jó esetben egy kivételkezelő el is kap. Ahhoz, hogy kivétellel információt adhassunk át, kivételkezelő osztályt kell létrehoznunk paraméterkezelő konstruktorral.

A kivétel elkapása:

try { utasítások } [**catch** (típus változónév) { utasítások] … [**finally** { utasítások }]

A try blokk tartalmazza az **ellenőrzött utasítások**at, emellett láthatóságot is definiál. A catch ágak sorrendje nagyon fontos, mert több ág is **elkap**hat egy **kivételt**, például az Exception kivételkezelő minden kivételt elkap.

A try blokkban elhelyezett utasításokban keletkezett kivételek esetén a catch parancsszó utáni blokk kapja meg a vezérlést.

- Ha a catch ágakban talál megfelelő típusú ágat, akkor lefutnak az utasításai, majd végrehajtódnak a finally ágban leírt utasítások és a program folytatódik a finally blokk utáni résszel.
- Ha egyetlen catch ág sem egyezik a szükségessel, lefut a finally blokk és tovább folytatódik a program.

Ezek alapján a catch ág akár teljesen is hiányozhat, ugyanakkor a finally ág akkor is lefut, ha nem volt kivétel.

try-with-resources (erőforrások):

formálisan:

try(erőforrásosztálynév azonosító [; erőforrásosztálynév azonosító]…){
} catch (kivételosztálynév azonosító) {
} [catch (kivételosztálynév azonosító) {

}] ....
finally { }

Olyan objektumokat lehet vele kezelni, amely osztályok megvalósítják a java.io.AutoCloseable interface-t. Azokhoz az objektumokhoz jó, amelyekhez erőforrás van rendelve. A try-ban megnyitja a fájlt, és nem kell a finally ág a fájl bezárásához, hanem az AutoCloseable pont erre szolgál.

Pl.:
try(BufferedReader br = new BufferedReader(new FileReader(path))) { return br.readLine(); }
....

2) C#:

A C# kivételkezelése hasonlít a Java nyelv kivételkezeléséhez, azonban…
C# -ban ha nem adunk meg egy **catch** ág után semmit, akkor létrejön egy **általános catch ág**, amely paraméter nélküli. Ilyen általános catch ágból csak egy szerepelhet, és csak az utolsó ág lehet. Különlegessége, hogy elkap minden kivételt, azt is, amelyet más nyelv váltott ki. A catch ág típusai nem lehetnek kompatibilisek, különben fordítási hibát kapunk. Ugyanakkor C#-ban a finally ágból nem lehet kilépni, csak vezérlés átadó utasítással.

throw [*kifejezés*] **;** (egy kivétel példányt kell adjon a kifejezésnek).
Továbbá:

- Kifejezés nélküli throw utasítás csak catch ágban szerepelhet, ahol a kivétel továbbadására szolgál (finally ág ekkor is lefut).
- A finally ágban bekövetkező kivétel továbbadódik.
- Metódus szinten nem kezelt kivétel továbbadódik a hívó metódusnak.
- Ha egy kivételt sehol sem kezelünk, akkor az aktuális szál félbeszakad és a folytatás implementációfüggő.

A C# nyelvben minden kivételosztály tartalmaz egy **Message** tulajdonságot, amely string típusú és tartalmazza a kivételhez tartozó üzenetet.

Java, Csomagok:
szerepe: nevek hatáskörének elhatárolása, típusok csoportosítása

Csomagban egymáshoz szorosan kötődő típusok vannak, ezek lehetnek osztályok, interfészek, enumerációk, annotációk. A csomagok osztályokat és interfészeket tartalmaznak, mert az enumeráció es az annotáció speciális osztályok (ezt nem értem, pedig így van leírva).

A csomagokat a forrásfájljaik elején a package kulcsszóval lehet jelölni. Max. 1 db package kulcsszó lehet a forrásfájl elején. Ha nincs, az adott forrásfájlban lévő eszközök a default csomagba kerülnek (névtelen csomag).

Erősen ajánlott a saját típusokat csomagokba elhelyezni. A csomagok egymáshoz viszonyítva hierarchiába rendezhetők, a hierarchiát a csomagok nevéből lehet kikövetkeztetni, hiszen a csomag neve egy olyan lista, amelyek neveit pont karakter határol el egymástól.

pl.: package hu.unideb.inf.prog2.shells, ahol a hu a legfőbb, shells a legalsó csomag

A csomagok egymásba ágyazhatóak.

Csomagokkal kapcsolatos lehetőségek

- a csomag egy hatáskör elhatároló eszköz, szóval hatáskört jelöl ki. A csomagokban definiált eszközök hatásköre a csomag.

- külső szinten a csomag lehet látható vagy csomag

- ha valamit publikussá teszünk a csomagban, akkor az nem csak az adott csomagban lesz látható, hanem másik csomagban is.

- más helyről egy adott eszköz hogyan hivatkozható? :

1. teljes minősített hivatkozással:
   pl.: hu.unideb.inf.prog2.shells.típusnév

2. A tartalmazó típus importálásával, majd típusnévvel
   program elején: import hu.unideb.inf.prog2.shells.típusnév;
   később amikor használni szeretném: típusnév

3. A tartalmazó csomag importálásával, majd egyszerűen típusnévvel
   import hu.unideb.inf.prog2.shells.*;
   …
   típusnév

4. Vannak olyan eszközök, amelyek bizonyos osztályban statikusak, pl a PI a math-ban
   A statikus tagokat statikus importtal tesszük elérhetővé:
   import static java.lang.Math.*; -> példa
   import static java.lang.tipusnév.*; -> általános

Névütközések megakadályozása

- Névütközés lehet, ha egy project több azonos nevű csomagot tartalmaz. Ez a névütközés automatikusan feloldható az explicit hivatkozással.

- különböző csomagok azonos nevű típusokat tartalmaznak:

- ha ezeket teljesen minősítve használjuk akkor nincs gond

- ha rövid hivatkozással hivatkozok a típusra, akkor fordítási hiba van.

```
import hu.unideb.inf.prog2.shells.Hallgato;
import hu.unideb.inf.prog1.shells.Hallgato;
….
hu.unideb.inf.prog2.shells.Hallgato;
hu.unideb.inf.prog1.shells.Hallgato;
```

A java.lang csomag implicit módon mindig importálva van -> nem kell őket minősíteni.

A csomaghierarchia és a könyvtárrendszer viszonya

Minden csomaghoz külön alkönyvtár van rendelve operációs rendszer szinten. Logikai szinten a csomagok egymásba ágyazva helyezkednek el, ezek a háttértárakon egymásba elhelyezett alkönyvtárak.

```
package mypackage;
class Main{
    public static void main (String[] args){
        sysout(„Main from mypackage.”);
    }
}
```

Csomagokban lévő osztályok main metódusát akarjuk indítani, akkor azt teljes hivatkozással kell.

C# névterek (namespace):

A C# programokban névtereket lehet definiálni (spacename). Egymásba skatulyázhatóak, egymásba lehet őket definiálni. Nevek hatáskörének elhatárolására szolgál.

```
namespace névtérnév1{
        .
        .
        namespace névtrénév2{
        }
}
```

A C# program is megengedi, hogy névtelen névtérbe helyezzük el az eszközeinket, ezt a tevékenységet hívjuk úgy, hogy **az illető szennyezi a névtelen névteret**.

A C# névterek nem különülnek el külön alkönyvtárakba, hanem ugyanabban az állományban lesznek.

Interfészek:

1) **Java:**

[módosító] interface név [extends interfacenév_lista]
törzs

public static final:

- absztrakt metódusokat
- alapértelmezett metódusok
- statikus metódusokat
- konstans deklarációkat

default típus_név([f.p.l.]) (throws kivételosztálynév_lista) törzs
static típus_név([f.p.l.]) (throws kivételosztálynév_lista) törzs

2) **C#:**

[módosító] interfacenév [: interfacenév_lista]

metódus jellegű kód:

- absztrakt metódus
- tulajdonság
- indexelő
- esemény

A C# szigorú, de lehetővé teszi ütköző nevek feloldását. Két lehetőség:
- expplicit interfészimplementáció
- implicit interfészimplementáció