Magas szintű programozási nyelvek 2
Előadás jegyzet

1. Rendszerfejlesztés

0. lépés: Elemzés (analízis)
1. lépés: Tervezés
   a, technológia független rész
   b, technológia függő rész
2. lépés: Megvalósítás (implementáció)
3. lépés: Működtetés, karbantartás, továbbfejlesztés

2. Az evolúció főbb állomásai

C# << Java << (Ada, Smalltalk 80, C++, Objective C) << ALGOL 60
A Simula 67-es nyelvekben jelentek meg az első objektumorientált elemek, de a Smalltalk 80-as volt az első igazi objektumorientált nyelv.

3. Az objektumorientált paradigma (OOP)

Létrejöttének oka, hogy tovább növeljék a programok absztrakciós szintjét.
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**

4. Absztrakt adattípus

Használatával a programozó lehetőséget kap arra, hogy magasabb szinten gondolkodjon a programról, ne az alacsony szintű megvalósítási kérdések legyenek a meghatározók.

Osztály: Az absztrakt adattípust valósítja meg, az osztály egy absztrakt nyelvi eszköz. Jellemzői:

- **Attribútumok**: statikus jellemzőket definiálnak.
- **Metódusok**: dinamikus jellemzőket definiálnak (eljárás-orientált nyelvekben ~ alprogramok).

Objektum: Konkrét nyelvi eszköz, amely mindig egy osztály példányaként jön létre. Az objektum létrejöttét **példányosítás**nak nevezzük. Egy osztályhoz több példány is létrehozható, amelynek az osztályhoz hasonló adatstruktúrával és viselkedésmóddal rendelkeznek. A memóriában helyezkednek el, ezért van memóriabeli címük.

- **Állapot**: a memóriacímen elhelyezkedő adat-együttes.
- **Kezdőállapot**: a példányosítás következtében az objektum ebbe az állapotba kerül.
- **Öntudat, objektumazonosító**: minden objektum csakis önmagával azonos, mert egyedi és megkülönböztethetetlen azonosítójuk van.
- **Üzenetküldés**: az objektumok között interfészek segítségével párhuzamos üzenetküldésre van lehetőség.

  Az **interfész** azt írja le, hogy egy osztály példányainak mely attribútum és metódus jellemzői látszanak más objektumok számára. Az interfészt maga az osztály definiálja.
  Üzenetküldés során egy adott objektum egy számára látható metódust **meghív és megcímzi** a másik objektumot. A fogadó objektum **visszatérési érték**kel **vagy output értékk**el válaszol. Az üzenetküldés megváltoztathatja az objektum állapotát.

5. Osztály- és példányszintű attribútumok és metódusok

Az osztály szintű attribútumok az osztályban képződnek és nem többszöröződhetnek, a példány szintű attribútumok azonban minden példányosításnál elhelyeződnek a memóriában.

- **Kiterjedés**: megadja, hogy az adott pillanatban hány példánya van az osztálynak (osztály szintű attribútum funkcionálhat így).

A példány szintű metódusok a példányok viselkedését határozzák meg.

- **Aktuális példány (this, self):** azon példány, amelyet a metódus éppen kezel. Osztályszintű metódusoknál nincs aktuális példány.

Beállító, lekérdező metódusok:

A példányszintű beállító metódusok hatására a példány állapotot változtathat, amelyet a paraméterekkel határozhatunk meg.
A lekérdező metódusok függvény jellegűek és az aktuális példány aktuális állapotával térnek vissza.
Az osztályszintű beállító metódusokkal az osztály attribútumait állíthatjuk be.

Konstruktor:

Az objektumorientált nyelvek használják a kezdőérték beállításához. Az élő példány tudja, hogy melyik osztály példányaként jött létre.

6. Öröklődés

Aszimmetrikus kapcsolat, az újrafelhasználhatóság eszköze az objektumorientált nyelvekben.

- **Szülőosztály**: más néven szuperosztály vagy alaposztály.
- **Gyermek osztály**: más néven alosztály vagy származtatott osztály.

Öröklődés során az alosztály átveszi (örökli) a szuperosztályának minden attribútumát és metódusát, amelyeket azonnal fel is tud használni. Az alosztály ezen felül **új** attribútumokat és metódusokat **definiál**hat, az átvett neveket **átnevezhet**i, **újradeklarálhat**ja, vagy megváltoztathatja a **láthatóság**aikat és **újraimplementálhat**ja 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. Egy új osztály tehát a fordítási időben jön létre és emiatt az öröklődés is 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.

Osztály hierarchia:

Egyszeres öröklődés: **fa-alakzat**
Többszörös öröklődés: **aciklikus gráf-alakzat**

- **Gyökérosztály**: nincs szuperosztálya
- **Levélosztály**: nincs alosztálya.

Névütközés: Akkor fordul elő, ha a szuperosztályok között van 2 azonos nevű, amely miatt az azonosítás nem lehetséges egyértelműen.

Az öröklődést 2 különböző megközelítésből szemlélhetjük:

Típus oldalról (ez a szemléletmód az absztrakciót helyezi középpontba)
- Osztály: típus.
- Alosztály: altípus.

Helyettesíthetőség: egy leszármazott osztály példánya minden helyen elhelyezkedhet a program szövegében, ahol az elődosztály példánya megjelenhet, azaz az elődosztály helyettesíthető a leszármazottal.

Strukturális oldalról
Az öröklődés, mint eszköz a program struktúrájának erősödését segíti.

7. Bezárás

Az alosztályok eszközeinek **láthatóság**át szabályozza. Szintjei:

- Publikus: az összes kliensosztály látja.
- Védett: az adott osztály és a leszármazott osztályok látják.
- Privát: csak az adott osztályban használhatóak (az alosztály azonban örökli a privát eszközöket).
- Programegység szerkezetén alapuló: nem minden nyelvben szerepel.

Az attribútumok privátok, a metódusok publikusak.

8. Metódusok újraimplementálása (polimorfizmus)

Polimorf metódusok:

Előfordulhat, hogy két metódusnak azonos a neve. A fordítási hibát csak úgy kerülhetjük el, ha eltérő ezen metódusok specifikációja.

Kötés (nyelvi mechanizmus):

Statikus kötés: már a fordításkor eldől a kérdés, a helyettesíthetőség nem játszik szerepet.
Dinamikus kötés: a kérdés csak futási időben dől el. A megoldás a helyettesíthetőségen alapul.

Metódusnevek túlterhelése:

Azonos nevű, eltérő implementációjú metódusokat tudunk létrehozni. A metódusok specifikációjának különbözniük kell.

9. Öröklődés, delegáció, továbbküldés

Osztály szintű öröklődés: már meglévő funkcionális osztályok segítségével használhatjuk újra és definiálhatunk új funkcionalitásokat.

A delegáció és a továbbküldés már **objektum szintű öröklődés**. Ekkor, ha:

- A, B egy-egy objektum
- M egy üzenet

Ha A nem érti meg az M üzenetet, akkor öröklődéskor ez hibát jelent.
Az A objektum azonban továbbküldheti ezt az üzenetet B objektumnak.

A delegálás és a továbbküldés közötti különbség a **self kezelés**ben van:

Delegálás esetén B-ben is A-t adja, tehát az aktuális példány A marad.
Továbbküldés esetén az A és B objektum megtartja saját identitását, tehát A-ban A-t, B-ben B-t kapjuk meg, ha hivatkozzuk.
Az M üzenet ekkor a B objektumon operál.

10. Absztrakt osztály

Segítségével viselkedésmintákat adhatunk meg, amelyeket majd valamelyik leszármazott osztály fog konkretizálni. Az absztrakt osztály tartalmaz absztrakt metódusokat, amelyeknek csak specifikációjuk van, implementációjuk nincs. Az absztrakt osztályból származtatható absztrakt és konkrét osztály (minden metódusához kötelező az implementáció) is. Egy osztály mindaddig absztrakt, amíg **legalább egy metódusa absztrakt**. Ezek az osztályok az öröklődési hierarchiában **levél osztály**ok lesznek.

11. Paraméterezett osztály (kollekció)

Generikusoknak felelnek meg.

12. Konténer (beágyazott osztály)

Olyan osztály, amelynek objektumai maguk is objektumokat tartalmaznak. Tulajdonképpen objektumokból álló adatszerkezetek realizálására szolgálnak. Ilyen osztály a **kollekció**, amelynek adattípusa például a verem, fa, lista, sor, halmaz vagy a multihalmaz. A konténerosztály elemeinek elérését szolgálja az **iterátor**, amely maga is az osztálynak egy objektuma. Ezzel lehet bejárni a konténer által tartalmazott objektumokat, függetlenül azok reprezentációjától.

13. Objektumok élettartalma

Az objektumokat is a memóriában tároljuk, ezért ők **tranziens**ek. Ez azt jelenti, hogy nem élik túl az őt létrehozó programot. Arra azonban lehetőség van, hogy az objektum állapotát I/O eszközökkel állományba mentsük és aztán egy másik objektum állapotát vissza ki tudjuk olvasni az állományból. Adatbázis-kezelő rendszerekben az objektumok **perzisztens**ek, azaz túlélik az őket létrehozó alkalmazást.

A már szükségtelen objektumok tárterületeit fel kell szabadítani, erre 2 megvalósítás született:

- Az objektumot **explicit módon** a programozónak kell megsemmisítenie.
- **Szemétgyűjtögetés** módszerével a rendszer automatikusan törli a szükségtelen objektumok tárterületét.

14. Reflexió

Egy rendszer reflexív, ha futás közben meg tudja vizsgálni saját végrehajtási állapotait. Erre szolgálnak a **meta-adatok** (adatok az adatokról).

A reflexió lehet:

- **Megfigyelő**: a rendszer csak olvashatja a belső állapotait.
- **Beavatkozó**: a rendszer módosíthatja is belső állapotait.

A reflexió végbemehet:

- **Magas szinten**: például, ha elérjük egy veremben az aktiváló rekord elemeit.
- **Alacsony szinten**: például, ha a memóriát, mint egész tömböt olvassuk.

Beavatkozó magas szintű reflexió esetén elképzelhető, hogy meg lehet változtatni az öröklődési hierarchiát.
Annak leírását, ahogyan egy objektumorientált rendszer alap szinten működik **meta-object protocol**-nak (MOP) hívjuk.
Totálisan dinamikus objektumorientált rendszerben a reflexió segítségével hozzáférhetünk a rendszer MOP-jához és módosíthatjuk azt.

15. UML alapfogalmak

- **Minta**: egy olyan általános probléma megoldása, amely működött a múltban és hasonló módon újra felhasználható a jövőben.
- **Idióma**: a nyelvi mintát hívjuk így, például hogyan kell megnyitni egy állományt.
- **Csomag**: típusokat és csomagokat tartalmaz, és egy névteret jelöl ki.
- **Osztálydiagram**: objektumokra szaggatja szét a valós világot.

16. Java

A Java programozási nyelvet a SUN 1995 óta fejleszti, a C++ javított kiadásaként érkezett. A Java-ban megírt programot a fordító egy bájtkódra fordítja le és ezt a programot a **Java Virtual Machine** futtatja, amely egy interpreter. A JVM **platformfüggetlen**. A Java karakterkészletét az **Unicode 16** szabvány alkotja, a kis- és nagybetűk a nyelvben meg vannak különböztetve.

Azonosító: _ és $betűt is tartalmazhat.

Kulcsszavak: standard azonosítók nincsenek.

Megjegyzések: sorvégéig tartó (//), tetszőleges elhelyezésű (/* … */), dokumentációs (javadoc nevű program dolgozza fel) (/** … */).

Címke: bármely utasítás címkézhető.

Típusok:

- Primitív és referencia (tömb, osztály, interfész).
- Beépített típusok (boolean, char, byte, short, int, long, float, double, csomagoló osztályok).

17. Java eszközök

Tömb:

Csak **egydimenziós** tömböket ismer, az **indexelés 0-ról** indul, az elemszámot **nem kötelező** előre megadni. A tömb jelölése: [].
Mivel a tömb referenciatípus, ezért objektumként kezeljük.
A **new** operátorral tudunk új tömböt példányosítani.

Iterál:

Értékeit a beépített típusok értékkészleteiből veheti fel.
Karakter konstans jelölése: ’a’, sztring konstans jelölése: ”ab”.

Nevesített konstans:

Beépített nevesített konstansok: NULL, true, false (ezek foglalt szavak is).
Saját nevesített konstanst nem lehet létrehozni!

Változódeklaráció:

típus név [= érték] **;**
Automatikus kezdőérték adás nincs, kezdőérték azonban adható.

Kétirányú elágaztatás:

if (*feltétel*) utasítás [**else** utasítás]
A feltétel nem lehet numerikus, csak logikai típusú.

Többirányú elágaztatás:

switch (*egész.kifejezés*) { **case** *egész.literál* **:** utasítások [**case** *egész.literál* **:** utasítások] … [**default :** utasítások] }

Előfeltételes ciklus:

while (*feltétel*) utasítás;

Végfeltételes ciklus:

Do utasítás **while** (*feltétel*);

For ciklus:

for (*[P1]; [P2]; [P3]*) utasítás;
for (*típus változó : kollekció*) utasítás; (~ foreach)

Vezérlésátadó utasítások:

break [*címke*] (címke nélkül a legbelső blokkra hivatkozik).
continue [*címke*] (a megadott címkére vonatkozik).
return [*kifejezés*] (módszerek befejeztetésére szolgál).

A **goto** utasítást **nem ismeri** a Java nyelv.

18. Osztályok és példányok a Java nyelvben

A Java nyelvben az osztályok önálló egységet alkotnak, **minden osztály önállóan fordítható**. A nyelv az osztályokat **csomagokba szervez**i. Egy osztály attribútumait (adattagjait) változódeklarációk, módszereit (tagfüggvényeit) függvénydefiníciók adják. Az osztály minden példánya saját készlettel rendelkezik az adattagokból. Egy osztály valamelyik módszerének meghívásánál meg kell adjuk, hogy melyik példányra hívtuk meg. Azt a példányt, amelyre meghívtuk a módszert **aktuális példány**nak nevezzük. A példányok adattagjait **példányváltozóknak** is nevezzük.

Az osztályváltozók és osztálymódszerek összefoglaló neve a **statikus tagok**, amelyek magához az osztályhoz kapcsolódnak. Osztályonként egy-egy osztályváltozó van, az osztálymódszerek pedig ezeken dolgoznak. Ezek az osztálymódszerek akkor is működnek, ha az osztálynak egyetlen példánya sincsen.

Az osztály egy absztrakt adattípus implementációja. Minden osztály egy **fej**ből és egy **törzs**ből áll.

A fej szerkezete:

[*módosító*] **class** név [**extends** *szuperosztály.név*] [**implements** *interfész.lista*] (az alosztályok között csak egyszeres öröklődés van)

Az osztály lehetséges módosítói: **abstract**, **final** (nem örökölhető), **public** (más csomagból is látható), **alapértelmezetten private**.

A törzs szerkezete:

{} jelek közé írt **tagdefiníció**k tetszőleges sorrendben.

Példányváltozó:

[*módosító*] típus név [= kezdőérték] [, név [= kezdőérték]] … ; (alapértelmezett kezdőértékek: 0, null, false)

Ha egy példányváltozó módosítója **final**, az azt jelenti, hogy az adattag értéke nem változtatható meg, azaz **konstans**ként viselkedik. Ha nincs kezdőértéke, akkor konstruktor segítségével kell neki értéket adni vagy inicializáló blokkban kell beállítani.

Példányváltozó láthatósága:

- ha nincs alapszó, alapértelmezetten **private** (az adott adattagnak csomagszintű láthatósága van).
- **protected** (védett láthatóság: az adott osztály és annak leszármazottai láthatják).
- **public** (nyilvános láthatóság: minden csomag láthatja az adattagot).

Példánymódszerek:

[*módosító*] fej (specifikáció) törzs (szignatúra)

- **abstract** módosító: az ilyen módszer csak absztrakt osztályban szerepelhet, törzsét valamely osztály leszármazottja adja meg.
- egy osztály mindaddig absztrakt, amíg legalább egy módszere absztrakt.

19. Paraméterkiértékelés

A Java nyelv az alábbi módon történik a paraméteregyeztetés: **sorrendi kötés, számbeli egyeztetés**, **típusbeli egyeztetés**.
Van azonban **változó paraméterszámú** paraméterlista is, jelölése: **…**, de a típust is meg kell adni.
Java-ban a paraméterátadás **érték szerinti**.

20. Statikus tagok és hivatkozás

A Java nyelvben a statikus tagokat a **static** módosítóval különböztetjük meg a többi, dinamikus tagoktól.
Példa a hivatkozásra: System **.** out **.** println(”szöveg”);

21. Példányosítás és élettartam

A példányosítás a **new** operátorral történik. Példányosításkor létrejön a példány és a változói, ezután meghatározódnak a kezdőértékek. Amikor egy példányra hivatkozunk nem a tárcímet, hanem magát az objektumot hivatkozzuk. Az aktuális példányra a **this** pszeudo-változóval hivatkozhatunk.

A Java Virtual Machine tartalmaz egy referenciaszámlálót, erre építi a **garbage collection** algoritmusát is, amelynek elve, hogy ha nincs egy objektumra hivatkozás, akkor az törölhető. Tehát egy objektum élettartama azon időintervallum, amely **a példányosítás és az objektumra való hivatkozás között telik el**.

22. A Java főprogramja

A főprogramot a Java nyelvben a publikus és statikus visszatérési érték nélküli **Main()** függvény jelöli. A főprogram akkor fejeződik be, ha annak vége van, vagy valahol a System osztály **exit** módszerét hívjuk meg.

23. Polimorfizmus, túlterhelés és elfedés

A Java nyelven több metódust ugyanazzal a névvel nevezhetünk meg, ha a formális paraméterek száma vagy típusa eltér. Ekkor a megfelelő kódot a fordító az aktuális paraméterek száma és típusa szerint választja ki.

A polimorfizmus elve szerint egy örökölt módszer implementációja tetszés szerint megváltoztatható. A Java nyelv a **dinamikus kötés** elvét vallja, azaz mindig az aktuális példány osztályában definiált vagy a hozzá legközelebbi örökölt kód kerül meghívásra. Az újradefiniált módszernek **felülről kompatibilis**nek kell lennie az örökölttel, így a bezárást nem lehet szűkíteni, csak **bővít**eni, hiszen azonos specifikációjúaknak kell lenniük az újradefiniált módszereknek. Az alosztályban a **super** minősítővel hivatkozhatunk a szuperosztály polimorf módszerére.

Az öröklődés során a leszármazott új adattagokat definiálhat, de **nem szűntethet meg** örökölt adattagot. A példánymódszereket azonban **újraimplementálhat**ja, illetve á**tdefiniálhat**ja az adattagokat és az osztálymódszereket, ezzel képes elrejteni az eredetieket a hozzáférés elől az adott és a leszármazott osztályaiban is. Az elfedett tagok osztálynév minősítéssel, a **super** minősítővel vagy típuskényszerítéssel érhetőek el. Példánymódszer azonban osztálymódszerrel sem fedhető fel, mert kötése statikus.

24. Konstruktor, inicializáló blokk és destruktor

Példányosítás után a példány **alapállapot**át 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 konstruktor törzsének első utasítása lehet egy adott osztálybeli, más osztálybeli vagy egy szülőosztálybeli konstruktor hívása is. Ehhez a **this**, illetve a **super** kulcsszót kell használnunk. 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 egy alosztály valamely konstruktorában nincs **this** vagy **super** hívás, akkor automatikusan beépül egy **super();** hívás, amely a szuperosztály alapértelmezett konstruktorát hívja meg. Tehát alosztály példányosításkor mindenképpen lefut a szülőosztály valamelyik konstruktora.

Inicializáló blokkból egy osztályon belül tetszőleges számú lehet. Létezik **példány** és **osztály** (static módosító jelzi) **inicializáló blokk** is. A példány-inicializáló blokk minden példányosításnál lefut. Általános célja, hogy a konstruktorok közös kódját helyezik el benne.

A **destruktor**t a **Finalize**() metódus valósítja meg, amely egy protected láthatóságú, visszatérési érték nélküli módszer. Ezt a módszert minden osztály örökli és átdefiniálhatja. Az objektum megszűntetésekor végrehajtandó műveleteket írhatjuk le benne. Meghívása automatikusan történik meg, még a példány tárhelyének felszabadulása előtt.

25. Interfészek a Java-ban

Az interfészt mindig egy osztály implementálja, azonban az interfész **nem objektum**! Az interfész-hierarchia alján mindig osztályok állnak. A többszörös öröklődés is megvalósítható az interfészekkel: az interfész, mint referenciatípus **mindenütt szerepelhet, ahol az osztály szerepelhet**, azaz előfordulhat formális paraméter típusaként, metódus visszatérési típusaként, változódeklarációnál vagy tag típusnál.

[*módosító*] **interface** név [**extends** *szuperinterfészek*] törzs

A Java a névütközéseket nem tudja kezelni, mindez többszörös öröklődés esetén keletkezhet.

Interfész implementálása:

módosító **class** név **implements** *interfésznevek* törzs

26. Csomagok és fordítási egységek a Java-ban

A csomagok tartalmazzák a Java kódjait és alkalmazásait, önálló fordítási egységeknek tekintjük őket. Tehát a fordítási egységeket csomagokba gyűjtjük. A csomag önmagában egy névteret is meghatároz. A csomagok között egy hierarchia épül fel, amely könyvtárszerkezetként jelenik meg, más szóval a Java hierarchia nem más, mint csomagok által felépített fa. A csomagokban deklarált eszközökre minősítéssel lehet hivatkozni.

Fordítási egység létrehozása:

[*csomagleírás*] [*importleírás*] típusdeklaráció**;**

package csomagnév; Ekkor az adott csomaghoz fog tartozni a lefuttatott kód.
Ha nincs csomagleírás, akkor névtelen csomagba kerül.

import [static] *minősített.név***;** Segítségével egy másik csomagban deklarált nyilvános típusokat tudunk elérni úgy, hogy csak egyszerű és nem pedig minősítet névvel hivatkozzuk.

Import Alakzat.Zárt.Ellipszis.Kör;
Import Alakzat.Zárt.Ellipszis.*; (így nem csak Kör-t érjük el)

Az importálás nem vonatkozik a csomag alcsomagjaira, azokat külön kell importálni.

27. Típus-egyenértékűség és típus-kényszerítés a Java nyelvben

Primitív numerikus típusok esetén:

- C-szerű, automatikus konverzió van (byte > char/short > int > long > float > double).
- A Java automatikusan string típusúra konvertál minden más értéket, ha konkatenáció esetén az egyik operandus sztring.

Referencia típusok esetén:

- Adott típusú objektum típusával egyenértékű (azaz konvertálható az adott típusra) azon osztály, amelynek a példánya maga az objektum.
- Null érték eltérő referenciatípusok esetén is egyenértékű.
- Tömb esetén: egyedeinek típusa egyenértékű.

Minden objektum ismeri a példányosító típusát, erre szolgál az **instanceof** parancs.

t **instanceof** Négyzet Akkor igaz a kifejezés, ha t Téglalap típusú értéke egy Négyzet vagy annak leszármazottja.

Szűkítő referenciakonverziót csak akkor lehet végrehajtani, ha az objektum leszármazott típusú.

28. A Java kivételkezelése

A Java nyelv kivételkezelési mechanizmusa hasonló az ADA nyelvéhez. Ha bekövetkezik egy speciális esemény valamely módszer futása közben, akkor egy **kivétel objektum** jön létre, amely információkat tartalmaz a kivétel fajtájáról és aktuális állapotáról. A módszer eldobja a kivételt és a JVM kezelésébe kerül. A módszer ezután felfüggeszti működését annál az utasításnál, ahol a kivétel bekövetkezett, majd bekövetkezik a kivételkezelés.

A kivételkezelés során a JVM keres egy megfelelő típusú és látható kivételkezelőt. Egy kivételkezelő megfelelő a keresési feltételeknek, ha típusa megegyezik a kivétel típusával, vagy őse annak. A láthatóságot maga a kivételkezelő definiálja. **A kivételkezelő maga egy blokk** és ebbe a blokkba tetszőlegesen ágyazható újabb blokk is. A JVM ezekből a blokkokból lépked kifelé, amíg a megfelelő kivételkezelőt meg nem találja. Ha megtalálta, lefut a kivételkezelő kódja, majd a kivételkezelő kódját követő utasítással folytatódik tovább a program. Ekkor azt mondjuk, hogy a kivételkezelő elkapta a kivételt. A kivételkezelőben bekövetkezett kivételt ugyanígy kezeli a JVM.

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, amelyeket 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.

29. Kollekciók és bejárásuk

A Java nyelvnek saját keretrendszere van a kollekciók kezeléséhez, melynek neve **Collection Framework**. Ez az osztályoknak és az interfészeknek olyan összefüggő rendszere, amely egy általános célú eszközkészletet ad. Ősinterfésze a **Collection**. Alapműveletei:

- int size();	Megadja, hogy hány elem van a kollekcióban.
- boolean isEmpty();	Megadja, hogy üres-e a kollekció.
- boolean Contains(Object element);	Megadja, hogy a kollekció tartalmazza –e a paraméterként adott elemet.
- boolean add(Object element);	Megpróbálja hozzáadni a paraméterként kapott elemet a kollekcióhoz.
- boolean remove(Object element);	Megpróbálja törölni a paraméterként kapott elemet a kollekcióból.
- boolean ContainsAll(Collection c);	}
- boolean addAll(Collection c);	} A paraméterként megadott kollekció összes elemére vonatkozik a művelet.
- boolean removeAll(Collection c);	}
- Iterator iterator();	Segítségével a kollekció elemei végigjárhatóak.
- boolean retainAll(Collection c);	A megadott kollekció elemeivel egyező elemek maradnak csak meg.
- void clear();	Törli a kollekcióból az összes elemet.
- Object[] toArray();	Egy tömbbe másolja a kollekció elemeit.
- Object[] toArray(Object[] a);	Az adott paraméter típusának megfelelő tömbbe másolja a kollekció elemeit.

A kollekciók alapinterfészei:

Collection > (Set > SortedSet, List, Queve)
Map > SortedMap

Kollekciókat iterátorokkal járhatunk be:

- public boolean nasNext();	Igaz értékkel tér vissza, ha van következő elem.
- public Object next();	A következő elemre lép.
- public void remove();	Az utoljára elért elemet törli.

30. Rendezés

Comparable interfész:

[int] **CompareTo** (object obj)	Negatív értéket ad, ha az aktuális példány rendezettség szerint előrébb van (kisebb), mint a paraméterként kapott példánytól.

Comparator interfész:

int **Compare** (object a, object b)	Ha **a** hátrébb van, mint **b**, akkor pozitív értéket ad vissza, tehát nem aktuális paraméterrel vizsgál.

Objektumok azonosságára vonatkozó vizsgálatok:

== és !=	Memóriacímet vizsgálnak.
Object Equals()	Újra-implementálható metódus.

31. Assertion (követelmény, programhelyességi előírás)

Lehetővé teszi a programmal szembeni elvárásaink tetszelését.

assert *feltétel* [: *kifejezés*] **;**	Ha a feltétel hamis, akkor kiváltódik a kivétel. Ha igaz, akkor a program adott pontján az elvárásunk teljesült.

Az előírás célja, hogy gyorsan és hatékonyan találja meg a hibákat, valamint hibadokumentáláshoz is hasznos.
Hamis feltétel esetén kiváltódik az **AssertionError kivétel**, és ha van megadott kifejezés a rendszer a kifejezés értékét átadja a kivétel konstruktorának. Ennek eredményeképpen részletes hibaüzenetet hozhatunk létre. Egy előírás nem változathatja meg a program állapotát.

Előírásokat módszerekhez is köthetünk, például megadhatjuk, hogy minek kell teljesülnie, miután az adott módszer lefutott.

32. Beágyazott osztályok

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.

33. 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.

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.

34. Szálak

A Java nyelvben a **Thread** osztály **run()** módszere adja meg a szálként futtatandó kódot. Ezt a módszert implementálnunk kell, és szükségünk van a **Runnable** interfészre is. A szálak példányosíthatóak, ezért új szálat, mint objektumot a **new** operátorral hozhatunk létre. Kezdetben csak létrejön a szál, elindítani a **start** módszerrel lehet, ekkor **futásra kész** állapotba kerül.

A szálak állapotai a lehetnek: új, futásra kész, futó (aktív), várakozó, halott.

A futásra kész állapotúak közül az **ütemező** választja ki a futtatandó szálat. Futáskor a run() kezd el futni. Egy szál akkor lesz **halott**, ha a **stop()** módszert hívjuk meg, vagy ha a run() kódja véget ér. Ezután a szál újra már nem hívható meg.

Hoare-féle monitor:

A szinkronizációt valósítja. Az adatokat és a műveleteket úgy zárja egységbe, hogy a monitorhoz tartozó műveleten kívül csak egy művelet lehet aktív. Az adatok csak a monitor által meghatározott műveleteken keresztül érhetőek el.

Feltételes változók: (lehet, hogy ez rosszul van definiálva)

Minden feltételes változóhoz tartozik egy sor, amelyben 2 művelet van értelmezve (felfüggesztés, várakozás). Felfüggesztés esetén az aktuális monitor folyamat blokkolódik, felfüggesztődik, várakozni kezd. A várakozási sorban várakozik az aktuális monitor. A továbbindításnál, ha van a várakozási sorban felfüggesztett folyamat, akkor az továbbindul, egyébként nem történik semmi.

Szinkronizáció:

A Hoare-féle monitor elvén működik. Ahhoz, hogy egy szál szinkronizálható legyen, a **synchronized** módosítóval jelölni kell. Egy várakozó szál futásra kész állapotba kerülhet:

- ha letelt a **sleep()** ideje,
- ha a **notify()** vagy **notifyAll()** metódus megszűnteti a szál(ak) várakozását,
- ha befejeződött az I/O művelet.

A JVM minden szinkronizáció előtt felépít egy monitort. Ha egy szál meghív egy módszert, akkor egyetlen másik szál sem hívhatja meg ugyanezen objektum synchronized metódusát. Azaz a szinkronizáció kezeli a konkurenciát.

Létezik **osztály szintű monitor** is, például ha statikus módszerhez adjuk meg a synchronized módosítót. Ekkor az osztályhoz épül fel egyetlen monitor és a synchronized módszereket csak egyetlen objektummal érhetjük el.

A **wait()** metódus elengedi a monitort és egy feltétel bekövetkeztére, vagy a notify() –re vár, amikor is újra magához közi a monitort, ha tudja. Ha sikerült magához kötnie, akkor az fut tovább.

A **yield()** módszer felfüggeszti a szál működését, amely visszakerül a várakozási sorba, hogy más szál is működhessen (nem időszakos).

Szálak összefűzése:

Szálakat a Thread osztály **join()** metódusával tudunk összefűzni. Összefűzött szálak esetében a hívó szál addig vár, amíg a meghívott be nem fejezi a működését. A join() –hoz megadható időparaméter is, amely megadja, hogy legfeljebb meddig várjon.

A szálakhoz **prioritás** is rendelhető (10 szintig) a **setPriority()** metódussal, amely indítás és futás közben is alkalmazható. Alapesetben minden szál 5ös szintű. Az ütemező mindig a legmagasabb prioritású, futásra kész szálat indítja el. Azonos prioritás esetén az érkezési sorrend és a **rang** alapján dönt. A **kommunikáció** paraméterek és osztály-attribútumokkal történik.

35. Generikusok

Osztályok, interfészek és metódusok esetén <> jelek között megadhatóak a generikus típusok, avagy a **típusváltozó**k. Ezek a paraméterezett típusok használhatóak ott is, ahol más szokványos paraméterek, de tőlük eltérően típusként is szerepeltethetőek. A generikusok azonban **primitív típusokkal nem paraméterezhetőek**. A típusváltozók az őket deklaráló osztályban és interfészekben láthatóak, kivéve a statikus inicializálókban és metódusokban.

Egy generikus osztály, illetve interfész deklaráció a típusok egy halmazát, más szóval egy **sablon**t definiál.

A generikus osztályokhoz **interfész**ek adhatóak, mint **korlát**ok. A korlátként megadott típusok sorrendje nem számít, de osztálytípus korlátnak az első helyen kell állnia. Ha egy típusváltozónak több korlátja van, akkor az ütköző metódusokra, amelyek azonos szignatúrájúak, de eltérő a visszatérési értékük, a rájuk történő hivatkozás fordítási hibát okoz.

Típustörlés:

Futásidőben nem jelennek meg a generikus típusok. Egy paraméterezett típusból a típustörlés eredményeképpen kapott típus (törölt típus):

- ha nincs korlátja: object típusú lesz,
- ha van osztály típusú korlátja: az adott osztály típusú lesz,
- ha csak interfész típusú korlátja van: a típusváltozó értéke a lexikografikusan első interfész típus lesz.

Nyers típusok:

Ha egy generikus típust a típusváltozók értékeinek megadása nélkül használunk, akkor az úgynevezett **nyers típus**t kapjuk, ami majdnem teljesen megegyezik a törölt típussal. A nyers típus arra szolgál, hogy **biztosít**sa az adott osztály generikus és nem generikus típusa közötti **átjárhatóság**ot. Fontos lehet tudni, hogy mi lesz az adott paraméterezett típus típustörésének az eredménye.

Generikus metódusok:

A generikus metódusok is elláthatóak típusparaméterekkel, amelyeket a metódus visszatérési típusa előtt kell feltüntetni. Generikus metódus hívásakor a **típusparamétereket nem kell megadni**, mert a fordító **automatikus**an határozza meg a típusokat a paraméterek típusa alapján egy sajátos **típuslevezetési technika** szerint. A típuslevezetéssel nem meghatározható típusparaméterek értéke **Object** lesz.

Generikusok és kivételkezelés:

Mivel futásidőben a generikus típusok nem értelmezhetőek, ezért nem állhatnak **catch** blokkfejekben.

36. Enum típusok

Olyan típusok, amelyek **mezői konstansoknak egy halmazából áll**. A nevesített konstansok nem rendelkeznek **saját névtér**rel, de az enum típusok **igen**. Sőt, a nevesített konstansok kódját újra kell fordítani, ha új konstanst adunk meg, az enum típusokét nem.

Az enum egy **osztály**t definiál. **Törzs**e tartalmazhat más metódusokat is, ezért a fordító automatikusan hozzáad néhány metódust a törzshöz, amikor létrehozza azt. Ilyen metódus például a **values()**, amely egy olyan tömbbel tér vissza, amely tartalmazza a felsorolásos típusnak a deklaráció sorrendjében megadott összes értékét.

Minden enum implicit módon kiterjeszti a **Java.lang.Enum** osztályt.

37. C#

A C# nyelvet a .NET fejlesztette ki, a Java utódjaként. A nyelv egyik különlegessége, hogy menet közben optimalizálja a programozó által megkonstruált kódot. Emellett kezeli a szemétgyűjtögetést, a memóriát, illetve a biztonsági szintet a kódhoz való hozzáféréshez.

Azonosító: _ betűt is tartalmazhat.

Kulcsszavak: alapszavak vannak, azonban standard azonosítók itt sincsenek.

Megjegyzések: sorvégéig tartó (//), tetszőleges elhelyezésű (/* … */).

Címke: bármely utasítás címkézhető.

Típusok:

- Érték (**egyszerű**: sbyte, ushort, uint, ulong, char, float, double, bool, decimal | **enum** | **struct**)
- Referencia (osztály, interfész, tömb, delegált)

38. C# eszközök

Literálok:

Decimális és hexadecimális alakban is megadhatóak.

Nevesített konstansok:

Beépített nevesített konstansok: null, false, true (ezek alapszavak is).

Saját nevesített konstans definiálása:

típus név = *kifejezés*; [, név = *kifejezés*] … **;**

Változók:

Deklaráció alakja: típus név [= *kifejezés*] [, név = [= *kifejezés*]] … **;**

Kifejezések:

Az operátorok objektumorientáltak és C nyelvben is használt precedencia-táblázat szükséges kiértékelésükhöz.

Utasítások:

- üres utasítás,
- utasítási kifejezés (értékadás, példányosítás, metódushívás),
- if utasítás,
- switch utasítás (case utasítást ugró utasítással kell zárni (például: kivételdobás, break, goto, continue, return),
- while, do while, for, foreach ciklus,
- goto (alakjai: goto címke, goto case *konstans.kifejezés*, goto default),
- using() (az utasítások végrehajtása után az erőforrások felszabadítása következik).

Blokk:

A blokkokat { } jelekkel jelöljük.
Egy blokk tartalmazhat változó és nevesített konstans deklarációt is.
A blokkok kezelése statikus, nem újradeklarálhatóak, de az élettartamuk dinamikus.

Attribútumok:

Az attribútumok saját eszközökhöz rendelhetőek, amelyek futás közben lekérdezhetőek, mint információk.
Az attribútumok az **Attribute** osztályból származnak.

39. Névtér és fordítási egység a C# nyelvben

A C#-ban egyszerre fordulnak le a fordítási egységek, azonban a neveknek egyedieknek kell lenniük. A C# fizikai egysége az **assembly**. Ezek a fizikai egységek külön telepíthetőek, gyűjteményeknek is nevezhetjük őket. Az assembly tartalmazhat típusokat, azok végrehajtható kódját, illetve hivatkozhat is más assembly-re. Egy assembly kiterjesztése: **exe**.

40. C# nyelvbeli szintaktika

Using direktívák:

using névtér.név; (másik névteret importál).

Globális attribútumok:

Assembly-k és modulok számára adnak információt.

Névtér-deklaráció:

namespace *minősített.név* { { [*using.direktívák* | *névtér.tag.deklarációk*] } } (hivatkozásuk minősített névvel történik).

Típusdeklaráció:

{ *osztály* | *interfész* | *struct* | *enum* | *delegált* } (lehet tag névtérben, osztályban és struktúrában).

Osztálydeklaráció:

[*attribútumok*] [*módosítók*] **class** név [*bázis*] törzs [**;**]

Bázis:

: { *osztálynév* | *interfész* [, *interfész*] … | *osztálynév*, *interfész* [, *interfész*] … }

Elérési szintjének legalább olyannak kell lennie, mint származási osztályának, az elérési szint tehát csak szűkíthető.

Törzs:

(*osztálydeklacárió*) { [*tagdeklarációk*] }

Tagok:

{ *nevesített konstans* | *mező* | *metódus* | *tulajdonság* | *esemény* | *indexelő* | *operátor* }

Static módosító nélkül a tag példány-szintű lesz.
Láthatósági módosítók: private, public, protected, internal, protected internal.

Nevesített konstans:

[*attribútumok*] [*módosítók*] **const** típus név = *kifejezés* [, név = *kifejezés*] … **;**

Mező:

[*attribútumok*] [*módosítók*] típus *változódeklarációk* **;**

Többszálú programok esetén használatos a **readonly** és a **volatile** attribútum (szinkronizálás szükséges a helyes működéshez).
A mező kezdőértékét beállíthatjuk explicit kezdőértékadással vagy konstruktorhívással.

Újra-deklaráció elfedhet egy örökölt nevet. Öröklődéskor a konstruktorok nem öröklődnek.
Absztrakt osztályban csak absztrakt módosító szerepelhet.

41. Metódusok a C# nyelvben

Metódusok szerkezete:

- **Fej**: [*attribútumok*] [*módosítók*] típus név ([*formális.paraméterlista*])
- **Törzs**: { *blokk* | **;** }

Legyen N egy metódus, A az aktuális paraméterlista, C egy deklarálandó típus, R pedig egy futásidejű parancs. Ekkor:

- C, N és A segítségével a rendszer paraméterkiértékeléssel kiválaszt egy M metódust C-ből vagy annak egy elődosztályából.
- Ha M nem virtuális, akkor A hívódik meg.
- Ha M virtuális, akkor az R-ben ismert implementációja fog lefutni.

Egy metódust felülírhatunk az **override** módosító segítségével, ha az virtuális, absztrakt vagy már alapból egy felülírt metódus. A felülírt metódus **specifikáció**jának **azonos**nak kell lennie az eredetivel és a **láthatóság**a **sem módosítható**. Külső metódusok felülírásához az **extern** módosító szükséges. A metódusok **túlterhelhetőek**.

42. Paraméterátadás a C# -ban

A paraméterátadás módja lehet:

- érték szerinti,
- referencia (**ref** módosító) szerinti,
- eredmény (**out** módosító) szerinti.

Van azonban egy negyedik paraméterátadási technika, amelyet **tömb paraméterátadás**nak hívunk. Így, ha az aktuális paraméterlista egy konvertálható típusú kifejezés, akkor érték szerinti lesz az átadás, egyébként változó paraméterszámú a metódus. Ekkor az aktuális paraméterlista értékeiből létrejön egy tömb-példány, és ebből operál a metódus.

43. Hozzáférők a C# nyelvben

A **hozzáférő** (tulajdonság ~ property) olyan tag, amely az objektum valamilyen belső állapotának elérését teszi lehetővé. Tekinthetjük úgy, mint a mező természetes kiterjesztését, azonban ennek nincs címe. A nevével hivatkozható, és hivatkozáskor megadja azokat az utasításokat, amelyekkel írni és olvasni tudjuk. Egy tulajdonság a **get** és **set** metódustól függően lehet [csak] olvasható { vagy | és } [csak] írható.

Get: Paraméter nélküli metódus. A deklarációban megadott típussal tér vissza. A blokkjának meg kell határoznia a tulajdonság értékét.
Set: Egyetlen implicit paramétere van, visszatérési értéke void. A tulajdonság állapotát állítja be.

44. Események

Az esemény a C# üzenet-kezelő eszköze. Tagonként definiálhatunk eseményeket, amelyek bekövetkeztében üzenetek generálódnak és ezekhez **üzenetkezelő**t definiálhatunk, amely maga egy végrehajtható kód. **Az esemény nem egyezik meg a kivétellel**! Az esemény egy olyan delegáltat tartalmazó mező, amely a delegálttal egy eseménykezelőt hivatkozik.

Az **eseményelérő** segítségével az **eseménykezelő**ből eltávolíthatunk, illetve hozzáadhatunk eseményt. Külső esemény nem tartalmazhat eseményelérőt. Egy esemény hozzáadható egy eseménykezelőhöz a += operátorral, illetve eltávolítható a -= operátorral.

Eseményelérő alakban az eseménykezelő megadja, hogy hogyan kell az eseménykezelőt hozzáadni, illetve elvenni az eseménytől. Egy eseményelérőhöz egy void típusú és egyetlen paramétert tartalmazó metódus tartozik. A blokk ennek a metódusnak a törzsét adja meg, ezért igazodnia kell hozzá.

45. Indexelők

Az indexelő olyan tag, amely segítségével az osztály egy objektuma - ha az kollekció - tömbként indexelhető.

```
public Osztály this[int index]
{
   get { return (Osztály)kollekció[index]; }
}
```

46. Operátorok

Az osztály példányaihoz definiálhatóak operátorok.

[*attribútumok*] [*módosítók*] { *unáris* | *bináris* | *konverziós* } *blokk* **;**

- **unáris operátor**: tipus **operator** { + | - | % | ~ | ++ | -- | true | false }
- **bináris operátor**: tipus **operator** {+ | - | * | / | % | & | | | ^ | << | >> | == | != | > | < | >= | <= }
- **konverziós**: { **implicit operator** típus (*típus.azonosító*) | **explicit operator** típus (*típus.azonosító*) }

47. Konstruktorok és destruktorok

Létezik osztály- és példányszintű konstruktor. A példány-konstruktorok felépítése a következő:

[*attribútumok*] [*módosítók*] osztálynév ([*formális.paraméterlista*]) [*inicializáló*] *blokk* **;** // **:** { **base** | **this** } ([*paraméterek*])

A C# nyelvben minden konstruktor törzsének elején **implicit** módon meghívódik egy másik konstruktor. Ezt adja meg az **inicializáló**. A **base** használta esetén a szuperosztály konstruktora hívódik meg. **This** esetén a saját osztálybeli konstruktor hívódik meg. A megfelelő konstruktor kiválasztása a paraméterlista-kiértékeléstől függ. Ha nincs megfelelő konstruktor, akkor az fordítási hiba. Ha nincs inicializáló, akkor implicit módon egy **base()** metódus kerül az osztályba.

Példányosításkor először az explicit mező **kezdőértékadás**ok hajtódnak végre a deklarációk sorrendjében, azután az implicit konstruktorhívás vagy inicializálás, majd a konstruktor törzse. Ha nem adunk meg konstruktort, automatikusan felépül egy, alapértelmezett **base()** hívással.

Alapértelmezett konstruktora absztrakt osztálynak is van, **protected** láthatósággal. A konstruktorok láthatósága **más esetekben public**. **Privát konstrukturok**nak akkor van értelme, ha meg akarjuk akadályozni, hogy az alapértelmezett konstruktor létrejöjjön. Például, ha nem akarjuk, hogy az osztálynak példánya legyen. A **statikus konstruktor**ok privátak és nem örökölhetőek, segítségükkel osztályokat inicializálhatunk.

A **destruktor**ok példányok explicit megszűntetésére valók. A destruktorok **nem örökölhető**ek, valamint explicit módon sem hivatkozhatóak. Egy ilyen destruktor akkor fut le, ha egy példányt az adott osztályból nem használ már semmilyen kód.

48. Interfészek

Egy interfész formai alakja:

[*attribútumok*] [*módosítók*] **interface** név [*bázis*] { *tagdeklarációk* } [**;**]

Minden interfész tartalmazhat metódust, tulajdonságot, eseményt és indexelő specifikációkat. Az interfészek között **többszörös öröklődés** van. Egy osztály vagy struktúra akárhány interfészt implementálhat. Ekkor az alosztályok minden – az interfészben definiált - tagot örökölnek, a névütközést pedig teljes minősítéssel kerülik meg.

49. Enum típusok

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. Deklaráció alakja:

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

50. Delegáltak a C# nyelvben

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 **egyszerű**. 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.

51. Struktúrák

A struktúráknak van adatrésze és tagfüggvényei is, de a struktúrát ettől függetlenül **értéktípus**nak tekintjük. A program működése során nem használják a dinamikus memóriát (halmot), csakis kizárólag a vermet.

Tag lehet egy struktúrában: nevesített konstans, mező, metódus, konstans, operátor, típus, tulajdonság, esemény, indexelő, konstruktor. Destruktor azonban nem lehet tag.

Az adatelemek alapértelmezetten nullázódnak. Értékadásnál az érték másolódik, nem pedig a referencia. A **this** kulcsszó mindig egy értéket jelöl. Példánykonstruktornál a this egy **out**-paraméter, példányfüggvénynél pedig egy **ref**-paraméter a struktúra típusával, mint egy változó.

52. Tömbök

C#-ban a tömbök egy absztrakt osztály leszármazottai. Lehetnek többdimenziósak is, indexeik típusa pedig int, uint, long, ulong vagy ezen típusokká konvertálható kifejezések.

53. Kivételkezelés a C# nyelvben

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.

54. Egy C# program futtatása

A C# nyelvnek van egy előfordítója, amelynek van egy opcionális feltételes előfordítója. Ez a feltételes előfordító egy nagyon hatékony optimalizáló eszköz.

A C# alkalmazás egy olyan assembly, amelynek van belépési pontja (ez a Main() függvény). Egy alkalmazás futtatásakor egy alkalmazási domain jön létre. Egy alkalmazásnak különböző példányai létezhetnek különböző alkalmazási domainekben ugyanazon a gépen, ugyanazon időben. Az alkalmazási domain úgy működik, mint egy **konténer**. Behatárolja az alkalmazásban használt típusok hatáskörét. A domainek között a példányok nem oszthatóak meg.

Egy alkalmazáson belül **több belépési pont** is megengedett, ekkor az indításkor meg kell adni, hogy melyik legyen a fő belépési pont. Ennek módja, hogy egy parancssori argumentummal megadjuk ezt a belépési pontot.

55. A funkcionális paradigma alapfogalmai

Középpontjában a függvények állnak. Más szóval **applikatív nyelvek**nek is hívják ezeket a nyelveket.

Ezekben a nyelvekben a program nem más, mint típusdeklarációk, osztálydeklarációk, függvénydeklarációk és függvénydefiníciók sorozata. A program része még a kezdeti kifejezés, amelyben tetszőleges hosszúságú, egymásba ágyazott függvény hívássorozat jelenhet meg. Egy ilyen program végrehajtását a **kezdeti kifejezés** kiértékelése jelenti. A kezdeti kifejezésben szereplő függvények meghívása úgy zajlik, hogy a hívást szövegszerűen a paraméterek figyelembevételével helyettesítjük a definíció törzsével.

A funkcionális nyelvek nyelvi rendszerei alapvetően **interpreter alakúak**, interaktívak, de tartalmaznak fordító programokat is. Fordító motorjuk (magjuk) egy **redukciós** (átíró) **rendszer**. Ha az egyes részkifejezések átírásának a sorrendje nincsen hatással a végeredményre, akkor ezt **konfluens**nek nevezzük.

A nyelv legfontosabb építőelemei a **saját függvények**. A függvények törzse kifejezésekből áll. Saját függvényt beépített vagy általunk már korábban definiált függvények segítségével tudunk definiálni. A beépített függvényeket a nyelvi rendszer biztosítja. A függvények alapértelmezetten lehetnek rekurzívak, illetve létrehozhatóak **kölcsönösen rekurzív függvények** is.

Kiértékelési stratégia:

A kezdeti kifejezés **redukálás**a mindig egy **redukálható részkifejezés** (redex) átírásával kezdődik. Hogy ha a kifejezés már nem redukálható tovább, akkor egy **normál formájú** kifejezést kaptunk.

A funkcionális nyelvek 2 kiértékelési stratégiát különböztetnek meg:

Lusta kiértékelési stratégia: Szöveg szerinti paraméterátadás.

A kifejezésben először a legbaloldalabbi, legkülső **redex** kerül átírásra, amely azt eredményezi, hogy ha egy kifejezés az függvényhívás, akkor az aktuális paraméterek kiértékelését csak akkor végzi el, ha szükség van rájuk. Mindig eljut a normálformáig, ha az létezik.

Mohó kiértékelési stratégia: Név szerinti paraméterátadás.

A kifejezésben először a legbaloldali, legbelső redex kerül átírásra. Az aktuális paraméterek kiértékelése történik meg először. Gyakran ez a hatékonyabb, de nem biztos, hogy véget ér, még ha a kifejezésnek létezik is a normálformája.

Tisztán funkcionális (tisztán applikatív) egy nyelv, ha nyelvi elemeinek nincs mellékhatása, illetve ha nincs benne lehetőség értékadásra vagy más eljárás orientált nyelvi elem használatára. Ekkor teljesül a **hivatkozási átláthatóság**. A nem tisztán funkcionális nyelvekben van mellékhatás és a nyelvben lehetőség van más nyelvi elemek használatára.

Hivatkozási átláthatóság:

Egy kifejezés értéke nem függ attól, hogy a program melyik részén fordul le. Tehát ugyanolyan kifejezés értéke a program bármely pontján azonos. A hivatkozási átláthatóságot megvalósító nyelvekben tehát nincsenek változók csak [nevesített] konstansai.

Funkcionálok:

Magasabb rendű függvények. Ezek a függvények eszközként tartalmaznak olyan függvényeket, amelyek paramétere vagy visszatérési értéke is maga egy függvény. A funkcionálok **proceduális absztrakciót** szolgálnak. A funkcionálokat megvalósító nyelvek egy részében a kivételkezelési eszköztár gyenge vagy nem is létezik, másik részében igen hatékony és bő készlet áll rendelkezésre.

A saját függvények definiálásra alkalmas függvénykészlet asszociatív, így a funkcionális nyelvben írt programok kiértékelése jól párhuzamosítható. Az elterjedt funkcionális nyelveknek általában van párhuzamosított változata.

Funkcionális nyelvek:

- HasKell (tisza, lusta)
- LISP (imperatív eszközöket is tartalmaz)
- CLOS (objektumorientált, mohó)
- Erlang
- F#