Jegyzet - 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

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.

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.

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.

Az objektumok kommunikációja üzenetküldés formájában történik.
Üzenetküldés
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- és példányszintű attribútumok és metódusok

- Az osztály szintű attribútumok az osztályban képződnek és nem többszöröződhetnek. Osztályhoz kötődnek, nem példányhoz. Egyszer kerülnek meghívásra, az osztály példányainak számától függetlenül. pl.: osztály ***kiterjedés***e, amely megadja, hogy az adott pillanatban hány példánya van az osztálynak.
- A példány szintű attribútumok azonban minden példányosításnál elhelyeződnek a memóriában .
- A példány szintű metódusok a példányok viselkedését határozzák meg.

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.

Beállító, lekérdező metódusok

Attribútumok lekérdezésére, beállítására szolgálnak.

- A beállító metódusok hatására valamely attribútum (akár az összes) értéke megváltozik, ezek a metódusok eljárás jellegűek -> az új értéket paraméterként határozzuk meg.
- A lekérdező metódusok függvény jellegűek és az aktuális példány aktuális állapotával térnek vissza.

Az osztályszintű lekérdező metódusok az osztály attribútumainak értékeivel térnek vissza. Az osztályszintű beállító metódusokkal az osztály attribútumait állíthatjuk be.
Példányszintű beállító metódusok használatával az aktuális példány állapotot vált.

Példányosítás

Lefoglalódik egy memóriaterület a példány számára. Itt elhelyezésre kerülnek a példány szintű attribútumok, így beállítódik a kezdőállapot. Az objektum innentől kezdve él, és öntudattal rendelkezik. Az aktuális példány kezelését a példányszintű metódusokba implicit módon beépülő paraméterrel oldják meg.

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
Az öröklődés további tulajdonsága a helyettesíthetőség, ami az öröklődésen alapul, az újrafelhasználhatóságnak egy megnyilvánulása.

Liskov - féle helyettesíthetőség-elve
Egy leszármazott osztály példánya minden helyen elhelyezkedhet a program szövegében, ahol az előd osztály egy példánya szerepel.

A modern programok a helyettesíthetőséget fordítási időben végzik. A leszármazott osztály objektumai mindig helyettesíti a szülő osztály objektumait, fordítva ez nem igaz.

Láthatatlan öröklődés
Privát eszközt örököl az alosztály, erre az eszközre az alosztály nem tud hivatkozni.

Delegáció, továbbküldés
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.

Bezárás
Az alosztályok eszközeinek láthatóságát szabályozza.
Szintjei:
- *Publikus:* az összes kliensosztály látja.
- *Védett:* az adott osztály és a leszármazott osztályok látják.
- *semmi:* az adott osztállyal egy csomagban lévők 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.

Az alábbi táblázat mutatja a módosítók hatását:

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

Metódusok újraimplementálása (polimorfizmus)
Egy alosztály az örökölt metódusokat újraimplementálhatja. Ez azt jelenti, hogy különböző osztályokban azonos metódusspecifikációkhoz különböző implementáció tartozik. Az ilyen metódusokat **polimorf metódus**oknak nevezzük. Ezek után a kérdés az, hogy ha meghívunk egy objektumra egy ilyen metódust, akkor melyik implementáció fog lefutni. Polimorf metódusok esetén kötéstől függő, hogy melyik implementáció fut le.
Kötés (nyelvi mechanizmus)
- *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.

Metódusnevek túlterhelése (overloading)
Ez annyit jelent, hogy egy osztályon belül azonos nevű, eltérő implementációjú metódusokat tudunk létrehozni. Ezeket a metódusokat a formális paraméter listájuk különbözteti meg.
A formális paraméterek sorrendje, száma, típusa. Vannak olyan nyelvek ahol ez sem elég.

Absztrakt osztály
Segítségével viselkedésmintákat adhatunk meg, amelyeket majd valamelyik leszármazott osztály fog konkretizálni. Az absztrakt osztály tartalmazhat absztrakt metódusokat, amelyeknek csak specifikációjuk van, implementációjuk nincs. Az absztrakt osztályok nem példányosít hatók, csak örököltethetők. Az absztrakt osztályból származtatható absztrakt és konkrét osztály is. Egy osztály mindaddig absztrakt, amíg legalább egy metódusa absztrakt. Ha egy osztálynak van olyan metódusa, amelyhez nem tartozik implementáció, akkor az absztrakt. Attól lesz absztrakt egy osztály, hogy annak deklarálom! Nem muszáj absztrakt metódusának lennie.

Konkrét osztály
Minden metódusához tartozik implementáció.

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.
Az interfész nem példányosítható, helyette implementálható egy osztály által, és kiterjeszthető más interfészek által(extends).

Paraméterezett osztály (kollekció)
Generikusoknak felelnek meg, amelyeknek egy jellegzetes példája a konténer osztály.

Konténer
Olyan osztály, amelynek objektumai maguk is objektumokat tartalmaznak. Tulajdonképpen objektumokból álló adatszerkezetek konkretizálására szolgálnak. Ilyen osztály a ***kollekció***, amelynek adattípusa például a verem, fa, lista, sor, halmaz vagy a multi halmaz. 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.

Objektumok élettartalma
Az objektumok a memóriában helyezkednek el, memóriában kezelt konstrukciók. Egy objektum mindig ***tranziens***. 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 így később létrehozható egy másik új objektum, amelynek állapota ugyan az lesz. Adatbázis-kezelő rendszerekben az objektumok ***perzisztens***ek, azaz túlélik az őket létrehozó alkalmazást. Minden objektumod explicit módon kell létrehozni (példányosítás). A már szükségtelen objektumok tárterületeit fel kell szabadítani, erre 2 megvalósítás született:
- Az objektumot explicit módon a programozónak kell megsemmisítenie. Pl.: C++
- Szemétgyűjtögetés módszerével a rendszer automatikusan törli a szükségtelen objektumok tárterületét. Pl.: Java, C#, SmallTalk

Automatizmus
A program futásával párhuzamosan vele szinkron, vagy aszinkron módon a rendszer figyeli azt, hogy az adott objektumot hivatkozza –e még bármely más eszköz. Ha nem, akkor azt az objektumot előbb vagy utóbb megszűntetheti.

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.

Az OO nyelveknek két nagy csoportja van
- ***Tiszta:*** Teljes mértékben az OO paradigma mentén épülnek fel, ezekben nem lehet más paradigma eszközeinek segítségével programozni. Ezen nyelvekben egyetlen osztályhierarchia létezik. Ez adja a nyelvi rendszert és a fejlesztői környezetet is egyben. Ezen nyelvekben a programozás azt jelenti, hogy definiáljuk a saját osztályainkat, azokat elhelyezzük az osztályhierarchiában, majd példányosítunk.

- ***Hibrid:*** A hibrid OO nyelvek valamilyen más paradigma (eljárás orientált, funkcionális, logikai, stb.) mentén épülnek fel, és az alap eszközrendszerük egészül ki OO eszközökkel. Ezen nyelvekben mindkét paradigma mentén lehet programozni. Általában nincs beépített osztályhierarchia (hanem pl. osztálykönyvtárak vannak), és a programozó saját osztályhierarchiákat hozhat létre.

Egyes tiszta OO nyelvek az **egységesség elvé**t vallják. Ezen nyelvekben egyetlen programozási eszköz van, az objektum. Ezekben a nyelvekben tehát minden objektum, a metódusok, osztályok is.

Az OO paradigma **imperatív paradigma**ként jött létre. Tehát ezek a nyelvek algoritmikusúak, és így eredendően fordítóprogramosak. A SIMULA 67 az első olyan nyelv, amely tartalmazza az OO eszközrendszert, de a paradigma fogalmait később a Smalltalk fejlesztői csapata tette teljessé. Aztán folyamatosan kialakultak a hibrid OO nyelvek, és megjelent a deklaratív OO paradigma (CLOS, Prolog++) is.

Java

A Java és C# egyszeres öröklődést, késői kötést, automatikus memóriakezelést valósítanak meg. 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 platform független.

Karakterkészlet

UTF-16 kódolású, a Unicode szabványnak megfelelő karakterkészlet. A kis- és nagybetűk a nyelvben meg vannak különböztetve. A nemzeti nyelvek karakterkészletét is magában foglalja.

Azonosító

betű karakteren kívül _ és $ jelet is tartalmazhat.

Címke

A végrehajtható utasítások megjelölésére szolgál, bármely végrehajtható utasítás megcímkézhető. Azért alkalmazzuk, hogy a program egy másik pontjáról hivatkozni tudjunk rá. A címke egy azonosító, az utasítás előtt áll kettősponttal elválasztva. A címkét a break vagy a continue paranccsal együtt használjuk úgy, hogy megadjuk a parancs után azt, hogy melyik egységre vonatkozik.

Megjegyzés

A megjegyzés olyan programozási eszköz, amely arra szolgál, hogy a program írója a kód olvasójának valamilyen információt szolgáltasson.

- sorvégéig tartó (//) - tetszőleges elhelyezésű (/* … */)
- dokumentációs (Javadoc nevű program dolgozza fel) (/** … */).

Tömb

A tömb a referenciatípusok közzé tartozik, saját jelölése van. Statikus adatszerkezetet valósítanak meg, fix elemszámúak lesznek. A tömböknél az elemeket 0-tól indexeljük. Az utolsó szabályos index az elemszám-1 - edik index. Általános deklarációs alakja: típus[] azonosító; vagy: típus azonosító[]; Az elemszámot nem feltétlenül a deklarációban kell megadni: a tömb az egy referenciatípus, és mint ilyen típus valamilyen osztályhoz kötődik. Így példányosítani kell. Így ha deklarációban nem adjuk meg a tömb elemszámát, az a példányosításnál fog eldőlni. Több lehetőség van erre: int[] a; a = new int[10]. A new kulcsszó a példányosítás operátora. A new hatására mindig valamilyen memóriaművelet zajlik a hátérbe. A new int[10] hatására valahol a memóriában elfoglalunk az objektum tárolására elegendő tárhelyet, melyben 10 db int értéket lehet tárolni. Az 'a' viszont már megszületett korábban, ő egy referenciaelem volt. Így a későbbiekben bármikor összerendelhető a tömb objektummal. Ő maga nem tartalmazza az objektumot, csak hivatkozik rá. Ha később lefoglalunk egy 15 elemű tömbnek elegendő tárterületet az 'a = new int[15];' utasítással akkor az a erre fog hivatkozni. Az előző objektumra, a 10 elemű tömbre már nem hivatkozik senki, tehát ez meg fog szűnni. Legközelebb a **garbage collector** fog vele foglalkozni, amikor fel akarja szabadítani a tárterületet.

Nevesített konstans

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

Kulcsszavak

Olyan karaktersorozat, amelynek a nyelv tulajdonít jelentést, és ez a programozó által nem megváltoztatható. Standard azonosítók nincsenek. Jelenleg a 8-as JAVA 50 darab karaktersorozatot definiál, melyek mindegyikét ASCII karakterek alkotják. Kulcsszóként való használatra vannak fenntartva, nem használhatók azonosítóként! A 'const' és a 'goto' fenntartott szavak, bár jelenleg nincsenek használatban. A 'true' és a 'false' logikai literálok, nem kulcsszavak. A 'null' a null literál, nem kulcsszó.

Változó

A változó olyan adatelem, amely azonosítóval van ellátva. Az egyik legfontosabb programozási eszközünk. A változókat használat előtt deklarálni kell. Ez azt jelenti, hogy meg kell adni a típusát, nevét, esetleg értékét is. Amikor egy változónak kezdeti értéket adunk akkor a változót **inicializál**juk. pl.: byte a; int x = 0;

Egy változót a program tetszőleges részén deklarálhatunk. A változó **érvényességi tartomány**a a programnak az a része, ahol a változó használható. A változódeklaráció helye határozza meg az érvényességi tartományt.

A változó lehet (A deklarálás helyétől függően.):

- **tagváltozó**: Az osztály vagy objektum része, az osztály egészében látható (alkalmazható). Az osztályváltozók deklarálásánál a 'static' kulcsszót kell használni.
- **lokális változó**: Egy kódblokkon belül alkalmazzuk. A láthatósága a deklaráció helyétől az őt körülvevő blokk végéig tart.

A metódusok formális paraméterei az egész metóduson belül láthatók.

A változót lehet véglegesen is deklarálni (**konstans változó**), a végleges változó értékét nem lehet megváltoztatni az inicializálás után. A végleges változók deklarációnál a 'final' kulcsszót kell használni: pl.: final int aKonstans = 0;

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.

Operátorok

38 db operátor van.

- Bitenkénti eltolás: >>> bal oldalra mindig egy 0- at fog behozni
  >> előjeltől függően hozza be a 0- at vagy 1-et

& bitenkénti ÉS ^ bitenkénti VAGY ?: három operandosú && logikai és

multiplikatív: %, +, -

Típusok csoportosítása

Alapvetően két nagy típus van: primités, és referencia típusok.

- Primitív és referencia (osztályok, interfészek, felsorolások(enumerációk), tömbök).
- Beépített típusok (boolean, char, byte, short, int, long, float, double, csomagoló osztályok).
- Speciális típus a null típus, amelynek nincs is külön neve.

A logika típusokkal logikai műveleteket, összehasonlító műveleteket lehet végrehajtani.

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.

Deklarációs és végrehajtható utasítások
- A deklarációs utasítások mögött nem áll tárgykód. A programozó a névvel rendelkező saját programozási eszközeit tudja deklarálni.
- A végrehajtható utasításokból generálja a fordító a tárgykódot. Melyek ezek?
  - Üres utasítás
  - Kifejezés utasítás (pl. 8; a = b;)
  - Vezérlő utasítások (szekvencia, feltételes utasítás, többszörös elágaztatás, ciklusok, vezérlésátadó utasítások)
  - Módszerhívások (metódushívások)
  - Objektumot létrehozó kifejezések

Alprogramok
A programot célszerű kisebb egységekre (alprogram) bontani, melyeket a program bizonyos pontjairól aktivizálhatunk. Így áttekinthetőbbé, olvashatóbbá válik a forrásprogram, valamint a többször előforduló tevékenységeket elegendő egyszer megírni (elkészíteni). A Java- ban metódusnak nevezzük az alprogramot. A metódus utasítások összessége, melyet meghívhatunk a metódus nevére való hivatkozással. Egy metódus lehet eljárás vagy függvény aszerint, hogy van-e visszatérési értéke.

Eljárás
Az eljárásnál nincsen visszatérési érték, egyszerűen végrehajtódik az eljárás nevére való hivatkozással. A végrehajtás után a program azzal az utasítással folytatódik, amelyik az eljárást meghívó utasítást követi. Az eljárás visszatérési típusa void (semleges), ami azt jelenti, hogy a visszatérési típus nincs definiálva.

Függvény
A függvénynél van visszatérési érték. A függvényt is egyszerűen a nevére való hivatkozással hívunk meg, de visszaad egy értéket, melyet a függvény neve képvisel.

Blokk
A blokk nulla vagy több utasítás kapcsos zárójelek között, amely használható bárhol, ahol az önálló utasítások megengedettek.
Jellemzői:
- { } zárójelek között szerepel.
- Címkézhető.
- Tetszőleges mélységben egymásba skatulyázható.
- A változó a blokk lokális változójaként deklarálható.
- A blokkon belül tetszőleges a deklarációs- és végrehajtható utasítások sorrendje.

A nyelv vezérlési szerkezete

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.

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.

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.

Osztályok és példányok a Java nyelvben
Az OO paradigma egyik alapfogalma.
Deklarálása:
1) Módosítók (opcionális) Pl.: public, static, final, abstract stb.
2) Osztály neve a „class” kulcsszó után
3) Szülő osztály neve az „extends” kulcsszó után (opcionális)
4) Az implementált interfészek nevei egymástól vesszővel elválasztva az „implements” kulcsszó után (opcionális)
5) Az osztály törzse {} között

Pl.: public class A {}
A Java nyelv egyszeres öröklődést vall az osztályok között, ezért ha ki van írva az extends kulcsszó, akkor az az osztály lesz az adott osztály szülőosztálya, amelyek a neve az extends kulcsszó után szerepel. Ha az extends hiányzik, akkor az adott osztály szülőosztálya az object osztály lesz, mint ahogyan a példában is. Tetszőleges számú interfészt implementálhat az osztály, kiterjeszteni viszont csak egyet terjeszthet ki, a szülőosztályát.
Az osztály egy absztrakt adattípus implementációja. Minden osztály egy fejből és egy törzsből áll.

A fej szerkezete:
[módosító] class osztálynév [extends szuperosztály_név] [implements interfész_lista] (az alosztályok között csak egyszeres öröklődés van)
Az osztály lehetséges módosítói:
1) láthatósági módosítók:
   Külső szinten definiált osztályhoz csak public és semmi használható, ha az osztály definíciója egy másik osztályon belül található, akkor mind a 4 használható.
   - public: bármelyik kliens osztály elérheti és használhatja az osztályt
   - protected: a leszármazott osztályok és az adott osztállyal egy csomagban lévő osztályok érhetik el.
   - semmi: csak azok az osztályok tudják elérni, amelyek vele egy csomagban vannak
   - private: csak a definiáló osztályból érhető el

2) abstract módosító: Lehet belső, és külső osztály egyaránt. Az osztályt a tervezője abstractnak definiálta. Nem lehet példányosítani, viszont más osztályok szülőosztálya lehet.

3) static módosító: Csak beágyazott osztályoknál fordulhat elő, osztályszintű tagságot jelent.

4) final módosító: Ez az osztály nem örököltethető, ő egy levélelem lesz az osztályhierarchiában, mindkét szinten megjelenhet.

Osztálynév
Azonosító jellegű karaktersorozat. Nagy kezdőbetű, amennyiben szóösszetétel, akkor a szavak kezdőbetűje nagybetű, a többi kicsi.

Extends szuperosztálynév
A definiált osztálynak közvetlen szülőosztálya, ha ez nincs, akkor közvetlenül az object osztály leszármazottja.

Implements interfésznév- lista
Az osztály milyen interfészeket akar implementálni, több név lehet, vesszővel vannak elválasztva.

A törzs szerkezete
{} jelek közé írt tagdefiníciók tetszőleges sorrendben. Nem blokk!!!
A törzs tartalmazhat mezőt (lehet példány és osztályszintű), metódusokat (-//-), konstruktorokat, típusdefiníciókat (leggyakoribb fajtájuk a beágyazott osztály, beágyazott interface, vagy beágyazott enumeráció).

Beágyazott osztályok/ belső osztályok
Osztályokat más osztályokon és módszereken belül is lehet definiálni, amelyeket beágyazott osztályoknak hívunk. Egy osztály **tagosztály**ának nevezzük azt az osztályt, amelyet az osztály tagjaként definiáltunk. Az ilyen osztály ugyanúgy örökölhető és elfedhető. A beágyazott osztályok két típusa a statikus tagosztályok és a nem statikus tagosztályok (belső osztály).

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 osztályoknak** hívjuk.

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, ha nincs, akkor a this változó azon osztály aktuális példányát fogja hivatkozni, amelynek a beágyazásból kifelé haladva van aktuális példánya.

Két speciális belső osztály: lokális és névtelen osztályok.

Lokális és névtelen osztályok

- A lokális osztályok 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 az 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.

Tagváltozók (mezők, fields)

[módosító] típus változónév [= kezdőérték];
A mezők az osztályban deklarált változók lesznek. *Két fajtájuk van:* osztályszintű és példányszintű mező. A mezők deklarációja közvetlen az osztály törzsén belül helyezkedik el.
Deklarációjuk:

1) Módosítók (opcionális) pl: public, final stb
2) Típus
3) Mezőnév
4) Kezdő értékadás (opcionális)
5) Pontosvessző

Példányszintű mező példa: private int eletkor;
Osztályszintű mező példa: public static final int NYUGDIJKORHATÁR = 65;

Módosítók:
láthatósági, static, final

- Static: Ha szerepel a static kulcsszó akkor osztályszintű mezők lesznek, különben példányszintű mezők lesznek.
- Final: A final módosító használatával a mező a program futása során csak egyszer kaphat értéket. Kaphatja rögtön a deklarációban, illetve kaphatja később is, de csak egyszer.
- Láthatósági módosítók csoportja:
    1) public – a deklaráló osztályból, az osztályból leszármazott osztályból, az osztállyal egy csomagban levő osztályokból és kliens osztályokból látható (a nagyvilágból elérhető)
    2) protected – Az adott osztályból és belőle leszármazott osztályból látható. Lehetővé teszi az azonos csomagból történő elérhetést is.
    3) (semmi) - Csomagszintű láthatóságot definiálja. Az adott osztályból, és a vele egy csomagban levő osztályokból látható. Pl: int a;
    4) private – Az ilyen mezőt csak a deklaráló osztályból lehet elérni és hivatkozni.

Ez a sorrend un. erősorrend. Az 1. valósítja meg a leglazább adatrejtést, a 4. a legszigorúbbat.

Kezdőérték adás '=' jel után lehet. Ha nem adunk neki kezdőértéket, akkor alapértelmezett értéket kap, a 0 bitkombinációt. Referencia típusnál null, logikainál false.

A tagváltozókkal egyből lehet dolgozni, de a lokális változókkal nem, mert nem kapnak automatikusan kezdőértéket.

Metódusok (methods):

Deklarálásuk:

[módosítók] típus metódusnév ([f.p.l]) [throws kivételosztálynév_lista] törzs

Pl: public static void main (String args){}

Módosítók lehetnek:

- statikus (osztályszintű), nem statikus (példányszintű).
- final: A metódus örökölhető, de nem implementálható újra.
  Ha az egyik osztály örököl a szülőtől egy metódust, azt újraimplementálhatja.
  Ez a jelenség a polimorfizmus. Ezt tiltja meg a final kulcsszó.
- abstract: Nem lesz törzse, nem ismerjük az imlementációját, a metódus törzse egy ';'
  A specifikációjában szerepelni fog az abstract szó. Azt, hogy egy absztrakt metódus mit fog csinálni, a leszármazott osztályban mondjuk meg: ott írjuk le a metódus implementációját. Pl.: public abstract void eltol (int x, int y)
- láthatósági: public, protected, (semmi), private

Típus:

Primitív, referencia és void. Az első kettő a függvényszerű metódusok, nekik mindig kell, hogy legyen visszatérési értékük. Szabályos befejeztetésük: return kifejezés;
A void metódusok eljárásszerűek nekik nem kell, hogy legyen visszatérési értékük. Szabályos befejeztetésük: return; , vagy a törzs végének az elérése is a metódus szabályos befejeztetése.

Neve:

Azonosító jellegű. Konvenció, hogy kis kezdőbetűvel kezdődik, és a szóösszetételek határán a szavak nagybetűssé válnak.

Formális paraméterlista:

() párral vannak határolva. A () –nek mindig meg kell jelenni, viszont lehet üres is. A C-ből örökölt formájú: vesszővel elválasztott típus paraméternév páros.

A kivételosztály nevek listája:

Annak a leírására szolgál, hogy milyen problémás eseteket, kivételes helyzeteket nem kezel a metódus. Melyek azok a kivételfajták, amelyek kiváltódhatnak a metódusban, és melynek kezelésével nem foglalkozik a metódus. Lehetnek olyan kivételek is, amelyekkel foglalkozik, de azt nem szoktuk odaírni.

A metódus törzse:

Az **egy blokk**, és minden olyan eszköz lehet benne, ami egy blokkban. Az osztály törzse viszont nem blokk! Az osztály törzsén belül a metódusnevek túlterhelhetőek (overloading). Deklarációs és végrehajtható utasítások sorozata. Eljárás szerű metódusok esetén lehet üres is. Függvényszerű metódus esetén kell kilépési pont.

A Java nyelv támogatja a metódusok újraimplementálását a polimorfizmust, és a metódusok túlterhelését is támogatja.

Konstruktor

Speciális metódus, amely a kezdőállapot beállítására szolgál. Ezek szolgálnak az objektumok létrehozására. Konstruktorok nem öröklődnek. Formálisan majdnem úgy néznek ki, mint a metódusok:

[módosítók] konstruktornév ([f.p.l]) [throws kivételosztálynév_lista] törzs

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.

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.
A szülőosztály konstruktorhívása: super (….);
A saját osztály másik konstruktorának hívása: this (….);
Alapértelmezett: super ();

Inicializáló blokk
{
…..
}

Konstruktorok közös kódrészleteit tartalmazhatja, nem kötelező megcsinálni.
Egy osztály törzsében több inicializáló blokk állhat, a végrehajtás a felírás sorrendjében történik.

Statikus inicializáló blokk
static {
…
}
Osztályok statikus adattagjai érhetőek el, tipikusan osztályszintű mezők inicializálására szolgál.

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.

Objektum létrehozása
Mindig valamilyen példányosítással történik.
A példányosítás külön művelettel történik: new konstruktorhívás
konstruktor nevének megadása(aktuális paraméter lista)
pl.: new Paint(3.0, 2.0);

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.

Polimorfizmus (polymorphism) *osztály*
Alapvető jelentése: több alakúság
Egy osztálynak az alosztályai definiálhatnak saját egyedi viselkedéseket ugyanazokhoz a funkcionalitásokhoz, amelyeket már szülőosztályukban már definiáltak.

Kivételkezelés
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. A Java- ban 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 (ez 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 kivételkezelőben bekövetkezett kivételt ugyanígy kezeli a JVM.
<u>*A Javaban 3 fajta kivétel van:*</u> hiba, ellenőrzött kivételek, run time exception
object -> throwable -> error rengeteg alosztály van…
->exception ->)
->) -> checked exceptions (a run time-on kívül)
…..)
-> run time exception

- Hiba (error): Olyan kivételek, amelyeket legtöbbször külső körülmény generálja, nincsenek hatással a programra. pl.: out of memory error

- Az ellenőrzött kivételekre a Java nyelv előír választási lehetőséget, amellyel a programozónak élni kell. Két lehetőség van: catch or specify. Szóval vagy el kell kapni, vagy elő kell írni, meg kell adni, deklarálni kell. Az errorokra és a run time exceptionokre ilyen elvárása nincs.
- 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.

Ellenőrzött kivételek esetén 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 kivételeknek. 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 Exceptionosztálynak 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 utasításokat, emellett láthatóságot is definiál. A catch ágak sorrendje nagyon fontos, mert több ág is elkaphat 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.
A virtuális gép a catch ágak FELÍRÁSI SORRENDJÉBEN keres ALKALMAS vagy MEGFELELŐ kivételkezelőt. A kivételkezelő az a programrész lesz, ami végrehajtja a kivételt.
Az alkalmas akkor teljesül, ha a kiváltódott objektum típusa megegyezik a kivételkezelő ágban szereplő kivételtípussal. Alkalmas az a kivételkezelő, amely a kiváltódott objektum közvetett vagy közvetlen ősosztálya. A finally rész opcionális, nem kötelező hogy legyen.
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, és van finally ág, akkor 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.
Ha a futtató rendszer nem talál alkalmas vagy megfelelő kivételkezelőt, akkor maga a try utasítás vált ki így egy kivételt, és így a program ennél az utasításnál megszakítja a futását, és átadja a vezérlést a futtató rendszernek. Ha a kivétel a mainben keletkezik, akkor keletkezik egy kezeletlen kivétel. Megjelenik majd a kivétel típusa, és az is hogy hol hajtódik végre.

- catch{kivételtípus1|kivételtípus2|…..} {utasítások}

Ennél az esetnél a változónak lényegében finalnak kell lennie, azaz a catch ágban a változóhoz nem rendelhetünk új értéket. A felsorolásban elől vannak a specializált kivétel osztályok, és hátrébb az általános kivétel osztályok, egyébként fordítási hiba lesz!

Ha valamelyik catch ág el tudja kapni a kivételt, akkor a catch or specify közül a catch teljesül, ha nem tudja egyik catch ág sem elkapni, akkor leellenőrzi, hogy a kivétel osztálynév_listában szerepel –e megfelelő osztálynév, mint a kivétel példányosító osztálya, vagy valamelyik ősosztályának neve, ha igen, akkor teljesül a specify. Ha egyik sem teljesül, akkor fog fordítási hibát jelezni.

Catch ág
Egy- egy blokkot nevezünk catch ágnak. A catch szótól a blokk végéig nevezzük kivételkezelőnek.

Erőfforás kezelő try utasítás
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(); } ….

Generikusok
Procedurális absztrakció fejlett eszköze. Típussal paraméterezhető programozási eszközök.
Felhasználásuk:
1) Erősebb típusellenőrzést tudunk így végrehajtani fordítási időben:
Azaz a Java fordítóra bízhatjuk a típus ellenőrzési tevékenységet. Azaz ha valami hibát talál a program a típus konverziónál, akkor le sem fordul így a program.

2) Kiválthatók vele a típuskényszerítések (nincs szükség castolásra)
List list = new ArrayList(); list.add(„hello”); String s = (String)list.get(0);
-> List<String> list = new ArrayList<String>(); list.add(„hello”); String s = list.get(0);

Előnyök:
- Ha a program további részeiben kiolvassuk a sztringet nem kell mindig castolni.
- A java 7-estől = new ArrayList<>(); -> a fordítóprogram a deklaráció alapján kideríti, hogy a típus milyen típus akar lenni.

< > diamond jel

3) Lehetővé teszi a programozónak, hogy általános típust tudjon írni a konkrét típustól függetlenül.

Milyen eszközöket biztosít a nyelv a generikus típusok használatára?
- generikus típusok
- generikus metódusok

Generikus típusok
class név<T1,T2,…..>{
}
T1,T2,…. ezek a típusparaméterek

A szintaktikához kötelező szimbólum a kacsacsőr!
A típusparaméterek 1-2 db karaktersorozatok:
- K - kulcsjellegű információval definiáljuk a paramétert
- N - szám jellegű információ
- T - csak általános típusként
- V - érték jellegű információt akarunk használni

pl.: public class Boksz<T>{
private T t;
public void set(T t) { this.t = t; }
public T get() { return t; }
}
T – bármilyen típusú adatra jó lesz a doboz.

Nyers típusok
Csak generikus osztályoknak van nyers típusa és interfészeknek. Nem nyert típusú referenciát hozzá lehet rendelni nyers típusú referenciához.
Ha egy generikus típust a típusváltozók értékeinek megadása nélkül használunk, akkor az úgynevezett **nyers típus**t kapjuk. A nyers típus arra szolgál, hogy **biztosít**sa az adott osztály generikus és nem generikus típusa közötti **átjárhatóság**ot.

Generikus típusoknál erősen ajánlott megadni a típusparamétereket, és ez alapján példányosítsunk és rendeljünk hozzá referenciát.

Jelölő interfész - a törzse nem tartalmaz semmit se

Generikus metódusok
Olyan metódusok, amelyeknek típusparamétereik vannak. Ezeknek a típusparamétereknek a deklarációját a kacsacsőrben kell, ezeknek a hatásköre csak oda terjed ki ahova őt deklarálták. Statikus és nem statikus metódusok egyaránt lehetnek generikusak, éppen úgy ahogyan az osztály konstruktorai is lehetnek azok.

Paraméterek korlátozása (bounded type parameters = korlátozott típusparaméterek)
Két irányból történhet a típusparaméterek korlátozása, alulról vagy felülről.

Felülről: a típusparaméterek nevét (egy betűs azonosítót) az extends szó fogja követni, majd az extends szó mögött már egy létező típus nevét adjuk meg. Ez a már létező típus lehet akár egy osztálynak akár egy interfésznek is a neve, és emiatt az extends szónak általános érvényű jelentése az, hogy a típusparaméternek vagy kiterjesztenie, vagy implementálnia kell az extends szó mögött szereplő osztályt vagy interfészt.

A típusparamétert nem csak metódusoknál lehet felülről korlátozni, hanem osztályoknál is.
Ha **több korlát**tal is rendelkezik, akkor így kell: <T extends B1 & B2 & …… & Bn>
Ezek közül valamelyik osztály lesz, a többinek pedig interfésznek kell lenni.
class A { …. }
interface B { …. }
interface C { …. }

class D<T extends A & B & C> { … }
Ha többszörösen korlátozott a típusparaméterünk, és szerepel a korlátozások között osztály, akkor az osztálynak kell az elsőnek lenni.

Mi köze van az öröklődéshez a generikus típusoknak?

Nincs öröklődési viszony közöttük. pl.: Box<Number> Box<Integer> alapból a number és az integer között lenne, mivel az integer a number leszármazottja. number ← integer
Viszont mind a ketten leszármazottjai lesznek az object osztálynak.
box<Number> → Object ← Box<Integer>
Collection<String> ← List<String> ← ArrayList<String> ezek továbbra is érvényesek.

Típuskövetkeztetés (type interence)

Ez a java fordítónak egy képessége, ami azt akarja jelenteni, hogy a java fordító képes megkeresni minden egyes metódushívásnál és a hozzá tartozó deklarációnál azt a típusparamétert/paramétereket, amellyel a hívás végrehajtható, kivitelezhető. Ez a következtető algoritmus meghatározza a paramétereknek a típusát, és amennyiben lehetséges akkor a metódusok esetében a visszatérési érték típusát is. A következtető algoritmus mindig a legspecifikusabb típust keresi meg, amellyel a metódus végrehajtható.

pl.: Map<String, List<String>> myMap = new HashMap<>();

Map<String, List<String>> myMap = new HashMap(); // itt nyers típussal dolgozunk. (kihagytuk a diamond szimbólumot a végéről)

List empty list—vissza aad egy üres listát
List<String> listOne = Collections.emptyList(); az egyenlőségjel utána részből a fordítóprogram nem fogja tudni , hogy milyen típusú adatok lesznek az üres listában, ezt majd onnan fogja tudni, hogy hogyan lett deklarálva, és innen fogja tudni hogy a generikus típusa a listának a String lesz.

Helyettesítő karakterek (wildcards)

A ? mint helyettesítő karakter egy ismeretlen típust jelöl. Sok helyen előfordulhat, egy paraméternek, mezőnek, vagy lokális változónak a típusaként (ezek a gyakoribbak), néha metódusok visszatérési típusaként is. Soha nem szokott előfordulni generikus metódushívás típusparamétereként, generikus osztály példányosításakor (példányosításában), és szupertípusként sem.

Alulról korlátozott típusok:

A super lesz a kulcsszó. Megjelenési formája pl.: <? super A> itt csak olyan osztályokat veszünk figyelemben, ami az a szülőosztálya….

Az alsó és felső korlátozást egyszerre nem lehet használni a típusparaméterezésnél!

Típustörlés (type erasuer)

A generikusok implementálásakor a java fordító hajtja végre a típustörlést a következő helyzetekben:

→ a generikus típusokban helyettesíti/lecseréli az összes típusparamétert a korlátjaikra, vagy az object típusra, ha a típusparaméterek nincsenek korlátozva. Ennél fogva a lefordított byte

kód, az már csak egyszerű hétköznapi osztályokat, interfészeket és metódusokat fog tartalmazni.

→ típuskényszerítéseket szúr be a kódba, ha szükséges, a típusbiztonság biztosítása/fenntartása érdekében

→ úgynevezett áthidaló metódusokat (bridge methods) épít be a kódba a polimorfizmus biztosítása érdekében, a generikus típusok kiterjesztésekor

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.

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.

Enum típusok

Olyan típusok, amelyek mezői konstansoknak egy halmazából áll. A nevesített konstansok nem rendelkeznek saját névtérrel, 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 osztályt definiál. Törzse 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.

Lambda kifejezés
Nézzük meg, hogy a Java 8 előtt milyen lehetőségünk egy swinges JButton-ra egy ActionListenert raknunk.

```
button.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent event) {
                System.out.println("gomb lenyomva");
        }
});
```

Ha ránézünk erre a kódrészletre, láthatjuk, hogy eléggé pazarló megoldás, mivel egy névtelen interfész implementációt kell írnunk. Ha több gombunk van, több ilyet kell írnunk, ami lássuk be elég olvashatatlan kódot eredményez, akármennyire is próbálunk figyelni erre.

Most vessünk egy pillantást a Java 8 legnagyobb újítására, a lambda-kifejezésekre. Az előző példa lambda-kifejezésekkel a következőképpen néz ki:

```
button.addActionListener(event -> System.out.println("gomb lenyomva"));
```

Lássuk be, ez sokkal szebb, olvashatóbb, és egyszerűbb megoldás, még akkor is ha az előző példának a nagy részét az IDE kigenerálja nekünk. Nézzük meg, hogy mit csinál ez a kód.

Ahelyett, hogy egy interfész implementációt adnánk át az addActionListener metódusnak, semmi ilyet nem teszünk, helyette egy kódrészletet adunk át neki. A lambda-kifejezés bal oldala a metódusnak a paramétere (a paraméter neve mindegy, hogy mi). A jobb oldal pedig az a kód, amit a Java lefuttat, ha bekövetkezik az adott esemény.

A másik nagyon fontos dolog, - amit már egyébként láthattunk a Java 7 esetében - az a típus kikövetkeztetés (type inference). A példában látható, hogy a kifejezés bal oldalára nem írtam típust, ez a type inference miatt van.

Stream
Minden programozási nyelv alapfeladatai közé tartozik a külvilággal való kommunikáció, amely a legtöbb esetben az adatok olvasását jelenti egy bemeneti eszközről (pl. billentyűzet), ill. az adatok írását egy kimeneti eszközre (pl. képernyő, fájl). A Java nyelv az i/o műveleteket **adatfolyamokon** (ún. streameken) keresztül, egységesen valósítja meg. Egy dologra kell csak figyelni, hogy bemeneti vagy kimeneti csatornát kezelünk-e. A csatornák kezelése Java osztályokon keresztül valósul meg, azonban ezek nem részei a Java alap eszközkészletének, ezért importálni kell őket a java.io csomagból.

A konzol kezelésére a Java három adatfolyamot biztosít. Ezeket az adatfolyamokat nem kell megnyitni vagy bezárni, e műveletek automatikusan történnek.

A standard adatfolyamok a következők:

- standard bemenet: System.in
- standard kimenet: System.out
- standard hiba: System.err

A konzol I/O műveletek használata esetén a java.util csomagból importálnunk kell a megfelelő osztályokat.