Tartalomjegyzék

[1 Az objektum orientált programtervezés alapelvei 3](#_Toc104838554)

[1.1 Kapszula (egységbezárás) 3](#_Toc104838555)

[1.2 Adatrejtés (information hiding) 3](#_Toc104838556)

[1.3 Öröklés (inheritance) 4](#_Toc104838557)

[1.4 Polimorfizmus 4](#_Toc104838558)

[1.5 (Absztrakció) 4](#_Toc104838559)

[2 Java jellemzői, java platform, java program felépítése, fordítás, futtatás menete. Csomagok. Import. 4](#_Toc104838560)

[2.1 Jellemzői: 4](#_Toc104838561)

[2.2 Platform: 5](#_Toc104838562)

[2.3 JAVA program felépítése, fordítás 5](#_Toc104838563)

[2.4 Csomagok - package 6](#_Toc104838564)

[2.5 import 6](#_Toc104838565)

[3 Osztálydefiníció, adattag, metódustag definíció. Egységbezárás alapelvének implementálása. 7](#_Toc104838566)

[3.1 Osztálydefiníció: 7](#_Toc104838567)

[3.2 Egységbezárás alapelvének implementálása 8](#_Toc104838568)

[4 Osztálydefiníció, adattag, metódustag definíció. Információ rejtés alapelvének implementálása. 9](#_Toc104838569)

[4.1 ld. 3.1 9](#_Toc104838570)

[4.2 Információ rejtés alapelvének implementálása 9](#_Toc104838571)

[5 Hivatkozás típusú (referencia) változók fogalma, jellemzői. Elemi tipusú és referencia változók összehasonlítása. 9](#_Toc104838572)

[5.1 A típusokat két nagy csoportra oszthatjuk: 9](#_Toc104838573)

[5.2 Az érték (elemi) típuscsalád tagjai: 9](#_Toc104838574)

[5.3 A referencia típuscsalád tagjai 9](#_Toc104838575)

[5.4 Érték típus 9](#_Toc104838576)

[5.5 Referencia típusok 10](#_Toc104838577)

[6 Műveletek referencia változókkal. Referencia statikus és dinamikus típusa. Referencia konverziók. 10](#_Toc104838578)

[6.1 Objektum 10](#_Toc104838579)

[6.2 Műveletek referencia változókkal 10](#_Toc104838580)

[6.3 Referencai statikus és dinamikus típusa 11](#_Toc104838581)

[6.4 Referencia konverziók 11](#_Toc104838582)

[7 Példányosítás szintaktikája, menete. Példány élettartama. Szemétgyűjtő. A final módosító. 11](#_Toc104838583)

[7.1 Példányosítás szintatktikája 11](#_Toc104838584)

[7.2 Menete 12](#_Toc104838585)

[7.3 Objektum élettartama 12](#_Toc104838586)

[7.4 Final módosító 12](#_Toc104838587)

[8 Konstruktorok fogalma, definíciója, használatának szabályai. 12](#_Toc104838588)

[8.1 Fogalma 12](#_Toc104838589)

[8.2 Szintaktikája 12](#_Toc104838590)

[8.3 Konstruktor használatának szabályai 13](#_Toc104838591)

[9 A this fogalma, szerepe. Osztályszintű adattagok, metódusok. 13](#_Toc104838592)

[9.1 A this fogalma, szerepe 13](#_Toc104838593)

[9.2 Osztályszintű adattagok, metodusok (static) 14](#_Toc104838594)

[9.3 Osztályszintű metodusok 14](#_Toc104838595)

[10 Öröklődés fogalma, használatának előnyei, hátrányai, jellemzői. Öröklődés szintaktikája. Tagok öröklődése. 14](#_Toc104838596)

[10.1 Öröklődés fogalma 14](#_Toc104838597)

[10.2 Öröklődés előnyei 15](#_Toc104838598)

[10.3 Öröklődés nehézségei 15](#_Toc104838599)

[10.4 Öröklődés jellemzői 15](#_Toc104838600)

[10.5 Öröklődés szintatktikája 16](#_Toc104838601)

[11 Polimorfizmus megjelenési formái. Függvény/operátor túlterhelés (overloading), metódusfelüldefiniálás (overriding). 16](#_Toc104838602)

[11.1 Polimorfizmus fogalma 16](#_Toc104838603)

[11.2 overloading – függvény/operátor túlterhelés 16](#_Toc104838604)

[11.3 overriding – metodusok felülírása 16](#_Toc104838605)

[12 Absztrakt metódusok, osztályok. Interface fogalma szerepe. Interface definíció, implementálás. 16](#_Toc104838606)

[12.1 Absztrakt metodus 16](#_Toc104838607)

[12.2 Absztrakt osztály 17](#_Toc104838608)

[12.3 Interface fogalma 17](#_Toc104838609)

[12.4 Interface definíció 17](#_Toc104838610)

[12.5 Interface implementálása 17](#_Toc104838611)

[12.6 Abstract osztály vs interfész 18](#_Toc104838612)

[13 Speciális szintaktikával definiálható osztályok: tömb, enum. Generikus osztályok. 18](#_Toc104838613)

[13.1 Tömb 18](#_Toc104838614)

[13.2 enum 18](#_Toc104838615)

[13.3 Generikus osztályok 19](#_Toc104838616)

[14 Kivételkezelés 19](#_Toc104838617)

[15 A lang csomag osztályai: Object, Comparable, System, String, StringBuilder, csomagolóosztályok. 21](#_Toc104838618)

[15.1 Object osztály 21](#_Toc104838619)

[15.2 String 21](#_Toc104838620)

[15.3 StringBuilder/Buffer (egyszál/több szál) 21](#_Toc104838621)

[15.4 Math 21](#_Toc104838622)

[15.5 Comparable osztály 22](#_Toc104838623)

[15.6 System és egyéb 22](#_Toc104838624)

[15.7 Csomagoló (wrapper) osztály 22](#_Toc104838625)

[16 Osztályok közötti kapcsolatok. 22](#_Toc104838626)

# Az objektum orientált programtervezés alapelvei

## Kapszula (egységbezárás)

* 1. az adatok és a hozzájuk tartozó eljárásokat egyetlen egységben kezeljük (objektum-osztály);
  2. Az osztály mezői tárolják az információkat;
  3. A metódusok kommunikálnak a külvilággal; -> Metódus: Olyan eljárás, mely része valamely objektumosztálynak, így az adott osztály attribútumaival végez valamilyen műveletet.
  4. Az osztály változóit csak a metódusokon keresztül változtathatjuk meg;
  5. A feladatok elvégzésének „hogyan”-ja az objektum belügye. Az objektum belseje sérthetetlen. Az objektummal csak az interfészen keresztül lehet kommunikálni.

### Osztály:

* 1. Egy felhasználó által készített típus, mely összetett adatszerkezet - elvileg tartalmazza az adott objektum adatait, és az azokat kezelő eljárásokat. Az objektum információt tárol, és kérésre feladatokat hajt végre. Az objektum felelős feladatainak korrekt elvégzéséért.
  2. A modellezés során nyert absztrakt adattípus implementációja: Egy programozó által definiálható típus, amely tartalmazhat:
     1. adatokat az objektum struktúrájának modellezésére;
     2. metódusokat az objektum viselkedésének modellezésére;
  3. Az osztály saját névteret képez, azaz több osztályban is lehet ugyanolyan nevű adat vagy művelet.
  4. Valamint az osztályon belül az adatok és műveletek korlátlanul elérik egymást.

**Objektumorientált program:** Egy OO program egymással kommunikáló objektumok összessége, melyben minden objektumnak megvan a jól meghatározott feladata.

### Objektum:

* 1. Egy változó, melynek típusa valamely objektumosztály, vagyis az osztály egy példánya.
  2. Egy osztály általában többször is példányosítható. A példányok ugyanazzal a viselkedésfajtákkal rendelkeznek, ugyanaz a struktúrája, de minden példány más állapottal rendelkezhet (azaz adatai más-más értékekkel rendelkezhetnek).

### Absztrakció:

Absztrakciós folyamat, amelyben a valós világban létező jelenséget (megoldandó problémát) valamilyen programozási eszköz absztrakciós szintjén képezünk le. A leglényegesebb tulajdonságok kiemelése és általánosítása.

## Adatrejtés (information hiding)

Az osztály műveleteinek implementációja az osztályon kívül nem látható. Az osztály adatai és műveletei kívülről csak szabályozottan érhetők el.

### Szabályok

* 1. az osztályok adatai a külvilág számára közvetlenül nem elérhetők (csak a metódusain keresztül);
     1. A programozó határozza meg, hogy írás, olvasás legyen-e a hozzáférés.
     2. A hozzáférések és az adatábrázolás egymástól független.
     3. Az adatok beállításakor lehet hozzákötni ellenőrzést.
  2. A metódusok közül csak azok érhetők el a többi objektum számára, amelyek az objektum kommunikációs felületéhez tartoznak.

## Öröklés (inheritance)

* 1. az objektum-osztályok továbbfejlesztésének lehetősége. Ennek során a származtatott osztály örökli ősétől azok attribútumait és metódusait, de ezeket bizonyos szabályok mellett újakkal egészítheti ki, és meg is változtathatja;
  2. Az eredeti osztályt ősosztálynak nevezzük (szülő);
  3. Az új, továbbfejlesztett osztályt származtatott osztálynak (gyerek);
  4. Egy ősből több leszármaztatott osztályt is készíthetünk;
  5. Egy származtatott osztálynak
     1. legfeljebb egy szülője lehet (pl.: Pascal, Java, C#) – öröklődési fa
     2. több szülője is lehet (pl.: C++) -- öröklődési gráf
  6. Metódusok törzsét megváltoztathatjuk;
  7. Mezők neveit, típusait általában nem változtathatjuk meg;
  8. Új mezőkkel és metódusokkal egészíthetjük ki az osztályt;

## Polimorfizmus

* 1. Ugyanarra a kérelemre a különböző objektumok különbözőképpen reagálnak.

Ha van egy "Kör" objektumom, ami egyben "Alakzat" is, akkor az fogadhatja azokat az üzeneteket, ami egy "Alakzatnak" érkezhet, de reagálhat rá, mint "Kör" is. A gyakorlatban ez lesz az objektum orientált programok egyik legfontosabb tulajdonsága, amit polimorfizmusnak nevezünk.

* 1. A származtatás során az ős osztályok metódusai képesek legyenek az új átdefiniált metódusok használatára újraírás nélkül is;
  2. Operátorok esetében az összeadás másként működik egész számokra mint stringekre.

## (Absztrakció)

Egyes források az OOP alapelvekhez sorolják az absztrakciót, amely a probléma megfelelő osztályokkal való modellezésével egyszerűsíti az összetett valóságot. Ez az a folyamat, mely során a valós objektumokat leképezzük program objektumokra (a lényeges jellemzők kiemelése és általánosítása). Lásd osztály és objektum alapfogalmak.

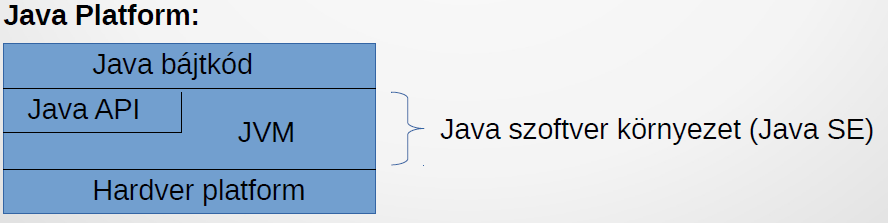
# Java jellemzői, java platform, java program felépítése, fordítás, futtatás menete. Csomagok. Import.

## Jellemzői:

* 1. C++ alapú szintaktika, de:
     1. mutatók helyett referencia változók (referencia kvázi alias egy már létező változóhoz; a pointer egy olyan változó, ami egy memória címet tartalmaz, ami egy másik változóra mutat)
     2. nincs többszörös öröklés;
     3. Java-ban nincs operátor overloading; (ugyanaz az operator név vagy szimbólum más-más műveletet eredményezne;)
  2. Teljesen objektum orientált;
  3. OOP alapelveket megvalósítja (4db. ld. 1. tétel)
  4. nincsen osztályon kívüli deklaráció, utasítás;
  5. Gazdag osztálykönyvtár választék;
  6. **Erősen típusos nyelv** (minden kifejezés és részkifejezés típusa fordítási időben ismert; előre el kell döntenünk és deklarálnunk kell, hogy a változókban milyen adatokat szeretnénk tárolni; az erősen tipizált nyelv szigorúbb gépelési szabályokkal rendelkezik a fordítás idején, ami azt jelenti, hogy a hibák és a kivételek nagyobb valószínűséggel történnek a fordítás során; addig a gyengén tipizált nyelv lazább gépelési szabályokkal rendelkezik, és kiszámíthatatlan vagy akár hibás eredményeket hozhat;)
  7. Minden Java fejlesztőeszköz ingyenes és nyílt forrású;
  8. Platformfüggetlen (WORA: Write Once Run Anywhere)
  9. Támogatja
     1. a többszálon futó programok fejlesztését
     2. az elosztott alkalmazások fejlesztését
  10. a webes alkalmazások fejlesztését

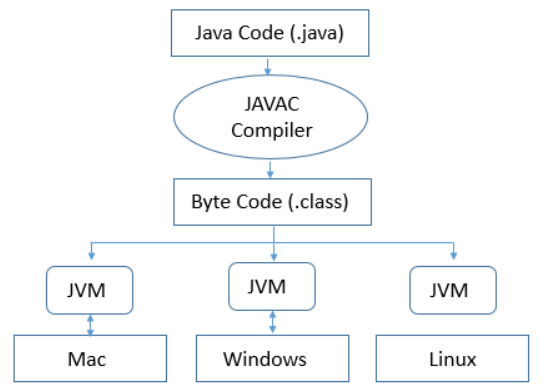
## Platform:

Java Program ->Forráskód -> Compiler -> Byte kód -> Fordító -> Gépi kód -> futó program



* 1. A platform maga egy HW és SW környezet, amelyen egy program fut; (a legtöbb platform leírható a OS-essel (MS, Linux, Mac OS) és az alatta lévő HW elemekkel.
  2. a JAVA SE (SW környezet) 2 elemből áll:
     1. JVM – Java Virtual Machine -> JAVA programozási nyelvhez készített virtuális gép. Alapvető feladata egy adott JAVA bájtkód (tehát egyetlen) futtatása platformfüggetlen módon.
     2. JAVA API (Application Programming Interface) gyűjtemény -> ez könyvtárakba csomagolt hasznos kész komponensek gyűjteménye;
  3. A JAVA platform független, azaz ugyanaz a JAVA program futatható más-más HW-SW környezetben -> Ezáltal a JAVA program kicsit lassabb lehet, bár ez a hátrány a VM technológia által jócskán csökkenthető;

## JAVA program felépítése, fordítás



* 1. **1. lépés a fordítás:** forráskód (fájlname.java, szöveges fájl – pl: Eclipse-ben létrehozva) -> fordító -> bájtkód (classname.class fájl, platform független, nem futtatható NEM gépikód)
  2. **2.** **lépés az értelmezés**: bájtkód (classname.class) -> interpreter (JVM része; soronként értelmezi és végrehajtja a bájtkódot) -> gépikód (a számítógép processzora számára értelmezhető egyetlen „nyelv”; minden programozási nyelvet gépi kóddá kell változtatni, hogy az végrehajtható legyen; pl 16-os számrendszerű gépikódra: B8 D4 07 00 00 90 48 75 FC C3; (tízesben vagy 2-esben is felírható lenne)

### JDK részei

* 1. javac: java fordító
  2. java: java értelmező
  3. jdb: Java debugger
  4. Javadoc: automatikus dokumentum generáló
  5. jar: Java archiváló

### JRE (Jva Runtime Environment)

* 1. JVM – Java Virtual Machine
  2. platform függő osztályok, könyvtárak; (API-k is)

### JVM

* 1. absztrakt számítógép
  2. ez futtatja a Java programokat
  3. futtatás előtt ellenőrzi a bájtkódot;

## Csomagok - package

* 1. Az osztályokat csomagokba soroljuk. -> az osztály definíció előtt package kulcsszóval jelöljük a csomagot!
  2. A csomagok révén az osztályok hozzáférése korlátozható.
  3. A csomagok lehetőséget biztosítanak a tagok csomagszintű hozzáféréséhez.
  4. Névteret alkotnak: Egy osztály neve csak a saját csomagjában kell, hogy egyedi legyen. Két külön csomagban lehet ugyanolyan nevű osztály, ugyanis az osztály teljes neve valójában: Csomagnév.Osztálynév
  5. Elnevezésük hierarchikus;
  6. Csomagneveknek egyedinek kell lennie;

## import

* 1. A csomag megadása után, de még az osztály definíció előtt, lehet import utasításokat megadni „import osztályteljesnév” formában. Hatására az osztálydefiníción belül az osztályra elég csak az osztálynévvel hivatkozni nem kell a teljes nevét kiírni.
  2. Az import nem szúr be kódot, csak ismertté teszi az osztálynevet.
  3. Ha két import csomagban is van azonos nevű osztály, akkor hivatkozáskor a teljes nevet ki kell írni.
  4. ”\*” karakter segítségével több osztály is importálható.
  5. Java **alapértelmezett csomag a java.lang** -> ezt akkor is tartalmazza a program, ha nem írjuk ki. Pl.: a String, vagy a Math osztályok ebben vannak, elég is csak az osztálynevet kiírni.
  6. Ha használni akarunk egy osztályt, akkor azt importálni szükséges.
  7. java.lang csomag: ezt nem kell importálni, a Java automatikusan megteszi;
  8. Math osztály: minden metódusa statikus, ezért az osztály nevével minősítve hívható; ha importáljuk, akkor meg minősítés nélkül;

# Osztálydefiníció, adattag, metódustag definíció. Egységbezárás alapelvének implementálása.

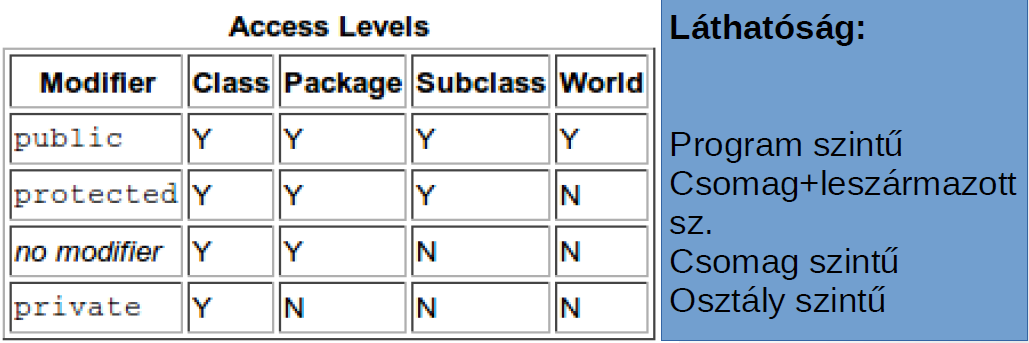
## Osztálydefiníció:

* 1. Az OOP program alapegysége;
  2. A programozó által definiált típus;
  3. Saját névteret képez;
  4. A forrásfájl neve kötelezően az osztály neve; egy fájl egy osztály;

Nagy betűvel kezdődjön; case sensitive; számmal nem kezdődhet;

* 1. Osztály módosítók
     1. **hozzáférés**: public, protected, „üres”, private,
     2. **abstract**: nem példányosítható osztály, amit csak ősosztályként használunk;
     3. **final**: végleges, nem lehet leszármazott osztálya;
     4. **static**: osztályszintű adattag (az osztályban meghatározott érték nem változik)
  2. A programok belépési pontja egy futtaható osztály; (main method)
     1. public static void main(String[] args)
        1. public: hozzáférési kategória (public = mindenki);

(public -> mindenki, protect -> no world, <üres> -> csak package, private -> csak osztály)



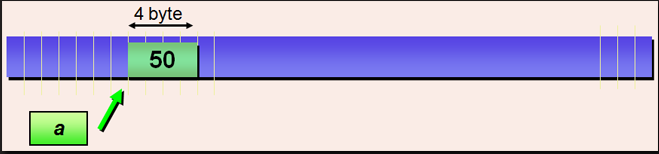
* + - 1. static: osztály szintű adattag vagy metódus
* minden példányban azonos lesz (osztott változó);
* akkor is végrehajtható, ha az osztálynak nincsenek példányai;
* az osztályszintű adattagok betöltődéskor elfoglalják helyüket a memóriában;
  + - 1. void: visszatérési érték (üres);
      2. String[]: sztringek tömbje, a program indító paraméterei;
  1. Az osztálytagok

definiálásuk sorrendje tetszőleges, de a szokásos:

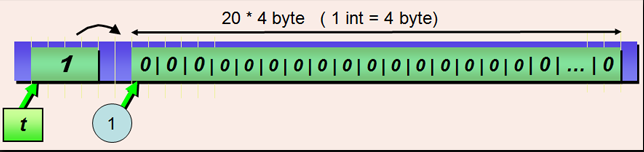
* + 1. **Adattag definíció**
       1. a lokális vátozókkal szemben külön sorban szokás őket definiálni;
       2. a lokális változókkal szemben az adattagok akkor is inicializálódnak, ha a programozó ezt külön nem teszi meg;
       3. **adattag hozzáférés módositó** -> **public** String name; **private** static int x;
       4. **az adattag típus** -> tetszőleges **elemi** vagy **referencia**;

elemi: public **int** salary; referencia: public **String** name;

ELEMI



REFERENCIA



* + - 1. az **adattag neve**: public Sring **name**; provate int x
    1. **Inicializáló blokkok** (osztályszintű, példányszintű);

kezdeti értékek megadása történik;

konvenció, hogy az összes adattagot inicializáljuk;

több is lehet egy osztályban;

a konstruktor előtt lefut;

* + 1. **Konstruktorok**:

**módosító azonosító (paraméterlista) {utasítások}**

**módosító**: csak hozzáférés módosítója lehet: public, protected, üres, (private???)

**azonosító**: kötelezően az osztálynév;

**paraméterek**: szintaktikája ugyanaz mint a metódusoké;

**utasítások**: return kivételével bármi;

az osztály kezdeti értékeinek beállítására szolgál az osztály példányosításakor; (ugyan nem szokás, de lehet más utasítás is)

HA NEM CSINÁLUNK KONSTRUKTORT, AKKOR KELETKEZIK EGY DEFAULT (üres)!

HA CSINÁLNUNK, AKKOR NINCS DEFAULT!

lehet több konstruktor is;

nem tekintjük metódusnak, mert nincs visszatérési értéke és nem lehet meghívni;

* + 1. **Metódusok**;

**módosítók típus név(paraméterlista)**

metódus hozzáférés **módosítók**:

public, protected, üres, private

final -> végleges, nem írható felül a leszármazott osztályokban

abstract -> kötelező felülírni a leszármazott osztályban

static -> osztályszintű metódus -> példány nélkül is meghívható;

**típus**: tetszőleges elemi vagy referencia típus ÉS void!

**név**: kisbetűvel szabadon;

**paraméterlista**

szabadon;

paraméterátadás érték szerinti;

* + 1. (Tagosztályok);

## Egységbezárás alapelvének implementálása

* 1. Egy nyelvi elemen belül találhatók az adatok és az adatokon műveleteket végző metódusok (ez a nyelvi elem az osztály). Az adatok CSAK ezeken a metódusokon keresztül érhetők el!
  2. Az osztályon belül az elemek (csak) a nevükkel hivatkoznak egymásra.
  3. Az osztályon belül a metódusok korlátlanul használhatják az adattagokat és a többi metódust.

# Osztálydefiníció, adattag, metódustag definíció. Információ rejtés alapelvének implementálása.

## ld. 3.1

## Információ rejtés alapelvének implementálása

### Eszköz

Az információrejtés alapelv megvalósításának eszköze a **hozzáférési módosítók** használata.

### Konvenciók

* 1. Egy osztály csak akkor legyen public, ha általános használatra szántuk.
  2. Az adattagok private (esetleg protected) módosítót kapjanak.
  3. A metódusok közül csak az legyen public, amelyen engedélyezzük az osztály adattagjainak elérését (módosítását) a többi osztály számára.

# Hivatkozás típusú (referencia) változók fogalma, jellemzői. Elemi tipusú és referencia változók összehasonlítása.

## A típusokat két nagy csoportra oszthatjuk:

* 1. referencia (reference) típus család,
  2. érték (value) típus család.

## Az érték (elemi) típuscsalád tagjai:

* 1. az összes egész szám típus (long, int, short, byte, ...)
  2. az összes tört szám típus (double, float)
  3. karakter típus (char)
  4. logikai típus (bool)

## A referencia típuscsalád tagjai

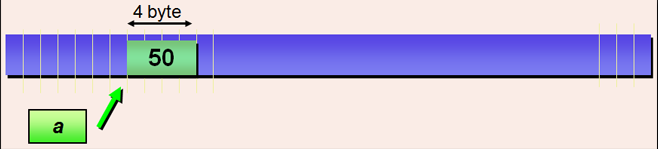
* 1. String
  2. tömbök (vektorok)
  3. listák
  4. objektumosztályok

## Érték típus

Egy érték típusú változóról mindig pontosan lehet tudni a memóriaigényét. Hiszen:

int a; -> 4 byte-on tárolható; double b; -> 8byte-on tárolható;

A memóriában az „a” változónak 4 byte foglalódik le, melyen majdan az értéket tárolni fogjuk. A program szövegében az a változó lényegében ezen **memória címre mutat, ahol maga az érték tárolódik**. Amíg a változó meg nem szűnik, addig **végig a memória ugyanazon a pontja** tartozik hozzá. Ha új értéket adunk neki, akkor is itt tárolódik.



## Referencia típusok

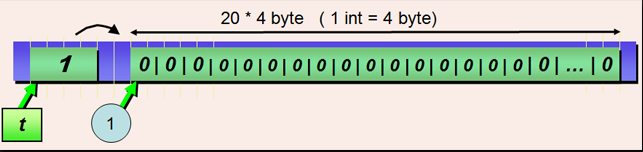
A referencia típuscsalád valamely típusába tartozó változók esetén a helyzet sokkal bonyolultabb, ugyanis nem tudjuk egy ilyen típusú változóról eldönteni, hogy mennyi helyet foglal.

Ugyanis pl. az int[] t = new int[20] esetében mondhatjuk, hogy 20 x 4byte, de mi a helyzet, ha csak létrehozzuk, de a méretét csak később adjuk meg? Vagy egy osztály elemeit szeretnénk egy tömbbe pakolni, vagy String értékeket, amelyeknek alapból nem tudjuk a hosszát?

A fordítóprogram kettéválasztja a referencia típusú változók kezelését a hozzájuk tartozó tényleges adattárolástól. Két memóriaterület tartozik ténylegesen egy referencia típusú változóhoz:

* 1. elsődleges memóriaterület, mindig 4 byte, tartalma egy memóriacím;
  2. másodlagos memóriaterület, mérete változó, itt történik a tényleges adattárolás;

A fenti példánál maradva az elsődleges memóriaterület az int[] t; létrehozásokor foglalódik 4 byte-on, aminek ekkor a tartalma üres! Majd a t = new int[20]; megadásakor foglalódik a második memóriaterület, és ennek a memóriaterületnek a címe be is íródik az első címterületbe, míg a konkrét adatok a második memóriaterületre.



A referenciákon értelmezett további műveletek nem a hivatkozott objektumra, hanem magára a referenciára vonatkozik. Tehát ha String a = „madár”; és String b = „madár” attól még a és b nem lesz egyenló, mert más a referenciájuk!

# Műveletek referencia változókkal. Referencia statikus és dinamikus típusa. Referencia konverziók.

## Objektum

Az osztály példányait a new Osztálynév() kifejezéssel hozhatjuk létre. Mivel ez referencia változó, ezért csak a 4byte-os referencia jön létre, ami kezdetben null, majd az objektum létrejöttével annak memória címét tartalmazza.

**Az objektum csak futás közben, dinamikusan jön létre.**

## Műveletek referencia változókkal

### Deklaráció

Circle c1, c2;

### Értékadás

c1 = new Circle(); c2 = c1; Circle c1 = new Circle(”alma”);

Circle c2 = new Circle(”alma”);

### Egyenlőség vizsgálat

boolean equal = (c1 == c2); -> IGAZ boolean equal = (c1 == c2); -> NEM IGAZ

|  |  |  |  |
| --- | --- | --- | --- |
| Művelet | C1 | C2 | Egyenlőek-e |
| deklarálás | String c1; | Sring c2; |  |
| értékadás | c1 = new String(„alma”); | c2 = c1 | IGEN |
|  |  |  |  |
| deklarálás és értékadás | c1 = new String(”alma”); | c2 = new String(”alma”); | NEM |

## Referencai statikus és dinamikus típusa

Ha pl. a Vehicle osztály egy kiterjesztése/leszármazott osztálya a Car.

Példányosítással létrehozunk egy Trabi-t a Car osztályban: Car trabi = new Car(4, 800, 4);

Ha a Vehicle osztályban volt egy wheelLoad metódus, akkor ez a Car osztályban is elérhető! Akár azt is megtehetem, hogy Vehicle típussal hozok létre Car példányt: Vehicle skoda = new Car(…);

Viszont 2 Car objektum nem tudja egymás metódusait meghívni, vagy a Vehicle példány sem a Car példányét.

### Referencia típus változói

* 1. A deklaráláskor megadott típus **(statikus** típus): ez határozza meg milyen műveleteket végezhetünk a referenciával hivatkozott objektummal (azaz **milyen metódusokat hívhatunk meg**).
  2. A hivatkozott objektum tényleges típusa (**dinamikus** típus): **amelyik osztályból példányosítom**. Ez csak olyan típus lehet, amely rendelkezik ugyanazokkal a műveletekkel, adatokkal, mint a statikus típus. Ez csak a statikus típus leszármazottai esetén garantált.

PÉLDA: „Vehicle skoda = new Car(4, 1300, 5);” példányosítás során a skoda referencia változó **STATIKUS** típusa Vehicle, míg **DINAMIKUS** típusa Car;

## Referencia konverziók

Elemi típusok között gyakorlatilag korlátok nélkül használható a bővebb típusok irányába (kivéve boolean: logikai <-> numerikus NINCS).

Osztályok között viszont csak egy öröklődési fán belül engedélyezett.

# Példányosítás szintaktikája, menete. Példány élettartama. Szemétgyűjtő. A final módosító.

## Példányosítás szintatktikája

Circle c = new Circle(10); -> „c” egy 10 egység sugarú kör példánya;

## Menete

A példányosítás nem más, mint egy osztályból egy teljes referencia változó, vagyis egy objektum létrehozása (referencia változó és tényleges adatok). Ennek pontos lépései:

* 1. Létrejön az objektum (lefoglalódik számára a hely a memóriában a kupac területen).
  2. Inicializálódnak az adattagok.
     1. Osztálydefinícióban megadott adattag-inicializálás
     2. Automatikus inicializálás (ha a definícióban nem volt)
     3. Java inicializáló blokkban megadott inicializálás
  3. Lefut a megfelelő konstruktor
  4. Visszatér az objektumra mutató referenciával.

## Objektum élettartama

* 1. **Objektum a példányosításkor jön létre**;
  2. Megszűnéskor: a Garbage Collector (automatikus szemétgyűjtő; a JVM indítja külön szálon) azokat az objektumokat megszűnteti, amelyekre egyetlen referencia sem hivatkozik;
  3. Megszűnés:
     1. lokális változó, ha kilépünk a hatáskörből;
     2. megváltozik a változó értéke;
     3. <null> értéket kap;

## Final módosító

* 1. Lokális változó definícióban: egyszer vehet fel értéket, ami utána nem is változtatható.
  2. Osztály adattag definícióban: egyszer vehet fel értéket, ezt utána megváltoztatni nem lehet.
  3. Paraméterlistában:
     1. Metódus definícióban: az ilyen típusút nem lehet felüldefiniálni a leszármazottban.
     2. Osztálydefinícióban: a final típusú osztálydefinícióból nem lehet leszármaztatni. (pl.: String)
  4. Ha referencia változóra vonatkozik a final módosító, akkor nem cserélhetem le a helyét egy másikra, de az értékét felülírhatom!

# Konstruktorok fogalma, definíciója, használatának szabályai.

## Fogalma

* 1. Konstruktor: speciális metódus, amelyet nem lehet explicite meghívni, hanem automatikusan hívódik meg az osztály példányosításakor.
  2. Szerepe az objektum induló állapotának beállítása (pl. adattagok inicializálása).
  3. Nem tekintjük metódusnak, mert nincs visszatérési típusa és nem lehet meghívni.

## Szintaktikája

módosító azonosító(paraméterlista) {utasítások}

* 1. módosító: csak hozzáférési módosítója lehet;
  2. azonosító: kötelezően az osztály neve;
  3. Visszatérési értéke nincs;
  4. Paraméterlista: szintaktikája ugyanaz, mint a metódusnál; a paraméterek példányosításkor kapnak értéket;
  5. utasítás: return kivételével tetszőleges utasítás használható;

## Konstruktor használatának szabályai

* 1. ha egy osztálynem éri el egy másik osztály konstruktorát, akkor nem példányosíthatja azt; (private konstruktor csak az osztályon belül példányosítható;)
  2. Konstruktor nem öröklődik, tehát nem is lehet felülírni;
  3. Egy osztálynak akkor is van konstruktora, ha nem csinálunk, mert a „default” konstruktor akkor is létrejön;
  4. Ha készítünk egyetlen egy konstruktort is, akkor a default konstruktor nem jön létre;
  5. Egy osztálynak lehet több konstruktora is;
  6. Egy konstruktorból meghívható egy másik konstruktor, amikor is a this mögé kell tenni a másik konstruktor paramétereit (szerepe: ha már ott elkészült egy jó inicializálás, akkor azt itt felhasználhatjuk);
  7. a konstruktor csak az előírt paraméterekkel tud példányosítani; (pl.: String helyett nem lehet üres)
  8. Konstruktorban tetszőleges utasítások lehetnek, de inicilizálásra szokás használni;

# A this fogalma, szerepe. Osztályszintű adattagok, metódusok.

## A this fogalma, szerepe

Először is tisztázzuk: 2 -féle this-el találkozhatunk

### this a Konstruktorban

”public Circle(double r) {this(0,0,r);}” → ez egy olyan konstruktor, amely csak a 3. paramétert kéri, az első 2 paramétert abból a konstruktorból veszi, amelyben csak pont az első 2 van megadva.

### this a metodusokban (pszeudo)

Metódusok hívásakor a példányszintű metódusnak van még egy paramétere, amely a paraméterlistában nem látható. Ez annak az objektumnak tartalmazza a referenciáját, amelyre a hívás szólt. Tehát c1.shift(2,3) esetén átadódik maga „c1” is.

Erre a láthatatlan paraméterre a metóduson belül a ”this” kulcsszóval hivatkozhatunk.

A metóduson belül minden olyan adattag vagy metódus hivatkozás, amely előtt nincs minősítés az valójában a this-el minősítődik.

A kör eltolását megvalósító metódus definíciója valójában (függetlenül attól, hogy explicite kiírom-e a this-t):

public void shift(int dx, int dy) {

this.x += dx;

this.y += dy;

}

Azt az objektumot, amelyre a this hivatkozik, aktuális objektumnak/példánynak nevezzük.

De lehet ilyen is

boolean greaterThan(Circle other) {

return this.radius > other.radius;

}

Ha a metódusban egy lokális változó neve megegyezik az osztályon (ill. hatáskörön) belüli valamely adattag nevével, akkor kötelező kiírni a this-t az adattag elé, mert a lokális változó elfedi az adattagot.

## Osztályszintű adattagok, metodusok (static)

* 1. **Egy osztály tag** (adattag, metódustag, stb.) lehet **osztályszintű vagy példányszintű**.
  2. Az **osztályszintű** tagok definíciójában szerepel a **static** módosító.
  3. **Példányszintű tag a példányhoz kötődik, osztályszintű tag az osztályhoz**.
  4. Javaban **a konstruktor mindig példányszintű**.
  5. Az osztály minden példánya saját készlettel rendelkezik a példányszintű adattagokból és egy közös készlet van az osztályszintű adattagokból.
  6. A példányszintű adattagoknak példányosításkor foglalódik hely, minden példány esetén külön. Az osztályszintű adattagoknak az osztály első hivatkozásakor (betöltődésekor) foglalódik hely.
  7. Circle c1 = new Circle(5);

Circle c2 = new Circle(10);

Ha radius osztályszintű (static) adattag, akkor c1.radius és c2.radius ugyanarra a memóriaterületre hivatkozik. Ha radius példányszintű adattag, akkor a két hivatkozás nem ugyanarra a memóriaterületre vonatkozik.

* 1. Az **osztályszintű adattag** olyan tulajdonságok modellezésére szolgál, amely **minden példányra közös**.
  2. Osztályszintű adattagra hivatkozni az osztály nevével minősítve kell.

## Osztályszintű metodusok

* 1. A metódus definíciók a példányok számára közösek.
  2. A példányszintű metódus megkapja a +1 láthatatlan paramétert a this-t, az osztályszintű metódus pedig nem.
  3. Az osztályszintű metódusokra hivatkozni az osztálynévvel minősítve kell.
  4. Az osztályszintű metódus tagok léteznek és használhatók már azelőtt is, hogy példányt létrehoznánk az osztályból.
  5. A main metódus osztályszintű, így a JVM hívhatja anélkül, hogy példányosítani kellene az osztályt.
  6. A **Math osztály minden metódusa** **static**. Ezért az osztály nevével minősítve hívhatók (pl. Math.sqrt(25)), vagy import static java.lang.Math.\*; után minősítés nélkül is.

# Öröklődés fogalma, használatának előnyei, hátrányai, jellemzői. Öröklődés szintaktikája. Tagok öröklődése.

## Öröklődés fogalma

Egy osztály létrehozható úgy is, hogy egy másik osztály leszármazottja. Ilyenkor az osztály:

* 1. örökli a szülője adattagjait, metódusait (tehát már nem kell ezeket definiálni), DE az ősosztály elemeinek az elérése a leszármazott osztályból nem feltétlenül garantált (adatrejtés láthatósági módosítókkal)
  2. lehet új adattagokat és metódusokat definiálni,
  3. lehet az örökölt metódusokat módosítani (felüldefiniálni, overriding), azaz a viselkedés a leszármazottban is létezik, csak másképpen.

## Öröklődés előnyei

* 1. Kód újrafelhasználás;
  2. A leszármazott osztályok definíciója egyszerűbb (csak a különbözőségeket tartalmazza, az öröklött adattagok és metódusok definícióit nem);
  3. A közös részek módosítása egyszerűbb (csak egy helyen, a szülőben szerepelnek);
  4. Egy leszármazott mindig helyettesítheti az őst (minden adattaggal és metódussal rendelkezik, amivel az ős);

## Öröklődés nehézségei

* 1. Alapos tervezést igényel: egy osztály tagjait úgy kell megírni, hogy ha később leszármaztatjuk, ne legyen probléma (ne kelljen újratervezni)
  2. Ha nem akarom, hogy az osztályból lehessen leszármaztatni, akkor ezt explicite meg kell adnom a definícióban (final)

## Öröklődés jellemzői

* 1. Egyszeres öröklés: egy osztálynak csak egy közvetlen őse lehet.
  2. Egy osztályból több osztályt is le lehet származtatni.
  3. Egy leszármazott osztály lehet egy másik osztály őse (ha nem végleges / lezárt).
  4. Egy osztály nem lehet saját maga őse.
  5. final módosítójú osztálynak nem lehet leszármazottja.
  6. Terminológia
     1. ősosztály
     2. leszármazott osztály
  7. **A konstruktor nem öröklődik**;
  8. Java-ban konstruktor első utasítása lehet:
     1. super(params): ősosztály konstruktorának hívása
     2. Ha az első utasítás nem this() vagy super() akkor első utasításként automatikusan beszúrásra kerül egy super() paraméter nélküli hívás.
  9. **Az inicializáló blokkok NEM öröklődnek.**
  10. **Minden adattagot örököl a leszármazott osztály**
  11. A leszármazott osztály új adattagokat is definiálhat.
  12. Ha a leszármazott osztályban ugyanolyan nevű adattagot definiálunk, mint egy örökölt tag, akkor elfedjük az örökölt tagot (**hiding**). Az örökölt, de elrejtett tagra a közvetlen leszármazottban még tudunk hivatkozni a super.tagnév-vel (ha elérhető), de a leszármazott osztály további leszármazottaiban már nem.
  13. **A leszármazott osztály örökli az ősosztály minden metódusát.**
  14. A leszármazott osztályban definiálhatunk egy örökölt metódussal azonos nevű, de eltérő paraméter szignatúrájú metódust (**overloading**, metódus túlterhelés).
  15. A leszármazott osztály felüldefiniálhat egy örökölt példányszintű metódust (**overriding**). Ekkor a két metódus neve és paraméter szignatúrája is megegyezik.
  16. A leggyakrabban használt, minden osztályban felüldefiniált metódus:

public String toString()

* 1. Ha a felüldefiniált metódust az @Override annotációval jelöljük, a fordító ellenőrzi a felüldefiniálás szintaktikai helyességét. Megadása opcionális. (Véletlenül más nevet írok, mint az ősosztályban: tostring -> toString -> HIBA más a bemeneti paraméter szignatúra -> HIBA)
  2. A leszármazott osztály elrejthet egy osztályszintű (static) metódust, ha az örökölt osztályszintű metódussal azonos nevű, paraméter szignatúrájú és visszatérési típusú metódust definiálunk (statikus polimorfizmus).
  3. Az elrejtett osztályszintű metódusra a leszármazottban tudunk hivatkozni (ha elérhető) az osztálynévvel minősítve.

## Öröklődés szintatktikája

class Osztaly extends Ososztaly {...}

* 1. Ha nem adunk meg ősosztályt (nincs extends kulcsszó), akkor automatikusan a **java.lang.Object** osztály az ős.
  2. Az **Object minden osztály őse**, tagjait minden osztály örökli.
     1. Adattagja nincs
     2. Egyetlen konstruktora paraméter nélküli
     3. Metódusai öröklődnek

# Polimorfizmus megjelenési formái. Függvény/operátor túlterhelés (overloading), metódusfelüldefiniálás (overriding).

## Polimorfizmus fogalma

Polimorfizmus: többalakúság. Bizonyos viselkedések működése függ a környezettől, ahol alkalmazzuk.

## overloading – függvény/operátor túlterhelés

### Metodusok túlterhelése

**Létezhet több azonos nevű, különböző szignatúrájú metódus egy osztályban**. A metódushívás aktuális paraméterei határozzák meg, hogy melyik metódus fog meghívódni. Ez már fordítási időben eldől (**statikus, fordításidejű kötés**).

### Operátorok kiterjesztése

Az operátorok jelentését ki lehet terjeszteni új típusokra. Pl. C++-ban és C#-ban. Javaban nincs, de a + operátor használható String típusú adatok összefűzésére is. De ez nem a programozó által definiált operátor kiterjesztés!

## overriding – metodusok felülírása

Az **öröklés** során bizonyos viselkedési formákat (metódusokat) **a** **származtatott osztályban** új tartalommal valósítunk meg (**de a metódus neve és szignatúrája megegyezik a szülő osztály**ban definiált **metóduséval**). Futási időben dől el, hogy melyik metódus kerül meghívásra (**dinamikus, futásidejű kötés**).

Pl.: síkidomok osztály kerület/terület metódusait az egyes származtatott osztályokban felüldefiniálom.

# Absztrakt metódusok, osztályok. Interface fogalma szerepe. Interface definíció, implementálás.

## Absztrakt metodus

Gyakran előfordul a tervezés során, hogy egy osztály szintjén tudjuk, hogy valamilyen metódus szükséges lesz a leszármazottakban, de még nem tudjuk megadni a definícióját.

* 1. Ezek a metódusok törzs nélküli metódusok (csak deklaráció).
  2. Az ilyen metódust az abstract minősítővel kell ellátni.
  3. Ha az osztály tartalmaz absztrakt metódust, az osztályt is az abstract minősítővel kell ellátni.
  4. Rögzít egy tervezési döntést (szükséges metodusok halmaza)
  5. Kényszeríti a leszármazott osztály(ok) programozóját meghatározott metódusok definiálására.

## Absztrakt osztály

* 1. Egy **absztrakt osztály arra szolgál, hogy ősosztálya legyen további osztályoknak**. A leszármazott osztály(ok) feladata az absztrakt metódusok felüldefiniálása.
  2. **Absztrakt osztályt nem lehet példányosítani**.
  3. **Absztrakt osztálynak lehet konstruktora** (ha vannak adattagjai), hogy a leszármazottból beállíthassuk az adattagok kezdőértékét.
  4. **Absztrakt osztály leszármazottja lehet absztrakt**, ha nem minden örökölt absztrakt metódust valósít meg. Azaz az absztrakt osztály absztrakt metódusait nem biztos, hogy a közvetlen leszármazott definiálja.
  5. Absztrakt osztálynak lehetnek nem absztrakt metódusai és egyéb (pl. static) tagjai is.
  6. Absztrakt metódusnak nem lehet módosítója a private, final, static mert akkor nem lehetne felüldefiniálni.

## Interface fogalma

* 1. **Az interfész egy viselkedésminta** leírása, amelyet egy osztály valósít meg.
  2. Az **interfész „szerződés” az interfész és az implementáló osztályok között**, aminek teljesülését a fordító ellenőrzi.
  3. Szemantikájában hasonlít az absztrakt osztályhoz. DE!
     1. Az **absztrakt osztály az osztályhierarchia része.** Leszármaztatással és az absztrakt metódusok felüldefiniálásával használjuk fel.
     2. Az **interfész nem kötődik egyik osztályhoz sem** (nem része az osztályhierarchiának). Csak egy mintát ír elő, amit **implementálni kell**.

**Implementálás: az interfészben deklarált metódusok megvalósítása**.

* 1. Használatának előnye: Amíg egy osztálynak csak egy közvetlen őse lehet, addig **egy osztály** **tetszőleges számú interfészt implementálhat**. (többszörös öröklés megvalósítása)

## Interface definíció

módosítók interface NameOfInterface {// interfész tagok}

* 1. **Minden interfész** implicite **absztrakt**, nem kell kiírni az abstract kulcsszót.
  2. Az **interfész minden metódusa** implicite **absztrakt** akkor is, ha nincs kiírva az abstract kulcsszó (csak deklarációk).
  3. Az **interfész** minden tagjának **automatikusan** **public** a hozzáférési módosítója (nem szoktuk kiírni).
  4. **Nincs konstruktora**, **nem példányosítható**. Nincsenek példány szintű adattagjai.
  5. **Interfészek** között lehet **többszörös** **öröklés**.
  6. Az interfész neve tetszőleges, az osztálynévhez hasonlóan nagy kezdőbetűs azonosító.
  7. Az interfész **csak absztrakt metódusokat és konstans (final static) adattagokat** tartalmazhat.
  8. **Többszörös öröklés lehetséges.** Pl.: egy leszármazott osztály implementál egy (vagy több) interfészt.

## Interface implementálása

módosítók class azonosító extends ősosztály implements interfészek {...}

**Egy osztály több interfészt implementálhat.** Interfészek: interface nevek vesszővel elválasztva.

Ha egy osztály implementál egy interface-t, akkor **meg kell valósítani az interface-ben levő összes metódust**, az örökölteket is. **Kivéve, ha az osztály abstract**, mert ekkor az interfész metódusait absztraktként jelölve **elhalaszthatjuk a metódusdefiníciót az absztrakt osztály leszármazottainak implementálásáig.**

## Abstract osztály vs interfész

* 1. Egy osztály megvalósíthat több interfészt. Egy interfész nem implementálhat interfészt.
  2. Egy osztálynak csak egy őse lehet, egy interfésznek lehet több őse is.
  3. Az interfész nem része az osztályhierarchiának.
  4. Ugyanazt az interfészt egymástól „független” osztályok is megvalósíthatják. Az absztrakt osztály absztrakt metódusait csak a leszármazottjai definiálhatják.

# Speciális szintaktikával definiálható osztályok: tömb, enum. Generikus osztályok.

## Tömb

* 1. Összetett típus. A tömbelemek indexelése 0-tól „méret-1”-ig.
  2. Tömb deklarálása: int[] myArray;
  3. A myArray referencia változó tetszőleges hosszúságú egész számokból álló tömbre hivatkozhat (még nem hivatkozik!). Csak a tömb változó jött létre (ami referencia típusú), a tömb még nem. []: a tömb típus jelölése
  4. **A tömb típusa lehet elemi v. referencia típus.** Referenciatömbök létrehozásakor csak a referenciák jönnek létre maguk az objektumok nem! Azokat külön létre kell hozni. Részletesebben később.
  5. Tömb deklarálása és létrehozása: int[] myArray = new int[10]; **A tömb változó objektumhivatkozást tartalmaz**.
  6. Értékadás tömbelemnek: myArray[0] = 5;
  7. **A tömb mérete** mindig ismert: Java: **myArray.length**
  8. A futtatórendszer tömbelemre hivatkozáskor ellenőrzi, hogy érvényes-e a megadott index. Indexhatár túllépésekor hiba váltódik ki. Java: IndexOutOfBoundsException
  9. Akkor is hibaüzenetet kapunk, ha nem létező tömb elemére hivatkozunk. Például: int[] myArray; myArray[0] = 5;
  10. A tömb mérete megadható kifejezéssel is, nemcsak konstanssal. Pl.: int[] myArray = new int[size\*2];
  11. Többdimenziós tömb típus nincs, de **egy tömb elemei lehetnek tömbök.**
  12. Tömbök tömbje: a tömb minden sora tömb, amelyek különböző méretűek lehetnek.
  13. Mátrix: minden sora azonos hosszúságú tömb.
  14. Tömbök tömbje (jagged array):

int[][] twoDimenArray = new int[2][];

//first row has 2 columns

twoDimenArray[0] = new int[2];

//second row has 3 columns

twoDimenArray[1] = new int[3];

## enum

* 1. **Konstansok fix méretű gyűjteménye.** Ezért **a neveik csupa NAGYBETŰ**-vel.
  2. **Enum** típus deklarálásakor **egy osztályt** definiálunk**. Minden Enum típus implicite kiterjeszti a java.lang.Enum osztályt** (azaz Enum típus nem származhat más osztályból).
  3. Osztály, de **nem lehet belőle leszármaztatni**.
  4. Osztály, de **nem lehet abstract** módosítója.
  5. **Az enum konstansok valójában** **public static final** referenciák az osztály egy-egy példányára.
  6. **Nem lehet példányosítani**.
  7. Az Enum osztály tagjai:
     1. Saját adattagok és metódusok
     2. Az Enum konstansait tartalmazó tömböt visszaadó metódus

public static enum-type[ ] values( )

* + 1. A paraméterben megadott Enum konstanst visszaadó metódus

public static enum-type valueOf(String str)

* 1. Definiálás (önálló osztályként vagy tagosztályként):

Példa:

public enum Day {SUNDAY, MONDAY, TUESDAY, WEDNESDAY,THURSDAY, FRIDAY, SATURDAY}

Hivatkozás:

Day day = Day.MONDAY;

## Generikus osztályok

Kacsacsőrök között egy általános <E> típus paraméter megadása.

Példa:

class Generikus<E> {

private E x;

public Generikus(E x) {

this.x = x;

}

public E getX() {

return x;

}

public void setX(E x) {

this.x = x;

}

}

A Generikus osztálynak van egy E típus paramétere. Ez az E az osztályon belül bárhol, ahol típust kell megadni, szerepelhet. A példában a Generikus osztálynak van egy adattagja, aminek a típusa az E lesz. A konstruktor beállítja ezt az adattagot, így paraméterben egy szintén E típusú paramétert vár. Az adattaghoz lesz egy getter, aminek így a visszatérési típusa is az E típus lesz, illetve egy setter, aminek pedig a paramétere lesz E típusú. Amikor a GenericsPelda main metódusában példányosítjuk ezt a Generikus osztályt, akkor egyszer E helyére Long kerül, és a konstruktor hívás paramétere is egy egész szám lesz, míg a másik példányosítás során String lesz. Valamennyi helyen, ahol a Generikus osztályban az E szerepelt, az az első esetben Longra, a másodikban Stringre "cserélődik", és ennek megfelelően kell a Generikus osztály metódusait paraméterezni.

# Kivételkezelés

* 1. **Kivétel definíció**: a programvégrehajtás során felmerülő esemény, amely megzavarja a normál programműködést. Java-ban **a kivétel objektum**, ami a kivétel bekövetkezésekor jön létre.
  2. **Hagyományos hibakezelésben** keverednek a funkcionális és a hibakezelő kódrészek, átláthatatlan, **nehézkesen** módosítható.
  3. **Kivételkezelés**: A hibakezelés egy formája, amelyben **elválasztható a hibát lekezelő kódrészlet** a normál működés kódjától, valamint jelezhetjük a kódunkban előforduló hibákat és előírhatjuk kezelésüket a kódot felhasználó másik kódrész számára.
  4. Kivétel (**exception**) létrejöhet egy végrehajtás során előforduló hiba hatására (I/O hiba, tömbindex túllépés, stb.).
  5. **Saját** magunk is létrehozhatunk **kivételeket** (pl. nullával osztás, negatív életkor, intervallum felsőhatára kisebb mint az alsóhatár, stb.)
  6. Amikor egy kivétel létrejön (kiváltódik), azt mondjuk, hogy **megdobódott a kivétel** (throwing an exception).
  7. A kivétel lekezelését a **kivétel elkapásának** nevezzük (catching an exception).
  8. **Java-ban a kivétel objektum**, ami a kivétel bekövetkezésekor jön létre.
  9. Ellenőrzött kilépés lehetséges.

Értesíteni a felhasználót. Vagy információt adni a kivétel fajtájáról, amit majd felhasználhatunk.

Kivételkezelő műveletek után folytatni az alkalmazást.

De például előtte minden adat elmenthető.

* 1. **Java-ban** a **try-catch-finally** háromrészből álló vezérlő szerkezet **szolgál a kivételkezelést** is tartalmazó kódrészlet írására.
  2. Ez a szerkezet lehetővé teszi, hogy a normál működés kódját elválasszuk a hibakezelő kódtól (ezáltal olvashatóbb kódot kapjunk) és hiba esetén a megfelelő hibakezelő kód fusson le.
     1. ”**try**”: normál működés kódja;
     2. ”**catch**”: ebből több is lehet -> mindben meg kell adni a kivétel tipusát és az azt lekezelő kódot;
     3. ”**finally**”: el is maradhat, de itt még elvarrhatjuk, amit el kell; (adatmentés, cleanup, fájl lezárása, erőforrás felszabadítása, stb.)
  3. **Ha a try-ban nincs hiba, akkor azonnal finally,** tehát ilyenkor a catch blokkok nem hajtódnak végre. **Ha van hiba, akkor a dobódott hibának megfelelő catch blokk hajtódik végre**. Ha a try-ban van **return** és az végrehajtódik, akkor a finally blook is kimarad. Ha egy származtatott exception osztályt hívunk, akkor ügyelni kell, hogy őt ne előzze meg az ősosztálya, mert akkor nem ő fog megfutni, hanem az őse.
  4. **Vannak kötelezően és nem kötelezően kezelendő kivételek.**
     1. **kötelező -> ha nincs kezelve, akkor a kód nem fordul le:** IOException; ClassNotFoundException; FileNotFoundException;
     2. **nem kötelező - > ha nincs kezelve, attól még a kód lefordul:** sss
  5. **Exception példák**
     1. NegativeArraySizeException,
     2. IOException
     3. FileNotFoundException,
     4. SQLException,
     5. ClassNotFoundException;
  6. Az **„Elkapni vagy előírni”** követelmény;

Akkor helyes a Java kód, ha kielégíti az „Elkapni vagy előírni” (Catch or Specify) követelményt. Ez azt jelenti, hogy azt a kódrészt, amelyik dobhat kivételt

* + 1. vagy–try – catch blokkba kell zárni (azaz a kivételt el kell kapni és le kell kezelni);
    2. vagy–olyan metódusban kell megadni, amelyik előírja, hogy kivétel(eke)t dobhat (throws);

Az a kód, amelyikben kötelezően lekezelendő kivétel dobódhat és nem teljesíti ezt a követelményt, nem fordul le!

* 1. **Kivétel megdobása**: kivételt bármely kódrészben dobhatunk: throw throwableObject;
  2. A dobott kivételt vagy az adott metóduson belül kell lekezelni (catch blokkban), vagy tovább dobhatjuk a catch blokkban (throw), vagy jelezzük a metódus fejlécében (throws), hogy a hívó metódus kezelje.
  3. **Saját kivétel osztály létrehozása**
     1. vagy az **Exception** osztályból származtatjuk -> ekkor kötelezően kezelendő lesz a hiba, ellenben nem fordul le a program.
     2. vagy a **RuntimeException** osztályból származtatjuk -> ekkor nem kötelező, lefordul a kód anélkül is.

# A lang csomag osztályai: Object, Comparable, System, String, StringBuilder, csomagolóosztályok.

5.PDF-ben!!!

A System, String osztályok végleges osztályok, nem lehet őket leszármaztatni.

## Object osztály

* 1. Minden osztály őse;
  2. Konstruktura üres: Object();
  3. Fontosabb metódusai:
     1. toString(); -> az objektum string leírásával tér vissza;
* célszerű minden osztályban felüldefiniálni úgy, hogy az adott osztály adattartalmára utaló Stringet adja vissza;
* minden objektum képes String-gé változni ezzel a metódussal;
  + 1. clone(); -> visszaadja az objektum másolatát; (KELL: java.lang.Clonable; enélkül Exception)

sekély <-> mélymásolat : csak referencia másolódik <-> új objektumként másolódik;

* + 1. equals(object obj); -> igaz, ha a paraméterben megadott objektummal egyező a hívó;
* ezt is célszerű minden osztályban felüldefiniálni;
* tartalmi egyezőség vizsgálata; public boolean equals(Object obj) {return this == obj;}
  + 1. getClass(); -> objektum futásidejű típusával tér vissza;
    2. hashCode() -> Objektum hash kódjának előállítása (a memóriacímből képzi a kódot).
       1. Ha az equals() metódus szerint két objektum egyenlő, akkor a hashCode() metódusnak uazt az értéket kell visszaadnia.
    3. finalize(); szemétgyűjtő hívja, mielőtt automatikusan megszűntetné az objektumot;

## String

* 1. Szerepe: Konstans szövegek tárolása;
  2. final osztály, azaz nem terjeszthető ki;
  3. A ”szöveg” hatására is létrejön egy String objektum és annak a referenciája található a helyén. Vagyis pl. ”szöveg”.length() helyes.
  4. A String objektumokra definiált a + operátor, mint összefűzés;
  5. Sok hasznos metódusa van, pl: length(), toUpperCase(), toLowerCase(), trim(), split(), isEmpty()

## StringBuilder/Buffer (egyszál/több szál)

* 1. Szerepük: Változtatható szövegek tárolása;
  2. Szöveg változtatásához használt metódusaik: append(), insert(), delete(), replace(). A reverse()

## Math

* 1. Matematikai konstansokat, függvényeket tartalmaz, minden tagja static;
  2. final osztály és nem példányosítható.
  3. fontosabbak: PI, E konstans adattagok; pow(), sqrt(), abs(), ceil(), floor(), round(), min(), max(), random();

## Comparable osztály

* 1. Gyakran használt interface, a relációkat he
  2. lyettesíti.
     1. Például az Arrays osztály sort() metódusával egy már feltöltött Kör objektumokat tartalmazó „a” tömb rendezése: Arrays.sort(a); de ez csak akkor működik, ha a Kör osztályom implementálja a Comparable interface-t.

## System és egyéb

## Csomagoló (wrapper) osztály

* 1. Minden primitív típusnak van nagybetűvel kezdődő csomagoló (wrapper) osztálya (Java).
  2. Minden elemi típusnak van Osztály megfelelője: Byte, Short, Integer, Long, Float, Double, Boolean, Character;
  3. Konverziós metódusokat is tartalmaznak:
     1. String → numerikus: parseTipus(String s), pl. parseInt(String s)
     2. Numerikus → numerikus: intValue(), floatValue(), stb.
     3. Numerikus → String: toString()
     4. Elemi típus String-é konvertálása:
     + String összefűzéssel: String s1 = "" + 5;
     + A String osztály valueOf metódusával: String s2 = String.valueOf(5);
     + A csomagoló osztály toString metódusával: String s3 = new Integer(5).toString();
     + A csomagoló osztály static toString metódusával: String s4 = Integer.toString(5);
  4. Szerepük: az elemi típusok használhatók legyenek, olyan helyen, ahol referencia típus kell. Pl. generikus osztályokban.
  5. Az ilyen típusú változók (referencia változók) objektum hivatkozásokat tartalmaznak. Az objektum hivatkozás azt adja meg, hogy egy adott érték hol található a memóriában.
  6. Ezek az osztályok tartalmazzák az adott primitív típushoz tartozó konstansokat és alapvető műveleteket.
  7. Mikor van rá szükség?
     1. Amikor primitív típusú értéket olyan kontextusban akarunk használni, ahol csak objektum hivatkozás szerepelhet.
     2. Amikor szükség van az osztály műveleteire (pl. típuskonverzió esetén).

# Osztályok közötti kapcsolatok.

EGYÉB FOGALMAK

1. **Imperatív (procedurális) programozás:** egy imperatív program parancsok soraival bírja a számítógépet működésre. Két fő jellemzője: a parancs és az állapot. Ilyenek: C, C# Pascal, C++, (Ellentéte a deklaratív programozás: ez arra összpontosít, hogy a program mit érjen el, anélkül, hogy meghatározná, hogy hogyan.)
2. **Annotáció:** Az osztályok definiálása mellett a Java nyelvben van egy olyan lehetőségünk, hogy úgynevezett annotációk segítségével a programot kiegészítő adatokkal (metaadatokkal) lássuk el. Ezeket különböző programkód elemekhez rendelhetjük (csomagokhoz, típusokhoz, metódusokhoz, attribútumokhoz, konstruktorokhoz, lokális változókhoz).

Felhasználási mód szerint az annotációknak három típusát különböztetjük meg:

* Információkat szolgáltat a fordítónak hiba detektáláshoz
* Fordítási és fejlesztési idejű utasításokat ad (kód, XML, dokumentum generáláshoz)
* Futásidejű utasításokat ad, amelyeket a java reflection által feldolgozhatunk (pl.: objektum tulajdonságainak ellenőrzéséhez)

Pl:

* @Override (előírja, hogy az adott metódus egy ős osztálybeli metódus felülírt változata);
* @Deprecated (adott osztály elavult, az osztály használata nem javasolt)

1. **Literál**: karakter vagy karakterlánc adatok megjelenési formája, melynek legfontosabb tulajdonsága a többi, deklarálható változóhoz képest, hogy ellentétben velük egy literálra nem hivatkozhatunk a program többi részéből.
   1. Boolean logikai literál 2 értéket vehet fel: true és false;
   2. char literál: 1db karaktert vehet fel; hivatkozhatunk rá:
      1. közvetlenül: ’J’ (szimpla aposztrof)
      2. unicode segítségével: ’\u0061’
   3. String literál: karakterekből álló szövegszekvencia; -> „ez egy szöveg” („” használat!)
   4. PL int literál típus:
      1. DECIMAL: int x = 101; -> 101 (klasszik)
      2. OCTAL: int x = 0146; -> 64 // The octal number should be prefix with 0.
      3. HEXA: int x = 0X123Face; ->
      4. BINARY: int x = 0b1111; -> 15
2. **Continue fgv**: 09\_1-1-a 30:20 percnél! (Ezzel vissza tudjuk küldeni egy adott pontra a futást.)
3. **enum**: az enum értékek nem String-ek hanem konstansok, ezért azokat közvetlenül össze lehet hasonlítani. Ha String lenne, akkor kellene az equals metódus valahogy így: bookArray[i].author.equalsIgnoreCase(bookArray[j].author) VAGY ÍGY dog.getBreed().equals(breed)
4. Comparable interfész használata tömb rendezésre: csak egy mező szerint lehet rendezni;
   1. abban az osztályban, amelynek az elemeit rendezni szeretnénk:
      1. implements Comparable:

public class UniEmployee extends Employee **implements Comparable<UniEmployee>**

* + 1. automatikusan is létrehozható a metódus, de az annotáció nem kell;
    2. meg kell írni az összehasonlítót a „return” mögé (kivonási sorrend cserével lehet növekvő vagy csökkenő sorrendet elérni);

**public** **int** compareTo(UniEmployee o) {

**return** **this**.getAge() - o.getAge();

* 1. A futtatható osztályban
     1. az „Arrays” osztály „sort” metódusát kell futtatni, valahogy így:

Arrays.sort(empArray);

1. Comparator interfész használata tömb rendezésre: több mező szerint is lehet rendezni;
   1. abban az osztályban, amelynek az elemeit rendezni szeretnénk
      1. új beágyazot **static** osztály létrehozása, amely implementálja a comparator interfészt az UniEmployee osztályra
      2. int metódus megírása
         1. bemenet 2 osztály példány
         2. return a 2 osztály összehasonlítandó mezőjének a „compareTo” fgv-e;

**public** **static** **class** NameSorter **implements** Comparator<UniEmployee>{

**public** **int** compare(UniEmployee o1, UniEmployee o2) {

**return** o1.getName().compareTo(o2.getName());

* 1. A használó osztályban
     1. az „Arrays” osztály „sort” metódusát kell futtatni, valahogy így(az empArray a futtatható osztályban a példány; az UniEmployee pedig az ez osztály, ahol implementáltuk a comparatot fgv-t;):

Arrays.*sort*(empArray, **new** UniEmployee.NameSorter());

*listArray*(empArray);

1. Beágyazott osztály: egy osztályon belül definiálok egy másik osztályt; pl. a comparator osztály ilyen. Ekkor „static” kulcsszóval hozom létre.

AZONBAN: szervezhetem ezeket egy külön osztályba is, amikor is tilos lesz a „static” kulcsszó!!!

1. **Dinamikus tömb**: ArrayList; LÉTREHOZÁSA a futtatható osztály Main metódusában:

ArrayList<UniEmployee> empArray = new ArrayList<UniEmployee>();

1. Comparator hegyek: Gy09\_1-2-a
2. Címkézett utasítás ciklusban
   1. **continue:** a ciklus blokk végére ugrik, azaz a ciklusmag további része nem hajtódik végre, és a következő iterációval folytatja a végrehajtást;
   2. **break:** kilép az aktuális blokkból és a blokkot követő utasításra ugrik a vezérlés. Csak ciklusban és switch-ben használható;
   3. **return:** metódusban használható, visszatér a metódus meghívásához és a rákövetkező utasítással folytatódik;
3. stringek azonossága:
   1. **pl string tömb esetén**: names[i].equals(names[j] eredméy -> true/false;
4. tömb műveletek -> binary search kell a vizsgához;
5. Java8 comparator: sokkal egyszerűbb, mert nem kell külön ösztályban definálni; **EZT KELL TUDNI!!**
6. throw VS throws utasítás KIVÉTELKEZELÉSBEN VIZSGÁRA TUDNI.
7. FILEREADER / FILEWRITER
8. FÁJLKEZELÉS LÉPÉSEI 78. oldal!!!
9. Interface getter és settere speciális, nem is generálható automatán.

public default Color getAlapertszin() {

return alapertszin;

}

public void setAlapertszin();