Pázmány Péter Katolikus Egyetem
Információs Technológiai és Bionikai Kar

OOP

JAVA PROGRAMOZÁS

3. GYAKORLAT

Amiről ma szó lesz

Objektumok

Csomagok

Öröklődés

Debugger használata

OOP vs. egyéb programozási módszerek

Az **objektumorientált programozás** alapötlete, hogy a tárgyi világ fogalmait használja fel a tervezés során. A részfeladatok avagy **felelősségi körök** ellátására **objektumokat** rendel, ezek – természetesen a programozó által definiált – interakciójából áll elő a feladat megoldása.

Minden objektum rendelkezik bizonyos tulajdonságokkal és képes bizonyos tevékenységekre. A forráskód szintjén ennek a két fogalomnak a **tagváltozók** és a **metódusok** felelnek meg.

Az OOP nem nyelvi elemek készlete, hanem egy tervezési módszertan, amit a legtöbb ma népszerű programnyelv nyelvi képességekkel is támogat. Objektumorientáltan programozni ennek hiányában is lehet, csak nehezebb. Példa: GTK+.

Az OOP alapfogalmai

- Objektum, példány, egyed
- Osztály
  - Változók
    - Osztályváltozók
    - Példányváltozók
  - Metódusok
    - Osztálymetódus
    - Példánymetódus
  - Konstruktor
  - Destruktor
  - Inicializáló blokk (lehet osztály vagy példány inicializáló)
- Láthatósági szintek
- Egységbezárás
- Adatelrejtés

Objektum, Osztály

Objektum:

- Program entitás: állapot és funkcionalitás
- Egyedi, zárt (jól definiált interfésszel rendelkezik, amelyen keresztül az állapota változtatható, műveletei elérhetők)
- Kapcsolatok objektumok között: ezek az osztályok

Osztály:

- Hasonló objektumok gyűjteménye:
  - Strukturális vagy funkcionális hasonlóság
- Objektumok reprezentációjának megadása
- Osztály: egy (felhasználói) típus
- Példányosítás: egy osztály egy objektumát képezzük

Objektum létrehozása

Egy egyed egy osztály megvalósítása, más szóval **példányosítása**. Javaban ezt így érjük el:

```
Dog a = new Dog();
```

A példányosítás a **memóriaterület lefoglalásával** és az **arra mutató referencia beállításával** kezdődik.

Ilyenkor lefut az osztály **konstruktora** (és inicializáló blokkjai), ahol meghatározhatjuk a kezdőértékeket, inicializálhatjuk az objektumot.

A „dog” referencia a **new operátorral** létrehozott objektumra mutat.

Ha egy tagváltozó esetében elmulasztjuk a példányosítást, akkor automatikusan null-ra inicializálódik az objektum. A lokális változókra ez nem igaz, az inicializálatlan lokális változók használata fordításidejű hiba.

Osztály létrehozása

Egy osztálynak lehetnek

- Példányváltozói, példánymetódusai
- Osztályváltozói, osztálymetódusai

A példányváltozókból minden egyedbe kerül egy-egy, az osztályváltozókból egyetlen egy létezik csak, ezen osztozik az összes objektum.

Példánymetódus: legalább egy példányváltozót felhasznál.

Osztálymetódus: csak osztályváltozókat használ.

Java esetén egy osztály **definíciója és deklarációja**, azaz az eljárások feje és megvalósítása **nem szétválasztható.**

Példa – Car

Valósítsunk meg egy `Car` osztályt Javaban!

```
package vehicles; // az osztályt tartalmazó csomag

public class Car { // az osztály kezdete

  // az osztály megvalósítása

} //az osztály vége
```

Példa – Car – változók

A változók felsorolásán túl a Java megengedi, hogy inicializáljuk is azokat, konstruktoron kívül

Változó deklarációja:
`[módosítók] típus változó [inicializáció]`

```
public class Car {
   static private int fuelPrice = 400; // statikus változó
   private String type;
   private double consumption;
   private double km = 0; // példa kezdőértékre
   private double fuel = 0;
   private int capacity;
   public String licence;
   //további metódusok

}
```

Konstruktor

Programkód, ami a **példányosításkor automatikusan** végrehajtódik.

Ennek **neve megegyezik az osztály nevével**, nincs visszatérési értéke.

Hasonlít a metódusokra, de nem pont ugyanolyan (**nem tag**, mert például nem öröklődik).

Minden osztálynak van konstruktora. Ha mi nem írtunk, a fordító automatikusan létrehozza és a példányosításkor meghívódik.

Konstruktorból **lehet több is**, túlterhelésükre ugyanaz vonatkozik, mint a függvények túlterhelésére.

Példa – Car – Konstruktor

```
public Car(String type, double consumption, int capacity) {
        this.type = type;
        this.consumption = consumption;
        this.capacity = capacity;
}

public Car(String type, double consumption, int capacity,
                double km) {
        this.type = type;
        this.consumption = consumption;
        this.km = km;
        this.capacity = capacity;
}
```

A `this` pszeudováltozó

Az osztálydefiníción belüli példánymetódusokban a **`this`** névvel hivatkozhatunk az adott objektumra.

Osztályszintű metódusokban nem használható.

Ha több konstruktort is definiálunk, az egyes konstruktorok elején, minden más utasítás előtt, van lehetőség valamelyik másik konstruktor végrehajtására a **`this`** kulcsszóval:

```
public Car (String type, double consumption, int capacity, double km) {
        this(type, consumption, capacity);
        this.km = km;
}
```

Metódusok

Alprogramok, melyek objektumokhoz vannak kapcsolva.

Érdekesség, hogy a **metódus neve megegyezhet egy változóéval** is.

A metódusok túlterhelhetők.

Túlterhelés: ugyanolyan azonosítójú (nevű), de különböző szignatúrájú függvények létrehozása.

Szignatúra: azonosító, paraméterlista

Függvény feje a Javaban: módosítók, a visszatérési érték típusa, azonosító név, paraméterlista, kivételek (melyek dobásra kerülhetnek)

Metódus definíciója: fej(specifikáció)+törzs

```
[módosítók] típus név([paraméterek]) [kivételek] {törzs}
```

Metódusok a Car osztályban

```
public void refuel(double liter) {
        fuel += liter;
}

private double fuelNeed(double km) {
        return (km / 100.0) * consumption;
}

public double cost(double km) {
        return fuelNeed(km) * fuelPrice;
}
```

Példa – Car - Metódusok

```
public double go(double km) {
    if (fuelNeed(km) <= fuel) {
        fuel -= fuelNeed(km);
        this.km += km;
        System.out.println("Megtettünk " + km + " km utat!");
        return km;
    } else {
        System.out.println("Nincs elég benzin "
            + km + " km úthoz!");
        return 0;
    }
}
```

Módosítók I.

Osztály módosítói:

- **`public`**: Az osztály bárki számára látható
- **`final`**: Végleges, nem lehet belőle leszármazni. Az öröklődésnél lesz róla szó bővebben.
- **`abstract`**: Az öröklődésnél lesz róla szó, csak leszármazni lehet belőle, példányosítani nem (a tervezés eszköze).
- (Üres): A csomagon (package) belül bárki számára látható.
- **Amik kombinálhatóak**:
  - **`public` + `final`**, **`public` + `abstract`**
- **Amik nem kombinálhatóak**:
  - **`final` + `abstract`**

Módosítók II.

Mező módosítói:

- **`public`**: Az osztály példányait használó bármely kód számára közvetlenül hozzáférhető.
- **`private`**: Csak azon osztály objektumai számára elérhetők, melyben meghatározták őket.
- **`protected`**: A definiáló osztály és leszármazottainak objektumai számára és a csomagon belül látható.
- (Üres): A csomagon (package) belül bárki számára látható.
- **`final`**: Végleges, azaz konstans érték**.** Nem ugyanaz mint a C++ const! Itt a referencia az az érték, ami nem változtatható, maga az objektum módosítható.
- **`static`**: Osztályváltozó, egy osztályhoz csak egy tartozik (vagyis az osztály minden példányához is ugyanaz a változó tartozik).
- **Amik kombinálhatóak**:
  - Láthatóság + **`final`** + **`static`**
- **Amik nem kombinálhatóak**:
  - Láthatósági szintek, egyszerre csak egy láthatósági szint használható (nyilvánvalóan)

Módosítók III.

Metódus módosítói:

- **`public`**: Az objektumot használó bármely kód számára közvetlenül hozzáférhető.
- **`private`**: A definiáló osztály metódusai számára látható.
- **`protected`**: A definiáló osztály és leszármazottainak objektumai számára és a csomagon belül látható.
- (Üres): A csomagon (package) belül bárki számára látható.
- **`static`**: Osztálymetódus, példányosítás nélkül használható, csak statikus változókat használhat.
- **`final`**: Végleges, a leszármazott nem definiálhatja felül. (Öröklődésnél bővebben)
- **`abstract`**: Absztrakt osztálynál használható, kötelező felüldefiniálnia, azaz megvalósítania a leszármazott osztálynak.
- **Amik kombinálhatóak**:
  - Láthatósági szint + **`static`** + **`abstract`**
- **Amik nem kombinálhatóak**:
  - **`abstract`** + **`final`**, **`abstract`** + **`static`**, **`final`** + **`static`**

Láthatóság

Ami hozzáfér		Amihez hozzáfér			
Csomag	Osztály	public	default	protected	private
Azonos	Ez	X	X	X	X
	Belső	X	X	X	X
	Leszármazott	X	X	X	
	Más	X	X	X	
Másik	Leszármazott	X		X	
	Más	X			

A `static` kulcsszó

Javaban **minden objektum** – maga az osztály is egy objektum.

A **`static` kulcsszó ennek a felfogásnak a nyelvi leképződése**, mivel ezzel az osztályhoz, mint objektumhoz rendelünk egy változót.

Ugyanez elmondható a **`static` metódusokról** is.

Mivel a **`static`** metódus az **osztályhoz, mint objektumhoz tartozik**, így nem használhat fel (olvashat, írhat) változókat az osztály példányaiból**.**

Példa a `static` használatára

```
static private int fuelPrice = 400;

static public int fuelCost(int liter) {
        return fuelPrice * liter;
}
```

A benzin minden autó számára ugyanannyiba kerül, típustól függetlenül.

Ennek megfelelően természetesen 10 liter benzin megvásárlása is ugyanannyi pénz minden autótulajdonos számára, függetlenül attól, hogy ő például ezzel hány kilométert tud megtenni.

Getter & Setter

Osztályok leírásánál legtöbbször **elrejtjük a belső változókat** a külvilág elől, mert például

- nem akarjuk, hogy megváltoztassák (integritás)
- módosítottan akarjuk megjeleníteni vagy átadni
- a változtatásra reagálni akarunk (esemény)
- az objektum felhasználói így garantáltan nem függenek az osztály belső struktúrájától, azt szabadon változtathatjuk
- vagy egyszerűen nem akarjuk, hogy lássák

Erre használjuk a **`private`** módosítószót a változóknál.

Ilyenkor az elérést a **getter** és **setter** függvények biztosítják.

Getter & Setter – Példa

```
public double getFuel() {
        // itt hajthatjuk végre a konverziót,
        // ellenőrzést stb
        return fuel;
}

public double getKm() {
        return km;
}

public void setConsumption (double newConsumption) {
    this.consumption = newConsumption;
}
```

Getter & Setter – gyorsan

Bal oldalt az osztály nevére jobbklikk

→ Source → Generate Getters and Setters...

Majd a listából kiválasztjuk, hogy mely változókhoz szeretnénk getter vagy setter függvényt generálni.

Ugyanígy konstruktorokat is lehet generálni.

toString()

A Javában **minden egyednek van egy függvénye, ami szöveges információt hordoz a példányról**, ez a public String **`toString()`** függvény.

Ez a metódus lényegében megadja az **adott objektum `String` reprezentációját**. A println a toString segítségével bármilyen objektumot “ki tud írni”.

```
System.out.println(objektum);
```

Ha nem hozzuk létre, az osztályunk örökli az ősosztálytól (erről bővebben később).

```
public String toString() {
    return type + ", benzin: " + getFuel() + ", km: " + getKm();
}
```

Az üres referencia, objektum élettartama

Ha egy **változó értéke `null`, akkor nem mutat objektumra.**

A `null` referencia minden osztály esetén használható.

Nincs olyan utasítás Javaban, amivel egy objektumot explicit módon meg lehet szüntetni.

Ha már nem hivatkozunk egy objektumra többet, akkor a referenciát `null`-ra lehet állítani. De ez magát az objektumot nem szünteti meg.

Szemétgyűjtés

A már nem használt objektumokat a garbage collector automatikusan kitörli a memóriából, a programozónak nem kell (és nincs is lehetősége) ezzel foglalkozni.

Inicializáló blokkok

Inicializáló blokk: Utasításblokk a tagok (példány- és osztályszintű változók és metódusok) és konstruktorok között, az osztálydefiníción belül.

```
public class A {
        int[] array = new int[10];

        {
                for (int i = 0; i < 10; i++) {
                        array[i] = (int) Math.random();
                }
        }
}
```

Inicializáló blokkok

Az előbb példány szintű inicializáló blokkot láttunk. Ugyanez megtörténhet **osztály szinten** is:

```
public class A {
        static int i = 10;
        static int ifact;
        static {
                ifact = 1;
                for (int j = 2; j <= 10; j++) {
                        ifact * = j;
                }
        }
}
```

Inicializáló blokkok – Miért is?

Osztályszintű inicializátor: az osztály inicializációjakor fut le, az osztályszintű konstruktorokat helyettesíti (hiszen olyanok nincsenek).

Példányinicializátor: példányosításkor fut le, a konstruktorokat egészíti ki; pl. oda írhatjuk azt, amit minden konstruktorban végre kell hajtani (névtelen osztályoknál is jó, mert ott nem lehet konstruktor).

Több inicializáló blokk is lehet egy osztályban.

Végrehajtás (mind osztály-, mind példányszinten): a változók inicializálásával összefésülve, definiálási sorrendben; nem hivatkozhatnak később definiált változókra.

Nem lehet benne `return` utasítás.

A példányinicializátor az osztály konstruktora előtt fut le, de az ősosztályok konstruktora után.

Nem szoktuk az „osztálytagok” közé sorolni.

Csomagok

Az osztályokat nagyobb funkcionális egységekbe, **csomagokba** szervezzük, a csomagokat pedig csomag-hierarchiákba.

Minősített hivatkozás pontokkal:

```
java.util
```

Egy típus teljes neve tartalmazza az őt befoglaló csomag nevét is:

```
java.util.ArrayList
```

Csomagok megadása

A **`package`** kulcsszó segítségével.

```
package warehouse;
public class Screw {
  //...
}
```

Csomag hierarchiákat is létrehozhatunk:

Példa: `package raktar.adatok` → egy adatok alcsomag a raktár csomagon belül. (Egy csomag: egy könyvtár a fájlrendszerben.)

Láthatósági szintek csomagokban

A **láthatósági módosítók** (az osztályokra vonatkozóan):
`public` – mindenki látja

Ha nem írunk módosítót az osztály elé, akkor a láthatósági szintje „félnyilvános”, azaz **csomagszintű** (**package-private**).

Az azonos csomagban levő osztályok hozzáférnek egymás félnyilvános tagjaihoz. (A protected lényegében ennek a kiterjesztése.)

Hivatkozás más csomagokra

Ha egy forrásfájlban használni akarunk egy típust egy másik csomagból:

Írjuk ki a teljes nevét:

```
java.util.ArrayList a = new java.util.ArrayList();
```

Importáljuk import utasítással:

```
import java.util.ArrayList;
import java.util.*;
ArrayList a = new ArrayList();
```

import `java.util.*` : nem csak egy adott osztályt tölt be, hanem a csomagban található összes osztályt.

Fordítási egység

Az, amit oda lehet adni a fordítónak, hogy lefordítsa.

Java-ban egy fordítási egység tartalma:

`package` utasítás (opcionális)

`import` utasítások (opcionális)

típusdefiníciók (**`public`** típusból (**`class`**-ból) csak egy)

A főprogram működése

Egy Java program egy **osztály „végrehajtásának”** felel meg

- ha van statikus main nevű metódusa.

A Java-nak megadott osztályt inicializálja a virtuális gép (pl. az osztályváltozóit inicializálja), és megpróbálja elkezdeni a `main`-jét.

- **Ha nincs `main`: futási idejű hiba.**

Nincs `return`!

Végetérés: ha a `main` véget ér, vagy meghívjuk a `System.exit(int)`-et. (Ennek paramétere a kilépési állapot, normál kilépés esetén 0).

Pár szó a JVM-ről

A **forrásfájlokból** a fordító **.class** kiterjesztéssel **bájtkódot** készít.

Az alkalmazás elindításánál a **`ClassLoader`** egy adott .class fájlt **a megfelelő osztályra való első hivatkozáskor** tölt be a JVM-re.

Ezután következik a **bájtkód ellenőrzése**, majd az osztályt leíró elemek a **`java.lang.Class`** osztályba kerülnek. Ekkor fut le az **osztályszintű inicializáló blokk**.

A `java.lang.Class` tartalmazza a használt osztályok reprezentációját. `forName(String name)` metódusa segítségével explicit betölthetjük a megfelelő osztály bájtkódját rá való **hivatkozás vagy példányosítás nélkül**. (Ez történik pl. különböző driverek betöltése esetén.)

Összefoglalás I.

Amit eddig megtanultunk:

Mi is az az OOP?

- Osztály, objektum
- Példányváltozók, példánymetódusok
- Osztályváltozók, osztálymetódusok (`static`)
- Konstruktorok, getterek és setterek, `toString()`
- Láthatósági szintek, és hogy miért használjuk ezeket
- Inicializáló blokkok
- `java.lang.Class`

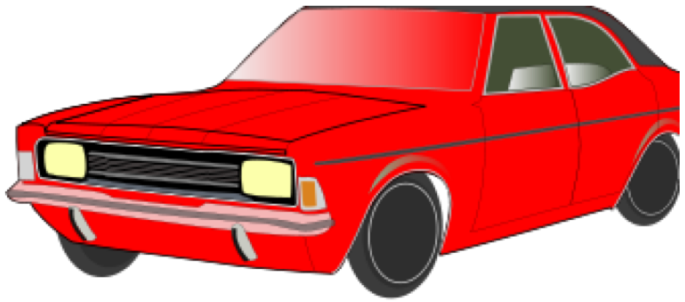
Csomagok

Hogyan működik egy Java program

Öröklődés - Motiváció

Új osztályt szeretnék egy meglévő alapján megírni.

- Egy osztály adattagjait, metódusait felhasználva szeretnénk új funkcionalitást létrehozni.
- Szeretnénk meglévő osztályainkat bővíteni új adattagok és metódusok felvételével.
- Szeretnénk már meglévő metódusokat, adattagokat felüldefiniálni az új funkciónak megfelelően.



A Java megoldás

```
class Child extends Parent { /* Child törzse */ }
```

- `extend` = kiterjeszteni
- Az ily módon megadott ún. **leszármazott osztály** (*Child*) **örökli a szülője** (*Parent*) **tulajdonságait**, azaz a belső állapotát leíró változóit és a viselkedést megvalósító metódusait.
- Természetesen az osztály törzsében ezeket tovább lehet bővíteni, a viselkedését módosítani, ezt sugallja az `extends` kifejezés a definíciónál.

Öröklődés, mint osztályok közötti kapcsolat

Javaban az **`extends`** kulcsszó által leírt öröklődés **is-a reláció**:

```
class Child extends Parent { /* osztály törzse*/ }

Child c;
```

A `c` típusa `Parent`, azon belül pedig `Child`

Ez tehát azt is jelenti, hogy **a leszármazott osztály létrehozásával egy altípust képzünk**.

Javaban a tartalmazási (has-a) relációt nem lehet öröklődés útján kifejezni.

Összefoglalva

Az öröklődés révén **kód-újrafelhasználás** valósul meg, ami:

- a kód redundanciáját csökkenti
- nem csak a programozást könnyíti meg, hanem az olvashatóságot és a karbantarthatóságot is növeli

Nem csupán csak kódmegosztást jelent, hanem **altípus képzést** is

- **tervezési szintű fogalom**

Osztályhierarchia

Az öröklődési reláció irányított gráf reprezentációját osztályhierarchiának is nevezik.

Javaban van egy "univerzális ősosztály", az `Object`, minden osztály ennek a leszármazottja.

Javaban csak egyszeres öröklődés van, tehát az osztályhierarchia Java esetén egy fa.

Ha nem adunk meg `extends`-et, akkor **`extends`** `Object` értendő.

`Object`: predefinit, a `java.lang`-ban van definiálva, olyan metódusai vannak, amelyekkel minden objektumnak rendelkeznie kell.

- Pl. `toString(), equals(), hashCode()`

Minden, ami nem primitív típusú (`int, char`, stb.), az `Object` leszármazottja.

Öröklődés lehetőségei

A leszármazott osztályban közvetlenül használhatjuk a szülőosztály `public` és `protected` metódusait, tagváltozóit (egy leszármazott osztály örökli a szülő osztály tagjait, így a szülő osztály `private` tagjaiból is van példánya, de ezekhez nem férhet közvetlenül hozzá).

- Deklarálhatunk olyan nevű adattagokat, amelyekkel a szülő osztály már rendelkezik, elfedve a szülőosztálybeli elemeket.
- Írhatunk példánymetódust, mely szignatúrája megegyezik egy szülőosztálybeli példánymetódussal, felüldefiniálva azt.
- Túlterhelhetünk egy szülőosztályban meglévő metódust.
- Írhatunk osztálymetódust, mely szignatúrája megegyezik egy szülőosztálybeli osztálymetódussal, elfedve azt.
- Írhatunk teljesen új metódusokat.
- Deklarálhatunk teljesen új adattagokat.

A **`super`** kulcsszóval hivatkozhatunk szülőosztálybeli elemekre (pl. konstruktorra, hiszen az nem öröklődik, vagy olyan függvényre, melyet a leszármazottban felüldefiniáltunk vagy elfedtünk.)

Példa: Taxi osztály Car osztályból

Hozzunk létre egy Taxi osztályt a már meglévő Car osztályból.

Vegyünk fel **újabb adattagokat** a Taxi osztályba:

- pénztárca: A sofőr pénzének regisztrálása
- Kilométerdíj: a fuvarozás díja kilométerenként

```
public class Taxi extends Car {
        protected double budget;
        protected double kmCost;
// függvények
}
```

Példa: `Taxi` konstruktorok

A leszármazott **nem örökli a szülő konstruktorát**. De van lehetőségünk a **leszármazott konstruktorának ELSŐ sorában meghívni a szülő valamelyik konstruktorát a `super(...)`** kulcsszóval.

Ha ezt nem tesszük meg, vagy ha nem is definiálunk konstruktort, **akkor is végrehajtódik a szülő paraméter nélküli konstruktora** (ha van ilyen, **ellenkező esetben fordítási hiba**).

```
public Taxi(String type, double consumption,
                        int capacity, double kmCost) {
        super(type, consumption, capacity);
        this.kmCost = kmCost;
        this.budget = 0;
}
```

Konstruktorok lefutási sorrendje

A lefutási sorrend példányosítás esetén a következő:

- Ősosztály static inicializáló blokkja
- Osztály static inicializáló blokkja
- Ősosztály inicializáló blokkja lefut
- Ősosztály konstruktora
- Osztály inicializáló blokkja lefut
- Osztály konstruktora

Amit ebből mindenki jegyezzen meg: egy leszármazott osztály konstruktora mindig a szülő osztály konstruktora után fut le. (Még akkor is, ha az nincs explicit meghívva!)

Nézzük meg ehhez a `runningorder` csomag tartalmát!

És többek között vegyük észre, hogy bár nem hívtuk meg explicit módon a leszármazottban a szülőosztály konstruktorát, mégis lefutott az ősosztály paraméter nélküli konstruktora!

`Taxi` felüldefiniálás I.

Taxi esetén máshogy számoljuk a **költségeket** (ehhez hozzájön a kilométer díj), továbbá mivel van már pénztárcánk, így a benne levő pénzt kezelni kell **tankolás** során, végül az osztály String reprezentációja is változik, tehát új **`toString()`** metódus is kell.

- **Felüldefiniálás**

Írhatunk a leszármazott osztályban **olyan példánymetódust, mely szignatúrája megegyezik egy szülőosztálybeli példánymetódussal**. Ezt nevezzük **felüldefiniálásnak**.

Taxi felüldefiniálás II.

A felüldefiniálás szabályai:

- **Szignatúra** megegyezik (különben **túlterhelés** lenne).
- **Visszatérési érték** típusa megegyezik, vagy a felüldefiniált metódus visszatérési értékének altípusa (kovariáns visszatérési érték).
- **Láthatóság** nem szűkíthető.
- Specifikált **kiváltható kivételek** nem bővíthetőek.
- Ezután a **felüldefiniált metódus** a **`super`** kulcsszó segítségével továbbra is **elérhető marad**.

Túlterhelés emlékeztető

Ugyanazt az azonosítót használjuk különböző szignatúrájú függvények esetében.

Egy névhez több funkciót rendelünk hozzá.

Fontos szabály: egyező azonosítójú függvények esetében a **paraméterezésnek különbözőnek kell lennie**. Ez az egyértelműség miatt fontos.

Visszatérési érték alapján nem lehet egy függvényt túlterhelni. Azaz nem lehet több ugyanolyan nevű és paraméterlistájú, de más visszatérési típusú függvény.

A függvényhívásnál a paraméterezés dönti el, hogy melyik függvény-implementáció fusson le.

Leszármazott osztályban lehetőségünk van túlterhelni egy szülőosztálybeli metódust.

Taxi Felüldefiniálás III.

```
@Override
public double cost(double km) {
        return super.cost(km) + kmCost * km;
}

@Override
public void refuel(double liter) {
        fuel += liter;
        budget -= liter * fuelPrice;
}

@Override
public String toString() {
        return super.toString() + ", a fuvarozó pénze: "
        + budget;
}
```

Taxi új metódusok

Írhatunk a leszármazott osztályban új metódusokat:

Fuvarozást megvalósító függvény

Olyan **költség függvény**, ami egy főre megmondja az utazás költségét, ha egyszerre több utast fuvaroz a taxi.

(A `go()` függvényt ne definiáljuk felül, hiszen két fuvar között a taxi rendes autóként közlekedik.)

Taxi új metódusok

```
public double transfer(double km) {
        if (go(km) == km) {
                budget += cost(km);
                return km;
        } else {
                return 0;
        }
}

public double costPerPerson(double km, int num) throws Exception {
        if (num > capacity + 1) {
                throw new Exception("Több utas, mint férőhely");
        }
        return cost(km) / num;
}
```

Van egy új osztályunk!

Amit még lehetett volna: **elfedések**

- Szülőosztályban meglévő adattagokkal megegyező nevű adattagok deklarálása a leszármazott osztályban (elfedve azokat).
- Szülőosztályban lévő osztálymetódussal megegyező szignatúrájú osztálymetódus írása a leszármazott osztályban (itt ugyanazok a kitételek érvényesek, mint a metódus felüldefiniálás esetén).
- Ezekre ez esetben nem volt szükség.

Nézzük meg a `Taxi` osztály kódját!

Elfedés Példa

```
class Apple {
        protected static int kgPrice = 200;
        public static int getKgPrice() {
        return kgPrice;
    }
}

class GreenApple extends Apple {
    protected static int kgPrice = 500;
    public static int getKgPrice() {
        return kgPrice;
    }
}
```

`final` kulcsszó

Osztály esetén: olyan osztályt jelöl, ami végleges, tehát nem lehet már belőle örökölni.

- Ilyen lesz majd a **`Bus`** osztályunk.

Metódus esetén: olyan metódust jelöl ami végleges, tehát öröklődés során már nem definiálható felül.

- A **`Car`** osztály halad és benzinigény függvénye legyen ilyen, hiszen ez a függvény minden autó (legyen az taxi, busz stb.) esetén helytálló (kiszámítunk egy benzin igényt, és ha ez megvan, megtesszük az utat).

Adattag esetén: konstans.

Bus osztály

Hozzuk létre a `Bus` osztályt a `Taxi` osztályból!

A buszon egy **vonaljeggyel** lehet utazni, szintén **van kilométerdíj, amit a sofőr kap (sofőr fizetése)**. → új osztályváltozó: jegyár

A költség ugyanúgy kerül kiszámításra, ahogy a taxinál, ellenben a **fuvar esetében az üzemeltető cég pénztárcáját** tekintsük, azaz a jegyárból vonjuk ki a buszvezető díját is!

Az osztály legyen **végleges** (`final`).

Az alábbi metódusokat kell megvalósítani:

- `public Bus(String type, double consumption, int capacity, double kmCost)`
- `public double transfer(double km)` → felüldefiniálás
- `public double transfer(double km, int num)` → túlterhelés
- `public double profit(double km, int num)` → új metódus
- `public double costPerPerson(double km, int num)` → felüldefiniálás

Bus osztály megvalósítás I.

```
public final class Bus extends Taxi {
        private static int ticketPrice = 350;

        public Bus(String type, double consumption,
                        int capacity, double kmCost) {
                super(type, consumption, capacity, kmCost);
        }
        // egyéb metódusok
}
```

Bus osztály megvalósítás II.

```
// felüldefiniálás
@Override
public double transfer(double km) {
    if (go(km) == km) {
        budget += ticketprice - kmCost * km;
        return km;
    } else return 0;
}

// túlterhelt metódus
public double transfer(double km, int num) {
    if (go(km) == km) {
        budget += num * ticketprice - kmCost * km;
        return km;
    } else return 0;
}
```

Bus osztály megvalósítás III.

```
// többi felüldefiniált metódus
@Override
public double costPerPerson(double km, int num) {
    return ticketPrice;
}
@Override
public String toString() {
    return "A busz típusa: " + type + ", a vállalat penze: "
    + budget;
}
public double profit(double km, int num) {
    return num * ticketPrice - cost(km);
}
public static double getTicketPrice() {
    return ticketPrice;
}
```

Van egy újabb osztályunk

Kész van a `Bus` osztály. Nézzük meg a kódját a `vehicles` csomagban, és a `main` függvényt!

Eddig specializáltunk. Mi van ha generalizálni akarunk?

Absztrakt osztályok I.

A tervezés eszközei (generalizálást, azaz általánosítást tesznek lehetővé).

Az **altípus reláció megadása** tervezés szempontjából sokszor **megkívánja**.

Absztrakt osztály: **hiányosan definiált osztály**. → Vannak benne olyan funkciók, amelyekről még nem tudjuk, hogy hogyan fognak működni, amelyek megvalósítását a leszármazottakra bízzuk.

- **`abstract`** módosító szó (osztály, és a nem implementált metódus előtt).
- Azt az osztályt, amiben van legalább egy `abstract` módosítóval jelölt függvény, szintén `abstract` módosítóval kell ellátni.

Nem lehet belőle példányosítani, de lehet belőle örökölni.

Absztrakt osztályok II.

Megjegyzés: Absztrakt osztály leszármazottja lehet maga is absztrakt, ha nem implementálja az ősosztályban specifikált függvényeket, vagy ha önmaga is tartalmaz nem implementált függvényeket. **Absztrakt osztályt úgy is létrehozhatunk, hogy a szülő nem absztrakt**.

Tegyük fel, hogy szeretnénk az autó mellé egy újabb járművet definiálni, a biciklit.

Definiáljunk egy absztrakt `Vehicle` osztályt, majd ebből örököltessük le a `Car` és a `Bicycle` osztályainkat.

Most tehát a specializálással ellentétes folyamatról van szó, az általánosításról!

Absztrakt `Vehicle` osztály

```
public abstract class Vehicle {
    protected String type;
    protected double km;
    protected int capacity;

    public abstract double go(double km);

    @Override
    public String toString() {
        this.type = "";
        return type + ", " + km + " km, férőhely: " + capacity;
    }
}
```

Ezután ügyeljünk arra, hogy a `Car` osztályban már ne legyen a `type`, a `km` és a `capacity` újra deklarálva!

A `Bicycle` osztály

A `Bicycle` osztályból további értelmes járművet már nem tudunk leszármaztatni, ezért ez legyen final.

```
public final class Bicycle extends Vehicle
{
    public Bicycle (String type, int
 capacity, double km) {
        this.type = type;
        this.capacity = capacity;
        this.km = km;
    }

    public Bicycle (String type) {
        this(type, 1, 0);
    }
    @Override
    public double go(double km) {
        this.km += km;
        System.out.println("Megtettünk "
+ km + " km utat.");
        return km;
    }
}
```

Absztrakt osztályból leszármaztatva a leszármazott köteles implementálni az absztrakt osztályban specifikált függvényeket! (különben maga is absztrakt kell maradjon)

Polimorfizmus

A **polimorfizmus**, vagy más néven **többalakúság** az a jelenség, hogy **egy rögzített típusú változó több típusú objektumra hivatkozhat.**

Ez az ún. **altípusos polimorfizmus** (van másfajta is).

```
Car a = new Car("Subaru", 10, 4, 0);
Car b = new Taxi("Honda", 5, 5, 300);
```

Egy A típus altípusa egy B típusnak, ha minden olyan helyzetben, ahol B típusú objektum használható, A típusú objektum is használható.

Statikus típus, dinamikus típus

Statikus típus: a deklarációnál megadott típus.

Dinamikus típus: a változó által mutatott objektum tényleges típusa.

A dinamikus típus vagy maga a statikus típus, vagy egy leszármazottja (különben fordítási hibát kapunk). A statikus típus állandó (fordítási időben ismert), a dinamikus típus változhat a program futása során.

Helyes:

- `Car a = new Car("Subaru", 10, 4, 0);`
- `Car b = new Taxi("Honda", 5, 5, 300);`
- `Vehicle j = new Bicycle("Magellan");`

Attól még, hogy az absztrakt osztály nem példányosítható, mint típust használhatjuk!

Csak maga a példányosítás tiltott, nem írhatunk olyat, hogy:

- `Vehicle j = new Vehicle();` → hiba

Helytelen (hibát is kapunk rá):

- `Car a = new Bicycle("Magellan");`
- `Taxi b = new Car("Honda", 5, 5, 300);`

Statikus típus, dinamikus típus, szerepük

A statikus típus dönti el azt, hogy milyen műveletek végezhetők a referenciaváltozón keresztül.

A dinamikus típus dönti el azt, hogy a végrehajtandó metódusnak melyik törzse fusson le.

Ld. később: dinamikus kötés

- `Car a = new Car("Subaru", 10, 4, 0);`
- `Car b = new Taxi("Honda", 5, 5, 300);`
- `System.out.println(a.toString());`
- `System.out.println(b.toString());` // A `Taxi` `toString()`-je fog lefutni
- `b.transfer(100);` // hibát kapnánk, hiszen a `Car`-nak nincs `transfer` azonosítójú függvénye.

Hogyan tudnánk mégis ezek után elérni valahogy a dinamikus típusnak megfelelő műveleteket?

- Típuskonverzió – később

Dinamikus kötés

Dinamikus kötésnek nevezzük azt a jelenséget, amikor egy objektumon a statikus típus által megengedett műveletet meghívva, a műveletnek a dinamikus típusnak megfelelő implementációja fog lefutni.

- Ezt már láthattuk az előző dián.

A dinamikus kötés jelensége természetesen csak akkor jön létre, ha a leszármazott osztályban van felüldefiniált (és nem túlterhelt!) metódus.

Javaban minden metódus olyan, mint C++-ban a virtual kulcsszóval ellátottak. Itt a dinamikus kötés automatikusan létrejön!

```
Car a = new Car("Subaru", 10, 4, 0);
Car b = new Taxi("Honda", 5, 5, 300);
System.out.println(a.toString());
System.out.println(b.toString());
```

Az eredmény:

```
Subaru, benzin: 0.0, km: 0.0, férőhely: 4

Honda, benzin: 0.0, km: 0.0, férőhely: 5, a fuvarozo penze: 0.0
```

Implicit és explicit típuskonverziók – primitív típusok

Automatikus típuskonverzió (de nem mindig injektív):

- `byte -> short -> int -> long -> float -> double`

Mindig **bővítő**! (így biztonságos)

```
double d;
int i = 6;
d = i;
d = d + i;
```

Explicit típuskonverzió: cast

A programozó explicit megmondja, hogy milyen típusú legyen a változó: (ez így viszont már nem biztonságos, adatot veszthetünk)

```
double d;
int i = 6;
d = i;
i = (int) d + 4;
```

Implicit típuskonverziók – objektumok

Altípusok esetén:

Szűkebb halmazba tartozó objektumokat **bővebb halmazba** tartozó objektumokként kezelünk.

Tehát egy **leszármazott** osztályba tartozó objektumot az **ősosztályba** tartozóként kezelünk.

Ekkor **automatikus típuskonverzió** jön létre.

```
Car b = new Taxi("Honda", 4, 4, 300);
Taxi c = new Bus("Ikarus", 6, 30, 200);
```

Explicit típuskonverziók – objektumok I.

```
Car b = new Taxi("Honda", 4, 4, 300);
Taxi c = (Taxi) b;
c.transfer(100);
```

A fenti példában nem kapnánk hibát, hiszen a `b` dinamikus típusa `Taxi`, így átkonvertálhatjuk a referenciát gond nélkül.

Figyelem: a konverzió nem magát az objektumot érinti!

De ha elnézünk valamit, és a `b` változónk mégsem `Taxi` típusú objektumra mutat, akkor **`java.lang.ClassCastException`-t** kaphatunk.

Ha ezt ki akarjuk védeni, használjuk az **`instanceof`** operátort.

Explicit típuskonverziók – objektumok II.

Az **`instanceof`** operátor egy **megadott objektumot hasonlít egy megadott típushoz:**

```
if (b instanceof Taxi) {
        Taxi c = (Taxi) b;
        c.transfer(100);
}
```

Fontos: objektumok esetén cast-olásra akkor van szükségünk, ha egy olyan műveletet szeretnénk elérni az objektumon, ami a statikus típusban nincs definiálva. Egyéb esetekben a dinamikus kötés miatt úgyis a felüldefiniált metódus futna le.

A konverzió csak a referencia típusát érinti, magát az objektumot nem. Nem lehet objektum referenciák típusát akármivé átalakítani, csak az öröklődési láncban felfelé vagy lefelé.

Feladat

Nézzük meg ehhez a `dinkotes` csomagban a `Main` osztály `main` függvényét!

Fontos fogalmak – áttekintés

Osztályhierarchia

Konstruktorok és az öröklődés

Felüldefiniálás vs. túlterhelés

Elfedés

Absztrakt osztályok

Véglegesítés

Polimorfizmus

Statikus és dinamikus típus

Dinamikus kötés

Debugger

Debugger: hibakereső eszköz

Miért hasznos?

- Lehetőséget nyújt arra, hogy a programunkat lépésről lépésre futassuk, és közben láthassuk az egyes változóink pillanatnyi értékeit.
- Segítségével általában hatékonyan tudunk hibát keresni a programunkban.
- Amikor hibás a program, általános módszer, hogy egyes helyeken kiíratjuk a változóink értékét, vagy kiíratunk üzeneteket, hogy lássuk, hol keletkezett a hiba. Debugger alkalmazásakor ezt nem kell megtennünk, hiszen külön ablakon keresztül láthatjuk változóink aktuális értékeit, továbbá lépésenként végigkövethetjük, hogy éppen mely sorok végrehajtása történik.

Debugger

A debugger lényege, hogy előre elhelyeztt ún. **breakpoint**ok elérésekor a program végrehajtása szünetel. Ezután a programozó eldöntheti, hogy hogyan történjen a breakpoint által jelölt utasítás végrehajtása:

- **step into**: Az utasítás végrehajtása sorról sorra. Ha ezen a ponton egy függvényhívás van, akkor belépünk a függvénytörzsbe és lépésről lépésre végrehajtjuk az abban foglaltakat.
- **step over**: A jelölt sorban egy függvényhívás van, de nem lépünk be a függvénytörzsbe, hanem továbblépünk.
- **step return**: Ha egy függvényben vagyunk, a return utasításra ugrik.
- **run to line**: Általunk megadott sorig végrehajtja a kódot.

Egyéb lehetőségek:

- **watch pointok** elhelyezése: Egy adott változóra egy watch pointot állíthatunk. Ez esetben a debugger minden egyes alkalommal megáll, amikor a változót elérik, vagy megváltoztatják.

Példa

Vegyük az autó projektet:

Helyezzünk el tetszőleges sorok mentén breakpointokat:

- Az adott sor mentén a kód melletti bal oldali keskeny sávra dupla kattintás.

```
107     /**
108      * @param km
109      *            - double
110      * @return double - ténylegesen megtett kilométer
111      */
112     public final double go(double km) {
113         if (fuelNeed(km) <= fuel) {
114             fuel -= fuelNeed(km);
115             this.km += km;
116             System.out.println("Megtettünk " + km + " km utat!");
117             return km;
118         } else {
119             System.out.println("Nincs elég benzin " + km + " km úthoz!");
120             return 0;
121         }
122     }
123
```

Példa

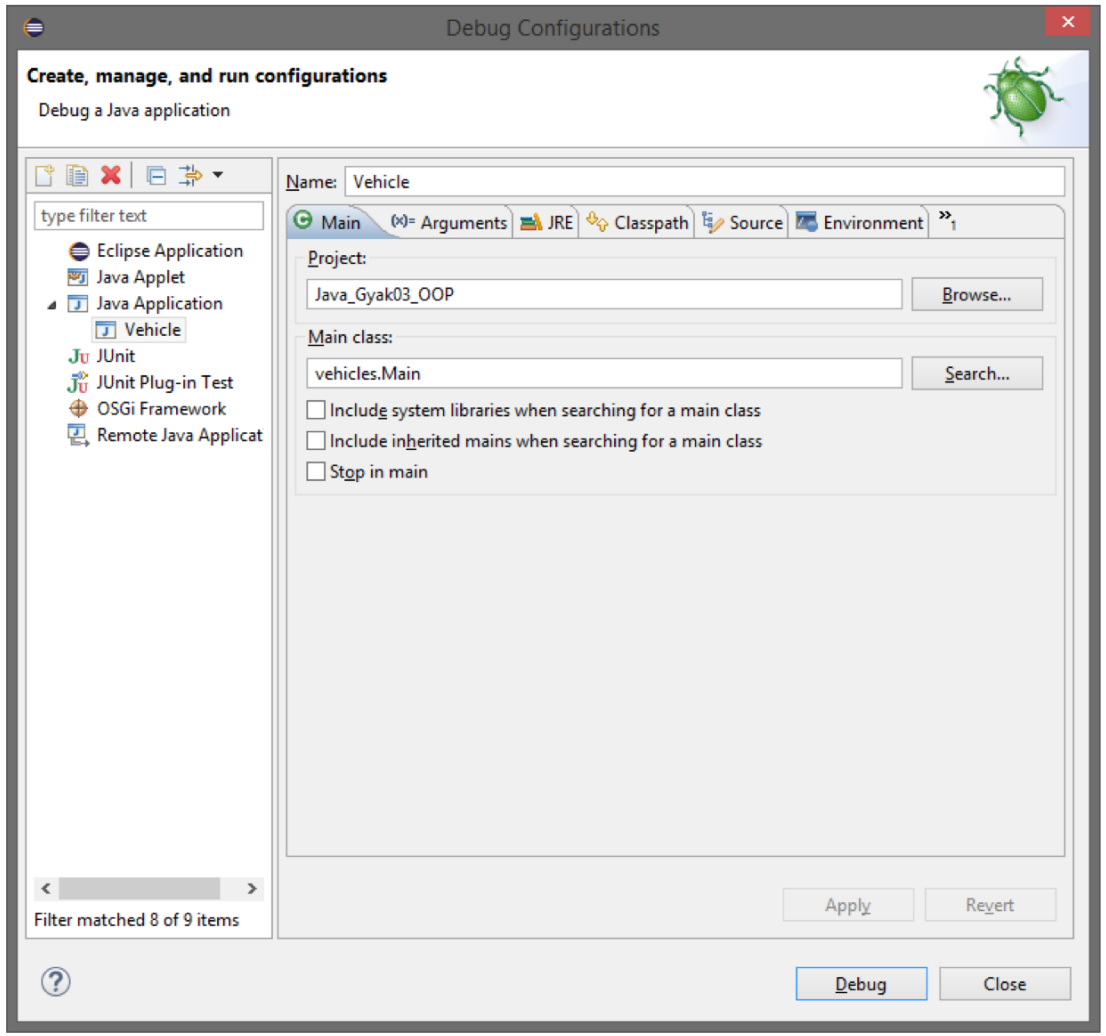
Ha ez megvan helyezzünk el az osztály adattagjaira watch pointokat (változó szintjében a keskeny sávra dupla kattintás).

```
3  /**
4   * @author sonil, divad Taxiból leszármazott Busz osztály, jegyár adataggal.
5   */
6  public final class Bus extends Taxi {
7
8      // új osztályváltozó
9      private static int ticketPrice = 350;
10
```

Példa

Majd indítsuk el a debuggert. (kisbogár ikon)

(Akár külön alkalmazásként is elindíthatjuk.)

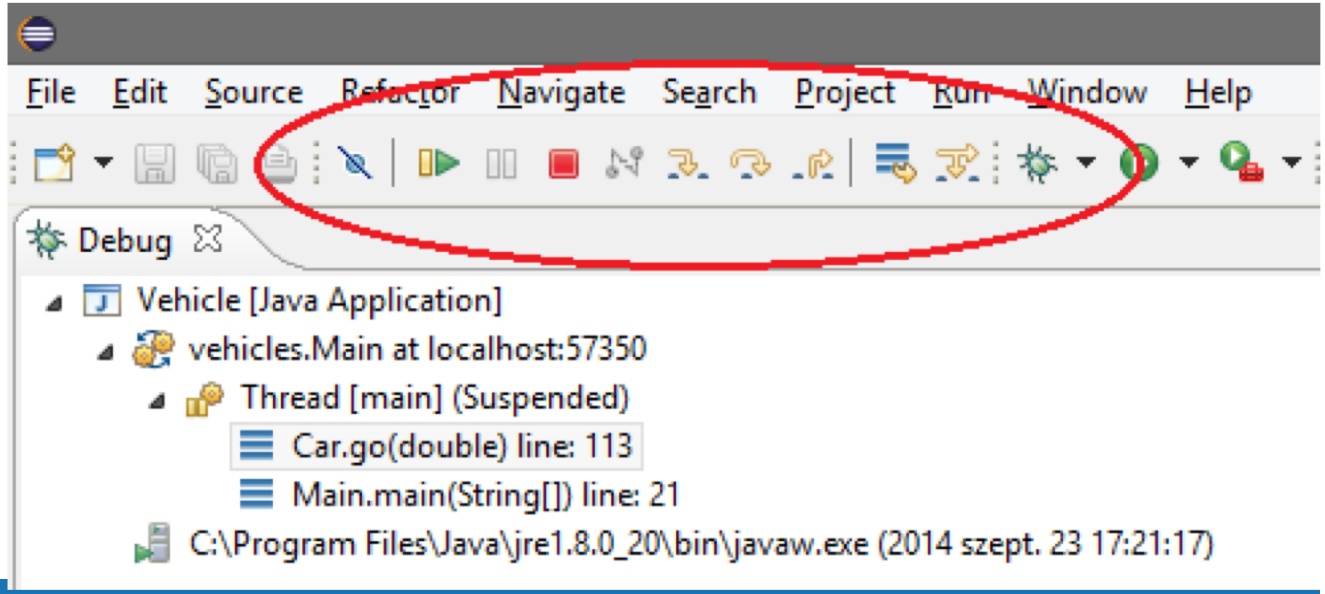


Példa

Mikor breakpoint-hoz érünk, próbáljuk ki a

- **step into**
- **step over**
- **step return**
- **run to line (jelöljük ki kurzorral a sort, ahova ugrani akarunk, majd Ctrl+R)**

lehetőségeket. Kiemelve láthatjuk, hogy éppen melyik sor végrehajtása történik.

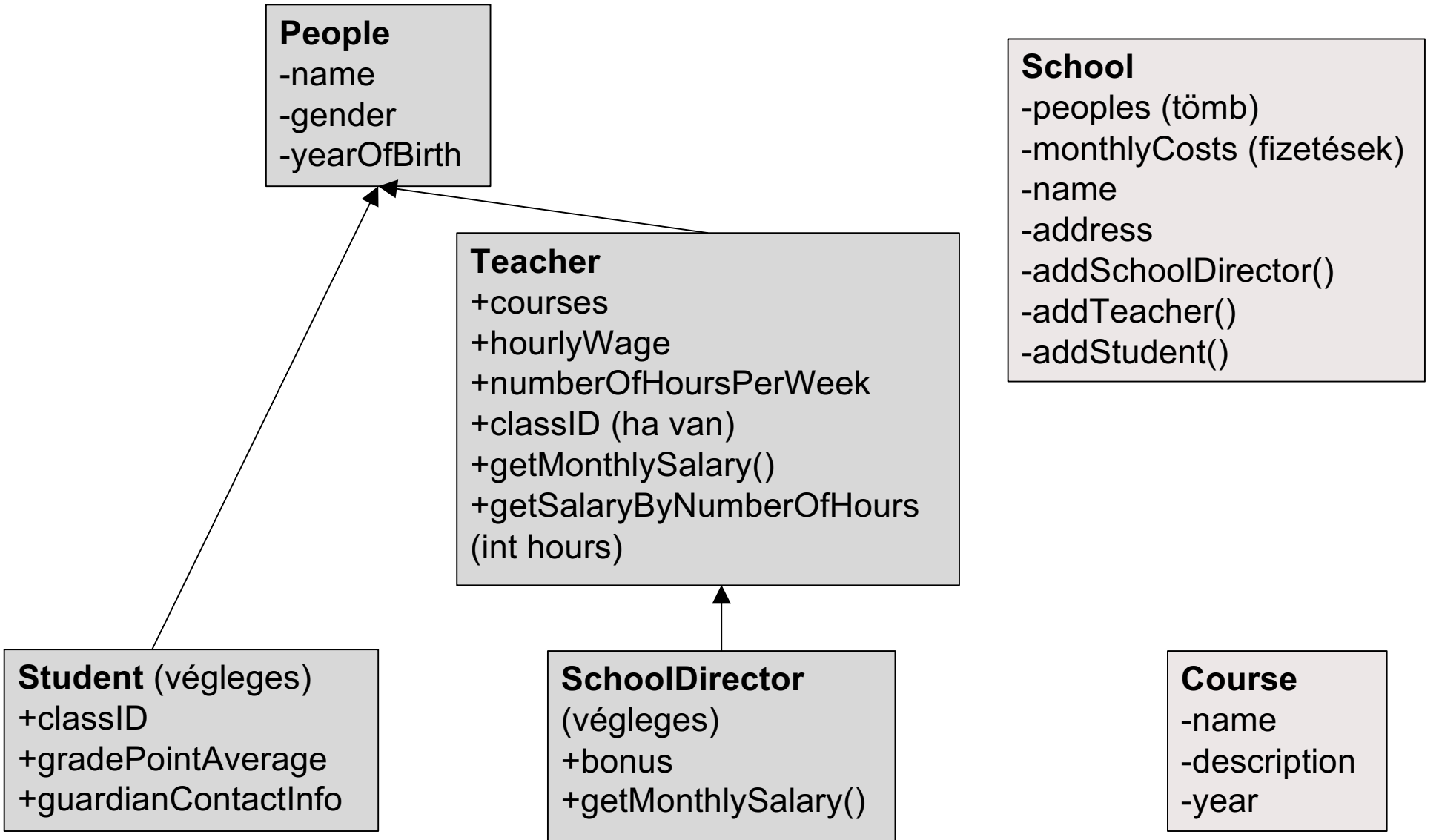


Gyakorló feladat G03F01

Iskola modell alkalmazás I.

Először is az iskolába járnak **emberek**, akiknek van nevük, születési évük és nemük.
Az iskolában tanítanak **tantárgyakat**, amelyeknek van nevük, leírásuk és hozzá vannak rendelve évfolyamok, amikor ezeket oktatni kell.
Az iskolába járó emberek között vannak **diákok**, akik egy osztályba tartoznak, akiknek van egy tanulmányi átlaguk, és akiknél el kell tárolni a szülő/nevelő elérhetőségeit.
Az iskolában tanítanak **tanárok**, akikhez bizonyos tárgyak vannak hozzárendelve. A tanárok között vannak osztályfőnökök, akiknek van osztályuk. A tanároknak van heti óraszámuk, órabérük ami alapján a havi fizetésük kiszámolható.
A tanárok között van **egy igazgató**, akinek a fizetése nem csak az órabér és a heti óraszám alapján képződik (van valamifajta plusz fizetés).
Végül van az **iskola**, amiben vannak tanulók, tanárok, igazgató, aminek van egy neve, címe és ahol kiszámolható, hogy havonta összesen mennyi fizetést kell kiosztani.

Gyakorló feladat G03F01



Gyakorló feladat G03F01

Végül írj egy **`Main`** osztályt, melyen keresztül bemutatod az iskolát (iskola adatai, tanári kar, diákok).

- **Tipp**: Az iskola `toString()` függvényét terheld túl a megfelelő módon, majd írasd ki az iskola példányát a konzolra. Az emberek listázásánál NE csak a név jelenjen meg, hanem a megfelelő osztály `toString()` függvényének eredménye. (látszódjon tehát, hogy ki tanár, diák, igazgató, minden adat…)

Az iskola adatainak kiíratása után írasd ki a tanárok havi fizetését (összes kiadás).
Adj hozzá még diákokat, tanárokat (és igazgatót is!) az iskola nyilvántartásához.
Kérdezd le újra az iskola havi kiadásait (1 hónap = 4 hét).
Tehát nagy vonalakban mutasd be az általad létrehozott osztályok funkcióit!
Az előbbi dián listázott metódusokon kívül természetesen, ha indokolt, legyenek implementálva **getter** és **setter** metódusok is, továbbá egyéb olyan (segéd) metódusok, amelyekkel az objektum állapotait tudjuk megváltoztatni, vagy egyéb okból szükségesek.

Gyakorló feladat G03F02

Feladat egy gráf osztály megírása, melyben az adatelemek/csúcsok `Stringek`.

- Egy gráf egy olyan adatszerkezet, melyben a tárolt adatelemek között sok-sok kapcsolat(él) van. Nincs semmilyen megkötés az adatelemek közötti kapcsolatokra.
- A gráfba maximum 10 adatelemet/csúcsot lehet felvenni.
- A kapcsolatokat az adatelemek között egy incidencia mátrix-szal reprezentáld.

A gráfnak a következő metódusai legyenek:

- Legyen paraméter nélküli konstruktora, és legyen olyan is, ami egy `String` tömböt vár, és a tömb elemeivel egyből feltölti a gráfot.
- Le lehessen kérni az aktuális méretét (hány adatelem van benne tárolva).
- Le lehessen kérni a max. méretét (max. hány adatelemet lehet belerakni).
- Lehessen új adatelemet felvenni.
- Lehessen két csúcs között kapcsolatot létrehozni (ha a paraméterben megadott adatelemek még nincsenek benne a gráfban, akkor automatikusan vétessenek fel).
- Lehessen két csúcs között megnézni, hogy van-e kapcsolat.
- Lehessen csúcsot törölni.
- Ki lehessen listázni egy adatelem esetén minden olyan adatelemet, amivel kapcsolatban van.
- Legyen benne megfelelő kivételkezelés!

Írj a gráf osztályba egy útkereső függvényt, ami két tetszőleges csúcs esetén megállapítja, hogy van-e köztük út.

Gyakorló feladat G03F03

Valósítsd meg a következőkben leírt osztályszerkezetet, és írj egy programot ami demonstrálja a képességeit!

- Hozzunk létre egy gerinces osztályt, amit ne lehessen példányosítani
- A gerinceseknél tároljuk el, és legyen kiírható a csigolyák száma (a porcos halaktól most tekintsünk el.), és a lábak száma; illetve valósítsuk meg a `halad` függvényt, ami kiírja haladásuk módját.
- Minden állatnak legyen neve, amit ne lehessen átkeresztelni futás alatt.
- A nevezéktant alaposan leegyszerűsítve hozzuk létre a `Hal`, a `Keteltu`, a `Hullo`, a `Madar` és az `Emlos` osztályt. Mindet lehessen példányosítani.
- Tároljuk el és tegyük lekérdezhetővé:
  - A halaknál az uszonyok számát; a lábak száma 0 legyen mindig. A hal úszik.
  - A kétéltűeknél azt, hogy megeszik-e a franciák – a kiíratásnál ezt egész mondatban tegye. A kétéltű úszik és megy.
  - A hüllőknél, hogy mérgezőek-e – a kiíratásnál ezt egész mondatban tegye. A hüllő csendesen siklik.
  - A madarak esetében a szárnyak számát is. A madár repül (pingvinek off)
  - Az emlősök esetén a fogak számát. Az emlős mászik.
- Az emlősökből hozzuk létre a `Kutya` és `Macska` osztályt, amelyeket nem lehet örököltetni, és a ahol
  - A kutyáknál tároljuk el, hogy ugat-e, és hozzá egy függvényt, hogy ennek alapján harap-e. A kutya farokcsóválva szalad.
  - A macskáknál tároljuk el azt is szövegként, hogy milyen színű a szőre. A macska sündörög.

Gyakorló feladat G03F04

Készítsd el a `Sikidom` absztrakt osztályt. Tulajdonságai: `kerulet`, `terulet`. Műveletei: `kerulet`, `terulet`, `toString` – ezek legyenek publikusak; Absztrakt műveletei: `keruletSzamit`, `teruletSzamit` – legyenek leszármazottak számára kezelhetőek.

Valósítsd meg hogy a `kerulet` és a `terulet` függvények adják visszatérési értékül a `kerulet`-et illetve a `terulet`-et, amelyek a `keruletSzamit` illetve a teruletSzamit értékeivel bírnak. A `toString`-be irassuk ki a teruletet és a kerületet.

A `Sikidom` leszármazottja a `Kor` osztály, aminek sugar a mezője. Hozzuk létre az osztály egyparaméteres konstruktorát, definiáljuk a `keruletSzamit` és a `teruletSzamit` (a sugár alapján), `toString` metódusokat. A `toString` írja le az osztály állapotát: miről van szó, sugár, területe, kerülete.

Gyakorló feladat G03F04

A `Sikidom` másik leszármazottja a `Trapez` osztály, 4 paraméteres konstruktora van. Ezért 4 mezővel bír, azaz a,b,c,d oldalak vannak (leszármozottai lesznek, figyeljünk a legjobb láthatóságra). Írjuk meg erre is a `keruletSzamit` és a `teruletSzamit` metódusokat az oldalak alapján. A `toString` írja le ismét az osztályt-t(mezők, terület, kerület).

Hozzuk létre a `Paralelogramma` osztályt , ami egy speciális trapéz. Mezői bővüljenek a szog-el. 3 paraméteres konstruktora van: a,b oldal és a szög. A szög segítségével határzozzuk meg a terültet. Alkossuk meg ismételten a `keruletSzamit`, `teruletSzamit`, `toString` függvényeket().

Készítsünk `Teglalap` osztályt, ami a `Paralelogramma` gyermeke. 2 paraméteres konstruktora van. Minimalizáljuk a megírandó függvényeket, tudjuk leírni a fentiekhez hasonlóan az osztályt.

Valósítsuk meg a `Negyzet` osztályt, aminek egyparaméteres konstruktora van. Cselekedjünk az előbbi szerint.

Gyakorló feladat G03F05

A feladat áruházak ‘modellezése’. Először hozz létre egy áru osztályt a következő tagváltozókkal: id, név, mennyiség (a raktárban), egységár. Hozd létre az ezeket fogadó konstruktort az osztályhoz.

Minden áru áfája megegyezik (20%) – a megfelelő nyelvi elemmel jelöld ezt.

Hozz létre egy függvényt, mely visszaadja az adott áruból a raktárban található mennyiség értékét.

Készíts egy statikus függvényt, mely egy adott bemeneti (nettó) értékre megmondja, mennyi az adóval terhelt (bruttó) ára
Készítsd el az `Üzlet` osztályt. Egy üzletnek van neve, a benne található árukat pedig egy előre meghatározott méretű (30) statikus [nem `static`!], `Áru` típusú tömbben tároljuk.

Az üzlethez tartozzon egy vásárlás függvény – bemenete egy áru azonosítója (`id`), illetve egy mennyeiség, míg kimenete a kívánt mennyiség ára (természetesen a megvásárolt terméket vond ki a raktárkészletből).

Gyakorló feladat G03F05

Hozz létre egy leszármazottat az `Exception` osztályból, majd ezzel jelezd, hogy ha olyan terméket próbálunk vásárolni, mely nincs raktáron.
A raktárat egy `Áru` típust fogadó függvénnyel és konzolon keresztül is lehessen feltölteni, azaz kérje be az áru adatait, majd mentse el a raktárban.
Amennyiben már létező azonosítót adunk meg, úgy a már eltárolt áru adatait (pl mennyiségét) módosítsa (azaz ne adja hozzá másodjára)
Az aktuális raktárkészletet konzolon keresztül lehessen kilistázni

Készítsd el az `Áru` leszármazottjaként az `Élelmiszer` osztályt. Új adattag a lejárati idő (stringként tárolva, később lesz szó a Java dátumformátumról), továbbá áfája eltérő (12%).

Az `Üzletből` leszármazva hozd létre a hipermarket osztályt, mely tartalmaz egy második, `Élelmiszer` típusú (30 elemszámú) tömböt is. Ennek megfelelően adj hozzá egy második, az élelmiszereket konzolról feltöltő függvényt is.

A `Hipermarket` a raktár kiíratását már fájlba végzi, melyben megjelennek a hagyományos áruk és élelmiszerek is, továbbá kiírja minden tétel nettó és bruttó értékét is.

Gyakorló feladat G03F05

Készíts egy futtató osztályt, mely létrehoz egy áruházat és egy hipermarketet, feltölti néhány áruval, majd kilistázza a raktárak tartalmát. Próbáld ki a konzolról beolvasást is.

A változókat a megfelelő módon rejtsd el, majd biztosítsd hozzájuk legalább az elérési függvényeket (amelyikhez szükséges, ahhoz a módosítót is)

Minden osztályod rendelkezzen releváns `toString()` metódussal

A futtató osztályt érdemes a többi osztállyal együtt fejleszteni, hogy ne csak a végén derüljenek ki a hibák.

Gyakorló feladat G03F06

A feladat egy egyetemi kurzus szimulálása.

A kurzus 14 hétből áll:

- minden héten elméleti kisZH, és gyakorlati házi feladat (ez utóbbit a következő héten értékelik),
- kivéve a 7. és a 14. héten, mert akkor nagyZH (a 6. héten feladott házit a 8. héten értékelik, a 13. héten pedig nincs házi),
- illetve az első héten, ekkor nem adnak fel házit, és nem íratnak kisZH-t.

A kurzuson 25 tanuló vesz részt:

- ezek közül néhányan "elméleti" és néhányan "gyakorlati" beállítottságúak
- ők rendre a kisZHkon, illetve a házikon teljesítenek jobban; a nagyZH-n egyformán teljesítenek.
- Ezeket a tanulókat generáld le objektumokként!
- Szimuláld továbbá egy olyan gyakorlatvezető teljesítményét, aki ellenőrzi a számonkéréseket, azokat majdnem hibátlanra írja meg, és az ő teljesítményével skálázd a végén az eredményt!

Gyakorló feladat G03F06

Mindenki véletlenszerűen teljesít, viszont a várható érték a fentiek szerint kell alakuljon.

A konzolra készíts táblázatos formában heti és féléves statisztikákat!

A feladat során nem használhatsz elágazást (és ebbe beleértjük az if-et, a switch-et és a nyilvánvalóan elágazás kiváltására használt feltételes ciklusokat), ellenben használd fel a tanult OOP technikákat!