Java tutorial

Objektum-elvű programozás (OOP)

- Objektumok, osztályok (adatközpontú)
- Eseményvezérelt programozás
  - vs. strukturált programozás
  - deklaratív / *imperatív*
- Adatabsztrakció (egységbe zárás, adatelrejtés)
- Polimorfizmus
- Öröklődés
- Dinamikus kötés

Objektumok

- (Program) entitás: állapot és funkcionalitás
- egyedi, zárt (interfész), konzisztencia

- Attribútumok, események
- Attribútum - változó (adattag)
- Eseménykezelő - alprogram (metódus)

Példa objektumokra

Kör

x-koord: 0
y-coord: 0
Sugár: 1 egység
Terület: 3.1415926...

Nagyít
Kicsinyít
Eltol

Alkalmazott

Név: Gipsz Jakab
Munkahely: ELTE
Fizetés: 200e Ft

FizetéstEmel
FeladatotAd

Példa objektumokra

Kör

Középpont: Origó
Sugár: 1 egység
Terület: 3.1415926...

Nagyít
Kicsinyít
Eltol

Alkalmazott

Név: Gipsz Jakab
Munkahely: ELTE
Fizetés: 200e Ft

FizetéstEmel
FeladatotAd

Példa objektumokra

Kör

Középpont: Origó
Sugár: 1 egység
Terület: 3.1415926...

Nagyít
Kicsinyít
Eltol

Alkalmazott

Név: Gipsz Jakab
Beosztás: tanársegéd
Fizetés: 200e Ft

FizetéstEmel
FeladatotAd

Kapcsolatok objektumok között

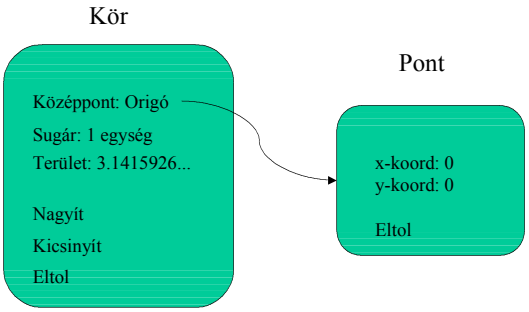
- **Osztály**: hasonló objektumok gyűjteménye
  - **Struktúrális hasonlóság (reprezentáció)**
  - Funkcionális hasonlóság (viselkedés)

  Típus: típusértékek halmaza

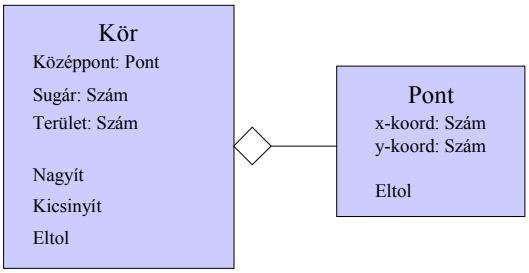
  Példányosítás (osztály ⇨ objektum)

  Relációk: is-a, has-a
- Aggregációk, asszociációk

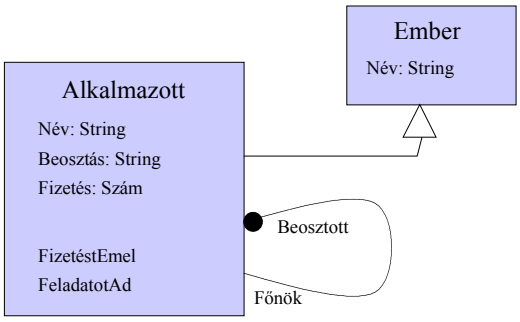
Példa objektumok kapcsolatára



Példa: osztályok és kapcsolataik



Példa: osztályok és kapcsolataik (2)



Osztályok, objektumok a Java nyelvben

Osztály = séma:

objektumok reprezentációjának megadása

Objektum: egy osztály egy példánya

minden objektum valamilyen osztályból származik példányosítással

Reprezentáció:

példány adattagok, példány metódusok

Osztály: típus

Java tutorial

Az Alkalmazott osztály

```
public class Alkalmazott {

  String név;
  String beosztás;
  int fizetés;

  void fizetéstEmel( int mennyivel ){
    fizetés += mennyivel;
  }

}
```

Alkalmazott.java

Az Alkalmazott osztály

```
public class Alkalmazott {

  String név;
  String beosztás;
  int fizetés;

  void fizetéstEmel( int mennyivel ){
    fizetés += mennyivel;
  }

}
```

Alkalmazott.java

Az Alkalmazott osztály

```
public class Alkalmazott {

  String név;
  String beosztás;
  int fizetés;

  void fizetéstEmel( int mennyivel ){
    fizetés += mennyivel;
  }

  int szobaszám;
}
```

Alkalmazott.java

```
javac Alkalmazott.java
java Alkalmazott
```

Java tutorial

Főprogram

```
public class Alkalmazott { ... }
```

Alkalmazott.java

```
public class Program {
  public static void main( String[] args ){
    Alkalmazott a;
    a = new Alkalmazott();
  }
}
```

Program.java

```
javac Program.java
java Program
```

Feladat

- Készítsd el a Pont osztályt!
- Tulajdonságok: x és y koordináta
- Művelet: eltolás
  - dx és dy értékekkel

Főprogram

```
public class Alkalmazott { ... }
```

Alkalmazott.java

```
public class Program {
  public static void main( String[] args ){
    Alkalmazott a;
    a = new Alkalmazott();
  }
}
```

Program.java

javac Program.java
java Program

Főprogram

```
public class Alkalmazott { ... }
```

Alkalmazott.java

```
public class Program {
  public static void main( String[] args ){
    Alkalmazott a = new Alkalmazott();
  }
}
```

Program.java

javac Program.java
java Program

Főprogram

```
public class Alkalmazott { ... }
```

Alkalmazott.java

```
public class Program {
  public static void main( String[] args ){
    new Alkalmazott();
  }
}
```

Program.java

javac Program.java
java Program

Objektumok tárolása

- Dinamikus memóriakezelés szükséges
- Ada, C: mutatók (pointerek)
- Java: referenciák

  `Alkalmazott a;`

  Az `a` változóban az objektum memóriabeli címét tároljuk. A deklaráció hatására nem jön létre objektum!

Objektum létrehozása

- Az `a` változóhoz objektum hozzárendelése

  `a = new Alkalmazott();`
- Példányosítás: valamilyen osztályból a **`new`** operátorral (memóriafoglalás a mellékhatás, a kezdőcím a kifejezés értéke)

  `new Alkalmazott()`
- Az **`a`** referencia a **`new`** operátorral létrehozott “objektumra mutat”

Java tutorial

Feladat

- Készíts főprogramot a Pont osztályhoz. Hozz létre benne egy Pont objektumot.

Java tutorial

Metódus meghívása

```
public class Alkalmazott {

  ...

  void fizetéstEmel( ... ){ ... }
}
```

```
public class Program {
  public static void main( String[] args ){
    Alkalmazott a = new Alkalmazott();
    a.fizetéstEmel(40000);
  }
}
```

Metódus meghívása

```
public class Alkalmazott {

  ...

  public void fizetéstEmel( ... ){ ... }
}
```

```
public class Program {
  public static void main( String[] args ){
    Alkalmazott a = new Alkalmazott();
    a.fizetéstEmel(40000);
  }
}
```

Adattag elérése

```
public class Alkalmazott {
  ...
  int fizetés;

  public void fizetéstEmel( ... ){ ... }
}
```

```
public class Program {
  public static void main( String[] args ){
    Alkalmazott a = new Alkalmazott();
    a.fizetés = 200000;
    a.fizetéstEmel(40000);
  }
}
```

Adattag elérése

```
public class Alkalmazott {
  ...
  public int fizetés;

  public void fizetéstEmel( ... ){ ... }
}
```

```
public class Program {
  public static void main( String[] args ){
    Alkalmazott a = new Alkalmazott();
    a.fizetés = 200000;
    a.fizetéstEmel(40000);
  }
}
```

Feladat

- Állítsd be a létrehozott Pont koordinátáit, told el a definiált metódussal, végül írd ki a képernyőre a koordináták új értékét.

Adattagok definiálása

- Adattag = példányváltozó
- Adattag megadása - változódeklaráció

```
public class Alkalmazott {
  ...
  int fizetés;
  public void fizetéstBeállít( ... ){...}
  public void fizetéstEmel( ... ){ ... }
}
```

Java tutorial

Adattagok inicializálása

```
public class Alkalmazott {
  ...
  int fizetés = 200000;
  public void fizetéstBeállít( ... ){...}
  public void fizetéstEmel( ... ){ ... }
}
```

```
public class Program {
  public static void main( String[] args ){
    Alkalmazott a = new Alkalmazott();
    a.fizetéstEmel(40000);
  }
}
```

Adattagok automatikus inicializálása: példa

```
public class Alkalmazott {
  ...
  int fizetés;
  public void fizetéstBeállít( ... ){...}
  public void fizetéstEmel( ... ){ ... }
}
```

Olyan, mint: `int fizetés = 0;`

```
public class Program {
  public static void main( String[] args ){
    Alkalmazott a = new Alkalmazott();
    a.fizetéstEmel(40000);
  }
}
```

Adattagok automatikus inicializálása: implicit kezdőérték

- Példányváltozók esetén történik
  - alprogram lokális változójára nincs (fordítási hibát okoz, ha előzetes értékadás nélkül próbáljuk használni!)
  - példányváltozók esetén nehéz lenne betartani ezt a szabályt, ezért inicializál automatikusan
- Pl. szám típusok esetén nulla (`0` vagy `0.0`), boolean esetén `false`, char esetén `\u0000`
- Nem illik kihasználni!

Feladat

- A Pont osztályban az x és y adattagokat explicit inicializáld 0-ra!

Java tutorial

Adattagok definiálása: példák

int fizetés=200000, pótlékok, levonások=fizetés/4;

Láthatóság változtatása:

public int fizetés = 200000;

Nem módosítható értékű változók (“konstansok”)

final double ADÓKULCS = 25;

Hivatkozás példányváltozókra

```
public class Alkalmazott {
  public int fizetés;
  public void fizetéstEmel( int mennyivel ){
    fizetés += mennyivel;
  }
  public boolean többetKeresMint(Alkalmazott másik){
    return fizetés > másik.fizetés;
  }
}
```

Osztálydefiníción belül

„saját” adattagra

más objektum adattagjára

```
public class Program {
  public static void main( String[] args ){
    Alkalmazott a = new Alkalmazott();
    a.fizetés = 200000;
  }
}
```

Minősített név

Más osztályban

Metódusok

- Metódus: alprogram, amely egy objektumhoz van kapcsolva
- Az „első paraméter” az objektum
- Például Javában `a.többetKeresMint(b)`
  Adában `többetKeresMint(a,b)`
- Üzenetküldéses szintaxis
- Mellékhatásos függvény
  - eljárás: **void** visszatérési érték

Metódusok definíciója

- Fejből (specifikációból) és törzsből áll

```
public void fizetéstEmel( int mennyivel ){
    fizetés += mennyivel;
}
```

- Fej: módosítók, a visszatérési érték típusa, azonosító név, paraméterlista, ellenőrzött kivételek
- Paraméterlista: `(int x, int y, char c)`
- Szignatúra: azonosító, paraméterek típusa

```
public void fizetéstEmel( int mennyivel ){
    fizetés += mennyivel;
}
```

Példák metódusdefiníciókra

```
int lnko( int a, int b ){
  while( a!=b )
    if( a>b ) a-=b; else b-=a;
  return a;
}
```

Alaptípusokra érték szerinti paraméterátadás

```
public void fizetéstDupláz(){
  fizetés *= 2;
}
```

Üres paraméterlista

Kilépés metódusból

- Ha nem **`void`** a visszatérési érték típusa, akkor kell **`return`** utasítás, amivel megadjuk a visszatérési értéket
- Ha nincs visszatérési érték (**`void`**), akkor is lehet **`return`** utasítás, amivel kiléphetünk

```
void f(...){
  while(...){
    ...
    if(...) return;
  }
}
```


Kilépés metódusból

- Ha nem **`void`** a visszatérési érték típusa, akkor kell **`return`** utasítás, amivel megadjuk a visszatérési értéket
- Ha nincs visszatérési érték (**`void`**), akkor is lehet **`return`** utasítás, amivel kiléphetünk


```
void f(...){
  while(...){
    ...
    if(...) break;
  }
}
```

Vezérlésmegszakító utasítások

- **`continue`** - kilép a ciklusmagból
- **`break`** - kilép a ciklusból
- **`return`** - kilép a metódusból

A visszatérési érték megadása

- A fordító ellenőrzi, hogy "függvény" esetén mindenféleképp lesz visszatérési érték, azaz a vezérlés mindig eljut egy return utasításhoz
- Hasonlóan a metódusok lokális változói kapcsán végzett ellenőrzéshez (inicializáltság)
- Fordítási idejű döntés

Java tutorial

Feladat

- Készítsd el a Kör osztályt!
- Tulajdonságok: középpont (Pont) és sugár (double)
- Műveletek: eltol és nagyít
  - eltol: dx és dy értékekkel a középpontot kell eltolni...
  - nagyít: a sugarat szorozni faktor-ral

Java tutorial

Közvetlen adathozzáférés megtiltása

```
public class Alkalmazott {
  ...
  int fizetés;
  public void fizetéstBeállít( int összeg ){
    fizetés = összeg;
  }
  public void fizetéstEmel( ... ){ ... }
}
```

```
public class Program {
  public static void main( String[] args ){
    Alkalmazott a = new Alkalmazott();
    a.fizetéstBeállít(200000);
    a.fizetéstEmel(40000);
  }
}
```

Típusinvariáns megőrzése

```
public class Alkalmazott {
  double fizetés, évesFizetés;
  public void fizetéstBeállít(int új){
    fizetés = új;
    évesFizetés = 12*fizetés;
  }
}
```

Feladat

- A Kör osztályban vezess be egy új attribútumot, a területet tartsuk benne nyilván. Írd meg a sugaratBeállít műveletet!
- Használd a Math.PI értéket...
- A műveletek legyenek publikusak, az attribútumok ne!
- Készíts lekérdező műveleteket a sugárhoz és a területhez, melyek publikusak. (Az adattagok nem publikusak!) A lekérdező műveletek neve megegyezhet a lekérdezett attribútum nevével.

Típusinvariáns megőrzése

```
public class Kör {
  double sugár, terület;
  public void sugaratBeállít(double r){
    sugár = r;
    terület = sugár*sugár*Math.PI;
  }
}
```

Referenciák ráállítása egy objektumra

- Referencia és objektum együttes létrehozása
  `Alkalmazott a = new Alkalmazott();`
- Referencia ráállítása meglévő objektumra
  `Alkalmazott b = a;`
  A két referencia ugyanarra az objektumra mutat.
  `b.fizetéstEmel(10000);`

Feladat

- Próbáld ki a Pont osztály egy objektumával!

Java tutorial

Üres referencia

- Ha egy változó értéke **`null`**, akkor nem mutat objektumra.
  `Alkalmazott c = null;`
- A **`null`** referencia minden osztályhoz használható.
- Példányváltozók automatikus inicializálásához ezt használja a Java
- **`c.fizetéstEmel(10000);`**
  futási idejű hiba:
  `NullPointerException`

Nem változtatható referencia

```
final Alkalmazott a = new Alkalmazott();
a.fizetéstBeállít(100000);
a = new Alkalmazott();
```

A referencia "konstans", nem lehet másik objektumra állítani, de a mutatott objektum megváltozhat.

Összetett típusok

- Összetett értéket csak objektummal lehet létrehozni: az egyetlen típuskonstrukció
- Minden összetett érték dinamikus
- Minden összetett értékre referencián keresztül lehet hozzáférni
- Pl. a tömbök is (speciális predefinit) osztályok

Objektum paraméterként

```
public boolean többetKeresMint(Alkalmazott másik){
    return fizetés > másik.fizetés;
}
public void keressenAnnyitMint(Alkalmazott másik){
    másik.fizetéstBeállít(fizetés);
}
```

Főprogramban: `a.keressenAnnyitMint(b);`

- Az objektumreferencia érték szerint adódik át: **referencia szerinti paraméterátadás**
- Olyan, mint a C-ben a cím szerinti

Mellékhatás

- A metódusoknak lehet mellékhatása
  - az objektumon, amihez tartozik
  - globális objektumokon (System.out)
  - paraméterként átadott objektumokon

```
public class Program {
    public static void fizetéstEmel
    ( Alkalmazott a, int mennyivel ){
          a.fizetéstEmel(mennyivel);
    }
    ...
}
```

Feladat

- A Pont és a Kör osztályokhoz készítsd el a távolság() metódust, mely megadja az objektum távolságát egy, a paramétereként átadott Pont objektumtól.
- A Kör osztályban definiálj példánymetódust, mely a paramétereként átadott pont objektumot a kör objektum középpontjába állítja:

```
public void középpontba( Pont p )
```

Az objektum élettartama

- Nincs olyan utasítás, amivel objektumot explicit módon meg lehet szüntetni
- A nyelv biztonságosságát növeli
- Szemétgyűjtés: ha már nem hivatkoznak egy objektumra, akkor azt meg lehet szüntetni.

```
Alkalmazott a = new Alkalmazott();
a = null;
```

- Nem biztos, hogy megszűnik a program vége előtt

Szemétgyűjtés

- Modern nyelvekben gyakori
- Biztonságosság
  - többszörös hivatkozás esetén: ha az egyik hivatkozáson keresztül megszüntetjük az objektumot, egy másikon keresztül meg továbbra is használni próbáljuk
- Hatékonyság: idő és tár
- Ciklikus hivatkozás
- Szemétgyűjtő algoritmusok

Java tutorial

Osztályszintű változók és metódusok

- A tyúk és a tojás: programot csak objektumhoz tudunk készíteni (metódust), objektumot pedig csak a program futása közben tudunk létrehozni.

```
class hello {
  public static void main( String[] args ){
     System.out.println("hello");
  }
}
```

Osztályszintű változók és metódusok 2

- Az objektumoknak vannak attribútumai és műveletei, amiket az osztálydefinícióban adunk meg: példányváltozók és (példány)metódusok.

```
public class Alkalmazott {
  int fizetés = 0;
  public void fizetéstEmel( int mennyivel ){ ... }
}
```

Osztályszintű változók és metódusok 3

- Az osztályoknak is lehetnek: osztályszintű változók és osztályszintű metódusok. Ezeket is az osztálydefinícióban adjuk meg, de a static módosítószóval:

```
static int nyugdíjKorhatár = 60;
public static void nyugdíjKorhatártEmel( int
mennyivel ){
   nyugdíjKorhatár += mennyivel;
}

int életkor;
```

Java tutorial

Osztályszintű változók és metódusok 4

- A példányváltozók minden objektumhoz (az osztály minden példányához) létrejönnek, az osztályszintű változók az osztályhoz jönnek létre; azaz csak egy van, és minden objektum osztozik rajta.
- A példánymetódusok egy objektumhoz tartoznak: az első (kvázi implicit) paraméter az objektum; osztályszintű metódusok az osztály műveletei, az "első paraméter" az osztály.
- Az osztályok is picit olyanok, mint az objektumok. (Azzal, hogy vannak attribútumaik és eseménykezelőik.)

Osztályszintű változók és metódusok 5

- Az osztályszintű változók:
  - kifejezhetik, hogy ugyanaz az attribútumérték van minden objektumhoz (nyugdíjKorhatár)
  - az osztály állapotát rögzíthetik (kávépénz)

```
static int kávépénz = 0;
public int kávéraBefizet(){
  fizetés -= 100;
  kávépénz += 100;
}
static public kávétVesz(){ ... }
```

Osztályszintű változók és metódusok 6

- A static dolgokhoz nem kell objektum; a főprogram is ilyen volt.
- A procedurális programozás imitálása:
  - osztály - modul
  - osztályszintű metódus - alprogram
- "Globális értékek", példa: System.out, Math.PI.

Főprogram működése

- Egy Java program egy osztály "végrehajtásának" felel meg: ha van statikus main nevű metódusa.
- A java-nak megadott nevű osztályt inicializálja a virtuális gép (pl. az osztályváltozóit inicializálja), és megpróbálja elkezdeni a main-jét. Ha nincs main: futási idejű hiba.
- Végetérés: ha a main végetér, vagy meghívjuk a System.exit(int)-et. (Kilépési állapot, nulla szokott a normális állapot lenni.)
- A main feje: amit eddig csináltunk. (Elvileg az args név helyett lehet más is, de az a megszokott.) A String[]-be kerülnek a virtuális gépnek átadott extra paraméterek. C-ben ugyanez: argc, argv.

Java tutorial

Példa

```
public class Argumentumok {
   public static void main( String[] args ){
     if (args.length > 0) {
       for (int i=0; i < args.length; i++)
         System.out.println(args[i]);
     } else {
       System.out.println("nincsenek argumentumok");
     }
   }
}
```

static változók inicializálása

- Ugyanúgy, mint a példányváltozóknál: az előfordulás sorrendjében.

```
static int i = 1;
static int j = 1;
```

- példányváltozók: minden példány létrehozásakor
- osztályváltozók: egyszer, az osztály inicializációjakor
- osztály inicializációja: az első rá vonatkozó hivatkozás kiértékelésekor (pl. példányosítás, metódus meghívása, változó hozzáférés)

static változók inicializálása 2

- inicializátorban lehet már deklarált osztályszintű változó, de nem lehet példányváltozó; példányváltozó inicializátorában viszont lehet osztályszintű változó

```
static int k = i+j-1;
boolean túlkoros = életkor > nyugdíjkorhatár;
```

- automatikus inicializáció van: implicit kezdőérték

Java tutorial

Hivatkozás osztályszintű változóra

- Objektumon és osztályon keresztül egyaránt.

```
Alkalmazott a = new Alkalmazott();
... a.nyugdíjKorhatár ...
... Alkalmazott.nyugdíjKorhatár ...
```

Hivatkozás osztályszintű változóra 2

- minősítés nélkül: az aktuális objektum osztályának osztályszintű változójára
- objektumos minősítéssel
- osztályos minősítéssel; objektumok, példányok hiányában is működik

```
public class Számozott {
  static int következő = 1;
  public final int SORSZÁM = következő++;
}
```

- A példányváltozók inicializátora minden példányosításkor kiértékelődik. (Ez a mellékhatásos kifejezéseknél fontos!)

... és osztálymetódusra

- Csak az osztályváltozókhoz férhet hozzá, a példányváltozókhoz nem. (Nincs aktuális példány, mert akkor is végrehajtható, ha nincs példány.)

```
a.nyugdíjKorhatártEmel(5);
Alkalmazott.nyugdíjKorhatártEmel(5);
```

Feladat

- A Kör osztályhoz írj illeszkedik() műveletet, mely eldönti, hogy a paraméterként megadott Pont objektum illeszkedik-e a körvonalra - egy adott tűréshatáron belül. A tűréshatár értéke a Kör osztály jellemzője.
- A Pont osztályhoz készíts műveletet, amely a paraméterként átadott két Pont objektum által meghatározott szakasz felezőpontját adja vissza.

A this pszeudováltozó

- Az osztálydefiníción belül a példánymetódusokban this névvel hivatkozhatunk az aktuális objektumra.
- A static metódusokban a this persze nem használható.
- Ez egy predefinit név.
- Noha a this.valami-hez általában nem kell a minősítés, időnként azért szükség lehet rá. És olyan is van, amikor maga a this kell (pl. átadni paraméterként).

```
boolean kevesebbetKeresMint( Alkalmazott másik ){
     return másik.többetKeresMint(this);
}

public void fizetéstBeállít( int fizetés ){
     this.fizetés = fizetés;
}
```

Feladat

- A Kör osztály sugaratBeállít metódusának formális paramétere legyen ugyanúgy elnevezve, mint a sugár attribútum.

Java tutorial

Névütközések

- Példányváltozó és formális paraméter neve megegyezhet. Példa: előbb... ELFEDÉS
- Metódusnév és változónév megegyezhet, mert a () megkülönbözteti őket a hivatkozáskor.

```
int fizetés;
public int fizetés(){ return fizetés; }
```

Nevek túlterhelése

- ugyanazt a nevet használhatom több metódushoz, ha különböző a szignatúra
  - szignatúra: név plussz paraméterek típusának sorozata
  - "metódusnév túlterhelése"
  - meghíváskor az aktuális paraméterek száma és (statikus) típusa alapján dönt a fordító (nem számít a visszatérési érték, mert anélkül is meg lehet egy metódust hívni)
  - valaminek illeszkednie kell, különben fordítási hiba

Példa

```
void fizetéstEmel( int növekmény ){
   fizetés += növekmény;
}

void fizetéstEmel(){ fizetés += 5000; }

void fizetéstEmel( Alkalmazott másik ){
   if (kevesebbetKeresMint(másik))
      fizetés = másik.fizetés;
}



a.fizetéstEmel(10000);
a.fizetéstEmel();
a.fizetéstEmel(b);
```

Példa

```
void fizetéstEmel( int növekmény ){
   fizetés += növekmény
}

void fizetéstEmel(){ fizetéstEmel(5000); }

void fizetéstEmel( Alkalmazott másik ){
   if (kevesebbetKeresMint(másik))
      fizetés = másik.fizetés;
}



a.fizetéstEmel(10000);
a.fizetéstEmel();
a.fizetéstEmel(b);
```

Feladat

- Készíts középpontos tükrözést végző műveleteket a Pont és a Kör osztályokban.
- A műveleteket meg lehessen hívni Pont objektummal is és két koordinátával (cx,cy) is!
- Valósítsd meg a középpontos tükrözés műveleteket úgy, hogy egymást hívják!

Öröklődés

- Osztály kiegészítése új tagokkal (példányváltozókkal, metódusokkal).
- Szülőosztály, gyermekosztály. Tranzitív lezárt: ős, leszármazott.
- öröklés: a szülő tagjaival is rendelkezik

```
public class Főnök extends Alkalmazott {

   int beosztottakSzáma = 0;
   public void újBeosztott( Alkalmazott beosztott ){
      ...
   }
  ...
}
```

Osztályhierarchia

- az öröklődési relációt gráfként megadva osztályhierarchiának is nevezik
- egyszeres öröklődés esetén ez egy (irányított) erdő
- Java-ban van egy "univerzális ősosztály", az Object, minden osztály ennek a leszármazottja
  - ha nem adunk meg extends-et, akkor implicit extends Object van
  - Object: predefinit, a java.lang-ban van definiálva
  - olyan metódusokat definiál, amelyekkel minden objektumnak rendelkeznie kell
  - minden, ami nem primitív típusú (int, char, stb.), az Object leszármazottja
- tehát az osztályhierarchia egy irányított fa

Java tutorial

Hozzáférési kategóriák

- Információ-elrejtés: nem jó, ha "mindenki mindent lát".
  - a program komponensei jól meghatározott, szűk interfészen keresztül kapcsolódnak össze
  - egy komponensen belül erős kohézió, komponensek között korlátozott, szűkített interfész
  - a program növekedése nem okoz exponenciális, csak lineáris komplexitás-növekedést
  - nyelvi támogatás ajánlott az információ elrejtéséhez, hogy ne a programozónak kelljen mindenre figyelni

Hozzáférési kategóriák 2

- Komponensek a Java programokban: elsősorban osztályok és csomagok szintjén.
  - Amivel mi most foglalkozunk: minden egy (névtelen) csomagban, pl. minden használt osztály egy könyvtárban lefordítva.
  - Osztályok szintjén: adattagok és metódusok.
- Példa:
  - Kör osztály (sugár és terület)
  - Alkalmazott osztály (fizetés és évesFizetés)
  - Csak egyszerre lehet babrálni, hogy a típusinvariáns megmaradjon: metóduson keresztül lehessen csak csinálni.

Módosító szavak: public, private, protected

- minden tag pontosan egy hozzáférési kategóriában lehet, ezért ezen módosítószavak közül max. egyet lehet használni egy taghoz
- **Félnyilvános tagok**: ha nem írunk semmit.
  - azonos csomagban definiált osztályok (objektumai).
- **Nyilvános tagok**: public
  - különböző csomagokban definiált osztályok is elérik
  - írás/olvasás szabályozása: nincs "read-only"
  - megoldás: lekérdező függvénnyel (akár ugyanazzal a névvel is lehet)

- **Privát tagok**: private
  - csak az osztálydefiníción belül érhető el
  - az osztály minden objektuma
  - jó lenne egy még szigorúbb is, de ilyen nincs
- **Védett tagok**: protected
  - a félnyilvános kategória kiterjesztése: azonos csomag, plussz a leszármazottak
  - Az örökölt tagok mindig ott vannak, de nem mindig érhetőek el (közvetlenül) a gyerekből, csak ha a szülő ezt megengedi a hozzáférési módosítokkal.
    - Pl. egy private változóhoz/metódushoz nem fér hozzá, csak esetleg közvetett úton, más (örökölt) metódusokon keresztül.

Feladat

- Készítsd el a SzínesPont osztályt a Pont osztály leszármazottjaként. Új tulajdonság: szín. Új műveletek: szín beállítása és lekérdezése. A szín attribútum legyen privát.

Inicializáció

A típusinvariáns megteremtése példányosításkor.

- példányváltozók inicializációja
- nehézkes a példányosítást "felparaméterezni" (de nem lehetetlen, lásd a Számozott példát)
- inicializálás egy extra metódussal, pl. init
- más megoldás lenne a Factory, amikor az osztály egy osztályszintű metódusát lehetne használni példányosításra

Factory metódus

```
public class Alkalmazott {
   static public Alkalmazott példányosít( ... ){
      Alkalmazott a = new Alkalmazott();
      ...
      return a;
   }
}
```

- még mindig veszélyes, jobb nyelvi szinten összekapcsolni a példányosítást és az inicializálást

Java tutorial

Konstruktorok

- programkód, ami a példányosításkor "automatikusan" végrehajtódik
- hasonlít a metódusokra, de nem pont ugyanolyan (nem tag, mert pl. nem öröklődik)
- a konstruktor neve = az osztály nevével
- paramétereket vehet át
- több (különböző szignatúrájú) konstruktor is lehet egy osztályban
- csak példányosításkor hajtódhat végre (new mellett)
- a visszatérési típust nem kell megadni, mert az adott

```
class Alkalmazott {

    ...

    public Alkalmazott( String név, int fizetés ){
         this.név = név;
         this.fizetés = fizetés;
         this.évesFizetés = 12*fizetés;
    }

    public Alkalmazott( String név ){
         this.név = név;
         this.fizetés = 40000;
         this.évesFizetés = 12*fizetés;
    }

    ...
}
```

- Módosítók közül csak a hozzáférési kategóriát adók használhatók (vannak egyébként mások is, pl. a final).
- A törzs olyan, mint egy void visszatérési értékű metódusé, a paraméter nélküli return-t használhatjuk.
- Szokás: ugyanazokat a neveket használhatjuk konstruktor formális paraméternek, mint a példányváltozóknak (this használata).
- Meghívhat egy másik konstruktort is, this névvel: az első utasítás lehet csak!

```
public Alkalmazott( String név ){
  this(név,40000);
  ...
}
```

Java tutorial

- A konstruktorok előtt a rendszer lefoglalja a tárat.
- ha a programozó nem ír konstruktort, akkor létrejön egy implicit, ami public, paraméter nélküli és üres törzsű

```
public Főnök(){}
```

- A hozzáférési kategóriák vonatkoznak a konstruktorokra is.
- Használat: new után paraméterek megadása.
  - Aktuális argumentumok a konstruktornak: ezek döntik el, hogy melyik konstruktor hívódik meg.
  - Ha nem írtunk konstruktort, akkor nem adunk át paramétert és az implicit konstruktor hívódik meg.

- A konstruktor nem örökölhető, de meghívható (a this-hez hasonlóan) a szülőosztálybeli konstruktor a legelső sorban: super névvel.

```
public class Főnök extends Alkalmazott {
   public Főnök( String név, int fizetés ){
      super(név,fizetés);
      beosztottakSzáma = 0;
   }
   public Főnök( String név ){
      this(név,100000);
   }
}
```

- Ha egy konstruktor nem hív meg másik konstruktort, akkor implicit módon egy paraméter nélküli `super()` hívás kerül bele; ha nincs paraméter nélküli konstruktora a szülőnek, akkor fordítási hiba.
- Az implicit konstruktor üres, tehát abba is bekerül implicit super.

```
public class SzínesPont extends Pont { int szín
= 0; ... }
```

implicit generálódik: `public SzínesPont() { super(); }`

```
public class Négyzet {
  int oldal;
  public Négyzet( int oldal ){ this.oldal =
oldal; }
}

public class SzínesNégyzet extends Négyzet {
  int szín = 0;
}
```

- A super megelőzi az osztálydefinícióban szereplő példányváltozó inicializálásokat, a többi része a konstruktornak viszont csak utánuk jön.
- Egy protected konstruktort csak super-ként lehet meghívni a csomagon kívül, new-val csak csomagon belül lehet.

Feladat

- A Pont és Kör osztályokhoz készíts konstruktorokat. A Pont osztályhoz csak egyet, aminek a koordinátákat lehet átadni. A Kör osztályhoz hármat:
  - aminek a sugár mellett egy Pont objektumot,
  - illetve a középpont koordinátáit lehet átadni,
  - valamint egy paraméter nélküli konstruktort
- Melyik Kör konstruktor hívhat meg egy másikat? Mit jelentenek a különböző lehetőségek?
- Miért nem fordul a SzínesPont osztály? Javítsd...

Java tutorial

Inicializáló blokkok

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

```
class Számozott {

      static int következő = 0;
      public final int SORSZÁM = következő++;
      int fact;
      {
         fact = 1;
         for (int j=2; j<=SORSZÁM; j++) fact *= j;
      }

      ...
}
```

- Osztályinicializátor és példányinicializátor (az utóbbi csak a Java 1.1 óta). Az osztályszintű inicializátor a static kulcsszóval kezdődik.

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

- 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 a "tagok" közé sorolni

Java tutorial

Destruktorok...

- ...márpedig nincsenek, hiszen szemétgyűjtés van
- mégis, tudomást szerezhetünk az objektum megsemmisítéséről, ami fontos bizonyos applikációknál
- finalize metódust kell írni (az Object-ben van definiálva). Adott forma...

```
protected void finalize()
 throws Throwable { ... }
```

- pontosan nem definiált, hogy mikor hívódik meg: ami biztos, hogy a tárterület újrafelhasználása előtt

"Destruktorok" osztályokhoz

- osztályszinten is van ilyen:

```
static void classFinalize()
  throws Throwable { ... }
```

- ha már nem rendelkezik példányokkal, és más módon sem hivatkoznak rá, akkor az osztály törölhető

Öröklődés megint

- az öröklődés révén kód-újrafelhasználás jön létre, ami
  - a kód redundanciáját csökkenti
  - nem csak a programozást könnyíti meg, hanem az olvashatóságot és a karbantarthatóságot is növeli
- az öröklődés nem csak kódmegosztást jelent, hanem altípus képzést is
  - tervezési szintű fogalom

Altípusosság

- egy parciális rendezés
- a gyermek rendelkezik a szülő összes attribútumával
- minden eseményre reagálni tud, amire a szülő
- ezért minden olyan helyzetben, amikor a szülőt használhatjuk, használhatjuk a gyermeket is:
  - **a gyermek típusa a szülő típusának egy altípusa**
  - (hiszen a típus = megszorítás arra, hogy egy értéket milyen helyzetben szabad használni)

Példa: Főnök része Alkalmazott

- az Alkalmazott műveleteit meghívhatjuk egy Főnökre is

```
  Főnök f = new Főnök("Jancsi",20000);
f.fizetéstEmel(20000);
int i = (new Főnök("Juliska")).fizetés();
```

- egy Alkalmazott formális paraméternek átadható egy Főnök aktuális

```
 Alkalmazott a = new Alkalmazott("Frédi");
if( a.többetKeresMint(new Főnök("Béni")) )
  ...
```

- egy Alkalmazott típusú referenciának értékül adható egy Főnök példány

```
Alkalmazott a = new Főnök("Winnetou");
```

Polimorfizmus, többalakúság

- ugyanaz a művelet meghívható Alkalmazottal és Főnökkel egyaránt:
  több típussal rendelkezik a művelet
- egy rögzített típusú (pl. Alkalmazott) változó hivatkozhat több különböző típusú objektumra (Alkalmazott, Főnök)
- ez a fajta polimorfizmus az ún. altípus polimorfizmus (subtype vagy inclusion polimorfizmus).
  Van másféle is, pl. parametrikus polimorfizmus, ami az Ada generic-re hasonlít

Polimorfizmus: Cardelli-Wegner

- univerzális
  - parametrikus
  - altípus (inclusion)
- ad-hoc
  - típuskényszerítés (coercion)
  - nevek túlterhelése (overloading)

Példa parametrikus polimorfizmusra (Ada)

```
generic
  type T is private;
procedure Swap ( a, b: in out T ) is
  c: T := a;
begin
  a := b;
  b := c;
end Swap;

procedure IntegerSwap is new Swap(Integer);
procedure BooleanSwap is new Swap(Boolean);
...
a: Integer := 42;
b: Integer := 33;
...
IntegerSwap(a,b);
```

Példa parametrikus polimorfizmusra (funkcionális nyelvek, pl. Clean)

```
swap :: (a,a) -> (a,a)
swap (x,y) = (y,x)
```

`swap (42,33)` eredménye `(33,42)`

Java tutorial

Java tutorial

Változók típusa

statikus: a változó deklarációjában megadott típus

dinamikus: a változó által hivatkozott objektum tényleges típusa

- a dinamikus mindig a leszármazottja a statikusnak (vagy maga a statikus)
- a statikus típus állandó, de a dinamikus típus változhat a futás során

```
Alkalmazott a = new Alkalmazott("Adél");
Főnök b = new Főnök("Balázs");
Alkalmazott c = new Főnök("Cecil");
a = c;
a = new Alkalmazott("András");
a = b;
```

Object típusú változók

- felelnek meg a típus nélküli mutatóknak
  - **mindenre hivatkozhatnak, ami nem primitív típusú**

```
Object o = new Alkalmazott("Olga");
o = new Kör();
```

nem megy:

```
o.fizetéstEmel(20000);
Alkalmazott a = o;
```

A statikus típus szerepe

- Egy objektumreferencia (vagy objektum kifejezés) **statikus típusa** dönti el azt, hogy **mit szabad** csinálni az objektummal
  - pl. azt, hogy milyen műveletek hívhatók meg rá
- Nem szabad például az alábbiakat, mert fordítási hiba:

```
Object o = new Alkalmazott("Olga");
o.fizetéstEmel(20000);

Alkalmazott c = new Főnök("Cecil");
Főnök b = c;
```

Különbség az altípusosság és a kódkiterjesztés között

- Példa:
  - a Négyzet az altípusa a Téglalapnak
  - a Téglalap megkapható a Négyzetből újabb adattagok felvételével
- az OO nyelvek többsége nem tesz különbséget a kettő között, mindkettőt ugyanazzal a nyelvi eszközzel (öröklődés) támogatják
- inkább az altípusosság mellett definiáljunk öröklődést
  - *tisztább tervezéshez vezet*
    (esetleg felesleges kód árán, mint pl. Négyzetben a "b")

Java-ban: Erős (strong) típusellenőrzés

- igyekszik fordítási időben típushelyességet biztosítani (static typing)
- esetenként futási idejű ellenőrzéseket is csinál (dynamic typing)

Típuskonverzió

- automatikus (implicit)
- explicit

Automatikus típuskonverzió

- altípusok esetén
  - *szűkebb halmazba tartozó értéket egy bővebb halmazba tartozó értékké konvertál*

1) **objektumok** esetén: egy leszármazott osztályba tartozó objektumot az ősosztályba tartozóként kezel

```
int i = (new Főnök("Juliska")).fizetés();
```

2) **primitív típusok** esetén is definiál a nyelv altípusokat, így:

```
b < c|s < i < l < f < d
```

az `l < f` esetén információvesztés lehet (pontosság)

```
b = 12
```

is jó, egész literált fordítási időben bájttá tud alakítani

a reprezentációt meg kell változtatni

Explicit típuskonverzió: típuskényszerítés

- *bővebből szűkebbet csinál: adatvesztő, nem biztonságos*
- pl. float-ból int-et: Math osztály kerekítő műveletével
- pl. egészek szűkítése esetén a felső bitek vesznek el
- objektumok esetén: ha hiba, akkor **`ClassCastException`** megelőzés: **`instanceof`** operátor

```
Object o = new Alkalmazott("Olga");
Alkalmazott a = (Alkalmazott) o;

((Alkalmazott)o).fizetéstEmel(20000);

if( o instanceof Alkalmazott )
    ((Alkalmazott)o).fizetéstEmel(20000);
```

Feladat

- Készítsd el a SzínesKör osztályt a Kör osztály leszármazottjaként. Új műveletei: a szín beállítása és lekérdezése.
- Ne vezess be új adattagot: a színes körök színét a középpontjuk színe határozza meg, amely nem közönséges pont, hanem színes pont.

Java tutorial

Nevek újrahasznosítása

- Felüldefiniálás (redefining)
  - öröklődés mentén
- Túlterhelés (overloading)
  - változó vs. metódus (zárójelek)
  - metódus vs. metódus (paraméterezés)
- Elfedés
  - példányváltozó vs. metódus paramétere (this)

Túlterhelésre példa

- Ugyanolyan névvel több művelet.
- A paraméterezés dönti el, mikor melyikre gondolunk.
- Gondoljunk a konstruktorokra...

```
class Alkalmazott {

  void fizetéstEmel(){ ... }

  void fizetéstEmel( int mennyivel ){ ... }

  ...
}
```

Túlterhelésre példa

- Ugyanolyan névvel több művelet.
- A paraméterezés dönti el, mikor melyikre gondolunk.
- Gondoljunk a konstruktorokra...

```
class Alkalmazott {

  void fizetéstEmel(){ fizetés += 1000; }

  void fizetéstEmel( int mennyivel ){
    fizetés += mennyivel;
  }

  ...
}
```

Túlterhelésre példa

- Ugyanolyan névvel több művelet.
- A paraméterezés dönti el, mikor melyikre gondolunk.
- Gondoljunk a konstruktorokra...

```
class Alkalmazott {

  void fizetéstEmel(){ fizetéstEmel(1000); }

  void fizetéstEmel( int mennyivel ){
    fizetés += mennyivel;
  }

  ...
}
```

Túlterhelésre példa

```
public static void main( String args[] ){
  Alkalmazott a = new Alkalmazott();
  a.fizetéstEmel(1500);
  a.fizetéstEmel();
}
```

```
class Alkalmazott {

  void fizetéstEmel(){ fizetéstEmel(1000); }

  void fizetéstEmel( int mennyivel ){
    fizetés += mennyivel;
  }

  ...
}
```

Java tutorial

Felüldefiniálás

- a gyermek osztályban bizonyos eseményekre másképp kell (vagy legalábbis lehet) reagálni, mint a szülőosztályban:
- a szülő- (vagy ős)osztálybeli metódust felüldefiniáljuk a gyermekben

```
class Téglalap {
  ...
  double kerület() { return 2*(a+b); }
}

class Négyzet extends Téglalap {
  ...
  double kerület() { return 4*a; }
}
```

- örökölt metódushoz új definíciót rendelünk
- csak példánymetódusok esetén

Ha mást kell csinálnia...

```
class Alkalmazott {
  ...
  int pótlék() { return nyelvvizsgákSzáma*5000; }
}

class Főnök extends Alkalmazott {
  ...
  int pótlék() {
    return nyelvvizsgákSzáma*5000 +
           beosztottakSzáma*1000;
  }
}
```

Az örökölt metódus

- használható a `super.valami()` is a felüldefiniált metódus elérésére

```
class Alkalmazott {
  ...
  int pótlék() { return nyelvvizsgákSzáma*5000; }
}

class Főnök extends Alkalmazott {
  ...
  int pótlék() {
    return super.pótlék() +
           beosztottakSzáma*1000;
  }
}
```

Feladat

- Készíts toString műveletet a Pont és a SzínesPont osztályokhoz, mely az ilyen objektumokról adatokat szolgáltat egy String formájában. (Az adatok az attribútumok értékei legyenek.) A metódus az alábbi specifikációval rendelkezzen:

  public String toString()

Java tutorial

Dinamikus kötés (late binding)

- mindig a dinamikus típus szerinti művelet hívódik meg
- futás közben választódik ki az a metódus, ami végrehajtódik
- C++ virtual -- Java: teljesen dinamikus

```
Alkalmazott a = new Alkalmazott(...);
Főnök f = new Főnök(...);
int i = a.pótlék();
int j = f.pótlék();
a = f;
int k = a.pótlék();
```

- még az örökölt metódus törzsében is dinamikus kötés van

```
class Alkalmazott {
 ...
 int teljesFizetés() { return fizetés() + pótlék(); }
}
```

Feladat

- A Pont osztályban definiáljunk **`println`** metódust, mely kiírja a pontot a szabványos kimenetre. (Ehhez, implicit módon, az előző feladatban írt toString metódust használjuk.)
- Nézzük meg, hogyan működik az örökölt **`println`** metódus a SzínesPont objektumokon!

Felüldefiniálás szabályai

- a szignatúra megegyezik
- a visszatérési típus megegyezik
- a hozzáférési kategória: nem szűkíthető

  private < félnyilvános < protected < public
- specifikált kiváltható kivételek: nem bővíthető

ha a szignatúra ugyanaz, akkor már nem lehet túlterhelés, ezért ha a többi feltétel nem oké, akkor fordítási hiba

```
protected Integer add( Vector v ) throws A, B {...}

public  Integer add( Vector v ) throws C {...}
```

Legyen: `class C extends A {...}`

Java tutorial

Altípusosság egy megközelítése

- "types as sets", típusértékhalmazok tartalmazása
- az altípusú érték mindig használható, ha a bővebb típusú kell
- kontra- és kovariancia, invariancia

```
class X {
  Integer add( Vector v ) throws A, B {...}
  ...
}

class Y extends X {
  public  Integer add( Vector v ) throws C {...}
}

class C extends A {...}
```

```
X x = new X();
Y y = new Y();
Vector v = new Vector();

... x.add(v) ... y.add(v) ...
```

A legfontosabb/legelterjedtebb OO nyelvek

- Simula 67
- Smalltalk
- C++
- Eiffel
- Java

Variancia

művelet paraméter: invariancia (pl. Java)

```
class Gyerek { ... }

class Sílelő extends Gyerek {
  void szobatársatRendel( Sílelő szobatárs ){ ... }
  Sílelő szobatárs(){ ... }
  ...
}

class SílelőLány extends Sílelő {
  void szobatársatRendel( Sílelő szobatárs ){ ... }
  Sílelő szobatárs(){ ... }
  ...
}
```

Variancia

művelet paraméter: kontra-variancia (*nem Java!*)

```
class Gyerek { ... }

class Sílelő extends Gyerek {
  void szobatársatRendel( Sílelő szobatárs ){ ... }
  Sílelő szobatárs(){ ... }
  ...
}

contra-class SílelőLány extends Sílelő {
  void szobatársatRendel( Gyerek szobatárs ){ ... }
  SílelőLány szobatárs(){ ... }
  ...
}
```

Variancia

művelet paraméter: ko-variancia
(pl. Eiffel, *nem Java!*)

```
class Gyerek { ... }

class Sielő extends Gyerek {
  void szobatársatRendel( Sielő szobatárs ){ ... }
  Sielő szobatárs(){ ... }
  ...
}

co-class SielőLány extends Sielő {
  void szobatársatRendel( SielőLány szobatárs ){ ... }
  SielőLány szobatárs(){ ... }
  ...
}
```

Általában megengedhető lenne művelet felüldefiniálása esetén

- kontravariancia (ellentétes változás) az alprogram paraméterében
  - a paraméter típusa bővebb: több paramétert elfogadó
- a visszatérési érték típusára kovariancia
  - a visszatérési érték szűkebb: nem ad olyat vissza, amit az ősbeli sem
- ahol egy bennfoglaló típusú valamit használok, ott lehet helyette egy altípusbelit használni; sőt, még több környezetben használhatom az altípusbelit, hiszen az speciálisabb, több információt hordozó

Példa kontravarianciára (NEM JAVA!)

```
Gyerek gy = new Gyerek();
Sielő s1 = new Sielő(), s2 = new Sielő();
SielőLány slány = new SielőLány();
slány.szobatársatRendel(gy);
s1.szobatársatRendel(s2);
s1.szobatársatRendel(slány);
s1 = slány;
s1.szobatársatRendel(s2);
```

Eiffel-ben (Bertrand Meyer) megengedett művelet felüldefiniálása esetén

- kovariancia (együttváltozás) a felüldefiniált alprogram paraméterében
  - az altípusbeli műveletben a szülőbeli paramétertípusának egy altípusa szerepel
- a visszatérési érték típusára is kovariancia

```
Sielő s1 = new Sielő(), s2 = new Sielő();
SielőLány slány = new SielőLány();
s1.szobatársatRendel(s2);
s1 = slány;
s1.szobatársatRendel(s2);  NEM JÓ!
```

Még egy példa Eiffel-ből

- Egy altípusból elhagyhatók bázistípusból örökölt műveletek.
- Ez megsérti azt a szabályt, hogy az altípusú érték mindig használható ott, ahol a bázistípusú érték.
- Bizonyos esetekben programozás-technikailag kényelmes tud lenni. Például:

  a Madár repül
  a Pingvin nem repül

Java tutorial

Visszatérve a Java-hoz

- invariancia (nem-változás) az alprogram paraméterében
  - az altípusban a paraméter típusa ugyanaz, mint a szülőben
- visszatérési értékre is invariancia
- viszont kivételekre (speciális visszatérési érték) ko-variancia, és láthatóságra (ami a paraméterekre hasonlít) kontra-variancia

- túlterhelés: a szignatúra különböző, tehát azt definiálja felül, amivel megegyezik a szignatúra
- felüldefiniálásnál igyekezzünk megőrizni a jelentést, inkább csak a kiszámítás módja legyen más

Heterogén adatszerkezetek

- Egy adatszerkezetben többféle osztályú objektumot szeretnénk tárolni.
  - néha jó, néha nem
- Például a predefinit **`Vector`** osztályban **`Object`**-ek vannak.
- Egy tömbbe is tehetünk különbözőket egy **közös őstípus** alapján.

```
Test[] t = {new Kocka(42.0),new Gömb(33.0) };
```

a **statikus** típus ugyanaz, a **dinamikus** típus eltérhet

Túlterhelés: választás a változatok között

- ha a szignatúrában szereplő típusok ős-leszármazott viszonyban állnak
  - fordítási időben történik a választás
  - az aktuális paraméterek statikus típusa dönt
  - altípusosság lehetséges
  - "legjobban illeszkedő"
  - fordítási hiba, ha nincs
  - típuskényszerítés
  - rossz stílus, kerülendő a többértelműség

```
void m(Alkalmazott a1, Alkalmazott a2) { ... }
void m(Alkalmazott a, Fonok f) { ... }
void m(Fonok f, Alkalmazott a) { ... }
```

Java tutorial

Elfedés

- osztályszintű metódusoknál nincs felüldefiniálás + dinamikus kötés
- a statikus metódusokat elfedni lehet
- nem csak a szignatúrának kell megegyezni, hanem... hasonló szabályok mint felüldefiniálásnál, különben fordítási hiba

```
class A {
  static void alma(int x){ ... }
}

class B extends A {
  static void alma(int x){ ... }
}
```

Osztálymetódus elfedése

- elfedett elérése:
  - super
  - minősítés
  - típuskényszerítés

```
Alkalmazott a = new Alkalmazott();
Főnök f = new Főnök();

a.nyugdíjKorhatár()
f.nyugdíjKorhatár()
Alkalmazott.nyugdíjKorhatár()
((Alkalmazott)f).nyugdíjKorhatár()
```

```
class Alkalmazott {
  ...
  static public int nyugdíjKorhatár() { return 65; }
  static public int A_nyugdíjKorhatár() {
      return nyugdíjKorhatár();
   }
}
class Főnök extends Alkalmazott {
  ...
  static public int nyugdíjKorhatár() {
    return super.nyugdíjKorhatár() + 5;
    return Alkalmazott.nyugdíjKorhatár() + 5;
  }
}
```

- statikus kiválasztás (fordítási időben)
- elfedett elérése öröklött metóduson keresztül

```
Alkalmazott a = new Alkalmazott();
Főnök f = new Főnök();

a.nyugdíjKorhatár()
f.nyugdíjKorhatár()
a = f;
a.nyugdíjKorhatár()
a.fizetés()
a.A_nyugdíjKorhatár()
```

```
class Alkalmazott {
  ...
  static public int nyugdíjKorhatár() { return 65; }
  static public int A_nyugdíjKorhatár() {
      return nyugdíjKorhatár();
   }
}
class Főnök extends Alkalmazott {
  ...
  static public int nyugdíjKorhatár() {
    return super.nyugdíjKorhatár() + 5;
    return Alkalmazott.nyugdíjKorhatár() + 5;
  }
}
```

- példánymetódust nem szabad elfedni

```
class Alkalmazott {
  ...
  static public int nyugdíjKorhatár() { return 65; }
  static public int A_nyugdíjKorhatár() {
      return nyugdíjKorhatár();
   }
}
class Főnök extends Alkalmazott {
  ...
  static public int nyugdíjKorhatár() {
    return super.nyugdíjKorhatár() + 5;
    return Alkalmazott.nyugdíjKorhatár() + 5;
  }
  static float nyugdíjKorhatár() { ... }     NEM JÓ!
  static int fizetés(){...}                  NEM JÓ!
}
```

Változók elfedése

- példány- vagy osztályszintű változók esetén
- statikus kiválasztás
- az elfedett változókhoz nem lehet közvetlenül hozzáférni, csak
  - super-es minősítéssel
  - típusos minősítéssel (osztályváltozó esetén)
  - típuskényszerítéssel
  - örökölt metóduson keresztül

Példa példányváltozó elfedésére

```
class a {
  int x = 0;
  void pr() { System.out.println(x); }
}

class b extends a {
  int x = 1;
  void pri(){ System.out.println(super.x); }
  static public void main( String[] args ){
    a v = new b();    b w = new b();
    System.out.println(v.x + " " + w.x);
    System.out.println(((a)w).x);
    v.pr();  w.pr();
    w.pri();
  }
}
```

Java tutorial

A final módosítószó

- "nem változtatható"
  - változó (példány, osztályszintű, lokális, paraméter)
  - metódus (példány, osztályszintű)
  - osztály

final változók

- kvázi konstansok
  - nem változtatható a változó értéke, ha már egyszer beállítottuk
  - `final double ADÓKULCS = 0.25;`

final változók

- kvázi konstansok
  - nem változtatható a változó értéke, ha már egyszer beállítottuk
  - `final double ADÓKULCS = 0.25;`
- üres konstansok (a fordító észreveszi a hibát)

```
final static int i = 100;
final static int ifact;
static {
  int j = 1;
  for(int k=1; k<100; k++)
    j *= k;
  ifact = j;
}
```

final változók

- kvázi konstansok
  - nem változtatható a változó értéke, ha már egyszer beállítottuk
  - `final double ADÓKULCS = 0.25;`
- üres konstansok (a fordító észreveszi a hibát)

```
final static int i = System.in.read();
final static int ifact;
static {
  int j = 1;
  for(int k=1; k<100; k++)
    j *= k;
  ifact = j;
}
```

Üres konstans értéke konstruktorokból

```
final String név;

public Alkalmazott( String név ){
  this(név,100000);
}

public Alkalmazott( String név, int fizetés ){
  this.név = név;
  this.fizetés = fizetés;
}
```

final változó: a referencia nem változhat

```
class A {
  int v = 10;

  public static void main( String args[] ){
    final A a = new A();
    a = new A(); // NEM SZABAD!
    a.v = 11;
  }
}
```

Java tutorial

final metódus és osztály

- "nem változtatható"
  - **nem lehet felüldefiniálni, illetve leszármaztatni belőle**

```
public class Object {
  public final Class getClass();
  ...
}

public final class System {
  ...
}
```

Pl. ha veszélybe sodorná a rendszer működését...

Java tutorial

Absztrakt osztályok

- hiányosan definiált osztály (valami nincs benne készen)
- nem lehet belőle példányosítani
- az altípus reláció megadása tervezés szempontjából sokszor megkívánja

Absztrakt osztályok

- hiányosan definiált osztály (valami nincs benne készen)
  - bizonyos műveleteknek még nem adjuk meg az implementációját
- nem lehet belőle példányosítani
- az altípus reláció megadása tervezés szempontjából sokszor megkívánja

Absztrakt osztályok

- hiányosan definiált osztály (valami nincs benne készen)
  - bizonyos műveleteknek még nem adjuk meg az implementációját
  - az `abstract` módosítószóval jelezzük
- nem lehet belőle példányosítani
- az altípus reláció megadása tervezés szempontjából sokszor megkívánja

Absztrakt osztályok

- hiányosan definiált osztály (valami nincs benne készen)
  - bizonyos műveleteknek még nem adjuk meg az implementációját
  - az `abstract` módosítószóval jelezzük
- nem lehet belőle példányosítani
- az altípus reláció megadása tervezés szempontjából sokszor megkívánja
  - csak azért kellenek, hogy "igazi" osztályok közös viselkedését csak egyszer kelljen leírni, vagy hogy "igazi" osztályok közös őssel rendelkezzenek

```
public abstract class Gyerek {

  protected Játék kedvencJáték; // és egyéb attribútumok

  ...

  public abstract double mennyireSzereti(Gyerek másik);

  public Gyerek legjobbBarát( Gyerek[] osztály ){
     int maxhely = 0;
     double maxérték = mennyireSzereti(osztály[0]);
     for( int i = 1; i<osztály.length; i++ ){
        double érték = mennyireSzereti(osztály[i]);
        if( érték > maxérték )
              { maxérték = érték; maxhely = i; }
     }
     return osztály[maxhely];
  }
}
                    Gyerek gyerek = new Gyerek();
```

```
public abstract class Gyerek {
  protected Játék kedvencJáték;      // és egyéb attribútumok ...
  public abstract double mennyireSzereti(Gyerek másik);
  public Gyerek legjobbBarát( Gyerek[] osztály ){ ... }
}

public class Fiú extends Gyerek {
  ...
  public double mennyireSzereti(Gyerek másik){
     double összeg = 0.0;
     if( kedvencJáték.equals(másik.kedvencJáték) )  összeg +=
10.0;
     ...
     return összeg;
  }
}

public class Lány extends Gyerek {
  ...
  public double mennyireSzereti(Gyerek másik){
     if( másik.kedvencJáték.equals(Játék.döglöttMacska) )  return
0.01;
     if( másik.kedvencJáték.equals(Játék.hajasBaba) )  return
12.0;
     ...
```

Feladat

- Valósítsd meg a Hasáb absztrakt osztályt.
  Tulajdonság: magasság.
  Absztrakt művelet: alapterület számítása.
  Másik művelet: térfogat számítása.
- Készítsd el a Henger és Kocka osztályokat, melyek a Hasáb konkrét leszármazottjai.

Java tutorial


```
public abstract class Hasáb {
  protected double magasság;
  public abstract double alapterület();
  public double térfogat(){
    return alapterület() * magasság;
  }
}

public class Henger extends Hasáb {
  ... // konstruktorok
  protected double sugár;
  public double alapterület(){
    return sugár*sugár*Math.PI;
  }
}

public class Kocka extends Hasáb {
  ... // konstruktorok
  public double alapterület(){
    return magasság*magasság;
  }
}


Hasáb h = new Kocka(10.0);
```

Absztrakt osztályok: összegezve

- nem példányosítható közvetlenül, előbb specializálni kell (megvalósítani az absztrakt műveleteket)
- a gyerek is lehet absztrakt
  - akár absztrakt a szülő, akár nem
- absztrakt statikus típussal rendelkező változók hivatkozhatnak valamilyen leszármazott konkrét dinamikus típusú objektumra
  - dinamikus kötés híváskor
- egy absztrakt metódus nem lehet private, final vagy static (sem native)

Java tutorial

Többszörös öröklődés

- egy osztály több osztálytól is örökölhet
  - az osztályhierarchia egy irányított körmentes gráf (de nem feltétlenül fa vagy erdő)
  - például: RészegesEgyetemista, KétéltűJármű, SzínesNégyzet
- vita a szakirodalomban
  - nagyon hasznos dolog
  - problémákat vet fel (állítólag)
- **Java**: kompromisszum (jó? nem jó?)
  - *osztályokra egyszeres öröklődés, de...*

Többszörös öröklődés "problémái"

- ugyanolyan attribútum/metódus többszörösen
  - class D extends B, C
  - B.valami C.valami
  - felüldefiniáláskor D.valami
- ismételt öröklődés: ugyanaz többször
  - class B extends A class C extends A
  - A.valami

Interfészek

- egy másik referencia típus
- az absztrakt osztályok definíciójára hasonlít
- típusspecifikáció-szerűség
- többszörös öröklődés (altípus reláció)
- nem új dolog, Objective-C protocol

Interfész: "absztrakt"

- teljesen absztrakt, egyáltalan nincs benne "kód", csak specifikáció
  - pl. nincs metódustörzs vagy példányváltozó benne

```
public interface Enumeration {
  public boolean hasMoreElements();
  public Object nextElement();
}
```

Interfész: nem példányosítható

- ugyanúgy, mint az absztrakt osztályok: nem példányosítható közvetlenül
- előbb "meg kell valósítani"

```
class ListaIterátor implements Enumeration {
  ...
  public boolean hasMoreElements(){ ... }
  public Object nextElement(){ ... }
}
```

```
HalmazIterátor
TömbIteráror
SorozatIterátor
```

stb...

Interfész: öröklődés

- Interfészek is kiterjeszthetik egymást: extends
- Akár többszörös öröklődés is lehet
  - ha a kódöröklés szintjén gondot is okozhat a többszörös öröklődés, azért a specifikáció öröklődése esetén nem
- Az osztályok megvalósíthatnak interfészeket
  - ugyanazt az interfészt többen is
  - ugyanaz az osztály többet is

Relációk a referencia-típusokon

- Öröklődés osztályok között
  - fa (egyszeres öröklődés, közös gyökér)
  - kódöröklés
- Öröklődés interfészek között
  - körmentes gráf (többszörös öröklődés, nincs közös gyökér)
  - specifikáció öröklése
- Megvalósítás osztályok és interfészek között
  - kapcsolat a két gráf között, továbbra is körmentes
  - specifikáció öröklése

Interfész megvalósítása

- Ha

  az I egy interfész,
  J az I egyik őse,
  az A osztály megvalósítja I-t,
  B leszármazottja az A-nak,

- akkor

  B megvalósítja J-t.

Java tutorial

Interfész: típus

- használhatók változódeklarációkban
- használhatók formális paraméterek specifikációjában

Interfész: típus

- használhatók változódeklarációkban
- használhatók formális paraméterek specifikációjában

- egy interfész típusú változó:
  - referencia, ami olyan objektumra mutathat, amely osztálya (közvetlenül vagy közvetve) megvalósítja az interfészt

```
I v = new A();
J w = new B();
```

Interfész: típus

- használhatók változódeklarációkban
- használhatók formális paraméterek specifikációjában

- egy interfész típusú formális paraméter:
  - megadható egy olyan aktuális paraméter, amely egy objektum, és amely osztálya (közvetlenül vagy közvetve) megvalósítja az interfészt

```
void m( I p ){...}      m(new A());
void n( J p ){...}      n(new B());
```

Java tutorial

Ha egy változó (vagy formális paraméter) deklarált típusa (azaz statikus típusa) egy interfész, akkor

- dinamikus típusa egy azt megvalósító osztály
  - csak ennyit használhatunk ki róla
- a változóra olyan műveleteket használhatunk, amelyek az interfészben (közvetlenül vagy közvetve) definiálva vannak

```
void elemeketKiír( Enumeration e ){
  while( e.hasMoreElements() )
    System.out.prinln(e.nextElement());
}

Vector v;
...
elemeketKiír( v.elements() );
```

Interfészek megadása

- módosítószók:
  - **abstract** (Automatikusan, azaz nincs hatása. Nem szokás.)
  - **public** (Mint osztályoknál; több csomag esetén érdekes.)
- kiterjesztés (öröklődés):
  - extends után lista
- példánymetódusok specifikációja és osztályszintű konstansok
  - nem lehetnek benne üres konstansok
- Az interfészek neve gyakran -ható, -hető (azaz -able)

`Comparable, Runnable (Futtatható)`

Metódusok az interfészekben

- használható módosítók:
  - abstract és public (automatikusan, nincs hatásuk)
- nem használhatók:
  - protected
  - private
  - static
  - final
  - (native, synchronized)

Változódeklarációk az interfészekben

- használható módosítók:
  - public final static (automatikusan, nincs hatásuk)
- nem használhatók:
  - protected
  - private
  - (volatile, transient)

Interfészt megvalósító osztály

- az osztályban egy implements klóz, benne több interfész felsorolható
- a metódusok megvalósítása public kell, hogy legyen
- a konstansok specifikációját természetesen nem kell megismételni

Java tutorial

Névütközés

- változók
- metódusok

- öröklődés során újabb entitás ugyanazzal a névvel vagy szignatúrával
- többszörös öröklődés során több helyről is
  - több helyről, de ugyanazt

Névütközések változódeklarációkban

- elfedés
- elérés minősítésen keresztül

- ha többszörösen örökölt
  - ha ugyanaz több úton, akkor csak egy lesz belőle
  - ha másik, akkor a hivatkozásnál fel kell oldani a többértelműséget, különben fordítási hiba

Névütközések metódusoknál

- lehet túlterhelés (különböző szignatúra)
- lehet felüldefiniálás, ha a szignatúra megegyezik; ilyenkor kell még (különben fordítási hiba):
  - visszatérési típus egyenlősége
  - a kiváltott kivételekre a szokásos kovariancia
  - a hozzáférési kategóriára nincs korlátozás, hiszen minden public

- többszörös öröklés
  - ha ugyanaz, akkor egyértelmű
  - ha több ugyanolyan szignatúrájú, akkor a "legnagyobb közös osztó" ("azt" kell venni felüldefiniálásnál és megvalósításnál)

Feladat

- Készítsd el a Színezhető interfészt, mely műveleteket definiál szín lekérdezésére és beállítására. A SzínesPont és SzínesKör osztályok valósítsák meg ezt az interfészt.