CODESPRING
OUR CODE YOUR SUCCESS
20 years

Codespring mentorprogram
Enterprise szoftverrendszerek fejlesztésének alapjai (MP I)

Let's win the race together!

Objektumorientált programozás Java-ban

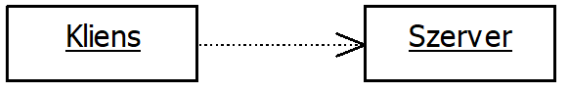
Simon Károly
simon.karoly@codespring.ro

Osztályok és objektumok

- ADT (Abstract Data Types): adatok + az adatokkal végrehajtható műveletek
- Osztályok: adattípus meghatározása, melynek alapján példányokat (objektumokat) hozhatunk létre
  - adatrejtés, öröklődés, polimorfizmus
- Objektum: adatok (attribútumok) + műveletek (metódusok – az adatokat feldolgozó kód)
- Objektum állapota: az attribútumok aktuális értékei határozzák meg
- Objektumok beazonosítása: referenciák segítségével

Osztályok és objektumok

- Kommunikáció: kliens – szerver modell



- egy objektum (kliens) kéri egy másik objektumtól (szerver) bizonyos művelet végrehajtását. A kérés egy üzenet, gyakorlatilag a szerver objektum valamelyik nyilvános (publikus) metódusának meghívása. Ehhez a kliensnek rendelkeznie kell a szerverre mutató referenciával:

```
public class Server {
  ...
}
```

```
public class Client {
  Server s;
  ...
}
```

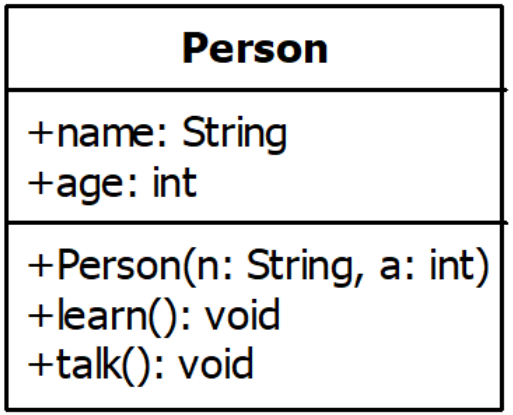
Objektumorientált fejlesztés

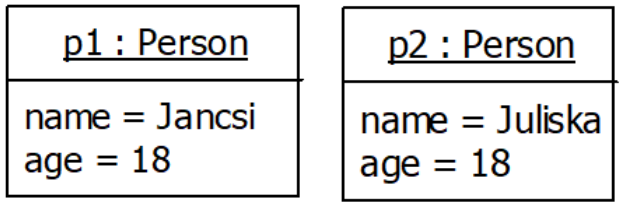
```java
public class Person {

    //az attributumok:
    public String name;
    public int age;

    //a konstruktor:
    public Person(String n, int a) {
        name = n;
        age = a;
    }

    //a metodusok
    //nem adunk meg konkret implementaciot:
    public void talk() {...}
    public void learn() {...}
}
```



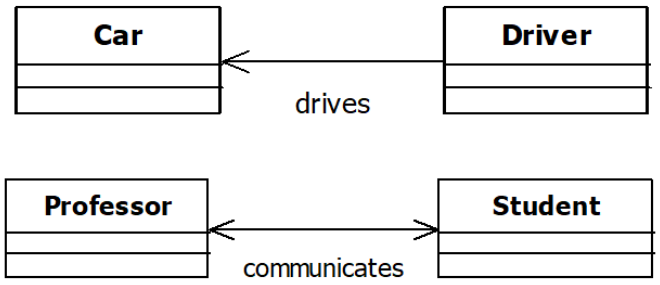


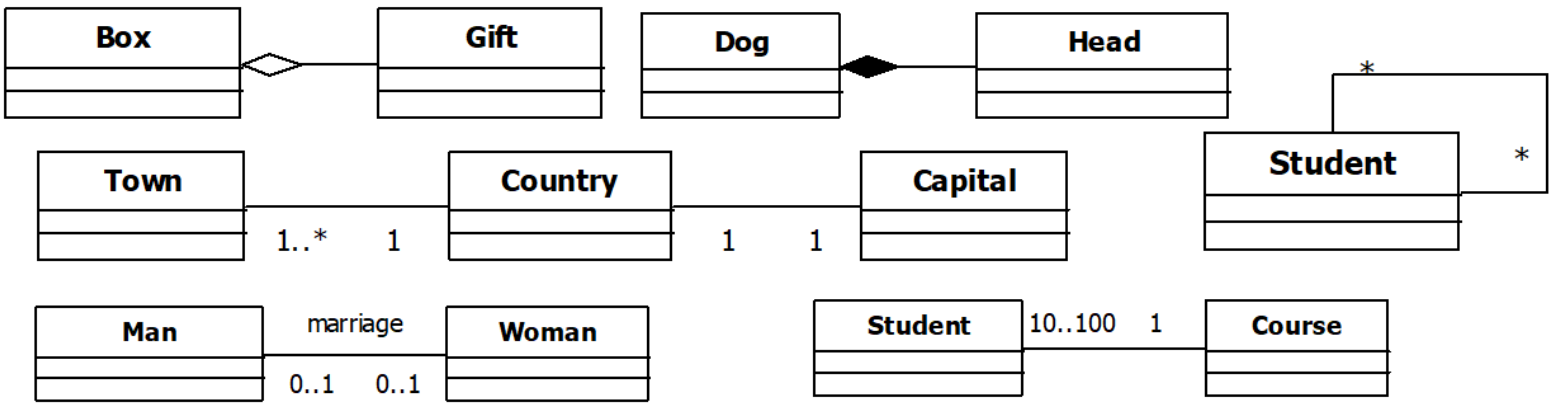
Példányosítás:

```java
Person p1 = new Person ("Jancsi", 18);
Person p2 = new Person("Juliska", 18);
```

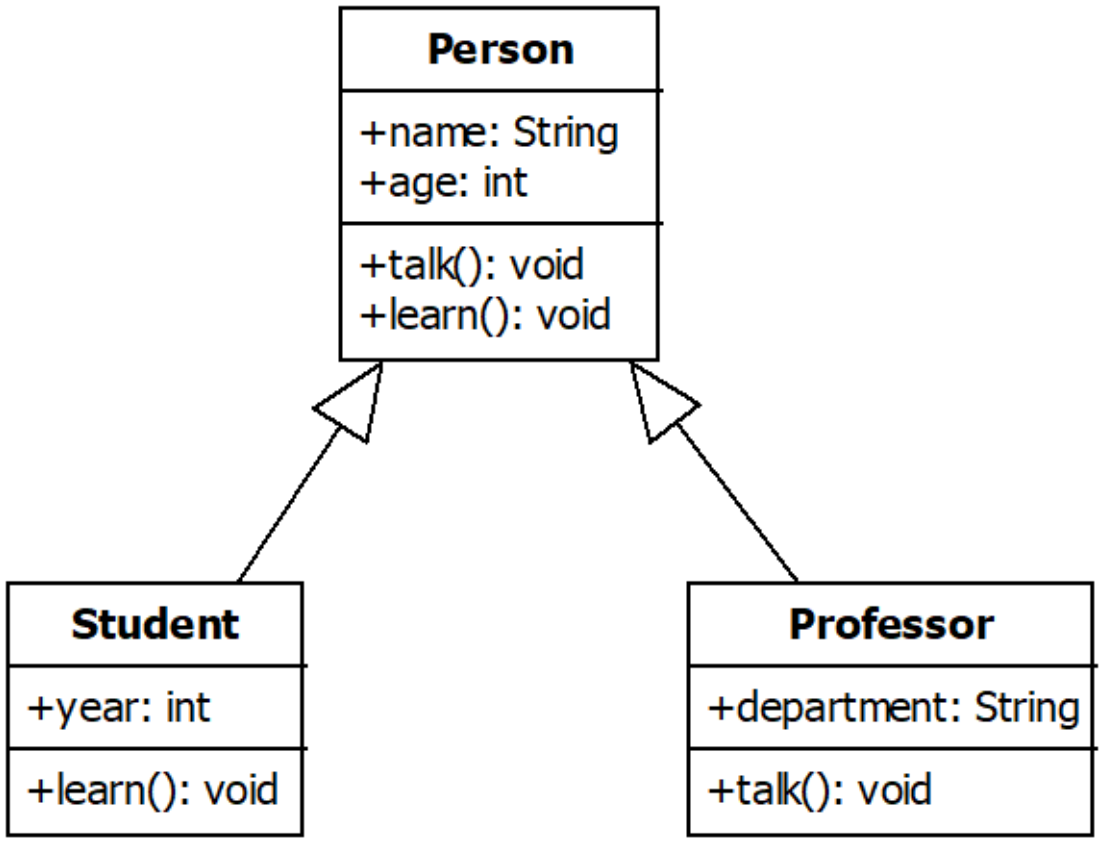
Osztályok közötti kapcsolatok

- Ismerettségi viszony
- Tartalmazási viszony
  - Gyenge (aggregation)
  - Erős (composition)
- Számosság szerint:
  - Egy az egyhez (one to one)
  - Egy a többhöz (one to many)
  - Több a többhöz (many to many)



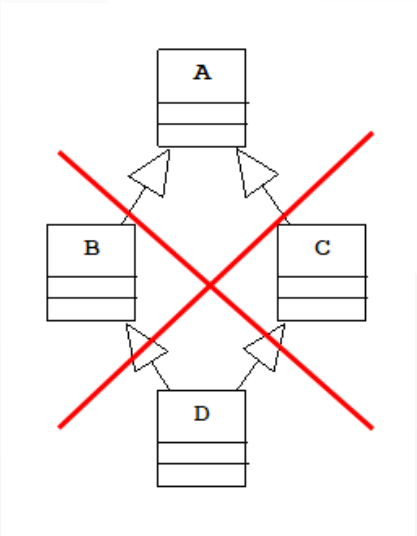


Öröklődés


Person
+name: String
+age: int
+talk(): void
+learn(): void
Student
+year: int
+learn(): void
Professor
+department: String
+talk(): void

Öröklődés

- metódusok túlterhelése (method overloading): az osztályon belül több metódus azonos névvel - a paraméterlista és esetenként a visszafordított típus különbözik. Statikus kötés – a metódus címe már a kompilálási fázisban ismert.
- metódusok újradefiniálása (method overriding): a származtatott osztályok újradefiniálják az alaposztály metódusait. Dinamikus kötés. Java-ban nem megengedett a többszörös öröklés (gyémántöröklés problémájának kiküszöbölése)



Öröklődés


```
class A {
    A(String a) {
      System.out.println("Az A osztály konstruktora " + a);
    }
}

class B extends A {
    B(String b) {
      super(b);
      System.out.println("A B osztály konstruktora " + b);
    }
}

public class Example {
    public static void main(String[] args) {
      B b = new B("Hi");
    }
}
```

- Alaposztály paraméteres konstruktorának meghívása, a származtatott osztály konstruktorának elején: **`super(...)`**
- **`super`** referencia: az alaposztály példányára hivatkozik
- **`this`** referencia: hivatkozás az aktuális példányra
- Adattagok inicializálása, konstruktorok (túlterhelés, láncolás stb.)

Hozzáférés módosítók

- Adatrejtés, attribútumok és metódusok láthatósága:
  - Nyilvános (bárhonnan elérhető): `public` (+)
  - Privát (csak osztályon belül látható): `private` (-)
  - Védett (a származtatott osztályokon belül látható): `protected` (#)
  - Csomagszintű (a csomagon belül látható), `package-private`: ha nem határozzuk meg másképpen, akkor ez az alapértelmezett

Típusmódosítók

- Osztályok esetében:
  - `final` – nem lehet belőle származtatni
  - `abstract` – alaposztályként alkalmazható, nem példányosítható (ha van legalább egy absztrakt metódusa, az osztályt is absztraktnak kell deklarálni)
- Metódusok esetében:
  - `static` – osztálymetódusok, meghívásukhoz nem szükséges példányosítás, az osztály nevével hívjuk meg őket: `osztalynev.metodus()` (pl. `Integer.toString(...)`). Csak statikus attribútumokkal végezhetnek műveleteket és nem hívhatnak meg nem statikus metódusokat.
  - `abstract` – csak absztrakt osztályokban deklarálhatóak, nincsenek implementálva, a származtatott osztályok „kötelesek” ezeket implementálni
  - `final` – a származtatott osztályokban nem újradefiniálhatóak
  - `native` – platformfüggő programozási nyelvben (pl. C++) vannak implementálva
  - `synchronized` – kritikus erőforrásokhoz való hozzáférés

Típusmódosítók

- Attribútumok esetében:
  - `static` – osztályszintű változó, mindenik példány ugyanazt az értéket használja
  - `final` – csak egyszer történhet értékadás, konstansok deklarációjánál alkalmazhatjuk. Figyelem: egy referencia esetében a final használata nem jelenti azt, hogy a referencia által azonosított objektum állapota (attribútumainak értéke) nem változhat, csak azt, hogy a referencia nem „átirányítható".
  - `transient` – nem képezi részét az objektum perzisztens állapotának
  - `volatile` – a változó értékét több végrehajtási szál változtathatja

Alapvető konvenciók

```
public class Person {

    private String fullName;
    private int age;

    public Person(String fullName, int age) {
        this.fullName = fullName;
        this.age = age;
    }

    public void setFullName(String fullName) {
        this.fullName = fullName;
    }

    public String getFullName() {
        return fullName;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public int getAge() {
        return age;
    }

}
```

- Elnevezési és kódolási konvenciók
- Privát adattagok, publikus getter/setter metódusok
- final típusmódosító használata is indokolt lehet a konstruktorok és setter metódusok paramétereinek esetében

Polimorfizmus

- Polimorfizmus (többalakúság): egy B típusú objektum egy adott helyzetben A típusúként jelenik meg, és A típusú objektumként használjuk. Természetesen ez csak akkor lehetséges, ha a két osztály (A és B) között származtatási viszony áll fent.
- Mivel a származtatott osztály örökli az alaposztály tulajdonságait, használható minden olyan helyzetben, ahol az ős használható. Az angol terminológiában ezt a helyzetet a „B is A" kifejezés érzékelteti (amennyiben B az A leszármazottja). Vigyázat: a kijelentés fordítottja már nem igaz.
- Egy egyszerű példa: a négyszög (A) osztályból származtatjuk a négyzet (B) osztályt. A négyzetről elmondható, hogy négyszög, de természetesen nem minden négyszög négyzet. A származtatott osztály az őshöz képest új tulajdonságokkal is rendelkezhet, így előállhatnak olyan helyzetek, amikor az ős nem helyettesítheti utódját. Egy négyzet (B) objektumot, viszont bármilyen helyzetben „kezelhetünk" négyszögként, így semmi akadálya, hogy egy négyszög típusú referencia egy négyzet objektumra mutasson.

Polimorfizmus

- Egy referencia esetében beszélhetünk statikus és dinamikus típusról vagy kötésről. A statikus típus az, amely a deklarációban szerepel, a dinamikus típus a referencia által aktuálisan beazonosított objektum típusa.
- Mi történik akkor, ha egy A típusúként deklarált referencia adott pillanatban egy B típusú objektumra mutat, és segítségével meghívunk egy metódust, melyet a B osztály újradefiniált? A metódusnak „melyik változata" fog érvényesülni?
  - Természetesen, ha újradefiniáltuk a metódust, valószínűleg azt szeretnénk, hogy az új, a konkrét típusnak megfelelő implementáció érvényesüljön. De ez nem minden nyelvben történik automatikusan így. Pl.: C++ - virtuális tagfüggvények
  - A Java nyelvben a metódusok újradefiniálásának esetében mindig az objektum konkrét típusának megfelelő implementáció érvényesül (dinamikus kötés). Azt is mondhatnánk, hogy a Java-ban minden tagfüggvény virtuális.

Absztrakt osztályok

- Származtatás és polimorfizmus: egy rendszeren belül létrehozható közös alap különböző, de azonos alaptulajdonságokkal is rendelkező objektumok részére. Ez lehetővé teszi, hogy azokban az esetekben, amikor csak a közös tulajdonságok relevánsak, azonos módon hivatkozzunk ezekre az objektumokra. Ezt egyszerűen megtehetjük akkor, ha az osztályok rendelkeznek egy közös őssel.
- Előállhatnak olyan esetek, amikor az ősosztálynak nem lehetnek példányai, vagy a rendszer szempontjából értelmetlen lenne a példányosítás.
- Példa: mértani alakzatokat megjelenítő felület. Az alakzatok rendelkeznek közös tulajdonságokkal (pozíció, szín, stb.), de nem lenne értelme, hogy egy általános alakzat objektumot hozzunk létre. A közös tulajdonságok nem lennének elegendőek a megjelenítéshez. Mégis hasznos lenne, ha a különböző alakzatoknak megfelelő osztályok rendelkeznénk egy közös őssel, hogy bizonyos esetekben egységesen hivatkozhassunk rájuk.

Absztrakt osztályok

- Absztrakt metódusok és osztályok: Java-ban abstract kulcsszóval jelöljük az absztrakt metódusokat, és nem adunk meg konkrét implementációt. Ha egy osztálynak van egy absztrakt metódusa, absztrakt osztályról van szó, és ezt a fejlécben jeleznünk kell.
- Az absztrakt metódusokat általában a származtatott osztályok implementálják. Ha ezt mégsem teszik, akkor az illető származtatott osztályt is absztraktnak kell deklarálni.
- Absztrakt osztály: közös tulajdonságok kiemelése + közös felület/viselkedési mód meghatározása
  - Csak ősosztályként alkalmazható, nem példányosítható

Interfészek

- Interfész általánosan: rendszerek közötti kommunikációnál egy adott rendszer interfésze írja le, hogy kívülről hogyan lehet hozzáférni a rendszerhez.
- Osztályoknál: tulajdonképpen az osztály interfészét a publikus adattagok és metódusok alkotják.
- Java-ban további jelentés: interfész = típus deklaráció, mely egy bizonyos viselkedési módot határoz meg.
- Konstansokból és nem implementált (absztrakt) metódusokból (metódus prototípusokból) áll.
  - Megjegyzés: az interfészeken belüli default, statikus és privát metódusokat később tárgyaljuk
- Azok az osztályok, amelyek „megvalósítják” (implementálják) az illető interfészt, kötelező módon implementálják az abban deklarált metódusokat (minden metódust).

Interfészek

- Úgy is tekinthetünk az interfészekre, mint „szerződésekre”, amelyeket az implementáló osztályoknak be kell tartaniuk. Ha egy osztály megvalósít egy adott interfészt, „vállalja azt”, hogy az interfésznek megfelelően fog „viselkedni”.
- Alkalmazás: hasonló funkcionalitásokkal bíró osztályok részére egy közös viselkedési mód meghatározása, függőségek feloldása.
- Egy osztály több interfészt is megvalósíthat (különbség az absztrakt osztályokhoz viszonyítva).
- Míg az absztrakt osztály tartalmazhat attribútumokat és nem absztrakt metódusokat is, az interfész nem (minden metódusa absztrakt, a későbbiekben tárgyalt default, statikus és privát metódusokon kívül).

Interfészek

- Interfész

```
public interface Resizable {
    public void resize(Dimension d);
}
```

- Megvalósító osztály

```
public class Circle implements Resizable {
    public void resize(Dimension d) {
        //a kör újraméretezését megvalósító kód
        ...
    }
}
```

- Main:

```
public static void main(String[] args) {
    Circle c = new Circle();
    c.resize(new Dimension(100,100));
    Resizable s = new Circle();
    s.resize(new Dimension(100,100));
}
```

- Túl szoros, fölösleges függőségek feloldása - példa:

```
public void resizeModels(List<Resizable> models) {
  for (Resizable r:models) r.resize(new Dimension(100,100));
}
```

Interfészek

- Függőségek feloldása, példa: egy metódus egy karakterláncokból álló listát vár, hogy az elemeit kiírja a konzolra. A lista többféleképpen megvalósítható. Például, az elemeket tárolhatjuk tömbben, vagy láncolt listát alkalmazhatunk. A feladat szempontjából ezek az implementációs részletek nem relevánsak. Csak az fontos, hogy a paraméterként kapott objektum rendelkezzen a listák alapvető tulajdonságaival, például lehessen egy iterátor segítségével bejárni, hogy kiírhassuk az elemeit. Fölösleges lenne megkötnünk a metódusunkat használó programozó kezét azzal, hogy rákötelezzük egy adott implementáció alkalmazására.
  - A megoldás: a paraméter típusát interfész segítségével határozzuk meg.
  - Az interfészek hasonlóan alkalmazhatóak a metódusok által visszatérített értékek típusának esetében is.

Interfészek

- Más példa: alkalmazás-programozási felületek (API – Application Programming Interface): egy cég komplex műveleteket megvalósító osztályokat tartalmazó szoftvercsomagot készít. A csomagot egy másik cég fogja felhasználni saját alkalmazásának fejlesztésekor. A tipikus eljárás, hogy az osztályok publikus interfészeknek lesznek a megvalósításai. A publikus interfészek segítségével lehet majd meghívni az osztályokon belül implementált metódusokat az implementációs részletek ismerete nélkül.
- Komponens alapú programozás, szolgáltatásorientált architektúrák
- Összefoglaló megjegyzés: érdemes “interfészekben gondolkodni”!

Interfészek

- Az interfészek esetében is beszélhetünk öröklődésről, egyik interfész lehet egy másik interfész leszármazottja. Itt nincs megszorítás a többszörös örökléssel kapcsolatban (mivel interfészek szintjén nem léphet fel a gyémánt öröklődésből származó probléma).
  - Alkalmazás példa: egy interfésznek már léteznek megvalósításai és ki szeretnénk egészíteni további metódusokkal. Az összes megvalósító osztályt módosítanunk kellene. Jobb megoldás lehet a származtatás (a származtatott interfészben kapnak helyet az új metódusok).
- Észrevétel: az interfészek megvalósításánál a „szerződés” betartása, csak azt jelenti, hogy az interfész metódusait a megvalósító osztály implementálja, tehát egy az osztály példányára mutató referencia segítségével ezek a metódusok meghívhatóak. Ez a „vállalás” az implementációs részletekkel, a megvalósítás hogyanjával kapcsolatban semmiféle garanciát nem jelent. Például, az is lehetséges, hogy egy adott interfészt megvalósító osztály valamelyik metódust olyan módon implementálja, hogy kivételt dob a metóduson belül, vagy egyszerűen üresen hagyja a metódus törzsét.

Beágyazott osztályok

- Belső osztály (inner class)

```
class A {

  A(){}

  class B {

    B() {}

    public void doIt() {
      System.out.println("Hello");
    }

  };

}

public class Example {

    public static void main(String[] args) {
      A a = new A();
      A.B b = a. new B();
      b.doIt();
    }

}
```

- osztályok deklarálása más osztályok belsejében
- nem tartalmazhatnak statikus metódusokat, kivéve a statikus beágyazott osztályokat
- hozzáférnek a külső osztály minden adattagjához és metódusához (a privátokhoz is)
- a nem statikus belső osztályok példányosítását minden esetben a külső osztály példányosítása előzi meg

Beágyazott osztályok

- Statikus beágyazott osztály (static nested class):

```
class A {

  A() {}

  static class B {

    B() {}

    public static void doIt() {
      System.out.println("Hello");
    }

  };

}

public class Example {

  public static void main(String[] args) {
    A.B.doIt();
  }

}
```

Beágyazott osztályok

- Név nélküli belső osztályok (anonymous inner classes):

```
...
addWindowListener(new WindowAdapter() {

    public void windowClosing(WindowEvent e) {
      System.exit(0);
    }

});
...
```

Object ősosztály

- `getClass()`: az objektum típusának futási időben történő meghatározása. Egy Class típusú objektumot térít vissza, amelynek segítségével az osztályról kérhetünk különböző információkat.
- `toString()`: az objektum egy szöveges reprezentációját téríti vissza. Az Object osztály implementációja az osztály nevét és a hash kódot fűzi egybe, de természetesen ez a metódus is újradefiniálható.
- `finalize()`: az objektum által foglalt memóriaterület felszabadítása előtt hívja meg a szemétgyűjtő. Az Object osztály nem ad implementációt erre a metódusra. A származtatott osztályokban tipikusan az objektum által foglalt erőforrások felszabadítására alkalmazzák.

Object ősosztály

- `equals()`: az objektumok egyenlőségét vizsgálja. Általában a metódust a tartalom, az állapot összehasonlítására szokás használni. Bár az Object osztály alapértelmezett implementációja csak a referenciák azonosságát vizsgálja, a származtatott osztályokban ez a metódus általában olyan módon van újradefiniálva, hogy mélyebb, tartalmi összehasonlításra adjon lehetőséget. Például String objektumok esetében akkor fog igaz eredményt (true értéket) adni, ha a karakterláncok azonos karakterekből állnak.
- `hashCode()`: egy egész értéket, az objektum hash kódját téríti vissza, amely gyakran szükséges, amikor az objektumokat hasító táblákban tároljuk. A metódus a kódot a példány aktuális állapotának függvényében képezi. Ha két objektum állapota azonos (tartalma megegyezik), hash kódjuk is megegyezik. Következményként, ha újradefiniáljuk az equals metódust, akkor általában a hashCode metódust is újra kell definiálnunk. Az Object osztály alapértelmezett metódusa a memóriacím alapján képezi a kódot. A metódus újradefiniálásánál fontos, hogy minden olyan adattagot felhasználjunk a kód generálásához, amelyet az equals metóduson belül felhasználtunk az összehasonlításhoz.

Object ősosztály

- `clone()`: másolat készítése az osztály egy már létező példányáról. A metódus fejléce:

```
protected Object clone() throws CloneNotSupportedException
```

- A másolat készítése csak akkor valósítható meg, ha az osztály implementálja a `Cloneable` interfészt. Az Object ősosztály ezt nem teszi meg. Ha egy az interfészt nem implementáló osztály példányáról szeretnénk a clone metódushívás segítségével másolatot készíteni, a `CloneNotSupportedException` típusú kivételt kapjuk futási időben.
- Ha a másolat elkészíthető, akkor egy az eredeti objektummal megegyező állapotú új objektumot kapunk eredményül. Megjegyzendő, hogy az adattagokról nem készül másolat (ezek nem lesznek „klónozva”), így alapértelmezetten a metódushívás egy sekély másolást (shallow copy) eredményez, nem készül mély másolat (deep copy). Természetesen a clone metódus újradefiniálható, és így mély másolat is készíthető.
- `wait()`, `notify()`, `notifyAll()` - később tárgyaljuk (végrehajtási szálak, szinkronizálás)

Csomagok

- Osztályok és interfészek csoportja
- jar tömörítő
- Létrehozás:

```
package mypackage;
public class MyBaseClass {
  ...
}

package mypackage;
public class MyDerivedClass extends MyBaseClass {
  ...
}
```

- Használat:

```
java.awt.Button b;
mypackage.MyBaseClass ob;
mypackage.MyDerivedClass od;
```

- Vagy:

```
import java.awt.Button;
import mypackage.MyBaseClass;
import mypackage.MyDerivedClass
```

- Vagy:

```
import java.awt.*;
import mypackage.*;
```

Feladat, példa