Programozási Ismeretek

Dr. Gergely Tamás
Dr. Ferenc Rudolf, Dr. Jász Judit, Dr. Kiss Ákos

Szegedi Tudományegyetem
Informatikai Intézet
Szoftverfejlesztés Tanszék

2024

(v0214)

Tartalom

Tartalom

Polimorfizmus

- A harmadik lényeges OOP tulajdonság az adatabsztrakció és öröklődés után
- Interfész és implementáció
  - Szétválasztja a „*mit?*” a „*hogyan?*”-tól.
  - Interfész – *Mit?*
    - Az objektum milyen üzeneteket fogadhat, mire tud reagálni?
  - Implementáció – *Hogyan?*
    - Az objektum hogyan reagál az (általa ismert) üzenetekre?

Polimorfizmus

Példa

```
class Hang {
    private int magassag;
    private Hang(int m) {
        magassag = m;
    }
    public String toString() {
        return "" + magassag + "Hz";
    }
    public static final Hang
        C = new Hang(262),
        D = new Hang(294),
        E = new Hang(330),
        F = new Hang(349),
        G = new Hang(392),
        A = new Hang(440),
        H = new Hang(494);
}
```

```
class Hangolo {
    public static void hangolj(Hangszer h) {
        h.szolj(Hang.C);
    }
}
```

```
class Hangszer {
    public void szolj(Hang h) {
        System.out.println("A hangszer "
                + h + "-es hangon szól.");
    }
}
```

```
class Zongora extends Hangszer {
    public void szolj(Hang h) {
        System.out.println("A zongora "
                + h + "-es hangon szól.");
    }
}
```

```
public class HangszerPelda {
    public static void main(String[] args) {
        Hangszer h = new Hangszer();
        Hangolo.hangolj(h);
        Zongora z = new Zongora();
        Hangolo.hangolj(z); // upcasting
    }
}
```

Hangszer.szolj()

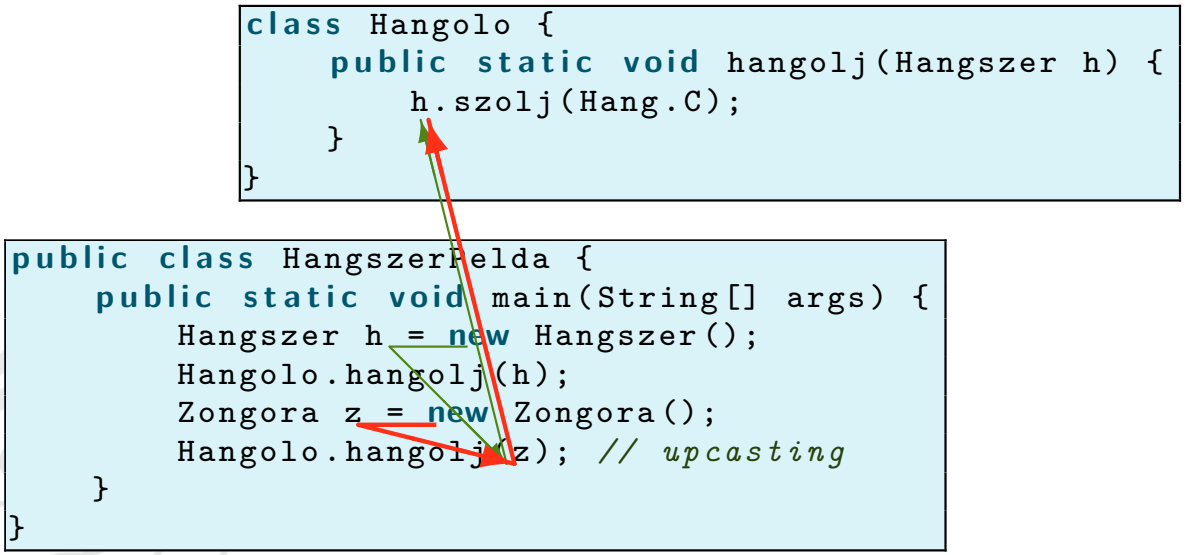
Zongora.szolj()

„Elfelejteni” a típust?

- Az előző példában a `Hangolo.hangolj(z)` hívás során „elveszik” a típus, mert a metódus `Hangszer` típusú objektumot vár `Zongora` tíousú objektum van átadva
- Egyszerűbb lenne, ha a `hangolj` metódus `Zongora` típusú objektumot várna?
  - Nem, mert akkor minden egyes hangszerre külön `hangolj()` metódust kellene írni,
  - sőt, ha új hangszerrel szeretnénk bővíteni a rendszert, akkor is mindig új metódust kellene írni (ez a karbantarthatóság rovására megy)!

Kései kötés

- Honnan tudja a fordító, hogy melyik `szolj()` metódust hívja?
  - A fordító **NEM** tudja!
  - Csak futás közben derül ki (kései kötés) az **objektum típusa** alapján

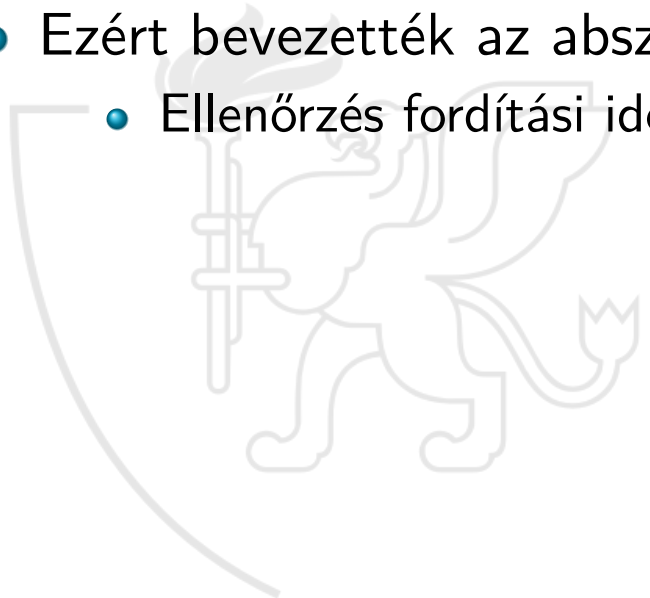


- A polimorfizmusnak köszönhetően tetszőleges számú új hangszert felvehetünk a `hangolj()` metódus megváltoztatása nélkül
- Egy jól megtervezett OO programban ezt a modellt követik, vagyis az objektumok az interfészek segítségével kommunikálnak egymással

Tartalom

Absztrakt osztályok és metódusok

- Az előző példában a `Hangszer` ősosztályban szereplő metódusok valójában „álmetódusok”
  - ha ezek véletlenül meghívódnak, akkor valami hiba van a programban, mert ezeknek a szerepe az, hogy interfész legyen
  - lehetne programfutás közben kiíratni, hogy hiba van, de ennek az ellenőrzéséhez sok tesztelés szükséges
  - magából az ősosztályból valójában nem kellene, hogy létrejöjjön objektum
- Ezért bevezették az absztrakt osztályokat és metódusokat
  - Ellenőrzés fordítási időben történik



Absztrakt osztályok és metódusok
(folyt.)

- Kulcsszó: `abstract`
- Absztrakt osztály:

```
abstract class Hangszer { /* ... */ }
```

  - nem példányosítható (fordítási hiba)
  - közös interfészt biztosít a leszármazottaknak
- Absztrakt metódus:

```
abstract public void szolj(Hang h);
```

  - csak deklarációja van (nincs definíciója – törzse)
- Ha egy osztálynak van legalább egy absztrakt metódusa, akkor neki is absztraktnak kell lennie
- Egy osztály lehet absztrakt akkor is, ha nincs absztrakt metódusa

Absztrakt osztály

Példa

```
class Hang {
    private int magassag;
    private Hang(int m) {
        magassag = m;
    }
    public String toString() {
        return "" + magassag + "Hz";
    }
    public static final Hang
        C = new Hang(262),
        D = new Hang(294),
        E = new Hang(330),
        F = new Hang(349),
        G = new Hang(392),
        A = new Hang(440),
        H = new Hang(494);
}
```

```
class Hangolo {
    public static void hangolj(Hangszer h) {
        h.szolj(Hang.C);
    }
}
```

```
abstract class Hangszer {
    abstract public void szolj(Hang h);
}
```

```
class Zongora extends Hangszer {
    public void szolj(Hang h) {
        System.out.println("A␣zongora␣"
                + h + "-es␣hangon␣szól.");
    }
}
```

```
public class AbstractHangszerPelda {
    public static void main(String[] args) {
        //Hangszer z = new Hangszer();
        Hangszer z = new Zongora(); // upcasting
        Hangolo.hangolj(z);
    }
}
```

fordítási hiba!

Zongora.szolj()

Futás közbeni típusazonosítás

- Run-time Type Identification (RTTI)
- Futás közben megállapítható, hogy egy ősosztály típusú referencia milyen konkrét típusú objektumra hivatkozik
- Java-ban a „hagyományos” RTTI-t bővítették egy ún. „`reflection`” mechanizmussal
  - akár egy a fordító számára ismeretlen osztály felépítése is lekérhető

Futás közbeni típuskonverzió

- *Upcast* (ősre konvertálás)
  - elveszítjük a konkrét típust
  - a konverzió biztonságos
- *Downcast* (leszármazottra konvertálás)
  - jó lenne visszanyerni a konkrét típust
  - a konverzió egyéb nyelvekben nem biztonságos
  - Java-ban ellenőrizve van (RTTI): Ha nem megfelelő típusra próbálunk downcast-olni akkor egy `ClassCastException` típusú kivétel dobódik

Objektumtípus ellenőrzés

- Az `instanceof` kulcsszó segítségével futásidőben ellenőrizhető, hogy egy objektum adott típusú-e vagy sem
- A „leszármazott típus ős típusú is” elv alapján működik

```
public class AlakzatTombPelda {
    public static
    void main(String[] args) {
        Alakzat a[] = new Alakzat[3];
        a[0] = new Kor();
        a[1] = new Haromszog();
        a[2] = new Negyzet();
        for (int i = 0; i < a.length; ++i) {
            Object o = a[i];          // upcast
            if (o instanceof Kor) {
                ((Kor)o).rajzolj(); // downcast
            }
        }
    }
}
```

Kor.rajzolj()

Tartalom

Interfészek

- Interfész osztály: teljesen absztrakt osztály
  - A Java interfész osztályok egy interfészt adnak meg, implementáció nélkül
  - Egy protokollt adnak meg az osztályok között
  - Lehetővé teszik a „többszörös öröklődés”-t
- Kulcsszó: `interface`
- Metódus deklarációk:
  - törzs nélkül (kivéve a `default` metódusokat Java 8-tól)
  - impliciten `public`
- Mezők:
  - impliciten `public`, `static` és `final`
- Az `implements` kulcsszóval lehet implementálni („származtatni” belőle)

Interfész

Példa

```
class Hang {
    private int magassag;
    private Hang(int m) {
        magassag = m;
    }
    public String toString() {
        return "" + magassag + "Hz";
    }
    public static final Hang
        C = new Hang(262),
        D = new Hang(294),
        E = new Hang(330),
        F = new Hang(349),
        G = new Hang(392),
        A = new Hang(440),
        H = new Hang(494);
}
```

```
class Hangolo {
    public static void hangolj(Hangszer h) {
        h.szolj(Hang.C);
    }
}
```

```
interface Hangszer {
    void szolj(Hang h);
}
```

impliciten public

```
class Zongora implements Hangszer {
    public void szolj(Hang h) {
        System.out.println("A zongora "
                + h + "-es hangon szól.");
    }
}
```

```
public class InterfaceHangszerPelda {
    public static void main(String[] args) {
        Zongora z = new Zongora();
        Hangolo.hangolj(z); // upcasting
    }
}
```

Zongora.szolj()

Interfészek
(folyt.)

- Interfész vagy absztrakt osztály?
  - ha ősosztályt készítünk, akkor mindig kezdjük azzal, hogy interfész lesz
  - ha mégis szükség lesz az ősosztály szintjén metódusokat implementálni vagy mezőket hozzáadni akkor módosítsuk absztrakt osztályra (esetleg konkrét osztályra)

„Többszörös öröklődés”

- Mivel az interfészek nem foglalnak tárterületet, lehetővé válik a biztonságos többszörös öröklődés
- Nem igazi többszörös öröklődés (mint pl. a C++-ban)
- Maximum egy darab „igazi” osztály (`class`) lehet az ősök között, a többi interfész kell legyen
- Interfész osztályok is származhatnak egymásból (`extends`), ezzel új interfészt kapunk

„Többszörös öröklődés”

Példa

```java
interface Kuzd {
    void kuzdj();
}
```

```java
interface Uszik {
    void usszal();
}
```

```java
interface Repul {
    void repulj();
}
```

```java
class Szereplo {
    public void kuzdj() {
        System.out.println("A szereplő küzd!");
    }
}
```

```java
class Akciohos extends Szereplo
                implements Kuzd, Uszik, Repul {
    public void usszal() {
        System.out.println("Az akcióhős úszik!");
    }
    public void repulj() {
        System.out.println("Az akcióhős repül!");
    }
}
```

```java
public class AkciohosPelda {
    public static void main(String[] args) {
        Akciohos ah = new Akciohos();
        ah.kuzdj();
        ah.usszal();
        ah.repulj();
    }
}
```

Tartalom

Konstansok csoportosítása

- Mivel a mezői automatikusan `static final`-ok, az `interface` a Java 1.5 verzió előtt jó eszköz volt arra, hogy konstansokat csoportosítsunk

```
public interface HonapInterface {
    int JANUARY = 1,   FEBRUARY = 2,
        MARCH = 3,     APRIL = 4,
        MAY = 5,       JUNE = 6,
        JULY = 7,      AUGUST = 8,
        SEPTEMBER = 9, OCTOBER = 10,
        NOVEMBER = 11, DECEMBER = 12;
}
```

```
public final class HonapInterfacePelda {
    public static void main(String[] args) {
        System.out.println(HonapInterface.MAY);
    }
}
```

Az enum

- Az 1.5 verziótól Java-ban is létezik az `enum`
- Ez valójában egy speciális osztálydeklaráció nyelvi támogatással, így használható például `switch`-ben is

```
public enum HonapEnum {
    JANUARY(31), FEBRUARY(28), MARCH(31),
    APRIL(30),   MAY(31),      JUNE(30),
    JULY(31),    AUGUST(31),   SEPTEMBER(30),
    OCTOBER(31), NOVEMBER(30), DECEMBER(31);

    private final int days;

    HonapEnum(int days) {
        this.days = days;
    }
    int numberOfDays() {
        return this.days;
    }
}
```

```
public final class HonapEnumPelda {
    public static void main(String[] args) {
        for (HonapEnum h : HonapEnum.values()) {
            System.out.print("Month␣"
                + (h.ordinal() + 1) + "␣is␣"
                + h + ",␣and␣it␣has␣"
                + h.numberOfDays());
            switch (h) {
                case DECEMBER:
                case JANUARY:
                case FEBRUARY:
                    System.out.print("␣frosty");
                default:
                    System.out.println("␣days.");
                    break;
            }
        }
    }
}
```

Tartalom

Belső osztályok

- Osztályon vagy metóduson belül deklarált osztályok
  - logikailag összetartozó osztályok csoportosítása
  - megadható a láthatóságuk
  - máshol nem használt algoritmus elrejtése a külvilágtól
  - a belső osztályból elérhetőek a „körülvevő” osztály elemei (kivéve ha `static` a belső osztály)
  - `.class` fájlnév: `OuterClass$InnerClass.class`
- **Nem** kompozíció!

Belső osztályok

Példa

```
public class Csomag {
    class Tartalom {
        private String tartalom = "";
        public Tartalom(String tartalom, int suly) {
            if (suly <= sulykorlat) {
                this.tartalom = tartalom;
            } else {
                this.tartalom = "---";
            }
        }
        public String miVanBenne() {
            return tartalom;
        }
    }
    class Cel {
        private String cim;
        Cel(String cim) {
            this.cim = cim;
        }
        String miACim() {
            return cim;
        }
    }
    private int sulykorlat;
    public Csomag(int sulykorlat) {
        this.sulykorlat = sulykorlat;
    }
    public void felad(String cel,
                      String mit, int suly) {
        Cel c = new Cel(cel);
        Tartalom t = new Tartalom(mit, suly);
        System.out.println("Cím: "
                           + c.miACim());
        System.out.println("Tartalom: "
                           + t.miVanBenne());
    }
    public static void main(String[] args) {
        Csomag cs = new Csomag(20);
        cs.felad("Tanzánia", "Elefánt", 5000);
    }
}
```