MILTON FRIEDMAN EGYETEM

PROGRAMOZÁSI TECHNOLÓGIÁK

1. óra - Bevezetés

2024.03.18

Szalai Patrik

szalai.patrik@uni-milton.hu

TÉMAKÖRÖK:

Félév információk

- Tankönyv
- Előadások
- Gyakorlat
- Vizsgázás
- Tesztlabor
- Kidolgozások
- Eszközök

1, Bevezetés

- Ismétlés gyakorlat
- OOP Alapelvek
- GOF1
- GOF2

FÉLÉV INFORMÁCIÓK

- **Tankönyv:** Kusper Gábor – Programozási technológiák
- **Gyakorlat:** Könyvben található példák és esetek bemutatása (C# és Java)
- **Eszközök:**
  Github (+ Desktop kliens), Preferált szövegszerkesztő, Visual Studio (. NET IDE)
- **Tesztlabor:** WIP
  Hasznos linkek, gyakorlati anyag github gyűjtemény, előadások anyaga (.pdf)

Tankönyv

Eszközök

FÉLÉV INFORMÁCIÓK

- **Tételsor: „PPP”**
  1, Principles – OO Tervezési alapelvek
  2, Practices – Jó gyakorlatok
  3, Patterns – Programtervezési minták

- **Beadandók:**
  1, Egy beadandó esszék – TBD
  2, Git-en Fork-ok, feladatok a félév során

- **Kötelező olvasmány:**
  Robert C. Martin „Uncle Bob” – Clean Code

- **Vizsga/Beugró:**
  TBD

A tárgy célja

- Alkalmazott tudás elsajátítása
- Alapelvek ismerete
- Tanultak gyakorlása kódban
- Munka fejlesztői eszközökkel

ISMÉTLÉS

- **Forráskód vizsgálata a félév során**
  - Az osztály felülete
  - Az osztály megvalósítása
- **Az osztály Felülete**
  - Interface
  - Minden hozzá tartozik, ami publikus
  - Alapvetően a Publikus metódusok alkotják
- **Az osztály Megvalósítása**
  - Implementation
  - Az osztály forráskódja
- **Absztrakció**
  - Elvonatkoztatás
  - Lényeges és lényegtelen tulajdonságok ELVÁLASZTÁSA
  - Lényeges tulajdonság kiemelése, lényegtelen figyelmen kívül hagyása
  - Ez **nem egyenlő az OOP Absztrakció alapelvével**, de fontos ismerni és alkalmazni a jelentést

Absztrakt osztályok

- Van Felülete
- DE Megvalósítása csak részleges

Konkrét osztályok

- Van Felülete
- Megvalósítása teljes

ISMÉTLÉS

- **Objektumok vizsgálata a félév során**
  - Az objektum felülete (típusa)
  - Az objektum viselkedése
  - Az objektum belső állapota
- **Az objektum Felülete**
  - Minden objektumnak van típusa
  - A típus meghatározza az objektum felületét is
  - Típustól függ, hogy milyen metódusokat tudok hívni
- **Az objektum Viselkedése**
  - Behavior
  - A futás közbeni forráskód
- **Az objektum Belső állapota**
  - Inner state
  - A mezők pillanatnyi értéke
  - Kezdő belső állapotot a konstruktor hozza létre

Futó forráskód

- Forráskód dinamikus vetülete

Megvalósítás

- Forráskód statikus vetülete

Különbség!
Pl.: if - else

Egységbezárás

- Belső állapotot kívülről ne lehessen módosítani
- Csak az osztály metódusai tudják változtatni!

Encapsulation

VISELKEDÉS ÉS MEGVALÓSÍTÁS

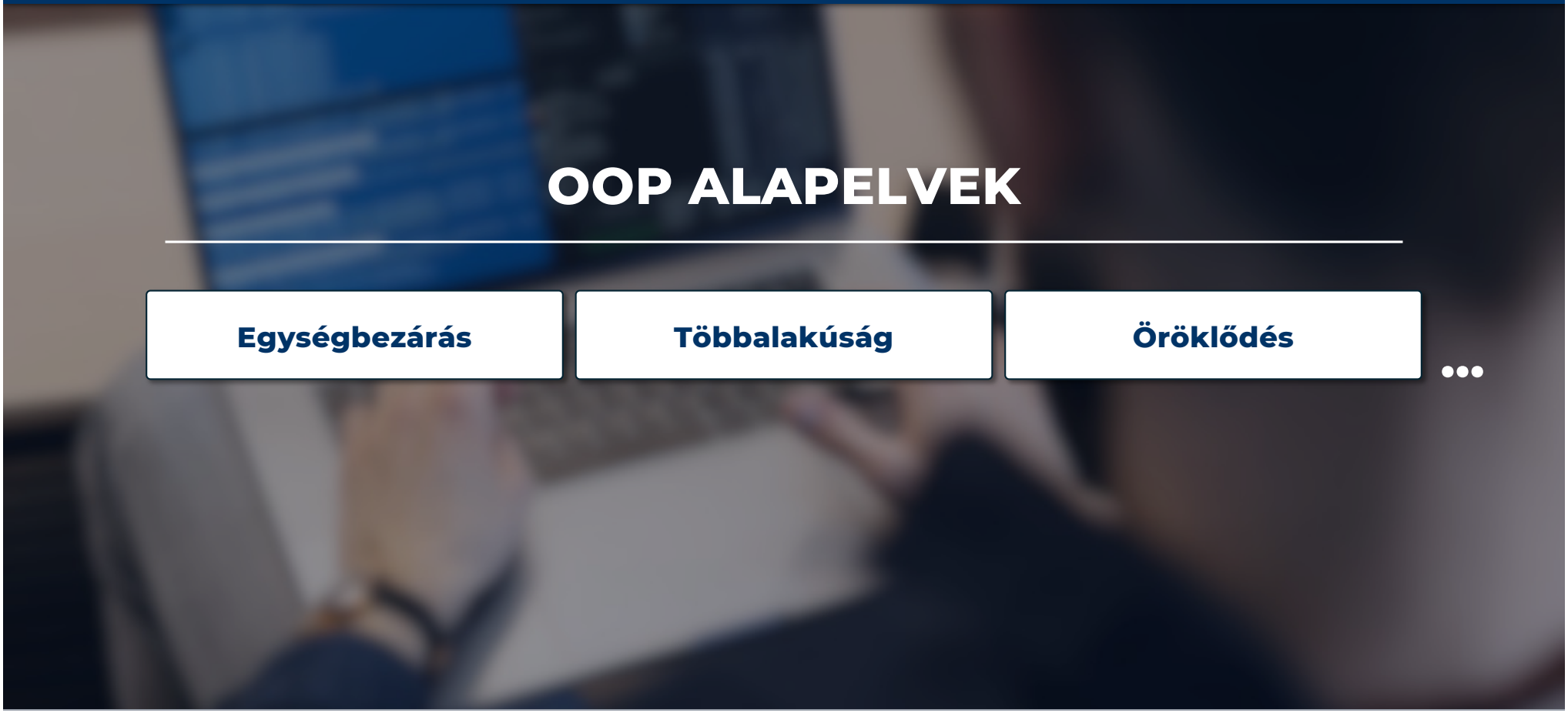
Futó forráskód

- Forráskód dinamikus vetülete

Megvalósítás

- Forráskód statikus vetülete

```
//C# Példa 1 – Dinamikus vs Statikus forráskód vetület
class Utlevel {
        bool ervenyes;
        public void SetErvenyesseg(bool ervenyes){
                this.ervenyes=ervenyes;
        }
        public string Valid(){
                if (ervenyes) return "Az útlevél érvényes.";
                else return "Az útlevél érvényessége lejárt."
        }
}
```



OOP ALAPELVEK

- Egységbezárás
- Többalakúság
- Öröklődés
- ...

OOP ALAPELVEK

- **Egységbezárás**
  - Az objektum belső állapota legyen megváltoztathatatlan
  - Lehetőleg NE használjunk publikus mezőket
  - Lehetőleg ne adjunk vissza olyan referenciát, mely egy ilyen mezőre mutat

```
// Java Példa 2 – Encapsulation sértés
class Kutya {
        ArrayList<String> nevek = new ArrayList<>();
        public ArrayList<String> getNevek() { return nevek; }
}

// Példa program
main() {
        Kutya k1 = new Kutya();
        k1.getNevek().add("Bobikutya");
}
```

- A ‚nevek' ArrayList közvetlenül publikálva van a külvilág számára a getNevek metódus által „Exposed/Kitett"
- Ezáltal külső kód által közvetlenül módosíthatóvá válik a 'Kutya' objektum

OOP ALAPELVEK

- **Egységbezárás – Javítsunk a kódon!**

```
// Javított példa
class Kutya {
        private ArrayList<String> nevek = new ArrayList<>();
        public void addNev(String név) { nevek.add(név); }
}

// Példa program
main() {
        Kutya k1 = new Kutya();
        k1.addNev("Bobbikutya");
}
```

- Egységbezártuk a ’nevek’ ArrayList-et a 'Kutya’ osztályba
- Ezt úgy értük el, hogy nem tettük nyilvánossá a ’nevek’ ArrayList-et, hanem helyette előre meghatározott metódust adtunk ‚addNev()’ a szabályozott működés betartatására.
- **Külső kód továbbra is interakcióba tud lépni a 'Kutya’ objektummal, de csak az általunk biztosított metódusokkal szerkesztheti azt, nem közvetlenül.**
- **Publikus szolgáltatást hoztunk létre.**

OOP ALAPELVEK

Egységbezárás – Szolgáltatás

```
// Szolgáltatás példa
class Kutya {
    private ArrayList<String> nevek = new ArrayList<>(); // Továbbra is private!
    public ArrayList<String> getNevek() { return nevek; } // Visszaraktuk a hibát a példába!
    public void addNev(String név) {
        if (!név.equals("Adolf")) nevek.add(név);
        // Vizsgálat: Nem lehet a kutya neve Adolf.
    }
}

// Szolgáltatás példa program
main() {
    Kutya k1 = new Kutya();
    k1.getNevek().add("Adolf"); // Megszegi az egységbezárást, így kikerülheti a vizsgálatot.
    k1.addNev("Adolf"); // Nem szegi meg, ezért működési logika szerint el lesz utasítva.
}
```

- **Hiába private** a 'nevek' ArrayList, a 'getNevek' továbbra is vissza ad referencát rá!
- Az egységbezárás betartása nélkül „lyukas" a vizsgálatunk, könnyen megkerülhető.

OOP ALAPELVEK

- **Egységbezárás – Szolgáltatás**
  - Publikus metódusokat hívhatunk szolgáltatásnak.
  - Fontos megjegyezni, hogy Objektum orientált programozási nyelvek között van olyan is, amiben nincs metódushívás, hanem helyette üzenetküldés van, ilyen például a Smalltalk.

```
// Szolgáltatás, mellyel hozzáadható új Kutya név
class Kutya {
    private ArrayList<String> nevek = new ArrayList<>();
    public void addNev(String név) {
        nevek.add(név);
    }
}

// Szolgáltatás példa program
main() {
    Kutya k1 = new Kutya();
    k1.addNev(”Bobbikutya");
}
```

OOP ALAPELVEK

- **Egységbezárás**
  - Az objektum belső állapota legyen megváltoztathatatlan
    - Csak az osztály metódusai változtathassák
  - Lehetőleg NE használjunk publikus mezőket
  - Lehetőleg ne adjunk vissza olyan referenciát, mely egy ilyen mezőre mutat

OOP ALAPELVEK

- **Többalakúság**
  - Polymorphism
  - Ha van egy objektumom, akkor annak több típusa lehet
  - Az objektumot bármelyik típusán keresztül lehet nézni

```
// C# Példa 3 - Többalakúság
class Allat{}
class Gerinces:Allat{}
class Macska:Gerinces{}
class HaziMacska:Macska{}

HaziMacska h1 = new HaziMacska(); // Helyes egyszerű példa.

Macska m1 = new HaziMacska();
HaziMacska h2 = new Macska();
```

- **Feladat:** Melyik sor helyes a kódunkban?

OOP ALAPELVEK

- **Többalakúság**
  - Polymorphism
  - Ha van egy objektumom, akkor annak több típusa lehet
  - Az objektumot bármelyik típusán keresztül lehet nézni

```
// C# Példa 3 – Többalakúság
class Allat{}
class Gerinces:Allat{}
class Macska:Gerinces{}
class HaziMacska:Macska{}

HaziMacska h1 = new HaziMacska(); // Helyes egyszerű példa.

Macska m1 = new HaziMacska(); // Helyes példa az öröklődési lánc szerint.
HaziMacska h2 = new Macska(); // Ez nem helyes, mivel az öröklődési lánc szerint a Macskának NINCS
Házimacska típusa!
```

- **Megoldás:** Csak az első sor, mégpedig az öröklődési lánc miatt.

OOP ALAPELVEK

Többalakúság - Öröklődés

```java
// Java-ban a Példa 3 – Többalakúság
class Allat extends Object{} // Nem kell kiírni, mert automatikus!
class Gerinces extends Allat{}
class Macska extends Gerinces{}
class HaziMacska extends Macska{}

HaziMacska h1 = new HaziMacska();
```

- Java-ban „extends” a szintaxisa az öröklődésnek.
- **Az öröklődési láncon önmagát ÉS felfelé minden altípust megkap!**

A h1-típusai tehát:

- Object
- Allat
- Gerinces
- Macska
- HaziMacska

OOP ALAPELVEK

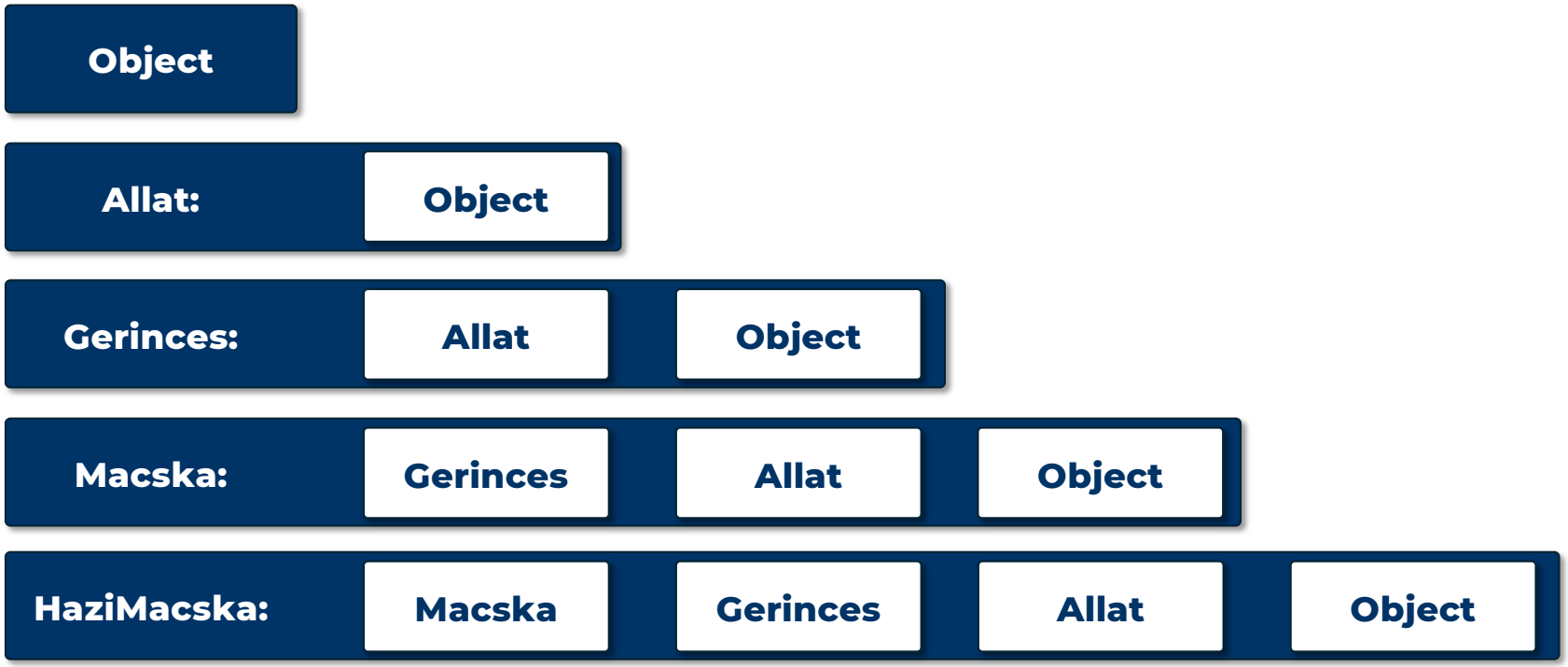
Többalakúság - Öröklődés

```
// C# Példa 3 – Többalakúság
class Allat{}
class Gerinces:Allat{}
class Macska:Gerinces{}
class HaziMacska:Macska{}

HaziMacska h1 = new HaziMacska(); // Helyes egyszerű példa.

Macska m1 = new HaziMacska(); // Helyes példa az öröklődési lánc szerint.
HaziMacska h2 = new Macska(); // Ez nem helyes, mivel az öröklődési sorrend szerint a Macskának NINCS
Házimacska típusa!
```

Öröklődési lánc:

OOP ALAPELVEK

Többalakúság - Öröklődés



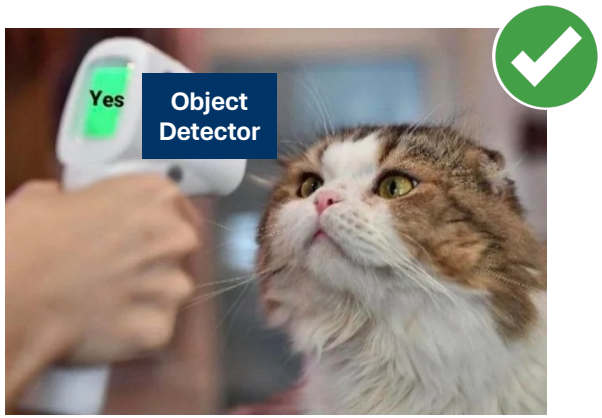
OOP ALAPELVEK

Többalakúság

- Egy objektumnak több típusa is van (az öröklődési láncon felfelé mindegyiket megkapja)
- Ezen típusok közül bármelyikként használható

Példa: A HáziMacska használható Macskaként, mert:

1. A Házimacska egy Macska
2. A Macska egy Gerinces
3. A Gerinces egy Állat
4. Az Állat egy Objektum



OOP ALAPELVEK

- **Többalakúság – „IS-A”**

  Példa: A HáziMacska használható Macskaként, mert:

  1. A Házimacska egy Macska
  2. A Macska egy Gerinces
  3. A Gerinces egy Állat
  4. Az Állat egy Objektum

- **Az öröklődés másik neve IS-A kapcsolat.**

  Példa: A HáziMacska használható Macskaként, mert:

  1. Házimacska **IS A** Macska
  2. A Macska **IS A** Gerinces
  3. A Gerinces **IS A** Állat (class)
  4. Az Állat **IS A** Objektum (base class)

- **Az IS-A kapcsolatnak 2 fajtája van:**
  - Öröklődés (inheritance) – Java kulcszava: 'extends’
  - Interface megvalósítás – Java kulcsszava: implements
    - C#-ban mindkettőt kettősponttal jelöljük!

IS-A KAPCSOLAT

- **Az IS-A kapcsolatnak 2 fajtája van:**
  - Öröklődés (inheritance)
  - Interface megvalósítás

```
// Öröklődés
class A { }
class B extends A { }
// B osztály az A-ból ered. B IS-A A
```

```
// Interface megvalósítás
interface InterfaceA { }
class Osztaly1 implements InterfaceA { }
// Osztaly1 IS-A InterfaceA
```

- **Öröklődés:**
  - Osztályok közötti kapcsolat
  - Az alosztály örökölheti az ősök tulajdonságait és viselkedését (metódusait)
  - Az **alosztály IS-A ősosztály**
- **Interface megvalósítás:**
  - Akkor fordul elő, mikor egy osztály egy- vagy több interface-t implementál
  - Az interface egy kapcsolatot definiál, mely leírja, hogy milyen metódusokat kell adniuk az implementációt alkalmazó osztályoknak
  - Az **implementációt alkalmazó osztály IS-A interface**

INTERFACE MEGVALÓSÍTÁSA

- **Interface**
- Egy „leírat/tervrajz/szerződés”, mely definiál egy szett metódust, melyet az interface-t implementáló osztálynak meg kell adnia.
- Nem tartalmaznak metódus megvalósítást, csak metódus szignatúrákat deklarálnak.

- **Az Interface-t „megvalósító” osztály**
- Mikor egy osztály implementál egy interface-t, elfogadja az interface által definiáltakat.
- Az osztály adja a megvalósítást az interface által leírt összes metódusra.
- Az interface-t adó osztály így „valósítja meg” vagy „implementálja” az interface-t.

```
// Az interface
interface Allat {
    void eszik();
    void alszik();
}


// Az osztály, mely „Megvalósítja” az interface-t
class Kutya implements Allat {
    @Override
    public void eszik() {
        System.out.println("A kutya eszik.");
    }
    @Override
    public void alszik() {
        System.out.println(”A kutya alszik.");
    }
}
```

INTERFACE-EK

```
interface ITudUgatni {
}
```

- **Létre hoztunk egy interface-t. Na de minek tettünk elé „I” betűt?**
  - Hungarian notation
  - Itt most CamelCase-t alkalmaztunk
- **Hungarian notation**
  - Charles Simonyi
  - A változó nevéből kitalálható a változó típusa.
    - Pl.: Minden int típusú változót **i** betűvel kezdek
    - Pl.: Minden pointer változó nevét **p** betűvel kezdek.
    - C#-ban „illik” az interface nevét **I** betűvel kezdeni.

INTERFACE-EK

```
interface ITudUgatni {
        String ugat(); // public abstract
}
```

- **Public Abstract**
  - Interface-en belül minden publikus és minden absztrakt
  - Ezért nem kötelező ezeket kiírni

Osztály	Interface	Absztrakt Osztály
Van Felülete	Van Felülete	Van Felülete
Van Megvalósítása	Nincs Megvalósítása	Van Megvalósítása
	Mivel minden Absztrakt	A Megvalósítás lehet részleges, vagy teljesen hiányozhat is, ha minden metódusa absztrakt

INTERFACE-EK

- **Absztrakt Osztály**
  - Arra használjuk, hogy ő legyen a hierarchia őse
  - A hierarchia tetején vannak
    - Ezek az Absztrakt ősök
    - Megfogalmazzák, hogy milyen szolgáltatásokat nyújtunk
    - Nem dolgozzák ki a szolgáltatásokat, azokat a gyermek osztályok teszik
  - Emlékezzünk vissza, mit jelent az Absztrakció

Absztrakt Osztály

- Van Felülete
- Van Megvalósítása
- A Megvalósítás lehet részleges, vagy teljesen hiányozhat is, ha minden metódusa absztrakt

INTERFACE-EK

- **Egy Macska „szolgáltatásai”**
  - Kódoljunk macska viselkedést... induljunk egy tarisznyarákból

Absztrakt Osztály

- Van Felülete
- Van Megvalósítása
- A Megvalósítás lehet részleges, vagy teljesen hiányozhat is, ha minden metódusa absztrakt

INTERFACE-EK

- **Egy Macska „szolgáltatásai”**
  - Kódoljunk macska viselkedést

```
// C# Példa 4 – Macska szolgáltatásai
abstract class Macska {
        public abstract String dorombol();
        public abstract bool eszikE();
}

// Java Példa 4 – Macska szolgáltatásai Java-ban
class HaziMacska extends Macska {
        @Override
        public String dorombol() { return "Intense dorombolások"; }
        @Override
        public boolean eszikE() { return true; }
}

main() {
        HaziMacska h1 = new HaziMacska();
        Macska h2 = new HaziMacska();
}
```

INTERFACE-EK

- **GOF – Gang of Four**
  - Objektumorientált tervezési alapelvek (GOF1 és GOF2)
  - Design Patterns könyv

```
// Java Példa 4 – Macska szolgáltatásai Java-ban
class HaziMacska extends Macska {
        @Override
        public String dorombol() { return "Intense dorombolások"; }
        @Override
        public boolean eszikE() { return true; }
}

main() {
        HaziMacska h1 = new HaziMacska();
        Macska h2 = new HaziMacska();
}
```

- **GOF1:** Program to an Interface, not an Implementation
- **Megvalósításra programozni:** Ha egy osztály kódjában felhasználjuk egy másik osztály implementációját
  - **Haszna:** Gyors és rövid kódot eredményez általában
  - **Veszélye:** Ha megváltozik az egyik osztály, akkor a másik osztályt is meg kell változtatni
    - **Azaz:** Implementációs függőséget okoz!

INTERFACE-EK

GOF1

```java
// Java Példa 5 – GOF1
class HaziAllat {
    double súly;
    String fajta;
    public HaziAllat(String fajta) {
        this.fajta = fajta;
        if (fajta == "Kutya") súly = 5.0;
        else if (fajta == "Macska") súly = 3.0;
        else súly = 1.0;
    }
}
```

Feladat: Mi a hiba a kódban?

INTERFACE-EK

GOF1

```
class HaziAllat {
     double súly;
     String fajta;
     public HaziAllat(String fajta) {
          this.fajta = fajta;
          if (fajta.equals("Kutya")) súly = 5.0;
          else if (fajta.equals("Macska")) súly = 3.0;
          else súly = 1.0;
     }
}
// Oltás osztály
class Oltas {
     int alapPerKg = 3;
     public int getMiliLiter(HaziAllat x) {
          if (x.getFajta().equals("Kutya")) return alapPerKg * 5;
          else (x.getFajta().equals("Macska")) return alapPerKg * 3;
          ...
     }
}
```

Feladat: Mi a hiba programtervezés szempontjából?

INTERFACE-EK

GOF1

```
class HaziAllat {
    double súly;
    String fajta;
    public HaziAllat(String fajta) {
        this.fajta = fajta;
        if (fajta.equals("Kutya")) súly = 5.0;
        else if (fajta.equals("Macska")) súly = 3.0;
        else súly = 1.0;
    }
}
// Oltás osztály – HA felhaszbálom a fenti implementációt, akkor implementációs függőség alakul ki!
class Oltas {
    int alapPerKg = 3;
    public int getMiliLiter(HaziAllat x) {
        if (x.getFajta().equals("Kutya")) return alapPerKg * 5; // Csak addig jó a kód, amíg minden kutya 5.0
        kg
        else (x.getFajta().equals("Macska")) return alapPerKg * 3;
        ...
    }
}
```

A hiba: A két osztály között implementációs függőség alakult ki. Ha megváltozik a HaziAllat kódja, akkor valószínűleg az Oltas kódját is meg kell változtatni.

INTERFACE-EK

- **Írjuk át a kódot felületre programozva!**
- Bevezetünk egy absztrakt osztályt

```java
// Java Példa 6 – GOF1 felületre programozás
abstract class AbsHaziAllat {
    public abstract double getSúly(); // Szolgáltatás
    public abstract String getFajta();
    // Nem kell konstruktor, azt majd megcsinálja a child
}
// Oltás osztály – Felületre programozva
class Oltas {
    int alapPerKg = 3;
    public int getMiliLiter(AbsHaziAllat x) {
        return alapPerKg * x.getSúly();
    }
}
```

- **Itt nincs implementációs függőség!**
- **Kérdés:** Miért probléma az implementációs függőség?

INTERFACE-EK

GOF1

```java
// Java Példa 4 – Macska szolgáltatásai Java-ban
class HaziMacska extends Macska {
        @Override
        public String dorombol() { return "Intense dorombolások"; }
        @Override
        public boolean eszikE() { return true; }
}

main() {
        HaziMacska h1 = new HaziMacska();
        Macska h2 = new HaziMacska();
}
```

Kérdés: Melyik sor felel meg a GOF1-nek?

INTERFACE-EK

GOF1

```
// Java Példa 4 – Macska szolgáltatásai Java-ban
class HaziMacska extends Macska {
        @Override
        public String dorombol() { return "Intense dorombolások"; }
        @Override
        public boolean eszikE() { return true; }
}

main() {
        HaziMacska h1 = new HaziMacska();
        Macska h2 = new HaziMacska();
}
```

- **Kérdés:** Melyik sor felel meg a GOF1-nek?
- **Válasz:** A 2-es sor, hiszen a lehető legabsztraktabb típust használom.
  - Rugalmas marad a kód
  - Nem vagyok egy megvalósításhoz hozzáláncolva

- **Cseréljük le a HáziMacskát egy KóborMacskára**

INTERFACE-EK

GOF1

```
class HaziMacska extends Macska {
        @Override
        public String dorombol() { return "Intense dorombolások"; }
        @Override
        public boolean eszikE() { return true; }
}
class KoborMacska extends Macska { ... }
main() {
        HaziMacska h1 = new HaziMacska();
        Macska h2 = new HaziMacska();
        h1 = new KoborMacska();
        h2 = new KoborMacska();
}
```

Kérdés: Melyik új sor felel meg a GOF1-nek?

GOF 1

- **Alapjai**
  - A típusnak a lehető legabsztraktabb (legősibb) típust kell használni
  - Aminek a szolgáltatásai még jók!
    - Avagy, a programom rendeltetéséhez szükséges szolgáltatásokat tartalmazza

```
class HaziMacska extends Macska {
        @Override
        public String dorombol() { return "Intense dorombolások"; }
        @Override
        public boolean eszikE() { return true; }
}
class KoborMacska extends Macska { ... }
main() {
        HaziMacska h1 = new HaziMacska();
        Macska h2 = new HaziMacska();
        h1 = new KoborMacska();
        h2 = new KoborMacska();
}
```

GOF 1

Melyik sor helyes?

```java
// Java Példa 7 – GOF1 példák, szolgáltatások
abstract class Macska {
        public abstract String dorombol();
        public abstract boolean eszikE();
}
class HaziMacska extends Macska {
        public boolean alszikE() { return true; }
        @Override
        public boolean eszikE() { return true; }
        public String dorombol() { return "Intense dorombolások"; }
}
main() {
        HaziMacska h1 = new HaziMacska();
        Macska h2 = new HaziMacska();
}
```

- **1. Kérdés:** Hány szolgáltatása van a HáziMacskának?
- **2. Kérdés:** Melyik sor helyes a GOF1 szerint, ha használom az „alszik” szolgáltatást?
- **3. Kérdés:** Melyik sor helyes, ha sehol sem használom a az „alszik” szolgáltatást?

GOF 2

- **GOF2:** Favour object composition over inheritance.
- Használjunk objektum összetételt öröklődés helyett, ahol csak lehet.

```
// Öröklődéssel
class Geringes {
          public String fut() { return "fut"; }
}
class Kutya extends Geringes {
     public String gyorsanFut() {
          return "gyorsan" + fut();
     }
}
main () {
          Kutya k1 = new Kutya();
          String s = k1.gyorsanFut();
}
// s = "gyorsanfut 1"
```

- Egyszerűen használjuk a 'fut' metódust, mert **megörökölte**

```
// Objektum összetétellel
class Geringes {
          public String fut() { return "fut"; }
}
class Kutya {
     Geringes g = new Geringes();
     public String gyorsanFut() {
          return "gyorsan" + g.fut();
     }
}
main () {
          Kutya k1 = new Kutya();
          String s = k1.gyorsanFut();
}
```

- Referencián keresztül használjuk a 'fut' metódust, mert **meghívtuk**

GOF 2

- **GOF2:** Favour object composition over inheritance.
- Használjunk objektum összetételt öröklődés helyett, ahol csak lehet.

```
// Objektum összetétellel
class Gerinces {
          public String fut() { return "fut"; }
}
class Kutya {
     Gerinces g = new Gerinces();
     public String gyorsanFut() {
          return "gyorsan" + g.fut();
     }
}
main () {
          Kutya k1 = new Kutya();
          String s = k1.gyorsanFut();
}
```

- Referencián keresztül használjuk a ‘fut’ metódust, mert **meghívtuk**
- **HA** van egy referenciám egy másik objektumra, akkor annak szolgáltatásait használhatom.
- Ezt nevezik **objektum összetételnek** (object composition)
- Az objektum összetétel másik neve: **HAS-A** kapcsolat

HAS-A KAPCSOLAT

```
// Objektum összetétel – HAS-A

// A Gitárosnak van egy Gitárja
class Gitar { ... }
class Gitaros { Gitar g; ... }

// A Kutyának van egy Hazdája, aki Ember
class Ember { ... }
class Kutya { Ember gazdi; ... }

// A Kutyának van egy Farka
class ? ...
```

- **A HAS-A kapcsolatnak a birtoklás erőssége szerint 2 fajtája van:**
  - Aggregáció (aggregation) – Megosztott tulajdonlás
  - Kompozíció (composition) – Kizárólagos tulajdonlás
- **Klasszikus példa:**
  Ha meghal a gitáros, akkor vele temetik a gitárját?
  - Ha igen, akkor kompozíció
  - Ha nem, akkor aggregáció

HAS-A KAPCSOLAT

- **GOF2:** Favour object composition over inheritance.
- Használjunk objektum összetételt öröklődés helyett, ahol csak lehet.
- **Avagy:** Használj HAS-A kapcsolatot IS-A kapcsolat helyett, ha lehet!

Miért?

Aggregáció

- Másnak is lehet referenciája a gitárra
- Lehet a banda tulajdonát képezi

Kompozíció

- Csak a kutyának lehet referenciája a farkára
- Csak ő csóválhatja

HAS-A KAPCSOLAT

Az IS-A kapcsolat:

- Mert az IS-A kapcsolat túlságosan merev.
- IS-A kapcsolat már a program írásában létrejön a forráskódban, még a fordítás előtt.
- **Emiatt dinamikusan nem változtatható.**
- Ún.: „Fehér-dobozos” újrahasznosítás, mert általában ismerem az ős forráskódját.
- **Kérdés:** Ez milyen „veszélyt” rejt?

A HAS-A kapcsolat:

- Ún.: „Fekete-dobozos” újrahasznosítás, mert általában nem ismerem az általam birtokolt referencia mögött álló osztály forráskódját.
- Dinamikusan változtatható.
- **Itt Direkt programozhatunk felületre.**

ÖSSZEFOGLALÁS

Alapok	Fogalmak	OOP Alapelvek
GOF	IS-A	HAS-A

ÖSSZEFOGLALÁS

Fogalmak:

Felület	**Megvalósítás**	**Absztrakció**
Osztály	**Interface**	**Szolgáltatás**

OOP Alapelvek:

Egységbezárás	**Többalakúság**	**Öröklődés**

Kapcsolatok:

GOF	**IS-A**	**HAS-A**

MILTON FRIEDMAN EGYETEM

KÖSZÖNÖM A FIGYELMET!

1. óra - Bevezetés

2024.03.18

Szalai Patrik

szalai.patrik@uni-milton.hu