Pázmány Péter Katolikus Egyetem
Információs Technológiai és Bionikai Kar

Interface, Lambda, Reflection

Java programozás

4. gyakorlat

Interface

Interface fogalma általánosan

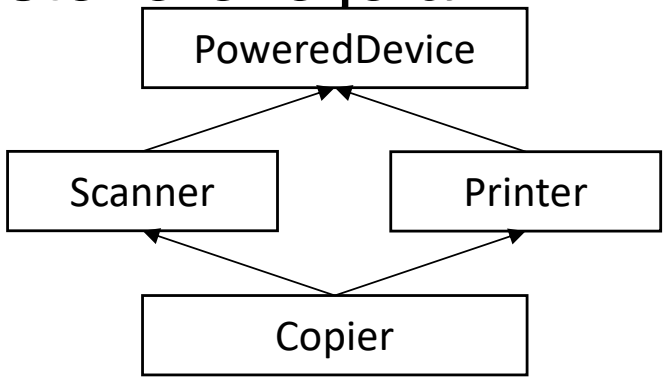
- Interface = felület
  - olyan felületet definiál, amelyen keresztül érintkezhetünk az osztállyal
  - a megvalósítást nem szabja meg!
  - konstansok, tagfüggvények meghatározása

A Java két alapvető építőeleme

- 1. alapvető építőelem: **`class`** (osztály)
  - Már tanult osztályok
  - Közös ősük a `java.lang.Object`
- 2. alapvető építőelem: **`interface`** (interfész)
  - Teljesen absztrakt osztályként viselkedik
  - Nincs közös ősük!
  - „Használati utasítás” az osztályhoz
  - Tartalmazhat:
    - Konstansokat
    - Tagfüggvények szignatúráit (a törzset nem!)
- 3. **`Enum`**

Többszörös öröklődés

- Ha mindkét ősosztályban megtalálható egy adott szignatúrájú függvény, akkor melyik őstől örökölje a leszármazott a megvalósítást?
- A C++ megoldása:
  - külön meg kell mondani, hogy melyiktől örökölje



  - `Scanner::f()` a `Scanner` ősosztályban levő `f()`
  - `Printer::f()` a `Printer` ősosztályban levő `f()`
  - `f()` a közös leszármazott `Copier` osztályban levő `f()`
  - Felmerülő nehézség
    - diamond problem (gyémánt probléma)
- A Java megoldása:
  - nincs többszörös öröklődés
  - viszont egy osztály több interfészt is megvalósíthat

Interfész a Javában - példa

- Például:
- ```
  public interface Edible {
        // konstans
        int MAX_DAY = 10;

        // metódus
        void eat();
  }
  ```
  ```

Interfész a Javában - megvalósítás

- Az interfészeket az osztályok valósíthatják meg.
- Kulcsszó:
  - `public class Apple implements Edible`
- Ha egy osztály megvalósít egy interfészt, akkor az interfész használható statikus típusként, mivel az interfész is egy típust definiál.
- **Mi az a statikus és mi az a dinamikus típus?**
- Például:
  - ```
    Edible apple = new Apple();
    Edible pear  = new Pear();
    ```
- feltéve, hogy:
  - ```
    Apple implements Edible
    Pear  implements Edible
    ```

Módosítók az interfészekben

- **Tagfüggvények**
  - Egy interfész tagfüggvénye mindig **`public abstract`**
    - Ezeket a módosítókat nem kötelező kiírni.
    - Mást kiírni hiba.
    - Miért?
- **Konstansok**
  - Minden konstans eleve **`public static final`**
  - Ezt sem kell kiírni

További lehetőségek

- Alapértelmezett metódus:
  - Lehetőségünk van nem csak a metódus deklarációját megadni, hanem az implementációját is
  - Erre a metódus fejében írt **`default`** kulcsszóval van lehetőségünk,
    majd megadva annak implementációját.
    - Természetesen ettől az implementáló osztály ezt ugyanúgy felüldefiniálhatja.
- Statikus metódus:
  - Lehetőségünk van továbbá statikus metódusok létrehozására is
  - Ezt az eddig tanultaknak megfelelően, a **`static`** kulcsszó használatával van lehetőségünk az implementáció megadásával együtt.

Interfész a Javában - öröklődés

- Interfészek között is lehet örököltetni és az osztályoktól eltérően
  az interfészek között van **többszörös öröklődés**.
- A leszármazott örököl minden konstanst és tagfüggvényt az ősinterfész(ek)től.
  - Konstansok:
    - A különböző nevű konstansok esetében nincs gond, mind használható.
    - Azonos nevű konstansok esetében nem tudja eldönteni melyiket használja, ekkor a leszármazotton keresztül nem tudunk hivatkozni ezekre.
      - Esetleg felüldefiniálva hivatkozhatunk az egyik ősére
  - Metódusok:
    - Aminek nem volt implementációja azzal nincs gond.
    - Aminek bármelyik ősben is volt implementációja (**`default`**) annál nekünk kell feloldanunk az ütközést. Erre több lehetőség van.
      - Újra absztaktnak jelölhetjük (`@Override …(); )`
      - Meghívhatjuk valamely ősosztálybeli implementációt (`Parent.super.defaultMethod(); )`

Interfész a Javában - észrevétel

- Két fontos észrevétel:
  - Egy osztály akármennyi interfészt megvalósíthat.
  - Egy interfészt akármennyi osztály megvalósíthat.
- Nézzük meg az eddig tanultakat az `interfaces` csomagban.

Beépített interfészek

- Az egységesebb kódok megírásában segítenek. Tipikusan egy-egy osztály „képességeit”, funkcióit jelezzük interfészeken keresztül.
- Például:
  - `Comparable` : összehasonlításhoz
  - `Cloneable` : marker interfész, engedélyezi az `Object clone()` metódusát
  - `Serializable` : marker interfész, engedélyezi egy objektum „sorosítását”, azaz olyan szöveggé alakítását, amiből később visszaállítható
  - `Runnable` : szálként futtatható

  - Később lesz szó róluk

Mire jó az interfész?

- **Használhatjuk** valamilyen funkcionalitás biztosítására.
  - Az implementáló osztálynak kötelező megvalósítania az interfész tagfüggvényeit.
  - Az osztálynak biztosan tudnia kell azt a funkciót.
- **Használhatjuk** több, hasonló osztály kezelőfelületének egységesítésére.
  - Az implementáló osztályokat ugyanúgy tudjuk kezelni függetlenül attól, hogy az adott problémára milyen megoldást nyújtanak.
  - Az interfész statikus típusként is használható, így az implementáló osztályokat „egy kalap alá vehetjük”.

Mire NEM jó az interfész?

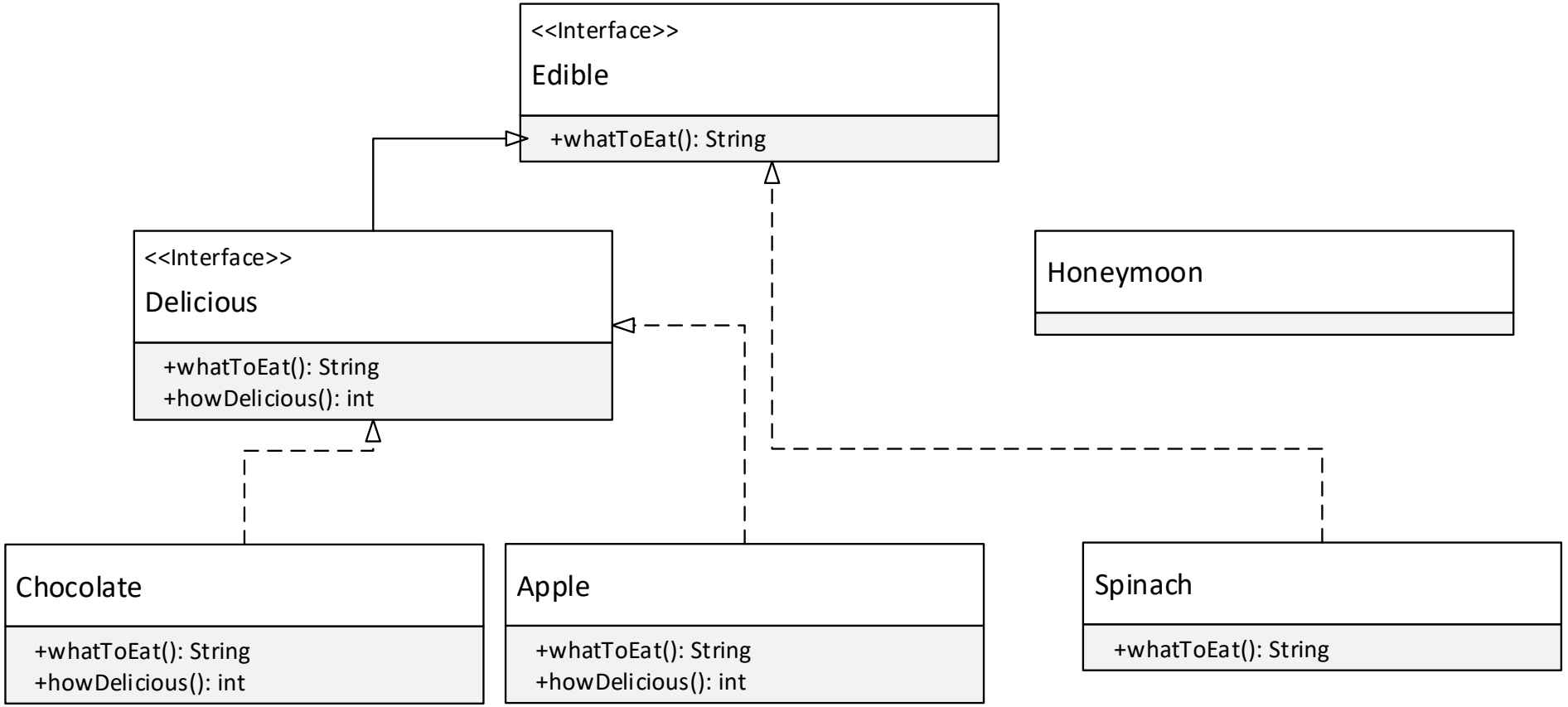
- **NE használjuk** belső szerkezet meghatározásához.
  - Az interfészben csak publikus konstansokat és tagfüggvényeket tudunk deklarálni.
  - Feltörné az enkapszulációt!
  - Az interfész nem utalhat a belső működésre
    - Bár a default interfészek használatával a lehetőségek száma növekedik, érdemes megfontolni, hogy valóban szükség van-e rájuk!
  - Minél általánosabbnak kell lennie.
  - A bemenetre minél általánosabb megkötéseket kell megadni.
  - Ehhez még következő gyakorlaton (Collection) részletesebben kitérünk.

Figyelem!

- Ha egy osztály megvalósít egy interfészt, akkor az osztálynak:
  - az interfész minden metódusát meg kell valósítania vagy
  - absztraktnak kell jelölnie!
- Ha az osztályunk leszármazott és **`interface`**-t is megvalósít, akkor mindig először szerepel az **`extends`** és csak utána az **`implements`** kulcsszó!

Példa

- Kódoljuk le az alábbi struktúrát!



Interfész implementálása

- Hozzuk létre az **`Apple`** osztályt.
- Ez egy finom gyümölcs, tehát megvalósítja a **`Delicious`** interfészt.
- A szokásos módon hozzuk létre, de még ne nyomjuk meg a **Finish** gombot!
- Keressük meg az ablakban az **Interfaces**... feliratot, tőle jobbra pedig az **Add**... gombot! Ezt használjuk!

Példa – Interfész implementálása

```
public interface Edible {
    String whatToEat();
}
```

```
public interface Delicious extends Edible {
    int howDelicious();
}
```

```
public class Chocolate implements Delicious {
    @Override public String whatToEat() {
        return "Chocolate";
    }

    @Override public int howDelicious() {
        return 2;
    }
}
```

Interfészek bővítése

- Mi történne, ha egy interfészt kibővítenénk egy új tagfüggvénnyel?
  - Az interfészt megvalósító osztályok „lerokkannak”
    - Hiszen lesz egy tagfüggvény, amit nem valósítanak meg.
- Megoldás:
  - ha nem kell mindenkinek az új tagfüggvény, akkor hozzunk létre egy új interfészt, ami:
    - Az eredeti interfészből örököl
    - Az új tagfüggvényt tartalmazza
  - ha kell mindenkinek az új tagfüggvény, akkor az alapértelmezett metódust használhatjuk, ezzel megadva az implementációt is
- Természetesen, előfordulhat, hogy bővíteni kell egy interfészt.

Példa - Interfészek bővítése

- ```
  public interface Edible {
      String whatToEat();
  }
  ```

- Új interfész létrehozásával:

- ```
  public interface
      EdibleWithCalories
            extends
  Edible {

    Integer
  howManyCalories();

  }
  ```

- Létező interfész módosításával:

- ```
  public interface Edible {

    String whatToEat();

    default Integer
        howManyCalories() {
      return null;
    }

  }
  ```
  ```

Példa – Öröklődés

```
public interface Edible {
        String whatToEat();
        default Integer howManyCalories() {
                return null;
        }

}
public interface Delicious extends Edible {
        int howDelicious();
}
public class Chocolate implements Delicious {
        @Override public String whatToEat() {
                return "Chocolate";
        }

        @Override public int howDelicious() {
                return 2;
        }
        @Override public Integer howManyCalories() {
                return 100;
        }
}
```

Belső osztályok, Lambda kifejezések

Nested class, Inner class

- Egy meglévő osztályon belül deklarált újabb osztály
- Eléri az outer class (az az osztály amiben implementáljuk) minden változóját és metódusát
- Lehet statikus és nem-statikus
- Statikus (nested):
  - Csak szervezésbeli jelentősége van
    - Mint bármely más osztály/objektum, külső osztálynak csak a statikus változóit és metódusait éri el
  - Példányosítható az osztály példányosítása nélkül
- Nem-statikus (inner):
  - Eléri a példányváltozókat és példánymetódusokat is
  - Csak a külső osztálybeli példány keretében értelmezhető
    - Csak az osztály példányosítása után példányosítható
- Láthatósága az eddigiektől eltérően lehet private és protected is

Példa – Inner & nested class

```
class Outer {
        private int a = 5;
        private static int s = 2;

        private class Inner {
                @Override
                public String toString() {
                        return "Inner: " + a;
                }
        }

        public static class StaticNested {
                @Override
                public String toString() {
                        return "Nested: " + s;
                }
        }
}
```

Anonymous inner class

- Az anonymous inner class a sima inner class-hoz nagyon hasonlító, egy másik osztályon belül definiált osztály, melyet azonban nem illetünk külön névvel
- Olvashatóbbá teszik a kódot, mivel a definíció oda kerül, ahol felhasználjuk (ezért leginkább rövid eseménykezelőkhöz javasolt): definíció és példányosítás egy helyen
- Alaptípusa vagy osztály (amiből öröklődik) vagy interfész (amit implementál)

- ```
  Edible a = new Edible() {
        @Override
        public String whatToEat() {
              return "AnonymousFood";
        }
  };
  ```
  ```

Példa

- Nézzük meg az innerclass és innerclass.demo csomagban lévő példát

Lambda kifejezések

- Java-ban lehetőség van lambda kifejezések használatára.
- A kifejezéssel tovább egyszerűsíthető egyszerű metódusokat tartalmazó névtelen osztályok leírása.
- Eseményvezérelt rendszerekben nagyon hasznos (pl. mobil programozás, GUI programozás): egy esemény bekövetkeztekor valami rövid, jól definiált dolgot akarunk csinálni.
- Nézzünk az alábbi metódust tartalmazó interfész megvalósítására példát:

```
interface IExample {
  int getResult(int x);
}
```

Lambda kifejezések

- Anonymous inner class-ként:
  - ```
    new IExample() {
      int getResult(int x) {
            return x * x;
      }
    }
    ```

- Lambda kifejezés(ek) használatával, a következők ekvivalensek:
  - `(x) -> x * x`
  - `(int x) -> x * x`
  - `(x) -> { return x * x; }`
  - `(int x) -> { return x * x; }`
    ```

Példa

- Nézzük meg a lambda és lambda.demo csomagban lévő példát

Kivételkezelés II.

Ismétlés

- A kivétel a programnak az elvárttól eltérő állapotát leíró objektum
- Ha a futás során valahol abnormális állapotot észlelünk, akkor ott egy kivételt “dobunk”, amivel jelezzük a problémát a külvilág (jó esetben a program többi része) felé
- A kivételbe elhelyezünk minden információt, ami a hiba kezeléséhez szükséges (pl. melyik beolvasott fájl hibás)

Ismétlés

- Egy catch-blokk egy adott típusú, vagy abból származtatott kivételt tud elkapni, ezért fontos a blokkok sorrendje
- Ha a try-blokk után nincs megfelelő típust váró catch, úgy a kivétel egy szinttel feljebb kerül
- Ha nincs felsőbb szint (azaz hívó metódus), akkor a program futása megszakad.
- A nem kezelt kivételek nem jutnak el az operációs rendszerig, a JVM kapja el őket.
- Ha egy metódus (tovább)dob egy ellenőrzött kivételt, akkor azt kötelező kiírni a metódus szignatúrájában.
- `public int pop() throws UnderFlowException`

Kivételek öröklődéskor

- A következő szignatúrájú absztrakt tagfüggvényt örököljük az ősosztályunktól / vesszük át egy interfésztől:

```
public abstract int readIntFromFile(String file)
        throws IOException, NumberFormatException;
```

- Amikor megvalósítjuk, a dobható kivételek körét nem bővíthetjük!
  - Miért nem?
  - Szűkíthetjük?

Kivételek öröklődéskor

- A következő szignatúrájú absztrakt tagfüggvényt örököljük az ősosztályunktól / vesszük át egy interfésztől:

```
public abstract int readIntFromFile(String file)
        throws IOException, NumberFormatException;
```

- Amikor megvalósítjuk, a dobható kivételek körét nem bővíthetjük!
  - Miért nem?
    - Mivel az ősosztályt, mint statikus típust használva, nem tudnánk eldönteni, hogy az osztály pontosan milyen kivételt dobhat, így nem tudunk felkészülni a lekezelésére.
  - Szűkíthetjük?
    - Igen. Ebben az esetben nem kell félnünk, hogy esetleg nem lesz lekezelve a kivételünk, mivel az ősosztály, mint statikus típus, még több kivételre is fel van készítve.

Class, Reflection

A Reflection API

- A Reflection API segítségével futásidőben tudjuk vizsgálni és módosítani az alkalmazásunk szerkezetét.
- Például:
  - Milyen tagfüggvényei vannak egy adott osztálynak?
    - Ezeknek mik a módosítói, neve, paraméterei, visszatérési értéke?
  - Milyen tagváltozói vannak?
    - Név, módosítók, típus?
  - Osztályhierarchia bejárása
  - stb.

Előnyök és hátrányok

- Előnyei:
  - **Rugalmasság**: akár igény szerint is importálhatunk csomagokat és osztályokat
  - **Teljes elérhetőség**: az API segítségével az összes osztály és objektum minden tulajdonsága elérhető
- Hátrányai:
  - **Enkapszuláció feltörése**: privát változókhoz is hozzá tudunk férni! Nem szabad visszaélni vele!
  - **Kisebb hatékonyság**
  - **Biztonsági korlátok**: pl. egy Java appletnek nincs elég jogosultsága az API használatára

A Reflection API alapja

- A különböző szerkezeteket osztályokkal reprezentáljuk.
- Például:
  - **`Package`** osztály: egy csomagot reprezentál
  - **`Class`** osztály: egy osztályt reprezentál
  - **`Field`** osztály: egy tagváltozót reprezentál
  - **`Method`** osztály: egy tagfüggvényt reprezentál
  - **`Constructor`** osztály: egy konstruktort reprezentál
  - Stb.

Egy osztály „befogása” (retrieve)

- Egy osztályt többféleképpen is „be tudunk fogni”, azaz megszerezni az őt reprezentáló `Class` objektumot:

- **1) Meglévő objektum alapján**
  - `String str = "ez egy String";`
    `Class c = str.getClass();`
  - A `String` osztályt fogtuk be, tehát `c` a `String` osztályt fogja reprezentálni.
  - `c`-nek semmilyen kapcsolata sem lesz `str`-rel!
  - Primitív típushoz nem használható.
    - Nem osztályok
    - nincsenek tagfüggvényeik
    - nincs `getClass()`!

Egy osztály „befogása” (retrieve)

- Egy osztályt többféleképpen is „be tudunk fogni”, azaz megszerezni az őt reprezentáló `Class` objektumot:

- **2) Statikus befogás**
  - `Class c = boolean.class;`
  - `Class d = java.io.PrintStream.class;`
  - `Class e = int[][][].class;`
  - Primitív típusok legegyszerűbb befogási módja.
  - Statikusan működik, nincs szükség példányra.

Egy osztály „befogása” (retrieve)

- Egy osztályt többféleképpen is „be tudunk fogni”, azaz megszerezni az őt reprezentáló `Class` objektumot:

- **3) Név alapján**
  - `Class c = Class.forName("java.lang.String");`
  - Primitív típushoz nem használható.

- Nézzük meg a `reflection` csomagot!

Gyakorló feladat G04F01

- A házi feladatban saját `Logger` osztályokat kell létrehoznotok egy megadott interfész nyomán.
- Az interfész tartalmazzon egy `log` nevű eljárást amely vár egy szöveges bemenetet.
- Továbbá implementáljátok egy `LoggerFactory` nevű osztályt a következő osztálymetódusokkal:
  - `Logger consoleLogger();`
  - `Logger fileLogger(String fileName);`
  - `Logger multiLogger(Logger[] logger);`
- Ezek által visszaadott `Logger` példányok rendre:
  - A konzolra írjon ki.
  - Adott nevű fájlba írjon ki.
  - Az átadott `Logger`ekkel írjon ki.

Gyakorló feladat G04F01

- Készítsetek továbbá egy `ThrowableLogger` interfészt
  - mely származik a `Logger` interfészből és
  - túlterheli a `Logger` interface eljárását egy plusz `Throwable cause` argumentumú eljárással.
- Készítsetek egy saját kivétel osztályt, melyet a program futása alatt valahol kiváltotok és lekezelésekor meghívjátok vele a `ThrowableLogger` példányotok megfelelő metódusát.
- Hozzatok létre egy tömböt és töltsetek bele különböző `Logger` példányokat.
- Logoljatok a létrehozott `Logger` példányokkal.

Gyakorló feladat G04F02

- Készíts egy valutaváltó alkalmazást!
  - A program USD, GBP, EUR, CHF és JPY pénznemekből tudjon HUF-ba és vissza váltani!
  - A használandó árfolyamok:
    - 1 USD 244,76 HUF
    - 1 GBP 397,34 HUF
    - 1 EUR 309,71 HUF
    - 1 CHF 256,52 HUF
    - 1000 JPY 2252,66 HUF
  - Hozd létre a **`Currency`** absztrakt ősosztályt.
    - Legyen egy **`final double`** mezője, ami az aktuális árfolyamot tárolja.
    - Egy másik **`double`** mezője a pénznemből nálunk levő készletet tárolja, pl. 150 USD. Alapértékét konstruktorban tudjuk MAJD megadni.
  - Hozz létre két **interfészt**.
    - **`BuyableCurrency`**: egy függvénye legyen, a **`buyCurrency()`**. Ezzel tudunk HUF-ot a pénznembe váltani.
    - **`SellableCurrency`**: egy függvénye legyen, a **`sellCurrency()`**. Ezzel tudjuk a pénznemet HUF-ba váltani.
    - Mindkét függvény double-t várjon és double-t adjon vissza!

Gyakorló feladat G04F02

- Minden külföldi pénznemet **külön** osztályként hozz létre.
  - Az osztályok örököljenek a **`Currency`** osztályból.
  - Mindegyik pénznemmel mindkét irányban kereskedünk,
    - kivéve az angol fonttal (GBP), amit nem akarunk eladni,
    - és a japán jennel (JPY), amiből nem akarunk venni.
  - Semmiképpen sem akarunk 1000 USD-nél többet eladni egy alkalommal!

- Néha szeretnénk letiltani a svájci frank (CHF) mozgását.
  - A CHF osztályának legyen egy **`boolean enabled`** adattagja, ami alapból **`true`**, azaz engedélyezi a kereskedést.
  - Legyen egy **`getter`** függvénye, ami lekérdezi az **`enabled`** értékét.
  - Legyen egy paraméter nélküli **`trigger`** függvénye, ami kapcsolóként működik, azaz az ellenkező állapotba állítja az **`enabled`** adattagot.
- Írj egy értelmes tesztelő Main osztályt is! Gondold át, hogy milyen hibalehetőségek jelentkezhetnek, és a hibákat saját kivételekkel jelezd!

Gyakorló feladat G04F03

- Az alábbi struktúrát kell megvalósítani
- Adott egy ősosztály `Sokszogek`:
  - Absztrakt osztály, nincs benne semmi. Csak leszármazottai lesznek.
- Adott két interfész: `KeruletSzamithato`:
  - Egy függvénye van: `float computeKerulet();`
- És egy másik: `KeruletTeruletSzamithato`:
  - A kerületen kívül van: `float computeTerulet();`
- Maga a feladat: Hozz létre egy háromszög, és egy négyszög osztályt. Mindkét osztály a sokszög oldalait tartalmazza, ennyi alapján valósítsa meg mindegyik a fentiek közül az ennyi adat alapján értelmezhető interfészt.
- Ha hibás adatot kap egy sokszög, vagyis ha olyan oldalhosszakat kap amikből nem lehet az adott sokszöget megrajzolni (pl.: háromszög – háromszög-egyenlőtlenség), azt kezelje kivételekkel. Legyen egy olyan kivétel ami minden sokszögre használható, és ebből származóan legyen az egyes sokszögtípusokra is külön-külön.

Gyakorló feladat G0F03

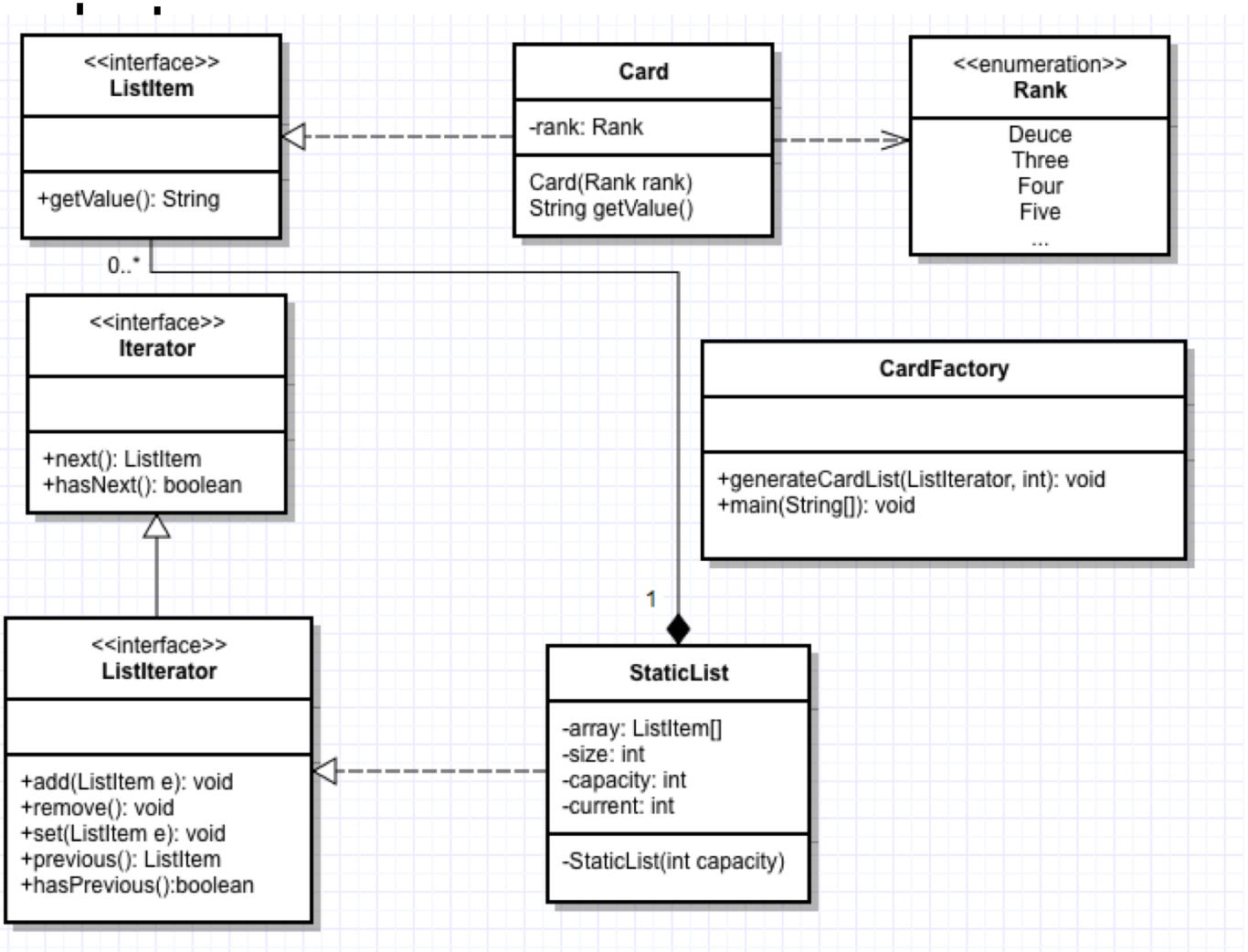
- Ezek után hozz létre egy `Main` osztályt, ahol a `main` függvényben létrehozol egy sokszögekből álló tömböt. Tegyél bele mindenféle sokszöget amit megvalósítottál (lehet többet is nem kell megállni kettőnél). Figyelj a kivételkezelésre.
- Hozz létre egy kör osztályt! Ez is valósítsa meg a megfelelő interfészt!
  Ezt is teszteld le!

Gyakorló feladat G04F04

- Készíts egy interfészt, neve legyen Operation, amely operate metódusa két egész számot vár, és visszatér szintén egy egésszel.
- Származz le ebből egy DescribableOperation interfész-el, ez bővítse ki az előzőt egy új describe metódussal, melynek nincs paramétere, és String-el tér vissza
  - Ez legyen default metódus, és az „Operation” szöveget adja vissza
- Készíts egy Operations osztályt, melynek legyen több static ill. nem static belső osztálya
  - Ezek mind a DescribableOperation-t valósítsák meg, és valami értelmes kétváltozós matematikai műveletet hajtsanak végre
  - A nevük is ezt reprezentálja
- Készíts egy Main osztályt, amiben egy tömbben tárold az összes deklarált műveleted egy-egy példányát.
  - A tömbben rakj hozz egy példányt lambdaként és névtelen osztályként is
- A felhasználótól kérj be két számot, majd azokra alkalmazd az összes műveletet a tömb segítségével, majd írd ki a művelet nevét illetve értékét!

Gyakorló feladat G04F05

- Implementáld az ábrán lévő interfészeket és osztál



Gyakorló feladat G04F05

- A ListItem interfészben egy getValue() metódus van, String visszatérési értékkel. Ezt az interfészt valósítja meg a Card osztály, melynek egy rank mezője van, melynek típusa egy Rank enum, ami a franciakártyák értékeit listázza (kettes, hármas, ..., király, ász). A Card-nak legyen egy konstruktora, amellyel beállítjuk az értékét.

- Az Iterator interfész olyan osztályt ír le, amin végig lehet egy irányba iterálni, next() és hasNext() metódusokkal rendelkezik. Ebből örököl a ListIterator interfész, ami add() (lista végére beszúrás), remove() (aktuális elem törlése), set() (aktuális elem lecserélése), previous() és hasPrevious() metódusokkal rendelkezik. Ezt az interfészt valósítja meg a StaticList osztály, ami rendelkezik egy ListItem tömbbel, maximális kapacitással, éppen aktuális mérettel és aktuális pozícióval, valamint a konstruktorán át meg lehet adni a maximális kapacitást.

- Hozz létre egy CardFactory osztályt, egy generateCardList osztálymetódussal, ami feltölti az átadott ListIterator-t megvalósító objektumot adott számú kártyával. Az osztály valósítsa meg a main()-t is, létrehozva egy StaticList-et valamilyen kapacitással, majd meghívva a generateCardList-et, végül az Iterator interfész képességeit kihasználva sorban kiírja a listában lévő kártyák értékét, a ListItem interfész segítségével.