Készítette: Jánki Zoltán Richárd
Dátum: 2016.09.26.

Apple Swift kurzus
4. gyakorlat

Protokoll:

- metódusok, adattagok, funkcionalitás tervezete
- Java-ban az interfészek megfelelője
- készíthető osztályhoz, struktúrához, enum-hoz, stb...
- megszorítások a típusra vonatkozóan (milyen metódusai, adattagjai, stb... legyen?)

szintaxis:

```
protocol MyProtocol {}
```

- a protokoll használatakor a típust követően kell rá hivatkozni a nevével
  pl.: `class MyClass : MyProtocol {}`

- a protokollban definiálásra kerülnek a típus adattagjai
- az adattagok tulajdonsága (számított, tárolt adat) nem kerül specifikálásra
- az adattagok *gettable* és *settable* jellemzőjét a változók neve után adjuk meg
  - *gettable* és *settable*: nem lehet konstans vagy csak olvasható adattag
  - csak *gettable*: bármilyen adattag lehet, szükség esetén settable is lehet
  - csak *settable*: nem lehetséges!

  - pl.:
```
protocol MyProtocol {
    var settableProperty: Type { get set }
    var noNeedToBeSet: Type { get }
}
```

- protokollban definiált metódusok fejlécét szükséges csak megírni (név, paraméterezés, visszatérési érték)

  pl.:
```
protocol MyProtocol {
    func funcName() -> Type
}
```

Delegate:

- egy tervezési minta
- korábbi nyelvekben függvénypointerként mutatták be
- egy objektum, mely koordinál, ha egy másik objektum valamilyen eseménybe ütközik
  pl.: ablak bezárásakor felkínálják a bezárás előtti adatmentést (adatvesztés megelőzése végett)

- egy protokoll kerül definiálásra, mely tartalmazza a delegate "kötelezettségét"
- a delegate implementációja osztályban történik meg
- a vizsgálandó objektumnak rendelkeznie kell egy **`delegate`** adattaggal

pl.:
```
protocol MyDelegate {
    func somethingStarted(param: ParamType)
}

class ImplementedDelegate : MyDelegate {
    func somethingStarted(param: ParamType) {
        //delegate code
    }
}
class MyClass {
    var delegate : MyDelegate?
}
```

Kiterjesztések:

- meglévő típusokhoz (osztály, struktúra, enum) újabb funkcionalitást rendelünk
- olyan típusok kiterjesztése, amelyek forráskódja nem érhető el
- új metódus, adattag, inicializáló függvény hozzáadása
- osztályon kívül definiálandó (jellemzően a programkód végén)

```
extension Type {
    var newVar: Type { return self }
}
```

Hibakezelés:

- azon folyamatok, amelyek nem feltétlen futnak le sikeresen, hibát dobhatnak
  pl.: fájl beolvasása lemezről
- saját hibatípusok is definiálhatóak (enum)
- a saját hibatípusok az **`Error`** ősosztályból öröklődnek
- a potenciális hibát dobó részeket **`do-(try)-catch`** blokkal kezeljük
- azon függvények, amelyek hibát dobhatnak, ellátandók **`throws`** kulcsszóval

```
enum MyError : Error {
    case enum1
}

func dangerousFunc(param: paramType) throws -> returnType {
    guard (statement) else {
        throw MyError.enum1
    }
    return something
}

do {
    try dangerousFunc(param: myParam)
} catch MyError.enum1 {
    print("I catched an error.")
}
```

Generikus típusok:

- cél: a kód általánosítása, újrafelhasználhatósága
- olyan kód írása, mely nem csak egy adott típusra működik, hanem akár mindre
- az **`Array`** és **`Dictionary`** is generikus kollekciók

pl.:
```
func swap(a: inout Int, b: inout Int) {   //jól működik Int-re
    let temp = a
    a = b
    b = temp
}

func swapGen<T>(a: inout T, b: inout T) { //működik minden típusra
    let temp = a
    a = b
    b = temp
}
```