


Programozás I.


Dr. Ferenc Rudolf
Dr. Jász Judit

Szegedi Tudományegyetem
Informatikai Intézet
Szoftverfejlesztés Tanszék

2021-02-08



- 
- 1 **Bemutakozás**
 - **Kurzus információk**
 - 2 **Objektumorientált tervezés**
 - A modellezés alapelvei
 - A „siker háromszöge”
 - UML
 - Objektumorientáltság alapfogalmak
 - 3 **Java alapok**
 - Bevezető
 - Az első java program
 - 4 **Java**
 - Programozási alapismeretek ismétlése
 - Objektumok, osztályok
 - Polimorfizmus
 - Annotációk
 - Objektumok kezelése
 - Hozzáférés szabályozása
 - Beágyazott osztályok
 - Újrafelhasználhatóság
 - Interfészek és enumerációk
 - Objektumok tárolása
 - 5 **Tervezési minták**
 - Hibakezelés kivételekkel
 - Sztringek
 - Generikus típusok
 - Típus információk
 - I/O
 - GUI; Belső, lokális és anonim osztályok, lambdák

- 
- 1 **Bemutakozás**
 - **Kurzus információk**
 - 2 **Objektumorientált tervezés**
 - A modellezés alapelvei
 - A „siker háromszöge”
 - UML
 - Objektumorientáltság alapfogalmak
 - 3 **Java alapok**
 - Bevezető
 - Az első java program
 - 4 **Java**
 - Programozási alapismeretek ismétlése
 - Objektumok, osztályok
 - Polimorfizmus
 - Annotációk
 - Objektumok kezelése
 - Hozzáférés szabályozása
 - Beágyazott osztályok
 - Újrafelhasználhatóság
 - Interfészek és enumerációk
 - Objektumok tárolása
 - 5 **Tervezési minták**
 - Hibakezelés kivételekkel
 - Sztringek
 - Generikus típusok
 - Típus információk
 - I/O
 - GUI; Belső, lokális és anonim osztályok, lambdák

- Dr. Ferenc Rudolf

szoba: Szoftverfejlesztés Tanszék (Dugonics tér 13.)
152. szoba

e-mail: `ferenc@inf.u-szeged.hu`

honlap: `http://www.inf.u-szeged.hu/~ferenc/`



- Előadás

- Előre felvett videók, melyek legkésőbb az előadás időpontjára elérhetőek lesznek.
- Előadás fóliák és jegyzet.

- Gyakorlat:

- Előre felvett videók és jegyzet melyek legkésőbb a gyakorlat időpontjára elérhetőek lesznek.
- Óra időpontjában konzultációs lehetőség lesz az adott heti tananyaggal kapcsolatban.
- Óra időpontjában kerülnek megíratásra a röpdolgozatok és zárthelyi dolgozatok.

- Tananyagok elérhetősége: <https://okt.sed.hu/>

- Két zárthelyi dolgozat (ZH) és három röpdolgozat
 - 1. röpdolgozat: 4. hét gyakorlat (max. 10 pont).
 - 2. röpdolgozat: 7. hét gyakorlat (max. 10 pont).
 - 1. ZH: 8. hét gyakorlat (max. 35pont).
 - 3. röpdolgozat: 11. hét gyakorlat (max. 10 pont).
 - 2. ZH: 13. hét gyakorlat (max. 35pont).
- Házi feladatok a BIRÓ rendszerben (6-ból 5 kötelező)
- Kötelező program benyújtása és elbírálások
- Pótlás
 - Azok pótolhatnak, akik igazoltan nem tudtak jelen lenni.
 - ZH pótlása csak az utolsó gyakorlaton, a javítóval egy időben lehetséges.
 - Röpdolgozat pótlása a gyakorlatvezetővel egyeztetett időpontban lehetséges.
 - A házi feladatok és a kötelező program nem pótolhatók.

• Javító ZH

- Azok javíthatnak, akik nem érték el valamelyik részteljesítésből a minimumot (50%), és teljesítették a házi feladatokhoz, valamint a kötelező programhoz kapcsolódó követelményeket.
- Az egész félévi anyagból.
- Érvényteleníti az addig megírt ZH-kat és röpdolgozatokat.
- Elérhető maximális pontszám: 50
- Időpont: az utolsó héten a szorgalmi időszakban az **előadás időpontjában.**



- A gyakorlati jegy a pontszám alapján alakul ki:
 - Röpdolgozatok: max. 30 pont; min. 15 pont
 - ZH-k: max. 70 pont; min. 35 pont

89	–	100	jeles (5)
76	–	88	jó (4)
63	–	75	közepes (3)
50	–	62	elégséges (2)
0	–	49	elégtelen (1)

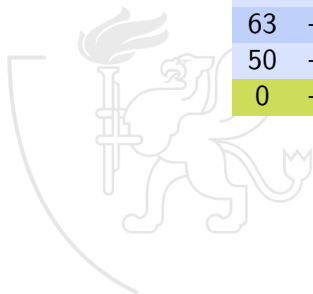
- **Jelen tájékoztatás a gyakorlati követelményekről nem teljes körű. A teljes körű tájékoztatás a Coospace gyakorlati színtereiben található.**

- A tárgy alapfeltétele a Programozás alapjai tantárgy. Aki azt nem teljesítette, **NE** vegye fel ezt a kurzust!
- Az előadásokhoz kiírt CooSpace tesztekből legalább 8-at teljesíteni kell.
 - Egy adott teszt akkor kerül elfogadásra, ha azt legalább 80%-osan kitöltötte a hallgató (többször kitölthető a teszt).
 - Minden teszt kitöltésének határideje a teszt kihirdetési hetének vasárnap 23 óra 59 perce.



- Az előadás teljesítéséhez sikeres vizsgát kell tenni.
- A sikeres vizsgához az elérhető pontszám legalább 50%-át el kell érni.
- A vizsgán semmilyen segédlet nem használható.

89	–	100	jeles (5)
76	–	88	jó (4)
63	–	75	közepes (3)
50	–	62	elégséges (2)
0	–	49	elégtelen (1)



- Objektum orientáltság
 - UML alapok (vizuális modellezés, jelölésrendszer, eszköz, modell, nézet, diagram)
 - Objektumok - állapota, viselkedése, identitása, élete
 - Osztály, csomag, osztálydiagram (asszociáció, aggregáció, öröklődés)
 - Objektum interfésze, implementáció elrejtése
 - Implementáció újrafelhasználása - kompozíció, aggregáció
 - Interfész újrafelhasználása - öröklődés, polimorfizmus
- Tervezési minták



• A Java nyelv

- Primitív típusok
- Osztályok - új típusok létrehozása, mezők, metódusok, csomagok
- Fordítás és futtatás, virtuális gép, futtató környezet
- Megjegyzések, dokumentáció, kódolási stílus
- Programfutas vezérlés, operátorok, precedencia, vezérlési szerkezetek, tömbök
- Inicializálás és takarítás, konstruktor, szemétygyűjtés
- Újrafelhasználhatóság - kompozíció, aggregáció, öröklődés, implementáció elrejtése
- Operáció kiterjesztés és felüldefiniálás, polimorfizmus, kései kötés
- Végso adatok, metódusok és osztályok
- Generikus osztályok
- Absztrakt és interfész osztályok, „többszörös öröklődés”, belso osztályok
- Hibakezelés kivételekkel és futás közbeni típusazonosítás (RTTI)
- Osztálykönyvtárak (kollekciók, iterátorok, I/O rendszer, GUI)

Segítség a kurzushoz

- Az előadás és gyakorlatok anyagai:
 - <https://okt.sed.hu>
- Könyvek - Objektumorientáltság
 - Vég Csaba: *Alkalmazásfejlesztés*. Logos2000 Kiadó, 1999.
 - Dr. Kondorosi Károly, Dr. László Zoltán, Dr. Szirmay-Kalos László *Objektumorientált szoftverfejlesztés*. Computer Books, 1999.
 - Ian Sommerville *Szoftverrendszerek fejlesztése*. Panem Kiadó, 2002.
 - Tarczali Tünde *UML diagramok gyakorlatban (.pdf)*. Typotex Kiadó, 2011.



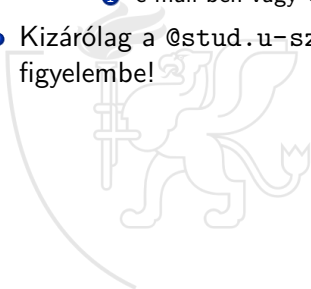
- Könyvek - JAVA

- **Bruce Eckel** *Thinking in Java*.
- Angster Erzsébet *Objektumorientált tervezés és programozás: JAVA (1.+2. kötet)*. 4KÖR Bt., 2003.
- Nyékyné Gaizler Judit (szerkesztő) *JAVA 2 (I.+II.+Referencia)*. ELTE TTK, 2001.
- Gál Tibor *JAVA programozás (egyetemi jegyzet)*. Műegyetemi Kiadó, 2002.
- Rogers Cadenhead *Tanuljuk meg a Java programozási nyelvet 24 óra alatt*. Kiskapu Kiadó, 2006.
- Benkő Tiborné, Tóth Bertalan *JAVA*. ComputerBooks, 2005.
- Dirk Louis, Peter Müller *JAVA 5*. Panem Kiadó, 2006.

- Web

- Oracle Java Tutorial
(<https://docs.oracle.com/javase/tutorial/java/index.html>)
- Csak az nem talál segédanyagot, aki nem keres.

- A gyakorlattal kapcsolatban
 - ① Gyakorlatvezető
 - ① Személyesen a gyakorlatokon
 - ② e-mail-ben vagy Coospace üzenetben
 - ② Fő gyakorlatvezető
 - ① e-mail-ben (antal@inf.u-szeged.hu) vagy Coospace üzenetben
 - ③ Ha a fő gyakorlatvezető úgy ítéli meg akkor: Előadó
 - ① e-mail-ben vagy Coospace üzenetben
- Kizárólag a @stud.u-szeged.hu címről jött leveleket vesszük figyelembe!




Programozás I.


Dr. Ferenc Rudolf
Dr. Jász Judit

Szegedi Tudományegyetem
Informatikai Intézet
Szoftverfejlesztés Tanszék

2021-02-08




- 
- 1 **Bemutakozás**
 - Kurzus információk
 - 2 **Objektumorientált tervezés**
 - A modellezés alapelvei
 - A „siker háromszöge”
 - UML
 - Objektumorientáltság alapfogalmak
 - 3 **Java alapok**
 - Bevezető
 - Az első java program
 - 4 **Java**
 - Programozási alapismeretek ismétlése
 - Objektumok, osztályok
 - Polimorfizmus
 - Annotációk
 - Objektumok kezelése
 - Hozzáférés szabályozása
 - Beágyazott osztályok
 - Újrafelhasználhatóság
 - Interfészek és enumerációk
 - Objektumok tárolása
 - 5 **Tervezési minták**
 - Hibakezelés kivételekkel
 - Sztringek
 - Generikus típusok
 - Típus információk
 - I/O
 - GUI; Belső, lokális és anonim osztályok, lambdák

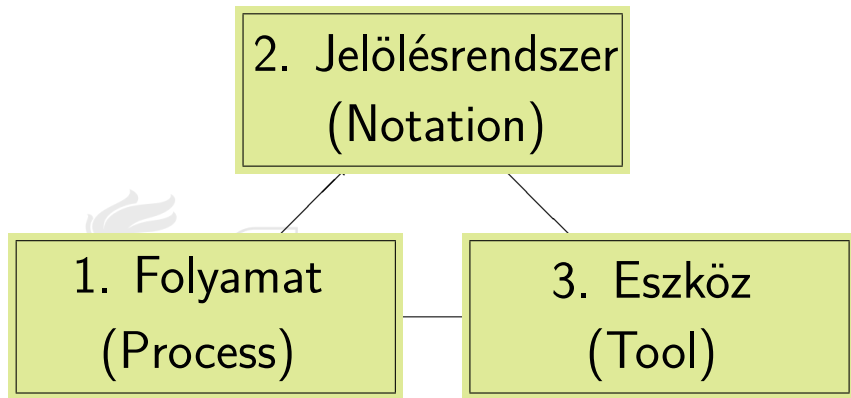
- 
- 1 **Bemutakozás**
 - Kurzus információk
 - 2 **Objektumorientált tervezés**
 - **A modellezés alapelvei**
 - A „siker háromszöge”
 - UML
 - Objektumorientáltság alapfogalmak
 - 3 **Java alapok**
 - Bevezető
 - Az első java program
 - 4 **Java**
 - Programozási alapismeretek ismétlése
 - Objektumok, osztályok
 - Polimorfizmus
 - Annotációk
 - Objektumok kezelése
 - Hozzáférés szabályozása
 - Beágyazott osztályok
 - Újrafelhasználhatóság
 - Interfészek és enumerációk
 - Objektumok tárolása
 - 5 **Tervezési minták**
 - Hibakezelés kivételekkel
 - Sztringek
 - Generikus típusok
 - Típus információk
 - I/O
 - GUI; Belső, lokális és anonim osztályok, lambdák

- Programozás lépései
 - modellezés és tervezés
 - kódolás
 - dokumentálás, tesztelés
- Modellezés
 - a probléma leírása a valós világból vett ötletekkel
 - a rendszer lényeges részeit vizsgáljuk
 - vizuális: szabványos grafikai eszközökkel
- Objektumorientált rendszerek:
 - analízis szintű gondolkodás
 - tervezés szintű gondolkodás

Miért modellezünk?

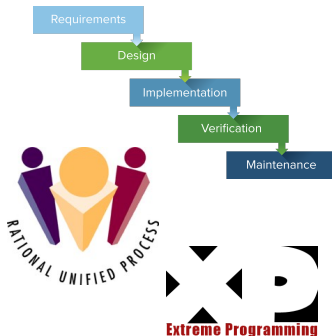
- „1 bitmap = 1 megaword” (ismeretlen szerző)
- Alkalmas üzleti folyamatok leírására
- Esettanulmányok a felhasználó szempontjából
- Kommunikációs eszköz
- Komplexitás kezelése
- Fejlesztési idő és rizikó csökkentése
- Szoftver architektúra definiálása
- Szoftver újrafelhasználhatóság

- 
- 1 **Bemutakozás**
 - Kurzus információk
 - 2 **Objektumorientált tervezés**
 - A modellezés alapelvei
 - A „siker háromszöge”
 - UML
 - Objektumorientáltság alapfogalmak
 - 3 **Java alapok**
 - Bevezető
 - Az első java program
 - 4 **Java**
 - Programozási alapismeretek ismétlése
 - Objektumok, osztályok
 - Polimorfizmus
 - Annotációk
 - Objektumok kezelése
 - Hozzáférés szabályozása
 - Beágyazott osztályok
 - Újrafelhasználhatóság
 - Interfészek és enumerációk
 - Objektumok tárolása
 - 5 **Tervezési minták**
 - Hibakezelés kivételekkel
 - Sztringek
 - Generikus típusok
 - Típus információk
 - I/O
 - GUI; Belső, lokális és anonim osztályok, lambdák



1. Folyamat

- Fejlesztési életciklust irányítja (folyamat leírása)
- Végrehajtandó lépések és a végrehajtás sorrendje
- Általában iteratív és inkrementális életciklust ír elő
- Minden iterációban
 - követelmények gyűjtése
 - analízis
 - tervezés
 - implementáció
 - tesztelés
- Példák
 - vízesés modell
 - Scrum
 - eXtreme Programming (XP)
 - Rational Unified Process (RUP)

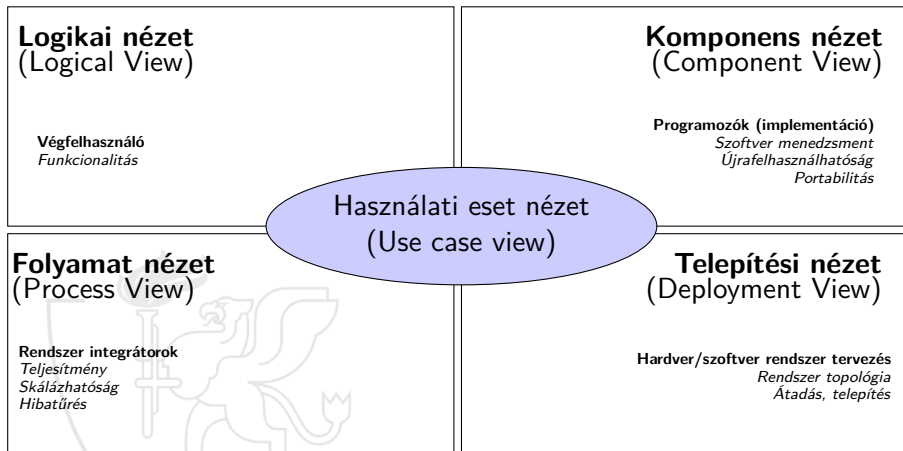


2. Jelölésrendszer

- Modellezéskor a rendszer architektúráját különböző nézetekkel írjuk le
 - logikai, komponens (implementációs), folyamat, telepítési (feladat-kiosztási)
- Használati eset nézet (esettanulmány)
 - az egészet összefogja és közös kommunikációs platformot létesít a résztvevő felek között
- Nem minden rendszer igényli az összes nézetet
- Új nézeteket is lehet definiálni
 - pl. adat, biztonság

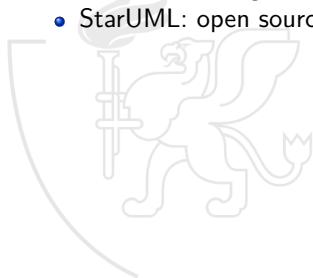



Az architektúra 4+1 nézete



3. Eszköz

- Ősi eszköz: papír és ceruza
- Nagyon sok UML eszköz van: egyszerű rajzoló programtól kifinomult objektummodellező eszközöig. Pl.
 - Rational Rose: teljes vizuális modellezés, kliens/szerver, osztott, valós idejű rendszerekhez
 - Borland Together ControlCenter
 - Microsoft Visio
 - EdrawMax: megosztható diagrammok
 - StarUML: open source megoldás 7 diagramtípussal



- 
- 1 **Bemutakozás**
 - Kurzus információk
 - 2 **Objektumorientált tervezés**
 - A modellezés alapelvei
 - A „siker háromszöge”
 - **UML**
 - Objektumorientáltság alapfogalmak
 - 3 **Java alapok**
 - Bevezető
 - Az első java program
 - 4 **Java**
 - Programozási alapismeretek ismétlése
 - Objektumok, osztályok
 - Polimorfizmus
 - Annotációk
 - Objektumok kezelése
 - Hozzáférés szabályozása
 - Beágyazott osztályok
 - Újrafelhasználhatóság
 - Interfészek és enumerációk
 - Objektumok tárolása
 - 5 **Tervezési minták**
 - Hibakezelés kivételekkel
 - Sztringek
 - Generikus típusok
 - Típus információk
 - I/O
 - GUI; Belső, lokális és anonim osztályok, lambdák

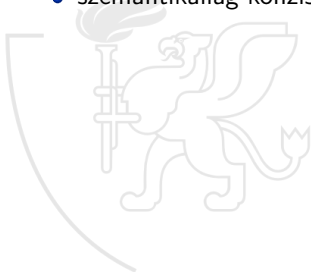
- Unified Modeling Language
(Egységesített Modellező Nyelv)
- Egy nyelv: szintaktikai és szemantikai szabályok összessége
- Szoftverrendszer elemeinek
 - vizualizálására
 - specifikálására
 - létrehozására
 - dokumentálására
- Teljes UML dokumentáció
 - <http://www.uml.org>

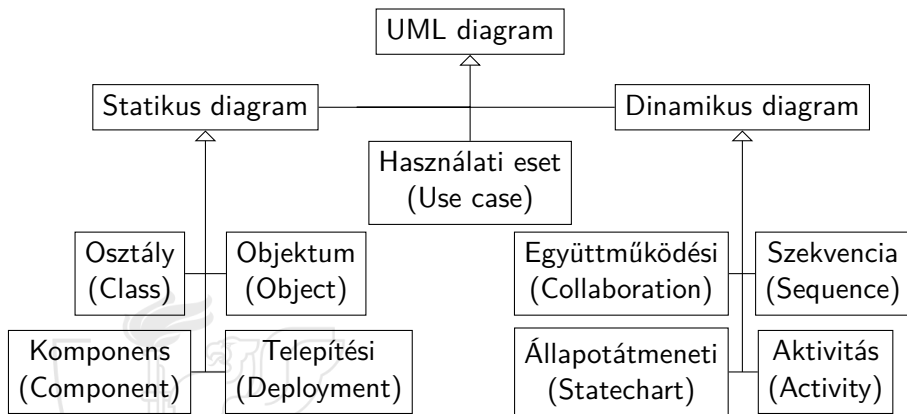



- Nyílt szabvány
(Object Management Group – OMG által)
- Teljes szoftverfejlesztési életciklust támogatja
- Különböző alkalmazási területekre alkalmazható
- Hatalmas tapasztalati tudásra épít
- Sok eszköz támogatja
(Rose, MSVC, Visio, G2Pro, Together, dia, Eclipse, NetBeans, Visual Paradigm...)
- Támogatók: Rational, HP, IBM, Microsoft, Oracle, Platinum, TI, Sun, DEC,...
- Bővebben: https://en.wikipedia.org/wiki/List_of_Unified_Modeling_Language_tools

Modell, nézet és diagram

- Modell: a rendszer teljes leírása
 - „adatbázis”
- Nézet: az architektúra „4+1” nézete
 - logikai, komponens...
- Diagram: a modell különböző vetületei
 - valamely résztvevő szemszögéből
 - részleges reprezentációja a rendszernek
 - szemantikailag konzisztens a többi nézettel





- 
- 1 **Bemutakozás**
 - Kurzus információk
 - 2 **Objektumorientált tervezés**
 - A modellezés alapelvei
 - A „siker háromszöge”
 - UML
 - **Objektumorientáltság alapfogalmak**
 - 3 **Java alapok**
 - Bevezető
 - Az első java program
 - 4 **Java**
 - Programozási alapismeretek ismétlése
 - Objektumok, osztályok
 - Polimorfizmus
 - Annotációk
 - Objektumok kezelése
 - Hozzáférés szabályozása
 - Beágyazott osztályok
 - Újrafelhasználhatóság
 - Interfészek és enumerációk
 - Objektumok tárolása
 - 5 **Tervezési minták**
 - Hibakezelés kivételekkel
 - Sztringek
 - Generikus típusok
 - Típus információk
 - I/O
 - GUI; Belső, lokális és anonim osztályok, lambdák

Objektum- és osztálydiagram

- Objektumok és osztályok vizuális reprezentációja
- Objektum orientált tervezéshez nélkülözhetetlen
- Jó kommunikációs eszköz



Az objektum fogalma

- Egy entitás ábrázolása (valós világból, vagy elvonatkoztatott)
Pl.:
 - személy adatai
 - poligon
 - kurzus
- Minden objektum rendelkezik:
 - állapottal
 - viselkedéssel
 - identitással



Objektum állapot

- Egy a lehetséges létezései közül
- Időben változó
- *Attribútumok* határozzák meg
- Példa:
 - egy Kurzus típusú objektum állapota nyitott, ha kevesebb, mint 20 jelentkező van, egyébként zárt



Objektum viselkedése

- Annak módja, hogyan reagál egy objektum más objektumok kéréseire
- Mindent definiál, amit az objektum „csinálhat”
- *Operációk* határozzák meg
 - egy Kurzus típusú objektumnak lehetnek hallgató_felvétel és hallgató_törlés operációi



Objektum identitása

- Minden objektum egyedi, akkor is, ha állapotuk azonos
- Példa:
 - a programozások és algoritmusok különböző objektumok, annak ellenére, hogy mindkettő valamilyen kurzus
 - ugyanabba, a Kurzus osztályba tartoznak

programozas : Kurzus

nyitott : bool = true

algoritmusok : Kurzus

nyitott : bool = false

Osztály fogalma

- Leírás objektumok csoportjához, amelyeknek közösek az
 - attribútumaik, operációik;
 - más objektumokkal való kapcsolataik és
 - szemantikus viselkedésük
- Egy minta objektumok létrehozásához (példányosításához):
 - minden objektum pontosan egy osztály példánya
 - az osztály az objektum típusa



Kurzus
nyitott : bool
hallgatoFelvetel() hallgatoTorles()

- Nagy rendszereknél elkerülhetetlen az osztályok csoportosítása
- Hierarchikus szerkezetet biztosít
- Magasabb szintű absztrakciót valósít meg
- Minden csomag tartalmaz(hat)
 - interfész osztályokat (publikus) és
 - implementációs osztályokat (belső, privát)



Kurzusok

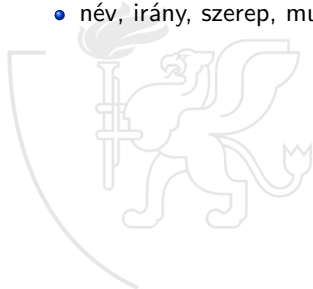
Kurzus

nyitott : bool

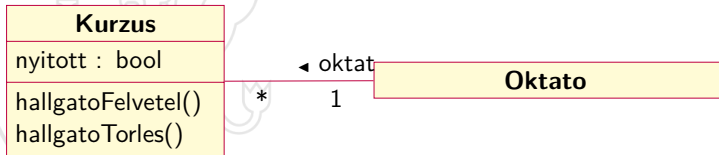
hallgatoFelvetel()
hallgatoTorles()

- Csomagokba szervezett osztályok önmagukban nem elegendőek (modell)
 - nehezen olvasható
- A modell különböző vetületei a tényleges megjelenítései a rendszernek
- Követhetik a modellt, de általában nem:
 - pl. a modell egy osztálya több különböző vetületen (diagramban) is megjelenhet
- Mindig van egy fődiagram, amely tipikusan a rendszer fő csomagjait tartalmazza
- Minden csomagnak is van saját fődiagramja, amely a publikus interfész osztályokat jeleníti meg

- Objektum-kölcsönhatások megvalósulásai
 - objektum-üzeneteknek a „csatornái”
- Kapcsolatok az osztálydiagramokon:
 - asszociáció
 - aggregáció és kompozíció (rész-egész kapcsolatok)
 - öröklődés
- Kapcsolatok tulajdonságai:
 - név, irány, szerep, multiplicitás, ...

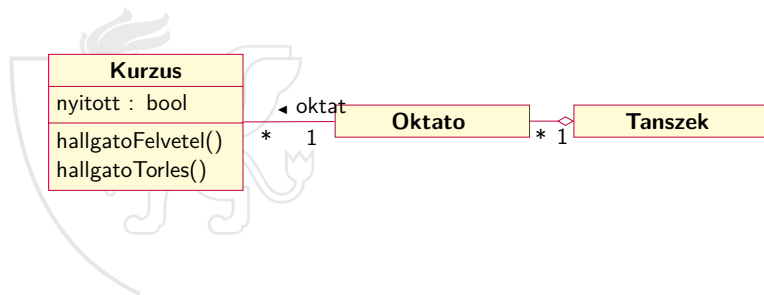


- Osztályok közötti kétirányú összeköttetés (navigálási irány megadható – üzenet iránya)
- „Használati kapcsolat”, létük egymástól általában független, de legalább az egyik ismeri és/vagy használja a másikat
- Szemantikus összefüggést ábrázol:
 - nem adatfolyam (mindkét irányba lehet információ/adat továbbítás)
- Gyakorlatilag az osztályokból létrejövő objektumok között van összefüggés



Aggregáció

- Asszociáció egy speciális formája
- „Rész-egész” kapcsolat (erősebb mint az asszociáció)
- Az egyik objektum fizikailag tartalmazza vagy birtokolja a másikat
- A rész-objektum(ok) léte az egész-objektumtól függ
- UML (rombusz a tartalmazó oldalon)



- Mikor legyen egy asszociáció helyett inkább aggregáció?
 - a kapcsolat leírásánál a „része” használható
 - az egész-objektum bizonyos műveletei automatikusan a rész-objektumokra is vonatkoznak (pl. megszűnés)
 - sok esetben nem egyértelmű (ugyanaz a kapcsolat két alkalmazásban másként jelenhet meg)
- Kétféle létezik
 - gyenge tartalmazás – általános aggregáció
 - erős tartalmazás (Kompozíció) – részek élettartama szigorúan megegyezik az egészével

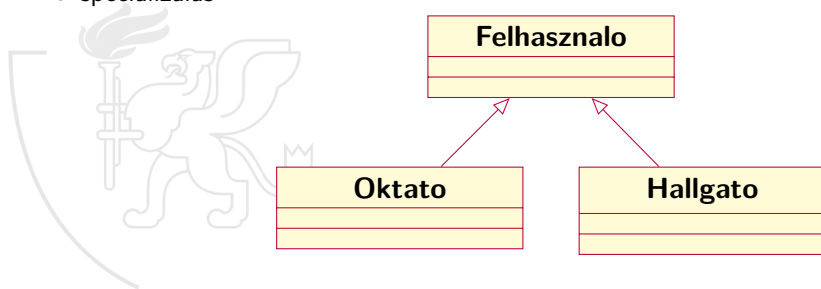


- Osztályok közötti kapcsolat (reláció), ahol egy osztály megosztja a struktúráját és/vagy a viselkedését egy vagy több másik osztállyal
- Öröklődési hierarchia
 - származtatott osztály örököl az ősz osztály(ok)tól
- Az attribútumokat és operációkat a lehető legfelsőbb szinten kell definiálni
- A származtatott (gyerek) osztály mindent örököl az őstől (relációkat is) és kiegészítheti ezeket sajátokkal

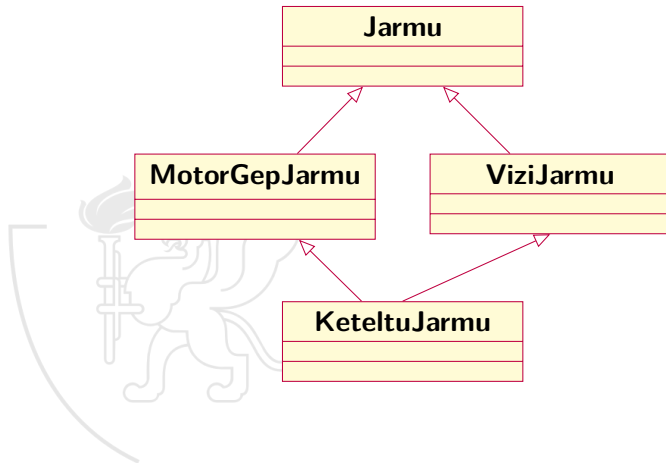


Öröklődés (folyt.)

- A származtatott osztály egy örökölt operáció saját implementációját is nyújthatja: polimorfizmus (felüldefiniálás, overriding)
- Az öröklődés relációnak nincs neve, multiplicitása
- Tipikus öröklődési szintek száma: 3-5
- Az újrafelhasználhatóság egyik alapeszköze
- Öröklődés feltárása
 - általánosítás és
 - specializálás



- Példa: a kételtű jármű egy motorgépjármű (ami egy jármű) és egyben egy vízi jármű is (ami ugyancsak egy jármű)



Többszörös öröklődés (folyt.)

- Problémák adódhatnak
Pl.:
 - név ütközések
 - többszörösen örökölt operációk, attribútumok
- Megoldható: C++ virtuális öröklődés
- Kevésbé karbantartható kódhoz vezet
- csak akkor szabad használni, ha tényleg szükséges, de akkor is csak nagy odafigyeléssel
- Java-ban nincs rá lehetőség

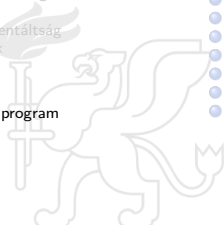
Programozás I.

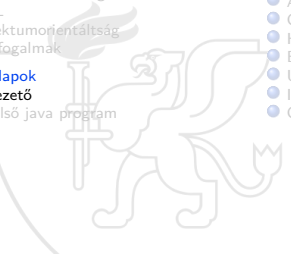
Dr. Ferenc Rudolf
Dr. Jász Judit

Szegedi Tudományegyetem
Informatikai Intézet
Szoftverfejlesztés Tanszék

2021-02-08



- 
- 1 **Bemutakozás**
 - Kurzus információk
 - 2 **Objektumorientált tervezés**
 - A modellezés alapelvei
 - A „siker háromszöge”
 - UML
 - Objektumorientáltság alapfogalmak
 - 3 **Java alapok**
 - Bevezető
 - Az első java program
 - 4 **Programozási alapismeretek ismétlése**
 - **Java**
 - Objektumok, osztályok
 - Polimorfizmus
 - Annotációk
 - Objektumok kezelése
 - Hozzáférés szabályozása
 - Beágyazott osztályok
 - Újrafelhasználhatóság
 - Interfészek és enumerációk
 - Objektumok tárolása
 - 5 **Tervezési minták**
 - Hibakezelés kivételekkel
 - Sztringek
 - Generikus típusok
 - Típus információk
 - I/O
 - GUI; Belső, lokális és anonim osztályok, lambdák

- 
- 1 **Bemutakozás**
 - Kurzus információk
 - 2 **Objektumorientált tervezés**
 - A modellezés alapelvei
 - A „siker háromszöge”
 - UML
 - Objektumorientáltság alapfogalmak
 - 3 **Java alapok**
 - **Bevezető**
 - Az első java program
 - 4 **Java**
 - Programozási alapismeretek ismétlése
 - Objektumok, osztályok
 - Polimorfizmus
 - Annotációk
 - Objektumok kezelése
 - Hozzáférés szabályozása
 - Beágyazott osztályok
 - Újrafelhasználhatóság
 - Interfészek és enumerációk
 - Objektumok tárolása
 - 5 **Tervezési minták**
 - Hibakezelés kivételekkel
 - Sztringek
 - Generikus típusok
 - Típus információk
 - I/O
 - GUI; Belső, lokális és anonim osztályok, lambdák

- A kurzus célja, hogy elsajátítsuk
 - az objektum orientált gondolkodásmódot,
 - az osztálydiagramok használatát és
 - a Java-t, mint tisztán objektum orientált nyelvet
- Java
 - újabb nyelv, sok jó tulajdonságot átvett: C, C++, Smalltalk, ...
 - komplexitást csökkenti a programozó szempontjából



- A komplexitást általában növeli
 - a követelmények bővülése: a nyelvek is bonyolultabbak lesznek
 - kompatibilitási problémák: C++ kompatibilis a C-vel, Perl a Seddel, Visual Basic a BASIC-kel, ...
- Komplexitás kezeléséből adódik
 - könnyebb robusztus kódot írni
 - lassabban fut
- Hátrányai viszont megtérülnek
 - gyorsabban meg lehet tanulni a nyelvet
 - könnyebb megbízható programot írni

A Java nyelv

A Java fő jellemzői:

- egyszerű
- objektumorientált
- előfordított
- értelmezett
- robusztus
- biztonságos
- semleges architektúrájú
- hordozható
- többszálú
- dinamikus

A Java nyelv (folyt.)

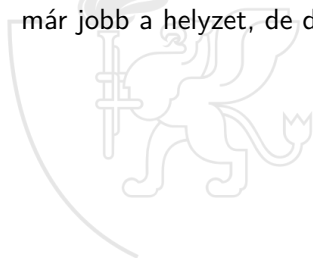
- Magas szintű programozási nyelv
 - nem a kódoláson van a hangsúly, hanem a tervezésen
 - elvontabb fogalmakat ábrázol (pl. interfész)
- Web programozás elősegítése
 - platform-független
 - beépített biztonsági rendszerek
- Programozók és programozási eszközök kommunikációjának elősegítése
 - nem „csak” egy nyelv
 - számos egyéb alkalmazhatóság (applet, beágyazott rendszerek, . . .)

Miért ennyire sikeres?

- Fő célkitűzés: Produktivitás
- Alapoktól indulva lett megtervezve, az eddigi programozási nyelvek tapasztalatait felhasználva
- OOP
- Jó osztálykönyvtárak
- Hibakezelés nyelvi szinten (kivételek)
- Nagy rendszerek írására is alkalmas
- Könnyű elsajátítani egy olyan szinten, amellyel már megbízható program írható
 - C++ nehezebb és ezért kevesebb a jó C++ programozó

Java vs. C++

- Régebben állították, hogy a Java leváltja majd a C++-t
 - már nem így van
- A két nyelv más-más területeken előnyös
- Ha Web programozás: Java
- Ha új nyelvet kell megtanulni: a Java könnyebb
- Ha a program performanciája (idő, memória) fontos: C++
- Első interpretált Java 20-50-szer lassabb volt mint a C/C++! Mára már jobb a helyzet, de drámai áttörés nem történt



A Java platform

Komponensei:

- Java API: több ezer használatra kész szoftverkomponens, amely csomagokba szervezett osztályokat és interfészeket tartalmaz
- Java Virtual Machine: a Java bájtkódot futtató virtuális gép
 - célja, hogy biztosítsa a hordozhatóságot, platform-független
 - JIT (Just in time) fordító: bájtkódot natív kódra fordítja → futtatja



- Natív kód

- a gép által végrehajtható utasítások sorozata
- csak a célgépen futtatható
- natív fordító: natív kódot készít

- Bájt kód

- a magas szintű nyelv és a natív kód közötti platform
- gépfüggetlen közbenső kód
- csupán annyival magasabb szintű a natív kódnál, hogy platform-független legyen (biztosítsa a hordozhatóságot)
- alap JVM (Java Virtual Machine – Java Virtuális Gép) direktben futtatja
- JIT: natív kóddá alakítja, majd úgy futtatja

- a Java a Web programozására is ad megoldást
- a Web elemei programozási szempontból
 - kliens/szerver elv
 - szerver: ahol az adat és a program van
 - kliens: aki eléri, feldolgozza, vagy csak megjeleníti
 - a Java mindkettőre ad megoldást



- Kezdetben minden interaktivitás a szerver oldalán volt
 - statikus HTML, CGI – Common Gateway Interface (bonyolult, lassú)
- A kliens programozása felé terelődött a hangsúly
 - tehermentesíti a szervert
- Hasonló a hagyományos programozáshoz, csak egy másik platformra
 - a web böngészőt egy korlátozott operációs rendszernek lehet tekinteni



Kliens programozás (folyt.)

- Szkript nyelvek
 - HTML-ben van benne a forráskód
 - speciális problémákra speciális nyelvek
 - JavaScript (semmi köze a Java-hoz), Ajax, ActionScript (Flash), Dart, TypeScript, ...
 - a legtöbb böngésző támogatja a JavaScriptet alapból
- Java technológia kliens programozáshoz: **applet**
 - csak böngészőn belül fut
 - minden böngészőn működik ahol van Java plug-in
 - nem a forráskód van értelmezve, hanem a lefordított kód kerül futtatásra (JAR: tömörített formátum)
 - elavult, a Java 1.9-ben már „deprecated”, a Java Web Startot ajánlják helyette

- Lehet egyszerű
 - fájl kérés
- Lehet bonyolult
 - adatbázis lekérés + HTML formázás
- Hagyományosan Perlt, CGI-t használtak, manapság inkább PHP-t, Java-t vagy .NET-et
- Java-ban: **servlet**, **JSP**, keretrendszerek



- Servlet
 - a szerveren tárolt és futó (általában kis méretű) alkalmazás, amely a hasonló funkciókat ellátó CGI-hez hasonlítva gyorsabb
- JSP (Java Servlet Pages)
 - servletekre épülő technológia, mely a tartalom és a megjelenítés szétválasztását és dinamikus oldalgenerálást céloz meg úgy, hogy a megjelenítendő HTML oldalt egy JAVA servlet állítja elő
- Keretrendszerek
 - Spring, Stripes, JBoss Seam, Struts, stb.



- A Java önálló programok írására is alkalmas
 - nem csak applet/servlet
- Előnyei
 - portabilitás
 - gyorsan írhatók megbízható, robusztus programok
- Android app-ok



Programozási irányelvek

- Az elegancia mindig kifizetődik
 - rövidtávon időpocsékolásnak tűnhet, de hosszútávon az újrafelhasználhatóságnak köszönhetően megéri
- Először legyen működőképes, utána legyen gyors
 - ha nem elég gyors, akkor profile-ozással határozzuk meg a lassú részeket
- Oszd meg és uralkodj
 - ha egy probléma túl bonyolultnak tűnik, képzeljük el a programunkat úgy, mintha a nehéz rész már meg lenne oldva, és csak használjuk azt
 - a nehéz rész általában újra felosztható



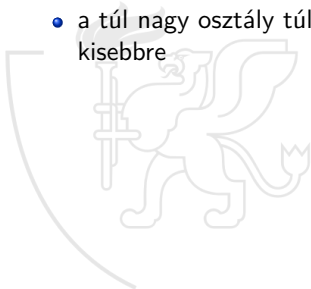
- Clean Code

- Osztály és metódus készítéskor használj olyan neveket, hogy ne kelljen kommentezni őket

- Az modellezés és tervezés eredményeképpen automatikusan létre lehessen hozni az osztályokat az interfészeikkel és más osztályokkal való kapcsolataikkal (főleg az öröklődéssel) együtt

- Az osztályok legyenek kompaktak

- a túl nagy osztály túl komplex lesz, ilyenkor célszerű szétdarabolni több kisebbre



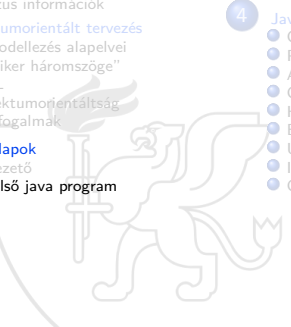
Programozási irányelvek (folyt.)

- Használd a függvény-kiterjesztést (overloading)
 - egy függvény ne foglalkozzon több különböző dologgal bemenő paraméter értéke alapján
- Vigyázz a „gigantikus objektum” szindrómára
 - tipikusan a kezdő OO programozók tévedése, hogy mindent belezsúfolnak egy osztályba és hagyományos procedurális módon oldják meg a feladatot
- Ha valami csúnyát kell csinálni, akkor az szigorúan legyen bezárva egy osztályba
- Ha nem hordozható kódot kell írni, akkor rejtse interfész mögé és zárd be egy implementációs osztályba

Programozási irányelvek (folyt.)

- Először a kompozíciót válaszd, amikor új osztályt készítesz a már meglévőkből
 - csak akkor használj öröklődést, ha a tervek azt indokolják
- Amikor készek a tervek ellenőrizd át újra és magyarázd el egy kívülálló szakembernek
 - egy friss szempár kihozhat még triviális hibákat is
- Gondolj rá, hogy a programkódot sokkal többször olvassák, mint írják
 - a névhasználat, a kommentek, magyarázatok, példák sokat segíthetnek a programmegértésben



- 
- 1 **Bemutakozás**
 - Kurzus információk
 - 2 **Objektumorientált tervezés**
 - A modellezés alapelvei
 - A „siker háromszöge”
 - UML
 - Objektumorientáltság alapfogalmak
 - 3 **Java alapok**
 - Bevezető
 - Az első java program
 - 4 **Java**
 - Programozási alapismeretek ismétlése
 - Objektumok, osztályok
 - Polimorfizmus
 - Annotációk
 - Objektumok kezelése
 - Hozzáférés szabályozása
 - Beágyazott osztályok
 - Újrafelhasználhatóság
 - Interfészek és enumerációk
 - Objektumok tárolása
 - 5 **Tervezési minták**
 - Hibakezelés kivételekkel
 - Sztringek
 - Generikus típusok
 - Típus információk
 - I/O
 - GUI; Belső, lokális és anonim osztályok, lambdák

Az első Java program

```
import java.util.*;
public class HelloDate {
    public static void main(String[] args) {
        System.out.println("Hello, a datum:");
        System.out.println(new Date());
    }
}
```

```
Hello, a datum:
Fri Nov 03 13:10:07 CET 2017
```

- java.util kell a Date-hez
- java.lang impliciten adott
- java.lang.System.out egy statikus PrintStream objektum
- Ha önálló programot írunk Java-ban, akkor
 - az adott fájlban kell lennie egy osztálynak, amelynek ugyanaz a neve, mint a fájlnek és
 - ez az osztály kell, hogy tartalmazzon egy statikus main metódust a fenti alakban

Fordítás és futtatás

- Kell egy „Java programming environment”
- Korábban JDK és JRE külön, mostmár egyetlen Java környezet (<http://www.oracle.com/technetwork/java/>)
- Fordítás: forrás (.java) → bytecode (.class)

```
javac HelloDate.java
```

Futtatás

```
java HelloDate
```

- Szokás build fájlokat használni
 - ant, gradle, makefile, stb.
- Fejlesztőkörnyezetek

- Több soros (C-szerű)

```
/* Ez egy  
több soros  
megjegyzés */
```

- Egy soros (C++ szerű)

```
// Ez egy megjegyzés a sor végéig
```



- Egy Java forráshoz hozzátartozik annak dokumentációja speciális megjegyzések formájában: `/** ... */`
- **javadoc**: program, amely összegyűjti ezeket a megjegyzéseket és készít egy HTML dokumentumot
- HTML-t is be lehet ágyazni, ezeket a javadoc beágyazza

```
/**  
    Megjegyzés a soron következő  
    osztály/operáció/attribútum-hoz  
*/  
class A...
```

- Tag-ek:

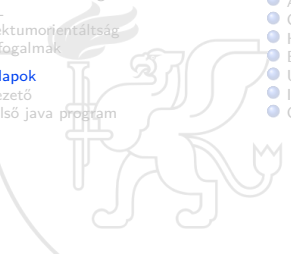
- **@see:** más osztályokra hivatkozás – linket szúr be
- **@version:** verzió
- **@author:** szerző
- **@param:** metódus paraméter szerepe
- **@return:** metódus visszatérési értékének leírása
- **@throws:** metódus milyen kivételeket dobhat



- A dokumentációt különböző formátumokban képes előállítani, pl.:
 - HTML, XML, PDF, RTF, DocBook, TeX
- Ellenőrizheti, hogy nem hiányzik-e valahonnan a dokumentációs megjegyzés, így a generált dokumentációban nem lehet üres folt
- Speciális osztályok, mint pl. JavaBeanek, illetve Servletek, dokumentálására is testre szabható



- Nem hivatalos Java standard
- Osztály nevek
 - nagy kezdőbetűvel kezdődnek
 - ha összetett szó, akkor nincs elválasztó karakter, hanem minden szó nagy betűvel kezdődjön.
 - Pl.: EzEgyJoOsztalyNev
- Metódus-, mező nevek:
 - ua., mint az osztály nevek, csak kisbetűvel kezdődnek
 - Pl.: ezJoTagNev
- {} zárójelezés, tabulálás mint a fóliákon

- 
- 1 **Bemutakozás**
 - Kurzus információk
 - 2 **Objektumorientált tervezés**
 - A modellezés alapelvei
 - A „siker háromszöge”
 - UML
 - Objektumorientáltság alapfogalmak
 - 3 **Java alapok**
 - Bevezető
 - Az első java program
 - 4 **Programozási alapismeretek ismétlése**
 - **Java**
 - Objektumok, osztályok
 - Polimorfizmus
 - Annotációk
 - Objektumok kezelése
 - Hozzáférés szabályozása
 - Beágyazott osztályok
 - Újrafelhasználhatóság
 - Interfészek és enumerációk
 - Objektumok tárolása
 - 5 **Tervezési minták**
 - Hibakezelés kivételekkel
 - Sztringek
 - Generikus típusok
 - Típus információk
 - I/O
 - GUI; Belső, lokális és anonim osztályok, lambdák

Minden objektum...

- A Java nyelv feltételezi, hogy objektum orientáltan fognak programozni benne
- A programozónak objektum orientáltan kell gondolkodnia
- Egy Java program alapvető komponenseit tekintjük át, és látni fogjuk, hogy minden objektum, még maga a Java program is



Objektum referenciák

- Adatok (objektumok) manipulálása általában
- direkt (pl. C-ben egy változó)
- indirekt (pl. C/C++-ban pointer; C++ és Java-ban referencia)
- Java-ban minden objektum, és mindig referenciával hivatkozunk rájuk (kivéve a primitív típusokat)
- Példa
 - televízió \longleftrightarrow objektum
 - távirányító \longleftrightarrow referencia



Primitív típusok

- A primitív típusú adatok (hasonlóan a C/C++-hoz) a stacken keletkeznek és direkt elérésűek (nem referencia által)
- Primitív típusok (fix bitszélesség):

Primitív típus	Méret	Min	Max	Wrapper
boolean	-	-	-	Boolean
char	16 bit	Unicode 0	Unicode $2^{16} - 1$	Character
byte	8 bit	-128	+127	Byte
short	16 bit	-2^{15}	$+2^{15} - 1$	Short
int	32 bit	-2^{31}	$+2^{31} - 1$	Integer
long	64 bit	-2^{63}	$+2^{63} - 1$	Long
float	32 bit	IEEE ₇₅₄	IEEE ₇₅₄	Float
double	64 bit	IEEE ₇₅₄	IEEE ₇₅₄	Double
void	-	-	-	Void

Primitív típusok (folyt.)

- boolean értéke: true vagy false
- *Wrapper*: objektumba csomagolja a primitív típust és a heapen tárolja
 - szükség lehet rá, ha pl. kollekcióban szeretnénk őket tárolni (Java 1.5-ben ez már automatizált – autoboxing)

```
char c1 = 'x';  
Character c2 = new Character('x');
```

- BigInteger: Akármekkora egész szám
- BigDecimal: Akármekkora fix pontos szám
- Tömb
 - referenciákat tárol ill. primitív típusnál értékeket
 - automatikusan inicializálódik (null vagy 0)
 - indexhatár ellenőrzés van futáskor (biztonságos, de lassabb)

- Értékadás

- primitív típusoknál értékmásolás
- objektum referenciáknál csak referenciamásolás (igazi objektum másolás: `clone()` metódussal)

- Matematikai : +, -, *, /, %

- Összevont: +=, -=, *=, /=, %=

- Egy operandusú: +, -, ++, --

- Relációk: ==, !=, <, >, <=, >=

- primitív típusoknál érték összehasonlítás
- objektum referenciáknál csak referencia összehasonlítás (igazi objektum összehasonlítás: `equals()` metódussal)

Operátorok (folyt.)

- Bitműveletek: $\&$, $|$, \wedge (xor), \neg (not)
- Értékadással összevont: $\&=$, $|=$, $\wedge=$
- Biteltolás: \ll , \gg , \ggg (unsigned)
- Értékadással összevont: $\ll=$, $\gg=$, $\ggg=$
- Logikai: $\&\&$, $||$, $!$
 - C/C++-szal ellentétben csak boolean értékekre használható
 - összetett kifejezés csak addig értékelődik ki, amíg ki nem derül egyértelműen az értéke



Operátorok (folyt.)

- Háromoperandusú if-else
`boolean-exp ? val0 : val1`
- Vessző: ,
 - csak for ciklusban használható
- Típuskonverzió: `(type)value`
 - primitív típusok között használható korlátok nélkül (kivéve a `boolean-t`)
 - osztályok között csak egy öröklődési fán belül engedélyezett
- Nincs `sizeof()`



- Precedencia szabály:

Operátor típusa	Operátorok
Unáris postfix operátorok	++, -- , [], ., (<param>)
Unáris prefix operátorok	++, --, +, -, , !
Multiplikatív operátorok	*, /, %
Additív operátorok	+, - ,
Bitenkénti léptető operátorok	<<, >>, >>>
Összehasonlító	<, >, <=, >=, (instanceof)
Egyenlőség	==, !=
Bitenkénti ÉS	&
Bitenkénti kizáró vagy	^
Bitenkénti VAGY	
Logikai és	&&
Logikai VAGY	
Feltételes	?:
Értékadás	=, (és összetettek, pl.: +=)

Vezérlési szerkezetek

- A Java támogat minden C vezérlési szerkezetet, kivéve a goto-t
- Szekvenciális
- Szelekciós vezérlés

```
if (booleanExpression)
    statement
else
    statement
```

- Kezdőfeltételes ismétléses vezérlés

```
while (booleanExpression)
    statement
```

Vezérlési szerkezetek (folyt.)

- Végfeltételes ismétléses vezérlés

```
do
    statement
while (booleanExpression);
```

- Számlálásos ismétléses vezérlés

```
for(initialization; booleanExpression; step)
    statement
```

- break [label]: megszakítja az aktuális (vagy a labellel címkézett) ismétlést és a szerkezetet követő utasítással folytatja
- continue [label]: megszakítja az aktuális (vagy a labellel címkézett) ismétlést és a következővel folytatja
- return: metódusból való visszatérés

Vezérlési szerkezetek (folyt.)

- Eset kiválasztásos szelekciós vezérlés
 - byte, short, char, int primitív típusokra
 - felsorolási típusokra
 - String-re és wrapper osztályokra (Character, Byte, Short, Integer)

```
switch (selector) {  
    case value1: statement; break;  
    case value2: statement; break;  
    case value3: statement; break;  
    // ...  
    default: statement;  
}
```

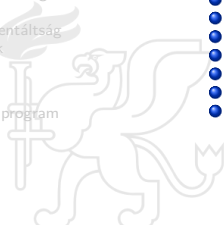
Programozás I.


Dr. Ferenc Rudolf
Dr. Jász Judit

Szegedi Tudományegyetem
Informatikai Intézet
Szoftverfejlesztés Tanszék

2021-02-08



- 
- 1 Bemutatók
• Kurzus információk
 - 2 Objektumorientált tervezés
• A modellezés alapelvei
• A „siker háromszöge”
• UML
• Objektumorientáltság alapfogalmak
 - 3 Java alapok
• Bevezető
• Az első java program
 - 4 Programozási alapismeretek ismétlése
• Java
• Objektumok, osztályok
• Polimorfizmus
• Annotációk
• Objektumok kezelése
• Hozzáférés szabályozása
• Beágyazott osztályok
• Újrafelhasználhatóság
• Interfészek és enumerációk
• Objektumok tárolása
 - 5 Hibakezelés kivételekkel
• Sztringek
• Generikus típusok
• Típus információk
• I/O
• GUI; Belső, lokális és anonim osztályok, lambdák
 - 6 Tervezési minták
• Minták áttekintése
• Gyártási minták
• Szerkezeti minták
• Viselkedési minták

- 
- 1 **Bemutakozás**
 - Kurzus információk
 - 2 **Objektumorientált tervezés**
 - A modellezés alapelvei
 - A „siker háromszöge”
 - UML
 - Objektumorientáltság alapfogalmak
 - 3 **Java alapok**
 - Bevezető
 - Az első java program
 - 4 **Java**
 - **Objektumok, osztályok**
 - Polimorfizmus
 - Annotációk
 - Objektumok kezelése
 - Hozzáférés szabályozása
 - Beágyazott osztályok
 - Újrafelhasználhatóság
 - Interfészek és enumerációk
 - Objektumok tárolása
 - 5 **Tervezési minták**
 - Hibakezelés kivételekkel
 - Sztringek
 - Generikus típusok
 - Típus információk
 - I/O
 - GUI; Belső, lokális és anonim osztályok, lambdák

- Minél bonyolultabb a programozás problémája, annál absztraktabb megoldások kellenek
- Assembly: absztrakció a gép felett
- Fortran, C: absztrakció assembly felett
 - még mindig gép-orientált elvonatkoztatás
- Probléma orientált absztrakció: LISP, APL
 - korlátozott az alkalmazhatóság
- OOP: Objektum orientált programozás
 - alkalmazhatóság-független absztrakció



- Objektum
 - alkalmazhatóság-független absztrakciója a probléma egy elemének
- Program
 - egymással kommunikáló objektumok összessége
- Minden objektumot kisebb objektumokból állítunk össze (akár alaptípusúakból)
- Osztály
 - az objektumok típusa
 - a `class` kulcsszóval definiáljuk
- Ugyanolyan típusú objektumok ugyanolyan üzeneteket fogadhatnak
 - a gyakorlatban ez relaxálható → polimorfizmus

- A sok egyedi objektum között vannak olyanok, melyeknek közös tulajdonságaik és viselkedéseik vannak, vagyis egyazon családba – osztályba tartoznak
- Simula-67 volt az első ilyen nyelv
- Az osztály egyben egy **absztrakt adattípus** is
 - adatok és a rajtuk végzett műveletek **egységbezárása (encapsulation)**
- Ugyanúgy viselkedik mint minden egyéb primitív típus
 - pl. változó (objektum) hozható létre



Objektum interfésze (folyt.)

- **Osztály:** tulajdonság (állapot) + viselkedés
 - mi lehet az állapot
- **Objektum:** az osztály egy példánya
 - mi az állapot
- Tulajdonság = **attribútumok** (adattagok, mezők)
- Viselkedés = **operációk** (metódusok, tagfüggvények)
- A viselkedések üzeneteken keresztül aktiválhatók
 - ~függvényhívás



Új típusok létrehozása

- Új, egy osztályba tartozó objektumokhoz típus hozzárendelése: `class` kulcsszóval. Pl.:

```
class Alakzat { /* az osztály törzse */ }  
// ...  
Alakzat a = new Alakzat();
```

- Önmagában még nem sok mindenre jó
 - csak létre lehet hozni,
 - de nem tud fogadni semmilyen üzenetet,
 - és nem tárol semmi hasznosat
- Testre kell szabni

Attribútumok hozzáadása

- Osztály attribútuma (mezője, adattagja) lehet
 - másik osztály típusú (referenciát tárol), létre kell hozni `new`-val (inicializálás)
 - primitív típusú (értéket tárol, inicializálódik)

```
class Alakzat {  
    int szin;  
    float terület;  
    Koordinata xy;  
}
```

```
Alakzat a = new Alakzat();  
// ...  
a.szin = 1;  
a.terulet = 23.99f;  
a.xy = new Koordinata(1,19);
```

- A tagok elérése: `"."` operátor segítségével
- Önmagában még csak adattárolásra használható

Operációk hozzáadása

- Funkcionalitást biztosít az objektumoknak, meghatározza, hogy milyen üzeneteket fogadhat
- Részei: név, paraméterek, visszatérési típus, törzs

```
visszateresiTípus operacioNev( /* paraméter lista */ ) {  
    /* metódus törzs */  
}
```

- Csak osztályoknak lehetnek operációi (metódusai)
- Metódus hívás = üzenet küldése az objektumnak

```
class Alakzat {  
    int szin;  
    float terület;  
    Koordinata xy;  
    int szine() {return szin;}  
}
```

```
Alakzat a = new Alakzat();  
// ...  
int sz = a.szine();
```

Osztályok használata

- Egy fájlban belül
 - egyszerűen használni kell, az sem baj, ha csak később lesz definiálva
- Külső osztályok (pl. Java osztálykönyvtárak)
 - `import` kulcsszóval be lehet hozni egy csomagot
 - pl. használni szeretném a láncolt listát

```
import java.util.LinkedList;
```

- pl. használni szeretnék mindent az utilból

```
import java.util.*;
```

- a default package és `java.lang` package automatikusan importálódik

Interfész és implementáció

- **Interfész:** mi az amit üzenhetünk (**deklaráció**)
- **Implementáció:** ami teljesíti a kérést (**definíció**)

```
public class Lampa {  
    private int fenyero = 100;  
    public void be() {fenyero = 100;}  
    public void ki() {fenyero = 0;}  
    public void fenyесit() {  
        if (fenyero < 100) fenyero++;  
    }  
    public void tompit() {  
        if (fenyero > 0) fenyero--;  
    }  
}  
...  
Lampa lampa1 = new Lampa();  
lampa1.be();
```

lampa1: Lampa

fenyero=100

Lampa

fenyero

be()

ki()

fenyесit()

tompit()

- OO programozás közben két feladatot szoktunk megoldani
 - osztályt gyártunk (magunknak vagy másoknak)
 - osztályt használunk (miénket vagy másét)
- Ha osztályt gyártunk, el kell rejtenünk az implementációt
 - a használója nem kell hogy ismerje, nem tud róla, így nem használhatja rosszul és nem is teheti tönkre
 - kevesebb lesz a programhiba
- Elérés vezérlése (láthatóság) – access specifiers (bővebben később)
 - elrejtésre
 - implementáció biztonságos módosítása
 - public, protected, private, alapértelmezett: „friendly” (package private: csomagon belül public, egyébként private)

this

- Egy operáció kódja csak egy példányban van a memóriában

```
class Alakzat {  
    public void rajzolj()  
    { /* ... */  
}
```

```
Alakzat a1 = new Alakzat();  
Alakzat a2 = new Alakzat();  
a1.rajzolj();  
a2.rajzolj();
```

- Honnan tudja a `rajzolj()`, hogy melyik objektumhoz lett hívva?
- Egy „titkos” implicit paramétert generál a fordító
 - hivatkozás az aktuális objektumra
 - `this`

*// a valóságban ez történik
// a fordító belsejében:*

```
class Alakzat { /*...*/  
    void rajzolj(Alakzat this) {  
        /*osztályon kívül van!*/  
    }
```

```
    rajzolj(a1);  
    rajzolj(a2);
```

this (folyt.)

- Mire jó? Amikor valamire explicite fel akarjuk használni, pl.:
 - ha a metódus formális paraméterneve megegyezik valamelyik attribútum nevével

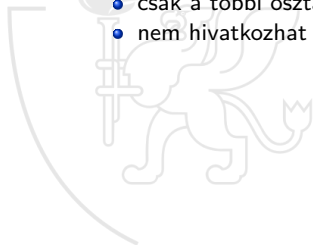
```
this.x = x; // az első az attribútum
```

- visszaadni egy metódussal

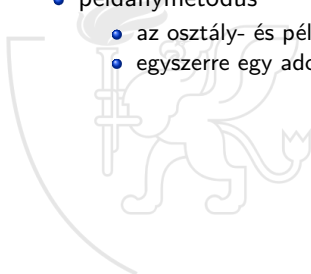
```
class Alakzat {  
    public void rajzolj() {  
        System.out.println("Alakzat");  
    }  
    public Alakzat kicsinyit() {  
        /*...*/ return this;  
    }  
}
```

```
Alakzat a1 = new Alakzat();  
a1.kicsinyit().kicsinyit();
```


- Az osztályhoz tartozik
 - az osztály minden objektumára egyformán érvényes
- Módosítója a `static`
- Osztálytag lehet
 - osztályváltozó (statikus attribútum)
 - egy darab él belőle és a statikus memóriaterületen tárolódik
 - az egyes objektumok osztoznak rajta
 - osztálymetódus (statikus operáció)
 - csak a többi osztálytagot látja
 - nem hivatkozhat a `this`-re



- Az objektumhoz (osztálypéldányhoz) tartozik
 - minden egyes objektumra különbözhet
- Minden tag, amelynek nincs static módosítója
- Példánytag lehet
 - példányváltozó
 - minden objektumban külön szerepel
 - értéke az objektum állapotára jellemző
 - példánymetódus
 - az osztály- és példánytagokat egyaránt látja
 - egyszerre egy adott objektumon dolgozik



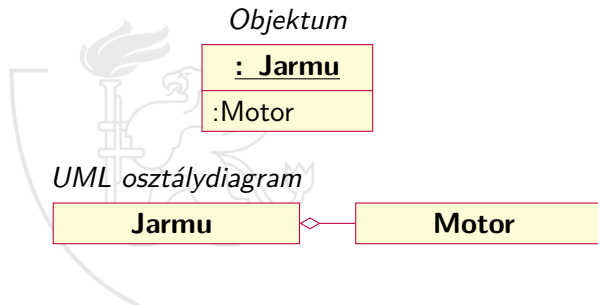
Osztálytag és példánytag

- Statikus metódus meghívható anélkül, hogy az osztályából objektumot hoznánk létre
- Statikus metódus csak az osztály statikus változóit és metódusait éri el
 - nem statikus metódusokat csak élő objektumokra lehet meghívni
- Kezdetben még nincsenek objektumok, így csak statikus metódusokat hívhatunk, ezért statikus a `main` is
 - `public static void main(String[] args)`
- Statikus metódust nem lehet felüldefiniálni (lásd később)
 - nem működik rá a polimorfizmus mechanizmus
 - korai kötést alkalmaz a fordító a függvényhívásra

- A metódus blokkjából hivatkozni lehet az osztály bármely tagjára
 - osztálymetódusból csak osztálytagra
- Egy adattag csak a fizikailag előtte deklarált tagokra hivatkozhat
 - osztályváltozó csak osztályváltozóra
- Lokális változóra csak az őt deklaráló metóduson belül lehet hivatkozni
- Metódusnak és attribútumnak lehet ugyanaz a neve
 - de nem ajánlott
- A lokális változók eltakarják az ugyanolyan nevű osztály- illetve példányváltozókat. Ilyen esetben:
 - az osztályváltozót az osztály nevével,
 - a példányváltozót a this referenciával minősítve lehet elérni

Implementáció újrafelhasználása

- Az újrafelhasználhatóság az OOP egyik legfontosabb előnye
- Kompozíció, aggregáció
 - objektum használata másik osztályban
 - futás közben változtatható
 - általában private
- (Öröklődés is egy alternatíva, de nem feltétlenül az a legjobb)

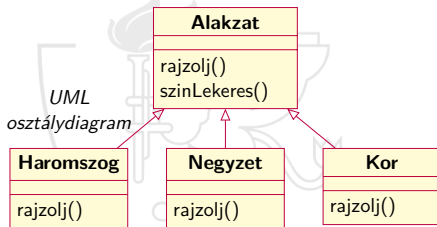


Interfész újrafelhasználása

- „Hasonló” osztályokat ne kelljen mindig újra és újra létrehozni
- Inkább „klónozni”, majd bővíteni/módosítani
- Ezt **öröklődésnek** nevezzük
 - ős, szülő, alap (base, super)
 - gyerek, leszármaztatott (derived, child)
 - ha az ős változik, a származtatott is „módosul”
 - elérhetőség fontos:
 - private nem elérhető (de az objektum része!)
 - protected, public igen
- Osztályhierarchiák

Öröklődés jelentése

- Hasonlóság kifejezése az ős felé
 - **általánosítás**
- Különbség a gyerek felé
 - **specializálás**
- A származtatott új típus lesz
 - az ős interfészét duplikálja
 - azonos típusú az ősse! (a kör az egy alakzat)



- Új attribútumokat veszünk fel
- Operációkat veszünk fel
 - teljesen új operációkat veszünk fel
 - módosítjuk az ős viselkedését (interfész marad)
 - **felüldefiniálás (overriding)**

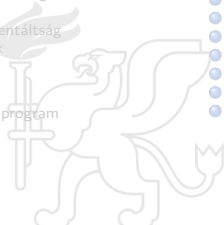


- Ősosztály típusú referencia mindig értékül kaphat leszármazott osztály típusú referenciát
 - leszármazottról az ősrre a konverzió implicit
 - csak egy részét látja és nem hivatkozik ki belőle, ezért biztonságos
- Leszármazott osztály típusú referencia típuskényszerítéssel értékül kaphat ősosztály típusú referenciát
 - ősről a leszármazottra a konverzió explicit kell legyen
(LeszarmazottTípus)osReferencia
 - csak öröklődési láncon belül lehet konvertálni
 - a statikus típusához képest nagyobb memóriát láthat, veszélyes lehet, ha rossz típusra konvertálunk
- Az objektum referencia **statikus** típusa a forráskódban deklarálásakor megadott osztály
- Az objektum referencia **dinamikus** típusa a futásidőben hivatkozott objektum osztálya

Egy gyökerű öröklődési hierarchia

- A legtöbb OOP nyelvben van egy beépített ősz
 - Java-ban: **Object**
 - minden osztálynak az őse (sajátjainknak is)
- Jó, mert nem kell mindig az alaptól hierarchiát építeni
- Bizonyos alap-funkcionalitások eleve elérhetőek lesznek
- Pl. argumentum-átadás is könnyebb (upcasting)
- Garbage collector működése egyszerűbb (lásd később)
- Kis megkötés



- 
- 1 **Bemutakozás**
 - Kurzus információk
 - 2 **Objektumorientált tervezés**
 - A modellezés alapelvei
 - A „siker háromszöge”
 - UML
 - Objektumorientáltság alapfogalmak
 - 3 **Java alapok**
 - Bevezető
 - Az első java program
 - 4 **Java**
 - Objektumok, osztályok
 - **Polimorfizmus**
 - Annotációk
 - Objektumok kezelése
 - Hozzáférés szabályozása
 - Beágyazott osztályok
 - Újrafelhasználhatóság
 - Interfészek és enumerációk
 - Objektumok tárolása
 - 5 **Tervezési minták**
 - Hibakezelés kivételekkel
 - Sztringek
 - Generikus típusok
 - Típus információk
 - I/O
 - GUI; Belső, lokális és anonim osztályok, lambdák

- Többalakúság
- Objektumok felcserélhetőségét biztosítja
- Objektumot őstípusa alapján kezeljük
 - a kód nem függ a specifikus típusoktól
 - utólag is lehet definiálni származtatottakat
- Az ős osztály interfészét használjuk
- A fordítóprogram nem tudja melyik konkrét operáció hívódik (örökölt vagy felüldefiniált)!
 - a programozó nem is kívánja megmondani
 - futás közben derül ki a konkrét típus alapján

- OOP kései kötés
 - fordítási időben csak az operáció kandidátusok adottak
 - Java-ban minden operáció hívás ilyen

Nem OOP – korai kötés

- hívott eljárás abszolút címe fordítási időben meg van adva (pl. C, Pascal)



Polimorfizmus (folyt.)

```
class Alakzat {  
    public void rajzolj() { /*A*/ }  
}  
  
class Haromszog extends Alakzat {  
    public void rajzolj() { /*H*/ }  
}  
  
class Negyzet extends Alakzat {  
    public void rajzolj() { /*N*/ }  
}  
  
class Kor extends Alakzat {  
    public void rajzolj() { /*K*/ }  
}
```

```
public class AlakzatPelda {  
    static void  
    csinald(Alakzat a) {  
        // ...  
        a.rajzolj();  
    }  
    public static  
    void main(String[] args) {  
        Kor k = new Kor();  
        Haromszog h  
            = new Haromszog();  
        Negyzet n  
            = new Negyzet();  
        csinald(k);  
        csinald(h);  
        csinald(n);  
    }  
}
```

„Elfelejteti” a típust?

- Az előző példában a `csinald(k)` hívás során „elveszik” a típus, mert a metódus `Alakzat`-ot vár, de `Kor` van átadva
 - egyszerűbb lenne, ha a `csinald` metódus `Kor`-t várna?
 - nem, mert akkor minden egyes alakzatra kellene külön `csinald` metódus!
 - sőt, ha újabb alakzatot szeretnénk hozzáadni, akkor mindig új metódus is kell (karbantarthatóság)!



- Könnyen bővíthető a program (nincs típus-specifikusság), a polimorfizmusnak köszönhetően tetszőleges új alakzatot felvehetünk.
- Alakzat helyett Kor van átadva
 - megtehető, mert a kör az egy alakzat is
 - **upcasting – beleolvasztás**
- A csinald metódusban nincs megkülönböztetés a típusok szerint!
- Melyik rajzolj() operáció fog hívódni?
 - fordítási időben nincs megkötve
 - futási időben derül ki

```
static void  
csinald(Alakzat a) {  
    // ...  
    a.rajzolj();  
}  
  
public static  
void main(String[] args) {  
    Kor k = new Kor();  
    ...  
    csinald(k);  
}
```

ős

származtatott

- Egy speciális mechanizmus működik, amely futás közben rendeli hozzá a megfelelő implementációt (kései kötés)

- **Egységbezáras** – absztrakt adattípus
 - adatok és a rajtuk végzett műveletek egységbezárasa
- **Öröklődés**
 - interfész újrafelhasználása
- **Polimorfizmus** - többalakúság
 - kései kötés

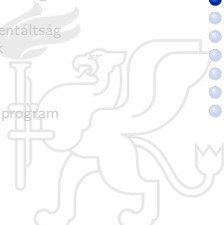


Az első Java program - értelmezve

```
import java.util.*;
public class HelloDate {
    public static void main(String[] args) {
        System.out.println("Hello, a datum:");
        System.out.println(new Date());
    }
}
```

```
Hello, a datum:
Fri Feb 16 14:19:05 CET 2018
```

- System tartalmaz egy statikus PrintStream típusú, out nevű objektumot (*aggregáció/kompozíció*)
- Date az Object-ből származik és felüldefiniálja a toString() metódust (*öröklődés*)
- println() az Object toString()-jét hívja, de ez kései kötésű hívás lesz, így a Date osztály toString()-je fog meghívódni (*polimorfizmus*)

- 
- 1 Bemutakozás
 - Kurzus információk
 - 2 Objektumorientált tervezés
 - A modellezés alapelvei
 - A „siker háromszöge”
 - UML
 - Objektumorientáltság alapfogalmak
 - 3 Java alapok
 - Bevezető
 - Az első java program
 - 4 Java
 - Programozási alapismeretek ismétlése
 - Objektumok, osztályok
 - Polimorfizmus
 - **Annotációk**
 - Objektumok kezelése
 - Hozzáférés szabályozása
 - Beágyazott osztályok
 - Újrafelhasználhatóság
 - Interfészek és enumerációk
 - Objektumok tárolása
 - 5 Tervezési minták
 - Hibakezelés kivételekkel
 - Sztringek
 - Generikus típusok
 - Típus információk
 - I/O
 - GUI; Belső, lokális és anonim osztályok, lambdák

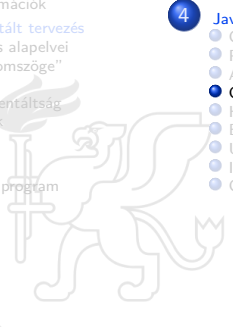
- Programmal kapcsolatos kiegészítő információkat (metaadatokat) írnak le
- Programkód elemeihez rendelhetők (csomagokhoz, típusokhoz, metódusokhoz, attribútumokhoz, konstruktorokhoz, lokális változókhoz)
- Plusz információt hordoznak a Java fordító ill. speciális eszközök számára
- Nincsenek közvetlen hatással annak a kódnak a működésére, amelyet annotálnak
- Felhasználási módjai:
 - Információkat szolgáltat a fordítónak hiba detektáláshoz
 - Fordítási és fejlesztési idejű utasításokat ad (kód, XML, dokumentum generáláshoz)
 - Futásidejű utasításokat ad, amelyeket a java reflection által feldolgozhatunk (pl.: objektum tulajdonságainak ellenőrzéséhez)
- Vannak eleve definiált annotációk, de mi magunk is írhatunk

Annotáció példák

```
public class A {  
    public void method() {}  
}  
  
public class B extends A {  
    @Override  
    public void method() {}  
}
```

```
/**  
 * @deprecated A D osztály használatát a 10.3-as verziótól  
 *             a D2 osztály váltja ki.  
 */  
@Deprecated  
public class D {}
```

```
public class C {  
    @SuppressWarnings({"deprecation"})  
    public void method() {  
        D obj = new D();  
    }  
}
```

- 
- 1 **Bemutakozás**
 - Kurzus információk
 - 2 **Objektumorientált tervezés**
 - A modellezés alapelvei
 - A „siker háromszöge”
 - UML
 - Objektumorientáltság alapfogalmak
 - 3 **Java alapok**
 - Bevezető
 - Az első java program
 - 4 **Java**
 - Programozási alapismeretek ismétlése
 - Objektumok, osztályok
 - Polimorfizmus
 - Annotációk
 - **Objektumok kezelése**
 - Hozzáférés szabályozása
 - Beágyazott osztályok
 - Újrafelhasználhatóság
 - Interfészek és enumerációk
 - Objektumok tárolása
 - 5 **Tervezési minták**
 - Hibakezelés kivételekkel
 - Sztringek
 - Generikus típusok
 - Típus információk
 - I/O
 - GUI; Belső, lokális és anonim osztályok, lambdák

- Registers
 - processzoron belül, gyors
- Stack
 - stack pointeren keresztül direkt elérésű memória a RAM-ban
- Heap
 - általános célú memória a RAM-on belül
- Static / Constant storage
 - konstans értékek helye, valamint maga a programkód
- Non-RAM storage
 - programon kívüli adatok

- Eddigiekből: OOP = egységbezárás + öröklődés + polimorfizmus
- Vannak egyéb fontos tényezők is
- Objektumok tárolási helye
 - **stack**: automatikus és gyors, de nem mindig megfelelő
 - **static**: statikus, nem flexibilis de gyors
 - **heap**: dinamikus, futás-közbeni, lassúbb
- Felszabadítás
 - **stack**: automatikus
 - **static**: automatikus
 - **heap**: Java-ban ez is automatikus

Objektumok élete Java-ban

- Java: mindig heap-en keletkeznek
 - kivéve a primitív típusokat
- Létre kell őket hozni
 - new kulcsszóval
- Felszabadítás
 - automatikus: *garbage collector* (szemétgyűjtő)
 - biztonságos, könnyebb a kezelés, de sokkal lassúbb



Inicializálás - konstruktor

- Régebbi nyelvekben a nem biztonságos programozás fő okai:
 - inicializálás hiánya
 - eltakarítás hiánya
- Java-ban egy különleges operáció hívódik meg minden objektum létrehozásakor
 - **konstruktor**
 - neve = az osztály neve
 - garantált inicializálás objektum létrejöttkor
 - helyette lehetne pl. `initialize()`, de ezt mindig kézzel kellene meghívni → hibalehetőség

```
class Alakzat {  
    /* attribútumok */  
    ...  
    Alakzat() {  
        /* inicializáló kód */  
        szin = 0;  
        terület = 0f;  
        xy = new Koordinata(0,0);  
    }  
}
```

Inicializálás - konstruktor (folyt.)

- A konstruktornak nincs visszatérési értéke
- A konstruktornak is lehetnek paraméterei
 - megadják, hogyan inicializáljunk
 - pl. csak úgy lehet objektumot létrehozni, hogy megadjuk a pozíciót is

```
class Alakzat {  
    /* attribútumok */  
    ...  
    public Alakzat(int x, int y) {  
        /* inicializáló kód */  
        szin = 0;  
        terület = 0f;  
        xy = new Koordinata(x,y);  
    }  
}
```

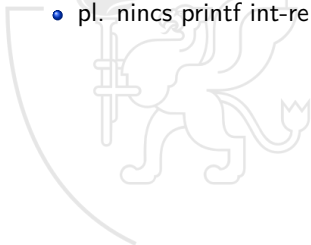
```
Alakzat a = new Alakzat(1,19);
```

A new operátor

- `new <OsztályNév>(<argumentumlista>)`
- Létrehoz egy új `OsztályNév` osztályú objektumot
 - lefoglalja számára a szükséges memóriát
 - meghívja az osztálynak a konstruktorát
 - visszaadja az új objektumra mutató referenciát
- Az objektum osztályát utólag megváltoztatni nem lehet



- Magasabb szintű nyelvekben neveket használunk
- Természetes nyelvben is lehet több értelme a szavaknak
 - a szövegkörnyezetből derül csak ki az értelme
- Programozásban ezt nevezzük **kiterjesztésnek (overloading)**
 - statikus polimorfizmus
 - (overriding = felüldefiniálás, más fogalom!)
 - túlterhelés (nem szerencsés magyar fordítás)
- Régebben, pl. C-ben, minden név egyedi volt
 - pl. nincs printf int-re és float-ra külön-külön



Operáció kiterjesztés (folyt.)

- Újabban szükségessé is vált
 - pl. a konstruktornak csak egy neve lehet, mégis különböző konstruálásokat szeretnénk
- Metódusok kiterjesztése (nem csak konstruktor)
 - ugyanaz a neve de más a paraméter-lista (akár üres is lehet)
 - ugyanaz a feladat, miért lenne több különböző nevű függvényünk?

```
class Alakzat {  
    public Alakzat() {  
        /* inicializáló kód */  
    }  
    public Alakzat(int x, int y) {  
        /* inicializáló kód */  
    }  
}
```

```
Alakzat a1 = new Alakzat();  
Alakzat a2 = new Alakzat(1,19);
```

Operáció kiterjesztés - default konstruktor

- Argumentum nélkül hívható konstruktor
 - **default** constructor
 - alapértelmezett beállításokkal rendelkező objektum létrehozására
- Ha nem definiálunk egy konstruktort sem, akkor a fordító készít egy defaultot
 - ha már van valamilyen (akár default akár nem) akkor nem készít



Operáció kiterjesztés (folyt.)

- A fordító a kiterjesztett metódusokat a paraméterlistájuk alapján különbözteti meg
- A hívás helyén az aktuális argumentumok száma és típusai határozzák meg
- Konvertálható: primitív típusoknál ha nincs pontos egyezés, akkor az adat konvertálódik (de csak nagyobb típusokra!)
- Függvény visszatérési érték nem használható a megkülönböztetésre
 - pl. `f()`; hívásnál a visszatérési érték nincs is használva



- Lokális változókat kötelező expliciten inicializálni
- Adattagok inicializálása
 - nincs expliciten inicializálva – a primitívek 0 alapértelmezett értéket kapnak, a nem primitívek pedig `null` értéket (nincs memória foglalva)
 - definíció helyén
 - instance initialization clause
 - konstruktor: az előző kettő után hívódik csak



Attribútumok inicializálása (folyt.)

- Példa:

```
class A {  
    char c;           // default  
    int i = 1;        // definíció helyén  
    float f = init(); // függvénnnyel  
    B a = new B();    // nem primitív  
    C c1;  
    C c2;  
    // instance initialization clause:  
    {  
        c1 = new C(1);  
        c2 = new C(2);  
    }  
    A() {i = 2;}      // default konstruktorral  
                    // előbb 1, utána 2  
    A(int i) {this.i = i;}  
}
```

- Nem csak a korrekt inicializálás fontos, hanem az erőforrások helyes kezelése is
- Java: **garbage collector**
 - csak memóriával foglalkozik
- Nem minden takarítást tud elvégezni
 - new nélküli memórafoglalás (pl. natív metódus által)
- Segítség: `finalize()`
 - takarítás előtt hívódik
 - nem destruktorkor!



- Fontos: a szemétgyűjtés végrehajtása nem szavatolt!
 - pl. még sok szabad memória van és elkerülhető a fölösleges lassítása a programnak
 - kiszámíthatatlan, hogy mikor fog hívódni
- Tehát ez nem azonos a destruktork szerepével
 - fontos, hogy a mindig végrehajtandó feladatokat ne a `finalize()`-ba tegyük (pl. fájl lezárás)



- Kézzel is lehet indítványozni a futtatását

```
System.gc();  
System.runFinalization();
```

- Működés: több algoritmus szerint, de ez mind rejtve marad
 - pl. referenciaszámlálás



- Tömb: névvel ellátott (egyetlen azonosítóval kezelt) azonos típusú elemek sorozata
 - Java-ban a tömb is egy objektum
 - deklarálás:

```
int [] a1;  
int a2 [];
```

- méretet nem lehet megadni, akkor foglalódik le amikor inicializáljuk
 - deklaráláskor csak egy referenciánk van tömb típusra
- Inicializálás
 - inicializáló kifejezéssel:

```
int [] a1 = {1, 3, 4};
```

- tömb hivatkozás másolással:

```
int [] a2 = a1;
```

- a lefoglalt terület elemei 0-ra lesznek inicializálva

- Tömb objektumok mérete
 - csak olvasható, pl.

```
for (int i = 0; i < a1.length; i++)  
    System.out.println(a1[i]);
```

- Biztonságos, mert túlindexelésnél nincs elszállás, hanem kivétel lesz dobva
- Ezek az extra funkcionalitások lassítják a programot
- Inicializálás futás közben
 - méret nem ismert előre

```
int i = /* ... */;  
int[] a1 = new int[i];
```

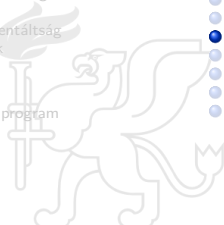
Tömbök (folyt.)

- Nem primitív típusok tömbjei
 - csak referenciák tárolódnak
 - ezért az elemeket new-val kell lefoglalni

```
Integer[] a1 = new Integer[3];  
a1[0] = new Integer(500);  
Integer[] a2 = new Integer[] {  
    new Integer(1),  
    new Integer(2),  
}; // new Integer[] elhagyható
```

- Többdimenziós tömbök
 - hasonló az egydimenzióshoz

```
int[][] a2d = { {1,2,3}, {4,5,6} };  
int[][][] a3d = new int[2][3][5];
```


- 
- 1 **Bemutakozás**
 - Kurzus információk
 - 2 **Objektumorientált tervezés**
 - A modellezés alapelvei
 - A „siker háromszöge”
 - UML
 - Objektumorientáltság alapfogalmak
 - 3 **Java alapok**
 - Bevezető
 - Az első java program
 - 4 **Java**
 - Programozási alapismeretek ismétlése
 - Objektumok, osztályok
 - Polimorfizmus
 - Annotációk
 - Objektumok kezelése
 - **Hozzáférés szabályozása**
 - Beágyazott osztályok
 - Újrafelhasználhatóság
 - Interfészek és enumerációk
 - Objektumok tárolása
 - 5 **Tervezési minták**
 - Hibakezelés kivételekkel
 - Sztringek
 - Generikus típusok
 - Típus információk
 - I/O
 - GUI; Belső, lokális és anonim osztályok, lambdák

- OO Tervezés
 - különválasztani a változó dolgokat a nem változóktól
- Osztálykönyvtárak esetén különösen fontos
 - kliens kódja változatlan maradhat, ha a könyvtár változik
 - kívülről elérhető részek ne változzanak, de
 - szabadon változhat a rejtett rész
- Az ilyen fajta elrejtést szolgálják az **elérést vezérlők (access specifiers)**



Implementáció elrejtése (folyt.)

- Elérések (növekvő szigorúság szerint):
 - public, protected, friendly (package private, nincs rá kulcsszó), private
- Csomag egységeinek elérését hogyan szabályozhatjuk?
 - csomagok: package
 - elérés vezérlésének teljes értelme a csomagok használatával jön elő
 - import-tal érünk el egy csomagot

```
import java.util.*;
```

- Lehet csak egy osztályra is kiadni

```
import java.util.ArrayList;
```

- import használata után a `java.util.ArrayList` helyett elég `ArrayList`-et írni
- Miért van rá szükség?
 - logikai csoportosítás
 - nevek ütközésének elkerülése
- Ha nincs megadva (mint az eddigi példákban), akkor minden új osztály egy ún. **default package** csomagba kerül
- Ha nagy rendszert írunk, célszerű csomagokat használni

- Egy-egy Java program írásakor fordítási egységeket készítünk (compilation-, translation unit)
 - .java → .class fájl(ok)
- Minden fordítási egységben lehet egy publikus osztály (`public class ...`), amelynek a neve meg kell egyezzen a fájl nevével (.java nélkül)
- Lehetnek a fájlban egyéb (rejtett) osztályok is
 - .java fájl fordításával .class fájlok jönnek létre (egy-egy minden egyes osztályhoz)
- Java-ban nincs linker (a .class fájlok önállóak)
 - lehet őket tömöríteni .jar fájlba (~zip)
 - .class fájlokat a virtuális gép értelmezi és hajtja végre egyenként

Csomagok megadása

- A csomag (package) nem más mint .class fájlok halmaza
- Melyik csomagba tartoznak?
- A forrásfájl első nem-komment sorába be kell írni:

```
package mypackage;
```

- Ez azt jelenti, hogy a benne levő osztályok a megadott csomaghoz fognak tartozni
- Használata, mint már láttuk

```
mypackage.MyClass m;
```

```
import mypackage.*;  
MyClass m;
```

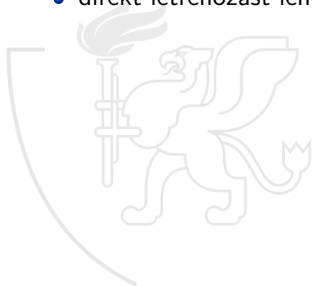
- Az egy csomagba tartozó osztályok egy lokális könyvtárba kerülnek

- Egyedi csomagnevek
 - a teljes elérési útvonal bele van kódolva
 - konvenció szerint az internet domain név is benne van fordított sorrendben (pl. hu.u-szeged.inf.java.*)
 - nem kötelező a használata, de javasolt
- Hogy a Java virtuális gép megtalálja, be kell állítani a CLASSPATH környezeti változót, amelynek tartalmaznia kell
 - minden olyan elérési útvonalat, ahol a Java csomagok elhelyezkednek a lokális rendszeren
 - .jar fájlok esetében a fájlnev is hozzá van adva (nem csak a könyvtárnev)
- Név ütközések
 - ha két import valami.* van és mindkettőben van olyan nevű osztály, mint amelyet használni szeretnénk (amíg nem használunk ilyet, addig nincs baj) explicite ki kell írni a csomag nevét az osztály elé

- Az osztály tagjai elé oda kell írni az elérés vezérlés kulcsszavát (csak arra az egy tagra lesz érvényes):
 - `public`, `protected`, `private`
 - ha nincs kiírva, akkor alapértelmezett a “friendly”
- Friendly (“package private”, “package access”)
 - csomagon belül minden osztály eléri (`public`)
 - csomagon kívül nem elérhető (`private`)
 - szorosan kapcsolódó osztályokat lehet így csoportosítani (egymást használják)



- Ha nem friendly-t használunk
 - minden osztálynak saját magának kell megadnia, mi az, ami elérhető benne és kinek
 - ami nem kell másnak az mindig legyen private
 - az interfész rész legyen public
 - leszármazottaknak (és azonos csomagban levőknek) legyen protected
- Konstruktorok is megadhatjuk az elérését
 - direkt létrehozást lehet így korlátozni



Elérés vezérlése (folyt.)

- public
 - mindenki elérheti
 - az osztály interfésze
- private
 - senki nem érheti el, kivéve az adott osztály saját metódusait
 - főleg rejtett implementációra és adattagokra alkalmazzák
- protected
 - az adott osztály, annak származtatott osztályai és a csomagjában levő osztályok elérik, a többiek nem



Elérés vezérlése (folyt.)

- Példa

```
class Suti {  
    protected void harap() {}  
}
```

```
class Ludlab extends Suti {  
    public void megesz() {harap(); /* ... */}  
}
```

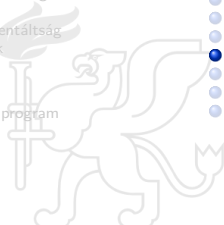
```
// másik csomag!  
public class SutiPelda {  
    public static void main(String[] args) {  
        Ludlab a = new Ludlab();  
        a.harap();    // nem elérhető  
        a.megesz();   // ok  
    }  
}
```

Osztály elérés vezérlése

- Könyvtáron (csomagon) belül az egyes osztályok elérését is lehet szabályozni
- public osztály: a kliens elérheti az osztályt

```
public class Alakzat { /*...*/ }
```

- .java fájlként csak egy publikus osztály lehet (de nem kötelező), az, amelyik a fájl nevét viseli
 - kis és nagy betű is számít!
- Publikuson kívül lehet még friendly, ha nincs semmi kiírva (private, protected csak inner class lehet)
 - csomagon kívül nem hozható létre belőle objektum

- 
- 1 **Bemutakozás**
 - Kurzus információk
 - 2 **Objektumorientált tervezés**
 - A modellezés alapelvei
 - A „siker háromszöge”
 - UML
 - Objektumorientáltság alapfogalmak
 - 3 **Java alapok**
 - Bevezető
 - Az első java program
 - 4 **Java**
 - Programozási alapismeretek ismétlése
 - Objektumok, osztályok
 - Polimorfizmus
 - Annotációk
 - Objektumok kezelése
 - Hozzáférés szabályozása
 - **Beágyazott osztályok**
 - Újrafelhasználhatóság
 - Interfészek és enumerációk
 - Objektumok tárolása
 - 5 **Tervezési minták**
 - Hibakezelés kivételekkel
 - Sztringek
 - Generikus típusok
 - Típus információk
 - I/O
 - GUI; Belső, lokális és anonim osztályok, lambdák

- Osztályon- vagy metóduson belüli osztályok
- Logikailag nagyon szorosan összetartozó osztályok csoportosítása
- Megadható a láthatóságuk
- Máshol nem használt algoritmus teljes elrejtése a külvilágtól
- A belső osztályból elérhetőek a „körülvevő osztály” elemei (kivéve, ha `static` a belső osztály)
- .class fájlnev: Külső\$Belső.class
- Nem kompozíció!

Belső osztályok - Példa

```
public class Csomag {  
    class Tartalom {  
        private int x = 19;  
        public int miVanBenne() {return x;}  
    }  
    class Cel {  
        private String cimzett;  
        Cel(String cimzett) {this.cimzett = cimzett;}  
        String miACim() {return cimzett;}  
    }  
    public void felad(String cel) {  
        Tartalom t = new Tartalom();  
        Cel c = new Cel(cel);  
        System.out.println(c.miACim());  
    }  
    public static void main(String[] args) {  
        Csomag cs = new Csomag();  
        cs.felad("Tanzania");  
    }  
}
```

- Belső osztályok
 - osztályon belül, vagy objektum példány által példányosítható
- Metóduson belüli belső osztályok, vagy lokális belső osztályok
 - csak a metóduson belül példányosítható
- Anonymous belső osztályok
 - egy helyen deklaráljuk és példányosítjuk
- (Példák később, a GUI résznél.)



.this és .new

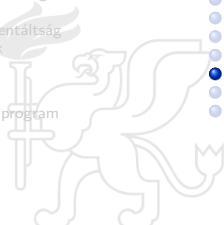
- **.this**
 - ha a belső osztályból a külső osztály objektumára szeretnénk hivatkozni
 - `KulsoOsztalyNeve.this`
- **.new**
 - belső objektum létrehozása külső class objektum által

```
public class Kulso {  
    public class Belso {  
        public Kulso kulso() {  
            return Kulso.this;  
        }  
    }  
    public static void main(String[] args) {  
        Kulso k = new Kulso();  
        Kulso.Belso kb = k.new Belso();  
    }  
}
```

Statikus belső osztályok

- Statikus tagja a külső osztálynak
- Tulajdonképpen egy sima osztály a másik osztály szkópjában belül
- A külső osztály példányosítása nélkül is elérhető
- Nem látja a külső osztály adattagjait és metódusait
- Példányosítása:

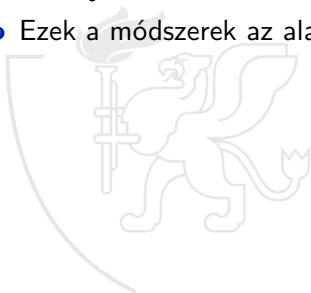
```
public class Kulso {  
    static class Belso {  
        public void foo() {  
            System.out.println("Statikus_belso_osztaly");  
        }  
    }  
    public static void main(String args[]) {  
        Kulso.Belso b = new Kulso.Belso();  
        b.foo();  
    }  
}
```

- 
- 1 **Bemutakozás**
 - Kurzus információk
 - 2 **Objektumorientált tervezés**
 - A modellezés alapelvei
 - A „siker háromszöge”
 - UML
 - Objektumorientáltság alapfogalmak
 - 3 **Java alapok**
 - Bevezető
 - Az első java program
 - 4 **Java**
 - Programozási alapismeretek ismétlése
 - Objektumok, osztályok
 - Polimorfizmus
 - Annotációk
 - Objektumok kezelése
 - Hozzáférés szabályozása
 - Beágyazott osztályok
 - **Újrafelhasználhatóság**
 - Interfészek és enumerációk
 - Objektumok tárolása
 - 5 **Tervezési minták**
 - Hibakezelés kivételekkel
 - Sztringek
 - Generikus típusok
 - Típus információk
 - I/O
 - GUI; Belső, lokális és anonim osztályok, lambdák

- OOP programozás (és a Java nyelv) egyik legvonzóbb tulajdonsága a kód hatékony újrafelhasználásának lehetősége
- De: újrafelhasználás \neq egyszerű másolás
- Régebbi nyelvekben a procedurális programozás volt az egyetlen lehetőség
- Ennél jobb az osztály újrafelhasználása
 - ne kelljen minden osztályt újból megírni, hanem építőköckaként felhasználni a meglévőket

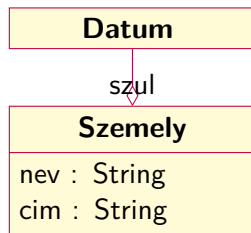


- A lényeg: mindezt úgy csinálni, hogy a meglévő kódot (osztályokat) ne módosítsuk
- Kompozíció
 - új osztályban meglévő osztályokból készítünk objektumot
- Öröklődés
 - új osztály létrehozása egy meglévő „altípusaként”
 - új funkcionalitás hozzáadásával
- Ezek a módszerek az alap-építőelemi az OOP-nek



- Kompozíció: összetétel, aggregáció, rész-egész kapcsolat
- Egy osztály egy attribútuma egy másik osztály vagy primitív típusú, pl.

```
class Szemely {  
    private String nev;  
    private Datum szul;  
    private String cim;  
}
```



- Minden OOP nyelvnek szerves része
- Java-ban minden új osztály implicite örököltetve van az Object-ből (direkt, vagy indirekt módon)
- Új osztály származtatása meglévőből: „olyan mint a régi”
 - bővíti azt (extends kulcsszó)
 - a származtatott az ős minden adatát és metódusát „megkapja” – öröklí az őstől
- Öröklődés
 - a származtatott- (al-, gyermek-) osztály egy speciális esete az ős osztálynak
 - az ős- (alap-, szülő-) osztály az általános esete a gyerek osztályoknak

- Az ősnek van néhány interfész metódusa (public)

```
class TisztitoSzer {  
    public void tisztits() {}  
    public void surolj() {}  
}
```

```
MosoPor mp = new Mosopor();  
mp.aztass();  
mp.tisztits();  
mp.surolj();
```

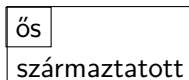
```
public class MosoPor extends TisztitoSzer {  
    // metódus módosítása (felüldefiniálás):  
    public void surolj() {super.surolj(); /* ... */}  
    // új metódus hozzáadása:  
    public void aztass() {}  
}
```

- A származtatott ezeket mind tartalmazza
 - az interfész újra fel van használva
- surolj() felüldefiniálja az ősét (overriding)
 - specializálja a viselkedését
 - az őst is hívhatja super segítségével (egyébként rekurzív lenne)
- aztass() új metódus, bővítése az ősnek

Ős osztály inicializálása

- Öröklődés nem csak az osztály interfészét „másolja”
- A származtatottból létrejövő objektumnak része az ősobjektum

- ős objektum ugyanolyan marad
+ ki van egészítve a
származtatott új elemeivel



- azonos a kezdőcímük
- Az őst is inicializálni kell
 - a származtatott konstruktorában hívni kell az ős konstruktorát (hiszen az tudja, hogyan kell)
 - a default ős-konstruktor automatikusan meghívódik (mindig a legősibb előbb és így lefelé)

Ős osztály inicializálása (folyt.)

- Ha nem default az ősz konstruktor, hanem vannak argumentumai, akkor kézzel kell meghívni a `super` segítségével
 - a konstruktor első utasítása az ősz konstruktorának hívása legyen

```
class Jatek {  
    Jatek(int i) { /* ... */ }  
}  
  
class TablaJatek extends Jatek {  
    TablaJatek(int i) { super(i); /* ... */ }  
}  
  
public class Sakk extends TablaJatek {  
    Sakk() { super(64); /* ... */ }  
}
```

- Sűrűn használatos a kettő együtt
- Ős osztály inicializálására a fordító kényszerít, de az adattag-objektumokra nekünk kell figyelni!



Kompozíció vagy öröklődés?

- Mindkettőnél az új osztály részobjektumokat tárol
- Mi a különbség és mikor melyiket használjuk?
- **Kompozíció:** egy meglévő osztály funkcionalitását felhasználjuk, de az interfészét nem
 - a kliens csak az új osztály interfészét látja
 - ezért a beágyazott objektum általában private elérésű
- **Öröklődés:** egy meglévő osztály speciális változatát készítjük el (specializálunk)
- Pl. melyik mi: Autó & Kerekek vagy Jármű & Autó

Kompozíció vagy öröklődés? (folyt.)

- Sokszor hajlamosak vagyunk arra, hogy mindenhol az öröklődést használjuk, mert az OOP tulajdonság!
- Nem mindig ez a jó
- Kezdetben indulunk ki inkább a kompozícióból, majd ha kiderül hogy az új osztály mégis egy speciális típusa a másiknak, akkor származtassunk
- Származtatásnak elsődlegesen akkor van létjogosultsága, ha ősré való konvertálás is szükséges lesz (lásd később)
- Túl sok származtatás és mély osztályhierarchiák nehezen karbantartható kódot eredményezhetnek!

Ösre konvertálás

- Mivel a származtatott az ősz osztály egy „új típusa”, logikus hogy mindenhol, ahol az ősz használható, ott a származtatott is használható
- Ez azt jelenti, hogy ahol ősz típusú objektumot vár a program, ott a származtatott egy implicit konverzió megy át az őszbe

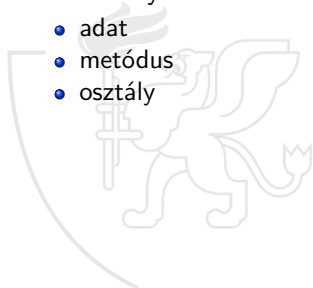
```
class Hangszer {  
    public void szolj() { /*...*/ }  
}
```

```
class Zongora extends Hangszer {  
    public void szolj() { /*...*/ }  
}
```

```
class Hangolo {  
    static void hangolj(Hangszer i) {  
        i.szolj();  
    }  
}
```

```
Zongora z = new Zongora();  
Hangolo.hangolj(z); // upcast - beleolvasztás
```

- A `final` kulcsszó
- Több jelentése van, de a lényeg: az, ami el van látva vele, az nem változhat (végső)
- Miért akarjuk korlátozni?
 - tervezési megfontolásból
 - hatékonyság miatt
- Három helyen használható
 - adat
 - metódus
 - osztály



- Konstans adat létrehozása hasznos lehet
 - ha már fordításkor eldönthető (konstans propagálás)
 - futás közben van inicializálva, de onnantól kezdve konstans
- Konstans primitív adat létrehozására egyszerűen ki kell írni a definíció elé (lehet static is)
- Nem primitív típusra használva nem az objektum lesz konstans, hanem a referencia
 - inicializálás után másra már nem mutathat, de maga az objektum megváltozhat
 - Java-ban az objektumok nem tehetők konstanssá, csak a viselkedésükkel lehet azt „szimulálni”
- Üres végsők (blank final)
 - a konstruktorban inicializálni kell
- Metódus paramétere is lehet final
 - csak olvasható

Végső adatok (folyt.)

- Példa:

```
class FinalData {  
    final int i = 9;  
    static final int i2 = 99; // csak egy van belőle  
    final Value v2 = new Value();  
    final int j = (int)(Math.random()*20); // futás közben!  
    final int k; // blank  
    FinalData(int i) {k = i;}  
    void f(final int i) {i++;} // hiba: i nem változhat  
}
```

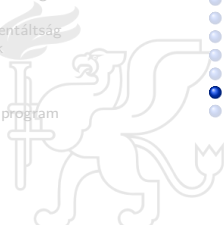


- Két dologra használjuk:
 - meggátoljuk a származtatott osztályokat abban, hogy felüldefiniálják és megváltoztassák a viselkedését
 - hatékonyság végett: a fordító számára megmondjuk, hogy a metódus hívását inline hívásokká alakíthatja:
 - hívás helyett a hívott függvény kódja „be lesz másolva”
- Minden `private` metódus `implicitely final` lesz
 - nem lehet felüldefiniálni, mert privát
 - ha mégis megpróbáljuk, akkor új metódust kapunk



- Osztályra is mondhatjuk, hogy legyen végső: `final class ...`
- Azt jelenti, hogy belőle nem lehet új osztályokat származtatni
- Biztonsági vagy hatékonysági megfontolásból
- Minden metódusa is implicite `final` lesz
 - nem lehet származtatni, ezért felüldefiniálni sem
 - ugyanazok a hatékonysági megfontolások érvényesek lesznek, mintha ki lenne írva



- 
- 1 **Bemutakozás**
 - Kurzus információk
 - 2 **Objektumorientált tervezés**
 - A modellezés alapelvei
 - A „siker háromszöge”
 - UML
 - Objektumorientáltság alapfogalmak
 - 3 **Java alapok**
 - Bevezető
 - Az első java program
 - 4 **Java**
 - Programozási alapismeretek ismétlése
 - Objektumok, osztályok
 - Polimorfizmus
 - Annotációk
 - Objektumok kezelése
 - Hozzáférés szabályozása
 - Beágyazott osztályok
 - Újrafelhasználhatóság
 - **Interfészek és enumerációk**
 - Objektumok tárolása
 - 5 **Tervezési minták**
 - Hibakezelés kivételekkel
 - Sztringek
 - Generikus típusok
 - Típus információk
 - I/O
 - GUI; Belső, lokális és anonim osztályok, lambdák

- Polimorfizmus

- A harmadik lényeges OOP tulajdonság az adatabsztrakció és öröklődés után
- Interfész és implementáció: szétválasztja a *mit* a *hogyan*-tól



Példa

```
class Hang {  
    private int magassag;  
    private Hang(int m) {magassag = m;}  
    public static final Hang  
        C = new Hang(0),  
        D = new Hang(1),  
        E = new Hang(2);  
}
```

```
class Hangszer {  
    public void szolj(Hang h) {  
        System.out.println(  
            "Hangszer.szolj()");  
    }  
}
```

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

```
class Zongora extends Hangszer {  
    public void szolj(Hang h) {  
        System.out.println(  
            "Zongora.szolj()");  
    }  
}
```

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

„Elfelejteti” a típust?

- Az előző példában a `Hangolo.hangolj(z)` hívás során „elveszik” a típus, mert a metódus `Hangszer`-t vár, de `Zongora` van átadva
 - egyszerűbb lenne, ha a `hangolj` metódus `Zongora`-t várna?
 - nem, mert akkor minden egyes hangszerre kellene külön `hangolj` metódus!
 - sőt, ha újabb hangszert szeretnénk hozzáadni, akkor mindig új metódus is kell (karbantarthatóság)!



- Honnan tudja a fordító, hogy melyik objektum szolgál metódusát hívja?
- **Nem** tudja!
- Csak futás közben derül ki az objektum dinamikus típusa alapján

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

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


- 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



- Az előző példában a Hangszer űrosztályban szereplő metódusok valójában „álmetódusok”
 - ha ezek véletlenül meghívódnak, akkor valami hiba van a programban, mert ezeknek a szerepe az, hogy interfész legyen
 - lehetne programfutas közben kiírni, hogy hiba van, de ennek az ellenőrzéséhez sok tesztelés szükséges
 - magából az űrosztályból valójában nem kellene, hogy létrejőjjön objektum
- Ezért bevezették az absztrakt osztályokat és metódusokat
 - fordítási időbeni ellenőrzés



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);
```

- 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 metódus nem lehet `private`, illetve `final`
 - nem lehetne felüldefiniálni és megvalósítani

Példa

```
class Hang {  
    private int magassag;  
    private Hang(int m) {magassag = m;}  
    public static final Hang  
        C = new Hang(0),  
        D = new Hang(1),  
        E = new Hang(2);  
}
```

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

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

```
class Zongora extends Hangszer {  
    public void szolj(Hang h) {  
        System.out.println(  
            "Zongora.szolj()");  
    }  
}
```

```
public class HangszerPelda {  
    public static void main(String[] args) {  
        Zongora z = new Zongora();           // -> Zongora.szolj()  
        //Hangszer z = new Hangszer();       // -> Fordítási hiba!  
        Hangolo.hangolj(z); // upcasting  
    }  
}
```

- Interfész: teljesen absztrakt osztály
- kulcsszó: `interface`
- metódus deklarációk
 - törzs nélkül
 - Java 8-tól megengedett a törzs deklarálása, feltéve, hogy `static`-ként, vagy `default`-ként lett deklarálv
 - impliciten `public` és `abstract`
 - nem lehet `private` (Java 9 előtt) és `final` (nem lehetne implementálni)
- mezők
 - impliciten `public`, `static` és `final` (kötelező inicializálni)
- Különválasztja az interfészt az implementáció(k)tól
- Egy protokollt valósít meg osztályok között
- Egy osztály több interfészből is származhat
 - több osztályból viszont nem
 - lehetővé teszi a „többszörös öröklést”

- Az `implements` kulcsszóval lehet implementálni („származtatni” belőle)

```
interface Hangszer {  
    void szolj(Hang h);    // impliciten public és abstract!  
}
```

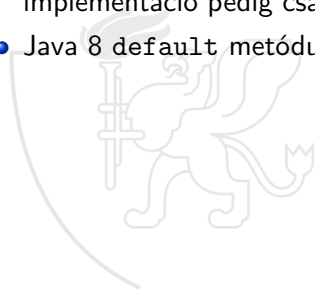
```
class Zongora implements Hangszer {  
    public void szolj(Hang h) {  
        System.out.println("Zongora.szolj()");  
    }  
}
```



- Interfész vagy absztrakt osztály?
 - ha őosztályt készítünk, akkor mindig kezdjük azzal, hogy interfész lesz
 - ha mégis szükség lesz az őosztály szintjén metódusokat implementálni vagy mezőket hozzáadni akkor módosítsuk absztrakt osztályra (esetleg konkrét osztályra)
- Interfész módosítói
 - `abstract`
 - felesleges, mert az interfész eleve absztrakt
 - kompatibilitási okokból van még jelen
 - ne használjuk
 - `public`
 - az interfész más csomagokból is elérhető lesz

„Többszörös öröklődés”

- Mivel az interfészek nem foglalnak tárterületet, lehetővé válik a biztonságos többszörös öröklődés
- Nem igazi többszörös öröklődés (mint pl. a C++-ban)
- Max. egy db. „igazi” osztály lehet az ősök között, a többi interfész kell legyen
- Valójában a specifikáció (interfész) örököltethető többszörösen, az implementáció pedig csak egyszeresen
- Java 8 default metódusai módosítják ezt a koncepciót



„Többszörös öröklődés” (folyt.)

- Interfész osztályok is származhatnak egymásból, ezzel új interfészt kapunk
- Egy interfésznek lehet több ősf interfésze
- Az interfészek öröklődési hierarchiájában nincsen közös ősf interfész, amelyből minden interfész származik (mint amilyen az osztályoknál az `Object`)



„Többszörös öröklődés” (folyt.)

- Példa

```
interface Kuzdo {  
    void kuzdj();  
}  
  
interface Uszo {  
    void usszal();  
}  
  
interface Repulo {  
    void repulj();  
}
```

```
class Szereplo {  
    public void kuzdj() { /*...*/ }  
}  
  
class AkcioHos  
    extends Szereplo  
    implements Kuzdo, Uszo, Repulo {  
    public void usszal() { /*...*/ }  
    public void repulj() { /*...*/ }  
}  
  
public class AkcioHosPelda {  
    public static  
    void main(String[] args) {  
        AkcioHos ah = new AkcioHos();  
        ah.kuzdj();  
    }  
}
```

Konstansok csoportosítása

- Mivel az interfész mezői hagyományosan `static`-ok és `final`-ok, jó eszköz arra, hogy konstansokat csoportosítsunk
 - Szimulálni lehet a C/C++-os `enum`-ot

```
interface Months {  
    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 HonapPelda {  
    public static void main(String[] args) {  
        System.out.println(Months.MAY);  
    }  
}
```

Enumerációk

- Java 1.5 nyelvi újítás
 - hasonlít a C/C++-os enum-ra

```
enum Months {  
    JANUARY, FEBRUARY, MARCH,  
    APRIL, MAY, JUNE, JULY,  
    AUGUST, SEPTEMBER, OCTOBER,  
    NOVEMBER, DECEMBER  
}
```

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

MAY

Enumerációk (folyt.)

Enum/HonapPelda2.java [1-23. sor]

```
1 enum Months {
2     JANUARY, FEBRUARY, MARCH,
3     APRIL, MAY, JUNE, JULY,
4     AUGUST, SEPTEMBER, OCTOBER,
5     NOVEMBER, DECEMBER
6 }
7
8 public final class HonapPelda2 {
9     public static void main(String[] args) {
10         Months m = Months.MARCH;
11         switch (m) {
12             case MARCH:
13                 System.out.println("Erkezik a tavasz:");
14                 break;
15             case JULY:
16                 System.out.println("Hurra, nyaralas!");
17                 break;
18             default:
19                 System.out.println("Atlagos honap...");
20                 break;
21         }
22     }
23 }
```

Erkezik a tavasz :)

Enumerációk (folyt.)


Enum/Months.java [1-27. sor]

```
1 enum Months {
2     JANUARY (31), FEBRUARY (28), MARCH (31),
3     APRIL (30), MAY (31), JUNE (30),
4     JULY (31), AUGUST (31), SEPTEMBER (30),
5     OCTOBER (31), NOVEMBER (30), DECEMBER (31);
6
7     private final int napokSzama;
8
9     Months(int napokSzama) {
10         this.napokSzama = napokSzama;
11     }
12
13     public int get() {
14         return napokSzama;
15     }
16
17     public static void main(String args[]) {
18         Months[] m = Months.values();
19         for (int i = 0; i < m.length; ++i) {
20             System.out.println(m[i] + " napjainak száma " + m[i].get());
21         }
22         // foreach szintaxissal (lásd később)
23         for (Months p : Months.values()) {
24             System.out.println(p + " napjainak száma " + p.get());
25         }
26     }
27 }
```

Enumerációk (folyt.)

```
JANUARY napjainak a száma 31
FEBRUARY napjainak a száma 28
MARCH napjainak a száma 31
APRIL napjainak a száma 30
MAY napjainak a száma 31
JUNE napjainak a száma 30
JULY napjainak a száma 31
AUGUST napjainak a száma 31
SEPTEMBER napjainak a száma 30
OCTOBER napjainak a száma 31
NOVEMBER napjainak a száma 30
DECEMBER napjainak a száma 31
```

- minden enum impliciten a `java.lang.Enum`-ból származik
- enum osztályok metódusai:
 - `values`
 - `private`, vagy `package private` konstruktor, amely inicializálja a konstansokat

- 
- 1 **Bemutakozás**
 - Kurzus információk
 - 2 **Objektumorientált tervezés**
 - A modellezés alapelvei
 - A „siker háromszöge”
 - UML
 - Objektumorientáltság alapfogalmak
 - 3 **Java alapok**
 - Bevezető
 - Az első java program
 - 4 **Java**
 - Programozási alapismeretek ismétlése
 - Objektumok, osztályok
 - Polimorfizmus
 - Annotációk
 - Objektumok kezelése
 - Hozzáférés szabályozása
 - Beágyazott osztályok
 - Újrafelhasználhatóság
 - Interfészek és enumerációk
 - Objektumok tárolása
 - 5 **Tervezési minták**
 - Hibakezelés kivételekkel
 - Sztringek
 - Generikus típusok
 - Típus információk
 - I/O
 - GUI; Belső, lokális és anonim osztályok, lambdák

- Csak nagyon egyszerű program esetében van előre meghatározott számú objektum ismert élettartammal
- Általában ez csak futási időben derül ki
 - kell, hogy legyen lehetőség objektumokat létrehozni bármikor, bárhol
 - ezeknek nem lehet mindig nevet adni. . .
- A megoldás: Kollektciók (konténerek)
 - Beépített típus: tömb (array)
 - Konténer osztályok a java.util könyvtárból



- Már bemutattuk a tömbök használatát
- Miért jobb a tömb a többi konténernél?
 - hatékonyság: Konstans idejű elérés, de fix a méret (flexibilis tömb: `ArrayList` konténer)
 - típus: Nem veszíti el a típust – adott típusú elemeket tárol, nem `Object`-eket
 - Java 1.5-ben a Generics-szel ez megoldódott
- Először mindig próbáljunk meg tömböt használni, és csak akkor válasszunk másik konténert, ha korlátba ütközünk



Az Arrays osztály

- A `java.util` könyvtár része
- Statikus metódusokat tartalmaz, amelyek segítséget nyújtanak a tömbkezelésben:
 - `equals()` – tömbök egyenlősége
 - `fill()` – tömb feltöltése adott értékkel
 - `sort()` – tömb rendezése
 - `binarySearch()` – tömbben keresés
 - `asList()` – List típusú listát készít belőle
- Mindegyik meg van valósítva (overload) minden primitív típusra és az `Object`-re

Tömbök egyenlősége, feltöltése

- Példa:

```
import java.util.*;
public class Tombok {
    public static void main(String[] args) {
        int[] a1 = new int[10];
        int[] a2 = new int[10];
        Arrays.fill(a1, 47);
        Arrays.fill(a2, 47);
        System.out.println(Arrays.equals(a1,a2));
    }
}
```

true

- Az általános rendezőalgoritmusoknak az elemek típusaitól függetleneknek kell lenniük
 - össze kell, hogy tudjon hasonlítani két elemet
 - típusoktól függetlenül kell, hogy működjön
- Callback technikával oldható meg
 - az elem saját maga hasonlítja magát össze egy ugyanolyan típusú másik elemmel
 - két módszer van: Comparable és Comparator



Tömbök rendezése (folyt.)

- „Természetes módszer”: Comparable
- A `java.lang.Comparable` interfész implementálása
- Egyetlen metódusa van: `compareTo()`
 - egy objektumot vár paraméterül
 - visszatérési érték: <0 , ha az aktuális objektum kisebb, mint az argumentum; $=0$, ha egyenlő és >0 , ha nagyobb, mint az argumentum



Tömbök rendezése (folyt.)

Holding/ComparablePelda.java [1-15. sor]

• Példa:

```
1 import java.util.*;
2
3 class Tancos implements Comparable {
4     double magassag;
5     public Tancos(double m) {
6         magassag = m;
7     }
8     public int compareTo(Object o) {
9         double másik = ((Tancos)o).magassag;
10        return (magassag < másik ? -1 : (magassag == másik ? 0 : 1));
11    }
12    public String toString() {
13        return String.format("%.2f", magassag) + " cm";
14    }
15 }
```

Tömbök rendezése (folyt.)

Holding/ComparablePelda.java [17–31. sor]

```
17 public class ComparablePelda {  
18     static void print(Tancos[] t) {  
19         for (int i = 0; i < t.length; i++)  
20             System.out.println(t[i]);  
21         System.out.println();  
22     }  
23     public static void main(String[] args) {  
24         Tancos[] t = new Tancos[10];  
25         for (int i = 0; i < t.length; i++)  
26             t[i] = new Tancos(Math.random()*100+100);  
27         print(t);  
28         Arrays.sort(t);  
29         print(t);  
30     }  
31 }
```

130,75	cm
177,28	cm
125,79	cm
115,60	cm
114,43	cm
176,19	cm
170,01	cm
194,40	cm
139,45	cm
152,51	cm
114,43	cm
115,60	cm
125,79	cm
130,75	cm
139,45	cm
152,51	cm
170,01	cm
176,19	cm
177,28	cm
194,40	cm

Tömbök rendezése (folyt.)

- Másik módszer: `Comparator`
- Ha nem implementálta az osztály a `Comparable` interfészt
- Egy külön osztályt kell létrehozni, amely a `java.lang.Comparator` interfészt implementálja
- Két metódusa van: `compare()` és `equals()`
 - általában csak a `compare()`-t kell implementálni, mert az `Object` őosztály implementálja a másikat
 - két objektumot vár paraméterül
 - visszatérési érték: <0 , ha az első argumentum kisebb; $=0$, ha egyenlő és >0 , ha nagyobb mint a második objektum

Tömbök rendezése (folyt.)

Holding/ComparatorPelda.java [1-19. sor]

• Példa:

```
1 import java.util.*;
2
3 class Tancos {
4     double magassag;
5     public Tancos(double m) {
6         magassag = m;
7     }
8     public String toString() {
9         return String.format("%.2f", magassag) + " cm";
10    }
11}
12
13 class TancosComparator implements Comparator {
14     public int compare(Object o1, Object o2) {
15         double m1 = ((Tancos)o1).magassag;
16         double m2 = ((Tancos)o2).magassag;
17         return (m1 < m2 ? -1 : (m1 == m2 ? 0 : 1));
18     }
19}
```

Tömbök rendezése (folyt.)

Holding/ComparatorPelda.java [21–35. sor]

```
21 public class ComparatorPelda {  
22     static void print(Tancos[] t) {  
23         for (int i = 0; i < t.length; i++)  
24             System.out.println(t[i]);  
25         System.out.println();  
26     }  
27     public static void main(String[] args) {  
28         Tancos[] t = new Tancos[10];  
29         for (int i = 0; i < t.length; i++)  
30             t[i] = new Tancos(Math.random()*100+100);  
31         print(t);  
32         Arrays.sort(t, new TancosComparator());  
33         print(t);  
34     }  
35 }
```

192,23	cm
140,96	cm
178,75	cm
179,86	cm
154,22	cm
148,87	cm
180,62	cm
168,70	cm
140,35	cm
127,21	cm
127,21	cm
140,35	cm
140,96	cm
148,87	cm
154,22	cm
168,70	cm
178,75	cm
179,86	cm
180,62	cm
192,23	cm

- Csak rendezett tömbben lehet keresni!
- `Arrays.binarySearch()` – bináris keresés
 - visszatérési érték: ≥ 0 , ha megtalálta (a keresett elem tömbindexe); < 0 , ha nem (melyik elem előtt kellene, hogy legyen)
 - ha több elem is megfelel a keresésnek, akkor bizonytalan, hogy melyiket találja meg (ilyenkor használjunk inkább `TreeSet`-et, lásd később)
 - ha `Comparator`-ral lett rendezve, akkor meg kell adni ugyanazt a `Comparator` objektumot a keresésnek is



Tömbben keresés (folyt.)

- Példa:

```
public class TombbenKereses {  
    public static void main(String[] args) {  
        int[] t = new int[100];  
        /* tömb feltöltése */  
        Arrays.sort(t);  
        /*...*/  
        int pozicio = Arrays.binarySearch(t,19);  
        if (pozicio >= 0) {  
            /* megvan */  
        }  
    }  
}
```

Tömbök másolása

- `System.arraycopy()` statikus metódus
- Meg van valósítva minden primitív típusra és az `Object`-re (csak a referenciák másolódnak!)

```
public class TombokMasolasa {  
    public static void main(String[] args) {  
        int[] i = new int[25];  
        int[] j = new int[25];  
        /*...*/  
        System.arraycopy(i, 0, j, 0, i.length);  
        /*...*/  
    }  
}
```

Kollekciók (konténerek)

- A konténer osztálykönyvtár egyike a leghatékonyabb programozási eszközöknek
- Java 2-ben teljesen újratervezték
 - a Java 1.x verziókban nagyon gyengére sikeredett
- Java 5-ben kibővítették generikus típusokra
- Két koncepció
 - **Kollekció (Collection)**: Egyéni elemek csoportja, általában valamilyen szabályossággal (listában sorrendiség, halmazban egyediség, stb.)
 - **Leképezés (Map)**: Kulcs-adat kettősök csoportja, a kulcsra gyors keresést biztosít



Kollekciók (folyt.)

Holding/Kollekciok.java [1-21. sor]

```
1 import java.util.*;
2
3 public class Kollekcio {
4     static Collection feltolt(Collection c) {
5         c.add("kutya");
6         c.add("macska");
7         c.add("macska");
8         return c;
9     }
10    static Map feltolt(Map m) {
11        m.put("kutya", "Odie");
12        m.put("macska", "Arlene");
13        m.put("macska", "Garfield");
14        return m;
15    }
16    public static void main(String[] args) {
17        System.out.println(feltolt(new ArrayList()));
18        System.out.println(feltolt(new HashSet()));
19        System.out.println(feltolt(new HashMap()));
20    }
21 }
```

```
[kutya, macska, macska]
[kutya, macska]
{kutya=Odie, macska=Garfield}
```


A Java 5 előtti kollekciók hiányossága

- Elveszítik a típust
 - **mindig** Object-eket tárolnak
 - általános célúak, nem tárolhatnak specifikus típusú objektumokat
- Bármilyen belerakható – nincs típusellenőrzés
 - pl. egy kutyákat tároló konténerbe nyugodtan bele lehet tenni macskákat
 - csak az Object referencia kerül bele (**upcast**)
 - **Downcast**-olni kell az elemet használat előtt!



A Java 5 előtti kollekciók ... (folyt.)

Holding/KutyakEsMacskak.java [1-30. sor]

```
1 import java.util.*;
2
3 class Kutya {
4     private int kutyaSz;
5     Kutya(int i) {kutyaSz = i;}
6     public void print() {
7         System.out.println(
8             "Kutya_#" + kutyaSz);
9     }
10 }
11 class Macska {
12     private int macskaSz;
13     Macska(int i) {macskaSz = i;}
14     public void print() {
15         System.out.println(
16             "Macska_#" + macskaSz);
17     }
18 }
19 public class KutyakEsMacskak {
20     public static void main(String[] args) {
21         ArrayList kutyak = new ArrayList();
22         for(int i = 0; i < 3; i++)
23             kutyak.add(new Kutya(i)); // upcast Object-re!!!
24         // nem problema berakni egy macskat a kutyak koze:
25         kutyak.add(new Macska(3)); // upcast Object-re!!!
26         for(int i = 0; i < kutyak.size(); i++) // downcast Kutya-ra!!!
27             ((Kutya)kutyak.get(i)).print();
28         // futas kozbeni hiba a 4. iteracioban (kiveteldobas)
29     }
30 }
```

A Java 5 előtti kollekciók ... (folyt.)

```
Kutya #0  
Kutya #1  
Kutya #2  
Exception in thread "main" java.lang.ClassCastException at  
KutyakEsMacskak.main(KutyakEsMacskak.java:27)
```



A Java 5 előtti kollekciók ... (folyt.)

Holding/KutyakEsMacskak2.java [1-27. sor]

• Interfészekkel sokszor megoldható

```
1 import java.util.*;
2
3 class Kutya {
4     private int kutyaSz;
5     Kutya(int i) {kutyaSz = i;}
6     public String toString() {
7         return "Kutya_#" + kutyaSz;
8     }
9 }
10 class Macska {
11     private int macskaSz;
12     Macska(int i) {macskaSz = i;}
13     public String toString() {
14         return "Macska_#" + macskaSz;
15     }
16 }
17 public class KutyakEsMacskak2 {
18     public static void main(String[] args) {
19         ArrayList kutyak = new ArrayList();
20         for(int i = 0; i < 3; i++)
21             kutyak.add(new Kutya(i)); // upcast Object-re!!!
22         // nem problema berakni egy macskat a kutyak koze:
23         kutyak.add(new Macska(3)); // upcast Object-re!!!
24         for(int i = 0; i < kutyak.size(); i++)
25             System.out.println(kutyak.get(i));
26     }
27 }
```

```
Kutya #0
Kutya #1
Kutya #2
Macska #3
```

Típust megőrző kollekció készítése

Holding/KutyaListaTeszt.java [1–20. sor]

```
1 import java.util.*;
2
3 class Kutya {
4     private int kutyaSz;
5     Kutya(int i) {kutyaSz = i;}
6     public String toString() {
7         return "Kutya_#" + kutyaSz;
8     }
9 }
10
11 class KutyaLista {
12     private ArrayList lista = new ArrayList();
13     public void add(Kutya k) {
14         lista.add(k);
15     }
16     public Kutya get(int index) {
17         return (Kutya)lista.get(index);
18     }
19     public int size() {return lista.size();}
20 }
```

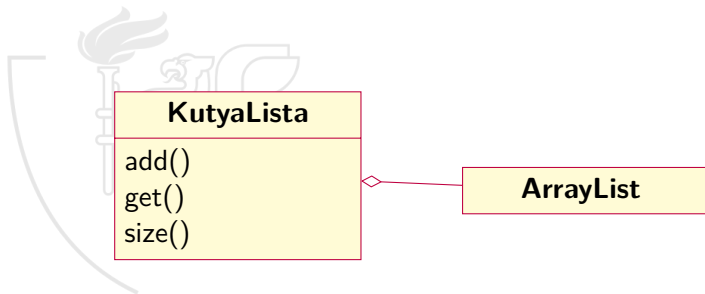
```

22 public class KutyaListaTeszt {
23     public static void main(String[] args) {
24         KutyaLista kutyak = new KutyaLista();
25         for(int i = 0; i < 3; i++)
26             kutyak.add(new Kutya(i)); // nincs upcast!
27         //kutyak.add(new Macska(3)); // forditasi hiba
28         for(int i = 0; i < kutyak.size(); i++)
29             System.out.println(kutyak.get(i)); // nem kell downcast!
30     }
31 }

```

KutyaListaTeszt.java:30:
add(Kutya) in Ku-
tyaLista cannot be
applied to (Macska)
kutyak.add(new
Macska(3))

Kutya #0
Kutya #1
Kutya #2



Generikus kollekciók

Holding/KutyakEsMacskakGen.java [1-18. sor]

```
1 import java.util.*;
2
3 class Kutya {
4     private int kutyaSz;
5     Kutya(int i) {kutyaSz = i;}
6     public void print() {
7         System.out.println(
8             "Kutya_#" + kutyaSz);
9     }
10 }
11 class Macska {
12     private int macskaSz;
13     Macska(int i) {macskaSz = i;}
14     public void print() {
15         System.out.println(
16             "Macska_#" + macskaSz);
17     }
18 }
```

Generikus kollekciók (folyt.)

Holding/KutyakEsMacskakGen.java [19–29. sor]

```
19 public class KutyakEsMacskakGen {  
20     public static void main(String[] args) {  
21         ArrayList<Kutya> kutyak = new ArrayList<Kutya>();  
22         for(int i = 0; i < 3; i++)  
23             kutyak.add(new Kutya(i)); // nincs upcast!  
24         //kutyak.add(new Macska(3)); // forditasi hiba  
25         for(int i = 0; i < kutyak.size(); i++)  
26             kutyak.get(i).print(); // nem kell downcast!  
27         //((Kutya)kutyak.get(i)).print();  
28     }  
29 }
```

KutyakEsMacskakGen.java:24: cannot find symbol
symbol : method add(Macska)

location: class java.util.ArrayList<Kutya>

kutyak.add(new Macska(3)); // forditasi hiba

^

1 error

Kutya #0

Kutya #1

Kutya #2

- Minden konténernek támogatnia kell az elemek hozzáadását, illetve elérését (ez a dolga)
- Pl. ArrayList esetében ez könnyű
 - `add()`; `get()`
- Ennél általánosabb, absztraktabb megoldás kell
 - általános kódot szeretnénk írni, amely nem függ a használt konténertől (ne kelljen újraírni)
- A megoldás az **Iterátorok** használata



Iterátorok (folyt.)

- Egy iterátor egy objektum, amely végighalad egy objektumsorozaton anélkül, hogy a felhasználó programozó tudná, hogy milyen konkrét belső struktúrája van a kollekciónak
- Java iterátor tudása
 - kérni lehet egy `Iterator` objektumot a konténertől az `iterator()` metódushívással
 - kérni lehet a következő elemet a `next()` metódussal (a legelsőöt is `next()`-tel kérjük!)
 - meg lehet kérdezni, hogy maradt-e még elem a sorozatban a `hasNext()` metódussal
 - törölni lehet az iterátorral lekért elemet `remove()`-val

Iterátorok (folyt.)

```
import java.util.*;

public class KutyakEsMacskak {
    public static void main(String[] args) {
        ArrayList kutyak = new ArrayList();
        for(int i = 0; i < 3; i++)
            kutyak.add(new Kutya(i));
        Iterator i = kutyak.iterator();
        while(i.hasNext())
            ((Kutya)(i.next())).print();
    }
}
```

Kutya #0

Kutya #1

Kutya #2

foreach

- Java 5 újítás

```
import java.util.*;

public class KutyaEsMacskak {
    public static void main(String[] args) {
        ArrayList kutyak = new ArrayList();
        for(int i = 0; i < 3; i++)
            kutyak.add(new Kutya(i));
        for(Object o : kutyak)
            ((Kutya)o).print();
    }
}
```

```
Kutya #0
Kutya #1
Kutya #2
```

foreach (folyt.)

```
import java.util.*;

public class KutyaEsMacskak {
    public static void main(String[] args) {
        ArrayList<Kutya> kutyak = new ArrayList<Kutya>();
        for(int i = 0; i < 3; i++)
            kutyak.add(new Kutya(i));
        for(Kutya k : kutyak)
            k.print();
    }
}
```

```
Kutya #0
Kutya #1
Kutya #2
```

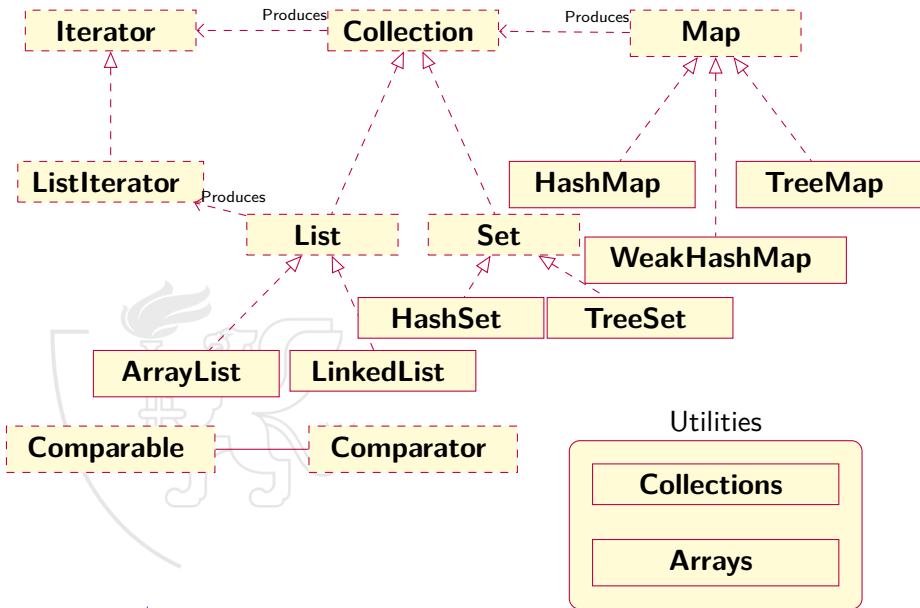
Iterátorok (folyt.)

Holding/HorcsogJarat.java [1-23. sor]

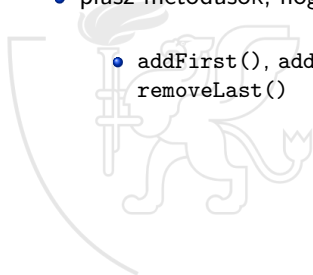
```
1 import java.util.*;
2
3 class Horcsog {
4     private int horcsogSorszam;
5     public Horcsog(int i) {horcsogSorszam = i;}
6     public String toString() {
7         return "Hello, en vagyok a(z) " + horcsogSorszam + ". horcsog";
8     }
9 }
10 class Kiiro {
11     static void irdKiMindet(Iterator i) {
12         while (i.hasNext())
13             System.out.println(i.next());
14     }
15 }
16 public class HorcsogJarat {
17     public static void main(String[] args) {
18         ArrayList v = new ArrayList();
19         for(int i = 0; i < 5; i++)
20             v.add(new Horcsog(i));
21         Kiiro.irdKiMindet(v.iterator());
22     }
23 }
```

```
Hello, en vagyok a(z) 0. horcsog
Hello, en vagyok a(z) 1. horcsog
Hello, en vagyok a(z) 2. horcsog
Hello, en vagyok a(z) 3. horcsog
Hello, en vagyok a(z) 4. horcsog
```

Osztályhierarchia



- List (interfész) – Elemeket tárol **adott sorrendben**
 - ListIterator – oda-vissza be lehet járni a listát és be lehet szúrni elemeket a listába elemek közé is
- ArrayList – Tömbbel megvalósított lista
 - gyors elérés de lassú beszúrás/törlés
- LinkedList – „Igazi” láncolt lista
 - gyors beszúrás/törlés de lassú elérés
 - plusz metódusok, hogy használható legyen stack-, queue- és deque-ként
 - `addFirst()`, `addLast()`, `getFirst()`, `getLast()`, `removeFirst()`, `removeLast()`



- Set (interfész) – **Egyedi elemeket** tárol sorrendiség nélkül
 - a tárolt elemeknek felül kell definiálni az `equals()`-t
- HashSet – **Gyors keresést** biztosító halmaz
 - a tárolt elemeknek érdemes felüldefiniálni a `hashCode()` metódust (hatékonyság miatt)
- TreeSet – **Rendezett** halmaz (fával megvalósított)
 - `first()` – a legkisebb elem
 - `last()` – a legnagyobb elem
 - `subSet(from,to)`, `headSet(to)`, `tailSet(from)`
 - visszaadják a fának egy részét

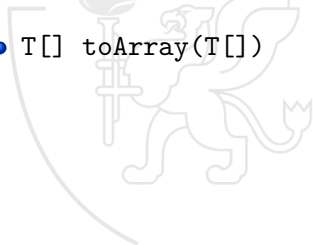


Collection (List, Set) funkcionalitása

- `boolean add(Object)` – elem hozzáadása
- `boolean addAll(Collection)` – elemek hozzáadása
- `void clear()` – töröl minden elemet
- `boolean contains(Object)` – true, ha tartalmazza az elemet
- `boolean containsAll(Collection)` – true, ha tartalmazza az elemeket
- `boolean isEmpty()` – true, ha üres
- `Iterator iterator()` – készít egy iterátort

Collection (List, Set) funkcionalitása (folyt.)

- `boolean remove(Object)` – elem törlése
- `boolean removeAll(Collection)` – elemek törlése
- `boolean retainAll(Collection)` – csak akkor tartja meg az elemet, ha az szerepel a paraméterben (metszetképzés)
- `int size()` – elemek száma
- `Object[] toArray()` – felépít egy tömböt, amely tartalmazza a kollekció elemeit
- `T[] toArray(T[])` – mint az előző, csak a paraméterben kapott tömb futásidejű típusának megfelelő típusú tömböt épít fel



- Map (leképezés) – **Kulcs-adat** objektum-párok csoportja, a kulcsra gyors keresést biztosít
- HashMap – **Hash-táblával** implementált, keresésre/beszúrára optimalizálva
- TreeMap – Piros-fekete **fával** implementált, **rendezetten tárolja** az elemeket
 - `firstKey()` – a legkisebb kulcs
 - `lastKey()` – a legnagyobb kulcs
 - `subMap(from,to)`, `headMap(from)`, `tailMap(to)`
 - visszaadják a fának egy részét

Map funkcionalitása

- `void clear()` – töröl minden elemet
- `boolean containsKey(Object)` – true, ha tartalmazza a kapott kulcsú elemet
- `boolean containsValue(Object)` – true, ha tartalmazza a kapott tartalmú elemet
- `Set entrySet()` – visszaadja halmazként a tárolt adatokat
- `Object get(Object key)` – a kapott kulcsú elemet adja vissza
- `boolean isEmpty()` – true, ha üres
- `Set keySet()` – visszaadja halmazként a tárolt kulcsokat

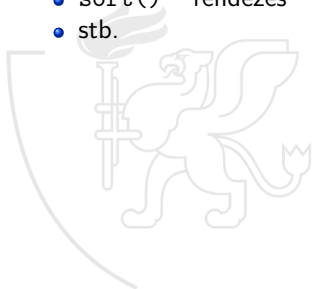
Map funkcionalitása (folyt.)

- `Object put(Object key, Object value)`
 - beszúr egy új elemet (párost)
- `void putAll(Map)`
 - beszúrja a kapott Map elemeit
- `Object remove(Object key)`
 - törli a kapott kulcsú elemet
- `int size()`
 - elemek száma
- `Collection values()`
 - visszaadja kollekcióként a tárolt elemeket



A Collections osztály

- A `java.util` könyvtár része
- Statikus metódusokat tartalmaz, amelyek segítséget nyújtanak a kollekciók kezelésében
 - `binarySearch()` – keresés
 - `copy()` – másolás
 - `fill()` – feltöltés (csere) adott értékkel
 - `reverse()` – elemsorrend megfordítás
 - `sort()` – rendezés
 - stb.




A Collections osztály (folyt.)

- Példa

```
import java.util.*;

public class ListaFeltoltes {
    public static void main(String[] args) {
        List list = new ArrayList();
        for(int i = 0; i < 5; i++)
            list.add("");
        Collections.fill(list, "Hello");
        System.out.println(list);
    }
}
```

```
[Hello, Hello, Hello, Hello, Hello]
```


- 
- 1 **Bemutakozás**
 - Kurzus információk
 - 2 **Objektumorientált tervezés**
 - A modellezés alapelvei
 - A „siker háromszöge”
 - UML
 - Objektumorientáltság alapfogalmak
 - 3 **Java alapok**
 - Bevezető
 - Az első java program
 - 4 **Java**
 - Programozási alapismeretek ismétlése
 - Objektumok, osztályok
 - Polimorfizmus
 - Annotációk
 - Objektumok kezelése
 - Hozzáférés szabályozása
 - Beágyazott osztályok
 - Újrafelhasználhatóság
 - Interfészek és enumerációk
 - Objektumok tárolása
 - 5 **Hibakezelés kivételekkel**
 - Sztringek
 - Generikus típusok
 - Típus információk
 - I/O
 - GUI; Belső, lokális és anonim osztályok, lambdák
 - 5 **Tervezési minták**
 - Minták áttekintése
 - Gyártási minták
 - Szerkezeti minták
 - Viselkedési minták

- Java alapfilozófia: Rosszul formált kód nem fog jól futni
- Ideális lenne, ha minden hibát fordítási időben ki lehetne deríteni, de a valóságban nem így van
- Korábbi nyelvekben (pl. C-ben) a hibakezelés inkább csak konvenció volt
 - tipikusan, a függvény visszaad egy hibakódot, vagy beállít egy jelzőt, amit a hívó fél kell(ene), hogy ellenőrizzen
 - ez általában elmarad: pl. ki ellenőrzi le, hogy a printf() mit ad vissza?



- A megoldás: Nyelvi szinten beépíteni és megkövetelni a hibakezelést
- Nem a Java találmánya, már az 1960-as években ismerték operációs rendszer szinten
- A Java kivételkezelése a C++-on alapul (esetleg az Object Pascal-on), a C++ kivételkezelése pedig az Ada-n
- Még egy jelentős előny: „Megtisztul” a kód, mert különválik az „igazi” kód a hibakezeléstől



- **Kivételes feltétel:** egy probléma, amely meggátolja a programfutást egy adott metódusban (vagy blokkban)
- Egyszerű példa a paraméter ellenőrzés: Érdemes leellenőrizni és kivételt „dobni” ahelyett, hogy folytatnánk

```
if (p < 0)
    throw new IllegalArgumentException();
```

- Kivétel dobás (**throw**)
 - kivétel objektum jön létre a **heap**-en
 - az aktuális programvégrehajtás megáll és a kivétel objektum referenciája „eldobódik”
 - a **kivételkezelő mechanizmus** veszi át az irányítást és a hívási veremben keres egy megfelelő **kivételkezelőt**, amely lekezeli a hibát és ott folytatja a program végrehajtását

Kivételek argumentumai

- Minden beépített kivételnek két konstruktora van: a default és ami egy String-et vár (ezt később fel lehet használni)

```
if (p < 0)
    throw new IllegalArgumentException("A_p_nem_lehet_negatív!");
```

- A kivételkezelés valójában egy alternatív „return” mechanizmus
 - a kivétel objektum van „visszaadva” (a metódus visszatérési típusa más!)
 - nem biztos, hogy csak egy szinttel tér vissza! (addig, amíg nem talál kivételt elkapó kódot)

Kivétel elkapása

- Ha valahol el lett dobva egy kivétel, akkor az feltételezi, hogy azt valahol máshol „elkapják”
- **Védett régió:** A kód olyan része, amely kivételeket hozhat létre és amelyet a hibakezelő kód követ

```
try {  
    // "normál kód", amelyben kivétel keletkezhet  
    // (védett régió)  
} catch (ExceptionType1 e1) {  
    // hibakezelő kód az e1 kivételre  
} catch (ExceptionType2 e2) {  
    // hibakezelő kód az e2 kivételre  
    throw e2; // tovább is lehet dobni  
} catch (Exception e) {  
    // hibakezelő kód az összes (megmaradt) kivételre  
} finally {  
    // végül (mindig lefut)  
}
```

Saját kivételek

Exceptions/KivetelPelda.java [1–20. sor]

- Saját kivételeket is lehet írni
- Származtatni kell valamelyik létező kivétel osztályból
 - pl. **Exception** (ős)osztályból

```
1 class SajatException extends Exception {
2     public SajatException(String s) {super(s);}
3 }
4 public class KivetelPelda {
5     public void f() throws SajatException {
6         System.out.println("Dobunk egy SajatException-t f()-bol");
7         throw new SajatException("f()-bol dobtak");
8     }
9     public static void main(String[] args) {
10         KivetelPelda kp = new KivetelPelda();
11         try {
12             // ...
13             kp.f();
14             // ...
15         } catch (SajatException e) {
16             System.err.println("Elkaptuk!");
17             System.err.println(e);
18         }
19     }
20 }
```

```
Dobunk egy SajatException-t f()-bol
Elkaptuk!
SajatException: f()-bol dobtak
```

Kivétel specifikáció

- Java-ban meg kell adni, hogy egy metódus milyen kivételeket dobhat a `throws` kulcsszóval
 - ez része a metódus-deklarációnak

```
void f() throws SimpleException, NullPointerException {  
    // ...  
}
```



- **Ősosztály:** Throwable, két leszármazottja van:
 - Error: általában nem kell vele foglalkozni (fordítási időbeni- és rendszer hibákat képvisel)
 - Exception: a Java programozó számára ez az „ősosztály”
- A kivételek leírása megtalálható a Java honlapján
 - <https://docs.oracle.com/javase/8/docs/api/java/lang/Exception.html>

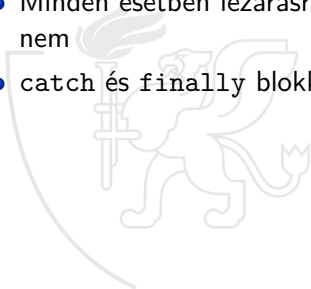


RuntimeException

- Java standard kivétel, amelyet nem kell külön megadni a kivétel specifikációban
 - Azon kivételek őse, amelyeket a virtuális gép dobhat normális működés közben, pl.:
 - NullPointerException
 - ClassCastException
 - IndexOutOfBoundsException
 - stb.
- <http://docs.oracle.com/javase/8/docs/api/java/lang/RuntimeException.html>

try-with-resources

- Lehetőség van egy-egy erőforrás létrehozására a `try` statementben közvetlen
- Tipikusan valamilyen külső erőforrás (pl. fájl) kezelésekor használatos, amelyet le kell zárni használat után
- Erőforrás lehet minden, ami implementálja a `java.lang.AutoCloseable` interfészt
- Minden esetben lezárásra kerül így az objektum, akár volt hiba, akár nem
- `catch` és `finally` blokkok csak ezután hívódnak meg



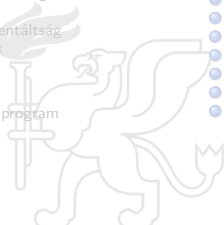
try-with-resources példa

- Java 7 előtt
 - (Fájlkezelést lásd később.)

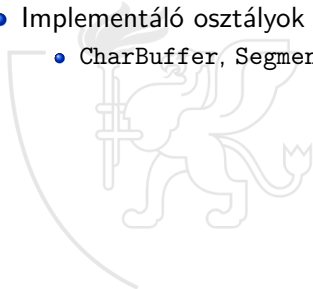
```
static String beolvas(String path) throws IOException {  
    BufferedReader br =  
        new BufferedReader(new FileReader(path));  
    try {  
        return br.readLine();  
    } finally {  
        if (br != null) br.close();  
    }  
}
```

- Java 7 után

```
static String beolvas(String path) throws IOException {  
    try (BufferedReader br =  
        new BufferedReader(new FileReader(path))) {  
        return br.readLine();  
    }  
}
```

- 
- 1 **Bemutakozás**
 - Kurzus információk
 - 2 **Objektumorientált tervezés**
 - A modellezés alapelvei
 - A „siker háromszöge”
 - UML
 - Objektumorientáltság alapfogalmak
 - 3 **Java alapok**
 - Bevezető
 - Az első java program
 - 4 **Java**
 - Programozási alapismeretek ismétlése
 - Objektumok, osztályok
 - Polimorfizmus
 - Annotációk
 - Objektumok kezelése
 - Hozzáférés szabályozása
 - Beágyazott osztályok
 - Újrafelhasználhatóság
 - Interfészek és enumerációk
 - Objektumok tárolása
 - 5 **Tervezési minták**
 - Hibakezelés kivételekkel
 - **Sztringek**
 - Generikus típusok
 - Típus információk
 - I/O
 - GUI; Belső, lokális és anonim osztályok, lambdák

- CharSequence interfész
- char értékek sorozatát jelképezi
- Funkcionalitása
 - `char charAt(int index)`
 - `int length()`
 - `CharSequence subSequence(int start, int end)`
 - `String toString()`
- Implementáló osztályok
 - `CharBuffer`, `Segment`, `String`, `StringBuffer`, `StringBuilder`



Sztringek

String/StringPelda.java [1-17. sor]

- A String osztály karakter füzéreket ábrázol
 - minden szöveg, mint pl. az "abc", ezen osztály példánya
- A String osztály objektumai konstansok
 - minden művelet, amely egy sztringet módosít, új sztringet hoz létre

```
1 public class StringPelda {
2     public static void main(String[] args) {
3         String str1 = "abc";
4         // ekvivalens az alábbival:
5         char data[] = {'a', 'b', 'c'};
6         String str2 = new String(data);
7         // további példák:
8         System.out.println("abc");
9         String cde = "cde";
10        System.out.println("abc" + cde);
11        String c = "abc".substring(2,3); // -> "c" (2)--(3-1) karakterek
12        System.out.println(c);
13        String d = cde.substring(1,2);   // -> "d" (1)--(2-1) karakterek
14        System.out.println(d);
15    }
16 }
```

```
abc
abccde
c
d
```

String

- A felüldefiniált + operátorral lehetőség van a sztringek összefűzésére és más objektumok szövegre konvertálására
 - toString() metódus hívás által, melyet minden osztály megörököl az Object-től

```
public class Concatenation {  
    public static void main(String[] args) {  
        String def = "def";  
        String s = "abc" + def + "ghi" + 47;  
        System.out.println(s);  
    }  
}
```

abcdefghi47

Sztring néhány metódusa

- `length()`
- `charAt()`
- `getChars()`
- `toCharArray()`
- `equals()`, `equalsIgnoreCase()`
- `compareTo()`, `compareToIgnoreCase()`
- `contains()`
- `regionMatches()`
- `startsWith()`, `endsWith()`
- `indexOf()`, `lastIndexOf()`
- `substring()`
- `replace()`
- `toLowerCase()`, `toUpperCase()`
- `trim()`
- `valueOf()`

Sztring formázás

- C-szerű string formázás (sokszor kényelmes)
- `String format(String format, Object... args)`
 - `format` nagyrészt ugyanaz, mint C-ben
 - `Object... args` változó számú paraméterlistát jelent



StringBuilder és StringBuffer

- Változó hosszúságú sztring
- Fő funkcionalitás:
 - append és insert, meg vannak valósítva minden adattípusra
- StringBuilder gyorsabb, de nem szál-biztos
- StringBuffer lassabb, de szál-biztos
- A metódusaik megegyeznek



Sztringek darabolása

String/Tokenizer.java [1–21. sor]

- Megadott formátumú sztring feldarabolható tokenizerrel
- Megadott elválasztóval egy tömbbe kerülnek a rész sztringek

```
1 import java.util.*;
2 public class Tokenizer {
3     public static void main(String args[]) {
4         String mydelim = " ";
5         String mystr = "JAVA Code String Tokenizer Geeks";
6
7         StringTokenizer geeks3 = new StringTokenizer(mystr, mydelim);
8         int count = geeks3.countTokens();
9         System.out.println("Number of tokens" + count);
10        for (int i = 0; i < count; i++)
11            System.out.println("token at " + i + " ]" + geeks3.nextToken());
12        while (geeks3.hasMoreTokens())
13            System.out.println(geeks3.nextToken());
14    }
15 }
```

```
Number of tokens : 5
token at [0] : JAVA
token at [1] : Code
token at [2] : String
token at [3] : Tokenizer
token at [4] : Geeks
```

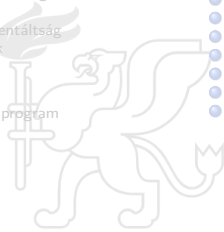
Sztringek összefűzése

String/StringJoin.java [1-12. sor]

- StringJoiner osztály Java 8-tól

```
1 import java.util.StringJoiner;
2
3 class StringJoin {
4     public static void main(String[] args) {
5         StringJoiner sj = new StringJoiner(";", "prefix-", "-suffix");
6         sj.add("Java");
7         sj.add("Code");
8         sj.add("String");
9         sj.add("Joiner");
10        System.out.println(sj);
11    }
12 }
```

prefix-Java;Code;String;Joiner-suffix

- 
- 1 **Bemutakozás**
 - Kurzus információk
 - 2 **Objektumorientált tervezés**
 - A modellezés alapelvei
 - A „siker háromszöge”
 - UML
 - Objektumorientáltság alapfogalmak
 - 3 **Java alapok**
 - Bevezető
 - Az első java program
 - 4 **Java**
 - Programozási alapismeretek ismétlése
 - Objektumok, osztályok
 - Polimorfizmus
 - Annotációk
 - Objektumok kezelése
 - Hozzáférés szabályozása
 - Beágyazott osztályok
 - Újrafelhasználhatóság
 - Interfészek és enumerációk
 - Objektumok tárolása
 - 5 **Tervezési minták**
 - Hibakezelés kivételekkel
 - Sztringek
 - **Generikus típusok**
 - Típus információk
 - I/O
 - GUI; Belső, lokális és anonim osztályok, lambdák

Generikus típusok (Generics)

- A Java 5 újdonsága (nyelvi bővítés)
- Lehetőség nyílik az osztályok paraméterezésére más típusokkal
- Nagyon hasonlít a C++ sablonokra
 - de vannak eltérések
- Csak az alapokat tekintjük át



Motiváló példa

```
import java.util.*;

class GenericsMotivacio {
    public static void main(String[] args) {
        // hagyományos
        List s1 = new LinkedList();
        s1.add(new Integer(0));
        // ...
        Integer i1 = (Integer)s1.iterator().next(); // downcast!
        System.out.println(i1);

        // generics
        List<Integer> s2 = new LinkedList<Integer>();
        s2.add(new Integer(0));
        // ...
        Integer i2 = s2.iterator().next(); // nem kell downcast!
        System.out.println(i2);
    }
}
```

0
0

- Interfész

- `interface InterfeszNev<T> {...}`

- Osztály

- `class OsztalyNev<T> {...}`

- Metódus

- `<T> T fuggvenyNev(T p);`

- Használat

- `OsztalyNev<Long> i = new OsztalyNev<Long>(new Long(1));`
 - `String s = x.fuggvenyNev("valami");`
 - Primitív típus nem használható, pl.
 - `OsztalyNev<int> s = new OsztalyNev<int>(1);`
 - fordítási hiba

Példák

Generics/GenericsPelda.java [1-29. sor]

```
1 class Generikus<E> {
2     private E x;
3     public Generikus(E x) {
4         this.x = x;
5     }
6     public E getX() {
7         return x;
8     }
9     public void setX(E x) {
10        this.x = x;
11    }
12}
13 class GenMetodus {
14     public String toString() {
15         return "GenMetodus";
16     }
17     <T> T f(T p) {return p;}
18}
19 class GenericsPelda {
20     public static void main(String[] args) {
21         //Generikus<long> a1 = new Generikus<long>(1); // fordítási hiba
22         Generikus<Long> a1 = new Generikus<Long>(new Long(1));
23         Generikus<String> a2 = new Generikus<String>("egy");
24         GenMetodus b = new GenMetodus();
25         Generikus<GenMetodus> a3 = new Generikus<GenMetodus>(b);
26         System.out.println(b.f("valami"));
27         System.out.println(b.f(9));
28     }
29}
```

- Java 5 - for-each megvalósítása generikus típusokra:

```
List<String> strings = new ArrayList<String>();  
  
//... add String instances to the strings list...  
  
for(String aString : strings){  
    System.out.println(aString);  
}
```

- Java 7 - Type inference: a fordító kikövetkezteti a példányosított kollekció típusát a hozzárendelt változó típusából:

```
List<String> strings = new ArrayList<>();
```

Saját tároló osztály bejárása

Generics/GenericsForEach.java [1-28. sor]

```
1 import java.lang.Iterable;
2 import java.util.Iterator;
3
4 class MyStack<E> implements Iterable<E> {
5     class Link {
6         E item;
7         Link next;
8         Link(E item, Link next) {this.item = item; this.next = next;}
9     }
10    Link top;
11
12    void push(E item) {
13        top = new Link(item, top);
14    }
15    public Iterator<E> iterator() {return new MyIterator<E>(top);}
16 }
17
18 class MyIterator<E> implements Iterator<E> {
19     MyStack<E>.Link cur;
20
21     MyIterator(MyStack<E>.Link top) {cur = top;}
22     public boolean hasNext() {return cur != null;}
23     public E next() {
24         E item = cur.item;
25         cur = cur.next;
26         return item;
27     }
28 }
```

Saját tároló osztály bejárása

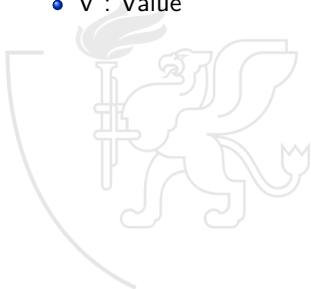
Generics/GenericsForEach.java [30–40. sor]

```
30 public class GenericsForEach {  
31     public static void main(String[] args) {  
32         MyStack<Double> collection = new MyStack<>();  
33         collection.push(1.5);  
34         collection.push(2.5);  
35         collection.push(3.5);  
36         for(Double d : collection) {  
37             System.out.println(d);  
38         }  
39     }  
40 }
```

3.5
2.5
1.5

Típus paraméterek elnevezési konvenciói

- Általában egyetlen nagybetű, így egyértelműen megkülönböztethető egyéb osztály vagy interfész nevétől
- Leggyakoribb paraméter elnevezések:
 - E : Element (tárolók használatánál)
 - K : Key
 - N : Number
 - T : Type
 - V : Value



Raw type-ok

- A raw type-ot a generikus osztály, vagy interfész neve alkotja a típus argumentumok nélkül

```
public class Box<T> {  
    public void set(T t) { /* ... */ }  
    // ...  
}  
Box<String> stringBox = new Box<>(); // paraméterezett típus  
Box rawBox = new Box();              // raw type
```

- Java 5 előtti kód (főleg konténerek) kompatibilis maradjon

- Néha szükség lehet, hogy a típus paraméterre valamilyen megszorítást tegyünk

- Felső korlát:

```
public class NaturalNumber<T extends Integer>
```

- Wildcard-ok, ismeretlen típusok

- upper bounded wildcard:

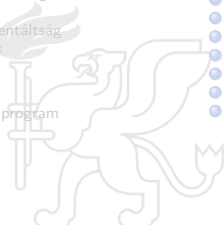
```
public void process(List<? extends Foo> list)
```

minden olyan listára, ami vagy a Foo, vagy annak leszármazottaiból áll

- lower bounded wildcard:

```
public void addNumbers(List<? super Foo> list)
```

minden olyan listára, ami vagy a Foo, vagy annak őseiből áll

- 
- 1 **Bemutakozás**
 - Kurzus információk
 - 2 **Objektumorientált tervezés**
 - A modellezés alapelvei
 - A „siker háromszöge”
 - UML
 - Objektumorientáltság alapfogalmak
 - 3 **Java alapok**
 - Bevezető
 - Az első java program
 - 4 **Java**
 - Programozási alapismeretek ismétlése
 - Objektumok, osztályok
 - Polimorfizmus
 - Annotációk
 - Objektumok kezelése
 - Hozzáférés szabályozása
 - Beágyazott osztályok
 - Újrafelhasználhatóság
 - Interfészek és enumerációk
 - Objektumok tárolása
 - 5 **Tervezési minták**
 - Hibakezelés kivételekkel
 - Sztringek
 - Generikus típusok
 - **Típus információk**
 - I/O
 - GUI; Belső, lokális és anonim osztályok, lambdák

Futás közbeni típusazonosítás

- Run-time Type Identification (RTTI)
- Futás közben megállapítható, hogy egy (pl. őssztály típusú) referencia milyen konkrét típusú objektumra hivatkozik
- Java-ban a „hagyományos” RTTI-t bővítették egy ún. „**reflection**” mechanizmussal
 - egy a fordító számára ismeretlen osztály felépítése is lekérhető futás időben
 - reflection annak a képessége, hogy a program futás közben lekérdezze (esetleg módosítsa) adott objektumot, típust, stb.



Példa

RTTI/AlakzatPelda.java [1–25. sor]

```
1 import java.util.*;
2
3 class Alakzat {
4     public void rajzolj() {
5         System.out.println(this + ".rajzolj");
6     }
7 }
8
9 class Haromszog extends Alakzat {
10     public String toString() {
11         return "Haromszog";
12     }
13 }
14
15 class Negyzet extends Alakzat {
16     public String toString() {
17         return "Negyzet";
18     }
19 }
20
21 class Kor extends Alakzat {
22     public String toString() {
23         return "Kor";
24     }
25 }
```

Példa (folyt.)

RTTI/AlakzatPelda.java [27-37. sor]

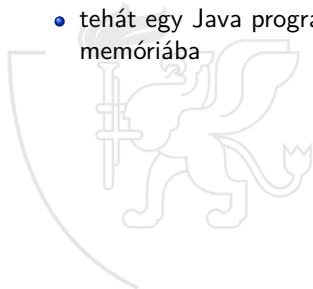
- RTTI implicit használata downcast (típuskonverzió gyerekre) esetén
- Hiba esetén kivétel dobódik

```
27 public class AlakzatPelda {  
28     public static void main(String[] args) {  
29         List s = new ArrayList();  
30         s.add(new Kor());  
31         s.add(new Haromszog());  
32         s.add(new Negyzet());  
33         Iterator i = s.iterator();  
34         while(i.hasNext())  
35             ((Alakzat)i.next()).rajzolj(); // RTTI használata impliciten  
36     }  
37 }
```

```
Kor.rajzolj  
Haromszog.rajzolj  
Negyzet.rajzolj
```

A Class objektum

- Speciális objektum, amely tartalmazza a „szokásos” objektumok létrehozásához szükséges információkat
- Minden osztályhoz a programban létezik egy Class típusú objektum (valójában ez az objektum van eltárolva a .class fájlban)
- Futásidőben, amikor szükség van egy objektumra, akkor a JVM betölti a megfelelő .class fájlból a Class objektumot (ha még nincs betöltve)
 - tehát egy Java program általában nem egyszerre töltődik be a memóriába



A Class objektum elérése

- Osztálynév segítségével
 - `Class.forName("OsztalyNev")`
- Class literál segítségével
 - `OsztalyNev.class`
- Object őssztály `getClass()` metódusa segítségével
 - `x.getClass()`, ahol x egy objektum



Példa

RTTI/AlakzatPelda2.java [1-28. sor]

```
1 class Alakzat {
2     static {
3         System.out.println("Alakzat_betoltese");
4     }
5     public void rajzolj() {
6         System.out.println(this + ".rajzolj");
7     }
8 }
9 class Haromszog extends Alakzat {
10     static {
11         System.out.println("Haromszog_betoltese");
12     }
13     public String toString() {
14         return "Haromszog";
15     }
16 }
17 public class AlakzatPelda2 {
18     public static void main(String[] args) {
19         new Alakzat();
20         try {
21             Class c = Class.forName("Haromszog");
22         } catch (ClassNotFoundException e) {
23             e.printStackTrace(System.err);
24         }
25     }
26 }
```

Szokásos betöltés

Név alapján betölti a megfelelő
Class objektumot futás közben

Alakzat betoltese
Haromszog betoltese

Class literálok

RTTI/AlakzatPelda3.java [19–23. sor]

- Minden osztálynak van egy statikus adattagja `class` névvel, amely az ő `Class` objektumára hivatkozik
- Példa

```
19 public class AlakzatPelda3 {  
20     public static void main(String[] args) {  
21         new Alakzat();  
22         Class c = Haromszog.class;  
23     }
```

```
Alakzat betöltése  
Haromszog betöltése
```


Objektumtípus ellenőrzése

RTTI/AlakzatPelda4.java [1–25. sor]

- Statikus: instanceof kulcsszó
- Ellenőrizhető futásidőben, hogy adott típusú-e egy objektum

```
1 import java.util.*;
2
3 class Alakzat {
4     public void rajzolj() {
5         System.out.println(this + ".rajzolj");
6     }
7 }
8
9 class Haromszog extends Alakzat {
10     public String toString() {
11         return "Haromszog";
12     }
13 }
14
15 class Negyzet extends Alakzat {
16     public String toString() {
17         return "Negyzet";
18     }
19 }
20
21 class Kor extends Alakzat {
22     public String toString() {
23         return "Kor";
24     }
25 }
```

Objektumtípus ellenőrzése (folyt.)

RTTI/AlakzatPelda4.java [27–40. sor]

```
27 public class AlakzatPelda4 {  
28     public static void main(String[] args) {  
29         List s = new ArrayList();  
30         s.add(new Kor());  
31         s.add(new Haromszog());  
32         s.add(new Negyzet());  
33         Iterator i = s.iterator();  
34         while(i.hasNext()) {  
35             Object o = i.next();  
36             if (o instanceof Haromszog)  
37                 ((Haromszog)o).rajzolj();  
38         }  
39     }  
40 }
```

Haromszog.rajzolj

Objektumtípus ellenőrzése (folyt.)

RTTI/AlakzatPelda5.java [27-47. sor]

- Dinamikus: `isInstance()` Class metódus

```
27 public class AlakzatPelda5 {
28     public static void main(String[] args) {
29         List s = new ArrayList();
30         s.add(new Kor());
31         s.add(new Haromszog());
32         s.add(new Negyzet());
33         Iterator i = s.iterator();
34         while(i.hasNext()) {
35             Object o = i.next();
36             if (Haromszog.class.isInstance(o))
37                 ((Haromszog)o).rajzolj();
38             // vagy:
39             try {
40                 if (Class.forName("Haromszog").isInstance(o))
41                     ((Haromszog)o).rajzolj();
42             } catch (ClassNotFoundException e) {
43                 e.printStackTrace(System.err);
44             }
45         }
46     }
47 }
```

Haromszog.rajzolj
Haromszog.rajzolj

Egyéb Class metódusok

- getName
- isInterface
- getMethods
- getConstructors
- getSuperclass
- getPackage
- stb.



Példa reflection-re: metóduslistázó


RTTI/KutyaPelda.java [1-30. sor]

```
1 import java.lang.reflect.*;
2
3 class Kutya {
4     private int kutyaSz;
5     Kutya(int i) {kutyaSz = i;}
6     public String toString() {
7         return "Kutya_#" + kutyaSz;
8     }
9     public int sorszam() {
10        return kutyaSz;
11    }
12}
13 public class KutyaPelda {
14     public static void main(String[] args) {
15         if (args.length != 1) System.exit(0);
16         try {
17             Class c = Class.forName(args[0]);
18             System.out.println(c.getName());
19             System.out.println(c.isInterface());
20             Method[] m = c.getMethods();
21             Constructor[] ctor = c.getConstructors();
22             for (int i = 0; i < m.length; i++)
23                 System.out.println(m[i]);
24             for (int i = 0; i < ctor.length; i++)
25                 System.out.println(ctor[i]);
26         } catch (ClassNotFoundException e) {
27             System.err.println(e);
28         }
29     }
30 }
```

Példa reflection-re: metóduslistázó (folyt.)

```
>java KutyaPelda Kutya
Kutya
false
public java.lang.String Kutya.toString()
public int Kutya.sorszam()
public final void java.lang.Object.wait(long,int) throws java.lang.InterruptedException
public final void java.lang.Object.wait() throws java.lang.InterruptedException
public final native void java.lang.Object.wait(long) throws java.lang.InterruptedException
public boolean java.lang.Object.equals(java.lang.Object)
public native int java.lang.Object.hashCode()
public final native java.lang.Class java.lang.Object.getClass()
public final native void java.lang.Object.notify()
public final native void java.lang.Object.notifyAll()
```

```
>java KutyaPelda KutyaPelda
KutyaPelda
false
public static void KutyaPelda.main(java.lang.String[])
public final void java.lang.Object.wait(long,int) throws java.lang.InterruptedException
public final void java.lang.Object.wait() throws java.lang.InterruptedException
public final native void java.lang.Object.wait(long) throws java.lang.InterruptedException
public boolean java.lang.Object.equals(java.lang.Object)
public java.lang.String java.lang.Object.toString()
public native int java.lang.Object.hashCode()
public final native java.lang.Class java.lang.Object.getClass()
public final native void java.lang.Object.notify()
public final native void java.lang.Object.notifyAll()
public KutyaPelda()
```

- 
- 1 **Bemutakozás**
 - Kurzus információk
 - 2 **Objektumorientált tervezés**
 - A modellezés alapelvei
 - A „siker háromszöge”
 - UML
 - Objektumorientáltság alapfogalmak
 - 3 **Java alapok**
 - Bevezető
 - Az első java program
 - 4 **Java**
 - Programozási alapismeretek ismétlése
 - Objektumok, osztályok
 - Polimorfizmus
 - Annotációk
 - Objektumok kezelése
 - Hozzáférés szabályozása
 - Beágyazott osztályok
 - Újrafelhasználhatóság
 - Interfészek és enumerációk
 - Objektumok tárolása
 - 5 **Tervezési minták**
 - Hibakezelés kivételekkel
 - Sztringek
 - Generikus típusok
 - Típus információk
 - **I/O**
 - GUI; Belső, lokális és anonim osztályok, lambdák

A Java I/O rendszer

- Adatfolyam alapú megközelítés
- Java 1.0: InputStream/OutputStream osztályok
 - bájt-orientált I/O
- Java 1.1: Reader/Writer osztályok
 - karakter-orientált I/O **unicode** támogatással
 - de bővíti az InputStream/OutputStream osztályhierarchiát is



A File osztály

IO/DirList.java [1-27. sor]

- Egy fájlt vagy könyvtárat képvisel

```
1 import java.io.*;
2 import java.util.*;
3
4 class DirFilter implements FilenameFilter {
5     String s;
6     DirFilter(String s) {this.s = s;}
7     // callback függvény:
8     public boolean accept(File dir, String name) {
9         // path torlese:
10         String f = new File(name).getName();
11         // resz-string?
12         return f.indexOf(s) != -1;
13     }
14 }
15
16 public class DirList {
17     public static void main(String[] args) {
18         File path = new File(".");
19         String[] list;
20         if (args.length == 0)
21             list = path.list();
22         else
23             list = path.list(new DirFilter(args[0]));
24         for(int i = 0; i < list.length; i++)
25             System.out.println(list[i]);
26     }
27 }
```

A File osztály (folyt.)

```
>java DirList  
DirFilter.class  
DirList.class  
DirList.java
```

```
>java DirList .class  
DirFilter.class  
DirList.class
```



A File osztály (folyt.)

IO/MakeDirectories.java [1–22. sor]

- Sok metódusa van. Pl.:

```
1 import java.io.*;
2
3 public class MakeDirectories {
4     public static void main(String[] args) {
5         if (args.length < 1) System.exit(1);
6         File f = new File(args[0]);
7         System.out.println(
8             "Abszolút útv.: \t" + f.getAbsolutePath() +
9             "\nNév: \t\t" + f.getName() +
10            "\nHossz: \t\t" + f.length() +
11            "\nModosítva: \t" + f.lastModified());
12         if (f.isFile()) System.out.println("fajl");
13         else if (f.isDirectory()) System.out.println("könyvt.");
14         if (f.exists()) {
15             System.out.println(f + " létezik, töröljük");
16             f.delete();
17         } else {
18             System.out.println("letrehozás... \t" + f);
19             f.mkdirs();
20         }
21     }
22 }
```

A File osztály (folyt.)

```
>java MakeDirectories proba.java
Abszolút utv.: D:\prog1\proba.java
Nev:          proba.java
Hossz:        667
Módositva:    1080123982328
fajl
proba.java letezik, toroljuk
```

```
>java MakeDirectories proba
Abszolút utv.: D:\prog1\proba
Nev:          proba
Hossz:        0
Módositva:    0
letrehozás... proba
```

```
>java MakeDirectories proba
Abszolút utv.: D:\prog1\proba
Nev:          proba
Hossz:        0
Módositva:    1080124219296
konyvt.
proba letezik, toroljuk
```

- Adatfolyam (stream)
 - objektum, amely valamilyen adatforrást vagy adatnyelőt jelképez
 - elrejti a fizikai közeget (pl. fájl, string)
- Két típus
 - `InputStream` – adatforrás folyam
 - `OutputStream` – adatnyelő folyam



InputStream típusai (adatforrások)

- `ByteArrayInputStream`
 - bájtok tömbje
- `StringBufferInputStream`
 - string objektum
- `FileInputStream`
 - fájl
- `PipedInputStream`
 - cső (pipe)
- `SequenceInputStream`
 - más adatforrás folyamatok sorozata (összegyűjtése)
- `FilterInputStream`
 - adatfolyam „dekoráló” osztályok őse

OutputStream típusai (adatnyelők)

- `ByteArrayOutputStream`
 - bájtok tömbje
- `FileOutputStream`
 - fájl
- `PipedOutputStream`
 - cső (pipe)
- `FilterOutputStream`
 - adatfolyam „dekoráló” osztályok őse
- Nincs string kimenet

FilterInputStream

- InputStream-ből származik
 - **Decorator** tervezési minta
- DataInputStream
 - primitív adattípusok olvasása (String is), pl. `readByte()`, `readFloat()`, stb.
- BufferedInputStream
 - pufferelt adatfolyam
- LineNumberInputStream
 - figyeli a sorszámot (`getLineNumber()`)
- PushbackInputStream
 - a legutóbb olvasott karaktert vissza lehet tenni az adatfolyamba

FilterOutputStream

- OutputStream-ből származik
 - **Decorator** tervezési minta
- DataOutputStream
 - primitív adattípusok írása (String is), pl. `writeByte()`, `writeFloat()`, stb.
- BufferedOutputStream
 - puffertelt adatfolyam (`flush()`)
- PrintStream
 - formázott kimenet (`print()`, `println()`)



Reader/Writer osztályok

- A legtöbb Java adatfolyam osztálynak (Java 1.0) megvan a megfelelő Reader/Writer párja (Java 1.1)
 - **unicode** támogatás
- Konvertáló osztályok, pl.
 - `InputStreamReader`, `OutputStreamWriter`



- InputStream
 - Reader
- OutputStream
 - Writer
- FileInputStream
 - FileReader
- FileOutputStream
 - FileWriter
- StringBufferInputStream
 - StringReader
- ByteArrayInputStream
 - CharArrayReader
- ByteArrayOutputStream
 - CharArrayWriter
- PipedInputStream
 - PipedReader
- PipedOutputStream
 - PipedWriter
- Az interfészek nagyon hasonlóak

Megfeleltetés (dekoráló)

- `FilterInputStream`
 - `FilterOutputStream`
 - `BufferedInputStream`
 - `BufferedOutputStream`
 - `DataInputStream`
 - `PrintStream`
 - `LineNumberInputStream`
 - `PushBackInputStream`
 - Az interfészek nagyon hasonlóak
- `FilterReader`
 - `FilterWriter`
 - `BufferedReader`
 - `BufferedWriter`
 - nincs
 - `PrintWriter`
 - `LineNumberReader`
 - `PushBackReader`

A RandomAccessFile osztály

- Nem része az adatfolyam öröklődési hierarchiáknak
- A fájlban tud előre/hátra mozogni, keresni
 - `getFilePointer()`
 - `seek()`
 - `length()`



Példák

IO/IOStreamDemo.java [1-14. sor]

```
1 import java.io.*;
2
3 public class IOStreamDemo {
4     public static void main(String[] args) throws IOException {
5         System.out.println("--_fajlbeolvasas_soronkent_--");
6         BufferedReader brf =
7             new BufferedReader(
8                 new FileReader("src/IOStreamDemo.java"));
9         String s;
10        StringBuilder f = new StringBuilder();
11        while((s = brf.readLine()) != null) {
12            f.append(s).append("\n");
13        }
14        brf.close();
```

```
-- fajlbeolvasas soronkent --
```

Példák (folyt.)

IO/IOStreamDemo.java [16–21. sor]

```
16 System.out.println("--_input_memoriabol_--");
17 StringReader sr = new StringReader(f.toString());
18 int c;
19 while ((c = sr.read()) != -1) {
20     System.out.print((char)c);
21 }
```

```
-- input memoriabol --
import java.io.*;

public class IOStreamDemo {
    public static void main(String[] args) throws IOException {
        System.out.println("-- fajlbeolvasas soronkent --");
        BufferedReader brf =
            new BufferedReader(
                new FileReader("src/IOStreamDemo.java"));
        String s;
        StringBuilder f = new StringBuilder();
        while((s = brf.readLine()) != null) {
            f.append(s).append("\n");
        }
        brf.close();

        System.out.println("-- input memoriabol --");
        StringReader sr = new StringReader(f.toString());
        int c;
        while ((c = sr.read()) != -1) {
            System.out.print((char)c);
        }
    }
}
```

Példák (folyt.)

IO/IOStreamDemo.java [23–33. sor]

```
23 System.out.println("--formattalt input memoriabol--");
24 try {
25     DataInputStream dis =
26         new DataInputStream(
27             new ByteArrayInputStream(f.toString().getBytes()));
28     while (true) {
29         System.out.print((char)dis.readByte());
30     }
31 } catch (EOFException e) {
32     System.err.println("Adatfolyam vege");
33 }
```

```
-- formattalt input memoriabol --
import java.io.*;

public class IOStreamDemo {
    public static void main(String[] args) throws IOException {
        System.out.println("-- fajlbeolvasas soronkent --");
        BufferedReader brf =
            new BufferedReader(
                new FileReader("src/IOStreamDemo.java"));
        String s;
        StringBuilder f = new StringBuilder();
        while((s = brf.readLine())!= null) {
            f.append(s).append("\n");
        }
        brf.close();
        ....
        Adatfolyam vege
    }
}
```


Példák (folyt.)

IO/IOStreamDemo.java [35–47. sor]

```
35 System.out.println("--fajl output--");
36 BufferedReader brs =
37     new BufferedReader(
38         new StringReader(f.toString()));
39 PrintWriter pw =
40     new PrintWriter(
41         new BufferedWriter(
42             new FileWriter("IOStreamDemo.out")));
43 int lineCount = 1;
44 while ((s = brs.readLine()) != null) {
45     pw.println(lineCount++ + ": " + s);
46 }
47 pw.close();
```

```
-- fajl output --
```

- IOStreamDemo.out-ba bekerül a forrás állomány úgy, hogy a sorok sorszámozva vannak

Példák (folyt.)

IO/IOStreamDemo.java [49–63. sor]

```
49 System.out.println("-- random access fajl irasa/olvasasa --");
50 RandomAccessFile raf =
51     new RandomAccessFile("IOStreamDemo.dat", "rw");
52 for(int i = 0; i < 5; i++) {
53     raf.writeDouble(i*1.414);
54 }
55 raf.seek(3*8);
56 raf.writeDouble(47.0001);
57 raf.seek(0);
58 for(int i = 0; i < 5; i++) {
59     System.out.println(i + ".ertekek:" + raf.readDouble());
60 }
61 raf.close();
62 }
63 }
```

```
-- random access fajl irasa/olvasasa --
0. ertekek: 0.0
1. ertekek: 1.414
2. ertekek: 2.828
3. ertekek: 47.0001
4. ertekek: 5.656
```

- IOStreamDemo.dat tartalma:

.....?özl'vČ'9@.zl'vČ'9@G?.FÜ]d@.zl'vČ'9

- Unix fogalom
 - **standard input:** a program bemenete (`System.in`)
 - **standard output:** a program kimenete (`System.out`)
 - **standard error:** a program hiba kimenete (`System.err`)
- Programokat egymáshoz lehet láncolni
 - a program kimenete egy másik program bemenete lehet közvetlenül



Standard I/O (folyt.)

- A `System.out` és `System.err` `PrintStream` típusú objektumok, ezért közvetlenül használhatóak
 - már sok példa volt rá...
- A `System.in` pedig `InputStream` típusú, ezért „dekorálni” kell



Példa standard input-ra

IO/Echo.java [1–14. sor]

```
1 import java.io.*;
2
3 public class Echo {
4     public static void main(String[] args) throws IOException {
5         BufferedReader in =
6             new BufferedReader(
7                 new InputStreamReader(System.in));
8         String s;
9         // ures sorra kilep
10        while ((s = in.readLine()).length() != 0) {
11            System.out.println(s);
12        }
13    }
14 }
```

```
haliho
haliho
hehe
hehe
papagaj
papagaj
```

Scanner osztály

IO/MyScanner.java [1–17. sor]

- Text olvasó, ami a primitív típusokat és sztringeket tudja felismerni reguláris kifejezésekkel
- Az inputot darabokra szedi (alapértelmezett: szóköz elválasztó)
- Az eredmény tokenek a megfelelő fajta next hívással konvertálhatóak különböző típusú

```
1 import java.util.Scanner;
2 import java.io.File;
3
4 public class MyScanner {
5     public static void main(String args[]) {
6         try {
7             Scanner sc = new Scanner(new File(args[0]));
8             while (sc.hasNextLong()) {
9                 long number = sc.nextLong();
10                System.out.println("Long: " + number);
11            }
12            if (sc.hasNext()) {
13                System.out.println("Next: " + sc.next());
14            }
15        } catch (Exception e) {}
16    }
17 }
```

Formatált kiírás

IO/Printf.java [1-17. sor]

- C-szerű lehetőség, sokszor hasznos és kényelmes
- `System.out.format()` és `System.out.printf()` metódusok

```
1 import java.util.*;
2
3 public class Printf {
4     public static void main(String[] args) {
5         double pi = Math.PI;
6
7         System.out.format("%f%n", pi);           // --> "3,141593"
8         System.out.format("%.3f%n", pi);         // --> "3,142"
9         System.out.printf("%10.3f%n", pi);        // --> "          3,142"
10        System.out.printf(Locale.US, "%10.4f%n", pi); // --> "          3.1416"
11
12        Calendar c = Calendar.getInstance();
13        System.out.format("%tB%te,%tY%n", c, c, c); // --> "március 19, 2018"
14        System.out.format("%tI:%tM%tp%n", c, c, c); // --> "12:16 du."
15        System.out.format("%tD%n", c);           // --> "03/19/18"
16    }
17 }
```

Objektumok szerializálása, mentése

IO/Student.java [1–13. sor]

- A szerializáció az a folyamat, amikor az objektumot bájtok sorozatával ábrázoljuk
- Állapotmentés, állapot átadás. Elmenthető.
- Magas szintű streamek: `ObjectInputStream` és `ObjectOutputStream`
- Szerializálhatósághoz implementálandó: `java.io.Serializable`
- Nem szerializálható mezőt `transient` jelzővel kell ellátni

```
1 public class Student implements java.io.Serializable {  
2     private String name;  
3     private String address;  
4     private transient int age;  
5     public Student(String name, String address, int age) {  
6         this.name = name;  
7         this.address = address;  
8         this.age = age;  
9     }  
10    public String toString() {  
11        return "Student_" + name + "_" + address + "_" + age;  
12    }  
13 }
```


Objektumok szerializálása, mentése (folyt.)

IO/SerializeDemo.java [1-17. sor]

```
1 import java.io.*;
2
3 public class SerializeDemo {
4     public static void main(String [] args) {
5         Student e = new Student("Kelemen_Ede", "Szeged", 25);
6         try {
7             ObjectOutputStream out =
8                 new ObjectOutputStream(
9                     new FileOutputStream("student.ser"));
10            out.writeObject(e);
11            out.close();
12            System.out.printf("Szerializalva ide: student.ser");
13        } catch (IOException i) {
14            i.printStackTrace();
15        }
16    }
17 }
```

Szerializalva ide: student.ser


Objektumok szerializálása, mentése (folyt.)

IO/DeserializeDemo.java [1-20. sor]

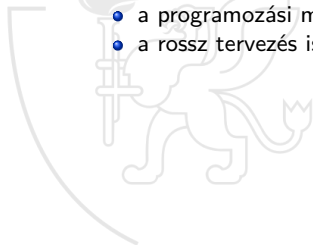
```
1 import java.io.*;
2
3 public class DeserializeDemo {
4     public static void main(String [] args) {
5         Student e = null;
6         try {
7             ObjectInputStream in =
8                 new ObjectInputStream(
9                     new FileInputStream("student.ser"));
10            e = (Student)in.readObject();
11            in.close();
12            System.out.println("Deszerializalt Student... " + e);
13        } catch (IOException i) {
14            i.printStackTrace();
15        } catch (ClassNotFoundException c) {
16            System.out.println("Student class not found");
17            c.printStackTrace();
18        }
19    }
20 }
```

Deszerializalt Student...Student Kelemen Ede Szeged 0

Tartalom

- 
- 1 **Bemutakozás**
 - Kurzus információk
 - 2 **Objektumorientált tervezés**
 - A modellezés alapelvei
 - A „siker háromszöge”
 - UML
 - Objektumorientáltság alapfogalmak
 - 3 **Java alapok**
 - Bevezető
 - Az első java program
 - 4 **Java**
 - Programozási alapismeretek ismétlése
 - Objektumok, osztályok
 - Polimorfizmus
 - Annotációk
 - Objektumok kezelése
 - Hozzáférés szabályozása
 - Beágyazott osztályok
 - Újrafelhasználhatóság
 - Interfészek és enumerációk
 - Objektumok tárolása
 - 5 **Tervezési minták**
 - Hibakezelés kivételekkel
 - Sztringek
 - Generikus típusok
 - Típus információk
 - I/O
 - **GUI; Belső, lokális és anonim osztályok, lambdák**

- Eredeti célkitűzés (Java 1.0)
 - olyan GUI könyvtár készítése, amely egyformán jól néz ki minden platformon
- Eredmény
 - **AWT** – Abstract Window Toolkit
 - olyan GUI könyvtár, amely egyformán rosszul néz ki minden platformon
 - ráadásul korlátozott
 - csak 4 font érhető el
 - nem érhetők el a speciálisabb GUI elemek
 - a programozási modellje is esetlen és nem objektum orientált
 - a rossz tervezés iskolapéldája (állítólag 1 hónap alatt készült el)



- Java 1.1
 - továbbfejlesztett AWT
 - objektum orientált programozási modell bevezetése
 - még mindig nem az igazi
- Java 2
 - **JFC** – Java Foundation Classes
 - **Swing** – a JFC GUI-val foglalkozó része
 - szépen megtervezett könyvtár
 - kár, hogy továbbra is függ az AWT-től
- Java 8
 - JavaFX – JDK/JRE része lett
 - 2008 óta fejlesztik
 - Java 11-től kezdve eltávolították, külön fejlődik

- Swing GUI komponensek
 - minden a nyomógomboktól a táblázatokig
- Külalak plug-in-ek (look-and-feel)
 - paramétrezhető külalak lehetősége
 - pl. az aktuális operációs rendszer ablakozójához igazodhat, de lehet akár saját egyedi is
- Elérhetőség API
 - fogyatékos felhasználók támogatása

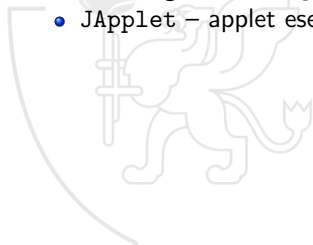


JFC komponensek (folyt.)

- Java 2D API
 - 2D grafika, szöveg és képek kezelése
 - nyomtatás támogatása
- „Húzd és ejtsd” támogatása (Drag-and-drop)
 - adatok mozgatása „húzd és ejtsd” módon Java és natív alkalmazások között
- Internacionalizálás
 - különleges karakterek támogatása, pl. japán, kínai, koreai



- Szükséges csomagok (általában)
 - `javax.swing.*`
 - `javax.swing.event.*`
 - `java.awt.*`
 - `java.awt.event.*`
- Minden Swing-es programnak kell, hogy legyen legalább egy legfelső szintű tárolója ezek közül:
 - `JFrame` – fő ablak
 - `JDialog` – másodlagos ablak (függ más ablaktól)
 - `JApplet` – applet esetén a böngészőn belüli terület



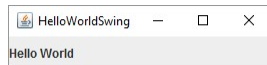
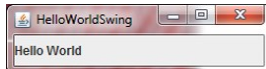
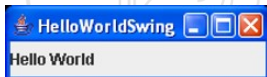
- A legfelső szintű tárolóknak van egy ún. tároló mezője
 - content pane: `java.awt.Container` típusú
 - összefogja tárolt komponenseket
 - kivéve a menüket
- A **Swing**-es komponensek őse
 - `JComponent`
 - kivéve a legfelső szintű tárolókat



Swing-es „Hello World”

Gui/HelloWorldSwing.java [1-13. sor]

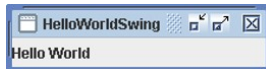
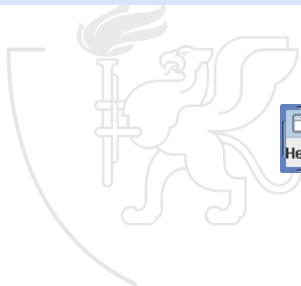
```
1 import javax.swing.*;
2
3 public class HelloWorldSwing {
4     public static void main(String[] args) {
5         JFrame frame = new JFrame("HelloWorldSwing");
6         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
7         JLabel label = new JLabel("Hello World");
8         frame.add(label);
9         //frame.getContentPane().add(label); // korábban
10        frame.pack();
11        frame.setVisible(true);
12    }
13 }
```



Swing-es „Hello World” #2

Gui/HelloWorldSwing2.java [1-13. sor]

```
1 import javax.swing.*;
2
3 public class HelloWorldSwing2 {
4     public static void main(String[] args) {
5         JFrame.setDefaultLookAndFeelDecorated(true);
6         JFrame frame = new JFrame("HelloWorldSwing");
7         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
8         JLabel label = new JLabel("Hello World");
9         frame.add(label);
10        frame.pack();
11        frame.setVisible(true);
12    }
13 }
```

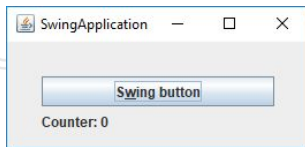


- A komponensek tudnak generálni eseményeket
 - „tüzelés”
 - minden egyes esemény külön típussal van ábrázolva
- Egy esemény egy vagy több „hallgatózóhoz” (listener) érkezik meg, amelyek reagál(hat)nak rá
 - olyan osztály, amely implementál valamilyen **listener** interfészt
- A komponens objektumban regisztrálni kell a listener-t
 - `komp.addXYZListener(listenerObj);`
 - `XYZ` == az esemény típusa
 - `Action` – tipikusan klikk v. Enter a komponensen
 - `Window` – ablak bezárása
 - `MouseMotion` – egér mozdul a komponens felett
 - `Component` – láthatóvá válik a komponens
 - `Focus` – a komponens megkapja a fókuszt
 - stb.

Eseménykezelés példa

Gui/SwingApplication.java [1-17. sor]

```
1 import javax.swing.*;
2 import java.awt.*;
3 import java.awt.event.*;
4
5 class OnButtonClick implements ActionListener {
6     private String prefix;
7     private int clicks;
8     private JLabel label;
9     public OnButtonClick(String prefix, int clicks, JLabel label) {
10         this.prefix=prefix;
11         this.clicks=clicks;
12         this.label=label;
13     }
14     public void actionPerformed(ActionEvent e) {
15         label.setText(prefix + clicks++);
16     }
17 }
```



Nyomógomb és eseménykezelés

Gui/SwingApplication.java [19–40. sor]

```
19 public class SwingApplication {
20     private static JFrame frame = new JFrame("SwingApplication");
21     private JPanel panel = new JPanel(new GridLayout(0,1));
22     private JButton button = new JButton("SwingButton");
23     private String prefix = "Counter: ";
24     private int clicks = 1;
25     private JLabel label = new JLabel(prefix + "0");
26
27     public SwingApplication() {
28         button.setMnemonic(KeyEvent.VK_W);
29         button.addActionListener(new OnButtonClick(prefix, clicks, label));
30         panel.add(button);
31         panel.add(label);
32         panel.setBorder(BorderFactory.createEmptyBorder(30,30,10,30));
33         frame.add(panel, BorderLayout.CENTER);
34     }
35     public static void main(String[] args) {
36         SwingApplication app = new SwingApplication();
37         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
38         frame.pack();
39         frame.setVisible(true);
40     }
```

- Szorosan és kizárólag az adott osztályhoz tartozó osztály forráskód szintű tartalmazása
 - Nem kompozíció!
- A belső osztály látja és használhatja a külső osztály tagjait
- Statikus belső osztály a külső osztály statikus tagjait látja és használhatja
- **.class** fájlnev: **Külső\$Belső.class**



Belső osztályok

Gui/SwingApplication2.java [5–32. sor]

```
5 public class SwingApplication2 {
6     class OnButtonClick implements ActionListener {
7         public void actionPerformed(ActionEvent e) {
8             label.setText(prefix + clicks++);
9         }
10    }
11    private static JFrame frame = new JFrame("SwingApplication");
12    private JPanel panel = new JPanel(new GridLayout(0,1));
13    private JButton button = new JButton("SwingButton");
14    private String prefix = "Counter: ";
15    private int clicks = 1;
16    private JLabel label = new JLabel(prefix + "0");
17
18    public SwingApplication2() {
19        button.setMnemonic(KeyEvent.VK_W);
20        button.addActionListener(new OnButtonClick());
21        panel.add(button);
22        panel.add(label);
23        panel.setBorder(BorderFactory.createEmptyBorder(30,30,10,30));
24        frame.add(panel, BorderLayout.CENTER);
25    }
26    public static void main(String[] args) {
27        SwingApplication2 app = new SwingApplication2();
28        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
29        frame.pack();
30        frame.setVisible(true);
31    }
32 }
```


Lokális osztályok

- Szorosan és kizárólag az adott blokkhoz (általában metódushoz) tartozó osztály forráskód szintű tartalmazása
- A lokális osztály látja és használhatja
 - a metódus osztályának tagjait
 - a metódus paramétereit és (lényegében) konstans változóit (amelyek sohasem változnak)



Lokális osztályok

Gui/SwingApplication3.java [5–32. sor]

```
5 public class SwingApplication3 {
6     private static JFrame frame = new JFrame("SwingApplication");
7     private JPanel panel = new JPanel(new GridLayout(0,1));
8     private JButton button = new JButton("SwingButton");
9     private String prefix = "Counter: ";
10    private int clicks = 1;
11    private JLabel label = new JLabel(prefix + "0");
12
13    public SwingApplication3() {
14        class OnButtonClick implements ActionListener {
15            public void actionPerformed(ActionEvent e) {
16                label.setText(prefix + clicks++);
17            }
18        }
19        button.setMnemonic(KeyEvent.VK_W);
20        button.addActionListener(new OnButtonClick());
21        panel.add(button);
22        panel.add(label);
23        panel.setBorder(BorderFactory.createEmptyBorder(30,30,10,30));
24        frame.add(panel, BorderLayout.CENTER);
25    }
26    public static void main(String[] args) {
27        SwingApplication3 app = new SwingApplication3();
28        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
29        frame.pack();
30        frame.setVisible(true);
31    }
32 }
```

Anonim osztályok

- Név nélküli lokális osztályok
- Lehetőséget ad arra, hogy egyszerre definiáljunk és példányosítsunk is egy osztályt
- Akkor érdemes használni, ha csak egyszer használnánk egy lokális osztályt
- Szintaxis
 - new operátor
 - Interfész, amit megvalósít vagy osztály, amit bővít
 - Konstruktor argumentumlista
 - Osztálytörzs

Anonim osztályok

Gui/SwingApplication4.java [5–31. sor]

```
5 public class SwingApplication4 {
6     private static JFrame frame = new JFrame("SwingApplication");
7     private JPanel panel = new JPanel(new GridLayout(0,1));
8     private JButton button = new JButton("SwingButton");
9     private String prefix = "Counter: ";
10    private int clicks = 1;
11    private JLabel label = new JLabel(prefix + "0");
12
13    public SwingApplication4() {
14        button.setMnemonic(KeyEvent.VK_W);
15        button.addActionListener(new ActionListener() {
16            public void actionPerformed(ActionEvent e) {
17                label.setText(prefix + clicks++);
18            }
19        });
20        panel.add(button);
21        panel.add(label);
22        panel.setBorder(BorderFactory.createEmptyBorder(30,30,10,30));
23        frame.add(panel, BorderLayout.CENTER);
24    }
25    public static void main(String[] args) {
26        SwingApplication4 app = new SwingApplication4();
27        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
28        frame.pack();
29        frame.setVisible(true);
30    }
31 }
```

- Egy darab metódussal rendelkező anonim osztályok tömörebb írásmódja
- Ilyen esetekben elvben „funktionalitást” adunk át argumentumként és nem objektumot
- „anonim metódus”
- Java 8 nyelvi újítása
- Szintaxis
 - Vesszővel elválasztott paraméterlista zárójelek között
 - Típusok elhagyhatók, egy paraméter esetében a zárójel is
 - -> token
 - Törzs: egyetlen kifejezés vagy utasítás blokk

Lambda kifejezések

Gui/SwingApplication5.java [5–29. sor]

```
5 public class SwingApplication5 {
6     private static JFrame frame = new JFrame("SwingApplication");
7     private JPanel panel = new JPanel(new GridLayout(0,1));
8     private JButton button = new JButton("SwingButton");
9     private String prefix = "Counter: ";
10    private int clicks = 1;
11    private JLabel label = new JLabel(prefix + "0");
12
13    public SwingApplication5() {
14        button.setMnemonic(KeyEvent.VK_W);
15        button.addActionListener((ActionEvent e) -> {
16            label.setText(prefix + clicks++);
17        });
18        panel.add(button);
19        panel.add(label);
20        panel.setBorder(BorderFactory.createEmptyBorder(30,30,10,30));
21        frame.add(panel, BorderLayout.CENTER);
22    }
23    public static void main(String[] args) {
24        SwingApplication5 app = new SwingApplication5();
25        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
26        frame.pack();
27        frame.setVisible(true);
28    }
29 }
```

Lambda kifejezések

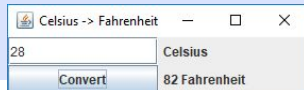
Gui/SwingApplication6.java [5–27. sor]

```
5 public class SwingApplication6 {
6     private static JFrame frame = new JFrame("SwingApplication");
7     private JPanel panel = new JPanel(new GridLayout(0,1));
8     private JButton button = new JButton("SwingButton");
9     private String prefix = "Counter: ";
10    private int clicks = 1;
11    private JLabel label = new JLabel(prefix + "0");
12
13    public SwingApplication6() {
14        button.setMnemonic(KeyEvent.VK_W);
15        button.addActionListener(e -> label.setText(prefix + clicks++));
16        panel.add(button);
17        panel.add(label);
18        panel.setBorder(BorderFactory.createEmptyBorder(30,30,10,30));
19        frame.add(panel, BorderLayout.CENTER);
20    }
21    public static void main(String[] args) {
22        SwingApplication6 app = new SwingApplication6();
23        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
24        frame.pack();
25        frame.setVisible(true);
26    }
27 }
```

Szövegmező

Gui/CelsiusConverter.java [5-32. sor]

```
5 public class CelsiusConverter {
6     JFrame frame = new JFrame("Celsius->Fahrenheit");
7     JPanel panel = new JPanel(new GridLayout(2,2));
8     JTextField celsiusText = new JTextField();
9     JButton convertButton = new JButton("Convert");
10    JLabel celsiusLabel = new JLabel("Celsius");
11    JLabel fahrenheitLabel = new JLabel("Fahrenheit");
12
13    public CelsiusConverter() {
14        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
15        convertButton.addActionListener(e -> {
16            int tempFahr = (int)((Double.parseDouble(celsiusText.getText()))*1.8+32);
17            fahrenheitLabel.setText(tempFahr + "Fahrenheit");
18        });
19        panel.add(celsiusText);
20        panel.add(celsiusLabel);
21        panel.add(convertButton);
22        panel.add(fahrenheitLabel);
23        celsiusLabel.setBorder(BorderFactory.createEmptyBorder(5,5,5,5));
24        fahrenheitLabel.setBorder(BorderFactory.createEmptyBorder(5,5,5,5));
25        frame.add(panel, BorderLayout.CENTER);
26        frame.pack();
27        frame.setVisible(true);
28    }
29    public static void main(String[] args) {
30        new CelsiusConverter();
31    }
32 }
```



Rádiógombok

Gui/RadioAndDialogDemo.java [1–20. sor]

```
1 import javax.swing.*;
2 import java.awt.*;
3 import java.awt.event.*;
4
5 public class RadioAndDialogDemo {
6     static JDialog dialog = new JDialog();
7     static JPanel panel = new JPanel(new BorderLayout());
8     ButtonGroup group = new ButtonGroup();
9     final String defaultCommand = "def";
10    final String yesNoCommand = "yn";
11    final String yesNoCancelCommand = "ync";
12
13    public static void main(String[] args) {
14        dialog.setTitle("RadioDialog");
15        dialog.setDefaultCloseOperation(JDialog.DISPOSE_ON_CLOSE);
16        new RadioAndDialogDemo();
17        dialog.add(panel);
18        dialog.pack();
19        dialog.setVisible(true);
20    }
```

Rádiógombok

Gui/RadioButtonDemo.java [22-48. sor]

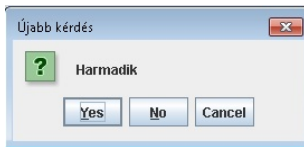
```
22 public RadioButtonDemo() {
23     panel.add(createRadioButtonsPanel(), BorderLayout.NORTH);
24     JButton button = new JButton("Valassz");
25     panel.add(button, BorderLayout.CENTER);
26     JLabel label = new JLabel("Valassz!", JLabel.CENTER);
27     button.addActionListener(e -> {
28         String command = group.getSelection().getActionCommand();
29         if (command == defaultCommand) {
30             JOptionPane.showMessageDialog(dialog, "Elso");
31             label.setText("OK");
32         } else if (command == yesNoCommand) {
33             int n = JOptionPane.showConfirmDialog(dialog, "Masodik",
34                 "Ujabb_kerdes", JOptionPane.YES_NO_OPTION);
35             if (n == JOptionPane.YES_OPTION) label.setText("Yes");
36             else if (n == JOptionPane.NO_OPTION) label.setText("No");
37         } else if (command == yesNoCancelCommand) {
38             int n = JOptionPane.showConfirmDialog(dialog, "Harmadik",
39                 "Ujabb_kerdes", JOptionPane.YES_NO_CANCEL_OPTION);
40             if (n == JOptionPane.YES_OPTION) label.setText("Yes");
41             else if (n == JOptionPane.NO_OPTION) label.setText("No");
42             else label.setText("Cancel");
43         }
44     });
45     label.setBorder(BorderFactory.createEmptyBorder(10,10,10,10));
46     panel.add(label, BorderLayout.SOUTH);
47     panel.setBorder(BorderFactory.createEmptyBorder(20,20,5,20));
48 }
```



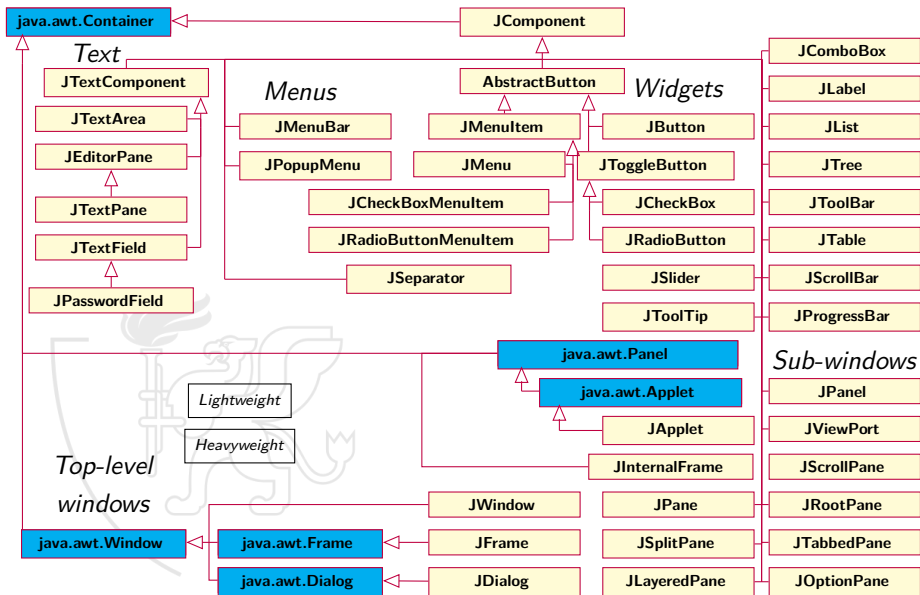
Rádiógombok

Gui/RadioAndDialogDemo.java [50–68. sor]

```
50 private JPanel createRadioButtonsPanel() {
51     JRadioButton[] radioButtons = new JRadioButton[3];
52     radioButtons[0] = new JRadioButton("ez az első");
53     radioButtons[0].setActionCommand(defaultCommand);
54     radioButtons[1] = new JRadioButton("ez a második");
55     radioButtons[1].setActionCommand(yesNoCommand);
56     radioButtons[2] = new JRadioButton("ez pedig a harmadik");
57     radioButtons[2].setActionCommand(yesNoCancelCommand);
58     for (int i = 0; i < radioButtons.length; i++)
59         group.add(radioButtons[i]);
60     radioButtons[0].setSelected(true);
61     JPanel panel = new JPanel();
62     panel.setLayout(new BoxLayout(panel, BoxLayout.PAGE_AXIS));
63     panel.add(new JLabel("A lista"));
64     for (int i = 0; i < radioButtons.length; i++)
65         panel.add(radioButtons[i]);
66     return panel;
67 }
68 }
```



További komponensek



További komponensek (folyt.)

- <http://docs.oracle.com/javase/tutorial/uiswing/components/>



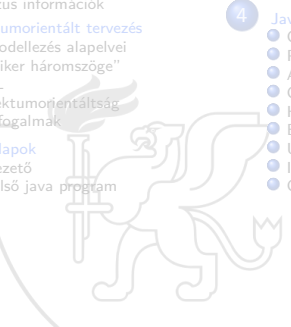
Programozás I.

Dr. Ferenc Rudolf
Dr. Jász Judit

Szegedi Tudományegyetem
Informatikai Intézet
Szoftverfejlesztés Tanszék

2021-02-08



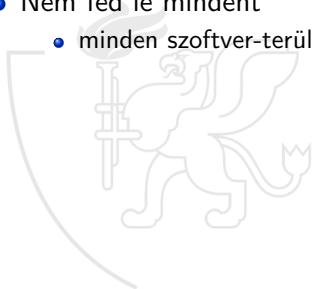
- 
- 1 **Bemutakozás**
 - Kurzus információk
 - 2 **Objektumorientált tervezés**
 - A modellezés alapelvei
 - A „siker háromszöge”
 - UML
 - Objektumorientáltság alapfogalmak
 - 3 **Java alapok**
 - Bevezető
 - Az első java program
 - 4 **Java**
 - Programozási alapismeretek ismétlése
 - Objektumok, osztályok
 - Polimorfizmus
 - Annotációk
 - Objektumok kezelése
 - Hozzáférés szabályozása
 - Beágyazott osztályok
 - Újrafelhasználhatóság
 - Interfészek és enumerációk
 - Objektumok tárolása
 - 5 **Tervezési minták**
 - Hibakezelés kivételekkel
 - Sztringek
 - Generikus típusok
 - Típus információk
 - I/O
 - GUI; Belső, lokális és anonim osztályok, lambdák

- Minták áttekintése
- Gyártási minták
- Szerkezeti minták
- Viselkedési minták

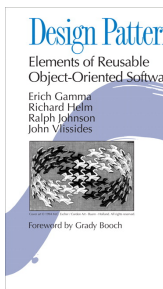
- Tervezési minta általában
Sokszor felmerülő probléma leírása + a megoldás magja, amelyet újra és újra alkalmazhatunk
- Egymással kommunikáló objektumok és osztályok leírása egy általános tervezési probléma megoldására bizonyos kontextuson belül
- Ami a *kódolásnak* az osztály-könyvtár, az a *tervezésnek* a tervezési minta gyűjtemény
- Nem OOP esetén is lehetnek speciális minták

Tervezési minták (folyt.)

- A legtöbb jó OO architektúra használ mintákat
- Részben ez határozza meg a szoftver minőségét
- A rendszer egyszerűbb, érthetőbb, karbantarthatóbb és újrafelhasználhatóbb lesz
- Nem új találmány
 - a programozói „folklór” része
- Nem fed le mindent
 - minden szoftver-területnek lehetnek további saját mintái

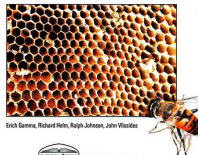


- Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides
 - **Design Patterns**
Elements of Reusable Object-Oriented Software
 - **Programtervezési minták**
Újrahasznosítható elemek objektumközpontú programokhoz
- Egyéb könyvek is vannak, de ez az egyik legfontosabb
- <http://www.oodeesign.com/>



Programtervezési minták

Újrahasznosítható elemek
objektumközpontú programokhoz



KISKAPU

Addison-Wesley

- A minta neve
 - fontosabb mint gondolnánk
 - kommunikáció eszköze
 - beszédes név kell, hogy legyen
- A probléma
 - mikor alkalmazandó a minta
 - a kontextust is definiálja




- A megoldás
 - a minta elemeinek absztrakt leírása, azok kapcsolataival, feladataival, közreműködésekkel
 - nem ír(hat)ja le a konkrét tervet, implementációt
- Következmények
 - sokszor figyelmen kívül hagyják, pedig fontos
 - minta alkalmazásának előnyei, hátrányos hatása méretre, időre (az újrafelhasználhatóság általában győz)



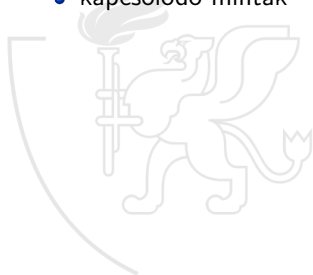
Mi nem tervezési minta?

- Nézőpont kérdése
 - a programozó képességeitől is függ
- Alapvető építőelemek nem azok (algoritmusok, absztrakt adattípusok)
 - pl. láncolt lista implementációja
- Teljes szoftverrendszer váz-szerkezete sem
 - Framework-alapú tervezés



- 
- 1 **Bemutakozás**
 - Kurzus információk
 - 2 **Objektumorientált tervezés**
 - A modellezés alapelvei
 - A „siker háromszöge”
 - UML
 - Objektumorientáltság alapfogalmak
 - 3 **Java alapok**
 - Bevezető
 - Az első java program
 - 4 **Java**
 - Programozási alapismeretek ismétlése
 - Objektumok, osztályok
 - Polimorfizmus
 - Annotációk
 - Objektumok kezelése
 - Hozzáférés szabályozása
 - Beágyazott osztályok
 - Újrafelhasználhatóság
 - Interfészek és enumerációk
 - Objektumok tárolása
 - 5 **Tervezési minták**
 - **Minták áttekintése**
 - Gyártási minták
 - Szerkezeti minták
 - Viselkedési minták
 - Hibakezelés kivételekkel
 - Sztringek
 - Generikus típusok
 - Típus információk
 - I/O
 - GUI; Belső, lokális és anonim osztályok, lambdák

- Gamma könyv szerint
- UML osztály- és objektum-diagramok hasznosak, de nem elegendők
- További elemek a leírásoknál
 - motiváció
 - alkalmazhatóság
 - implementációs kérdések
 - ismert alkalmazások
 - kapcsolódó minták



A katalógus szerkezete

- Rendeltetés szerinti csoportosítás
 - Gyártási (Creational)
 - objektumok létrehozása
 - Szerkezeti (Structural)
 - osztályok vagy objektumok összetétele, szerkezete
 - Viselkedési (Behavioral)
 - osztályok vagy objektumok kölcsönhatása és felelősség megosztása
- Hatáskör szerinti csoportosítás
 - Osztályok (Class)
 - osztályok közötti statikus viszonyok
 - Objektumok (Object)
 - objektumok közötti dinamikus (futás közbeni) viszonyok

A katalógus mintái – áttekintés

Scope	Class	Purpose		
		Creational	Structural	Behavioral
	Class	Factory Method (107)	Adapter (class) (139)	Interpreter (243) TemplateMethod (325)
	Object	AbstractFactory(87) Builder (97) Prototype (117) Singleton (127)	Adapter(object)(139) Bridge(151) Composite (163) Decorator (175) Facade (185) Flyweight (195) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Observer (293) State (305) Strategy (315) Visitor (331)

- **Factory Method** (Virtual Constructor)
 - interfész definiálása egy objektum létrehozásához, de csak a származtatott osztályok döntenek el a tényleges osztályt
- **Abstract Factory** (Kit)
 - kapcsolódó vagy függő objektumcsaládok létrehozása a konkrét osztály megnevezése nélkül
- **Builder**
 - összetett objektum létrehozásának elkülönítése a tényleges reprezentációtól
- **Prototype**
 - objektumok létrehozása egy prototípus példány másolásával
- **Singleton**
 - biztosítja, hogy egy osztályból csak egy objektum keletkezzen amely globálisan elérhető

A szerkezeti minták áttekintése

- **Adapter** (Wrapper)

- osztály újrafelhasználása új, kompatibilis interfésszel

- **Bridge** (Handle, Body)

- absztrakció és implementáció közötti csatolás lazítása, hogy a kettő egymástól függetlenül legyen változtatható

- **Composite**

- rész-egész szerkezetek leképezése objektum-hierarchiára rész és egész azonos kezelésével



A szerkezeti minták (folyt.)

- **Decorator** (Wrapper)

- további felelősségek/tulajdonságok dinamikus csatolása az objektumhoz

- **Facade**

- magasabb szintű egyesített interfész egy nagyobb rendszer több különböző interfészéhez

- **Flyweight**

- nagy számú apró, megosztott objektum hatékony kezelése

- **Proxy** (Surrogate)

- szurrogátum (helyettes) egy másik objektum számára a hozzáférés közben tartása végett

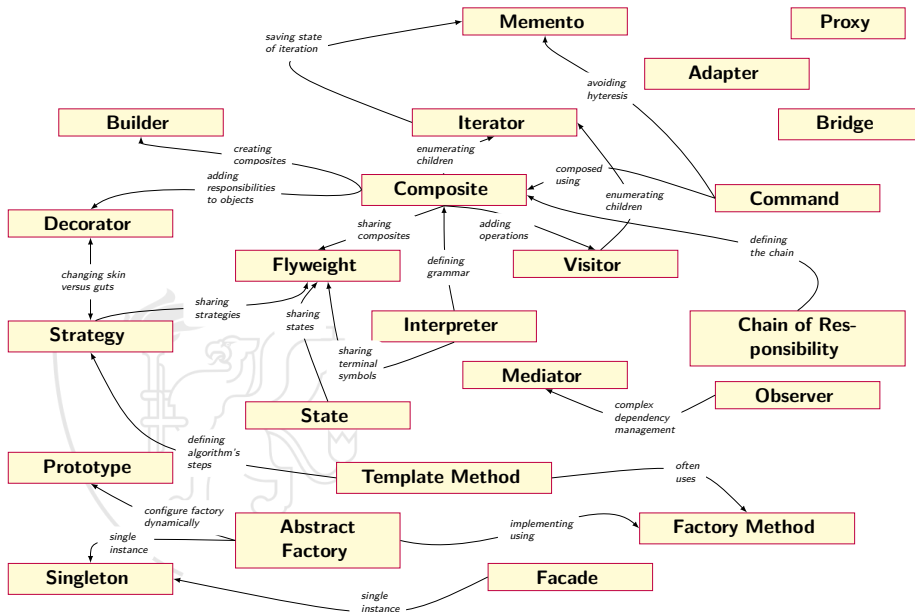
A viselkedési minták áttekintése


- **Interpreter**
 - egy nyelv nyelvtanának reprezentálása a hozzá tartozó interpreterrel
- **Template Method**
 - egy algoritmus vázának definiálása, amelyben bizonyos műveleteket csak a származtatott osztályok definiálnak
- **Chain of Responsibility**
 - a kérés/parancs küldője és végrehajtója közötti közvetlen csatolás megszüntetése
- **Command (Action, Transaction)**
 - kérések objektumként való ábrázolása azok sorba állíthatósága, naplózhatósága végett
- **Iterator (Cursor)**
 - tároló-objektum elemeinek sorozatos elérése a reprezentációtól függetlenül
- **Mediator**
 - objektumok együttműködési módját egységbezáró objektum közvetlen csatolás megszüntetésével

A viselkedési minták (folyt.)

- Memento (Token)
 - objektum belső állapotáról pillanatfelvétel készítése a későbbi visszaállítás végett
- Observer (Dependents, Publish-Subscribe)
 - egy objektum állapotának megváltozásáról a tőle függő objektumok értesítése
- State (Objects for States)
 - objektum viselkedésének megváltoztatása belső állapotváltozás hatására (osztály-váltás)
- Strategy (Policy)
 - több, azonos feladatú, de különböző algoritmus egységbezárásával azok felcserélhetővé tétele
- Visitor
 - objektum-hierarchián végezhető művelet reprezentálása a kérdéses osztályok megváltoztatása nélkül

Minták közötti kapcsolatok



- 
- 1 **Bemutakozás**
 - Kurzus információk
 - 2 **Objektumorientált tervezés**
 - A modellezés alapelvei
 - A „siker háromszöge”
 - UML
 - Objektumorientáltság alapfogalmak
 - 3 **Java alapok**
 - Bevezető
 - Az első java program
 - 4 **Java**
 - Programozási alapismeretek ismétlése
 - Objektumok, osztályok
 - Polimorfizmus
 - Annotációk
 - Objektumok kezelése
 - Hozzáférés szabályozása
 - Beágyazott osztályok
 - Újrafelhasználhatóság
 - Interfészek és enumerációk
 - Objektumok tárolása
 - 5 **Tervezési minták**
 - Hibakezelés kivételekkel
 - Sztringek
 - Generikus típusok
 - Típus információk
 - I/O
 - GUI; Belső, lokális és anonim osztályok, lambdák
 - 6 **Gyártási minták**
 - Minták áttekintése
 - Gyártási minták
 - Szerkezeti minták
 - Viselkedési minták

- Példányosítás folyamatában segítenek
 - maga a folyamat elkülönüljön a példányosítás módjától
 - mi jön létre, ki hozza létre, hogyan jön létre, és mikor
- Melyiket használjuk?
 - Factory Method általánosan használható
 - Abstract Factory flexibilisebb, de nagyobb osztályhierarchiához vezet
 - Prototype kisebb alkalmazásoknál jó
 - Singleton egyértelmű, hogy hol kell



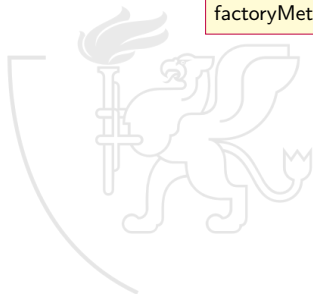
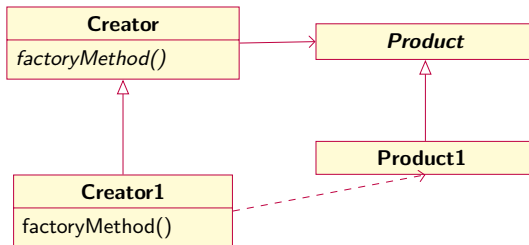
Factory Method

- Objektum létrehozása úgy, hogy a klienst nem terheljük a létrehozás logikájával
- Az ősosztály definiálja az összes szokásos és általános viselkedést, de magát a létrehozási részleteket átadja az alosztályoknak



Factory Method (folyt.)

- Szerkezet



Factory Method példa

- Kismeretű kollekciók létrehozása körülményes az adatok egyesével való hozzáadásával

```
import java.util.ArrayList;
import java.util.List;
public class FactoryMethodsExample {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("Java");
        list.add("JavaFX");
        list.add("Spring");
        list.add("Hibernate");
        list.add("JSP");
        for(String l : list){
            System.out.println(l);
        }
    }
}
```

Java
JavaFX
Spring
Hibernate
JSP

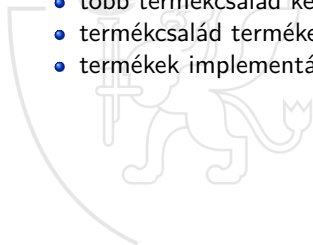
Factory Method példa (folyt.)

- Java 9 kibővíti a különböző kollekciók interfészét az of metódussal
- Nem módosítható tárolók az eredmények

```
import java.util.List;
public class FactoryMethodsExample {
    public static void main(String[] args) {
        List<String> list
            = List.of("Java", "JavaFX", "Spring", "Hibernate", "JSP");
        for (String l : list) {
            System.out.println(l);
        }
    }
}
```

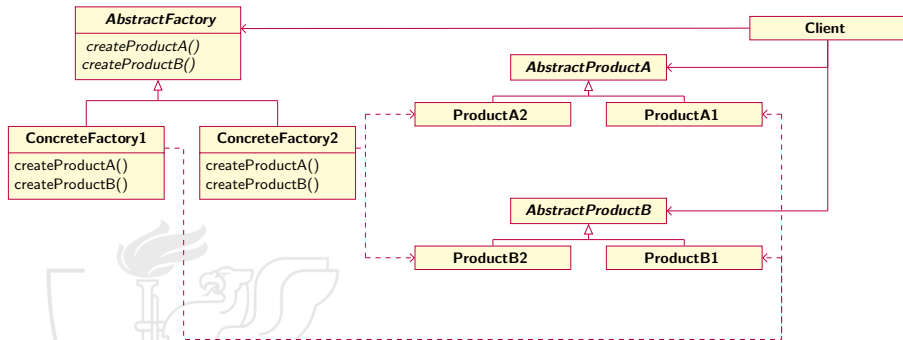
Abstract Factory

- Cél
 - kapcsolódó vagy függő objektumcsaládok létrehozása a konkrét osztály megnevezése nélkül
- Motiváció
 - pl. osztálykönyvtár felhasználói felület kezeléshez (különböző külalakok különböző UI elemeket definiálnak)
- Alkalmazhatóság
 - a rendszer független a termékek szerkezetétől, gyártásától
 - több termékcsalád kell, hogy legyen
 - termékcsalád termékeit együtt kell használni
 - termékek implementációja rejtett, interfész adott



Abstract Factory (folyt.)

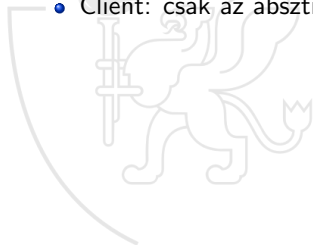
- Szerkezet



Abstract Factory (folyt.)

- Résztvevők

- AbstractFactory: interfészt deklarál kreáló operációkhoz (pl. WidgetFactory)
- ConcreteFactory: kreáló operációkat implementál konkrét termékcsaládhoz (pl. MotifWidgetFactory)
- AbstractProduct: interfészt deklarál egy termékhez (pl. Window, ScrollBar)
- Product: hozzátartozó konkrét gyárhoz implementál egy konkrét terméket (pl. MotifWindow)
- Client: csak az absztrakt interfészeket használja



Abstract Factory (folyt.)

- Következmények
 - konkrét osztályok elszigetelése (a kliens absztrakt interfészen keresztül kommunikál)
 - a termékek konzisztenciája biztosítva van
 - új terméktípusok bevezetése bonyolult
- Ismert alkalmazások
 - felhasználói felület könyvtár kül. look-and-feel elemekkel
 - platform független felhasználói felület
- Kapcsolódó minták
 - AbstractFactory megvalósítások sokszor a Factory Method mintával
 - ConcreteFactory tipikusan Singleton

Abstract Factory példa

```
public interface Shape {  
    void draw();  
}
```

```
public class Rectangle implements Shape {  
    @Override  
    public void draw() {  
        System.out.println("Inside Rectangle::draw() method.");  
    }  
}
```

```
public class RoundedRectangle implements Shape {  
    @Override  
    public void draw() {  
        System.out.println("Inside RoundedRectangle::draw() method.");  
    }  
}
```

Abstract Factory példa (folyt.)

```
public class Square implements Shape {  
    @Override  
    public void draw() {  
        System.out.println("Inside Square::draw() method.");  
    }  
}
```

```
public class RoundedSquare implements Shape {  
    @Override  
    public void draw() {  
        System.out.println("Inside RoundedSquare::draw() method.");  
    }  
}
```

```
public abstract class AbstractFactory {  
    abstract Shape getShape(String shapeType);  
}
```

Abstract Factory példa (folyt.)

```
public class ShapeFactory extends AbstractFactory {
    @Override
    public Shape getShape(String shapeType) {
        if (shapeType.equalsIgnoreCase("RECTANGLE")) {
            return new Rectangle();
        } else if (shapeType.equalsIgnoreCase("SQUARE")) {
            return new Square();
        }
        return null;
    }
}
```

```
public class RoundedShapeFactory extends AbstractFactory {
    @Override
    public Shape getShape(String shapeType) {
        if (shapeType.equalsIgnoreCase("RECTANGLE")) {
            return new RoundedRectangle();
        } else if (shapeType.equalsIgnoreCase("SQUARE")) {
            return new RoundedSquare();
        }
        return null;
    }
}
```

Abstract Factory példa (folyt.)

```
public class FactoryProducer {  
    public static AbstractFactory getFactory(boolean rounded) {  
        if (rounded)  
            return new RoundedShapeFactory();  
        else  
            return new ShapeFactory();  
    }  
}
```

```
public class AbstractFactoryPatternDemo {  
    public static void main(String[] args) {  
        AbstractFactory shapeFactory=FactoryProducer.getFactory(false);  
        Shape shape1 = shapeFactory.getShape("RECTANGLE");  
        shape1.draw();  
        Shape shape2 = shapeFactory.getShape("SQUARE");  
        shape2.draw();  
        AbstractFactory shapeFactory1=FactoryProducer.getFactory(true);  
        Shape shape3 = shapeFactory1.getShape("RECTANGLE");  
        shape3.draw();  
        Shape shape4 = shapeFactory1.getShape("SQUARE");  
        shape4.draw();  
    }  
}
```

Abstract Factory példa (folyt.)

```
Inside Rectangle::draw() method.  
Inside Square::draw() method.  
Inside RoundedRectangle::draw() method.  
Inside RoundedSquare::draw() method.
```

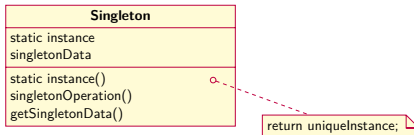


- Cél
 - biztosítja, hogy egy osztályból csak egy objektum keletkezzen, amely globálisan elérhető
- Motiváció
 - pl. egy operációs rendszerben csak egy fájlrendszer kezelő lehet (maga az osztály tartja számon a saját példányát)
- Alkalmazhatóság
 - pontosan egy példány létezhet amelyeket a kliensek elérhetnek
 - az egyedüli példány bővíthető kell, hogy legyen (származtatással)



Singleton (folyt.)

- Szerkezet és implementáció



```
class Singleton {  
    public static Singleton instance() {  
        if (instance == null)  
            instance = new Singleton();  
        return instance;  
    }  
    protected Singleton() {}  
    // ...  
    private static Singleton instance = null;  
    // ...  
}
```


Singleton (folyt.)

- Résztvevők


- Singleton: definiálja az instance operációt, amely elérhetővé teszi a klienseknek az egyedüli példányt. Az egyetlen példányt is ő készíti

- Következmények

- az egyedüli példány ellenőrzött elérése
- globális változók helyett jobb alternatíva
- a Singleton specializálható is származtatással
- könnyedén módosítható hogy fix számú példánya legyen

- Ismert alkalmazások

- alkalmazás inicializáló adatainak tárolása
- operációs rendszer ablakkezelője

- 
- 1 **Bemutakozás**
 - Kurzus információk
 - 2 **Objektumorientált tervezés**
 - A modellezés alapelvei
 - A „siker háromszöge”
 - UML
 - Objektumorientáltság alapfogalmak
 - 3 **Java alapok**
 - Bevezető
 - Az első java program
 - 4 **Java**
 - Programozási alapismeretek ismétlése
 - Objektumok, osztályok
 - Polimorfizmus
 - Annotációk
 - Objektumok kezelése
 - Hozzáférés szabályozása
 - Beágyazott osztályok
 - Újrafelhasználhatóság
 - Interfészek és enumerációk
 - Objektumok tárolása
 - 5 **Tervezési minták**
 - Hibakezelés kivételekkel
 - Sztringek
 - Generikus típusok
 - Típus információk
 - I/O
 - GUI; Belső, lokális és anonim osztályok, lambdák
- 5 **Szerkezeti minták**
 - Minták áttekintése
 - Gyártási minták
 - **Szerkezeti minták**
 - Viselkedési minták

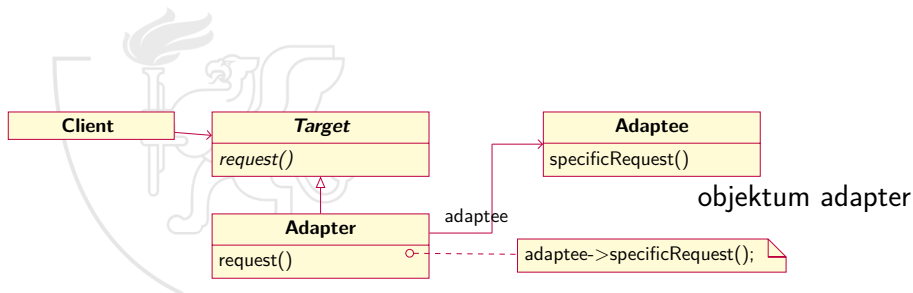
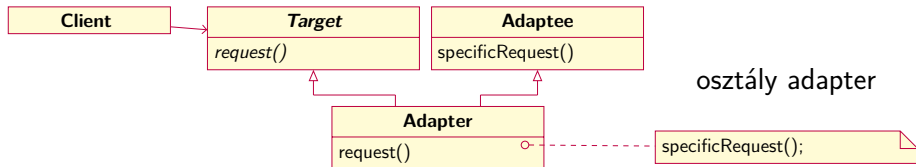
- Nagyobb szerkezetek létrehozására osztályok vagy objektumok összetételével
 - osztály szerkezeti minta: öröklődéssel hozunk létre új interfészeket, implementációkat (többszörös öröklődés). Pl. Adapter
 - objektum szerkezeti minta: objektumok összetétele új funkcionalitás létrehozásához. Pl. Composite (fontos: futás közben változhat az összetétel)
- Szinte minden szoftver területen hasznosíthatók



- Cél
 - osztály/objektumok újrafelhasználása új, kompatibilis (kliens által ismert) interfészszel
 - két típus: osztály adapter és objektum adapter
- Motiváció
 - pl. grafikus szerkesztőnél fontok kirajzolására használni egy (már meglevő) könyvtárat, amely pl. egy szövegszerkesztőben volt használatos
- Alkalmazhatóság
 - meglevő osztályt akarunk használni, de annak nem megfelelő az interfésze
 - újrafelhasználható osztályt akarunk írni, amelyet több nem kapcsolódó osztály fog használni

Adapter (folyt.)

• Szerkezet



- Résztvevők

- Target: alkalmazás-specifikus interfész, amelyet a kliens használ (pl. Shape)
- Client: a Target interfész szerint használja az objektumokat (pl. DrawingEditor)
- Adaptee: megvalósítja a szükséges funkcionalitást és definiálja az adaptálandó interfészt (pl. TextView)
- Adapter: illeszti az Adaptee interfészét a Target interfészhez (pl. TextShape)

- Működés

- a kliens az Adapter interfészét használja, az pedig leképezi a hívásokat az Adaptee interfészére

- Ismert alkalmazások

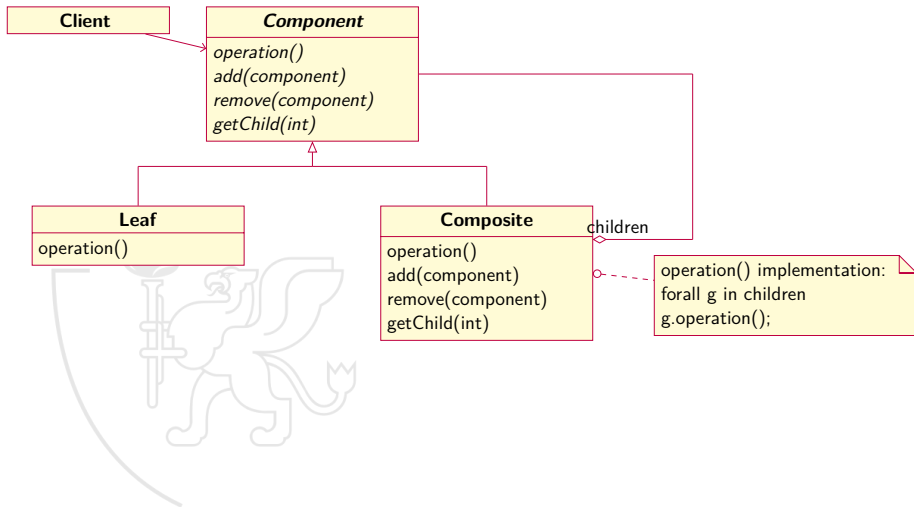
- motivációnál leírt szöveg-kirajzolás
- felhasználói felület rajzolásánál egyéb grafikus elemek kirajzolása (pl. dialógusra kép)
- absztrakt adattípusok adaptálása (pl. asszociatív tömb megvalósítása piros-fekete fával)



- Cél
 - rész-egész szerkezetek leképezése objektum-hierarchiára (fa-szerkezetre), miközben a kliens a rész és egész egységeket azonosan kezeli
- Motiváció
 - rajzszerkesztő, melyben a primitívekből összetett rajzelemek építhetők fel rekurzívan és ezek a primitív rajzelemekkel azonos módon használhatók (pl. azonos művelet hívása a kirajzoláshoz)
- Alkalmazhatóság
 - rész-egész szerkezetek hierarchiáját kell leírni
 - a kliensnek nem kell hogy észrevegye a különbségeket a kompozíciók és a primitívek között

Composite (folyt.)

• Szerkezet



- Résztvevők

- Component: interfészt deklarál a részek (gyerekek/szülő) elérésére/manipulálására (pl. Graphic). Definiálja a hierarchiában az alapértelmezés szerinti közös viselkedést
- Leaf: a kompozíció primitív eleme, gyerekek nélkül. Definiálja a primitív objektumok viselkedését (pl. Rectangle, Line, Text, ...)
- Composite: definiálja az összetett objektumok viselkedését, tárolja a gyerek komponenseket és implementálja a rájuk vonatkozó interfész Component operációkat (pl. Picture)
- Client: a szerkezet objektumait a Component interfészen keresztül manipulálja



• Működés

- ha a kliens üzenete egy Leaf-re vonatkozik, akkor az a kérést közvetlenül kezeli, ha viszont Composite-ra, az a gyerekeinek továbbítja az kérést esetleges egyéb tevékenységek mellett
- rekurzív fa-szerkezet alakul ki, amelynek elemeit a kliens egységesen kezeli
- a kliens egyszerűbb, nem is kell hogy tudjon arról, hogy az primitívet kezel, vagy összetett
- könnyű új komponenseket hozzáadni: Leaf vagy Component leszármazottjai (a klienst nem kell módosítani)
- túl általánossá válhat tőle a rendszer: a szerkezet egyes elemei akármik lehetnek és futás közbeni típusvizsgálatra lehet szükség

- Ismert alkalmazások

- szinte minden jól megtervezett OO rendszerben van, pl. szinte minden felhasználói felület könyvtár használja
- szintaktikus elemzők is használják a belső reprezentáció tárolására

- Kapcsolódó minták

- sokszor a Decorator mintával együtt szerepel
- Iterator mintával lehet bejárni a gyerekeket
- Visitor minta hasznos az egész szerkezet bejárásánál



Composite példa

```
public interface Department {  
    void printDepartmentName();  
}
```

```
import java.util.ArrayList;  
import java.util.List;  
  
public class Institute implements Department {  
    private String name;  
    private List<Department> childDepartments = new ArrayList<>();  
  
    public void printDepartmentName() {  
        childDepartments.forEach(Department::printDepartmentName);  
    }  
  
    public void addDepartment(Department department) {  
        childDepartments.add(department);  
    }  
  
    public void removeDepartment(Department department) {  
        childDepartments.remove(department);  
    }  
}
```

Composite példa (folyt.)

```
public class SoftwareEngineering implements Department {  
    private String name = "Department_of_Software_Engineering";  
    public void printDepartmentName() {  
        System.out.println(name);  
    }  
}
```

```
public class TechnicalInformatics implements Department {  
    private String name = "Department_of_Technical_Informatics";  
    public void printDepartmentName() {  
        System.out.println(name);  
    }  
}
```

Composite példa (folyt.)

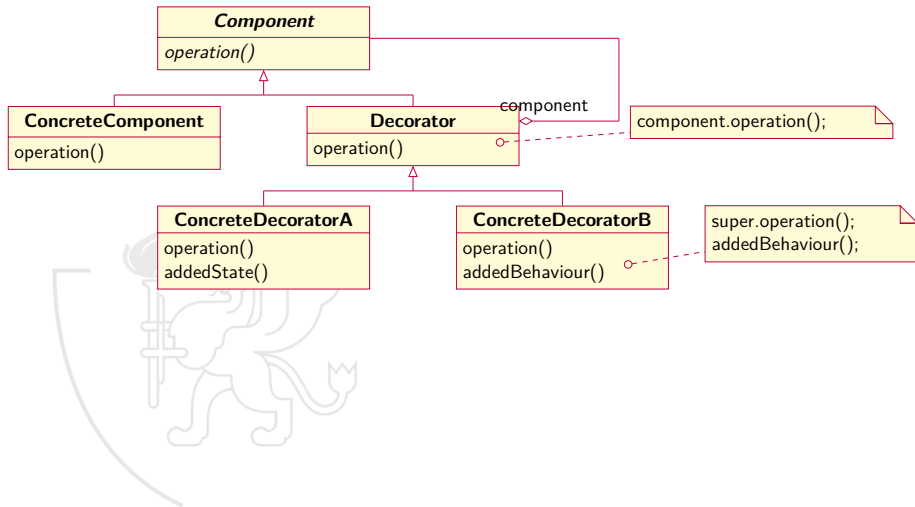
```
public class CompositeDemo {  
    public static void main(String args[]) {  
        Department sed = new SoftwareEngineering();  
        Department ti = new TechnicalInformatics();  
  
        Institute headDepartment = new Institute();  
        headDepartment.addDepartment(sed);  
        headDepartment.addDepartment(ti);  
  
        headDepartment.printDepartmentName();  
    }  
}
```

Department of Software Engineering
Department of Technical Informatics

- Cél
 - további felelősségek/tulajdonságok dinamikus csatolása az objektumhoz
- Motiváció
 - további funkcionalitást akarunk rendelni egyes objektumokhoz, de nem egy egész osztályhoz
 - öröklődéssel is megoldható, de a futásidejű választás és változtatás körülményes (és mindegyiket kell bővíteni)
 - egy új dekorátor objektum, amely tartalmazza az eredeti objektumot is, miközben a kliensnek minden művelet átlátszó Pl.: görgetősáv vagy keret hozzáadása egy ablakhoz
- Alkalmazhatóság
 - a származtatás nem lehetséges, vagy nem praktikus

Decorator (folyt.)

- Szerkezet



- Résztvevők

- Component: interfészt definiál az objektumokhoz, amelyekhez később, dinamikusan további felelősségeket kapcsolhatunk (pl. VisualComponent)
- ConcreteComponent: definiál egy objektumot, amelyhez további felelősség kapcsolható (pl. TextView)
- Decorator: tárol egy hivatkozást a Component objektumra és vele megegyező interfészt definiál
- ConcreteDecorator: felelősségeket csatol a komponens objektumhoz (pl. BorderDecorator, ScrollDecorator)



- Működés


- a dekorátor a dekorált komponens interfészéhez alkalmazkodik, az üzeneteket továbbítja, de közben saját funkciót is ellát, pl. keretet rajzol

- Következmények

- flexibilisebb mint a statikus öröklődés: futás közben módosíthatók a felelősségek
- a dekorátor objektum transzparens, de baj lehet az objektumok identitásával
- sok kicsi és hasonló objektum keletkezik, ezért nehéz a rendszer megértése és tesztelése

- Ismert alkalmazások

- sok felhasználói felület kezelő könyvtár használja a motivációnál leírt módon

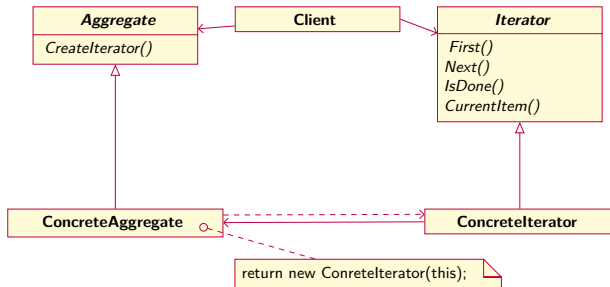
- 
- 1 **Bemutakozás**
 - Kurzus információk
 - 2 **Objektumorientált tervezés**
 - A modellezés alapelvei
 - A „siker háromszöge”
 - UML
 - Objektumorientáltság alapfogalmak
 - 3 **Java alapok**
 - Bevezető
 - Az első java program
 - 4 **Java**
 - Programozási alapismeretek ismétlése
 - Objektumok, osztályok
 - Polimorfizmus
 - Annotációk
 - Objektumok kezelése
 - Hozzáférés szabályozása
 - Beágyazott osztályok
 - Újrafelhasználhatóság
 - Interfészek és enumerációk
 - Objektumok tárolása
 - 5 **Tervezési minták**
 - Hibakezelés kivételekkel
 - Sztringek
 - Generikus típusok
 - Típus információk
 - I/O
 - GUI; Belső, lokális és anonim osztályok, lambdák
- **Minták áttekintése**
 - **Gyártási minták**
 - **Szerkezeti minták**
 - **Viselkedési minták**

- Algoritmusokkal foglalkoznak
- Felelősségek szétosztása objektumok között
- Nem csak osztályokat és objektumokat, hanem a kommunikációt is leírják
- Osztály viselkedési minták öröklődéssel valósulnak meg, pl. Template Method (algoritmus absztrakt leírása)
- Objektum viselkedési mintáknál egyenrangú objektumok kommunikálnak anélkül, hogy mindegyik ismerje mindegyiket (pl. Mediator)



- Cél
 - tároló-objektum elemeinek sorozatos elérése a reprezentációtól függetlenül
- Motiváció
 - egy tároló (aggregátor) objektum (pl. lista) elemeinek elérése függetlenül a belső reprezentációtól, többféle bejárással
 - a kliensnek ne kelljen foglalkoznia az implementációval
 - a lista interfészét ne kelljen módosítani
- Alkalmazhatóság
 - egységes interfészre van szükség különböző tároló szerkezetekhez
- Konkrét megvalósítás: `java.util.Iterator`

Szerkezet

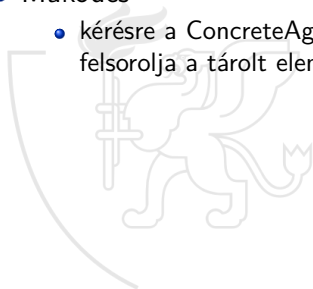


• Résztvevők

- Iterator: interfész a tárolt elemek elérésére és bejárására
- Concreteliterator: implementálja az Iterator interfészt és nyilvántartja az aktuális bejárési pozíciót
- Aggregate: interfész az Iterator objektum létrehozására
- ConcreteAggregate: implementálja az iterátor létrehozásának interfészét és visszaadja a konkrét iterátort (ez maga a tároló)

• Működés

- kérésre a ConcreteAggregate létrehoz egy új Concreteliterator-t amely felsorolja a tárolt elemeket (tárolja az aktuális pozíciót és léptetni tud)



- Következmények
 - tároló bejárására tetszőleges variáció lehet új iterátorokkal: fordított irányú, filterezett, ...
 - ha van iterátor, a tároló interfésze sokkal egyszerűbb
 - maga az iterátor tárolja az iterálás állapotát, ezért több bejárás is lehetséges egyszerre
- Ismert alkalmazások
 - pl. konténer osztálykönyvtárak
- Kapcsolódó minták
 - Composite mintánál sokszor használják
 - Memento használható arra, hogy az iterálás állapotát rögzítsük

Köszönöm a figyelmet!

