**Objektumorientált programozás**

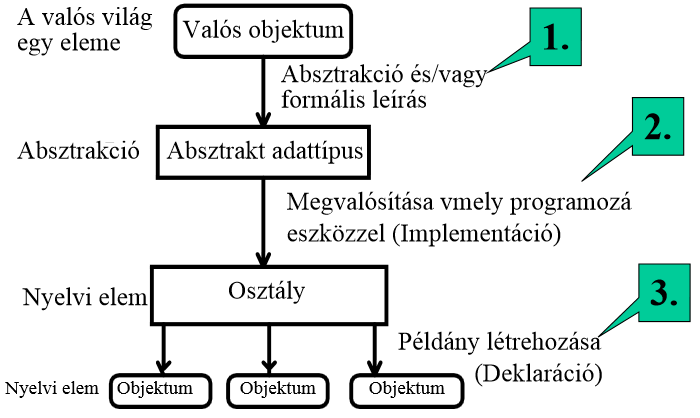
az objektumorientált programozás (object-oriented programming, oop) egy programozási módszertan. ellentétben a korábbi programozási módszertanokkal, nem a műveletek megalkotása áll a középpontban, hanem az egymással kapcsolatban álló programegységek hierarchiájának megtervezése. az objektumorientált gondolkodásmód lényegében a valós világ modellezésén alapul.

*absztrakció*: a valós világ végtelen és bonyolult. minél mélyebbre megyünk az elemzésében, annál bonyolultabb. a lebontásban tehát valahol meg kell állnunk, illetve szelektálnunk kell. az absztrakció egy szemléletmód, mellyel a valós világot leegyszerűsítjük úgy, hogy csak a lényegre koncentrálunk. az egyes objektumoknak csak azon tulajdonságait vesszük figyelembe, melyek a cél elérése érdekében feltétlenül szükségesek.

a legegyszerűbb matematikai fogalmak is, mint a szám vagy a pont fogalma, magasfokú, és történetileg szinte mindig több évszázad, évezred alatt végbemenő absztrakció eredményei. e folyamat során dolgok (tárgyak, fogalmak) egy összességét tekintve elvonatkoztatunk azon tulajdonságoktól, melyek a vizsgálat szempontjából lényegtelenek, és csak bizonyos kiemelt tulajdonságokat veszünk figyelembe.

*programkészítés:* absztrakciós folyamat, amelyben a valós világban létező jelenséget (megoldandó problémát) valamilyen programozási eszköz absztrakciós szintjén képezünk le.

*a programkészítés egy másik megközelítése:* a program a valós világ egy szeletének működő modellje. ebben a szemléletben a valós világ egy absztrakt modelljét (analízis modell) kell leképezni egy programozási eszközre. oop esetén az analízis modell elkészítése során a valóságot objektumok halmazaként tekintjük. ezen objektumok egymással kapcsolatban vannak és együttműködnek.



*objektum*: a valós világ elemeinek programozási modellje. az objektum adatokat tárol, és kérésre tevékenységeket végez.

*osztály*: az egymáshoz hasonló objektumok általánosítása, gyűjtőfogalma. programozás technikai szempontból egy típus.

az objektum által tárolt adatokat szokás az objektum mezőinek, adattagjainak vagy tagváltozóinak hívni. az objektumokat függvények vagy eljárások hívásával kérjük meg a tevékenységek elvégzésére. ezek az alprogramok az objektum metódusai. az objektum tehát a hozzá tartozó mezők és metódusok összessége.

minden egyes objektum egy-egy jól meghatározott osztályhoz tartozik. az osztály forráskódjában definiáljuk a hozzá tartozó objektumok mezőit és metódusait. az egyes objektumok a megfelelő osztály példányai – az objektumok az osztályok példányosításával jönnek létre.

**Objektumorientált programozás alapelvei**

***1. egységbezárás (encapsulation)***

az összetartozó adatokat és a hozzájuk kapcsolódó tevékenységeket egyetlen objektumban fogjuk össze.

az osztály mezői tárolják az információkat.

a metódusok kommunikálnak a külvilággal.

információ elrejtés: a feladatok elvégzésének hogyanja az objektum belügye (a módszerek implementációja rejtett). az objektum belseje sérthetlen. az objektummal csak az interfészeken keresztül lehet kommunikálni.

interface: a külvilág számára elérhető módszerek együttese.

***2. öröklődés (inheritance)***

osztályok továbbfejlesztésének lehetősége. ennek során a származtatott osztály örökli ősétől azok attribútumait, és metódusait, de ezeket bizonyos szabályok mellett újakkal egészítheti ki, és meg is változtathatja.

ősosztály (szülő) -> származtatott osztály (gyerek)

egy ősből több leszármazott osztályt is készíthetünk

egy osztálynak legfeljebb egy szülője lehet javaban – öröklődési fa

egy osztálynak több szülője is lehet c++-ban – örölődési gráf

***3. sokalakúság (polimorfizmus)***

egy referencia által mutatott objektum típusa egy osztályhierarchián belül futásidőben dől el, azaz „többalakú”. ennek megfelelően egy osztályban definiált metódusok az alosztályokban felüldefiniálhatók, így csak futásidőben kerül meghatározásra (kései kötés), hogy ténylegesen melyik kód kapja meg a vezérlést. tehát polimorfizmus esetén egy objektumnak több típusa is lehet.

a gyermek típus rendelkezik a szülőjének összes attribútumával, és metódusával. tehát minden eseményre reagálni tud, amire a szülő is. a gyermek minden olyan helyzetben használható, amiben a szülő is. erre mondhatjuk azt, hogy a gyermek típusa a szülő típusának egy altípusa.

- ugyanaz a művelet meghívható a szülő és a gyermek osztállyal is, tehát több típussal rendelkezik a művelet.

- egy rögzített típusú változó hivatkozhat több különböző típusú objektumra (szülő típusú referenciának értékül adható egy leszármazott típusú példány).

- egy metódus aktuális paraméterének típusa lehet a megadott formális paraméter leszármazottja.

*statikus típus:* a változó deklarációjában megadott típus. fordítás során egyértelműen eldől, nem változhat futás során. a statikus típus határozza meg, hogy mit szabad csinálni az objektummal (pl. hogy milyen műveletek hívhatók meg rá).

*dinamikus típus*: a változó által hivatkozott objektum típusa. vagy a statikus típus leszármazottja, vagy maga a statikus típus. futás során változhat.

*statikus kötés (korai kötés):* fordításidejű hozzárendelése a hívott metódusnak, adattagnak az objektumhoz. (pl. c++: nem virtuális függvények, java: osztálymetódusok - statikus metódusok).

*dinamikus kötés (kései kötés):* futásidejű hozzárendelése a hívott metódusnak, adattagnak az objektumhoz. (pl. c++: virtuális függvények, java: példánymetódusok).

*virtuális függvény:* a származtatás során az ősosztályok metódusai képesek legyenek az új átdefiniált metódusok használatára újraírás nélkül is. ezt virtuális metódusokon keresztül érhetjük el. javaban minden metódus virtuális.

tehát: ha létrehozunk egy osztályt és benne egy függvényt, majd ebből az osztályból származtatunk egy újat és abban szintén deklaráljuk az előbbi függvényt ugyanazzal a névvel, akkor ennek az lesz az eredménye, hogy elrejtjük az ősosztálybeli függvényt. ha viszont az ősosztályban található függvényt virtuálisként deklaráljuk, akkor a leszármazott osztályban lehetőségünk van felülírni azt. ennek következményeként mindig az a függvény hívódik meg, amelyik osztály típusaként tároljuk az adott példányt.

**Java: alapok**

Általános célú programozási nyelv: olyan programozási nyelv, amit széles körűen fel lehet használni szoftverek írására. A java általános célú.

Obketumorientált programozás (OOP): olyan programozási paradigma, amely a programokat objektumokból építi fel. A java objektum orientált programozási nyelv. A 8 primitív típuson (boolean, char, byte, short, int, long, float, double) kívül minden az Object osztály leszármazottja.

Platform független: a Javában íródott programok hasonlóan fognak futni különböző hardvereken. Ezt úgy lehet megvalósítani, hogy a Java fordítóprogram csak egy úgynevezett Java bájtkódra fordítja le a forráskódot, ami aztán futtatva lesz a virtuális gépben.

Vannak olyan Java fordítóprogramok, amelyek natív gépi kódra fordítják le a forráskódot.

Garbage Collection: egy formája a biztonságos memóriakezelésnek. A garbage collector (GC, szemétgyűjtő) megkísérli eltávolítani a memóriából azokat az objektumokat, amelyeket az alkalmazás már nem használ. A javaban nem kell memóriát felszabadítanunk, hanem a virtuális gép megteszi ezt helyettünk.

A Java-nak nincs hivatalos standardizációja, az Oracle implementációja számít a de facto standardnak. Ez az Oracle implementáció két részre van osztva. Az egyik része a Java Runtime Environment (JRE), a másik pedig a Java Development Kit (JDK).

Java SE: Java Platform, Standard Edition. Egy széles körben elterjedt és használt platform specifikáció a Java nyelv programozásához. Széleskörben használatos általános API-kat határoz meg, és tartalmazza a Java nyelv specifikációt (JLS) és a Virtuális gép specifikációt.

Java EE: Java Platform, Enterprise Edition. Abban különbözik a Standard Edition-től, hogy több programkönyvtárat (API-t) tartalmaz. Mindkettőnek a legismertebb implementációja a JRE+JDK páros.

A java forráskódot először le kell fordítani bájtkódra egy fordítóprogrammal (compiler)

.java -> .class

A bájtkódot a Java virtuális gép (JVM) interpretálja

A JDK fordítóprogramja: javac

fordítás: javac HelloWorld.java

futtatás: java HelloWorld

jar fájl futtatása: java –jar filename.jar

path kiíratása: echo %PATH%

path beállítása: set PATH=%PATH%;”c:\Program Files\Java\jdk.1.6.0\_18\bin\”

Tehát a java három szerepet tölt be: programozási nyelv, köztesréteg (middleware) és platform.

**Java: objektum és osztály**

Az objektum egy létező alkatrésze a programnak, amely egy terv alapján készül el, ezt a tervet hívjuk osztálynak. Tehát az osztály a sablon és amikor konkrét dolgokat hozunk létre a sablon alapján az az objektum.

Egy objektum adatokból és műveletekből áll. Az adatot, melyeket változókkal írunk le, attribútumnak nevezzük. A műveletek az objektum függvényei, amelyeket metódusnak hívunk.

Egy új osztály létrehozásakor annak alapja egy másik osztály. Ez a származtatás. Az új osztályban a szülő osztály metódusai és attribútumai használhatóak, és saját attribútumokat és metódusokat is tartalmazhat. Minden objektumnak van szülője a Javaban (Object class).

package kockapoker;

public class Dobokocka {

private int number;

public Dobokocka(int n){

this.number = n;

}

public int getNumber{

return number;

}

}

*package kockapoker:* a csomag neve, ahol az osztály található. Nem kötelező, de elvárt a megadása. A csomagnév meghatározza, hogy a fájlban lévő osztály melyik csomag része. Csomagokba vannak szervezve a Java környezet osztályai és az általunk írt osztályok is. Egy csomag tartalmazhat alcsomagot, amelynek szintén lehet alcsomagja, vagyis a csomagok fa struktúrát alkotnak.

Ha a csomagunkat több eltérő helyen lévő alkalmazásból is szeretnénk használni, akkor a könyvtárat egy, a CLASSPATH környezeti változó által leírt központi helyre kell tennünk.

A CLASSPATH értéke könytárak sorából áll (Windows-on pontosvesszővel elválasztva). Amikor a rendszer keresni kezdi az a.b.C nevű osztályt, veszi az elsõ könyvtárat a CLASSPATH-en. Ehhez hozzáfûzi a csomag nevét, így lesz egy alkönytárnév és ebben keresi meg az osztálynév.class fájlt. Ha nem találja, továbblép a következõ könyvtárra a CLASSPATH-en és ezt addig csinálja, amíg a könyvtárak a CLASSPATH-en el nem fogynak.

Tegyük fel, hogy a CLASSPATH értéke a következõ: c:\Users\javadev;c:\Users\javalib. Tegyük fel továbbá, hogy a rendszer az a.b.C nevû osztályt keresi. Veszi tehát az elsõ könyvtárat a CLASSPATH-en, a c:\Users\javadev-et. Ehhez hozzáilleszti a csomagnevet, vagyis az a.b-t és kapja a c:\Users\javadev\a\b könyvtárat. Itt megpróbálja fellelni a C.class fájlt. Hogy nem találta meg, akkor továbblép a c:\Users\javalib-re és felkeresi a c:\Users\javalib\a\b\C.class fájlt. Ha ez megvan, siker. Ha nem, hibaüzenet.

set CLASSPATH=”c:\java\csomagok;c:\java\egyeb” - Ez a sor két könyvtárat állít be a CLASSPATH-nak.

CLASSPATH kiíratása java programon belül: System.getProperty("java.class.path"), kiíratás parancssorban: echo %CLASSPATH%

*public class Dobokocka:* a publikus osztály neve. Minden osztályt névvel kell ellátnunk, amely név a Java nyelv előírásait teljesíti. Ha az osztály neve több szóból áll, akkor minden szót nagybetűvel kezdünk, és egybeírjuk - szóköz nélkül (például: OsztályNév). Lehetőleg csak angol karaktereket használjuk. Egy .java fájlban csak egy publikus osztály lehet, és ennek az osztálynak a nevének meg kell egyeznie a fájl nevével. Ezenkívül lehet akárhány láthatóság módosító nélküli osztály egy fájlban, de ezeket csak az adott package-en belül lehet látni.

Az osztály írásánál el kell döntenünk, hogy az objektum adataihoz és metódusaihoz milyen hozzáférést engedélyezünk egy külső felhasználó számára. Ez a láthatóság, ami lehet: public, protected, private. Ahhoz, hogy az ősosztály adatait és műveleteit lássa a leszármazott osztály, legalább protected láthatóság kell.

*private*: a változót csak az osztályon belül lehet elérni és használni, a leszármazott osztályok sem használhatják.

*protected*: a változót az azonos csomagban lévő illetve leszármazott osztályok is elérik.

*public*: a változó nyilvános, bárki számára elérhető és módosítható.

Osztályon belül a változó neve egyedi kell legyen. Elnevezési konvenció: minden változónevet kisbetűvel írunk, ha a név több szóból áll, akkor az első szót leszámítva minden további szót nagybetűvel kezdünk, és egybeírjuk - szóköz nélkül (például: változóNév).

A *konstruktor* az a művelet, amely inicializálja az objektumot. A hozzá tartozó kód akkor fut le, amikot az adott osztályt példányosítjuk. Egy osztályhoz annyiféle konstruktort készítünk, ahányféle képpen lehetővé tesszük a példányok inicializálását. A konstruktor legfőbb ismérve, hogy a neve azonos az osztály nevével és a név után kerek zárójelet találunk, a zárójelek között pedig a bemeneti paramétereket. A paraméter nélküli konstruktort alapértelmezett konstruktornak nevezzük. A konstruktor neve előtt is lehetnek módosítók: private, protected, public.

Ha nem adunk meg semmilyen konstruktort, akkor a fordító automatikusan biztosít egyet: ez a paraméterek nélküli alapértelmezett konstruktor (default constructor). Az alapértelmezett konstruktort csak akkor kapja meg automatikusan egy osztály, ha semmilyen konstruktora nincs. Ha már csak egyet is megadunk, akkor ez a szabály nem él. Amennyiben az osztályunk egy másik osztály leszármazottja (nem közvetlenül az Objecté), abban az esetben a szülő paraméterek nélküli konstruktorát biztosítja a fordító. Fordítási idejű hibát fogunk látni, ha utóbbi esetben az ősosztály alapértelmezett konstruktora nem elérhető vagy nincs.

Tehát ha az ősosztálynak csak paraméteres konstruktora van, akkor a gyerek konstuktorának első sorában ennek kell szerepelnie: super(paraméterek); A paraméterlistának az ősosztály valamelyik konstruktorára illeszkednie kell. Amikor a gyerek automatikusan megkapja a szülő paraméternélküli konstruktorát, akkor is egy implicit super() hívással kezdődik a konstruktor végrahajtása. Ha van alapértelmezett konstruktora az ősosztálynak, a gyerek akkor is használhatja a super hivatkozást.

Változók elé kitehető a static minősítő. Az ilyen változó az osztály minden objektuma számára egy példányban létezik, azok osztottan használják. Szokás osztályváltozónak is hívni. Ha egy változó nem static, akkor minden objektum saját példánnyal rendelkezik belőle. Az ilyet példányváltozónak szokás hívni. A metódusok is lehetnek static-ok (osztálymetódus). Meghívásukhoz nem szükséges példányosítás, az osztály nevével hívjuk meg őket: Osztalynev.metodus(). Csak statikus attribútumokkal végezhetnek műveleteket és nem hívhatnak meg nem statikus metódusokat.

public class Dobokocka

{

private static int kockaDB=0;

private int sorszam;

public Dobokocka(){

kockaDB++;

sorszam = kockaDB;

}

}

Metódusok felüldefiniálása (method overriding): egy osztály leszármazottjában a metódust felül lehet definiálni. Ilyenkor az új metódus visszatérési típusának, nevének, és paraméter szignatúrájának meg kell egyeznie az ősosztálybeli metóduséval. Az új metódus hozzáférési szintje nem lehet szűkebb az eredeti metódusénál, bővebb lehet.

Metódusnév túlterhelése (method overloading): az osztályon belül több metódus azonos névvel – a paraméterlista és esetenként a visszafordított típus különbözik.

Java-programunk futtatásakor a memóriában elég szabad helynek kell rendelkezésre állnia a végrehajtandó parancsok beolvasásához. Ezenkívül szabad helyet kell találni a program által használt adatok tárolásához is. A számítógép memóriája úgy képzelhető el, mint bitek (vagy szokványosabban bájtok, ahol 1 bájt 8 bit) hosszú sorozata. Minden egyes bájtnak van egy címe, amivel azonosítani tudjuk őket. Ezekhez a konkrét memóriahelyekhez akkor tudunk hozzáférni, ha ismerjük a címét.

Például tegyük fel, hogy végrehajtunk egy metódust, amely tartalmaz egy ilyen deklarációs utasítást: int a, b, c; Ekkor a rendszer megállapítja, mekkora helyre van szükség ezeknek a változóknak (int – egyenként 32 bitre), majd megvizsgálja, hogy a memóriában talál-e ennyi szabad helyet. Ha igen, lefoglal három darab négybájtos blokkot, és feljegyzi ezek címét egy külső táblázatba. Ha később hivatkozunk ezekre a változókra a kódban, a rendszer megkeresi a változó nevét a táblázatban, és megnézi, a változó hol van eltárolva a memóriában és ezután tud hozzáférni a memóriahelyhez a tárolt érték lekérdezése vagy egy új érték beírása céljából.

Az osztályszintű hatáskörrel rendelkező változók akkor jönnek létre, amikor létrehozzuk az osztály egy példányát, és addig maradnak érvényben, amíg az adott objektum meg nem szünik. Egy metódusban deklarált lokális változó csak akkor jön létre, amikor a metódus elkezd működni, és akkor szűnik meg (törlődik a memóriából), amikor a metódus befejeződik.

Van különbség egy primitív adattípusú változó, és egy adott osztály példányaként létrejövő objektum deklarációja között. A primitív változót a deklarációja után azonnal felhasználhatjuk. Az objektumot viszont ahhoz, hogy használni tudjuk, a deklarációja után még inicializálnunk is kell.

MyClass myObject = new MyClass();

Mivel egy osztály sok információt tartalmazhat, hatékonyabb az összes hozzá tartozó információt egy összefüggő memóriaterületen tárolni úgy, hogy a tárolt információ belső szerkezete az osztály minden egyes példányára megegyezik. Így ha tudjuk a példánynak lefoglalt memóriaterület kezdőcímét, ehhez a címhez egy eltolási értéket adva megkaphatjuk az osztály bármely változóját.

Tehát ha egy objektumot deklarálunk, elég az objektumnak lefoglalt memóriterület kezdőcímét tárolnunk. A myObject nevű változó egy referencia, amely a példány számára lefoglalt tárterület kezdőcímét tartalmazza, így a segítségével a létrehozott példányra hivatkozhatunk. Tehát egy osztálytípussal rendelkező változó esetén egy referenciát deklarálunk.

MyClass myObject;

Ez a sor nem hoz létre egy teljes MyClass objektumot, csak egy helyet foglal a memóriában, ahol a majdan létrejövő példány címét tároljuk. Ekkor a példány még nem létezik. Minden referenciának automatikusan null lesz a kezdőértéke, amikor létrehozzuk, ezzel jelezve, hogy még nincs példány hozzárendelve. A példány létrehozása:

myObject = new MyClass();

A new egy unáris java-operátor, amelynek hatására létrejön az új példány. Egyetlen operandusa van, amely annak az osztálynak a nevét mondja meg, amelynek a példányát létre akarjuk hozni. (Valójában az operandusa a konstruktor). A new operátor visszatérési értéke az a memóriacím, amelyen a példánynak lefoglalt memóriaterület kezdődik.

Tehát a myObject = new MyClass(); kifejezés helyet foglal a MyClass osztály egy példányának, majd ennek címét a myObject referenciához rendeli, így később ennek segítségével hivatkozhatunk a példányra.

void ertekSzerint(int primitiv){

System.out.println("kezdeti érték: " + primitiv);

primitiv++;

System.out.println("végérték: " + primitiv);

}

Hívjuk meg a fenti metódust úgy, hogy az alábbi változót adjuk meg neki paraméternek: int probaErtek = 1; Ekkor a képernyőre először ez íródik ki: kezdeti érték: 1, majd ez: végérték: 2. A kérdés az, a metódus lefutása után mi lesz a probaErtek értéke? 1 vagy 2? Amikor egy primitív változót adunk át egy metódusnak, akkor a metódus lemásolja magának az átadott változó értékét. Más szavakkal az ertekSzerint(int primitiv)-et a fordító úgy tekinti, mint egy lokális változó deklarációját, azaz a memóriában primitiv néven lefoglal neki egy területet. E változó kezdőértéke az az érték lesz, amelyet a metódus hívásakor átadunk a metódusnak.

Tehát minden művelet, amelyet a változón végrehajtunk, csak a metóduson belüli helyi változatot érinti. Az eredeti probaErtek változatlan marad, mert fizikailag más memóriaterületen helyezkedik el. A paraméterek átadásának ezen formáját érték szerinti paraméterátadásnak hívjuk, a javában minden primitív változó mindig érték szerint adódik át, így ilyen változó értékét lehetetlen azáltal megváltoztatni, hogy átadjuk egy metódusnak.

void cimSzerint(MyClass paramObject){

paramObject.setNumber(2);

}

Hívjuk meg a metódust: MyClass probaObject = new MyClass(); cimSzerint(probaObject); Ebben az esetben a cimSzerint()-ben végrehajtott változások tartósak, a probaObject-et érintik. Ez a viselkedés az előzőtől eltérőnek tünhet. Mikor a probaObject-et átadjuk a cimSzerint()-nek, a referencia aktuális értékéről készül másolat a paramObject-be. Ugyanúgy, mint az ertekSzerint()-ben, a probaObject és a paramObject itt is független változók.

A különbség most az, hogy mivel a két referencia értéke megegyezik, ugyanazt a példányt hivatkozzák a memóriában. A paraméterátadás ezen változatát cím szerinti paraméterátadásnak nevezzük. (Szigorú értelemben véve a cím szerinti paraméterátadás más jelent, hiszen ez is érték szerinti paraméterátadás, csak itt memóriaértékeket adunk át érték szerint.)

Egy változónak van statikus és dinamikus típusa. (Nem összekeverendő a statikus változó a változó statikus típusával. Ez két különböző dolog). Egy változó statikus típusa az, amit a deklarációjában megadtunk. Ez a változó teljes élettartalma alatt változatlan. Egy változó dinamikus típusa az általa éppen hivatkozott objektum tényleges típusa. Ez a program futásakor bármikor változhat. A változó dinamikus típusa csak a statikus típus, vagy annak leszármazottja lehet.

List lista1 = new ArrayList(); List lista2 = new LinkedList();

A lista1 és a lista2 statikus típusa is List, a lista1 dinamikus típusa ArrayList, a lista2 dinamikus típusa pedig LinkedList.

Kötés (binding): egy metódushíváshoz a megfelelő definíció megtalálása. Ehhez a metódusnév nem elég. Ha az aktuális objektum osztályában a név egyedi (beleértve az örökölt metódusokat is), a hatáskör és a név elegendő az azonosításhoz. Az objektum statikus típusa alapján az azonosítást a fordítóprogram el tudja végezni. Ez a korai (statikus) kötés (fordításidejű hozzárendelése a hívott metódusnak az objektumhoz).

Lehet azonban, hogy egy osztályon belül egy metódusnév túl van terhelve. De ez is még korai kötés, mivel a fordítóprogram az aktuális paraméterek statikus típusa alapján dönteni tud. Azonban felüldefiniált metódusoknál már más a helyzet. Egy ilyen metódushívásnál dönteni kell, hogy az örökölt, vagy a saját változat hívódjon meg. A döntés alapja a hivatkozás dinamikus típusa. Ez a dinamikus (késői) kötés: futásidejű hozzárendelése a hívott metódusnak az objektumhoz.

Egy leszármazott osztály elérheti az ősosztály metódusait a super.metódusnév(…) formájú hivatkozással. Tehát ha van egy osztályunk, ami egy másik osztály leszármazottja, és ebben az ősosztály egyik metódusa felülvan definiálva (mondjuk ennek a neve myMethod), akkor ebben az osztályban a this.myMethod() a felüldefiniált, a super.myMethod() pedig az eredeti, ősosztálybeli metódust hívja meg.

**Java: absztrakt osztály és interface**

Előfordul a tervezés során, hogy egy osztály szintjén tudjuk, hogy valamilyen metódus szükséges lesz a leszármazottakban, de még nem lehet megadni az implementációját. Lehet törzs nélküli metódust is definiálni. Az ilyen metódust az abstract minősítővel kell ellátni. Ha az osztály tartalmaz absztrakt metódust, az osztályt is az abstract minősítővel kell ellátni. Tehát az, hogy egy osztály absztrakt, annyit jelent, hogy tartalmazhat absztrakt metódusokat. Absztrakt osztályt nem lehet példányosítani, csak más osztályt származtatni belőle, ahol meg kell adni az absztrakt metódusok implementációját (ha nem adjuk meg az implementációkat, akkor a származtatott osztályt is absztraktként kell deklarálni).

public abstract class Sikidom

{

public abstract double terulet();

public abstract double kerület();

}

Az absztrakt metódusok szerepe: rögzít egy tervezési döntést (szükséges metódusok halmaza). Kényszeríti a leszármazott osztályok programozóját meghatározott metódusok definiálására.

Az interface viselkedésmódot definiál. Gyakorlatilag egy művelethalmaz deklarációját jelenti. Az információ rejtés (zártság) elve miatt általában csak metódusokból áll. Egy interfész kényszeríti az osztály készítőjét a megfelelő elemek implementálására. Szintaktikája hasonló az osztályéhoz, de a class kulcsszó helyett az interface kulcsszót kell használni. Az interfész nem tartalmazhat végrehajtható kódot, azt a megvalósító osztályban kell megadni (implements). Ha egy osztály implementál egy interfészt, akkor annak minden egyes metódusát implementálnia kell. Ez alól egyetlen kivétel az absztrakt osztály. A fordító akkor sem fog panaszkodni, ha az interfész metódusai meg sem jelennek, mint absztrakt metódus az absztrakt osztályban. Ezeket majd a származtatott osztályokban kell csak implementálni.

Az interfész tulajdonképpen a „legabsztraktabb osztály”. Ha absztrakt osztályunk csak absztrakt metódusokat tartalmaz, akkor azt inkább interfészként érdemes deklarálni. Így egy interfész sem példányosítható. De lehet referencia statikus típusa, ahogyan az absztrakt osztály is. Az implementáló osztály lehet egy ilyen referencia dinamikus típusa. PL: List lista1 = new ArrayList(); Tehát a java megengedi, hogy egy bizonyos osztályúnak deklarált objektumot úgy hozzunk létre, hogy valójában az osztálynak egy származtatott osztályát pédányosítjuk. Ilyenkor, ha egy olyan metódust hívunk meg, ami a példányosított osztályban benne van ugyan, de a statikus típust megadó osztályban nincs, akkor hibát fogunk kapni, mert csak futási időben derül ki, hogy mi a dinamikus típus, melyben a metódus is található. A hiba javítására két mód van.

Az első, hogy definiáljuk a metódust az ősosztályban is, amely nem csinál semmit, csak azért van, hogy a fordító ne jelezzen hibát. Ez a metódus lehet üres törzsű, de lehet törzs nélküli absztrakt. Másik megoldás a típuskényszerítés. PL: ((Teglalap)test).TeglalapMethod. A fordító ellenőrzi, hogy a kényszerített osztály leszármazottja-e az ősosztálynak, ha igen, akkor megnézi, hogy ez a leszármazott osztály tartalmazza-e a hívott metódust.

Az interfészek esetében is beszélhetünk öröklődésről, egyik interfész lehet egy másik interfész leszármazottja. Erre alkalmazás példa: egy interfésznek már léteznek megvalósításai és ki szeretnénk egészíteni további metódusokkal. Az összes megvalósító osztályt módosítanunk kellene. Jobb megoldás lehet a származtatás (a származtatott interfészben kapnak helyet az új metódusok). Interfészek esetében létezik többszörös öröklődés.

Absztrakt osztály és interfész közötti különbségek:

- az interface tisztán osztály vázat definiál, az absztrakt osztály megadhat implementációt is (azaz az interface-ben minden metódus absztrakt, míg az absztakt osztálynál nem feltétlenül, lehet hogy csak egy)

- interface adattagja csak public static final lehet

- egy osztály több interfészt is implementálhat egyszerre, függetlenül attól, hogy azok mennyire állnak kapcsolatban egymással

a instance of B – Ez a logikai művelet akkor igaz, ha az „a” változó dinamikus típusa leszármazottja a „B” típusnak, egyébként hamis.

**Java: beágyazot osztályok**

A legtöbbet olyan osztályokat használunk, amelyek nem másik osztályba vagy interfészbe vannak beágyazva. Az ilyen osztályt legfelső szintű osztálynak nevezzük (top level class).

A Java nyelv megengedi osztályok deklarálását más osztályokban is, ezeket beágyazott osztályoknak nevezzük (nested class).

Egy beágyazott osztály kétféle lehet: statikus (static nested class) vagy nem statikus, azaz belső osztály (inner class).

Statikus metódusokat csak a statikus beágyazott osztályok használhatnak. A statikus beágyazott osztály nem érheti el a nem statikus tagokat (csak ha példányreferencián keresztül teszi azt) és osztályreferencián keresztül hivatkozunk rá.

A nem statikus belső osztályok példányosítását minden esetben a külső osztály példányosítása előzi meg.

Deklarálhatunk egy belső osztályt egy metóduson belül is. Ezek a lokális osztályok (local classes).

Léteznek még név nélküli osztályok is (anonymous classes). Az osztály példányosítása és definiálása egyszerre, egy helyen történik.

**Java: generikus osztályok**

Ha általános célú típusokat (osztályokat, interfészeket) definiálunk, minden paraméter és visszatérési érték csak a lehető legáltalánosabb típusú lehet (tipikusan Object). Ebből következően csak futási idejű típusellenőrzés van, a kódot pedig olvashatatlanná teszik az explicit típuskonverziók (castok). Erre a problémára megoldást nyújt a generics. A Java generic-ek, azaz generikus osztályok valamilyen típussal paraméterezett osztályok. Segítségükkel problémafüggetlenül lehet algoritmusokat, adatszerkezeteket definiálni. A generikus típusokból típusparaméterek megadásával paraméterezett típusok (parameterized type) hozhatóak létre. A paraméteres típusok mindenhol használhatóak, ahol szokványos típusok, tehát más paramérerezett típusok típusparamétereiként is. A primitív típusok sem használhatóak típusparaméterként.

ArrayList<Integer> list = new ArrayList<>();

itt a paraméterezett típus (osztály): ArrayList<Integer>

Egy generikus osztály, illetve interfész deklaráció típusok egy halmazát definiálja. A halmaz egyes elemeit a megadott típusparaméterek határozzák meg.

class Pair<N,M>{

public N getFirst() {…}

public M getSecond() {…}

}

A < és > jelek között a típusváltozók szerepelnek. A típusváltozókat a deklaráció további részében típusként használhatjuk.

A típusváltozókhoz az extends kulcsszóval korlátokat is megadhatunk, amelyek a behelyettesíthető típusparaméterek körét szűkítik. Korlátként megadhatunk egy osztályt vagy egy interfészt, illetve & jelekkel elválasztva tetszőleges számú további interfészt. A megadott korlátok ellenőrzését a fordító ellenőrzi.

class Pair<N extends Number&Comparable, M extends Iterator>{…}

A korlátok célja a generikus típus paraméterezhetőségének szabályozása. A korlátok alkalmazásával az explicit típuskonverziók is szükségtelenné válnak.

class Valami<E>{

E e=…;

Iterator i = ((List) e).iterator(); //futásidejű típusellenőrzés

}

class Valami<E extends List>{ //fordításidejű típusellenőrzés

E e=…;

Iterator i = e.iterator(); //nincs explicit típuskonverzió

}

Ha egy típusparaméter konkrét értékére nincs szükség, használhatunk határozatlan típusparamétereket (Wildcard):

public void printList (List<?> l){

for(Iterator<?> i = l.iterator(); i.hasNext();)

…

}

A fenti metódus segítségével tetszőleges elemtípusú lista elemeit kiírathatjuk. A ? joker mellett megadhatunk felső, illetve alsó korlátot is az extends és a super kulcsszóval.

class MyList<E> {

public void addAll (List<? extends E> list) {…}

public <T> void assign (T value, List<? super T> list) {…}

}

A metódusok is elláthatóak típusparaméterekkel. A típusparamétereket a metódus visszatérési értéke előtt kell feltüntetni.

public <E> void csere(E[] a, int i, int j){…}

A Collection <?> minden kollekcióra ráillik. Ilyenkor object-ként hivatkozunk az elemekre.

public void print(Collection<?> c) { for (Object o : c) {System.out.println(o); } }

**Java: kivételkezlés**

A kivétel (exception) egy olyan esemény, amely a program futása során léphet fel, megszakítva annak helyes működését. Ilyen hibákat előidézhet:

- a felhasználó (rossz program használat),

- a programozó (rosszul megírt kód, vagy rossz hibakezelés),

- a környezet (pl: JVM, hardware, hivatkozási hibák, stb.).

Amikor egy hiba bekövetkezik egy metódusban, akkor létrehozásra kerül egy kivétel objektum, ami a futtatókörnyezetnek adódik tovább. Ez a kivétel objektum információt tartalmaz a hibáról, annak típusáról, és a program állapotáról amikor kiváltódott. A kivétel objektum létrehozását és átadását a futtatókörnyezetnek, kivétel dobásnak (throw) nevezzük.

Ha egy metódus dob egy kivételt, akkor a futtatókörnyezet megpróbálja azt lekezelni. És ezt hogy végzi? Nos, a hívási stack-ben elkezdi keresni azt a metódust, amely tartalmaz olyan kódrészletet, amely le tudja kezelni az adott kivételt (exception handler). A keresés sorrendje a hibát kiváltó metódussal kezdődik, és a metódushívások sorrendjének megfelelően visszafelé keres. Az a hibakezelő kerül kiválasztásra, amely az adott hibatípust tudja kezelni.

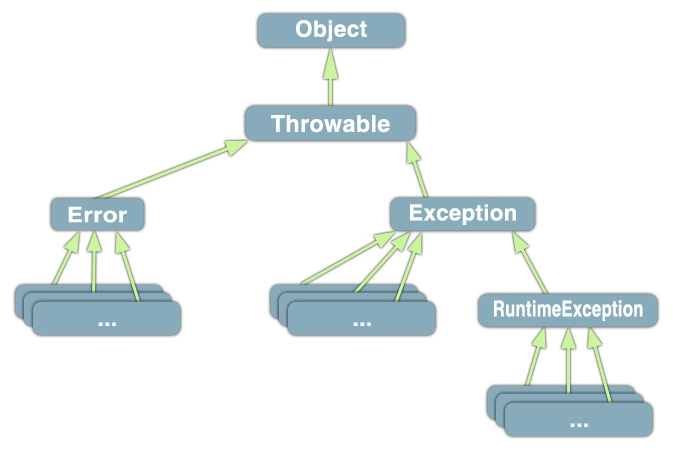
Ha sikerült találni egy megfelelő hibakezelős részt, akkor az a rész elkapja (catch) a hibát. Ha nem sikerült találni hibakezelőt, akkor a futtatókörnyezet terminál.

Ahhoz, hogy megértsük a Java kivételkezelését, ismertünk kell a három kivétel kategóriát:

- *Checked exceptions* – Ezek az ellenőrzött kivételek, amelyek kezelését nem hagyhatja figyelmen kívül a programozó. Ezek megfelelősége már fordítási időben kiderül. (ősosztály: java.lang.Exception, pl. java.lang.ClassNotFoundException, CloneNotSupportedException)

- *Runtime exceptions* – Ezek a futás idejű, vagy nem ellenőrzött kivételek. Ezek a program futása során léphetnek fel. Ezeket meg lehet előzni, ha a programozó jó tesztekkel kiszűri a problémás eseteket, és megírja hozzájuk a kivételkezelést. Ezeket a fordító nem ellenőrzi fordításkor. (ősosztály: java.lang.RuntimeException, pl. ArrayIndexOutOfBoundsException, NumberFormatException, DivisionByZeroException)

- *Errors* – Ezek nem kivételek, hanem olyan kritikus hibák, amelyekkel nem igazán lehet mit kezdeni (pl: elfogy a JVM memóriája futás közben).



A try-catch-finally utasításszerkezet biztosítja a kivételkezelést.

*A* try blokk egy olyan rész, amely közrezárja a kritikus utasításokat. Általános alakja így fest:

try {

// kritikus utasítások

}

A try blokk törzsében figyeljük azt a problémás kódrészletet, amely kivételt dobhat. Ha ez bekövetkezik, vagyis létrejön egy kivétel, akkor az tovább adódik a hozzárendelt hibakezelő catch résznek. A try blokkot mindig követnie kell legalább egy catch blokknak, és köztük semmilyen más utasítás nem szerepelhet.

try {

// kritikus utasítások

} catch (Exception1 e1) {

// Exception1 kezelése

} catch (Exception2 e2) {

// Exception2 kezelése

}

Egy catch ág az argumentumában meghatározott kivételt képes csak elkapni/lekezelni. A try blokkban kiváltott kivételt az a catch ág fogja lekezelni, amelybe osztályhierarchia szerint beleillik. Ez mit is jelent? A kivételek is osztályok, ráadásul származtatva vannak. Ezért a különböző catch ágakban az osztályhierarchiának megfelelően kell meghatározni a kivételtípusokat. Mindig a legszűkebbtől (hierarchia alja) kell kezdeni és haladni a bővebb felé. A fenti általános alakban az Exception1 lejjebb van a hierarchiában, mint az Exception2, ezért ő az első.

A finally blokk opcionális, megléte nem kötelező. A finally mindig végrehajtódik. Akkor is ha kiváltódott egy kivétel, akkor is ha nem. Ebbe a részbe általában olyan utasítások kerülnek, amelyek erőforrásokat szabadítanak fel (pl: fájlok, streamek lezárása).

try {

// kritikus utasítások

} catch (Exception1 e1) {

// Exception1 kezelése

} finally {

// ...

}

**Maven**

Egy alkalmazás készítése során nem célszerű minden programrészletet saját magunk elkészíteni, hiszen lehet, hogy egyes problémákra (pl. e-mail küldés smtp protokollon keresztül) már léteznek mások által kifejlesztett, elterjedt, jól bejáratott megoldások. Egy nagyobb szoftver projekt temérdek ilyen framework-öt, programcsomagot tartalmazhat, melyeket összefoglaló néven függőségeknek (dependency) szoktunk nevezni. Például egy Java projekt függőségei általában jar fájlok szoktak lenni.

Réges-régen, egy szomorúbb korban, amikor becsatlakoztunk egy projektbe, vagy elkezdtünk egy újat, saját maguknak kellett ezeket a függőségeket letölteni, beállítani az IDE-t, annak érdekében, hogy a projekt minden függőséget figyelembe véve, megfelelően forduljon, mely sok bosszúságot és álmatlan éjszakát okozhatott. Egykor az Apache Ant eszköz szolgált a konfigurációs problémák megoldására, de ez mára már elavulttá vált a script fájljainak bonyolult nyelvezete miatt, valamint ennek az eszköznek a használata esetén is magunknak kellett letölteni a függőségeket, így könnyen előfordulhatott, hogy a kollégáimmal különböző verziójú függőségekkel dolgoztunk, s ez beláthatatlan következményekkel járhatott.

Manapság az Apache Maven eszközt használjuk projektjeink építéséhez, menedzseléséhez. A Maven használatával egy egyszerű xml fájlban tudjuk megadni a projekt felépítéséhez szükséges összes információt. A különböző függőségek összevadászásával sem kell többet bajlódnunk, hiszen azokat a Maven tölti le egy repository-ból az xml fájlban megadott információk alapján. A Maven Central Repository rengeteg projektet tesz könnyen elérhetővé, de ezen felül léteznek más (pl. gyártók által készített) repository-k is, sőt saját magunk is létrehozhatunk ilyet.

Telepítés

A Maven futtatásának előfeltétele, hogy gépünkre fel legyen telepítve a Java Development Toolkit (JDK). Ennek telepítése után ellenőrizzük le, hogy a rendszer JAVA\_HOME környezeti változója a JDK mappájára mutat, valamint azt is, hogy a PATH környezeti változó tartalmazza-e a $JAVA\_HOME/bin útvonalat.

Amint ez megvan, elkezdhetjük a Maven telepítését:

1. A “http://maven.apache.org/download.cgi” oldalról töltsük le az operációs rendszerünknek megfelelő Maven változatot.

2. A letöltött csomagot ki kell csomagolni egy általad kiválasztott mappába.

3. Hozzuk létre a M2\_HOME környezeti változót, s értékéül adjuk meg annak a mappának az elérési útvonalát ahova a mavent kicsomagoltuk.

- Linux: A konzolba, vagy .bashrc fájlba írjuk be a következőt: export M2\_HOME=”<maven\_mappa>”

- Windows: Nyissuk meg a “Környezeti változók” ablakot, s készítsünk új környezeti változót (M2\_HOME), értéke pedig ’<maven\_mappa>’ legyen.

4. Hozzuk létre a M2 környezeti változót, melynek értéke a Maven HOME könyvtárának bin alkönyvtára lesz.

-Linux: a konzolba, vagy .bashrc fájlba írjuk be a következőt:export M2=$M2\_HOME/bin

-Windows: a “Környezeti változók” ablakban készítsünk egy új környezeti változót M2 névvel és %M2\_HOME%\bin értékkel

5. Végül a PATH környezeti változóhoz adjuk hozzá az M2 környezeti változót

-Linux: a konzolba, vagy .bashrc fájlba írjuk be a következőt:export PATH=$M2:$PATH

-Windows: a “Környezeti változók” ablakban szerkesszük a PATH változót, s az értékének végére illesszük be a következőt: %M2%

Ellenőrzésképpen egy konzol ablakban futtassuk le az mvn –version parancsot.

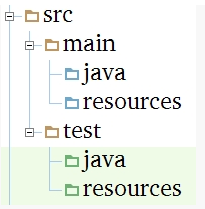
A Maven első futtatása után létrejön egy .m2 mappa, mely tartalmazni fogja a letöltött függőségeket, valamint az itt található settings.xml-ben különböző beállításokat adhatunk meg.

Egyes IDE-khez lehetséges, hogy fel kell telepíteni egy Maven plugint annak érdekében, hogy könnyen lehessen Maven projekteket létrehozni.

Első használat – Egy egyszerű maven projekt

Minden maven projekt alapja a pom.xml fájl, mely gyakorlatilag minden fontos információt tartalmaz, mint pl. a projekt neve, verziója, csomagolási formátuma, a fordítási beállítások, s itt kell megadni azt is, hogy milyen függőségei vannak a projektnek (pl. miyen jar-okat töltsön le a maven fordítás során).

A Maven projektek forrás állományai egy előre definiált könyvtár struktúrában vannak. Egy Java projekt struktúrája alapvetően a következőképpen néz ki:



A példában látható main mappában projektünk forrásállományai, míg a test mappában az egyes osztályok tesztjeit tartalmazó fájlok találhatók. A resources almappákban úgynevezett erőforrás állományok (pl. property fájlok, xml leírók, stb.), míg a java könyvtárban java forrásállományok találhatók.

Egy Hello World alkalmazás az alábbi parancs lefuttatásával hozható létre konzolból:

*mvn archetype:generate -DgroupId=hu.zerotohero -DartifactId=hello-world -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false*

Ahogy láthatjuk létrejött a projekt a fent bemutatott fájlszerkezettel, valamint a nélkülözhetetlen pom.xml-el, melynek tartalma:

<project xmlns=”http://maven.apache.org/POM/4.0.0″ xmlns:xsi=”http://www.w3.org/2001/XMLSchema-instance”

xsi:schemaLocation=”http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4\_0\_0.xsd”>

<modelVersion>4.0.0</modelVersion>

<groupId>hu.zerotohero</groupId>

<artifactId>hello-world</artifactId>

<packaging>jar</packaging>

<version>1.0-SNAPSHOT</version>

<name>hello-world</name>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>3.8.1</version>

<scope>test</scope>

</dependency>

</dependencies>

</project>

Az artifactId nem mást tartalmaz, mint a projektünk nevét, míg a groupId tag-ek között található a projektet készítő szervezet azonosítója, mely jelen esetben hu.zerotohero (mindig a szervezet dönti el, hogy mi lesz az azonosító). A packaging részben adhatjuk meg, hogy milyen típusú allományt készítsen a Maven a projektünkből. A name tag-ek között található a projekt neve, a version tag-ek közt pedig jelenlegi verziószáma. A dependencies részben tudjuk felsorolni a projekt függőségeit.

Ahogy láttuk, új Maven projekt létrehozható kézzel is, de ezt ajánlott az IDE-re bízni, amennyiben lehetséges.

A projekteket például le tudjuk fordítani konzolból az “mvn clean install” paranccsal, de az IDE-kből is lehetséges a fordítás és a futtatás is. Fordítás elött a Maven letölti a projekt függőségeit egy előre beállított mappába, majd a fordítás eredménye a projekt könyvtárában létrejött target mappában lesz megtalálható (pl. jar fájl, war fájl, attól függően, hogy mit állítottunk be). A Hello World “alkalmazást” futtatva:

*java -cp target/hello-world-1.0-SNAPSHOT.jar hu.zerotohero.App*

megkapjuk az eredményt.