Objektumorientált programozás

Az élet szép, környezetünk tele van fákkal, virágokkal, repdeső madarakkal, vidáman futkározó állatokkal.

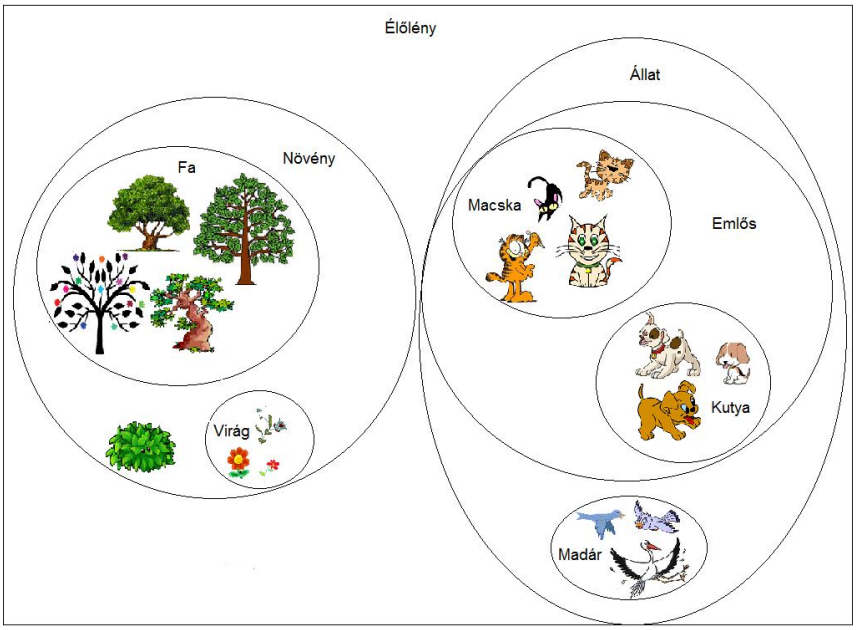
Ez a – nem művészi értékű, de idillikus – kép azt a pillanatot mutatja, amikor még nincs ott az ember. Ha ő is megérkezik, akkor jó esetben gyönyörködik benne, de egy kis idő után igényét érzi annak, hogy valakinek meséljen a látottakról. Ehhez viszont meg kell alkotnia a fa, virág, madár, kutya, macska, stb. fogalmát. De hogyan érti meg a hallgatósága, hogy mire gondol, mit láthatott, amikor elmeséli az élményeit? Csak akkor tudják elképzelni a hallottakat, ha bennük is élnek ezek a fogalmak, és maguk is „látják” a képet. Persze, nem ugyanazt fogják látni, mint a mesélő, sőt, a hallgatóság minden tagja mást és mást képzel el, de mégis megtalálják a közös hangot, mert jó esetben a fogalmaik lényege azonos: tudják, hogy a fa olyan valami, amelynek „földbe gyökerezett a lába”, a szél hatására hajlongani tud, az állatok viszont változtathatják a helyüket, a madarak repülni tudnak, a kutyák ugatni, stb.

Ha viszont le kell fényképezni egy adott fát, vagy hazahozni a rétről a család kutyáját, akkor már nem elég fogalmi szinten gondolkozni, hanem a konkrét „példánnyal” kell foglalkoznunk.

De hogyan alakulhattak ki ezek a közös fogalmak, illetve hogyan alakulhatnak ki újabbak? Úgy, hogy állandóan osztályozzuk a látottakat. A kép szereplőinél maradva először talán csak annyit veszünk észre, hogy mindegyikük él, aztán azt, hogy vannak köztük egy helyben maradó, illetve a helyüket változtató élőlények, később esetleg még további megkülönböztető jegyeket fedezünk fel, és finomítjuk a fogalmainkat. (A kisgyerek is hasonlóan fedezi fel a világot, bár ez a felfedezés kétirányú, eleinte inkább a konkrét ismeretek felől indulva jutunk el elvontabb szintig, majd a már meglévő tudásunkba kell integrálni az újdonságokat. De maga a fogalomalkotás hasonló az itt tárgyaltakhoz.)

Térjünk vissza a képen látottakhoz. Észrevettük tehát, hogy csupa *élőlény* látható rajta. Aztán megkülönböztettük a helyváltoztatásra képes és képtelen lényeket. Az előbbiek az *állat*ok, az utóbbiak a *növény*ek. De még további lényegi különbségeket is észrevehetünk: a növények között vannak sokáig élő fás szárúak (*fa*), illetve rövid életű lágy szárúak (*virág*). Az állatok egy része röpdös a levegőben (*madár*), más részük a földön szaladgál, és így tovább. Amikor azt tapasztaljuk, hogy lényegi eltérés van a vizsgált élőlények között, akkor külön osztályba soroljuk őket, ha azt észleljük, hogy bár van eltérés, de sokkal fontosabbak a közös jellemzők, akkor azonos osztályba kerülnek. Ez a fajta csoportosítási, osztályozási képesség alapvető része az emberi gondolkozásnak, és ugyanez az alapja az **objektum-orientált** gondolkozásmódnak is.

A csoportokba (osztályokba) sorolás hatására létrejöhet a következő fogalom-hierarchia, vagy más néven, osztály-hierarchia:



Ahogy látható, különböző kapcsolatok vannak az egyes csoportok között. Mindegyik benne van az élőlényeket szimbolizáló téglalapban, de vannak egymástól független csoportok, illetve vannak olyanok is, ahol az egyik tartalmazza a másikat. Például a kutyák csoportja az emlősök csoportján belülre van rajzolva, az pedig az állatok csoportján belülre. Ez a tartalmazás logikus, hiszen egy kutya egyúttal emlős is, és minden emlős az állatok csoportjába tartozik. Ezért amikor a kutya fogalmát akarjuk meghatározni, vagyis azokat a jellemzőket, amelyek alapján egy élőlényt kutyának tekintünk, akkor elég csak a specialitásokat kiemelni, anélkül, hogy az emlősökre, illetve az állatokra vonatkozó egyéb tulajdonságokat külön részletezni kellene. Az ilyen tartalmazási relációt (vagyis azt, amikor közöljük, hogy a kutya egyúttal emlős is, vagyis a kutya fogalma az emlős fogalmának kibővítése) **öröklődés**nek (esetleg származtatásnak vagy kibővítésnek) nevezzük.

De mi van az egyes csoportokon belül? Szemmel láthatóan a kezdőkép konkrét élőlényei. Vagyis az osztályozás mindig kétirányú:

Az egyik irány az **absztrakció**. Ennek során megpróbáljuk kiemelni az azonos osztályba került dolgok közös jellemzőit: megtartjuk a lényegesnek vélt tulajdonságokat, és elhagyjuk a lényegteleneket.

A másik irány: a kialakult osztályok használata, vagyis ha definiáltunk egy osztályt, akkor hogyan lehet olyan példányokat létrehozni, amelyek ehhez az osztályhoz tartoznak. (Esetünkben: ha pl. bemegyünk egy kertészetbe fát vásárolni, akkor valóban fát kapjunk.)

Az **osztály** tehát egy absztrakt fogalom (amolyan tervrajz féle), az osztályba sorolt konkrét dolgok pedig az osztály **példány**ai, vagy más szóval **objektum**ok.

Programozási szempontból azt is mondhatjuk, hogy az osztály egy összetett típust jelent, ahol mi magunk (vagy az adott programnyelv létrehozói) definiáljuk azt, hogy mit is értünk ez alatt a típus alatt, az objektumok pedig ilyen típusú változók.

Alapfogalmak:

A valós világ objektumainak kétféle jellemzője van: mindegyiknek van valamilyen állapota (valamilyen tulajdonsága), és mindegyik viselkedik valamilyen módon. Például egy kutya tulajdonsága lehet a neve, színe, fajtája; viselkedése pedig az, hogy ugat, csóválja a farkát, stb. Az osztályozás során pontosan ezeket a tulajdonságokat és viselkedést kell leírnunk, illetve meghatároznunk.

Az **objektumorientált programozás (OOP)** egy, az osztály-hierarchiára épülő programozási módszer, amely lehetővé teszi különböző bonyolult változók (objektumok) létrehozását és kezelését. Egy o**bjektum-orientált program** az egymással kapcsolatot tartó, együttműködő objektumok összessége, ahol minden objektumnak megvan a jól meghatározott feladata.

Az **osztály** egy-egy fogalom definiálására szolgál. Leírásakor egy-egy speciális típust határozunk meg abból a célból, hogy később ilyen típusú változókkal tudjunk dolgozni. Az `Osztaly` típusú változó majd `Osztaly` típusú objektumot tartalmaz. Egy osztály tulajdonképpen egy objektum „tervrajzának” vagy sémájának tekinthető.

A **példány** egy konkrét, az osztályra jellemző tulajdonságokkal és viselkedéssel rendelkező **objektum**. Mindkét elnevezés használatos (példány, objektum). Egy időben több azonos típusú objektum is lehet a memóriában, és két objektumot akkor is különbözőnek tekintünk, ha azonos tulajdonságaik vannak. (Pl. két Bodri nevű puli nyilván két különböző kutya.)

Egy program objektumai hasonlóak a valós világ objektumaihoz, vagyis nekik is vannak állapotaik (tulajdonságaik) és viselkedésük. Ezeket az állapotokat úgynevezett mezőkben (vagy adattagokban) tároljuk, a viselkedést pedig a metódusok írják le. Mivel az azonos osztályba tartozók hasonló módon viselkednek, ezért a hozzájuk tartozó metódusokat az osztályok definiálásakor határozzuk meg. Azt is, hogy milyen mezőkkel kell rendelkeznie egy-egy ilyen osztálynak (azaz ilyen típusnak), de a mezők konkrét értékét már az objektumok, azaz a konkrét példányok tartalmazzák.

De hogyan jönnek létre ezek a példányok? A **konstruktor** hozza létre őket. Ez egy speciális, visszatérési típus nélküli metódus, amelyben inicializáljuk az objektum bizonyos állapotait, és helyet foglalunk számára a memóriában. Az, hogy helyet kapnak a memóriában, azt jelenti,

hogy minden egyes példány, az összes adattagjával együtt helyet kap. Egy kivétel lehet, amikor minden egyes példányhoz azonos értékű adattag tartozik. (Például minden magyar állampolgár 18 éves korában válik választópolgárrá.) Az ilyen adatot fölösleges annyi példányban tárolni, ahány objektum van, elég csak egyszer. Ezeket, az azonos típusú objektumok által közösen használható adatokat, **statikus** adatoknak nevezzük, illetve a rájuk hivatkozó változókat statikus változóknak. Léteznek statikus metódusok is, ezeket az őket tartalmazó osztály példányosítása nélkül tudjuk meghívni.

Létrejöttük után az objektumok „önálló lények", kommunikálni tudnak egymással. Bár ennél kicsit többet jelent a kommunikáció, de első közelítésben mondhatjuk azt, hogy gyakorlatilag azt jelenti, hogy az egyik objektum meg tudja hívni a másik valamelyik metódusát.

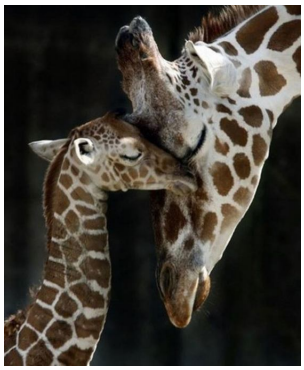
Ugyanakkor nem szabad megengednünk azt, hogy kívülről bárki belepiszkálhasson egy objektum állapotába, vagyis hogy egy objektum megváltoztathassa egy másik adattagjának értékét, illetve lekérhesse azt annak ellenére, hogy a másik esetleg titokban szeretné tartani. (Nem feltétlenül örül annak valaki, ha bárki megnézheti, mennyi pénz van a bankszámláján, és nyilván nem lehet kívülről megváltoztatni valaki születési dátumát.)

Azt az elvet, hogy egy objektumot ne lehessen kívülről nem várt módon manipulálni, az **egységbezárás** (vagy az **információ elrejtése**) elvének nevezzük. Ennek lényege, hogy csak meghatározott metódusokon keresztül módosítható az állapot. Erre mutat egy kis példát a mellékelt kép: a tanárnak nem feltétlenül kell tudnia, hogy a vizsgázó könyvekből vagy internet alapján készült fel. Őt csak az érdekli, hogy tud-e a diák – azaz, visszafordítva az OOP nyelvére, hogy elvárt módon működik-e az objektum.

Egységbe zárás – az információ elrejtése

Az, hogy elvárt módon működik (vagyis pl. az előbb említett diák jól felel), azt jelenti, hogy meg tudjuk hívni az objektum megfelelő metódusát, és az úgy működik, ahogyan kell. De ahhoz, hogy meg tudjuk hívni, a mezővel ellentétben, a metódus nem lehet rejtett. Lehetnek olyan metódusok, amelyekhez bárki hozzáférhet, de lehetnek olyanok is, amelyeket csak belső használatra szánunk, illetve olyanok is, amelyet csak bizonyos körülmények között akarunk megosztani. Azt, hogy ki érheti el a metódusokat, a **láthatóság** szabályozza.

Az egyes objektumorientált nyelvek között lehet eltérés, de az alapvető három láthatósági típus a nyilvános (**public**), rejtett (**private**) és a védett (**protected**) mód. Ezen kívül a default láthatóság, vagyis az, amikor nincs láthatósági módosító a metódusnév előtt. (Láthatósági módosítok lehetnek osztálynév és mezőnév előtt is, de ezek, illetve a láthatóság pontosabb definiálása majd a megfelelő helyen időben sorra kerül.)



Ahogy az induló példában már szó volt róla, az osztályok között kapcsolat is lehet. Az egyik leggyakrabban használt kapcsolat a már említett **öröklődés**.

A minket körülvevő világban gyakran előfordul, hogy két tárgy (élőlény, stb.) között hasonlóságot tapasztalunk. A képen látható zsiráf-gyerek is hasonlít a mamájára, sok-sok biológiai tulajdon-

ságot örököl tőle. Ugyanakkor saját tulajdonságokkal (is) rendelkező, önálló egyéniség.

Hasonló a helyzet az egymással öröklési kapcsolatban lévő osztályokkal. Programozási szempontból az egyik alapvető elvárás, hogy a kódunkban lehetőleg ne legyen kódismétlés. Többek között ezt hivatott megoldani az öröklődés. Ha egy osztály egy másik osztály minden nyilvános tulajdonságát és metódusát tartalmazza, de vagy egy kicsit bővebb annál, vagy bizonyos metódusai kicsit eltérően működnek, mint a másik megfelelő metódusa, akkor ezt az osztályt célszerű származtatni (örökíteni) a másikból, és csak az eltérő tulajdonságait, metódusait tárolni, a közösekre pedig hivatkozni. Azt az osztályt, amelyet alapul veszünk, szülő-, vagy ős-osztálynak nevezzük, azt amelyik kibővíti ezt, utód-, vagy származtatott-osztálynak.

Természetesen olyan is lehet, hogy egy osztály sok dolgot tartalmaz egy másik osztályból, de nem mindent, illetve a közös tulajdonságokon kívül vannak még saját specialitásai is. Ekkor is alkalmazható az öröklődés, csak ekkor létre kell hoznunk egy közös ős-osztályt, amelyből mindkettő öröklődhet. Ha ez a közös ősosztály valóban csak arra kell, hogy mindkét osztály tudjon örökölni tőle, de nem akarunk saját példányokat létrehozni belőle, akkor célszerű **absztrakt**, azaz nem példányosítható **osztály**ként definiálni. Ilyen lehet pl. a bevezető példa emlős osztálya, hiszen nincs egyetlen emlős példány sem, csak kutyák és macskák vannak. Egy absztrakt osztályban lehetnek absztrakt metódusok, vagyis olyanok, amelyeknek nem írjuk meg a törzsét. Esetünkben ilyen lehet például a „beszél” metódus, amelyet majd elég lesz a kutya, illetve macska osztályban kifejteni, hiszen a kutya ugatva „beszél”, a macska nyávogva.

Ennél komolyabb absztrakció is lehet, amikor csak azt soroljuk fel, hogy egyáltalán milyen metódusokat akarunk majd megvalósítani, de maga a megvalósítás hiányzik. Ezt nevezzük **interfész**nek, de ennek tárgyalására majd a megfelelő fejezetben kerül sor.

Az öröklődés kapcsán még egy fogalmat kell megemlítenünk, mégpedig a **polimorfizmus** fogalmát. A szó görög eredetű, és többalakúságot jelent. Ezt legegyszerűbben a már elkezdett példán tudjuk megvilágítani.

Mivel a kacsa nem emlős, ezért tekintsük a képen szereplő állatokat az `Allat` osztályból származtatott `Kutya`, `Macska`, `Kacsa` osztály egy-egy példányának. Az `Allat` osztályban megírt (esetleg absztrakt) `beszel()` metódust más-más módon írja felül az utód osztályok megfelelő metódusa. Ha a képen látható állat példányokat egy közös listában szeretnénk szerepeltetni, akkor kénytelenek leszünk `Allat` típusúnak deklarálni őket. Fordítási időben nem derül ki, hogy az adott lista esetében melyik `beszel()` metódust kell futtatni, futásidőben azonban – vagyis amikor kiderül, hogy melyik utód-osztályba tartozik a konkrét példány – ez egyértelművé válik.

Azt, hogy ilyen későn (vagyis csak futási időben, amikor sorra kerül az adott példány) dől el, hogy melyik metódust kell futtatni, **késői kötés**-nek nevezzük.

Összefoglalva: az objektumorientált programozás legfontosabb alapfogalmai az osztály és az objektum, legfontosabb alapelvei pedig az egységbezárás, öröklődés és polimorfizmus.

Van azonban még egy fontos alapelv, az újrahasznosíthatóság elve, amely persze nem csak az OOP programokra igaz. Vagyis az, hogy úgy írjunk programot, hogy azt ne csak egyszer, egy adott szituációban tudjuk felhasználni.

Ennek elnevezésére még angol mozaikszó is született: WORA („Write once, run anywhere”) vagy WORE („Write once, run everywhere”).

Vagyis úgy írjuk meg a programjainkat, hogy annak elemeit néhány egyszerű cserével könnyedén fel lehessen használni egy másik szoftver létrehozásakor.

Ennek eléréséhez célszerű betartani a következő elveket:

- *Modularitás elve*: Próbáljuk meg a komplex feladatot kisebb részekre bontani, mégpedig úgy, hogy egy-egy rész egy-egy kisebb, önálló feladatot oldjon meg.
- *Fokozatos fejlesztés elve*: A fejlesztés elején előbb csak egy egyszerű implementációt hozzunk létre, teszteljük, majd bővítsük tovább, de úgy, hogy minden bővítési lépést tesztelünk.
- *Az adatreprezentáció rugalmasságának elve*: Az elv lényege, hogy bármikor könnyedén ki tudjuk cserélni a kód futásához használt adatokat. Ennek legelemibb módja, hogy **SOHA** nem égetünk be adatokat. (De nem csak ezt jelenti az elv, eleve rugalmasan kell kezelni bármiféle adatot.)