Info 3

## Az objektumorientált paradigma alapfogalmai:

* + **Egységbezárás:** Az adatok és a metódusok osztályba való összezárását jelenti. Tulajdonképpen az objektum egységbezárja az állapotot (adattagok értékei) a viselkedésmóddal (műveletekkel). Következmény: az objektum állapotát csak a műveletein keresztül módosíthatjuk.
  + **Absztrakció**: Elvonatkoztatás. Segítségével privát implementációkat rejthetünk el egy nyilvános interfész mögé. Példa: java.util csomagban List interfész és az interfészt implementáló ArrayList, illetve LinkedList osztályok. Az absztrakció lehetővé teszi, hogy mindkét osztály példányait ugyanazon List interfész műveletein keresztül kezeljük.
  + **Öröklődés**: egy [objektumot](https://hu.wikipedia.org/wiki/Objektum) vagy egy [osztályt](https://hu.wikipedia.org/w/index.php?title=Oszt%C3%A1ly_(sz%C3%A1m%C3%ADt%C3%A1stechnika)&action=edit&redlink=1) alapjául választunk egy másik objektumnak (prototípus-alapú öröklődés) vagy osztálynak (osztály-alapú öröklődés), megtartva a hasonló implementációt. Más néven egy új osztályt (alosztályt) származtatunk egy már létező szuper- vagy alaposztályból, amiket aztán osztályok hierarchiájává formázunk. A legtöbb osztály-alapú objektum-orientált nyelvben egy öröklődésen keresztül létrehozott objektum a „gyermekobjektum” megkapja az ő „szülőobjektumának” minden tulajdonságát és viselkedését, leszámítva a konstruktorokat, destruktorokat, túlterhelt operátorait és barát függvényeit. Az öröklődés lehetővé teszi, hogy már létező osztályokra hozzunk létre újakat,[[1]](https://hu.wikipedia.org/wiki/%C3%96r%C3%B6kl%C5%91d%C3%A9s_(objektumorient%C3%A1lt_programoz%C3%A1s)#cite_note-1) ezzel annak egy új implementációját megalkotva úgy, hogy közben a viselkedését megtartja (interfész implementálása). Ez hasznos, mivel újrahasznosításra kerül a kód, emellett publikus osztályok és interfészek segítségével szabadon bővíthető marad az eredeti szoftver. Objektumok vagy osztályok közötti öröklődés által kialakult kapcsolatok jól szemléltethetők [irányított gráfokkal](https://hu.wikipedia.org/wiki/Gr%C3%A1felm%C3%A9leti_fogalomt%C3%A1r#Ir%C3%A1ny%C3%ADtott_gr%C3%A1fok).
  + **Polimorfizmus**: Többalakúság. Egy típuselméleti fogalom, amely szerint egy ősosztály típusú változó hivatkozhat ugyanazon közös ősosztályból származó (vagy ugyanazon interfészt megvalósító) osztályok példányaira. A polimorfizmus lehet statikus és dinamikus. (a) statikus polimorfizmus: metódusok túlterhelése, függvénysablonok, osztálysablonok. Satikus, fordításidejű kötés. (b) dinamikus polimorfizmus: metódusok felülírása. Dinamikus, futásidejű kötés.
  + **Programozási paradigma:** Programozási mód. Alapvetően a program felépítésére használt eszközkészletet jelenti, vagyis milyen egységek képezik a program alkotóelemeit. (moduláris programozás, objektumorientált programozás, általánosított programozás, aspektusorientált programozás stb.)
  + **Absztrakt adattípus:** Az adattípus leírásának legmagasabb szintje, amelyben az adattípust úgy specifikáljuk, hogy az adatok ábrázolására és a műveletek implementációjára semmilyen előírást nem adunk. Lehetőleg matematikai fogalmakat használva írjuk le az adattípust (halmazok és ezeken értelmezett műveletek).
  + **Objektumorientált programozás:** Olyan programozási paradigma, amely a programokat objektumokból építi fel. A program működése tulajdonképpen objektumok kommunikációját jelenti. Legfontosabb alapelvei: egységbezárás, öröklődés, polimorfizmus.
  + **Osztály:** Az osztály egy felhasználói típus, amelynek alapján példányok (objektumok) hozhatók létre. Az osztály alapvetően adat és metódus (művelet) definíciókat tartalmaz.
  + **Objektum:** Információt (adatokat) tárol és kérésre műveleteket végez. Van állapota, viselkedése és futásidőben azonosítható.
  + **Üzenet:** Objektumhoz továbbított kérés. Válaszként az objektum végrehajtja a kért műveletet.
  + **Információ elrejtése:** Az objektum elrejti az adatait és bizonyos műveleteit. Ez azt jelenti, hogy nem tudjuk pontosan, hogy egy objektumban hogyan vannak az adatok ábrázolva, sőt a műveletek implementációit sem ismerjük. Az információk elrejtése az objektum biztonságát szolgálja, amelyeket csak a ellenőrzött műveleteken keresztül érhetünk el.
  + **Származtatás:** Olyan osztályok között értelmezett viszony, amely segítségével egy általánosabb típusból (ősosztály) egy sajátosabb típust tudunk létrehozni (utódosztály). Az utódosztály adatokat és műveleteket (viselkedésmódot) örököl, kiegészíti ezeket saját adatokkal és műveletekkel, illetve felülírhat bizonyos műveleteket. A kód újrafelhasználásának egyik módja. Megkülönböztetünk egyszeres és többszörös örökítést.
  + **Metódusok túlterhelése:** Több azonos nevű, különböző szignatúrájú függvény. A függvényhívás aktuális paraméterei meghatározzák, hogy melyik függvény fog meghívódni. Ezt már a fordításidőben eldől (statikus, fordításidejű kötés).
  + **Metódusok felülírása:** Egy osztályhierarchián belül az utódosztály újradefiniálja az ősosztály metódusát. (azonos név, azonos szignatúra). Ha ősosztály típusú mutatón vagy referencián keresztül érjük el az osztályhierarchia példányait és ezen keresztül meghívjuk a felülírt metódust, akkor futási időben dől el, hogy pontosan melyik metódus kerül meghívásra. (dinamikus, futásidejű kötés).
  + **Absztrakt osztály:** Olyan osztály, amelynek van legalább egy absztrakt művelete. Felületet határoz meg és nem lehet példányosítani. Absztrakt osztály az absztrakt műveleteinek implementálását az utódosztályokra bízza.
  + **Interfész:** Viselkedésmódot definiál. Gyakorlatilag egy művelethalmaz deklarációját jelenti. Ha egy osztály implementál egy adott interfészt, akkor példányai az interfészben meghatározott viselkedéssel fognak rendelkezni. Csak konstans adattagokat tartalmazhat és minden tagja nyilvános.
  + **Konstruktor:** Az a művelet, amely inicializálja az objektumot. Automatikusan hívódik. Egy osztályhoz annyiféle konstruktort készítünk, ahányféle képpen lehetővé tesszük a példányok inicializálását .
  + **Destruktor:** A konstruktorral ellentétes művelet, általában a konstruktorban lekötött erőforrásokat szabadítja fel. Az objektum megsemmisítése előtt hajtódik végre és automatikusan hívódik.
  + **Függőség:** Komponensek közötti függőség mértéke. Megkülönböztetünk laza és szoros csatolású rendszereket. A laza csatolású rendszerek esetében, a rendszer valamely komponensének változása nem vonja maga után a többi komponenes módosítását.
  + **Aggregáció:** Rész-egész kapcsolat. A részek alkotják az egészet. Például az autó motor, váz és kerekek aggregációja. A részek túlélhetik az egészet.
  + **Kompozíció:** Sajátos aggregáció, amikor a rész szorosan hozzátartozik az egészhez. A részek nem élik túl az egészet. Például az emberi agy szorosan hozzátartozik az emberhez
  + **Delegálás:** Implementációs mechanizmus, melynek során egy objektum továbbítja (delegálja) a kérést egy másik objektum fele. A delegált objektum fogja feldolgozni a kérést. Példa: Java eseménykezelés (az eseményfigyelő fele továbbítódik a kérés)
  + **Iterátor:** Olyan típus, amely pozíciót határoz meg egy halmazban (tároló, adatfolyam). Műveletein keresztül biztosítja a tároló bejárását, azaz a tárolt elemek egymás utáni feldolgozását.
  + **Algoritmus:** Általánosan megvalósított függvény, amely minimális követelményt támaszt azon adatokkal szemben, amelyeken végrehajtódik.
  + **Függvényobjektum:** Függvényként viselkedő objektum. Az az előnye a függvénymutatóhoz képest, hogy mint objektum, állapotot is tárol, nemcsak függvényként viselkedik. Megvalósítás: olyan osztállyal, amelyben értelmezzük a függvényhívás operátort. Ezen kívül az osztály tartalmazhat adattagokat és más segédműveleteket is.
  + **Függvénysablon:** Típusparaméterekkel ellátott függvény, amely egy függvénycsaládot határoz meg.
  + **Osztálysablon:** Típusparaméterekkel ellátott osztály, amely egy típuscsaládot határoz meg
  + **Privát örökítés:** Hozzáférés-szűkítő hatása van. Az ősosztálytól átvett adat és metódustagok privát tagokká alakulnak, ezáltal az utódosztály már nem biztosítja az ősosztály által meghatározott viselkedésmódot. Az utódosztály az ősosztály implementációját örökli és nem annak interfészét.
* **Osztály**: Az osztály egy olyan felhasználói típus, ami adattagok mellett tagfüggvényeket is tartalmazhat. Egy olyan alaprajz, sablon, utasítások halmaza, ami meghatározza a belőlük készült objektumokat. A

## **Objektum**: Egy változó, ami egy osztályból lett példányosítva.

## **Példányosítás**: Egy objektumosztályból konkrét objektum készítése. A konstruktor általában azért felelős, hogy az objektum [adattagjait](https://wiki.prog.hu/index.php?title=Adattag&action=edit&redlink=1) az osztály szempontjából értelmes, érvényes állapotba hozza, bizonyos nyelvekben pedig az objektum által használt memóriaterületet lefoglalása is feladata. A már nem használt objektum által lefoglalt memória és egyéb erőforrások felszabadításáért a [destruktor](https://wiki.prog.hu/wiki/Destruktor) felelős.

## **Öröklődés**: Az öröklődés során beszélhetünk Ős (Base), illetve Származtatott (derived) osztályokról. Az öröklődés az osztályok továbbfejlesztését jelenti, ennek során a származtatott osztály örökli az ős osztály attribútumait, metódusait, ezek mellett újakkal egészítheti ki vagy megváltozhatja a meglévőket bizonyos szabályok szerint. Egy ősből több származtatott osztályt is készíthetünk. Egy származtatott osztálynak viszont programozási nyelvtől függően változó őse lehet. Java-ban egy (öröklődési fa), C++-ban több (öröklődési gráf). A legerősebb kapcsolat két osztály között az öröklődés. Kódújrahasználhatóság szempontjából az ősosztály nagyon hasznos. Modellezésnél x EGY y (is-a) kapcsolattal jelöljük. Bővíthetőségre nyitott, többféle viselkedés definiálható, ugyanazon a felelősségen belül. Tiltható az öröklődés.

## **Osztályhierarchia**:

## **Polimorfizmus**: Ugyan az a név különböző forma. A polimorfizmus sokoldalúságot jelent. Ez több formában jöhet elő. Legtöbbször a leszármaztatott osztály őséből származó függvények felülírásakor használjuk. Szükséges egy fogalom bevezetése. Függvény szignatúra (prototípus): Programozási nyelvenként változik, de a függvény neve és paraméter listája beazonosít egy függvényt. (cppben a visszatérési érték is hozzájön ,javaban nem)

* + Metódus túlterhelés(overload): Függvény paraméterlistáját (vagy még a visszatérési értékét is) megváltoztatva, de nevét megtartva átdefiniálhatjuk a függvényünket. Pl két integer összeadását, ugyan azzal a függvénynévvel két double-re íratjuk át.
  + Generikus programozás:
  + Metódus felülírás(override): Egy metódus felülírásakor megmarad a szignatúrája, viszont ugyanúgy felüldefináljuk a metódus működését. Pl.: Ősosztályban implementált metódusokat a származtatáskor felüldefiniáljuk, hogy a származtatott osztályunk viselkedését tudjuk implementálni. Láthatósága a metódusnak csak szűkebb lehet. Tiltható a felüldefiniálás.

## **A bezárási eszközrendszer:**

Az absztrakt adattípus az attribútumainak és metódusainak láthatóságát szabályozhatja, ezt valósítja meg a bezárási eszközrendszer. Általában 3 szintű bezárás van, mezők és metódusok esetében:

* + **Public:** minden osztály számára látható
  + **Private:** az osztályon kívülről nem látható. Az eszközöket csak az adott osztály láthatja.
  + **Protected:** az osztályon kívülről nem látható, private-ként viselkedik. Az osztályon belül és annak leszármazottjai számára látható.

## **Absztraktosztályok és interfészek:** Mindkettő absztrakciót használ, elrejtik a belső megvalósítást.

* + java

## **Interfészek:** Az interfész egy teljesen absztrakt osztály, minden függvénye absztrakt és publikus. Az összes változója konstans, publikus és konstans. Nem tartalmazhatnak megvalósítást, csak metódus prototípusokat. Egy interfész több interfészt is kiterjeszthet, de normális, absztrakt osztályokat nem. Implements kulcsszó.

* + - **Absztraktosztály**: Az absztrakt osztály tartalmazhat absztrakt, illetve nem absztrakt metódusokat, amik lehetnek publikusak vagy védettek. Tartalmazhatnak megvalósítást is. Változói lehetnek konstansok, statikusak illetve ezek ellenkezői is bármilyen láthatósággal. Egy absztrakt osztály csak osztályt terjeszthet ki, legyen az absztrakt vagy nem.
  + Cpp
    - Csak absztrakt osztályok vannak, amik virtuális függvényeket tartalmaznak. Ezeket az osztályokat nem lehet példányosítani és nem lehetnek nem statikus adattagjai. Nem lehetnek visszatérési típusok, paraméter típusok. Pointerek és referenciákat lehet deklarálni rájuk.

## **Típustagok:**

* **Metódus túlterhelés:**
  + Egy osztályon belül lehet több ugyanolyan nevű metódus is, ha a paraméterlistájuk különbözik – ezt hívjuk túlterhelésnek (overloading).
  + A fordító a paraméterlista szerint tesz különbséget az azonos nevű metódusok között.
  + Pl: static double osszeg( double x, double y ){ return x + y; } static double osszeg( int x, double y ){ return x + y; }
  + Nem csak static metódusoknál!
* **Bezárási eszközrendszer:** Az absztrakt adattípus (osztály) az attribútumainak és metódusainak láthatóságát szabályozhatja, ezt valósítja meg a bezárási eszközrendszer. Általában 3 szintű bezárás van:
  + **- privát láthatóság**: kívülről nem látható. Az eszközöket csak az adott osztály látja. Jelölése: - A hipermarket példában az Élelmiszer osztályból készített példányok nettó\_ár attribútumát csak maguk az Élelmiszer osztályból példányosított objektumok érhetik el. A műszaki\_cikk osztályból készített objektumok nem látják az Élelmiszer osztály példányainak nettó\_ár attribútumait, mivel azok privátak. Ahhoz az attribútumhoz csak a getNettó() metóduson keresztül férhetnek hozzá.
  + **- védett láthatóság:** az osztály és a leszármazottjai férhetnek hozzá az eszközökhöz. Jelölése: #
  + **- publikus láthatóság:** minden osztály láthatja az osztály eszközeit. A kliens osztályok is. Jelölése: +
* **Absztrakt osztályok:** Az absztrakt osztály legfőbb feladata, hogy olyan általános, absztrakt megoldásokat tartalmazzon, amik az objektum saját adatain, állapotain dolgoznak. Ezen a ponton viszont nem mindig lehet a problémát teljes egészében megoldani. Maradnak még olyan részek, amiket rá kell bízni a leszármazott osztályokra. Az, hogy az absztrakt osztályban lehetnek olyan metódusok, amiket nem fejtesz ki, ennek következménye. Definiálod a metódust annak paramétereivel, ezzel bebiztosítva az osztályt, hogy a metódus majd létezni fog, amikor meghívnád. Ennek az absztrakt metódusnak viszont nem kell publikusnak lennie, de nem lehet privát. Vagy publikus (public) vagy védett (protected), hiszen a privát metódusokat a leszármazott sem éri el. Tehát egy absztrakt metódus vagy csak azért készül, hogy az absztrakt ős hivatkozhasson rá, ezzel áthárítva a megvalósítás felelősségét a gyermek osztályra, vagy azért, hogy a funkció kívülről is elérhető legyen az objektumon. Előfordulhat, hogy az ősosztálynak nincs szüksége a metódusra. Ekkor viszont, főleg több metódus esetén, érdemes megfontolni az interfész alkalmazását, mivel gyakorlatilag semmi sem fogja kötni az ősosztályhoz a funkciót, ami ezek szerint azért lett nyilvános, mert szükség van a nyilvánosságára és nem csak egy extra lehetőség a nyilvánosság számára. Érdekesség, hogy statikus metódus is lehet absztrakt. Felteheted a kérdést, hogy ebben mégis mi az érdekes. Nos, sem a Java, sem a C#, de még a C++ sem támogatja az absztrakt és a statikus jelzőt egyszerre. Statikus híváshoz ismerni kellene az osztálynak a pontos nevét és nem csak egy őstípust. Ehhez vagy dinamikusan kellene tudni összeállítani az osztály és metódus nevét, amit majd valami, valahol, valahogyan meghív, vagy az ősnek kellene tudnia hivatkozni az utolsó leszármazottra, úgy, ahogy a szülőre is tud. Ez nem olyan triviális. Már ahol egyáltalán lehetséges. PHP-ben viszont az 5.3-as verzió óta a "static" kulcsszó használható a "self" helyett is, ami így nem az aktuális osztályra, hanem a leszármazottra hivatkozik. Ezen kívül dinamikusan is könnyedén előállítható egy meghívandó metódus. Persze az, hogy valamire van lehetőség, még nem jelenti azt, hogy tanácsos is élni vele.

Absztrakt osztály és interfész közötti különbségek

* az absztrakt osztály hasonló az interfészhez, de megadhat részleges implementációt, amit a származtatott osztályok egészíthetnek ki
* ha ansztrakt osztályunk csak absztrakt metódusokat tartalmaz, akkor azt inkább interfészként kell deklarálni
* az interfészek biztosítják a Java nyelvben a többszörös öröklődést; egy osztály több interfészt is implementálhat egyszerre, függetlenül attól, hogy azok mennyire állnak kapcsolatban egymással (pl. Serializable, Comparable, Cloneable stb.)
* az absztrakt osztály leszármazott osztályai közös implementációt is tartalmaznak (az absztrakt osztályban nem absztraktként definiált metódusok), de egyedi viselkedést is megadnak (ezek az absztrakt metódusok)
* absztrakt osztály tartalmazhat nem *static* és *final* mezőket is
* interfészekben minden metódus implicit absztraktként van deklarálva

Absztrakt osztály mint interfész implementáció

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 metódus meg sem jelenik, mint absztrakt metódus az absztrakt osztályban. Ezeket majd a származtatott osztályokban kell csak implementálni.

* **Típustagok:** tulajdonságok, metódusok, konstruktorok, mezők