1. OOP története, alapfogalmai és alapelvei.

Az OOP nyelvek és a procedurális nyelvek viszonya

- A procedurális nyelvek esetében az adatokat egyszerű vagy összetett rekordokban tároljuk, ám a velük működő metódusok külön programelemekként lebegnek a kódban.
- Az OOP nyelvek legjelentősebb újítása, hogy rekordok helyett a hozzájuk nagyon hasonló objektumokat használja, melyek egyaránt magukba foglalhatnak adatokat és funkciókat, azaz rekordokat és metódusokat. Így a rekordok egy egységgé válnak az őket használó metódusokkal. Ez az egységbezárás.
- Az egységbezárás a C# nyelvben már a tanulmányok kezdete óta jelen van a különféle meghívott metódusok formájában (Pl.: a Console.WriteLine() a Console osztályban található WriteLine() metódus, vagy a Math.PI a Math osztályban megbúvó PI rekord, ami a π nagy pontosságú változatát adja vissza az őt meghívó programrésznek).

Az OOP története

- Alapjait a ’60-as években fektették le. 1969-ben Alan Kay fogalmazta meg az alapelveit PhD dolgozatában. Az objektumok fogalma először az AI - fejlesztésben és a LIPS programozási nyelvben kezdett kibontakozni, de konkrét eszközként a Simula 67 programozási nyelvben jelent meg, magával hozva többek között az osztály az öröklődés, a virtuális metódusok és a garbage collection fogalmát.

OOP alapfogalmak

- **Osztály**
  - Összetett adattípust és az ezeket az adatokat feldolgozó metódusokat leíró típusdefiníció.
- **Mező**
  - Az osztályban szereplő adattároló, változó.
- **Metódus**
  - Az osztályban található eljárás vagy függvény, ami az osztály mezőivel végez műveletet.
- **Property**
  - Különleges adatfajta, leginkább a mezőkhöz hasonlít. A Getterrel és Setterrel részletesen szabályozható, hogy milyen adatot küld vagy fogad el.
- **Adatrejtés**
  - Az OOP egyik fontos elve, hogy az egyes objektumoknak csak a program működéséhez szükséges minimum számú információt kell kifelé adniuk. Ez az osztályon belül a mezők, metódusok vagy konstruktorok elé írt Private (csak a saját osztályán belül látható), Protected (a saját osztályán és annak gyerekosztályain belül látható), illetve Public (bárki számára látható) kulcsszóval tehető meg.
- **Konstruktor**
  - Speciális metódus, mely a példányosításkor fut le. Jellemzően inicializálási feladatokra, alaphelyzetbe állításra, például a paraméterek feldolgozására használják.
  - Egy osztályban lehet több konstruktor is, amiket a fordító a paramétereik alapján különböztet meg.
  - Az osztálynak mindig van legalább egy konstruktora. Ha min nem hozunk létre, akkor fordításkor létrejön egy 0 paraméteres, üres törzsű konstruktor. (Ha viszont manuálisan létrehozunk egyet, akkor az automatikus nem jön létre!)
  - Szintaktika: a metódus neve meg kell egyezzen az őt tároló osztály nevével!
- **Példányosítás**
  - Az osztály önmagában csak egy „terv”, egy séma, mely a példányosításkor („new” kulcsszó!) kap adatokat, értelmet, egyéniséget.

- **Objektum**
  - Az adott osztály egy példánya.
    - „Auto” osztály:
      - Gyártó;
      - Rendszám;
      - Szín;
    - „Auto” példány (Auto x = new Auto)
      - Volkswagen;
      - ABC – 123;
      - Sárga;
- **Destruktor**
  - Speciális metódus, memóriafelszabadítást végez (C#-ban legtöbbször nincs rá szükség a Garbage Collector miatt, C++-ban viszont gyakran!)

Az OOP három alapelve és azok jelentősége.

- **Egységebezárás** (encapsulation)
  - A mezőknek és a velük dolgozó metódusoknak egy egységet kell alkotniuk, melyeket védeni kell a „külvilágtól” az adatrejtéssel.
    - Adatrejtés
      - Az OOP egyik fontos része, hogy az egyes objektumoknak csak a program működéséhez szükséges minimum számú információt kell kifelé adniuk. Ez az osztályon belül a mezők, metódusok vagy konstruktorok elé írt Private (csak a saját osztályán belül látható), Protected (a saját osztályán és annak gyerekosztályain belül látható), illetve Public (bárki számára látható) kulcsszóval tehető meg.
  - Az érkező és elküldött adatokat minden esetben ellenőrizni kell (erre valók a propertyk).
- **Öröklődés** (inheritance)
  - Egy általánosan definiált osztályból legyen lehetőség létrehozni specializált osztályokat, melyek tartalmazzák, egyes esetekben felülírják, és mindenek előtt kibővítik az ősük mezőinek és metódusainak listáját. Az így létrejött gyerekosztály az őse minden tulajdonságával rendelkezik. Kevesebb nem lehet az ősénél!
- **Sokalakúság, polimorfizmus** (polymorphism)
  - A gyerekosztályok által felüldefiniált metódusokat a rendszer automatikusan választja a szülőosztályban lévők helyett.
  - Egy Osztálynak több változata is lehet, a rendszer pedig automatikusan el tudja dönteni, hogy melyik változatban definiált metódust alkalmazza.

2. Adatrejtés. Property. Konténerosztályok.

A mező és metódus használata.

- A mezők létrehozásakor törekednünk kell arra, hogy a működéséhez szükséges legkevesebb memóriát foglalja le.
- A mezők lehetőleg ne legyenek semmilyen módon elérhetőek a külvilág számára, a bennük tárolt adatot metódusokon keresztül, ellenőrzötten lehessen lekérdezni vagy módosítani!
- Szabályok:
  - A mező legyep private, vagy protected!
  - A mező kizárólag egy metóduson (vagy egy propertyn) keresztül kaphat kívülről értéket!
  - A metódus ellenőrizze a változtatásokat és kezelje a kivételeket!
- Példányszintű és osztályszintű mezők:

Példányszintű mezők	Osztályszintű mezők
	Létrehozásuk a „static” kulcsszóval történik
Definiálásuk az osztályban, a metódusokon kívül történik.	
Védelmi szinttel rendelkeznek. Ha egy mező védelmi szintjét nem adjuk meg, az automatikusan „private” lesz.	
Típussal rendelkeznek. Ez lehet alapértelmezett, de a programozó által generált.	
Elnevezésükkor a változó névadási szabályokat használjuk.	
Kezdőértéküket a konstruktor állítja be.	
Értékük a példányhoz kötődik és minden példánynál más. Elérésük: Példánynév.Változónév	Értékük az osztályhoz kötődik és minden példánynál egységes. Ha az egyikben megváltozik, az a többire is kihat. Elérésük: Osztálynév.Változónév
Élettartamuk dinamikus, a példány élettartamával egyezik meg.	Élettartamuk statikus, az első hivatkozástól a program futásának végéig a memóriában maradnak.

- Példányszintű és osztályszintű metódusok:

Példányszintű metódusok	Osztályszintű metódusok
	Létrehozásuk a „static” kulcsszóval történik
Védelmi szinttel rendelkeznek. Ha egy metódus védelmi szintjét nem adjuk meg, az automatikusan „private” lesz.	
Fel tudja használni a saját osztályának példány- és osztályszintű mezőit és más példány és osztályszintű metódusait.	Kizárólag az osztályának osztályszintű metódusaira és mezőire hivatkozhatunk. Legtöbbször akkor használjuk, ha az osztályt nem kell példányosítani, vagy a metódus végrehajtásához nincs szükség példányazonosításra.
Saját példányra a „this” kulcsszóval hivatkozunk.	
Elérésük: Példánynév.Metódusnév()	Elérésük: Osztálynév.Metódusnév()

Védelmi szintek, azok kapcsolata a hatáskör fogalmával. Az adatrejtés jelentősége.

A mezők és metódusok védelmi szintjei meghatározzák a hatáskörüket is.

A C# védelmi szintjei:

- **Private**
  - A mezők és metódusok alapértelmezett védelmi szintje.
  - Kizárólag az adott osztályon belül érhető el.
- **Protected**
  - Az adott osztályban és annak gyermekosztályaiban érhető el.
- **Public**
  - A program bármely részéről elérhető.
- **Internal**
  - Az adott szerelvényen belül elérhető (A szerelvény az osztályok csoportosításának egy módja, a fordításkor több tárgykód készül. Általában egy .exe és DLL-ek (Dinamicly linked Library, ez maga a szerelvény vagy Assembly(egység)). A DLL-eket hozzá kell linkelni a programhoz, ekkor elérhetővé válnak a benne leírt osztályok.
- **Internal protected**
  - Az adott szerelvényből és az ő leszármazottjaiból elérhető.

Az adatrejtés is része az Egységbezárás alapelvnek. Az OOP egyik fontos elve, hogy az egyes objektumoknak csak a program működéséhez szükséges minimum számú információt kell kifelé adniuk. Mivel minden osztály a saját adatainak épségéért és ellenőrzéséért felel, nem engedhetjük meg, hogy kívülről ellenőrizetlenül módosíthatóak legyenek az adataik.

Property. Elrejtett mező elérése property-n vagy metóduson keresztül.

A property fontos eszköze az adatok védelmének. Az osztályt kívülről szemlélő számára egy egyszerű mezőnek tűnik, ám bemenő és kiküldött adatait részletesen lehet szűrni és ellenőrizni a „getter”-el és „setter”-el.

- **Getter:**
  - Kimenő adatot szabályozó elem. Kötelező része egy „Return”, mely az előkészített adatot dobja vissza a meghívás helyére.
- **Setter:**
  - A beérkező adatot szabályozó elem. A „value” kulcsszóval tudunk arra az értékre hivatkozni, amit kívül a property értékül kapott.

Előny:

• Ellenőrizhető az adatváltozás, komplex hibakezelés

• Tisztább kód

Hátrány:

• A sebessége megtévesztő lehet. A programozó egyszerű mezőnek látja, nem feltétlenül tudja, hogy milyen bonyolult kód van mögötte.

A vizsgáló ágak külön-külön is kaphatnak védelmi szintet, alapértelmezetten mindkettő „public”. Ha a property privát, a vizsgáló ágai nem kaphatnak publikus elérést, a fordított eset viszont lehetséges.

A property mezőként funkcionál, ha a vizsgáló ágainak nem adunk törzset (Származtatott property). Ellenkező esetben magában nem tárol adatot, egyfajta „kapuvá” alakul, melyhez tartozik egy privát mező (ennek a neve megegyezik a property nevével, csak azzal ellentétben kisbetűvel kezdődik).

```
public int MyProperty { get; set; }
```

```
public int MyProperty { get; private set; }
```

```
private int myProperty;
       public int MyProperty {
            get
            {
                return myProperty;
            }
            set
            {
                value = myProperty;
            }
        }
```

Konténerosztályok használata.

A konténerosztály egy olyan osztály, mely magába foglalja az adott objektumokat, valamint olyan metódusokat tartalmaz, melyek az objektumokat kívülről kezelik (például összehasonlítják, rendezik).

- A konténernek ugyanúgy be kell tartani az OOP alapelveit, melyek közül az egységbezárás a legkritikusabb.
- Törekedni kell arra, hogy ne legyen szükség a programban a konténer elemeinek közvetlen lekérdezésére, de ha ezt nem tudjuk elkerülni, akkor biztosítsuk a megfelelő védelmet. Mivel a konténer tárolója legtöbbször egy lista, a lista pedig referenciaváltozó, ezért egy ideiglenes listába gyűjtve adjuk ki az elemeinek Icloneable által kreált klónjait. Így a lista és az elemek is különválnak az eredetijüktől a memóriában, esetleges változtatásuk nem hat ki a konténer elemeire.
- A konténer elemeivel végzett szűrő, számoló, rendező műveleteket a konténerbe kell implementálni! Ezek metódusokon vagy propertyken keresztül, biztonságosan adjanak ki eredményt! Ugyanígy közvetetten kell megoldani az elemek hozzáadását és törlését is!

Indexelők.

- Az indexer egy speciális property, mely az adott osztály példányait indexelhetővé teszi, hasonlóan egy tömbhöz vagy listához.
- Szabályai:
  - Védelmi szinttel rendelkezik
  - Visszatérési típussal rendelkezik
  - A neve kötelezően „this”
  - Formális paraméterlistával rendelkezik, melyet [] között kell megadni!

`public Klon this[int index]`
`{`
`get`
`{`
`foreach (var item in this.KlonLista)`
`{`
`if (index == item.Azonosito)`
`{`
`return item;`
`}`
`}`
`throw new Exception("Az id nem található!");`
`}`
`}`
	A formális paraméterlista bármilyen típust bekérhet.
	A Foreach végigmegy a konténer elemein és visszaad egy elemet, amennyiben annak megadott mezője (jelen esetben az „Azonosito”) megegyezik az „index”-el
	Elhelyezhető benne hibakezelés is.

3. Példány- és osztályszint. Névterek.

Példány- és osztályszintű mezők, élettartami kérdések.

- A mezők létrehozásakor törekednünk kell arra, hogy a működéséhez szükséges legkevesebb memóriát foglalja le.
- Példányszintű és osztályszintű mezők:

Példányszintű mezők	Osztályszintű mezők
	Létrehozásuk a „static” kulcsszóval történik
Definiálásuk az osztályban, a metódusokon kívül történik.	
Védelmi szinttel rendelkeznek. Ha egy mező védelmi szintjét nem adjuk meg, az automatikusan „private” lesz.	
Típussal rendelkeznek. Ez lehet alapértelmezett, de a programozó által generált.	
Elnevezésükkor a változó névadási szabályokat használjuk.	
Kezdőértéküket a konstruktor állítja be.	
Értékük a példányhoz kötődik és minden példánynál más. Elérésük: Példánynév.Változónév	Értékük az osztályhoz kötődik és minden példánynál egységes. Ha az egyikben megváltozik, az a többire is kihat. Elérésük: Osztálynév.Változónév
Élettartamuk dinamikus, a példány élettartamával egyezik meg.	Élettartamuk statikus, az első hivatkozástól a program futásának végéig a memóriában maradnak.

Konstansok.

Működésük továbbra is változatlan:

- Egy változó a „const” kulcsszóval kiegészítve
- Kötelező deklaráláskor az inicializálása
- Értékét nem lehet megváltoztatni

A konstansok osztályszintű adatok, automatikusan statikusak lesznek!

Példány- és osztályszintű metódusok és property-k.

- A metódusok és propertyk működése ebből a szempontból nagyon hasonló.
- A propertyk vizsgáló ágai külön-külön is kaphatnak védelmi szintet, alapértelmezetten mindkettő „public”. Ha a property privát, a vizsgáló ágai nem kaphatnak publikus elérést, a fordított eset viszont lehetséges.

Példányszintű metódusok/propertyk	Osztályszintű metódusok/propertyk
	Létrehozásuk a „static” kulcsszóval történik
Védelmi szinttel rendelkeznek. Ha egy metódus védelmi szintjét nem adjuk meg, az automatikusan „private” lesz.	
Fel tudja használni a saját osztályának példány-	Kizárólag az osztályának osztályszintű

és osztályszintű mezőit és más példány és osztályszintű metódusait.	metódusaira és mezőire hivatkozhatunk. Legtöbbször akkor használjuk, ha az osztályt nem kell példányosítani, vagy a metódus végrehajtásához nincs szükség példányazonosításra.
Saját példányra a „this” kulcsszóval hivatkozunk.	
Elérésük: Példánynév.Metódusnév()	Elérésük: Osztálynév.Metódusnév()

Névterek fogalma, hierarchiája és használata.

• A névtér az osztályok körüli közeg. Egy névtéren belül minden osztálynak egyedi nevet kell adni, de különböző névterekben van lehetőség azonos nevű osztályokat létrehozni.
• A BCL az osztályokat komplex névtérrendszerben tárolja.
• Ha saját névteret használunk, akkora benne szereplő osztályokra a „Névtér.Osztálynév” úttal hivatkozunk.
• A névtereket lehet egymásban is létrehozni, így növelhetjük a nagy programok kódjának követhetőségét. ilyenkor a belső osztályokra „Névtér.Névtér2. … .NévtérN.Osztálynév”
• Beépített névterek pl.: System; System.Windows.Media. …
• Egy névtér közvetlenül tartalmazhatja:
  ◦ Class
  ◦ Struct
  ◦ Enum
  ◦ Interface
  ◦ Delegate
• A névtér használata opcionális, a névtér nélküli osztályok is működőképesek. A névterek nélkül létrehozott osztályok egy globális névtérben kapnak helyet.
• Ha a kód elején „using” kulcsszóval megadjuk a használt névtereket, akkor az abban szereplő osztályok használatához nem kell mindig a teljes hivatkozást kiírnunk. (System.IO.StreamReader)

Ismert névterek a .NET keretrendszerben.

System.Windows.Media
System.Drawing
System.IO
System

4. Öröklődés

Az öröklődés szabályai.

Az öröklődés az OOP egyik alapelve. Egy általánosan definiált osztályból legyen lehetőség létrehozni specializált osztályokat, melyek tartalmazzák, egyes esetekben felülírják, és mindenek előtt kibővítik az ősük mezőinek és metódusainak listáját. Az így létrejött gyerekosztály az őse minden tulajdonságával rendelkezik. Kevesebb nem lehet az ősénél!

- A C# -ban úgy jelezzük az öröklődést, hogy az osztályunk neve mögé „:” után írjuk a választott ősének nevét.
- A gyermekosztály örökli az ős összes mezőjét, propertyjét és metódusát.
- Az osztálynak kötelessége kezelni az ősének legalább egy publikus konstruktorát, ha az ki van fejtve! Ezt a gyerekosztály konstruktora mögé írt „base” kulcsszóval tudjuk meghívni.
- Ha az ős egy absztrakt osztály vagy egy interfész, a gyermekosztálynak kötelessége kifejteni az ős(ök) absztrakt metódusait!
- A különböző osztályfajtákra vonatkozó szabályok:
  - Egy osztálynak csak egyetlen másik osztály vagy absztrakt osztály lehet az őse!
  - Egy osztálynak bármennyi interfész őse lehet, akár osztály vagy absztrakt osztály ős mellett is!

Öröklődés különböző védelmi szintek esetén.

- Az ősosztály Public és Protected elemei megjelennek és elérhetőek a gyermekosztályban, mintha csak a sajátjai lennének. Az ősosztály Private elemeit, bár a gyermekosztály szintén megörökli, azonban nem éri el őket.

Mezők újradefiniálása.

- Ha az ősosztály egy Private elemével azonos nevű elemet hozunk létre a gyermekosztályban, az nem okoz fennakadást, mert a gyermekosztály számára a gyakorlatban nem elérhető az ősosztály Private eleme.
- Ha azonban az ősosztály egy Public vagy Protected elemét akarjuk az előbbi módon újradefiniálni a gyermekosztályban, az el fogja fedni a régi elemet. Futtatáskor ez nem okoz fennakadást, a keretrendszer mégis figyelmezteti a programozót, hogy a korai kötést tudatosan használva írja ki a „new” kulcsszót.

Metódusok és property-k újradefiniálási lehetőségei.

- A metódusok és propertyk is ugyanazokon a módokon definiálhatóak újra.
  1. Korai kötéssel, a „new” kulcsszó használatával
  2. Késői kötéssel, a „virtual” és az „override” kulcsszavak használatával
  3. Megadhatóak az ősben absztrakt elemekként, amik késői kötést alkalmaznak, de az ősben nincsenek definiálva, így a gyermekosztályra hárul a feladat, hogy ezt megtegye.
  4. A propertyk újradefiniálásánál nem lehet külön csak a gettert vagy csak a settert.

Korai és késői kötés.

- **Korai kötés**
  - A fordításkor hajtódik végre.
  - Oldhatatlan, megváltoztathatatlan kötés, a fordítóprogram sosem fogja felülbírálni.
  - Gyorsítja a végrehajtást.
  - A „new” kulcsszóval készíthető el.
  - Problémák lépnek fel, amikor az őt tartalmazó gyermekosztály az ősének a konstruktorán keresztül próbálja meghívni. Ekkor a fordító kikerüli a felüldefiniálást és az ősben található változatot fogja használni.
- **Késői kötés**
  - Futásidőben hajtódik végre
  - Futáskor a program képes eldönteni, hogy a felülírt elem melyik változatát használja.
  - Az ősben a „virtual”, a gyermekosztályban „override” kulcsszóval készíthető el.
  - Lassabb futású
  - Kiküszöböli a korai kötés problémáját, mert mikor az ősbe visszalépve a virtuális metódushoz ér, megkeresi annak a legújabb változatát és azt futtatja le a gyermekosztályban.

5. Konstruktorok.

A konstruktor fogalma és feladata.

- Speciális metódus, mely a példányosításkor fut le. Jellemzően inicializálási feladatokra, alaphelyzetbe állításra, például a paraméterek feldolgozására használják.
- Egy osztályban lehet több konstruktor is, amiket a fordító a paramétereik alapján különböztet meg.
- Szintaktika: a metódus neve meg kell egyezzen az őt tároló osztály nevével!

Alapértelmezett konstruktor.

- Az osztálynak mindig van legalább egy konstruktora. Ha min nem hozunk létre, akkor fordításkor létrejön egy 0 paraméteres, üres törzsű konstruktor. (Ha viszont manuálisan létrehozunk egyet, akkor az automatikus nem jön létre!)

Konstruktor hívása konstruktorból. Konstruktorhívási lánc.

- Egy osztálynak az operator owerloading elve alapján több konstruktora is lehet. Ha egy konstruktorból meg akarunk hívni egy másik konstruktort, akkor a hívó neve mögé „:” -ot követően „this” kulcsszóval és a megfelelő paraméterlistával hivatkozunk a meghívandóra.
- Konstruktorhívási lánc
  - A szülő-gyerek osztálykapcsolatokban a gyerekosztály konstruktorának futása előtt egy láncreakció indul meg, minek során az Objacet osztálytól az aktuális gyerekosztályig az összes, szülő-gyerek kapcsolatban résztvevő osztály konstruktora lefut.

Az osztályszintű konstruktor.

Tulajdonságok:
- A neve megegyezik az osztály nevével.
- Nincs visszatérési típusa.
- **Nincs paramétere.**
- **El van látva a static jelzővel.**
- **Kötelezően private, ezért azt nem kell kiírni.**
- **Csak egyetlen osztályszintű konstruktor lehet egy osztályban!**

Az osztályszintű konstruktor a statikus mezők beállítására szolgál. Ezt a rendszer az osztály első példányosításakor hívja meg, így lehetőséget biztosít arra, hogy a statikus mezőket maga a felhasználó inicializálja.

Object factory.

- Az Objezt factory egy többlépcsős eljárás a példányosításra.
- Ilyenkor az osztály konstruktora privát, ezért közvetlenül nem lehet példányosítani.
- Az osztályba bekerül egy statikus, publikus, az osztály objektumával visszatérő metódus, mely a megfelelő paramétereket bekérve, közvetetten elvégzi a példányosítást.
  - Mivel az osztály része, ez a metódus hozzáfér a privát konstruktorhoz.
- Előnye, hogy a példányosító metódusban lehetőségünk van különféle, akár statikus mezőkhöz kapcsolódó vizsgálatokat végezni, adott esetben a példányosítást is megtagadni.

6. Típuskompatibilitás.

OOP területén a típuskompatibilitás fogalma.

Az öröklődés elve szerint egy ősosztály minden közvetett és közvetlen leszármazottja rendelkezik az adott ős minden mezőjével és metódusával.
A típuskompatibilitás a fenti elv miatt kimondja, hogy bármilyen gyermekosztály átkonvertálható a saját közvetlen, vagy közvetett ősévé (vagyis kompatibilis vele).

- Ez a szabály fordítva nem igaz, mivel a gyermekosztály legtöbbször több információt tartalmaz ősénél. (pl.: Minden pulikutya emlős, de nem minden emlős pulikutya. → A „pulikutya” osztály minden esetben átkonvertálható az őséül szolgáló „emlős” osztállyá).

Az is és as operátorok.

Bár alakjuk és működésük nagyon hasonló, a két operátor külön szerepet tölt be a típuskompatibilitás témakörében.

- **Is**
  - Logikai igaz értékkel tér vissza, amennyiben a baloldalán található példány kompatibilis a jobboldalára írt osztállyal.
  - If (pulikutya **is** Emlős) = „true”
- **As**
  - Típuskényszerítés, ráveszi a futtatókörnyezetet, hogy az adott példányt egy tőle különböző típusként kezelje.
  - Általában együtt használják az „is” operátorral, hogy a típuskompatibilitás hiányából származó hibákat elkerüljék.

Típuskompatibilitás haszna kollekciók használatakor.

Ha egy osztálynak van x számú gyermekosztálya és azoknak is van y számú gyermekosztálya, akkor is meg lehet oldani, hogy az összes előbb felsorolt típusú objektum egyetlen listába kerüljön. A lista típusának a közös őst kell megadni.

Object típus és annak virtuális metódusai.

Az Object osztály minden további osztálynak az őse, ezért minden osztállyal kompatibilis.
Ha egy metódus object típusú paramétert vár, az azt jelenti, hogy a típuskompatibilitás alapján bármilyen típusú paramétert elfogad.
Az Objezt fontosabb virtuális metódusai:

- **GetType()**
  - Az objektum típusát adja vissza.
- **ToString()**
  - Az objektumot (alapértelmezetten az objektum típusát) stringgé alakítja.
  - Ha azt akarjuk, hogy az objektumnak ezt a metódusát használó függvények más eredményt adjanak, az adott osztályban „override”-olni kell a ToString() metódust.
- **Equals()**
  - Két objektum egyenlőségét vizsgálja.
  - Bool értékkel tér vissza.
  - Alapértelmezetten két objektum memóriacímét hasonlítja össze.
  - Újraírásnál meg kell állapítanunk a két összehasonlítandó értéket:
    - A metódus egyik paramétere a saját osztálya, melyre „this” kulcsszóval hivatkozhatunk, a másik pedig az „obj” névan bekért paraméter.

        - Összehasonlítás előtt az obj-t konvertáljuk a megfelelő típusúra!
- **GetHashCode()**
    - Feladata, hogy egy olyan int értéket generáljon, mely az objektumot valamilyen szinten azonosítja.
    - A Hash kód egy információhalmaz részéből vagy egészéből keletkezik. Determinisztikus, vagyis ugyanazt az információhalmazt megkapva mindig ugyanazt az eredményt adja.
    - Felhasználása: azonosságvizsgálat, például az Equals() metóduson belül.

7. Sealed, static és abstract osztályok.

Sealed osztályok.

- Létrehozásához a „sealed” kulcsszót kell az osztály neve elé írni.
- Jelentése: lepecsételt osztály.
- Az ezzel a kulcsszóval megjelölt osztályokat példányosítani lehet, de ősül választani nem.
- Ilyen például a „Console” osztály.

Static osztályok. Singleton.

A Statikus osztályokból a statikus mezőkhöz hasonlóan csak egy darab jön létre a memóriában.
Másik elnevezésük a Singleton.
Tulajdonságai:

- Csak osztályszintű elemeket tartalmazhat.
- Nincs és nem is generálódik konstruktor.
- Nem lehet ősül választani.

Absztrakt osztályok.

- Létrehozásukhoz az „abstract” kulcsszót kell az osztály neve elé írni.
- Ha tartalmaz absztrakt metódust, az osztályt is hasonlóan meg kell jelölni.
- Az absztrakt osztály tartalmazhat nem absztrakt metódust is.
- Ha egy absztrakt osztályt egy másik absztrakt osztály ősül választ, az ős absztrakt metódusait nem köteles felülírni.
- Az absztrakt osztályt nem lehet példányosítani.
- Azokat az absztrakt osztályokat, amik kizárólag absztrakt elemeket tartalmaznak, „Interface”-eknek nevezzük.

Absztrakt metódusok és virtuális metódusok összehasonlítása.

Absztrakt metódus	Virtuális metódus
Nincs törzse	Van törzse
Kötelező felülírni	Nem kötelező felülírni
Késői kötést használ	

8. Bővítő metódusok. Operator overloading.

A this kulcsszó fogalma és használati lehetőségei.

Egy osztályon belül az adott osztályra a „this” kulcsszóval hivatkozhatunk.

- Használható az azonos nevű mezők egyértelműsítésére például konstruktorokban
- Az „Equals” felülírásakor így tudjuk meghívni az osztály belső értékeit.

Bővítő metódusok.

A meglévő osztályainkba lehetőségünk van kívülről metódusokat beépíteni

- Csak statikus osztályba írható!
- Csakis Public Static metódus építhető be!
- Segítségével írhatunk olyan metódust, mely a „string” osztályt bővíti ki: megmondja, hogy az a string, amire meg van hívva például 4 betűből áll-e.
- Szintaxis:

public static	bool	MetodusNev	(	this	string	szoveg	, …, ...	);
Csak ilyen lehet!	Visszatérési érték		A „this” kötelező! Utána annak az osztálynak a nevét írjuk, amibe be akarunk építeni.			A kibővített osztály aktuális példánya.	Ezután jönnek a rendes paraméterek!	

- A törzsében elvégezzük a műveletet, majd visszaadjuk a bool értéket!
- (string) szo.MetodusNev() →Ellenőrzi, hogy a „szo” tartalma 4 betű hosszú-e.

Operator overloading

Gyakran előfordul a gyakorlatban, hogy olyan dolgokat akarunk megvalósítani a programnyelv operátoraival, melyre az nincs felkészítve (például egy Date mezőhöz hozzáadni egy unit értéket a napok számának növelése érdekében).

- Erre nyújt megoldást az Operator overloadnig, melyet egy metódushoz hasonlóan kell létrehozni.
- Létrehozás:

public	static	Days	operator	+	(	Day datum	,	Uint hozzaad	);
Védelmi szint	Mindig statikusnak kell lennie!	Annak a mezőnek a típusa, amin végrehajtjuk a változtatást	A név helyére az „operator” kulcsszó tekül	A túltöltendő operátor	Paraméterlista. Fontos, hogy annyi paramétert és olyan sorrendben adjunk meg, ahogy azt az adott operátor megköveteli.				

- A metódus törzsében elvégezzük a konkrét műveletet és visszatérünk azzal a „datum” értékkel, amiben a napok számát megnöveltük.
- Ezek után ha a kódban leírjuk a „datum+2” parancsot, akkor hibakód helyett a fenti függvény fog lefutni és a „datum” nevű, Date értékben meg fog növekedni a napok száma 2-vel.
- Túlterhelhető operátorok:

Unáris operátorok:	+, -, !, ++, --
A bináris operátorok legtöbbje, az értékadó operátorokat kivéve:	+, -, *, /, %, ==, !=, <, >, <=, >=
Kasztolás operátorok	

9. Interface.

Interface szintaktikai szabályai.

Az interfész egy speciális osztályfajta, vagy úgynevezett fungcionális definíció. Önmagában nem tárol információt, propertyk metódusok definícióival szolgál az őket ősül választó osztályok számára.

Interface és absztrakt osztály összehasonlítása.

- Kizárólag absztrakt elemeket tartalmaz.
  - Az elemeknek nincs metódustörzse.
- Minden eleme csak publikus lehet, ezért ezt nem kell jelezni.
- Nem tartalmazhat mezőt.
- Nem tartalmazhat osztályszintű elemet.
- Még definíció szinten sincs konstruktor vagy destruktor.
- A gyermekosztályban kifejtéskor nincs szükség az „override” kulcsszóra.

Interface implementálása, öröklődéssel való kapcsolata.

- Interfész implementálhat interfészt.
- Egy osztály bármennyi interfészt implementálhat.

Ismert interface-ek a .NET keretrendszerben.

- **ICompareable, IComparer**
  - Összehasonlítást lehetővé tévő interfészek.
- **IEnumerable, IEnumerator**
  - Bejárhatóságot biztosító interfészek (foreach)
- **ICollection, IList**
  - Az IEnumerable leszármazottjai
  - Count propertyt ad hozzá
  - Listákkal kapcsolatos metódusokat (Add, Insert, Remove, Contains, IndexOf) ad hozzá

yield kulcsszó jelentősége és használata.

Leegyszerűsíti az enumerátorok használatát, melyeket a fordító automatikusan implementál. Használata hibakezelési és memóriahasználati okokból is nagyon fontos.

- **Yield return;**
  - Visszaadja az enumerátort
  - Vegyünk két metódust. Az egyik a GetNumber(int szam) függvény, ami egy listát ad vissza, mely „szam” mennyiségű random számot tartalmaz. A másik metódus meghívja ezt a függvényt, egy listát készít az értékekből, majd egy foreach-el kiírja annak tartalmát.

```
static public void Write()
{
    var Numbers = GetNumbers(1000);
    foreach (var item in Numbers)
    {
        Console.WriteLine(item);
    }
}
static public IEnumerable<int> GetNumbers(int szam)
{
    List<int> tmp = new List<int>();
    for (int i = 0; i < szam; i++)
    {
        tmp.Add(i);
    }
    return tmp;
}
```

    - Ebben a kódban előbb legenerálódik az összes szám és csak ezután lesznek kiíratva.
    - Mivel a List() gyermeke az IEnumerable interfésznek, ezért kompatibilis is vele.
  - A Yield return használatával az eredmény nem, a kód csak egy kicsit, de a működés nagyon megváltozik:

```
static public void Write()
{
    var Numbers = GetNumbers(1000);
    foreach (var item in Numbers)
    {
        Console.WriteLine(item);
    }
}
static public IEnumerable<int> GetNumbers(int szam)
{
    for (int i = 0; i < szam; i++)
    {
        yield return i;
    }
}
```

    - Mostmár kötelező az IEnumerable használata!
    - Ahelyett, hogy egyszerre legenerálná, a foreach haladásával egyidejűleg generálja az új értékeket, mindig visszalépve a belső ciklusva az „in” kulcsszó után.
    - Bizonyítás: Helyezzünk el a for-ciklusban egy „if”-et, mely mondjuk i=30 esetén hibaüzenettel leállítja a programot. Az első megoldás esetében a leállás pillanatában a konzol üres, mert a program nem jutott el a kiíratásig. A második megoldásnál ellenben az első 30 szám kiírásra került, mert a GetNumbers() i változója minden foreach körönként növekedett egyel.
- **Yield break**: Megállítja a folyamatot (úgy, mint a sima break egy ciklust).

10. Kivételkezelés.

Hagyományos hibajelzési technikák, azok hátrányai.

- „if” ágakkal lehetőségünk van észlelni a metódusok által dobott hibakódokat, ám ezek túlbonyolítják a programot, vagy még több hibát okozhatnak.
- Lehet írni az elágazásokhoz „default” ágakat hibakezelési feladatokra, ám ez egy általános, alkalmazkodásra képtelen megoldás.

Kivételkezelés filozófiája.

- A kivétel olyan futás közben fellépő hiba, mely megszakítja az utasítások normális végrehajtási rendjét. Ezek lehetnek például erőforrási hibák (hiányzó fájl), bemeneti hibák (helytelen adatok a felhasználótól), fejlesztési hibák (túlindexelés). Lehetnek továbbá Inplicit (a futtatókörnyezet által automatikusan kiváltott) és explicit (a programozó által kiváltott) kivételek.
- A kivételkezelés az a művelet, melynek során felkészítjük a programot az esetleges futási hibákra, hogy az visszatérhessen eredeti állapotába.

Kivételosztályok.

A C# hibakezelése hibaobjektumokon alapszik. A hiba mivoltától függően létrejön a bekövetkezésekor egy hibaobjektum, mely információkat tartalmaz a hibáról.
A kivételkezelés során a hibaobjektum elkapásával tud a program tájékozódni a hiba jellegéről és kiválasztani a legjobb megoldást.
Minden hibaobjektum Exception típusú példány, vagy annak egy leszármazottja.
Hibaobjektumokat a programozó is tud írni. Egy átlagos osztálynak kell ősül választani az Exception osztály. Ezután az osztályban elérhetővé válnak a hibaobjektumok részei, mint név, vagy leírás. Ezeket átírva készíthetünk saját hibaobjektumot, melyekkel még részletesebb hibakezelés végezhető.
Nevezetes hibaobjektumok:

- **FileNotFoundException** (nem található a keresett fájl)
- **IndexOutOfRangeException** (az index a határon kívülre mutatott)
- **OutOfMemoryException** (elfogyott a memória)

Kivételek dobása és elkapása.

A fejlesztés során a programban keletkeznek olyan blokkok, amik nagy eséllyel futhatnak hibára. Ezeket a blokkokat érdemes Try-Catch-ekkel védeni.
Az adott hibát vizsgáló Try-Catch-et a hiba forrásához a lehető legközelebb kell tenni, hogy ne akadályozza a többi kód futását.
Try-Catch működése:

- A Try ág törzsében szerepel a kritikus blokk
- Ehhez tartozhat több Catch ág is, melyek különböző hibakódokat kapnak paraméterül, ezeket vizsgálják.
- Ha létrejön egy hibakód, az azt kezelő Catch ág törzse fut le.
- A Catch ágakat érdemes a legspeciálisabbtól a legáltalánosabbig írni, hogy egy általános ág ne kapjon el olyan hibát, amit egy specializált ág jobban megoldana
- A Catch-blokk után írhatunk egy „Finaly” ágat, ami mindig lefut, függetlenül attól, hogy volt-e hiba vagy nem. Ebbe érdemes tenni például fájlok bezárásának kódját és hasonlóan fontos dolgokat, amiknek mindig le kell futniuk.

11. Generikusok.

Boxing és unboxing.

Különféle adatok Object példányokban való tárolása során ütközhetünk abba a hibába, hogy az értéktípusú adatot nem lehet eltárolni egy referenciatípusú object-ben. Erre a problémára megoldás a „Boxing”, mely a nevéből adódóan „Becsomagol” egy értéktípusú változót egy referenciatípusú objektumba. A művelet fordítottja az „Unboxing”, „Kicsomagolás”.

- Boxing
  - Egy referenciatípusú változónak egy értéktípusú értéket adunk.
  - ```
    double d = 12.4;
    object o = d;
    ```
  - Lépései:
    - Az értékről másolat készül a memóriában
    - A másolat bekerül egy referenciatípusú változóba (Ez egy object lesz, mivel az referenciatípus és bármilyen adatot elfogad).
  - A művelet jelentős memória igénnnyel jár, megkétszerezi az értékadáshoz szükséges bájtok számát.
- Unboxing
  - A Boxing ellentettje: Az object változó értékét megkapja a megfelelő értéktípusú változó
  - ```
    double x = (double)o;
    ```

Általános célú programozás Object használatával.

Az Object osztály minden osztálynak az őse, így a típuskompatibilitás elve alapján minden osztállyal kompatibilis, bármilyen típusú adatot tárolni tud.
Erre a tulajdonságára épülnek az általános célú algoritmusok, melyek általában kereséseket, rendezéseket hajtanak végre bármilyen típusú adatokkal.
Az Object osztálynak vannak különféle virtuális metódusai. Ezeket a gyermekosztályokban lehetőségünk van specializálni.

- **GetType()**
  - Az objektum típusát adja vissza.
- **ToString()**
  - Az objektumot (alapértelmezetten az objektum típusát) stringgé alakítja.
  - Ha azt akarjuk, hogy az objektumnak ezt a metódusát használó függvények más eredményt adjanak, az adott osztályban „override”-olni kell a ToString() metódust.
- **Equals()**
  - Két objektum egyenlőségét vizsgálja.
  - Bool értékkel tér vissza.
  - Alapértelmezetten két objektum memóriacímét hasonlítja össze.
  - Újraírásnál meg kell állapítanunk a két összehasonlítandó értéket:
    - A metódus egyik paramétere a saját osztálya, melyre „this” kulcsszóval hivatkozhatunk, a másik pedig az „obj” névan bekért paraméter.
    - Összehasonlítás előtt az obj-t konvertáljuk a megfelelő típusúra!
- **GetHashCode()**
  - Feladata, hogy egy olyan int értéket generáljon, mely az objektumot valamilyen szinten azonosítja.
  - A Hash kód egy információhalmaz részéből vagy egészéből keletkezik. Determinisztikus, vagyis ugyanazt az információhalmazt megkapva mindig ugyanazt az eredményt adja.
  - Felhasználása: azonosságvizsgálat, például az Equals() metóduson belül.

Generikusok írása. Kikötések a típusparaméterre.
Az objectek használatával a nagy memóriaigény mellett egy másik nagy probléma is van: bár maga az object bármilyen adatot be tud fogadni, az adatokat kezelő programrészek legtöbbször már nem ennyire rugalmasak, így egy nem megfelelő típusú adat futásidejű hibát fog eredményezni.
A generikus fogalma: Típussal paraméterezhető osztály, interfész, metódus, stb.
Saját generikus írása:

- Írjunk meg egyet a fogalomban említett programrészek közül! A paraméter kereteként ezúttal <> jeleket használjuk! Az itt felsorolt paraméterek nem adatok, hanem típusok lesznek.
- Példányosításkor az adott generikus elem neve mögé ugyanúgy <> között be kell írni a paraméterekhez hasonlóan a használni kívánt típusokat, valamint a generikuson belül is a típusváltozókkal kell hivatkozni az adott elemek típusára (amik ugye példányosításonként változhatnak).
- Így bár az object-ekhez hasonlóan többféle típusú adat fogadására is alkalmas lesz egy osztály, ezeket példányosításonként fixáljuk. Így például ha van egy általános konténerosztály és abból háromfélét akarunk gyártani háromféle adat tárolására, akkor, bár ugyanazt az osztályt használják, mégis csak az adott típusú adatot fogadják el.

```
class Node<T> {
   T data;
   public T Data {
      get { return data; }
      set { data = value; }
   }
   ...
   Node<T> nextNode;
   ...
}
```

- Ha egy Node objektumot <string> típus megadásával gyártunk le, akkor a későbbiekben csak stringet lehet majd beletölteni.

```
Node<int, string> n1 = new Node<int, string>
(szam, szoveg);
```

- Bár a fenti példában nem látható konstruktor, ha a példányosításkor megadunk egy <T,K> típusparamétert és a konstruktor is T, K típusú adatokat kér, akkor természetesen csak ilyen típusú adatokkal tudunk példányosítani.

Kikötések:
A program helyes működése érdekében a paraméterlista megírása utáni „where T : ” kulcsszóval tudunk kikötést tenni arra, hogy csak akkor fogatjuk el az adott T típusparamétert, ha ősül választotta a „:” után megnevezett osztályokat.

```
public T Maximum<T>(T a, T b, T c)
   where T: IComparable
{
   T x = a;
   if (b.CompareTo(x) > 0) x = b;
   if (c.CompareTo(x) > 0) x = c;
   return x;
}
```

Két értéket akkor lehet összehasonlítani, ha a típusuk őse az IComparable interfésznek. A kikötéssel megtehetjük, hogy előre levizsgáljuk ezt és le sem engedjük futni a folyamatot.

Ismert generikusok a .NET keretrendszerben. (System.Collections.Generic)

- Lista List<T>
- Verem Stack<T>
- Sor Queue<T>
- „Szótár” Dictionary<T>

12. Delegate.

Delegate fogalma és használata.

A memóriában kétféle tartalmat tárolunk: adatot és kódot. Az adatra referencia típusú változókkal hivatkozunk, míg a kódra delegete-ekkel.
Delegate: Metódus referencia.
Használata:

1. Típus deklarálása:

```
delegate int Operation(int a, int b);
```

   - Ugyanúgy történik mint bármilyen metódus esetében, annyi a különbség, hogy elé kerül egy „delegade” kulcsszó.
   - Tekintsünk az „Operations” elemre úgy, mint egy üres változóra, mely a megadottakkal azonos viszatérési értékű és paraméterlistájú metódusokat tud tárolni.

2. A delegate-re illeszkedő metódus(ok) megírása

```
static int Addition(int x, int y) {
  return x + y;
}

static int Division(int x, int y) {
  if (y == 0)
    throw new Exception("Nem oszthatsz 0-val!");

  return x / y;
}
```

   - A visszatérési érték és a paraméterlista azonos legyen a delegate-évl.

3. Delegate példányosítása

```
Operation op;
```

4. A delegate-nek értékül adjuk a rá passzoló tetszőleges metódust.

```
op = Addition;
```

5. Ezentúl ha a delegate-et futtatjuk, a 4. pontban értékül adott metódus fog rajta keresztül lefutni.

```
op(132, -83);
```

Lehetőségünk van Generikus delegate-eket létrehozni

```
delegate T Operation<T>(T a, T b);
```

Ez hasonlóan zajlik, mint a határozott típusú értékadás, ám a példányosításkor meg kell határoznunk a típust.

```
Operation<int> op;
```

Callback metódusok.

Van arra lehetőség, hogy egy metódust delegate-ként átadunk egy másik metódusnak.
Nagy jelentősége van a 3-rétegű alkalmazásfejlesztési elvben, melynek egyik kitétele, hogy az egyes rétegekben lévő programrészek ne érjenek át másik rétegekbe. Ha egy metódusnak olyan

dolgot kell csinálni, ami kiér az ő rétegéből, akkor paraméterként meghív egy delegate-et, amiben szereplő metódussal meg tudja oldani a feladatot.

Események.

Napjainkban az Eseményvezérelt programozás a legelterjedtebb programozási forma.
Az esemény, vagyis „event”, úgy írható le, mint delegate-ek listája, azok egyszerre való meghívása. Egy event meghívásakor az összes benne tárolt delegate lefut.
Event létrehozása:

1. Deklarálunk egy delegate típust.

```
delegate void LogProc(string message);
```

2. Létrehozzuk az eventet
   - Az „event” kulcsszóval kezdünk, megadjuk a delegate-típust, amit tárolni szeretnénk és az event nevét.

```
event LogProc log;
```

3. Hozzáadjuk a delegate példányokat.

```
log += LogSQL;
```

4. Esemény kiváltása
   - Leírjuk az event nevét, utána pedig a benne szereplő delegate-ek paraméterlistáját.

```
log("Hiba: lemez megtelt");
```

   -

Anonim delegate-ek.

Az anonim delegate-ek, ahogy a nevük is mutatja, név nélküli, egyszer használatos delegate-ek. Ezek nem csak a lustaságot szolgálják ki, de nagyon sok új programozási elem épül rájuk, mint például a Lamda-kifejezések.

```
delegate(<paraméterlista>) {
  <metódustörzs>
}
```

Használhatjuk ezt a formát, ha a delegate-et csupán egy eventhez akarjuk hozzáadni és nem önállóan használni.

13. Lambda kifejezések és LINQ.

Lambda kifejezés fogalma, szintaxisa és jelentősége.

- Megjelenése óta a C# programnyelv rengeteget fejlődött. Az évek során megfigyelhető volt az imperatív kezdetektől a deklaratív irányba való haladás.
- A lamda kifejezések a C# új elemei, erősen deklaratív formában vezetik be az SQL-szerű adatkezelést.
- A szintaxisa legtöbbször az SQL és a C# szintaxisának keveréke.
- A lamda kifejezés operátora a „=>”. Ennek bal oldalán egy tetszőleges nevű változó áll (a típusát nem kell megadnunk), a jobb oldalán pedig a rá vonatkozó kifejezés van.
- A lamda kifejezés egy anonim delegate, csak más, intuitívabb szintaxissal.
- `x => x.EndsWith("dinnye")` → Adja vissza x-et, ha az „dinnye”-val végződik!
- A lamda kifejezésekkel hosszú kódokat, kódismétléseket igénylő lekérdezéseket iktathatunk ki.

Ismert generikus delegate-ek a .NET keretrendszerben és ezek lambda kifejezésekkel való kapcsolata.

A pamda kifejezések hátterében különféle delegate-ek futnak.

Predicate<T>
- Egy számparamétere van, bool értéket ad vissza:

Action<T1,…,T16>
- 16 -féle, különböző típusú paramétere lehet. Eljárást reprezentál, ezért nincs visszatérési értéke.

Func<T1,...T16, TR>
- 16 -féle, különböző típusú paramétere lehet. Függvényt reprezentál, melynek TR a visszatérési típusa.

LINQ bővítő metódusok és használatuk.

A LINQ jelentése magyarul: nyelvbe ágyazott lekérdezés. A C# 3.0 verzióval kerültek a nyelvbe, az adatbázis kezelés világából hozott parancsokat és szűréseket hozták magukkal.

Szűrés Bizonyos elemek kiszűrése megadott feltételek alapján	Where
Projekció Leképezés más típusú elemekre	Select, SelectMany
Rendezés Rendezés megadott szempont alapján	OrderBy, ThenBy
Csoportosítás Csoportosítás megadott szempont alapján	GroupBy
Joinok Elemek összekapcsolása megadott kulcsok alapján	Join, GroupJoin
Quantifiers Logikai kvantorok	Any, All

Particionálás Kollekció feldarabolása megadott feltétel alapján	Take, Skip, TakeWhile, SkipWhile
Halmazműveletek	Distinct, Union, Intersect, Except
Elemek Egyetlen elem kiolvasása	First, Last, Single, ElementAt
Aggregáció Kivonatkészítés konkrét érték formájában	Count, Sum, Min, Max, Average
Konverzió	ToArray, ToList, ToDictionary
Kasztolás Az elemek típus szerinti szűrése, vagy konvertálása	OfType<T>, Cast<T>