Eötvös Loránd Tudományegyetem
Természettudományi Kar

Alkalmazott Modul III

1. gyakorlat

Objektumorientált keretrendszerek, C# alapismeretek

© 2011.09.20. Giachetta Roberto
groberto@inf.elte.hu
http://people.inf.elte.hu/groberto

Objektumorientált keretrendszerek
Történelem

- 1967-ben a *Simula 67* támogatta először az objektumorientált programozást
- 1980-ban a *Smalltalk* nyelv volt az első, amely teljesen objektumorientáltan épült fel, és megadta az összes őt követő nyelv alaptulajdonságait, úgymint:
  - minden implementált elem (változók, konstansok, metódusok) egy objektum, és egyben egy osztály példánya, az osztályok egy *teljes származtatási hierarchiá*ban vannak
  - a memóriakezeléshez *szemétgyűjtő*t használ
  - *dinamikus programozás* támogatása
  - teljesen hordozható kódot biztosít *virtuális gép*en történő futtatás segítségével

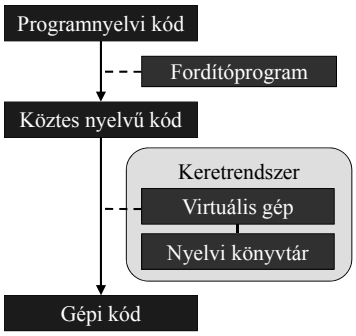
Objektumorientált keretrendszerek
A virtuális gép

- A hordozhatóság akadálya, hogy a fordítással keletkezett alacsonyszintű programkód gépfüggő, ezért a *Smalltalk* programokat egy gépfüggetlen *köztes nyelv*re (*Intermediate Language*) fordították
- A köztes nyelvű program egy értelmező szoftver segítségével futtatható, amely futás közben alakítja gépi kóddá a programkódot, ezt az értelmezőt nevezzük virtuális gépnek (*Virtual Machine*)
- Ezt a félig fordított, félig értelmezett megoldást nevezzük *futásidejű fordítás*nak, vagy röpfordításnak (*Just In Time Compilation*)
  - a futtatáskor optimalizációk történnek, így az értelmezést követően gyorsabb lehet a futás, mint teljes fordítás esetén

Objektumorientált keretrendszerek
A szoftver keretrendszer

- Mivel a fordítás egy része, és az összeszerkesztés futási időben történik, ezért nem kötelező minden programkomponenst beágyazni a programba, a hivatkozások feloldása történhet futtatáskor is
  - ezáltal csökkenthető a program mérete és a betöltés ideje
  - a kiemelt komponenseknek jelen kell lenniük a gépen
  - célszerű a beépített könyvtárak kiemelése
- *Szoftver keretrendszer*nek nevezzük a kiemelt programkönyvtár és a virtuális gép együttesét
  - tartalmazza az API gépfüggetlen, absztrakt lefedését
  - felügyeli a programok futásának folyamát
  - biztosítja a memóriakezelést, szemétgyűjtést

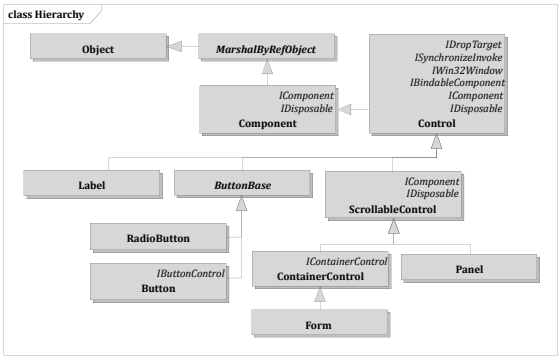
Objektumorientált keretrendszerek
Futásidejű fordítás



Objektumorientált keretrendszerek
Teljes származtatási hierarchia

- Teljes származtatási hierarchiában van egy egyetemes ős osztály, minden más osztály ennek leszármazottja (akkor is, ha nincs megjelölve ősként)
  - az ős definiálja az alapértelmezett viselkedését minden objektumnak (pl. szöveggé alakítás, másolás,…)
  - minden beépített osztály egy származtatási hierarchia mentén van implementálva, és ezek között absztrakt osztályok is találhatóak
  - ezen láncok mentén specializálódnak, illetve kiegészülnek az osztályok viselkedései
  - a hierarchia mentén találhatóak megkötések is a származtathatóságra és a felüldefiniálhatóságra

Objektumorientált keretrendszerek
Teljes származtatási hierarchia



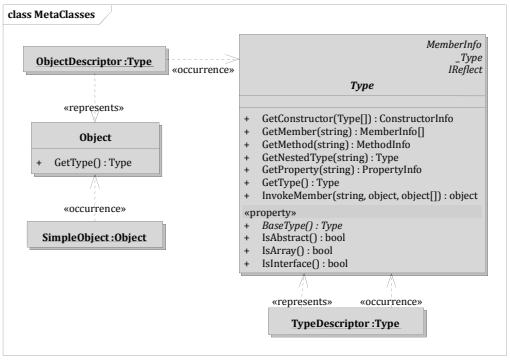
Objektumorientált keretrendszerek
Memóriakezelés

- Az objektumorientált nyelvekben célszerű referenciákon, illetve mutatókon keresztül hivatkozni az objektumokra, hatékonysági és élettartam szabályozási okok folytán
  - mutatók esetén a létrehozást és törlést manuálisan kell megírni, ám a törlés sokszor elmarad, ezért a memóriában maradnak a felesleges, nem hivatkozott objektumok (a szemét), ezért célszerű ezt automatikusan elvégezni
- A virtuális gép feladata, hogy felügyelje a program által elfoglalt memóriaterületet, és a benne lévő memóriaszemetet eltávolítsa, ezt nevezzük *szemétgyűjtés*nek (*Garbage Collection*)
  - ciklikusan ellenőrzi a memóriát és a hivatkozásokat, és kiüríti a nem hivatkozott memóriaterületeket

Objektumorientált keretrendszerek
Metaosztályok és dinamikus programozás

- Mivel minden objektum, maguk az osztályok is objektumok, és az ő viselkedési mintájukat is definiálni kell, erre a célra szolgálnak a *metaosztályok*
  - közös felületet biztosít osztálytulajdonságok és azok részleteinek lekérdezésére, metódusok futtatására, módosítására és példányosításra
  - az objektumok és osztályok lehetőséget adnak a metaosztály lekérdezésére és az osztályhoz tartozó *metaobjektum* létrehozására
  - a metaosztály is egy objektum, a típusa szintén metaosztály
- A metaobjektumokon át bármely osztályt módosíthatunk futás közben, ezáltal dinamikusan programozhatóvá válik a rendszer

Objektumorientált keretrendszerek
Metaosztályok és dinamikus programozás



Objektumorientált keretrendszerek
A Java

- 1991-ben indult el a *Java* fejlesztése a *Sun Microsystems*-nél, amelynek célja egy objektumorientált, általános célú, hordozható kódú, sokplatformos programozási nyelv megalkotása volt
  - könnyű programozhatóságot, ugyanakkor lassabb programfuttatást eredményezett, a programok a Java virtuális gépen (*JVM*) futottak
  - pár év alatt, különösen a mobil és webes alkalmazásoknak köszönhetően nagy népszerűségre tett szert
  - a *Microsoft* saját virtuális gépet, és nyelvi könyvtárat fejlesztett ki (*Virtual J++*), amely gyorsabb futtatást tett lehetővé

Objektumorientált keretrendszerek
A .NET Framework

- 1998-tól a Microsoft új célja a futásidejű fordítás kiterjesztése a létező Microsoft nyelvekre (Visual C++, Visual Basic), ezáltal egy közös virtuális gépen futhatnak a programok
  - egységes köztes nyelv vezethető be, így a nyelvek tetszőleges mértékben kombinálhatóak egymással
  - egységes programkönyvtárak használhatóak
- 2001-re készült el a *.NET Framework*, és a dedikáltan ráépülő nyelv, a C#
  - azóta integrált része a Windows rendszereknek
  - hozzáférést biztosít az összes Microsoft technológiához (*COM*, *ODBC*, *DirectX*)

Objektumorientált keretrendszerek
A .NET Framework

- Egységes virtuális gép: *Common Language Runtime* (*CLR*)
- Egységes köztes nyelv: *Common Intermediate Language* (*CIL*)
- Egységes típusrendszer: *Common Type System* (*CTS*)
- Teljeskörű programkönyvtár
  - *Base Class Library* (*BCL*): gyűjtemények, I/O kezelés, adatkezelés, hálózat, párhuzamosítás, XML, …
  - *Framework Class Library* (*FCL*): WinForms, ASP.NET, LINQ, WPF, WCF
- Biztosítja a programok védettségét és hordozhatóságát
  - memóriakezelés felügyelete: *Managed Code*
  - köztes kód védelme: *Code Access Security* (*CAS*)

Objektumorientált keretrendszerek
A .NET Framework története

- 1.1 (2003): megnövelt *ASP.NET* funkcionalitás, biztonság. Compact Framework megjelenése
- 2.0 (2005): eléri a Java funkcionalitását
  - gyorsabb programfuttatás, 64 bites támogatás
  - sablonok, parciális osztályok, névtelen metódusok támogatása, új adatkezelés (ADO.NET)
- 3.0 (2007): új technológiák a kommunikációra és megjelenítésre (WCF, WPF, WF, CardSpace)
- 3.5 (2008): funkcionális programozás, nyelvbe ágyazott lekérdezések (LINQ), AJAX támogatás
- 4.0 (2010): párhuzamosítás automatizálása (PLINQ, TPL), szerződés alapú programozás (DbC)

Objektumorientált keretrendszerek
A .NET Framework nyelvei

- Összesen 44 nyelv biztosít támogatást a .NET keretrendszer felé, a hivatalosan támogatott nyelvek:
  - *C#*: a Visual J++ utódnyelve, amely eltávolodik a Java szintaxisától, több ponton visszatér a C++-hoz
  - *J#*: a Visual J++ utódnyelve, megmarad a teljes Java szintaxisnál, a .NET környezetre építve (ma már nem használatos)
  - *Visual Basic .NET*: a Visual Basic továbbfejlesztése
  - *C++/CLI (C++.NET)*: C++ szintaxis kiegészítve a .NET könyvtárakra memóriafelügyelettel
  - *F#*: funkcionális programozási nyelv
  - *JScript.NET*: JavaScript alapú szkriptnyelv

Objektumorientált keretrendszerek
Kritika a .NET-tel szemben

- A futásidejű fordítás miatt
  - szükséges a keretrendszer megléte
  - lassúbb programindítás (nagyságrendi elmaradás a fordított programokhoz képest, azonban jelentős előny a Java-val szemben)
  - a köztes kód teljes mértékben visszafejthető bármely felsőbb szintű nyelvbe (ez valamelyest kivédhető kódzavaró eszközökkel, de nem teljesen)
- A memóriafelügyelet miatt
  - megnövelt erőforrásigény (mutatók folyamatos ellenőrzése)
  - késleltetés a szemétgyűjtő futásakor (valós idejű alkalmazások implementálása lehetetlen)

C# alapismeretek
A nyelv lehetőségei

- A C# *tisztán objektumorientált programozási nyelv*, amely teljes mértékben a *.NET Framework*re támaszkodik
  - hordozható kód, memóriafelügyelet
  - szintaktikailag nagyrészt C++, megvalósításában Java
  - egyszerűsített szerkezetet biztosít, nem választható el a deklaráció a definíciótól
  - strukturált felépülés névterekkel
  - lehetőséget ad komponens-alapú, elosztott, lokalizált, beágyazott fejlesztésre
  - 2.0-tól sablonok használata, elosztott típusok
  - 3.0-tól támogatja a funkcionális paradigmát
  - a forrásfájl kiterjesztése: `.cs`

C# alapismeretek
Azonosítók és literálok

- Kódolás: Unicode 3.0
  - azonosítókat a szabványnak megfelelően lehet írni
  - a kulcsszavak a @ karakterrel használhatóak azonosítóként
- Számábrázolás:
  - egész számok alapértelmezésben 10-es számrendszerben ábrázolódnak, de lehet 8-as (`0` előtag, pl. `0173`), illetve 16-os számrendszert használni (`0x` előtag, pl. `0x3de2`)
  - hosszú (`long`) számokat `L` utótaggal (pl. `132L`), előjel nélküli (`unsigned`) számokat `U` utótaggal jelölünk (pl. `12U`)
  - valós számok alapértelmezetten dupla pontosságúak, egyszeres pontosságot `F` utótaggal jelölünk (pl. `13.1F`)
  - valós számok megadhatóak kitevős formában is

C# alapismeretek
A „Hello, World!” program

```
namespace Hello // névtér
{
  class HelloWorld // osztály
  {
      static void Main() // statikus főprogram
      {
          System.Console.WriteLine("Hello, World!");
          // kiírás konzol képernyőre
      }
  }
}
```


C# alapismeretek
Névterek

- A névterek biztosítják a rendszer strukturáltságát, lényegében csomagoknak felelnek meg
  - minden osztálynak névtérben kell elhelyezkednie, nincs globális, névtelen névtér, így a program szerkezete: `namespace <nevek> { <osztályok> }`
  - hierarchikusan egymásba ágyazhatóak, és ezt a névtérben pont elválasztóval jelöljük, pl.:

```
namespace Outer { ... }
namespace Outer.FirstInner { ... }
// a fenti névtéren belüli névtér
namespace Outer.FirstInner.DeepInner { ... }
// a belső névtéren belüli névtér
namespace Outer.SecondInner { ... }
```


C# alapismeretek
Névterek

  - a .NET könyvtárai is hierarchikus névterekben találhatóak közvetlenül csak az aktuális névtérbeli osztályok láthatóak, de más névterekre is hivatkozhatunk
- Névtereket használni a `using <névtér>` utasítással lehet, ekkor a névtér összes típusa elérhető lesz
  - pl.:
```
using System;
using System.Collections.Generic;
```
  - az utasítás a teljes fájlra vonatkozik, így általában a névtérhasználattal kezdjük a kódfájlt
  - a típusnév előtt is megadhatjuk a használandó névteret (így nem kell `using`), pl.: `System.Collections.Stack l;`
  - típusnév ütközés esetén mindenképpen ki kell írnunk a teljes elérési útvonalat

C# alapismeretek
Típusosság

- A nyelv három típuskategóriát különböztet meg:
  - *érték*: érték szerint kezelendő típusok, mindig másolódnak a memóriában, és a blokk végén törlődnek
  - *referencia*: biztonságos mutatókon keresztül kezelt típusok, amelyeknél csak a memóriacím másolódik, a szemétgyűjtő felügyeli és törli őket, amint elvesztik az összes referenciát
  - *mutató*: nem biztonságos mutatatók, amelyek csak felügyeletmentes (unsafe) kódrészben használhatóak
- Minden típus objektumorientáltan van megvalósítva
- Minden típus a teljes származtatási hierarchiának megfelelően egy .NET Framework-beli osztály, vagy annak leszármazottja, a keretrendszertől független típusok nem hozhatóak létre

C# alapismeretek
Osztályok felépítése

- Az osztály négy féle elemből építhető fel:
  - *mező* (*field*): attribútum érték, amely lehet változó, konstans, vagy olvasható (lehet objektum- és osztályszintű)
  - *metódus* (*method*): osztály tagfüggvénye, amely eléri az osztály minden elemét (amennyiben nem osztályszintű)
  - *tulajdonság* (*property*): attribútumként funkcionáló érték, amely igazából metódusok segítségével van megvalósítva, lényegében a beállító és lekérdező művelet absztrakciója
  - *esemény* (*event*): eseményvezérelt programoknál kiváltható tulajdonság, amelyhez eseménykezelő metódus rendelhető, mely reagál rá
- A típusok tetszőlegesen egymásba ágyazhatóak

C# alapismeretek
Primitív típusok

- *Primitív típus*nak nevezzük a nyelv alaptípusait, amelyek a központi könyvtárban (a `System` névtérben) vannak megvalósítva
  - a C#-ban valójában nem primitívek, csak egyszerű formában használható osztályok, ezért *intelligens*ek
- Érték szerinti primitívek:
  - logikai: `bool`
  - egész számok (előjeles és előjel nélküli): `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`
  - lebegőpontos számok: `float`, `double`
  - fixpontos szám: `decimal` ($1.0*10^{-28}$ - $7.9*10^{28}$)
  - karakter: `char`

C# alapismeretek
Primitív típusok

- Referencia szerinti primitívek:
  - objektum (ősosztály): `object`
  - szöveg: `string`
- Minden rövidített C# típusnév ekvivalens egy .NET osztálynévvel, amely a `System` névtérben található, pl.:
  `int == System.Int32`, `object == System.Object`
- A rövidített névvel is elérhetőek az intelligens osztály tulajdonságok
  - osztályszinten, pl.:
    ```
    int.MaxValue // a maximális int érték
    ```
  - objektumszinten, pl.:
    ```
    int a = 1; a.ToString(); // kiírás szövegesen
    ```


C# alapismeretek
Típuskonverziók

- *Szigorúan típusos* a nyelv
  - minden értéknek fordítási időben ismert a típusa
  - az implicit (automatikus) típuskonverziók korlátozva vannak a nagyobb típushalmazba
    - pl.: `byte` ⟶ `short`, `ushort`, `int`, …, `double`
      `int` ⟶ `long`, `float`, `decimal`, `double`
      `float` ⟶ `double`
  - nem lehet automatikus konverzióra támaszkodni olyan típusok között, ahol nem garantált, hogy nem történik értékvesztés, és ez fordítási időben kiderül
    - pl.: `float` ⟶ `int`
    - ekkor explicit konverziót kell alkalmaznunk

C# alapismeretek
Típuskonverziók

- az explicit típuskonverzió fordítási időben felügyelt, és kompatibilitást ellenőriz, pl.:
  ```
  int x; double y = 2, string z;
  x = (int)y; // engedélyezett
  z = (string)y;
  // hiba, mert int és string nem kompatibilisek
  ```
- tetszőleges primitív típuskonverzióra a `Convert` osztály statikus metódusai használhatóak, illetve szövegre történő konverzió több módon is elvégezhető:
  ```
  int x; double y = 2, string z;
  x = Convert.ToInt32(y);
  z = Convert.ToString(y); // z = y.ToString();
  x = Int32.Parse(z); // x = Convert.toInt32(z);
  ```


C# alapismeretek
Felsorolási típus

- A *felsorolási típus* (`enum`) értékek egymásutánja, ahol az értékek egész számoknak felelődnek meg (automatikusan 0-tól sorszámozva, de ez felüldefiniálható)
  - pl.: `enum Munkanap { Hétfő, Szerda = 2, Csütörtök}`
  - a hivatkozás a típusnéven át történik, pl.:
    `Munkanap mn = Munkanap.Hétfő`
  - egy változó több értéket is eltárolhat, így több értékre is igaz lehet, pl.:
    `Munkanap mn = Munkanap.Hétfő | Munkanap.Szerda`
  - többágú elágazásban használható feltételként
  - ez is egy osztály a `System` névtérben:
    `public abstract class Enum : ValueType, …`

C# alapismeretek
Elemi osztály

- Az *elemi osztály* (*struct*) egy egyszerűsített osztály, amely:
  - mindig érték szerint kezelődik, a példány automatikusan megsemmisül a blokk végén
  - nem szerepelhet öröklődésben (sem ősként, sem leszármazottként), de implementálhat tetszőleges számú interfészt
  - automatikus őse a `System.ValueType` (ami leszármazottja a `System.Object`-nek, ugyanakkor definiálja az érték szerinti kezelést)
    ```
    [<láthatóság>] [<módosítók>] struct <név>
      [: <interfészek>] {
       <definíciók>
    }
    ```


C# alapismeretek
Elemi osztály

- További megkötések:
  - nem lehet alapértelmezett konstruktora, és destruktora (az érték szerinti kezelés miatt)
  - nem lehet az elemeit inicializálni (minden elem az alapértelmezett értéket kapja meg)
  - nem alkalmazhatóak az `abstract`, `virtual`, `sealed`, `protected` kulcsszavak
- Általában egyszerű, rekordszerű szerkezethez használjuk, amelyek érték szerinti kezelése, másolása nem rontja a program teljesítményét
  - ugyanakkor tetszőleges mértékig növelhetjük a tulajdonságaikat

C# alapismeretek
Elemi osztály

- pl.:

```
namespace My.Numbers {
   struct Racionalis // racionális szám osztálya
   {
      public int Szamlalo; // adattagok
      public int Nevezo;

      public Racionalis(int sz, int n)
      { // paraméteres konstruktor
         Szamlalo = sz; Nevezo = n;
         if (Nevezo == 0) Nevezo = 1;
      }
   }
}
```


C# alapismeretek
Referencia osztály

- A *referencia osztály* (*class*) a teljes értékű osztály, amely származtatásban is szerepelhet

```
[<láthatóság>] [<módosítók>] class <név>
  [: <ős>, <interfészek>]
{
   <definíciók>
}
```

  - csak egy őse lehet, de tetszőleges számú interfészt valósíthat meg
  - mezőit lehet közvetlenül inicializálni, vagy nulla paraméteres konstruktor segítségével
  - az öröklődés miatt lehet absztrakt osztály, és szerepelhetnek benne absztrakt és virtuális elemek

C# alapismeretek
Referencia osztály

- Cím szerint, felügyelt mutatók segítségével kezelt típusok, ezért külön példányosítani kell a **`new`** utasítással, és megfelelő konstruktor meghívásával
  - a felügyelt mutató alapértelmezetten **`null`** értéket vesz fel, és nem fordulhat elő, hogy rossz címre mutat, a referenciát levenni az objektumról szintén a **`null`** értékadással lehet
  - másoláskor csak a memóriacíme másolódik, hacsak nem manuálisan másolunk (**`Clone`**), ehhez meg kell valósítania az **`ICloneable`** interfészt
  - megsemmisítését a szemétgyűjtő végzi, de (az **`IDisposable`** interfészt megvalósító osztályoknál) lehetőségünk van előzetes törlés elvégzésére (**`Dispose`**), ekkor az objektum java része megsemmisül

C# alapismeretek
Referencia osztály

- Pl.:

```
namespace My.Numbers {
   class Racionalis {
   // most már referencia osztály
      public int Szamlalo; // adattagok
      public int Nevezo;

      public Racionalis(int sz, int n)
      { // paraméteres konstruktor
         Szamlalo = sz; Nevezo = n;
         if (Nevezo == 0) Nevezo = 1;
      }
   }
}
```


C# alapismeretek
Interfész

- Az *interfész* (*interface*) deklarációk halmaza, egy osztályfelület, amely nem példányosítható, csak arra szolgál, hogy osztályok közös tulajdonságait összefogja

```
[<láthatóság>] interface <név> {
   <deklarációk>
}
```

  - a többszörös öröklődés kiküszöbölésére szükséges
  - nem tartalmaz megvalósítást, azaz lényegében minden eleme absztrakt (de nem kell kiírni a kulcsszót)
  - tartalmazhat metódusokat és tulajdonságokat
  - minden eleme publikus, ezért nem adunk meg láthatóságot
  - a névkonvenció szerint az interfész nevét I betűvel kezdjük

C# alapismeretek
Interfész

- Pl.:

```
namespace My.Numbers {
   interface IDoubleCompatible {
   // valós típussal kompatibilis felület

      void FromDouble(double d);
      // csupán a metódus deklarációja szerepel,
      // a törzse nem, se láthatósága

      double ToDouble();
   }
}
```

C# alapismeretek
Interfész

- Pl.:

```
namespace My.Numbers {
    public class Complex : IDoubleCompatible {
        public double R, I; // publikus adattagok

        public Complex(double r, double i)
        { R = r; I = i; } // publikus konstruktor

        // interfész megvalósítása:
        public void FromDouble (double d)
        { R = d; I = 0; }
        public double ToDouble() { return R; }
    }
}
```


C# alapismeretek
Tömbök

- A tömbök is objektumok, a `System.Array`, vagy leszármazottjának példányai, emiatt referenciaként tárolt és intelligens (pl. a méretét a `Length` tulajdonsággal érhetünk el)
- Deklarációra több lehetőség is adott:
  - *`<típus>`*`[]` szokványos egydimenziós tömb
  - *`<típus>`*`[][]` szokványos egymásba ágyazott tömb
  - *`<típus>`*`[,]` kétdimenziós tömb
- Definícióhoz létre kell hozni a tömböt a méret megadásával, vagy az értékek megadásával, pl.:

```
int[] t = new int[4]; // minden eleme 0 lesz
int[] t = new int[] {1, 2, 3, 4}; // t.Length == 4
int[,,] m = new int[10,5,2]; // 3 dimenziós
```


C# alapismeretek
Vezérlési szerkezetek

- A szekvenciapont a `;`
- Ugrás: `goto` *`<címke>`*`;`
- Elágazások:
  - kétágú:

```
if(<feltétel>)
    <mag>;
else
    <mag>;
```

    - a feltétel logikai típusú
    - a csellengő `else` mindig az utolsó elágazáshoz tartozik
    - rövidítve (használható értékadáshoz is):
      `<feltétel>` `?` *`<mag>`* `:` *`<mag>`*`;`

C# alapismeretek
Vezérlési szerkezetek

- többágú:

```
switch(<változó>){
    case <konstans> :
        <mag>; break; // kilépés az elágazásból
    case <konstans> :
        <mag>;
        goto case <konstans> // ugrás címkére
    //…
    default: // alapértelmezett ág
        <mag>; break;
}
```

  - alkalmazható egész és szöveg változókra
  - alapértelmezett (`default`) ág nem kötelező
  - a lezárás (`break`), vagy továbbadás (`goto`) kötelező

C# alapismeretek
Vezérlési szerkezetek

- Ciklusok:
  - számláló:

```
for (<inicializálás>; <feltétel>; <léptetés>)
    <mag>;
```

  - előtesztelő: `while (`*`<feltétel>`*`)` *`<mag>`*`;`
  - utántesztelő: `do {` *`<mag>`*`; } while (`*`<feltétel>`*`);`
  - felsoroló:

```
foreach(<deklaráció> in <kifejezés>) <mag>;
```

    - egy gyűjtemény értékein tud végighaladni
    - kifejezés értékének kell `GetEnumerator()` metódus
  - ciklusból kilépés (`break`), feltétel kiértékeléshez ugrás (`continue`) lehetséges

C# alapismeretek
Operátorok

- Az operátorok olyan függvények, amelyek speciális meghívással rendelkeznek, vezérlőkaraktereken, vagy kulcsszavakon keresztül
  - a meghívás rögzített formában történik (*prefix*, *postfix* vagy *infix* jelölés mellett)
  - az operátorok *precedenciá*val rendelkeznek, amely halmozás esetén megszabja a hívási sorrendet
  - az operandusok száma minden esetben rögzített, vannak egy-, két-, illetve háromoperandusú műveletek, és ez rögzített minden operátorhoz (a `+` és `–` operátortoknak van egy-, illetve kétoperandusú változata is)
- A beépített típusok előre definiált operátorokkal rendelkeznek

C# alapismeretek
Operátorok

- Aritmetikai:
  - pozitivitás (`+a`), negáció (`-a`)
  - értéknövelés (`a++`, `++a`), értékcsökkentés (`a--`, `--a`)
  - összeadás (`a + b`), kivonás (`a - b`), szorzás (`a * b`), osztás (`a / b`), maradékképzés (`a % b`)
  - túlcsordulás ellenőrzés (`checked`, `unchecked`)
- Értékadás:
  - egyszerű (`a = b`)
  - összetett (`a += b`, `a -= b`, `a *= b`, `a /= b`, `a %= b`, `a <<= b`, `a >> b`, `a &= b`, `a |= b`, `a ^= b`)
  - feltételes (`a ? b : c`)

C# alapismeretek
Operátorok

- Függvényhívás (`a()`)
- Logikai:
  - érték összehasonlítás (`a < b`, `a > b`, `a <= b`, `a >= b`, `a == b`, `a != b`)
  - tagadás (`!a`), és (`a && b`), vagy (`a || b`)
- Memória:
  - memóriajelenlét ellenőrzés (`??`)
  - referencia (`&a`), dereferencia (`*a`)
  - taghivatkozás (`a.b`), mutató általi taghivatkozás (`a->b`)
  - méretlekérdezés (`sizeof a`, `sizeof(`*`típus`*`)`)
  - memóriaterület lefoglalás (`new` *`típus`*)

C# alapismeretek
Operátorok

- Indexelés (`a[b]`)
- Típuskezelés:
  - típusazonosítás (`is`), típuskezelés (`as`)
  - explicit típuskonverzió (`(`*`típus`*`) a`)
  - típusazonosítás (`typeof a`, `typeof(`*`típus`*`)`)
- Bitenkénti:
  - eltolás balra (`a << b`), eltolás jobbra (`a >> b`)
  - komplementer képzés (`~ a`), bitenkénti és (`a & b`), bitenkénti vagy (`a | b`), különbségképzés (`a ^ b`)

C# alapismeretek
Operátorok precedenciája

- Az operátorok precedenciája meghatározza, a műveletek halmozása esetén milyen sorrendben történik a végrehajtás
  - a precendencia típusfüggetlen, és rögzített
  - a magasabb precendenciájú hajtódik előbb végre
  - zárójelek használatával befolyásolhatjuk a végrehajtási sorrendet
- C# precedenciák:
  1. `++` (postfix), `--` (postfix), `[]`, `()`, `.`, `new`, `typeof`, `checked`, `unchecked`
  2. `+` (unáris), `-` (unáris), `!`, `~`, `++` (prefix), `--` (prefix), (`(`*`típus`*`)`)
  3. `*`, `/`, `%`
  4. `+`, `-`

C# alapismeretek
Operátorok precedenciája

5. `<<`, `>>`
6. `>`, `<`, `>`, `<`, `is`, `as`
7. `==`, `!=`
8. `&`
9. `^`
10. `|`
11. `&&`
12. `||`
13. `?:`
14. `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `<<=`, `>>=`, `&=`, `|=`, `^=`

- Pl.:
  `a * b - c == 7` jelentése: `((a * b) - c) == 7`
  `c = a == b % 2` jelentése: `c = (a == (b % 2))`
  `++a++` jelentése: `++(a++)`

C# alapismeretek
Kivételkezelés

- A program által kiváltott bármilyen nem szabályos tevékenység kivételként jelenik, és lekezelhető
- A kivétel általános osztálya az `Exception`, de a műveletek rendszerint ennek valamely speciálisabb változatát keltik
- Kivételt kezelni egy kivételkezelő (`try-catch-finally`) szakasszal tudunk, amelyben meg kell adnunk az elfogandó kivétel típusát, és kivétel esetén lefuttathatunk egy megadott utasítássorozatot:

```
try { <kivételkezelt utasítások> }
catch (<elfogott kivétel típusa>){
   <kivételkezelő utasítások>
}
finally { <mindenképp lefuttatandó utasítások> }
```

C# alapismeretek
Kivételkezelés

- Kivételkezelő szakaszt bármely metóduson belül elhelyezhetünk a programban
  - ha a **`try`** blokkban kivétel keletkezik, akkor a vezérlés a **`catch`** ágra ugrik, az utána következő utasítások így nem futnak le
  - a program ellenőrzi, hogy a kivétel típusa egyezik-e, vagy speciális esete a **`catch`**-ben megadottnak, különben tovább dobja a kivételt
  - ha elfogta a kivételt, akkor futtatja a **`catch`** ág utasításait
  - a **`finally`** blokk használata nem kötelező, de amennyiben van, úgy az abban lévő utasításokat akkor is futtatja, ha keletkezett lekezelt kivétel, és akkor is, ha nem

C# alapismeretek
Kivételkezelés

- Lehetőségünk van különböző típusú kivételek elfogására is, amennyiben több **`catch`** ágat készítünk a szakaszhoz
  - a kivétel típusát sorban egyezteti az ágakon, és az első találat ágát futtatja
  - amennyiben biztosan el akarunk kapni bármilyen kivételt, kapjuk el az általános **`Exception`** típust is
  - pl.:

```
try { // kivételkezelt utasítások
   …
} // kivételkezelő ágak:
catch (ArgumentException) { … }
catch (NullReferenceException) { … }
catch (Exception) { … }
```


C# alapismeretek
Kivételkezelés

- A kivételek üzenettel rendelkeznek, amelyet a kivétel **`Message`** tulajdonságán keresztül kérhetünk le
- Kivételt mi is kiválthatunk tetszőleges pontján a programnak, amelyet egy őt meghívó metódusban kezelni tudunk
- Kivételt kiváltani a **`throw`** utasítással tudunk:

```
throw new <kivétel típusa>(<kivétel szövege>);
```

  - az utasításra a program futása megszakad, és a következő kivételelfogó utasításra kerül a vezérlés
  - a kivétel típusa lehet egy beépített kivételtípus (pl. **`ArgumentException`**, **`IndexOutOfRangeException`**, **`Exception`**), valamint mi is definiálhatunk kivétel típusokat
  - a kivétel szövegét nem kötelező megadni

C# alapismeretek
Generikus típusok

- Generikus programozásra futási időben feldolgozott sablon típusok (*generic*-ek) segítségével van lehetőség
  - a sablon fordításra kerül, és csak a futásidejű fordításkor helyettesítődik be a konkrét értékre
  - a sablonos osztályt szokás szervernek, a példányosítását kliensnek nevezni
  - pl.:

```
class GenericClass<T>{
   public T item; // használható a T típusként
}
//...
GenericClass<int> gc = new GenericClass<int>();
gc.item = 1;
```


C# alapismeretek
Generikus típusok

- Mivel szigorú típusellenőrzés van, ezért fordítási időben a sablonra csak az **`Object`**-ben értelmezett műveletek használhatóak, ezt a műveletkört növelhetjük megszorításokkal
  - a megszorítás (**`where`**) korlátozza a típus behelyettesítési értékeit, és ezáltal bővíti az alkalmazható metódusok és tulajdonságok körét
  - a behelyettesítéskor így csak az adott típus, vagy annak leszármazottja szerepelhet
  - korlátoznál csak egy osztály, és mellette tetszőleges számú interfész adható meg, ekkor a behelyettesített típusnak valamennyit meg kell valósítania, ez fordítási időben van ellenőrizve

C# alapismeretek
Generikus típusok

- Pl.:

```
class GenericClass<T> where T
    : Control, ICloneable, IDisposable {
    // T-re és példányaira használható lesz az
    // őszosztály, és a fenti interfészek összes
    // művelete
}

GenericClass<Button> gc;
gr = new GenericClass<Button>();
// példányosításkor csak a fentieket megvalósító
// osztály szerepelhet, különben fordítási hiba
```