.NET

C# 1.0, 2.0

Albert István
ialbert@aut.bme.hu
QB. 221, 463-1664

BME, Automatizálási és Alkalmazott Informatikai Tanszék

BME AUT
Automatizálási és
Alkalmazott
Informatikai Tanszék

C# 1.0

A fájlok struktúrája

- Nincs header fájl: „definíció a deklarálásnál"
  - mint a Visual Basic, Pascal, Modula, Java
- A kód és deklaráció egy helyen
  - a kód konzisztens és könnyen kezelhető
  - csapatmunka esetén sokkal közérthetőbb
  - deklaráció metaadatokon át jobban elérhető
- Feltételes fordítás (prepocesszor) van, de makró nincs

A típus rendszer

- A .NET CLS típus rendszerére épül
- Közvetlen elérés a .NET típus rendszeréhez
- Minden Objektum
  - Látszólag minden objektum a System.Object-ből származik

```
Console.Write( 256.256.ToString() );
```

- Megkülönböztetés érték és a referencia típusok között
  - érték: egyszerű típusok, enum, struct
  - referencia: interfész, osztály, tömb, sztring, ...

Az egyszerű típusok

- Egész típusok
  - **`int, uint (32 bit), long, ulong (64 bit)`**
  - **`byte, sbyte`** (8 bit)**`, short, ushort`** (16 bit)
- IEEE lebegőpontos típusok
  - **`double`** (pontosság: 15-16 számjegy)
  - **`float`** (pontosság: 7 számjegy)
- Pontos szám típus
  - **`decimal`** (28 szignifikáns számjegy)
- Karakter típusok
  - **`char`** (egy unicode karakter, **nem** cserélhető fel az **`int`**-tel)
  - **`string`** (gazdag funkcionalitás, referencia szerinti típus)
- Boolean típus
  - **`bool`** (külön típus, **nem** cserélhető fel az **`int`**-tel)

A felsorolt típus (enum)

- Nevesített elemek használata a számok helyett
- Erősen típusos
- Jobban használható a "Color.Blue" mint a "3"
  - Jobban olvasható, könnyebben kezelhető
  - Még mindig olyan könnyű, mint az int
- Mögötte egész számot tárol
  - Tetszőleges integer típussal használható
  - Az egyes tagokhoz rendelt értékek explicit megadhatók

Statementek

- C-szerű: Flow Control és Loop
  - `if (<bool expr>) { ... } else { ... }`
  - `switch(<var>) { case <const>: ...; }`
  - `while (<bool expr>) { ... }`
  - `for (<init>;<bool test>;<modify>) { ... }`
  - `foreach( típus <var> in <var> ) { ... }`
  - `do { ... } while (<bool expr>);`
- Nem C szerű:
  - `lock(<object>){ ... };`
    - nyelvvel járó kritikus szekció szinkronizáció lehetőség
  - `checked { ... }; unchecked { ... };`
  - `checked ( ... ); unchecked ( ... );`
    - uncheck: aritmetikai műveletek hibáinak kezelése, nullával osztás NEM

Interfészek

- Metódusok és tulajdonságok absztrakt definíciója
  - Komponens alapú gondolkozás
- Struktúra és szemantika definiálása speciális célra
  - Interfészek = szerződések
- Támogatja a többszörös interfész implementációt

```
interface IPersonAge
{
    int YearOfBirth { get; set; }
    int GetAge();
}
```

Osztályok

- Kód és adat implementációja
- Interfészeket implementálhat
- Legfeljebb egy osztályból örökölhet
- Osztályok tartalmazhatnak
  - mezők: tag-változó
  - tulajdonságok: érték elérése get/set metódus párokon át, lehet csak olvasható/írható is !
  - metódusok: funkcionalitás
  - speciális: event, indexer, delegate, ...
- Fizikai struktúra (assembly=szerelvény)
- Logikai struktúra (namespace)

Struktúrák = érték típusok

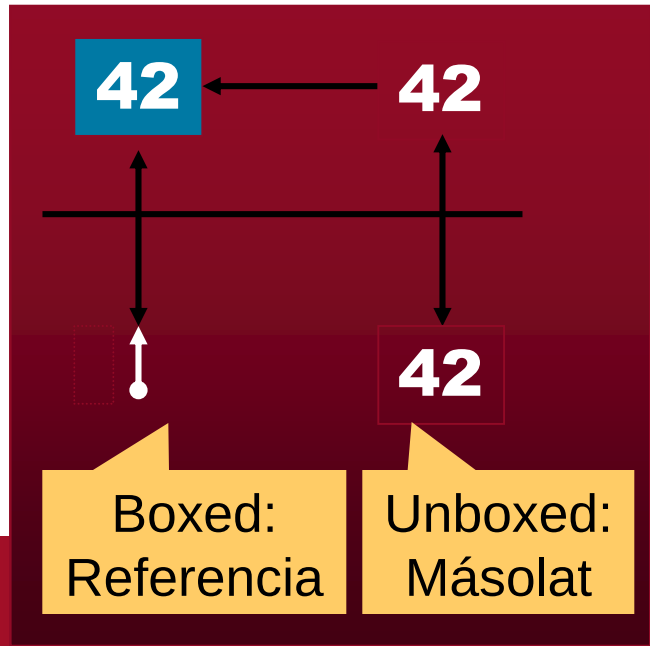
- Adat és kód csoportosítva
  - Hasonló az osztályhoz, de:
    - nincs öröklődés, csak interfész implementáció
    - *mindig érték szerint átadva (másolás)*
  - Osztály vs. struktúra
    - struktúra ⇨ Könnyű adat kontainer, érték típus
    - osztály ⇨ Gazdag objektum referenciákkal, referencia típus
- „C++-szal ellentétben a `struct` nem egy olyan `class` ahol minden `public`!”

```
struct Point
{
    double X;
    double Y;
    void MoveBy(double dX, double dY)
    { X+=dX; Y+=dY; }
}
```

Boxing és Unboxing

- Érték szerinti típusok dobozolódnak
- "Boxing" lehetővé teszi az érték típusú objektumok referenciakénti átadását
- Az értéktípust a CLR a heapre másolja és kiegészíti típus információval, így referencia típusként működik

```
object a = 42;
```



Névterek

- Minden definíciónak namespace-ben kell lennie
  - Elkerüli a névütközéseket
  - Egyszerűbbé teszi az API-k átlátását
- Egymásba ágyazható (ajánlott)
- Osztályok, típusok csoportosítása
- A **`namespace`** kulcsszóval deklarálhatóak
- Hivatkozás a **`using`** kulcsszóval

Tulajdonságok

- Mezőeléréshez metódus rendelése
- Felhasználása
  - csak olvasható tagok implementációja
  - elérés validációja
  - számolt vagy képzett értékek
  - felfedhet interfészen keresztül örökölt értékeket

```
string Name
{
    get { return name; }
    set { name = value; }
}
```

Indexerek

- Konzisztens mód a gyűjtemények elemeinek elérésére
- Indexelt elérést biztosít a tartalmazott objektumokhoz
- A „tulajdonságok”-ra épül, de paraméterezhető
- Az index tetszőleges típusú lehet

```
object this[string index]
{
  get { return Dict.Item( index ); }
  set { Dict.Add( index, value ); }
}
```

Operátorok

- C-szerű:
  - Logikai: `&& || ^`
  - Aritmetikus: `* / + - % << >>`
  - Relácionális: `== != < > >= <=`
- C-hez hasonló:
  - Integer: `&` és `|` bináris AND/OR
  - Bool: `&` és `|` logikai operátor teljes kiértékeléssel

    ```
    1 == 0 && isValid( a )
    1 == 0 & isValid( a ) ⇨ isValid hívás
    ```
- Nem C-szerű:
  - is: Runtime-Type tesztelése
  - as: Type-Cast, nincs exception, null lesz
  - typeof: Runtime-Type lekérése

Operátor felüldefiniálás

- Legtöbb operátor felüldefiniálható
  - aritmetikai, relációnális, kondícionális és logikai operátorok
- Nem felüldefiniálható:
  - Hozzárendelés operátor
  - Speciális operátorok (`sizeof, new, is, typeof`)

```
Total static operator + ( int Amount, Total t )
{
      t.total += a;
      return t;
}
```

Operátor felüldefiniálás a Frameworkben

- String
- Decimal, DateTime, TimeSpan
- Point, Rectangle
- SQLString, SQLInt, ...

Tömbök

- Nullától indexelt, erősen típusos, a .NET `System.Array` osztályán alapul
- Deklaráláskor típus és dimenziók fixek, nincs korlát
  - `int[] EgyDim;`
  - `int[,] KetDim;`
  - `int [][] Beagyazott;`
- Létrehozás **`new`**-val (korlátokkal vagy inicializálással)
  - `EgyDim = new int[20];`
  - `KetDim = new int[,] { {1,2,3}, {4,5,6} };`
    - `KetDim[0,0] == 1 !`
  - `Beagyazott = new int[1][];`
  - `Beagyazott[0] = new int[]{1,2,3};`
    - `Beagyazott[0][2] == 3 !`

Hozzáférési jogok

- Hasonlít a C++ modellre
  - **`public`** ⇨ mindenki hívhatja/elérheti
  - **`protected`** ⇨ leszármazottak hívhatják/érhetik el
  - **`private`** ⇨ csak pontosan ennek az osztálynak a tagjai
- Kibővíti a C++ modellt
  - **`sealed`** ⇨ Nemlehet belőle származtatni
  - **`internal`** ⇨ publikus elérés de csak szerelvényen belül
  - **`protected internal`** ⇨ szerelvényen belül és a leszármazott típusokra

Típus-verzió kezelés

- Valós életbeli probléma:
  - Két ember két helyen két szoftverrészletet ír
  - A osztálya B osztályán alapul
  - B implementál egy metódust: **`CalcResult`**
  - Következő verzió, A is készít egy ilyet: **`CalcResult`**
    - **Kérdés:** Akarja B, hogy a CalcResult-t A felülírja ?
    - **Válasz:** Nem jellemző …
- Megoldás: a szándékot jelezni kell öröklésкor, egyébként figyelmeztet a fordító

Virtual

- Futási időben dől el, hogy melyik függvény (vagy tulajdonság) hajtódik végre – a származtatási láncban az utolsó
- Az *override* kulcsszóval lehet felüldefiniálni a virtuális függvényeket
- Nem használható a *static, abstract és override* kulcsszavakkal együtt !

Override

- Az override megváltoztatja az eredeti virtuális vagy absztrakt fv (vagy tulajdonság) implementációját
- A felüldefiniált fv-nek virtuálisnak vagy absztraktnak kell lennie
- Az *override* nem változtathatja meg az elem elérési szintjét (public, protected, …)
- Nem használható a *new, static, virtual, abstract* kulcsszavakk együtt !

A new módosítószó

- Teljesen új függvényt hoz létre, megszakítja a virtuális fv-láncot
- Konstanst, mezőt, tulajdonságot, függvényt vagy típust lehet újként definiálni (azaz a régit elfedni)
- Nem használható az *override*-dal együtt

Összefoglalva

A virtuálissal lehet megcsinálni, hogy mindig a származott osztály függvénye hívódjon meg, még ha az alaposztályból hívsz, akkor is. A *virtuális* függvényt az *override*-dal lehet felüldefiniálni.

Egyébként új függvény létrehozására (a szignatúra csak véletlenül azonos) a *new* használandó (beágyazott osztályra, stb-re is).

Lehet *new virtual*: ilyenkor egy új lánc indul, az eddigi osztályok az előző virtuális fv-t hívják meg, az ez után következők viszont az ebben a láncban lévő utolsót.

Delegate - metódusreferencia

- Funkció pointerek
  - statikus metódusokra
  - Tagfüggvényekre – eltárolja a this referenciát is
- Deklarálás – mint egy metódus fejléc

```
public delegate void ClickHandler
        (object sender, EventArgs  e );
```

- A ClickHandler típusként használható
  - Lehet belőle példányt létrehozni stb.
- Erősen típusos, biztonságos
  - Ellenőrizhető

Események

- Nyelvbe épített esemény modell
- Deklarálás – a delegate-re alapozva

```
event ClickHandler OnClicked;
```

- Eseményt hívni *csak* a tulajdonos osztályból lehet:

```
...
OnClicked(this,PointerLocation);
...
```

- Eseményre feliratkozás kezelőfüggvény hozzáadásával

```
Button.OnClicked += MyClickHandler;
```

```
class Test {
  static void Logger( string s ) {
       Console.WriteLine( s );
  }
  public static void Main() {
       MyClass instance = new MyClass();
  // nem fordul: eventnél nem lehet ilyet csinálni
   // csak delegatenél
       instance.Log = new MyClass.LogHandler( Logger )

   // ez jó
       instance.Log += new MyClass.LogHandler(.)
       instance.Process();
} }
```

Attribútumok

- Lehetővé teszi, hogy bővítsük a kódelemzést
  - > Útmutatás a futtatókörnyezetnek

```
[Transaction(TransactionOption.Required)]
class MyBusinessComponent { ... }
```

- Mindennek lehet attribútuma
- Programból lekérdezhető
- Deklaratív programozási irány

Bővítés egyedi attribútummal

- Implementáljuk az osztályt *Attribute névvel
- Felderítés a .NET reflection segítségével

```
typeof(<object-expr>)
   . GetCustomAttributes();
```

Attribútumok felhasználása

- Tulajdonságok kategorizálása
- Metódushívások tranzakcióbafoglalása
- Biztonsági rendszer
- Sorosítás
- Kód konfigurálás
- XML, Web szolgáltatások
- Natív funkciók elérése
- …

Konstans vagy csak olvasható mezők

konstans

Fordítás-idejű kiértékelés

Mindig statikus

csak olvasható

Futásidejű kiértékelés

Statikus vagy tag változók

```
class Math
{
  public const double Pi = 3.14;
}

class Color
{
  public static readonly Color Red   = new Color(...);
  public static readonly Color Blue  = new Color(...);
  public static readonly Color Green = new Color(...);
}
```

Statikus konstruktorok

- Globális változók inicializálása
- Az első példány létrejötte illetve az első statikus függvényhívás, változóelérés előtt meghívódik

```
class CharInfo
{
   static bool[] isAlpha;
   static int Min = -9999;
   static int Max = 9999;

   static CharInfo() {
      isAlpha = new bool[256];
      for (int c = 'A'; c <= 'Z'; c++) isAlpha[c] = true;
      for (int c = 'a'; c <= 'z'; c++) isAlpha[c] = true;
   }
}
```

Destruktorok

- Mark-and-sweep szemétgyűjtés – nemdeterminisztikus destruktorok
  - a végrehajtás ideje nem ismert
  - a sorrend nem ismert
  - a szál (!) nem ismert
- A destruktor *csak* a külső hivatkozásait engedheti el
- Destruktorokkal rendelkező objektumok tipikusan implementálják az IDisposable interfészt

Destructorok

- Az Object.Finalize nem elérhető C#-ból

```
public class Resource: IDisposable
{
   ~Resource() {...}
}
```

```
public class Resource: IDisposable
{
   protected override void Finalize() {
      try {
         ...
      }
      finally {
         base.Finalize();
      }
   }
}
```

Destructorok

```
public class Resource: IDisposable
{
   public void Dispose() {
      Dispose(true);
      GC.SuppressFinalize(this);
   }

   protected virtual void Dispose(bool disposing) {
      if (disposing) {
         // Dispose dependent objects
      }
      // Free unmanaged resources
   }

   ~Resource() {
      Dispose(false);
   }
}
```

Volatile mezők

```
class Foo
{
    public int x;
    public int y;
}
```

```
foo.y = 5;
foo.x = 1;
```

```
if (foo.x == 1) {
    Console.WriteLine(foo.y);
}
```

Nem biztos, hogy az 5-öt írja ki !

A nem-volatile mezők írási és olvasási sorrendje felcserélődhet, ha egy szálon nem látszik a különbség.

Volatile mezők

```
class Foo
{
    public volatile int x;
    public int y;
}
```

Volatile mező !

Mindig 5-öt ír ki !

```
foo.y = 5;
foo.x = 1;
```

```
if (foo.x == 1) {
    Console.WriteLine(foo.y);
}
```

A volatile írás nem tehető későbbre.
A volatile olvasás nem tehető előbbre.

Mutatók (pointer)

- Gazdag .NET CLS támogatás
  - Referencia szerinti átadás ref kulcsszóval

```
void increment(ref int value, int by)
```

  - Kimeneti paraméterek az out kulcsszóval

- A mutatók használata általában szükségtelen
  - De használható ha teljesítmény okokból vagy más kóddal való együttműködés miatt szükséges

Beépített gyűjtemények, foreach

- Egyszerű támogatás gyűjtemények iterálására
  - használható tömbökre és más gyűjteményekre
- Használható saját osztálynál is
  - interfész implementálása
  - visszaadott objektumot implementálja az interfészt
    - (Reset(), MoveNext(), Current)

```
Point[] Points = GetPoints();
foreach( Point p in Points )
{
    MyPen.MoveTo(Point.x,Point.y);
}
```

Kivétel kezelés

- (`try`) próbáljuk meg futtatni ezt ...
- ...ha hiba van kapjuk (`catch`) el amit le tudunk kezelni...
  - Típus szűrő
- ... (`finally`) végül takarítsunk mindkét esetben

Megjegyzések XML-ben

- Konzisztens út a kódból való dokumentáció készítésére
- "///" megjegyzések exportálása
- Fordító generálja az XML dokumentációt (/doc)
- Előredefiniált sémákkal rendelkezik
  - Például XSLT-vel formázható

Különbségek: C++ és C#

- A C# nagyon hasonlít a C/C++-ra
- Megszüntet sok hibalehetőséget:
  - szigorúbb típusellenőrzés, kevés type-cast
  - nincs „átesés” a **`switch`**-ben
  - a boolean kifejezések erősen típusosak
  - jobb elérés védelem, nincs trükközés a headerekkel
  - Ritkán használunk mutatókat
  - nem kell üldözni a memória szivárgást

C# 2.0

BME AUT
Automatizálási és
Alkalmazott
Informatikai Tanszék

Tartalom

- CLR 2.0 és C# újdonságok
  - **Sablonok (generics)**
  - Név nélküli metódusok (anonymous methods)
  - Nullable types
  - Iterátorok
  - Egyebek
  - Ko- és kontra variancia

Generikus típusok

CLR

- Írjunk olyan kódot, ami több típussal használható
  - Tipikusan gyűjtemény osztályok
  - .NET 1.x megoldás:
    - objectként kezelés
    - saját gyűjtemény osztály
  - Így működnek
    - Gyűjtemény osztályok (pl. ArrayList)
    - IComparer, …
- Ez messze nem tökéletes megoldás… ☹

Generikus típusok

- Az élet generikus típusok nélkül
  - Érték típusoknál be és kidobozolás (teljesítmény)
  - Object-ként kezelés
    - Castolni kell, csak futási időben derül ki, ha hiba van

```
ArrayList list = new ArrayList();
list.Add( new Person() );
…
int i = (int)list[0];
```

  - ArrayList, stb: nincs kikényszerítve, hogy ne keveredjenek az objektumok

```
ArrayList list = new ArrayList();
list.Add( new Person() );
list.Add( 12 );
```

Generikus típusok

- Mik a generikus típusok ?
  - Olyan kód, ami paraméterként típust kap
  - Generikus osztályok, metódusok, interfészek stb.
- Előnyök
  - Kód újrafelhasználhatóság
  - Típusellenőrzés
  - Dobozolás és kasztolás megszüntetése
  - Kevesebb, karbantarthatóbb kód

Sablonok

```
public class List<T>
{
   private T[] elements;
   private int count;

   public void Add(T element) {
      if (count == elements.Length) Resize(count * 2);
      elements[count++] = element;
   }

   public T this[int index] {
      get { return elements[index]; }
      set { elements[index] = value; }
   }

   public
      get
   }
}
```

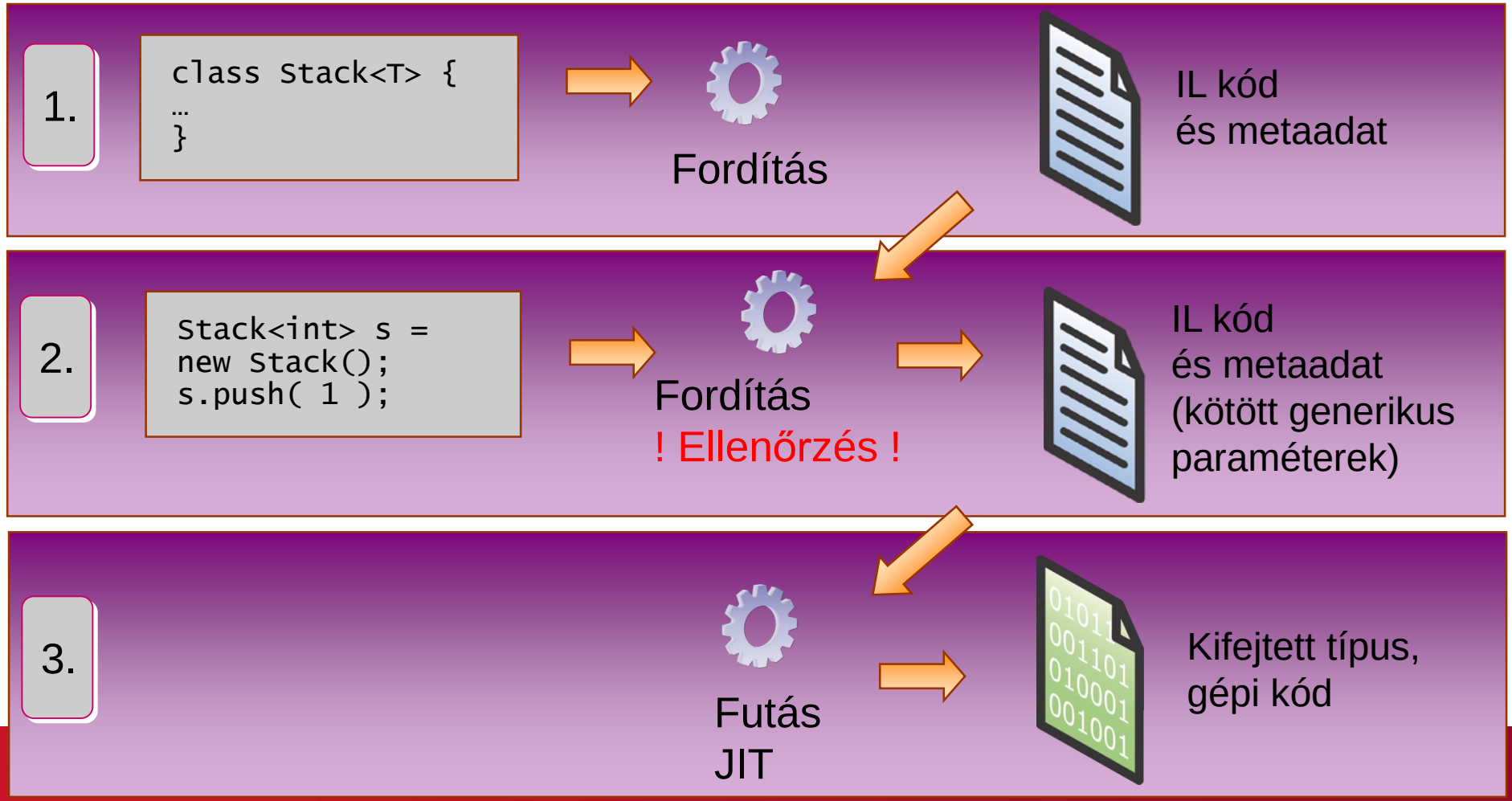
```
List<int> intList = new List<int>();

intList.Add(1);             // No boxing
intList.Add(2);             // No boxing
intList.Add("Three");       // Compile-time error

int i = intList[0];         // No cast required
```

Generikus típusok

- Hogyan működnek a generikus típusok? (például a C++ sablonokkal ellentétben)
  - Futásidőben készülnek el, nem fordításidőben
  - Deklaráláskor (fordítás időben) kerülnek ellenőrzésre, nem futásidőben
  - Érték- és referencia típusokkal is működnek
  - Teljes futásidejű típusinformáció (reflexió) támogatás
  - A JIT fordító a sablon kódot specifikus hivatkozásokra cseréli
  - Alkalmazás tartomány specifikus
- Érték típusok
  - A JIT fordító minden felhasznált értéktípushoz egy külön osztályt készít
- Referencia típusok
  - A JIT fordító ugyanazt a típust használja mindegyik referencia típushoz

Generikus típusok és a CLR

- Hogyan működnek a generikus típusok (például a C++ sablonokkal ellentétben) ?



Generikus típusok

- Generikus osztályok, struktúrák, interfészek, metódusreferenciák, metódusok
- T-t használhatjuk int-tel, float-tal, stb.

```
public struct Point<T>
{
    public T X;
    public T Y;
}

Point<float> point;
point.X = 1.2f;
point.Y = 3.4f;
```

- Kényszerek is használhatók a típusokon

Generikus típus kényszerek

- Kényszer kategóriák
  - Érték- vagy referencia típus (struct, class)

```
class MyClass<T> where T: struct
```

  - Interfész vagy alaposztály
  - Legyen default konstruktora: new()

```
class Dictionary<K,V>: IDictionary<K,V>
   where K: IComparable<K>
   where V: IKeyProvider<K>, IPersistable, new()
{
   public void Add(K key, V value) { ... }

   public V Find(K key) {
        if ( c.CompareTo( key ) ) { … }
   }

}
```

Generikus metódusok

- Az argumentum/visszatérési érték típusa paraméter
- Statikus is lehet
- Nem kell megadni a típust, kikövetkezteti

```
public class MyClass
{
        public static void Swap<T>( ref T lhs, ref T rhs )
        {
                T temp;
                temp = lhs;
                lhs = rhs;
                rhs = temp;
        }
}
…
int a = 2, b = 3;
MyClass.Swap<int>( ref a, ref b );
MyClass.Swap( ref a, ref b ); // kikövetkezteti a típust
```

Sablonok a keretrendszerben

- Gyűjteményoszályok
- Gyűjteményinterfészek
- Gyűjteményalaposzályok
- Egyéb osztályok
- Reflexió

LinkedList<T>
List<T>
Dictionary<K,V>
SortedDictionary<K,V>
Stack<T>
Queue<T>

IList<T>
IDictionary<K,V>
ICollection<T>
IEnumerable<T>
IEnumerator<T>
IComparable<T>
IComparer<T>

Collection<T>
KeyedCollection<T>
ReadOnlyCollection<T>

Nullable<T>
EventHandler<T>
Comparer<T>

Sablon metódus referenciák

- **Action** – Végrehajt egy műveletet az adott objektumon

```
public sealed delegate void Action<T>( T obj );
```

- **Predicate** – Feltételek ellenőrzéséhez

```
public sealed delegate bool Predicate<T>( T obj );
```

- **Converter** - Konvertáláshoz

```
public sealed delegate U Converter<T, U>( T from );
```

- **Comparison** – Összehasonlít két azonos típusú értéket

```
public sealed delegate int Comparison<T>( T x, T y );
```

A List<T> metódusai

- **ForEach** – Egy adott műveletet végrehajt minden objektumon

```
public void ForEach( Action<T> action );
```

- **Find/FindIndex/FindAll/FindLast –** Szűrés, keresés

```
public List<T> FindAll( Predicate<T> match );
```

- **ConvertAll** – Konvertál egy másik listára

```
public List<U> ConvertAll<U>( Converter<T, U> converter );
```

- **TrueForAll** – Ellenőrzi az összes elemet

```
public bool TrueForAll( Predicate<T> match );
```

Tartalom

- CLR 2.0 és C# újdonságok
  - Sablonok (generics)
  - **Név nélküli metódusok (anonymous methods)**
  - Nullable types
  - Iterátorok
  - Egyebek
  - Ko- és kontra variancia

Névtelen metódusok

```
class MyForm : Form
{
   ListBox listBox;
   TextBox textBox;
   Button addButton;

   public MyForm() {
      listBox = new ListBox(...);
      textBox = new TextBox(...);
      addButton = new Button(...);
      addButton.Click += delegate {
         listBox.Items.Add(textBox.Text);
      };
   }
}
```

Névtelen metódusok

- Metódus referencia létrehozás helyett kód
- A metódusreferencia típusát automatikusan kitalálja
  - A kód lehet paraméterek nélküli
  - Vagy kaphat paramétereke
  - De a visszatérési értéknek meg kell egyeznie

```
button.Click += delegate { MessageBox.Show("Hello"); };
```

```
button.Click += delegate(object sender, EventArgs e) {
   MessageBox.Show(((Button)sender).Text);
};
```

Local variable capturing 1

```
static D[] F() {
      D[] result = new D[3];
      for (int i = 0; i < 3; i++) {
            int x = i * 2 + 1;
            result[i] = () => {
                        Console.WriteLine(x); };
      }
      return result;
}

static void Main() {
      foreach (D d in F()) d();
}

produces the output:
1
3
5
```

Local variable capturing 2

```
However, when the declaration of x is
moved outside the loop:
static D[] F() {
      D[] result = new D[3];
      int x;
      for (int i = 0; i < 3; i++) {
            x = i * 2 + 1;
            result[i] = () => {
                  Console.WriteLine(x); };
      }
      return result;
}
the output is:
5
5
5
```

Tartalom

- CLR 2.0 és C# újdonságok
  - Sablonok (generics)
  - Név nélküli metódusok (anonymous methods)
  - **Nullable types**
  - Iterátorok
  - Egyebek
  - Ko- és kontra variancia

Mi a hiba az alábbi kódban?

```
SqlDataReader rs = cmd.ExecuteReader();
while (rs.Read())
{
    int Age;
    Age = rs["Age"];
    Console.WriteLine(Age);
}
```

```
Age
---------
30
50
null
21
```

Nullable types

- System.Nullable<T>
  - Null tartalmú értéktípusok
  - Egy értéktípus ami a T-ből és egy bool-ból áll

```
public struct Nullable<T> where T: struct
{
    public Nullable(T value) {...}
    public T Value { get {...} }
    public bool HasValue { get {...} }
    ...
}
```

```
Nullable<int> x = new Nullable<int>(123);
...
if (x.HasValue) Console.WriteLine(x.Value);
```

Nullable Types

- A T? ugyanaz, mint a System.Nullable<T>

```
int? x = 123;
double? y = 1.25;
```

- A null literál használható

```
int? x = null;
double? y = null;
```

- Explicit és implicit konverziók

```
int i = 123;
int? x = i;          // int --> int?
double? y = x;       // int? --> double?
int? z = (int?)y;    // double? --> int?
int j = (int)z;      // int? --> int
```

Nullable Types

- Beépített és saját operátorok

```
int? x = GetNullableInt();
int? y = GetNullableInt();
int? z = x + y;
```

- Összehasonlító operátorok

```
int? x = GetNullableInt();
if (x == null) Console.WriteLine("x is null");
if (x < 0) Console.WriteLine("x less than zero");
```

- A ?? (default value) operátor

```
int? x = GetNullableInt();
int i = x ?? 0;
```

Nullable CLR támogatás - boxing

- Dobozolás (boxing):

```
int? x = null;
Console.WriteLine( x == null ); // true
object o = x;
Console.WriteLine( o == null ); // ???
    Ennek is igaznak kéne lennie!
```

- Megoldás: a JIT fordító más kódot generál dobozoláshoz nullable típusok esetén: a tárolt értéket dobozolja.

Nullable CLR támogatás - unboxing

- Kidobozolás (unboxing):

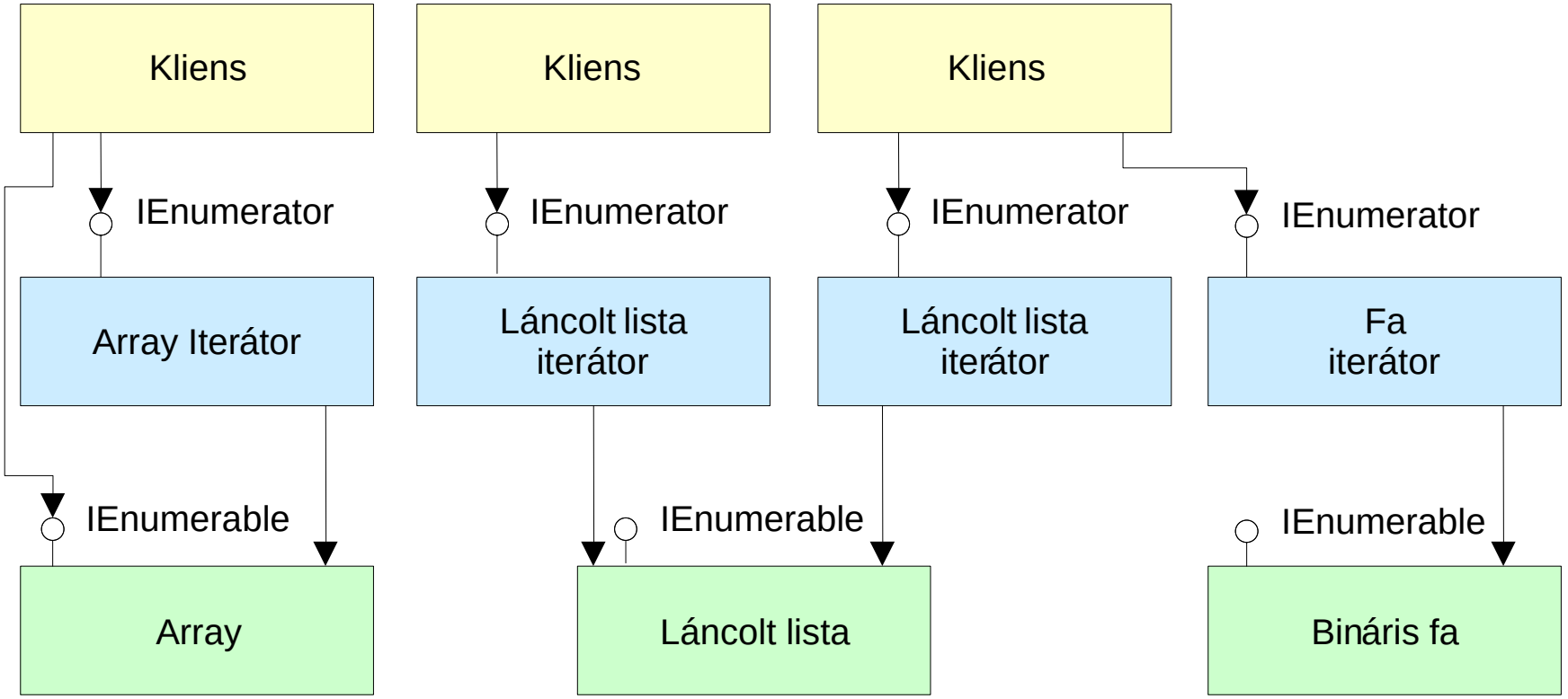
```
int x = 123;
object o = x;
int y = (int?)o;  // hiba
   Működnie kéne!
```

- Megoldás: JIT T kidobozolásakor T-t vagy Nullable<T>-t is vissza tud adni.

Tartalom

- CLR 2.0 és C# újdonságok
  - Sablonok (generics)
  - Név nélküli metódusok (anonymous methods)
  - Nullable types
  - **Iterátorok**
  - Egyebek
  - Ko- és kontra variancia

Iterátor minta áttekintés



Iterátorok

- A foreach az enumeration mintára épül
  - GetEnumerator() metódus

```
foreach (object obj in list) {
   DoSomething(obj);
}
```

```
Enumerator e = list.GetEnumerator();
while (e.MoveNext()) {
   object obj = e.Current;
   DoSomething(obj);
}
```

- foreach: egyszerű iterálás
  - De nehéz iterátort írni!

Iterátorok

```
public class List
{
   internal object[
   internal int cou

   public IEnumerat
      return new L
   }
}
```

```
public class ListEnumerator : IEnumerator
{
   List list;
   int index;

   internal ListEnumerator(List list) {
      this.list = list;
      index = -1;
   }

                                    false;

                                    dex]; }
```

```
public class List
{
   internal object[] elements;
   internal int count;

   public IEnumerator GetEnumerator() {
      for (int i = 0; i < count; i++) {
         yield return elements[i];
      }
   }
}
```

Iterátor

- Olyan metódus, a
  egy sorozat követ
  - **yield return** és **yi**
  - IEnumerator-t va

```
public class Test
{
    public IEnumera
        yield return
        yield return
    }
}
```

```
public IEnumerator GetEnumerator() {
    return new __Enumerator(this);
}

private class __Enumerator : IEnumerator
{
    object current;
    int state;

    public bool MoveNext() {
        switch (state) {
            case 0:
                current = "Hello";
                state = 1;
                return true;
            case 1:
                current = "World";
                state = 2;
                return true;
            default:
                return false;
        }
    }

    public object Current {
        get { return current; }
    }
}
```

Tartalom

- CLR 2.0 és C# újdonságok
  - Sablonok (generics)
  - Név nélküli metódusok (anonymous methods)
  - Nullable types
  - Iterátorok
  - **Egyebek**
  - Ko- és kontra variancia

Statikus osztályok

- Csak statikus tagok
- Nem használható változó, parameter, mező, … típusaként
- Például System.Console, …Environment

```
public static class Math
{
    public static double Sin(double x) {...}
    public static double Cos(double x) {...}
    ...
}
```

Tulajdonságok elérése

- Szétválasztott hozzáférésszabályzás
  - Az egyik taghoz korlátozottabb hozzáférés rendelhető
  - Tipikusan a set {…} jobban korlátozott

```
public class Customer
{
    private string id;

    public string CustomerId {
        get { return id; }
        internal set { id = value; }
    }
}
```

Figyelmeztetés vezérlés

- #pragma warning
  - (fordító paraméterként is)

```
using System;

class Program
{
    [Obsolete]
    static void Foo() {}

    static void Main() {
#pragma warning disable 612
        Foo();
#pragma warning restore 612
    }
}
```

Névtér alias minősítő

C# fordító

- Ütközések elkerülésére
- A global:: a globális névteret jelenti

```
using IO = System.IO;

public class MyClass
{
   public void Method1() { ; }
}

namespace MyApp
{
  public class MyClass
  {
     public void Method1()
     {
        global::MyClass o = new global::MyClass();

        IO::Stream s = IO::File.OpenRead("foo.txt");
     }
  }
}
```

Extern szerelvény alias

C# fordító

- Probléma: két szerelvényben ugyanaz a típusnév (ugyanabban a névtérben)

eszkozok.dll

```
namespace Franko
{
   public class Cuccok
   {
      public static void F() {...}
   }
}
```

vegyes.dll

```
namespace Franko
{
   public class Cuccok
   {
      public static void F() {...}
   }
}
```

```
extern alias Eszkozok;
extern alias Vegyes;

class Program
{
   static void Main() {
      Eszkozok.Franko.Cuccok.F();
      Vegyes.Franko.Cuccok.F();
   }
}
```

Parancssorba:

```
C:\>csc /r:Eszkozok=eszkozok.dll
   /r:Vegyes=vegyes.dll teszt.cs
```

Részleges típusok

```
public partial class Customer
{
    private int id
    private string
    private string
    private List<O
}
```

```
public partial cl
{
    public void Su
        orders.Add(
    }

    public bool Ha
        return orde
    }
}
```

```
public class Customer
{
    private int id;
    private string name;
    private string address;
    private List<Orders> orders;

    public void SubmitOrder(Order order) {
        orders.Add(order);
    }

    public bool HasOutstandingOrders() {
        return orders.Count > 0;
    }
}
```

Részleges típusok

- Egy osztály forráskódja több fájlra bontható
  - IDE generált és saját kód
- Csak fordítás alatt !
  - Nem kerülhet több szerelvénybe
  - Nincs linkelés
- A részleges osztályokat explicit jelölni kell
- A dizájnerek ahol csak lehet, részleges osztályokat generálnak

Tartalom

- CLR 2.0 és C# újdonságok
  - Sablonok (generics)
  - Név nélküli metódusok (anonymous methods)
  - Nullable types
  - Iterátorok
  - Egyebek
  - **Ko- és kontra variancia**

Tömbök kovarianciája

- Ez vajon működik ?

```
string [ ] sa = “apple knife ”.Split();
object [ ] oa = sa;                          // kovariancia
```

- És ez ?

```
foreach( var v in oa )
Console.WriteLine(v);
```

- No és ez?

```
oa[0] = “12”;
```

- Vagy ez ?

```
oa[0] = 12;
 ...
```

Ko- és kontravariancia definíció

- Egy programnyelv típusrendszerében egy típus konverzió:
  - **Kovariáns**, ha megőrzi a típusok rendezését ami a specifikustól a generikus felé rendez;
    - Arrays: string [ ] => object [ ] // nem biztonságos
  - **Kontravariáns**, ha megfordítja a rendezést;
  - **Invariáns**, ha egyik sem teljesül.

CLR V1

- Referencia típusok tömbjei kovariánsok C# 1.0-tól
  - Érték típusok nem, hiszen nincs blittable konverzió köztük (az enum-int stb nem támogatott)
- A [ ] explicit konvertálható B [ ]-re akkor és csak akkor ha van implicit konverzió A-ból B-be és A és B referencia típusok

Futás idejű ellenőrzések

- A tömb egy elemének beállítása
  - Az új elemnek implicit konvertálhatónak kell lennie az eredeti típusra

    objectArray[4] = “alma”;
- Tömb elem referencia vagy kimenő paraméterként való átadása
  - Csak a pointer kerül átadásra így a tömb nélkül nem lehet ellenőrizni a típust
  - Ezért a paraméter típusának a tömbelem típusának kell lennie

MIÉRT VAN OTT EGYÁLTALÁN ???

"Unfortunately, **this particular kind of covariance is broken**. It was added to the CLR because Java requires it and the CLR designers wanted to be able to support **Java-like languages**. We then up and added it to C# because it was in the CLR. This decision was quite controversial at the time and I am not very happy about it, but there's nothing we can do about it now." /Eric Lippert/

- http://blogs.msdn.com/b/ericlippert/archive/2007/10/17/covariance-and-contravariance-in-c-part-two-array-covariance.aspx

Miért van Java-ban?

- Mert nélküle nem lehetne ilyen általános metódusokat írni:
  - Array.sort(object [ ] arr);
  - Ez generikus típusoknál talán felesleges, de eredetileg egyik platformon se voltak generikus típusok
    - (Egy Array.swap metódus ezt ugyan megoldaná, de van egy csomó más helyzet is…)

Mi a helyzet a generikus típusokkal?

1. IEnumerable<object> ie = new List<string>();

- Ez biztonságos?
- Működik?

És ezek?

2. Action<string> a = (object a) => { cw(a); };
3. List<object> l = new List<string>();

Mi van a delegate-ekkel?

- class Control : Object { }
- class Button : Control { }

Melyik biztonságos?

```
delegate Control delegateControl_Control( Control o );

static void theOldWay2()  {
  delegateControl_Control    Control_Control;
  Control_Control = fn_Object_Button;
  Control_Control = fn_Button_Object;
}

static object fn_Object_Button( Button o ) {
      Console.WriteLine( o . Name ); }
static Button fn_Button_Object( object o ) {
      Console.WriteLine( o . ToString() ); }
```

C# V2

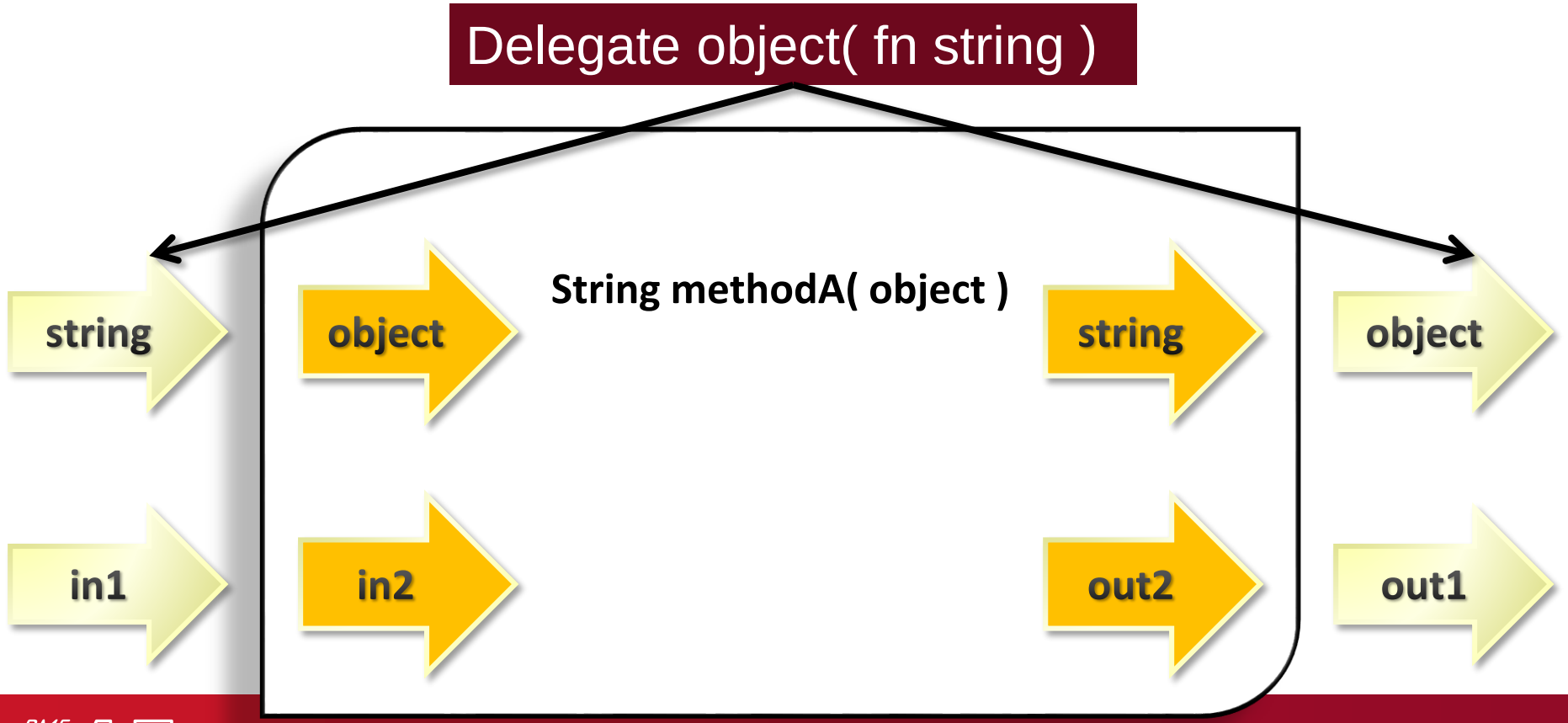
- C# 2.0 támogatja a delegate-ek ko- és kontravarianciáját
- Sok eseménykezelőnél vagy generikus eseménykezelőknél nagyon jól jön
  - ASP.NET, WPF stb

Mi a hasonlóság a két esetben?

- Tekintsük a tömböket és metódusokat feketedoboznak
  - A delegate átadja a paramétereket a funkciónak
- Objektumok bemennek (írás) és kijönnek (olvasás)
- Kifelé (olvasás, kovariancia): az objektumnak legalább azt a kell tudnia, amit a **hívó elvár**
- Befelé (írás, kontravariancia): az objektumnak legalább azt tudnia kell, amit a **feketedoboz elvár**
  - Tehát származtatott osztálynak kell lennie

Feketedoboz

- Kontravariancia: in1 : in2
- Kovariancia: out2 : out1



Melyik tehát a biztonságos ?

Interfész:

1. IEnumerable<object> ie = new List<string>();
2. IEnumerable<string> ie = new List<object>();

Delegate:

1. Action<object> a = (string a) => { cw(a); };
2. Action<string> a = (object a) => { cw(a); };

C# v4.0

- Ko- és kontravariancia interfészek és delegate-ek generikus típusparaméterekkel!
  - Hasonló feketedobozok mint korábban
- A generikus típusargumentumok megjelölhetők: be- vagy kimenetek
- Fordító ellenőrzések
  - Az „in” típus argumentumok csak paraméterek lehetnek
  - Az „out” típus argumentumok csak visszatérési értékek vagy kimeneti paraméterek lehetnek

Ko- és kontravariancia

```
string[] strings = GetStringArray();
Process(strings);
```

.NET tömbök kovariánsak

```
void Process(object[] objects) {
   objects[0] = "Hello";        // Ok
   objects[1] = new Button();   // Exception!
}
```

…de **nem biztonságosak**

```
List<string> strings = GetStringList();
Process(strings);
```

A generikus típusok v4 előtt invariánsak voltak

```
void Process(IEnumerable<object> objects) {
   // IEnumerable<T> is read-only and
   // therefore safely co-variant
}
```

C# 4.0 támogatja a **biztonságos** ko- és kontravarianciát

Biztonságos ko- és kontravariancia

```
public interface IEnumerable<out T>
{
   IEnumerator<T> GetEnumerator();
}
```

out = kovariáns, csak kimenetként

```
public interface IEnumerator<out T>
{
   T Current { get; }
   bool MoveNext();
}
```

Kevésbé specifikusként lehet kezelni

```
IEnumerable<string> strings = GetStrings();
IEnumerable<object> objects = strings;
```

```
public interface IComparer<in T>
{
   int Compare(T x, T y);
}
```

in = kontravariáns, csak bemenetként

Kevésbé generikusként lehet kezelni

```
IComparer<object> objComp = GetComparer();
IComparer<string> strComp = objComp;
```

Variancia C# 4.0-ban

- Interfészekhez és delegate-ekhez
- “Statically checked definition-site variance”
- Érték típusok mindig invariánsak
  - IEnumerable<int> *nem* IEnumerable<object>
  - Hasonlóan a korábbi tömbökhoz
- Ref és out paramétereknek invariáns típusoknak kell lennie

- Generikus ko- és kontravarianciát a .NET CLR 2.0-tól már támogatta !

Variancia a .NET Framework-ben

Interfaces

System.Collections.Generic.IEnumerable<out T>
System.Collections.Generic.IEnumerator<out T>
System.Linq.IQueryable<out T>
System.Collections.Generic.IComparer<in T>
System.Collections.Generic.IEqualityComparer<in T>
System.IComparable<in T>

Delegates

System.Func<in T, …, out R>
System.Action<in T, …>
System.Predicate<in T>
System.Comparison<in T>
System.EventHandler<in T>

C# újdonságok

- Generikus típusok
- Tulajdonság elérhetőség
- Statikus osztályok
- Névtelen műveletek
- External szerelvény alias
- Iterátorok
- Delegát kovariancia és kontravariancia
- Részleges típusok
- Nullable types
- Fix méretű bufferek
- Friend szerelvények
- Inline warning szabályozás
- Névtér alias minősítő