Alkalmazásfejlesztés II.

Dr. Dombi József Dániel

Szegedi Tudományegyetem
Informatikai Intézet
Szoftverfejlesztés Tanszék

2024

Tartalom

Fordítás ideje: 2024.09.11 13:09:47 git branch: nincs nincs

Tartalom

Előadók

Nappali tagozat

- Dr. Dombi József Dániel

szoba: Szoftverfejlesztés Tanszék (Árpád tér 2.) 4. szoba
telefon: +36 62 54-4235 / 4235
e-mail: dombijd@inf.u-szeged.hu
honlap: http://www.inf.u-szeged.hu/~dombijd/

Előadás

Követelmények

- Jelentkezések az Neptun-on keresztül a kurzusfelvételi időszak végéig (lehetőleg minél hamarabb).
- A tárgy alapfeltétele az Alkalmazásfejlesztés I tantárgy. Aki azt nem teljesítette, **NE** vegye fel ezt a kurzust!
- Az előadás helyszíne TIK Alagsor I-II előadó
- Az előadás látogatása nem kötelező, de *a gyakorlat épít az előadáson elhangzottakra*.
- Vizsgára jelentkezés előfeltétele a gyakorlat sikeres (legalább elégséges (2) szinten történő) teljesítése.
- Vizsgára jelentkezés az Neptun-on keresztül az aktuális szabályzatoknak (TVSZ, TTIK Tanulmányi Ügyrend) megfelelően.

Előadás

Számonkérés és értékelés

- Az előadás teljesítéséhez sikeres vizsgát kell tenni.
- A sikeres vizsgához az elérhető pontszám legalább 50%-át teljesíteni kell.
- A kollokviumon semmilyen segédlet nem használható.

89	–	100	jeles (5)
76	–	88	jó (4)
63	–	75	közepes (3)
50	–	62	elégséges (2)
0	–	49	elégtelen (1)

Megajánlott jegy

- Újdonság ebben a félévben, hogy félévközi teljesítmény alapján megajánlott jegyet lehet kapni.
- Feltételek
  - Gyakorlaton legalább 4 (jó), vagy 5 (jeles) érdemjegy esetén. A gyakorlati jegy lesz a megajánlott érdemjegy.
  - Az előadás coospace színterén meg fog jelenni 12 teszt. Sikeres teszt kitöltés alapján a hallgató 1 pontot kap. Aki megajánlott jegyet szeretne, annak legalább 10 pontot össze kell gyűjtenie.
  - Abban az esetben lesz lehetőség megajánlott jegyet kérni, ha a megtartott előadásokon a résztvevők száma MINDEN (akár online, akár offline esetben) alkalommal eléri a 40 főt. Amennyiben egy alkalommal is ez a szám alá esik, akkor senki se kérhet megajánlott jegyet.

Az előadás anyaga

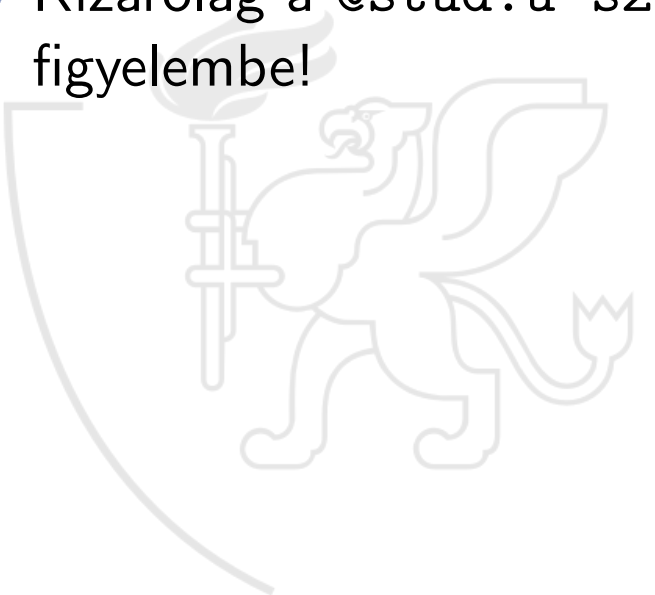
- Microsoft .NET Framework és .NET Core
  - Fordítás, futtatás
  - C# nyelv (OOP)
  - LINQ, kifejezés fák
  - C# nyelv érdekességek, szálkezelés
  - Asztali alkalmazás készítése
  - Adatbázis kezelés, ORM használat
  - Webes alkalmazás készítése

Segítség a kurzushoz

- Az előadás fóliái
  - `Coospace színtérbe kerülnek fel`
- Könyvek
  - Start building websites and services with ASP.NET Core 7, Blazor, and EF Core 7, 7th Edition
  - Apps and Services with .NET 7: Build practical projects with Blazor, .NET MAUI, gRPC, GraphQL, and other enterprise technologies
  - Pro ASP.NET Core 6: Develop Cloud-Ready Web Applications Using MVC, Blazor, and Razor Pages
  - C# 10 in a Nutshell: The Definitive Reference
  - Larry O'Brien, Bruce Eckel: *Thinking in C#*, Prentice Hall, 2002. ISBN-13: 978-0130385727
  - Tony Northrup, Shawn Wildermuth, Bill Ryan, *Microsoft .NET Framework 2.0 Foundation*, Microsoft Press, 2006. ISBN-13: 978-0735622777.
  - Reiter István, *C# jegyzet*, devPortal, 2010. ISBN-13: 978-6155012174. https://devportal.hu/content/CSharpjegyzet.aspx

Probléma esetén

- A kurzussal kapcsolatban
  1. Gyakorlatvezető
     1. Személyesen a gyakorlatokon
     2. e-mail -ben
  2. Ha a gyakorlatvezető úgy ítéli meg akkor: Előadó
     1. Személyesen az előadások szüneteiben
     2. e-mail -ben, mindkét előadónak.
- Kizárólag a `@stud.u-szeged.hu` címről jött leveleket vesszük figyelembe!



Alkalmazásfejlesztés II.

Dr. Dombi József Dániel

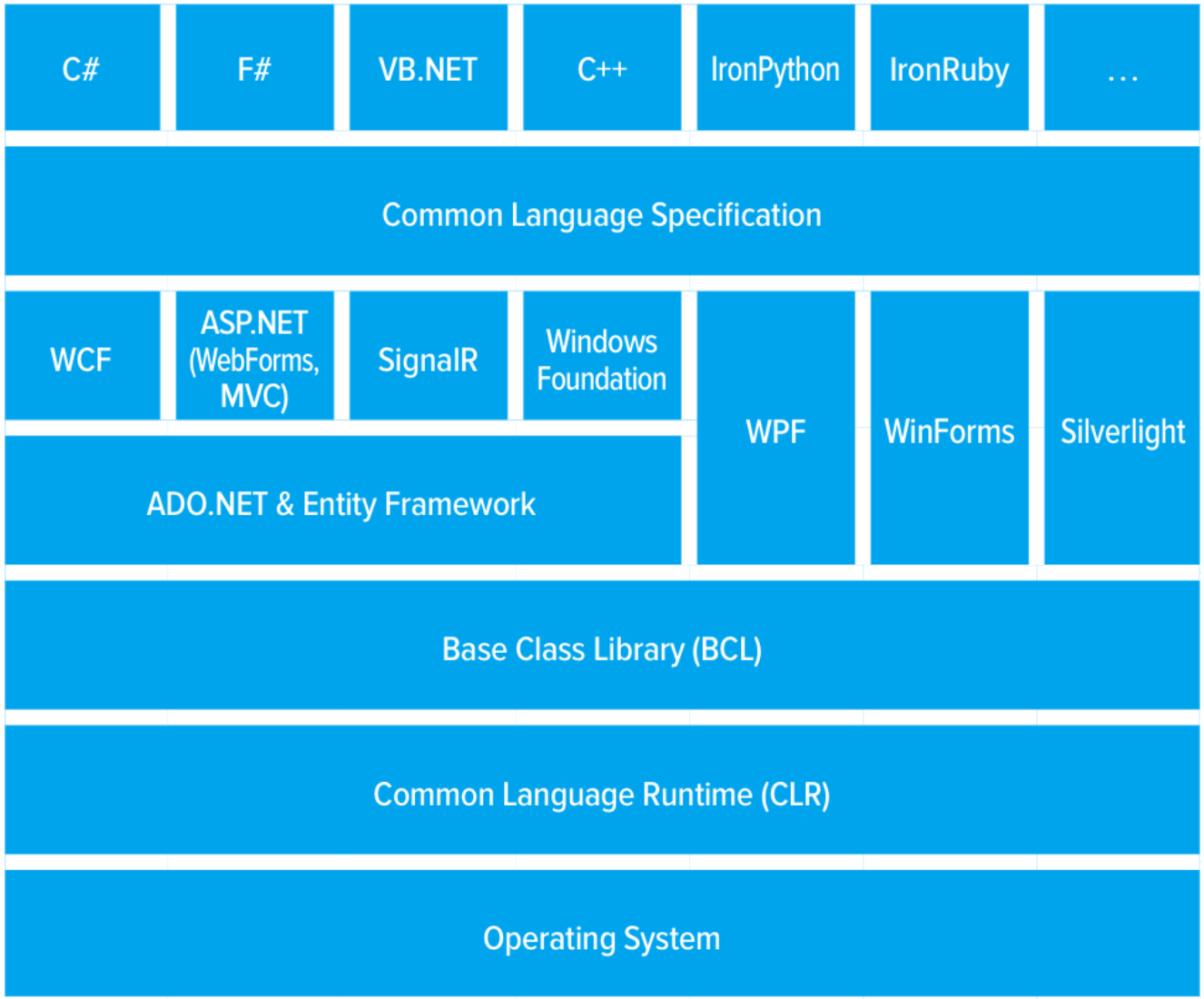
Szegedi Tudományegyetem
Informatikai Intézet
Szoftverfejlesztés Tanszék

2024

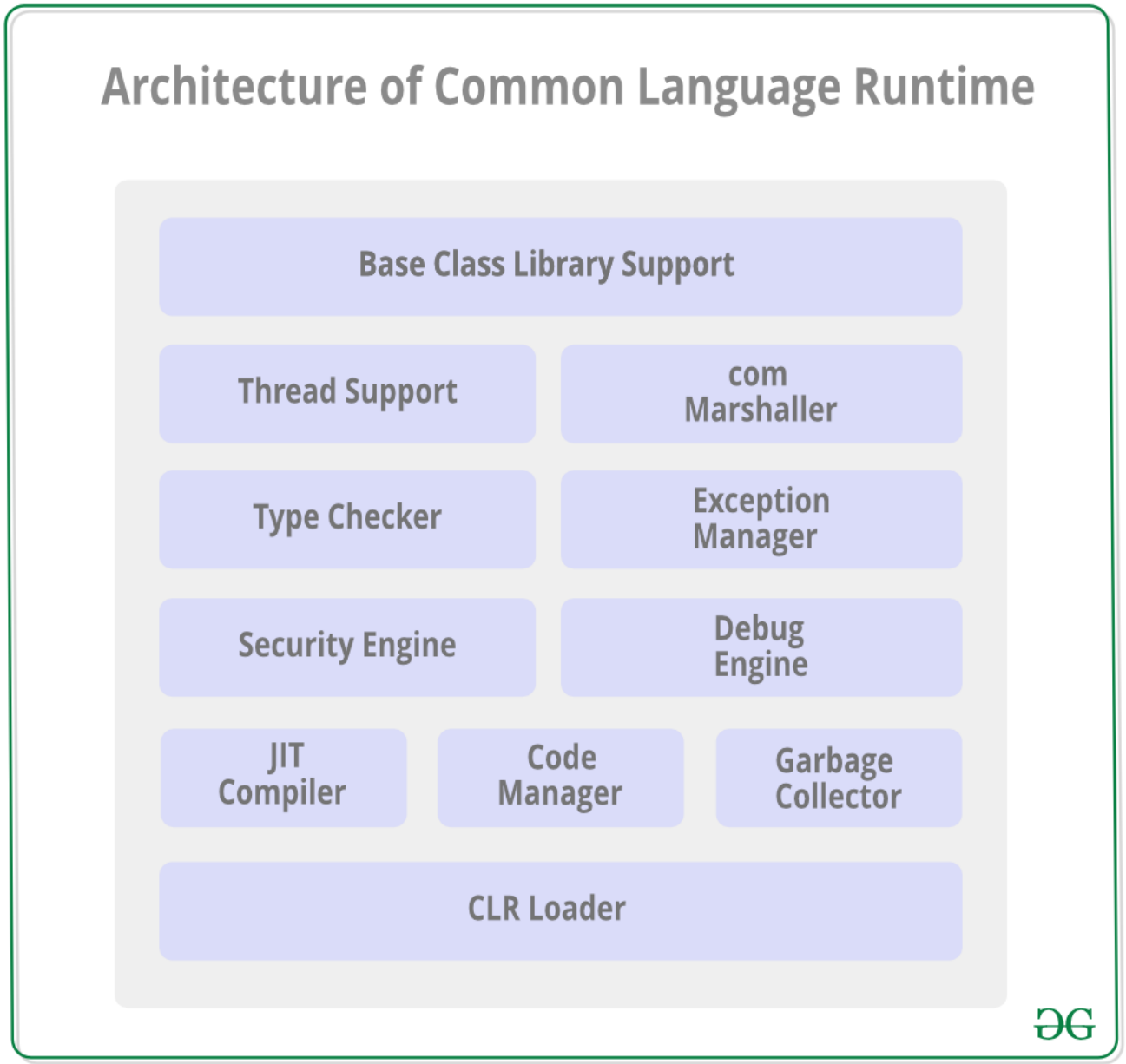
Tartalom

Fordítás ideje: 2024.09.11 13:09:47 git branch: nincs nincs

Microsoft .NET Framework



Common language runtime



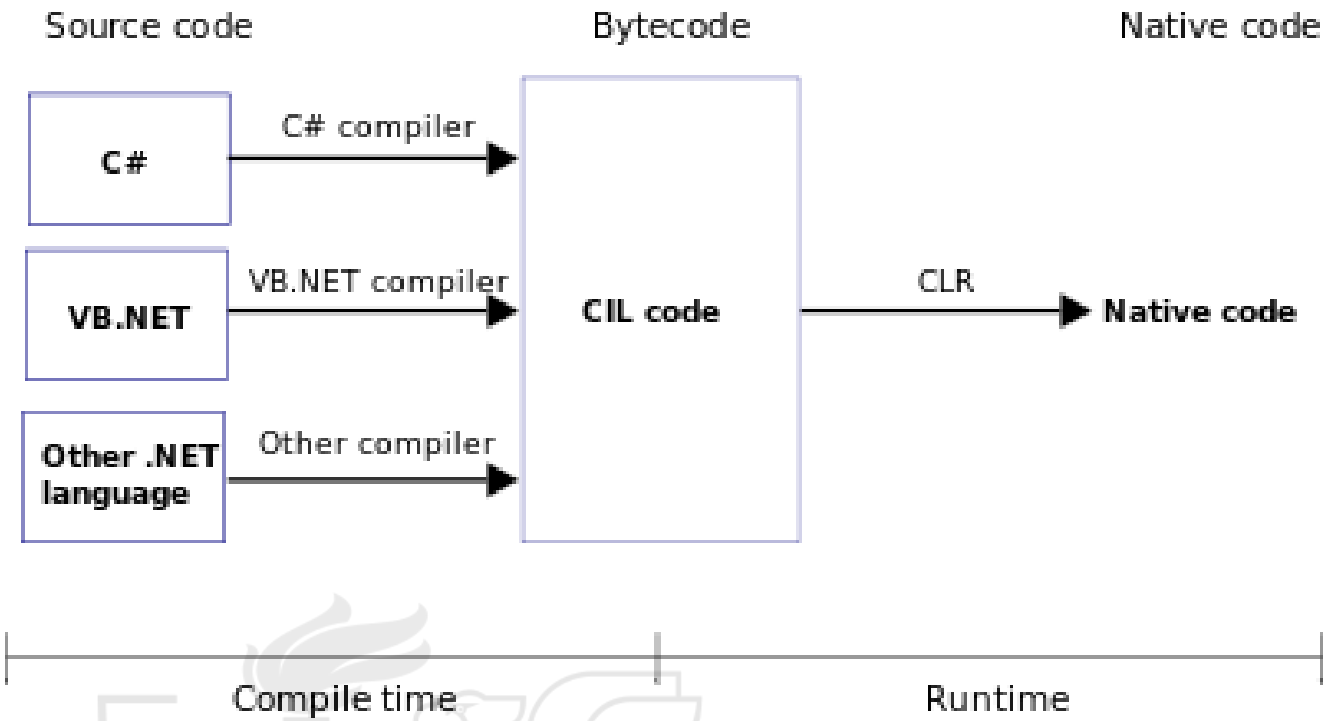
.NET keretrendszer

- Átfogja a használható programozási nyelveket
  - Nyelvközi öröklődést és hibakeresést tesz lehetővé
  - Jól integrálódik az eszközökkel
- Objektum orientált és konzisztens
  - Növeli a produktivitást mert csökkenti a megtanulandó API-k számát
- A közös típus rendszeren készült

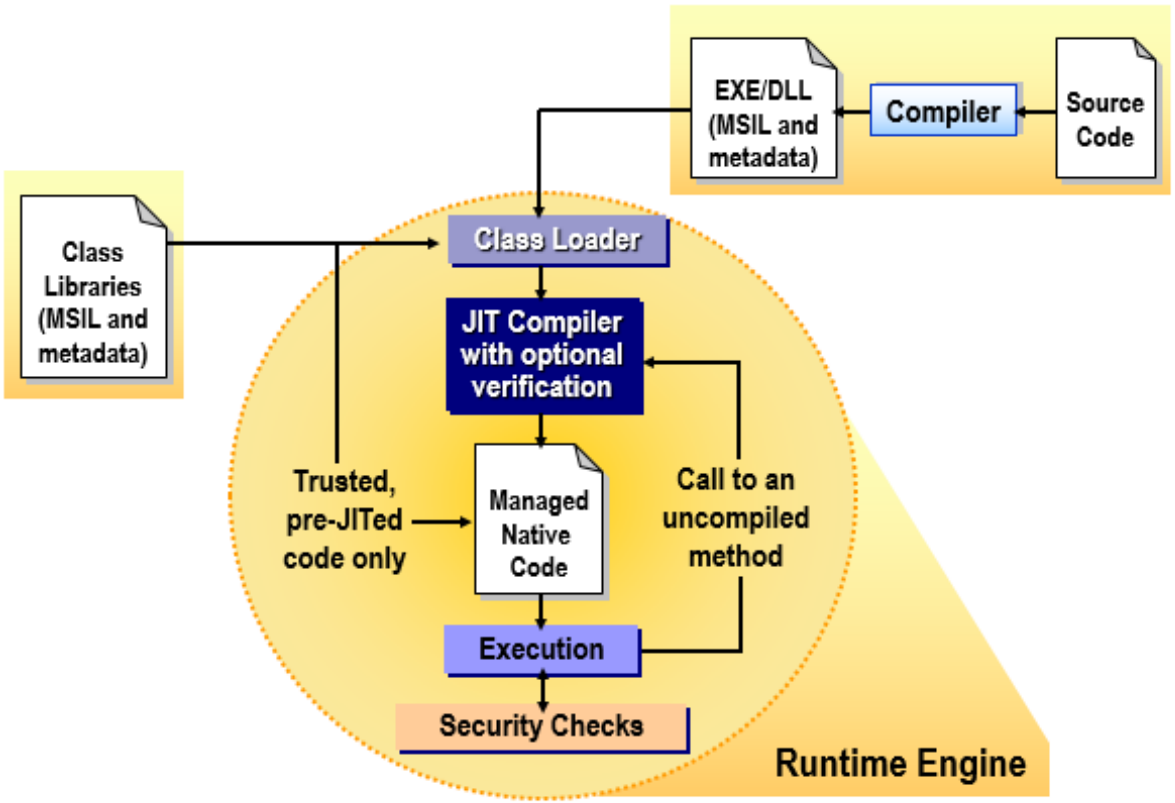
Microsoft Intermediate Language

- A köztes nyelv
- Minden támogatott fordító erre fordít
- JIT fordítók fordítják gépközeli nyelvre
- Például egy Visual Basic .NET-es projektben lévő osztályt felhasználhatunk C#-os projektben, vagy akár örökölhetünk is belőle.

Common Language Runtime



Runtime engine



Köszönöm a figyelmet!

Alkalmazásfejlesztés II.

Dr. Dombi József Dániel

Szegedi Tudományegyetem
Informatikai Intézet
Szoftverfejlesztés Tanszék

2024

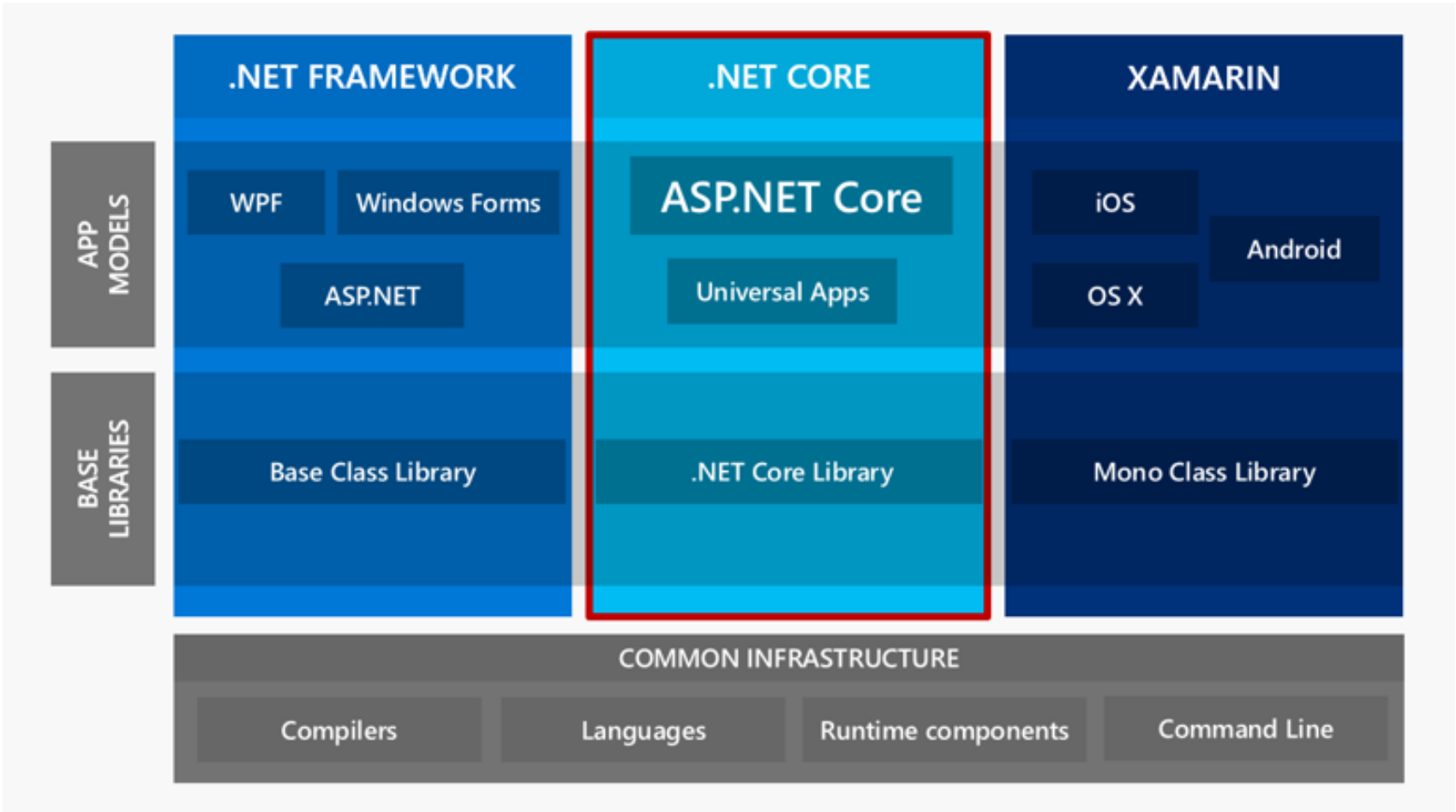
Tartalom

Fordítás ideje: 2024.09.25 16:25:50 git branch: nincs nincs

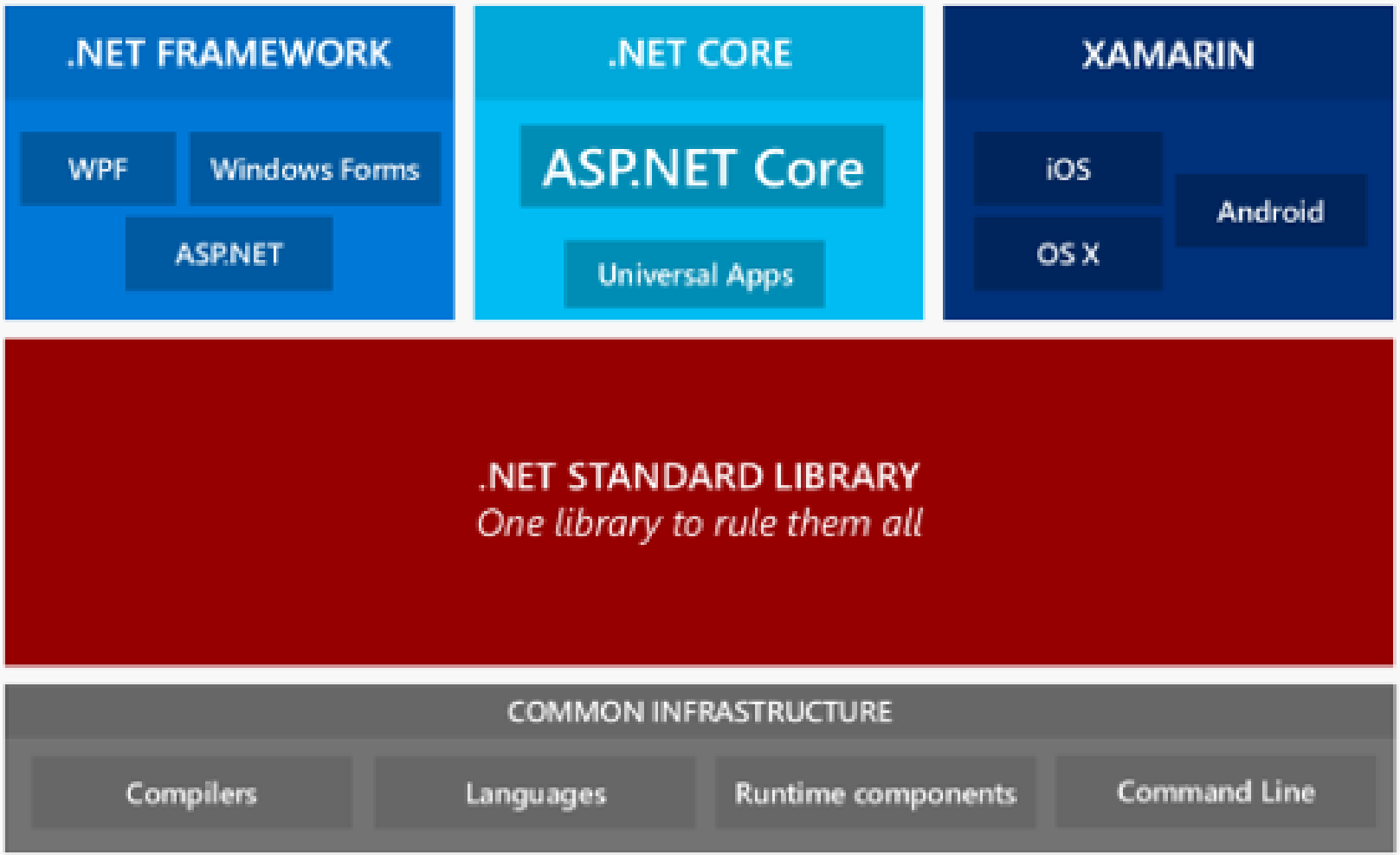
.NET

- Microsoft .NET Framework (4.8)
  - Csak Windows operációs rendszer alatt működik
  - Microsoft tervezte és fejlesztette
  - zárt forráskodú
  - Technológiák: WinForms, WPF, WCF, EF, ASP.NET, ASP.NET MVC,...
- Microsoft .NET Core (3.1) -> .NET 5 -> .NET 6 -> .NET 7 -> .NET 8 -> .NET 9
  - Platform független (Linux, Mac, Windows)
  - Közösség fejleszti
  - Nyílt forráskodú (MIT licence): https://github.com/dotnet/core
  - Technológiák: ASP.NET Core, EF Core
  - Technológiák: (3.0+): WinForm Core, WPF Core
- A két rendszer nem 100%-ban kompatibilis: https://docs.microsoft.com/en-us/dotnet/core/porting/

.NET family - keretrendszerek



.NET family - Standard library létjogosultsága .NET 3.0-ig???



.NET keretrendszer

- .NET futtató környezete szükséges
- Távolról és helyi gépről is futtatható
- Nem készítenek Registry bejegyzéseket
- Nem tehet tönkre más alkalmazást
- Megszűnik a DLL pokol
- A helyi fájlok törlésével eltávolítható

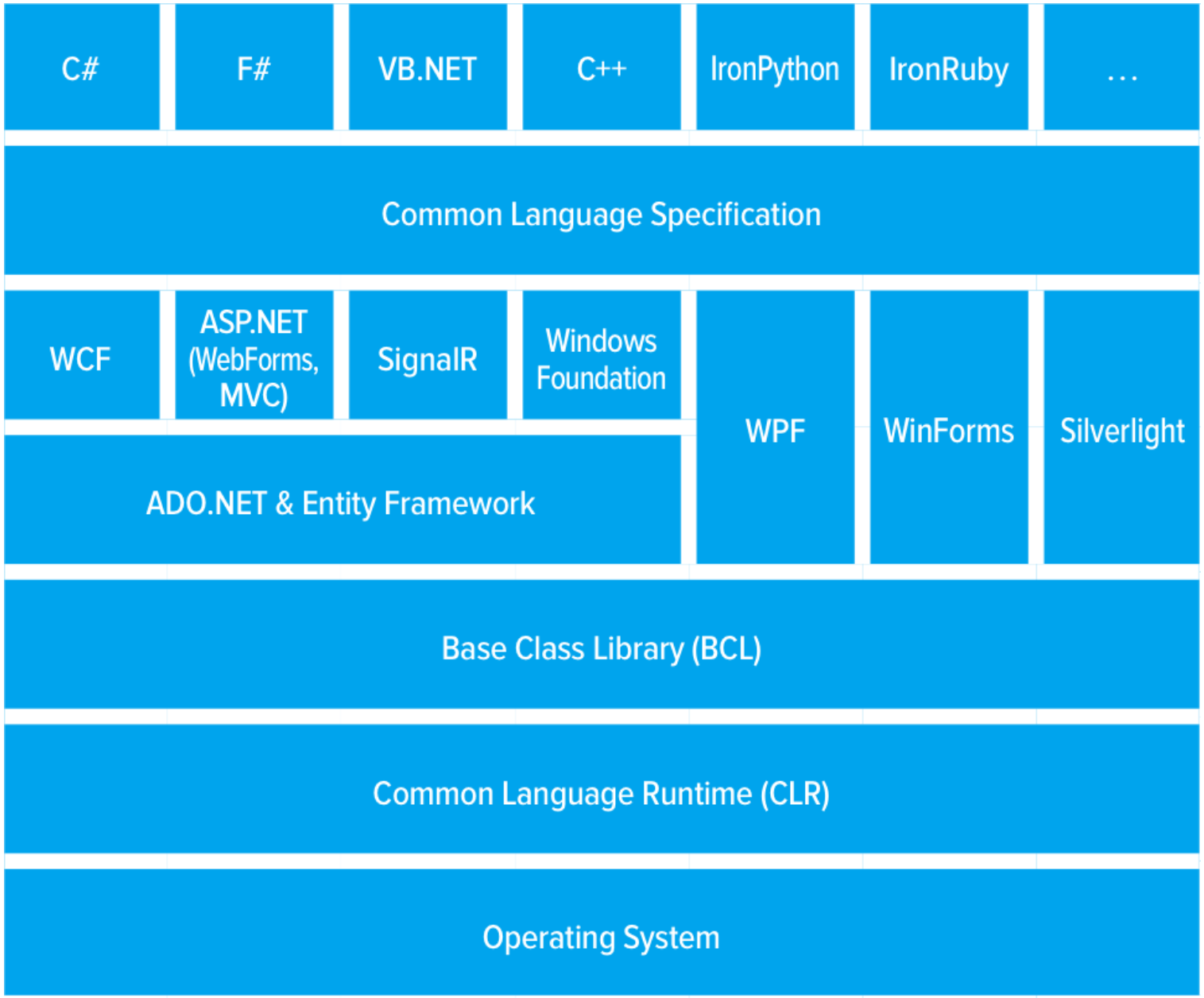
.NET Assembly

- Alapegysége a lefordított projektnek és verziózásnak
- Szerelvény lehet
  - futtatható .net alkalmazás (exe) (köztes kód!)
  - library (dll)
- Egy, vagy több osztályt is tartalmazhatnak.
- Tartalmaz egy leíró file-t (metadata), ami leírja a tartalmát, készítőjét, verzióját

.NET szemétgyűjtés

- Egyszerű szemét gyűjtő algoritmus
  - Addig vár míg a menedzselt kód szálai várakozó állapotba nem kerülnek
  - Az elérhető objektumokból gráfot készít
  - Átmozgatja az elérhető objektumokat a tömörített halomba
  - Az elérhetetlen objektumok területe felszabadul
  - Frissíti az átmozgatott objektumok referenciáit
- Körkörös referencia hivatkozások automatikusan kezelve vannak
- Külső erőforrások kezelése nem automatikus (pl: file, db, http, stb.)

Microsoft .NET Framework



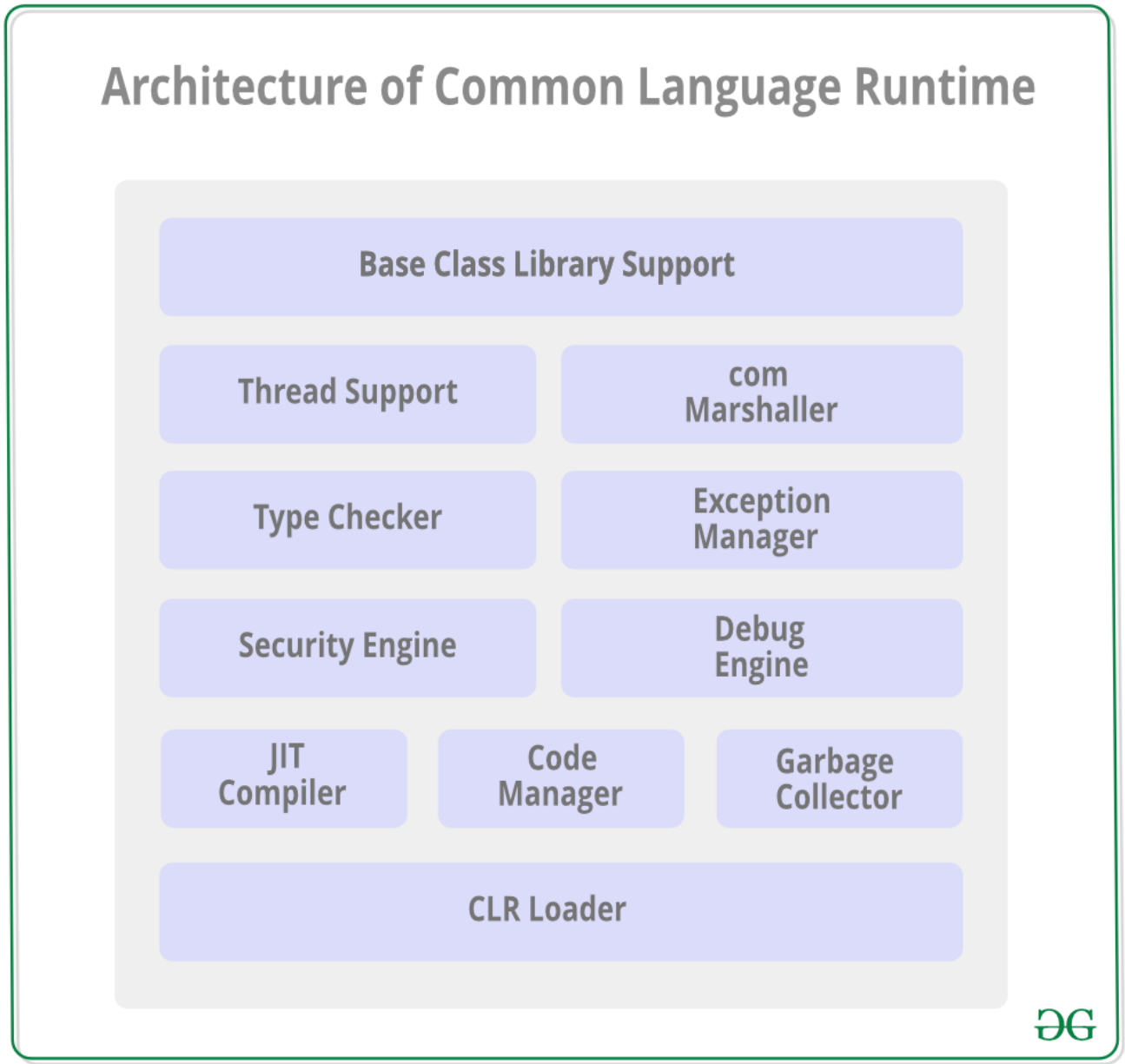
.NET keretrendszer

- Átfogja a használható programozási nyelveket
  - Nyelvközi öröklődést és hibakeresést tesz lehetővé
  - Jól integrálódik az eszközökkel
- Objektum orientált és konzisztens
  - Növeli a produktivitást mert csökkenti a megtanulandó API-k számát
- A közös típus rendszeren készült
- Fejlesztő környezetként használhatjuk a Visual Studiot.

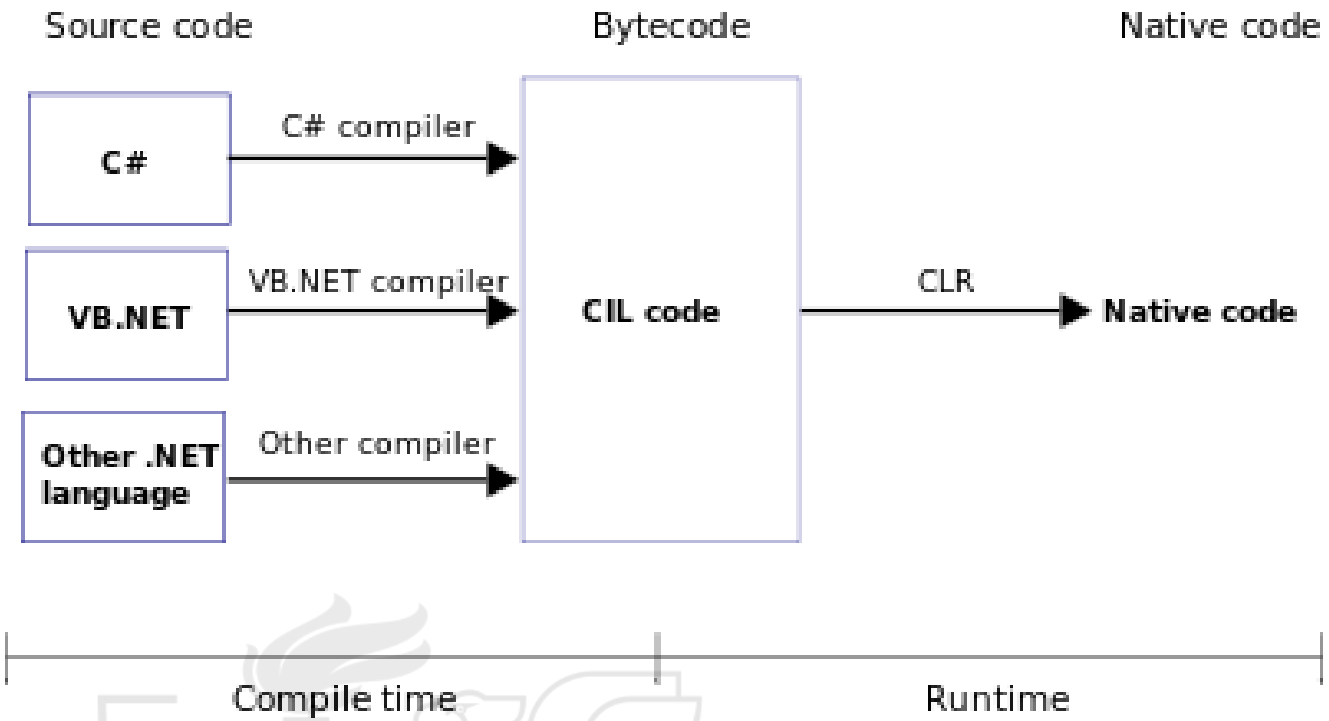
Microsoft Intermediate Language

- A köztes nyelv
- Minden támogatott fordító erre fordít
- JIT fordítók fordítják gépközeli nyelvre
- Például egy Visual Basic .NET-es projektben lévő osztályt felhasználhatunk C#-os projektben, vagy akár örökölhetünk is belőle.

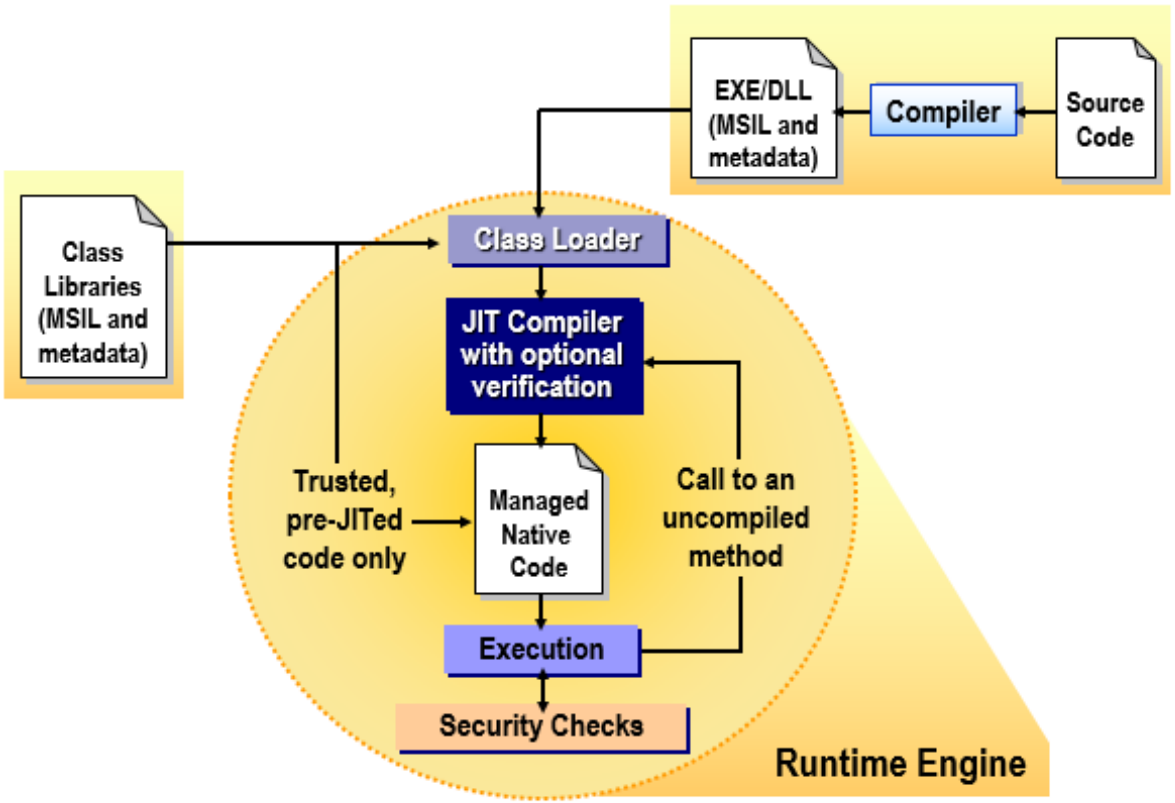
Common language runtime



Common language runtime



Runtime engine



Alkalmazásfejlesztés II.

Dr. Dombi József Dániel

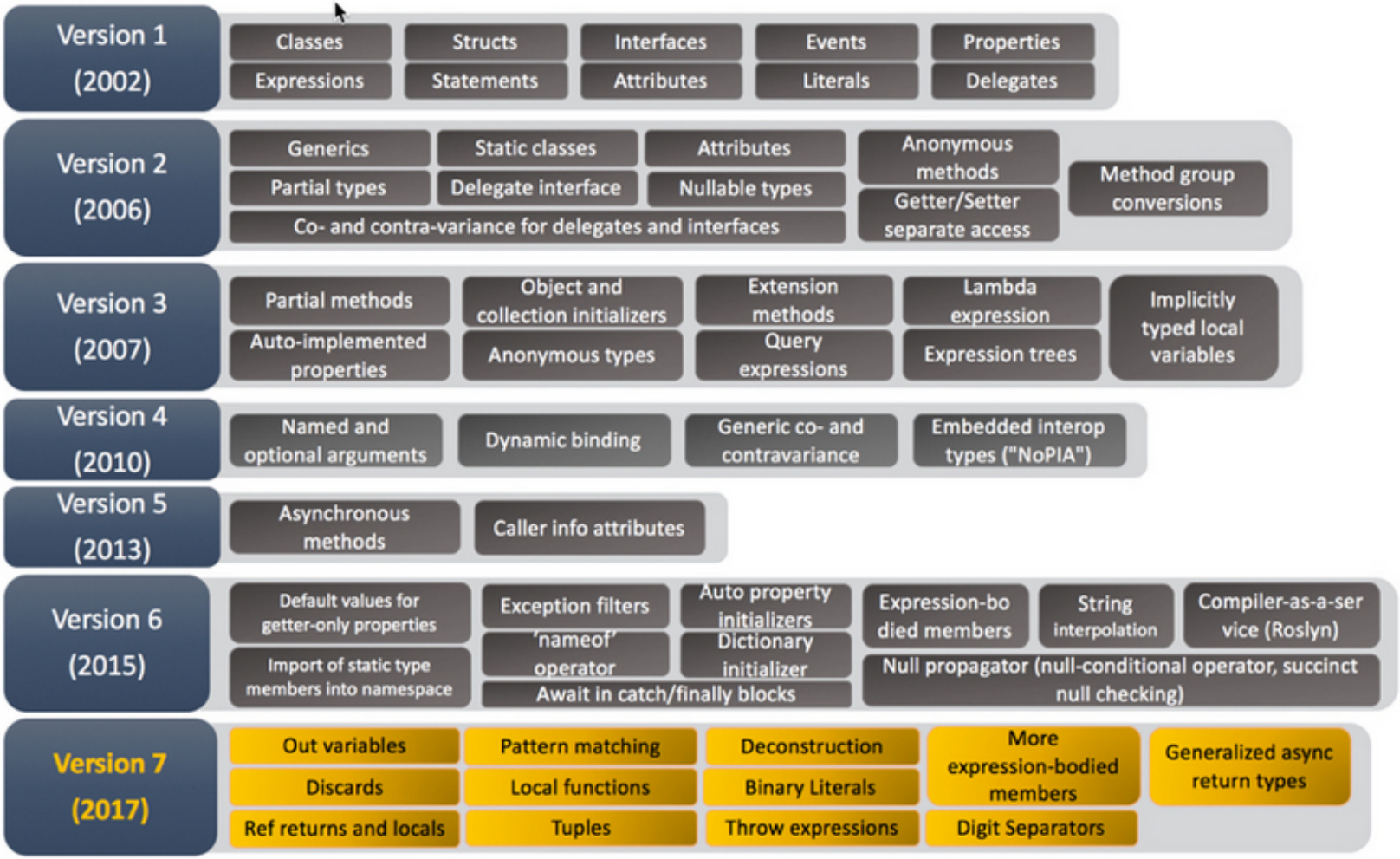
Szegedi Tudományegyetem
Informatikai Intézet
Szoftverfejlesztés Tanszék

2024

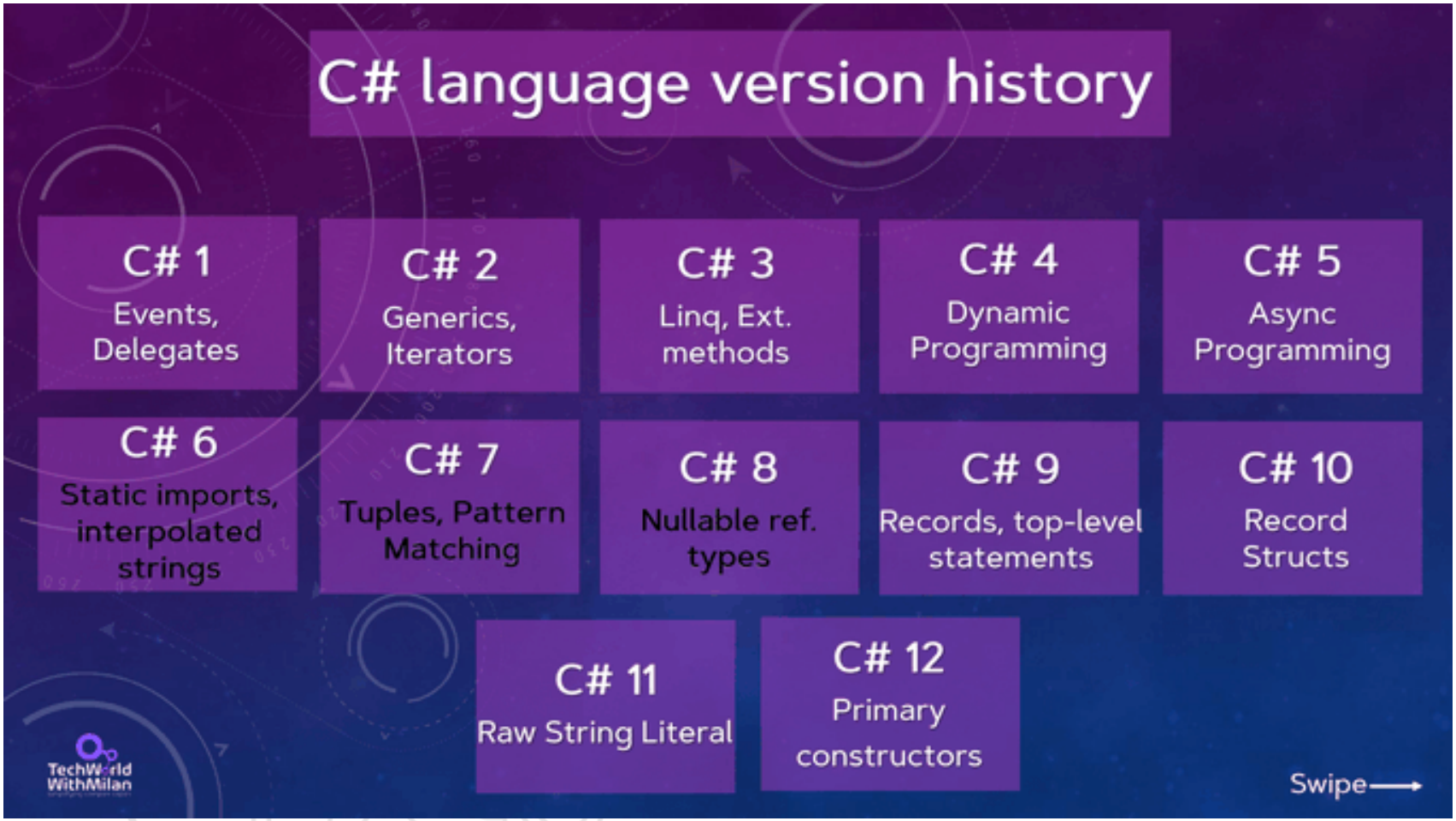
Tartalom

Fordítás ideje: 2024.09.25 16:25:50 git branch: nincs nincs

C# nyelv 7.0-ig


Version 1 (2002)
Classes
Structs
Interfaces
Events
Properties
Expressions
Statements
Attributes
Literals
Delegates
Version 2 (2006)
Generics
Static classes
Attributes
Anonymous methods
Method group conversions
Partial types
Delegate interface
Nullable types
Getter/Setter separate access
Co- and contra-variance for delegates and interfaces
Version 3 (2007)
Partial methods
Object and collection initializers
Extension methods
Lambda expression
Implicitly typed local variables
Auto-implemented properties
Anonymous types
Query expressions
Expression trees
Version 4 (2010)
Named and optional arguments
Dynamic binding
Generic co- and contravariance
Embedded interop types ("NoPIA")
Version 5 (2013)
Asynchronous methods
Caller info attributes
Version 6 (2015)
Default values for getter-only properties
Exception filters
Auto property initializers
Expression-bodied members
String interpolation
Compiler-as-a-service (Roslyn)
'nameof' operator
Dictionary initializer
Import of static type members into namespace
Await in catch/finally blocks
Null propagator (null-conditional operator, succinct null checking)
Version 7 (2017)
Out variables
Pattern matching
Deconstruction
More expression-bodied members
Generalized async return types
Discards
Local functions
Binary Literals
Ref returns and locals
Tuples
Throw expressions
Digit Separators

C# nyelv 12.0-ig

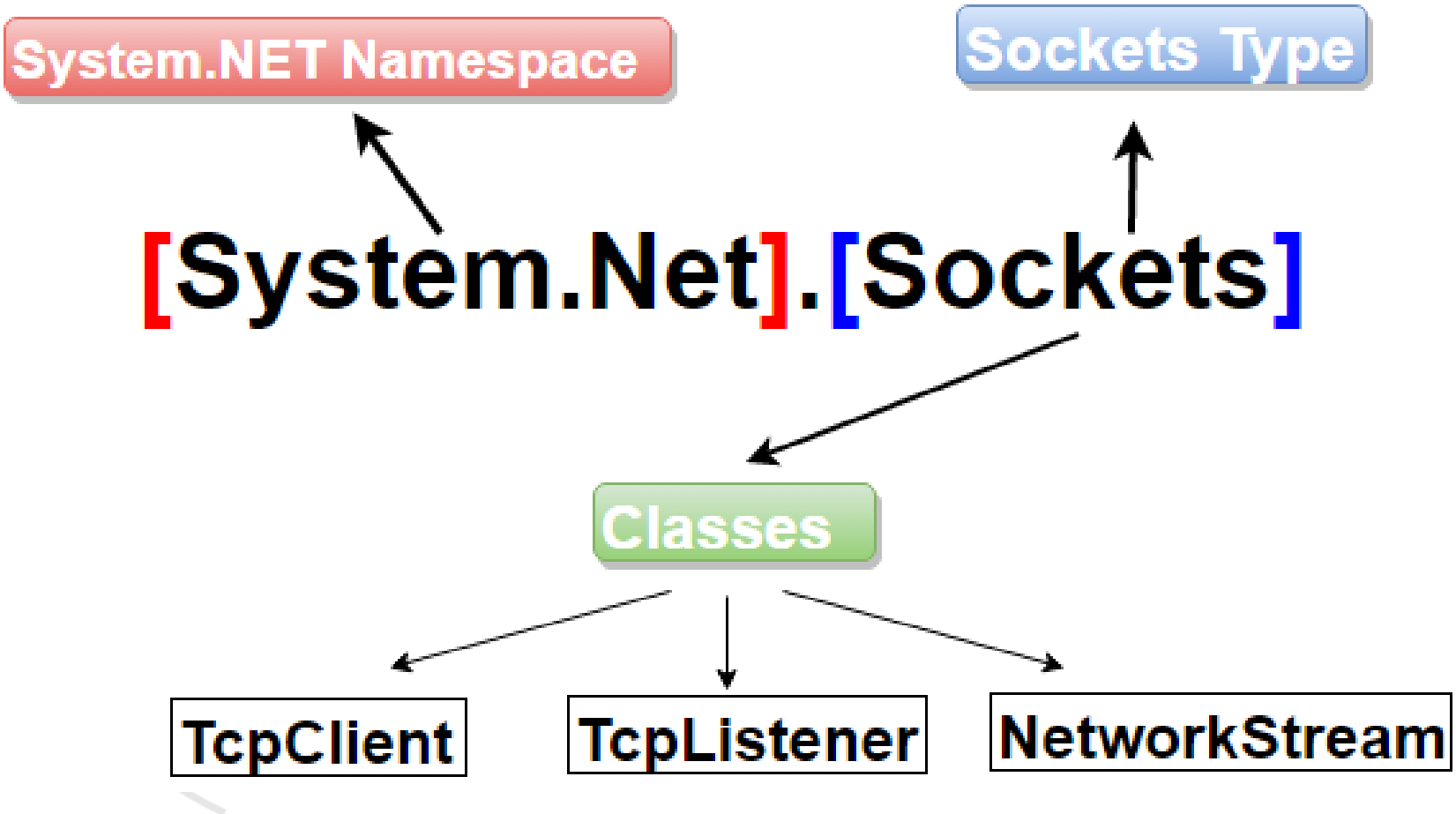


Bevezetés

- Objektum-orientált nyelv
- Program belépési pontja
  - static void Main(string[] args)
- Konzolra írás, konzolról olvasás
  - Console.Write("x = {0}", x);
  - Console.WriteLine(...)
  - x = Console.Read();
- Region (kód formázás segítése)

Névterek

- A típusok csoportosításának eszköze
- A namespace nyelvi kulcsszóval definiálható



Memória kezelés

- Az érték típusok (value type) a vermet használják
  - A memória foglalás és felszabadítás automatikus és biztonságos
- Menedzselt objektumok (reference type) a heapen tárolódnak
  - New operátor
  - Szemét gyűjtő takarít

Érték típusok (Value Type)

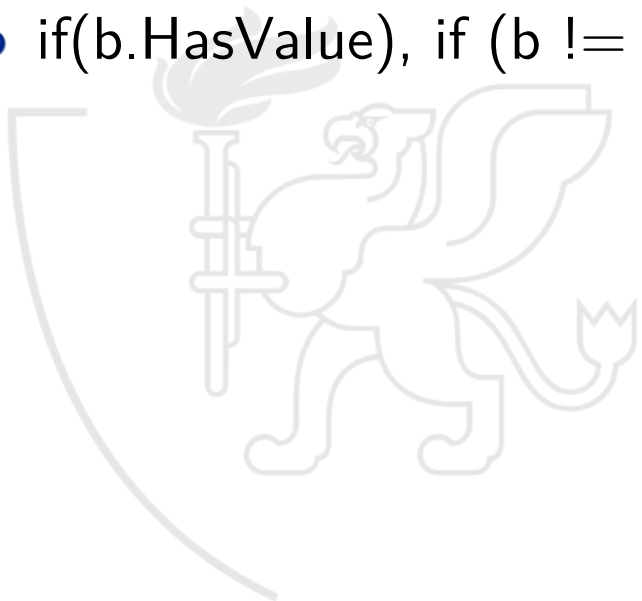
- A .NET keretrendszer legegyszerűbb típusai.
- Érték típusú változók közvetlenül a hozzájuk rendelt értéket tartalmazzák.
- Érték típusú változók tárolása a stack-ben történik.
- Mindegyik érték típus a **System.ValueType** alap típusból származik.
- Érték típusok:
  - Struct - struktúrák létrehozása: adatok és hozzá tartozó műveletek. C#7.2-től lehet readonly (nem módosítható, immutable) struktúrák létrehozása is támogatott.
  - integral numeric types: : sbyte, byte, int, uint, short, ...
  - floating point: float, double, decimal
  - bool (true, false)
  - char
  - enum (alapesetben int értéket kapnak 0-tól kezdve)
  - Tuple (C# 7.0): (double Sum, int Count) t2 = (4.5, 3);
  - Nullable

Érték típusok deklarálás

- Ahhoz, hogy ezeket a típusokat használjuk, deklarálnunk kell őket.
- A konstruktor alapértelmezett 0, vagy null értéket ad.
- Csak (kis- és nagybetűs) betűk, számok és aláhúzás-karakterek lehetnek a nevében.
- A névnek betűvel kell kezdődnie (aláhúzás karakter annak számít)
- A C# case-sensitive nyelv!

Nullable típus

- Nullable<type>, vagy type? (? operátor használata)
- Olyan érték típusú változók, amelyek null értéket is felvehetnek.
- A null értékkel azt jelölhetjük, ha a változó még nem kapott értéket.
- A nullable típusoknak **HasValue** és **Value** tulajdonságaik lesznek. A HasValue megállapítja, hogy a változónak van-e értéke.
- int? b = 10;
- if(b.HasValue), if (b != null), int c = b ?? -1;



Statements

- Elágazás
  - if - else if - else
  - switch - case - break - default
- Iterációk
  - while
  - do while
  - for, végtelen ciklus: for(;;)
  - foreach (azoknál a típusoknál, amelyek megvalósítják az IEnumerable interface-t)

Iterációk

```
Int32[] numbers = new Int32[10];
for (int i = 0; i < 10; i++)
{
    numbers[i];
}
foreach (int number in numbers)
{
    Console.WriteLine(number);
}
```

Kivétel kezelés

- A try kulcsszó nyitja a logika blokkját.
- A catch kulcsszó nyitja a hibakezelő blokkot. Szűrhető, hogy mely hibát akarjuk lekezelni.
- A finally kulcsszó nyitja a cleanup blokkot, amely mindenképpen lefut
  - Csak egy lehet belőle, a catchek után
  - Ha van finally, nem kötelező a catch
- Az Exception-ök olyan típusok, amelyek a System.Exception-ből származnak.
- A throw kucslszó segítségével az Exception-t tovább lehet dobni.
  - throw ex;
  - throw;
  - throw new Exception("inner ex", ex);

Kivétel kezelés

```
StreamReader stReader = new StreamReader(@"C:\Doc.txt");
try
{
    Console.WriteLine(stReader.ReadToEnd());
}
catch (Exception ex)
{
    Console.WriteLine("Error: " + ex.Message);
    // throw;
    // throw ex;
    // throw new Exception("Something wrong happened", ex);

    // without exact reason
    // throw new Exception("Something wrong happened");
}
finally
{
    stReader.Close();
}
```

Const vs read only

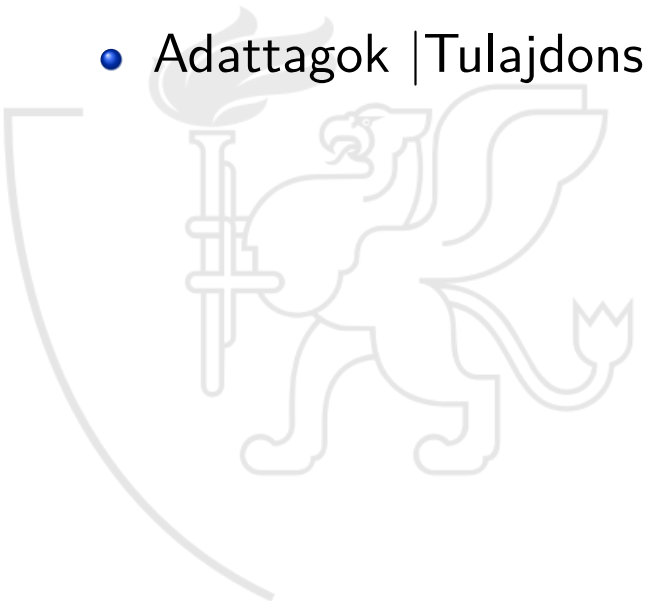
- Const – compile-time konstans
- Readonly – runtime konstansok definiálásra is alkalmas
  - Pl.: public static readonly uint timeStamp = (uint)DateTime.Now.Ticks;
  - Ez a kódrészlet public const ... esetén fordítási hibát eredményezne
- const esetén a member csak a deklaráció helyén kaphat értéket, ameddig a readonly esetében a konstruktorban is

Referencia típus (Reference Type)

- A referencia típusú változók a hozzájuk rendelt érték memóriacímét tartalmazzák.
- A referencia típusú változók a korábbi programozási nyelvek mutatóinak (pointer) feleltethetők meg.
- Beépített referencia típusok: object, string, dynamic
- **new** operátorral hozhatóak létre
- Heapben találhatók
- Megszüntetésükről a GC gondoskodik.

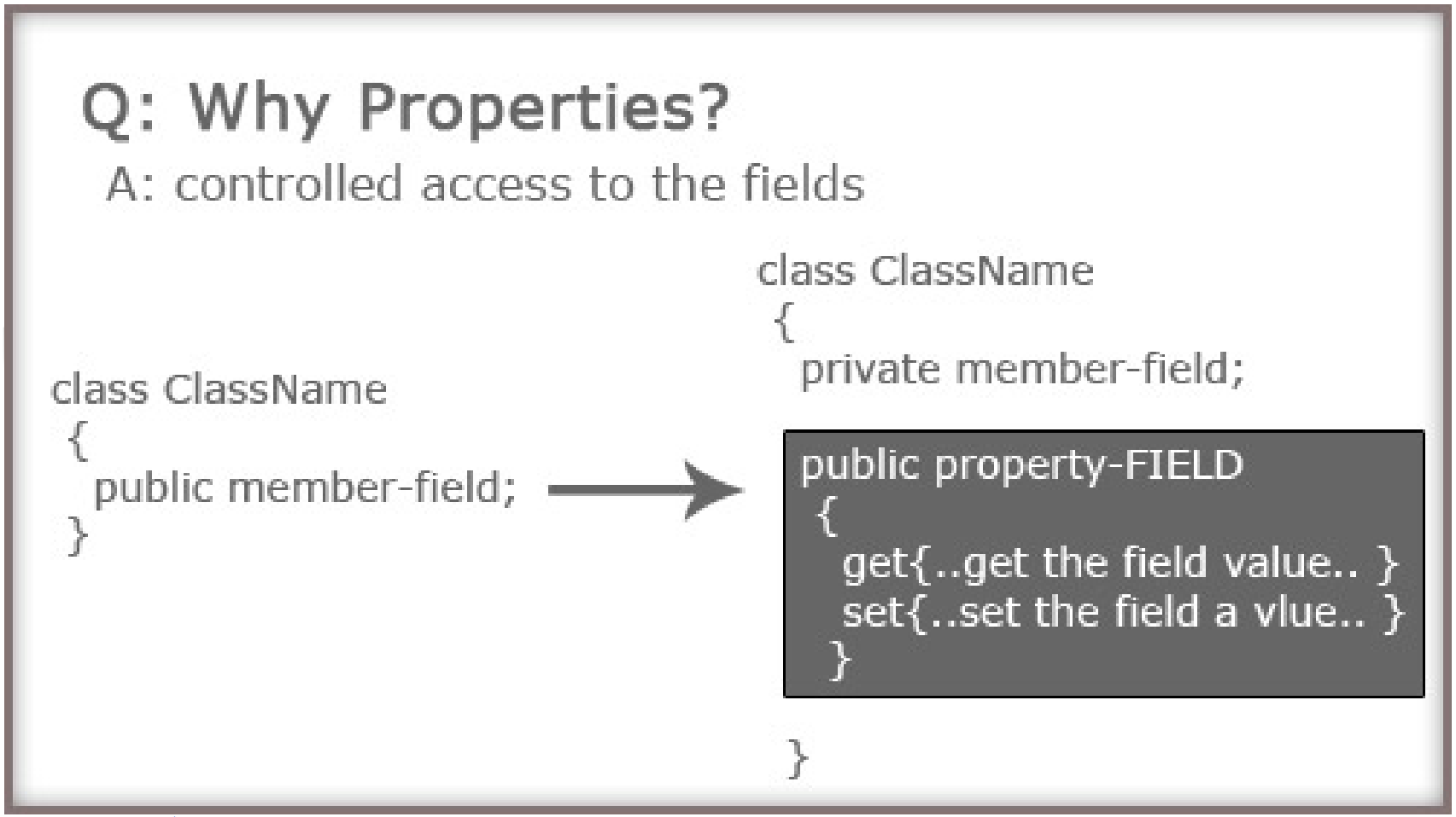
Osztályok (class)

- A **class** kulcsszó vezeti be
- Öröklés
  - Minden osztály a System.Object leszármazottja
  - Minden osztálynak csak egy őse lehet...
- Polimorfizmus
  - Őstípusként deklarált objektum értékül kaphat utódtípusút.
- Elemei:
  - Adattagok |Tulajdonságok |Műveletek |Események | Beágyazott típusok



Property

- **get, set** kulcsszavak
- Egy mezőt szabályozottan írhatóvá és/vagy olvashatóvá tévő metóduspár



Property

```
class Person{

//belső member változó
private string name;

//property
public string Name
{
        get { return this.name; }
        set { this.name = value; }
}

//auto property
public int Age
{
        get;
        set;
}
}
```

Partial kulcsszó

- Lehetővé teszi egy osztály, struktúra, interfész vagy metódus több fájlba történő szétdarabolását.
- FONTOS: az összetartozó partial elemeknek egy assembly-ben kell lenniük.
- Tipikus felhasználás: UI és code-behind

virtual kulcsszó

- Lehetővé teszi, hogy adott implementációt a leszármazott osztályban felüldefiniáljunk.
- Leszármaztatott osztályban az override kulcsszót kell használnunk

Virtual

```
public class Shape
{
    protected double _x, _y;
    public virtual double Area()
    {
        return _x  * _y;
    }
}
public class Circle : Shape
{
    public override double Area()
    {
        return 3.14 * _x * _x;
    }
}
```

Sealed kulcsszó

- Ha osztálynál használjuk akkor jelezzük, hogy nem lehet belőle leszármaztatni.
- sealed class B : A
- Metódusnál pedig meg tudjuk határozni, hogy a leszármazott osztályban ne lehessen felüldefiniálni a függvényt

Sealed method

```
class Y
{
    sealed protected void F() { Console.WriteLine("Y.F"); }
    protected virtual void F2() { Console.WriteLine("Y.F2"); }
}

class Z : Y
{
    // Attempting to override F causes compiler error CS0239.
    // protected override void F() { Console.WriteLine("Z.F"); }

    // Overriding F2 is allowed.
    protected override void F2() { Console.WriteLine("Z.F2"); }
}
```

Abstract módosító

- Hiányzó, vagy nem teljes implementáció
- Lehet használni: class, method, property, indexer, event
- Ha osztálynál használjuk
  - jelezzük, hogy ősosztályként akarjuk használni
  - nem lehet példányosítani
  - nem lehet használni a sealed módosítóval együtt
  - leszármazott osztályban minden abstract metódust implementálni kell
- Ha metódusnál használjuk
  - azok alapból virtualis metódusok
  - csak abstract osztályban hozhatóak létre
  - leszármazott osztályban a metódusnál override kulcsszót kell használni.
  - virtual és static nem használható

Abstract class and methods

```
abstract class BaseClass
{
    protected int _x = 100;

    public abstract void AbstractMethod() {...}; // Abstract method

    public abstract int X { get; }  // Abstract properties
}
class DerivedClass : BaseClass
{
    public override void AbstractMethod()
    {
        _x++;
    }

    public override int X   // overriding property
    {
        get
        {
            return _x + 10;
        }
    }
}
```

Láthatósági szintek

- private
  - Csak a típuson belül
- protected
  - Csak az típuson és a leszármazottjain belül
- internal
  - Csak az adott assembly-n belül
- protected internal
  - Protected VAGY Internal
- public
  - Korlátlan hozzáférés

Initializerek használata

- Konstruktor nélkül is lehet inicializálni egy objektumot, vagy listát.
- Olyan mintha a létrehozás után a property-ket használnánk, de egyszerűbb az írásmód.
- A kollekciónak implementálnia kell az IEnumerable interfészt

Initializer használata

```
struct Cat {
    int Age;
    string Name;
 }
//Object initializer
Cat cat = new Cat { Age = 10, Name = "Fluffy" };

//collection initializer
List<Cat> cats = new List<Cat> {          new Cat(){Name="S", Age=8 },
        new Cat(){Name="A", Age=14 }
};
```

Using utasítás

- IDisposable interfész
- Determinisztikus erőforrás felszabadítás
- GC nem tudja felszabadítani a nem menedzselt erőforrásokat (DB kapcsolat, fájl kezelés)
- A using biztosítja, hogy az Dispose() metódus megfelelően legyen meghíva a példányosított objektumon
  - még akkor is, ha kivétel dobódik
  - egy try-finally generálódik a háttérben

Implicit típusú lokális változók

- Nem kell kiírni a lokális változó típusát. A változó típusa a jobb oldali kifejezés típusa lesz
- **var** == a jobboldal típusa
- Továbbra is erősen típusos!
- A fordító kitalálja a környezetből hogy milyen típusú egy lokális változó
- var list = new List<string>(); == List<string> list = new List<string>();
- hasznos, ha nem akarjuk kiírni az osztály nevét pl: BoundTreeRewriterWithStackGuardWithoutRecursionOnTheLeftOfBinaryOperator

Köszönöm a figyelmet!

Alkalmazásfejlesztés II.

Dr. Dombi József Dániel

Szegedi Tudományegyetem
Informatikai Intézet
Szoftverfejlesztés Tanszék

2024

Tartalom

Fordítás ideje: 2024.10.02 16:19:24 git branch: nincs nincs

Delegate

- Erősen típusos függvény pointer”
- A System.MulticastDelegate és System.Delegate osztályok leszármazottai
- Csak a fordító származtathat belőlük
- Eseménykezelők is delegate-ek
- delegate példa: public delegate void PlayCallBack();

Event

- Küldő/fogadó osztály, objektum
- Feliratkozás (+=), leiratkozás (-=) operátorokkal
- 1 eseményre több feliratkozó lehet, illetve egy feliratkozó feliratkozhat több eseményre is
- Pl.: gombnyomás esemény a UI-on

Event

```
class A {
        //EventHandler egy delegate
        public event EventHandler AnEvent;
}
A a = new A();
a.AnEvent += 'esemenykezelo'
a.AnEvent -= 'esemenykezelo'
```

Event

```
//delegate definiálása
public delegate void PlayCallback();
private event PlayCallback MozartStarted;

//feliratkozás
Listener lis = new Listener();
MozartStarted += new PlayCallback(lis.Listen);

//feliratkozottak értesítése
public void StartMozart() {
        if (MozartStarted != null) {
                MozartStarted();
        }
}
```

Delegate

- A delegate típus deklarációja hasonló a függvény deklarációval
- Van egy visszatérési értéke és bármennyi bármilyen típusú paramétere lehet.
- A System.Actions és System.Func általános gyakran használt delegate-ek.
- System.Action: lehet bemenő paramétere és nincs visszatérési értéke
- System.Func: lehet bemenő paramétere és van visszatérési értéke
- System.Pred: lehet bemenő paramétere és logikai visszatérési értéke van
- System.Func-nál az utolsó paraméter típusa a visszatérési érték.

Delegate

```
delegate void TestDelegate(string s);
//teszt függvény
static void M(string s)
{
        Console.WriteLine(s);
}

//példányosítjuk a delegate típust
TestDelegate testDelA = new TestDelegate(M);
//hívás
testDelA("Hello. My name is M and I write lines.");
```

Action

```
delegate void DisplayMessage(string message);

private static void ShowWindowsMessage(string message)
{
      MessageBox.Show(message);
}

DisplayMessage messageTarget = ShowWindowsMessage;
//ugyanaz Action-nel. Sablon paraméterben határozzuk
// meg a bemenő típust.
Action<string> messageTarget = ShowWindowsMessage;
```

Func

```
delegate string[] ExtractMethod(string stringToManipulate,
                                int maximum);

private static string[] ExtractWords(string phrase, int limit)
 {
// csinálunk valamit
return new string[5];
 }

ExtractMethod extractMeth = ExtractWords;
//ugyanaz Func-val. Az utolsó sablon paraméter
// a return value típusa.
Func<string, int, string[]> extractMethod = ExtractWords;
```

Alkalmazásfejlesztés II.

Dr. Dombi József Dániel

Szegedi Tudományegyetem
Informatikai Intézet
Szoftverfejlesztés Tanszék

2024

Tartalom

Fordítás ideje: 2024.10.02 16:19:24 git branch: nincs nincs

Anonymous függvények

- Ahol delegate típust kell megadnunk tudunk használni névtelen függvényeket.
- Lambda kifejezéssel, vagy névtelen eljárássokkal tudunk létrehozni névtelen függvényeket.
- Ha egy függvényt csak egy delegate-nél akarunk használni, akkor nem szükséges külön deklarálni.

anonymous függvények

```
delegate void TestDelegate(string s);
static void M(string s)
{
        Console.WriteLine(s);
}

TestDelegate testDelA = new TestDelegate(M);

//anonymous method
TestDelegate testDelB = delegate(string s) { Console.WriteLine(s); };

//Lambda kifejezés
TestDelegate testDelC = (x) => { Console.WriteLine(x); };
testDelA("Hello. My name is M and I write lines.");
testDelB("Hello. My name is M and I write lines.");
testDelC("Hello. My name is M and I write lines.");
```

Alkalmazásfejlesztés II.

Dr. Dombi József Dániel

Szegedi Tudományegyetem
Informatikai Intézet
Szoftverfejlesztés Tanszék

2024

Tartalom

Fordítás ideje: 2024.10.02 16:19:24 git branch: nincs nincs

Lambda kifejezés

- Lambda kifejezések a következő alakúak lehetnek:
  - (input-parameters) => expression
  - (input-parameters) => <sequence-of-statements>
- A paraméter listát és törzset a => operátor választja el.
- a Lambda kifejezés delegate típusra konvertálódik. Ha nincs visszatérési érték, akkor Action, ha van visszatérési érték, akkor Func.
- Ha csak egy input paraméter van, akkor a zárójelek() opcionálisak, egyébként meg kötelező kiírni.
- a Fordító az input paraméterek típusát legtöbb esetben meg tudja határozni, ha nem, akkor nekünk kell megadni.

Lambda kifejezések

```
//x bemenő paraméter és a return érték az x*x
Func<int, int> square = x => x * x;
Console.WriteLine(square(5));
// Output: 25

//zárójel opcionális, nem kötelező kiírni
Func<int, int> square = (x) => x * x;
```

Lambda kifejezések

```
        //paraméter nélküli Lambda kifejezés
Action line = () => Console.WriteLine();

        //két input paraméterrel rendelkező lambda kifejezés,
        //A paraméterek ,-vel vannak elválasztva.
        Func<int, int, bool> testForEquality = (x, y) => x == y;

        //Az input paraméterek típusát is megadhatjuk
        Func<int, string, bool> isTooLong =
        (int x, string s) => s.Length > x;
```

Lambda kifejezések

```
//Statement lambda
Action<string> greet = name =>
{
    string greeting = "Hello␣{name}!";
    Console.WriteLine(greeting);
};
greet("World");
// Output: Hello World!
```

Alkalmazásfejlesztés II.

Dr. Dombi József Dániel

Szegedi Tudományegyetem
Informatikai Intézet
Szoftverfejlesztés Tanszék

2024

Tartalom

Fordítás ideje: 2024.10.02 16:19:24 git branch: nincs nincs

LINQ

- LINQ segítségével C# nyelvi szinten tudunk lekérdezéseket megadni.
- Általában a lekérdezések szöveg tipusúak és nincs fordítási támogatás.
- Hátrány, hogy meg kell ismernünk az egyes adat típusokhoz tartozó lekérdezési nyelvet: pl: SQL adatbázis, XML dokumentum, web szolgáltatás stb.
- LINQ segítségével query expression-öket tudunk írni és a szintaxisa a query-nek támogatja az alap lekérdezéseket mint például szűrés, rendezés, csoportosítás.

LINQ

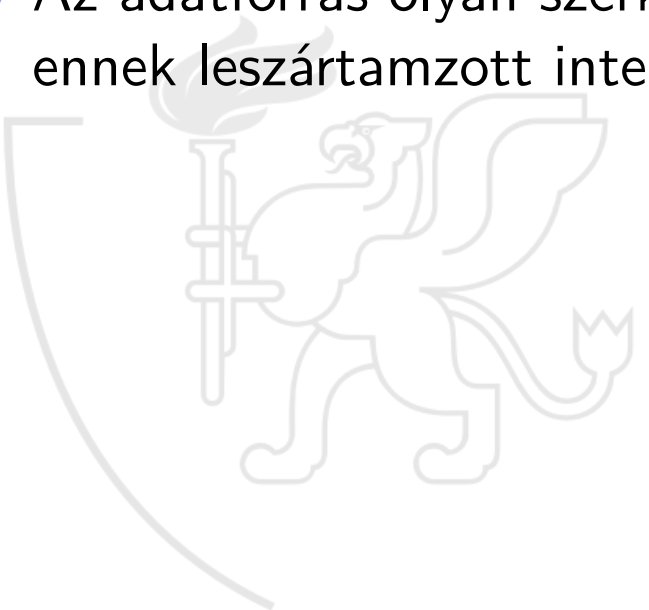
```
//példa tömb létrehozás
int[] scores = new int[] { 97, 92, 81, 60 };

//lekérdezés definiálsa
IEnumerable<int> scoreQuery =
from score in scores
where score > 80
select score;

//Lekérdezés végrehajtása
foreach (int i in scoreQuery)
{
 Console.Write(i + "␣");
}
```

LINQ

- Query expression-öket lekérdezésre és adat transzformációra használhatjuk a LINQ-et támogató adatszerkezeteken.
- LINQ lekérdezéseket C# nyelven fogalmazzuk meg.
- A változók a LINQ-ben erősen típusolt. A fordító ki tudja találni.
- LINQ lekérdezés akkor hajtódik végre, ha lekérdezési változón iterálunk (pl: foreach).
- A LINQ lekérdezés átkonvertálódik Standard Query Operátor műveletekre.
- Támogatott adatszerkezetek: LINQ TO XML, LINQ TO OBJECTS, LINQ TO ENTITIES,...

LINQ felépítése

- A query egy olyan lekérdezés, amely adatokat kap az adatforrástól.
- LINQ lekérdezés során végig object-ekkel foglalkozunk.
- A LINQ query három részből áll:
  - Adatforrás meghatározás
  - Lekérdezés megfogalmazása
  - Lekérdezés végrehajtása
- Az adatforrás olyan szerkezet, amely megvalósítja IEnumerable, vagy ennek leszártamzott interface-ét. (pl: IQueryable)



LINQ query

- A lekérdezés során meg kell adnunk a feltételeket, hogy az adatforrásból mit szeretnék lekérdezni.
- A query opcionálisan tartalmazhatja, hogy az adatot hogyan rendezzük, csoportosítsuk, vagy hogyan nézzen ki.
- A lekérdezést úgynevezett query variable-ba tároljuk.
- Query szintaxisa három fő részből áll
  - from: az adatforrás meghatározása
  - where: szűrési feltétel
  - select: a visszatérési elemek tipusát határozza meg.

LINQ query végrehajtás

- A query variable csak a lekérdezési parancsot tartalmazza.
- A query akkor értékelődik ki, ha alkalmazzuk például foreach-ben.
- Az eredmény a foreach során értékelődik ki és a foreach-ben lévő változó fogja tartalmazni a query egyes elemeit.
- A query akkor is kiértékelődik, ha aggregációs műveletet használunk (count, max, First, Average). Ebben az esetben először végig kell iterálni az adatforráson, hogy meghatározzuk az értéket. Viszont az érték egyetlen egy elem lesz.
- A query akkor is kiértékelődik, ha a ToList, vagy ToArray metódusokat meghívjuk. Ebben az esetben elmentjük az értékeket.

LINQ

```
var evenNumQuery =
    from num in numbers
    where (num % 2) == 0
    select num;
//query végrehajtása
int evenNumCount = evenNumQuery.Count();

//query azonnali végrehajtása
List<int> numQuery2 =
    (from num in numbers
     where (num % 2) == 0
     select num).ToList();
```

LINQ Generikus objektumok

```
//az eredmény Customer object lesz és ezt ki is tudjuk használni
//a lekérdezés során. Lást a customer-nek van city property-je.
IEnumerable<Customer> customerQuery =
    from cust in customers
    where cust.City == "London"
    select cust;

foreach (Customer customer in customerQuery)
{
    Console.WriteLine(customer.LastName + ", " + customer.FirstName);
}
```

LINQ query

- Az első lépés a data source meghatározása: **from** változó **in** datasource
- Változót úgy kell kezelni, mintha foreachben használnánk. A változó tipusát nem kell megadnunk.
- Tovább lehet módosítani az eredményt a **let** kulcsszóval, azaz tovább bonthatjuk az adatforrás eredményét.
- A **where** kulcsszóval szűrést tudunk végrehajtani. A where feltételben a logikai kifejezést kell kiértékelni. A kiértékelés során azokat az elemeket kapjuk vissza, ami eleget tesz a feltételnek. Logikai operátorokat lehet használni.
- Az orderby segítségével tudjuk rendezni az eredményt: ascending, descending.
- A csoportosítást a **group** segítségével tudjuk elvégezni.
- A **select** segítségével a projekciót tudjuk megadni.

LINQ lekérdezések

```
//customers-ből kérdezünk le és a változó név a cust.
var queryAllCustomers = from cust in customers
                        select cust;

string[] strings =
{
"A penny saved is a penny earned.",
"The early bird catches the worm.",
};

// A mondatokat bontsuk szét szavakra.
// A let segítségével tudjuk szavakra bontani.
var earlyBirdQuery =
from sentence in strings
let words = sentence.Split(' ')
from word in words
let w = word.ToLower()
select w;
```

LINQ lekérdezések

```
//szűrés. Tudjuk, hogy customer objektumról van szó
//és annak van City propertyje.
var queryLondonCustomers =
from cust in customers
where cust.City == "London"
select cust;

// logikai és-re példa
where cust.City=="London" && cust.Name == "Devon"

//logikai vagy-ra példa
where cust.City == "London" || cust.City == "Paris"
```

LINQ lekérdezések

```
//név szerint rendezés A->Z.
var queryLondonCustomers3 =
from cust in customers
where cust.City == "London"
orderby cust.Name ascending
select cust;
```

LINQ lekérdezések

```
//Csoportosítás során az eredmény beágyazott lista.
var queryCustomersByCity =
      from cust in customers
      group cust by cust.City;

// customerGroup is an IGrouping<string, Customer>
foreach (var customerGroup in queryCustomersByCity)
{
//a csoport kulcsa
Console.WriteLine(customerGroup.Key);
//csoportban lévő elemek.
foreach (Customer customer in customerGroup)
{
Console.WriteLine("␣␣␣␣{0}", customer.Name);
}
}
```

LINQ lekérdezések

```
 // Teljes objektumok visszaadása
 IEnumerable<Student> studentQuery1 =
 from student in app.students
 where student.ID > 111
 select student;

// Studentnek van egy Last propertyje, ami a keresztnevet tartalmazza
IEnumerable<String> studentQuery2 =
from student in app.students
where student.ID > 111
select student.Last;

//Studentnek a Scoore tömbbe tároljuk az eredményét
//A legelső eredményét növeljük 10%-val.
IEnumerable<double> studentQuery5 =
from student in app.students
where student.ID > 111
select student.Scores[0] * 1.1;
```

LINQ lekérdezések

```
// Eddigi eredményének átlaga
IEnumerable<double> studentQuery6 =
from student in app.students
where student.ID > 111
select student.Scores.Average();

// Új típus létrehozása new-val. Ekkor var-t kell használnunk.
var studentQuery7 =
from student in app.students
where student.ID > 111
select new { student.First, student.Last };
foreach (var item in studentQuery7)
{
Console.WriteLine("{0}, {1}", item.Last, item.First);
}
```

Köszönöm a figyelmet!

Alkalmazásfejlesztés II.

Dr. Dombi József Dániel

Szegedi Tudományegyetem
Informatikai Intézet
Szoftverfejlesztés Tanszék

2024

Tartalom

Fordítás ideje: 2024.10.09 16:20:39 git branch: nincs nincs

Szálak vs Process

- A processz az operációs rendszer egysége, minden alkalmazás külön processz
- Egy alkalmazás összes szála egy processzben fut
- A processzek teljesen izoláltak
- A szálak közös (heap) memóriához férnek hozzá

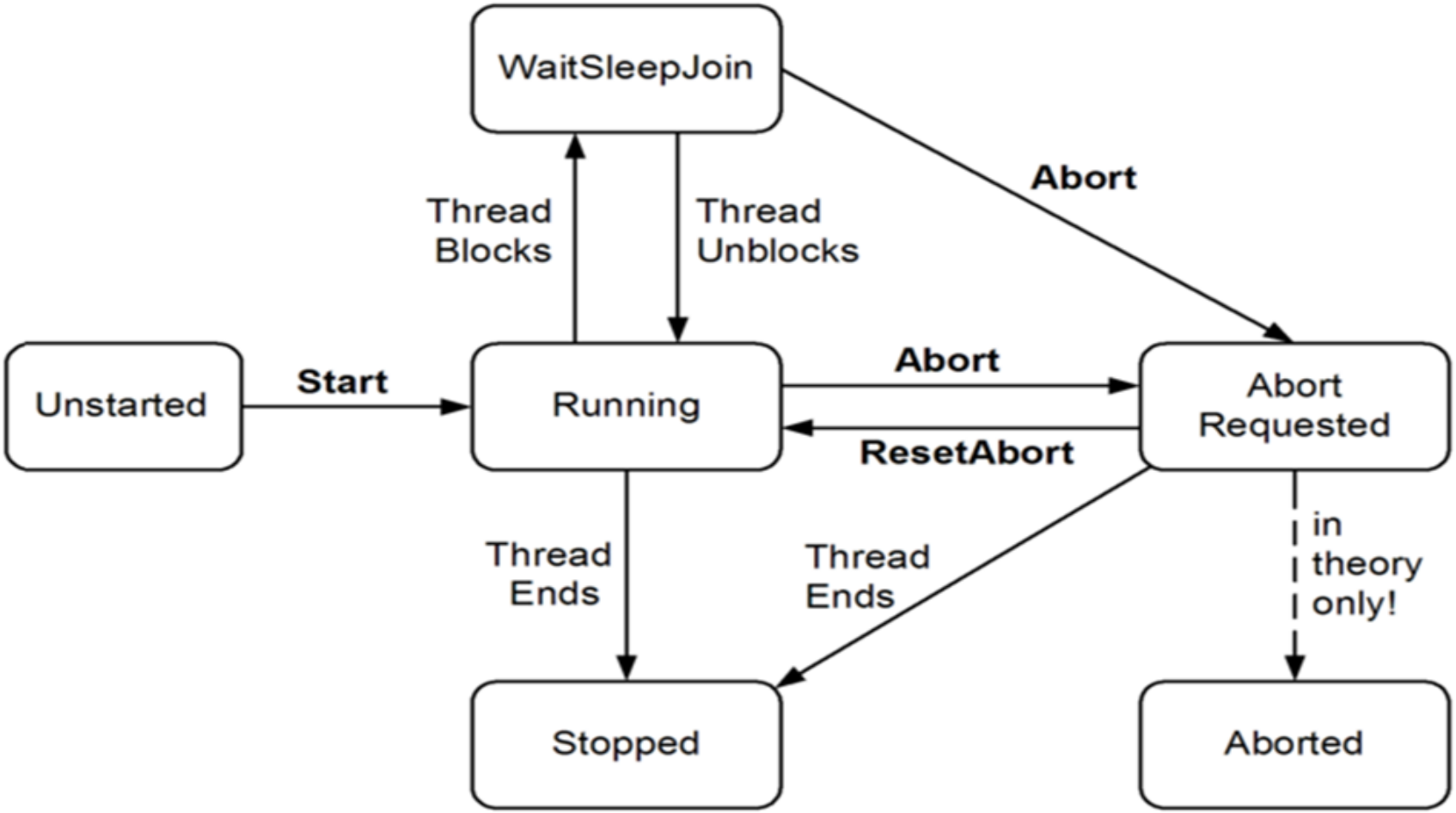
Threads alapok

- A C# támogatja a párhuzamos kódvégrehajtást
- Egy programnak induláskor a CLR és az operációs rendszer egy szálat indít
- Futás közben új szálak indíthatók
- A többszálúság alapja: Thread osztály

Thread

```
using System;
using System.Threading;

class ThreadTest {
        static void Main() {
                Thread t = new Thread (WriteY);
                t.Start();       // Run WriteY on the new thread
                while (true)
                // Write 'x' forever
                Console.Write ("x");
        }
        static void WriteY() {
                while (true)
                // Write 'y' forever
                Console.Write ("y");
        }
}
//result: xxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyxx...
```

Thread state



Threads Prioritás

- Egy szál prioritása meghatározza, hogy mekkora időszeletet kap ugyanazon processz többi szálához képest
- Vezérlése: **Priority** tulajdonság
- Lehetséges értékek a következő skáláról: enum ThreadPriority Lowest, BelowNormal, Normal, AboveNormal, Highest
- Egy szál prioritását továbbra is behatárolja az alkalmazás processzének prioritása

Threads Háttérszál

- Alapértelmezés szerint a szálak az előtérben futnak
- Amíg legalább egy szál fut előtérben, addig az alkalmazás „életben marad”
- Ha már csak háttérben fut szál, az alkalmazás terminál (a szálakkal együtt)
- Állapotvezérlés: **IsBackground** tulajdonság

Background task

```
Thread worker =
        new Thread (delegate() { Console.ReadLine(); });
worker.IsBackground = true;
worker.Start();
```

Threads zárolás

- A zárolás biztosítja, hogy egy kódrészlethez egyszerre csak egy szál férjen hozzá
- **lock** ... szerkezetet használhatjuk.
- Csak az a szál oldhatja fel a zárolt block-ot, aki zárolta.
- A lock akkor is feloldásra kerül, ha exception dobódik.
- Különböző megosztandó erőforrásokhoz különböző lock objektumokat használjunk.

Zárolás

```
public class Account
{
    private readonly object balanceLock = new object();
    private decimal balance;

    public decimal Debit(decimal amount)
    {
    lock (balanceLock)
    {
    ...
        balance = 3;
    }
return balance;
}
```

Alkalmazásfejlesztés II.

Dr. Dombi József Dániel

Szegedi Tudományegyetem
Informatikai Intézet
Szoftverfejlesztés Tanszék

2024

Tartalom

Fordítás ideje: 2024.10.09 16:20:39 git branch: nincs nincs

Async programozás cél

- Alkalmazás teljesítményének növelése.
- Alkalmazás folyamatosságának biztosítása.
- Hagyományos megoldás elbonyolítja a kódot.
- Az alkalmazás több feladatot is el tud látni egyszere.

Async tipikus felhasználás

- Web elérés (HttpClient)
- File elérés (StreamWriter, StreamReader)
- Kép kezelés (MediaCapture)
- WCF programming (Sync and Async operation)

Probléma

```
public void GetData()
{
        MyServiceClient client =new MyServiceClient();
        client.DoWorkCompleted += client_DoWorkCompleted;
        client.DoWorkAsync('Parameter');
}


void client_DoWorkCompleted(object sender,
        DoWorkCompletedEventArgs e)
{
        if (e.Error == null)
        {var result = e.Result;
        // update ui
        }
        else{//error handling}}
```

Probléma

- Kód szeparáció!
- Callback nem UI thread-en fut.
- Kép kezelés (MediaCapture)
- További async műveletek tovább bonyolítják a kódot.

Megoldás az Async alkalmazása

- Fordítási trükk.
- Callback egyből az async után található
- Callback ugyanazon a szálon fut
- Bármennyi await alkalmazható

Megoldás async

```
public async void GetDataAsync()
{
        try
        {
        MyServiceClient client =new MyServiceClient();
        var result = await client.DoWorkAsync('Parameter');
        // update ui
        }
        catch (Exception ex)
        {
        // error handling
        }
}
```

Async

- Az Asnyc használatakor két fontos kulcszó az **async** és **await**
- Asszinkron és szinkron eljárásokat tudunk készíteni.
- Az eljárásoknál az **async** jelzi, hogy az eljárás tartalmazhat asszinkron hívást.
- Bármennyi await alkalmazható

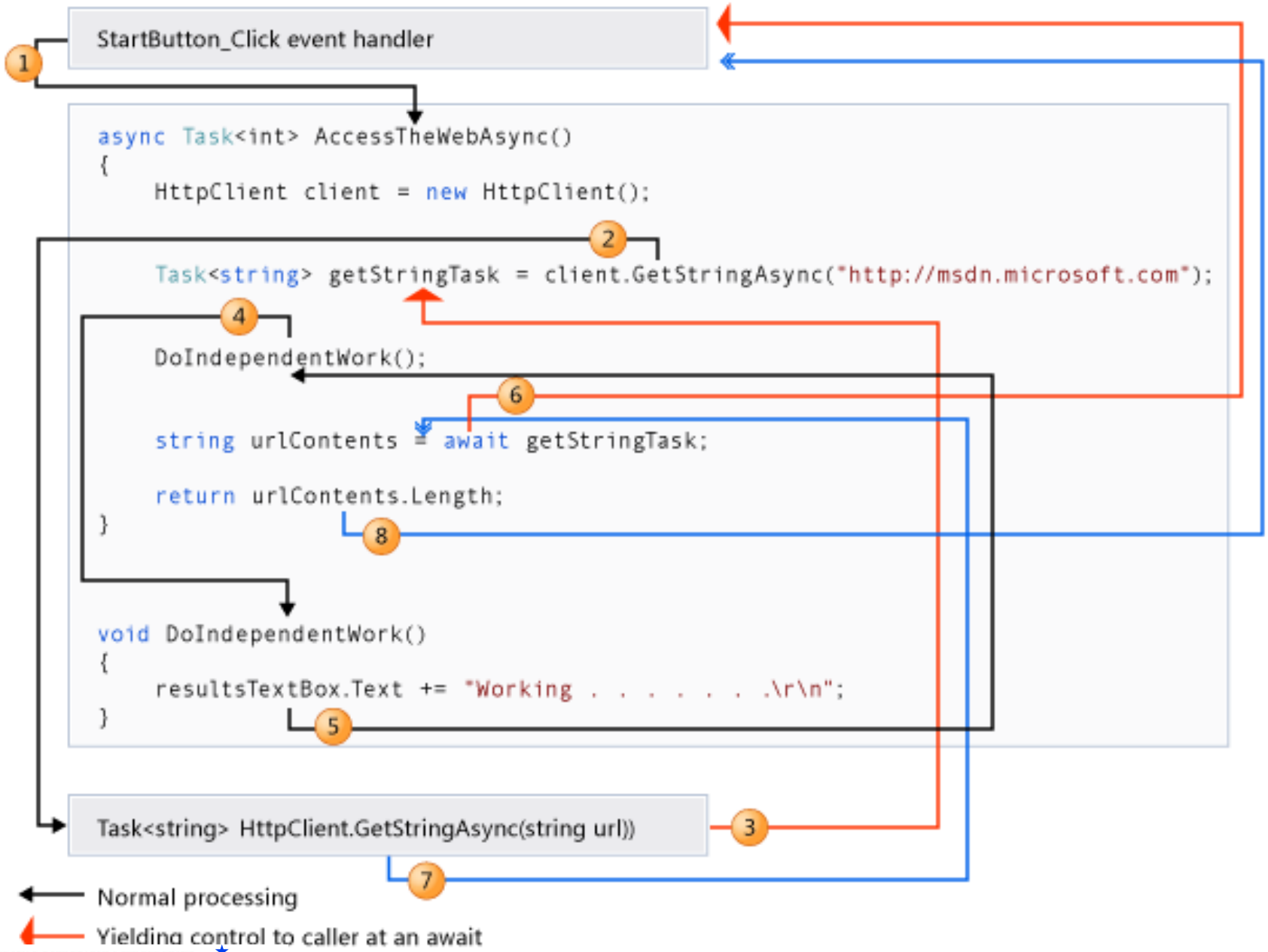
Async példa

```
async Task<int> AccessTheWebAsync()
{
    using (HttpClient client = new HttpClient())
    {
        Task<string> getStringTask =
        client.GetStringAsync("https://docs.microsoft.com");

        DoIndependentWork();

        string urlContents = await getStringTask;

        return urlContents.Length;
    }
}
```

Async függvény

- Függvényt megjelöljük async kulcsszóval.
- A visszatérési érték lehet: void, Task, Task<type>
- A függvény neveket Async suffix-al látjuk el.
- await segítségével futtatjuk le és várjuk meg az eredményt.
- több await is lehet egy függvényen belül
- await hatására a függvényből kilépünk és ha az asszinkron metódus befejeződik, akkor térünk vissza.

Thread state



Async visszatérési érték

- Async három féle visszatérési értékkel térhet vissza:
  - Task<TResult>: aminek van visszatérési értéke
  - Task: Ami Async metódust hajt végre, de nincs visszatérési értéke
  - void
  - C# 7.0-tól kezdve bármely típus, amely megvalósítja a ICriticalNotifyCompletion interface-t (GetAwaiter eljárás).

Task<TResult>

- Azon async eljárásoknál használjuk, ahol van return is, amely a TResult tipussal tér vissza.
- ha nem hívjuk meg a await-et, akkor Task<TResult>-tal tér vissza. Ennek van egy **Result** property-je.
- await hívásra a függvény végrehajtódik és a Result-ba kerül az eredmény.
- Ha még nincs kiértékelve a függvény és meghívjuk a Result-ot, akkor blokkolja a szálat és végrehajtja az utasítást.

Task<TResult>

```
static async Task<int> GetLeisureHours()
    {
        //Task.FromResult is a placeholder for actual work
                            //that returns a string.
var today =
await Task.FromResult<string>(DateTime.Now.DayOfWeek.ToString());

        // válasz feldolgozása
        int leisureHours;
        if (today.First() == 'S')
            leisureHours = 16;
        else
            leisureHours = 5;

        return leisureHours;
    }
//int tipussal tér vissza
int result = await GetLeisureHours()
```

Task

- Azon async eljárásoknál használjuk, ahol nincs return.
- Ezek lehetnek void-ok is, ha szinkron módon akarjuk végrehajtani. Ebben az esetben exception-t nem kapunk el.
- Task visszatérési érték esetén a hívó eljárásnál az await-et kell használnunk.
- Ebben az esetben a Task-nál nincs Result property.

Task

```
static async Task WaitAndApologize()
    {
        // Task.Delay is a placeholder for actual work.
        await Task.Delay(2000);
        // Task.Delay delays the following line by two seconds.
        Console.WriteLine("\nSorry for the delay. . . .\n");
    }

static async Task DisplayCurrentInfo()
    {
        await WaitAndApologize();
        Console.WriteLine($"Today is {DateTime.Now:D}");
        Console.WriteLine($"The time: DateTime.Now.TimeOfDay:t}");
        Console.WriteLine("The current temperature is 76 degrees.");
    }
```

Task exception

```
public async Task DoSomethingAsync()
{
    Task<string> theTask = DelayAsync();

    try
    {
        string result = await theTask;
        Debug.WriteLine("Result: " + result);
    }
    catch (Exception ex)
    {
        Debug.WriteLine("Exception Message: " + ex.Message);
    }
    Debug.WriteLine("Task IsCanceled: " + theTask.IsCanceled);
    Debug.WriteLine("Task IsFaulted:  " + theTask.IsFaulted);
    if (theTask.Exception != null)
    {
        Debug.WriteLine("Task Exception Message: "
            + theTask.Exception.Message);
        Debug.WriteLine("Task Inner Exception Message: "
            + theTask.Exception.InnerException.Message);
    }
}
```

Köszönöm a figyelmet!