Deep dive

Albert István

ialbert@aut.bme.hu

Q.B. 221, 1662

BME AUT

Automatizálási és
Alkalmazott
Informatikai Tanszék

Tartalom

Mit ír ki?

```
1 reference
static void addValue( int v )
{
    v++;
}

0 references
static void Main(string[] args)
{
    int a = 1;
    addValue(a);
    Console.WriteLine(a);
```


Span : a nagy dobás

- Tetszőleges résztömb biztonságos elérése és módosítása

```
var arr = new byte[10];
Span<byte> bytes = arr; // Implicit cast from T[] to Span<T>
```

- Ugyanarra az objektumra mutat, de belülre

```
Span<byte> slicedBytes = bytes.Slice(start: 5, length: 2);
slicedBytes[0] = 42;
slicedBytes[1] = 43;
```


Memória kezelő algoritmus típusok

- Determinisztikus
  - > Manuális (C, C++)
  - > Referencia számlált (VB 6)
- Heurisztikus
  - > Mark and sweep
  - > Copy and Collect
  - > Mark and compact

Gyenge referenciák

- Úgy hivatkozunk az objektumra, hogy a GC felszabadíthatja
  - > Memória hiány esetén a GC tud takarítani
- Használat előtt egy erős referenciát szerzünk az objektumra
- Használati esetek
  - > Klasszikus cache
  - > Az alkalmazás által használt objektumokon időzített feladat elvégzése

Unsafe Code

- Teljes kontrol
  - > Extrém optimalizáció
  - > Meglévő bináris struktúrák kezelése
  - > Együttműködés régi kóddal
- Unsafe szerelvény, kód
  - > Pointer típusok, pointer aritmetika
  - > ->, *, [], & operátorok
  - > Unsafe kasztolás
  - > GC elkerülése

Layout

- A mezők a deklarálás sorrendjében vannak a memóriában
- Alapértelmezett igazítás: 32/64 bit
  - > StructLayout attribútummal megváltoztatható
- 32 bit: data >= 4 byte akkor 4-byte-ra igazított
- 64 bit: data >= 8 byte akkor 8-byte-ra igazított
- Lehet nem igazított
  - > HW vagy OS kezeli
  - > Komoly teljesítmény probléma!

Mit ír ki?

```
1 reference
static void addValue( int v )
{
    v++;
}

0 references
static void Main(string[] args)
{
    int a = 1;
    addValue(a);
    Console.WriteLine(a);
```

Mit ír ki?

```
1 reference
static void addRefValue(ref int v)
{
    v++;
}

0 references
static void Main(string[] args)
{
    int a = 1;
    addRefValue(ref a);
    Console.WriteLine(a);
```

Mit ír ki?

```
2 references
class A { public int V; }

1 reference
static void addValue(A a)
{
    a.V++;
}

0 references
static void Main(string[] args)
{
    var a = new A { V = 1 };
    addValue(a);
    Console.WriteLine(a.V);
```

Mit ír ki?

```
1 reference
static void addOtherValue(A a)
{
    a = new A { V = 1 };
    a.V++;
}

0 references
static void Main(string[] args)
{
    var a = new A { V = 1 };
    addOtherValue(a);
    Console.WriteLine(a.V);
```

Mit ír ki?

```
1 reference
static void addOtherRefValue(ref A a)
{
    a = new A { V = 1 };
    a.V++;
}

0 references
static void Main(string[] args)
{
    var a = new A { V = 1 };
    addOtherRefValue(ref a);
    Console.WriteLine(a.V);
```

Mit ír ki?

```
1 reference
static void replace(string s)
{
    s.Replace('a', 'b');
}

0 references
static void Main(string[] args)
{
    string a = "aaa";
    replace(a);
    Console.WriteLine(a);
```

Mit ír ki?

```
1 reference
static void replaceRef(ref string s)
{
    s.Replace('a', 'b');
}

0 references
static void Main(string[] args)
{
    string a = "aaa";
    replaceRef(ref a);
    Console.WriteLine(a);
```

Mit ír ki?

```
1 reference
static void replaceRef2(ref string s)
{
    s = s.Replace('a', 'b');
}

0 references
static void Main(string[] args)
{
    string a = "aaa";
    replaceRef2(ref a);
    Console.WriteLine(a);
```

Típusok a .NET-ben

- Érték típus
  - int
  - Inline, verem
  - Másolat
- Referencia típus
  - Class A
  - Heap
  - Referencia

Érték típusok

- Például
  - Int, DateTime, Decimal, enum,
- Másolódnak
  - Nem „egy példány” van belőlük
  - A méret számít, kis objektumok előnyben
- Korlátozások
  - Nem örökölhetnek
  - Nem lehet belőlük örökölni

Mit ír ki?

```
2 references
class B
{
    2 references
    public virtual void Do()
    { Console.WriteLine("B"); }
}

1 reference
class C : B
{
    2 references
    public override void Do()
    { Console.WriteLine("C"); }
}

0 references
static void Main(string[] args)
{
    B v = new C();
    v.Do();
```

Öröklés …

- Miért jó örökölni?
  - Különböző típusok közösen kezelhetőek: alaposztály
  - Virtuális metódusok megváltoztatják a viselkedést

- Hogyan működik?
  - Honnan tudja, hogy melyik Do-t kell meghívni?

```
0 references
static void Main(string[] args)
{
    B v = new C();
    v.Do();
```

Típusinformáció

- Referencia típus
  - Típus információt tárol a példánynál…
  - Virtuális metódusok címei, interfészek, GetType(), …
- Érték típus
  - A típus információ csak a „változóból” jön!
  - A változó típusa alapján hívja meg a metódust!
  - Így az int csak 4 byte!

```
0 references
static void Main(string[] args)
{
    B v = new C();
    v.Do();
```

„Virtuális” metódus hívás – int vs class

```
            int a = 1;
00007FFBAC070B87  mov          dword ptr [rbp+38h],1
            a.ToString();
00007FFBAC070B8E  lea          rcx,[rbp+38h]
00007FFBAC070B92  call         00007FFC07F003D0
00007FFBAC070B97  mov          qword ptr [rbp+28h],rax
00007FFBAC070B9B  nop

            v.ToString();
00007FFBAC070B9C  mov          rcx,qword ptr [rbp+40h]
00007FFBAC070BA0  mov          rax,qword ptr [rbp+40h]
00007FFBAC070BA4  mov          rax,qword ptr [rax]
00007FFBAC070BA7  mov          rax,qword ptr [rax+40h]
00007FFBAC070BAB  call         qword ptr [rax]
00007FFBAC070BAD  mov          qword ptr [rbp+20h],rax
00007FFBAC070BB1  nop
```

Érdekesség: GetType

```
            object obj = a;
00007FFBAC070BF7  mov          rcx,7FFC081B1B68h
00007FFBAC070C01  call         00007FFC0BB3D2C0
00007FFBAC070C06  mov          qword ptr [rbp+48h],rax
00007FFBAC070C0A  mov          rcx,qword ptr [rbp+48h]
00007FFBAC070C0E  mov          eax,dword ptr [rbp+6Ch]
00007FFBAC070C11  mov          dword ptr [rcx+8],eax
00007FFBAC070C14  mov          rcx,qword ptr [rbp+48h]
00007FFBAC070C18  mov          qword ptr [rbp+60h],rcx
```

```
            a.GetType();
00007FFBAC070CA0  mov          rcx,7FFC081B1B68h
00007FFBAC070CAA  call         00007FFC0BB3D2C0
00007FFBAC070CAF  mov          qword ptr [rbp+48h],rax
00007FFBAC070CB3  mov          rcx,qword ptr [rbp+48h]
00007FFBAC070CB7  mov          eax,dword ptr [rbp+6Ch]
00007FFBAC070CBA  mov          dword ptr [rcx+8],eax
00007FFBAC070CBD  mov          rcx,qword ptr [rbp+48h]
00007FFBAC070CC1  call         00007FFC0BA4A200
00007FFBAC070CC6  mov          qword ptr [rbp+30h],rax
00007FFBAC070CCA  nop
```

Dobozolás - boxing

- Minden esetben, amikor példányhoz tartozó típus információra van szükség
- Teljesítmény
  - Memória nyomás, GC terhelés
  - CPU terhelés: másolás, virtuális hívás

```
int a = 1;
object obj = a;
obj.ToString();

IComparable ic = a;
ic.CompareTo(2);

object.ReferenceEquals(a, 3);
```

Dobozolás overhead

- Mekkora egy „üres object” mérete?
- 32 bit: két pointer, 8 byte
- 64 bit: két pointer, 16 byte
  - > + alignment: 8 byte

```
memory pressure for 1000000 int in array: 4000 Kb
memory pressure for 1000000 int in List<int>: 4000 Kb
memory pressure for 1000000 int in List<object>: 31999 Kb
Done.
Press any key to continue . . .
```

A mérés

```
var m0 = GC.GetTotalMemory(true);
var a = new int[1_000_000];
var m1 = GC.GetTotalMemory(true);
Console.WriteLine($"memory pressure fo

m0 = GC.GetTotalMemory(true);
var l = new List<int>(1_000_000);
for (int i = 0; i < l.Capacity; i++)
    l.Add(0);
m1 = GC.GetTotalMemory(true);
Console.WriteLine($"memory pressure fo

m0 = GC.GetTotalMemory(true);
var l2 = new List<object>(1_000_000);
for (int i = 0; i < l2.Capacity; i++)
    l2.Add(0);
m1 = GC.GetTotalMemory(true);
Console.WriteLine($"memory pressure fo
```

Span : a nagy dobás

- Tetszőleges résztömb biztonságos elérése és módosítása

```
var arr = new byte[10];
Span<byte> bytes = arr; // Implicit cast from T[] to Span<T>
```

- Ugyanarra az objektumra mutat, de belülre

```
Span<byte> slicedBytes = bytes.Slice(start: 5, length: 2);
slicedBytes[0] = 42;
slicedBytes[1] = 43;
```

A verem használatával

- A vermen lehet gyorsan, biztonságosan foglalni helyet

```
Span<byte> bytes = stackalloc byte[2]; // Using C# 7.2 stackalloc support for spans
bytes[0] = 42;
bytes[1] = 43;
Assert.Equal(42, bytes[0]);
Assert.Equal(43, bytes[1]);
bytes[2] = 44; // throws IndexOutOfRangeException
```

- A vermen lévő memória nem használható úgy, mint egy heapen lévő objektum!

C#: ref return

- A visszaadott objektum módosítható!
- Például egy indexer lehet ilyen:

```
public ref T this[int index] { get { ... } }
```

- Például : érték típusok esetén a korábbi indexer nem módosítható:

```
struct MutableStruct { public int Value; }
...
Span<MutableStruct> spanOfStructs = new MutableStruct[1];
spanOfStructs[0].Value = 42;
Assert.Equal(42, spanOfStructs[0].Value);
var listOfStructs = new List<MutableStruct> { new MutableStruct() };
listOfStructs[0].Value = 42; // Error CS1612: the return value is not a variable
```

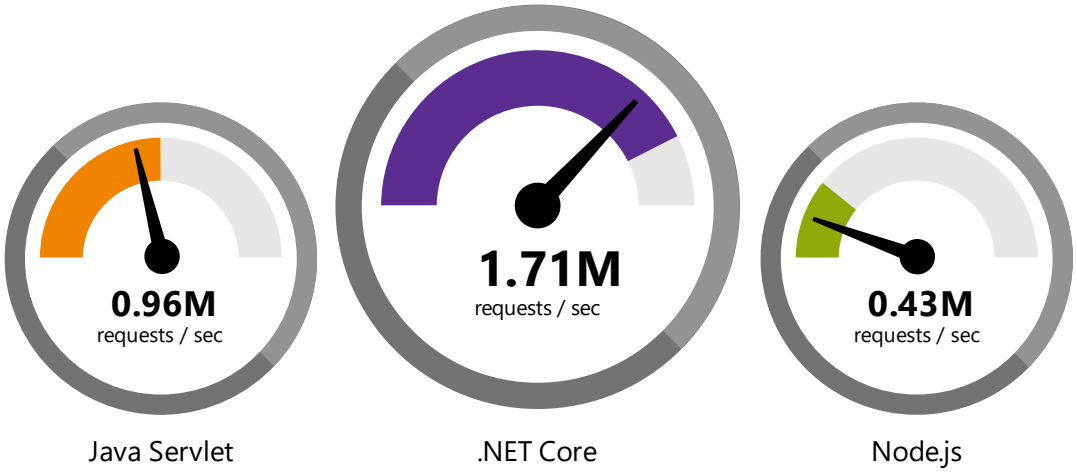
span megvalósítása, felhasználás

- Speciális CLR, JIT támogatás

```
public readonly ref struct Span<T>
{
  private readonly ref T _pointer;
  private readonly int _length;
  ...
}
```

- Szöveg feldolgozás (HTTP, JSON, …)
- Formázás
- Memory pool
- Natív kód interop

Már a .NET Core 2 is gyors

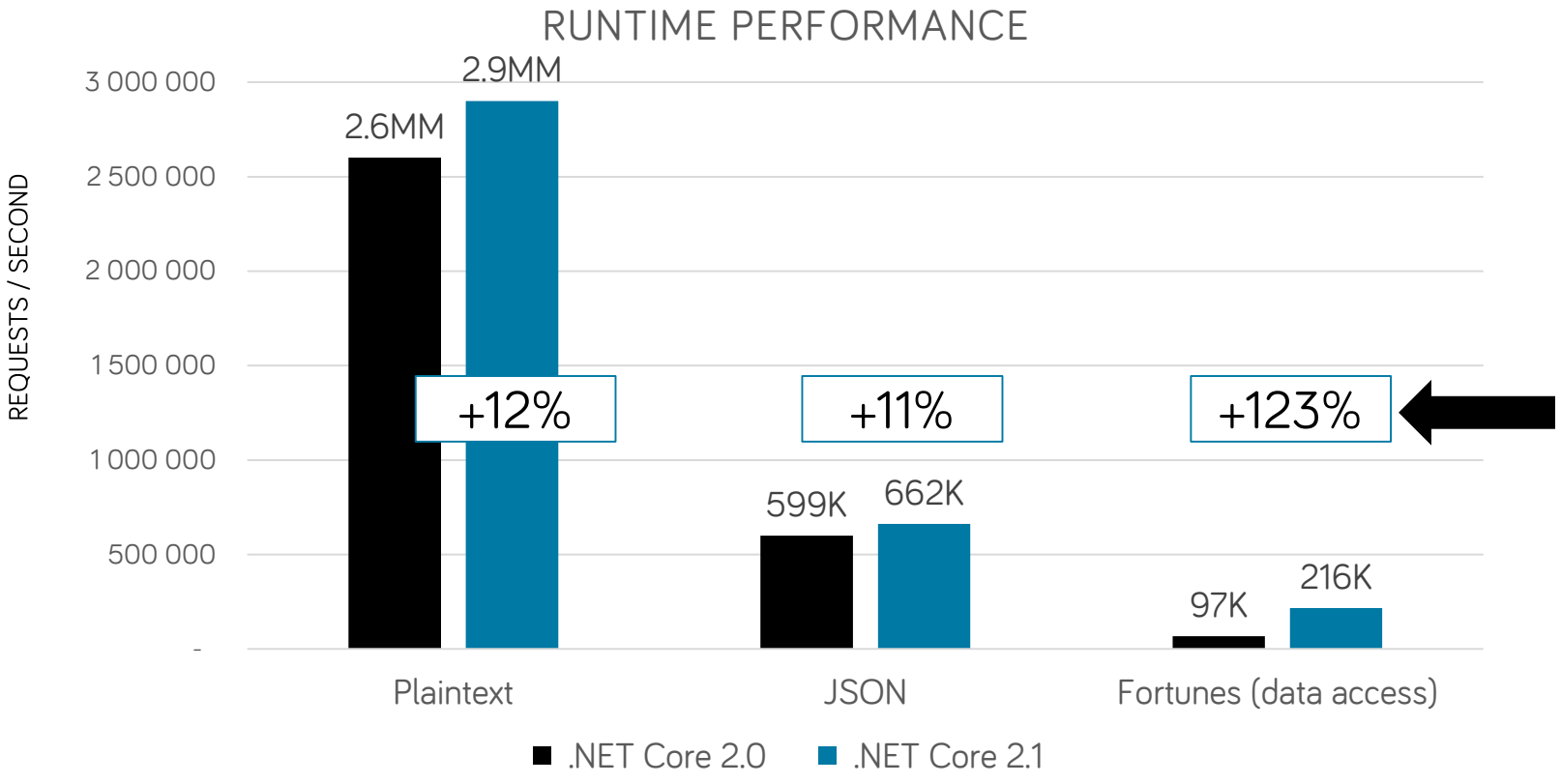


Data sourced from official tests available at TechEmpower Round 14.

"Using the same-size server, we were able to go from 1,000 requests per second per node with Node.js to 20,000 requests per second with .NET Core."
— Raygun

https://www.microsoft.com/net/customers

A .NET Core 2.1 még gyorsabb!



Data sourced from tests in our labs on same hardware as TechEmpower

Memória kezelő algoritmus típusok

- Determinisztikus
  - Manuális (C, C++)
  - Referencia számlált (VB 6)
- Heurisztikus
  - Mark and sweep
  - Copy and Collect
  - Mark and compact

Referencia számlálás

- Minden komponens nyilván tartja, hogy hányan használják
- Megszünteti saját magát, amikor az utolsó kliens is elengedi a referenciát rá
- Például: COM, nagy C++ projektek
- **Hátrány: körkörös referenciák feloldása**

Mark and Sweep

- Egymás után foglalja az objektumokat a heapen
- Valamikor megjelöli a nem használt objektumokat
- Felszabadítja az általuk használt memóriát
  - C Runtime Heap (manuális), LOH (automatikus)
  - Szabad területeket láncol egymáshoz
- Hátrányok
  - Allokáció: a szabad terület listát kell bejárni
  - Fregmentáció

Copy and Collect

- Két heapet tart fenn
- Csak az egyiken foglal
- A szemétgyűjtés során a használt objektumokat átmásolja az egyik heapről a másikra
  - > És megcseréli a szerepeket
- SSCLI
- Hátrányok
  - > Minden objektumot másolni kell
  - > Hatalmas memória igény

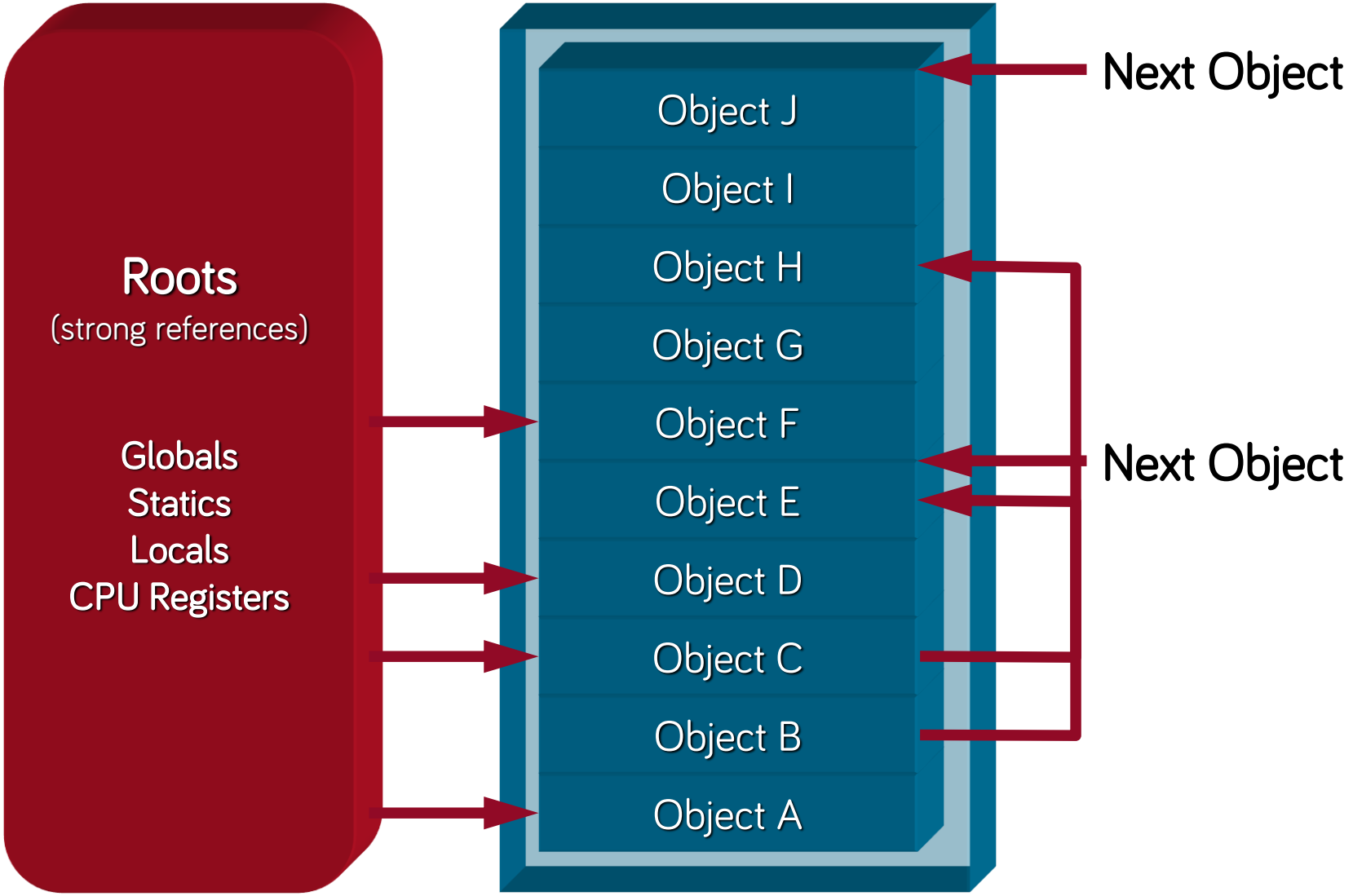
Mark and Compact

- Hasonló a Mark and Sweephez
- Egyetlen heapet használ
- CLR, egyes JVM implementációk
- Előnyök
  - Gyors allokáció
  - Nincs fregmentáció
  - Jó lokalitás (időben és helyben)
- Hátrányok
  - Nagy objektumok gyűjtése lassú
  - Minden referenciát módosítani kell

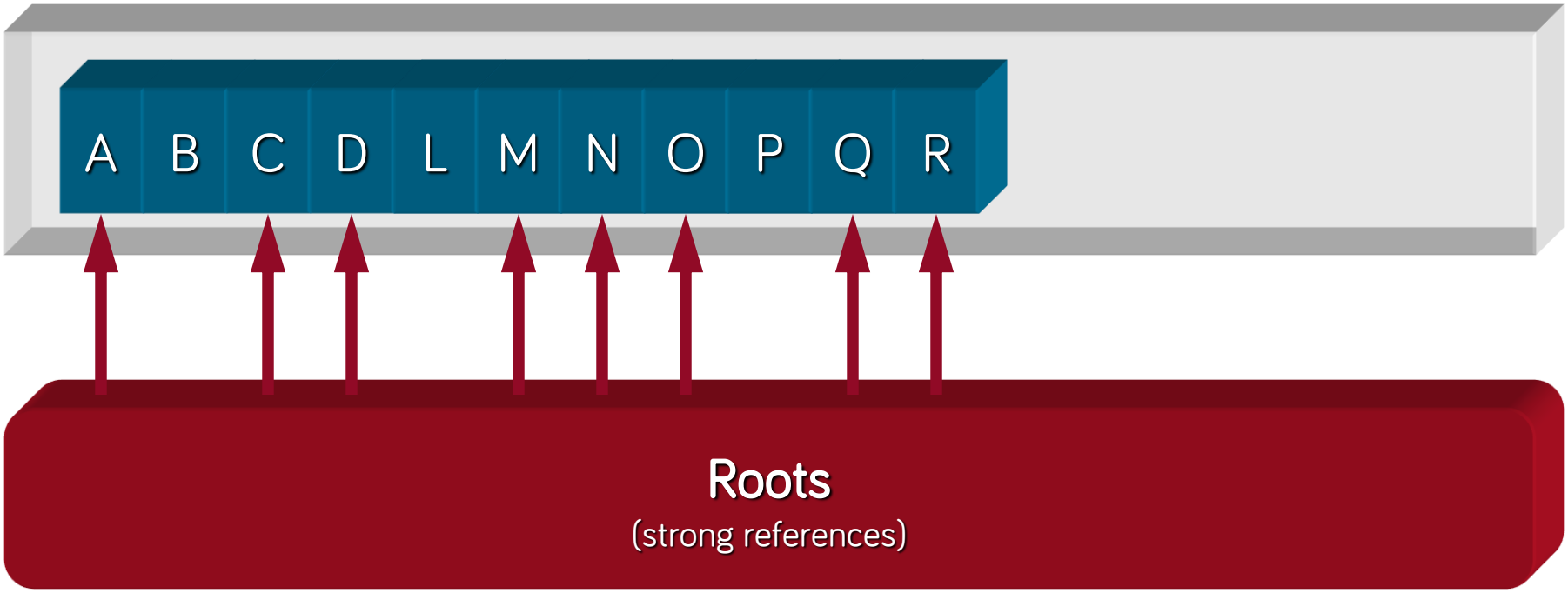
GC a .NET-ben

- Mark and Compact
- Generációkat használ
- Mark and Sweep a nagy objektumokhoz

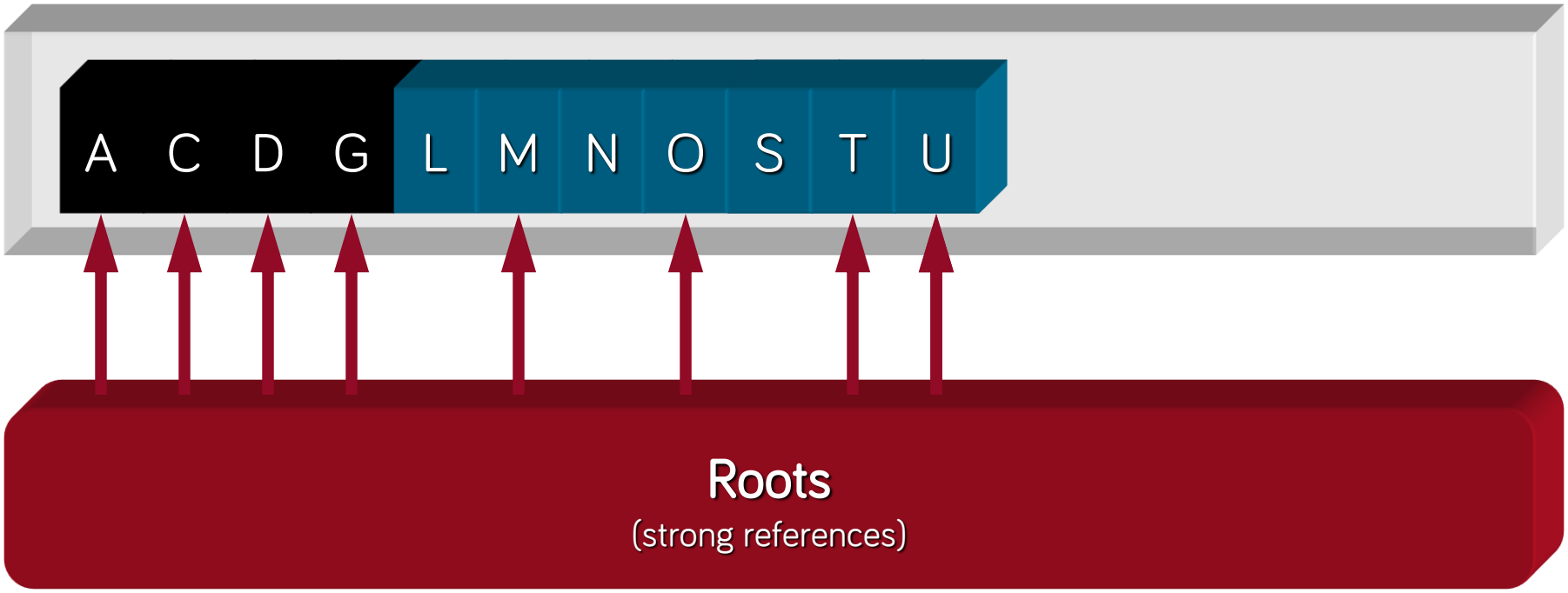
Mark and Sweep GC



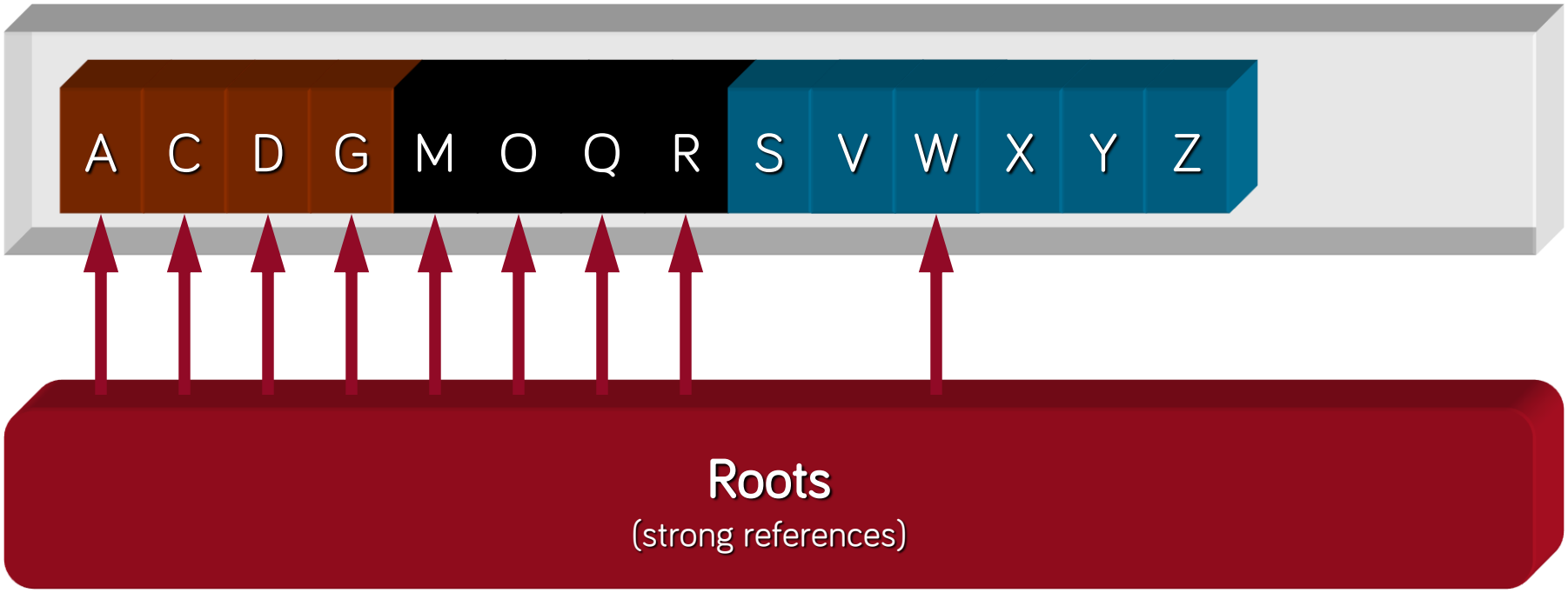
Garbage Collection
Generations



Garbage Collection
Generations



Garbage Collection
Generations



Működés

- Több generáció - weak generation hypothesis:

  Fiatal objektumok hamar halnak.

- Ha már nincs hely a 0-ás generáción, az első generációt is összegyűjti
  - > ha nem segít, akkor a 2-at is
  - > a második generáció után már nincs több
- Melyik generációban van: GC.GetGeneration
- A régebbi generációk általában 10-szer ritkábban gyűjtődnek

Generációk előnyei és problémái

- Nincs szükség hatalmas heapeket gyűjteni
  - A munka feldarabolódik
- A Mark fázis jelentősen lecsökkenhet ha figyelmen kívül hagyjuk a régi generációhoz tartozó objektumok közti hivatkozásokat
- De hivatkozhatnak új objektumokra is…

Megoldás: Write barriers

- Bittömb, ahol minden bit a memória egy tömbjéért felelős (pl 128 byteért)
  - Minden írás, ami ebbe a memóriatartományba történt, beállítja ezt a bitet egyre.
- Legközelebb csak a módosított tartományban lévő objektumokat kell figyelembe venni
  - Sokkal kevesebb objektumot kell megvizsgálni
- Implementáció
  - Minden írás +10-100 CPU ciklus
  - GetWriteWatch alapú

Működés - LOH

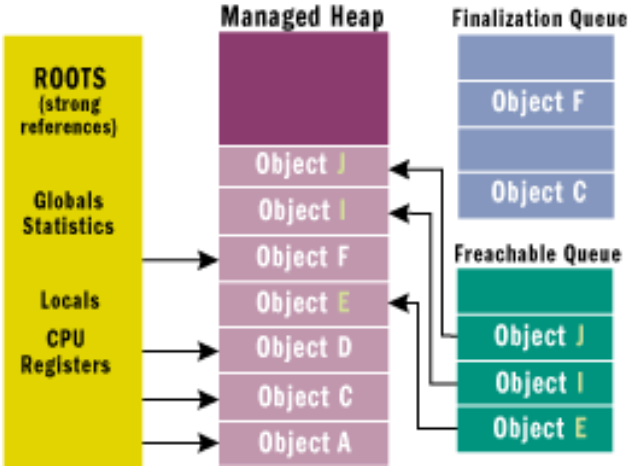
- Large Object Heap
  - A nagy objektumokat ne mozgassuk - lassú
  - Külön heap számukra
  - Kb 85kb-os küszöbérték
  - Az itt lévő objektumokat sose mozgatja – olyan mint C-s

GC: mikor, hol történik

- GC-nél a felügyelt szálakat meg kell állítani:
  - Thread hijacking - használjunk rövid metódusokat
  - Safe points
  - Fully interruptable code

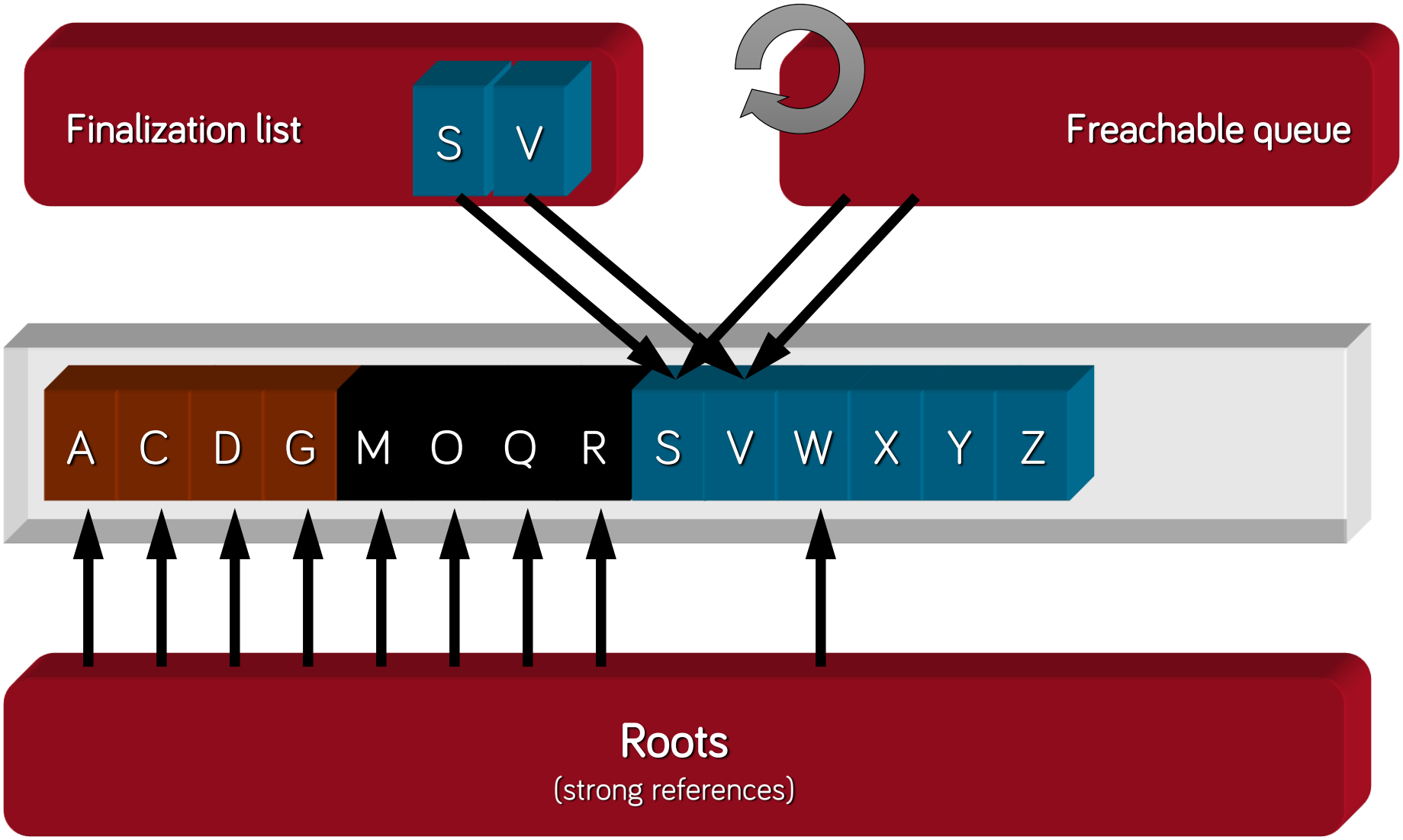
A nem determinisztikus destruktor

- Destruktor helyett Finalize metódus
  - Ha nincs Finalize vagy SuppressFinalize - rögtön felszabadul
  - ha van, akkor még hivatkozott – nem törlődik, csak legközelebb
  - Másik szál – pl. más thread local storage



- A hívás ideje nem determinisztikus:
  - Később, mint amikor a szemétgyűjtés lefut
- A sorrend sem determinisztikus !
  - pl: StreamWriter, FileStream: stream flush, de a fájl már be van zárva – a stream-et be *kell* zárni.
- *Újjászületés*: a Finalize-ben hivatkozást hozunk létre
- A Finalizer-es objektum mindenképp az 1. generációba kerül

Finalization működése



Finalization és a Dispose minta

- Finalization: ~C(): nem determinisztikus erőforrás takarítás
- Költségek: objektumok megőrzése; finalizer szál; könyvelési költség; hívás
- Használjuk ritkán – inkább a Dispose-t
  - IDisposable megvalósítása
  - Hívjunk GC.SuppressFinalize
  - Minél kevesebb mező, hamar nullázni
  - Dispose hamar; try/finally; C# using

Destructorok

- Mark-and-sweep szemétgyűjtés – nemdeterminisztikus destruktorok
  - a végrehajtás ideje nem ismert
  - a sorrend nem ismert
  - a szál nem ismert (TLS)
    - ThreadStaticAttribute
- 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();
      }
   }
}
```

Dispose tervezési minta

```
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);
    }
}
```

Gyenge referenciák

- Úgy hivatkozunk az objektumra, hogy a GC felszabadíthatja
  - Memória hiány esetén a GC tud takarítani
- Használat előtt egy erős referenciát szerzünk az objektumra
- Használati esetek
  - Klasszikus cache
  - Az alkalmazás által használt objektumokon időzített feladat elvégzése

Gyenge referencia példa

```
void SomeMethod()
{
    // Create a strong reference to a new Object.
    object o = new object();

    // Create a strong reference to a short WeakReference object.
    // The WeakReference object tracks the Object's lifetime.
    WeakReference wr = new WeakReference(o);

    o = null; // Remove the strong reference to the object.

    // if GC happens here, the object 'o' is collected
    // try to get a strong reference to the object
    o = wr.Target;
    if (o == null)
    {
        // A GC occurred, object's memory was reclaimed.
    }
    else
    {
        // A garbage collection did not occur and I can successfully
        // access the Object using o.
        // Object will not be collected as long as
        // I have this string reference of o.
```

Unsafe Code

- Teljes kontrol
  - Extrém optimalizáció
  - Meglévő bináris struktúrák kezelése
  - Együttműködés régi kóddal
- Unsafe szerelvény, kód
  - Pointer típusok, pointer aritmetika
  - ->, *, [], & operátorok
  - Unsafe kasztolás
  - GC elkerülése

Using the unsafe keyword

- Típusokon, mezőkön

```
public unsafe struct Node
{   public Node* Parent;   }

public struct Node
{   public unsafe Node* Parent;   }
```

- Utasítás blokkhoz

```
unsafe {   MyStruct* pi = stackalloc MyStruct[10]; }
```

C# pointerek

- Nem kezeli a GC
- Mire mutathat
  - blittable típusra
  - unmanaged típusra
  - Referencia típusra
- Amik…
  - Beépített integer típusok (int, byte, enum, …)
  - Érték típusok nem felügyelt típushivatkozásokkal
    - Beágyazott fix hosszú integer típusú tömbbel
  - Másik mutató
  - Nem használhat generikus típus argumentumot!

Blittable típus

```
struct BlittableStruct
{
    int a;
    unsafe int* pa;
    MyStruct mys;
    unsafe fixed byte data[10];
};
```

- FYI: beágyazott tömbök nem az ECMA része ☺

Mutat konverziók

- Implicit
  - Null bármilyen mutatóra
  - Bármilyen pointer void *-ra
- Explicit
  - Bármilyen pointerről bármilyen pointerre

    int * = (int *)pChar;
  - Integerről bármilyen pointerre

    int * = (int *)123;
  - Bármilyen pointerről integerre

    int = (int)pChar;

Pointer aritmetika 1

- Nem felügyelt típus mérete: sizeof

```
int size = sizeof( MyStruct );
```

- Cím számítás és érték kinyerés

```
int *pa = &a;       // indirection
int b = *pa;        // dereferencing
```

- Tag elérés mutató alapján

```
int a = pMyStruct -> A; // pont ugyanaz mint:
int a = (*pMyStruct) . A;
```

Pointer aritmetika 2

- Növelés, csökkentés

  int a = *pInteger++; // incremented by sizeof(int)

- Hozzáadás integer típusokkal

  pInteger += 4; // 4 * sizeof(int)

- Pointerek kivonása

  int diff = pInt1 – pInt2;

- Aritmetikai műveletek (<, ==, >, !=, <=, >=)

  bool b = pInt1 == pInt2;

Rögzítés (fix/pin)

- Pointer egy felügyelt referencia típus tagjára

  int *pa = & myInst . FieldA;

- Probléma: a GC mozgathatja az objektumot
- Megoldás: fixed statement

  fixed(int *pa = & myInst . FieldA) { … }

  fixed(int *pa = intArray) { … }

  fixed(int *pa = &intArray[12]) { … }

  fixed(char *ps = “apple”) { … }

- **A GC-nek ez nagyon rossz!**

Tömb foglalás a vermen

- Nem felügyelt típusú tömb foglalható a vermen

  int *pi = stackalloc int[10];

- Nincs index ellenőrzés
  - > Gyors de veszélyes!
- A lefoglalt memória nincs inicializálva!

```
static void overflow(string dummy)
{
    unsafe
    {
        uint* pi = stackalloc uint[10];
        for (uint i = 0; i < 13; i++)
            *(pi++) = 0xfefefefe;
    }
}
```

Unsafe kód

- Unsafe azt jelenti, hogy nem ellenőrizhető!
- Explicit ki kell írni C#-ban
- Explicit meg kell jelölni a szerelvényt, fordítási opció
- **Nem használhatod „véletlenül” !**

Layout

- A mezők a deklarálás sorrendjében vannak a memóriában
- Alapértelmezett igazítás: 32/64 bit
  - StructLayout attribútummal megváltoztatható
- 32 bit: data >= 4 byte akkor 4-byte-ra igazított
- 64 bit: data >= 8 byte akkor 8-byte-ra igazított
- Lehet nem igazított
  - HW vagy OS kezeli
  - Komoly teljesítmény probléma!

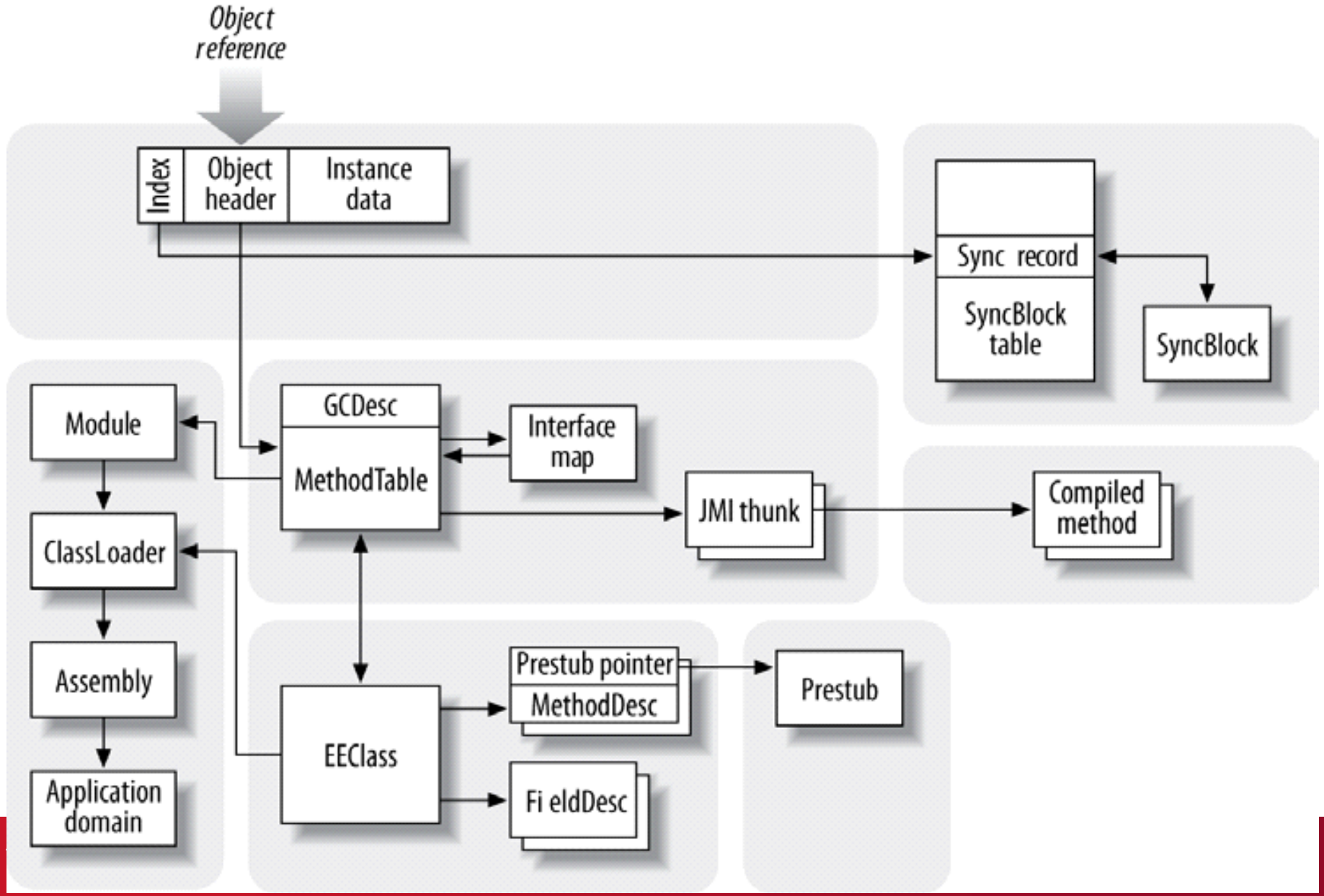
Layout opciók

- StructLayout attribútum
  - Class vagy struct esetén
  - Auto: hatékony
    - C#: alapértelmezett classokhoz
  - Sequential: a deklaráció sorrendjét követi
    - Pack paramétert használ, lehet nem folytonos!
    - C#: alapértelmezett struktúrákhoz
  - Explicit:
    - Felhasználó által definiált méret, mező eltolás, **a mezők átfedhetnek**
    - Felügyelt referenciák nem fedhetnek át!

A referenciák mutatók az objektumokra

- Az objektumok mozognak!
- Objektum fizikai memória layoutja:
  - Sync block index (-4/8 byte @ 32/64 bit)
    - 4 byte 32 biten, 8 byte 64 biten
  - Mutató a metódus táblára (…method desc)
    - 4 byte 32 biten, 8 byte 64 biten
    - Size, type info (EEClass)
    - Metódus pointerek (precode)
  - Példány adat (+4/8 byte @ 32/64 bit)

Memória layout



Migráció 32 bitről 64 bitre

- IntPtr platformfüggő (IntPtr . Size)
- Pointer aritmetika rendben

Kérdések?

Albert István
ialbert@aut.bme.hu