



Vývoj aplikací v prostředí .NET

© Katedra řídicí techniky,
ČVUT-FEL Praha

Přednáška – 11. týden



[Royo]

Fragmentace paměti

*Jedno úmrtí
znamená tragédii,
miliony mrtvých
statistiku.*

[Josef Stalin]

■ Důvody fragmentace

- a) proměnlivé chování programu
- b) program žádá velké bloky dat, ale uvolňuje pouze malé
- c) jednotlivá úmrtí
 - ❑ ruší se tím data, která spolu nesousedí, a tak nelze scelit jejich paměťové oblasti a učinit je použitelné pro velké objekty.



- Create a list of text, e.g a dictionary

```
public class Record
```

```
{ private Record linkBack; private int hashCode; private char [] data;  
  public Record(string text, Record previos)  
  { linkBack = previos; data = new char[text.Length];  
    text.CopyTo(0,data,0,text.Length); hashCode = data.GetHashCode();  
  }
```

```
/* Other methods of Record...*/
```

```
}
```

```
/* Example of building list from a method*/
```

```
static void CreateList()
```

```
{ Record last = null;  
  System.Text.StringBuilder sb = new System.Text.StringBuilder();  
  /* a long text */ for (int j = 1; j < 2040; j++) sb.Append((char)(j + 32));  
  for (int i = 1; i < 100000; i++) last = new Record(sb.ToString(), last);
```

```
}
```

Example

```
static void CreateList()
```

```
{    Record last = null;
```

```
    System.Text last {ConsoleApplication.Program.Record} .Text.StringBuilder
```

```
    /* a long te
```

```
    for (int j =
```

```
    for (int i = 1
```

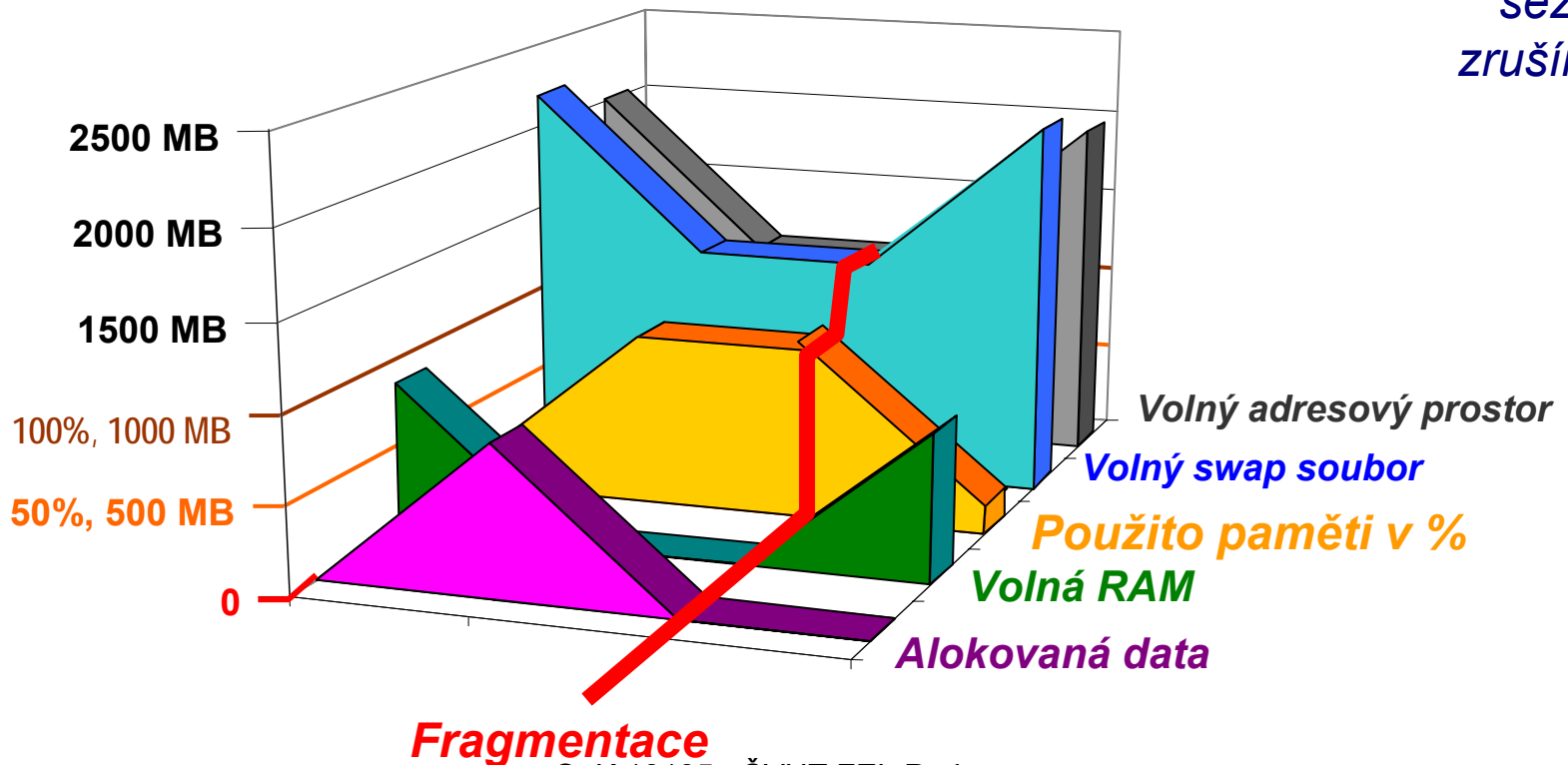
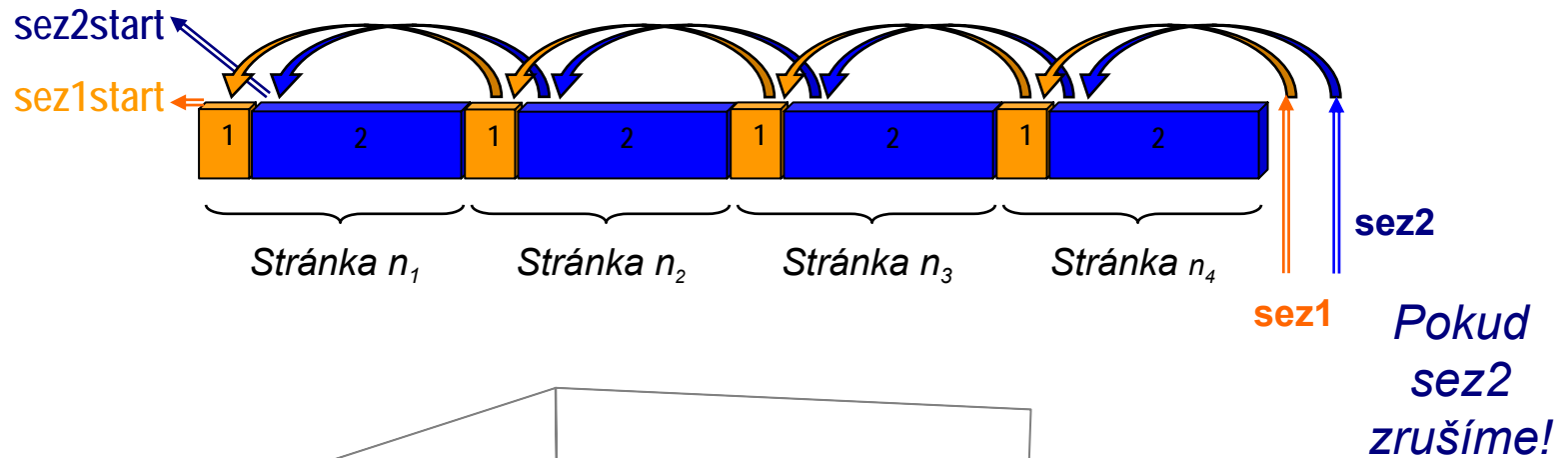
```
        last = new
```

```
    }
```

```
}
```

+	data	{Dimensions:[2039]}
-	hashCode	3950370
-	linkBack	{ConsoleApplication.Program.Record} (j + 32));
+	data	{Dimensions:[2039]}
-	hashCode	27471992
-	linkBack	{ConsoleApplication.Program.Record}
+	data	{Dimensions:[2039]}
-	hashCode	58082536
-	linkBack	{ConsoleApplication.Program.Record}
+	data	{Dimensions:[2039]}
-	hashCode	65555084
-	linkBack	{ConsoleApplication.Program.Record}
+	data	{Dimensions:[2039]}
-	hashCode	65707585
-	linkBack	{ConsoleApplication.Program.Record}
+	data	{Dimensions:[2039]}
-	hashCode	52015011
+	linkBack	{ConsoleApplication.Program.Record}

Příklad fragmentace v C++



Příklad obsahuje hned dvě podstatné chyby:

1. Vytváří fragmentaci

- Tu v C# sice odstraní garbage collector (GC), ale program zvyšuje jeho zátěž...

2. Odkazy jsou rozprostřené po paměti

- Musí se neustále načítat stránky z disku do RAM při prohledávání obřího seznamu.
- *Jak lze situaci zlepšit?*
 - odkazy lze umístit do souvislých bloků (polí) a využít unmanaged C++ DLL bez GC (*důvod dále...*)
 - či si naprogramovat vlastní odkládání dat na disk,
 - **nebo ještě lépe** – přes ADO.NET využít databázi, ta je na seznamy prvotřídní specialistka 😊
(*některá z příštích přednášek*)

```

struct HASH
{ int hashCode; int dataSize; WCHAR * dataField;
public : HASH(int hash, int size, WCHAR * data)
    { hashCode=hash; dataSize=size; dataField=new WCHAR[size];
      memcpy(dataField,data,size); }
HASH() { hashCode=0; dataSize=0;dataField=NULL; }
};

static HASH * hashArray=NULL; static int hashCount=0;
#define MAX_HASH 1000000
.RECORD_API int AddRecord(int hash, int size, WCHAR * data)
{ if(hashArray==NULL) hashArray = new HASH[MAX_HASH];
  if(hashCount<MAX_HASH)
    { hashArray[hashCount++]=HASH(hash,size,data); return TRUE; }
  return FALSE;
}

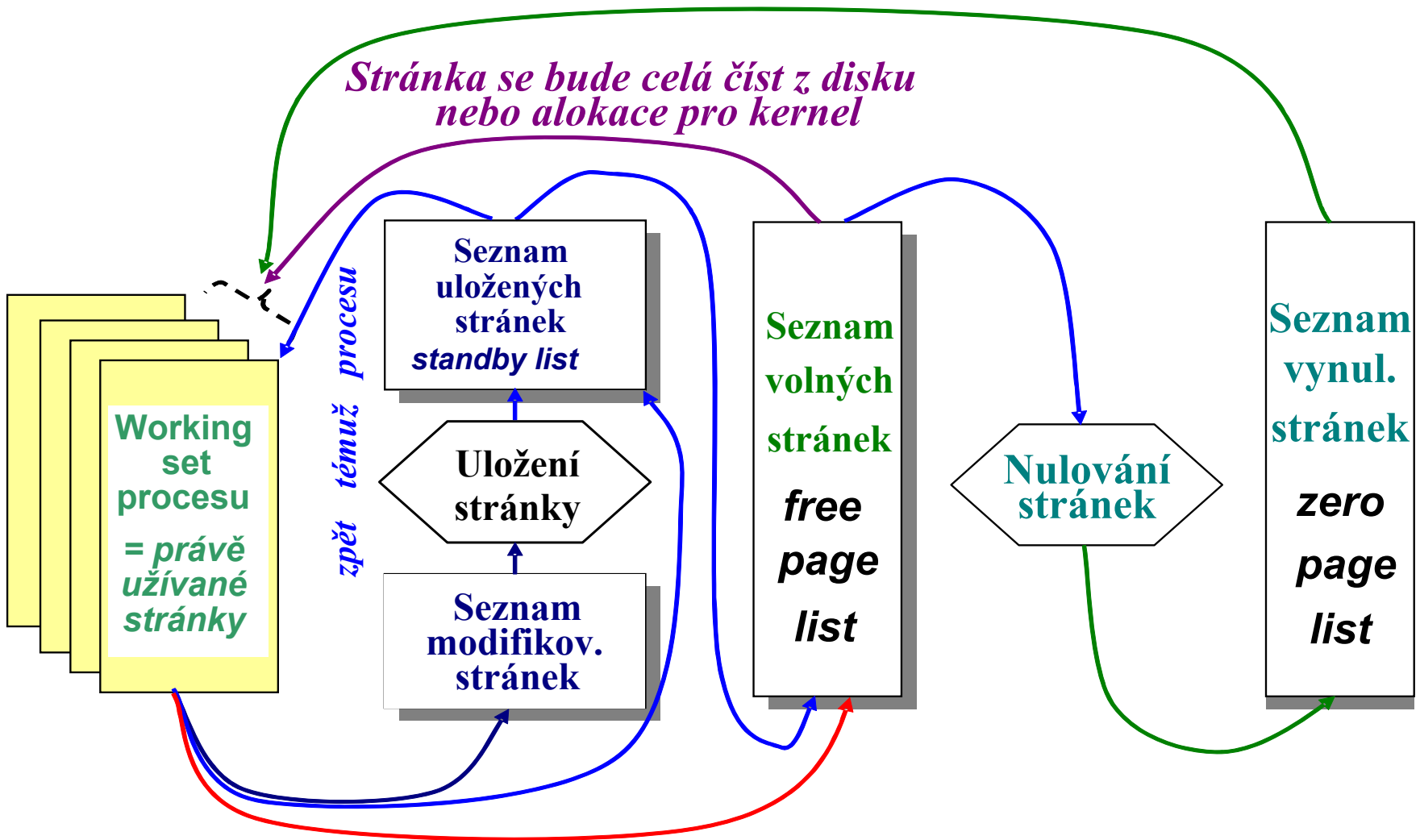
[DllImport("Record.dll")]
static extern int AddRecord(int hash, int size, char[] data);

```


Práce se stránkami

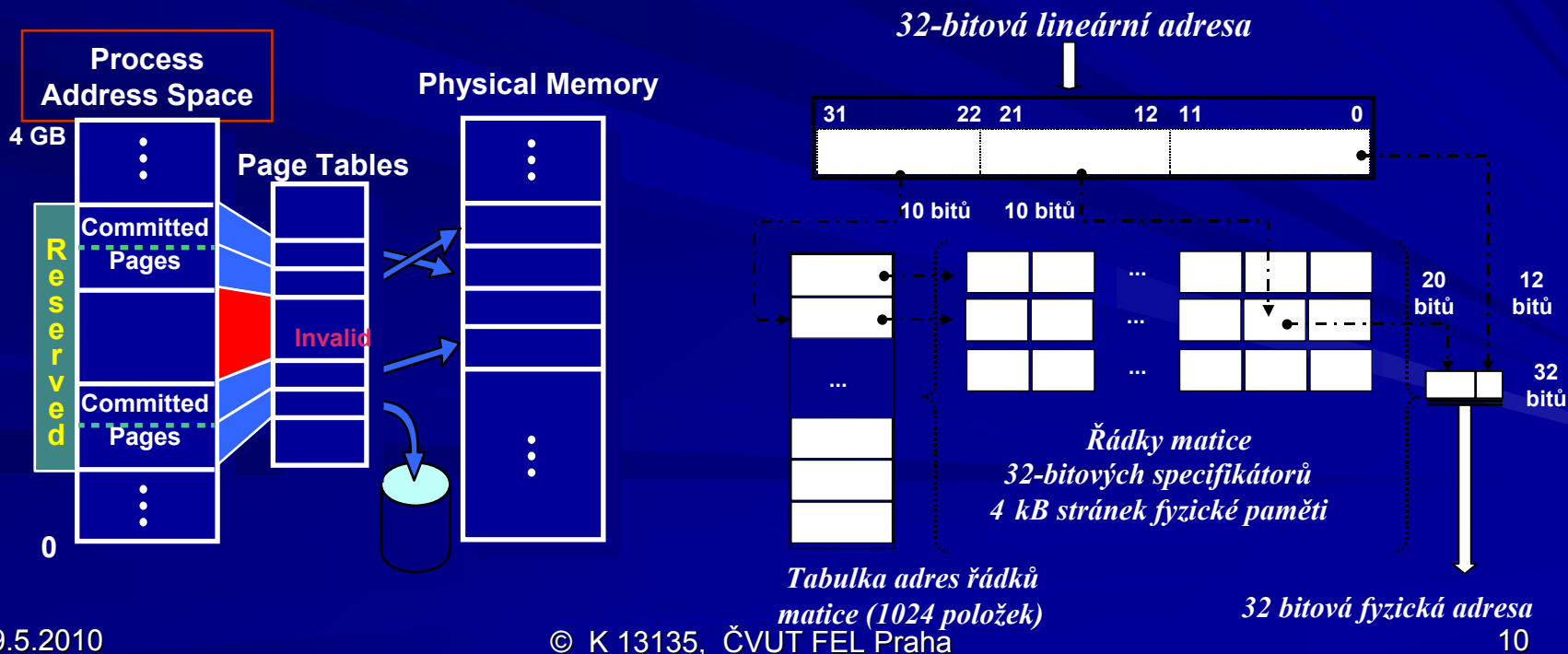
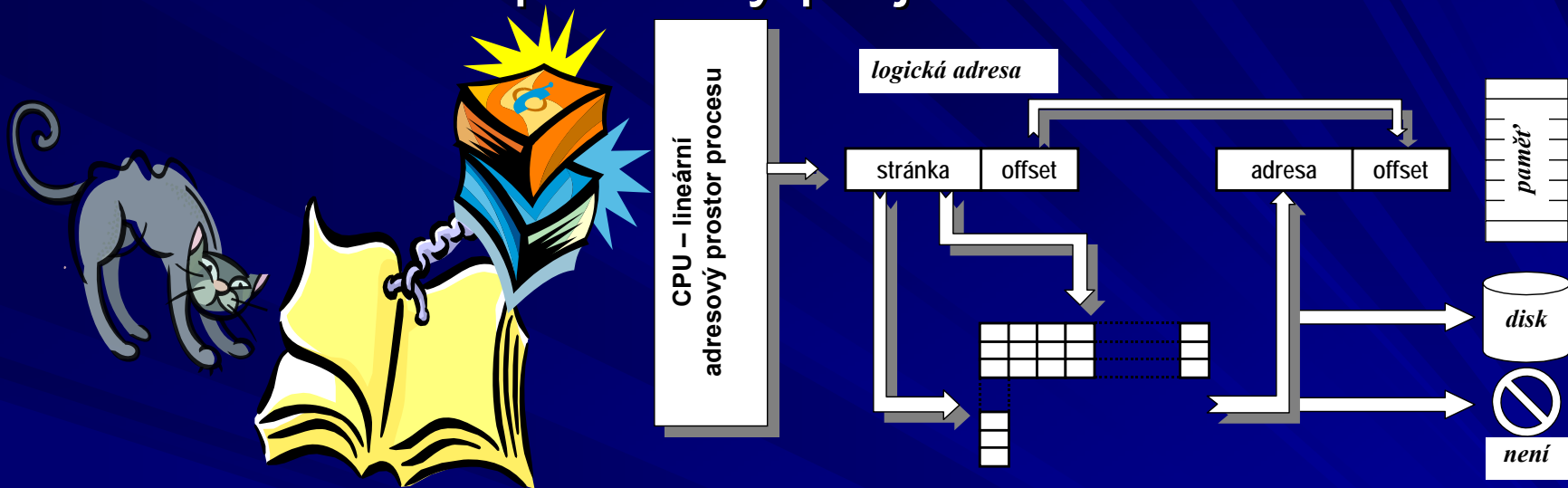
Stránka přidělená některému procesu

*Stránka se bude celá číst z disku
nebo alokace pro kernel*



Konec procesu

Jak se prakticky projeví stránkování Windows?



Task Manager (W2K): Process Information

Task Manager (W2K): Process Information

VM= virtual memory

<u>Image Name</u>	<u>Mem Usage</u>	<u>VM Size</u> (nutno přidat do výpisu)
jméno exe, <i>ale bez cesty</i>	working set, <i>ale včetně sdílených stránek</i>	privátní (nesdílená) paměť procesu

Task Manager (W2K): Memory Management Information

Physical Memory

Total memory: velikost RAM

Available memory: standby + free + zero

System cache: standby + working sets

Commit

Charge Total: VM size + kernel

Limit: fyzická paměť
+ okamžitý swap
file

Start -> Administrative Tools -> Performance

Start -> Run: **perfmon.exe**

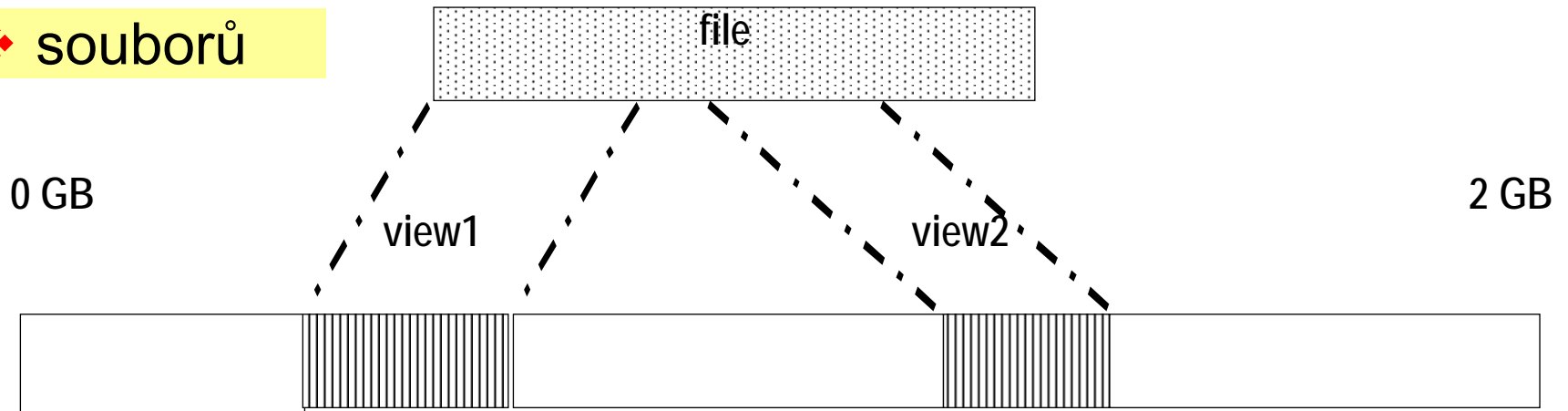
- **Memory, Pages/Sec** - jedno z nejvíc matoucích měřítek - zahrnuje všechny aktivity, včetně souborů, které se stávají součástí swap souboru.

- **Memory: Pages Input/sec**- kolik stránek se četlo ze SWAP do volné RAM

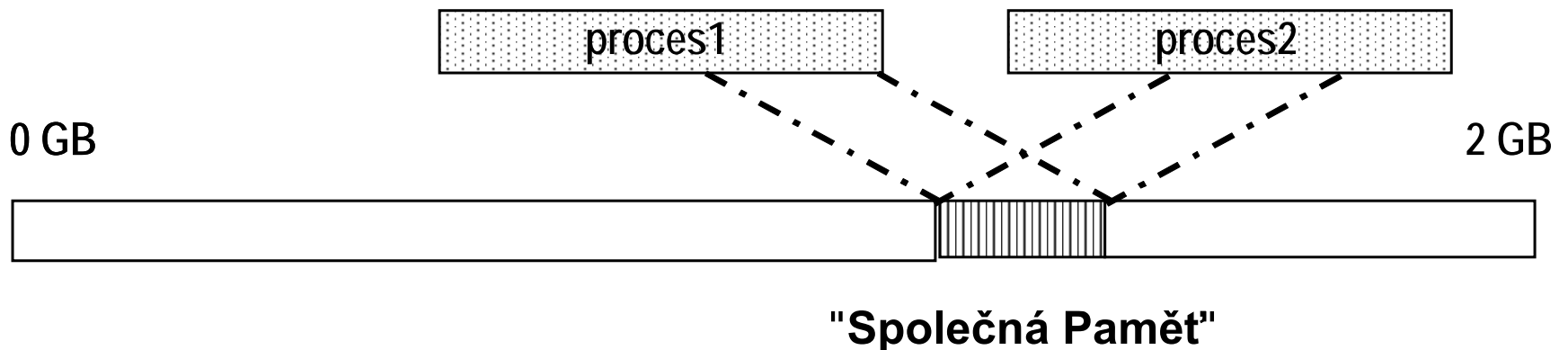
- **Memory, Pages Output/Sec** – kolik stránek se uložilo do SWAP

Mapování pomocí stránkování

❖ souborů



❖ sdílená paměť



Mapování souborů

```
m_hFile= CreateFile (filename),           // open MYFILE.TXT
        GENERIC_READ,                     // open for reading
        FILE_SHARE_READ,                 // share for reading
        NULL,                             // no security
        OPEN_EXISTING,                   // existing file only
        FILE_ATTRIBUTE_NORMAL,           // normal file
        NULL);                           // no attr. template
```

```
m_dwSize = GetFileSize(m_hFile, NULL);
```

```
m_hFileMappingObject = CreateFileMapping ( m_hFile
        NULL,                             // Default security.
        PAGE_READONLY,                     // Read/write permission.
        0, 0,                             // High-Low-order Max. size of hFile.
        NULL );                           // No name of file-mapping object
```

```
m_lpMapAddress ::MapViewOfFile
        m_hFileMappingObject             // into address space
        FILE_MAP_READ,                   // access mode
        0, 0,                             // high-low-order 32 bits of file offset
        0 );                             // number of bytes to map
```

při **m_hfile** =
0xFFFFFFFF
se paměť sdílí

■ Zdrojový kód

<http://www.pinvoke.net/default.aspx/kernel32/MapViewOfFile.html> !

Možnosti zlepšení

- Dobrá manuální správa
- Automatická správa
 - reference
 - garbage collector



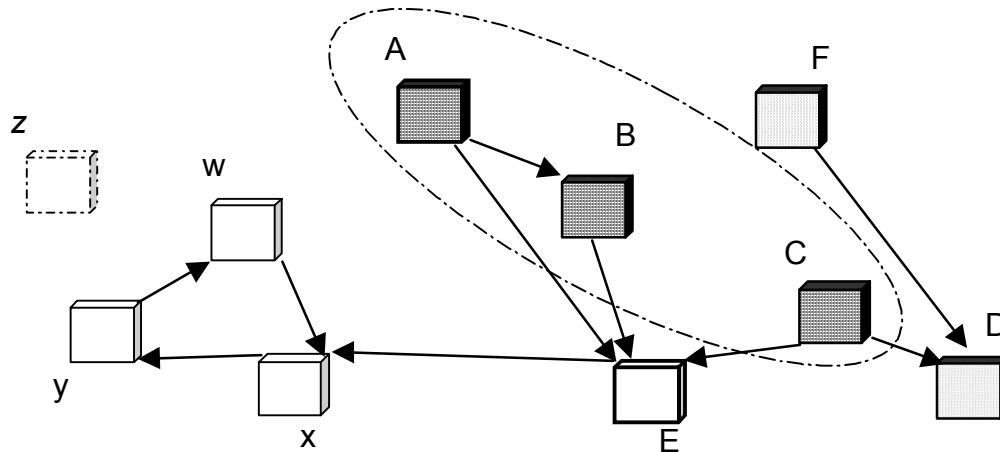
Čínský sběrač smetí

Manuální správa dynamických alokací

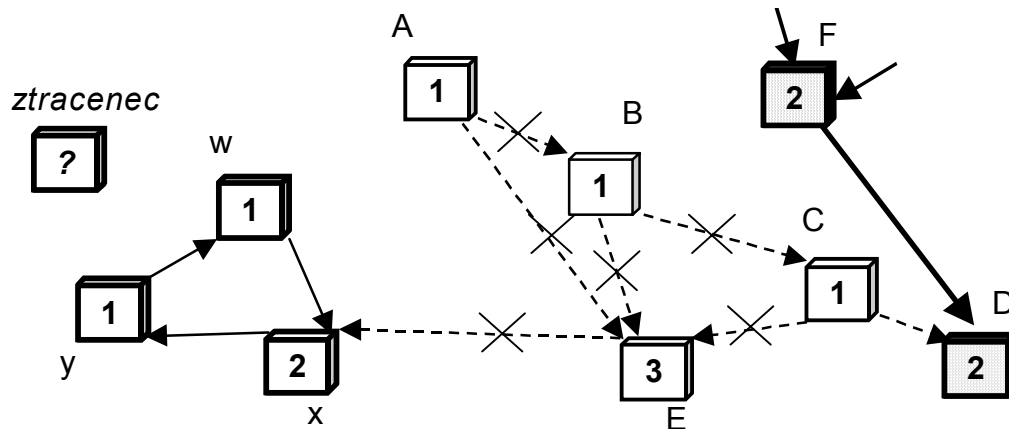
- *vše ponecháno na programátorovi*
 - + lze tak "někdy" vytvořit efektivní správu paměti,
 - - tvorba programu je pracná a s velkým rizikem chyb,
 - - neřeší se riziko fragmentace.



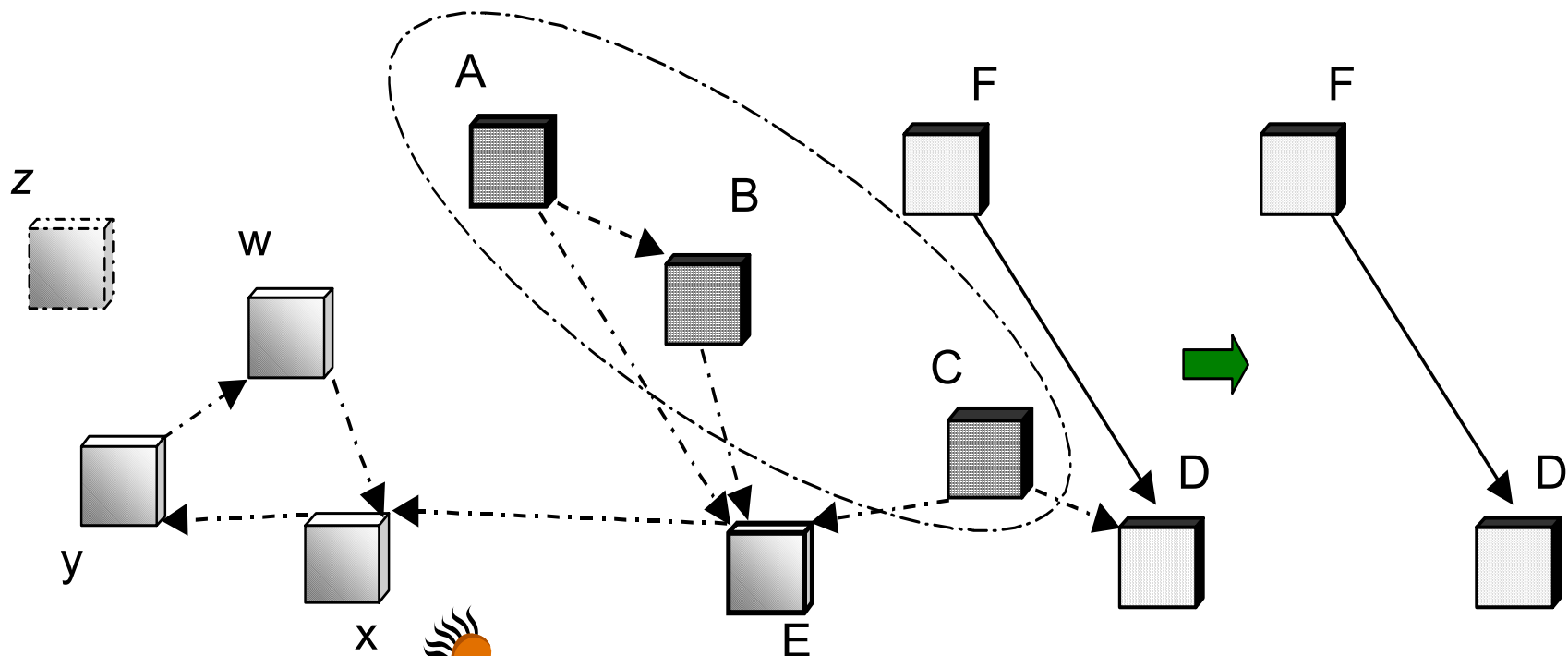
Počítání referencí



- + snadná implementace
- + průběžné uvolňování
- při práci s objekty se navíc provádí manipulace s referencemi;
- neuvolní se zapomenuté objekty (memory leak)
- neruší se alokační cykly
- problémy s transakcemi při paralelních operacích (*dataraces*)



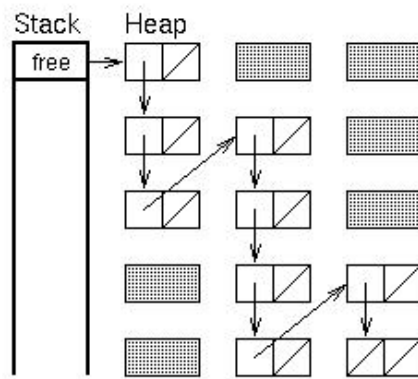
Značkování objektů => úplné uvolnění paměti



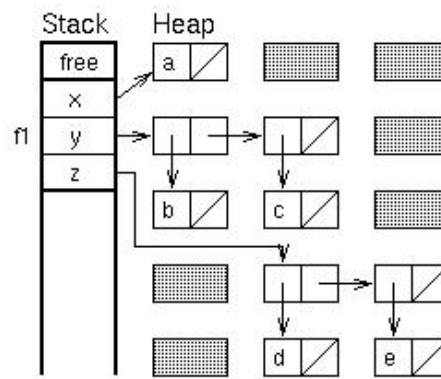
+ uvolní vše

- program je zastavený během běhu GC;
- objekty musí být trvale v RAM paměti, nelze je odložit na disk

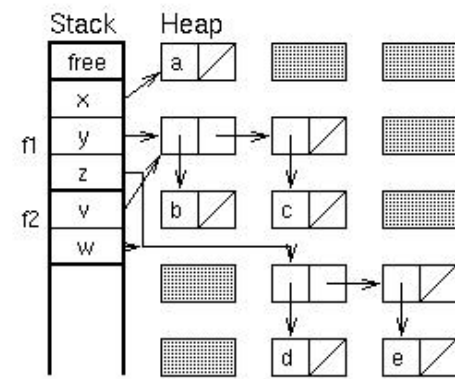
LISP – jeden z prvních jazyků s GC



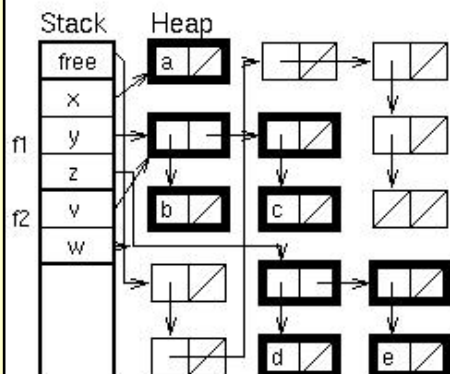
(a) Initial state



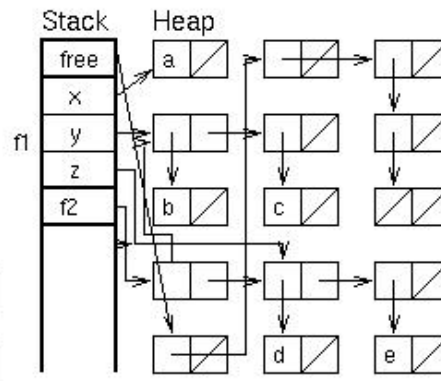
(b) Invoke f1 with a, (b,c), (d,e)



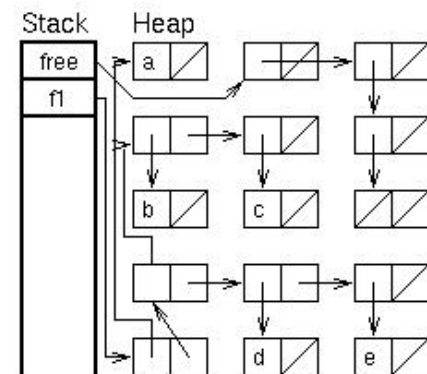
(c) Invoke f2 with arguments (b c)(d e)



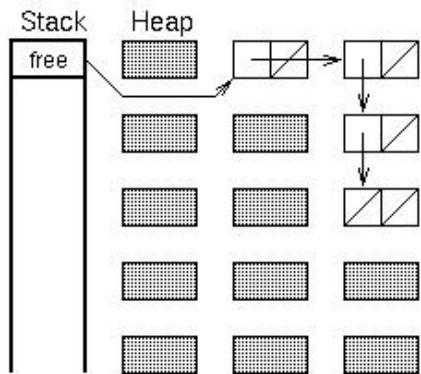
(d) Garbage collection - mark 9 nodes






(e) Compute f2 = ((b c) d e)



(f) Compute f1 = (a (b c) d e)



(g) f1 nodes freed

-  Active (allocated) nodes
-  Inactive (garbage) nodes
-  Null pointer (end of list)

Trasovací algoritmy pro GC

- + ruší zapomenuté objekty a cykly
- + optimálnější než počítání referencí při běhu programu
- + lze řešit i zcelování paměti.
- všechny objekty musí být viditelné pro GC;
- při trasování se musí pozastavit běh programu

Pozn. Existují sice GC algoritmy, které pracují souběžně s programem, např. “Tricolor GC”, ale ty jsou implementačně náročné a pomalejší.

.NET GC - garbage collector

Krédo .NET GC:

Alokuj objekt pomocí **new** a zapomeň na to...

**Předpoklady, na nichž je GC algoritmus
vybudovaný**

- čím novější objekt, tím kratší je jeho životnost;
- čím starší je objekt, tím delší je jeho životnost;
- GC části haldy (heap) je rychlejší než celé haldy

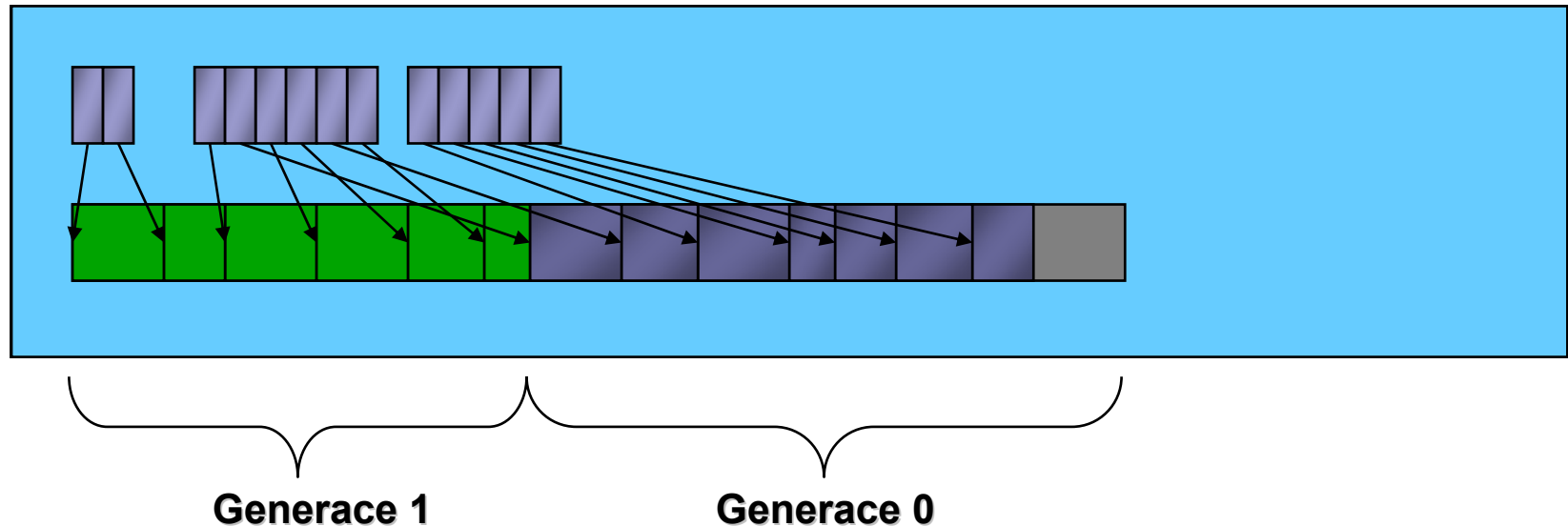


GC používá třífázový běh

- **Mark** - značkování → označí se objekty určené k zrušení
- **Sweep** - zametání → odstraní objekty z paměti
- **Compact** - zacelení → spojí paměť v jednolitý celek



Generace v .NET



Všechny objekty v haldě jsou generace 0

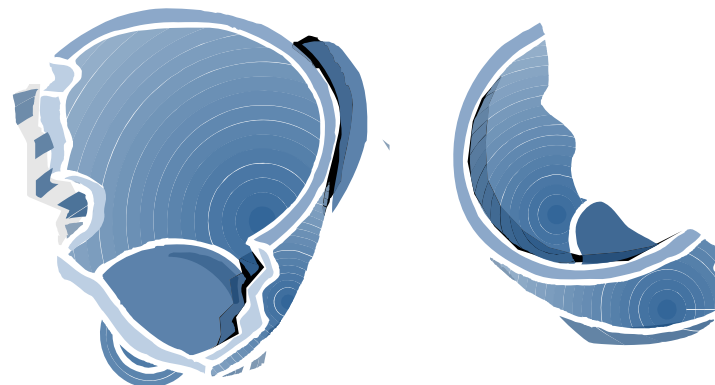
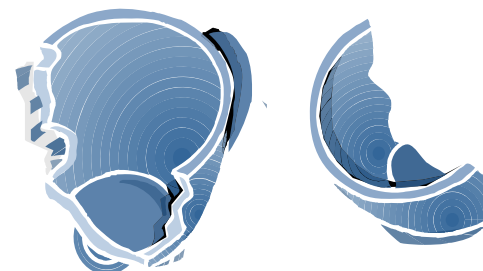
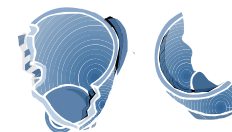
Dostupné odkazy udržují objekty naživu

Pokud 0. generace dosáhne předem určené velikosti $> G_0$, provede se GC a živé objekty se povýší do generace 1

Nové objekty opět vznikají v nulté generaci.

Velikosti generací pro spuštění GC

- 0. generace ~256 KB
- 1. generace ~2 MB
- 2. generace ~10 MB



GC odstraňuje objekt

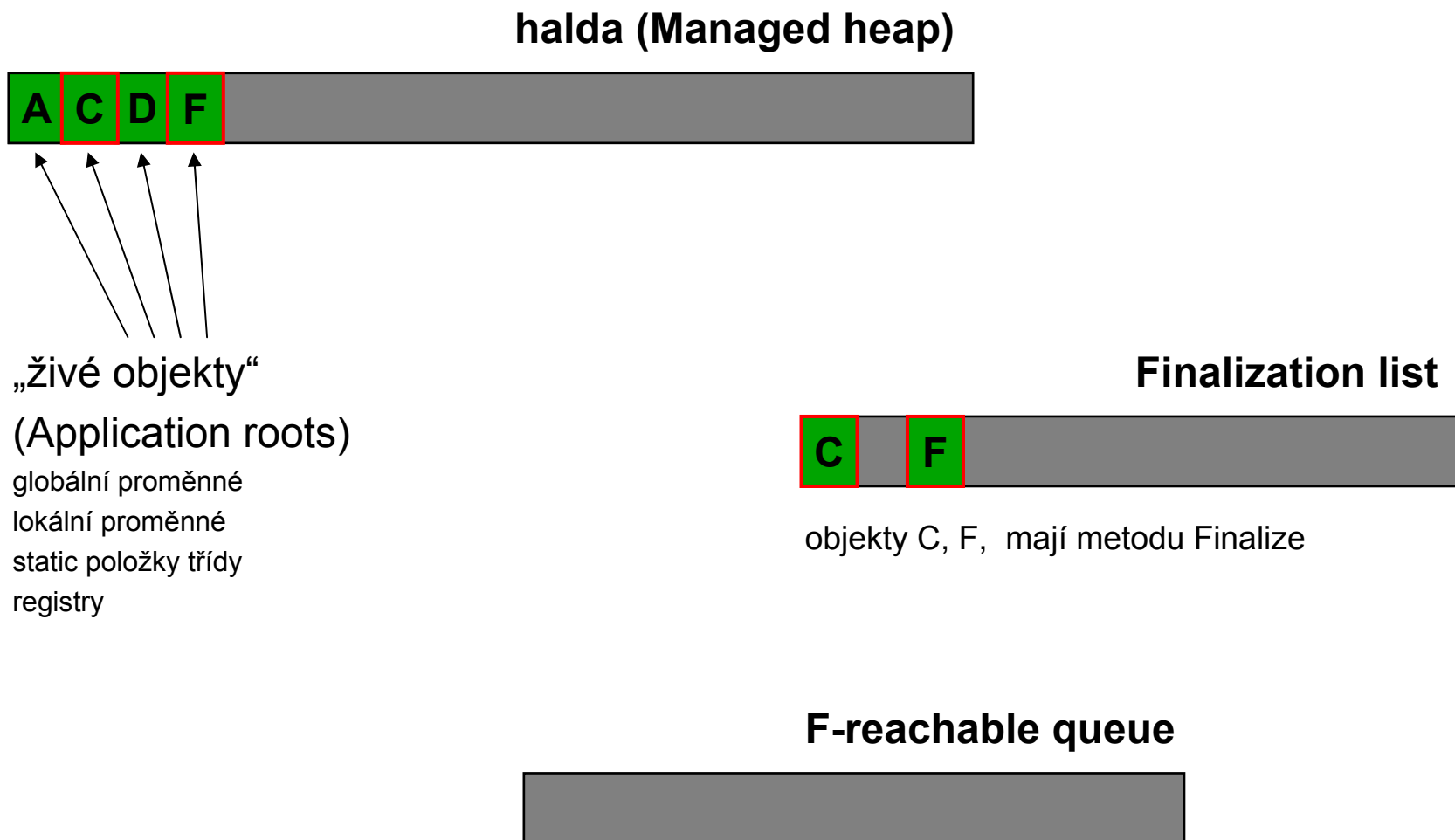
1. objekt nemá Finalize

- ☐ pak se okamžitě odstraní z paměti

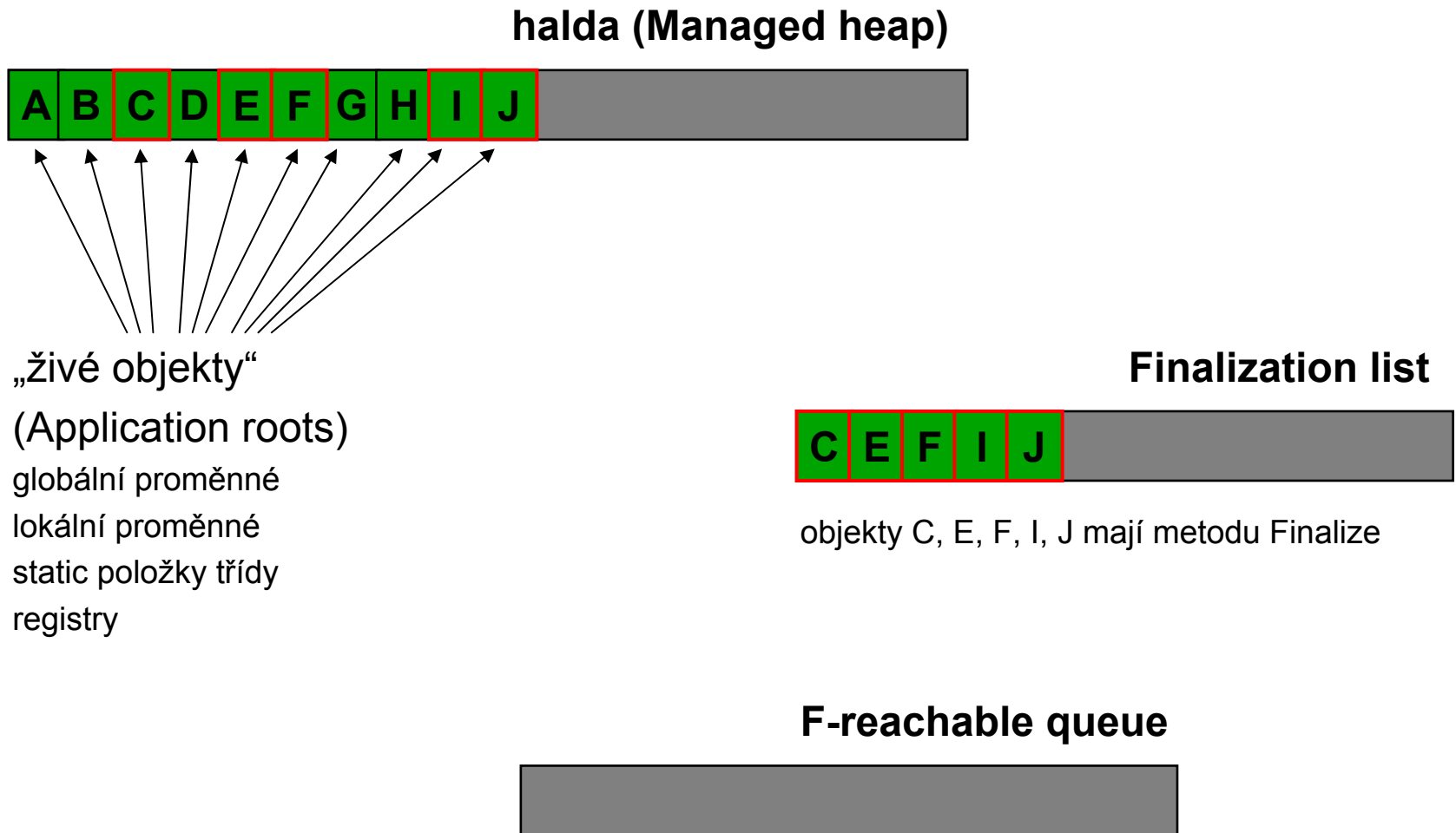
2. objekt má metodu Finalize (C# destruktorka)

- ☐ objekty s Finalize se evidují ve zvláštním seznamu - **Finalization list**
 - objekt se přesune do fronty **F-reachable queue**
 - objekt zatím stále žije....
 - thread na pozadí prochází frontu F-reachable queue a volá metody Finalize čekajících objektů
 - teprve po zavolání metody se objekt bude moci odebrat „do věčných lovišť“

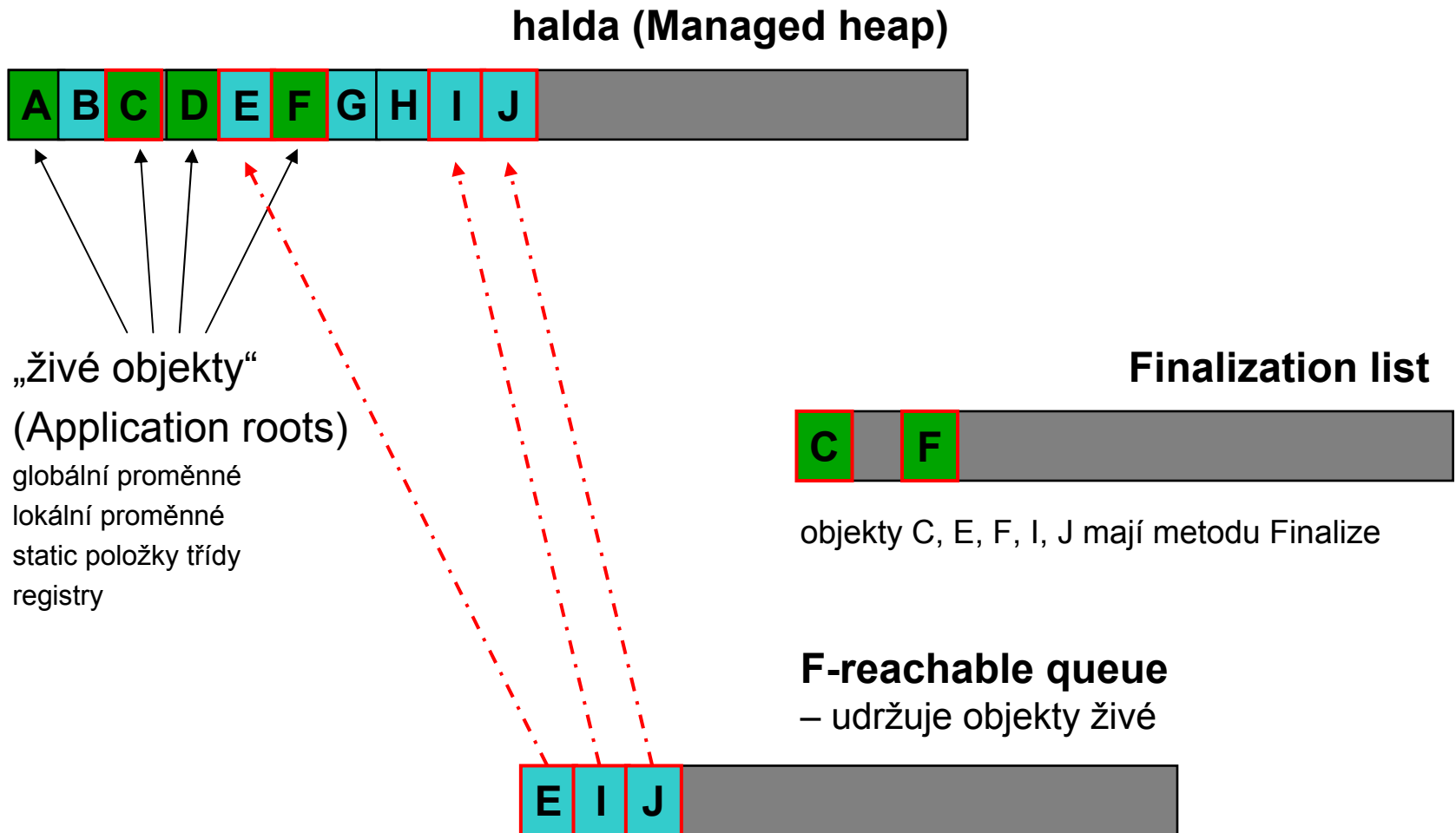
Garbage collector a metoda Finalize 1/12



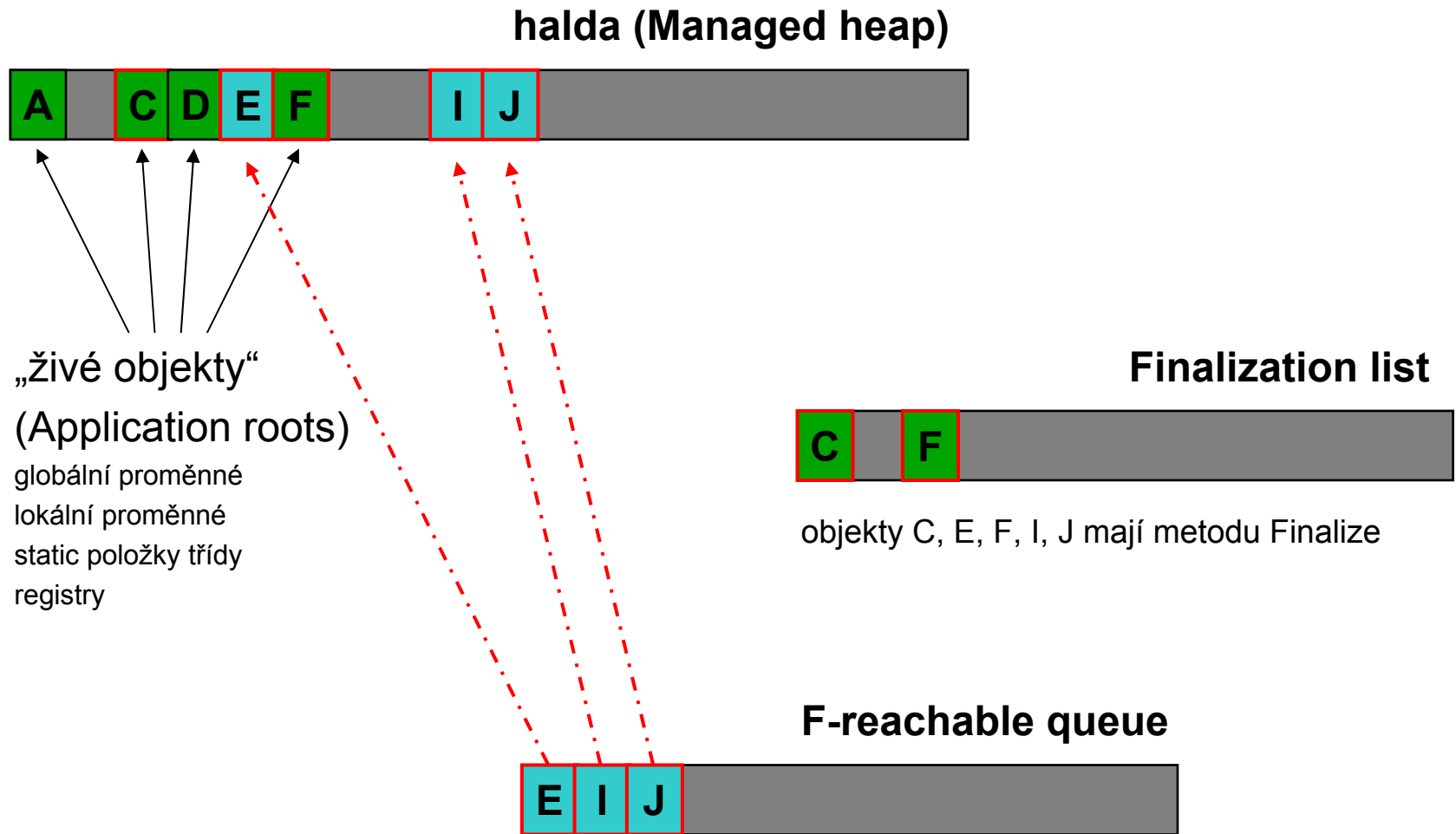
Garbage collector a metoda Finalize 2/12



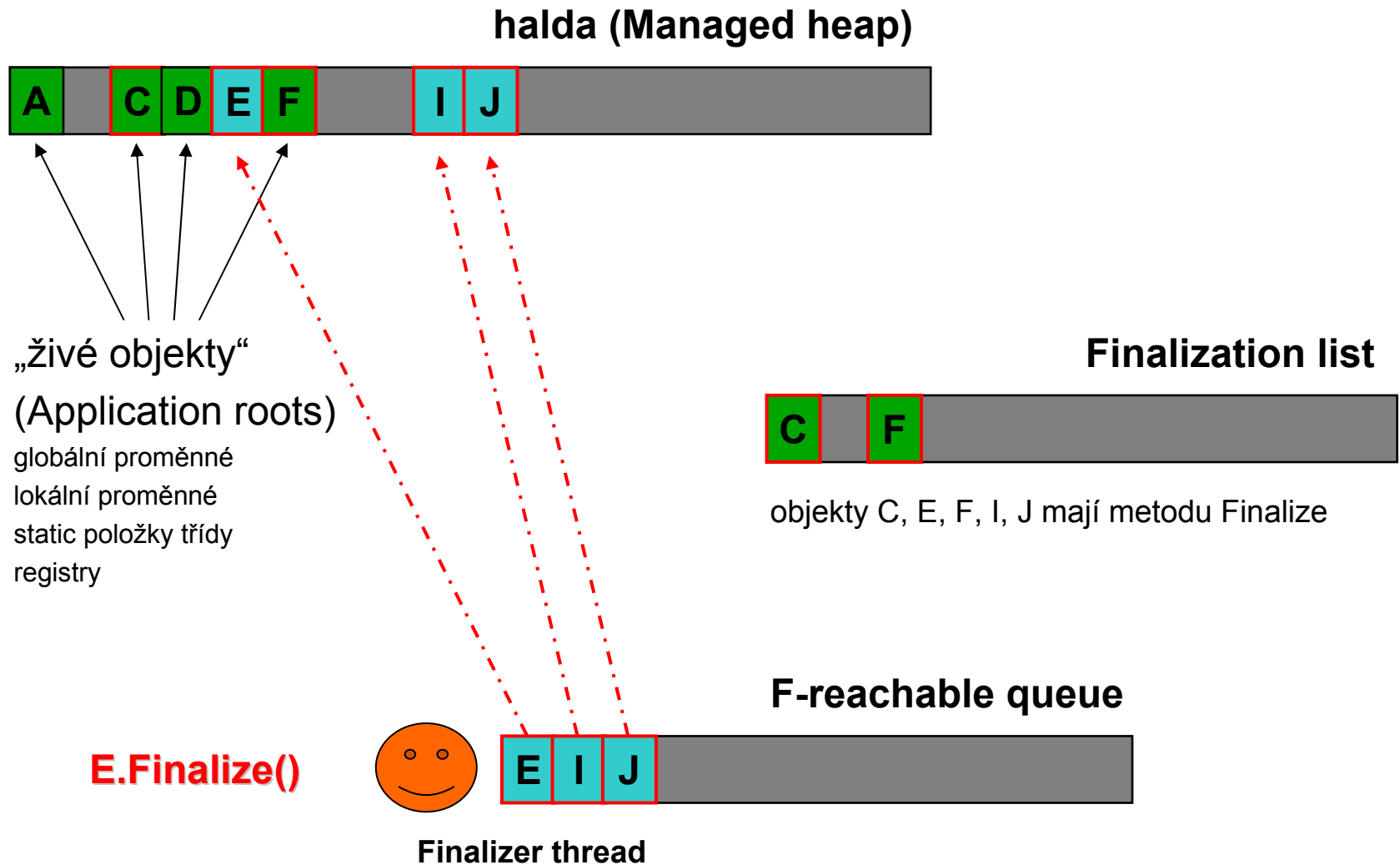
Garbage collector a metoda Finalize 3/12



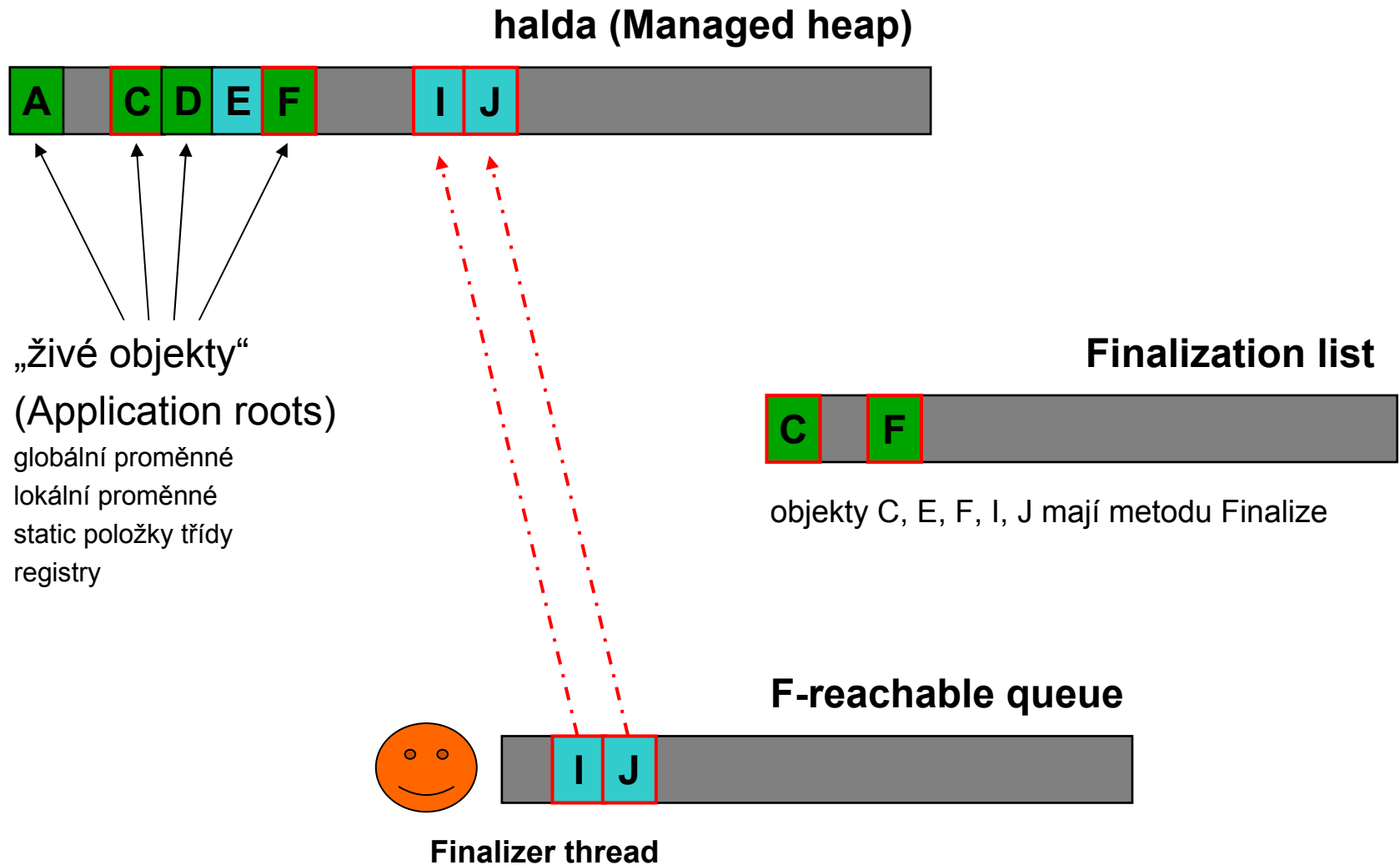
Garbage collector a metoda Finalize 4/12



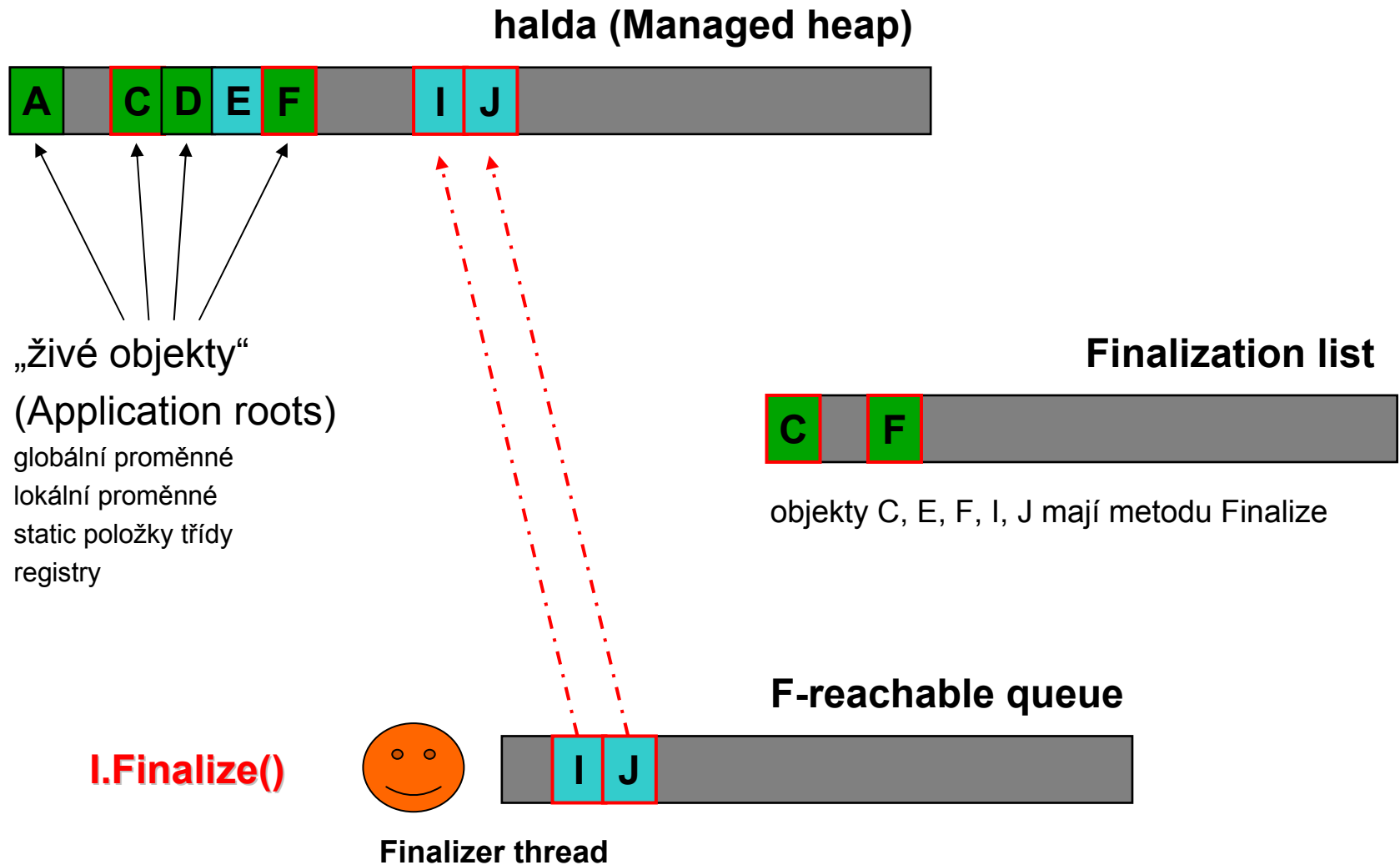
Garbage collector a metoda Finalize 5/12



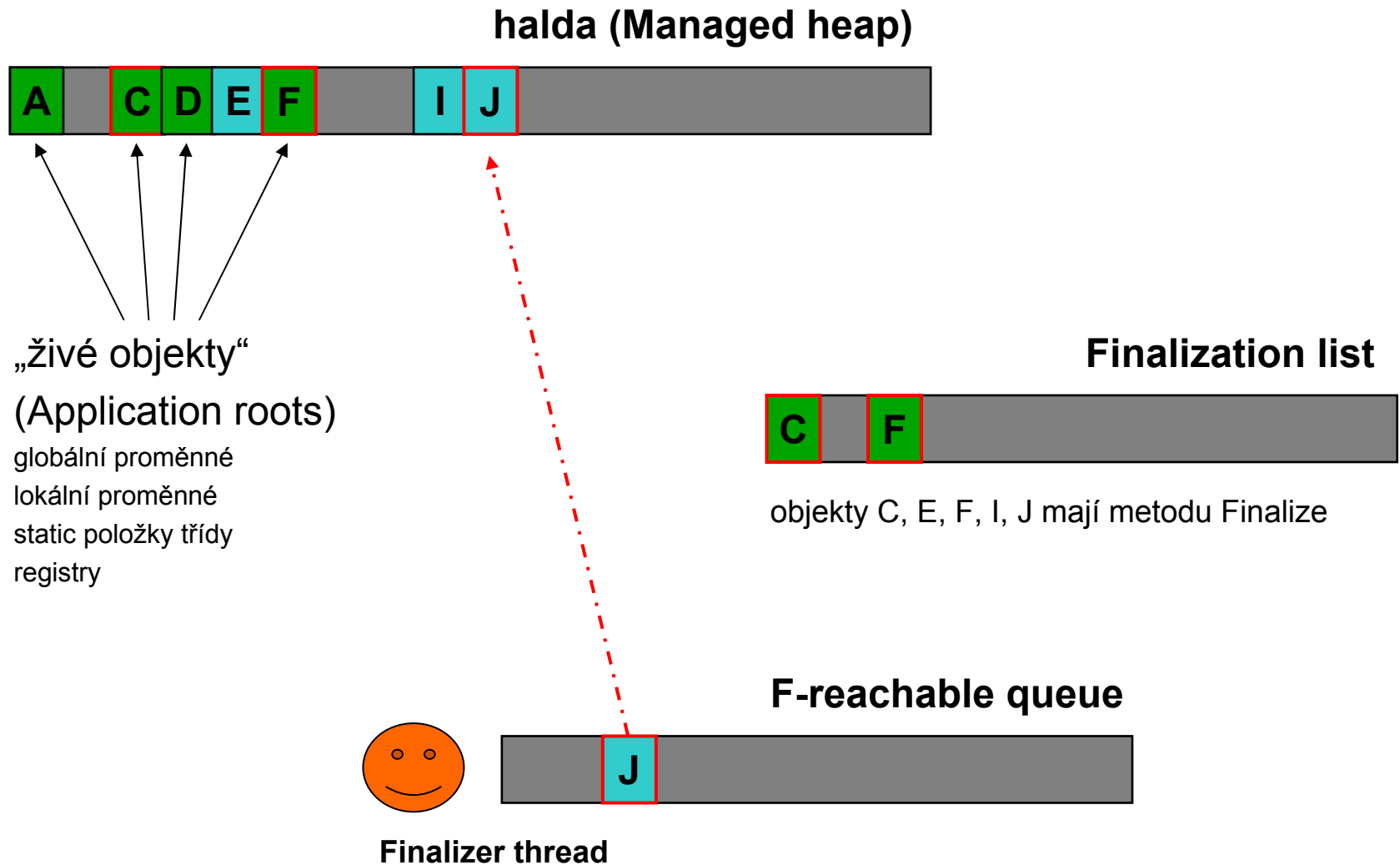
Garbage collector a metoda Finalize 6/12



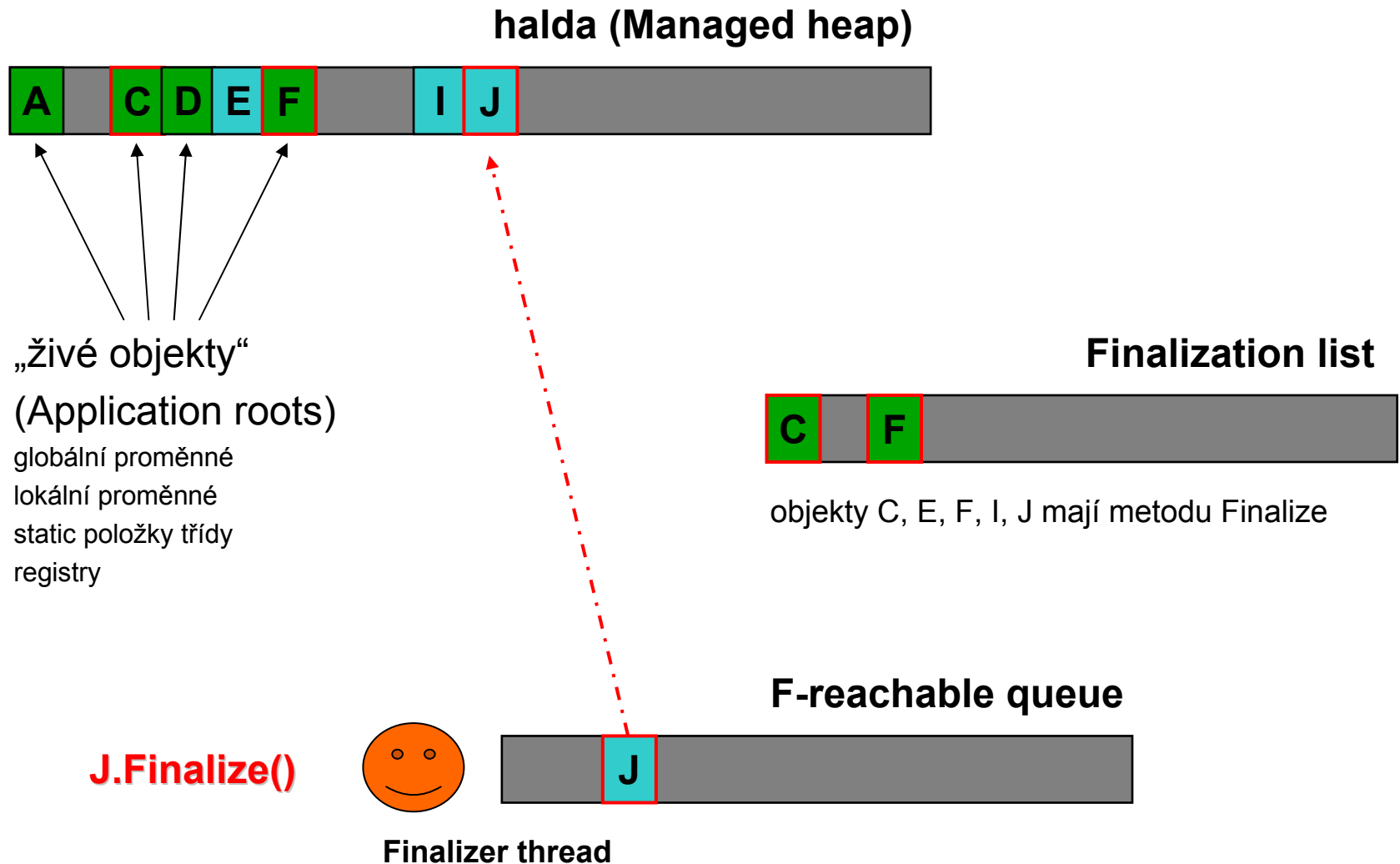
Garbage collector a metoda Finalize 7/12



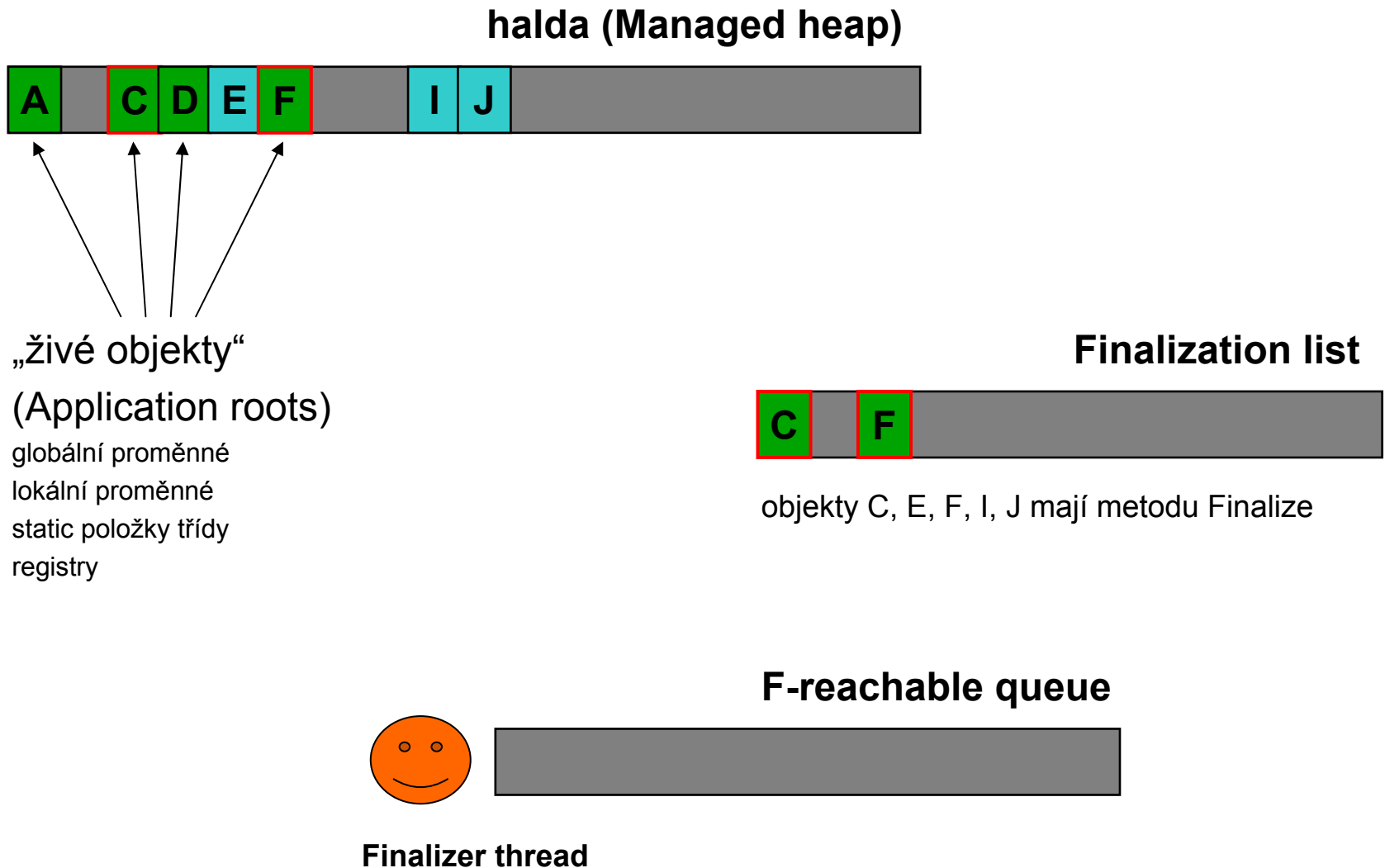
Garbage collector a metoda Finalize 8/12



Garbage collector a metoda Finalize 9/12

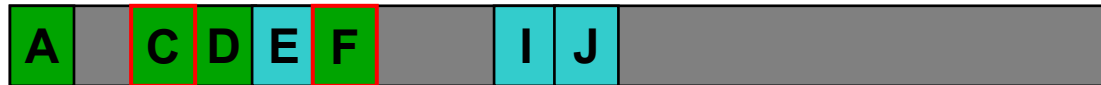


Garbage collector a metoda Finalize 10/12



Garbage collector a metoda Finalize 11/12

halda (Managed heap)



„živé objekty“
(Application roots)

globální proměnné
lokální proměnné
static položky třídy
registry

Další garbage collection...

Finalization list



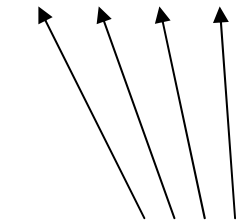
objekty C, E, F, I, J mají metodu Finalize

F-reachable queue



Garbage collector a metoda Finalize 12/12

halda (Managed heap)



„živé objekty“

(Application roots)

globální proměnné

lokální proměnné

static položky třídy

registry

Finalization list



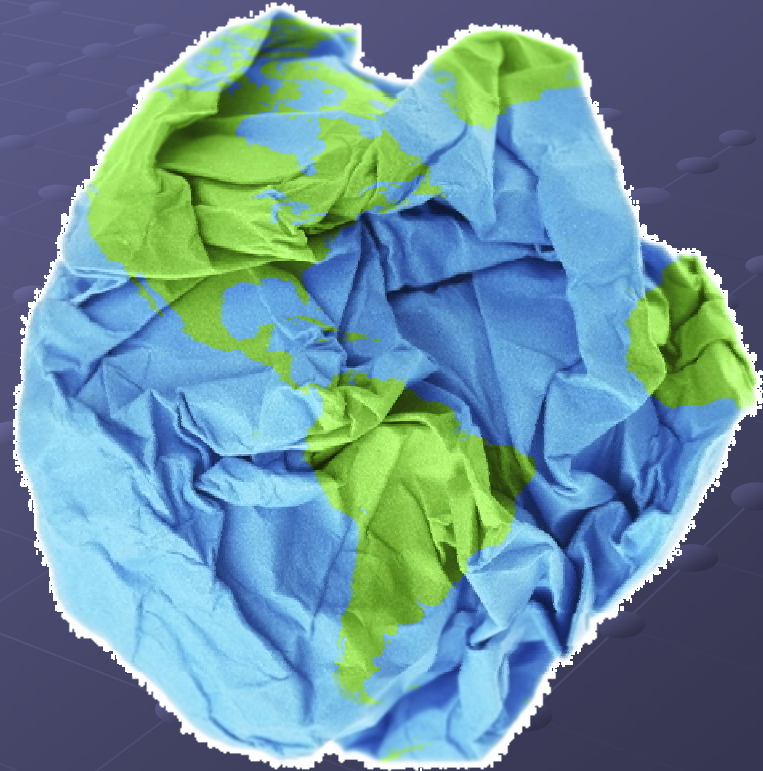
objekty C, E, F, I, J mají metodu Finalize

F-reachable queue



Využití GC

- Finalize
- Dispose
- Weak reference



[<http://www.5minutesforgoinggreen.com/>]

Deklarace Finalize

// Metoda Finalize uvolňuje "unmanaged" zdroje. Nelze ji redeklarovat přímo.

```
class MuJObject {  
    protected override void Finalize()  
    {  
        /* moje příkazy */  
    }  
}
```

// Nutno zapsat ve formě C# destrukturu

```
class MuJObject  
{  
    ~MuJObject()  
    { /* moje příkazy */ }  
}
```



// Výsledek překlada

```
class MuJObject  
{  
    protected override void Finalize()  
    { try { /* moje příkazy*/ }  
      finally { base.Finalize(); }  
    }  
}
```

Překladač si sám generuje metody Finalize → záruka jejich správného tvaru.

Deterministický destruktork – Dispose

- Metoda Finalize() řeší vnitřní složitější vazby mezi objekty
- Nehodí se však pro systémové zdroje
— GC ji volá se zpožděním.
- Někdy potřebujeme likvidační metodu
— tu lze pojmenovat libovolně, avšak doporučuje se **Dispose** zahrnuté v interface **IDisposable**

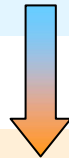
// Příklad použití IDisposable

```
class MojeTridaSDispose : IDisposable
{
    public void Dispose()
    {
        /* úklid, rušení systémových zdrojů */
        GC.SuppressFinalize(this); // již jsme uklidili
    }
}
```

```
interface IDisposable
{ void Dispose();
}
```

IDisposable dovoluje využít **using** konstrukce

```
using (MojeTridaSDispose mtd = new MojeTridaSDispose ())  
{ /* operace s mojí třídou */  
}
```



```
// výsledek překladač  
try { MojeTridaSDispose mtd = new MojeTridaSDispose();  
      /* operace s mojí třídou */  
    }  
finally { if (mtd != null) mtd.Dispose(); }
```

```
//Hodně systémových tříd implementuje IDisposable, viz. cvičení  
using (FileStream fs = File.Open( @"X:\VYUKA\RTIME\PJR\ECG\PERSON1.ECG",  
                                   FileMode.Open, FileAccess.Read)  
    )  
{  
    /* čtení EKG dat... */  
}
```

Alternativní oživení zrušeného objektu.

Povolíme možné zrušení objektu při nedostatku paměti, pokud bude znovu potřeba, otestujeme, zda nedošlo mezitím ke zrušení, pokud ne, obnovíme.

Příklad: "strom adresářů na disku" - uživatel sice přepnul do jiného okna, ale může se náhodou vrátit zpět.

System.WeakReference

```
ObjektAdresar a1=new ObjektAdresar( /* cesta */);  
/* Po skončení práce vložíme a1 do seznamu slabých referencí */  
System.WeakReference wra1 = new WeakReference(a1);  
a1=null; // zrušíme odkaz na objekt, GC může uvolnit  
/****** nějaká jiná práce *****/  
// Uživatel se vrátil zpět — podíváme se, jestli objekt a1 existuje... .  
Object o = wra1.Target; // zkusíme převést na silnou referenci  
if(o == null) // Podařilo se?  
{ a1=new ObjektAdresar( /* cesta */ );  
// bylo již zrušeno -> adresář nutno znovu načíst.  
}  
else { a1 = (ObjektAdresar) o; }  
// Stále existuje, přetypujeme na původní objekt.
```

II. Speciality C# převzaté z C++

Generické typy



■ Možno aplikovat na

- Class, struct, interface, delegate types

```
class Dictionary<K, V> { ... }  
struct HashBucket<K, V> { ... }  
interface IComparer<T> { ... }  
delegate R Function<A, R>(A arg);
```

```
Dictionary<string, Customer> customerLookupTable;  
Dictionary<string, List<Order>> orderLookupTable;  
Dictionary<string, int> wordCount;
```

- Parametry můžeme použít u
 - Class, struct, interface, delegate types
 - Methods

```
class Utils
{
    public static T[] CreateArray<T>(int size) {
        return new T[size];
    }

    public static void SortArray<T>(T[] array) {
        ...
    }
}
```

```
string[] names =
    Utils.CreateArray<string>(10);
names[0] = "Jones";
...
Utils.SortArray(names);
```

```
static void Swap<T>(ref T lhs, ref T rhs)
{
    T temp;  temp = lhs;  lhs = rhs;  rhs = temp;
}
```

```
public static void TestSwap()
{
    int a = 1; int b = 2;
    Swap<int>(ref a, ref b);
    Swap(ref a, ref b); // vynechání typu - překladač doplnil
    System.Console.WriteLine(a + " " + b);
}
```

■ Možno přidat omezení

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

- Žádné nebo primární omezení (constraint)
 - `class / struct`
- Žádné nebo více druhotných omezení
 - Interface, typový parametr
- Žádné nebo omezení konstruktoru
 - `new()`

- where T: struct
- where T : class
- where T : new()
- where T : <base class name>
- where T : <interface name>
- where T : U

```
class Base { }  
class Test<T, U>  
    where U : struct  
    where T : Base, new() { }
```

Primární

```
class Link<T> where T: class { ... }
```

```
class Nullable<T> where T: struct { ... }
```

Druhotné

```
class Relation<T, U> where T: class where U: T { ... }
```

Nevázané - Unbounded Type Parameters

= parametry nemající omezení se nazývají "Unbound" (nevázané?)

- **Nelze** pro ně obecně použít `!=` a `==` , *neboť příslušný typ nemusí operace podporovat.*
- **Lze je** sice porovnávat jen s `null` – *ale pro hodnotové typy se vrací vždy `false`*
- **Lze je** však převést na **`System.Object`**, *neboť C# dovoluje cokoliv převést na `Object`.*
- **Lze je** také explicitně převést na jakoukoliv interface, *použitý typ musí pak interface ale implementovat, jinak se hlásí chyba.*

Naked Type Constraints

//naked (nahý?) type constraint

```
public class SampleClass<T, U, V> where T : V { }
```



```
class List<T>
```

```
{
```

```
    void Add<U>(List<U> items) where U : T
```

```
        { /* ... */ }
```

```
}
```



Souhrn: Klíčové slovo where

```
interface IComparable<T> {  
    int CompareTo(T other);  
}
```

Interface a třídy mohou mít instance

```
class Set<T> : IEnumerable<T>  
where T : IComparable<T>  
{
```

Podmínkou lze odkazovat na parameter typu
("F-bounded polymorphism")

```
    private TreeNode<T> root;  
    public static Set<T> empty = new Set<T>();  
    public void Add(T x) { ... }  
    public bool HasMember(T x) { ... }  
}
```

I typ statics může mít parameter

```
Set<Set<int>> s = new Set<Set<int>>();
```

Typový parameter může být
referenční nebo hodnotový

Generics

- Collection classes

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

- Collection interfaces

ICollection<T>
IEnumerable<T>
IEnumerator<T>
IComparable<T>
IComparer<T>

- Collection base classes

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

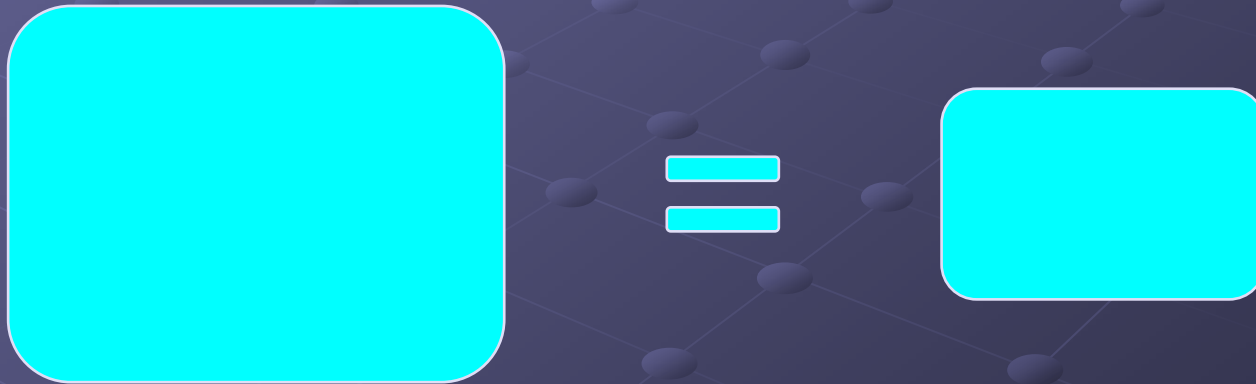
- Utility classes

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

- Reflection

Variance .NET 4.0

- Co-variance
- Contra-variance






```
string astring = ...;  
Dump(astring);
```

OK


```
void Dump(object obj) {  
    ...  
}
```



```
string[] strings = ...;  
Dump(strings);
```

OK

```
void Dump(object[] objs) {  
    ...  
}
```



```
string[] strings = ...;  
Dump(strings);
```

Runtime chyba

```
void Dump(object[] objs) {  
    objs[0] = new JinyObjekt();  
}
```



```
IEnumerable<string> strings = ...;  
Dump(strings);
```

.NET 3.x Chyba při překladu

```
void Dump(IEnumerable<object> objs) {  
    ...  
}
```

```
Converter<object, RotateTransform> c = ...;  
Apply(c);
```

OK <- class RotateTransform:Transform

```
void Apply(Converter<object, Transform> c) {  
    ...  
}
```



```
Action<Base> b = (target) => {  
Console.WriteLine(target.GetType()  
.Name); };
```

```
Action<Derived> d = b;  
d(new Derived());
```

- Pokud se T používá jako výstup, lze aplikovat $X<T\text{Odvozena}>$ na definici $X<T\text{Zakladni}>$
 - Nazývá se to *covariance*
- Pokud se T používá jako vstup, lze aplikovat $X<T\text{Zakladni}>$ na $X<T>$
 - Nazývá se to *contravariance*

Implementace



The Dynamic Tower (Dynamic Architecture Building, Da Vinci Tower) 420-metrů, 80 pater, Dubai

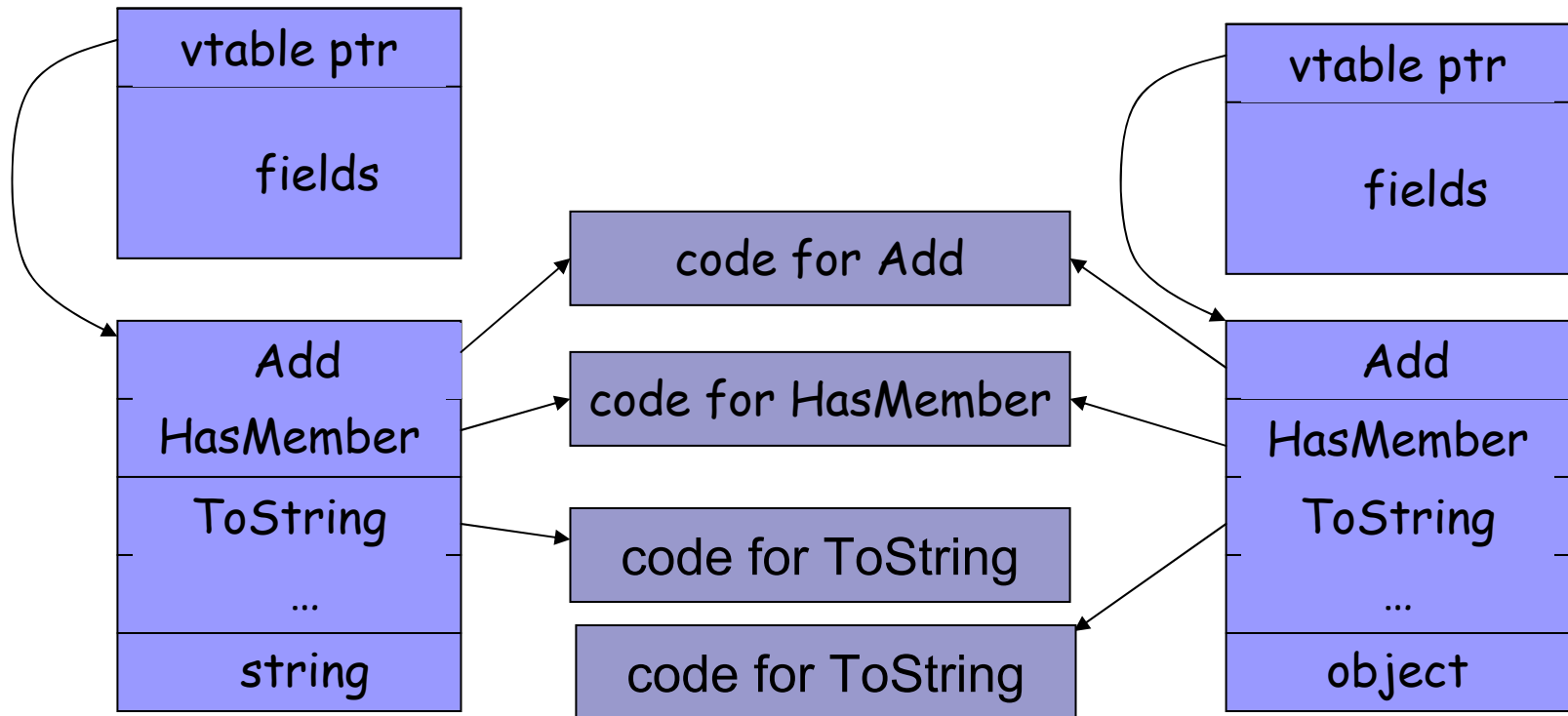
■ Polymorfní metody

- ☐ Instance se vytváří na žádost
- ☐ Lze sdílet kód mezi instancemi
- ☐ Run-time přetypování se řeší na základě interních “slovníků”

Kód se pokud možno sdílí

$x : \text{Set}\langle \text{string} \rangle$

$y : \text{Set}\langle \text{object} \rangle$



■ Non-generic quicksort:

```
void Quicksort(object[] arr, IComparer comp)
```

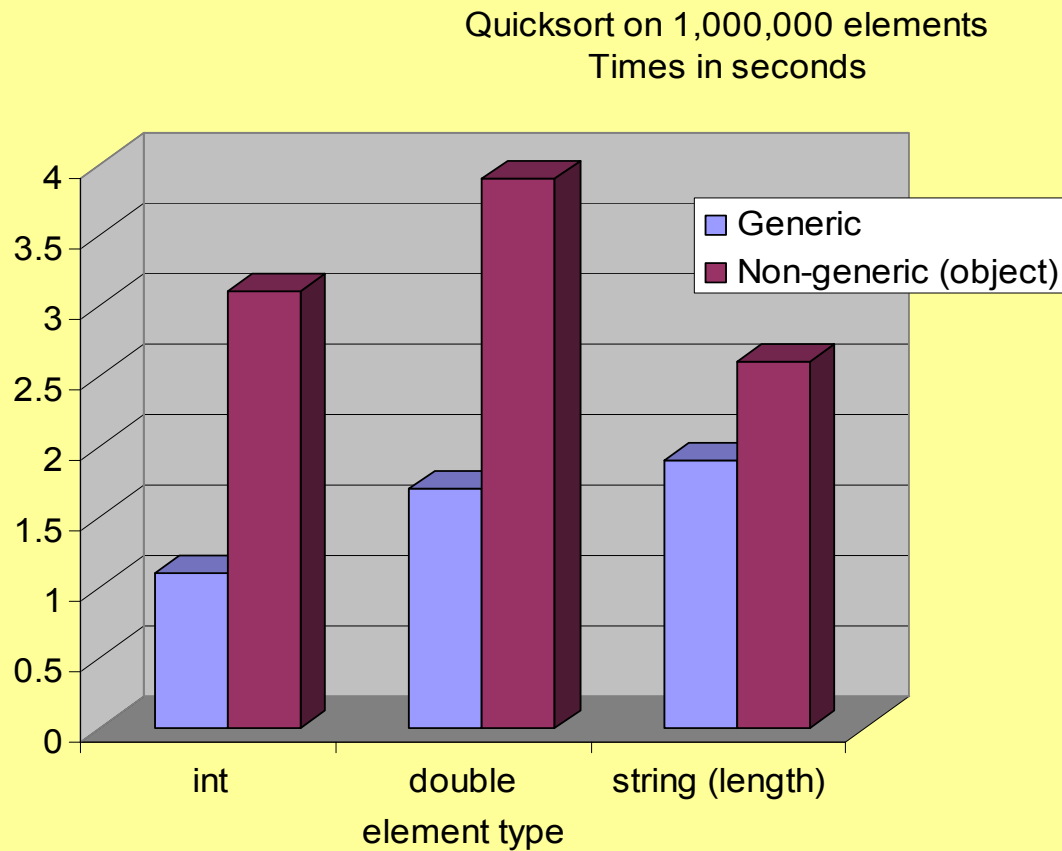
- porovnání vyžaduje přetypování

■ Generic quicksort

```
void GQuicksort<T>(T[] arr, IComparer<T> comp)
```

- přímé porovnání

Performance



Převzato od Claudio Russo MSR Cambridge