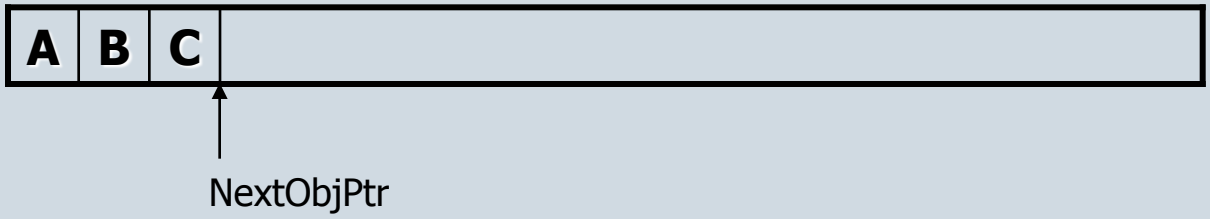
Стратегии на управление на памет и събиране на боклук в .NET платформа

**** Бъгове при досегашните стратегии***

**** управляван heep в Common Language Runtime средата: последователно заемане, съседни в програма-съседни в кеша***

**** heep на C runtime библ.:поддържа свързан списък на блокове памет.*** (защо не може горната стратегия?- преобразуванията в типове в run-time и вътрешни референции няма как да се определят. Тук метаданново описание.)



Ами когато паметта свърши??

Алгоритъм за събиране на боклук

-OutOfMemoryException

-Формиране на корени (roots) и боклук във всеки момент.

Корени могат да са: всички глобални и статични референтни типове; локални пром. от референтен тип; регистрови променливи. Корени са и обекти, реферирани от обхващащи повиквания нагоре из стека на нишката.

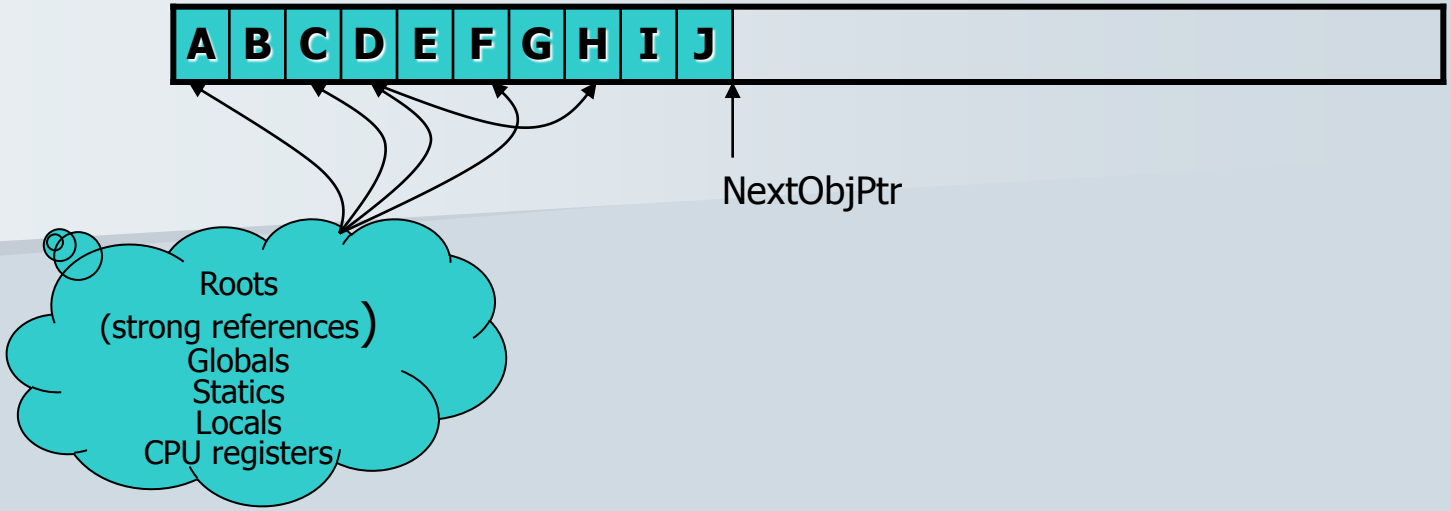
-Таблица на JIT компилатора с области на код и достижимост на обекти :

Примерна таблица, създадена от JIT компилатора, съдържаща съответствия на отместванията в собствения код с корените на метод

Отместван на началния байт	Отместване на крайния байт	Корени
0x00000000	0x00000020	`this, arg1, arg2, ECX, EDX`
0x00000021	0x00000122	`this, arg2, fs, EBX`
0x00000123	0x00000145	`fs`

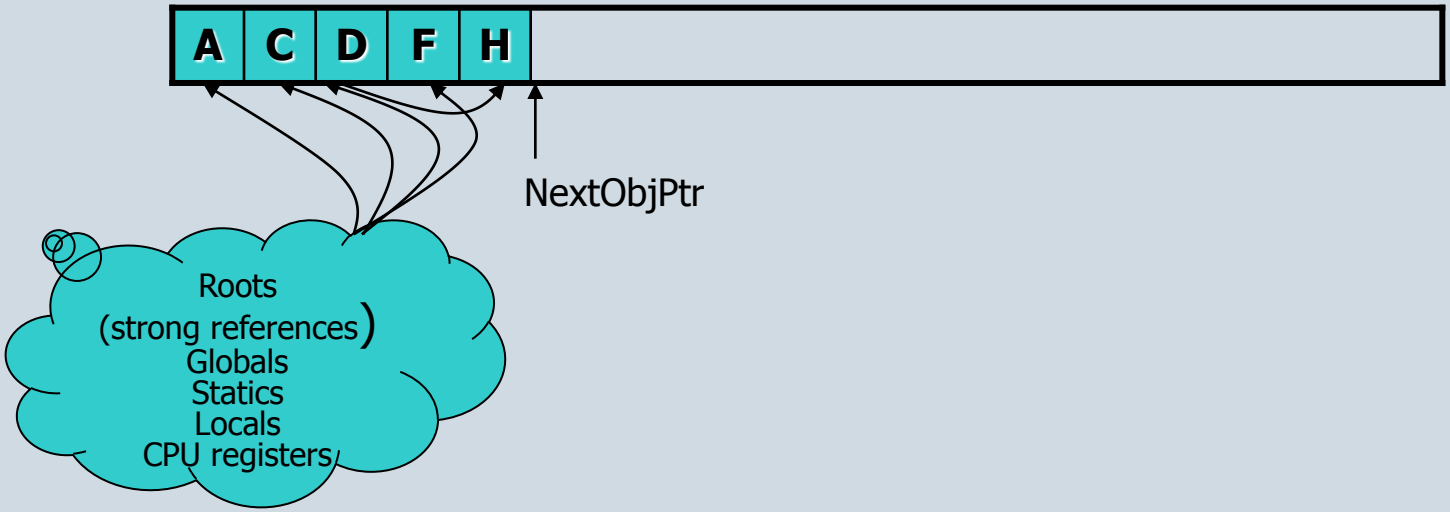
граф на достижимите обекти от корените
(създаван на база на таблицата динамично от JIT компилатора):

Managed heap преди събиране на боклука



memcpy()

Managed heap – след събиране на боклука (обекти и указатели се разместват)



 # може и да не пишете код, управляващ времето на живот на обектите;
 # вече няма неизползвани обекти – всички те ще се премахнат скоро;
 # освобождаване на обект и липса на достъп до обекта са еквивалентни твърдения.

```
Пример:
class My  {
static void XX() {
            ArrayList a = new ArrayList();          //'a' е корен
            for(Int32 x=0; x<1000; x++)
                {a.Add(new Object());}              //'a' реферира 1000 обекта
            Console.WriteLine( a.Lenght);           // използваме 'a'
//липсва друго използване на 'a' – то може да се почисти като боклук
……                }
            }
```

Какво да стане в момента на унищожаване? Това се определя от Finalize(), Dispose(), Close(), които програмистът пише.

Напр. типове, които обвиват неуправляеми ресурси (файл,mutex,сокет..) изискват явно финализиране – Finalize() за почистването си.

Пример за добра (от структурна гледна точка) финализация:

```
protected override void Finalize(){
                try { CloseHandle(handle); } // ако тук има exception?
                finally { base.Finalize(); }
```

Това прави и деструкторът клас в управляем код, ако срещне:

```
                ~MyClass(){
                        CloseHandle(handle);
                        }
```

В C#, Managed C++,.. (всички __gc класове наследяват Finalize() от Object) това става в синтаксиса на деструктора:

Т.е. ако имаме: ***~MyClass()***
{ Console::WriteLine(S"Finalizing process ..");}

То компилаторът автоматично го е разширил към:

```
MyClass::Finalize()
        {Console::WriteLine(S"Finalizing process ..");
          MyBaseClase::Finalize();}
virtual ~MyClass()
        {System::GC::SuppressFinalize(this); A::Finalize();}
```

//"A" обединява всички обръщения към обекти, реферирани в конструктора **, вкл и горния за да**
// може да извиква и техните финализации
// финализацията е отложен процес. Т.е. за даден период от време обектът остава в паметта

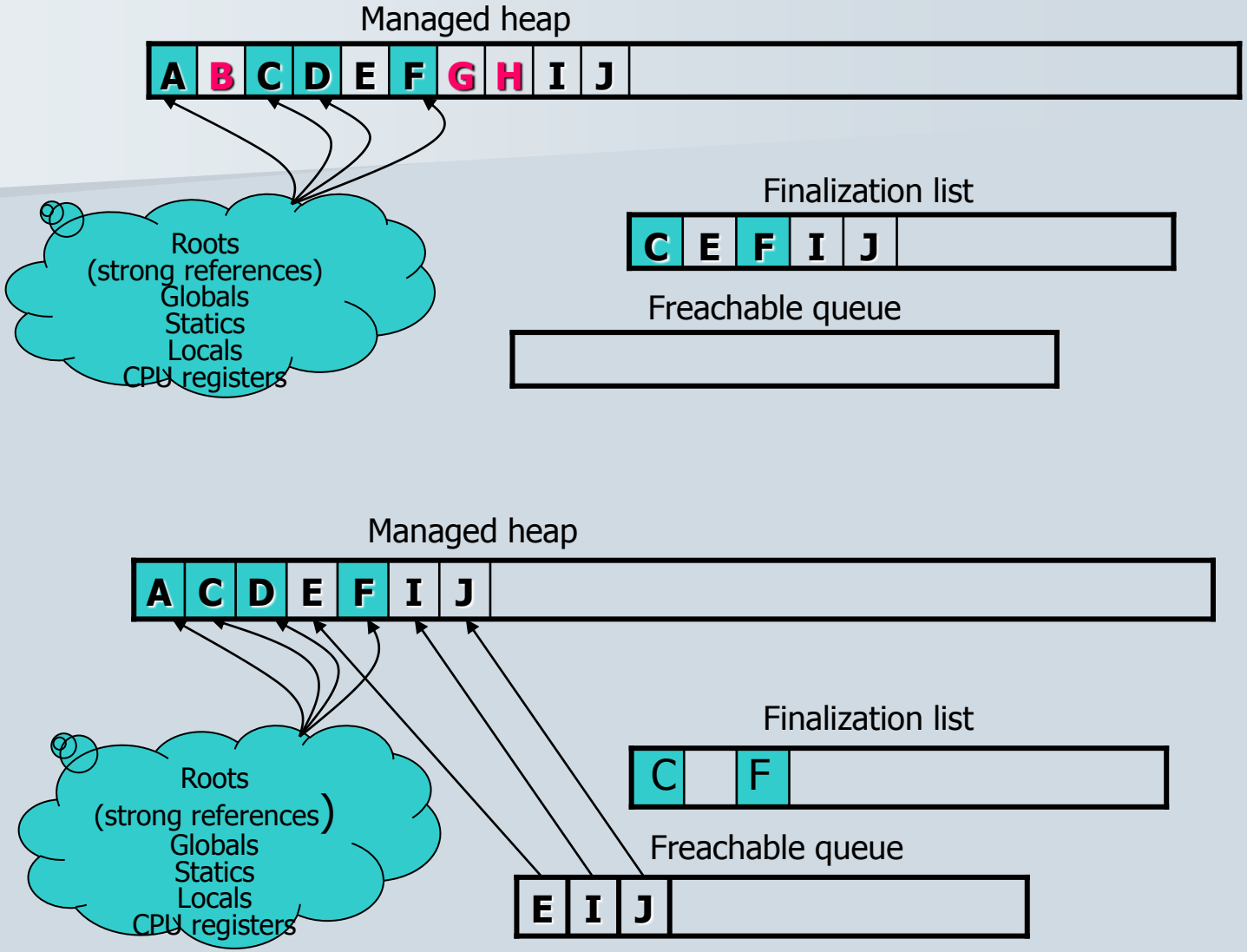
Недостатъци на Finalize():

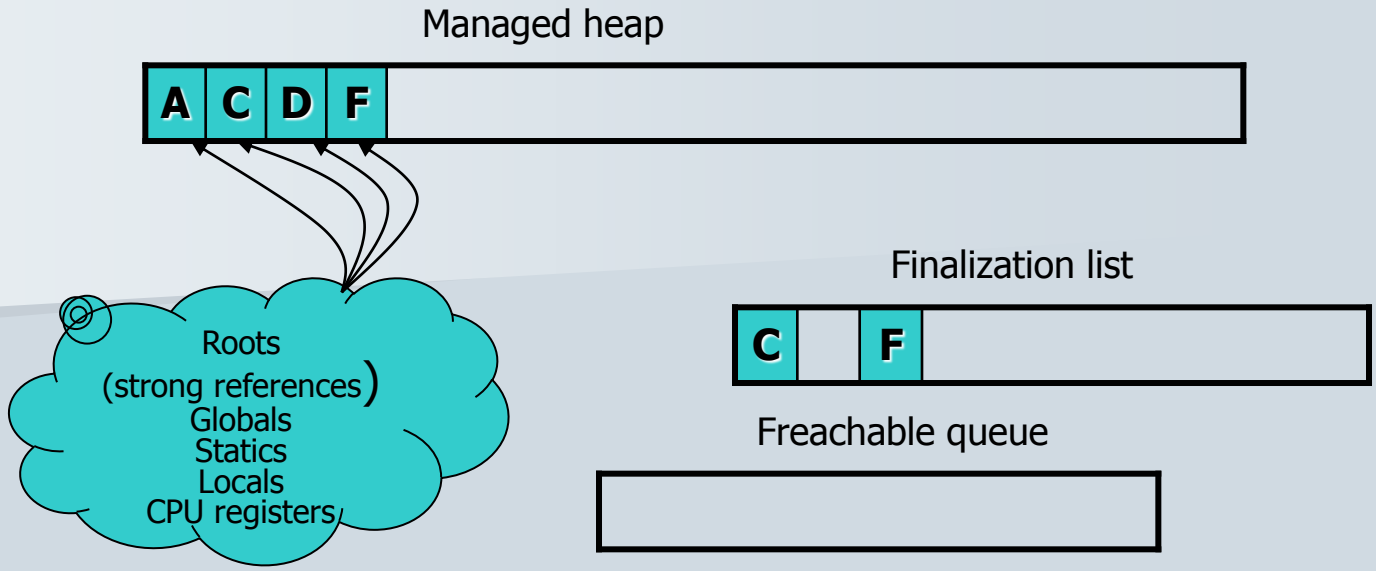
- ***финализацията е по-бавен процес от деструкцията преди*** ***(формира се списък на финализиране, а самото унищожаване става при последващ почистващ пас)***
- ***финализирането за някои обекти се отлага във времето, поради системата за повишаване в поколение***
- ***вътрешни референции могат да объркат процеса на финализирането им. няма гаранция за реда на изпълнения на Finalize(). Следователно, да се избягва вграждане на обекти, когато за тях и за обграждащия ги е предвиден Finalize() (те може да са вече финализирани).***
- ***няма контрол кога се изпълнява Finalize()***

Кога се вика Finalize():

1. ***поколение 0 се е запълнило;***
2. ***явно обръщение към System.GC.Collect()***
3. ***процес завършва – вика се Finalize() за всички обекти в него.***
4. **Не се вика по подразбиране, ако обект излезе от обхват, или завърши метод**

Как работи финализирането





•Високоприоритетна нишка изпълнява Finalize() от Freachable (reachable with finalize())опашката, когато в нея има обекти за всеки от тях.

•Викането на Finalize() е от отделна нишка, следователно методът не трябва да е зависим от викащата го нишка (например да съдържа код, които пипа в локалното пространство на викащата го нишка)

• докато обект е в freachable, той е достижим и не е боклук и паметта му не се възстановява. т.е. Той е предвиден за унищожение, но все още не е. Значи програмиста го счита за недостижим, но той реално е реферирал и указатели към него 'неволно' могат да се ползват

• За даден интервал може да се счита, че обектът е "съживен"

• След изпълняване на Finalize(), freachable опашката се изпразва. Вече нищо не реферира тези обекти.

• при следващото събиране на боклука, паметта за тези обекти се освобождава

Унищожение с Finalize()– не се знае КОГА. Finalize не е public и не може да се вика

модел на 'явно ликвидиране'

(всеки тип, дефиниращ Finalize, следва да имплементира този модел)

Интегриране на Finalize() и Dispose()

```
__gc class sealed Tester : public IDisposable
{   bool bDisposed;
public:
   Tester()                                  {  Console::WriteLine(S"Tester constructor");  }
   ~Tester()                                 {  Console::WriteLine(S"Tester finalizer");  Dispose(false);}
public void Dispose()                        {GC.SuppressFinalize(this); Dispose(true);}
public void Close()                          {Dispose(true);}
private  void Dispose(bool bDisposed) {if(bDisposed) {/*явно финализиране/затваряне*/
                                  Console::WriteLine(S"Tester dispose");
                                  /* код за освобождаване на ангажираните ресурси в обекта*/}
                                       else{ */неявна финализация с Finalize()*/}
```

Dispose() и Close() трябва да са public и невиртуални – за да не се предефинират Но Dispose() е интерфейсен метод и следователно е виртуален. Затова 'sealed' за класа или поне за Dispose()

```
// This is the entry point for this example
int _tmain(void)
{Console::WriteLine(S"Finalization Test");    Tester* pt = new Tester();
   /* Call Dispose */ pt->Dispose();
   /* Try calling a method on the object: */ pt->a.Finalize(); //System.ObjectDisposedException
   Console::WriteLine(S"End of Test");    return 0;}
// ресурс трябва да бъде освободен (Dispose) и след това изтрит (напр.
// File.Delete(..)). Иначе се счита, че той е използван (напр. от друг процес).
// Почистване с Dispose форсира финализацията, но не унищожава обекта от
//паметта – това прави събирането на боклука.   Т.е. могат ( рисковано) да се викат
// методи на обекта – до определен момент те ползват валидни указатели.
```

Коментари относно реализацията на модела на 'явна финализация'

- *Всъщност сме имплементирали интерфейс*

```
public interface IDisposable    {
                                 void Dispose(bool)
                                 }
```

**методите Dispose() и Close() нормално са public .*

Що се отнася до интерфейсния метод Dispose(bool), то в C# имплеметацията на интерфейсен метод е по подразбиране public, sealed. Така че тук следва да внимаваме за модификатора на достъп на Dispose(bool) :

private (ако класът е sealed) или

protected (ако не е Sealed). Тогава наследниците на класа ще могат да предефинират само Dispose(bool) и да използват наготово реализациите на Finalize(), Close() и Dispose().

- *Целият почистващ код е в Dispose(bool)*

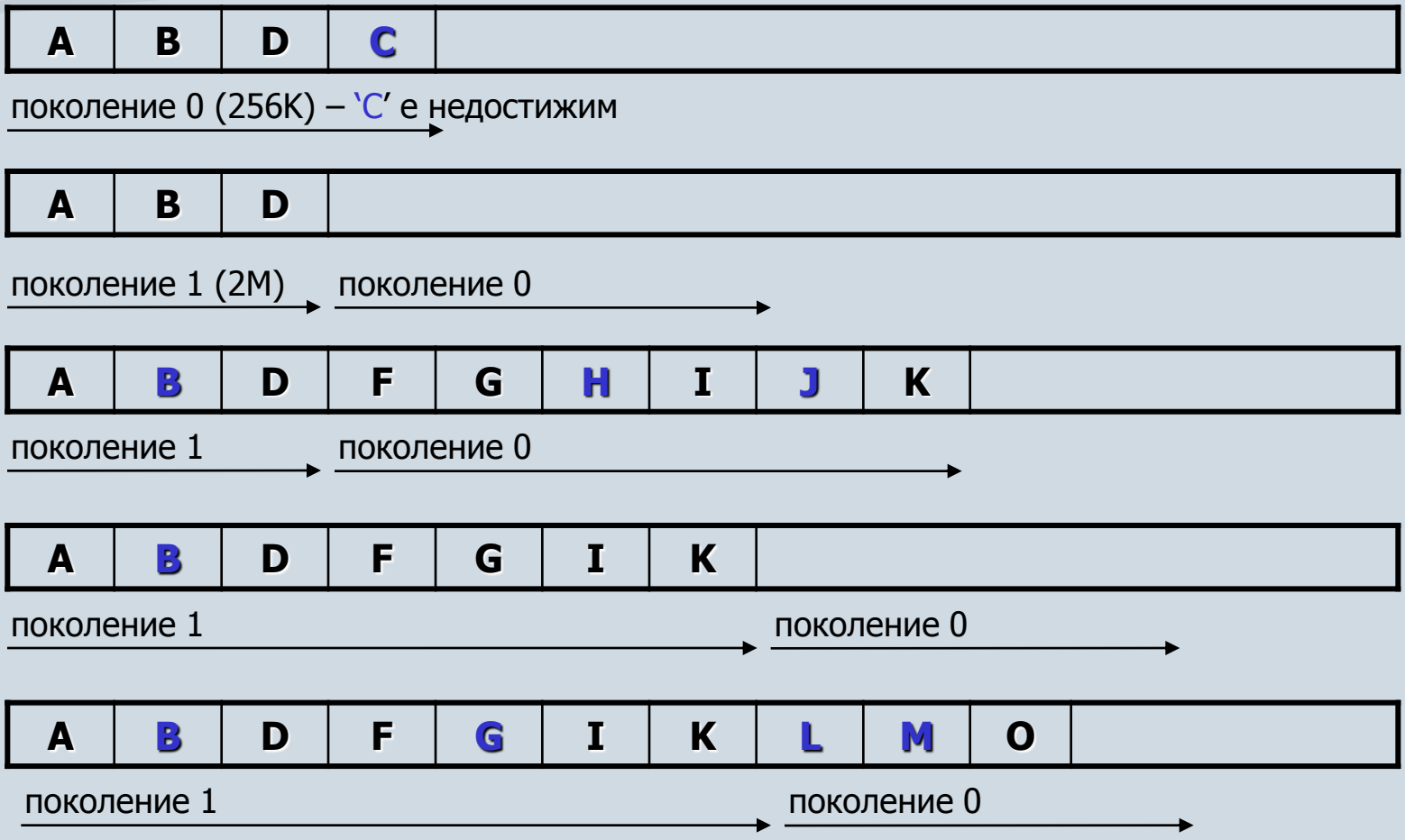
- *в Dispose(bool) следва да предвидим проверка, изключваща възможност от повтарящи се действия(напр. 2 пъти да се затвори handle) – при явна и неявна финализация, както и от многократни викане на Close() и Dispose() от различни нишки(в C# с констр. напр. lock)*

- *Трябва да се предвиди проверка и дали викането е от явно финализиране или от GC системата (това става с флага bool , подаван на Dispose(bool)). При CG -финализиране – да се изключи код в Dispose(bool), реферираш други managed (вградени) обекти, докато при явна финализация – такива референции да допустими (и дори нужни).*

- ***SupressFinalize()** блокира стандартното викане на Finalize() за да няма повторение. Тогава GC системата просто ще унищожи обекта, без да го слага в freachable опашката*

Поколения. Допускания:

- ***нов обект → по-кратък живот; стар обект →по-дълго може да живее***
- ***събиране на част от heap е по-бързо от събиране на целия heap.***
- ***неизследвани обекти в управлявания heap са от поколение 0.***



- ***Системата е самонастройваща се: напр. ако след освобождаване на поколение 0 не се е освободила реално много памет, то границата се разширява напр на 512;***
- ***В WEB (напр. ASP.NET) където вследствие клиентски заявки се формират куп обекти, системата работи с по-висока ефективност***
- ***Големи обекти (над 100 KB) се заделят от отделен heap. Финализацията работи по същия начин, но без преместване за свиване на паметта (това би било по-бавно). Те се считат като че винаги са в поколение 2 – т.е. по-рядко се финализират. Ако обаче се ползва явно финализиране за унищожението на големи обекти, то се форсира честа обработка и през всички поколения (0,1,2), което силно влошава производителността***

Програмно активиране системата за събиране на боклука:

1) void GC.Collect(Int32 Generation); //указвате кое поколение да се събира
// събира указаното поколение и тези под него (напр при '1': поколение1 и 0)

2) void Gc.Collect()
3) void Collect(Int32 generation, GCollectionMode mode)

Където 'mode' може да е:

- ***Default*** **- за момента действа както и без флаг;**
- ***Forced*** **- форсира стартиране на GC за всички поколения;**
- ***Optimized*** **– опитва с дефрагментация или GC за младшо поколение , докато се удовлетворят исканията за памет.**

- **добре е стартиране на GC ако знаете, че някакви обекти вече не са ви нужни;**
- **добре е стартиране на GC ако сте токущо съхранили някакви данни във файл;**
- **Или при 'upload ' на 'Windows Form' контрол;**
- **Можете да повикате и метода GC.WaitForPendingFinalizers();**
- **Или ако знаете, че GC ще изисква доста време за събиране на обектите, можете явно да го повикате предварително, в по-ненатоварен времеви интервал.**

Можете да получите и информация в кое поколение е определен обект:

Int2 GetGeneration(Object obj);

Предварително оценяване за наличност на големи обеми памет (.NET)

- **MemoryFailPoint class в System.Runtime namespace:**

```
public sealed class MemoryFailPoint : CriticlalFinalizerObject, IDisposable
{
        public MemoryFailPoint(Int32 sizeInMegabytes);
        ~MemoryFailPoint();
        public void Dispose();
}
```

Начин на използване на механизма:

-Създава се инстанция на класа с подаден обем памет в MB, които оценъчно ще се заделят

- проверките са следните:

- **има ли достатъчно място (в paging file) , като се взема предвид и предходно заделяне от MemoryFailPoint конструктор (ако има такова);**
- **ако няма памет – GC се стартира в опит да освободи;**
- **ако пак не достига – опитва се с разширение на paging file (при неуспех – exception);**
- **при успех – заявената памет се 'заделя' в thread safe режим, така че същото може да се направи и многократно в други нишки (вкл. и еднотипни).**

В реакция на InsufficientMemoryException (дъщерен на OutOfMemoryException) можете да добавите код (напр кеширане на някакви данни, редуциране на обеми и т.н.)

Използване на техниката:

```
Using System;
Using System.Runtime;
...

public static class Program {
        public static void Main() {
                try {
                        // 'логическо' резервиране на 1.5 GB памет
                        using (MemoryFailPoint mfp = new MemoryFailPoint(1500)) {
                        // алгоритъм , който консумира тази памет

                        } // завършва с повикване на Dispose - 'логически' освобождава
                    }
        catch (InsufficientMemoryException e) {
          // каквото трябва да се случи, ако липсва подобен обем свободна памет
                Console.WriteLine(e);
                }
        }
}
```