Лекция 12. Управление на паметта и ресурсите в .NET. Същност и управление на механизма за "събиране на боклука"

1. Управление на паметта и ресурсите в C#

1. 1. Заделяне на памет

За променливи от стойностен тип

- Паметта, нужна за променлива от стойностен тип (value type) в C# се заделя в стека и се освобождава автоматично, когато променливата излезе от обхват (след като излезе от блока или от метода).

За променливи от референтен тип

- Паметта, нужна за променлива от референтен тип (reference type) в C# се заделя (алокира) в специална област от динамичната памет, наречена „managed heap" и както в почти всички други езици заделянето на „динамична памет" се извършва "ръчно" - обявява изрично от програмиста с ключовата дума new, например: **myObject x = new myObject();**

1. Управление на паметта и ресурсите в C#

1.2. Придобиване (и освобождаването) на ресурс, различен от памет

- „Ръчно" във всички езици става и придобиването и освобождаването на ресурс, различен от памет (например ресурс предоставен от операционната система, конекции към БД, манипулатори на файлове и др.).

1. Управление на паметта и ресурсите в C#

1.3. Освобождаването на заделената памет

За разлика от заделянето, освобождаването на заделената памет в различните езици се решава по 3 различни начина:

- ръчно,
- автоматично или
- смесено.

Ръчното освобождаване е присъщо за C/C++. В C++, чрез оператор delete се извиква код за разрушаване на инстанцията (наричан „деструктор"), след което се освобождава и паметта, която е била заета от нея.

Основни недостатъци на ръчното освобождаване на заделената памет:

- забравянето на оператор delete – води до неефективно използване на паметта (до "висящи референции"), а понякога, при продължително работещи приложения - до изтичане на памет („memory leak"). Това е проблем, който преследва програмистите от много време. Разработени са редица помощни програми, които помагат да се открият точно такива неосвободени блокове от памет или системни ресурси.
- преждевременното подаване на delete - води до грешка, породена от опит за достъп до вече унищожен обект ("невалидна референция").

- .Net използва система за **автоматично освобождаване** на заделената памет (наречена система за почистване на боклук, Garbage Collector, GC), която **обикновено** се задейства автоматично при недостиг на памет.
- Казахме, че когато едно .NET приложение създава нов обект, паметта, необходима за него се заделя в регион, наречен **динамична памет** или **managed heap**. След като обектът е създаден, приложението използва неговата функционалност, и когато обектът стане ненужен, програмистът не се грижи за освобождането му (в C# не съществува оператор delete), а просто го изоставя. В някакъв неизвестен за програмиста момент (най-често при недостиг на памет в managed heap), GC се стартира автоматично, идентифицира всички обекти, които вече не се използват от приложението (третира ги като „отпадъци") и освобождава заетата от тях памет.

Основни недостатъци на този подход:

- програмистът не знае в кой момент ще се стартира GC, нито колко време ще работи, което не се харесва на някои програмисти.
- почистването е доста времеемка операция: когато се стартира GC, всички нишки на приложението заспиват и остават в това състояние докато той завърши своята работа.

Все пак, алгоритъмът, по който работи GC е много добре оптимизиран, така че загубата на контрол и практически недоловимото забавяне от „заспиването" на нишките се компенсират от предимствата, които автоматичното почистване дава.

- Трябва да се подчертае също, че GC се грижи само за обекти, заделени в „managed heap". Така, ако обектът капсулира ресурс, различен от памет, например файлов манипулатор, за неговото освобождаване трябва да се погрижим сами.

- Освобождаването на памет чрез GC не е нова идея. GC се използва и при други езици, като например при Java, разбира се с друг алгоритъм на работа. В тази тема ще разгледаме само този GC алгоритъм, който се използва в .Net.
- Много подробни обяснения по тази тема има в книгите на Наков и колектив, както и на http://msdn.microsoft.com/en-us/magazine/bb985010.aspx, http://msdn.microsoft.com/en-us/magazine/bb985011.aspx.

Кога се стартира Garbage Collector в .Net?

1. При завършване на приложението.
2. Ако при опит за добавяне на нов обект, CLR установи, че в managed heap няма достатъчно място за добавянето му.
3. Чрез изрично извикване на метод GC.Collect() за стартиране на Garbage Collector.

- Съществуват и езици (например, Visual Basic преди VB.NET) със **смесено освобождаване** на заделената памет.
- При тях, паметта може да се освобождава както "ръчно" от програмиста, така и автоматично от система за почистване на боклук.

2. Същност и управление на механизма за "събиране на боклука" (Garbage Collection)

- Когато един процес се инициализира, CLR резервира последователен блок от памет в heap-а за самия процес. Този блок от памет се нарича **managed heap**.
- Когато едно приложение създаде даден обект използвайки оператора **new**, CLR алокира необходимия блок от памет в managed heap за този обект.
- Със създаването на нови и нови обекти в managed heap, той започва да се пълни.
- **Какво става когато искаме да инстанцираме нов обект, но вече нямаме достатъчно памет в heap-а? Ето тук е мястото където Garbage Collector (GC) влиза в действие. Ето какво прави той:**

1. **Открива неизползваните обекти и освобождава заетата от тях памет**. С помощта на ***CLR***, ***GC*** създава списък на тези обекти, които се използват в момента („валидни" обекти). Останалите обекти се считат за боклук (garbage) и могат да бъдат унищожени, а заетата памет - освободена за други обекти.
2. **Финализация**. Изпълнява се специална нишка с висок приоритет, която за всеки запис във Freachable queue изпълнява финализиращия код на съответния обект (в деструктори се пише код за освобождаване на неменажирани ресурси) и след това **изтрива записа от** Freachable queue. Процесът на финализация е разгледан по-подробно в следващата секция.

3. **Пренарежда heap-а.** GC пренарежда паметта, като премахва създалите се "дупки" в heap-а, резултат от освободената памет на невалидните обекти (боклука). При необходимост, валидните обекти се изместват в паметта, така че да заемат разположени последователно блокове от памет. Останалата част от heap-а е свободна за създаване на нови обекти.
4. **Актуализира всички референции към преместените обекти:** тъй като валидните обекти са били изместени в паметта, всички референции към тях са станали невалидни. GC коригира тези референции, така че да сочат към новите местоположения в паметта за съответните обекти.
5. **Удовлетворяване на неизпълнената заявка:** оператора ***new*** се извиква отново и така заявката за памет вече се удовлетворява.

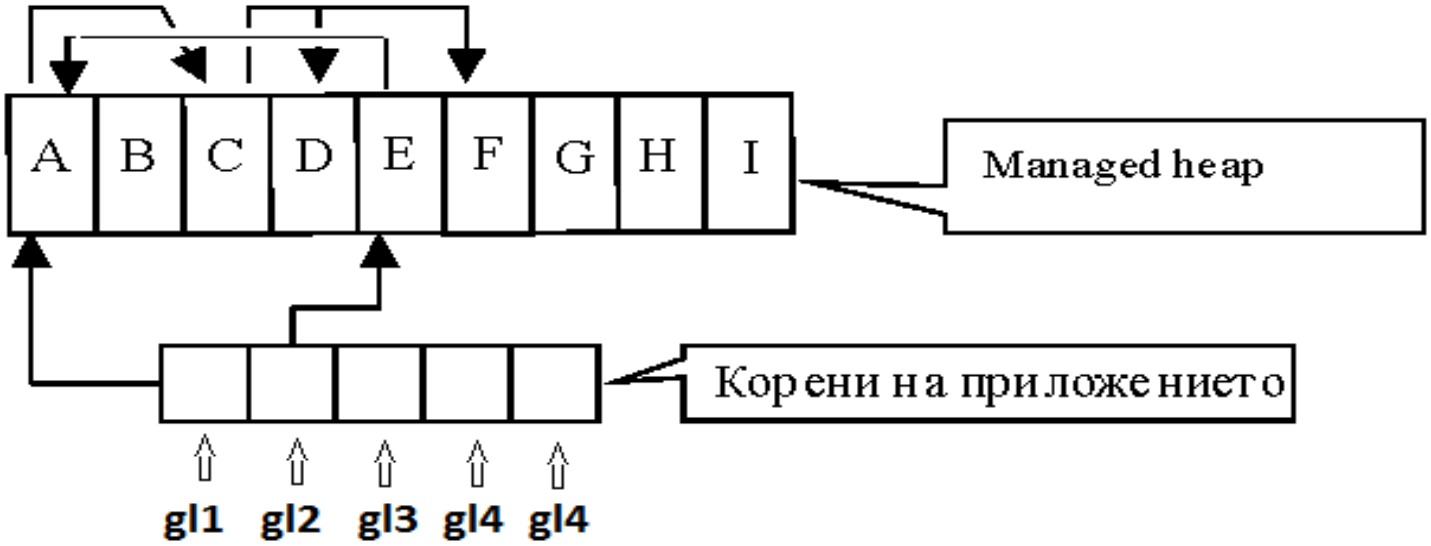
Интересен е въпроса: как се съставя списъка с обекти, които вече не се използват?

- Всяко приложение има набор от **root обекти** (**програмни корени, application roots**), които се променят с изпълнението на приложението. Това са обекти, които GC може да ползва като начална точка за да определи другите обекти, които се използват от приложението в момента.

- Корените (**root обекти**) представляват области от паметта, които сочат към обекти от managed heap, или са установени на **null**, например:
  - всички глобални или статични обекти в едно приложение, съдържащи референции към обекти в heap-а се считат за root обекти.
  - Всички локални променливи или параметри в стека към момента, в който се изпълнява GC, които сочат към обекти в heap-а, също принадлежат към корените.
  - Регистрите на процесора, съдържащи указатели към към managed heap, също са част от корените.
  - Към корените на приложението спада и Freachable queue (за Freachable queue по-подробно ще стане дума в секцията за финализация на обекти).

- Когато GC се стартира, той преглежда текущите root обекти и построява граф на всички обекти, които са достижими от корените на приложението, по рекурсивен алгоритъм за обхождане на граф в дълбочина. Това са реално обектите, които се използват в текущия момент. Останалите обекти са "боклук" и се счита, че могат да бъдат унищожени, а заетата от тях памет - освободена.

- Когато GC започва своята работа, той тръгва с предположението, всички обекти в **managed heap** са отпадъци, т.е. че никой от корените не сочи към обект от паметта.
- Примерът по долу представя работата на GC, свързана с построяване на граф на използваните обекти, достъпни от програмните корени. Ако един програмен корен (например, една глобална променлива gl1) сочи към обекта A от managed heap, то A ще се добави към графа. Ако A съдържа указател към C, а последният от своя страна към обектите D и F, всички те също стават част от графа. Така GC обхожда рекурсивно в дълбочина всички обекти, достъпни от съответната глобалната променлива gl1:



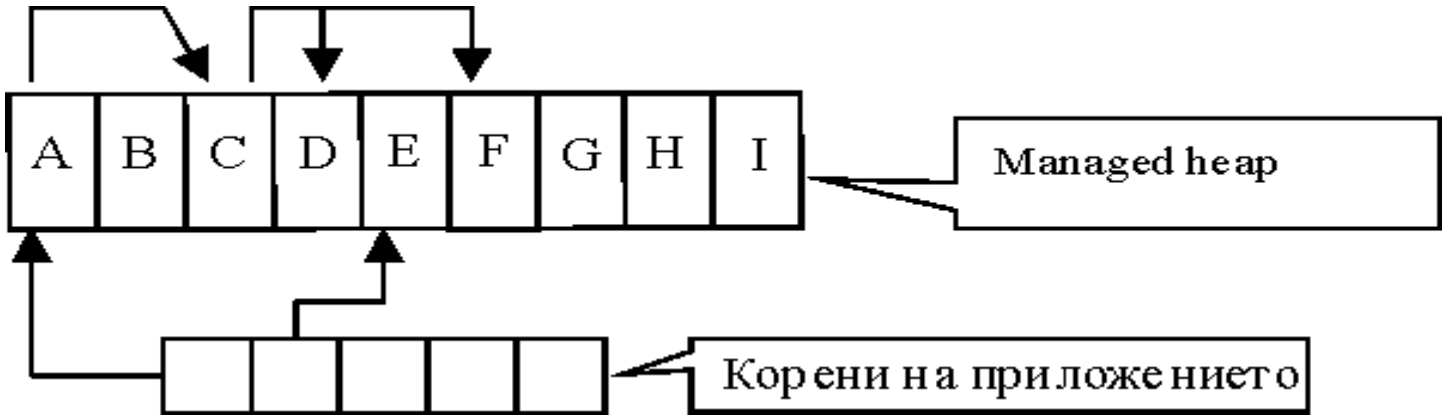
Когато приключи с построяването на този клон от графа, GC преминава към следващия корен (gl2) и обхожда всички достъпни от него обекти. В нашия случай към графа ще бъде добавен обект E.

Ако по време на работата GC се опита да добави към графа обект, който вече е бил добавен (в случая E сочи към A, който е добавен вече в графа), той спира обхождането на тази част от клона. Това се прави с две цели:

- значително се увеличава производителността, тъй като не се преминава през даден набор от обекти повече от веднъж;
- предотвратява се попадането в безкраен цикъл, ако съществуват циклично свързани обекти.

След обхождането на всички корени на приложението, графът ще съдържа всички обекти, които по някакъв начин са достъпни от приложението. В показания с фигурата пример това са обектите A, C, D, E и F.

Всички обекти, които не са част от създадения граф, не са достъпни и следователно се считат за отпадъци: в нашия пример това са обектите B, G, H и I.



Събирането на отпадъците е скъпа операция: работата на GC има значително отражение върху производителността на цялото приложение:

- Построяването на графа на достъпните обекти, преместването на обекти и дефрагментирането, за премахване на дупките в **managed heap** са времепоглъщащи операции, особено при многонишкови (**multithreading**) приложения.
- В този момент CLR спира работата на всички нишки, тоест нишките „заспиват" (**суспендиране на нишки**), за да се увери, че те няма да използват невалидни референции към паметта.

Предвидени са различни механизми в GC алгоритъма, с цел да се осигури колкото е възможно по-дълга работа на нишките и да намалят загубите. Облекчаващ факт все пак е, че GC се стартира само когато има нужда от това (т.е. когато има недостиг на памет).

- По подразбиране, CLR създава допълнителна конкурентна нишка, в която GC да работи, а работата на GC конкурентно, намалява производителността.
- Възможно е, с помощта на конфигурационната настройка ***gcConcurrent*** да се укаже на средата (CLR) да пусне GC в същата нишка, в която работи самото приложение. Това става, както е показано по-долу:

```
<configuration>
<runtime>
<gcConcurrent enabled = "false"/>
</runtime>
</configuration>
```

- За приложения, базирани например на потребителски интерфейс, има смисъл GC да се стартира конкурентно, така че приложенията да не изглеждат като „блокирали" или „забили" по време на работа на GC.
- Но при приложения, работещи на заден план (background applications), които не зависят от потребителски интерфейс, не се препоръчва използване на конкурентно изпълнение на GC.

Програмно стартиране на GC.

- Казахме, че по подразбиране, GC се включва автоматично тогава, когато се създава дадена инстанция на обект, и когато няма достатъчно памет за да се удовлетвори тази заявка.
- Възможно е и програмно да се накара GC да се включи във вашето приложение. .Net Framework предоставя един клас - System.GC, който се използва точно за това. Следният код принуждава GC да се стартира:

```
GC.Collect();
```

- Това извикване на статичния метод Collect() на клас GC без параметри, предизвиква пълно почистване на всички поколения памет. Извикването на същия метод, с аргумент номер на поколение, предизвиква почистване на всички поколения, започвайки от 0 до указаното.
- Като цяло, препоръчва се да оставите GC да се включи тогава, когато CLR прецени. Въпреки това, в някои случаи програмистът може да извика метод GC.Collect(). Например, добро място да извикате GC е, когато приложението не работи, например чакайки някаква намеса на потребителя.

3. Финализацията на обекти в .NET

- Както посочихме, GC ни освобождава напълно от грижите по освобождаване на заделената памет. Но GC не може да се използва за почистване на обекти, които капсулират някакъв системен ресурс, например файлов манипулатор (обект от тип System.IO.FileStream), връзка към база от данни (обект от тип System.Data.OleDb.OleDbConnection) и др.
- Именно за такива обекти се изпълнява **финализация, тоест**: **„преди обекта да бъде унищожен, CLR изпълнява даден код"**.

- За да се изпълни финализация, класът трябва да имплементира специален метод, наречен **Finalize()**. Класът **System.Object** дефинира метод **Finalize()**: **public virtual Finalize()**
- Но, създателите на C# са забранили препокриването (override) на този виртуален метод, както и директното му извикване.
- За извикване на **Finalize()**, **в класа** се дефинира **деструктор, който се извиква автоматично от GC, и който на практика се преобразува във Finalize() метод**. Трябва да се отбележи, че извикването на деструкторите в C# е автоматично от GC, но не е детерминистично. Това означава, че програмистът няма възможност да знае кога ще се изпълнят финализаторите.

```
class MyClass
{…
   ~MyClass () //Деструктор
   {
   //финализиращ код
   //например, код за освобождаване на
   //неменажирани ресурси
   }
}
```

- CLR поддържа две структури, които са свързани с финализацията. Това са т.нар.
  - **Finalization List и**
  - **Freachable Queue.**
- Finalization list съдържа указатели към всички обекти в heap-а, които трябва да бъдат финализирани (тоест имат **деструктори**), но все още се използват от приложението (не са идентифицирани от GC като отпадък все още).
- Когато даден обект се идентифицира като отпадък, GC проверява дали във **Finalization list** съществува указател към този обект. Когато такъв указател няма (какъвто е случаят с обект "B" в примера), неговата памет просто се освобождава, за ползване от други обекти.

- Когато обаче за определения като отпадък обект има указател във **Finalization list**, GC не може просто да унищожи обекта, тъй като преди това трябва да се изпълни неговия финализиращ код
- Указателят към този обект бива изтрит от Finalization list, но бива добавен към **Freachable queue**, с което самият обект продължава да "живее" в динамичната памет.

- Опашката **Freachable** съдържа указатели към всички обекти, чиито деструктори вече могат да се извикат. Всеки обект, за който има запис във **Freachable queue** се **третира като достъпен, а не като отпадък**. В момента в който, за даден обект (например G) се направи запис в Freachable queue – то обектът (в случая G) се **съживява**, т.е. той се добавя към графа на достъпните обекти и вече не се счита за отпадък. Нещо повече – защото е оцелял при едно преминаване на GC, той минава от Поколение 0 в Поколение 1.

- След това CLR стартира специална нишка с висок приоритет, която за всеки запис във Freachable queue изпълнява **Finalize()** метода на съответния обект, след което **изтрива записа от опашката**.

- При следващото почистване на Поколение 1 от GC по някое време (почистването на поколение 1 може да стане след доста стартирания на GC) вече обект G ще бъде счетен за недостъпен, тъй като
  - никой от корените на приложението не сочи към обекта

  и
  - записът от Freachable queue вече е изтрит.
- Вече паметта, заемана от него ще бъде освободена. Трябва да се отбележи обаче, че тъй като обектът вече е в по-високо поколение, преди това да се случи е възможно да минат още няколко преминавания на GC.

Какво представляват поколенията за които споменахме?

- **Поколенията (generations)** са механизъм в GC, чиято единствена цел е подобряването на производителността на GC. Вместо да обхожда всички обекти от heap-а, GC обхожда само част от тях, класифицирайки ги по определен признак, а именно - по поколение.
- В основата на механизма на поколенията стоят следните предположения:

- колкото по-нов е един обект, толкова по-вероятно е животът му да е кратък.
- **Поколение 0** съдържа новосъздадените обекти – тези, които никога не са били проверявани от GC. При инициализацията от CLR се определя праг за размера на Поколение 0.
- колкото по-стар е обектът, толкова по-големи са очакванията той да живее дълго. Пример за такива обекти са глобалните променливи.
- обектите, създадени по едно и също време обикновено имат връзка помежду си и имат приблизително еднаква продължителност на живота.

- Да предположим, че приложението иска да създаде нов обект - F. Добавянето на този обект би предизвикало препълване на Поколение 0. При това положение - стартира се GC.
- Сега оцелелите при преминаването на GC обекти стават част от Поколение 1 (защото са оцелели при едно преминаване на GC). Поколение 1 има праг за своя размер, който се определя от CLR при инициализацията, и който е по-голям от този на Поколение 0. Новият обект F, както и всички други новосъздадени обекти ще бъдат част от Поколение 0.

- Нека сега предположим, че е минало още известно време, през което приложението е създавало обекти в динамичната памет.
- За да освободи памет GC ще прегледа отново само обектите от Поколение 0, тъй като Поколение 1 не е достигнало прага си. Това се диктува от правилото, че по-старите обекти обикновено имат по-дълъг живот и следователно почистването на Поколение 1 не е вероятно да освободи много памет, докато в Поколение 0 е твърде възможно много от обектите да са отпадъци.

- И така, GC почиства отново Поколение 0, оцелелите обекти преминават в Поколение 1, а тези, които преди това са били в Поколение 1, просто си остават там.
- Много вероятно е в Поколение 1 да има вече обекти, които междувременно са станали недостъпни и следователно подлежат на унищожение, но живеят, тъй като Поколение 1 не е проверено при това преминаване на GC.

- С течение на времето Поколение 1 бавно ще расте. Идва момент, когато след поредното почистване на Поколение 0, Поколение 1 достига своя праг. В този случай приложението просто ще продължи да работи, тъй като Поколение 0 току-що е било почистено, а новите обекти, се добавят в Поколение 0.
- Когато Поколение 0 следващият път достигне своя праг и GC се стартира, той едва сега ще провери размера на Поколение 1. Тъй като той е достигнал своя праг, GC този път ще почисти както Поколение 0, така и Поколение 1.
- Вижда се, че минават няколко почиствания на Поколение 0, преди да дойде времето и на поколение 1.

В резюме:

- Както видяхме, цикълът на живот на обект, нуждаещ се от финализация е интересен. Обектът умира, след това референция към него се добавя към **Freachable queue,** при което обектът се съживява, нещо повече, минава в Поколение 1, неговият **Finalize()** метод се изпълнява и указателят към него се изтрива от Freachable queue. В по-късен етап, но неизвестно кога (тъй като обектът е Поколение 1), GC просто ще освободи заетата от него памет.

Потискане на финализацията

- Извършва се с помощта на метода **GC.SuppressFinalize()**, който приема като параметър инстанция на тип. Общият вид на метода е:

```
public static void GC.SuppressFinalize(object o),
```

Изчакване до приключване на финализацията

- Извикването метод **GC.WaitForPendingFinalizers()** предизвиква обработване на всички финализатори, на маркираните за финализация обекти в резултат от извикването на метода **GC.Collect()**.
- Така ще се освободи памет, в резултат на унищожаване на неуправляваните ресурси, обвити в маркираните за финализация обекти. Разбира се, това ще доведе до изразходване на време за финализацията на обектите.

Недостатъци на финализацията

- Финализацията е неефективна - обектите, имащи деструктор, престояват в паметта значително по-дълго от останалите обекти (защото са от по-високо поколение). Освен това, представете си, че обектът G от горния пример държи референции към други обекти. Тези обекти (и обектите, реферирани пряко или непряко от тях) също ще се съживят и ще останат в паметта, докато живее и обект G. Това означава, че GC не може да освободи паметта в момента, когато установи, че обектите са отпадъци.
- **Finalize()** методите се изпълняват от отделна нишка на CLR. Това усложнява и натоварва допълнително процеса на GC. В тези методи трябва да се избягва сложен код, отнемащ много време.

- CLR не дава никакви гаранции за реда, в който ще се извикат отделните финализатори. Това означава, че не е безопасно във **Finalize()** метод да се обръщате към друг обект, поддържащ финализация, защото неговият финализатор може вече да е бил изпълнен и състоянието на обекта в този случай е непредвидимо. Също, не е безопасно да извиквате статични методи. Тези методи вътрешно могат да използват обекти, които вече са били финализирани и резултатите отново са непредсказуеми.

Какви са препоръките?

- След всички недостатъци на финализацията, какво трябва да се прави, когато се налага да освободим някакъв системен ресурс? ***Microsoft препоръчва използването на финализацията да става съвместно с имплементирането на интерфейса Idisposable!!!***

Интерфейсът IDisposable

- Интерфейсът **Idisposable** се препоръчва от Microsoft в тези случаи, в които искате да гарантирате моментално освобождаване на ресурсите, тъй като вече видяхме, че използването на **деструктор** не гарантира това.
- Използването на **IDisposable** се състои в имплементирането му от класа обвиващ някакъв неуправляван ресурс (файл, сокет, връзка към база данни и т.н..). Освобождаването на ресурса става с изрично извикване на метода Dispose() на интерфейса IDisposable.

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

Пример, имплементиране на IDisposable

```
...
// Обявяване на ресурса, например FileStream fs
FileStream fs = new FileStream("proba.txt",...);
try
{
   // Използваме ресурса fs
}
finally
{  //освобождаваме ресурса.
      ((IDisposable)fs).Dispose();
 //fs.Close() – методът Close() също извиква
   Dispose()
}
```