Поколения памет

Поколенията (generations) са механизъм в garbage collector, чиято единствена цел е подобряването на производителността. Основната идея е, че почистването на част от динамичната памет винаги е по-бързо от почистването на цялата памет. Вместо да обхожда всички обекти от хийпа, garbage collector обхожда само част от тях, класифицирайки ги по определен признак. В основата на механизма на поколенията стоят следните предположения:

- колкото по-нов е един обект, толкова по-вероятно е животът му да е кратък. Типичен пример за такъв случай са локалните променливи, които се създават в тялото на даден метод и излизат от обхват при неговото напускане.
- 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.

Много изследвания потвърждават валидността на изброените твърдения за голям брой съществуващи приложения. Нека разгледаме по-подробно поколенията памет и това как те се използват за оптимизация на производителността на .NET garbage collector.

Поколение 0

Когато приложението се стартира, първоначално динамичната памет не съдържа никакви обекти. Всички обекти, които се създават, стават част от Поколение 0. Казано накратко Поколение 0 съдържа новосъздадените обекти – тези, които никога не са били проверявани от garbage collector. При инициализацията на CLR се определя праг за размера на Поколение 0.Да предположим, че приложението иска да създаде нов обект, F. Добавянето на този обект би предизвикало препълване на Поколение 0. В този момент трябва да започне събиране на отпадъци и се стартира garbage collector.

Почистване на Поколение 0

Garbage collector процедира по описания по-горе алгоритъм и установява че обекти B и D са отпадъци. Тези обекти се унищожават и оцелелите обекти A, C и E се пренареждат в долната (или лява) част на managed heap. Динамичната памет непосредствено след приключването на събирането на отпадъци изглежда по следния начин:

Сега оцелелите при преминаването на garbage collector обекти стават част от Поколение 1 (защото са оцелели при едно преминаване на garbage collector). 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. Managed heap сега изглежда по следния начин:

Добавянето на нов обект J, би предизвикало препълване на Поколение 0, така че отново трябва да се стартира събирането на отпадъци. Когато garbage collector се стартира, той трябва да реши кои обекти от паметта да прегледа. Както Поколение 0, така и Поколение 1 има праг за своя размер, който се определя от CLR при инициализацията. Този праг е по-голям от този на Поколение 0. Да предположим че той е 2MB.

В случая Поколение 1 не е достигнало прага си, така че garbage collector ще прегледа отново само обектите от Поколение 0. 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, garbage collector почиства отново Поколение 0, оцелелите обекти преминават в Поколение 1, а тези, които преди това са били в Поколение 1, просто си остават там.

Забележете, че обект C, който междувременно е станал недостъпен и следователно подлежи на унищожение, в този случай остава в динамичната памет, тъй като е част от Поколение 1 и не е проверен при това преминаване на garbage collector.

Следващата фигура показва състоянието на динамичната памет след това почистване на Поколение 0.

Както вероятно се досещате, с течение на времето Поколение 1 бавно ще расте. Идва момент, когато след поредното почистване на Поколение 0, Поколение 1 достига своя праг от 2 MB. В този случай приложението просто ще продължи да работи, тъй като Поколение 0 току-що е било почистено и е празно. Новите обекти, както винаги, ще се добавят в Поколение 0.

51.NET Framework и системата за управление на общи типове. Типовете в CLR.
CLR поддържа много езици за програмиране. За да се осигури съвмес-тимост на данните между различните езици е разработена общата система от типове (Common Type System – CTS). CTS дефинира поддържаните от CLR типове данни и операциите над тях. Всички .NET езици използват типовете от CTS. За всеки тип в даден .NET език има някакво съответствие в CTS, макар че понякога това съответ-ствие не е директно. Обратното не е вярно – съществуват CTS типове, които не се поддържат от някои .NET езици. По идея всички езици в .NET Framework са обектно-ориентирани. Common Type System също се придържа към идеите на обектно-ориентираното програмиране (ООП) и по тази причина описва освен стандартните типове (числа, символи, низове, структури, масиви) и някои типове данни свър-зани с ООП (например класове и интерфейси).
Типовете данни в CTS биват най-разнообразни:
- примитивни типове (primitive types – int, float, bool, char, …)
- изброени типове (enums)
- класове (classes)
- структури (structs)
- интерфейси (interfaces)
- делегати (delegates)
- масиви (arrays)
- указатели (pointers)

Всички тези типове повече или по-малко вече са ни познати от езика C#, но всъщност те са част от CTS. Езикът C# и другите .NET езици използват CTS типовете и им съпоставят запазени думи съгласно своя синтаксис. Например типът **System.Int32** от CTS съответства на типа **int** в C#, а типът **System.String** – на типа **string**.

52.Стойностни типове. Стандартни и user-defined.

Стойностни и референтни типове

В CTS се поддържат две основни категории типове: **стойностни типове** (value types) и **референтни типове** (reference types). Стойностните типове съдържат директно стойността си в стека за изпълнение на прог-рамата, докато референтните типове съдържат строго типизиран указател (референция) към стойността, която се намира в динамичната памет. По- нататък ще разгледаме подробно разликите между стойностните и рефе- рентните типове и особеностите при тяхното използване.

Стойностни типове (value types)

Стойностни типове (типове по стойност) са повечето примитивни типове (**int**, **float**, **bool**, **char** и др.), структурите (**struct** в C#) и изброените типове (**enum** в C#).

Стойностните типове директно съдържат стойността си и се съхраняват физически в работния стек за изпълнение на програмата. Те не могат да приемат стойност **null**, защото реално не са указатели.

Стойностните типове и паметта

Стойностните типове заемат необходимата им памет в стека в момента на деклариране им и я освобождават в момента на излизане от обхват (при достигане на края на програмния блок, в който са декларирани). Заде-лянето и освобождаване на памет за стойностен тип реално се извършва чрез единично преместване на указателя на стека и следователно става много бързо.

Горното обяснение е малко опростено. Всъщност ако стойностен тип има за член-данни само стойностни типове, при инстанциране целият тип ще се задели в стека. Ако, обаче, стойностен тип (например структура) съдържа като член-данни референтни типове, стойностите им ще се запи-шат в динамичната памет.

Стойностните типове наследяват System.ValueType

CLR се грижи всички стойностни типове да наследяват системния тип **System.ValueType**. Всички типове, които не наследяват **ValueType** са референтни типове, т.е. реално са указатели към динамичната памет (адреси в паметта).

Предаване на стойностни типове

При извикване на метод стойностните типове се подават по стойност, т.е. предава се копие от тях. При подготовка на извикването на метод CLR копира подаваните като параметри стойностни типове от оригиналното им местоположение в стека на ново място в стека и подава на извиквания метод направените копия. Ако извикваният метод промени стойността на подадения му по стойност параметър, при връщане от извикването про-мяната се губи. Това поведение важи, разбира се, само ако параметрите се подават по подразбиране, без да се използват ключовите думи в C# **ref** и **out**, които ще разгледаме по-нататък в следващите теми.

53. Елементи на типа: методи, събития, полета, properties. Примери.

В структурите може да се съдържат:

1. методи (static – могат да се викат дори без да има инстанция от типа и instance- викани само над дефинирана инстанция от типа). За тях е недопустимо дефиниране на default constructor method.

```
using System;
namespace ValueTypeMethods
 { struct Sample
        { public static void SayHelloType()
           {Console.WriteLine("Hello from type");}
          public void SayHelloInstance()
           {Console.WriteLine("Hello from instance");}

          static void Main(string[] args)
//стартовата точка е статичен метод,независимо дали на value или на
// reference type. Може да не се казва main(), макар в C# да носи това име.
           { SayHelloType();
             Sample s = new Sample()
             s.SayHelloInstance();
           }
        }
 }
```

2. Поле на типа -fields (static or instance). Има тип и име

3. properties(static or instance). "Логически полета" с тип, име и набор методи,управляващи достъпа до тях.Компилаторът избира подходащия метод.

```
using System;
namespace ValueType
{   struct Point
  {private int xPosition, yPosition;
   public int X
        {       get {return xPosition;}         // генерира се метод get_X
                set {xPosition = value;}        // генерира метод set_X
        }
   public int Y
        {       get {return yPosition;}
                set {yPosition = value;}
        }
   }
   class EntryPoint
    {   static void Main(string[] args)
        {       Point p = new Point();   // не в heap, а в стек, защото "p" е value type
                p.X = 44;                //компилаторът генерира повикване на set_X
                p.Y = 55;
                Console.WriteLine("X: {0}", p.X);
                Console.WriteLine("Y: {0}", p.Y);
        }
    }
}
```

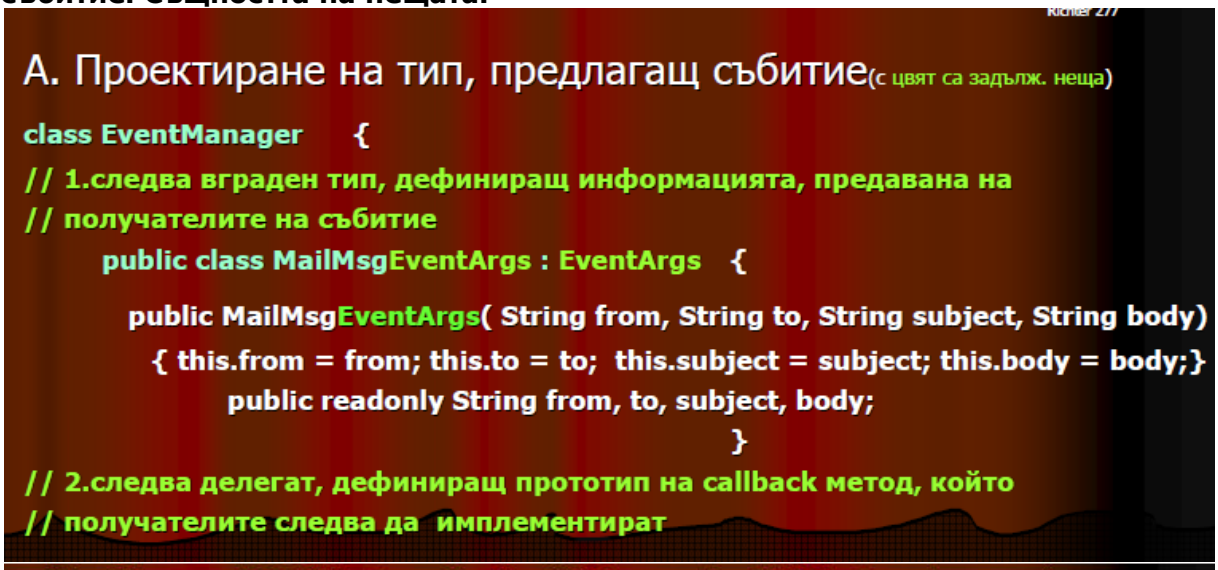
54.Събития. Кратък пример.

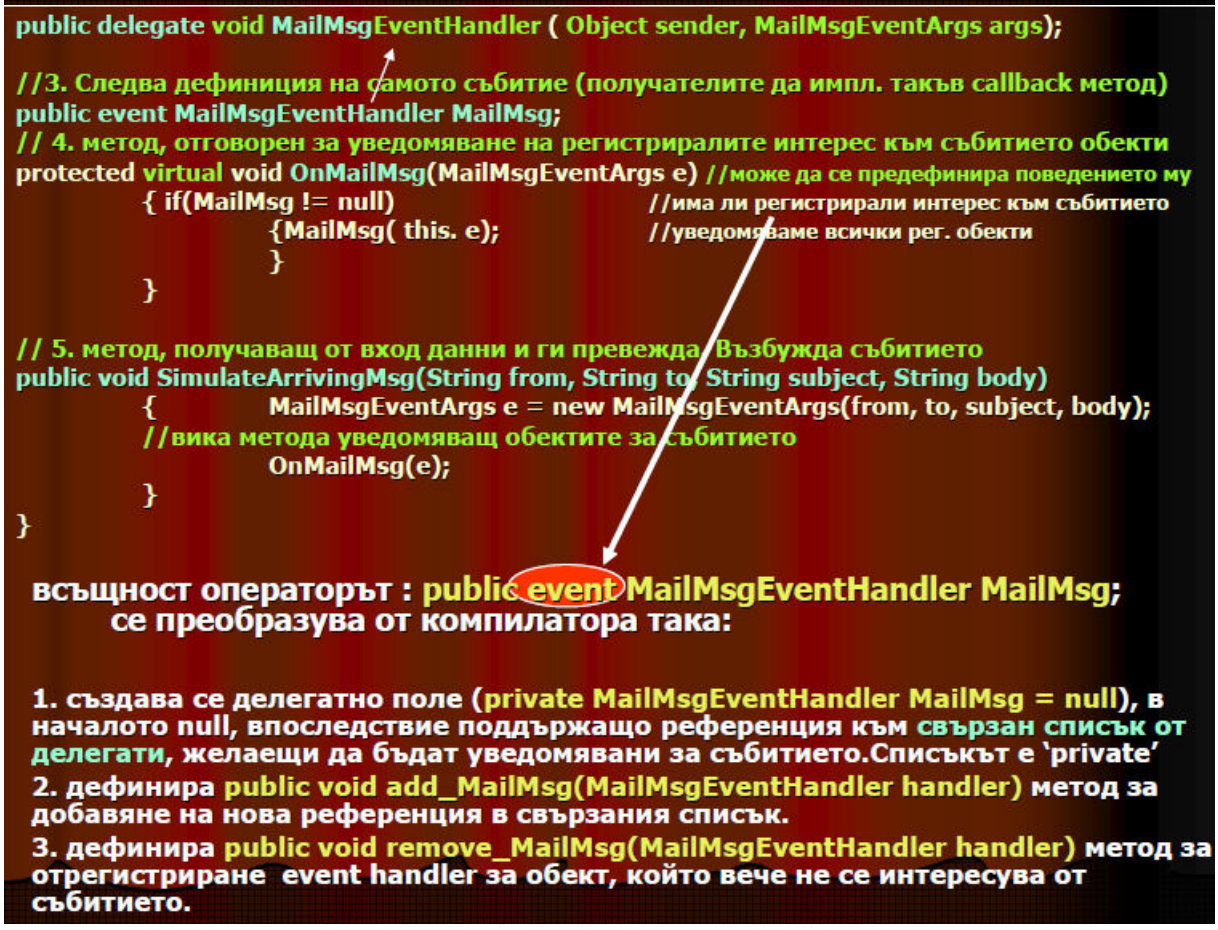
Събитията могат да се разглеждат като съобщения за настъпване на някакво действие. В компонентно-ориентираното програмиране компонен-тите изпращат събития (events) към своя притежател за да го уведомят за настъпването на интересна за него ситуация. Този модел е много характе-рен например за графичните потребителски интерфейси, където контро-лите уведомяват чрез събития други класове от програмата за действия от страна на потребителя. Например, когато потребителят натисне бутон, бутонът предизвиква събитие, с което известява, че е бил натиснат. Раз-бира се, събития могат да се предизвикват не само при реализиране на потребителски интерфейси. Нека вземем за пример програма, в която като част от функционалността влиза трансфер на файлове. Приключването на трансфера на файл може да се съобщава чрез събитие.

Изпращачи и получатели
Обектът, който предизвиква дадено събитие се нарича **изпращач на събитието (event sender)**. Обектът, който получава дадено събитие се нарича **получател на събитието (event receiver)**. За да могат да получават дадено събитие, получателите му трябва преди това да се абонират за него (subscribe for event).
За едно събитие могат да се абонират произволен брой получатели. Изпращачът на събитието не знае кои ще са получателите на събитието, което той предизвиква. Затова чрез механизма на събитията се постига по-ниска степен на свързаност (coupling) между отделните компоненти на програмата.
Събитията в .NET Framework
В компонентния модел на .NET Framework абонирането, изпращането и получаването на събития се поддържа чрез делегати и събития. Реализа-цията на механизма на събитията е едно от главните приложения на делегатите. Класът, който публикува събитието, дефинира делегат, който абонатите на събитието трябва да имплементират. Когато събитието бъде предизвикано, методите на абонатите се извикват посредством делегата. Тези методи обикновено се наричат **обработчици** на събитието. Делега-тът е multicast делегат, за да могат чрез него да се извикват много обра-ботващи методи (на всички абонати).

Извикването на събитие може да стане само в класа, в който то е дефинирано. Това означава, че само класът, в който се дефинира събитие, може да предизвика това събитие. Това е наложително, за да се спази шаблонът на Публикуващ/Абонати – абонираните класове се информират при промяна на състоянието на публикуващия и именно публикуващият е отговорен за разпращане на съобщенията за промяната, настъпила у него.

55.Проектиране на тип, предлагащ събитие. Проектиране на тип, използващ събитие. Същността на нещата.





4. Към методите от 2. и 3. са добавени атрибути за синнхронизация т.е. те са нишково обезопасени и много слушатели могат да работят едновременно с тях.

5. Методите са public, защото и събитието е било декларирано public

6. в метаданните се добавя описанието за event, типа делегат, методите add и remove

- Б. проектиране на тип, слушащ за събитие

```
class Object1
{    // подаване като параметър в конструктора на обекта със събитието EventManager
     public Object1(EventManager mm)
      {  // добавяме референция към списъка слушатели на събитието MailMsg  (сега това е callback метод
        // с име Object1Msg и имащ същата сигнатура като създадения  в класа EventManager
            // делегатен тип – MaiMsgEventHandler
        mm.MailMsg += new EventManager.MailMsgEventHandler(Object1Msg);
            // конструира се делегатен обект, обвиващ сега метода Object1Msg като  се вика
            // mm.add_EventManager(new EventManager.MailMsgEventHandler(Object1Msg)) за регистрация

      }
    // следва описание на callback  метода , който EventManager ще извика при събитието
    private void Object1Msg( Object sender, EventManager.MailMsgEventArgs e)
            {.....}

     public void Unregister( EventManager mm)
            {// конструираме инстанция на MailMsgEventHandler делегата, реферираш callback метода
             // Object1Msg  и го отрегистрираме като елемент от списъка.
             //  C# не допуска директно викане на add и remove, но от езици без събития – е възможно
              EventManager.MailMsgEventHandler callback =
                                                  new EventManager.MailMsgEventHandler(Object1Msg);
              mm.MailMsg -= callback;          //вика mm.remove_MailMsg(callback)
            }
}
```

56.Пакетирани типове (boxed types). Проблеми с достъпа.
Стойностните типове се съхраняват в стека на прило-жението и не могат да приемат стойност **null**, докато референтните типове съдържат указател (референция) към стойност в динамичната памет и могат да бъдат **null**.
Понякога се налага на референтен тип да се присвои обект от стойностен тип. Например може да се наложи в **System.Object** инстанция да се запише **System.Int32** стойност. CLR позволява това благодарение на т. нар. "**опаковане**" на стойностните типове (**boxing**).
В .NET Framework стойностните типове могат да се използват без преобразуване навсякъде, където се изискват референтни типове. При нужда CLR опакова и разопакова стойностните типове автоматично. Това спестява дефинирането на обвиващи (wrapper) класове за примитивните типове, структурите и изброените типове, но разбира се, може да доведе и до някои проблеми, които ще дискутираме по-късно.
Опаковане (boxing) на стойностни типове
Опаковането (boxing) е действие, което преобразува стойностен тип в рефере
нция към опакована стойност. То се извършва, когато е необхо-димо да се преобразува стойностен тип към референтен тип, например при преобразуване на **Int32** към **Object**: int i = 5;
object obj = i; // i се опакова

Особености при опаковането и разопаковането
При използване на автоматично опаковане и разопаковане на стойности трябва да се имат предвид някои особености:
- Опаковането и разопаковането намаляват производителността. За оптимална производителност трябва да се намали броят на опако- ваните и разопакованите обекти.

- Опакованите типове са копия на оригиналните стойности, поради което, ако променяме оригиналния неопакован тип, опакованото копие не се променя.

При работа с опаковани обекти трябва да се внимава, защото ако не бъдат съобразени някои особености, може да се наблюдава странно пове-дение на програмата . Основната причина за този резултат е фактът, че при преобразуване към интерфейс структурите се опаковат и съответно се създава копие на данните, намиращи се в тях. Опаковането е съвсем в реда на нещата, като се има предвид, че структурите са стойностни типове, а интерфейсите са референтни типове.

57.Референтни типове.
Референтни типове (типове по референция) са указателите, класовете, интерфейсите, делегатите, масивите и опакованите стойностни типове. Физически референтните типове представляват указател към стойност в динамичната памет, но за CLR те не са обикновени указатели, а специ-ални типово-обезопасени указатели. Това означава, че CLR не допуска на един референтен тип да се присвои стойност от друг референтен тип, който не е съвместим с него (т.е. не е същия тип или негов наследник). В резултат на това в .NET езиците грешките от неправилна работа с типове са силно намалени.
Референтните типове и паметта
Всички референтни типове се съхраняват в **динамичната памет** (т. нар. **managed heap**), която се контролира от системата за почистване на паметта (garbage collector). Динамичната памет е специално място от паметта, заделено от CLR за съхранение на данни, които се създават динамично по време на изпълнението на програмата. Такива данни са инстанциите на всички референтни типове.
Когато инстанция на референтен тип престане да бъде необходима на програмата, тя се унищожава от системата за почистване на паметта (т. нар. garbage collector).
Когато инстанцираме референтен тип с оператора **new**, CLR заделя място в динамичната памет, където ще стоят данните и един указател в стека, който съдържа адреса на заделеното място. Веднага след това заделената памет се занулява (освен ако програмистът не инициализира заделената променлива, например чрез извикване на подходящ конструктор).
Ако референтен тип (например клас) съдържа член-данни от стойностен тип, те се съхраняват в динамичната памет. Ако референтен тип съдържа член-данни от референтен тип, в динамичната памет се заделят указатели (референции) за тях, а техните стойности (ако не са **null**) също се заде-лят също в динамичната памет, но като отделни обекти.

Референтните типове и производителността
Понякога се приема, че заделянето на динамична памет е бърза операция, защото в текущата реализация (.NET Framework 1.1) физически се импле-ментира чрез преместване на един указател. Освобождаването на памет, обаче, е сложна и времеотнемаща операция, която се извършва от време на време от системата за почистване на паметта (garbage collector).
Ако изчислим средното време, необходимо за заделяне и освобождаване на динамична памет, се оказва, че заделянето и освобождаване на стой-ностните типове е значително по-бързо от референтните типове. Когато производителността е важна за нашата система, трябва да се съобразя-ваме с особеностите на стойностните и референтните типове и начина, по който те заделят и освобождават памет.
Глобално погледнато, нещата около управлението на динамичната памет в .NET Framework са доста комплексни, но в тази тема няма да се спираме на тях. По-нататък, в темата за управление на паметта и ресурсите, ще им обърнем специално внимание.

58.Интерфейсни типове. Използване на интерфейси със стойностни типове
Интерфейсът IComparable
Често пъти освен за равенство е необходимо обектите да се сравняват спрямо някаква подредба (например лексикографска за низове или по големина за числови типове). В .NET Framework типовете, които могат да бъдат сравнявани един с друг, трябва да имплементират интерфейса **System.IComparable**.
Интерфейсът дефинира един-единствен метод – **CompareTo(object)**. Този метод трябва да реализира сравняването и да връща:
- **число < 0** – ако подаденият обект е по-голям от **this** инстанцията

- **0** – ако подаденият обект е равен на **this** инстанцията

- **число > 0** – ако подаденият обект е по-малък от **this** инстанцията

IComparable се използва от .NET Framework при сортиране на масиви и колекции и при някои други операции, изискващи сравнение по големина.
Системни имплементации на IComparable
IComparable е имплементиран от много системни .NET типове, като например от примитивните стойностни типове **System.Char**, **System.Int32**, **System.Single**, **System.Double**, от символните низове (**System.String**) и от изброените типове (**System.Enum**). Това улеснява разработчиците при всекидневната им работа и често пъти им спестява излишни усилия.

Интерфейсите IEnumerable и IEnumerator
В програмирането се срещат типове, които съдържат много на брой ин-станции на други типове. Такива типове се наричат **контейнери** или още **колекции**. Колекции например са масивите, защото съдържат много на брой еднакви елементи.
Често пъти се налага да се обходят всички елементи на дадена колекция. За да става това по стандартен начин, в .NET Framework са дефинирани интерфейсите **IEnumerable** и **IEnumerator**.
Интерфейсът IEnumerable
Интерфейсът **System.IEnumerable** се имплементира от колекции и други типове, които поддържат операцията "обхождане на елементите им в някакъв ред". Този интерфейс дефинира само един метод – методът **GetEnumerator()**. Той връща итератор (инстанция на **IEnumerator**) за обхождане на елементите на дадения обект.
Обектите, поддържащи **IEnumerable** интерфейса, могат да се използват от конструкцията **foreach** в C# за обхождане на всичките им елементи.
Интерфейсът **IEnumerable** е реализиран от много системни .NET типове, като **System.Array**, **System.String**, **ArrayList**, **Hashtable**, **Stack**, **Queue**, **SortedList** и др. с цел да се улесни работата с тях.

Интерфейсът IEnumerator
Интерфейсът **System.IEnumerator** имплементира обхождане на всички елементи на колекции и други типове. Той реализира прост итератор чрез следните методи и свойства:
- Свойство **Current** – връща текущия елемент.

- Метод **bool MoveNext()** – преминава към следващия елемент и връща **true**, ако той е валиден.

- Метод **Reset()** – премества итератора непосредствено преди първия елемент (установява го в начално състояние).

59. Проблеми при присвояване и съвместимост на типовете.

Макар C# да не инициализира автоматично локалните променливи, компилаторът предупреждава за неправилното им използване. Например следният код ще предизвика грешка при опит за компилация:

```
int value;
value = value + 5;
```

Преобразуването на типове също е безопасно. CLR не позволява да се извърши невалидно преобразуване на типове – да се преобразува променлива от даден тип към променлива от тип, който не е съвместим с първия. При опит да бъде направено това, възниква изключение.

Неявното преобразуване на типове е разрешено само за съвместими типове, когато не е възможна загуба на информация. При явно преобразуване на типове, ако те не са съвместими, се хвърля InvalidCastException по време на изпълнение. Например следният код предизвиква изключение по време на изпълнение:

60.Делегати. Дефиниране, използване.
Делегатите са референтни типове, които описват сигнатурата на даден метод (броя, типа и последователността на параметрите му) и връщания от него тип. Могат да се разглеждат като "обвивки" на методи - те представляват структури от данни, които приемат като стойност методи, отговарящи на описаната от делегата сигнатура и връщан тип.
Делегатът се инстанцира като клас и методът се подава като параметър на конструктора. Възможно е делегатът да сочи към повече от един метод, но на това ще се спрем подробно малко по-нататък. Съществува известна прилика между делегатите и указателите към функ-ции в други езици, например Pascal, C, C++, тъй като последните пред-ставляват типизиран указател към функция. Делегатите също съдържат силно типизиран указател към функция, но те са и нещо повече – те са напълно обектно-ориентирани. На практика делегатите представляват класове. Инстанцията на един делегат може да съдържа в себе си както инстанция на обект, така и метод.
Едно от основните приложения на делегатите е реализацията на "обратни извиквания", т.нар. callbacks. Идеята е да се предаде референция към метод, който да бъде извикан по-късно. Така може да се осъществи например асинхронна обработка – от даден код извикваме метод, като му подаваме callback метод и продължаваме работа, а извиканият метод извиква callback метода когато е необходимо. Със средствата на делега-тите е възможно даден клас да позволи на потребителите си да предос-тавят метод, извършващ специфична обработка, като по този начин обработката не се фиксира предварително.Делегатите в .NET Framework са специални класове, които наследяват **System.Delegate** или **System.MulticastDelegate**. От тези класове обаче явно могат да наследяват само CLR и компилаторът. Всъщност, те не са от тип делегат – тези класове се използват, за да се наследяват от тях типове делегат.

Всеки делегат има **"списък на извикване" (invocation list)**, който представлява наредено множество делегати, като всеки елемент от него съдържа конкретен метод, реферирал от делегата. Делегатите могат да бъдат единични и множествени.

Единичните делегати наследяват класа **System.Delegate**. Тези делегати извикват точно един метод. В списъка си на извикване имат единствен елемент, съдържащ референция към метод.
ножествени (multicast) делегати
Множествените делегати наследяват класа **System.MulticastDelegate**, който от своя страна е наследник на класа на **System.Delegate**. Те могат да викат един или повече метода. Техните списъци на извикване съдър-жат множество елементи, всеки реферирал метод. В тях може един и същ метод да се среща повече от веднъж. При извикване делегатът активира всички реферирани методи. Множествените делегати могат да участват в комбиниращи операции.
Езикът C# съдържа запазената дума **delegate**, чрез която се декларира делегат. При тази декларация компилаторът автоматично наследява типа **MulticastDelegate**., т.е. създава множествен делегат. Затова ще обърнем по-голямо внимание именно на този вид делегати.

Мета-данните описват различни характеристики на асемблитата и съдържимото в тях:
- име на асемблито (например **`System.Windows.Forms`**)

- версия, състояща се от 4 числа (например `1.0.5000.0`)

- локализация, описваща език и култура (например неутрална или en-US или bg-BG)

- цифров подпис на създателя (незадължителен)

- изисквания към правата за изпълнение

- зависимости от други асемблита (описани с точното име и версия)

- експортирани типове

- списък със дефинираните класове, интерфейси, типове, базови класове, имплементирани интерфейси и т.н.

- списък с дефинираните ресурси

Освен тези данни за всеки клас, интерфейс или друг тип, който е дефиниран в асемблито, се съдържа и следната информация:
- описание на член-променливите, свойствата и методите в типовете

- описание на параметри на методите, връщана стойност на метода за всеки метод

- приложени атрибути към асемблито, методите и другите елементи от кода

Асемблитата в .NET Framework

Асемблитата са основна съставна част на всеки софтуерен продукт, базиран на .NET Framework. Те са най-малката и основна част при разпространение на .NET приложения. Асемблитата се състоят от компилирани .NET типове (интерфейси, класове, структури и др.), метаданни и ресурси (**.bmp**, **.jpeg**, **.ico** файлове, **.resource** и **.resx** ресурси и други). Компилираните типове представляват изпълним програмен код във вид на инструкции на междинния език IL. Метаданните описват асемблитата и типовете в тях. Ресурсите могат да бъдат вградени или записани като външни файлове.
Асемблитата могат да бъдат статични и динамични. Статичните асемблита се съхраняват във файл в portable executable (PE) формат, докато динамичните се изпълняват директно от паметта и не се записват (във файл) преди изпълнението им .NET Framework предлага стандартни средства и инструменти за създаване на динамични асемблита и позволява тяхното изпълнение и съхранение с помощта на класовете от пространството **System.Reflection.Emit**.

Метаданни и манифест на асембли

Всяко асембли, независимо дали е статично или динамично, съдържа в себе си метаданни (информация, която го описва).
Метаданните включват описание на съдържаните в асемблито типове и информация за него самото.

Манифест на асембли

Информацията за асемблито описва как са свързани съдържаните елементи помежду си – това е т. нар. **манифест**. Манифестът съдържа всички метаданни, нужни за описанието на идентичността на асемблито, информация за неговата версия, необходимите му права, асемблитата и версиите им, нужни за изпълнението му, както и допълнителна информация, необходима за извличането на типовете и ресурсите.
Манифестът може да се съдържа в самото асембли (в неговия **.exe** или **.dll** преносим изпълним файл) заедно с останалите ресурси или като самостоятелен файл, който съдържа само информацията на манифеста. Следващата илюстрация показва различните начини, по които се съхранява манифеста в асемблитата.

При асемблита, които съдържат един файл, манифестът е вмъкнат в PE файла и образува асембли от един файл. Възможно е създаването на многомодулно асембли с външен манифест или манифестът може да е вмъкнат в един от файловете.