2. Въведение в .NET среда. .NET framework архитектура. Поддържане на единна езикова среда.

Microsoft дефинират платформата .NET като съвкупност от технологии..NET платформата осигурява стандартизирана инфраст
използване, хостинг и интеграция на .NET приложения и XML уеб услуги, базирана на .NET сървърите на Microsoft, средствата

Компоненти на .NET Framework

Можем да разделим .NET Framework на два основни компонента:

- **Common Language Runtime (CLR)** – средата, в която се изпълнява управляваният код на .NET приложенията. Представлява
контролирано изпълнява .NET кода и осигурява различни услуги, като управление на сигурността, управление на паметта и др.

- **Framework Class Library (FCL)** – представлява основната библиотека от типове, които се използват при изграждането на .NE
основната функционалност за разработка, необходима за повечето приложения, като вход/изход, връзка с бази данни, работа с
приложения, използване на уеб услуги, изграждане на графичен потребителски интерфейс и др.

Архитектура на .NET Framework

Архитектурата на .NET Framework често пъти се разглежда на нива, както това е :

Операционна система

Операционната система управлява ресурсите, процесите и потребителите на машината. Тя предоставя и някои услуги на прило
COM+, MSMQ, IIS, WMI и други.

Средата, която изпълнява .NET приложенията (CLR), е обикновен процес в операционната система и се управлява от нея, както

Common Language Runtime

Общата среда за изпълнение Common Language Runtime (CLR) управлява процеса на изпълнение на .NET код. Тя се грижи за з
даване на паметта, управлява конкурентността, грижите за сигурността на приложенията и изпълнява други важни задачи, свър
кода.

Base Class Library

BCL представлява богата обектно-ориентирана библиотека с основни кла-сове, които осигуряват базова системна функционал
изход, работа с колекции, символни низове, мрежови ресурси, сигурност, отдалечено извикване, многонишковост и др.

ADO.NET и XML

Слоят на ADO.NET и XML предоставя удобен начин за работа с релационни и други бази от данни и средства за обработка на X

Езици за програмиране

.NET Framework позволява на разработчика да използва различни езици за програмиране, както и да интегрира в едно приложе
разработвани на различни езици. Възможно е дори клас, написан на един език, да бъде наследен и разширен от клас, написан
Microsoft .NET Framework поддържа стандартно езиците C#, VB.NET, Managed C++ и J#.Съвместимостта на езиците за програм
се дължи на архитектурни решения, които ще разгледаме в детайли след малко.

Common Language Runtime

След като се запознахме накратко с архитектурата на .NET Framework, нека сега разгледаме в детайли и най-важният компонен
Common Language Runtime (CLR) е сърцето на .NET Framework. Той пред-ставлява **среда за контролирано изпълнение на уп**
практика CLR е тази част от .NET Framework, която изпълнява компили-раните .NET програми в специална изолирана среда.

В своята същност CLR представлява виртуална машина, която изпълнява инструкции, на езика IL (Intermediate Language), езикт
всички .NET езици. CLR е нещо като виртуален компютър, който обаче не изпълнява асемблерен код за процесор Pentium, AMD
код.

Управляван код

Управляваният код (managed code) е кодът, който се изпълнява от CLR. Той представлява поредица от IL инструкции, които с
компилацията на .NET езиците. По време на изпълнение управляваният код се компилира допълнително до машиннозависим к
платформа и след това се изпълнява директно от процесора.

Intermediate Language (IL)

Междинният език Intermediate Language (IL), е език за програмиране от ниско ниво, подобен на асемблерните езици. За разлика
много по-високо ниво, отколкото асемблерите за съвременните мик-ропроцесори.

IL е обектно-ориентиран език. Той разполага с инструкции за заделяне на памет, за създаване на обект, за предизвикване и обр
за извикване на виртуални методи и други инструкции, свързани с обектно-ориентираното програмиране.

11. Windows Forms в .NET. Контроли и йерархия на графичните конролите. Създаване на дъщерни форми и контроли. П

Windows Forms е стандартната библиотека на .NET Framework за изграж-
дане на прозоречно-базиран графичен потребителски интерфейс (GUI) за
настолни (desktop) приложения. Windows Forms дефинира набор от
класове и типове, позволяващи изграждане на прозорци и диалози с
графични контроли в тях, чрез които се извършва интерактивно взаимо-
действие с потребителя.Windows Forms е типична компонентно-ориентирана библиотека за създаване на GUI, която предостав
писане на програмен код да се създава гъвкав графичен потребителски интерфейс.

Контролите в Windows Forms

Windows Forms съдържа богат набор от стандартни контроли: форми,
диалози, бутони, контроли за избор, текстови полета, менюта, ленти с
инструменти, статус ленти и много други.

Наследяване на форми и контроли

Windows Forms е проектирана така, че да позволява лесно наследяване и
разширяване на форми и контроли. Това дава възможност за преизполз-

ване на общите части на потребителския интерфейс.

Наследяване на форми
Наследяването на форми позволява повторно използване на части от потребителския интерфейс. Чрез него е възможно да променим наведнъж общите части на много форми. За целта дефинираме една базова форма, която съдържа общата за всички наследници функционалност.
Базовата форма е най-обикновена форма. Единствената особеност е, че контролите, които могат да се променят от наследниците, се обявяват като **protected**. При наследяване на форма се наследява класът на базовата форма.Не всички класове от Windows Forms са ко обикновени .NET компоненти, например **Menu**, **Timer** и **ImageList**. Изглежда малко странно защо менюто не е контрола, но това компонентата **Menu** реално няма графичен образ и представлява списък от **MenuItem** елементи. **MenuItem** класът вече има гра следователно е контрола.

12. Опционални и списъчни контроли. Основни пропъртита и събития. Приложение – пример.
Контролите в Windows Forms са текстовите полета, етикетите, бутоните, списъците, дърветата, таблиците, менютата, лентите с лентите и много други. Windows Forms дефинира базови класове за контролите и класове-наследници за всяка контрола. Базов е класът **System.Windows.Forms.Control**. Пример за контрола е например бутонът (класът **System.Windows.Forms.Button**).Всяка контрола обработва собствените си **събития**Когато главната приложение получи съобщение,свързано с някоя от неговите форми, тя препраща съобщението до обработчика на съобщения Този обработчик от своя страна проверява дали съобщението е за самата форма или за някоя нейна контрола. Ако съобщение обработва директноот съответния обработчик на събития. Ако съобщението е за някоя от контролите във формата, то се преда която получи съобщението, може да е обикновена контрола или контейнер-контрола.Когато обикновена контрола получи съобщ директно. Когато контейнер-контрола получи съобщение, тя проверява дали то е за нея или е за някоя от вложените контроли. докато съобщението достигне до контролата, за която е предназначено.Класът **System.Windows.Forms.Form** е базов клас за в Forms GUI приложенията. Той представлява графична форма - прозорец или диалогова кутия, която съдържа в себе си контрол навигацията между тях. Повечето прозорци имат рамка и специални бутони за затваряне, преместване и други стандартни опер прозорците и стандартните контроли по тяхната рамка зависят от настройките на графичната среда на операционната система. Програмистът има само частичен контрол над външния вид на прозорците. Клас класовете **Control**, **ScrollableControl** и **ContainerControl** и наследява от тях цялата им функционалност, всичките им свойства, **CheckBox** е кутия за избор в стил "да/не". Свойството й **Checked** задава дали е избрана.
RadioButton е контрола за алтернативен избор. Тя се използва в групи. Всички **RadioButton** контроли в даден контейнер (напр една група и в нея само един **RadioButton** е избран в даден момент. **ListBox** контролата се използва за изобразяване на списъ които потребителят може да избира чрез щракване с мишката върху тях. По-важните свойства на тази контрола са:
- **Items** – колекция, която задава списъка от елементи, съдържащи се в контролата.
- **SelectionMode** – разрешава/забранява избирането на няколко елемента едновременно.
- **SelectedIndex**, **SelectedItem**, **SelectedIndices**, **SelectedItems** – връщат избрания елемент (или избраните елементи).

ComboBox представлява кутия за редакция на текст с възможност за dropdown алтернативен избор.
- **Text** – съдържа въведения текст.
- **Items** – задава възможните стойности, от които потребителят може да избира.
- **DropDownStyle** – задава стила на контролата – дали само се избира стойност от списъка или може да се въвежда ръчно и друга стойност.

13. Диалози – стандартни и потребителски. Видове и приложение. Пример за употреба.
При разработката на Windows Forms приложения често пъти се налага да извеждаме диалогови кутии с някакви съобщения или разгледаме стандартните средства за такива ситуации.
Стандартни диалогови кутии Класът MessageBox ни позволява да извеждаме стандартни диалогови кутии, съдържащи текст, бутони и икони:
- съобщения към потребителя
- въпросителни диалози

Показването на диалогова кутия се извършва чрез извикване на статичния метод Show(…) на класа MessageBox.
Следният код, например, ще покаже диалогова кутия със заглавие "Предупреждение" и текст "Няма връзка с интернет":
MessageBox.Show("Няма връзка с Интернет.", "Предупреждение");
Пример за стандартна диалогова кутия с малко повече функционалност:
bool confirmed =MessageBox.Show("Наистина ли ще изтриете това?",

"Въпрос", MessageBoxButtons.YesNo,
MessageBoxIcon.Question) == DialogResult.Yes;
Този код ще покаже диалогова кутия със заглавие "Въпрос" и текст "Наистина ли ще изтриете това?". Преди текста ще има икон
нея, а под него – бутони Yes и No. Ако потребителят
натисне Yes, променливата confirmed ще има стойност true, в противен случай ще има стойност false.
Извикване на диалогови кутии Освен стандартните диалогови кутии можем да използваме и потреби-
телски дефинирани диалогови кутии. Те представляват обикновени форми и се извикват модално по следния начин:
DialogResult result = dialog.ShowDialog();
Методът ShowDialog() показва формата като модална диалогова кутия. Типът DialogResult съдържа резултата (OK, Yes, No, Ca
на диалога. Задаването на DialogResult може да става автоматично, чрез свойството DialogResult на бутоните, или ръчно – пред
чрез свойството му DialogResult.

14. SDI и MDI приложения. Структура и пример.

MFC прави лесно да се работи едновременно с един документен интерфейсни (SDI) и няколко документни интерфейсни (MDI)
SDI приложенията позволяват само един отворен прозорец на документ кадър по кадър. MDI приложенията позволяват няколк
отворени в една и съща инстанция на приложението. Приложението MDI има прозорец, в който няколко MDI прозорци, които са
си, могат да бъдат отворени, всяка от които съдържа отделен документ. В някои приложения, детето прозорци могат да са от р
например прозорците на графиките и таблиците прозорци. В този случай, на лентата с менюта може да се променят MDI прозор
,които са активирани.
Осн. развичия в SDI и MDI прилож. Много изгледи в MDIпроблемът за синхронизация на промените в изглед.
Разлики между SDI и MDI:
1.В MDI има повече от един отворен документ, докато в SDI , за да отвори втори трябва да затворим първия.
2.В MDI могат да се поддържат различни типове документи.
3.При MDI в менюто има опция Windows за превключване на прозорците.
4.MDI има поне 2 менюта , а SDI има едно .Първото при отворен документ, а второто при затворен.
5.В SDI има една рамка, а в MDI има главна и дъщерна рамка.
Множество изгледи под един документ в MDI приложения
Използва се многодокументния шаблон- CMultiDocTemplate.Тъй като документа е същия не се създава нов документ, а само но
рамката си остават същите.MFC осигурява списъчна структура за обхождане на всички изгледи(като например при UpdateAllVie
Document, за да получи данни.Ако в някое View променим данните и искаме промените да се отразят в други View, налага се да
UpdateAllView, който пък вика OnUpdate на всеки View.Този метод не се вика автоматично-ние трябва да го осигурим. UpdateAll
ги обновява, като инвалидизира (Invalidate) целия прозорец.Ако искаме оптимизация ние трябва да променим реализацията на
да променим параметрите, с които се вика OnUpdate.
В SDI има един обект на приложението на документа.
Това означава , че за да отвори нов документ трябва да затворим стария.
В MDI имаме много документални обекти, затова не е необходимо припокриване.
При избиране на нов се създава нов обект.Ако е ….. Template то се взема от него документен шаблон и се създава , а ако са по
екрана от кой точно шаблон да се вземе.
Отваряне на съществуващ документ.

15. Свързване с база данни. Свързване на данни с контроли (Data Binding). DataGrid. Master-Details.Пример.

База от данни се нарича всяка организирана колекция от данни.
Свързване на данни
Свързването на данни (data binding) осигурява автоматично прехвърляне на данни между контроли и източници на данни. Може
масив, съдържащ имена на градове, с ComboBox контрола и имената от масива ще се показват в нея.Всички Windows Forms ко
свързване на данни (data binding). Можем да свържем което и да е свойство на контрола към източник на данни.
Контролата DataGrid
DataGrid контролата визуализира таблични данни. Тя осигурява навигация по редове и колони и позволява редактиране на дан
данни най-често се използват ADO.NET DataSet и DataTable. Чрез свойството DataSource се задава източникът на данни, а чре
DataMember – пътят до данните в рамките на източника. По-важни
свойства на контролата са:
- ReadOnly – разрешава / забранява редакцията на данни.
- CaptionVisible – показва / скрива заглавието.
- ColumnHeadersVisible – показва / скрива заглавията на колоните.
- RowHeadersVisible – показва / скрива колоната в ляво от редовете.
- TableStyles – задава стилове за таблицата.
o MappingName – задава таблицата, за която се отнася дефинира-
ният стил.
o GridColumnStyles – задава форматиране на отделните колони –
заглавие, ширина и др.
Master-Details навигация
Навигацията "главен/подчинен" (master-details) отразява взаимоотношения от тип едно към много (например един регион има м
Forms се поддържа навигация "главен/подчинен". За да илюстрираме работата с нея, нека разгледаме един пример: Имаме Da
таблици – едната съдържа имена на държави, а другата – имена на градове. Те са свързани помежду си така, че на всяка държ
съответстват определени градове от втората

таблица:
Тогава можем да използваме две DataGrid контроли – първата, визуализираща държавите, а втората, визуализираща градовет
текущо избраната държава от първата контрола. За целта контролите се свързват с един и същ DataSet. На главната контрола
данни главната таблица. На подчинената контрола се задава за източник на данни релацията на таблицата:

```
// Bind the master grid to the master table
DataGridCountries.DataSource = datasetCountriesAndTowns;
DataGridCountries.DataMember = "Countries";
// Bind the detail grid to the relationship
DataGridTowns.DataSource = datasetCountriesAndTowns;
DataGridTowns.DataMember = "Countries.CountriesTowns";
```

16. GDI+. Методи за изчертаване на основните фигури. Запълване, контур и специални ефекти.

Пакетът **System.Drawing** осигурява достъп до GDI+ функциите на
Windows:
- повърхности за чертане
- работа с графика и графични трансформации
- изчертаване на геометрични фигури
- работа с изображения
- работа с текст и шрифтове
- печатане на принтер

Той се състои от няколко пространства:
- **System.Drawing** – предоставя основни класове като повърхности,
моливи, четки, класове за изобразяване на текст.
- **System.Drawing.Imaging** – предоставя класове за работа с изобра-
жения, картинки и икони, класове за записване в различни файлови
формати и за преоразмеряване на изображения.
- **System.Drawing.Drawing2D** – предоставя класове за графични транс-
формации – бленди, матрици и др.
- **System.Drawing.Text** – предоставя класове за достъп до шрифтовете
на графичната среда.

- **System.Drawing.Printing** – предоставя класове за печатане на
принтер и системни диалогови кутии за печатане.

Класът Graphics

Класът **System.Drawing.Graphics** предоставя абстрактна повърхност за
чертане. Такава повърхност може да бъде както част от контрола на
екрана, така и част от страница на принтер или друго устройство.
Най-често чертането се извършва в обработчика на събитието **Paint**. В
него при необходимост се преизчертава графичния облик на контролата.
Параметърът **PaintEventArgs**, който се подава, съдържа **Graphics** обекта.
Graphics обект може да се създава чрез **Control.CreateGraphics()**. Той
задължително трябва да се освобождава чрез **finally** блок или с конс-
трукцията **using**, защото е ценен ресурс.

Чрез примера ще илюстрираме работата с GDI+ чрез пакета **System.Drawing –** чертане на геометрични фигури с четки и моли
текст със зададен шрифт.

```
private void MainForm_Paint(object sender,
System.Windows.Forms.PaintEventArgs e)
{
Graphics g = e.Graphics;
g.SmoothingMode = SmoothingMode.AntiAlias;
Brush brush = new SolidBrush(Color.Blue);
g.FillEllipse(brush, 50, 40, 350, 250);
brush.Dispose();
Pen pen = new Pen(Color.Red, 2);
g.DrawRectangle(pen, 40, 50, 200, 40);
pen.Dispose();
brush = new SolidBrush(Color.Yellow);
Font font = new Font("Arial", 14, FontStyle.Bold);
g.DrawString(".NET Framework", font, brush, 60, 60);
brush.Dispose();

font.Dispose();
```

}

17. Вход/Изход в .NET. Работа с файлове, директории, потоци, четци и писци.
Потоците в обектно-ориентираното програмиране са една абстракция, с която се осъществява вход и изход от дадена програма
концепция са аналогични на потоците в други обектно-ориентирани езици, напр. Java, C++ и Delphi (Object Pascal).
Потокът е подредена серия от байтове, която служи като абстрактен канал за данни. Този виртуален канал свързва програмата
съхранение или пренос на данни (напр. файл върху хард диск), като достъпът до канала е последователен. Потоците предоста
запис на поредици от байтове от и към устройството. Това е стандартният механизъм за извършване на входно-изходни опера
Framework.Потоците в .NET Framework се делят на две групи – **базови** и **преходни**. И едните, и другите, наследяват абстрактн
System.IO.Stream, базов за всички потоци.Базовите потоци пишат и четат директно от някакъв външен механизъм за съхранен
система (например класът **FileStream**), паметта (**MemoryStream**) или данни, достъпни по мрежата (**NetworkStream**). По-нататъ
FileStream в точката "Файлови потоци". Преходните потоци пишат и четат от други потоци (най-често в базови потоци), като пр
добавят допълнителна функци-оналност, например буфериране (**BufferedStream**) или кодиране (**CryptoStream**). По-подробно
BufferedStream в точката "Буферирани потоци". За четене на данни от поток се използва методът **int Read(byte[] buffer, int off**
най-много **count** на брой байта от текущата позиция на входния поток, увеличава позицията и връ-ща броя прочетени байтове
на потока. Четенето може да блокира за неопределено време. Например, ако при четене от мрежа извикаме метода **NetworkSt**
налични данни за четене, операцията блокира до тяхното получаване. В такива случаи е уместно да се използва свойството **Ne**
DataAvailable, което показва дали в потока има пристигнали данни, които още не са прочетени, т. е. дали последваща операци
ще върне резултат веднага.

Писане в поток

Методът **Write(byte[] buffer, int offset, int count)** записва в изход-ния поток **count** байта, като започва от зададеното отместван
тази операция е блокираща, т.е. може да предизвика забавяне за неопределено време. Не е гарантирано, че байтовете, записа
достигнали до местоназначението си след успеш-ното изпълнение на метода. Възможно е потокът да буферира данните и да н
веднага.Файловите потоци в .NET Framework са реализирани в класа **FileStream**, който вече беше използван в примера за пото
Stream, той поддържа всичките му методи и свойства (четене, писане, позициониране) и добавя някои допълнителни.Четенето
потоци, както и другите по-рядко използвани операции, се извършват както при всички наследници на класа **Stream** – с методит

Файловите потоци поддържат пряк достъп до определена позиция от файла чрез метода **Seek(…)**.

Четците и писачите (readers and writers) в .NET Framework са класове, които улесняват работата с потоците. При работа напри
програмистът може да чете и записва единствено байтове. Когато този поток се обвие в четец или писач, вече са позволени че
различни структури от данни, например примитивни типове, текстова информация и други типове. Четците и писачите биват дв
Двоичните четци и писачи осигуряват четене и запис на примитивни типове данни в двоичен вид – **ReadChar()**, **ReadChars()**, **R**
др. за четене и съответно **Write(char)**, **Write(char[])**, **Write(Int32)**, **Write(double)** – за запис. Може да се чете и записва и **string**,
вид на масив от символи и префиксно се записва дължината му – **ReadString()**, респ. **Write(string)**.Текстовите четци и писачи о
на текстова инфор-мация, представена във вид на низове, разделени с нов ред. Базови текстови четци и писачи са абстрактнит
TextWriter. Основните методи за четене и запис са следните: - **ReadLine()** – прочита един ред текст.

- **ReadToEnd()** – прочита всичко от текущата позиция до края на потока.

- **Write(…)** – вмъква данни в потока на текущата позиция.
Работа с директории. Класове Directory и DirectoryInfo
Класовете **Directory** и **DirectoryInfo** са помощни класове за работа с директории. Ще изброим основните им методи, като отбел
са статични, а за **DirectoryInfo** – достъпни чрез инстанция.
- **Create()**, **CreateSubdirectory()** – създава директория или подди-ректория.

- **GetFiles(…)** – връща всички файлове в директорията.

- **GetDirectories(…)** – връща всички поддиректории на директорията.

- **MoveTo(…)** – премества (преименува) директория.

- **Delete()** – изтрива директория.

- **Exists()** – проверява директория дали съществува.

- **Parent** – връща горната директория.

- **FullName** – пълно име на директорията.

20. Таймер (Timer). Работа с таймери. Пропъртита, събития. Пример. Таймери

Често в приложенията, които разработваме, възниква необходимост от изпълняване на задачи през регулярни времеви интервали. Таймерите предоставят такава услуга. Те са обекти, които известяват приложението при изтичане на предварително зададен интервал от време. Таймерите са полезни в редица сценарии, например, когато искаме да обновяваме периодично потребителския интерфейс с актуална информация за статуса на някаква задача или да проверяваме състоянието на променящи се данни.

System.Timers.Timer

Класът предоставя събитие за изтичане на времевия интервал **Elapsed**,

което е делегат от тип **ElapsedEventHandler**, дефиниран като:

```
public delegate void ElapsedEventHandler(
object sender, ElapsedEventArgs e);
```

При изтичане на интервала, указан в свойството **Interval**, таймерът от тип **System.Timers.Timer** ще извика записалите се за събитието методи, използвайки нишка от пула. Ако използваме един и същ метод за получаване на събития от няколко таймера, чрез аргумента **sender** можем да ги разграничим. Класът **ElapsedEventArgs** чрез свойството **DateTime SignalTime** ни предоставя точното време, когато е бил извикван метода. За стартиране и спиране на известяването, можем да извикаме съответно **Start()** и **Stop()** методите. Свойството **Enabled** ни позволява да инструктираме таймера да игнорира събитието **Elapsed**. Това прави **Enabled** функционално еквивалентно на съответните **Start()** и **Stop()** методи. Когато приключим с таймера, трябва да извикаме **Close()**, за да освободим съответните системни ресурси.

System.Threading.Timer

System.Threading.Timer прилича на **System.Timers.Timer** и също използва пула с нишки. Основната разлика е, че той позволява малко по-разширен контрол – може да указваме кога таймера да започне да отброява, както и да предаваме всякаква информация на метода за обратни извиквания чрез обект от произволен тип. За да ползваме **System.Threading.Timer**, трябва в конструктора му да подадем делегат от тип **TimerCallback**, дефиниран като:

```
public delegate void TimerCallback(object state);
```

При всяко изтичане на времевия интервал, ще бъдат извиквани методите в този делегат. Обикновено като обект за състояние има полза да подаваме създателя на таймера, за да можем да използваме същия метод за обратни извиквания за обработка на събития от м
Другият параметър в конструктора на таймера е времевият интервал. Той може и да бъде променен впоследствие с извикване на **Change(…)** метода. **System.Threading.Timer** не предлага удобен начин за стартиране и спиране. Неговата работа започва веднага след конструирането му (точно след изтичането на подаденото стартово време) и прекъсването му става само чрез **Dispose()**. Ако искаме да го рестартираме трябва да създадем нов обект.

System.Windows.Forms.Timer

Пространството от имена **System.Windows.Forms** съдържа още един клас за таймер, който е със следната дефиниция:

```
public class Timer : Component, IComponent, Idisposable
{
public Timer();
public bool Enabled{virtual get ; virtual set;}
public int Interval {get; set;}
public event EventHandler Tick;
public void Start();
public void Stop();
```

}
Въпреки, че методите на **System.Windows.Forms.Timer** много приличат на тези на **System.Timers.Timer**, то **System.Windows.Forms.Timer** не използва пула с нишки за обратните извиквания към Windows Forms приложението. Вместо това, през определено време той пуска Windows съобщението **WM_TIMER** в опашката за съобщения на текущата нишка.
Използването на **System.Windows.Forms.Timer** се различава от употребата на **System.Timers.Timer**, само по сигнатурата на делегата за обратни извиквания, който в случая е стандартният **EventHandler**.
26.Изключения в .NET . Дефиниране на собствено изключение.
Собствени изключения
В .NET Framework програмистите могат да дефинират собствени класове за изключения и да създават класови йерархии с тях. голяма гъвкавост при управлението на грешки и необичайни ситу- ации. В по-големите приложения изключенията се разделят в за всяка категория се дефинира по един базов клас, а за конкретните представители на категориите се дефинира по един клас- по един абстрактен базов клас за категорията изключения, свързани с клиентите (**CustomerException**) и за категорията изключ поръчките (**OrderException**). Наследниците на **OrderException** и **CustomerException** също могат да се подреждат в класова йе собствени подкатегории.

При работата на приложението, използващо класовата йерархия от примера могат да се прихващат наведнъж всички грешки, с само някои конкретни от тях. Това дава добра гъвкавост при управлението на грешките.
Добре е да се спазва правилото, че йерархиите трябва да са широки и плитки, т.е. класовете на изключения трябва да са производни на тип, който се намира близо до **System.Exception**, и трябва да бъдат не повече от две или три нива надълбоко. Ако дефинираме тип за изключение, който няма да бъде базов за други типове, маркираме го като **sealed**, а ако не искаме да бъде инстанциран директно, го правим абстрактен.**Дефиниране на собствени изключения** За дефинирането на собствени изключения се наследява класът **System. ApplicationException** и му се създават подходящи конструктори и евен-туално му се добавят и допълнителни свойства, даващи специфична информация за проблема. Препоръчва се винаги да се дефинират поне следните два конструктора:
MyException(string message);
MyException(string message, Exception InnerException);
27.Правила за работа с изключения в .NET среда
Правила за работа с изключения
1. разработвате библиотека: ако прихванете всички изкл. , как разработващия приложение с библиотеката ще знае че нещо се е случило
2. разработвате библиотека с типове – не винаги знаете кое е грешка, кое не. Оставете това на викащия
3. Избягвайте код , прихващащ всичко: catch(System.Exception) {……}
4. Ако операция е частично завършена изключение и следва възстановяване в начално съст.: най-добре прихванете уведомете (с друго изкл.) викащата страна.
5. След прихващане и обработка на изключение, често е добре да уведомите извикващия: подавате същото (само с throw) (това е начина за преобразуваме изключението от нещо специфично, към общоразбираемо за потребител).

Необработвани съобщения (такива, които никой catch не разпознава)
Най напред следва да се разработи единна политика за тях – напр. въведен текст се съхранява и се визуализира диалогов прозо
-1.При отдалечено викана процедура или web услуга или сървърно-базиран код, който подава exception, то той се изпълня обкръжение на try/catch. Тъи като exception обекта е сериализиран, той може да се предава през граница на Domain – т.е клиентското приложение.
-2. В общия случай, необработени съобщения могат да се насочват за обработка към дефинирана в в приложението деле регистрирана като
event handle от тип System.UnhandledExceptionEventHandler към стандартния тип за изключния: System.AppDomain.Unha
AppDomain.CurentDomain.UnhandledException +=
new UnhandledExceptionEventHandler(MyUnhandledExceptionFunction);
3. Необработваните изключения в приложения, базирани на Windows Forms се прихващат така: цялата WinProc ф-ия всъщ обхващащ я автоматично try/catch.
При наличие на необработено по-долу изключение, catch блокът извиква виртуалния метод OnThreadException() дефини System.Windows.Forms.Control и предефиниран в Application
Той визуализира стандартен прозорец за 'unhandled exception'
Можете да предефинирате поведението чрез ваш метод от делегатен тип
System.Threading.ThreadExceptionEventHandler
и след това да свържете този метод с ThreadException събитието на класа Application
-4. Необработени съобщения в ASP.NET
ASP обхваща кода на приложението в собствен try блок и предопределя начин за обработка. Може да се намесите като ре метод към събитие Error на класа System.Web.UI.Page или на клас System.Web.UI.UserControl

(методът може и да се вика за всяко необработено изключение от която и да е страница на приложението – ако callback метода е свързан с Error събитие на клас System.Web.HTTPApplication)

5. Необработени изключения в среда ASP.NET XML

Отново обхващащ кода try блок на ASP.NET подава SoapException обект. Той се сериализира в XML вид и може да се предава към друг компютър или приложение, работещо като клиент на XML Web услугата.

30. Същност на механизма на сериализация. Сериализиране на обекти с вградени класове.

Сериализация

В съвременното програмиране често се налага да се съхрани състоянието на даден обект от паметта и да се възстанови след известно време. Това позволява обектите временно да се съхраняват на твърдия диск и да се използват след време, както и да се пренасят по мрежата и да се възстановяват на отдалечена машина.

Проблемите при съхранението и възстановяването на обекти са много и за справянето с тях има различни подходи. За да се намалят усилията на разработчиците в .NET Framework е изградена технология за автоматизация на този процес, наречена **сериализация**. Нека се запознаем по-подробно с нея.

Какво е сериализация (serialization)?

Сериализацията е процес, който преобразува обект или свързан граф от обекти до поток от байтове, като запазва състоянието на неговите полета и свойства. Потокът може да бъде двоичен (binary) или текстов (XML).

Запазване на състоянието на обект

Сериализацията се използва за съхранение на информация и запазване на състоянието на обекти. Използвайки сериализация, дадена програма може да съхрани състоянието си във файл, база данни или друг носител и след време да го възстанови обратно.

можем да сериализираме обект
и да го запишем в бинарен файл със средствата на .NET Framework:

```
string str = ".NET Framework";
BinaryFormatter f = new BinaryFormatter();
using (Stream s = new FileStream("sample.bin", FileMode.Create))
{
f.Serialize(s, str);
}
```

При сериализирането на обекта в потока се записват името на класа, името на асемблито (assembly) и друга информация за обекта, както и всички член-променливи, които не са маркирани като **[NonSerialized]** (употребата на този атрибут ще обясним по-нататък в тази тема). При десериализацията информацията се чете от потока и се пресъздава обектът.

Методи за сериализация

public static MemberInfo[] GetSerializableMembers(Type)

Методът приема като параметър типа на класа, който ще бъде сериализиран, и връща като резултат масив от **MemberInfo** обекти, съдържащи информация за сериализируемите членове на класа.

public static Object[] GetObjectData(Object, MemberInfo[])

Методът приема като параметри обект, който ще бъде сериализиран и масив с членовете, които трябва бъдат извлечени от обекта. За всеки от тях се извлича стойността, асоциирана с него в сериализирания обект и тези стойности се връщат като масив от обекти. Дължината му е същата, като дължината на масива с членовете, извличани от обекта.

35. Стратегии на управление на памет и събиране на 'боклук' в .NET среда. Алгоритъм за " събиране на боклук"

Как работи garbage collector?

Вече беше споменато, че ако добавянето на нов обект би довело до препълване на хийпа, трябва да се осъществи почистване на паметта. В този момент, CLR стартира системата за почистване на паметта, т.нар. garbage collector. **Н опростено обяснение. Garbage collector се**

стартира когато Поколение 0 се запълни. Поколенията се разглеждат в следващата секция.
Първото нещо, което трябва да се направи, за да може системата за почистване на паметта да започне работа, това е да се пр
приложението, изпълняващи управляван код. по време на събирането на отпадъци е твърде вероятно обектите да се премест
динамичната памет, нишките не трябва да могат да достъпват и модифицират обекти докато трае почистването. CLR изчаква в
в безопасно състояние, след което ги приспива. Съществуват няколко механизма, чрез които CLR може да приспи дадена ниш
различни механизми е
стремежът да се намали колкото се може повече натоварването и нишките да останат активни възможно най-дълго.
Освобождаване на неизползваните обекти
След като всички управлявани нишки на приложението са безопасно "приспани", garbage collector проверява дали в managed h
вече не се използват от приложението. Ако такива обекти съществуват, заетата от тях памет се освобождава. След приключва
работата по събиране на отпадъци се възобновява работата на всички
нишки и приложението продължава своето изпълнение. Както вероятно се досещате, откриването на ненужните обекти и освоб
заети от тях, не е проста задача. В тази
секция накратко ще опишем алгоритъмът, който .NET garbage collector използва за нейното решаване.
За да установи кои обекти подлежат на унищожение, garbage collector построява граф на всички обекти, достъпни от нишките н
момент. Всички обекти от динамичната памет, които не са част от графа се считат за отпадъци и подлежат на унищожаване. Въ
garbage collector може да знае кои обекти са достъпни и кои не? **Корените на приложението** са точката, от която
системата за почистване на паметта започва своята работа.
Корени на приложението
Всяко приложение има набор от корени (**application roots**). Корените представляват области от паметта, които сочат към обект
установени на **null**. Например всички глобални и статични
променливи, съдържащи референции към обекти се считат за корени на приложението. Всички локални променливи или парам
в който се изпълнява garbage collector, които сочат към обекти, също принадлежат към корените. Регистрите на процесора, съд
обекти, също са част от корените. Към корените на приложението спада и Freachable queue (за Freachable queue по-подробно ц
за финализация на обекти в настоящата глава.
Засега просто приемете че тази опашка е част от вътрешните структури, поддържани от CLR и се счита за един от корените на
компилаторът компилира IL инструкциите на даден метод в
процесорни инструкции, той също съставя и вътрешна таблица, съдър- жаща корените за съответния метод. Тази таблица е до
collector. Ако се случи garbage collector да започне работа, когато методът се изпълнява, той ще използва тази таблица, за да о
корените на приложението към този момент. Освен това се обхожда и стекът на извикванията за съответната нишка и се опред
извикващи методи (като се използват техните вътрешни таблици). Към получения набор от корени, естествено, се включват и т
щи се в глобални и статични променливи.Трябва да се помни, че не е задължително даден обект да излезе от обхват за да бъд
компилаторът може да определи__
кога този обект се достъпва от кода за последен път и веднага след това
го изключва от вътрешната таблица на корените, с което той става
кандидат за почистване от garbage collector. Изключение правят случаите,
когато кодът е компилиран с /**debug** опция, която предотвратява почист-
ването на обекти, които са в обхват. Това се прави за улеснение на
процеса на дебъгване – все пак при трасиране на кода бихме искали да
можем да следим състоянието на всички обекти, които са в обхват в
дадения момент.
Алгоритъмът за почистване на паметта
Когато garbage collector започва своята работа, той предполага че всички
обекти в managed heap са отпадъци, т.е. че никой от корените не сочи
към обект от паметта. След това, системата за почистване на паметта
започва да обхожда корените на приложението и да строи граф на
обектите, достъпни от тях.
Нека разгледаме примера, показан на следващата фигура. Ако глобална
променлива сочи към обект A от managed heap, то A ще се добави към
графа. Ако A съдържа указател към C, а той от своя страна към обектите
D и F, всички те също стават част от графа. Така garbage collector обхожда
рекурсивно в дълбочина всички обекти, достъпни от глобалната промен-
лива A:
Когато приключи с построяването на този клон от графа, garbage collector
преминава към следващия корен и обхожда всички достъпни от него
обекти. В нашия случай към графа ще бъде добавен обект E. Ако по време
на работата garbage collector се опита да добави към графа обект, който
вече е бил добавен, той спира обхождането на тази част от клона. Това се
прави с две цели:
- значително се увеличава производителността, тъй като не се преми-
нава през даден набор от обекти повече от веднъж;
- предотвратява се попадането в безкраен цикъл, ако съществуват
циклично свързани обекти (например A сочи към B, B към C, C към D

и D обратно към A).
След обхождането на всички корени на приложението, Графът съдържа всички обекти, които по някакъв начин са достъпни от приложението. В посочения на фигурата пример, това са обектите A, C, D, E и F.

Всички обекти, които не са част от този граф, не са достъпни и следователно се считат за отпадъци. В нашия пример това са обектите B, G, H и I. След идентифицирането на достъпните от приложението обекти, garbage collector преминава през хийпа, търсейки последователни блокове от отпадъци, които вече се смятат за свободно пространство. Когато такава област се намери, всички обекти, намиращи се над нея се придвижват надолу в паметта, като се използва стандартната функция **memcpy(…)**. Крайният резултат е, че всички обекти, оцелели при преминаването на garbage collector, се разполагат в долната част на хийпа, а **NextObjPtr** се установява непосредствено след последния обект. Фигурата показва състоянието на динамичната памет след приключване на работата на garbage collector.

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.

36. Финализация в .NET среда.

Какво е финализация?

Накратко, финализацията позволява да се почистват ресурси, свързани с даден обект, преди обектът да бъде унищожен от garbage collector. Обяснено най-просто, това е начин да се каже на CLR "преди този обект да бъде унищожен, трябва да се изпълни ето този код".
За да е възможно това, класът трябва да имплементира специален метод, наречен **Finalize()**. Когато garbage collector установи, че даден обект вече не се използва от приложението, той проверява дали обектът дефинира **Finalize()** метод. Ако това е така, **Finalize()** се изпълнява и на по-късен етап (най-рано при следващото преминаване на garbage collector), обектът се унищожава. Този процес ще бъде разгледан детайлно след малко. Засега просто трябва да запомните две неща:
- **Finalize() не може да се извиква явно**. Този метод се извиква само от системата за почистване на паметта, когато тя прецени, че даденият обект е отпадък.
- Най-малко **две** преминавания на garbage collector са необходими за да се унищожи обект, дефиниращ **Finalize()** метод. При първото се установява че обектът подлежи на унищожение и се изпълнява финализаторът, а при второто се освобождава и заетата от обекта памет. Всъщност в реалния живот почти винаги са необходими повече от две събирания на garbage collector поради преминаването на обекта в по-горно поколение.

37 Модел на явна финализация в .NET среда.Интегриране на Finalize() и Dispose()

Когато се създава нов обект, CLR проверява дали типът дефинира **Finalize()** метод и ако това е така, след създаването на обекта в динамичната памет (но преди извикването на неговия конструктор), указател към обекта се добавя към Finalization list. Така Finalization list съдържа указатели към всички обекти в хийпа, които трябва да бъдат финализирани (имат **Finalize()** методи), но все още се използват от приложението (или вече не се използват, но още не са проверени от garbage collector).
Създаването на обект, поддържащ финализация изисква една допълнителна операция от страна на CLR – поставянето на указател във Finalization list и следователно отнема и малко повече време.

Взаимодействието на garbage collector с обектите, нуждаещи се от финализация, е твърде интересно. Нека разгледаме следния пример.
Фигурата по-долу показва опростена схема на състоянието на динамичната памет точно преди да започне почистване на паметта. Виждаме че хийпът съдържа три обекта – A, B и C. Нека всички те са от Поколение 0. Обект A все още се използва от приложението, така че той ще оцелее при преминаването на garbage collector. Обекти B и C, обаче, са недостъпни от корените и се определят от garbage collector-а като отпадъци.

И така, garbage collector първо определя обект B като недостъпен и следователно – подлежащ на почистване. След това указателят към обект B се изтрива от Finalization list и се добавя към опашката Freachable. В този момент обектът се **съживява**, т.е. той се добавя към графа на достъпните обекти и вече не се счита за отпадък. Garbage collector пренарежда динамичната памет. При това обект B се третира както всеки друг достъпен от приложението обект, в нашия пример – обект A.
След това CLR стартира специална нишка с висок приоритет, която за всеки запис във Freachable queue изпълнява **Finalize()** метода на съответния обект и след това **изтрива записа от опашката**.
При следващото почистване на Поколение 1 от garbage collector, обект B ще бъде третиран като недостъпен (защото записът вече е изтрит от Freachable queue и никой от корените на приложението не обекта) и паметта, заемана от него ще бъде освободена. Забележете, че тъй като обектът вече е в по-високо поколение, преди това да се случи е възможно да минат още няколко преминавания на garbage collector,

Интерфейсът **IDisposable** се препоръчва от Microsoft в тези случаи, в които искате да га**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 разработе типове (Common Type System – CTS). CTS дефинира поддържаните от CLR типове данни и операциите над тях. Всички .NET е от CTS. За всеки тип в даден .NET език има някакво съответствие в CTS, макар че понякога това съответ-ствие не е директно. съществуват CTS типове, които не се поддържат от някои .NET езици. По идея всички езици в .NET Framework са обектно-ори System също се придържа към идеите на обектно-ориентираното програмиране (ООП) и по тази причина описва освен стандар 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 .N типовете и им съпоставят запазени думи съгласно своя синтаксис. Например типът **System.Int32** от CTS съответства на типа **i** **System.String** – на типа **string**.рантирате моментално освобождаване на ресурсите (вече знаете, че използването на **Finalize()**
Използването на **IDisposable** се състои в имплементирането на интерфейса от класа, който обвива някакъв неуправляван ресу ресурса при извикване на метода **Dispose()**.n

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). С съдържат директно стойността си в стека за изпълнение на прог-рамата, докато референтните типове съдържат строго типизир към стойността, която се намира в динамичната памет. По- нататък ще разгледаме подробно разликите между стойностните и р 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
в C#).
Стойностните типове директно съдържат стойността си и се съхраняват физически в работния стек за изпълнение на програма
приемат стойност **null**, защото реално не са указатели.
Стойностните типове и паметта
Стойностните типове заемат необходимата им памет в стека в момента на деклариране то им и я освобождават в момента на и
достигане на края на програмния блок, в който са декларирани). Заде-лянето и освобождаване на памет за стойностен тип реа
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 инстанци
задели в стека. Ако, обаче, стойностен тип (например структура) съдържа като член-данни референтни типове, стойностите им
динамичната памет.
Стойностните типове наследяват System.ValueType
CLR се грижи всички стойностни типове да наследяват системния тип **System.ValueType**. Всички типове, които не наследяват **V**
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 извикване
подаваните като параметри стойностни типове от оригиналното им местоположение в стека на ново място в стека и подава на
направените копия. Ако извикваният метод промени стойността на подадения му по стойност параметър, при връщане от извик
губи. Това поведение важи, разбира се, само ако параметрите се подават по подразбиране, без да се използват ключовите дум
разгледаме по-нататък в следващите теми.

54.Събития. Кратък пример.

Събитията могат да се разглеждат като съобщения за настъпване на някакво действие. В компонентно-ориентираното програм
изпращат събития (events) към своя притежател за да го уведомят за настъпването на интересна за него ситуация. Този модел
например за графичните потребителски интерфейси, където контро-лите уведомяват чрез събития други класове от програмата
потребителя. Например, когато потребителят натисне бутон, бутонът предизвиква събитие, с което известява, че е бил натисна
могат да се предизвикват не само при реализиране на потребителски интерфейси. Нека вземем за пример програма, в която ка
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
получател на събитието (event receiver). За да могат да получават дадено събитие, получателите му трябва преди това да се
(subscribe for event).
За едно събитие могат да се абонират произволен брой получатели. Изпращачът на събитието не знае кои ще са получателите
предизвиква. Затова чрез механизма на събитията се постига по-ниска степен на свързаност (coupling) между отделните компо
Събитията в .NET Framework
В компонентния модел на .NET Framework абонирането, изпращането и получаването на събития се поддържа чрез делегати и
на механизма на събитията е едно от главните приложения на делегатите. Класът, който публикува събитието, дефинира деле
събитието трябва да имплементират. Когато събитието бъде предизвикано, методите на абонатите се извикват посредством де
обикновено се наричат **обработчици** на събитието. Делега-тът е multicast делегат, за да могат чрез него да се извикват много
всички абонати).

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

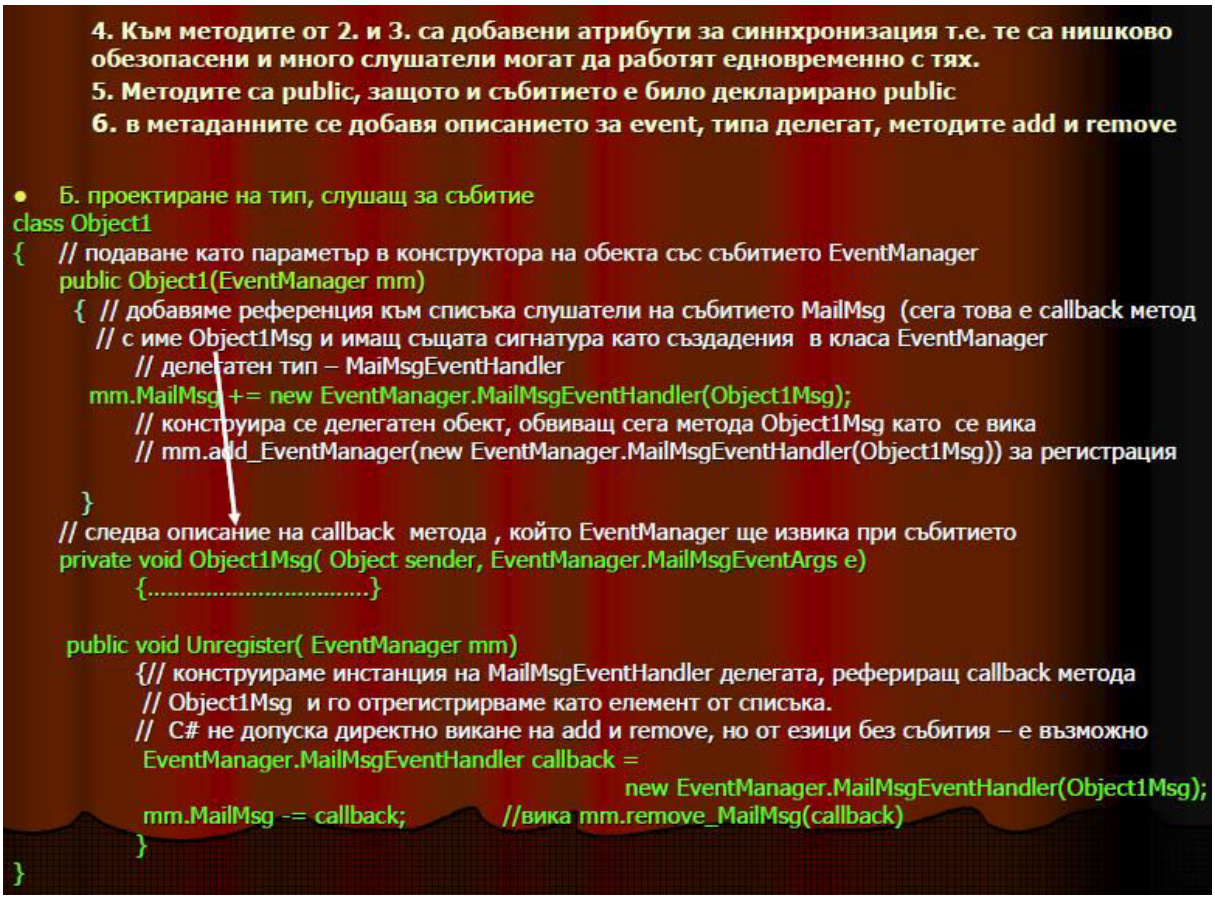
55.Проектиране на тип, предлагащ събитие. Проектиране на тип, използващ събитие. Същно

A. Проектиране на тип, предлагащ събитие(с цвят са задълж. неща)

```
class EventManager     {
// 1.следва вграден тип, дефиниращ информацията, предавана на
// получателите на събитие
        public class MailMsgEventArgs : EventArgs   {

          public MailMsgEventArgs( String from, String to, String subject, String body)
            { this.from = from; this.to = to;  this.subject = subject; this.body = body;}
                  public readonly String from, to, subject, body;
                                                              }
// 2.следва делегат, дефиниращ прототип на callback метод, който
// получателите следва да  имплементират
public delegate void MailMsgEventHandler ( Object sender, MailMsgEventArgs args);

//3. Следва дефиниция на самото събитие (получателите да импл. такъв callback метод)
public event MailMsgEventHandler MailMsg;
// 4. метод, отговорен за уведомяване на регистриралите интерес към събитието обекти
protected virtual void OnMailMsg(MailMsgEventArgs e) //може да се предефинира поведението му
            { if(MailMsg != null)                    //има ли регистрирали интерес към събитието
                        {MailMsg( this. e);          //уведомяваме всички рег. обекти
                        }
            }

// 5. метод, получаващ от вход данни и ги превежда. Възбужда събитието
public void SimulateArrivingMsg(String from, String to, String subject, String body)
            {           MailMsgEventArgs e = new MailMsgEventArgs(from, to, subject, body);
            //вика метода уведомяващ обектите за събитието
                        OnMailMsg(e);
            }
}
```

всъщност операторът : public event MailMsgEventHandler MailMsg;
се преобразува от компилатора така:

1. **създава се делегатно поле (private MailMsgEventHandler MailMsg = null), в началото null, впоследствие поддържащо референция към свързан списък от делегати, желаещи да бъдат уведомявани за събитието.Списъкът е 'private'**
2. **дефинира public void add_MailMsg(MailMsgEventHandler handler) метод за добавяне на нова референция в свързания списък.**
3. **дефинира public void remove_MailMsg(MailMsgEventHandler handler) метод за отрегистриране event handler за обект, който вече не се интересува от събитието.**



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 типов
(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** и
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 ти
опакова и разопакова стойностните типове автоматично. Това спестява дефинирането на обвиващи (wrapper) класове за прими
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 сто
Физически референтните типове представляват указател към стойност в динамичната памет, но за CLR те не са обикновени ука
типово-обезопасени указатели. Това означава, че CLR не допуска на един референтен тип да се присвои стойност от друг рефе
съвместим с него (т.е. не е същия тип или негов наследник). В резултат на това в .NET езиците грешките от неправилна работа
намалени.

Референтните типове и паметта

Всички референтни типове се съхраняват в **динамичната памет** (т. нар. **managed heap**), която се контролира от системата за п
(garbage collector). Динамичната памет е специално място от паметта, заделено от CLR за съхранение на данни, които се създа
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 почистван
garbage collector).

Когато инстанцираме референтен тип с оператора **new**, CLR заделя място в динамичната памет, където ще стоят данните и ед
съдържа адреса на заделеното място. Веднага след това заделената памет се занулява (освен ако програмистът не инициализ
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 рефер
данни от референтен тип, в динамичната памет се заделят указатели (референции) за тях, а техните стойности (ако не са **null**)
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
ментира чрез преместване на един указател. Освобождаването на памет, обаче, е сложна и времеотнемаща операция, която с
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
освобождаван**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
за числови типове). В .NET Framework типовете, които могат да бъдат сравнявани един с друг, трябва да имплементират интер
System.IComparable.

Интерфейсът дефинира един-единствен метод – **CompareTo(object)**. Този метод трябва да реализира сравняването и да връщ
- **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 сравн

Системни имплементации на IComparable

IComparable е имплементиран от много системни .NET типове, като например от примитивните стойностни типове **System.Cha**
System.Single, **System.Double**, от символните низове (**System.String**) и от изброените типове (**System.Enum**). Това улеснява р
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 **конт**
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 Framew
интерфейсите **IEnumerable** и **IEnumerator**.

Интерфейсът IEnumerable

Интерфейсът **System.IEnumerable** се имплементира от колекции и други типове, които поддържат операцията "обхождане на е
ред". Този интерфейс дефинира само един метод – методът **GetEnumerator()**. Той връща итератор (инстанция на **IEnumerator**
елементите на дадения обект.

Обектите, поддържащи **IEnumerable** интерфейса, могат да се използват от конструкцията **foreach** в C# за обхождане на всички
Интерфейсът **IEnumerable** е реализиран от много системни .NET типове, като **System.Array**, **System.String**, **ArrayList**, **Hashtab**
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 пр
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).

е на стой-ностните типове е значително по-бързо от референтните типове. Когато производителността е важна за нашата систе
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 тем
тях. По-нататък, в темата за управление на паметта и ресурсите, ще им обърнем специално внимание.

60.Делегати. Дефиниране, използване.
Делегатите са референтни типове, които описват сигнатурата на даден метод (броя, типа и последователността на параметрит
тип. Могат да се разглеждат като "обвивки" на методи - те представляват структури от данни, които приемат като стойност мето
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 по
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
ставляват типизиран указател към функция. Делегатите също съдържат силно типизиран указател към функция, но те са и нещ
обектно-ориентирани. На практика делегатите представляват класове. Инстанцията на един делегат може да съдържа в себе с
обект, така и метод.
Едно от основните приложения на делегатите е реализацията на "обратни извиквания", т.нар. callbacks. Идеята е да се предаде
който да бъде извикан по-късно. Така може да се осъществи например асинхронна обработка – от даден код извикваме метод,
метод и продължаваме работа, а извиканият метод извиква callback метода когато е необходимо. Със средствата на делега-тит
да позволи на потребителите си да предос-тавят метод, извършващ специфична обработка, като по този начин обработката не
предварително.Делегатите в .NET Framework са специални класове, които наследяват **System.Delegate** или **System.MulticastD
обаче явно могат да наследяват само CLR и компилаторът. Всъщност, те не са от тип делегат – тези класове се използват, за д
типове делегат.

Всеки делегат има **"списък на извикване" (invocation list)**, който представлява наредено множество делегати, като всеки еле
конкретен метод, реферирам 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 извикван
елемент, съдържащ референция към метод.
ножествени (multicast) делегати
Множествените делегати наследяват класа **System.MulticastDelegate**, който от своя страна е наследник на класа на **System.D
викат един или повече метода. Техните списъци на извикване съдър-жат множество елементи, всеки реферирщ метод. В тях м
се среща повече от веднъж. При извикване делегатът активира всички реферирани методи. Множествените делегати могат да
операции.
Езикът C# съдържа запазената дума **delegate**, чрез която се декларира делегат. При тази декларация компилаторът автоматич
MulticastDelegate., т.е. създава множествен делегат. Затова ще обърнем по-голямо внимание именно на този вид делегати.

19. LINQ – Language INtegrated Query. Query Expressions – Заявки вградени в езика. Ламбда

LINQ – Language INtegrated Query
(заявки вградени в езика)

- Основни фундаменти на LINQ
- LINQ - използвани езици C# 3.0 VB 9
- Особености
    - Lambda Expressions
    - Query Expressions
        - Delegate functions
        - Type inference
        - Anonymous types
        - Extension methods

- Expression trees

Инициализация на обекти

Invoice i = new Invoice { CustomerId = 123, Name = “Test” };

Is equivalent to:

Invoice I = new Invoice();

i.CustomerId = 123;

i.Name = “Test”;

Lambda Expressions
Predicates

- Predicate
  - (p) => p.Gender == “F”
- Projection
  - (p) => p.Gender ? “F” : “Female”
  - “Each person *p* becomes string “Female” if Gender is “F””

- **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);
        }
    }
}
```

- **24. Повторно генериране на изключението. Изключения във вложени конструкции.**
- Повторногенерираненаизключе- ние

• Използва се, когато прихванатото изключение не може да бъде обработено;

• Синтаксис:

```
catch(ExceptionType parameter){
// ...
throw;
}
```

• Повторно генериране на изключение може да се из- пълни само в рамките на catch-блок;

• Повторно генерираното изключение се обработва от следващият catch-блок;

Пример: Повторно генериране на из- ключение

```
#include <iostream>
#include <exception>
using namespace std;

void fun(void) {
try {
cout<<"Exception thrown in fun()"<<endl;
throw exception();
cout<<"This should not be printed"<<endl;
}
catch(exception& ex){
cout<<"Exception handled in fun()"<<endl;
throw;
}
cout<<"This should not be printed"<<endl;
}
```

```
17      int main(int argc ,char* argv[
])){
18
19  try     {
20      fun();
21      cout<<"This should not be printed"<<endl;
22  }
23  catch(const exception& ex){
24      cout<<"Exception handled in main()"<<endl;
25  }
26
27  cout<<"Program can continue"<<endl;
28
29  return 0;
30   }
```

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:

18. Windows Presentation Foundation (WPF).XAML.Контроли и логическо дърво. Примери

Нова рендираща система, базирана на DirectX
–Осигурява поддръжка на хардуерно ускорение

–Поддръжка на ефекти

–Вградена поддръжка на 3D
•**Добра интеграция** на 2Dи 3D UI
Независим от резолюцията!

•Декларативно програмиране –XAML

•Добри инструменти за разрабтване на GUI –Blend

•Стилове и теми

•Вградени анимации
•Композиране на елементи

•Разделяне на данните(**Data**)от поведението (**Behavior**)

•Лесно разпространение
–ClickOnce

–Browser(XBAP

XAML:
XML базиран език=>тагове и атрибути

•Декларативен
–Разделение на описание от поведение
•Описва .NETобекти

•Използва се за описване на потребителски интерфейс–работи с класовете от WPFплатформата

Как да създадем бутон:
<!--XAML-->
<Button Content="OK"/>
//C#
Button b= newButton() {Content = "Ok" };
Еквивалентно
Таг –класа на обекта
Атрибут –променя стойност на свойство

Prorerty елементи:

Не създават нови обекти

•Присвоява стойност на свойство
<Button>
<Button.Content>
<RectangleHeight="40"
Width="40"
Fill="Black" />
</Button.Content>
</Button>Таг –класа, собственик на свойството и името на свойството

Основни класове на WPF:

•DispatcherObject

•DependencyObject

•Visual

•UIElement

•FrameworkElementи Control

•Shapesи Text, ContentPresenter

•Control, ContentControl, UserControl

•Window
Контроли на WPF:
•Content Controls
–Buttons
•Button

•RepeatButton

•ToggleButton

•CheckBox

•RadioButton
Items Controls
–ItemsControl

–ListBox

–ListView

–ComboBox

–Menus
•Menu

•ContextMenu

–TreeView

–ToolBar

–StatusBar

2. Въведение в .NET среда. .NET framework архитектура. Поддържане на единна езикова среда.

Microsoft дефинират платформата .NET като съвкупност от технологии..NET платформата осигурява стандартизирана инфраст
използване, хостинг и интеграция на .NET приложения и XML уеб услуги, базирана на .NET сървърите на Microsoft, средствата

Компоненти на .NET Framework

Можем да разделим .NET Framework на два основни компонента:

- **Common Language Runtime (CLR)** – средата, в която се изпълнява управляваният код на .NET приложенията. Представлява
контролирано изпълнява .NET кода и осигурява различни услуги, като управление на сигурността, управление на паметта и др.

- **Framework Class Library (FCL)** – представлява основната библиотека от типове, които се използват при изграждането на .NE
основната функционалност за разработка, необходима за повечето приложения, като вход/изход, връзка с бази данни, работа с
приложения, използване на уеб услуги, изграждане на графичен потребителски интерфейс и др.

Архитектура на .NET Framework

Архитектурата на .NET Framework често пъти се разглежда на нива, както това е :

Операционна система

Операционната система управлява ресурсите, процесите и потребителите на машината. Тя предоставя и някои услуги на прило
COM+, MSMQ, IIS, WMI и други.

Средата, която изпълнява .NET приложенията (CLR), е обикновен процес в операционната система и се управлява от нея, както

Common Language Runtime

Общата среда за изпълнение Common Language Runtime (CLR) управлява процеса на изпълнение на .NET код. Тя се грижи за з
даване на паметта, управлява конкурентността, грижите за сигурността на приложенията и изпълнява други важни задачи, свър
кода.

Base Class Library

BCL представлява богата обектно-ориентирана библиотека с основни кла-сове, които осигуряват базова системна функционал
изход, работа с колекции, символни низове, мрежови ресурси, сигурност, отдалечено извикване, многонишковост и др.

ADO.NET и XML

Слоят на ADO.NET и XML предоставя удобен начин за работа с релационни и други бази от данни и средства за обработка на X

Езици за програмиране

.NET Framework позволява на разработчика да използва различни езици за програмиране, както и да интегрира в едно приложе
разработвани на различни езици. Възможно е дори клас, написан на един език, да бъде наследен и разширен от клас, написан
Microsoft .NET Framework поддържа стандартно езиците C#, VB.NET, Managed C++ и J#.Съвместимостта на езиците за програм
се дължи на архитектурни решения, които ще разгледаме в детайли след малко.

Common Language Runtime

След като се запознахме накратко с архитектурата на .NET Framework, нека сега разгледаме в детайли и най-важният компонен
Common Language Runtime (CLR) е сърцето на .NET Framework. Той пред-ставлява **среда за контролирано изпълнение на уп**
практика CLR е тази част от .NET Framework, която изпълнява компили-раните .NET програми в специална изолирана среда.

В своята същност CLR представлява виртуална машина, която изпълнява инструкции, на езика IL (Intermediate Language), език
всички .NET езици. CLR е нещо като виртуален компютър, който обаче не изпълнява асемблерен код за процесор Pentium, AMD
код.

Управляван код

Управляваният код (managed code) е кодът, който се изпълнява от CLR. Той представлява поредица от IL инструкции, които с
компилацията на .NET езиците. По време на изпълнение управляваният код се компилира допълнително до машиннозависим к
платформа и след това се изпълнява директно от процесора.

Intermediate Language (IL)

Междинният език Intermediate Language (IL), е език за програмиране от ниско ниво, подобен на асемблерните езици. За разлика
много по-високо ниво, отколкото асемблерите за съвременните мик-ропроцесори.

IL е обектно-ориентиран език. Той разполага с инструкции за заделяне на памет, за създаване на обект, за предизвикване и обр
за извикване на виртуални методи и други инструкции, свързани с обектно-ориентираното програмиране.

11. Windows Forms в .NET. Контроли и йерархия на графичните конролите. Създаване на дъщерни форми и контроли. П

Windows Forms е стандартната библиотека на .NET Framework за изграж-
дане на прозоречно-базиран графичен потребителски интерфейс (GUI) за
настолни (desktop) приложения. Windows Forms дефинира набор от
класове и типове, позволяващи изграждане на прозорци и диалози с
графични контроли в тях, чрез които се извършва интерактивно взаимо-
действие с потребителя.Windows Forms е типична компонентно-ориентирана библиотека за създаване на GUI, която предостав
писане на програмен код да се създава гъвкав графичен потребителски интерфейс.

Контролите в Windows Forms

Windows Forms съдържа богат набор от стандартни контроли: форми,
диалози, бутони, контроли за избор, текстови полета, менюта, ленти с
инструменти, статус ленти и много други.

Наследяване на форми и контроли

Windows Forms е проектирана така, че да позволява лесно наследяване и
разширяване на форми и контроли. Това дава възможност за преизполз-

ване на общите части на потребителския интерфейс.

Наследяване на форми
Наследяването на форми позволява повторно използване на части от потребителския интерфейс. Чрез него е възможно да променим наведнъж общите части на много форми. За целта дефинираме една базова форма, която съдържа общата за всички наследници функционалност.
Базовата форма е най-обикновена форма. Единствената особеност е, че контролите, които могат да се променят от наследниците, се обявяват като **protected**. При наследяване на форма се наследява класът на базовата форма.Не всички класове от Windows Forms са ко обикновени .NET компоненти, например **Menu**, **Timer** и **ImageList**. Изглежда малко странно защо менюто не е контрола, но това компонентата **Menu** реално няма графичен образ и представлява списък от **MenuItem** елементи. **MenuItem** класът вече има гра следователно е контрола.

12. Опционални и списъчни контроли. Основни пропъртита и събития. Приложение – пример.
Контролите в Windows Forms са текстовите полета, етикетите, бутоните, списъците, дърветата, таблиците, менютата, лентите с лентите и много други. Windows Forms дефинира базови класове за контролите и класове-наследници за всяка контрола. Базов е класът **System.Windows.Forms.Control**. Пример за контрола е например бутонът (класът **System.Windows.Forms.Button**).Всяка контрола обработва собствените си **събития**Когато главната приложение получи съобщение,свързано с някоя от неговите форми, тя препраща съобщението до обработчика на съобщения Този обработчик от своя страна проверява дали съобщението е за самата форма или за някоя нейна контрола. Ако съобщение обработва директноот съответния обработчик на събития. Ако съобщението е за някоя от контролите във формата, то се преда която получи съобщението, може да е обикновена контрола или контейнер-контрола.Когато обикновена контрола получи съобщ директно. Когато контейнер-контрола получи съобщение, тя проверява дали то е за нея или е за някоя от вложените контроли. докато съобщението достигне до контролата, за която е предназначено.Класът **System.Windows.Forms.Form** е базов клас за в Forms GUI приложенията. Той представлява графична форма - прозорец или диалогова кутия, която съдържа в себе си контрол навигацията между тях. Повечето прозорци имат рамка и специални бутони за затваряне, преместване и други стандартни опер прозорците и стандартните контроли по тяхната рамка зависят от настройките на графичната среда на операционната система. Програмистът има само частичен контрол над външния вид на прозорците. Клас класовете **Control**, **ScrollableControl** и **ContainerControl** и наследява от тях цялата им функционалност, всичките им свойства, **CheckBox** е кутия за избор в стил "да/не". Свойството й **Checked** задава дали е избрана.
RadioButton е контрола за алтернативен избор. Тя се използва в групи. Всички **RadioButton** контроли в даден контейнер (напр една група и в нея само един **RadioButton** е избран в даден момент. **ListBox** контролата се използва за изобразяване на списъ които потребителят може да избира чрез щракване с мишката върху тях. По-важните свойства на тази контрола са:
- **Items** – колекция, която задава списъка от елементи, съдържащи се в контролата.
- **SelectionMode** – разрешава/забранява избирането на няколко елемента едновременно.
- **SelectedIndex**, **SelectedItem**, **SelectedIndices**, **SelectedItems** – връщат избрания елемент (или избраните елементи).

ComboBox представлява кутия за редакция на текст с възможност за dropdown алтернативен избор.
- **Text** – съдържа въведения текст.
- **Items** – задава възможните стойности, от които потребителят може да избира.
- **DropDownStyle** – задава стила на контролата – дали само се избира стойност от списъка или може да се въвежда ръчно и друга стойност.

13. Диалози – стандартни и потребителски. Видове и приложение. Пример за употреба.
При разработката на Windows Forms приложения често пъти се налага да извеждаме диалогови кутии с някакви съобщения или разгледаме стандартните средства за такива ситуации.
Стандартни диалогови кутии Класът MessageBox ни позволява да извеждаме стандартни диалогови кутии, съдържащи текст, бутони и икони:
- съобщения към потребителя
- въпросителни диалози

Показването на диалогова кутия се извършва чрез извикване на статичния метод Show(…) на класа MessageBox.
Следният код, например, ще покаже диалогова кутия със заглавие "Предупреждение" и текст "Няма връзка с интернет":
MessageBox.Show("Няма връзка с Интернет.", "Предупреждение");
Пример за стандартна диалогова кутия с малко повече функционалност:
bool confirmed =MessageBox.Show("Наистина ли ще изтриете това?",

"Въпрос", MessageBoxButtons.YesNo,
MessageBoxIcon.Question) == DialogResult.Yes;
Този код ще покаже диалогова кутия със заглавие "Въпрос" и текст "Наистина ли ще изтриете това?". Преди текста ще има икон
нея, а под него – бутони Yes и No. Ако потребителят
натисне Yes, променливата confirmed ще има стойност true, в противен случай ще има стойност false.
Извикване на диалогови кутии Освен стандартните диалогови кутии можем да използваме и потреби-
телски дефинирани диалогови кутии. Те представляват обикновени форми и се извикват модално по следния начин:
DialogResult result = dialog.ShowDialog();
Методът ShowDialog() показва формата като модална диалогова кутия. Типът DialogResult съдържа резултата (OK, Yes, No, Ca
на диалога. Задаването на DialogResult може да става автоматично, чрез свойството DialogResult на бутоните, или ръчно – пред
чрез свойството му DialogResult.

14. SDI и MDI приложения. Структура и пример.

MFC прави лесно да се работи едновременно с един документен интерфейсни (SDI) и няколко документни интерфейсни (MDI)
SDI приложенията позволяват само един отворен прозорец на документ кадър по кадър. MDI приложенията позволяват няколк
отворени в една и съща инстанция на приложението. Приложението MDI има прозорец, в който няколко MDI прозорци, които са
си, могат да бъдат отворени, всяка от които съдържа отделен документ. В някои приложения, детето прозорци могат да са от р
например прозорците на графиките и таблиците прозорци. В този случай, на лентата с менюта може да се променят MDI прозор
,които са активирани.
Осн. развичия в SDI и MDI прилож. Много изгледи в MDIпроблемът за синхронизация на промените в изглед.
Разлики между SDI и MDI:
1.В MDI има повече от един отворен документ, докато в SDI , за да отвори втори трябва да затворим първия.
2.В MDI могат да се поддържат различни типове документи.
3.При MDI в менюто има опция Windows за превключване на прозорците.
4.MDI има поне 2 менюта , а SDI има едно .Първото при отворен документ, а второто при затворен.
5.В SDI има една рамка, а в MDI има главна и дъщерна рамка.
Множество изгледи под един документ в MDI приложения
Използва се многодокументния шаблон- CMultiDocTemplate.Тъй като документа е същия не се създава нов документ, а само но
рамката си остават същите.MFC осигурява списъчна структура за обхождане на всички изгледи(като например при UpdateAllVie
Document, за да получи данни.Ако в някое View променим данните и искаме промените да се отразят в други View, налага се да
UpdateAllView, който пък вика OnUpdate на всеки View.Този метод не се вика автоматично-ние трябва да го осигурим. UpdateAll
ги обновява, като инвалидизира (Invalidate) целия прозорец.Ако искаме оптимизация ние трябва да променим реализацията на
да променим параметрите, с които се вика OnUpdate.
В SDI има един обект на приложението на документа.
Това означава , че за да отвори нов документ трябва да затворим стария.
В MDI имаме много документални обекти, затова не е необходимо припокриване.
При избиране на нов се създава нов обект.Ако е ….. Template то се взема от него документен шаблон и се създава , а ако са по
екрана от кой точно шаблон да се вземе.
Отваряне на съществуващ документ.

15. Свързване с база данни. Свързване на данни с контроли (Data Binding). DataGrid. Master-Details.Пример.

База от данни се нарича всяка организирана колекция от данни.
Свързване на данни
Свързването на данни (data binding) осигурява автоматично прехвърляне на данни между контроли и източници на данни. Може
масив, съдържащ имена на градове, с ComboBox контрола и имената от масива ще се показват в нея.Всички Windows Forms ко
свързване на данни (data binding). Можем да свържем което и да е свойство на контрола към източник на данни.
Контролата DataGrid
DataGrid контролата визуализира таблични данни. Тя осигурява навигация по редове и колони и позволява редактиране на дан
данни най-често се използват ADO.NET DataSet и DataTable. Чрез свойството DataSource се задава източникът на данни, а чре
DataMember – пътят до данните в рамките на източника. По-важни
свойства на контролата са:
- ReadOnly – разрешава / забранява редакцията на данни.
- CaptionVisible – показва / скрива заглавието.
- ColumnHeadersVisible – показва / скрива заглавията на колоните.
- RowHeadersVisible – показва / скрива колоната в ляво от редовете.
- TableStyles – задава стилове за таблицата.
o MappingName – задава таблицата, за която се отнася дефинира-
ният стил.
o GridColumnStyles – задава форматиране на отделните колони –
заглавие, ширина и др.
Master-Details навигация
Навигацията "главен/подчинен" (master-details) отразява взаимоотношения от тип едно към много (например един регион има м
Forms се поддържа навигация "главен/подчинен". За да илюстрираме работата с нея, нека разгледаме един пример: Имаме Da
таблици – едната съдържа имена на държави, а другата – имена на градове. Те са свързани помежду си така, че на всяка държ
съответстват определени градове от втората

таблица:
Тогава можем да използваме две DataGrid контроли – първата, визуализираща държавите, а втората, визуализираща градовет
текущо избраната държава от първата контрола. За целта контролите се свързват с един и същ DataSet. На главната контрола
данни главната таблица. На подчинената контрола се задава за източник на данни релацията на таблицата:

```
// Bind the master grid to the master table
DataGridCountries.DataSource = datasetCountriesAndTowns;
DataGridCountries.DataMember = "Countries";
// Bind the detail grid to the relationship
DataGridTowns.DataSource = datasetCountriesAndTowns;
DataGridTowns.DataMember = "Countries.CountriesTowns";
```

16. GDI+. Методи за изчертаване на основните фигури. Запълване, контур и специални ефекти.

Пакетът **System.Drawing** осигурява достъп до GDI+ функциите на
Windows:
- повърхности за чертане
- работа с графика и графични трансформации
- изчертаване на геометрични фигури
- работа с изображения
- работа с текст и шрифтове
- печатане на принтер

Той се състои от няколко пространства:
- **System.Drawing** – предоставя основни класове като повърхности,
моливи, четки, класове за изобразяване на текст.
- **System.Drawing.Imaging** – предоставя класове за работа с изобра-
жения, картинки и икони, класове за записване в различни файлови
формати и за преоразмеряване на изображения.
- **System.Drawing.Drawing2D** – предоставя класове за графични транс-
формации – бленди, матрици и др.
- **System.Drawing.Text** – предоставя класове за достъп до шрифтовете
на графичната среда.

- **System.Drawing.Printing** – предоставя класове за печатане на
принтер и системни диалогови кутии за печатане.

Класът Graphics

Класът **System.Drawing.Graphics** предоставя абстрактна повърхност за
чертане. Такава повърхност може да бъде както част от контрола на
екрана, така и част от страница на принтер или друго устройство.
Най-често чертането се извършва в обработчика на събитието **Paint**. В
него при необходимост се преизчертава графичния облик на контролата.
Параметърът **PaintEventArgs**, който се подава, съдържа **Graphics** обекта.
Graphics обект може да се създава чрез **Control.CreateGraphics()**. Той
задължително трябва да се освобождава чрез **finally** блок или с конс-
трукцията **using**, защото е ценен ресурс.

Чрез примера ще илюстрираме работата с GDI+ чрез пакета **System.Drawing –** чертане на геометрични фигури с четки и моли
текст със зададен шрифт.

```
private void MainForm_Paint(object sender,
System.Windows.Forms.PaintEventArgs e)
{
Graphics g = e.Graphics;
g.SmoothingMode = SmoothingMode.AntiAlias;
Brush brush = new SolidBrush(Color.Blue);
g.FillEllipse(brush, 50, 40, 350, 250);
brush.Dispose();
Pen pen = new Pen(Color.Red, 2);
g.DrawRectangle(pen, 40, 50, 200, 40);
pen.Dispose();
brush = new SolidBrush(Color.Yellow);
Font font = new Font("Arial", 14, FontStyle.Bold);
g.DrawString(".NET Framework", font, brush, 60, 60);
brush.Dispose();

font.Dispose();
```

}

17. Вход/Изход в .NET. Работа с файлове, директории, потоци, четци и писци.
Потоците в обектно-ориентираното програмиране са една абстракция, с която се осъществява вход и изход от дадена програма
концепция са аналогични на потоците в други обектно-ориентирани езици, напр. Java, C++ и Delphi (Object Pascal).
Потокът е подредена серия от байтове, която служи като абстрактен канал за данни. Този виртуален канал свързва програмата
съхранение или пренос на данни (напр. файл върху хард диск), като достъпът до канала е последователен. Потоците предоста
запис на поредици от байтове от и към устройството. Това е стандартният механизъм за извършване на входно-изходни опера
Framework.Потоците в .NET Framework се делят на две групи – **базови** и **преходни**. И едните, и другите, наследяват абстрактн
System.IO.Stream, базов за всички потоци.Базовите потоци пишат и четат директно от някакъв външен механизъм за съхранен
система (например класът **FileStream**), паметта (**MemoryStream**) или данни, достъпни по мрежата (**NetworkStream**). По-нататъ
FileStream в точката "Файлови потоци". Преходните потоци пишат и четат от други потоци (най-често в базови потоци), като пр
добавят допълнителна функци-оналност, например буфериране (**BufferedStream**) или кодиране (**CryptoStream**). По-подробно
BufferedStream в точката "Буферирани потоци". За четене на данни от поток се използва методът **int Read(byte[] buffer, int off**
най-много **count** на брой байта от текущата позиция на входния поток, увеличава позицията и връ-ща броя прочетени байтове
на потока. Четенето може да блокира за неопределено време. Например, ако при четене от мрежа извикаме метода **NetworkSt**
налични данни за четене, операцията блокира до тяхното получаване. В такива случаи е уместно да се използва свойството **Ne**
DataAvailable, което показва дали в потока има пристигнали данни, които още не са прочетени, т. е. дали последваща операци
ще върне резултат веднага.

Писане в поток

Методът **Write(byte[] buffer, int offset, int count)** записва в изход-ния поток **count** байта, като започва от зададеното отместван
тази операция е блокираща, т.е. може да предизвика забавяне за неопределено време. Не е гарантирано, че байтовете, записа
достигнали до местоназначението си след успеш-ното изпълнение на метода. Възможно е потокът да буферира данните и да н
веднага.Файловите потоци в .NET Framework са реализирани в класа **FileStream**, който вече беше използван в примера за пото
Stream, той поддържа всичките му методи и свойства (четене, писане, позициониране) и добавя някои допълнителни.Четенето
потоци, както и другите по-рядко използвани операции, се извършват както при всички наследници на класа **Stream** – с методит

Файловите потоци поддържат пряк достъп до определена позиция от файла чрез метода **Seek(…)**.

Четците и писачите (readers and writers) в .NET Framework са класове, които улесняват работата с потоците. При работа напри
програмистът може да чете и записва единствено байтове. Когато този поток се обвие в четец или писач, вече са позволени че
различни структури от данни, например примитивни типове, текстова информация и други типове. Четците и писачите биват дв
Двоичните четци и писачи осигуряват четене и запис на примитивни типове данни в двоичен вид – **ReadChar()**, **ReadChars()**, **R**
др. за четене и съответно **Write(char)**, **Write(char[])**, **Write(Int32)**, **Write(double)** – за запис. Може да се чете и записва и **string**, н
вид на масив от символи и префиксно се записва дължината му – **ReadString()**, респ. **Write(string)**.Текстовите четци и писачи о
на текстова инфор-мация, представена във вид на низове, разделени с нов ред. Базови текстови четци и писачи са абстрактнит
TextWriter. Основните методи за четене и запис са следните: - **ReadLine()** – прочита един ред текст.

- **ReadToEnd()** – прочита всичко от текущата позиция до края на потока.

- **Write(…)** – вмъква данни в потока на текущата позиция.
Работа с директории. Класове Directory и DirectoryInfo
Класовете **Directory** и **DirectoryInfo** са помощни класове за работа с директории. Ще изброим основните им методи, като отбел
са статични, а за **DirectoryInfo** – достъпни чрез инстанция.
- **Create()**, **CreateSubdirectory()** – създава директория или подди-ректория.

- **GetFiles(…)** – връща всички файлове в директорията.

- **GetDirectories(…)** – връща всички поддиректории на директорията.

- **MoveTo(…)** – премества (преименува) директория.

- **Delete()** – изтрива директория.

- **Exists()** – проверява директория дали съществува.

- **Parent** – връща горната директория.

- **FullName** – пълно име на директорията.

20. Таймер (Timer). Работа с таймери. Пропъртита, събития. Пример. Таймери

Често в приложенията, които разработваме, възниква необходимост от изпълняване на задачи през регулярни времеви интервали. Таймерите предоставят такава услуга. Те са обекти, които известяват приложението при изтичане на предварително зададен интервал от време. Таймерите са полезни в редица сценарии, например, когато искаме да обновяваме периодично потребителския интерфейс с актуална информация за статуса на някаква задача или да проверяваме състоянието на променящи се данни.

System.Timers.Timer

Класът предоставя събитие за изтичане на времевия интервал **Elapsed**,

което е делегат от тип **ElapsedEventHandler**, дефиниран като:

```
public delegate void ElapsedEventHandler(
object sender, ElapsedEventArgs e);
```

При изтичане на интервала, указан в свойството **Interval**, таймерът от тип **System.Timers.Timer** ще извика записалите се за събитието методи, използвайки нишка от пула. Ако използваме един и същ метод за получаване на събития от няколко таймера, чрез аргумента **sender** можем да ги разграничим. Класът **ElapsedEventArgs** чрез свойството **DateTime SignalTime** ни предоставя точното време, когато е бил извикван метода. За стартиране и спиране на известяването, можем да извикаме съответно **Start()** и **Stop()** методите. Свойството **Enabled** ни позволява да инструктираме таймера да игнорира събитието **Elapsed**. Това прави **Enabled** функционално еквивалентно на съответните **Start()** и **Stop()** методи. Когато приключим с таймера, трябва да извикаме **Close()**, за да освободим съответните системни ресурси.

System.Threading.Timer

System.Threading.Timer прилича на **System.Timers.Timer** и също използва пула с нишки. Основната разлика е, че той позволява малко по-разширен контрол – може да указваме кога таймера да започне да отброява, както и да предаваме всякаква информация на метода за обратни извиквания чрез обект от произволен тип. За да ползваме **System.Threading.Timer**, трябва в конструктора му да подадем делегат от тип **TimerCallback**, дефиниран като:

```
public delegate void TimerCallback(object state);
```

При всяко изтичане на времевия интервал, ще бъдат извиквани методите в този делегат. Обикновено като обект за състояние има полза да подаваме създателя на таймера, за да можем да използваме същия метод за обратни извиквания за обработка на събития от м
Другият параметър в конструктора на таймера е времевият интервал. Той може и да бъде променен впоследствие с извикване на **Change(…)** метода. **System.Threading.Timer** не предлага удобен начин за стартиране и спиране. Неговата работа започва веднага след конструирането му (точно след изтичането на подаденото стартово време) и прекъсването му става само чрез **Dispose()**. Ако искаме да го рестартираме трябва да създадем нов обект.

System.Windows.Forms.Timer

Пространството от имена **System.Windows.Forms** съдържа още един клас за таймер, който е със следната дефиниция:

```
public class Timer : Component, IComponent, Idisposable
{
public Timer();
public bool Enabled{virtual get ; virtual set;}
public int Interval {get; set;}
public event EventHandler Tick;
public void Start();
public void Stop();
```

}
Въпреки, че методите на **System.Windows.Forms.Timer** много приличат на тези на **System.Timers.Timer**, то **System.Windows.Forms.Timer** не използва пула с нишки за обратните извиквания към Windows Forms приложението. Вместо това, през определено време той пуска Windows съобщението **WM_TIMER** в опашката за съобщения на текущата нишка.
Използването на **System.Windows.Forms.Timer** се различава от употребата на **System.Timers.Timer**, само по сигнатурата на делегата за обратни извиквания, който в случая е стандартният **EventHandler**.
26.Изключения в .NET . Дефиниране на собствено изключение.
Собствени изключения
В .NET Framework програмистите могат да дефинират собствени класове за изключения и да създават класови йерархии с тях. голяма гъвкавост при управлението на грешки и необичайни ситу- ации. В по-големите приложения изключенията се разделят в за всяка категория се дефинира по един базов клас, а за конкретните представители на категориите се дефинира по един клас- по един абстрактен базов клас за категорията изключения, свързани с клиентите (**CustomerException**) и за категорията изключ поръчките (**OrderException**). Наследниците на **OrderException** и **CustomerException** също могат да се подреждат в класова йе собствени подкатегории.

При работата на приложението, използващо класовата йерархия от примера могат да се прихващат наведнъж всички грешки, с само някои конкретни от тях. Това дава добра гъвкавост при управлението на грешките.
Добре е да се спазва правилото, че йерархиите трябва да са широки и плитки, т.е. класовете на изключения трябва да са производни на тип, който се намира близо до **System.Exception**, и трябва да бъдат не повече от две или три нива надълбоко. Ако дефинираме тип за изключение, който няма да бъде базов за други типове, маркираме го като **sealed**, а ако не искаме да бъде инстанциран директно, го правим абстрактен.**Дефиниране на собствени изключения** За дефинирането на собствени изключения се наследява класът **System. ApplicationException** и му се създават подходящи конструктори и евен-туално му се добавят и допълнителни свойства, даващи специфична информация за проблема. Препоръчва се винаги да се дефинират поне следните два конструктора:
MyException(string message);
MyException(string message, Exception InnerException);
27.Правила за работа с изключения в .NET среда
Правила за работа с изключения
1. разработвате библиотека: ако прихванете всички изкл. , как разработващия приложение с библиотеката ще знае че нещо се е случило
2. разработвате библиотека с типове – не винаги знаете кое е грешка, кое не. Оставете това на викащия
3. Избягвайте код , прихващащ всичко: catch(System.Exception) {……}
4. Ако операция е частично завършена изключение и следва възстановяване в начално съст.: най-добре прихванете уведомете (с друго изкл.) викащата страна.
5. След прихващане и обработка на изключение, често е добре да уведомите извикващия: подавате същото (само с throw) (това е начина за преобразуваме изключението от нещо специфично, към общоразбираемо за потребител).

Необработвани съобщения (такива, които никой catch не разпознава)
Най напред следва да се разработи единна политика за тях – напр. въведен текст се съхранява и се визуализира диалогов прозо
-1.При отдалечено викана процедура или web услуга или сървърно-базиран код, който подава exception, то той се изпълня обкръжение на try/catch. Тъи като exception обекта е сериализиран, той може да се предава през граница на Domain – т.е клиентското приложение.
-2. В общия случай, необработени съобщения могат да се насочват за обработка към дефинирана в в приложението деле регистрирана като
event handle от тип System.UnhandledExceptionEventHandler към стандартния тип за изключния: System.AppDomain.Unha
AppDomain.CurentDomain.UnhandledException +=
new UnhandledExceptionEventHandler(MyUnhandledExceptionFunction);
3. Необработваните изключения в приложения, базирани на Windows Forms се прихващат така: цялата WinProc ф-ия всъщ обхващащ я автоматично try/catch.
При наличие на необработено по-долу изключение, catch блокът извиква виртуалния метод OnThreadException() дефини System.Windows.Forms.Control и предефиниран в Application
Той визуализира стандартен прозорец за 'unhandled exception'
Можете да предефинирате поведението чрез ваш метод от делегатен тип
System.Threading.ThreadExceptionEventHandler
и след това да свържете този метод с ThreadException събитието на класа Application
-4. Необработени съобщения в ASP.NET
ASP обхваща кода на приложението в собствен try блок и предопределя начин за обработка. Може да се намесите като ре метод към събитие Error на класа System.Web.UI.Page или на клас System.Web.UI.UserControl

(методът може и да се вика за всяко необработено изключение от която и да е страница на приложението – ако callback метода е свързан с Error събитие на клас System.Web.HTTPApplication)

5. Необработени изключения в среда ASP.NET XML

Отново обхващащ кода try блок на ASP.NET подава SoapException обект. Той се сериализира в XML вид и може да се предава към друг компютър или приложение, работещо като клиент на XML Web услугата.

30. Същност на механизма на сериализация. Сериализиране на обекти с вградени класове.

Сериализация

В съвременното програмиране често се налага да се съхрани състоянието на даден обект от паметта и да се възстанови след известно време. Това позволява обектите временно да се съхраняват на твърдия диск и да се използват след време, както и да се пренасят по мрежата и да се възстановяват на отдалечена машина.

Проблемите при съхранението и възстановяването на обекти са много и за справянето с тях има различни подходи. За да се намалят усилията на разработчиците в .NET Framework е изградена технология за автоматизация на този процес, наречена **сериализация**. Нека се запознаем по-подробно с нея.

Какво е сериализация (serialization)?

Сериализацията е процес, който преобразува обект или свързан граф от обекти до поток от байтове, като запазва състоянието на неговите полета и свойства. Потокът може да бъде двоичен (binary) или текстов (XML).

Запазване на състоянието на обект

Сериализацията се използва за съхранение на информация и запазване на състоянието на обекти. Използвайки сериализация, дадена програма може да съхрани състоянието си във файл, база данни или друг носител и след време да го възстанови обратно.

можем да сериализираме обект
и да го запишем в бинарен файл със средствата на .NET Framework:

```
string str = ".NET Framework";
BinaryFormatter f = new BinaryFormatter();
using (Stream s = new FileStream("sample.bin", FileMode.Create))
{
f.Serialize(s, str);
}
```

При сериализирането на обекта в потока се записват името на класа, името на асемблито (assembly) и друга информация за обекта, както и всички член-променливи, които не са маркирани като **[NonSerialized]** (употребата на този атрибут ще обясним по-нататък в тази тема). При десериализацията информацията се чете от потока и се пресъздава обектът.

Методи за сериализация

public static MemberInfo[] GetSerializableMembers(Type)

Методът приема като параметър типа на класа, който ще бъде сериализиран, и връща като резултат масив от **MemberInfo** обекти, съдържащи информация за сериализируемите членове на класа.

public static Object[] GetObjectData(Object, MemberInfo[])

Методът приема като параметри обект, който ще бъде сериализиран и масив с членовете, които трябва бъдат извлечени от обекта. За всеки от тях се извлича стойността, асоциирана с него в сериализирания обект и тези стойности се връщат като масив от обекти. Дължината му е същата, като дължината на масива с членовете, извличани от обекта.

35. Стратегии на управление на памет и събиране на ‘боклук’ в .NET среда. Алгоритъм за “ събиране на боклук”

Как работи garbage collector?

Вече беше споменато, че ако добавянето на нов обект би довело до препълване на хийпа, трябва да се осъществи почистване на паметта. В този момент, CLR стартира системата за почистване на паметта, т.нар. garbage collector. **Ето опростено обяснение. Garbage collector се**

стартира когато Поколение 0 се запълни. Поколенията се разглеждат в следващата секция.
Първото нещо, което трябва да се направи, за да може системата за почистване на паметта да започне работа, това е да се пр
приложението, изпълняващи управляван код. по време на събирането на отпадъци е твърде вероятно обектите да се преместя
динамичната памет, нишките не трябва да могат да достъпват и модифицират обекти докато трае почистването. CLR изчаква в
в безопасно състояние, след което ги приспива. Съществуват няколко механизма, чрез които CLR може да приспи дадена нишк
различни механизми е
стремежът да се намали колкото се може повече натоварването и нишките да останат активни възможно най-дълго.
Освобождаване на неизползваните обекти
След като всички управлявани нишки на приложението са безопасно "приспани", garbage collector проверява дали в managed he
вече не се използват от приложението. Ако такива обекти съществуват, заетата от тях памет се освобождава. След приключва
работата по събиране на отпадъци се възобновява работата на всички
нишки и приложението продължава своето изпълнение. Както вероятно се досещате, откриването на ненужните обекти и освоб
заети от тях, не е проста задача. В тази
секция накратко ще опишем алгоритъмът, който .NET garbage collector използва за нейното решаване.
За да установи кои обекти подлежат на унищожение, garbage collector построява граф на всички обекти, достъпни от нишките на
момент. Всички обекти от динамичната памет, които не са част от графа се считат за отпадъци и подлежат на унищожаване. Въ
garbage collector може да знае кои обекти са достъпни и кои не? **Корените на приложението** са точката, от която
системата за почистване на паметта започва своята работа.
Корени на приложението
Всяко приложение има набор от корени (**application roots**). Корените представляват области от паметта, които сочат към обект
установени на **null**. Например всички глобални и статични
променливи, съдържащи референции към обекти се считат за корени на приложението. Всички локални променливи или парам
в който се изпълнява garbage collector, които сочат към обекти, също принадлежат към корените. Регистрите на процесора, съд
обекти, също са част от корените. Към корените на приложението спада и Freachable queue (за Freachable queue по-подробно ще
за финализация на обекти в настоящата глава.
Засега просто приемете че тази опашка е част от вътрешните структури, поддържани от CLR и се счита за един от корените на
компилаторът компилира IL инструкциите на даден метод в
процесорни инструкции, той също съставя и вътрешна таблица, съдър- жаща корените за съответния метод. Тази таблица е до
collector. Ако се случи garbage collector да започне работа, когато методът се изпълнява, той ще използва тази таблица, за да о
корените на приложението към този момент. Освен това се обхожда и стекът на извикванията за съответната нишка и се опред
извикващи методи (като се използват техните вътрешни таблици). Към получения набор от корени, естествено, се включват и т
щи се в глобални и статични променливи.Трябва да се помни, че не е задължително даден обект да излезе от обхват за да бъд
компилаторът може да определи__
кога този обект се достъпва от кода за последен път и веднага след това
го изключва от вътрешната таблица на корените, с което той става
кандидат за почистване от garbage collector. Изключение правят случаите,
когато кодът е компилиран с /**debug** опция, която предотвратява почист-
ването на обекти, които са в обхват. Това се прави за улеснение на
процеса на дебъгване – все пак при трасиране на кода бихме искали да
можем да следим състоянието на всички обекти, които са в обхват в
дадения момент.
Алгоритъмът за почистване на паметта
Когато garbage collector започва своята работа, той предполага че всички
обекти в managed heap са отпадъци, т.е. че никой от корените не сочи
към обект от паметта. След това, системата за почистване на паметта
започва да обхожда корените на приложението и да строи граф на
обектите, достъпни от тях.
Нека разгледаме примера, показан на следващата фигура. Ако глобална
променлива сочи към обект A от managed heap, то A ще се добави към
графа. Ако A съдържа указател към C, а той от своя страна към обектите
D и F, всички те също стават част от графа. Така garbage collector обхожда
рекурсивно в дълбочина всички обекти, достъпни от глобалната промен-
лива A:
Когато приключи с построяването на този клон от графа, garbage collector
преминава към следващия корен и обхожда всички достъпни от него
обекти. В нашия случай към графа ще бъде добавен обект E. Ако по време
на работата garbage collector се опита да добави към графа обект, който
вече е бил добавен, той спира обхождането на тази част от клона. Това се
прави с две цели:
- значително се увеличава производителността, тъй като не се преми-
нава през даден набор от обекти повече от веднъж;
- предотвратява се попадането в безкраен цикъл, ако съществуват
циклично свързани обекти (например A сочи към B, B към C, C към D

и D обратно към A).
След обхождането на всички корени на приложението, Графът съдържа всички обекти, които по някакъв начин са достъпни от приложението. В посочения на фигурата пример, това са обектите A, C, D, E и F.

Всички обекти, които не са част от този граф, не са достъпни и следователно се считат за отпадъци. В нашия пример това са обектите B, G, H и I. След идентифицирането на достъпните от приложението обекти, garbage collector преминава през хийпа, търсейки последователни блокове от отпадъци, които вече се смятат за свободно пространство. Когато такава област се намери, всички обекти, намиращи се над нея се придвижват надолу в паметта, като се използва стандартната функция **memcpy(…)**. Крайният резултат е, че всички обекти, оцелели при преминаването на garbage collector, се разполагат в долната част на хийпа, а **NextObjPtr** се установява непосредствено след последния обект. Фигурата показва състоянието на динамичната памет след приключване на работата на garbage collector.

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.

36. Финализация в .NET среда.

Какво е финализация?

Накратко, финализацията позволява да се почистват ресурси, свързани с даден обект, преди обектът да бъде унищожен от garbage collector. Обяснено най-просто, това е начин да се каже на CLR "преди този обект да бъде унищожен, трябва да се изпълни ето този код".
За да е възможно това, класът трябва да имплементира специален метод, наречен **Finalize()**. Когато garbage collector установи, че даден обект вече не се използва от приложението, той проверява дали обектът дефинира **Finalize()** метод. Ако това е така, **Finalize()** се изпълнява и на по-късен етап (най-рано при следващото преминаване на garbage collector), обектът се унищожава. Този процес ще бъде разгледан детайлно след малко. Засега просто трябва да запомните две неща:
- **Finalize() не може да се извиква явно**. Този метод се извиква само от системата за почистване на паметта, когато тя прецени, че даденият обект е отпадък.
- Най-малко **две** преминавания на garbage collector са необходими за да се унищожи обект, дефиниращ **Finalize()** метод. При първото се установява че обектът подлежи на унищожение и се изпълнява финализаторът, а при второто се освобождава и заетата от обекта памет. Всъщност в реалния живот почти винаги са необходими повече от две събирания на garbage collector поради преминаването на обекта в по-горно поколение.

37 Модел на явна финализация в .NET среда.Интегриране на Finalize() и Dispose()

Когато се създава нов обект, CLR проверява дали типът дефинира **Finalize()** метод и ако това е така, след създаването на обекта в динамичната памет (но преди извикването на неговия конструктор), указател към обекта се добавя към Finalization list. Така Finalization list съдържа указатели към всички обекти в хийпа, които трябва да бъдат финализирани (имат **Finalize()** методи), но все още се използват от приложението (или вече не се използват, но още не са проверени от garbage collector).
Създаването на обект, поддържащ финализация изисква една допълнителна операция от страна на CLR – поставянето на указател във Finalization list и следователно отнема и малко повече време.

Взаимодействието на garbage collector с обектите, нуждаещи се от финализация, е твърде интересно. Нека разгледаме следния пример.
Фигурата по-долу показва опростена схема на състоянието на динамичната памет точно преди да започне почистване на паметта. Виждаме че хийпът съдържа три обекта – A, B и C. Нека всички те са от Поколение 0. Обект A все още се използва от приложението, така че той ще оцелее при преминаването на garbage collector. Обекти B и C, обаче, са недостъпни от корените и се определят от garbage collector-а като отпадъци.

И така, garbage collector първо определя обект B като недостъпен и следователно – подлежащ на почистване. След това указателят към обект B се изтрива от Finalization list и се добавя към опашката Freachable. В този момент обектът се **съживява**, т.е. той се добавя към графа на достъпните обекти и вече не се счита за отпадък. Garbage collector пренарежда динамичната памет. При това обект B се третира както всеки друг достъпен от приложението обект, в нашия пример – обект A.
След това CLR стартира специална нишка с висок приоритет, която за всеки запис във Freachable queue изпълнява **Finalize()** метода на съответния обект и след това **изтрива записа от опашката**.
При следващото почистване на Поколение 1 от garbage collector, обект B ще бъде третиран като недостъпен (защото записът вече е изтрит от Freachable queue и никой от корените на приложението не обекта) и паметта, заемана от него ще бъде освободена. Забележете, че тъй като обектът вече е в по-високо поколение, преди това да се случи е възможно да минат още няколко преминавания на garbage collector,

Интерфейсът **IDisposable** се препоръчва от Microsoft в тези случаи, в които искате да га**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 разработе типове (Common Type System – CTS). CTS дефинира поддържаните от CLR типове данни и операциите над тях. Всички .NET е от CTS. За всеки тип в даден .NET език има някакво съответствие в CTS, макар че понякога това съответ-ствие не е директно. съществуват CTS типове, които не се поддържат от някои .NET езици. По идея всички езици в .NET Framework са обектно-ори System също се придържа към идеите на обектно-ориентираното програмиране (ООП) и по тази причина описва освен стандар 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 .N типовете и им съпоставят запазени думи съгласно своя синтаксис. Например типът **System.Int32** от CTS съответства на типа **i** **System.String** – на типа **string**.рантирате моментално освобождаване на ресурсите (вече знаете, че използването на **Finalize()**
Използването на **IDisposable** се състои в имплементирането на интерфейса от класа, който обвива някакъв неуправляван ресу ресурса при извикване на метода **Dispose()**.n

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). С съдържат директно стойността си в стека за изпълнение на прог-рамата, докато референтните типове съдържат строго типизир към стойността, която се намира в динамичната памет. По- нататък ще разгледаме подробно разликите между стойностните и р 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
в C#).
Стойностните типове директно съдържат стойността си и се съхраняват физически в работния стек за изпълнение на програма
приемат стойност **null**, защото реално не са указатели.
Стойностните типове и паметта
Стойностните типове заемат необходимата им памет в стека в момента на деклариране то им и я освобождават в момента на и
достигане на края на програмния блок, в който са декларирани). Заде-лянето и освобождаване на памет за стойностен тип реал
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 инстанцир
задели в стека. Ако, обаче, стойностен тип (например структура) съдържа като член-данни референтни типове, стойностите им
динамичната памет.
Стойностните типове наследяват System.ValueType
CLR се грижи всички стойностни типове да наследяват системния тип **System.ValueType**. Всички типове, които не наследяват **V**
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 извикване
подаваните като параметри стойностни типове от оригиналното им местоположение в стека на ново място в стека и подава на
направените копия. Ако извикваният метод промени стойността на подадения му по стойност параметър, при връщане от извик
губи. Това поведение важи, разбира се, само ако параметрите се подават по подразбиране, без да се използват ключовите дум
разгледаме по-нататък в следващите теми.

54.Събития. Кратък пример.

Събитията могат да се разглеждат като съобщения за настъпване на някакво действие. В компонентно-ориентираното програм
изпращат събития (events) към своя притежател за да го уведомят за настъпването на интересна за него ситуация. Този модел
например за графичните потребителски интерфейси, където контро-лите уведомяват чрез събития други класове от програмата
потребителя. Например, когато потребителят натисне бутон, бутонът предизвиква събитие, с което известява, че е бил натисна
могат да се предизвикват не само при реализиране на потребителски интерфейси. Нека вземем за пример програма, в която ка
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 даден
получател на събитието (event receiver). За да могат да получават дадено събитие, получателите му трябва преди това да се
(subscribe for event).
За едно събитие могат да се абонират произволен брой получатели. Изпращачът на събитието не знае кои ще са получателите
предизвиква. Затова чрез механизма на събитията се постига по-ниска степен на свързаност (coupling) между отделните компо
Събитията в .NET Framework
В компонентния модел на .NET Framework абонирането, изпращането и получаването на събития се поддържа чрез делегати и
на механизма на събитията е едно от главните приложения на делегатите. Класът, който публикува събитието, дефинира делег
събитието трябва да имплементират. Когато събитието бъде предизвикано, методите на абонатите се извикват посредством де
обикновено се наричат **обработчици** на събитието. Делега-тът е multicast делегат, за да могат чрез него да се извикват много о
всички абонати).

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

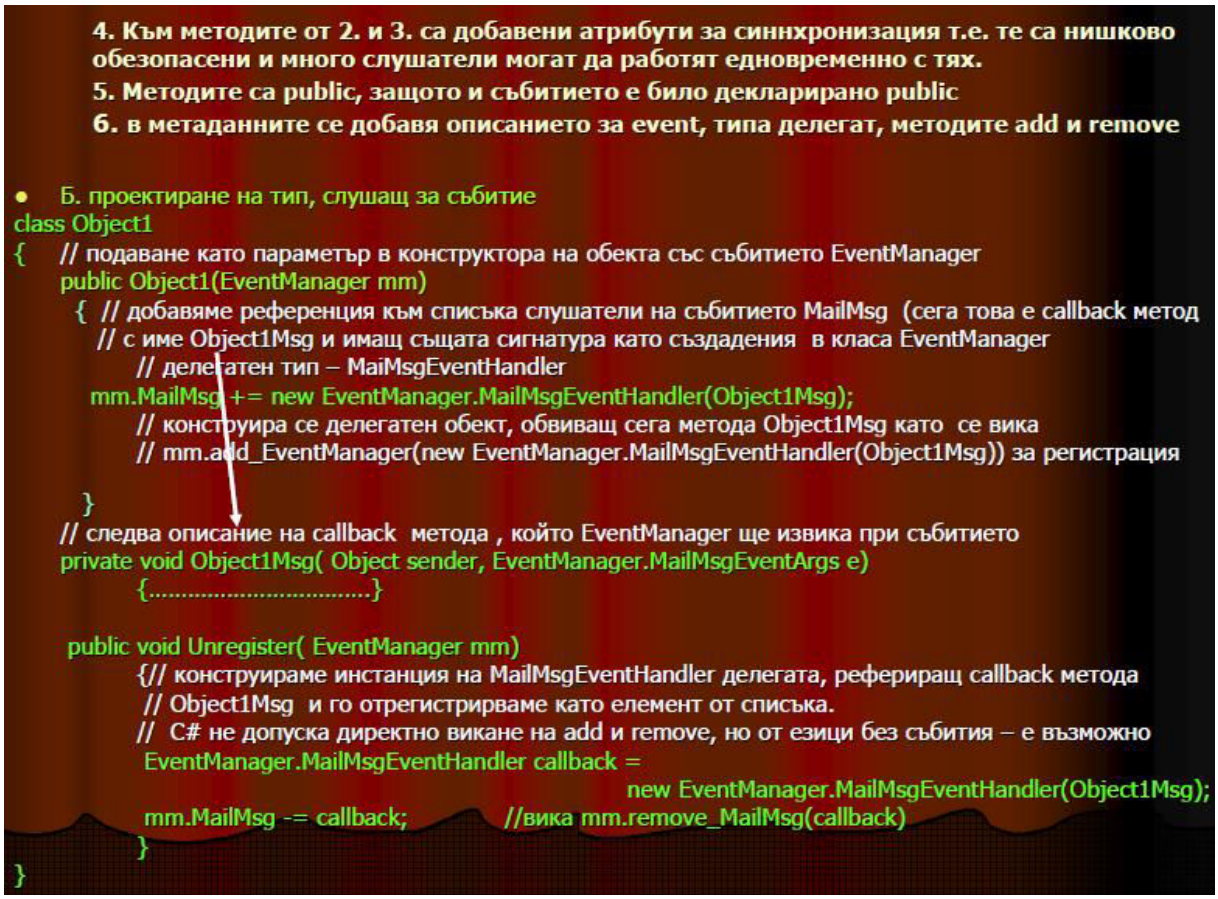
55.Проектиране на тип, предлагащ събитие. Проектиране на тип, използващ събитие. Същно

A. Проектиране на тип, предлагащ събитие(с цвят са задълж. неща)

```
class EventManager     {
// 1.следва вграден тип, дефиниращ информацията, предавана на
// получателите на събитие
        public class MailMsgEventArgs : EventArgs   {

          public MailMsgEventArgs( String from, String to, String subject, String body)
            { this.from = from; this.to = to;  this.subject = subject; this.body = body;}
                  public readonly String from, to, subject, body;
                                                          }
// 2.следва делегат, дефиниращ прототип на callback метод, който
// получателите следва да  имплементират
public delegate void MailMsgEventHandler ( Object sender, MailMsgEventArgs args);

//3. Следва дефиниция на самото събитие (получателите да импл. такъв callback метод)
public event MailMsgEventHandler MailMsg;
// 4. метод, отговорен за уведомяване на регистриралите интерес към събитието обекти
protected virtual void OnMailMsg(MailMsgEventArgs e) //може да се предефинира поведението му
            { if(MailMsg != null)                    //има ли регистрирали интерес към събитието
                        {MailMsg( this. e);          //уведомяваме всички рег. обекти
                        }
            }

// 5. метод, получаващ от вход данни и ги превежда. Възбужда събитието
public void SimulateArrivingMsg(String from, String to, String subject, String body)
            {           MailMsgEventArgs e = new MailMsgEventArgs(from, to, subject, body);
            //вика метода уведомяващ обектите за събитието
                        OnMailMsg(e);
            }
}
```

всъщност операторът : public event MailMsgEventHandler MailMsg;
се преобразува от компилатора така:

1. създава се делегатно поле (private MailMsgEventHandler MailMsg = null), в началото null, впоследствие поддържащо референция към свързан списък от делегати, желаещи да бъдат уведомявани за събитието.Списъкът е 'private'
2. дефинира public void add_MailMsg(MailMsgEventHandler handler) метод за добавяне на нова референция в свързания списък.
3. дефинира public void remove_MailMsg(MailMsgEventHandler handler) метод за отрегистриране event handler за обект, който вече не се интересува от събитието.



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 типов
(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** и
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 ти
опакова и разопакова стойностните типове автоматично. Това спестява дефинирането на обвиващи (wrapper) класове за прими
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 сто
Физически референтните типове представляват указател към стойност в динамичната памет, но за CLR те не са обикновени ука
типово-обезопасени указатели. Това означава, че CLR не допуска на един референтен тип да се присвои стойност от друг рефе
съвместим с него (т.е. не е същия тип или негов наследник). В резултат на това в .NET езиците грешките от неправилна работа
намалени.

Референтните типове и паметта

Всички референтни типове се съхраняват в **динамичната памет** (т. нар. **managed heap**), която се контролира от системата за п
(garbage collector). Динамичната памет е специално място от паметта, заделено от CLR за съхранение на данни, които се създа
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 почистван
garbage collector).

Когато инстанцираме референтен тип с оператора **new**, CLR заделя място в динамичната памет, където ще стоят данните и ед
съдържа адреса на заделеното място. Веднага след това заделената памет се занулява (освен ако програмистът не инициализ
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 рефе
данни от референтен тип, в динамичната памет се заделят указатели (референции) за тях, а техните стойности (ако не са **null**)
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
ментира чрез преместване на един указател. Освобождаването на памет, обаче, е сложна и времеотнемаща операция, която с
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
освобождаван**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
за числови типове). В .NET Framework типовете, които могат да бъдат сравнявани един с друг, трябва да имплементират интер
System.IComparable.

Интерфейсът дефинира един-единствен метод – **CompareTo(object)**. Този метод трябва да реализира сравняването и да връщ
- **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 срав

Системни имплементации на IComparable

IComparable е имплементиран от много системни .NET типове, като например от примитивните стойностни типове **System.Cha
System.Single**, **System.Double**, от символните низове (**System.String**) и от изброените типове (**System.Enum**). Това улеснява р
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 **конт
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 Framew
интерфейсите **IEnumerable** и **IEnumerator**.

Интерфейсът IEnumerable

Интерфейсът **System.IEnumerable** се имплементира от колекции и други типове, които поддържат операцията "обхождане на е
ред". Този интерфейс дефинира само един метод – методът **GetEnumerator()**. Той връща итератор (инстанция на **IEnumerator**
елементите на дадения обект.

Обектите, поддържащи **IEnumerable** интерфейса, могат да се използват от конструкцията **foreach** в C# за обхождане на всички
Интерфейсът **IEnumerable** е реализиран от много системни .NET типове, като **System.Array**, **System.String**, **ArrayList**, **Hashtab
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 пр
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).

е на стой-ностните типове е значително по-бързо от референтните типове. Когато производителността е важна за нашата систе
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 тем
тях. По-нататък, в темата за управление на паметта и ресурсите, ще им обърнем специално внимание.

60.Делегати. Дефиниране, използване.
Делегатите са референтни типове, които описват сигнатурата на даден метод (броя, типа и последователността на параметри
тип. Могат да се разглеждат като "обвивки" на методи - те представляват структури от данни, които приемат като стойност мето
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 по
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
ставляват типизиран указател към функция. Делегатите също съдържат силно типизиран указател към функция, но те са и нещ
обектно-ориентирани. На практика делегатите представляват класове. Инстанцията на един делегат може да съдържа в себе с
обект, така и метод.
Едно от основните приложения на делегатите е реализацията на "обратни извиквания", т.нар. callbacks. Идеята е да се предаде
който да бъде извикан по-късно. Така може да се осъществи например асинхронна обработка – от даден код извикваме метод,
метод и продължаваме работа, а извиканият метод извиква callback метода когато е необходимо. Със средствата на делега-тит
да позволи на потребителите си да предос-тавят метод, извършващ специфична обработка, като по този начин обработката не
предварително.Делегатите в .NET Framework са специални класове, които наследяват **System.Delegate** или **System.MulticastD
обаче явно могат да наследяват само CLR и компилаторът. Всъщност, те не са от тип делегат – тези класове се използват, за д
типове делегат.

Всеки делегат има **"списък на извикване" (invocation list)**, който представлява наредено множество делегати, като всеки еле
конкретен метод, реферирам 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 извикван
елемент, съдържащ референция към метод.
ножествени (multicast) делегати
Множествените делегати наследяват класа **System.MulticastDelegate**, който от своя страна е наследник на класа на **System.D
викат един или повече метода. Техните списъци на извикване съдър-жат множество елементи, всеки реферирщ метод. В тях м
се среща повече от веднъж. При извикване делегатът активира всички реферирани методи. Множествените делегати могат да
операции.
Езикът C# съдържа запазената дума **delegate**, чрез която се декларира делегат. При тази декларация компилаторът автоматич
MulticastDelegate., т.е. създава множествен делегат. Затова ще обърнем по-голямо внимание именно на този вид делегати.

19. LINQ – Language INtegrated Query. Query Expressions – Заявки вградени в езика. Ламбда

LINQ – Language INtegrated Query
(заявки вградени в езика)

- Основни фундаменти на LINQ
- LINQ - използвани езици C# 3.0 VB 9
- Особености
    - Lambda Expressions
    - Query Expressions
        - Delegate functions
        - Type inference
        - Anonymous types
        - Extension methods

- Expression trees

Инициализация на обекти

Invoice i = new Invoice { CustomerId = 123, Name = “Test” };

Is equivalent to:

Invoice I = new Invoice();

i.CustomerId = 123;

i.Name = “Test”;

Lambda Expressions
Predicates

- Predicate
  - (p) => p.Gender == “F”
- Projection
  - (p) => p.Gender ? “F” : “Female”
  - “Each person *p* becomes string “Female” if Gender is “F””

- **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);
        }
    }
}
```

- **24. Повторно генериране на изключението. Изключения във вложени конструкции.**
- Повторногенерираненаизключе- ние

• Използва се, когато прихванатото изключение не може да бъде обработено;

• Синтаксис:

```
catch(ExceptionType parameter){
// ...
throw;
}
```

• Повторно генериране на изключение може да се из- пълни само в рамките на catch-блок;

• Повторно генерираното изключение се обработва от следващият catch-блок;

Пример: Повторно генериране на из- ключение

```
#include <iostream>
#include <exception>
using namespace std;

void fun(void) {
try {
cout<<"Exception thrown in fun()"<<endl;
throw exception();
cout<<"This should not be printed"<<endl;
}
catch(exception& ex){
cout<<"Exception handled in fun()"<<endl;
throw;
}
cout<<"This should not be printed"<<endl;
}
```

```
17      int main(int argc ,char* argv[
])){
18
19  try     {
20      fun();
21      cout<<"This should not be printed"<<endl;
22  }
23  catch(const exception& ex){
24      cout<<"Exception handled in main()"<<endl;
25  }
26
27  cout<<"Program can continue"<<endl;
28
29  return 0;
30   }
```

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:

18. Windows Presentation Foundation (WPF).XAML.Контроли и логическо дърво. Примери

Нова рендираща система, базирана на DirectX
–Осигурява поддръжка на хардуерно ускорение

–Поддръжка на ефекти

–Вградена поддръжка на 3D
•**Добра интеграция** на 2Dи 3D UI
Независим от резолюцията!

•Декларативно програмиране –XAML

•Добри инструменти за разработване на GUI –Blend

•Стилове и теми

•Вградени анимации
•Композиране на елементи

•Разделяне на данните(**Data**)от поведението (**Behavior**)

•Лесно разпространение
–ClickOnce

–Browser(XBAP

XAML:
XML базиран език=>тагове и атрибути

•Декларативен
–Разделение на описание от поведение
•Описва .NETобекти

•Използва се за описване на потребителски интерфейс–работи с класовете от WPFплатформата

Как да създадем бутон:
<!--XAML-->
<Button Content="OK"/>
//C#
Button b= newButton() {Content = "Ok" };
Еквивалентно
Таг –класа на обекта
Атрибут –променя стойност на свойство

Prorerty елементи:

Не създават нови обекти

•Присвоява стойност на свойство
<Button>
<Button.Content>
<RectangleHeight="40"
Width="40"
Fill="Black" />
</Button.Content>
</Button>Таг –класа, собственик на свойството и името на свойството

Основни класове на WPF:

•DispatcherObject

•DependencyObject

•Visual

•UIElement

•FrameworkElementи Control

•Shapesи Text, ContentPresenter

•Control, ContentControl, UserControl

•Window
Контроли на WPF:
•Content Controls
–Buttons
•Button

•RepeatButton

•ToggleButton

•CheckBox

•RadioButton
Items Controls
–ItemsControl

–ListBox

–ListView

–ComboBox

–Menus
•Menu

•ContextMenu

–TreeView

–ToolBar

–StatusBar

11. Windows Forms в .NET. Контроли и йерархия на графичните конролите. Създаване на дъщерни форми и контроли. Пример.

Windows Forms е стандартната библиотека на .NET Framework за изграждане на прозоречно-базиран графичен потребителски интерфейс (GUI) за настолни (desktop) приложения. Windows Forms дефинира набор от класове и типове, позволяващи изграждане на прозорци и диалози с графични контроли в тях, чрез които се извършва интерактивно взаимодействие с потребителя.Windows Forms е типична компонентно-ориентирана библиотека за създаване на GUI, която предоставя възможност с малко писане на програмен код да се създава гъвкав графичен потребителски интерфейс.

Контролите в Windows Forms

Windows Forms съдържа богат набор от стандартни контроли: форми, диалози, бутони, контроли за избор, текстови полета, менюта, ленти с инструменти, статус ленти и много други.

Наследяване на форми и контроли

Windows Forms е проектирана така, че да позволява лесно наследяване и разширяване на форми и контроли. Това дава възможност за преизползване на общите части на потребителския интерфейс.

Наследяване на форми

Наследяването на форми позволява повторно използване на части от потребителския интерфейс. Чрез него е възможно да променим наведнъж общите части на много форми. За целта дефинираме една базова форма, която съдържа общата за всички наследници функционалност.
Базовата форма е най-обикновена форма. Единствената особеност е, че контролите, които могат да се променят от наследниците, се обявяват като **protected**. При наследяване на форма се наследява класът на базовата форма.Не всички класове от Windows Forms са контроли. Някои са обикновени .NET компоненти, например **Menu**, **Timer** и **ImageList**. Изглежда малко странно защо менюто не е контрола, но това е така, защото компонентата **Menu** реално няма графичен образ и представлява списък от **MenuItem** елементи. **MenuItem** класът вече има графичен образ и следователно е контрола.

12. Опционални и списъчни контроли. Основни пропъртита и събития. Приложение – пример.

Контролите в Windows Forms са текстовите полета, етикетите, бутоните, списъците, дърветата, таблиците, менютата, лентите с инструменти, статус лентите и много други. Windows Forms дефинира базови класове за контролите и класове-наследници за всяка контрола. Базов клас за всички контроли е класът **System.Windows.Forms.Control**. Пример за контрола е например бутонът (класът **System.Windows.Forms.Button**).Всяка контрола обработва собствените си **събития**Когато главната нишка на Windows Forms приложение получи съобщение,свързано с някоя от неговите форми, тя препраща съобщението до обработчика на съобщения на съответната форма. Този обработчик от своя страна проверява дали съобщението е за самата форма или за някоя нейна контрола. Ако съобщението е за формата, то се обработва директноот съответния обработчик на събития. Ако съобщението е за някоя от контролите във формата, то се предава на нея. Контролата, която получи съобщението, може да е обикновена контрола или контейнер-контрола.Когато обикновена контрола получи съобщение, тя го обработва директно. Когато контейнер-контрола получи съобщение, тя проверява дали то е за нея или е за някоя от вложените контроли. Процесът продължава, докато съобщението достигне до контролата, за която е предназначено.Класът **System.Windows.Forms.Form** е базов клас за всички форми вWindows Forms GUI приложенията. Той представлява графична форма - прозорец или диалогова кутия, която съдържа в себе си контроли и управлява навигацията между тях. Повечето прозорци имат рамка и специални бутони за затваряне, преместване и други стандартни операции. Външният вид на прозорците и стандартните контроли по тяхната рамка зависят от настройките на графичната среда на операционната система. Програмистът има само частичен контрол над външния вид на прозорците. Класът **Form** е наследник на класовете **Control**, **ScrollableControl** и **ContainerControl** и наследява от тях цялата им функционалност, всичките им свойства, събития и методи.

CheckBox е кутия за избор в стил "да/не". Свойството й **Checked** задава дали е избрана.

RadioButton е контрола за алтернативен избор. Тя се използва в групи. Всички **RadioButton** контроли в даден контейнер (например форма) образуват една група и в нея само един **RadioButton** е избран в даден момент. **ListBox** контролата се използва за изобразяване на списък със символни низове, които потребителят може да избира чрез щракване с мишката върху тях. По-важните свойства на тази контрола са:

- **Items** – колекция, която задава списъка от елементи, съдържащи се в контролата.
- **SelectionMode** – разрешава/забранява избирането на няколко елемента едновременно.
- **SelectedIndex**, **SelectedItem**, **SelectedIndices**, **SelectedItems** – връщат избрания елемент (или избраните елементи).

ComboBox представлява кутия за редакция на текст с възможност за dropdown алтернативен избор.

- **Text** – съдържа въведения текст.
- **Items** – задава възможните стойности, от които потребителят може да избира.
- **DropDownStyle** – задава стила на контролата – дали само се избира стойност от списъка или може да се въвежда ръчно и друга стойност.

13. Диалози – стандартни и потребителски. Видове и приложение. Пример за употреба.
При разработката на Windows Forms приложения често пъти се налага да извеждаме диалогови кутии с някакви съобщения или с някакъв въпрос. Нека разгледаме стандартните средства за такива ситуации.
Стандартни диалогови кутии Класът MessageBox ни позволява да извеждаме стандартни диалогови кутии, съдържащи текст, бутони и икони:
- съобщения към потребителя
- въпросителни диалози
Показването на диалогова кутия се извършва чрез извикване на статичния метод Show(…) на класа MessageBox.
Следният код, например, ще покаже диалогова кутия със заглавие "Предупреждение" и текст "Няма връзка с интернет":
MessageBox.Show("Няма връзка с Интернет.", "Предупреждение");
Пример за стандартна диалогова кутия с малко повече функционалност:
bool confirmed =MessageBox.Show("Наистина ли ще изтриете това?",
"Въпрос", MessageBoxButtons.YesNo,
MessageBoxIcon.Question) == DialogResult.Yes;
Този код ще покаже диалогова кутия със заглавие "Въпрос" и текст "Наистина ли ще изтриете това?". Преди текста ще има икона с въпросителен знак в нея, а под него – бутони Yes и No. Ако потребителят натисне Yes, променливата confirmed ще има стойност true, в противен случай ще има стойност false.
Извикване на диалогови кутии Освен стандартните диалогови кутии можем да използваме и потребителски дефинирани диалогови кутии. Те представляват обикновени форми и се извикват модално по следния начин:
DialogResult result = dialog.ShowDialog();
Методът ShowDialog() показва формата като модална диалогова кутия. Типът DialogResult съдържа резултата (OK, Yes, No, Cancel и др.) от извикването на диалога. Задаването на DialogResult може да става автоматично, чрез свойството DialogResult на бутоните, или ръчно – преди затварянето на диалога чрез свойството му DialogResult.

14. SDI и MDI приложения. Структура и пример.
MFC прави лесно да се работи едновременно с един документен интерфейсни (SDI) и няколко документни интерфейсни (MDI) приложения.
SDI приложенията позволяват само един отворен прозорец на документ кадър по кадър. MDI приложенията позволяват няколко документа да бъде отворени в една и съща инстанция на приложението. Приложението MDI има прозорец, в който няколко MDI прозорци, които са прозорци сами по себе си, могат да бъдат отворени, всяка от които съдържа отделен документ. В някои приложения, детето прозорци могат да са от различен тип, като например прозорците на графиките и таблиците прозорци. В този случай, на лентата с менюта може да се променят MDI прозорците на различни видове ,които са активирани.
Осн. развичия в SDI и MDI прилож. Много изгледи в MDIпроблемът за синхронизация на промените в изглед.
Разлики между SDI и MDI:
1.В MDI има повече от един отворен документ, докато в SDI , за да отвори втори трябва да затворим първия.
2.В MDI могат да се поддържат различни типове документи.
3.При MDI в менюто има опция Windows за превключване на прозорците.
4.MDI има поне 2 менюта , а SDI има едно .Първото при отворен документ, а второто при затворен.
5.В SDI има една рамка, а в MDI има главна и дъщерна рамка.
Множество изгледи под един документ в MDI приложения
Използва се многодокументния шаблон- CMultiDocTemplate.Тъй като документа е същия не се създава нов документ, а само нов изглед.Т.е. документа и рамката си остават същите.MFC осигурява списъчна структура за обхождане на всички изгледи(като например при UpdateAllView).Всеки изглед вика Get Document, за да получи данни.Ако в някое View променим данните и искаме промените да се отразят в други View, налага се да се използва UpdateAllView, който пък вика OnUpdate на всеки View.Този метод не се вика автоматично-ние трябва да го осигурим. UpdateAllView обхожда всеки View и ги обновява, като инвалидизира (Invalidate) целия прозорец.Ако искаме оптимизация ние трябва да променим реализацията на UpdateAllView и по-точно да променим параметрите, с които се вика OnUpdate.
В SDI има един обект на приложението на документа.
Това означава , че за да отвори нов документ трябва да затворим стария.
В MDI имаме много документални обекти, затова не е необходимо припокриване.
При избиране на нов се създава нов обект.Ако е ….. Template то се взема от него документен шаблон и се създава , а ако са повече то се извежда та екрана от кой точно шаблон да се вземе.
Отваряне на съществуващ документ.

15. Свързване с база данни. Свързване на данни с контроли (Data Binding). DataGrid. Master-Details.Пример.

База от данни се нарича всяка организирана колекция от данни.

Свързване на данни

Свързването на данни (data binding) осигурява автоматично прехвърляне на данни между контроли и източници на данни. Можем например да свържем масив, съдържащ имена на градове, с ComboBox контрола и имената от масива ще се показват в нея.Всички Windows Forms контроли поддържат свързване на данни (data binding). Можем да свържем което и да е свойство на контрола към източник на данни.

Контролата DataGrid

DataGrid контролата визуализира таблични данни. Тя осигурява навигация по редове и колони и позволява редактиране на данните. Като източник на данни най-често се използват ADO.NET DataSet и DataTable. Чрез свойството DataSource се задава източникът на данни, а чрез свойството DataMember – пътят до данните в рамките на източника. По-важни
свойства на контролата са:

- ReadOnly – разрешава / забранява редакцията на данни.
- CaptionVisible – показва / скрива заглавието.
- ColumnHeadersVisible – показва / скрива заглавията на колоните.
- RowHeadersVisible – показва / скрива колоната в ляво от редовете.
- TableStyles – задава стилове за таблицата.

o MappingName – задава таблицата, за която се отнася дефинира-
ният стил.
o GridColumnStyles – задава форматиране на отделните колони –
заглавие, ширина и др.

Master-Details навигация

Навигацията "главен/подчинен" (master-details) отразява взаимоотношения от тип едно към много (например един регион има много области). В Windows Forms се поддържа навигация "главен/подчинен". За да илюстрираме работата с нея, нека разгледаме един пример: Имаме DataSet, съдържащ две таблици – едната съдържа имена на държави, а другата – имена на градове. Те са свързани помежду си така, че на всяка държава от първата таблица съответстват определени градове от втората
таблица:

Тогава можем да използваме две DataGrid контроли – първата, визуализираща държавите, а втората, визуализираща градовете, съответстващи на текущо избраната държава от първата контрола. За целта контролите се свързват с един и същ DataSet. На главната контрола се задава за източник на данни главната таблица. На подчинената контрола се задава за източник на данни релацията на таблицата:

```
// Bind the master grid to the master table
DataGridCountries.DataSource = datasetCountriesAndTowns;
DataGridCountries.DataMember = "Countries";
// Bind the detail grid to the relationship
DataGridTowns.DataSource = datasetCountriesAndTowns;
DataGridTowns.DataMember = "Countries.CountriesTowns";
```

17. Вход/Изход в .NET. Работа с файлове, директории, потоци, четци и писци.
Потоците в обектно-ориентираното програмиране са една абстракция, с която се осъществява вход и изход от дадена програма. Потоците в C# като концепция са аналогични на потоците в други обектно-ориентирани езици, напр. Java, C++ и Delphi (Object Pascal).
Потокът е подредена серия от байтове, която служи като абстрактен канал за данни. Този виртуален канал свързва програмата с устройство за съхранение или пренос на данни (напр. файл върху хард диск), като достъпът до канала е последователен. Потоците предоставят средства за четене и запис на поредици от байтове от и към устройството. Това е стандартният механизъм за извършване на входно-изходни операции в .NET Framework.Потоците в .NET Framework се делят на две групи – **базови** и **преходни**. И едните, и другите, наследяват абстрактния клас **System.IO.Stream**, базов за всички потоци.Базовите потоци пишат и четат директно от някакъв външен механизъм за съхранение, като файловата система (например класът **FileStream**), паметта (**MemoryStream**) или данни, достъпни по мрежата (**NetworkStream**). По-нататък ще разгледаме класа **FileStream** в точката "Файлови потоци". Преходните потоци пишат и четат от други потоци (най-често в базови потоци), като при това посредничество добавят допълнителна функци-оналност, например буфериране (**BufferedStream**) или кодиране (**CryptoStream**). По-подробно ще разгледаме **BufferedStream** в точката "Буферирани потоци". За четене на данни от поток се използва методът **int Read(byte[] buffer, int offset, int count)**. Той чете най-много **count** на брой байта от текущата позиция на входния поток, увеличава позицията и връ-ща броя прочетени байтове или **0** при достигне края на потока. Четенето може да блокира за неопределено време. Например, ако при четене от мрежа извикаме метода **NetworkStream.Read(…)**, а не са налични данни за четене, операцията блокира до тяхното получаване. В такива случаи е уместно да се използва свойството **NetworkStream. DataAvailable**, което показва дали в потока има пристигнали данни, които още не са прочетени, т. е. дали последваща операция **Read()** ще блокира или ще върне резултат веднага.

Писане в поток

Методът **Write(byte[] buffer, int offset, int count)** записва в изход-ния поток **count** байта, като започва от зададеното отместване в байтовия масив. И тази операция е блокираща, т.е. може да предизвика забавяне за неопределено време. Не е гарантирано, че байтовете, записани в потока с **Write(…)**, са достигнали до местоназначението си след успеш-ното изпълнение на метода. Възможно е потокът да буферира данните и да не ги изпраща веднага.Файловите потоци в .NET Framework са реализирани в класа **FileStream**, който вече беше използван в примера за потоци. Като наследник на **Stream**, той поддържа всичките му методи и свойства (четене, писане, позициониране) и добавя някои допълнителни.Четенето и писането във файлови потоци, както и другите по-рядко използвани операции, се извършват както при всички наследници на класа **Stream** – с методите **Read()**, **Write()** и т. н.

Файловите потоци поддържат пряк достъп до определена позиция от файла чрез метода **Seek(…)**.

Четците и писачите (readers and writers) в .NET Framework са класове, които улесняват работата с потоците. При работа например само с файлов поток, програмистът може да чете и записва единствено байтове. Когато този поток се обвие в четец или писач, вече са позволени четенето и записа на различни структури от данни, например примитивни типове, текстова информация и други типове. Четците и писачите биват двоични и текстови. Двоичните четци и писачи осигуряват четене и запис на примитивни типове данни в двоичен вид – **ReadChar()**, **ReadChars()**, **ReadInt32()**, **ReadDouble()** и др. за четене и съответно **Write(char)**, **Write(char[])**, **Write(Int32)**, **Write(double)** – за запис. Може да се чете и записва и **string**, като той се представя във вид на масив от символи и префиксно се записва дължината му – **ReadString()**, респ. **Write(string)**.Текстовите четци и писачи осигуряват четене и запис на текстова инфор-мация, представена във вид на низове, разделени с нов ред. Базови текстови четци и писачи са абстрактните класове **TextReader** и **TextWriter**. Основните методи за четене и запис са следните: - **ReadLine()** – прочита един ред текст.

- **ReadToEnd()** – прочита всичко от текущата позиция до края на потока.

- **Write(…)** – вмъква данни в потока на текущата позиция.
Работа с директории. Класове Directory и DirectoryInfo
Класовете **Directory** и **DirectoryInfo** са помощни класове за работа с директории. Ще изброим основните им методи, като отбележим, че за **Directory** те са статични, а за **DirectoryInfo** – достъпни чрез инстанция.
- **Create()**, **CreateSubdirectory()** – създава директория или подди-ректория.

- **GetFiles(…)** – връща всички файлове в директорията.

- **GetDirectories(…)** – връща всички поддиректории на директорията.

- **MoveTo(…)** – премества (преименува) директория.

- **Delete()** – изтрива директория.
- **Exists()** – проверява директория дали съществува.
- **Parent** – връща горната директория.
- **FullName** – пълно име на директорията.

18. Windows Presentation Foundation (WPF).XAML.Контроли и логическо дърво. Примери

Нова рендираща система, базирана на DirectX
–Осигурява поддръжка на хардуерно ускорение

–Поддръжка на ефекти

–Вградена поддръжка на 3D
•**Добра интеграция** на 2Dи 3D UI
Независим от резолюцията!

•Декларативно програмиране –XAML

•Добри инструменти за разрабтване на GUI –Blend

•Стилове и теми

•Вградени анимации
•Композиране на елементи

•Разделяне на данните(**Data**)от поведението (**Behavior**)

•Лесно разпространение
–ClickOnce

–Browser(XBAP
XAML:
XML базиран език=>тагове и атрибути

•Декларативен
–Разделение на описание от поведение
•Описва .NETобекти

•Използва се за описване на потребителски интерфейс–работи с класовете от WPFплатформата

Как да създадем бутон:

```
<!--XAML-->
<Button Content="OK"/>
//C#
Button b= newButton() {Content = "Ok‟ };
```

Еквивалентно
Таг –класа на обекта
Атрибут –променя стойност на свойство

Prorerty елементи:

Не създават нови обекти

•Присвоява стойност на свойство
<Button>
<Button.Content>
<RectangleHeight="40"
Width="40"
Fill="Black" />
</Button.Content>
</Button>Таг –класа, собственик на свойството и името на свойството

Основни класове на WPF:

•DispatcherObject

•DependencyObject

•Visual

•UIElement

•FrameworkElementи Control

•Shapesи Text, ContentPresenter

•Control, ContentControl, UserControl

•Window
Контроли на WPF:
•Content Controls
–Buttons
•Button

•RepeatButton

•ToggleButton

•CheckBox

•RadioButton
Items Controls
–ItemsControl

–ListBox

–ListView

–ComboBox

–Menus
•Menu

•ContextMenu

–TreeView

–ToolBar

–StatusBar

19. LINQ – Language INtegrated Query. Query Expressions – Заявки вградени в езика. Ламбда изрази

LINQ – Language INtegrated Query
(заявки вградени в езика)

- Основни фундаменти на LINQ
- LINQ - използвани езици C# 3.0 VB 9
- Особености
    - Lambda Expressions
    - Query Expressions
        - Delegate functions
        - Type inference
        - Anonymous types
        - Extension methods
        - Expression trees

Инициализация на обекти

Invoice i = new Invoice { CustomerId = 123, Name = “Test” };

Is equivalent to:

Invoice I = new Invoice();

i.CustomerId = 123;

i.Name = “Test”;

Lambda Expressions
Predicates

- Predicate
    - (p) => p.Gender == “F”
- Projection
    - (p) => p.Gender ? “F” : “Female”
    - “Each person *p* becomes string “Female” if Gender is “F””

20. Таймер (Timer). Работа с таймери. Пропъртита, събития. Пример.
Таймери
Често в приложенията, които разработваме, възниква необходимост от изпълняване на задачи през регулярни времеви интервали. Таймерите предоставят такава услуга. Те са обекти, които известяват приложението при изтичане на предварително зададен интервал от време. Таймерите са полезни в редица сценарии, например, когато искаме да обновяваме периодично потребителския интерфейс с актуална информация за статуса на някаква задача или да проверяваме състоянието на променящи се данни.
System.Timers.Timer

Класът предоставя събитие за изтичане на времевия интервал **Elapsed**,

което е делегат от тип **ElapsedEventHandler**, дефиниран като:
```
public delegate void ElapsedEventHandler(
object sender, ElapsedEventArgs e);
```
При изтичане на интервала, указан в свойството **Interval**, таймерът от тип **System.Timers.Timer** ще извика записалите се за събитието методи, използвайки нишка от пула. Ако използваме един и същ метод за получаване на събития от няколко таймера, чрез аргумента **sender** можем да ги разграничим. Класът **ElapsedEventArgs** чрез свойството **DateTime SignalTime** ни предоставя точното време, когато е бил извикван метода. За стартиране и спиране на известяването, можем да извикаме съответно **Start()** и **Stop()** методите. Свойството **Enabled** ни позволява да инструктираме таймера да игнорира събитието **Elapsed**. Това прави **Enabled** функционално еквивалентно на съответните **Start()** и **Stop()** методи. Когато приключим с таймера, трябва да извикаме **Close()**, за да освободим съответните системни ресурси.

System.Threading.Timer

System.Threading.Timer прилича на **System.Timers.Timer** и също използва пула с нишки. Основната разлика е, че той позволява малко по-разширен контрол – може да указваме кога таймера да започне да отброява, както и да предаваме всякаква информация на метода за обратни извиквания чрез обект от произволен тип. За да ползваме **System.Threading.Timer**, трябва в конструктора му да подадем делегат от тип **TimerCallback**, дефиниран като:
```
public delegate void TimerCallback(object state);
```
При всяко изтичане на времевия интервал, ще бъдат извиквани методите в този делегат. Обикновено като обект за състояние има полза да подаваме създателя на таймера, за да можем да използваме същия метод за обратни извиквания за обработка на събития от множество таймери.
Другият параметър в конструктора на таймера е времевият интервал. Той може и да бъде променен впоследствие с извикване на **Change(…)** метода. **System.Threading.Timer** не предлага удобен начин за стартиране и спиране. Неговата работа започва веднага след конструирането му (точно след изтичането на подаденото стартово време) и прекъсването му става само чрез **Dispose()**. Ако искаме да го рестартираме трябва да създадем нов обект.

System.Windows.Forms.Timer
Пространството от имена **System.Windows.Forms** съдържа още един клас за таймер, който е със следната дефиниция:
```
public class Timer : Component, IComponent, Idisposable
{
public Timer();
public bool Enabled{virtual get ; virtual set;}
public int Interval {get; set;}
public event EventHandler Tick;
public void Start();
public void Stop();
}
```
Въпреки, че методите на **System.Windows.Forms.Timer** много приличат на тези на **System.Timers.Timer**, то **System.Windows.Forms.Timer** не използва пула с нишки за обратните извиквания към Windows Forms приложението. Вместо това, през определено време той пуска Windows съобщението **WM_TIMER** в опашката за съобщения на текущата нишка. Използването на **System.Windows.Forms.Timer** се различава от употребата на **System.Timers.Timer**, само по сигнатурата на делегата за обратни извиквания, който в случая е стандартният **EventHandler**.

24. Повторно генериране на изключението. Изключения във вложени конструкции.

Повторногенерираненаизключе- ние

- Използва се, когато прихванатото изключение не може да бъде обработено;
- Синтаксис:

```
catch(ExceptionType parameter){
    // ...
    throw;
}
```

- Повторно генериране на изключение може да се из- пълни само в рамките на catch-блок;
- Повторно генерираното изключение се обработва от следващият catch-блок;

Пример: Повторно генериране на из- ключение

```
1  #include <iostream>
2  #include <exception>
3  using namespace std;
4
5  void fun(void) {
6      try {
7          cout<<"Exception thrown in fun()"<<endl;
8          throw exception();
9          cout<<"This should not be printed"<<endl;
10     }
11     catch(exception& ex){
12         cout<<"Exception handled in fun()"<<endl;
```

```
13                  throw;
14              }
15              cout<<"This should not be printed"<<endl;
16      }
```

```
17  int main(int argc,char* argv[]){
18
19      try {
20          fun();
21          cout<<"This should not be printed"<<endl;
22      }
23      catch(const exception& ex){
24          cout<<"Exception handled in main()"<<endl;
25      }
26
27      cout<<"Program can continue"<<endl;
28
29      return 0;
30  }
```

26.Изключения в .NET . Дефиниране на собствено изключение.
Собствени изключения
В .NET Framework програмистите могат да дефинират собствени класове за изключения и да създават класови йерархии с тях. Това осигурява много голяма гъвкавост при управлението на грешки и необичайни ситу- ации. В по-големите приложения изключенията се разделят в логически в категории и за всяка категория се дефинира по един базов клас, а за конкретните представители на категориите се дефинира по един клас- наследник. Създава се по един абстрактен базов клас за категорията изключения, свързани с клиентите (**CustomerException**) и за категорията изключения, свързани с поръчките (**OrderException**). Наследниците на **OrderException** и **CustomerException** също могат да се подреждат в класова йерархия и да дефинират собствени подкатегории.

При работата на приложението, използващо класовата йерархия от примера могат да се прихващат наведнъж всички грешки, свързани с клиентите или само някои конкретни от тях. Това дава добра гъвкавост при управлението на грешките.
Добре е да се спазва правилото, че йерархиите трябва да са широки и плитки, т.е. класовете на изключения трябва да са производни на тип, който се намира близо до **System.Exception**, и трябва да бъдат не повече от две или три нива надълбоко. Ако дефинираме тип за изключение, който няма да бъде базов за други типове, маркираме го като **sealed**, а ако не искаме да бъде инстанциран директно, го правим абстрактен.

Дефиниране на собствени изключения
За дефинирането на собствени изключения се наследява класът **System. ApplicationException** и му се създават подходящи конструктори и евен-туално му се добавят и допълнителни свойства, даващи специфична информация за проблема. Препоръчва се винаги да се дефинират поне следните два конструктора:
MyException(string message);
MyException(string message, Exception InnerException);

27.Правила за работа с изключения в .NET среда

Правила за работа с изключения

1. разработвате библиотека: ако прихванете всички изкл. , как
разработващия приложение с библиотеката ще знае че нещо се е случило
2. разработвате библиотека с типове – не винаги знаете кое е грешка, кое не. Оставете това на викащия
3. Избягвайте код , прихващащ всичко: catch(System.Exception) {……}
4. Ако операция е частично завършена изключение и следва възстановяване в начално съст.: най-добре прихванете всичко, възстановете и уведомете (с друго изкл.) викащата страна.
5. След прихващане и обработка на изключение, често е добре да уведомите извикващия: подавате същото (само с throw) или друго изключение (това е начина за преобразуваме изключението от нещо специфично, към общоразбираемо за потребител).

Необработвани съобщения (такива, които никой catch не разпознава)

Най напред следва да се разработи единна политика за тях – напр. въведен текст се съхранява и се визуализира диалогов прозорец с информация и т.н.;

-1.При отдалечено викана процедура или web услуга или сървърно-базиран код, който подава exception, то той се изпълнява в сървърно обкръжение на try/catch. Тъи като exception обекта е сериализиран, той може да се предава през граница на Domain – т.е. обратно към клиентското приложение.

-2. В общия случай, необработени съобщения могат да се насочват за обработка към дефинирана в в приложението делегатна функция, регистрирана като event handle от тип System.UnhandledExceptionEventHandler към стандартния тип за изключния: System.AppDomain.UnhandledException . Пример:

```
AppDomain.CurentDomain.UnhandledException +=
new UnhandledExceptionEventHandler(MyUnhandledExceptionFunction);
```

3. Необработваните изключения в приложения, базирани на Windows Forms се прихващат така: цялата WinProc ф-ия всъщност, се вика в обхващащ я автоматично try/catch.

При наличие на необработено по-долу изключение, catch блокът извиква виртуалния метод OnThreadException() дефиниран в System.Windows.Forms.Control и предефиниран в Application

Той визуализира стандартен прозорец за 'unhandled exception'

Можете да предефинирате поведението чрез ваш метод от делегатен тип
System.Threading.ThreadExceptionEventHandler
и след това да свържете този метод с ThreadException събитието на класа Application

-4. Необработени съобщения в ASP.NET

ASP обхваща кода на приложението в собствен try блок и предопределя начин за обработка. Може да се намесите като регистрирате свой callback метод към събитие Error на класа System.Web.UI.Page или на клас System.Web.UI.UserControl
(методът може и да се вика за всяко необработено изключение от която и да е страница на приложението – ако callback метода е свързан с Error събитие на клас
System.Web.HTTPApplication)

5. Необработени изключения в среда ASP.NET XML

Отново обхващащ кода try блок на ASP.NET подава SoapException обект. Той се сериализира в XML вид и може да се предава към друг компютър или приложение, работещо като клиент на XML Web услугата.

30. Същност на механизма на сериализация. Сериализиране на обекти с вградени класове. Сериализация

В съвременното програмиране често се налага да се съхрани състоянието на даден обект от паметта и да се възстанови след известно време. Това позволява обектите временно да се съхраняват на твърдия диск и да се използват след време, както и да се пренасят по мрежата и да се възстановяват на отдалечена машина.

Проблемите при съхранението и възстановяването на обекти са много и за справянето с тях има различни подходи. За да се намалят усилията на разработчиците в .NET Framework е изградена технология за автоматизация на този процес, наречена **сериализация**. Нека се запознаем по-подробно с нея.

Какво е сериализация (serialization)?

Сериализацията е процес, който преобразува обект или свързан граф от обекти до поток от байтове, като запазва състоянието на неговите полета и свойства. Потокът може да бъде двоичен (binary) или текстов (XML).

Запазване на състоянието на обект

Сериализацията се използва за съхранение на информация и запазване на състоянието на обекти. Използвайки сериализация, дадена програма може да съхрани състоянието си във файл, база данни или друг носител и след време да го възстанови обратно.

можем да сериализираме обект
и да го запишем в бинарен файл със средствата на .NET Framework:

```
string str = ".NET Framework";
BinaryFormatter f = new BinaryFormatter();
using (Stream s = new FileStream("sample.bin", FileMode.Create))
{
f.Serialize(s, str);
}
```

При сериализирането на обекта в потока се записват името на класа, името на асемблито (assembly) и друга информация за обекта, както и всички член-променливи, които не са маркирани като **[NonSerialized]** (употребата на този атрибут ще обясним по-нататък в тази тема). При десериализацията информацията се чете от потока и се пресъздава обектът.

Методи за сериализация

public static MemberInfo[] GetSerializableMembers(Type)

Методът приема като параметър типа на класа, който ще бъде сериализиран, и връща като резултат масив от **MemberInfo** обекти, съдържащи информация за сериализируемите членове на класа.

public static Object[] GetObjectData(Object, MemberInfo[])

Методът приема като параметри обект, който ще бъде сериализиран и масив с членовете, които трябва бъдат извлечени от обекта. За всеки от тях се извлича стойността, асоциирана с него в сериализирания обект и тези стойности се връщат като масив от обекти. Дължината му е същата, като дължината на масива с членовете, извличани от обекта.

35. Стратегии на управление на памет и събиране на ‘боклук’ в .NET среда. Алгоритъм за “ събиране
на боклук”

Как работи garbage collector?

Вече беше споменато, че ако добавянето на нов обект би довело до препълване на хийпа, трябва да се осъществи почистване на паметта. В този момент, CLR стартира системата за почистване на паметта, т.нар. garbage collector. **Всъщност това е опростено обяснение. Garbage collector се стартира когато Поколение 0 се запълни. Поколенията се разглеждат в следващата секция.**

Първото нещо, което трябва да се направи, за да може системата за почистване на паметта да започне работа, това е да се приспят всички нишки на приложението, изпълняващи управляван код. по време на събирането на отпадъци е твърде вероятно обектите да се преместят на нови адреси в динамичната памет, нишките не трябва да могат да достъпват и модифицират обекти докато трае почистването. CLR изчаква всички нишки да достигнат в безопасно състояние, след което ги приспива. Съществуват няколко механизма, чрез които CLR може да приспи дадена нишка. Причината за тези различни механизми е стремежът да се намали колкото се може повече натоварването и нишките да останат активни възможно най-дълго.

Освобождаване на неизползваните обекти

След като всички управлявани нишки на приложението са безопасно "приспани", garbage collector проверява дали в managed heap има обекти, които вече не се използват от приложението. Ако такива обекти съществуват, заетата от тях памет се освобождава. След приключване на работата по събиране на отпадъци се възобновява работата на всички нишки и приложението продължава своето изпълнение. Както вероятно се досещате, откриването на ненужните обекти и освобождаването на ресурсите, заети от тях, не е проста задача. В тази секция накратко ще опишем алгоритъмът, който .NET garbage collector използва за нейното решаване. За да установи кои обекти подлежат на унищожение, garbage collector построява граф на всички обекти, достъпни от нишките на приложението в дадения момент. Всички обекти от динамичната памет, които не са част от графа се считат за отпадъци и подлежат на унищожаване. Възниква въпросът как garbage collector може да знае кои обекти са достъпни и кои не? **Корените на приложението** са точката, от която системата за почистване на паметта започва своята работа.

Корени на приложението

Всяко приложение има набор от корени (**application roots**). Корените представляват области от паметта, които сочат към обекти от managed heap, или са установени на **null**. Например всички глобални и статични променливи, съдържащи референции към обекти се считат за корени на приложението. Всички локални променливи или параметри в стека към момента, в който се изпълнява garbage collector, които сочат към обекти, също принадлежат към корените. Регистрите на процесора, съдържащи указатели към обекти, 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 в настоящата глава.

Засега просто приемете че тази опашка е част от вътрешните структури, поддържани от CLR и се счита за един от корените на приложението). Когато JIT компилаторът компилира IL инструкциите на даден метод в процесорни инструкции, той също съставя и вътрешна таблица, съдър- жаща корените за съответния метод. Тази таблица е достъпна за garbage collector. Ако се случи garbage collector да започне работа, когато методът се изпълнява, той ще използва тази таблица, за да определи кои са корените на приложението към този момент. Освен това се обхожда и стекът на извикванията за съответната нишка и се определят корените за всички извикващи методи (като се използват техните вътрешни таблици). Към получения набор от корени, естествено, се включват и тези, намира- щи се в глобални и статични променливи.Трябва да се помни, че не е задължително даден обект да излезе от обхват за да бъде считан за отпадък. JIT компилаторът може да определи__ кога този обект се достъпва от кода за последен път и веднага след това го изключва от вътрешната таблица на корените, с което той става кандидат за почистване от garbage collector. Изключение правят случаите, когато кодът е компилиран с /**debug** опция, която предотвратява почист- ването на обекти, които са в обхват. Това се прави за улеснение на процеса на дебъгване – все пак при трасиране на кода бихме искали да можем да следим състоянието на всички обекти, които са в обхват в дадения момент.

Алгоритъмът за почистване на паметта

Когато garbage collector започва своята работа, той предполага че всички обекти в managed heap са отпадъци, т.е. че никой от корените не сочи към обект от паметта. След това, системата за почистване на паметта започва да обхожда корените на приложението и да строи граф на обектите, достъпни от тях.

Нека разгледаме примера, показан на следващата фигура. Ако глобална променлива сочи към обект A от managed heap, то A ще се добави към графа. Ако A съдържа указател към C, а той от своя страна към обектите D и F, всички те също стават част от графа. Така garbage collector обхожда рекурсивно в дълбочина всички обекти, достъпни от глобалната промен- лива A:

Когато приключи с построяването на този клон от графа, garbage collector преминава към следващия корен и обхожда всички достъпни от него обекти. В нашия случай към графа ще бъде добавен обект E. Ако по време на работата garbage collector се опита да добави към графа обект, който

вече е бил добавен, 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 (например A сочи към B, B към C, C към D и D обратно към A).
След обхождането на всички корени на приложението, Графът съдържа всички обекти, които по някакъв начин са достъпни от приложението. В посочения на фигурата пример, това са обектите A, C, D, E и F.

Всички обекти, които не са част от този граф, не са достъпни и следователно се считат за отпадъци. В нашия пример това са обектите B, G, H и I. След идентифицирането на достъпните от приложението обекти, garbage collector преминава през хийпа, търсейки последователни блокове от отпадъци, които вече се смятат за свободно пространство. Когато такава област се намери, всички обекти, намиращи се над нея се придвижват надолу в паметта, като се използва стандартната функция **memcpy(…)**. Крайният резултат е, че всички обекти, оцелели при преминаването на garbage collector, се разполагат в долната част на хийпа, а **NextObjPtr** се установява непосредствено след последния обект. Фигурата показва състоянието на динамичната памет след приключване на работата на garbage collector.

36. Финализация в .NET среда.
Какво е финализация?
Накратко, финализацията позволява да се почистват ресурси, свързани с даден обект, преди обектът да бъде унищожен от garbage collector. Обяснено най-просто, това е начин да се каже на CLR "преди този обект да бъде унищожен, трябва да се изпълни ето този код".
За да е възможно това, класът трябва да имплементира специален метод, наречен **Finalize()**. Когато garbage collector установи, че даден обект вече не се използва от приложението, той проверява дали обектът дефинира **Finalize()** метод. Ако това е така, **Finalize()** се изпълнява и на по-късен етап (най-рано при следващото преминаване на garbage collector), обектът се унищожава. Този процес ще бъде разгледан детайлно след малко. Засега просто трябва да запомните две неща:
- **Finalize() не може да се извиква явно**. Този метод се извиква само от системата за почистване на паметта, когато тя прецени, че даденият обект е отпадък.
- Най-малко **две** преминавания на garbage collector са необходими за да се унищожи обект, дефиниращ **Finalize()** метод. При първото се установява че обектът подлежи на унищожение и се изпълнява финализаторът, а при второто се освобождава и заетата от обекта памет. Всъщност в реалния живот почти винаги са необходими повече от две събирания на garbage collector поради преминаването на обекта в по-горно поколение.

37 Модел на явна финализация в .NET среда.Интегриране на Finalize() и Dispose()
Когато се създава нов обект, CLR проверява дали типът дефинира **Finalize()** метод и ако това е така, след създаването на обекта в динамичната памет (но преди извикването на неговия конструктор), указател към обекта се добавя към Finalization list. Така Finalization list съдържа указатели към всички обекти в хийпа, които трябва да бъдат финализирани (имат **Finalize()** методи), но все още се използват от приложението (или вече не се използват, но още не са проверени от garbage collector).
Създаването на обект, поддържащ финализация изисква една допълнителна операция от страна на CLR – поставянето на указател във Finalization list и следователно отнема и малко повече време.
Взаимодействието на garbage collector с обектите, нуждаещи се от финализация, е твърде интересно. Нека разгледаме следния пример.
Фигурата по-долу показва опростена схема на състоянието на динамичната памет точно преди да започне почистване на паметта. Виждаме че хийпът съдържа три обекта – A, B и C. Нека всички те са от Поколение 0. Обект A все още се използва от приложението, така че той ще оцелее при преминаването на garbage collector. Обекти B и C, обаче, са недостъпни от корените и се определят от garbage collector-а като отпадъци.

И така, garbage collector първо определя обект B като недостъпен и следователно – подлежащ на почистване. След това указателят към обект B се изтрива от Finalization list и се добавя към опашката Freachable. В този момент обектът се **съживява**, т.е. той се добавя към графа на достъпните обекти и вече не се счита за отпадък. Garbage collector пренарежда динамичната памет. При това обект B се третира както всеки друг достъпен от приложението обект, в нашия пример – обект A.
След това CLR стартира специална нишка с висок приоритет, която за всеки запис във Freachable queue изпълнява **Finalize()** метода на съответния обект и след това **изтрива записа от опашката**.
При следващото почистване на Поколение 1 от garbage collector, обект B ще бъде третиран като недостъпен (защото записът вече е изтрит от Freachable queue и никой от корените на приложението не сочи към
обекта) и паметта, заемана от него ще бъде освободена. Забележете, че тъй като обектът вече е в по-високо поколение, преди това да се случи е възможно да минат още няколко преминавания на garbage collector,

Интерфейсът **IDisposable** се препоръчва от Microsoft в тези случаи, в които искате да гарантирате моментално освобождаване на ресурсите (вече знаете, че използването на **Finalize()** не го гарантира).
Използването на **IDisposable** се състои в имплементирането на интерфейса от класа, който обвива някакъв неуправляван ресурс и освобождаването на ресурса при извикване на метода **Dispose()**.n