1. Въведение в програмирането в среда Windows. Вход чрез опашки и съобщения. Независим интерфейс. Многозадачност. Управление на паметта.

спомени от DOS (конзолно приложение):

**програма системни средства на DOS -управление на файл; -управление на памет; -управление на изч. процес; -управление на драйвери; -вход/изход; -обработка на грешки.**

**апаратна част BIOS драйвери**

BIOS драйвери - системни ср-ва в DOS:

-управление на файл;

-управление на памет;

-управление на изч. процес;

-управление на драйвери;

-обработка на грешки;

-вход/изход.

Системните ср-ва се използват само ако програмата ги вика. Също се използва функционалност от драйвери и BIOS комуникацията може да се осъществи директно, или чрез посредничеството на драйвери, BIOS. Директното е чрез директно свързване с апаратната част.

Програмна среда на windows:

Апаратна част: Win компоненти

Памет GDI

Диск Kernel

User

Драйвери на Win (display, printer, kbd).

BIOS файлова с-ма:

Windows компонентите използват драйвери или директна комуникация с апаратната част. Тези компоненти са:

- GDI- у-ва за графичната среда;

- Кernel- у-ва за изчислителния процес и паметта;

- User- у-ва за потребителски интерфейс и диалозите. Всичко минава през Windows, който менажира задачите, заявките и т.н.

Разликата между Windows и DOS е че в Windows винаги се минава през Windows за да се получи достъп навсякъде.

Особенности на ОС:

1. Вход чрез опашки и съобщения- отказваме се от DOS. Идеята е директно да извикаме ф-я или основно изчисли-телния процес да се изпълнява после-дователно. В DOS програмата лесно се проследява, но става по-трудно четима.

2. В Windows се отказваме от директно извикване, последователното изпълнение на командите във файла.

Kbd таймер мишка

А.EXE

WINDOWS B.EXE

Ако имаме приложение А.ЕХЕ, то стрелките са от къде можем да получаваме съобщения - от самата себе си, от клавиатурата, от мишката, от софтуерни таймери и т.н. Съобщението е ср-во за насочване на изчислителния процес за изпълнение на определен програмен код, който е фрагмент от програмата. Приложвнието в Windows се състои от множество фрагменти, които се изпълняват ако са изпълнени определени събития (дошли са определени съобщения). Съобщението е 1 число, което има буквен еквивалент в Windows. Има дефинирани около 150 съобщения които могат да се обработят или не. Всяко съобщение е съпътствано от параметри (поне 2). Msgто идентифицира събитието, а параметрите дават допълнителна инф за msgто. Ако msgто е свързано с промяна размера на прозореца, то някъде ще се съдържа новата инф за размерите и след това ще се прерисува. Има независимост от устройство за изходен интерфейс. Изхода обикновено е на екран, но може да е на диск, интернет, принтер. Изхода трябва да е еднакъв т.е. каквото се вижда на екрана това да излезе на принтера (примерно). Въпросът е да не се налагат промени. Ако се смени устройствения контекст (DC) в рамките на който се прави изхода, просто ще се насочи в др.посока. Мощен клас за управление в MFC е CDC (Device Context). DC предоставя служебно създадена структура от Д, в която има набор от функции чрез които могат да се настройват у-ва. Когато се направи Device Context, а то трябва да се направи преди изхода, тогава изхода се извежда към него.

Многозадачност.

В DOS няма многозадачност. Първите версии на Windows (3.0, 3.11, 95) не са многозадачни. 98 е преходен Windows. Windows NT и 2000 са многозадачни.

Характеристики на многозадачна ОС:

-Възможност за чист паралелизъм (ако хардуера го позволява) или превключване без намеса;

-Изолирани адресни пространства;

-Защити;

-Права на достъп;

-Приоритети;

-Преходи в състоянията ня процесите (отложен, блокиран и т.н.);

Правило: изчислителният процес се управлява от поток приложения и всеки процес има собствено приложение => превключване на задачи става само ако се изчерпат msgтата в опашката на 1вото, а се появят в опашката на другото msg.

Усъвършенстване управлението и извикването на файлове.

Има много механизми, които позволяват качване и сваляне на фрагменти от програмата, което позволява разширение на паметта (виртуалната памет, swap и т.н., асоциране на файл с ОП.

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

* Компонентният модел на .NET
  + позволява дефиниране на поведението на компонентите
    - по време на дизайн (design-time behavior)
    - по време на работа (runtime behavior)
  + осигурява лесна преизползваемост (reusability)
  + дефинира компоненти и контейнери
* CLR осигурява междуезикова съвместимост на компонентите
* Основната функционалност на компонентния модел се намира в System.ComponentModel

6. Обща класификация на класове в MFC среда. Структура на Windows приложение в MFC среда. Основни класове. Последователност при конструиране на класовете. Изчислителен процес.

**. Обща класиф-я на класовете в MFC среда.**

1.Общи класове - това са групите класове за управление и промяна на файлове, низове, изключителни ситуации и често използвани класове.

2. Класове, които управляват видимости (меню, графични обекти, контекстни у-ва, прозорци, диалози).

3. Класове, които структурират приложение:

- класове, св. с двуслойната архитектура и техните подразделения;

-класове, св с упр-е на нишката;

-класове, св. с упр-е на блокировките;

4. Класове колекции – служат като контейнери за съхраняване и възстановяване на други обекти.

5. OLE класове.

6. Database класове – 7-8 класа свързани с обработката на структурирани данни- извличане, корекция, добавяне на записи, изкл.ситуации, SQL, изобразяване на записи.

7. Контролите- 20,30 класа.

8. Window Socket класове – тук се включват класовете за работа с клиент-сървър приложения, работа в мрежи

Структура на приложение с MFC подкрепа:

Hello.h

Class CMyApp:public SWinApp

{public: virtual BOOL InitInstance();};

class CMainWindow:public CFrameWnd

{public:CMainWindow();

protected: afx\_msg void OnPaint();

DECLARE\_MESSAGE\_MAP()};

Hello.cpp

#include<afxwin.h>

#include “Hello.h”

CMyApp myApp; //обект на приложението- той е само един в MFC приложението и има глобална видимост

BOOL CMyApp::InitInstance() //създава обект от класа CMainWindow и копира неговия адрес в член-променливата m\_pMainWnd

{m\_pMainWnd=new CMainWindow;

m\_pMainWnd->ShowWindow (m\_nCmdShow)//показва прозорецът

m\_pMainWnd-> UpdateWindow();

return TRUE;}

BEGIN\_MESSAGE\_MAP (CMainWindow, CFrameWnd)

ON\_WM\_PAINT()

END\_MESSAGE\_MAP()

CMainWindow::CMainWindow()

}Create (NULL,\_T(“The Hello Application”));}

void CMainWindow::OnPaint()

{CPaintDC dc(this);

CRect rect;

GetClientRect(&rect);

dc.DrawText({T(“Hello”),-1,&rect, DT\_SINGLELINE\_CENTER);}

Към приложението:

Със ChelloApp the Application – започва програмата;

AfxWinMain е първата използвана функция. Тя е скрита и не я пишем. Тази функция вика първата функция от нашето приложение. ChelloWnd.

Всяко приложение се структурира в нов thread и няма previous instance. Тогава се стартира нов thread за приложението и се структурира нова опашка от която после се теглят съобщенията.

Следващото което се изпълнява е pMainWnd. InitIntrance създава обекти от ChelloWnd като първо изпълнява конструктора m\_pMainWnd = new(HelloWnd). След коетосъздава прозореца и после го визуализира. Тук нямаме Proc функция която да насочи изчисленията. Тук това става чрез картата на съобщенията. Чрез картата на съобщенията, възникващите съобщения се насочват към обработващите ги функции. Картата на съобщенията автоматично свързва съобщението с метода. В случая: WM\_PAINT с On\_Paint().

CframeWnd е наследник на FrameWnd, който създава toolbar и т.н. и е от по-висока инстанция.

Create има около 10 параметъра аналози на параметрите в С++ (за Windows).

**8 Методика на изгр. на работоспособно приложение във визуална среда.**

Cdocument CView display printer аrchive data.

Едно изображение може да работи с множество документи, както и един документ – в много изображения.

Едно приложение във визуална среда има следните класове:

1. Документен клас (Cdocument) – в него систематизирана цялата обработка на данните.

2. Клас изображение (CView) – неговите методи са отговорни за изобразяване на данните, входа и изхода.

3. Клас Frame – това е най-голямото изображение. Това е мястото където се съдържа клиентската област на изображението

4. Клас на приложението – този клас обединява останалите и осъществява взаимодействието със средата (Windows).

С тези класове се генерират имена и файлове, к. изграждат приложението. ! да се добави код, за да стане функционална програма.

Търсим метод подходящ за CHelloView класа. Този метод ще е On Draw. Той е метод на базовия клас, вика се винаги, когато има необходимост от прерисуване

Hello World във визуална среда. Използваме метода OnDrаw:

void CHelloView::OnDraw(CDC\* pDC){

CHelloDoc\* pDoc = GetDocument(); //указател към документа

ASSERT\_VALID(pDoc);

// добавен код

CRect rcClient;

GetClientRect( rcClient );

pDC->DrawText( "Hello World", -1, rcClient, DT\_SINGLELINE|DT\_CENTER |DT\_VCENTER );

}

Взимаме клиентската област

GetClientRect; Попълваме я; извикваме я; компилираме и пускаме.

Курсор на мишка – изображение на мишката;

1.Създаване на курсор чрез WNDCLASS :

CString strWndClass = AfxRegisterWndClass(…,AfxGetApp() LoadStandartCursor( IDC\_CROSS),……..);

//в констр. на CMainWindow…………

Create(strWndClass,…………);

// в CFrameWnd::Create() – при създаване на прозореца – рамка.

2. С API ф-ия ::SetCursor(m\_hCursor);

3. Курсора да е на пясъчен часовник CWaitCursor.

pOldWnd – указател на прозорец който до сега е имал фокуса.

void CMainWindow::OnSetFocus( CWnd\* pOldWnd){………}

Същото е и при OnKillFocus той ще има за параметър новия прозорец, той ще получи фокус. Това важи само за прозорци, които се намират в един и същи thread.

2. Каретка-това е тази чертичка или правоъгълниче, което показва къде ще се изпише буквата. Тя трябва да се управлява от нас.

CWnd има 7 ф-ии за работа с каретка:

CreateCaret създава я от битмап

GetCaretPos;

SetCaretPos;

ShowCaret;

HideCaret;

За разлика от изображението на мишката при каретката е локално, т. е. в рамките на прозореца;

3. Мишка-при движение на мишката WM\_MOUSEMOVE . Съобщенията от мишкатасе отлагат в опашката на прозореца над която е тя, а не на фокусирания.

- При натиснат бутон: WM\_LBUTTONDOWN; WM\_LBUTTONUP;

WM\_LBUTTONDBLCLK;

Как можем да определим кога курсор влиза и излиза от рамката на прозорец:

влизането : в опашка се отлага WM\_MOUSEMOVE.В windows 98 се появи и интересна ф-ия ::TrackMouseEvent() (за да проверим можем ли да я използваме: #define \_WIN32\_WINNT 0x0400 ). С тази ф-ия можем да укажем приложението ( с параметър HWND) да получава съобщния WM\_MOUSELEAVE и WM\_MOUSEHOVER. С Track\_Mouse Exent() - можем да се регистрираме не искаме да получава и информация когато мишката напусне нашият прозорец и когато мишката остава неподвижна за да направим това нещо правим #define WM\_MOUSELEAVE и WM\_MOUSEHOVER нямат привързване към функция т.е. трябва да предефинираме картата на съобщенията. Тези 2 съоб-я не са обвързани към ф-ии. Те са стандартни, но трябва да се редактира КС. Има и друго msg, к. се появява при движение на мишката, без да натискаме бутон. Благодарение на възможността да се капсулира потока на msgs дори ако мишката излезе от прозореца, може да се прихваща msg на мишката при излизане от прозореца. SetCapture показва, над к. прозорец е мишката.

Още едно съобщение свързано с движението на командата MOUSEWILL

Прихващане на мишка:

CWnd::SetCapture()

::ReleaseCapture()

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

Класът System.Windows.Forms.Control е основа на всички графични Windows Forms контроли

Неговите свойства са типични за всички Windows Forms контроли

По-важните свойства на класа Control:

Anchor, Dock – задават по какъв начин контролата се "закотвя" за контейнера си

Bounds – задава размера и позицията на контролата в нейния контейнер

BackColor – задава цвета на фона

ContextMenu – задава контекстно меню (popup menu) за контролата

По-важните свойства на класа Control:

Controls – съдържа колекция от вложените контроли (ако има)

CanFocus – връща дали контролата може да получава фокуса

Enabled – позволява забраняване на контролата (тя става видима, но неактивна)

Font – задава шрифта (име, стил, размер)

ForeColor – задава цвета на контролата

Location – съдържа позицията на контрола в нейния контейнер

Parent – задава контейнер-контролата, съдържаща текущата контрола

По-важните свойства на класа Control:

Size – съдържа размерите на контролата

TabIndex – определя реда при навигация с клавиша [TAB]

TabStop – задава дали контролата може да се фокусира при навигация с [TAB]

Text – задава текст, свързан с контролата

Visible – задава видимост на контролата

По-важни методи на класа Control:

Focus() – фокусира контролата

Hide(), Show() – скрива/показва контролата

**Дъщерни контроли**

Прозорец, к. има специфика в обработването, предопределена от типа и е дъщерен към по общ. Най-често предава msg към родителя си. Най-честото съобщ. е WM\_COMMAND, а най простият контрол е BUTTON.

определяне на родител:

hwndParent = GetParent(hwndChild);

съобщения към родител: SendMessage(hwndParent,message,wParam,lParam);

Бутони (pushbutton, check box, radio, group box)

WM\_COMMAND

SendMessage(hwndButton, BM\_SETSTATE,1,0); // симулира натиск.

Това е прозорец, който има някаква специфика при обработването и се явява дъщерен на един по общ прозорец, най-често съобщения към родителя и най-честото съобщение е window command.

Прихващане на функции (Windows subclassing)

1. lpfnScrollProc = MakeProcInstance ((FARPROC)ScrollProc, hInstance);

2. Чрез GetWindowLong() се извлича адреса на стандартната функция.

3. Чрез SetWindowLong() се установява новия адрес .

4. Връщане към стандартната Windows функция:

CallWindowProc (lpfnStandart, hwnd, message, wParam, lParam);

Контрол за редактиране (edit )

CreateWindow(“edit”, …);

бутонът изпраща WM\_COMMAND с нотификационен код ( в lParam).

Контроли в 32 битови версии

над 20 типа. 6 са “класически”- от първите версии. Имат поддръжка на MFC:

CButton,CListBox;CEdit;

CComboBox; CScrollBar;

CStatic.

Създаване на бутон:

CButton m\_wndPushButton;

m\_wndPushButton.Create(\_T(“Start”), WS\_CHILD | WS\_VISIBLE | BS\_PUSHBUTTON, rect, this, IDC\_BUTTON);

CEdit

CEdit::SetMargins() // задава размерите на полетата вляво и дясно

CEdit::SetRect() // задава редакторската област в контрола

CEdit::SetLimitText() // задава макс. брой символи (подрзабират се 30000.

CEdit::SetWindowText()

CEdit::GetWindowText()

ф-иите са наследени от CWnd

::Clear()

::Cut() за селе-

::Copy() ктиран

::Paste() текст

Пример: edit контрол,

който приема само числови стойности:

class CNumEdit : public Cedit

{protected:

afx\_msg void OnChar(UINT nChar,UINT nRepCnt, UINT nFlags);

DECLARE\_MESSAGE\_MAP(};

BEGIN\_MESSAGE\_MAP(CNumEdit, CEdit)

ON\_WM\_CHAR()

END\_MESSAGE\_MAP()

void CNumEdit::OnChar((UINT nChar, UINT nRepCnt, UINT nFlags)

{

if((( nChar >= \_T(‘0’)) && (nChar <= \_T(‘9’)))) CEdit::OnChar(nChar, nRepCnt, nFlags);

}

Искаме да добавим филтър, к. да да допуска само цифри. И искаме функционалност, подобна на edit контрол. Затова правим клас CNumEdit наследник на CEdit. Трябва да се добави проверка за цифра. Това се добавя в метода свързан с натискане на клавиш(OnChar)За целта се декларира собствена ф-я OnChar с параметри като на стандартна ф-я.

1. Прототип

2. Свързване на ф-ята със съобщението-се добавя записа в нашата карта чрез DECLARE\_ MESSAGE\_MAP

3. Реализация-трябва да се направи карта на съобщенията

Искаме да добавим само една ф-я. Затова към нашия контрол ще постъпват много съобщ., к. той няма да ги обработва, а ще ги препраща към родителския. Само когато се получи OnChar ще го прихване. Записваме ф-ята OnChar, проверяваме дали е цифра от 0 до 9, ако не е нищо. Ако е: филтър и се пренасочва към CEdit.

13. Диалози – стандартни и потребителски. Видове и приложение. Пример за употреба.

Класът MessageBox позволява извеждане на стандартни диалогови кутии:

съобщения към потребителя

въпросителни диалози

Потребителските диалогови кутии се извикват така:

ShowDialog() показва модално диалога

Типът DialogResult съдържа резултата (OK, Yes, No, Cancel и др.)

Задаване на DialogResult:

Автоматично – чрез свойството DialogResult на бутоните

Ръчно – преди затваряне на диалога чрез свойството му DialogResult

В mainform clasa:

**private void ButtonCallDialog\_Click(**

**object sender, System.EventArgs e)**

**{**

**DialogForm dialog = new DialogForm();**

**if (dialog.ShowDialog() == DialogResult.OK)**

**{**

**string userName = dialog.UserName;**

**MessageBox.Show("You entered: " + userName);**

**}**

**else**

**{**

**MessageBox.Show("You canceled the dialog.");**

**}**

**}**

В dialogform clasa: **public string UserName {**

**get { return TextBoxName.Text; }**

**}**

14. SDI и MDI приложения. Структура и пример. Свързване с база данни. Свързване на данни с контроли (Data Binding). DataGrid. Master-Details. Пример.

DataGrid контролата визуализира таблични данни

Осигурява навигация по редове и колони

Позволява редактиране на данните

Слаба функционалност

Ще бъде заменена в бъдещи версии на .NET Framework

Използва се най-често с ADO.NET DataSet и DataTable

DataSource – задава източника на данни

DataMember – задава пътя до данните в рамките на източника

По-важни свойства на DataGrid

ReadOnly – разрешава / забранява редакция

CaptionVisible – показва / скрива заглавието

ColumnHeadersVisible – показва / скрива заглавията на колоните

RowHeadersVisible – показва / скрива колоната вляво от редовете

TableStyles – задава стилове за таблицата

Активен е само първият стил

MappingName – задава таблицата, за която се отнася дефинираният стил

GridColumnStyles – задава форматирането на отделните колони – заглавие, ширина и др.

MDI (Multiple Document Interface) приложенията поддържат работа с няколко документа едновременно

MDI контейнери (MDI parents)

Съдържат други форми

Задава им се IsMdiContainer = true

Обикновено имат меню Window за смяна на активната форма (MdiList = true)

MDI формите (MDI children)

Съдържат се в контейнер-формата

Задава им се MdiParent = <контейнер>

Master-Details навигацията отразява взаимоотношения тип "1 към много"

Например: 1 регион има много области

В ADO.NET DataSet обектите се поддържат релации тип "главен/подчинен"

Използват се DataRelation обектите в DataSet

Windows Forms поддържа Master-Details навигация

На главната контрола се задава за източник на данните главната таблица

На подчинената контрола се задава за източник на данните релацията на таблицата

Контролите се свързват с един и същ DataSet

Primeri:

***// 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";**

**SDI документния шаблон се създава още в InitInstanse():**

**CSingleDocTemplate \* pDocTemplate;**

**pDocTemplate = new CSingleDocTemplate( IDR\_MAINFRAME, RUNTIME\_CLASS( CMyDoc),** **RUNTIME\_CLASS( CMainFrame), RUNTIME\_CLASS( CMyView));**

**AddDocTemplate(pDocTemplate);**

**// следват редове зарег. на док. типове в REG, създаване обектите от шаблона, показване рамката и изгледа.**

**Включва се цикълът за обр. съобщения, като командните съобщения (от меню, акселер. , toolbar) се насочват за обработка към обектите - изглед**🡪 **документ**🡪**шаблон**🡪**рамка**🡪**приложение**🡪**::DefWindowProc() (command routing).**

SDI - приложения определена архитектура (класове); В SDI в дадне момент има един документ и в общия случай едно изображение.

В МDI в даден момент се работи с няколко документа от различен тип.

В началото още се указва какво приложение SDI или МDI ще се прави

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

Vij v GDI.pdf dokumenta

16. Вход/Изход в .NET. Работа с файлове, директории, потоци, четци и писци.

Какво представляват потоците?

Абстракцията "поток" е основният начин за осъществяване на входно-изходна активност в съвременните обектно-ориентирани езици (C#, C++, Java, Delphi)

Потоците:

са подредени серии от байтове

представляват абстрактни канали за данни, до които достъпът се осъществява последователно

предоставят механизъм за четене и писане на поредица байтове от и към устройства за съхранение или пренос на данни

В .NET Framework повечето входно-изходни операции използват потоци.

* В .NET Framework потоците са два вида:
  + Базови потоци (base stream)
    - четат и пишат данни от и към външен механизъм за съхранение на данни
    - примери: FileStream, MemoryStream, NetworkStream
  + Преходни потоци (pass-through streams)
    - четат и пишат в други потоци, като добавят допълнителна функционалност (напр. буфериране, кодиране и компресиране)
    - например: BufferedStream и CryptoStream
* Основните операции с потоци са:
  + конструиране (създаване)
    - потокът се свързва с механизма за пренос/ съхранение на данни или с друг поток
    - като параметър в конструктора на класа се подава информация за този механизъм
    - например при файлов поток се посочва име на файл, а при низов поток – съответен низ
  + четене
    - извличат се данни от потока
    - в зависимост от типа на потока тези данни се извличат по различен начин
    - например при файлов поток данните се прочитат от текущата позиция във файла
  + писане
    - изпращат се данни в потока
    - в зависимост от типа на потока тези данни се изпращат по различен начин
    - например при писане във файл данните се записват във файла от текущата позиция
  + позициониране
    - премества текущата позиция на потока (ако се поддържа позициониране)
    - позиционирането става спрямо текущата позиция, началото или края на потока
    - например при файлов поток се променя текущата позиция във файла
  + затваряне
    - завършва се работата с потока и се освобождават използваните ресурси
    - например при файлов поток се записват данните от вътрешните буфери, които не са все още записани на диска и се затваря файла
  + други операции
    - изпразване на вътрешните буфери (flush)
    - поддържа се и асинхронно четене и писане (което ще разгледаме в темата за работа с нишки и синхронизация)
    - някои специални потоци поддържат и други специфични за тях операции

Четци и писачи:

Четците и писачите са класове, които:

улесняват работата с потоци

позволяват четене и писане на различни структури от данни, например примитивните типове, текстова информация и други типове

биват двоични и текстови

Класовете BinaryReader и BinaryWriter

осигуряват четене и записване на примитивните типове данни в двоичен вид

ReadChar(), ReadChars(), ReadInt32(), ReadDouble(), …

Write(char), Write(char[ ]), Write(Int32), Write(Double), …

позволяват четене и писане на string, като го записват като масив от символи, предхождан от дължината му: ReadString(), Write(string)

Имаме бинарен файл със записи във формат (име: string, възраст: int)

За добавяне и четене на записи можем да ползваме следния код: Класовете TextReader и TextWriter

осигуряват четене и записване на текстова информация (низове, разделени с нов ред)

използват се по същия начин като класа Console (има ReadLine(), WriteLine(…), …)

символът за нов ред е различен за различните платформи:

LF (0x0A) – в Unix и Linux

CR LF (0x0D 0x0A) – в Windows и DOS

ReadLine() – прочита текстов ред

ReadToEnd() – прочита всичко до края на потока

Write(…) – пише текст в потока

WriteLine(…) – пише текстов ред в потока

Класовете TextReader и TextWriter са абстрактни и не се използват директно

Използват се следните класове:

StreamReader – чете текстови данни от поток

StringReader – чете текстови данни от низ

StreamWriter – пише текстови данни в поток

StringWriter – пише текстови данни в низ, използва вътрешно StringBuilder

* Directory и DirectoryInfo са класове, които предоставят функционалност за:
  + създаване на директории и поддиректории – Create(), CreateSubdirectory()
  + извличане на всички файлове – GetFiles()
  + извличане на всички поддиректории – GetDirectories()
  + местене (преименуване) – MoveTo(…)
  + изтриване – Delete()
  + извличане на горната директория – Parent
  + проверка за съществуване – Exists()
  + извличане на пълното име – FullName
* Класът Directory предоставя тези методи статични, а DirectoryInfo чрез инстанция

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

Въведение в XAML - XML

Предимства на XAML:

XAML код е кратко и ясно да се чете

Разделяне на дизайнер код и логика

Графичните инструменти за проектиране като Expression Blend изискват XAML като източник.

18. 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”;

Query Expressions:

Въвеждане на SQL-Like синтаксис Език

Съставител на традиционните C # (с удължители Методи)

**from *itemName* in *srcExpr***

**join *itemName* in *srcExpr* on *keyExpr* equals *keyExpr***

**(into *itemName*)?**

**let *itemName* = *selExpr***

**where *predExpr***

**orderby (*keyExpr* (ascending | descending)?)\***

**select *selExpr***

**group *selExpr* by *keyExpr***

**into *itemName query-body***

Lambda Expressions:

Lambda израз синтаксис:

(argumentList) => израз

oneArgument => израз

Аргументи по избор написали.

Механизъм от типа извод.

Ламбда изразите са естественото продължение на анонимните методи, които бяха въведени в C#

2.0. За да проследим развитието ще започнем от самото начало като обясним накартко делегатите.

Още с първата си версия C# предостави специален вид променлива, която сочи към методи –

делегатите. Това позволява делегати да се предават като параметри, да се променят и да се

извиква сочената функция. Използването на делегати позволява елегантно промяна на

програмния поток в зависимост от функциите, към които сочат.

**Делегати**

В следващият пример са извършени следните операции:

* Дефиниране на делегат Operation, който приема два параметъра.
* Реализиран метод Multiply със същите параметри като делегата Operation.
* В метода се създава променлива deleg, сочеща към инстанция на делегата Operation,

която сочи към метода Multiply. Променливата deleg се извиква като нормален метод.

* **От анонимни методи към ламбда изрази**

Ламбда изразите правят синтаксиса още по сбит. Прототипът е следният:

(Списък с параметри) => тяло на метода

Типът на параметрите не е задължителен и може да се пропусне:

(x, string s) => s.Length > x

Ако тялото на метода съдържа повече от един ред тогава цялото тяло се огражда с {}.

Използвайки лабда изразите горният пример изглежда така:

**static void Main(string[] args){**

**Operation deleg3 = (long i, int j) => {return i \* j; };**

**long res = deleg2(14, 13);**

**{**

Правила за използване на ламбда изрази:

* Скобите на списъка с параметри може да се пропусне, ако параметъра е един - n =>

n<10, освен ако не се указва типа на параметрите – (string n) => n<10.

* Скобите на списъка с параметри e задължителен ако параметрите са повече от един или

няма входни параметри.

* Може да се достъпват променливи от външният метод също като при анонимните методи.

Ламбда изразите са много често използвани в LINQ, както ще видим по късно в това ръководство.

За да покажем силата на тази конструкция ще разгледаме следният код

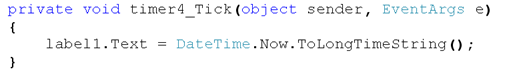
19. Таймер (Timer). Работа с таймери. Пропъртита, събития. Пример.

Използване на Таймер:

* Създаване на приложение
* Добавяне на контрола Timer
* Основни събития (Events)
* Основни свойства (Properties)
* Примери използващи Timer контрол

Използване на таймер за визуализация на дигитален часовник

1. Добавяме контрол Label -
2. Добавяме контрол Timer
3. Задаваме интервал на Timer – 1000ms=1s
4. Определяме функционалността на таймера, която ще се изпълнява на всяка 1 секунда
5. Стартираме Timer



20. Езикови средства за създаване устойчив код. Предоставени от 32 битова среда възможности за целта. Въведение в SEH механизма. Терминираща обработка (\_\_finally). Възможни двусмислия

SEH(Structure Exception Handling) структурирана обр. на изключителни ситуации.

Това е вграден механизъм за обработка на софтуерни(прекъсвания) и хардуерни(деление на нула) изкл. ситуации.

Добре структуриран код: използващ предоставените от средата средства за спряване с изкл. ситуации. Няма толкова проверки, добре блокуван и се знае кое кога ще се изпълни.

В 32 битова среда има следните възможности за прихващане и обработка на exception:

1.Може да се ползва SEH механизма на ОС.

2.Може да се ползва собствения за езика механизъм за обработка на exceptions.

3.Възможна е комбинация то 1 и 2.

При откриване на exception става:

1.Изпълняваната нишка се прекратява.

2.Управлението се предава от usermode в kernelmode (ядрото на ОС поема управлението).

3.Търси се начин за реакция чрез exception блок. Ако има то той се изпълнява, а ако няма диспечера генерира служебен exception handler. Ако има блок exception handler, то:

1. Да се изчисти паметта от излишни данни.

2. Да се възстановят операциите в БД (ако има).

3. Да се освободят заетите ресурси.

4. В LOG файл може да се запише информация.

5. Диалогов прозорец с полезна информация.

1.Терминираща обработка (\_\_finally).

\_\_finally представлява блок за който може да се гарантира че винаги ще се изпълни.

BOOL Myfunc()

{int\* p = 0;

\_\_try{ // използва се p }

\_\_finally{delete p;}

return fReturn;}

finally предшества блок, к. винаги ще се изпълнии то преди края на ф-ята. Ако имаме и finally и exception, то finally се изпълнява след exception блока.

Finally се изпълнява в следните случаи:

1) след изход от try секция

2) след прекратяването на try блока чрез exception

3) при нормален изход

Return по възможност ! да е извън try и finally блоковете. Иначе return може да е в try, finally или извън 2та блока.

. Ако е в try блока връщаната ст-ст не се връща в този момент, т.е компилатора съхранява резултата от връщане в междинна памет и ще го вурне след като handlera се изпълни. Това означава, че ако след finally блока има още 1 return той няма да върне ст-ст.

Единственият начин е да не се изпълни finally е да прекратим нишката(abroad прекъсване).

21. Филтър за обработка на изключения (\_\_except). Събиране информация при събитие.

Той се изчислява и стойността определя начина на реакция при случване на exception.

1)EXCEPTION\_EXECUTE\_HANDLER – term.handler – има стойност 1:

Ще се изпълни блокът след except и управлението ще се подаде след блока except.

2)EXCEPTION\_CONTINUE\_SEARCH (0) –dispatcher ще търси друг. Има стойност 0. Той показва на ОС да пренебрагне нашият exception блок и да търси обхващащ exception блок.

3)EXCEPTION\_CONTINUE\_EXECUTION –пренебрегва (-1) - има стойност –1:

Той пренебрагва exception блока т.е. той не се изпълнява, няма да се търси друг, а управлението ще се предаде на следващата конструкция.

ОС изработва множество идентификатори, които указват причината за exception-а.

\_\_except ( GetExceptionCode() == EXCEPTION\_ACCESS\_VIOLATION )

{

cout << “прихваната изкл. ситуация” <<endl;

}

// опит за четене/запис без инициал. на указател или без съответни права.

По-общ анализ:

\_\_ except( ExceptionFilter( GetExceptionCode()))

където:

int ExceptionFilter( int nException )

{ int nReturn;

switch( nException )

{

case EXCEPTION\_INT\_DIVIDE\_BY\_ZERO:

nReturn= EXCEPTION\_EXECUTE\_HANDLER;

break;

………………….

default: nReturn = EXCEPTION\_CONTINUE\_SEARCH;

}

return nReturn;

}

22. Вграден в С++ механизъм за реакция на изкл. събития (C++ Exception Handling). Стандартни обекти в ANSI C++ стандарт. Обработка на изключения при конструиране на обект.

**Вграден в С++ механизъм за р-я на изключителни събития**

C++Exception Handling

int Divide( int n1, int n2 )

{ int nReturn = 0;

try{if( !n2 ) throw

range\_error();

nReturn = n1 / n2;}

catch ( range\_error& e )

{cout << “опит за / на 0” << endl;}

return nReturn;}

В try блока се генерират изключителни ситуации. Може да се генерира дъщерна изключителна ситуация, а да се прихване родителска. Трябва try и catch да са в една функция. Ако няма catch, то се търси в обхващащата функция дали има catch, ако никъде няма то се извиква terminate. При генериране на изключителни ситуации с throw автоматично се изпълняват деструкторите на всички обекти създадени от началото на try блока.

Standart Exception Library

exeption

logic\_error runtime\_error

domain\_error range\_error

(вътр. грешка) overflow er. invalid\_argument, length\_error,out\_of\_range

- runtime error - групира всички проблеми, к. не могат да се групират при изпълнението

- logic error - групира всички проблеми, к. се пораждат при изпълн. на ф-ята.

SomeFunc()

{try{…throw range\_error( “ невъзможност за заделяне достатъчно памет”);

……}

catch( runtime\_error e)

{cout << e.what() << endl ;

// e.what() – стандартна (извиква съобщение)

}}

Интересен синтаксис е следният:

catch (…) // прихваща за обработка всички exceptions.

Важно е да се обхване чрез try. Конструкторът, не връща стойност т.е. не може да информира за успех или неуспех относно работата му.

{cout << “ ……sssdcg “ << endl;

throw; }}

Windows отделя нишка, к. се занимава с анимации.

Контрол календар

CMonthCalCtrl – този контрол позволява на потребители да въвеждат дати чрез избиране от календар, вместо да ги набират в edit контрол. Той може да поддържа единичен или множествен избор.

ето пример за задаване текущата дата:

m\_wndCal.SetCurSel(CTime(2002, 3, 25, …);

Контрол за избор на дата и час CdateTimeCtrl – контролът прилича на edit, но вместо да показва обикновени текстови низове, показва дати и часове.

24. Съвместяане на механизмите за реакция на изкл. събития (Win32 SEH и C++ Exception Handling).

**Съвместяване двата механизма – Win32 SEH  и C++ Exception handling**

**При SEH може да пропуснем деструкция на обект;**

**В SEH ако не прихванем точните exceptions, може да пропуснем събитие;**

**Повечето С++ библиотечни ф-ии хвърлят С++ exceptions**

***class CWin32Except***

***{ unsigned int m\_nCode;***

***public:***

***CWin32Except( unsigned int nCode ) : m\_nCode(nCode) {};***

***unsigned int Code() const {return m\_nCode;};***

***}***

**използване:**

**void SEH\_MyFunc( unsigned int nCode, EXCEPTION\_POINTERS\* pExp )**

**{ throw CWin32Except( nCode ); }**

***void DoAccessViolation()***

***{ int\* p =0; \*p = 32; }***

**void main(void)**

**{ *\_se\_translator\_function fn\_old;***

***fn\_old = \_set\_se\_translator( SEH\_MyFunc );***

**try**

**{ DoAccessViolation(); }**

**catch( CWin32Except& e )**

**{ cout << “ изключение с CWin32Except обект ” << endl;**

**cout << “Exception is “ << e.Code() << endl;}**

***\_set*\_se\_translator( fn\_old); }**

* 25. Изключения в .NET . Дефиниране на собствено изключение .

**изключението не е грешка. То е просто нарушаване на допусканията за даден програмен интерфейс.**

**отказ от HRESULT**

**изключение не може да се игнорира. Ако прил не прихване обект –изкл., CLR**

**прекратява работата му**

**няколко десетки стандартни типа на изключения, всички наследили System.Exception**

**стек на повикванията за catch.Претърсването става от конкретните (типове, чиито базов клас се намира по-далеч от System.Object) към общите. Така и трябва да се подреждат в кода (иначе компил. генерира грешка, защото ако има по-далечни catch блокове, собственият catch остава недостижим)**

**ако липсва филтър – генерира се “необработено съобщение”**

**след откриване на подходящ catch – изпълняване на всички finally от блока – подал изкл. *докато стигне ( и там спира* ) блока с catch филтъра, поел го за обработка.. След това се изпълнява този catch блок.Неговият finally – след това.**

**в края на catch блока:**

**а. подаване същото изключение** 🡪 **нагоре;**

**б. подава се друго изключение с по-богата информация нагоре;**

**с. нишката излиза от catch** 🡪**към finally, ако има**🡪**към инструкциите след него**.

**“трагични” съобщения: OutOfMemoryException (finally може да се изпълни, може и не); StackOverflowException; ExecutionEngineException**

**System.Exception съдържа полезни свойства: Message; Source; StackTrace (списък на виканите методи до този – подал изкл); TargetSite (метода); HRESULT**

**обща йерархия в .NET Framework Class Library (FCL):**

**System.Exception**

**= System.ApplicationException**

**= System.SystemException**

**подавайте конкретен клас изключение ( който няма производни класове)**

**Не наследявайте други потребителски изключения – съществуващият код ще прихваща и Вашето ново изключение, което ще е грешно.**

**йерархиите exceptions да са широки , но плитки. Тези без производни – sealed**

***Дефиниране на собствено изключение***

**1. да има 3 public конструктора:**

**а. без параметри (по подразбиране)**

**б. с параметър string – задаващ съобщение**

**в. с параметър String и инстанция на производен на Exception тип ( за генерирано вътрешно съобщение)**

**Винаги викайте в тях и конструктора на базовия тип изключение!**

**2. наследява Exception (пряко или през други – това е ваше решение!)**

**3. може да има и други констуктори**

**4. винаги да е сериализуем за да може изкл. да се предава през граница на процес, машина или кл. приложение или в log.. За целта:**

**- [Serializable] атрибут**

**- ако има дефинирани свои даннови членове: да наследява ISerializable; да имплементира метода му GetObjectData(); да има конструктор с 2 фиксирани парам. – SerializationInfo и StreamingContext**

**Структура. В property се указва откъде/накъде**

**се сериализира.**

26. Правила за работа с изключения в .NET среда

**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 🡪подава изключение 🡪 и следва възстановяване в начално съст.: най-добре сега прихванете всичко, възстановете структурите си 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).**

**Внимание: CLR управлява изключенията така: първо намира catch филтър, който може да е доста нагоре в стека. След това “развива” стека на вложените изкл. като изпълнява всички finally блокове , до и без този съответстващ на прихващащия филтър.Следва изпълнение на catch блока и след това неговия finally.**

***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.**

**Всичко е ОК!**

* **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 в йерарйията изключение, catch блокът извиква**

**виртуалния метод OnThreadException() дефиниран в System.Windows.Forms.Control**

**и предефиниран в класа 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* (з страницата) или на клас *System.Web.UI.UserControl (за контрол)*

Методът Ви може и да се вика за всяко необработено изключение от която и да е страница на

Вашето приложение – ако callback методът 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. XML кодът на този обект се десериализира в SoapException

обект отново и се подава към нишката на клиента за да бъде изключението прихванато.

27. Вход/изход и сериализация. Работа с файлови обекти без сериализация. Цел и предимства на архивните обекти. Предефиниране на операциите за сериализация.

**Вход/изход и сериализация**

**Работа без сериализация** – базов клас CFile.

CFile myfile;

CFileException\* e;

if ( file.Open( \_T(“My.txt”), CFile::modeReadWrite, &e))

{// работим с файла}

else{………e.ReportError();}

същото може и така:

try{CFile file (\_T(“MyFile.txt”), CFile::modeReadWrite);…..}

catch( CFileException& e)

{eReportError();

eDelete();}

CFile – капсулира всички AVI ф-ии. Има над 25 м-да-за всички възможни метода над файла.

- **отваряне на файл** – има предвид “е” клас. Декл. ме обект от CFile exception. Отваряне на файл в режим за четене и запис. Може с try и catch. При отваряне на файл се показва режима.

- **затваряне:**

А. file.Close();

В. CFile обект се затваря авт. при излизане извън обсег.

- **четене/ запис**

BYTE buff[0x4000];

CFile file(………………);

DWORD length = file.GetLenght();

while(length)

{ UINT nByteRead = file.Read(buff, sizeof(buff));

lenght -= nByteRead;

}

**- file.Write(buff, nByteRead);**

// записва определен брой байтове от буфера

**- file.Seek**( относит. отместване в байтове, спрямо какво)

- препоръчително е четенето да се обхване oт try catch

- изтриване (Remove())

- преименоване на файлове (Rename())

**Производни на CFile класове:**

Методите за работа с CFile класове могат да се използват и в др. случаи. Напр. MemFile и SharedFile, DataObject – пренасянето на данни м/у приложения, може да се стандартизира чрез него, а за програмата си е все същото. SocketFile – предоставя файл с подобен интерфейс. Stdio File – специален файл към текстови данни.

Internet File – все едно се работи с файл, но на практика се работи с протоколите от нета.

CMemFile, CSharedFile

COleDataObject::GetFileData

CSocketFile, стои между архивния обект и CSocket обекта.

CStdioFile , наследник на CFile.

Пример:

try {

CStdioFile

file(\_T(“My.txt”), CFile::modeRead);…..}

catch( CfFileException\* e){}

CInternetFile, CGopherFile и CHttpFile .

Изброяване на файлове и директории

:: FindFirstFile

:: FindNextFile()

:: FindClose()

Универсален подход на I/O – използване на архиви (базов клас CArchive)

Ето пример: искате в отоворен файл да запишете 2 променливи a,b:

file.Write( &a, sizeof(a));

file.Write( &b, sizeof (&b));

Ето другият подход:

CArchive ar( &file, CArchive::store);

ar << a << b;

Архива е един междинен обектмежду нас и запомнящата среда, скриващ спецификацията на запомнянето, а представящ на нас само стандартният диск. Едни и същи операции за вход/изход да бъдат използвани за данни с различен тип, независимо от средата. За да се осъществи това посредата ! да се постави нещо. Архива е винаги между нашето приложение и запомнящата среда. Функцията на междинният слой е като един транслатор. Най-често архива се асоцира с диск. Първо създаваме архива – обект, асоцираме го със запомнящата среда. Командите за вход/изход трябва да се предефинират за нашите данни и обекти. Всички примитивни типове имат предефинирани в MFC операции за << >> (BYTE, WORD, LONG, DWORD, float, double, intchar, char, unsigned int). Оперторите са предефинирани и за непримитивни типове, за които има стандартни MFC класове:

пример: Cstring string; ar<<string;

също и за: Ctime, Crect, Csize, ColeDateTime, cole variant …, както и за структури SIZE, POINT, RECT.

Ако се създават собствени класове и ако искаме те да поддържат всички обекти от тези класове и ако направим класа наследник на Cobject тогава не е нужно да предефинираме командите за вход/изход.

28. Създаване на собствени класове, поддържащи сериализация. Пряко използване на Serialize().

Ако създаваме собствени класове и ако искаме те да поддържат променливи от този клас и ако направим този клас наследник на CObject, тогава не е неоход. да предеф. командите, компилатора автом. го прави.

- Новосъздаденият клас наследява CObject;

- В декларацията на класа DECLARE\_SERIAL( име на класа)

- Предефинирвате Serialize() на базовия клас и сериализирате данновите членове на производния клас (ще поясним по-късно).

- Подразбиращ се конструктор на класа.

- В частта имплементация на класа IMPLEMENT\_SERIAL

Конструктор по подразбиране – това е конструктор без параметри. Когато класът поддържа сериализация този конструктор задължително трябва да го има. Динамичното създаване на oбекти използва винаги подразбиращият конструктор.

>>оператор за сесериализиране, <<оператор за сериализиране.

**Кога се използва Serialize():**

- При десериализиране на конструиран обект, не знаем името и сме заделили памет.

- Ако обектът е сериализиран със Serialize() трябва да се десериализира със Serialize().

- Ако създаваният клас има даннов елемент проявление или наследник на класа CОbject също се използва Serialize(), защото операциите << >> са дефинирани в CОbject да работят с указатели към обекти, а не с обекти.

- Ако сериализираният клас има даннов елемент указател към Cоbject или негов наследник, тогава имаме 2 възможности

1. Първо трябва да се конструира данновият обект във функцията за сериализация и след това да се използва указателя. Използването става по 2 начина:

PObject // указател

- PОbject ->Serialize();

- ar>>m.mydata>>p.Object

IMPLEMENT\_SERIAL – този макрос иска три параметъра.

пример:

class CLine : public CObject

{DECLARE\_SERIAL

(CLine)

protected: CPointm\_ptFrom; CPointm\_ptTo;

public: CLine(){}

CLine( CPoint from, CPoint to)

{m\_ptFrom = from; m\_ptTo = to;}

void Serialize (Carchive&ar );};

Kато дефинираме ф-ията Serialize така:

void CLine::Serialize( CArchive& ar)

{CObject::Serialize( ar);

if ( ar.IsStoring())

ar << m\_ptFrom << m\_ptTo;

else ar >> m\_ptFrom >> m\_ptTo;}

Добавяме и някъде в имплементацията:

IMPLEMENT\_SERIAL( CLine, CObject, 1)

IMPLEMENT\_SERIAL( CLine, CObject, 1 | VERSIONABLE\_SCHEMA)

В примера са спазени всички правила за сериализация описани по-горе.

29. Версии при сериализации. Работа с указатели към обекти и проблеми при използване на псевдоними на обекти при сериализация.

Ако след време искаме да променим нашия клас, тогава ! да преопишем Serialize, но пък написните преди това програми ! да се информират че версията е различна. Това става като се сравнят номерата на проектите: IMPLEMENT\_SERIAL(cLine, CObject, 1| VERSIONABLE\_SHEMA)

Ако открием разлика тогава се генерира exeption. Но ако искаме да подтиснем се използва VERSIONABLE\_SHEMA и се приема че може да се работи с различни версии. Когато работим с различни версии ни трябва да предвидим код в Serialize(). Предефинирането работи за указатели към такива обекти-наследници, а не за самите обекти. Така че:

CLine\* pLine = new CLine( CPoint( 0,0), CPoint( 40,40));

ar << pLine; //работи добре

CLine line( CPoint( 0,0), CPoint( 40,40));

ar << line; // няма да работи

Aко искаме да сериализираме по стойност, а не по указател, е добре така:

CLine line( CPoint( 0,0), CPoint( 40,40));

ar << &line; //ще работи за сериализация

при десериализация:

CLine\* pLine;

ar >> pLine; //всичко е добре

CLine line = \*pLine;

// операция “=” е предеф. в CLine като копиране;

delete pLine; // вече ненужен

Когато се обръщаме директно към Serialize се отказваме от проверка за проекти(само една версия) Операторите за сериал. преоб-ват CSrting в UniCode

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

Kak действа сериализацията:

-**Formatter извършва сериализацията. Той ползва типа FormatterService FormatterService, който има само static методи (не може инстанция) и подпомага операциите.**

**1.Вика се негов метод, който определя полетата , които подлежат на сериализация/десериализацияя;**

**2.Вика се друг метод на FormatterService който чете тези полета и попъква с тях масив 3. Formatter попълва в служебна информация в асемблито за типа и stream. При десериализация, тази информация се чете предхождащо 4. Formatter последователно записва елементите в stream**

-**Ако искаме да имаме пълен контрол как ще действа сериализацията, то следва да имплементираме**

-**System.Runtime.Serialization.ISerializable интерфейса: public interface ISerializable { Void GetObjectData(SerializationInfo info, StreamingContext context } както и IDeserializationCallback имащ метода: OnDeserialization(Object sender);**

Съдържа информация дали сериализацията ще се прави в друг процес, машина, файл, друго приложение и т.н.) Този обект съдържа стойностите, които ше се серилизират.

Използвайки го formatter обекта, през GetObjectData() го

попълва с данните. В процеса помага и SerializationInfo.AddValue(),

OSHte vij PDFa s IME Serialization.pdf stranica 10!!

31. Run-time сериализация в .NET.

**Пролемът е : преобразуване на обект или групирани обекти в stream of bytes и обратно. .NET решава въпроса за почти всички типове (почти) автоматично.**

-**По подразбиране обектите не са сериализуеми. Следователно повикване на обект.Serialize(stream, инстанция); ще генерира SerializationException; За да стане дефиниция на обект сериализуема - [Serializable] атрибут;**

-**[Serializable] може да се добавя към : class, struct, enum, delegate**

-**Всички полета на сериализуем тип автоматично стават сериализуеми, независимо от модификатора за достъп.**

-**Понякога се налага някои полета да се изключат от сериализацията: напр mutex , който ще се десериализира в различен процес. Тогава: [Serializable] class Circle { private Double radius; [NonSerialized] private Double m\_area; ….**

32. Управление на паметта в Windows. Защита на пространството, виртуална памет, разширяване на оперативната памет към диск, валидизиране на страници, охраняеми страници, shared блокове, динамична резервация и ангажиране на блокове памет. Copy-on-write механизъм.

**Виртуална памет**

Осн. х-ка е изплзването и въвеждането на на вирт. памет. Всеки просец използва собствено адресно пространство 2(32)=4GB. Те се разделят поравно по default на 2 части- потребителски и системна. В рамките на това адр. Простр. Се отказваме от сегментите. Това променя работата с у-тели.Виртуалното адр. Пространство се образува като част от HDD. За да се ускори адр.простр се разделя на страници с фикс. Дължина 4К. Определя се от архитектурата на процесора. Страница е базовата ед. Памет ,к. се обменя м/у процесора и вирт. памет. В тези страници има флагове за различни нужди(дали стр. е в паметта). Една стр. може да е protected или shared. Динамичното заделяне също става на 3 стъпки:1. Резервиране 2. Ангажира. З.Заделя

Целта- ускоряване: **1.** Информираме mem manager-a na win, че резервираме дадена памет и др.процеси не може да я ползват. **2.** Съвместяваме табл.с адреси. **3**. Обръщаме се към нея.

Резервирането става по страници. Има guard page и exeption: EXEPTION GUARD PAGE,к. се генерира при първият опит за достъп до охранявана станица.. Memory се алокира с shared( блока физически е на едно място, но е в адр. Пространство на 2 процеса)

Copy-on-write- при общ блок в адр.простр. на 2 процеса, за времето когато четат този блок е единствен, ако единия пробва да пише win managera прави копие на блока и насочва промените към копието.

HEAP: В 16Bit среда има локален и глобален heap. В 32 Heap-a e общ.

2 подхода:

1. Използване С С++ класове за работа с heap-a.

2. Използване API ф-ии на средата. Пример:

HANDLE hfile=::CreateFile(..)

HANDLE hMAP=::CreateFileMapping(hfile,…)

LPVOID lpvFile=::MapViewofFile(hMap,..)

DWORD dwfilesize=::GetFileSize(hfile,..)

::UnmapViewofFile(lpvFile);

::CloseHandle(hMap)

::CloseHandle(hFile)

Можем да създадем heap като укажем синхронен достъп на повече от една thread..

Във всеко проявление на класа ни да се структурира собствен heap. Удачно е щото при него се изолираме от влияние на други threat-ове. Може да предефинираме new() delete() ако заделяме памет за специфични обекти. Собствения heap авт. се унищожава като се унищож. общата за класа.(така гарантираме че няма да останат неизползвани хип-ове)

New()- проверява създаден ли е до момента private heap. Ако не-създава и инициализира брояч. Заделя памет в хип-а. Инкрементира брояча.

Delete()- декрементира. Освобождава памет от хип-а.

33. Работа с динамични блокове памет. Предимства на използването на собствени динамични блокове в рамките на процес, нишка или клас.

В 16 битова среда имаме локално и глобално адресно пространство.

При 32 битова среда поради линейното пространство няма локално и глобално адресно п-во, а то е едно (2GB потребителско адресно п-о). Има 2 функции за работа с хипове:

- С++ функции MFC- по – универсални

- с API функции на ОС – специфични и когато се гони бързо действие и големи обеми.

**Има 2 вида хип:**

- автоматично зареждане на приложението(по подразбиране)

- динамично зареждане – формиране на собствен потребителски хип.

Има API ф-ии heapCreate, с к. може да се съдаде heap. След което може да се заделя дин. област.

**Препоръки за работа със собствения хип**:

\* в рамките на своя клас се създава собствен хип, така че методите на класа да работят със собствения хип и той да се унищожи от дестр-ра. Предимствата:

- безопасност – другата нишка да се намеси в хипа.

- до голяма степен се избягва фрагментацията.

- в С++ може да се предефинира New и Delete - чрез предефинирането на New може да се направи оптимално заделяне на памет. **Схемата реализираща предефинирането на New е следната:**

1) проверява дали е заделен локален хип с New ако не е го заделяме.

2) заедно със създаването се задава и брояч на New и Delete.

3) заделя се необходимия брой байтове, които се заделени

4) инкрементираме брояча

Схема реализираща предефинирането на Delete е следната:

1) освобождава блока

2) декремнтира брояча

3) проверява ако е 0 унищожава handler към файла

Има функция – heapwin за създаден вече (от нас) хип. Големите блокове алокират собствен хип и връщат указател към тях. За малки блокове използват памет от вече алокиран хип.

За големи блокове след интензивно използване на Delete трябва ни чести интервали за викане – heapwin, който всъшност освобождава тези блокове от хипа.

На етапа Commit паметта е заделена в Paging файла, а в паметта се заделя в ОП, едва когато се опитаме да запишем.

**Права на достъп:** само за четене, само за запис или др. Една страница в паметта може да е охраняема:

EXCEPTION\_GUARD\_PAGE

Ако имаме опция guard то трябва четене от паметта. MemoryManager отделя последната част от ОП като guard. Ако се опитаме да четем охраняемата граница, това става , когато е свършило останалото място и генериме EXEPTION\_GUARD\_PAGE, която може да се прихване и обработи, т.е. ако ни трябва още памет, то може за да заделим и да преместим охраняемата страница назад.

Една страница (част от паметта) може е shared, т.е да се използва от повече от 1 процес:

П-с1 Физ. памет П-с2

Share блок се помества в адресното пространство на 2 или повече процеса, но фактически е задлено само едно физическо адресно пространство.

Друг механизъм е Copy\_On\_Write

П-с1 Физ. памет П-с2

Едно .exe стартирано става процес. Ако 2 процеса само четат паметта то тя е една. Ако единият процес запише данни в този блок: то за другият процес се появяват нови данни. В този момент Copy\_On\_Write заделя нова ОП и копира там блока, тогава вече и двата процеса работят с различни блокове.

Memory Manager постоянно размества страници в паметта, но може да заключим страници за разместване и те тогава не могат да се разместват. Адреса е 32 – битов,отказваме се от 16 – битовата сегментна организация, (в 16 – битовата сегмента е 2.., а тук е 2..=4GB). Имаме по–големи сегменти.

Отделните процеси са в отделни сегменти. Друга новост тук е виртуалната организация на паметта. Тя се реализира чрез мощен SWAP механизъм. Трета новост е страничната организация на адресното пространство, т.е. работи се с блокчета, а не с цели сегмнети.

4GB адресно пространство може да се счита за разделено на 2 по 2GB.

- за потребителски програми

- за системни програми

34. Memory- mapped файлове. Избягване на фрагментацията при работа с малки блокове памет.

**Memory mapped file** – средство за оптимално управление на паметта. Състой се от асоцииране на файла или част от файл с ОП. Файла или част от файл се асоциира с виртуалното адресно пространство и тогава над него могат да се правят определени операции без да се качва.

*HANDLE hFile=::CreateFile(...)//създаваме Handle*

*HANDLE hMap=::CreateFileMapping(hFile,...);*

*LPVOID IpvFile=::MapViewoffile(hMap,...);//мапва целия файл*

*DWORD dwFilesize=::GetFileSize(hFile,...)//използваме файл*

*::UnmapViewofFile(IpvFile);*

*::CloseHandle(hMap);*

*::CloseHandle(hFile);*

Два процеса могат да ползват общ *hMap* т.е. те имат обща памет:

*GlobalAlloc(...,GMEM\_SHARED...);*

В Win32 не прави shared блок, както в Win16. Обща памет, но не общ файл както по – горе без *CreateFile(...)* и с подаване на параметри 0хFFFFFFFF вместо hFile.

*GlobalAlloc* е наследена в Win16. Тук в 32 битовата среда всъщност не действа , но може да се запише.

**Съвет при работа с динамична памет:**

- хипът се фрагментира при продължителна работа. Решението е по – често да се унищожава и да се създава нов или да се прекомпонира.

- викайте heapmin в NT за освобождаване на големи блокове

- не викайте HeapFree за малки блокове заделени с new.

- стекът вече не е ограничен до 64К и става толкова голям, колкото е необходимо.

Как да пестим пространство от swap файла?

- EXE и DLL файлове не са в swap file – те се включват във виртуалното адресно пространство на всеки процес. Добре е и константните данни по някакъв начин да се прикачат към тях, за да не попадат в swap file.

**Низове :**

Ако те са непроменяеми за цялото изпълнение , декларирайте const *charmystr[]=”my string”;*

Низът се съхранява заедно с кода – в секцията за инициализирани константни данни . тя се съхранява заедно с EXE файла. В тази секция не можем да слагаме само С++ обекти, създадени чрез конструктор. Например:

*const Crect my\_rect(0,0,100,100);* се поставя в .bss, който се вкарва в swap file и всеки процес има копие на този обект. По – лошо е:

*const CString my\_str(“new instance”);*  което води до :

1. поставяне на *CString* обект в .bss секцията
2. масивът от символите в .data (иниц. неконст. данни )
3. заемане памет за копие на символите за всеки стартиран процес.

Нищо не попада в EXE и всичко харчи памет.

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

\* бъгове

\* управляван heep в Common Language Runtime средата. Последоват. заемане, първо в кеш.

\* heep на С runtime библ.:поддържа свързан списък на блокове памет.

**Алгоритъм за събиране на боклук**

OutOfMemoryException

корени (roots)- пример

граф на достижимете обекти от корените

# няма неизползвани обекти; # освободен 🡨 🡪недостижим обект.

пр: class My {

static void XX() {

ArrayList x=new ArrayList(); //а е корен

for(Int32 x=0;x<1000;x++)

{

a.Add(new Object());

} //а реферира 1000 обекта

Console.WriteLine( a.Lenght); // използваме а

//липсва друго използване на а#### започва почистване на боклука

// 1001 обекта могат да се почистят

……}}

36. Финализация в .NET среда.

**Финализиране на обекти в .NET**

Типове, които обвиват ресурси, освен памет (файл, mutex, сокет..)изискват явно финализиране – Finalize() за почистването си. Пример:

protected override void Finalize()

{

try { CloseHandle(handle); }

finally { base.Finalize(); }

**В C#, Managed C++ (всички \_\_gc класове наследяват Finalize() от Object).** Това става в синтаксиса на деструктора:

Ако имаме: ~MyClass()

{ Console::WriteLine(S”Finalizing..”);}

компилаторът го разширява до: MyClass::Finalize()

{Console::WriteLine(S”Finalizing..”); MyBaseClase::Finalize();}

virtual ~MyClass()

{System::GC::SuppressFinalize(this); A::Finalize();}

//”А” обединява всички обръщения към обекти, реферирани в конструктора

недостатъци на Finalize():

\*заделяне на финализирани обекти е по-бавно

\*осводождаването се забавя, особено при масиви от такива обекти

\*финализиран обект, рефериращ други обекти забавя финализирането им

\*няма контрол кога се изпълнява Finalize()

\*няма гаранция за реда на изпълнения на Finalize(). 🡪 във Finalize() да не се викат вътрешни обекти.

Кога се вика Finalize():

1. поколение 0 се е запълнило;
2. явно обръщение към System.GC.Collect()

процес завършва – вика се Finalize() за всички обекти в него

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

Недостатъци на Finalize():

финализацията е по-бавен процес от деструкцията преди (формира се списък на финализиране, а самото унищожаване става при последващ почистващ пас)

финализирането за някои обекти се отлага във времето, поради системата за повишаване в поколение

вътрешни референции могат да объркат процеса на финализирането им. няма гаранция за реда на изпълнения на Finalize(). Следователно, да се избягва вграждане на обекти, когато за тях и за обграждащия ги е предвиден Finalize() (те може да са вече финализирани).

няма контрол кога се изпълнява Finalize

Кога се вика Finalize():

поколение 0 се е запълнило;

явно обръщение към System.GC.Collect()

процес завършва – вика се Finalize() за всички обекти в него.

Не се вика по подразбиране, ако обект излезе от обхват, или завърши метод

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

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

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

За даден интервал може да се счита, че обектът е “съживен”

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

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

Всъщност сме имплементирали интерфейс

public interface IDisposable {

void Dispose(bool)

}

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

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

private (ако класът е sealed) или

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

•Целият почистващ код е в Dispose(bool)

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

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

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

38. Поколения в .NET среда. Управление на поколенията.

**Поколения. Допускания:**

• **нов обект**  **по-кратък живот; стар обект** **по-дълго може да живее**

• **събиране на част от heap е по-бързо от събиране на целия heap.**

• **неизследвани обекти в управлявания heap са от поколение 0.**

|  |  |  |
| --- | --- | --- |
| **А** | **B** | **D** |
|  |  |  |

нов обект 🡪 по-кратък живот

\*събиране нач аст от heap е по-бързо от събиране на целия heap.

\*неизследвани обекти в управлявания heap са от поколение 0.

А B C D E

поколение 0

ABD

пок1 поколение 0

ABDFGHI JK

пок1 пок 0

ABDFGIK

пок 1пок 0

активиране системата за събиране на боклука:

void GC.Collect(Int32 Generation);

void Gc.Collect();

39. Програмна архитектура документ / изображение. Основни класове, структуриращи приложението: документен клас, клас на изображение, класове – рамки.

**Архитектура документ/изображение: клас на приложението и документ шаблон.**

*Document Template(документен шаблон)*

Такъв има всяко приложение. Той е различен в MDI и SDI. Прави връзка между ( при това само по 1) document class , view, frame. Има CSingleDocTemplate и CMultipleDocTemplate.

Съдържа:

* документен низ (7 полета),

*\nMy\_app\My\_app\n\n\nMy\_app.Document\nMy\_app Document*

* Иконата на изображението
* Менюто при активно изображение.

SDI документния шаблон се създава още в InitInstanse():

*CSingleDocTemplate \* pDocTemplate;*

*PDocTemplate = new CSingleDocTemplate( IDR\_MAINFRAME, RUNTIME\_CLASS( CMyDoc), RUNTIME\_CLASS( CMainFrame), RUNTIME\_CLASS( CMyView));*

*AddDocTemplate(pDocTemplate);*

// следват редове за създаване обектите от шаблона, показване рамката и изгледа. Включва се цикълът за обр. съобщения, като съобщенията се разделят към обектите - приложение, рамка, изглед, документ, прозорец (command routing).

Ако създадем приложение и кажем че разширението е .авс, то след InitInstance(),Windows знае че след като кликнем на файла .авс тотрябва ад се стартира нашето приложение.

**Насочването на** съобщенията **за обработка:**

Командните съобщения се насочват първо към изгледа и после към документа (той обработва само командхите съобщения), след което ако няма обработка той отива в Template ако и там няма , отиват във Frame, aкo и там няма обработка отиват в Application(обекта приложение) и там ако няма отиват в default (обработват се по подразбиране)

При двуслойна архитектура се обособяват няколко класове групи:

1)документен клас- обикновено името и класовете включват DOC

2)view class – обикновено името и класовете включват view частицата. Класът View е дъщерен на рамката (Frame) и се явява клиентската част на приложението. То трябва да има производен на СView класа и всички извеждания отиват към методите на тези класове.

3) Frame class – класове на рамката. Това е най-големия прозорец, в който се разполагат изображението. Някои съобщения могат да се пренасочат за обработка от Frame, а не от View.

4)Application class – включва стартова точка на програмата, връзката между Frame/Veiw/Document

5)Document class – отговаря за обработка на данни, сериализация, документация. Документния клас притежава специфичните интерфейси за обработка на данните. Има методи за обхойдане на всички обекти създадени от него или за UpdateAllView …

View class

40. NET Framework и системата за управление на общи типове. Типовете в CLR.

source File🡪C++, C#, Eiffel, Python, Visual Basic, Jscript, Fortran,

IL - Езиково-специфичен компилатор🡪C++, C#

Изпълним файл🡪IL Code & Metadata

Runtime🡪Изпълнима Среда

CLR Type System  
Value Types

Reference Types

Built-in Value Types

User-DefinedValue Types

Reference Types

ObjectTypes

Interface Types

User – Defined

Reference Types

Pointer Types

**Дефиниция** за стойностен тип (наследил System.ValueType) и референтен тип.

заемане пространство от стека и от хийпа

референтните обръщения са силно типизирани.

garbage collector мести обекти в хийп иосъвременява референциите

CLR генерира указатели към user-defined типовете, при необходимост (липсата на връзка)

41. Стойностни типове. Стандартни и user-defined.

1. ВГРАДЕНИ

CIL FCL описние CLS поддръжка

bool System.Boolean true-false Y

char System.Char Unicode character Y

int8 System.SByte signed 8 bits N

int16 System.Int16 signed 16 bits Y

int32 System.Int32 signed 32 bits Y

int64 System.Int64 signed 64 bits Y

unsigned int8System.Byte unsigned 8 bits Y

unsigned int16System.UInt16 unsigned 16 bits N

unsigned int32 System.UInt32 unsigned 32 bits N

unsigned int64 System.UInt64 unsigned 64 bits N

float32 System. Single IEEE 32 bits FP Y

float64 System.Double IEEE 64 bits FP Y

native int System.IntPtr signed native int Y

native unsigned int System.UIntPtr unsigned native int N

2. Булеви

3. characters16 bits Unicode

4. Integers

5. Floating Point types

User-defined Value Types: в стека; липсва default constructor; биват изброими и структури изброими

using System

namespace Enumeration

{ struct EnumerationSample

{ enum Month { January = 1, February,…… }

static int Main(string[] args)

{ Console.WriteLine(“{0} is month {1}”, Month.September, (int) Month.September);

return 0;

}}}

структури (C++ 🡪 class; C# 🡪 struct за value type и class за reference type

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

**Типове методи, събития, полета, properties.**

В структурите може да се съдържат:

. методи (static or instance)

fields (static- когато типът няма стойност - or instance). Има тип и име

properties ( static or instance). “Логически полета” с тип

events ( static or instance). Той има име и тип (сигнатурата на callback методa). След дефиниране на event автомат. се генерират методи add\_EventName и remove\_eventName за добавяне и премахване на listeners за събитието.

Tип с дефинирани в него събития уведомява други обекти за възникването им.

Други обекти могат да регистрират или отрегистрират интерес към събитието, като го отработят по различен начин.

кратък пример:

using System;

namespace EventSample

{

public delegate void Adelegate();

struct EventClass

{public event ADelegate AnEvent;

public void InvokeEvent()

{

if (AnEvent != null)

AnEvent();

}

static void CallMe()

{Console.WriteLine(“ I got called”);)}

static void Main(string[] args)

{

EventClass e = new EventClass();

e.AnEvent += new ADelegate(CallMe);

e.AnEvent += new ADelegate(CallMe);

e.AnEvent += new ADelegate(CallMe);

e.InvokeEvent();

}}}

43. Събития. Кратък пример.

**Събития (events)** В компонентно-ориентираното програмиране компонентите изпращат събития (events) към своя притежател за да го уведомят при настъпване на интересна за него ситуация. Обектът, който предизвиква дадено събитие, се нарича изпращач на събития (event sender). Обектът, който получава дадено събитие, се нарича получател на събитието (event receiver). За да получават дадено събитие получателите му предварително се абонират за него (subscribe for event).  
 В компонентния модел на .NET Framework абонирането, изпращането и получаването на събитията се поддържа чрез делегати и събития. Събитията в C# са специални инстанции на делегати, декларирани с ключовата дума event. За променливите от тип събитие C# компилаторът автоматично дефинира операторите **+=** и **-=** съответно за абониране за събитието и за премахване на абонамент. Събитията могат да предефинират кода за абониране и премахваме на абонамент.  
**Разлика между събитие и делегат** Събитията, декларирани с ключовата дума event не са еквивалентни на член-променливите от тип делегат. ***public MyDelegate m; ≠ public event MyDelegate m;***Събитията могат да бъдат членове на интерфейс, а делегатите не могат. Извикването на събитие може да става само от класа, в който е дефинирано. Достъпът до събитията по подразбиране е синхронизиран.

**Конвенция за събитията** В .NET Framework се използва утвърдена конвенция за събитията:  
Делегатите, които се използват за събития:  
-имат имена образувани от глагол + EventHandler (SomeVerbEventHandler)  
-връщат void и приемат два параметъра – обект-изпращач от тип System.Object и обект, съдържащ данни за събитието от тип, наследник на System.EventArgs  
Пример: ***public delegate ItemChangedEventHandler( object aSender, ItemChangedEventArgs aEventArgs);***-Събитията се обявяват като public, започват с главна буква и завършват с глагол, например: ***public event ItemChangedEventHandler ItemChanged;***-За предизвикване на събитие се създава protected void метод с име в стил OnVerb, например:  
***protected void OnItemChanged() { … }***-Методът-получател (обработчик) на събитието има име Обект\_Събитие: ***private void OrderList\_ItemChanged() { … }*Събития и интерфейси**Събитията (events) могат да бъдат членове на интерфейси:   
***public interface IClickable   
{  
 event ClickEventHandler Click;  
}***При имплементацията на събитие от интерфейс за него могат да се реализират специфични add и remove методи. За разлика от свойствата, при събитията имплементацията на методите add и remove не е задължителна.  
**Интерфейси, събития, делегати**В .NET поведението "обратно извикване" (callback) може да се реализира чрез интерфейси, делегати или събития  
**Кога да използваме интерфейси?**  
Когато даден обект предоставя съвкупност от много callback методи.  
**Кога да използваме събития?**  
Когато разработваме компоненти, които трябва да известяват своя притежател за нещо, когато търсим съвместимост с компонентния модел на .NET.  
**Кога да използваме делегати?**  
Когато имаме единичен callback метод, който не е свързан с компонентния модел на .NET.

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

**Проектиране на тип, предлагащ събитие. Проектиране на тип използващ събитие.**

Проектиране на тип, предлагащ събитие

class EventManager{

// 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;}

// следва делегат, дефиниращ прототип на callback метод, който

// получателите следва да имплементират

public delegate void MailMsgEventHandler ( Object sender, MailMsgEventArgs args);

//дефиниция на самото събитие – от същия тип

public event MailMsgEventHandler MailMsg;

// метод, отговорен за уведомяване на регистриралите интерес към събитието обекти

protected virtual void OnMailMsg(MailMsgEventArgs e)

{ if(MailMsg != null) //има ли регистрирали интерес към събитието

{MailMsg( this. e); //уведомяваме всички рег. обекти

}}

// 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 дефиниране на : public event MailMsgEventHandler MailMsg; се преобразува от компилатора така:

\* създава се private делегатно поле, в началото null, впоследствие поддържащо референция към свързан списък от делегати, желаещи да бъдат уведомявани за събитието.

\* дефинира public void add\_MailMsg(MailMsgEventHandler handler) метод, добавящ референция в свързания списък.

\* дефинира public void remove\_MailMsg(MailMsgEventHandler handler) метод, отрегистриращ event handler за обект, който вече не се интересува от събитието.

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

Методите са public, защото и съзитието е било декларирано public

Б. проектиране на тип, слушащ за събитие

class Object1 {

// предаване на EventManager към конструктора

public Object1(EventManager mm)

{// добавяме референция към слушателя (FaxMsg callback метода) към MailMsg списъка на

// EventManager

mm.MailMsg += new EventManager.MailMsgEventHandler(FaxMsg);

// конструира се делегатен обект, обвиващ метода FaxMsg в интерфейс за да може да бъде

// външно повикан с Invoke() – вж. делегати) }

// следва описание на callback метода , който EventManager ще извика при събитие

private void FaxMsg( Object sender, EventManager.MailMsgEventArgs e)

{…………………………….}

public void Unregister( EventManager mm)

{// конструирваме инстанция на MailMsgEventHandler делегата, рефериращ callback метода

// FaxMsg и го отрегистрирваме като елемент от списъка

EventManager.MailMsgEventHandler callback\_element = new EventManager.MailMsgEventHandler(FaxMsg);

mm.MailMsg -= callback\_element;

}}

*Sealed Value Type*

\* value type не се наследява

\* sealed

45. Пакетирани типове (boxed types). Проблеми с достъпа.

възможни за всеки value type, вкл. user-defined, автоматично се генерират в обвиващ обект (boxing).

unboxing – връща value type в норм. състояние. Изработва указател към данната (до момента съдържана в boxed обект), копира данните на boxed обекта във value type и в стек. Помества boxed обекта в heap следователно всеки value type, може да се разглежда като обект, при необходимост и да взаимодейства с всеки друг обект (всички са наследници на Object). Общото наследство наподобява IUnknown.

Всеки box type поддържа интерфейси

using System;

struct Point

{public Int32 x,y;

public void Change(Int32 x, Int32 y) { this.x = x; this.y = y; }

public override String ToString()

{ return Strinf.Format( “{0}, {1}”, x, y);}}

class App

{ static void Main()

{

Point p = new Point();

p.x = p.y = 1;

Console.WriteLine(p);// (1,1)

p.Change(2, 2);

Console.WriteLine(p);// (2, 2)

Object o = p;// boxing

Console.WriteLine( о );(2, 2)

(( Point) o).Change(3, 3);

// защото Change() е метод на Point

// “ о “ се разопакова. Полетата му се копират във value type Point в стека и се променят на 3

Console.WriteLine( o );// (2, 2) – това е “пакетираният” обект

46. Референтни типове.

Reference Types

\* комбинира инф. за местоположение и за съдържание.

\* местоположението е “Type Safe”, т.е. само assignment-compatible типове могат да се пазят там.

\* garbage collector мести тези обекти свободно.

**А. Object Types**

Наследяват Object (System.Object).

Основни методи: Equals; Finalize (вика се преди gc да освободи паметта); GetType (важно е че методът е невиртуален – т.е. не е възможно някой да се представи за друг);

MemberwiseClone; ToString.

using System;

namespace Override

{

class Sample

{ static void Print( params Object[] objects)

{ foreach(Object o in objects)

Console.WriteLine( o.ToString());

// вика подходящие метод за всеки аргумент, ако съществува }

static void Main( string[] args)

{

Object o = new Object();

String s = “Nakov”;

int I = 33;

Print( o,s, i); // int boxed element

}}}

**Предаване променлив брой параметри:**

static Int32 Add(params Int32[] values )

{

Int32 sum = 0;

for( Int32 x = 0; x< valies.Lenght; x++)

sum += values[x];

return sum; }

Викаме го така: Add( 1,2,3,4,5,6)

което е по-нагледно от:

Add(new Int32[] { 1,2,3,4,5,6})

\* ключовата дума params добавя атрибут към параметъра [ParamArray].

\* Компилаторът претърсва за съответстващи методи без params и след това за такива с params. Тогава пакетира параметрите в масив и вика метода с указател към масива.

\* само последният параметър на метод може да е с params

\* данните трябва да могат да се пакетират в едномерен масив

\* метод с променлив брой параметри от различен тип се декларира:

Method ( params Object[] objects)

Важен CLR вграден тип (освен Object) е String

\* той е sealed

\* всеки метод, модифициращ низ, създава нов низ

\* основни методи:

конструктори ( няма default) за различни аргументи – char, масив char, указат. към char

Compare; Concatenate; Format; IndexOf; Insert; Length….

using System;

namespace StringSample

{

class Sample

{

static void Main(string[] args)

{ String s = “Nakov”; // без new (може, защото е вграден)

Console.WriteLine(“Length is:” + s.Lenght); // вземаме property – дължина

// преобразъва се в boxed тип. String + Object ?

// съществува overrided concatenation method на class String с параметър Object. Console.WriteLine(s.ToLower()); // създава се временен string object

}}}

47. Интерфейсни типове. Използване на интерфейси със стойностни типове.

**Интерфейсни типове**

**Interface Types**

\* въведени в COM и CORBA

\* целта е не само функционална наследяемост. но и поделяне на общ външен вид – интерфейс

\* интерфейсният тип е частична спецификация на тип

\* поддържа се наследяемост на интерфейсни типове

\* интерфейсният тип съдържа:

- методи (статични и на инстанция);

- полета (статични);

Property - - events

**Методите на интерфейс са public, abstract virtual.**

**Д. Интерфейси и Value типове**

using System;

namespace InterfaceSample

{public delegate void Changed();

interface IPoint

{ int X

{ get; set;}

int Y

{ get; set;}

}

struct Point : IPoint // value type , наследил интерфейс

{ private int xValue, yValue;

public int X

{ get { return xValue;}

set { xValue = value;}

}

public int Y

{ get { return yValue;}

set { yValue = value;}

}

}

public class EntryPoint

{ public static int Main()

{

Point p = new Point();

p.X = p.Y = 33;

IPoint ip = p; // interface pointer – в heap, value типа в стека

Console.WriteLine(“X: {0}, Y: {1}”, ip.X, ip.Y);

// ip има достъп само до reference типове, докато p е value type. Следователно p е опакован и

// ip има достъп до опакования обект и членове.

}

}}

Присвояване и съвместимост при съвпадение на типовете - когато присвоявания обект е от подтип;

когато имат общ интерфейс .

using System;

namespace AssignmentCompability

{

class Sample

{static void Main(string[] args)

{System.Int32 I = 42;

Object o;

String s = “Nakov”;

IComparable ic;

o = s; //OK

ic = s ;//OK – обект s поддържа IComparable

o = i; //OK влиза в box и се присвоява към обект

ic = i; // OK i се пакетира

i = s; // грешка при компилация

s = (String) o;

// runtime error: в момента “о” съдържа i . Свива се обсега на boxed обект към подобект

} }}

49. Делегати. Дефиниране, използване.

50. Вътрешно представяне на делегат.

51. CLR поддръжка за делегатите. ZAedno sa ako se padne edno ot tqh diktuvai vsichko .

*Делегати*  
- callback функции в Windows

- callback функции, реализиращи специфични обработки: qsort от runtime библиотеката приема callback ф-ия , сортираща елементите на масив

- callback не носи инф. за брой и тип параметри, връщана стойност, конвенция на повикване.

- callback функциите не са типово-обезопасени

Делегатите представляват .NET типове, които описват сигнатурата на даден метод (броя, типа и последователността на параметрите му) и връщания от него тип. Делегатите приличат на указателите към функции в C и C++ – съдържат силно-типизиран указател (референция) към метод. Те са структури от данни, които приемат като стойност методи, отговарящи на описаната от делегата сигнатура. Чрез тях се осъществяват "обратни извиквания" (callbacks).  
Могат да сочат както към статични методи, така и към методи на инстанция.  
Делегатите в .NET Framework са специални класове и могат да бъдат два вида:  
**-Единични (single-cast) делегати**: Съдържат референция към един единствен метод наследяват класа System.Delegate  
**-Множествени (multicast) делегати:** Съдържат свързан списък от референции към методи наследяват класа System.MulticastDelegate.  
В C# могат да се декларират само Multicast делегати (чрез запазената дума delegate).  
**Multicast делегати** При извикване на multicast делегат, се изпълняват последователно един след друг всички методи от неговия списък. Ако multicast делегат връща стойност или променя ref или out параметър, резултатът е само от последния извикан метод от списъка с методи на делегата. Ако при извикване на multicast делегат някои от методите в неговия списък хвърли изключение, следващите методи от списъка не се извикват. На практика single-cast делегати почти не се използват и под делегат обикновено се има предвид multicast делегат.  
**Класът System.MulticastDelegate** Наследник на System.Delegate и е базов клас за всички делегати в C#. Съдържа метод Combine за сливане на списъците от методи на няколко делегата от еднакъв тип, метод Remove за премахване на метод от списъка за извикване, метод GetInvocationList(), който връща масив от делегати – по един за всеки от методите в списъка за извикване на делегата. Има свойство Method, което описва сигнатурата на методите в делегата.

Дефинират се с кл. дума delegate

конструират се техни инстанции с new

вика се callback метода, като вместо име на метод, подаваме променлива, реферираща делегатен обект

*как CLR и компилаторът осигъряват поддръжката за делегати:  
нека имаме:*

делегатният обект е обвивка около метод и обект , с който методът работи

Обрущението към callback метод е като с ф-ия. Всъщност това е променлива , реферираща делегатен обект. Компилаторът извиква Invoke(). Все едно:

feedback.Invoke( items[item], item, items.Lenght);

Invoke() може да се вика явно от други езици

Обсег на член на тип

public

assembly

Family

Family or assembly

Family and Assembly

Compiler-controlled

Private

55. Същност на Java EE. Описание на многослойно приложение. Java EE компоненти за различните слоеве.

Същност на Java EE

набор от API-та и runtime среда за разработка и изпълнение на реални бизнес приложения (enterprise applications)

•разширение на Java SE (Standard Edition) предлагащо възможности за ORM мапинг, отдалечени извиквания, уеб услуги и много други

•дизайн базиран на сътрудничещи си отделни *компоненти* за създаване на многослойна архитектура

•залага на "*convention over configuration*"

Многослойно приложение:

* стандартен pattern за създаване на сървърни приложения
* •базира се на групи от компоненти с обща цел наречени слоеве (tiers) - като части от асембли-линия
* •всеки слой изпълнява функциите си и делегира останалата работа на следващия слой

Java EE компоненти за различните слоеве.

Презантационен слой.

servlet, jsp, facelet, listener, filter, custom tag

Бизнес-логичен слой.

ejb (session, message-driven), interceptor, timer

persistence layer :entity,listener.

database layer – извън обхвата на Java

56. Същност и приложение на сървлети. Жизнен цикъл на сървлет. Споделяне на информация и съхранение на състояние (сесия).

сървлетът "сервира" информация като отговор от получена заявка

•програмно API позволяващо прихващане на заявка обслужвана от сървъра и генериране на отговор

•клас имплементиращ Servlet интерфейса

•може да се използва за всякакъв вид заявки (GenericServlet), но най-често се използва заедно с HTTP протокола (HttpServlet)

Приложение на сървлети:

динамично генериране на съдържание

•обработка на HTTP заявки:

▫GET – за представяне на данни

▫POST – за получаване на данни от HTML форми

•обработка на множество паралелни заявки (по-добро scalability спрямо CGI)

•платформено независимо сървърно API

•позволява препращане на заявки към други сървлети (дава възможност за load balancing)

•основа на JSP и JSF технологиите

Жизнен цикъл на сървлет:

**load** - контейнерът зарежда сървлет класа при стартиране на уеб модула или при първа заявка

•**instantiate -** контейнерът създава инстанция от сървлет класа използвайки конструктора му (например new MyServlet())

•**init** - преди сървлетът да може да обслужва заявки, контейнерът го инициализира (чрез метод init()) и подава параметрите от web.xml-a .

**service** - след успешна инициализация, сървлетът може да обслужва заявки; контейнерът създава ***отделна нишка*** за всяка заявка и вика service() метода на сървлета

•**destroy** - когато сървлетът вече не е нужен, контейнерът вика метод destroy(); подобно на init() това се случва веднъж в жизнения цикъл

•**unload** – сървлет класът се освобождава от JVM-а на контейнера

Споделяне на информация

уеб компонентите споделят информация под формата на обекти складирани като атрибути на 4 scope обекта

•те се манипулират чрез методи setAttribute() и getAttribute() на scope-а

•scope-овете са (с намаляваща видимост):

▫*application* (javax.servlet.ServletContext) - данните ще са видими от всички уеб компоненти в приложението

▫*session* (javax.servlet.http.HttpSession) - данните ще са видими за уеб компоненти в текущата сесия (може да се използва за прехвърляна на данни между заявки)

▫*request* (javax.servlet.ServletRequest) - данните ще са видими за уеб компоненти от текущата заявка

▫*page* (javax.servlet.jsp.JspContext) - само в дадената JSP страница .

Съхранение на състояние:

сесията се използва за корелация на поредица от заявки

•представя се като HttpSession обект достъпен от request обекта, чрез метод getSession()

•най-лесният начин за следене на сесията е чрез подаване на уникален идентификатор:

▫идентификаторът може да се подава като *cookie*

▫уеб контейнерът може да включи идентификатора като параметър на всяка заявка (URL rewriting)

57. Същност на JSP. Жизнен цикъл на JSP. JSP директиви. JSP синтактични елементи (директива, скриплет, израз и др.).

Същност на JSP:

технология за създаване на статични или динамични view-та

•използва HTML синтаксис и лесно се интегрира с JavaScript и CSS

•транслира се до сървлет, който се изпълнява на сървъра и връща генерираната репрезентация

•по-малко писане на Java код

•голям набор от Java EE сървъри поддържащи JSP спецификацията (дава portability)

JSP директиви:

JSP директивите влияят на цялостната структура на генерирания сървлет клас

•синтаксис: **<%@директива атрибут="стойност" %> <jsp:directive.име атрибут="стойност"/>**

JSP синтактични елементи (директива, скриплет, израз и др.).

JSP директивите влияят на цялостната структура на генерирания сървлет клас

•синтаксис: **<%@директива атрибут="стойност" %> <jsp:directive.име атрибут="стойност"/>**

•**page** директива - дефинира атрибути за JSP-то

▫дефинира import пакети: <%@ page import="java.util.\*" %>

▫MIME тип на резултата: <%@ page contentType="text/plain" %>

▫дефиниране на имплицитна сесия: <%@ page session="true" %>

▫задаване на JSP URL за грешки: <%@ page errorPage="url" %>

**include** директива - позволява да включат други страници преди JSP-то да се конвертира до сървлет

▫пример: <%@ include file="/opa/header.jsp" %>

•**tablib** директива - декларира използването на библиотека с custom тагове

▫пример: <%@ taglib uri="uri" prefix="op" %>

•XML формат на директивите:

▫<jsp:directive.page attribute="value" />

▫<jsp:directive.include file="url" />

▫<jsp:directive.taglib uri="uri" prefix="op"/>

JSP изрази

JSP изразите се използват за директно вмъкване на резултата от Java израз в изходния резултат от JSP-то

•синтаксис: **<%= Java израз %>**

•примери:

▫<%= new java.util.Date() %>

▫<%= Math.PI %>

▫<%= *request*.getRemoteHost() %>

▫<%= *session*.getMaxInactiveInterval() %>

JSP скриплетиJSP скриплетите позволяват директното влагане на Java код в \_jspService() метода на JSP-то

•синтаксис: **<% Java код; %>**

•скриплетите имат достъп до стандартните JSP променливи (request, response, session, …)

•пример: <% for (int i=0; i<10; i++) { %> <%= i %> \* <%= i %> = <%= i\*i %> <br> <% } %>

58. Същност и предимства на библиотеките от тагове. Същност и предимства на Unified Expression Language.

Същност и предимства на библиотеките от тагове :съвкупност от пакетирани потребителски тагове

•всяка библиотека има префикс и URI идентификатор

•всяка таг библиотека се състои от:

▫TLD дескриптор описващ таговете

▫JAR с компилираните класове и ресурси на таговете

•използване:

▫TLD дескрипторът се слага в /WEB-INF

▫JAR-ът се поставя в /WEB-INF/lib

▫регистриране библиотеката в JSP-то: <%@ taglib prefix="mytags" uri="WEB-INF/my.tld" %>

▫използване на тага: <mytags:opa someattr="value" />

Същност на Unified Expression Language.

позволява лесното достъпване на обекти, техните пропъртита и методи от JSP и JSF страници

•синтаксис: **${ el израз }**

•замества JSP action таговете:

▫<jsp:useBean id="user" scope="request" />

▫<jsp:setProperty name="user" property="name" value="Penn" />

▫<jsp:getProperty name="user" property="name"/>

предимства на Unified Expression Language.

дава кратка нотация за достъп до данни в scope-ове page, request, session и application ${course}

•лесен достъп до елементи на колекция ${students[2]}

•лесен достъп до пропъртита на bean обекти: ${course.presenter.name}

•достъп до параметри на заявката, хедъри, cookies, …

${param["studentCount"]}

${cookie["dateOfLastVisit"]}

набор от прости оператори (+, -, \*, /, =, <, >, ==, &&, ||, ?:, empty, not): ${(2 + 5) \* 3} ${course.studentCount - 1} ${not empty presenter ? "тук съм" : "успах се" }

•автоматично конвертиране на типове към стринг

•възможност за извикване на функции: ${fn:length("It’s the cycle of life…")}

•връща празен стринг вместо exception: ${thisIsNull.prop}  ""

59. MVC – същност и компоненти. Пример за използване на MVC в едно Java EE приложение.

MVC – същност и компоненти:

софтуерен *pattern* разделящ презентацията на информация от работата на потребителя с нея

•използва се при визуални приложения

•разделя приложението на:

▫**model** - съхранява данните, състоянието и бизнес логиката; информира view-то при промяна на състоянието \*

▫**view** - представя визуално данните от модела

▫**controller** - приема потребителските команди и на тяхна база манипулира данните в модела; праща сигнали за промяна на view-то

видимост между компонентите:

▫моделът не знае за останалите компоненти

▫view-то знае само за модела

▫контролерът знае за модела и view-то

•съставни части на *pattern-а*:

▫composite pattern (view)

▫strategy pattern (view и контролер)

▫observer pattern (view и модел)

Пример за използване на MVC в едно Java EE приложение.

потребителят изпраща команда за добавяне на елемент в списъка:

▫контролерът получава командата и модифицира модела

▫моделът информира view-то за промяната

▫view-то визуализира новия елемент

•потребителят изпраща команда за скролиране:

▫контролерът я получава и праща сигнал до view-то

▫view-то скролира списъка

60. Същност и предимства на EJB. Общи изисквания към EJB класовете. Видове EJB-та.

Същност на EJB:

ENTERPRISE JAVABEANS

•платформа за създаване на portable, reusable и scalable бизнес приложения използвайки Java

•server-side *компоненти* използвани за изграждане на бизнес логиката и кода за съхранение на данни

•POJO класове капсулиращи логиката на приложението и разчитащи на услугите на контейнера

компонентният характер на EJB-тата позволява:

▫повторно използване на бизнес логика (например един модул за таксуване на кредитни карти, може да се вика от различни сайтове на компанията)

▫добра организация на приложението

•услугите на EJB контейнера позволяват:

▫директно използване на тествана комплексна функционалност (като транзакции, сигурност, AOP)

▫фокусиране върху бизнес логиката

•анотациите дават лесен начин за декларативно дефиниране на поведение от разработчика

Видове EJB-та:

*session* - извиква се от клиент за изпълнението на определена бизнес операция (например таксуване на кредитна карта), биват:

▫*stateful* - запазва състоянието между извикванията

▫*stateless* - не запазва състояние, независими операции

▫*singleton* - съществува една единствена инстанция

•*message-driven* - също имплементира бизнес логика, но се стартира чрез пращане на съобщение на messaging сървър; позволява асинхронност

•*entity* - POJO съответстващо на запис в дадена таблица и носещо ORM информация (в анотации)

Общи изисквания към EJB класовете:

* POJO клас имплементиращ един или повече бизнес интерфейса дефиниращи операциите (защо?)\*
* •бизнес интерфейсът използва анотациите @Local, @Remote или @WebService
* •не може да е abstract или final (защо?)
* •трябва да има конструктор без параметри (защо?)
* не може имената на методите да започват с ejb…()

61. Съпоставка на stateless и stateful бизнес компонентите. Жизнени цикли и callback методи:

stateless

не поддържа състояние между извикванията

•не се интересува от клиента

•работата се извършва с едно единствено извикване на метод

•може да има един или повече тематично свързани метода

•позволява poоling

•дава по-добро scalability

•*най-популярния вид*

stateful

поддържа състояние между извикванията

•последователни заявки от един клиент се обработват от едно и също EJB

•пазенето на състояние пречи на използването на pooling

•използва се повече памет

•използва се passivation/ activation за намаляване на паметта

•използва се @Remove метод

Жизнени цикли steatless

1.Създава се bean инстанцията използвайки конструктора без параметри.

2.Контейнерът инжектира ресурси.

3.Контейнерът складира инстанциите в pool.

4.При всяка нова заявка се взима свободен bean от pool-a (ако има свободен).

5.Изпълнява се избрания бизнес метод.

6.Връща се bean-ът обратно в басейна.

7.Унищожава се bean-ът.

Stateless EJB *callbacks:*

методи които се извикват в определена фаза от жизнения цикъл на bean-а (сходни с init() и destroy() методите при сървлети)

•обозначават се с анотации

•може да има няколко еднакви callback-а за един bean

•биват:

▫@PostContruct – извиква се в края на инициализирането на bean-а (след инжекцията на ресурси, стъпка 2) за осъществяване на комуникация с други системи

▫@PreDestroy – извиква се преди унищожението на bean-a (стъпка 7) за освобождаване на заети ресурси

Stateful EJB *callbacks:*

поддържат се стандартните callback-ове (@PostContruct и @PreDestroy)

•дефинират се два нови callback-а свързани с управлението на състоянието:

▫@PrePassivate – извиква се преди *пасивирането* на bean-а (тогава се сериализират данните в bean-a и се складират на диска вместо в паметта); целта е да се освободят конекции/данни неподдържащи складиране

▫@PostActivate – извиква се след активацията на bean-а, с цел повторна инициализация на загубени конекции с външни системи

62. Същност на Dependency Injection и сравнение с JNDI lookup. Същност на декларативната сигурност. Примерни анотации.

Същност на Dependency Injection:

цели да направи връзките между компоненти максимално *loosely coupled*

•компонент А извиква Б само чрез интерфейса му и никога не създава инстанция от Б директно

•реалните връзки се правят чрез конфигурация, а не в кода

•връзките се правят от контейнера  DI или IoC

•позволява лесна подмяна на имплементацията

Dependency Injection и сравнение с JNDI lookup

създаване чрез конструктор: BidManager manager = new BidManagerBean();

•взимане чрез JNDI: InitialContext context = new InitialContext(); BidManager manager = (BidManager)context.lookup("java:comp/env/BidManager");

•получаване чрез DI: @WebServlet("/actions/add-bid") public class AddBidServlet extends HttpServlet { **@EJB** private BidManager placeBid; … }

Същност на декларативната сигурност:

позволява декларативно да се зададе *authentication* (верифициране на самоличността) и *authorization* (определяне на правата) на потребител

•дефинират се роли (roles) и се задава достъпа им до определена функционалност

•ролите се мапват на групи и потребители на сървъра при деплой и конфигурация

•възможно е да се правят проверки в кода вместо декларативно

Примерни анотации:

•@DeclareRoles - задава списък от роли известни за приложението

•@PermitAll - дава достъп на всички роли в приложението до даден метод

•@DenyAll – забранява достъпа на всички роли в приложението до даден метод

•@RolesAllowed – задава конкретни роли имащи достъп до даден метод

•@RunAs – позволява промяна на ролята на текущия потребител

63. Същност на EJB транзакциите. Видове. Предназначение на прихващането на съобщения. Примерни анотации.

Същност на EJB транзакциите

групират набор от операции и гарантират всичко или нищо (например местене на пари между сметки)

•спазват ACID принципите

•могат да обхващат една система или няколко транзакционални системи

•разчита на JTA (Java Transaction API)

Видов транзакции:

спрямо обхват:

▫локални (един ресурс и manager, single commit)

▫глобални (N ресурси, Transaction Manager + N \* Resource Managers, 2PC)

•спрямо контрол:

▫CMT – транзакции управлявани от контейнера (чрез декларирани анотации)

▫BMT – транзакции управлявани от потребителя (чрез UserTransaction; ut.begin(); ut.commit(); ut.rollback())

Предназначение на прихващането на съобщения:

interceptor-ите предлагат опростени AOP възможности за EJB-та

•прихващат извикването на бизнес методи

•позволяват групирането на обща логика срещана в много класове (cross-cutting concerns)

•често се използва се за по-прости задачи – logging, tracing и auditing .

64. Сравнение между ООП и релационното представяне на данните. Прилики и проблеми. Същност на ORM.

Конфликт между бази данни и ООП

•идентичност

•състояние

•поведение

•капсулация

•наследяване

•полиморфизъм

•таблици

•редове

•колони

•обединение

•сечение

Същност на ORM.

преобразуване на обекти във формат за релационни бази данни и обратно

Проблеми:

идентичност ①

•наследяване ②

•еднопосочни асоциации ③

•комплексни типове (гранулярност) ④

•извличане на данни (QbE, QbA, QbL) ⑤

•зареждане на данни и кеширане ⑥

•*две схеми на данните*

•*капсулация и др.*

* клас ↔ таблица
* •свойство ↔ колона
* •асоциация ↔ foreign key
* •int ↔ INTEGER, String ↔ VARCHAR

65. JPA обект за съхранение (entity). Задаване на идентификатор, достъп до данните, вложени обекти. Пример.

JPA интерфейс за работа с entity-та

•еквивалентен на Hibernate Session интерфейса

•базови методи:

▫**persist**

▫**merge**

▫**find**

▫**flush**

▫**lock**

▫**remove**

▫**detach**

▫**refresh**

▫**clear**

▫**close**

66. Релации между entity-та. Видове. Примери.

Релации между entity-та:

* One-To-One
* •Many-To-One
* •One-To-Many
* •Many-To-Many

Характеристики на релацията:

* -роли:
* ▫източник (собственик)
* ▫дестинация
* -кардиналност:
* ▫едно
* ▫много
* -посока:
* ▫еднопосочна
* ▫двупосочна

OneToOne еднопосочна:

**@Entity**

public class Employee {

**@Id** private int id;

...

**@OneToOne**

**@JoinColumn(name="PSPACE\_ID")**

private ParkingSpace parkingSpace;

...

}

OneToOne двупосочна

**@Entity**

public class ParkingSpace {

**@Id** private int id;

... **@OneToOne(mappedBy="parkingSpace")**

private Employee employee;

...

}

ManyToOne еднопосочна ③

**@Entity**

public class Employee {

...

**@ManyToOne**

private Department department;

...

}

OneToMany двупосочна

**@Entity**

public class Department {

**@Id** private int id;

...

**@OneToMany(mappedBy="department")**

private Collection<Employee> empls;

...

}

ManyToMany двупосочна

**@Entity**

public class Project {

**@Id** private int id;

...

**@ManyToMany(mappedBy="prjs")**

private Collection<Employee> empls;

...

}

67. EntityManager – създаване и използване. Реализация на ООП наследяване в JPA.

EntityManager – създаване и използване:

чрез EntityManagerFactory

**EntityManagerFactory** emf =

**Persistence**.createEntityManagerFactory("emp");

**EntityManager** em = emf.createEntityManager();

...

em.close();

emf.close();

чрез Dependency Injection (DI)

**@PersistenceUnit(unitName="EmployeeService")**

EntityManagerFactory emf;

**@PersistenceContext(unitName="EmployeeService", type=PersistenceContextType.EXTENDED)**

EntityManager em;

**@PersistenceContext(unitName="EmployeeService")**

EntityManager em;

Използване:

Person p = new Person ();

EntityManager em = ...;

em.**persist**(p);

Long id = ...;

Person p = em.**find**(Person.class, id);

em.**remove**(p);

em.**flush**();

Наследяване:

три стратегии

▫обща таблица за клас йерархия (SINGLE\_TABLE)

▫join на наследници (JOINED)

▫една таблица за конкретен (TABLE\_PER\_CLASS)

•@MappedSuperclass (съхранение на данните от родителите)

68. Начини за извличане на данни в JPA. Същност и особености на JPQL. Пример

Начини за извличане на данни в JPA:

Query by Example:

▫EntityManager.find(*class*, *id*)

•Query by API:

▫JPA Criteria API

•Query by Language:

▫JPQL = Java Persistence Query Language

Същност и особености на JPQL:

вдъхновен от SQL и HQL

•*entity*-та вместо таблици

•OOP модел за обхождане на релации ('.')

**SELECT e**

**FROM Employee e**

**WHERE e.department.name = 'NA42'**

**AND e.address.state IN ('NY','CA')**

* единичен и множество резултати
* •функции за агрегиране (count, max, etc.)
* •прост синтаксис за join операции
* •update и delete заявки за bulk промени на данни
* •под-заявки и параметризация на заявките

JPQL Пример:

private static final String QUERY = **"SELECT e.salary**

**FROM Employee e WHERE e.dept.name = :deptName"**

public long queryEmpSalary(String deptName) {

return em.**createQuery**(QUERY, Long.class)

.**setParameter**("**deptName**", deptName)

.**getSingleResult**();

}