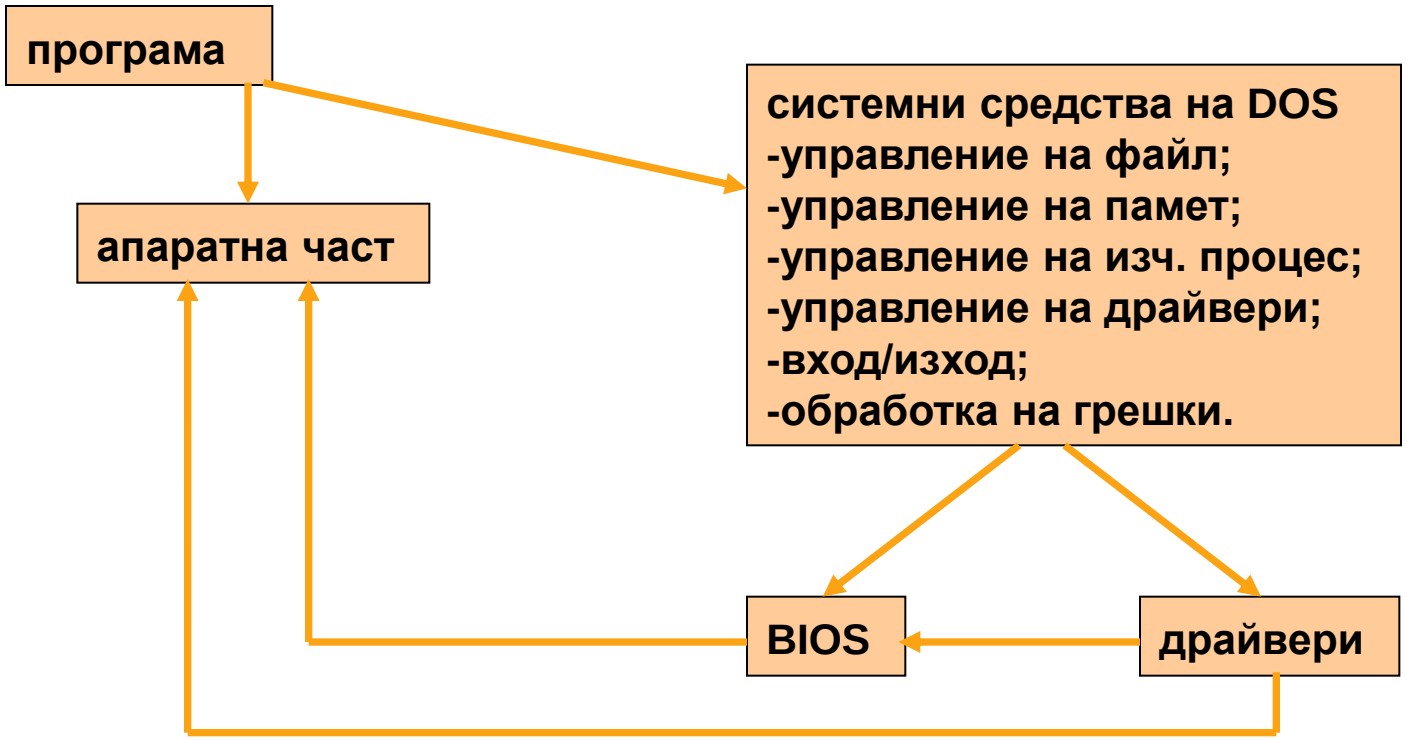
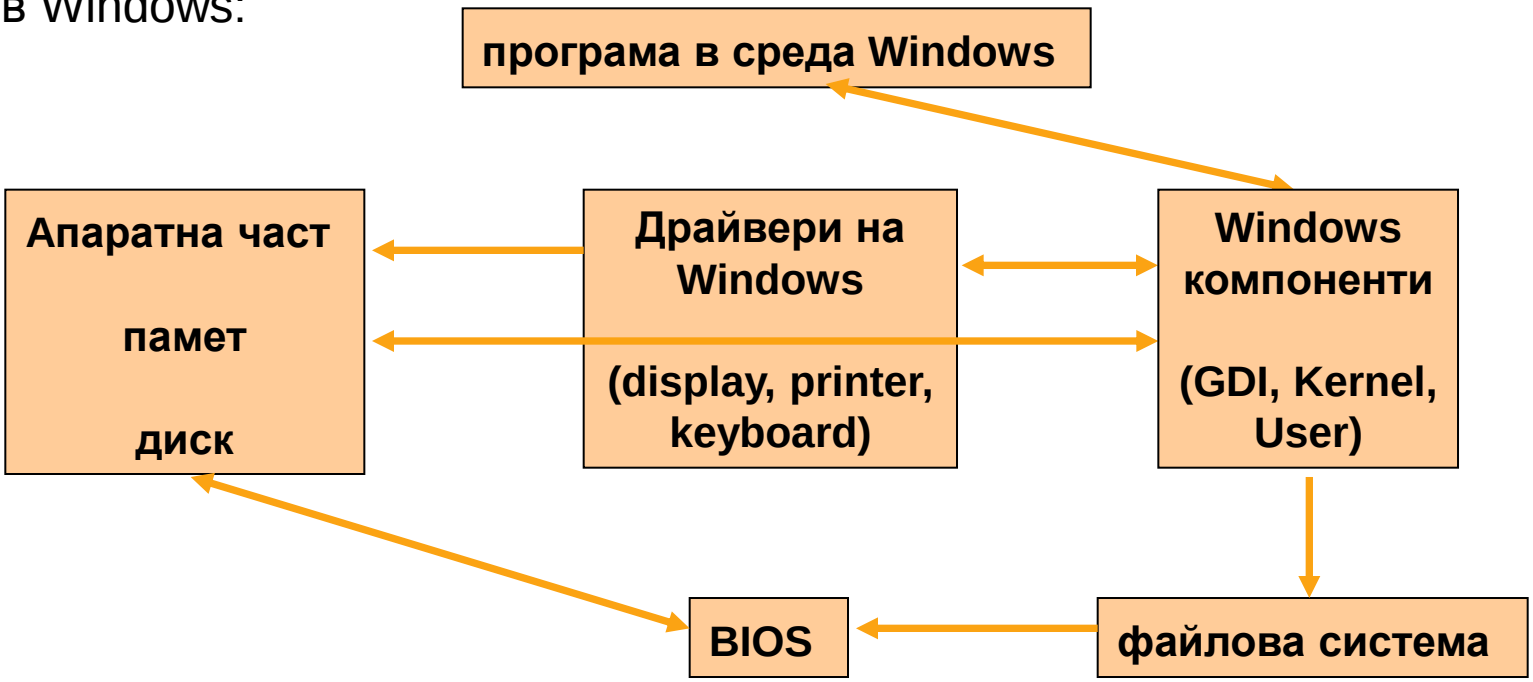
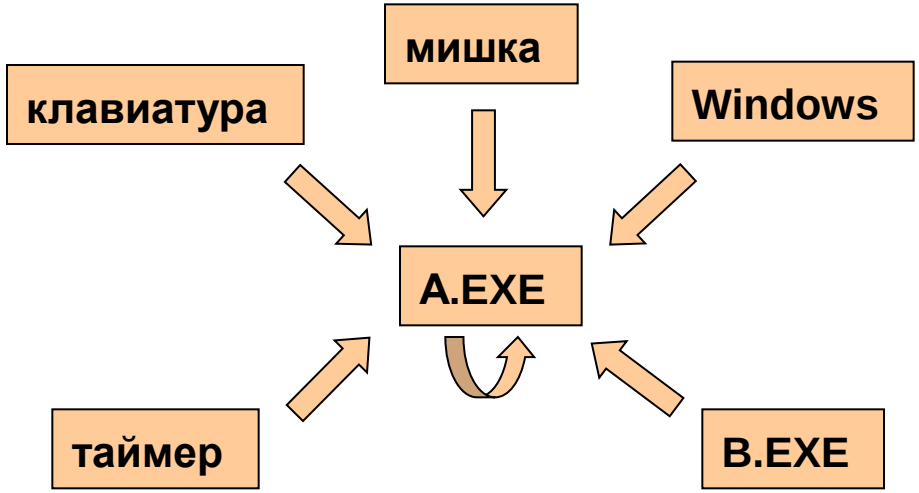
Въведение в програмирането в среда Windows

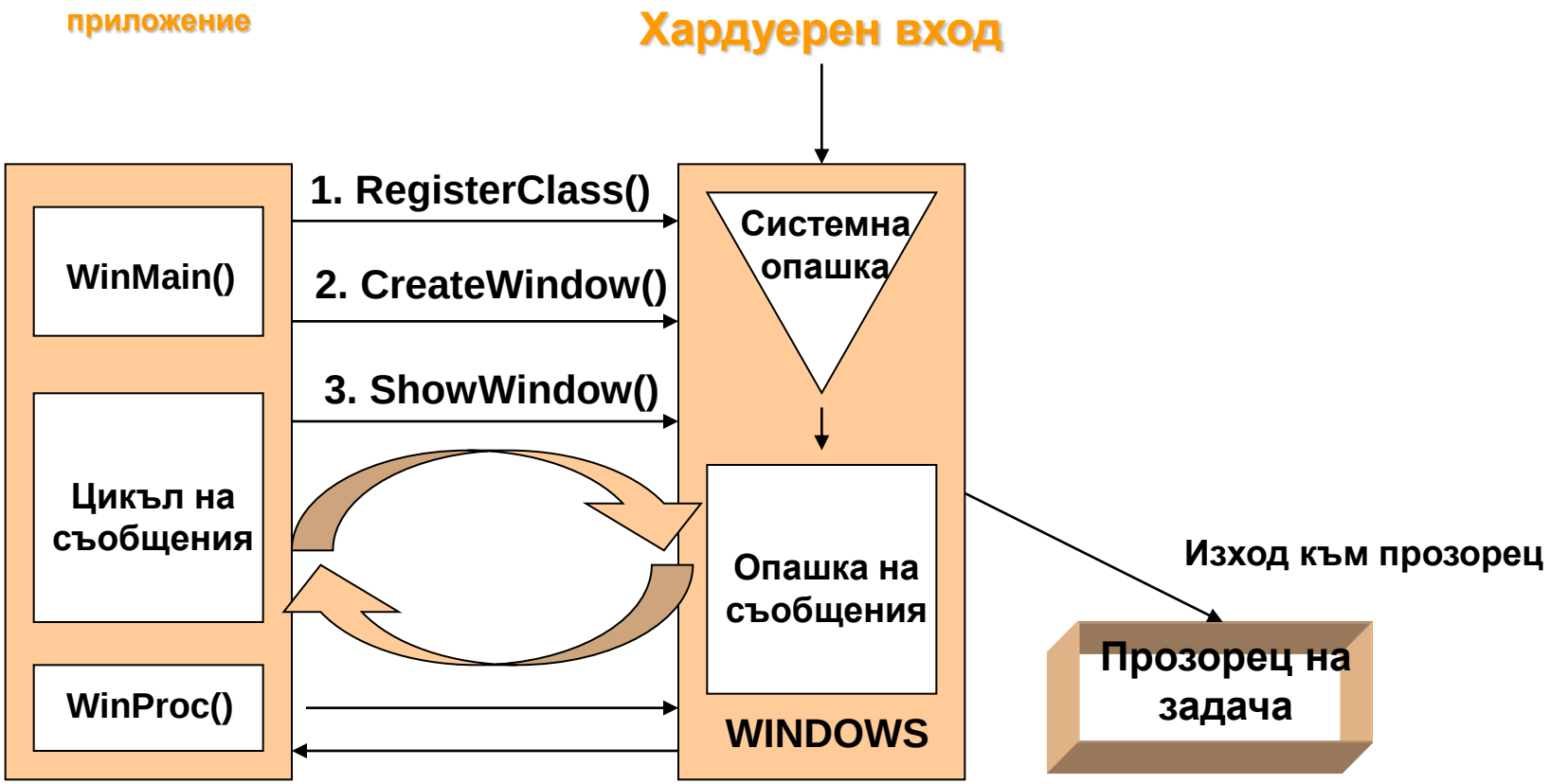
спомени от DOS (конзолно приложение):



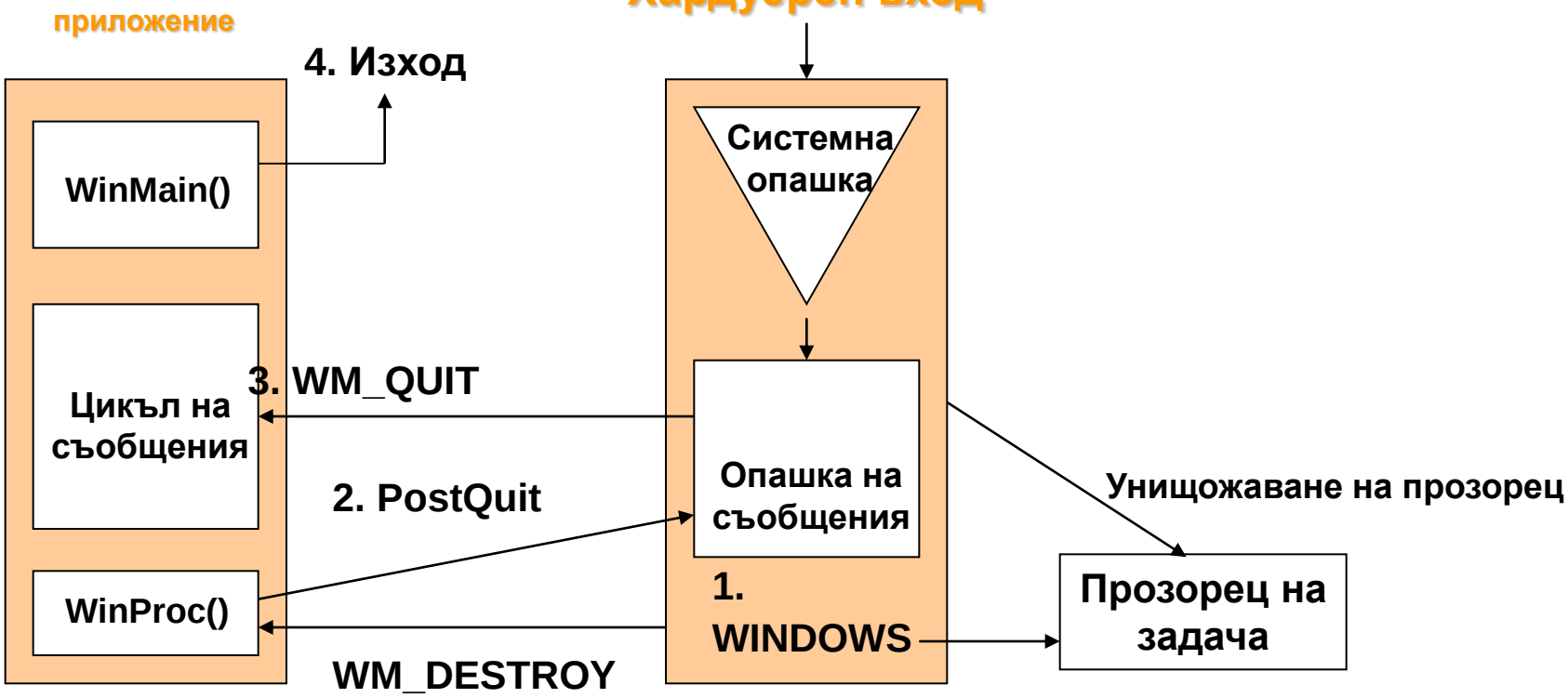
вече в Windows:



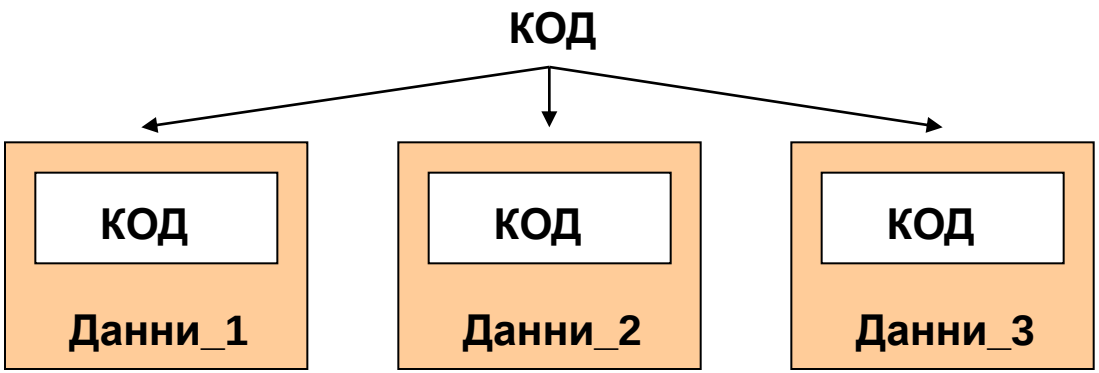



приложение
Хардуерен вход
1. RegisterClass()
2. CreateWindow()
3. ShowWindow()
WinMain()
Цикъл на съобщения
WinProc()
Системна опашка
Опашка на съобщения
WINDOWS
Изход към прозорец
Прозорец на задача

Хардуерен вход



Последователност на събития при изход от приложение



Стартовата точка към кода:

Добрият, стар DOS (конзолно приложение):

```
#include <stdio.h>
main()
{
        printf("Hello, world \n");
}
```

година на въвеждане	*език*	*интерфейс*
1985	C	Windows API
1992	C++	**MFC (Microsoft Foundation Class)**
2001	C# или C++	Windows Forms (част от .NET Framework)

-необходимост от компилатори и поддържаща среда (CLR – common language runtime);
-интегрирана развойна среда Visual Studio .NET (поддържа множество езици);
-безплатна версия: .NET Framework SDK;

версия 1 – старото Windows API

```
/* файл с разширение .C */
#include  <windows.h>
long FAR PASCAL WndProc(HWND, WORD, WORD, LONG);
int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance, LPSTR lpszCmdParam, int nCmdShow)
{           static char szAppName[] = “Hello”;
            HWND hwnd;
            MSG msg;
            WNDCLASS wndclass;

        if(!hPrevInstance)
                    {
                    wndclass.style = CS_HREDRAW | CS_VREDRAW ;
                    wndclass.lpfnWndProc = WndProc;          wndclass.cbClsExtra = 0;
                    wndclass.cbWndExtra = 0;        wndclass.hInstance = hInstance;
                    wndclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
                    wndclass.hCursor = LoadCursor(NULL, IDC_ARROW);
                    wndclass.hbrBackground = GetStockObject(WHITE_BRUSH);
                    wndclass.lpszMenuName = NULL;wndclass.lpszClassName = szAppName;
                    RegisterClass(&wndclass);
                    }
```

```
hwnd = CreateWindow (szAppName, “Name of the program”,
            WS_OVERLAPPEDWINDOW, CW_USEDEFAULT,
            CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, NULL, NULL,
                                                        hInstance, NULL);
ShowWindow(hwnd, nCmdShow);
UpdateWindow(hwnd);
```

```
while(GetMessage(&msg, NULL, 0, 0))
{           TransalteMessage(&msg);
            DispatchMessage(&msg);}
            return msg.wParam;  }
```

```
long FAR PASCAL WndProc(HWND whnd, WORD message, WORD wParam, LONG lParam)
{
  HDC hdc;
  PAINTSTRUCT ps;  RECT rect;
  switch(message)
          {case WM_PAINT:
                    hdc = BeginPaint(hwnd, &ps);
                    GetClientRect(hwnd, &rect);
                    DrawText(hdc, “Hello”,-1, &rect, DT_SINGLELINE | DT_CENTER |
                    DT_VCENTER);
                    EndPaint(hwnd, &ps);
                    return 0;
          case WM_DESTROY:
                    PostQuitMessage(0);
                    return 0;
          }
  return DefWindowProc( hwnd, message, wParam, lParam);
}
```

Само за информация:

```
/* помощен файл с разширение .DEF */
NAME Hello
Description “My first Windows program”          EXETYPE
Windows
STUB ‘WINSTUB.EXE’
CODE PRELOAD MOVEABLE, DISCARDABLE
DATA PRELOAD, MOVEABLE, MULTIPLE
HEAPSIZE 1024                    STACKSIZE 8196
EXPORTS WndProc
```

```
typedef struct tagMSG
{
        HWND hwnd;
        WORD message;
        WORD wParam;
        LONG   lParam;
        DWORD time;
        POINT pt;
} MSG;
```

версията на Hello.c е направо скандална:

150 реда код + евентуални ресурсни файлове за диалог и меню

Историята на цялото развитие от този момент е съсредоточена в борбата за намаляване на този код в нещо по-малко и елегантно

Малко допълващи пояснения

-**Порядък на четене и обработка на съобщения:**

= получени от PostMessage();
= Нормални съобщения (мишка, клавиатура);
= има ли инвалидизирана област?
Извиква се DoPaint() → отлага се WM_PAINT в опашката;
= проверка на флаг от timer, да → DoTimer() →WM_TIMER

особености на WM_PAINT:

*** винаги само 1 обобщено съобщение;**
*** InvalidateRect() поставя WM_PAINT;**
*** ValidateRect() валидизира указана област, която може да е различна от тази в WM_PAINT. Изчиства WM_PAINT от опашката**

Версия 2: работа в обектна среда:

В .h файл

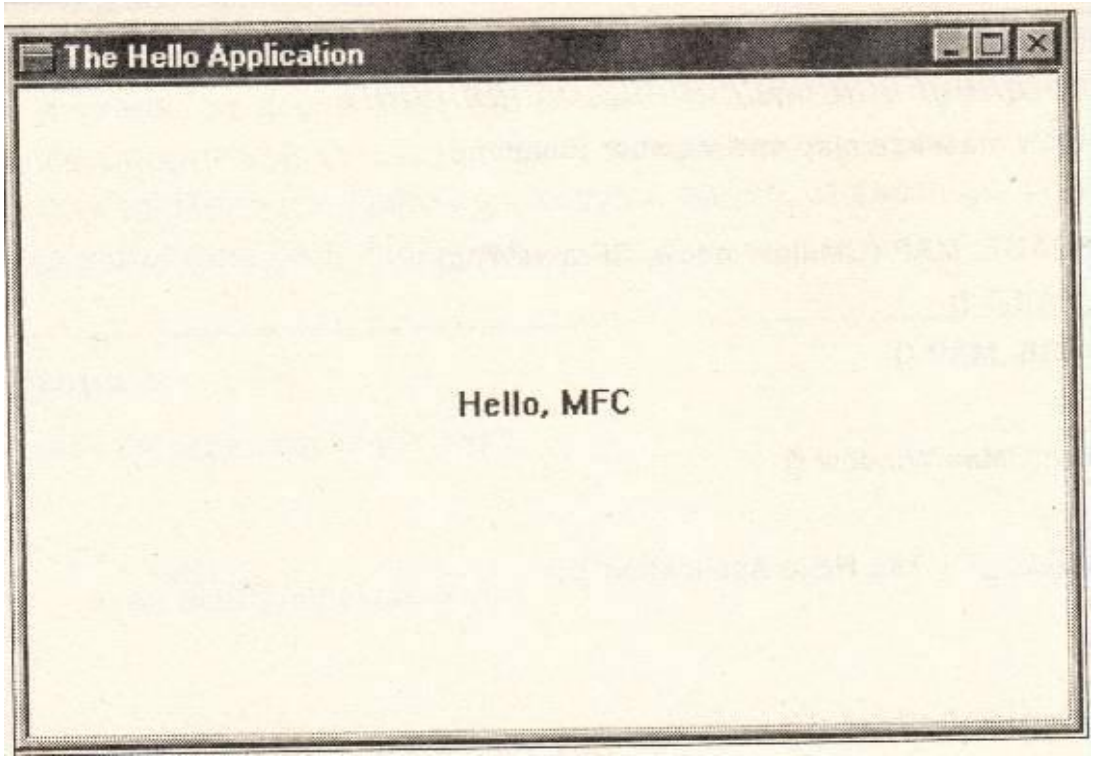
```
class CHelloApp : public CWinApp
        {public:                        BOOL InitInstance();        };
class CHelloWnd : public CFrameWnd
        {public:   CHelloWnd();
        protected:          afx_msg void OnPaint();DECLARE_MESSAGE_MAP()};
```

В .cpp файл

```
#include <afxwin.h>
#include “Hello.h”
BOOL CHelloApp :: InitInstance()
{           m_pMainWnd= new CHelloWnd;                  //m_pMainWnd е пром на CWinApp
m_pMainWnd->ShowWindow(m_nCmdShow );
m_pMainWnd->UpdateWindow();             return TRUE;        }
BEGIN_MESSAGE_MAP( CHelloWnd, CFrameWnd )
            ON_WM_PAINT()
END_MESSAGE_MAP()
CHelloWnd::CHelloWnd()
{                   Create( NULL, _T(“The Hello Application"));          }
void CHelloWnd::OnPaint()                                   // CWnd::OnPaint()
{           CRect rcClient;
            GetClientRect(&rcClient);
            CPaintDC dc( this );        // вика BeginPaint(),EndPaint() съответно в констр/деструктор
            dc.DrawText( _T(" Hello ,MFC"), -1, &rcClient,
                                        DT_SINGLELINE | DT_CENTER  |DT_VCENTER );
}

CHelloApp theApplication;
```

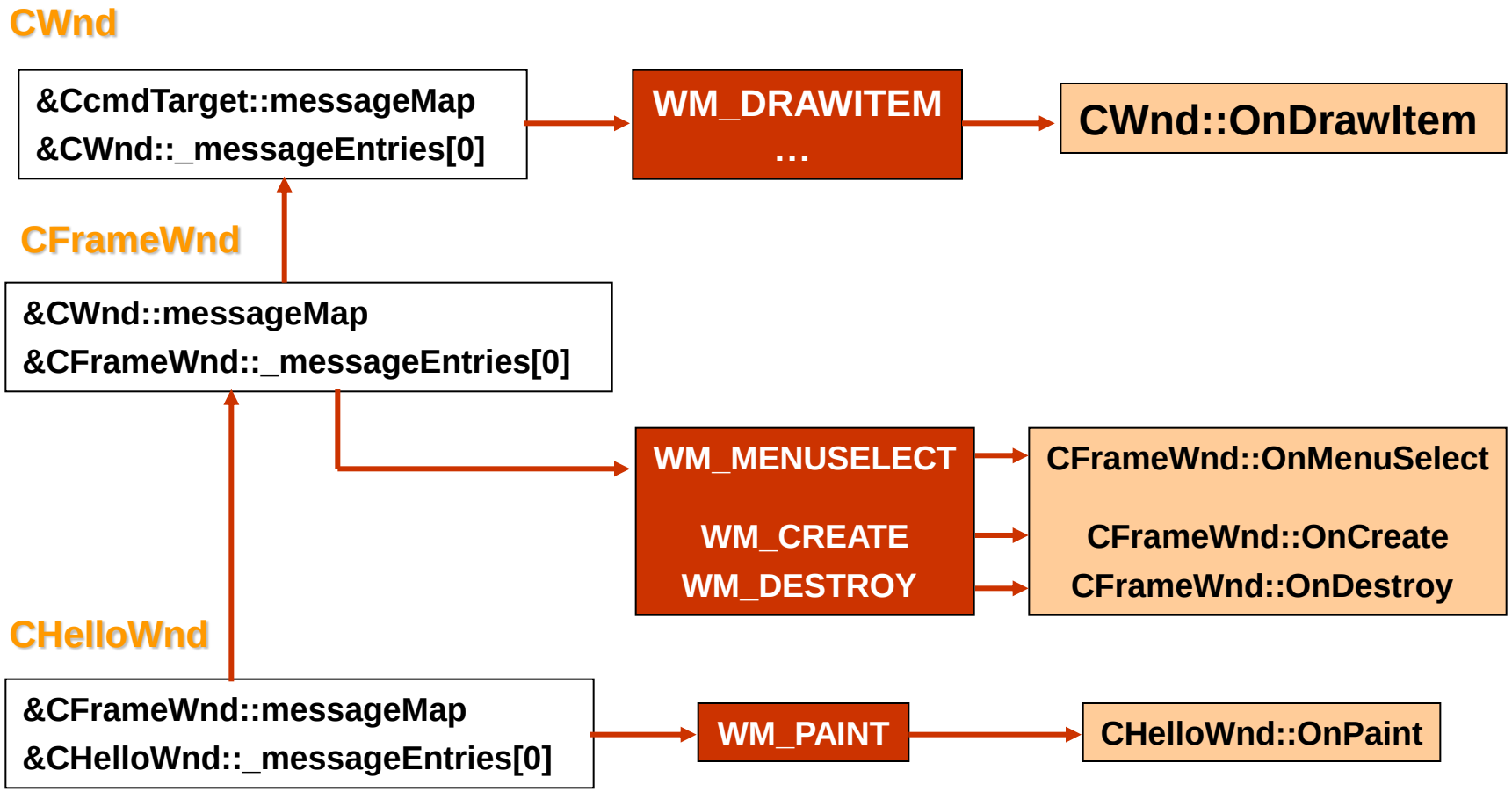
And the visualization produced by the previous code:



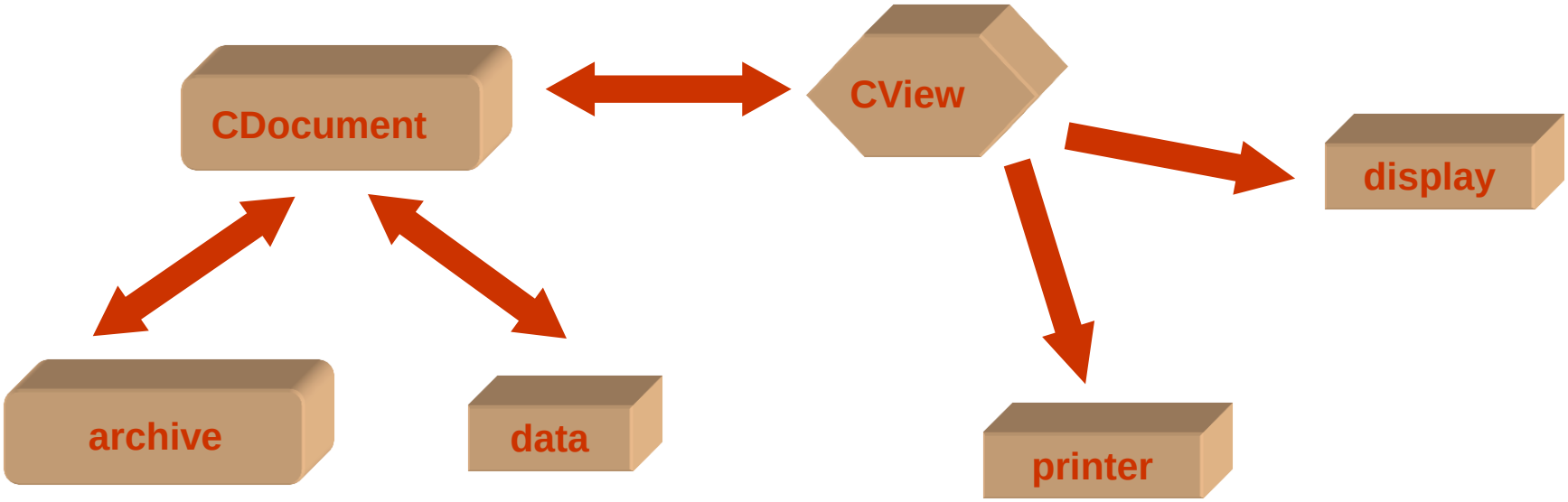
Съобщения и карта на съобщения

```
ON_MESSAGE(WM_MYMESSAGE, OnMyMessage)
afx_msg LPRESULT OnMyMessage(WPARAM wParam,LPARAM lParam);
```

message map структури _messageEntries масиви member функции



Версия 3 – в старата вече визуална среда




```
void CHelloView::OnDraw(CDC* pDC)                    //метод на CView
{
        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 );
}
```

версия 4 – преход към .NET (C# - ако въобще има разлика)

Вариант само с конзолно приложение:

Как така main(), който е метод на ConsoleHelloWorld е стартовата точка на програмата? Класът още не е създаден!!
ЗАТОВА main() е static !!!

namespaces; using..

HelloWorld.cs

```
class ConsoleHelloWorld
{
        pubic static void Main()
        {
                System.Console.WriteLine("Hello world!");
        }
}
```

-компилация (от команден ред) или от Visual Studio среда
- продуцира се малък по обем .exe файл на MS Intermediate Language (MSIL)
- след стартиране, .NET средата компилира междинния код към собствен машинен език за конкретния процесор и свързва с подходящите .DLL
-създал се е управляван код (managed code). Това е код, който може да се анализира от друга програма по време на изпълнение, за да се определя обхвата на действията в него. Това особено личи при управление на паметта и в Internet среда.

пространства от имена (над 90 са започващите със System);

- логически групира класове;
- съдържа: клас, структура, интерфейс, изброявания, делегати
-класове с еднакви имена в различни пространства (напр. Timer);
-класове извън всякакви пространства (напр. ConsoleHelloWorld) отиват в глобалното пространство на имена

Малко теория по именоване:

конвенция за именоване при програмиране в Windows

(Унгарска нотация)

1. **име на клас**

смес от малки и главни букви, започваща с главна и евентуално съдържаща вложени главни.

2. име на поле, променлива и обект

първата буква е малка, вътре може да включва големи букви

3. име на променлива от стандартен тип

има префикс от стандартни малки букви:

bool	**b**
byte	**by**
short	**s**
int	**l,x,y,cx,cy**
long	**l**
float	**f**
char	**ch**
string	**str**
object	**obj**

4. име на обект, инстанция на клас

малки букви свързани с или повтарящи името на класа. Напр:

Form	**form;**
PaintEventArgs	**pea;**

5. име на променлива – масив, започва с ‘a’

каква е разликата между конзолното и Windows-приложение:

1. начина на комилация: /target :exe и /target :/winexe за отделна компилация
и Console Application или Windows Application за Visual Studio.NET среда
2. конзолно приложение, стартирано през Windows създава прозорец за конзолен изход
Windows Application не създава конзолен прозорец и всеки изход към конзола
отива в небитието
3. За конзолно- не е нужна .NET CLR среда (поне System, System.Drawing, System.Windows.Forms)
4. Конзолно и windows приложения могат свободно да се смесват
5. реалната разлика е начинът по който програмата получава потребителски вход

вариант на нашата програма (само с диалогов прозорец, без форма):

```
class MessageBoxHellowWorld
{
        public static void Main()
        {
        System.Windows.Forms.MessageBox.Show("Hello world!");
        }
}
```

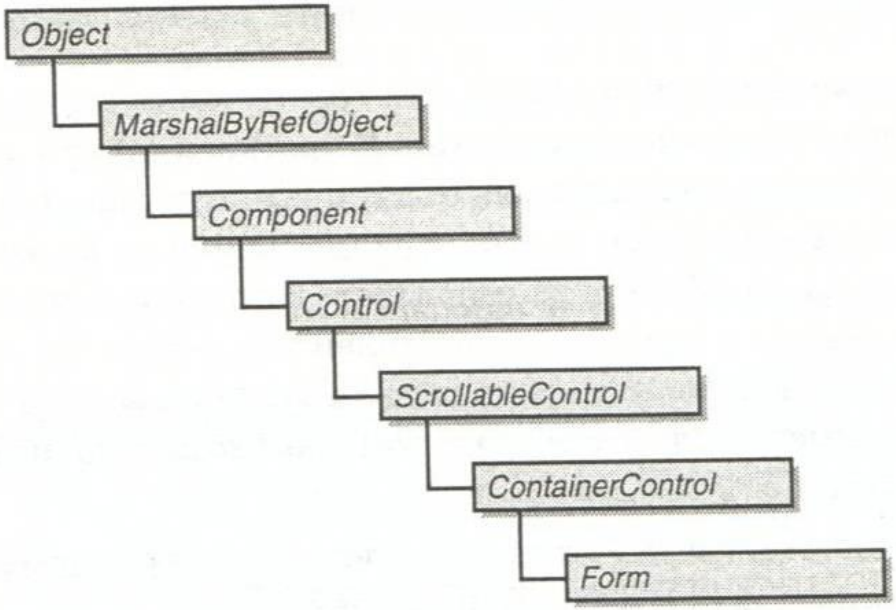
Единственият метод, освен тези на Object

System.Windows.Forms е пространство от имена с 200 класа и 100 изброявания, 40 делегата, 7 интерфейса и 4 структури

статичният метод Show() на MessageBox има 12 предефиниции. Позволява: различни бутони, икони в диалога,подразбиращ се бутон, enumeration DialogResult в което Show() връща информация за натиснат бутон и др.

Втори вариант на програмата – с форма (преди това беше ‘прозорец’)

(понятието ‘форма’ включва: caption bar, menu bar, client area)
В .NET класът Form е дефиниран в System.Windows.Forms и е наследник на:



*забележка****
конструкция от вида:

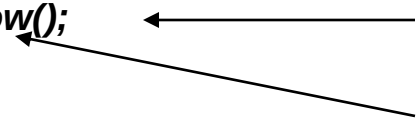
new System.Windows.Forms.Form();

създава, но не визулизира форма

Едва следващият код ще я визуализира:

```
using System.Windows.Forms;

class ShowForm
{
        public static void Main()
        {
                Form form = new Form();
                form.Show();
        }
}
```



алтернатива е:
form.Visible = true;

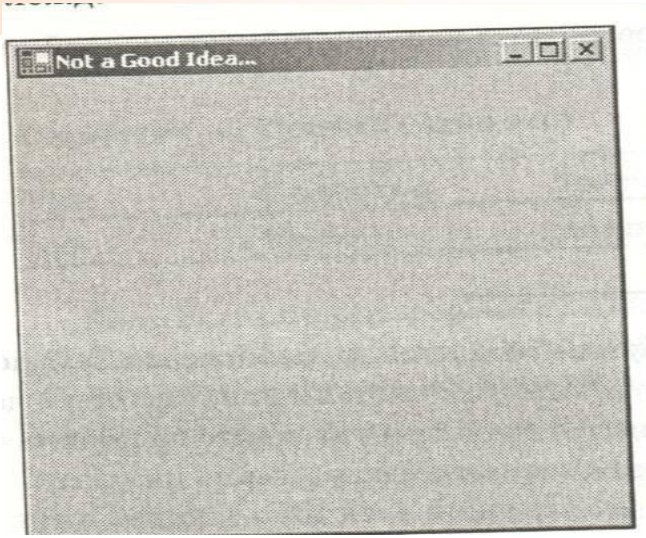
прави формата видима,
но само за миг.
Това е защото когато Windows приложение завърши, то изчиства след себе си всичко.
Това е друга разлика с конзолното приложение

Един начин да направим формата видима, например за 2+2 сек.
е като добавим :

```
using System.Threading;

form.Show();
Thread.Sleep(2000);
form.Text = “My form”;
Thread.Sleep(2000);
}
```

Вариант с форма и то работеща нормално:



да създадем приложение с форма:

```
using System.Windows.Forms;

class NewBadVersion
{
        public static void Main()
        {
                Form form = new Form();
                form.Text = "Not a Good Idea..";
                form.Visible = true;

                Application.Run();          //статичен метод на класа Application
        }
}
```

- вече имаме видима форма, вкл. поддръжка на мишка, местене, resize и т.н.
-за съжаление като затворим формата, приложението остава действащо като Task (Run() не връща управление, само прави формата невидима). Само с Ctrl+C ще ни спаси.

Да подобрим решението:

```
class NewBetterVersion
{
        public static void Main()
        {       Form form = new Form();
                form.Text = "better version";
                Application.Run(form);    }
}
```

Приложението завършва нормално; формата става видима автоматично. добавя се код за цикъла на съобщения. при затваряне, затварят се и всички форми, създадени в програмата.

свойства на формата:
Text
Visible
BackColor
Width
Height
FormBorderStyle
Cursor
StartPosition
…

Всичко е ОК, обаче къде да поставим нашия код, след като Run() не връща управление?

Или дискусия за: взаимодействие на форма с потребител

-събития (events). ***Очевидно, Run() някъде чака за събития.***
-събитията са толкова съществен елемент, че са вложени типове в .NET Framework
-събитията са и членове на класовете
-всяка програма може да дефинира свой event handler и го прикрепи към събитие
-прототипът на събитията съответства на делегатен тип
-това става така:
-1. имаме делегатен тип за събитието, дефиниран в някое namespace
-public delegate void PaintEventHandler(object objSender, PaintEventrgs pea);
-2. дефинираме свой статичен метод в класа, който ще обработва събитието
-static void MyPaintHandler(object objSender, PaintEventArgs pea) { ….}
-3. прикачаме своя манипулатор на събитие към исканото събитие
-form.Paint += new PaintEventHandler(MyPaintHandler);

-objSender е обектът към който се прилага събитието – form в случая;
-pea е клас , дефиниран в System.Windows.Forms и има 2 свойства:
-Graphics - свойството е инстанция на клас Graphics от System.Drawing
-ClipRectangle - инвалидизираният правоъгълник

.NET трета версия – форма + манипулатор на събитие (все още не е окончателна)

```
using System;
using System.Drawing;
using System.Windows.Forms;

class PaintEvent
{
        public static void Main()
        {
                Form form = new Form();
                form.Text   = “paint event”;
                form.Paint += new PaintEventHandler(MyHandler);
                Application.Run(form);
        }
static void MyHandler(object  objSender, PaintEventArgs pea)
        { //да изработим Graphics обект:за рисуване, писане и т.н в кл. област !!
                Graphics grfx = pea.Graphics;
                grfx.Clear(Color.Chocolate);
        }
}
```

using малко прилича на #include в C/C++. Всъщност, са по-сродни с ‘with’ на Visual Basic – не причиняват действия, само спестяват писане.

-не викайте MessageBox.Show() в манипулатор на събитие Paint:това инвалидизира част от кл. област и води до ново прерисуване и т.н.

- в манипулатор не правете нищо, което се натрупва. Напр. създаване на шрифт с удвоен размер (чрез промяна на property Font на класа Control или наследник.)
Това води до непрекъснато удвояване на размера на шрифта при всяко генериранер на събитие Paint!

трета-последна (вече реална) версия на програмата:
не се създава форма, а се наследява в наша форма


```
using System;
using System.Drawing;
using System.Windows.Forms;

class HelloWorld: Form
{
public static void Main()
        {
        Application.Run(new HelloWorld());
        }
public HelloWorld()    //това е конструктор по подразбиране. Той вика всички
                       //родителски конструктори, вкл и на Form
        {
        Text = "Hello World"; //няма нужда от HelloWorld.Text.    Може this.Text
        BackColor = Color.White;
        }
protected override void OnPaint(PaintEventArgs pea)
        {
        Graphics grfx = pea.Graphics;
        grfx.DrawString("Hello Windows Forms!", Font, Brushes, Black, 0,0);
}
```

липсва objSender, защото формата, към която се прилага повикването на OnPaint е винаги this

манипулатори на събития или On… методи ? Същността... за напреднали...

- Събитие можем да отработим с предефиниране на On.. метод
- Събитие можем да отработим и чрез код в наш манипулатор, който прикрепяме през eventHandler за клас, наследник на Control е препоръчително да се предефинира OnPaint()

protected override void OnPaint(PaintEventArgs pea) { ……}

подходът е препоръчителен за всички стандартни събития, дефинирани в Windows Forms

inside:

може да се допусне, че OnPaint() е преинсталиран манипулатора на събитие Paint?

Всъщност, OnPaint() в Control претърсва и вика всички инсталирани манипулатори за Paint, в който и да е наследник !!!

Последователността при наследяване от класа Control е следната:

1.Когато се вика On...() метод в наследяващ клас, то всъщност са предефинирани всички On...() методи на родителите и само кода в него ще бъде изпълнен

2. препоръчва се On...() методите в наследил клас да викат base.OnPaint(). Така ще се достигне до On...() метода със същото име в класа Control. Понякога това се пропуска.

(Това е задължително за предефинирани On…() методи на наследник на Form, когато ще дефинирате в бъдеще клас, съдържащ инстанция на горния клас в своя код и ако този нов клас ще добави свой манипулатор на същото събитие. Ако липсва base.OnPaint() например, в кода на On..() метода, събитието ще се прихване от кода на On…() метода на вградения клас, няма да се повика диспечера от Control и никога няма да се открие и изпълни кода на манипулатора в новия клас (виж Петцолд, стр 108))

3. Методът On...() на Control претърсва за всички инсталирани манипулатори за това събитие и последователно ги изпълнява.

4. След като всички инсталирани манипулатори за събитието са изпълнени, On...() от Control връща управлението на On...() метода в текущия клас

5. Едва тогава On..() метода на текущия клас се изпълнява