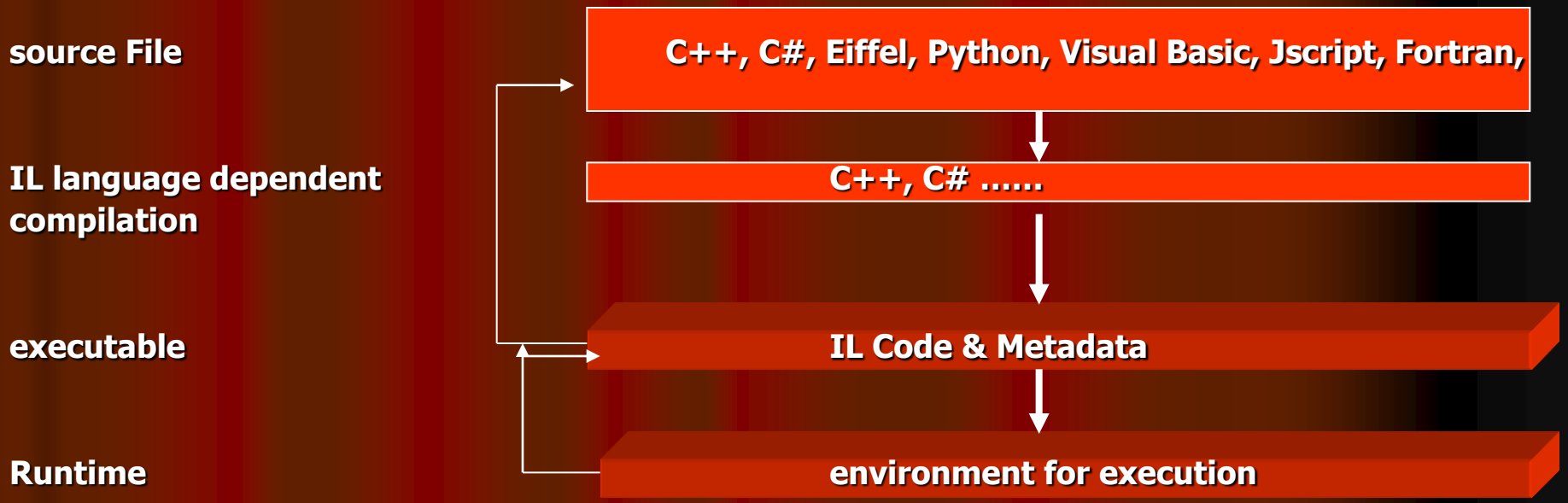
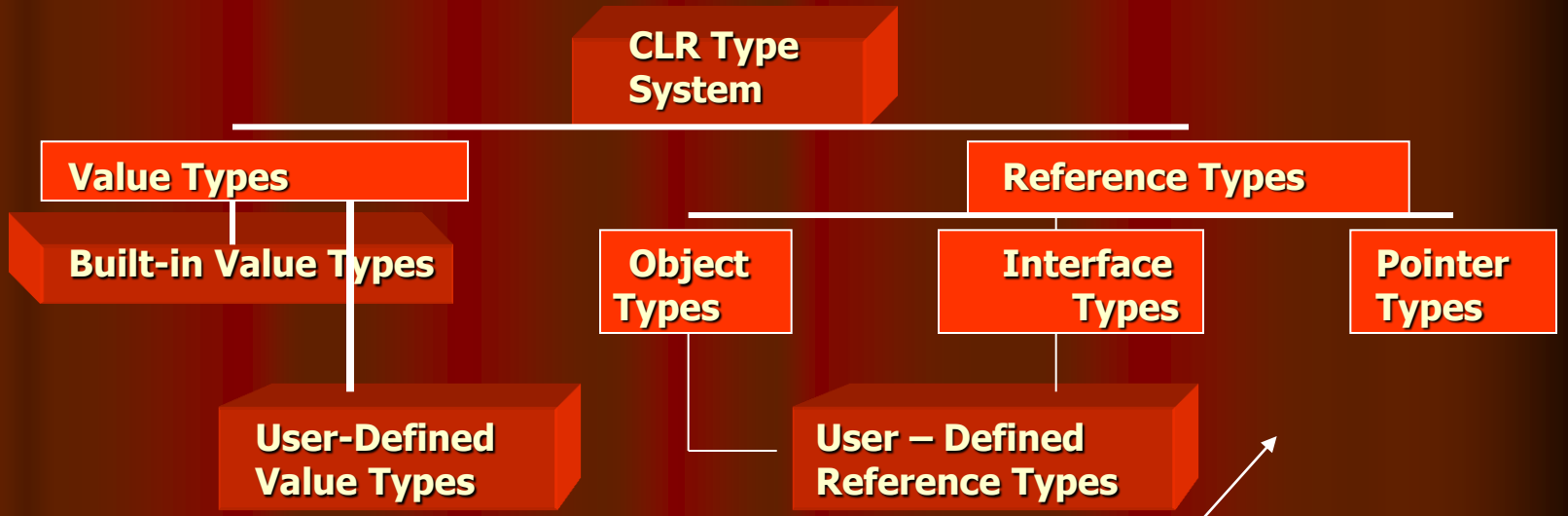
.NET Framework
environment for programming and common types control

* Support by many programming styles and languages
* Increased security, versions control
* Common types system





- Definitions of value type (inherited System.ValueType) and referent type.
- Value type can not be inherited. It's sealed. No virtual methods.
- Referent types can be created as child of any class(excluding ValueType and Enum).
- Memory is engaged from stack (value types) or heap (reference types). The last are managed by GC.
- Referent calls are strongly typed. We are speaking for: identity and equality.
- garbage collector moves objects **in the heap only** and associates new values for references accordingly.
- The programmer can not define own pointer type , but CLR is generating pointers to user-defined types, if needed

Value Types

Common IL

Framework Class Library **(FCL)**

in-line CIL	FCL support	description	CLS
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 int8	System.Byte	unsigned 8 bits	Y
unsigned int16	System.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. bool
3. characters 16 bits Unicode
4. Integers
5. Floating Point types

- User-defined Value Types: in stack; no default constructor; are of type enum or structures
  - enumerations

```
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;
                    }
            }
      }
```

❑ structures (C++ → class; C# → struct for value type or class for referent types)

- In the structure must be:
- **1. methods (static – can be called without instance of type and instance- can be call after an instance exists).**

No default constructor method is possible for them.

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

          static void Main(string[] args)
//starting point of the program is static method, not important if for a value or
// a reference type. The name may be different from main(),( in C# must be Main()
            { SayHelloType();
              Sample s = new Sample()
              s.SayHelloInstance();
            }
        }
 }
```

Като говорим за методи – да споменем и за метода – **конструктор на тип**
(това е нещо, различно от метода – конструктор на инстанция)

-Може да го има както при референтни, така и при стойностни типове
-Може да е само 1. никога няма параметри. Такъв не се създава по подразбране.
-Изглежда така:

```
class my_class{
        static My_class(){
        // код, който ще се изпълни първия път, когато имаме достъп го типа
```

-Конструкторът на тип се изпълнява само веднъж за цялото време на съществуване на типа в рамките на процеса.
-конструкторите на тип са и private, за да не могат да се викат външно, освен от CLR

2. fields of type -fields (static or instance). Possesses: type and name

3. properties(static or instance). "logical field" having type, name and set of methods that will control the manipulations with them. The compiler chooses the suitable method. Can be parameterless or parameterfull (called indexers in C#)

```
using System;
namespace ValueType
{   struct Point
   {private int xPosition, yPosition;
    public int X
            {       get {return xPosition;}        // method get_X is generating
                    set {xPosition = value;}       // method set_X is generating
            }
    public int Y
            {       get {return yPosition;}
                    set {yPosition = value;}
            }
    }
    class EntryPoint
     {      static void Main(string[] args)
            {       Point p = new Point();   //not in the heap - in the stack, "p" is of value type
                    p.X = 44;                //the compiler is generating  a call to set_X
                    p.Y = 55;
                    Console.WriteLine("X: {0}", p.X);
                    Console.WriteLine("Y: {0}", p.Y);
            }
     }
}
```

-**Methods that wrap the access to a field are called 'accessors'**

example:

```
public sealed class Employee {
        private String m_name;
        private Int32 m_age;
        public String Name {
                get { return(m_name);}
                set { m_name = value;} // the 'value' keyword always identifies the new value
                                }
        public Int32 Age {
                get { return(m_Age);}
                set {
                        if(value < 0)
                          throw new ArgumentOutOfRangeException( "value", value.ToString(),
                                "The value must be >=0");
                        m_Age = value;
        }}}


....
e.Name = "Nakov";
String new_name = e.Name;
e.Age = 100;
e.Age = -3;             //throws an exception
Int32 new_Age = e.Age
```

-Properties could be static, instance, virtual and so on;
-Properties can be marked with access modifier also;
-Each property has name and type (not a 'void');
- it's not possible to overload properties – with the same name and different type
- the compiler automatically defines get_xxx and set_xxx methods , where 'xxx' is the name of the property

Parameterfull properties (параметърни свойства)

Accepts 1 or more parameters. In C# are called 'indexers' (индексатори) .

Are exposed using array-like syntax:

```
public sealed class BitArray {
        private Byte[] byteArray;
        public BitArray(Int32 numBits) { …byteArray = new Byte[…]; }

        …..

        public Boolean this[Int32 bitPos] {
                get {……}
                set { …..}  // промяна на бит може да стане с операциите <<
                                // % 8 , § , | )
                                                }
}
```

Използване:

```
BitArray ba = new BitArray(12);
 ba[3] = ….;
 Console.WriteLine("Bit is" + (ba[3] ? "On" : "Off"));
}
```

Може да се счита, че индексаторът (в C#) е начин да се предефинира оператор '[]'

4. **events** (static or instance) are used to exponent **asynchrony** changes.
Must have a name and a type *(that is the signature of the callback method, used in the implementation in the client application – listener). After an event had been defined, methods of name add_EventName and remove_eventName are automatically defined and they' ill be used in the listeners associated with this event. Client objects are able to register its interest (became listeners). (a parallel with notifications used in COM technology can be made here)*
A type witch defined into some events informs other objects when event fires.
Other objects can register or unregister its interest to an event and can define own reaction to it.

- Our first short example:

```
using System;
namespace EventSample
{
        public delegate void ADelegate();
        struct EventClass
        {       public event ADelegate AnEvent;          //creates an event in this class
                public void InvokeEvent()
                 {      if (AnEvent != null)
                        AnEvent();
                 }
                static void CallMe()
                 {      Console.WriteLine(" I got called");)      }
```

```
static void Main(string[] args)
        { // listeners for this event are registered (add_...)
                EventClass e = new EventClass();
                e.AnEvent += new ADelegate(CallMe);
                e.AnEvent += new ADelegate(CallMe);
                e.AnEvent += new ADelegate(CallMe);
                e.InvokeEvent();
        }
}
}
```

- A. Дефиниране на тип, експониращ event (colored are the mandatory elements)

```
class EventManager     {
// 1.following is an inline type, that defines the information sent to event listeners
        public class MailMsgEventArgs : EventArgs {

            public MailMsgEventArgs( String from, String to, String subject, String body)
              { this.from = from; this.to = to;  this.subject = subject; this.body = body;}
                    public readonly String from, to, subject, body;
                                                                }
// 2.following is a delegate, that defines a prototype for a  callback method, that
// must be implemented by all listeners
```

Тип в System.EventArgs

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

//3. following is the definition of the event (listeners must implement such callback method)
public event MailMsgEventHandler MailMsg;
// 4. a method, responsible for the notification of objects- listeners for this event
protected virtual void OnMailMsg(MailMsgEventArgs e) //may be overloaded
        { if(MailMsg != null)                    //does exist a registered listener?
                {MailMsg( this. e);              //listeners are notificated
                }
        }

// 5. method that receives input data and translates them. Fires the event
public void SimulateArrivingMsg(String from, String to, String subject, String body)
        {       MailMsgEventArgs e = new MailMsgEventArgs(from, to, subject, body);
        //calls the method that notificates  all objects for the event
                OnMailMsg(e);
        }
}
```

- **Inside, the operator :** `public event MailMsgEventHandler MailMsg;`
  is transformed by the C# compiler in:

Винаги е private, независимо как е дефинирано събитието. Списъкът не е достъпен

1. делегатно поле (`private MailMsgEventHandler MailMsg = null`), that in the start moment is null, then initialized with a reference to a linked list of delegates, waiting to be informed for this event. The list is 'private'

2. defined new method: `public void add_MailMsg(MailMsgEventHandler handler)`
this method will add a new reference into the linked list in case of new listener.

3. new defined method `public void remove_MailMsg(MailMsgEventHandler handler)`
responsible for un-register all event handlers for objects that are lost its interest to the event.

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

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

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

- **Б. проектиране на тип, слушащ за събитие**

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

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

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

Докато обект е регистриран като събитие в друг обект – той не подлежи на GC обработки!

Sealed Value Type

* value type не се наследява
* не допуска virtual методи
* тип може да се маркира като sealed (C#, Sealed в CLR) – не може да бъде базов;
* тип може да се маркира като abstract (C#, Abstract в CLR) – не допуска инстанция, но може да се ползва за базов. Ако наследникът не е абстрактен, от него могат да се правят инстанции;

*засега не е допустимо Abstract Sealed. Затова, ако ще дефинирате собствен тип само със static членове , маркирате го sealed. Дефинирате му и private, безпараметърен конструктор (не може да се вика отвън за инстанция, а и пречи на компилатора да създаде свой по подразбиране) и така постигате абстракцията.
Такива са напр. обекти Math, Console.

Boxed types

Стойностните типове са олекотени, в стека а не в хипа, не се управляват от GC, не се сочат от указатели.
Това е достатъчно в много случаи, но не винаги...
Например, искаме да създадем масив (стандартен тип ArrayList pp;)
от точки- дефинирани като
struct Point{...}.
После да викаме pp.Add(my_point);
Или : Point new_point = (Point) pp[0];

Но точката е структура и е в стека. Как ще я вкараме в обект, в хипа и ще вземем референция към нея
как е допустимо и какво се е случило ?

- Всеки стойносен тип може да се конвертира в референтен (операцията е boxing)
- възможност за всеки value type, вкл. user-defined
- автоматично се генерира обвиващ обект (boxing) след копиране на стойността на value типа в boxed **обект** който се е алокирал в heap.
- Операция unboxing – връща value type в нормално състояние. Изработва указател към данната (до момента съдържана в boxed обект), копира данните на boxed обекта във value type (в стека).
- boxed обектите са 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 в компонентното програмиране.
- Всеки boxed type поддържа интерфейси ,чийто методи могат да се ползват.

Като имаме предвид казаното, следва да съобразим следствията от преобразуванията по-долу

```
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 String.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( o );         (2, 2)
                        (( Point) o).Change(3, 3);      // защото Change() е метод на Point
// " o " се разопакова. Полетата се копират във value type Point в стека
// и се променят на 3
                        Console.WriteLine( o );         // (2, 2) – това е "пакетираният" обект
```

Някои езици (C++) позволяват директна промяна в боксирания обект, но повечето (напр C#) – не . За да се спасим от проблема може да реализираме наследяване на интерфейс с интерфейсен метод Change() - това ще разгледаме по-късно)

Reference Types

* комбинира информация 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.Затова достъпът до тях е през силно типизирани референции, не директно

- A. **Object Types**: всички класове го наследяват; винаги са в GC heap; всички наследяват Object: System.Object

Основни методи на типа (с възможност за припокриване) са:

Equals (реализацията проверява или identity или equality);

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);

ToString (връща информация за типа, ако не се припокрие).

```
using System;
namespace Override
{
        class Sample
        { static void Print( params Object[] objects)
                { foreach(Object o in objects)
                        Console.WriteLine( o.ToString());
// вика подходящия  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, вика се собствения ToString()
        }                       }}
```

предаване променлив брой параметри:

```
static Int32 Add( params Int32[] values )
{
        Int32 sum = 0;
        for( Int32 x = 0; x< values.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 в CLR вграден тип (освен Object) е String

*** той е sealed**

*** всеки метод, модифициращ низ, създава нов низ (immutable)**

*** основни методи:**

конструктори (няма default) за различни аргументи – char, масив char, указатели 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

// преобразува се (boxed) в тип обект:  String + Object ?
// съществува overrided concatenation метод на  класа String с параметър Object.
//Той се  ползва

        Console.WriteLine(s.ToLower());          // създава се временен string object
                }
                        }}
```

Поради свойството immutable на низовете, програмна конструкция от вида:

str.ToUpper();
макар и валидна, няма да направи нищо (str променливата не е тази, която ще се промени). По-добре:

string strUpper = str.ToUpper();

Класът String имплементира IComparable, ICloneable, IConvertible, IEnumerable
И следователно методите в тях.

Проблеми по производителността, породени от честото пресъздаване на String обекти:

```
string str = String.Empty;
For (int I = 0; i<10000;i++)
            str +="asdfghj";
...
//всяко слепване води до създаване на нов низ и заделя памет.Всеки предишен
//низ се маркира за унищожаване. 10 000 низа → бавно и непроизводително!
```

По-удачно е използване на типа StringBuilder дефиниран в System.Text:
(там низът може да бъде променян чрез динамично презаделяне на памет с удвояване при всяко доближаване до границата на заделения буфер).
При това конвертирането е свободно:

```
string str = strbuilder.ToString()          // прави преобразуване към низ
```

Подобрената предишна програма с използване на този клас изглежда така:

```
…
StringBuilder sb = new StringBuilder();
 for( i = 0; i< 10000; i++)
 sb.Append("asdfgh");

 string str = sb.ToString();

// тази версия на програмата работи хиляди пъти по-бързо
```

Б. Въведение в 'Interface Types'

* въведени в COM и CORBA
* целта е не само функционална наследяемост(inheritance), но и възможност за поделяне на общ външен договор (интерфейс) между несвързани обекти.
* интерфейсният тип е частична спецификация на тип
* поддържа се наследяемост на интерфейсни типове
* интерфейсният тип съдържа:

    методи (статични и на инстанция)
    полета (статични) но не и в C#
    property
    events

* методите на интерфейс са public, abstract, virtual
* Framework поддържа множество интерфейсни типове
* CLR не поддържа множествено наследяване на типове, но поддържа такова за интерфейси

Пример с използване на масив и преобразуване към интерфейс

```
using System;
using System.Collection;

namespace StringArray
{       class EntryPoint
        {               static void Main(string[] args)
                        {String [] names = {“ Nakov”, Gotceva”, Najdenov”};
// класът Array за всякакви типове се създава автоматично при нужда и  винаги
// поддържа IEnumerator ( Array поддържа и: Clear(); GetLenght();Sort(),IsSynchronized().. )
// Sort() използва  IComparable, който следва да е реализиран от array типа

                        IEnumerator i = names.GetEnumerator();
                        while(i.MoveNext())
                                Console.WriteLine(i.Current);
                        }
        }
}
```

В. Pointer type

биват:

unmanaged function pointer
managed pointers (CLS съвместими и GC управляеми)
unmanaged pointer referring values (CLS несъвместими).Не всички езици ги поддържат
синтаксисът е езиково специфичен

Г. User- Defined Object Type

```
using System;
namespace ObjectType

        { public delegate void ADelegate();
        public interface IChanged
                    { event ADelegate AnEvent;  }
        public interface IPoint
                    {
                     int X { get; set}
                     int Y {get; set;}
                    }
```

```
class Point: IChanged, IPoint
{
        private int xPosition, yPosition;
        public event ADelegate AnEvent;
        public int  X
        {       get { return xPosition;}
                set { xPosition = value; AnEvent();}
        }
        public int Y
        {
                get {return xPosition;}
                set { yPosition = value; AnEvent();}
        }
class EntryPoint
{
        static void CallMe()                    // делегат
                { Console.WriteLine(" I got called")'          }
        static void Main(string [] args)
                { Point p = new Point();
                IChanged ic = p;
                IPoint ip = p;
                ic.AnEvent          += new ADelegate(CallMe);
                ip.X = 44;          // интерфейсът вика само своите методи
                ip.Y = 55;
                Console.WriteLine("X: {0}  Y:  {1}", p.X, p.Y);
                }
}}
```

Д. Използване на интерфейси с Value типове

1.можем да викаме интерфейсен метод и през value тип

2. Ако викаме метод на value тип през интерфейсния указател(който е в хипа), минаваме през пакетиране и прехвърляне през heap

```
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.
                        }
            }}
```

Е. Присвояване и съвместимост

правила:

1. при съвпадение на типовете;
2. когато присвояваният обект е от подтип;
3. 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 "o" съдържа  i . Свива се обсега  на boxed обект
    към подобект
                    }
        }}
```

Делегати

(Рихтер,430стр)

- callback функции в Windows
- callback функции, реализиращи специфични обработки: qsort от runtime библиотеката приема callback ф-ия , сортираща елементите на масив
- callback функциите не носят инф. за брой и тип параметри, връщана стойност, конвенция на повикване.
  т.е. callback функциите не са типово-обезопасени

.NET и делегатите

```
using System;
using Syste.Windows.Forms;
using System.IO;

class Set
{       private Object[] items;
        public Set(Int32 numItems)
                { items = new Object[numItems];
                for( Int32 i = 0; i< numItems; i++)       items[i] = i;}

        public delegate void Feedback( Object value, Int32 item, Int32 numItems);
        // декларира се public делегат и се описва сигнатурата му
```

```
        public void ProcessItems(Feedback feedback)        // параметърът е референция
                    // към обект за Feedback делегат, който се ползва за callback повиквания
                    { for( Int32 item = 0; item<items.Lenght; item++)
Не е
функция !!!                 {if(feedback != null)
                                { feedback(items[item], item+1; item.Lenght);      }
                                / /  ако има callback ф-ии, те се извикват за всеки обект
                    }}
}

class App           // тук ще викаме статични и методи на инстанция, посредством делегати
        { static void Main()
                    {StaticCallbacks();
                    InstanceCallbacks()
                    }
        static void StaticCallbacks()
                    {       Set setOfItems = new Set(5);
                            setOfItems.ProcessItems(null);          // 1. без callback методи
                            setOfItems.ProcessItems( new
Set.Feedback(App.FeedbackToConsole));  // 2. конструира се делегат от тип Feedback
                                        // обвиващ метода FeedbackToConsole() )
                            setOfItems.ProcessItems( new
Set.Feedback(App.FeedbackToMsgBox));            // 3.
                                        Set.Feedback fb = null; // ще създаваме верига делегати
                            fb += new Set.Feedback(App.FeedbackToConsole);
                            fb += new Set.Feedback(App.FeedbackToMsgBox);
                            setOfItems.ProcessItems(fb); // 4. за всеки елемент, през който се
// итерира, ще се викат последователно всички делегатни методи от списъка
                    }
```

```
static void FeedbackToConsole( Object value, Int32 item, Int32 numItems)
        { ……….…….}
static void FeedbackToMsgBox( Object value, Int32 item, Int32 numItems)
        { ……….…….}
}
static void InstanceCallbacks()
        {
        Set setOfItems = new Set(5);
        App appobj = new App(); //обектът е нужен само за instance метода си
        setOfItems.ProcessItems( new
                                 Set.Feedback(appobj.FeedbackToFile));
        }
void FeedbackToFile( Object value, Int32 item, Int32 numItems_
                { ……….…….}
}
```

- 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 осигуряват поддръжката за делегати

Делегатът
Е
Клас !!!

имахме в кода следния оператор:

```
public delegate void Feedback( Object value, Int32 item, Int32 numItems);
```

компилаторът дефинира клас за делегата:

```
public class Feedback : System.MulticastDelegate        // класът е public, защото делегатът е public
{           public Feedback( Object target, Int32 methodPtr);  // конструктор. сигнатурата му е различна??!!!
public void virtual Invoke( Object value, Int32 item, Int32 numItems);
                                                  // методът има същият прототип като делегата
// следват методи осигуряващи асинхронното повикване на callback метода
public virtual IAsyncResult BeginInvoke( Object value, Int32 item, Int32
           numItems,AsyncCallback callback, Object object);
public virtual void EndInvoke(IAsyncResulet result);
}
```

- Делегатът е клас, като всеки друг клас !!
- Делегатът наследява MulticastDelegate и важните полета: _target; _methodPtr (за статични методи е null, за останалите – Int32 индекс от метаданните), _prev (при вериги от делегати)
- делегатният обект е обвивка около метод и обект , с който методът работи
- Обръщението към callback метод е като с ф-ия (но такава няма !!). 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 някои езици (от C# - не, от VB – да). Методът знае кой метод (_methodPtr) и за кой обект (_target) от своя страна да повика.