Масиви

Масив в C# е обект:

– базов клас System.Array**;**

– свойство Length **– брой на елементите в масива.**

1. Декларация на масив

<тип>[ ] <име_на_масив>;

2. Създаване на масив

<име_на_масив> = new <тип>[размер];

Пример: Едномерен масив

```
using System;
class SingleArray
{   private int[] a;
    public SingleArray (int number)
    {   a = new int [number];
        Console.WriteLine ("Въведете {0} цели числа.",number);
        for (int i = 0; i < number; i++)
        {   Console.Write ("a[{0}]=", i);
            a[i] = int.Parse(Console.ReadLine());
        }
    }
    public void PrintArray()
    {   Console.WriteLine ("Масив");
        for (int i = 0; i < a.Length; i++)   Console.Write (a[i] + " ");
        Console.WriteLine ();
    }
    static void Main()
    {   SingleArray x = new SingleArray(5);
        x.PrintArray ();
    }
}
```

3. Многомерен масив

<тип>[, , ...,] <име_на_масив>;

или

<тип>[][]...[] <име_на_масив>;

Метод Array.GetLength (int) **– определя дължината на всеки размер на масива.**

Метод Aray.Rank () **– връща реда (броя на размерностите) на масива.**

Пример: Двумерен масив

```
using System;
class TwoDimArray
{
    private float[,] a;
    public TwoDimArray (int number1, int number2)
    {
        a = new float [number1, number2];
        Console.WriteLine ("Въведете матрица {0}x{1} " +
                    "от реални числа.", number1, number2);
        for (int i = 0; i < a.GetLength(0); i++)
            for (int j = 0; j < a.GetLength(1); j++)
            {
                Console.Write ("a[{0},{1}]=", i, j);
                a[i,j] = float.Parse (Console.ReadLine());
            }
    }
```

```
    public void PrintArray()
    {
        Console.WriteLine ("Масив");
        for (int i = 0; i < a.GetLength(0); i++)
        {   for (int j = 0; j < a.GetLength(1); j++)
                Console.Write ("{0,5:f2}", a[i,j]);
            Console.WriteLine ();
        }
    }
    static void Main()
    {
        TwoDimArray x = new TwoDimArray (5, 3);
        x.PrintArray ();
    }
}
```

Пример: Масив с различна дължина на редовете – масивът се разглежда като масив от масиви

```
using System;
class JaggedArray
{
    private int[][] a;
    public JaggedArray (int rows, params int[] n)
    {
        a = new int [rows][];
        for (int i = 0; i < rows; i++)
            a[i] = new int [n[i]];
        for (int i = 0; i < a.Length; i++)
        {   Console.WriteLine("Въведете {0} цели числа.", a[i].Length);
            for (int j = 0; j < a[i].Length; j++)
            {   Console.Write ("a[{0},{1}]=", i, j);
                a[i][j] = int.Parse (Console.ReadLine ());
            }
        }
    }
```

```
    public void PrintArray ()
    {
        Console.WriteLine("Масив с различна дължина на редовете");
        for (int i = 0; i < a.Length; i++)
        {
            for (int j = 0; j < a[i].Length; j++)
                Console.Write ("{0,5}", a[i][j]);
            Console.WriteLine ();
        }
    }
    static void Main()
    {
        JaggedArray x = new JaggedArray (3, 2, 5, 3);
        x.PrintArray ();
    }
}
```

Индексатори (Indexers)

Индексатор – разглежда обект като масив; използва се като елегантен масив.

1. **Дефиниране на индексатор**
   а) има индекс като аргумент;
   б) this **се използва като име на индексатор.**

```
class <име_на_клас>
{
   public <тип> this [<тип_индекс> индекс]
   {
      get {  // Връща необходимите данни        }
      set {  // Установява необходимите данни  }
   }
}
```

2. **Създаване на индексатор – създаваме обект от този клас и разглеждаме обекта като масив:**

```
<име_на_клас> <име_на_обект> = new <име_на_клас>();
<име_на_обект> [индекс] = <произволен_обект>;
```

Пример: Масив от низове (списък от низове)

Клас System.Collections.ArrayList **– създава динамичен масив от обекти.**

Метод ArrayList.Add **– добавя обект в края на колекцията.**

Свойство ArrayList.Count **– връща текущия брой на елементите в колекцията.**

```
using System;
using System.Collections;
class MyListBox
{  protected ArrayList data = new ArrayList();
   public object this[int idx]
   {
     get
      {  if (idx>-1 && idx < data.Count)
           return data[idx];
         else
           throw new InvalidOperationException("Индексът е извън областта");
      }
     set
     {   if (idx >-1 && idx < data.Count)
           data[idx] = value;
         else if (idx == data.Count)
           data.Add (value);
         else
          throw new InvalidOperationException ("Индексът е извън областта");
       }
   }
}
```

```
class IndexersApp
{
   static void Main()
   {
      MyListBox list = new MyListBox();
      list[0] = "Георги"; list[1] = "група 80"; list[2] = "ФКСУ";
      Console.WriteLine ("{0}, {1}, {2}", list[0], list[1], list[2]);
    }
}
```

Пример: Шахматна дъска 8x8 с индексатор с два параметъра: първият се изменя от A до H, а вторият – от 0 до 7.

```
using System;
class Grid
{
   const int NumRows = 8;
   const int NumCols  = 8;
   string[,] cells = new string[NumRows, NumCols];
   public string this[char c, int col]
   {
       get
       {
            c = Char.ToUpper (c);
            if (c < 'A' || c > 'H')
            {
                throw new ArgumentException();
            }
            if (col < 0 || col > NumCols-1)
            {
               throw new IndexOutOfRangeException();
            }
            return cells[c - 'A', col];
        }
```

```
        set
        {
          c = Char.ToUpper(c);
          if (c < 'A' || c > 'H')
          {
              throw new ArgumentException();
          }
          if (col < 0 || col > NumCols-1)
          {
              throw new IndexOutOfRangeException();
          }
          cells[c - 'A', col] = value;
      }
  }
}
class Test
{   static void Main()
    {    Grid g = new Grid();
         g['A',0] = "топ";
         g['H',7] = "пешка";
    }
}
```

Обработка на низове

Низ – последователност от символи.

Клас System.String **(псевдоним** string**) представя константен символен низ.**

Класове за обработка на низове:

- StringBuilder
- StringFormat
- StringCollection
- **и др.**

Пример:

```
public string Replace (char oldChar, char newChar);
public string Replace (string oldValue, string newValue);
public string Insert (int startIndex, string value);
public string ToUpper();
```

```
using System;
class TestStringApp
{
   static void Main (string[] args)
   {
      string a = "приход";
      // Замества всяко 'и' с 'е'
      string b = a.Replace ('и', 'е');
      Console.WriteLine (b);                  // преход
      string c = b.Insert (3, "раз");         // преразход
      string d = c.ToUpper ();
      Console.WriteLine (d);                  // ПРЕРАЗХОД
   }
}
```

Пример:

```
public int CompareTo (string strB);
// Сравнение на низове (CompareTo) и предефиниран оператор ==
if (d==c)                                  // или if (d.CompareTo (c) == 0)
   Console.WriteLine ("еднакви");
else
   Console.WriteLine ("различни");
```

Резултати:

различни

Пример:

```
// Replace не променя низа.
// Резултатът се присвоява на променливата.
string q = "приход";
q = q.Replace ('и', 'е');
Console.WriteLine (q);
```

Пример:

```
public string Substring (int startIndex);
public string Substring (int startIndex, int length);
```

```
// Слепване и индексиране на низ
string e = "Здравей"+", ";
e += "приятел";
Console.WriteLine (e);                     // Здравей, приятел
string f = e.Substring(1,7);
Console.WriteLine (f);                     // дравей,
for (int i = 0; i < f.Length; i++)
     Console.Write("{0,-3}", f[i]);
Console.WriteLine();                       // д р а в е й ,
```

Пример:

```
public bool StartsWith(string value);
public string Remove(int startIndex, int count);
```

```
string g = null;
if (f.StartsWith ("др"))
      g = f.Remove(2,3);
Console.WriteLine (g);              // дрй,
```

Пример:

```
public static string Format(string format, params object[] args);
```

```
int x = 16;
decimal y = 3.57m;
string h = String.Format("Артикул {0} се продава за {1:C}", x, y);
Console.WriteLine (h);       // Артикул 16 се продава за 3,57 лв
```

Пример:

```
// Слепване на низ с произволен тип данна
// (всички типове наследяват object.ToString)
string t = "Артикул "+12+" се продава за "+3.45+" лв";
Console.WriteLine (t);
```

Резултати:

```
Артикул 12 се продава за 3,45 лв
```

Пример:

```
// String.Format и Console.WriteLine имат последен
// аргумент params object[]
Console.WriteLine("Здравей {0} {1} {2} {3} {4} {5} {6} {7} {8}",
                        123, 45.67, true, 'A', 4, 5, 6, 7, '8');
string u=String.Format("Здравей {0} {1} {2} {3} {4} {5} {6} {7} {8}",
                          123, 45.67, true, 'A', 4, 5, 6, 7, '8');
Console.WriteLine (u);
```

Резултати:

```
Здравей 123 45,67 True A 4 5 6 7 8
Здравей 123 45,67 True A 4 5 6 7 8
```

Пример:

```
public string[] Split ();
char[] separator = new char[]{' ',',','.',':','/','\\','\''};
```

```
// Методът String.Split разцепва низ на поднизове чрез
// разделителни символи
string str = "Две хубави очи";
char[] seps=new char[]{' '};
string[] s = str.Split (seps);
foreach (string ss in s)
    Console.WriteLine (ss);
```

Резултати:

```
Две
хубави
очи
```

Недостатък: Split **не работи добре, когато един до друг има няколко разделителя (получава се празен ред).**

```
public string[] Split(params char[] separator,
                      StringSplitOptions.RemoveEmptyEntries);
```

Клас System.Text.StringBuilder **– представя променлив символен низ.**

Пример:

```
public StringBuilder Append (string value);
public StringBuilder AppendFormat (string format, params object[] args);
using System.Text;
```

```
StringBuilder sb =new StringBuilder("Приход");          // Приход
sb.Replace ('и', 'е');                                  // Преход
sb.Insert (3, "раз");                                   // Преразход
sb.Append (" на средства");                  // Преразход на средства
sb.AppendFormat (",{0}:{1}", 123, 45.6789);
                                  // Преразход на средства,123:45.6789
sb.Remove (sb.Length-3, 3);
                                  // Преразход на средства,123:45.6
Console.WriteLine (sb.ToString().ToUpper());
                                  // ПРЕРАЗХОД НА СРЕДСТВА,123:45.6
Console.WriteLine (sb.ToString().ToLower());
                                  // преразход на средства,123:45.6
```

Изброим тип enum

Изброим тип – подреден списък от имена, които могат да се използват и отпечатват в програмата.

1. Дефиниране на изброим тип

```
<модификатор> enum <име_на_типа> {идентификатор0, ...,
                                  идентификаторN}
```

идентификатор0 **има стойност** 0**.**

Допуска се

```
идентификатор = стойност
```

2. Въвеждане

```
<промелива_от_изброим_тип> =
(<име_на_типа>) Enum.Parse (typeof(<име_на_типа>), низ);
```

Ако низ **не съвпада с** идентификатори**, се получава изключението** System.ArgumentException**.**

3. Извеждане

```
<промелива_от_изброим_тип >.ToString ()
```

4. Присвояване

```
<промелива_от_изброим_тип> = <име_на_типа>.идентификатори;
```

Пример:

```
using System;
class EnumClass
{
   enum Season {Spring, Summer, Autumn, Winter};
   static void Main(string[] args)
   {
      Season s=(Season)Enum.Parse(typeof(Season),"Winter");
      Console.WriteLine ("Сезонът е {0}.", s);
      Season first = Season.Spring;
      Console.WriteLine ("Първият сезон е {0}.", first);
      Console.WriteLine ("Сезонът {0} е с номер {1}.", first, (int)first);
   }
}
```

Резултати:

```
Сезонът е Winter.
Първият сезон е Spring.
Сезонът Spring е с номер 0.
```

Интерфейси

Интерфейс – характеризира поведението на класовете, независимо от тяхната йерархия.

1. Дефиниране на интерфейс

```
[атрибути] [модификатори] interface <име_на_интерфейс> :
                                    <списък_от_интерфейси>
{
    // Декларация на методи
    // Декларация на свойства
    // Декларация на индексатори
    // Декларация на събития
}
```

2. Реализиране на интерфейс – класът трябва да дефинира всички членове на интерфейса.

Пример: Полиморфизъм с интерфейси

```
using System;
interface ISpeaker
{
   void Speak();
}
class Philosopher : ISpeaker
{  private string philosophy;
   public Philosopher (string thoughts)
   {   philosophy = thoughts;   }
   public void Speak()
   {   Console.WriteLine (philosophy);   }
   public void Pontificate()
   {   for (int i = 1; i <= 3; i++)
         Console.WriteLine (philosophy);
   }
}
class Dog : ISpeaker
{  public void Speak()
   {   Console.WriteLine ("Джаф!");   }
}
```

```
class Talking
{
   static void Main(string[] args)
   {
      ISpeaker current;
      current = new Dog();
      current.Speak();
      current = new Philosopher("Аз мисля, следователно съществувам.");
      current.Speak();
      ((Philosopher)current).Pontificate();
   }
}
```

Резултати:

```
Джаф!
Аз мисля, следователно съществувам.
Аз мисля, следователно съществувам.
Аз мисля, следователно съществувам.
Аз мисля, следователно съществувам.
```

3. Проверка за реализация

а) оператор is – проверява по време на изпълнение дали даден тип е съвместим с друг тип;

израз is тип

израз **– от референтен тип**

Стойността на израза е true, ако:

израз **≠** null **и**

израз **може да се преобразува до** тип

```
Dog dog = new Dog();
if (dog is ISpeaker)
   Console.WriteLine ("Реализира интерфейса ISpeaker");
```

б) оператор as – конвертира съвместими типове;

резултатът се запазва в локална променлива и се проверява дали е от валиден тип.

обект = израз as тип

израз**,** тип **– от референтен тип**

еквивалентно на

израз is тип ? (тип)израз : (тип)null

```
ISpeaker speaker;
speaker = dog as ISpeaker;
if (null != speaker)
   Console.WriteLine("Реализира интерфейса ISpeaker");
else
    Console.WriteLine("Не реализира интерфейса ISpeaker");
```

4. Скриване на имена чрез интерфейс

- **Обичайният начин за извикване на реализиран метод от интерфейс е принудително преобразуване на инстанцията на класа до типа на интерфейса и тогава методът се извиква.**

```
Dog dog = new Dog();
((ISpeaker)dog).Speak();
```

Реализираният метод се извиква чрез принудително преобразуване на типа

- **Извикване на реализирания метод без принудително преобразуване на типа – интерфейсните методи са public.**

```
dog.Speak();
```

Реализираният метод се извиква без принудително преобразуване на типа

Не е подходящо, когато клас реализира няколко интерфейса с много членове.

- **Използва се техниката скриване на имената - не позволява реализираните членове на интерфейса да стават public членове на класа:**
  - **Премахва се public модификаторът за достъп от реализирания член на интерфейса.**
  - **Името на реализирания член се определя с името на интерфейса.**

```
class Dog : ISpeaker
{
   void ISpeaker.Speak()
   {  Console.WriteLine ("Джаф!!");  }
}
…
Dog dog = new Dog();
dog.Speak();
((ISpeaker)dog).Speak();
```

Премахва се public модификаторът и се определя името на члена с името на интерфейса

Грешка – Speak() не е public метод

OK – реализираният метод е извикан чрез принудително преобразуване на типа

- **Избягване на двусмисленост на имената**

Реализирането на много интерфейси, които съдържат членове с еднакви имена, може да предизвика колизия на имената.
IWindow, IFile **и** MyApplication **поддържат операцията затваряне:**

```
public interface IWindow
{  void Close();  }

public interface IFile
{  void Close();  }

public class MyApplication : IWindow, IFile
{  public void Close()
   {  Console.WriteLine("Затваряне на приложението...");  }
}
...
MyApplication application = new MyApplication();
application.Close();
```

Колизия на имена

Кой метод Close е извикан?

```
...
MyApplication application = new MyApplication();
Console.WriteLine("IWindow? {0}", application is IWindow);
Console.WriteLine("IFile? {0}", application is IFile);
application.Close();
```

Резултати:

```
IWindow? True
IFile? True
Затваряне на приложението…
```

Внимание!
Проблем: Класът е реализирал само една версия на Close(), не две.

За да се избегне колизията с имената се използва изрично (експлицитно) определяне на имената на членовете:

- **Премахва се public модификаторът за достъп от реализирания член на интерфейса.**
- **Името на реализирания член се определя с името на интерфейса.**

```
public class MyApplication : IWindow, IFile
{
   void IWindow.Close()
   {  Console.WriteLine("Затваряне на прозорец...");  }

   void IFile.Close()
   {  Console.WriteLine("Затваряне на файл...");  }

   public void Close()
   {  Console.WriteLine("Затваряне на приложение...");  }
}
```

```
...
MyApplication application = new MyApplication();
Console.WriteLine("IFile? {0}", application is IFile);
((IFile)application).Close();
Console.WriteLine("IWindow? {0}", application is IWindow);
((IWindow)application).Close();
application.Close();
```

Резултати:

```
IFile? True
Затваряне на файл…
IWindow? True
Затваряне на прозорец…
Затваряне на приложение…
```

5. Проблеми с наследяване и интерфейси

- **Клас наследява базов клас и интерфейс, които имат идентични имена на методи**

```
public class Data
{   public void Close()
    {   Console.WriteLine("Затваряне на данни...");   }
}
public interface IWindow
{   void Close();   }
public class MyApplication : Data, IWindow { }
. . .
MyApplication application = new MyApplication();
application.Close();
```

Резултати:

```
Затваряне на данни…
```

Извикан е методът Data.Close

```
. . .
MyApplication application = new MyApplication();
IWindow window = application as IWindow;
if (null != window)
   window.Close();
```

Резултати:

```
Затваряне на данни…
```

Извикан е наследеният метод Data.Close

- **Производен клас има метод с идентично име на метод в базов клас, реализиращ метод на интерфейс**

```
public interface IWindow
{   void Close();   }
public class Data : IWindow
{
   public void Close()
   {   Console.WriteLine("Затваряне на данни...");   }
}
public class MyApplication : Data
{
   public new void Close()
   {   Console.WriteLine("Затваряне на приложение...");   }
}
```

```
...
MyApplication application = new MyApplication();
((IWindow)application).Close();
application.Close();
```

Резултати:

```
Затваряне на данни…
Затваряне на приложение…
```

Извиква IWindow.Close – реализация на интерфейса

Извиква MyApplication.Close – предефиниран наследен метод Data.Close (чрез new)

6. Комбиниране на интерфейси

Няколко интерфейса могат да се комбинират – класът трябва да реализира само комбинирания резултат.

```
public class Data { }
public interface IWindow
{  void Close(); }
public interface IRead
{  void Read(); }
public interface ICombine : IWindow, IRead
{ }
public class Component : Data, ICombine
{  public void Close()
   {   Console.WriteLine("Затваряне на компонента...");   }
   public void Read()
   {   Console.WriteLine("Четене на данни...");   }
}
```

```
. . .
MyApplication application = new MyApplication();
application.Read();
application.Close();
```

Резултати:

```
Четене на данни…
Затваряне на компонента…
```

Шаблони (Generics) в .NET Framework

Шаблоните са класове, структури, интерфейси и методи, които имат типизирани параметри за един или повече от типовете, които те съхраняват или използват.

```
public class Generic<T>
{
   public T Field;
}

Generic<string> s = new Generic<string>();
s.Field = "Низ";

Generic<int> i = new Generic<int>();
i.Field = 123;
```

Предимства

- **прехвърля отговорността за сигурността на типа от потребителя към компилатора;**
- **отпада необходимостта от писане на код за проверка коректността на типа на данните, защото се прилага по време на компилация;**
- **отпада необходимостта от принудително преобразуване на типа и се намалява възможността за възникване на грешки по време на изпълнение.**

Интерфейси и колекции в .NET Framework

System.Collections.Generic	System.Collections
IComparer<T>	IComparer
IComparable<T>	System.IComparable
Dictionary<K,T>	HashTable
LinkedList<T>	-
List<T>	ArrayList
Queue<T>	Queue
SortedDictionary<K,T>	SortedList
Stack<T>	Stack
ICollection<T>	ICollection
IDictionary<K,T>	IDictionary
IEnumerable<T>	IEnumerable
IEnumerator<T>	IEnumerator
IList<T>	IList

IEnumerator

```
// Извършва проста итерация на колекцията.
public interface IEnumerator
{
   // Връща текущия елемент в колекцията.
   object Current { get; }

   // Премества номератора към следващия елемент на колекцията.
   bool MoveNext ();

   // Установява номератора в начална позиция, която е
   // преди първия елемент в колекцията.
   void Reset ();
}
```

IEnumerator<T>

```
// Извършва проста итерация на типизирана колекция.
public interface IEnumerator<T> : IDisposable, IEnumerator
{
   // Връща текущия елемент в типизираната колекция.
    T Current { get; }
}
```

IEnumerable

```
// Представя номератор, който позволява проста итерация на колекцията.
public interface IEnumerable
{
   // Връща номератор, който може да итерира колекцията.
   IEnumerator GetEnumerator ();
}
```

IEnumerable<T>

```
// Представя номератор, който позволява проста итерация на колекция от
// определен тип.
public interface IEnumerable<T> : IEnumerable
{
   // Връща номератор, който може да итерира колекцията.
   IEnumerator<T>   GetEnumerator ()
}
```

IDictionaryEnumerator

```
// Номерира елементите на речник.
public interface IDictionaryEnumerator
{
   // Връща ключа и стойността на текущия елемент на речник.
   DictionaryEntry Entry { get; }

   // Връща ключа на текущия елемент на речник.
   object Key { get; }

   // Връща стойността на текущия елемент на речник.
   object Value { get; }
}
```

ICollection

```
// Дефинира методи за всички колекции.
public interface ICollection : IEnumerable
{
  // Връща броя на елементите в ICollection.
  int Count {get; }
  // Връща стойност, показваща дали достъпът до ICollection е
  // синхронизиран.
  bool IsSynchronized { get; }
  // Връща обект, който се използва за синхронизиране на
  // достъпа до ICollection.
  object SyncRoot { get; }
  // Копира елементите от ICollection в array,
  // стартирайки от определен индекс от array.
  void CopyTo (Array array, int index);
}
```

ICollection<T>

```
// Дефинира методи за всички колекции.
public interface ICollection<T> : IEnumerable<T>, IEnumerable
{
  // Връща броя на елементите в ICollection.
  int Count { get; }
  // Връща стойност, показваща дали достъпът до ICollection е само за
  // четене.
  bool IsReadOnly { get; }
}
```

IList

```
// Представя колекция от обекти, до които достъпът може
// да се осъществи чрез индекс.
public interface IList
{
  // Добавя елемент към IList.
  void Add (object value);

  // Определя дали IList съдържа определена стойност.
  bool Contains (object value);

  // Премахва първото срещане на определен обект от IList.
  void Remove (object value);
 …
}
```

IDictionary

```
// Представя колекция от двойки ключ-стойност.
public interface IDictionary
{
  // Връща IDictionaryEnumerator за IDictionary.
  IDictionaryEnumerator GetEnumerator ();
  // Добавя елемент с даден ключ и дадена стойност към IDictionary.
  void Add (object key, object value);
  // Определя дали IDictionary съдържа елемент с определения ключ.
  bool Contains (object key);
  // Премахва елемента с определения ключ от IDictionary.
  void Remove (object key);
…
}
```

IDictionary<TKey,TValue>

```
// Представя типизирана колекция от двойки ключ-стойност.
public interface IDictionary<TKey,TValue> :
  ICollection<KeyValuePair<TKey,TValue>>,
  IEnumerable<KeyValuePair<TKey,TValue>>, IEnumerable
{
  // Връща/установява стойността на елемента с определения ключ.
  TValue this [TKey key] { get; set; }
  // Връща ICollection, съдържаща ключовете на IDictionary.
  ICollection<TKey> Keys { get; }
  // Връща ICollection, съдържаща стойностите в IDictionary.
  ICollection<TValue> Values { get; }
  // Добавя елемент с дадения ключ и стойност към IDictionary.
  void Add (TKey key, TValue value);
  // Определя дали IDictionary съдържа елемент с дадения ключ.
  bool ContainsKey (TKey key);
  // Премахва елемента с дадения ключ от IDictionary.
  bool Remove (TKey key);
  // Връща стойността, съответстваща на дадения ключ.
  bool TryGetValue (TKey key, out TValue value);
}
```

Типизирана структура KeyValuePair<TKey,TValue> – **съдържа двойка типизиран ключ-типизирана стойност.**

```
struct KeyValuePair<TKey,TValue>
{
  public KeyValuePair(TKey key, TValue value);
  public TKey Key{ get; }
  public TValue Value{ get; }
}
```

IComparer

```
// Представя метод, който сравнява два обекта.

public interface IComparer
{
  // Сравнява два обекта и връща стойност, показваща дали единият
  // е по-малък, равен или по-голям от другия.
  int Compare (object x, object y);
}
```

IComparer<T>

```
// Представя метод, който сравнява два обекта.
public interface IComparer<T>
{
  // Сравнява два обекта и връща стойност, показваща дали единият
  // е по-малък, равен или по-голям от другия.
  int Compare (T x, T y);
}
```

IComparable

```
// Дефинира общ метод за сравняване.
// Така стойностните типове и класовете, реализиращи този общ
// метод, създават метод за сравняване на специфични типове.
public interface IComparable
{
  // Сравнява текущия екземпляр с друг обект от същия тип.
  int CompareTo (object obj);
}
```

IComparable<T>

```
// Дефинира общ метод за сравняване.
// Така стойностните типове и класовете, реализиращи този общ
// метод, създават метод за сравняване на специфични типове.
public interface IComparable<T>
{
  // Сравнява текущия екземпляр с друг обект от същия тип.
  int CompareTo (T other);
}
```

Пример: Колекция ArrayList – реализира IList; едномерен масив с елементи от тип Object, чийто размер се увеличава автоматично; притежава методи за добавяне, вмъкване и премахване на елемент; лявата граница винаги е 0.

```
using System.Collections;

ArrayList list = new ArrayList();
list.Add ("Петър");
list.Add ("Ана");

IEnumerator myEnumerator = list.GetEnumerator();
while (myEnumerator.MoveNext())
  Console.WriteLine (myEnumerator.Current);

foreach (string element in list)
  Console.WriteLine(element);
```

Пример: Колекция SortedList – реализира IDictionary, ICollection, IEnumerable; представя колекция от двойки ключ-стойност; елементите на SortedList са сортирани спрямо техните ключове в съответствие със специфичната реализация на IComparer или IComparable; достъпни са чрез ключ или индекс.

```
using System.Collections;

SortedList bookList = new SortedList();
bookList.Add (123456, "В името на розата");
bookList.Add (234567, "Доктор Живаго");

IDictionaryEnumerator myEnumerator = bookList.GetEnumerator();

while (myEnumerator.MoveNext())
  Console.WriteLine (myEnumerator.Key + "\t" + myEnumerator.Value);

foreach (DictionaryEntry element in bookList)
  Console.WriteLine(element.Key + "\t" + element.Value);
```

Пример: Колекцията SortedDictionary<TKey,TValue> реализира IDictionary<TKey,TValue>, ICollection<KeyValuePair<TKey,TValue>>, IEnumerable<KeyValuePair<TKey,TValue>>, IDictionary, ICollection и IEnumerable; представя колекция от двойки ключ-стойност (KeyValuePair<TKey,TValue>), които са сортирани спрямо ключовете им и са достъпни чрез ключ и чрез индекс.

```
using System.Collections.Generic;

SortedDictionary<string, int> phoneBook = new SortedDictionary<string, int>();
phoneBook.Add("Ann", 8871234);
phoneBook.Add("Peter", 8528765);

IEnumerator<KeyValuePair<string, int>> myEnumerator =
                                          phoneBook.GetEnumerator();
while (myEnumerator.MoveNext())
  Console.WriteLine(myEnumerator.Current.Key + "\t" +
                    myEnumerator.Current.Value);

foreach (KeyValuePair<string, int> element in phoneBook)
  Console.WriteLine(element.Key + "\t" + element.Value);
```

Пример: Използване на интерфейса IComparable с методи за сортиране и търсене в масиви и колекции. Необходимо е предефиниране на метода CompareTo.

```
public int CompareTo (object o);
public static void Sort (Array array, int startindex, int length);
```

```
using System;
class Rational : IComparable
{  private int numerator, denominator;
   public Rational (int numer, int denom)
   {  numerator = numer;
      denominator = denom;
      Reduce ();
   }
   private void Reduce()
   {  int common = Gcd (Math.Abs(numerator), denominator);
      numerator /= common;
      denominator /= common;
   }
   private int Gcd (int n1, int n2)              // НОД
   {  while (n1 != n2)
         if (n1 > n2)
            n1 -= n2;
         else
            n2 -= n1;
      return n1;
   }
```

```
   public override string ToString()
   {
      return numerator + "/" + denominator;
   }
   public int CompareTo (object o)
   {
      Rational op2 = (Rational)o;
      int difference = numerator*op2.denominator-op2.numerator*denominator;
      if (difference < 0)
         return -1;
      else if (difference>0)
         return 1;
      else
         return 0;
      }
```

```
      // Предефинира принудително преобразуване от Rational във float
      public static explicit operator float(Rational op)
      {
         return (float)op.numerator / op.denominator;
      }

      // Предефинира подразбиращо се преобразуване от Rational в double
      public static implicit operator double(Rational op)
      {
         return (double)op.numerator / op.denominator;
      }
}
```

```
class TestRational
{
   static void Main()
   {
      Rational x, y;
      x = new Rational (1, 4);
      y = new Rational (1, 3);
      int flag = x.CompareTo(y);
      if (flag < 0) Console.WriteLine ("x<y");
      else if (flag>0) Console.WriteLine ("x>y");
      else Console.WriteLine ("x=y");
      Rational[] list = new Rational[3] {new Rational (1, 3),
                        new Rational (1, 4), new Rational (2, 5)};
      Array.Sort (list, 0, 3);
      foreach(Rational r in list)
         Console.WriteLine (r);
      float f = (float)z; Console.WriteLine(f);
      double r = z; Console.WriteLine(r);
   }
}
```

Управление на паметта

1. **Управление на времето на живот на обектите от CLR, използвайки механизма за събиране на боклука (garbage collection)**
2. **Достъп до паметта директно чрез указатели**

Събиране на боклука

new **отделя памет в хийпа.**

Системата за събиране на боклука (garbage collector – GC) автоматично изчиства паметта на хийпа, когато обектът не е нужен вече, като извиква автоматично метода Object.Finalize **в отделна нишка за изпълнение.**

```
using System;
public class SuperClass
{
   private string name;
   public SuperClass(string name) {   this.name = name;   }
   public override string ToString() {   return name;   }
}
class Program
{
   static void Process()
   {
      SuperClass s = new SuperClass("SuperClass");
      Console.WriteLine(s);
   }
   static void Main(string[] args)
   {
      Process();
      Console.WriteLine("Край на Main");
   }
}
```

new отделя памет в хийпа

s вече е извън обсега

GC освобождава паметта на обекта, защото s вече не съществува – извиква автоматично Object.Finalize

Резултати:
SupperClass
Край на Main

Предефиниране на Finalize

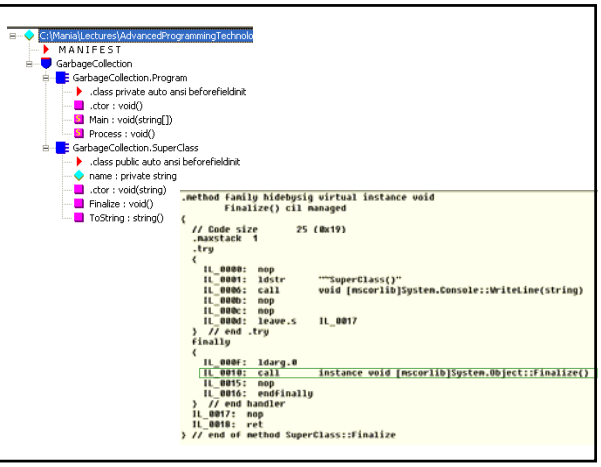
- **Синтактически методът е идентичен на деструктор**

```
using System;
public class SuperClass
{
  …
  // Предефиниране на Finalize
  ~SuperClass() {   Console.WriteLine("~SuperClass()");   }
}
```

Резултати:

```
SupperClass
Край на Main
~SuperClass()
```

Finalize се извиква като последна операция



Предефиниране на Finalize в производни класове

- **обектите се конструират „отвътре навън“ – първо се извиква конструкторът на базовия клас;**
- **обектите се разрушават „отвън навътре“ – деструкторът на производния клас завършва своята задача преди извикване на деструктора на базовия клас.**

```
using System;

public class SuperClass
{
  private string name;
  public SuperClass(string name)
  {
    this.name = name;
    Console.WriteLine("SuperClass()");
  }
  public override string ToString() {   return name;   }
  ~SuperClass() {   Console.WriteLine("~SuperClass()");   }
}

public class DerivedClass : SuperClass
{
  public DerivedClass(string name) : base(name)
  {   Console.WriteLine("DerivedClass()");   }
  ~DerivedClass() {   Console.WriteLine("~DerivedClass()");
  }
}
```

```
class Program
{
  static void Process()
  {
    DerivedClass d = new DerivedClass("DerivedClass");
    Console.WriteLine(d);
  }
  static void Main(string[] args)
  {
    Process();
    Console.WriteLine("Край на Main");
  }
}
```

Резултати:

```
SuperClass()
DerivedClass()
DerivedClass
Край на Main
~DerivedClass()
~SuperClass()
```

Принудително събиране на боклука

- **методът GC.Collect() форсира събирането на боклука;**
- **методът GC.WaitForPendingFinalizers() преустановява текущата нишка, докато опашката от финализатори, чакащи да се изпълнят от тази нишка, се изпразни.**

```
class Program
{   static void Process()
    {   SuperClass s = new SuperClass("SuperClass");
        Console.WriteLine(s);
    }
    static void Main(string[] args)
    {   Process();
        Console.WriteLine("Прави нещо");
        GC.Collect();
        GC.WaitForPendingFinalizers();
        Console.WriteLine("Край на Main");
    }
}
```

форсира събирането с GC.Collect

Резултати:

```
SuperClass()
SuperClass
Прави нещо
~SuperClass()
Край на Main
```

```
class Program
{
  static void Process()
  {
    SuperClass s = new SuperClass("SuperClass");
    Console.WriteLine(s);
    s = null;
    GC.Collect();
    GC.WaitForPendingFinalizers();
  }
  static void Main(string[] args)
  {
    Process();
    Console.WriteLine("Прави нещо");
    //GC.Collect();
    //GC.WaitForPendingFinalizers();
    Console.WriteLine("Край на Main");
  }
}
```

Установява референцията на null преди да форсира събирането с GC.Collect

Резултати:

```
SuperClass()
SuperClass
~SuperClass()
Прави нещо
Край на Main
```

Шаблон Dispose

- **Методът Dispose() изчиства ресурсите, дори ако референции към обекта са живи – изрично управление на деструктора за изчистване**

```
public void Dispose()
```

  - **Не позволява на GC да извика метода Finalize чрез метода GC.SuppressFinalize()**
- **Програмистът трябва изрично да извика метода Dispose**
- **Ако програмистът не извика Dispose, ще бъде извикан финализаторът – следователно финализаторът трябва да извика Dispose**

```
public class SuperClass
{
  private string name;
  public SuperClass(string name)
  {
    this.name = name;
    Console.WriteLine("SuperClass()");
  }
  public override string ToString() {   return name;   }
  ~SuperClass()
  {
    Dispose();
    Console.WriteLine("~SuperClass()");
  }
  public void Dispose()
  {
    // Тук се пише кодът за изчистване …
    Console.WriteLine("Dispose()");
    GC.SuppressFinalize(this);
  }
}
```

```
class Program
{
  static void Process()
  {
    SuperClass s = new SuperClass("SuperClass");
    Console.WriteLine(s);
    s.Dispose();
    s = null;
    GC.Collect();
    GC.WaitForPendingFinalizers();
  }
  static void Main(string[] args)
  {
    Process();
    Console.WriteLine("Край на Main");
  }
}
```

Резултати:

```
SuperClass()
SuperClass
Dispose()
Край на Main
```

Интерфейс IDisposable

```
public interface IDisposable
{
  void Dispose();
}
```

  - **Не позволява на GC да извика метода Finalize чрез метода GC.SuppressFinalize()**
- **Програмистът трябва изрично да извика Dispose**
- **Ако програмистът не извика Dispose, финализаторът ще бъде извикан – следователно финализаторът трябва да извика Dispose**

```
public class SuperClass : IDisposable
{  …  }
```