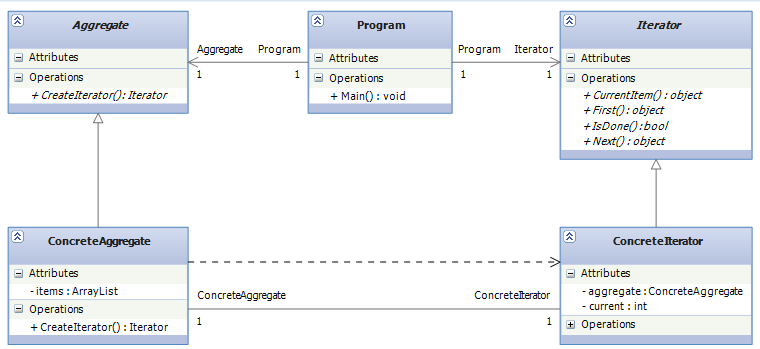
Пользовательские коллекции

Коллекция–этокласс,предназначенныйдлягруппировки связанныхобъектов,управленияимииобработкиихвциклах.

Коллекцииявляютсяважныминструментомпрограммиста,но решениеоихпримененииневсегдаоказываетсяочевидным.

Коллекции стоит применять, если:

* + •Отдельные элементы используются для одинаковых целей и одинаково важны.
  + •На момент компиляции число элементов не известно или не зафиксировано.
  + •Необходима поддержка операции перебора всех элементов.
  + •Необходима поддержка упорядочивания элементов.
  + •Необходимо использовать элементы из библиотеки, от которой потребитель ожидает наличия типа коллекции.



abstract class Aggregate

{

public abstract Iterator CreateIterator();

public abstract object this[int index] { get; set; }

}

class ConcreteAggregate : Aggregate

{

private readonly ArrayList elements = new ArrayList();

private ConcreteIterator iterator;

public override Iterator CreateIterator()

{

iterator = new ConcreteIterator(this);

return iterator;

}

public int Count

{

get { return elements.Count; }

}

public override object this[int index]

{

get { return elements[index]; }

set { elements.Insert(index,value); }

}

}

abstract class Iterator

{

public abstract object First();

public abstract object Next();

public abstract bool IsDone();

public abstract object CurrentItem();

}

class ConcreteIterator : Iterator

{

private readonly ConcreteAggregate aggregate;

private int current;

public ConcreteIterator(ConcreteAggregate aggregate)

{

this.aggregate = aggregate;

}

public override object First()

{

return aggregate[0];

}

public override object Next()

{

object element = null;

if (current < aggregate.Count - 1)

element = aggregate[++current];

return element;

}

public override object CurrentItem()

{

return aggregate[current];

}

public override bool IsDone()

{

return current >= aggregate.Count-1;

}

}

Системные колекции

ArrayList

* IEnumerator делает версию колекции, если удалить елемент в foreach версия изменится и будет исключение

// Вставка элементов в заданное положение используя метод InsertRange.

var moreString = new[] { "goodnight", "see ya" };

list.InsertRange(4, moreString);

// Вставка элементов в заданное положение используя индексатор.

// (!) При использовании индексатора элемент не вставляется в набор, а перезаписывается прежний объект, бывший в этом элементе.

list[3] = "Hey All 2";

// Удаление из набора одиночных элементов используя метод Remove.

list.Add("Hello");

list.Remove("Hello");

// Удаление из набора одиночных элементов с заданным индексом используя метод RemoveAt.

list.RemoveAt(0);

// Удаление из набора, группы элементов с заданным диапазоном используя метод RemoveRange.

list.RemoveRange(0, 4);

// Другие методы для добавления и удаления элементов набора - Contains, IndexOf, Clear.

string myString = "My String";

if (list.Contains(myString))

{

int index = list.IndexOf(myString);

list.RemoveAt(index);

}

else

{

list.Clear();

}

}

Итерация по колекции

for (int i = 0; i < list.Count; i++)

{

Console.WriteLine(list[i]);

}

// 2.

IEnumerator enumerator = list.GetEnumerator();

while (enumerator.MoveNext())

{

Console.WriteLine(enumerator.Current);

}

// 3.

foreach (string item in list)

{

//// Never do this!!!

list.Remove(item);

Console.WriteLine(item);

}

IComparer

class Program

{

static void Main()

{

var list = new ArrayList {2, 3, 1};

list.Sort(new DescendingComparer());

foreach (int item in list)

Console.WriteLine(item);

// Задержка.

Console.ReadKey();

}

}

public class DescendingComparer : IComparer

{

// Проверяет равенство двух объектов без учета регистра строк.

readonly CaseInsensitiveComparer comparer = new CaseInsensitiveComparer();

public int Compare(object x, object y)

{

// Для сортировки по убыванию.

// Объекты, переданные для сравнения, меняются местами.

int result = -1 \* comparer.Compare(x, y);

//int result = comparer.Compare(y, x);

return result;

}

}

Queue

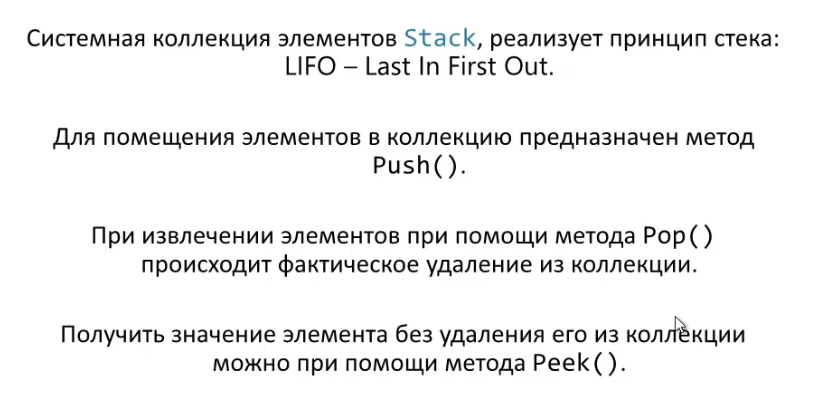
Peek – не удаляет елемент из очереди

var queue = new Queue();

queue.Enqueue("An item"); // Enqueue() - добавляет элемент в конец очереди.

Console.WriteLine(queue.Dequeue()); // Dequeue() - возвращает первый элемент очереди, удаляя его.

Stack



var stack = new Stack();

stack.Push("First");

stack.Push("Second");

stack.Push("Third");

stack.Push("Fourth");

// Peek() - возвращает элемент с вершины стека, не удаляя его.

if(stack.Peek() is string)

{

Console.WriteLine(stack.Pop());

}

// Count - возвращает количество элементов в стеке.

while (stack.Count > 0)

{

Console.WriteLine(stack.Pop());

}

Hashtable



var emailLookup = new Hashtable();

// Метод Add принимает в качестве первого параметра ключ,

// а в качестве второго - значение

emailLookup.Add("sbishop@contoso.com", "Bishop, Scott");

// emailLookup.Add("sbishop@contoso.com", "Bishop, Scott2");

// Особенности использования индексаторов.

// Использование индексатора эквивалентно вызову Add,

// если такого индекса не существует.

emailLookup["s.bishop@contoso.com"] = "Bishop, Scott";

// Если такой индекс существует, то происходит только замена значения.

emailLookup["sbishop@contoso.com"] = "-------------";

Console.WriteLine(emailLookup["sbishop@contoso.com"]);

Console.WriteLine(emailLookup["s.bishop@contoso.com"]);

Итерация по Hashtable

Console.WriteLine(new string('-', 20));

foreach (DictionaryEntry name in emailLookup)

{

Console.WriteLine(name.Value);

}

Console.WriteLine(new string('-', 20));

foreach (object name in emailLookup.Values)

{

Console.WriteLine(name);

}

public class Fish

{

private readonly string name;

public Fish(string name)

{

this.name = name;

}

public override int GetHashCode()

{

return name.GetHashCode();

}

public override bool Equals(object obj)

{

var otherFish = obj as Fish;

if (otherFish == null)

{

return false;

}

return otherFish.name == name;

}

}

class Program

{

static void Main()

{

var duplicates = new Hashtable();

var key1 = new Fish("Herring");

var key2 = new Fish("Herring");

duplicates[key1] = "Hello";

duplicates[key2] = "Hello2";

Console.WriteLine(duplicates.Count);

// Delay.

Console.ReadKey();

}

}

SortedList

var sort = new SortedList();

sort["First"] = "1st";

sort["Second"] = "2nd";

sort["Third"] = "3rd";

sort["Fourth"] = "4th";

sort["fourth"] = "4th";

foreach (DictionaryEntry entry in sort)

{

Console.WriteLine("{0} = {1}", entry.Key, entry.Value);

}

\\*\*

var sort = new SortedList(new DescendingComparer());

sort["First"] = "1st";

sort["Second"] = "2nd";

sort["Third"] = "3rd";

sort["Fourth"] = "4th";

sort["fourth"] = "44th";

foreach (DictionaryEntry entry in sort)

{

Console.WriteLine("{0} = {1}", entry.Key, entry.Value);

}

// Delay.

Console.ReadKey();

}

}

public class DescendingComparer : IComparer

{

CaseInsensitiveComparer comparer = new CaseInsensitiveComparer();

public int Compare(object x, object y)

{

// Для сортировки по убыванию.

// Объекты, переданные для сравнения, меняются местами.

int result = comparer.Compare(y, x);

return result;

}

}

HybridDictionari

// HybridDictionary - Рекомендуется к использованию в тех случаях, когда невозможно определить размер коллекции заранее.

var emailLookup = new HybridDictionary();

emailLookup["sbishop@contoso.com"] = "Bishop, Scott";

emailLookup["chess@contoso.com"] = "Hell, Christian";

emailLookup["djump@contoso.com"] = "Jump, Dan";

foreach (DictionaryEntry entry in emailLookup)

{

Console.WriteLine(entry.Value);

}

OrderedDictionari

/ OrderedDictionary - Размещение элементов соответствует порядку их добавления в коллекцию,

// что позволяет автоматически сохранять элементы в хронологическом порядке.

var emailLookup = new OrderedDictionary

{

{"sbishop@contoso.com", "Bishop, Scott"},

{"chess@contoso.com", "Hell, Christian"},

{"djump@contoso.com", "Jump, Dan"}

};

foreach (DictionaryEntry entry in emailLookup)

{

Console.WriteLine(entry.Value);

}

BitArray

static void Main()

{

BitArray bits = new BitArray(3);

bits[0] = false;

bits[1] = true;

bits[2] = false;

Console.WriteLine(bits.Length);

// Для изменения размера необходимо воспользоваться свойством Length.

bits.Length = 4;

bits[3] = true;

Console.WriteLine(bits.Length);

// Delay.

Console.ReadKey();

}

/\*\*\*\*\*\*\*\*\*\*\*

static void Main()

{

var bits = new BitArray(3);

bits[0] = false;

bits[1] = true;

bits[2] = false;

var moreBits = new BitArray(3);

moreBits[0] = true;

moreBits[1] = true;

moreBits[2] = false;

var xorBits = bits.Xor(moreBits);

foreach (bool bit in xorBits)

{

Console.WriteLine(bit);

}

// Delay.

Console.ReadKey();

}

BitVector32

static void Main()

{

var vector = new BitVector32(0);

int firstBit = BitVector32.CreateMask(); // ... 0000 0001 B - 1D

int secondBit = BitVector32.CreateMask(firstBit); // ... 0000 0010 B - 2D

int thirdBit = BitVector32.CreateMask(secondBit); // ... 0000 0100 B - 4D

Console.WriteLine(firstBit + " " + secondBit + " " + thirdBit); // Test

vector[firstBit] = true;

vector[secondBit] = true;

Console.WriteLine("{0} должно быть 3", vector.Data);

Console.WriteLine(vector.ToString());

var newVector = new BitVector32(4);

Console.WriteLine(newVector);

bool bit1 = newVector[firstBit];

bool bit2 = newVector[secondBit];

bool bit3 = newVector[thirdBit];

Console.WriteLine("bit1 = {0}, bit2 = {1}, bit3 = {2}", bit1, bit2, bit3);

// Delay.

Console.ReadKey();

}

}

Несколько чисел в одной памяти int

static void Main()

{

BitVector32.Section firstSection = BitVector32.CreateSection(10); // 0xA Hex - 1010 Bin

BitVector32.Section secondSection = BitVector32.CreateSection(50, firstSection); // 0x32 Hex - 110010 Bin

BitVector32.Section thirdSection = BitVector32.CreateSection(500, secondSection); // 0x1F4 Hex - 111110100 Bin

BitVector32.Section fourthSection = BitVector32.CreateSection(500, thirdSection);

var packedBits = new BitVector32(0);

packedBits[firstSection] = 10;

packedBits[secondSection] = 50;

packedBits[thirdSection] = 500;

packedBits[fourthSection] = 499;

Console.WriteLine(packedBits[firstSection]);

Console.WriteLine(packedBits[secondSection]);

Console.WriteLine(packedBits[thirdSection]);

Console.WriteLine(packedBits[fourthSection]);

Console.WriteLine(packedBits); // packedBits = {BitVector32{0000000000000 111110100 110010 1010}}

Console.WriteLine(packedBits.Data);

// packedBits.Data = 512 810

// Delay.

Console.ReadKey();

}

NameValueCollection

static void Main()

{

var nv = new NameValueCollection

{

{"Key", "Some Text"},

{"Key", "More Text"}

};

foreach (string s in nv.GetValues("Key"))

Console.WriteLine(s);

Console.WriteLine(new string('-', 20));

// Использование индексатора отличается от использования метода Add()

// Индексатор не добавляет новое значение а заменяет существующее.

nv["First"] = "1st";

nv["First"] = "FIRST";

Console.WriteLine(nv.GetValues("First").Length); // 1

foreach (string s in nv.GetValues("First"))

Console.WriteLine(s);

Console.WriteLine(new string('-', 20));

// Метод Add добавляет новое значение в коллекцию.

nv.Add("Second", "2nd");

nv.Add("Second", "SECOND");

Console.WriteLine(nv.GetValues("Second").Length); // 2

foreach (string s in nv.GetValues("Second"))

Console.WriteLine(s);

// Practical usage example with App.Config

Console.WriteLine(new string('-', 40));

Console.WriteLine("App.Config appSettings values");

var appSettings = ConfigurationManager.AppSettings;

foreach (string appSetting in appSettings)

{

Console.WriteLine("{0}: {1}", appSetting, appSettings[appSetting]);

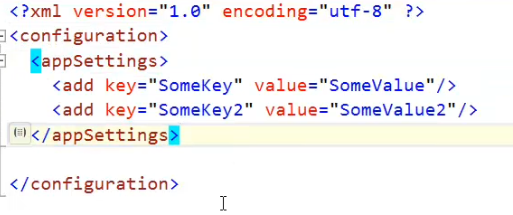
}

Console.WriteLine( appSettings["SomeKey"]);

// Delay.

Console.ReadKey();

}



Dictionari generic

var dict = new Dictionary<int, string>();

dict[3] = "Three";

dict[4] = "Four";

dict[1] = "One";

dict[2] = "Two";

dict[2] = "Tw1o";

string str = dict[3];

foreach (KeyValuePair<int, string> i in dict)

{

Console.WriteLine("{0} = {1}", i.Key, i.Value);

}

Console.WriteLine(new string('-', 25));

foreach (var value in dict.Values)

{

Console.WriteLine(value);

}

* Наборы — классы, предназначенные для группировки связанных объектов, управления ими и обработки их в циклах, — являются одним из основных инструментов программиста.
*  В .NET Framework в пространстве имен System.Collections реализовано большое количество специализированных наборов, которые значительно упрощают и ускоряют процесс разработки.
*  ArrayList - простой, поддерживающий индексирование и изменение размера набор объектов. Является простым неупорядоченным контейнером для объектов любого типа. В наборах разрешается хранить значимые типы, но сначала их необходимо преобразовать к ссылочному типу, выполнив операцию упаковки (boxing). Метод AddRange позволяет добавлять диапазоны (или группу элементов) любых объектов, поддерживающих интерфейс ICollection (включая массивы, объекты ArrayList и большинство других наборов). Методы Add и AddRange добавляют элементы в конец набора. Поскольку наборы ArrayList являются динамическими, они поддерживают вставку объектов в заданном положении. Для этого ArrayList предоставляет методы Insert и InsertRange. Интерфейс IEnumerable унифицирует перебор элементов набора в цикле. .NET Framework также поддерживает общий интерфейс — образец API наборов. Этот интерфейс называется ICollection и происходит от интерфейса IEnumerable. Это означает, что любой набор, поддерживающий интерфейс ICollection, также поддерживает интерфейс IEnumerable. Интерфейс
* ICollection содержит в себе 3 свойства(Count, IsSynchronized, SyncRoot) и один метод (CopyTo).
*  .NET Framework поддерживает еще один интерфейс для доступа к элементам списка: IList. Этот интерфейс наследуется непосредственно от ICollection. Если класс поддерживает интерфейс IList, он также поддерживает интерфейсы ICollection и IEnumerable.
*  Последовательные списки рекомендуется использовать в том случае, когда необходимо обрабатывать списки объектов последовательно, а не в произвольном порядке. Последовательные списки представлены двумя классами (Queue и Stack).
*  Queue - набор объектов, организованный по принципу «первым вошел, первым вышел» (FIFO; first-in, first-out). В отличии от ArrayList, в котором обращение к элементу и удаление его из набора представлены различными операциями, Queue объединяет эти операции в комбинированном методе Dequeue. Эти операции исходно логически связаны в силу специфики класса Queue. Создав экземпляр класса, можно вызывать метод Enqueue для добавления элементов в очередь и метод Dequeue для удаления элементов из нее. Класс Queue позволяет добавлять в список дублирующиеся элементы и null-значения, поэтому стандартные методы Dequeue и Реек не помогут определить, пуста ли очередь Queue.
*  Stack - набор объектов, организованный по принципу «последним вошел, первым вышел» (LIFO; last-in, first-out). Работу данного класса можно сравнить с колодой карт из которой карты можно убрать и ложить обратно только сверху. Работа с классом Stack похожа на работу с классом Queue: создают экземпляр класса Stack, затем для добавления элементов в стек вызывают метод Push, а для удаления — метод Pop. Оба класса Stack и Queue поддерживают метод Реек для просмотра следующего элемента в наборе без его удаления.
*  Hashtable - набор пар объектов «имя-значение», предоставляющих доступ к элементам как по имени, так и по индексу в наборе. Чтобы получить доступ к данным, уже добавленным в словарь, достаточно вызвать индексатор с требуемым ключом. Если из словаря требуется вывести значения, следует указать итератору, что он работает с объектами типа DictionaryEntry. Класс Hashtable требует уникальности хэш-кодов, а не связанных с ними значений. Если же попытаться сохранить разные значения с одним и тем же ключом, то второе значение заменит первое. Класс Hashtable чувствителен к регистру. Существенным недостатком данного класса является высокое потребление ресурсов, при использовании маленьких наборов (меньше десяти элементов) это снижает производительность.
*  Интерфейс IEqualityComparer предоставляет возможность реализовать настраиваемое сравнение коллекций в отношении равенства. Это означает, что можно создать свое собственное определение равенства и указать, что это определение должно использоваться для типа коллекции, которая принимает интерфейс IEqualityComparer. В .NET Framework конструкторы типов коллекций Hashtable, NameValueCollection и OrderedDictionary принимают этот интерфейс. Данный интерфейс поддерживает только сравнения в отношении равенства. Настройка сравнения для сортировки или упорядочения реализуется с помощью интерфейса IComparer.
*  SortedList - упорядоченный набор пар объектов «имя-значение». Чтобы упорядочить коллекцию, представленную SortedList вызов метода Sort не требуется. SortedList сам упорядочит элементы в момент добавления
*  ListDictionary - набор, подходящий для хранения небольших списков объектов. Он оптимален для небольших наборов (до 10-ти элементов), поскольку устроен как простой массив.
*  HybridDictionary - набор, в котором элементы хранятся в ListDictionary, если их мало, либо в Hashtable, если их много. Используется в тех случаях, когда определить размер коллекции изначально невозможно.
*  BitArray - компактный набор значений типа Boolean. Класс BitArray поддерживает нединамическое изменение размера. Размер набора необходимо указывать при создании экземпляра класса BitArray. После этого для изменения размера можно воспользоваться свойством Length. Элементы набора BitArray принимают одно из двух значений: true или false, поэтому сама концепция добавления или удаления элементов к такому набору неприменима.
*  Структура BitVector32 очень удобна для управления отдельными битами больших чисел. BitVector32 хранит свои данные как 32-разрядное целое число. Все операции над объектами BitVector32 в действительности изменяют значение целого внутри структуры.

|  |
| --- |
| \ |

* Структура BitVector32 позволяет последовательно создавать битовые маски, вызывая статический метод CreateMask. Эти маски можно использовать вместе с индексатором структуры BitVector32 для установки или получения значения отдельного бита, соответствующего маске. Также коллекция BitVector32 очень удобна для работы с отдельными битами и поддерживает упаковку битов. Упаковка часто позволяет сэкономить память, необходимую для хранения маленьких чисел.
*  StringCollection - простой, поддерживающий изменение размера, набор строк. Код, добавляющий строки в набор, выглядит так же, как и в приведенном выше примере с использованием ArrayList. Единственное отличие в том, что попытка добавления объекта с типом, отличным от строкового, влечет ошибку компиляции. К тому же теперь, получая строки из набора, вы работаете не с объектами, а именно со строками, что устраняет необходимость приведения типов.
*  StringDictionary - набор пар строк «имя-значение», предоставляющих доступ к элементам как по имени, так и по индексу в наборе. Он работает как Hashtable, но и ключ, и значение должны быть строками. Важно понимать, что по умолчанию ключи для объектов StringDictionary нечувствительны к регистру, поэтому ключи "Fourth" и "FOURTH" эквиваленты.
*  NameValueCollection - набор пар строк «имя-значение», предоставляющих доступ к элементам как по имени, так и по индексу в наборе. В классе NameValueCollection допускается несколько значений, соответствующих одному ключу, к тому же, значения можно получать не только по ключу, но и по индексу. Таким образом, при работе с классом NameValueCollection можно хранить несколько значений с одним ключом. Метод Add позволяет делать это.
*  Класс CollectionUtil поддерживает создание объектов Hashtable и SortedList, нечувствительных к регистру. Для создания этих объектов достаточно вызвать метод CreateCaseInsensitiveHashtable или CreateCaselnsensitiveSortedList.
*  Обобщения — это типы, которые принимают имена других типов в качестве параметров, и используют их. Вместо того, чтобы создавать строго типизированные наборы для каждого типа, достаточно создать набор, которые можно настроить на работу с любым типом.
*  Класс-обобщение LinkedList имеет одну особенность — реализацию перечислителя ILinkedListEnumerator, что позволяет перечислять значения списка без использования объектов LinkedListNode.

**Закрепление материала**

*  Что представляет собой коллекция ArrayList?
*  В каких случаях лучше использовать Hashtable, ListDictionary, HybridDictionary, OrderedDictionary?
*  Благодаря какому интерфейсу в коллекциях можно реализовать настраиваемое сравнение в отношении равенства? Для каких коллекций данный интерфейс можно применить?
*  Как получить все значения, соответствующие отдельному ключу в наборе NameValueCollection?
*  Перечислите основные отличия между NameValueCollection и StringDictionary?
*  Сраните ArrayList и StringCollection опишите основные преимущества StringCollection.
*  Перечислите основные коллекции, которые не чувствительны к регистру. Как можно исправить данную проблему?
*  В чем основное преимущество обобщенных коллекций? Какие обобщенные коллекции вам известны?

IO

/Держать подклуючения нужно как можно менше по времени

.NETFrameworkпредоставляетдоступкинформацииодисках, файлахидиректорияхпосредствомметодовисвойствклассов:

DriveInfo

FileInfo

DirectoryInfo

FileSystemWatcher

Path

КлассDriveInfoпредоставляетинформациюо системныхдискахиихсостоянии.Являетсяsealed инеможетвыступатьвролибазового.FileInfo

КлассFileInfoпредназначендляработыс файлами.Внемопределеныметодыдлясоздания, удаления,копирования,перемещенияиоткрытия файлов.Являетсяsealedинеможетвыступатьв ролибазового.DirectoryInfo

КлассDirectoryInfoсодержитметодыдля создания,перемещенияиперечисления директорийи/илиподдиректорийкомпьютера. Являетсяsealedинеможетвыступатьвроли базового.FileSystemWatcher

КлассFileSystemWatcherспециализируетсяна отслеживанииизмененийвсистеме.Вызывает соответствующиесобытияприсоздании, удаленииилиизменениифайловидиректорий.Path

СтатическийклассPathпредназначен дляработыспутямифайловойсистемы. Всеоперацииклассавыполняютсяв кросс-платформеннойманере.Классы потоков

Байтовые потоки:

BufferedStream

FileStream

MemoryStream

UnmanagedMemoryStream

Символьные потоки:

StreamReader

StreamWriter

StringReader

StringWriter

Двоичные потоки:

BinaryReader

BinaryWriterStream

АбстрактныйклассStream,является базовымдлявсехклассовпотоков.Потоки

BufferedStream–заключает байтовый поток в оболочку и добавляет буферизацию.

FileStream–поток для осуществления файлового ввода/вывода.

MemoryStream–поток для работы с памятью.

UnmanagedMemoryStream–поток для работы с неуправляемой памятью.Потоки

На вершине иерархии классов символьных потоков находятсяабстрактные классы TextReaderи TextWriter.

StreamReader/StreamWriter–предназначены для ввода/вывода символов из байтового потока, являются оболочкой для потоков байтового ввода/вывода.

StringReader/StringWriter–предназначены для ввода/вывода символов в символьную строку.Потоки

Используются дляфайловоговвода/вывода данных имеющих не строковой тип.

BinaryReader/BinaryWriter-представляют собой оболочку для байтовых потоков.

Позволяют сохранять информацию непосредственно в двоичном формате, что исключает необходимость предварительного преобразования данных.

Сжатие данных

System.IO.Compression-пространствоимен,вкотором определеныклассыдляработы сосжатиемданных.

Сжатие данных

GZipStreampеализуетстандартныйалгоритм сжатиябезпотерь,включаетциклическийконтроль суммыдляобнаруженияповрежденныхданных.

Сжатие данных

DeflateStreamвыполняетсжатие/декомпрессию данныхпоалгоритмуЛемпеля-ЗиваLZ77с использованиемкодированияпометодуХаффмана.

System.IO.IsolatedStorage-пространствоимен,вкотором определеныклассыдляработыс изолированнымхранилищем.

**Изолированное хранилище рекомендуетсяиспользовать для:**

* + •Промежуточного хранения и загрузки элементов управления
  + •Хранения общих компонентов
  + •Хранения личных данных пользователей на сервере
  + •Перемещения личных данных пользователей

**Изолированное хранилище не рекомендуется использовать для:**

* + •Хранения конфиденциальных и важных данных
  + •Хранения кода
  + •Хранения данных развертывания

DirectoryInfo

tatic void Main()

{

// Экземпляр класса DirectoryInfo, ссылается на директорию - C:\Windows\assembly.

var directory = new DirectoryInfo(@"C:\Windows\assembly");

// Вывод информации о каталоге (Директории).

if (directory.Exists) // Если указанная директория существует, то выводим о ней информацию.

{

Console.WriteLine("FullName : {0}", directory.FullName); // Полное Имя директории, (включая путь).

Console.WriteLine("Name : {0}", directory.Name); // Имя директории, (НЕ включая путь).

Console.WriteLine("Parent : {0}", directory.Parent); // Родительская директория.

Console.WriteLine("CreationTime : {0}", directory.CreationTime); // Время создания директории.

Console.WriteLine("Attributes : {0}", directory.Attributes.ToString()); // Аттрибуты.

Console.WriteLine("Root : {0}", directory.Root); // Корневой диск, на котором находится директория.

Console.WriteLine("LastAccessTime: {0}", directory.LastAccessTime); // Время последнего доступа к директории.

Console.WriteLine("LastWriteTime : {0}", directory.LastWriteTime); // Время последнего изменения файлов в директории.

}

else

{

Console.WriteLine("Директория с именем: {0} не существует.", directory.FullName);

}

\*\*\*\*///

var directory = new DirectoryInfo(@".");

// Проверка на существование указанной директории.

if (directory.Exists)

{

// Выводим информацию о каталоге.

Console.WriteLine("FullName : {0}", directory.FullName);

Console.WriteLine("Name : {0}", directory.Name);

Console.WriteLine("Parent : {0}", directory.Parent);

Console.WriteLine("CreationTime: {0}", directory.CreationTime);

Console.WriteLine("Attributes : {0}", directory.Attributes.ToString());

Console.WriteLine("Root : {0}", directory.Root);

Console.Write("\n");

// Получаем все файлы с расширением .txt.

FileInfo[] files = directory.GetFiles("\*.txt");

// Сколько файлов с расширением .txt в данной директории.

Console.WriteLine("Найдено {0} \*.txt файлов\n", files.Length);

// Выводим информацию о каждом файле.

foreach (FileInfo file in files)

{

Console.WriteLine("File name : {0}", file.Name);

Console.WriteLine("File size : {0}", file.Length);

Console.WriteLine("Creation : {0}", file.CreationTime);

Console.WriteLine("Attributes: {0}", file.Attributes.ToString());

Console.Write("\n");

}

}

else

{

Console.WriteLine("Директория с именем: {0} не существует.", directory.FullName);

}

// Delay.

Console.ReadKey();

}

Создание дерикторий

var directory = new DirectoryInfo(@".");

Console.WriteLine(directory.FullName);

// Создание в TESTDIR новых подкаталогов.

if (directory.Exists)

{

// Создаем D:\TESTDIR\SUBDIR.

directory.CreateSubdirectory("SUBDIR");

// Создаем D:\TESTDIR\MyDir\SubMyDir.

directory.CreateSubdirectory(@"MyDir\SubMyDir");

Console.WriteLine("Директории созданы.");

}

else

{

Console.WriteLine("Директория с именем: {0} не существует.", directory.FullName);

}

Удаление директорий

static void Main()

{

// Выводим информацию о дисках.

string[] drives = Directory.GetLogicalDrives();

Console.WriteLine("Имеющиеся диски:");

foreach (string drive in drives)

Console.WriteLine("- {0}", drive);

var directory = new DirectoryInfo(@".");

Console.WriteLine("\nГотовимся удалять:\n");

Console.WriteLine(directory.FullName + @"\MyDir\SubMyDir");

Console.WriteLine(directory.FullName + @"\SUBDIR");

Console.WriteLine("\nНажмите Enter для удаления.\n");

// Задержка перед удалением.

Console.ReadKey();

try

{

// Удаление каталогов.

Directory.Delete(@".\SUBDIR");

// Второй параметр определяет, будут ли удалены также и все вложенные подкаталоги.

Directory.Delete(@"D:\TESTDIR\MyDir", true);

Console.WriteLine("Каталоги успешно удалены.");

}

catch (Exception e) { Console.WriteLine(e.Message); }

// Delay.

Console.ReadKey();

}

}

Создания и удаления файлов

static void Main()

{

// Создаем новый файл в корневом каталоге диска D:

var file = new FileInfo(@"D:\Test.txt");

// FileMode.OpenOrCreate - ЕСЛИ: существует ТО: открыть ИНАЧЕ: создать новый

// FileAccess.Read - только для чтения,

// FileShare.None - Совместный доступ - Нет.

FileStream stream = file.Open(FileMode.OpenOrCreate, FileAccess.Read, FileShare.None);

// Выводим основную информацию о созданном файле.

Console.WriteLine("Full Name : {0}", file.FullName);

Console.WriteLine("Attributes : {0}", file.Attributes.ToString());

Console.WriteLine("CreationTime: {0}", file.CreationTime);

Console.WriteLine("Нажмите любую клавишу для удаления файла.");

Console.ReadKey();

// Закрываем FileStream.

stream.Close();

// Удаляем файл.

file.Delete();

Console.WriteLine("Файл успешно удален.");

// Delay.

Console.ReadKey();

}

Запись байтов в файл

static void Main()

{

// Создаем новый файл в корневом каталоге диска D:

var file = new FileInfo(@"D:\Test.txt");

// FileMode.OpenOrCreate - ЕСЛИ: существует ТО: открыть ИНАЧЕ: создать новый

// FileAccess.Read - только для чтения,

// FileShare.None - Совместный доступ - Нет.

FileStream stream = file.Open(FileMode.OpenOrCreate, FileAccess.Read, FileShare.None);

// Выводим основную информацию о созданном файле.

Console.WriteLine("Full Name : {0}", file.FullName);

Console.WriteLine("Attributes : {0}", file.Attributes.ToString());

Console.WriteLine("CreationTime: {0}", file.CreationTime);

Console.WriteLine("Нажмите любую клавишу для удаления файла.");

Console.ReadKey();

// Закрываем FileStream.

stream.Close();

// Удаляем файл.

file.Delete();

Console.WriteLine("Файл успешно удален.");

// Delay.

Console.ReadKey();

}

MemoryStream

Выполняет запись/чтение не на жесткий диск,а на управляемую кучу.

static void Main()

{

// Создаем объект класса MemoryStream.

var memory = new MemoryStream();

// Задаем требуемый объем памяти.

// memory.Capacity = 256;

// Записываем байты в поток.

for (var i = 0; i < 256; i++)

{

memory.WriteByte((byte)i);

}

// Переставляем внутренний указатель на начало.

memory.Position = 0;

// Считывам байты из потока.

for (int i = 0; i < 256; i++)

{

Console.Write(" " + memory.ReadByte());

}

Console.WriteLine("\n" + new string('-', 80));

// Сохраняем данные из MemoryStream в массив байт.

byte[] array = memory.ToArray();

foreach (byte b in array)

{

Console.Write(b);

}

Console.WriteLine("\n" + new string('-', 80));

// Сохраняем данные из MemoryStream в файл.

var file = new FileStream("Dump.dat", FileMode.Create, FileAccess.ReadWrite);

memory.WriteTo(file);

// Переставляем внутренний указатель на начало.

file.Position = 0;

// Считывам байты из файла.

for (int i = 0; i < 256; i++)

{

Console.Write(" " + file.ReadByte());

}

memory.Close();

file.Close();

// Delay.

Console.ReadKey();

StreamWriter(запись строк)

static void Main()

{

// Создаем файл.

var file = new FileInfo("Text.txt");

// С помощью экземпляра StreamWriter записываем в файл несколько строк текста.

StreamWriter writer = file.CreateText();

writer.WriteLine("Первая строка текста...");

writer.WriteLine("Вторая строка текста...");

// Вставляем символ начала новой строки.

writer.Write(writer.NewLine);

writer.WriteLine("Третья строка чисел:");

for (int i = 0; i < 10; i++)

{

writer.Write(i + " ");

}

// Вставляем символ начала новой строки.

writer.Write(writer.NewLine);

// Метод Close() автоматически очищает все буферы!

writer.Close();

Console.WriteLine("Файл Text.txt создан и в него записан текст.");

// Delay.

Console.ReadKey();

}

StreamReader(чтение строк)

static void Main()

{

// Создаем файл.

// var file = new FileInfo("Text.txt");

// С помощью экземпляра StreamWriter записываем в файл несколько строк текста.

// StreamWriter writer = file.CreateText();

var file2 = new FileStream("text.txt", FileMode.OpenOrCreate, FileAccess.ReadWrite);

var writer = new StreamWriter(file2, Encoding.GetEncoding(1251));

writer.WriteLine("Первая строка текста...");

writer.WriteLine("Вторая строка текста...");

// Вставляем символ начала новой строки.

writer.Write(writer.NewLine);

writer.WriteLine("Третья строка чисел:");

for (int i = 0; i < 10; i++)

{

writer.Write(i + " ");

}

// Метод Close() автоматически очищает все буферы.

writer.Close();

Console.WriteLine("Файл Text.txt создан и в него записан текст.");

// Выводим информацию из файла на консоль при помощи StreamReader.

Console.WriteLine("Содержимое файла Text.txt:\n");

StreamReader reader = File.OpenText("Text.txt");

string input;

// Выводим содержимое файла в консоль.

while ((input = reader.ReadLine()) != null)

{

Console.WriteLine(input);

}

reader.Close();

// Delay.

Console.ReadKey();

}

Копирования файлов

static void Main()

{

// Создаем объект для работы с файлом.

var file = new FileInfo(@"C:\Windows\notepad.exe");

// var dir = new DirectoryInfo(".");

// Копируем содержимое файла.

try

{

file.CopyTo(@"D:\notepad.exe");

Console.WriteLine("Файл успешно скопирован!");

}

catch (Exception e)

{

Console.WriteLine(e.Message);

}

Path

static void Main()

{

// Создание строки, содержащей адрес.

string path = @"C:\Windows\notepad.exe";

Console.WriteLine(path);

// Вывод на экран значений свойств класса-объекта Path.

Console.WriteLine("Ext: {0}", Path.GetExtension(path));

// ChangeExtension не изменяет расширение у файла — он просто создает строку с другим расширением,

// которую вы сами должны использовать для реального переименования (например, через статический метод Move класса File)

Console.WriteLine("Change Path: {0}", path = Path.ChangeExtension(path, "bak"));

Console.WriteLine(path);

// Delay.

Console.ReadKey();

}

FileSystemWatcher (Отслеживает изменения)

// Создание наблюдателя и сосредоточение его внимания на системном диске.

var watcher = new FileSystemWatcher { Path = @"D:\axaxa" };

// Зарегистрировать обработчики событий.

watcher.Created += new FileSystemEventHandler(WatcherChanged);

// watcher.Deleted += WatcherChanged;

// Начать мониторинг.

watcher.EnableRaisingEvents = true;

// Delay.

var change = watcher.WaitForChanged(WatcherChangeTypes.All);

Console.WriteLine(change.ChangeType);

Console.ReadKey();

}

// Обработчик события.

static void WatcherChanged(object sender, FileSystemEventArgs e)

{

Console.WriteLine("Directory changed({0}): {1}", e.ChangeType, e.FullPath);

}

Записи/Чтение в поток

static void Main()

{

// Создаем поток для работы с памятью.

Stream stream = new MemoryStream();

// Добавляем в поток данные.

Console.WriteLine("Запись в поток начата...");

AppendToStream(stream, new byte[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 });

Console.WriteLine("Данные успешно записаны!");

// Считываем назад.

Console.WriteLine("\nСодержимое потока:");

DumpStream(stream);

// Удаляем поток.

stream.Close();

// Delay.

Console.ReadKey();

}

// Метод для вывода содержимого потока на экран.

static void DumpStream(Stream stream)

{

// Установить курсор на начало потока.

stream.Position = 0;

// Обработать поток в цикле и показать его содержимое.

while (stream.Position != stream.Length)

{

Console.WriteLine("{0:x3}", stream.ReadByte());

}

}

// Метод для добавления данных в поток.

static void AppendToStream(Stream stream, byte[] data)

{

// Установить курсор в конец потока.

stream.Position = stream.Length;

// Добавить байты.

stream.Write(data, 0, data.Length);

}

}

Разные способы для чтения StreamReader

// Открываем файл для чтения.

FileStream file = File.Open(@"D:\test.txt", FileMode.OpenOrCreate, FileAccess.Write);

// Создаем поток для чтения данных из файла.

StreamReader reader = new StreamReader(file);

// Читаем до конца.

Console.Write(reader.ReadToEnd());

// Закрываем файл и удаляем поток.

reader.Close();

//file.Close();

Console.WriteLine("\n");

// Еще раз открываем файл, используя другой способ.

reader = File.OpenText(@"D:\test.txt");

// Читаем до конца и закрываем файл.

Console.Write(reader.ReadToEnd());

reader.Close();

Console.WriteLine("\n");

// Читаем весь текст, содержащийся в файле.

Console.WriteLine(File.ReadAllText(@"D:\test.txt"));

// Задержка.

Console.ReadKey();

}

Поиск по файлу

static void Main()

{

// Подготовка файла.

var reader = File.OpenText(@"D:\test.txt");

// Последовательный обход файла.

while (!reader.EndOfStream)

{

// Чтение файла построчно.

string line = reader.ReadLine();

// Если нужный текст найден, прекратить чтение.

if (line != null && line.Contains("Andrew"))

{

// Обнаружив слово "Andrew", уведомить пользователя и прекратить чтение файла.

Console.WriteLine("Found:");

Console.WriteLine(line);

break;

}

}

SrtingReader

string s = "Hello all!" + Environment.NewLine + "This is a multi-line \n\r text string.";

var reader = new StringReader(s);

// Метод Peek возвращает целочисленное значение, чтобы определить, произошла ошибка или достигнут конец файла.

// Это позволяет пользователю сначала проверить, не равно ли возвращенное значение -1, прежде чем приводить его к типу Char.

while (reader.Peek() != -1)

{

string line = reader.ReadLine();

Console.WriteLine(line);

}

SrtingWriter

var writer = new StringWriter();

writer.WriteLine("Hello all ");

writer.Write("This is a multi-line ");

writer.WriteLine("text string.");

Console.WriteLine(writer.ToString());

BinaryWriter

// Создаем файл.

FileStream file = File.Create(@"D:\TEMP\test.txt");

// Сообщаем с файлом наш БайнариРайтер.

var writer = new BinaryWriter(file);

// Записываем в файл разные данные.

long number = 100;

var bytes = new byte[] { 10, 20, 50, 100 };

string s = "hunger";

writer.Write(number);

writer.Write(bytes);

writer.Write(s);

// Удаляем поток.

writer.Close();

BinaryReader

// Открываем файл.

FileStream file = File.Open(@"D:\TEMP\test.txt", FileMode.Open);

// Сообщаем поток с файлом.

var reader = new BinaryReader(file);

// Читаем из файла разные данные.

long number = reader.ReadInt64();

byte[] bytes = reader.ReadBytes(4);

string s = reader.ReadString();

// Удаляем поток.

reader.Close();

// Выводим на экран то, что удалось прочитать.

Console.WriteLine(number);

foreach (byte b in bytes)

{

Console.Write("[{0}]", b);

}

Console.WriteLine();

Console.WriteLine(s);

// Delay.

Console.ReadKey();

BufferedStream

/Собирает все данные, а уже потом передает их в FileStream

// Использвание BufferedStream.

// Буфер — это блок байтов памяти, который используется для кэширования данных,

// тем самым уменьшая количество обращений к операционной системе.

// Таким образом, буферизация повышает скорость чтения и записи.

// Методы Read и Write класса BufferedStream обслуживают буфер автоматически.

FileStream file = File.Create(@"D:\test.txt");

BufferedStream buffered = new BufferedStream(file);

StreamWriter writer = new StreamWriter(buffered);

writer.WriteLine("Some data...");

buffered.Position = 0;

// При сбросе буфера buffered, данные из него попадают в связанный поток - file.

writer.Close();

// Delay.

Console.ReadKey();

Zip – архив

// Создание файла и архива.

FileStream source = File.OpenRead(@"D:\test.txt");

FileStream destination = File.Create(@"D:\archive.zip");

// Создание компрессора.

GZipStream compressor = new GZipStream(destination, CompressionMode.Compress);

// Заполнение архива информацией из файла.

int theByte = source.ReadByte();

while (theByte != -1)

{

compressor.WriteByte((byte)theByte);

theByte = source.ReadByte();

}

// Удаление компрессора.

compressor.Close();

РОБОТА С ТЕСКСТОМ

String

Символ @ указывает на то что управляющие символы не учитываются.

В таблицу интернирования попадают строки на момент запуска программы.Во время работы таблица интернирования не действует и создается новая строка на куче

// Метод String.Intern() - извлекает системную ссылку на указанный строковой литерал из пула интернирования.

string stringNew = String.Intern(Console.ReadLine());

// Создаем две строки.

// Эти строковые переменные ссылаются на одно место в памяти.

string string1 = "c:\\windows\\system32";

string string2 = @"c:\windows\system32";

// Демонстрация того, что ссылки действительно совпадают.

Console.WriteLine("string1 = " + string1);

Console.WriteLine("string2 = " + string2);

Console.WriteLine("Object.ReferenceEqual(string1, string2): {0}", Object.ReferenceEquals(string1, string2));

// Попытка создать еще одну строковую переменную, которая будет ссылаться на существующий литерал в пуле интернирования.

Console.WriteLine("\nEnter some string:");

// Метод String.Intern() - извлекает системную ссылку на указанный строковой литерал из пула интернирования.

string stringNew = String.Intern(Console.ReadLine());

// Сравнение.

Console.WriteLine("Object.ReferenceEqual(string1, stringNew): {0}", ReferenceEquals(string1, stringNew));

CultureInfo

// Получение информации о текущей культуре.

CultureInfo currentCulture = CultureInfo.CurrentCulture;

Console.WriteLine("Текущая локаль: {0}.", currentCulture);

// Информация о всех культурах доступных в системе.

CultureInfo[] cultureInfo = CultureInfo.GetCultures(CultureTypes.AllCultures);

Console.WriteLine("В системе определены {0} различные культуры.",

cultureInfo.Length);

foreach (CultureInfo ci in cultureInfo)

{

Console.WriteLine(ci.EnglishName + " | " + ci.ToString());

}

// Все культуры созданные пользователем.

cultureInfo = CultureInfo.GetCultures(CultureTypes.UserCustomCulture);

if (cultureInfo.Length < 1)

{

Console.WriteLine("Пользователь не создавал собственные локали.");

}

else

{

Console.WriteLine("Локали, созданные пользователем:");

RegionInfo

// Получение информации о текущем регионе.

RegionInfo regionInfo = RegionInfo.CurrentRegion;

Console.WriteLine("EnglishName региона:\t{0}.", regionInfo.EnglishName);

Console.WriteLine("NativeName региона:\t{0}.", regionInfo.NativeName);

Console.WriteLine(new string('-', 35));

Console.WriteLine("CurrencySymbol региона:\t{0}", regionInfo.CurrencySymbol);

Console.WriteLine("CurrencyEnglishName:\t{0}", regionInfo.CurrencyEnglishName);

Console.WriteLine("CurrencyNativeName:\t{0}", regionInfo.CurrencyNativeName);

Console.WriteLine(new string('-', 35));

// Получение информации о текущем формате даты: названия дней.

string[] days = CultureInfo.CurrentCulture.DateTimeFormat.DayNames;

Console.WriteLine("Дни недели:");

foreach (string day in days)

{

Console.WriteLine(day);

}

Console.WriteLine(new string('-', 35));

// Получение информации о формате даты в немецком языке: названия дней.

days = CultureInfo.GetCultureInfo("de-DE").DateTimeFormat.DayNames;

Console.WriteLine("Дни недели на немецком:");

foreach (string day in days)

{

Console.WriteLine(day);

}

// Delay.

Console.ReadKey();

Manifest

Для настроек запуска приложения.

Add->Applicaion Manifest.

IFormatProvider

private double real;

private double imaginary;

public Complex(double real, double imaginary)

{

this.real = real;

this.imaginary = imaginary;

}

// Реализация IFormattable

public string ToString(string format, IFormatProvider formatProvider)

{

var builder = new StringBuilder();

if (format == "TEST")

{

// Генерация отладочного вывода для данного объекта

builder.Append(this.GetType() + "\n");

builder.AppendFormat(" действительная:\t{0}\n", real);

builder.AppendFormat(" мнимая:\t\t{0}\n", imaginary);

}

else

{

builder.Append(" ( ");

builder.Append(real.ToString(format, formatProvider));

builder.Append(" : ");

builder.Append(imaginary.ToString(format, formatProvider));

builder.Append(" ) ");

}

return builder.ToString();

}

}

public class Program

{

static void Main()

{

var my = CultureInfo.CurrentCulture;

var us = new CultureInfo("uk-UA");

var complex = new Complex(12.3456, 1234.5678);

string stringComplex = complex.ToString("F", my); // F - по умолчанию = F2.

Console.WriteLine(stringComplex); // RU - запятая в числе.

stringComplex = complex.ToString("F2", us);

Console.WriteLine(stringComplex); // US - точка в числе.

Console.WriteLine("\nОтладочный вывод:\n{0:TEST}", complex);

// Delay.

Console.ReadKey();

}

Encoding

// Строка для изменения кодировок.

string leUnicodeStr = "здорово!";

// Настройки кодировок.

// Unicode - Получает кодировку для формата UTF-16 с прямым порядком байтов.

Encoding leUnicode = Encoding.Unicode;

// BigEndianUnicode - Получает кодировку для формата UTF-16 с обратным порядком байтов.

Encoding beUnicode = Encoding.BigEndianUnicode;

// UTF8 - Получает кодировку для формата UTF-8.

Encoding utf8 = Encoding.UTF8;

// Массивы байтов для хранения конвертированной строки.

byte[] leUnicodeBytes = leUnicode.GetBytes(leUnicodeStr); // В Unicode.

byte[] beUnicodeBytes = Encoding.Convert(leUnicode, beUnicode, leUnicodeBytes); // В Endian Unicode.

byte[] utf8Bytes = Encoding.Convert(leUnicode, utf8, leUnicodeBytes);

Encoding (Запись в файл с кодировкой)

// Записываем файл в кодировке UTF32

var swUtf32 = new StreamWriter("text.txt", true, Encoding.UTF32);

swUtf32.WriteLine("Hello, world!");

swUtf32.Close();

//File.WriteAllText("text.txt","Hello World!",Encoding.BigEndianUnicode);

//Console.WriteLine(File.ReadAllText("text.txt"));

// Delay.

StringBuilder

var builder = new StringBuilder();

builder.Append("StringBuilder ").Append("является ").Append("очень ... ");

string build1 = builder.ToString();

builder.Append("быстрым");

string build2 = builder.ToString();

Регулярные выражения

System.Text.RegularExpressions - пространство имён где, хранятся нужные методы для работы с Regex'ами.

МЕТАСИМВОЛЫ - это символы для составления Шаблона поиска.

\d Определяет символы цифр.

\D Определяет любой символ, который не является цифрой.

\w Определяет любой символ цифры, буквы или подчеркивания.

\W Определяет любой символ, который не является цифрой, буквой или подчеркиванием. С пробелом

\s Определяет любой непечатный символ, включая пробел. Пробелы табы

\S Определяет любой символ, кроме символов табуляции, новой строки и возврата каретки. Все кроме пробелов табов

. Определяет любой символ кроме символа новой строки.

\. Определяет символ точки.

\*/

КВАНТИФИКАТОРЫ - это символы которые определяют, где и сколько раз необходимое вхождение символов может встречаться.

^ - c начала строки.

$ - с конца строки.

+ - одно и более вхождений подшаблона в строке.

// | - символ для указания вариантов шаблона (ИЛИ).

string pattern = "test|str|aaaa";

\*/

namespace RegularExpressions

{

class Program

{

static void Main()

{

// Данный шаблон определяет символы цифр.

const string pattern = @"\d";

// Создаем экземпляр класса Regex и передаем в качестве аргумента конструктора шаблон.

var regex = new Regex(pattern);

while (true)

{

// Строка с которой будет сравниваться шаблон.

string @string = Console.ReadKey().KeyChar.ToString();

if (@string == " ")

break;

// Проверка соответствия строки шаблону.

// Метод IsMatch - сравнивает принимаемую в первом параметре строку с шаблоном.

// IsMatch - метод возвращающий bool. True - в случае, если шаблон соответсвует строке, false - в противном случае.

bool success = regex.IsMatch(@string);

Console.WriteLine(success ?

" соответствует шаблону \"{0}\"" :

" НЕ соответствует шаблону \"{0}\"",

pattern);

}

}

// Нужная подстрока(\d+) должна быть в конце строки.

pattern = @"\d+$"; // "123", "test123", "4556yyy789"

// Вся строка полностью от начала(^) и до конца($) должна соответствовать тому, что между ^ и $ (\d в данном случае).

pattern = @"^\d\*\D+\d+$"; // "23"

Regex.Replace

/ Регулярные выражения.

/\*

Метод Regex.Replace заменяет в первом параметре - строке (myString)

подстроку соответствующую шаблону (String) во втором параметре, на подстроку-вставку в третем параметре (Test).

Regex.Replace("myString","String","Test"); - результат myTest

Шаблонные переменные - обьявляются в текущем шаблоне таким образом:

(?<somevar>\d+) - объявление переменной.

${somevar} - использование шаблонных переменных в новом шаблоне.

Пример: Regex.Replace(@"test123firststr456secondstr",

@"test(?<first>\d+)firststr(?<second>\d+)secondstr",

@"test${second}firststr${first}secondstr")

Этот пример вернёт строку test456firststr123secondstr он изменит шаблон так, что подстрока 123 и 456 поменяются местами.

В этом же примере используется новая конструкция {1,2} и подобные,

она определяет кол-во повторов нужного нам элемента(\S, \d, etc). Формат: {min,max}.

Пример: \d{2,4} - числа длиной от двух до четырёх символов (12, 123, 1234) 12345 - уже не выходит.

\*/

namespace RegularExpressions4

{

class Program

{

static void Main()

{

// Замена подстроки соответствующей шаблону - пробелом.

Console.WriteLine(Regex.Replace("test123aaa4x5x6bbb789ccc", // Исходная строка.

@"\d+", // Шаблон.

" ")); // Символ для замены.

Console.WriteLine(Regex.Replace("test123aaa4x5x6bbb789ccc", // Исходная строка.

@"\d", // Шаблон.

" ")); // Символ для замены.

Console.WriteLine(Regex.Replace("02/05/1982", // Исходная строка.

@"(?<месяц>\d{1,2})/(?<день>\d{1,2})\/(?<год>\d{2,4})", // Шаблон.

//@"(?<месяц>\d{1,2})\/(?<день>\d{1,2})\/(?<год>\d{2,4})", // Шаблон.

"${день}-${месяц}-${год}")); // Новый формат.

Console.WriteLine(Regex.Replace(@"test123firststr456secondstr", // Исходная строка.

@"test(?<var1>\d+)firststr(?<var2>\d+)secondstr", // Шаблон.

@"test${var2}firststr${var1}secondstr")); // Новый формат.

Console.WriteLine(Regex.Replace("5 is less than 10", // Исходная строка.

@"\d", // Шаблон.

m => (int.Parse(m.Value) + 1).ToString())); // Функция изменения совпадения

Regex (Почта)

Regex regex;

string pattern;

string text;

// Возможно указать необходимые символы между скобок [].

pattern = "^[qwerty]+$";

text = "qwec"; // Анализируемая строка.

regex = new Regex(pattern);

Console.WriteLine("{0} == {1} : {2}\n", text, pattern, regex.IsMatch(text));

// Строка может состоять только из символов - [qwerty]. Например: qqq, yyqyqyyyq, eeer ...

pattern = "^[qwerty]+$";

text = "qrwere"; // Анализируемая строка.

regex = new Regex(pattern);

Console.WriteLine("{0} == {1} : {2}\n", text, pattern, regex.IsMatch(text));

pattern = "^[abcdefghigklmnopqrstuvwxyz]+$";

text = "test"; // Анализируемая строка.

regex = new Regex(pattern);

Console.WriteLine("{0} == {1} : {2}\n", text, pattern, regex.IsMatch(text));

// Второй способ, a-z это замена более длинного шаблона abcdefghigklmnopqrstuvwxyz

pattern = @"^[a-z]+$";

text = "test"; // Анализируемая строка.

regex = new Regex(pattern);

Console.WriteLine("{0} == {1} : {2}\n", text, pattern, regex.IsMatch(text));

// 0-9 это замена 01234567890.

pattern = "^[a-z0-9]+$";

text = "test007"; // Анализируемая строка.

regex = new Regex(pattern);

Console.WriteLine("{0} == {1} : {2}\n", text, pattern, regex.IsMatch(text));

// 0-9 это замена 01234567890.

pattern = "^[a-z0-9]+$";

text = "test 007"; // Анализируемая строка.

regex = new Regex(pattern);

Console.WriteLine("{0} == {1} : {2}\n", text, pattern, regex.IsMatch(text));

// Шаблон даты.

pattern = @"^\d{2}-\d{2}-\d{4}$";

text = "02-05-1982"; // Анализируемая строка.

regex = new Regex(pattern);

Console.WriteLine("{0} == {1} : {2}\n", text, pattern, regex.IsMatch(text));

// Квантификатор \* значит, что вхождение 0 и более раз...

pattern = @"^\d\*$";

text = "ertty"; // Анализируемая строка.

regex = new Regex(pattern);

Console.WriteLine("{0} == {1} : {2}\n", text, pattern, regex.IsMatch(text));

// Квантификатор \* значит, что вхождение 0 и более раз...

pattern = @"^\d\*$";

text = ""; // Анализируемая строка.

regex = new Regex(pattern);

Console.WriteLine("{0} == {1} : {2}\n", text, pattern, regex.IsMatch(text));

// Простой шаблон сравнения e-mail.

pattern = @"^[0-9a-z\_-]+@[\S]+\.\S{2,4}$";

text = "test@mail123.rлu"; // Анализируемая строка.

regex = new Regex(pattern);

Console.WriteLine("{0} == {1} : {2}\n", text, pattern, regex.IsMatch(text));

Match

namespace RegularExpressions7

{

class Program

{

static void Main()

{

string input = "";

input += "<a href='http://cbsystematics.com'>Home-page</a>";

input += "<a href='http://google.com'>Search</a>";

input += "<a href='https://ya.ru'>Yandex</a>";

input += "<a href='https://yandex.ru'>Yandex Full</a>";

input += "<a href='http://microsoft.com'>Microsoft</a>";

var regex = new Regex(@"href='(?<link>\S+)'>(?<text>\S+)</a>");

Console.WriteLine(input);

// for(текущее\_значение\_m; условие; действие\_при\_каждой\_итерации).

for (var m = regex.Match(input); m.Success; m = m.NextMatch())

{

// {0,-25} - значит что выделить 25 знакомест под вывод {0}. (-) - значит "прижаться" влево :)

Console.WriteLine("ССЫЛКА: {0,-25} на: {1,-4} позиции с именем: {2}", m.Groups["link"],

m.Groups["link"].Index,

m.Groups["text"]);

}

Console.WriteLine(new string('-', 25));

foreach (Match m in regex.Matches(input))

{

Console.WriteLine("ССЫЛКА: {0,-25} на: {1,-4} позиции с именем: {2}", m.Groups["link"],

m.Groups["link"].Index,

m.Groups["text"]);

}

Console.WriteLine(new string('-',20));

var htmlQuery = from Match m in regex.Matches(input)

where m.Groups["link"].Value.StartsWith("https")

select m;

foreach (var m in htmlQuery)

{

Console.WriteLine("ССЫЛКА: {0,-25} на: {1,-4} позиции с именем: {2}", m.Groups["link"],

m.Groups["link"].Index,

m.Groups["text"]);

}

// Delay.

Console.ReadKey();

}

Regex regex = new Regex(@"[0-9A-Za-z\_.-]+@[0-9a-zA-Z-]+\.[a-zA-Z]{2,4}");

MatchCollection collection = regex.Matches("русский edu@cbsystematics.com текст123ещерусскийsupport@cbsystematics.comтекст");

foreach (Match element in collection)

{

Console.WriteLine("{0,-25} на {1,-3} позиции.", element.Value, element.Index);

}

XML. Конфигурация. Реестр

eXtensibleMarkup Language

текстовыйформат,предназначенныйдляхраненияструктурированныхданныхдля обменаинформациеймеждупрограммами,атакжедлясозданиянаегоосновеболее специализированныхязыковразметки

XML Path Languageth

XPath(XMLPathLanguage)—языкзапросовкэлементамXML-документа.РазработандляорганизациидоступакчастямдокументаXMLвфайлахтрансформацииXSLTиявляетсястандартомконсорциумаW3C.XPathпризванреализоватьнавигациюпоDOMвXML.

RegistryРеестр

РеестрWindowsилисистемныйреестр*(WindowsRegistry)*—иерархическипостроеннаябаза данныхпараметровинастроеквбольшинствеоперационныхсистемMicrosoftWindows.

Реестрсодержитинформациюинастройкидляаппаратногообеспечения,программного обеспечения,профилейпользователей,предустановки.БольшинствоизмененийвПанели управления,ассоциациифайлов,системныеполитики,списокустановленногоПОфиксируются вреестре.

Разделы реестра

Реестр

* •HKEY\_CLASSES\_ROOT–содержит информацию о ассоциациях расширения файлов и зарегистрированных COM объектах.
* •HKEY\_CURRENT\_USER –содержит настройки для текущего пользователя.
* •HKEY\_LOCAL\_MACHINE–содержит настройки локального компьютера.
* •HKEY\_USERS–содержит настройки пользователя по умолчанию.
* •HKEY\_CURRENT\_CONFIG–содержит информацию о настройках, которые не являются специфическими для пользователя.
* •HKEY\_PERFORMANCE\_DATA–содержит информацию о производительности программных компонентов.
* •HKEY\_DYN\_DATA–содержит динамические изменяемые данные реестра, на данный момент является устаревшим.

Отображения текста xml-файла

// Примечание: В окне свойств для файла book.xml, свойству - Copy to Output Directory, присвойте значение - Copy always.

// Это необходимо чтобы файл book.xml копировался в папку рядом с \*.exe файлом.

namespace XML

{

class Program

{

static void Main()

{

// Загрузка XML из файла.

var document = new XmlDocument();

document.Load("books.xml");

// Показ содержимого XML.

Console.WriteLine(document.InnerText);

Console.WriteLine(new string('-', 20));

// Показ кода XML документа.

Console.WriteLine(document.InnerXml);

Загрузка содержимого xml-файла

FileStream stream = new FileStream("books.xml", FileMode.Open);

XmlTextReader xmlReader = new XmlTextReader(stream);

while (xmlReader.Read())

{

Console.WriteLine("{0,-10} {1,-10} {2,-10}",

xmlReader.NodeType,

xmlReader.Name,

xmlReader.Value);

}

xmlReader.Close();

Загрузка содержимого xml из строки

string xmlData = "<?xml version='1.0' encoding='utf-8' ?><Book><Title>CLR via C#</Title></Book>";

var stringReader = new StringReader(xmlData);

var reader = new XmlTextReader(stringReader);

while (reader.Read())

{

Console.WriteLine("{0,-15} {1,-10} {2,-10}",

reader.NodeType.ToString(),

reader.Name,

reader.Value);

}

reader.Close();

Обход всех елементов xml-файла

var document = new XmlDocument();

document.Load("books.xml");

XmlNode root = document.DocumentElement;

// Напечатает "document.DocumentElement=ListOfBooks"

Console.WriteLine("document.DocumentElement = {0}", root.LocalName);

foreach (XmlNode books in root.ChildNodes)

{

Console.WriteLine("Found Book:");

foreach (XmlNode book in books.ChildNodes)

{

Console.WriteLine(book.Name + ": " + book.InnerText);

}

Console.WriteLine(new string('-',40));

Чтение атрибутов

var reader = new XmlTextReader("books.xml");

while (reader.Read())

{

if (reader.NodeType == XmlNodeType.Element)

{

// Проверка на тип узла необходима, иначе будут найдены не только открывающие элементы (XmlNodeType.Element),

// но и закрывающие (XmlNodeType.EndElement).

if (reader.Name.Equals("Title")) // Закомментировать и выполнить.

{

Console.WriteLine("<{0}>", reader.GetAttribute("FontSize"));

}

}

Чтение атрибутов

static void Main()

{

var reader = new XmlTextReader("books.xml");

while (reader.Read())

{

if (reader.NodeType == XmlNodeType.Element)

{

if (reader.HasAttributes)

{

while (reader.MoveToNextAttribute())

{

Console.WriteLine("{0} = {1}", reader.Name, reader.Value);

}

}

}

}



Запись в xml-файл

var xmlWriter = new XmlTextWriter("books.xml", null);

xmlWriter.WriteStartDocument(); // Заголовок XML - <?xml version="1.0"?>

xmlWriter.WriteStartElement("ListOfBooks"); // Корневой элемент - <ListOfBooks>

xmlWriter.WriteStartElement("Book"); // Книга 1 - <Book

xmlWriter.WriteStartAttribute("FontSize"); // Атрибут - FontSize

xmlWriter.WriteString("8");

// xmlWriter.WriteString("18"); // ="8"

xmlWriter.WriteEndAttribute(); // >

xmlWriter.WriteString("Title-1"); // Title-1

xmlWriter.WriteEndElement(); // </Book>

xmlWriter.WriteEndElement(); // </ListOfBooks>

xmlWriter.Close();

Форматирование xml-файла

var xmlWriter = new XmlTextWriter("books.xml", null)

{

Formatting = Formatting.Indented,

IndentChar = '\t',

Indentation = 1,

QuoteChar = '\''

};

// Включить форматирование документа (с отступом).

// Для выделения уровня элемента использовать табуляцию.

// использовать один символ табуляции.

// Аналогично можно указать выравнивание с помощью четырех пробелов.

xmlWriter.IndentChar = ' ';

xmlWriter.Indentation = 4;

// По умолчанию строки в XML файл записываются с помощью двойных кавычек.

// Использование одиночных кавычек производится так:

xmlWriter.WriteStartDocument();

xmlWriter.WriteStartElement("ListOfBooks");

xmlWriter.WriteStartElement("ListOfBooks", "http://localhost/test");

xmlWriter.WriteStartElement("prefix", "ListOfBooks", "http://localhost/test");

xmlWriter.Close();

Выборка из xml-файла

// Создание XPath документа.

var document = new XPathDocument("books.xml");

XPathNavigator navigator = document.CreateNavigator();

// Прямой запрос XPath.

XPathNodeIterator iterator1 = navigator.Select("ListOfBooks/Book/Title");

while (iterator1.MoveNext())

{

Console.WriteLine(iterator1.Current);

}

Console.WriteLine(new string('-',20));

// Скомпилированный запрос XPath

XPathExpression expr = navigator.Compile("ListOfBooks/Book/Price");

XPathNodeIterator iterator2 = navigator.Select(expr);

while (iterator2.MoveNext())

{

Console.WriteLine(iterator2.Current);

}

Редактирование

// Создание XPath документа.

var document = new XPathDocument("books.xml");

// Единственное назначение XPathDocument - создание навигатора.

XPathNavigator navigator = document.CreateNavigator();

// При создании навигатора при помощи XPathDocument возможно выполнять только чтение.

Console.WriteLine("Навигатор создан только для чтения. Свойство CanEdit = {0}.", navigator.CanEdit);

// Используя XmlDocument навигатор можно использовать и для редактирования.

var xmldoc = new XmlDocument();

xmldoc.Load("books.xml");

navigator = xmldoc.CreateNavigator();

Console.WriteLine("Навигатор получил возможность редактирования. Свойство CanEdit = {0}.", navigator.CanEdit);

// Теперь можно попробовать что-то записать в XML-документ.

// Выполняем навигацию к узлу Book.

navigator.MoveToChild("ListOfBooks", "");

navigator.MoveToChild("Book", "");

// Проводим вставку значений.

navigator.InsertBefore("<InsertBefore>insert\_before</InsertBefore>");

navigator.InsertAfter("<InsertAfter>insert\_after</InsertAfter>");

navigator.AppendChild("<AppendChild>append\_child</AppendChild>");

navigator.MoveToNext("Book", "");

navigator.InsertBefore("<InsertBefore>1111111111</InsertBefore>");

navigator.InsertAfter("<InsertAfter>2222222222</InsertAfter>");

navigator.AppendChild("<AppendChild>3333333333</AppendChild>");

// Сохраняем изменения.

xmldoc.Save("books.xml");

Вычисление выражения xml-файла

double sum = 0;

// Создание XPath документа.

XPathDocument document = new XPathDocument("books.xml");

XPathNavigator navigator = document.CreateNavigator();

// Вычисляющий запрос с предварительной компиляцией.

XPathExpression expression = navigator.Compile("sum(ListOfBooks/Book/Price/text())");

Console.WriteLine(expression.ReturnType);

if (expression.ReturnType == XPathResultType.Number)

{

sum = (double)navigator.Evaluate(expression);

Console.WriteLine(sum);

}

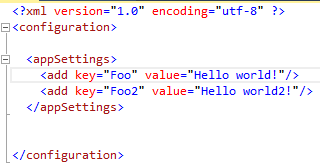
// Вычисляющий запрос без предварительной компиляции.

// Кроме выборки производится арифметическое вычисление.

sum = (double)navigator.Evaluate("sum(//Price/text())\*10");

Console.WriteLine(sum);

Файл кофигурации (Чтение данных)



// 2

NameValueCollection allAppSettings = ConfigurationManager.AppSettings;

Console.WriteLine(allAppSettings["Foo"]);

Console.WriteLine(allAppSettings[0]);

Console.WriteLine(new string('-', 12));

// 3

for (int i = 0; i < allAppSettings.Count; i++)

{

Console.WriteLine(allAppSettings[i]);

}

Console.WriteLine(new string('-', 12));

// 4

foreach (string item in allAppSettings)

{

Console.WriteLine(allAppSettings[item]);

}

Реестр

Структура реестра

// Registry - это класс, предоставляющий эксклюзивный доступ к ключам реестра для простых операций.

// RegistryKey - класс реализует методы для просмотра дочерних ключей, создания новых или чтения и модификации существующих,

// включая установку уровней безопасности для них.

RegistryKey[] regKeyArray = new RegistryKey[] { Registry.ClassesRoot,

Registry.CurrentUser,

Registry.LocalMachine,

Registry.Users,

Registry.CurrentConfig,

Registry.PerformanceData

// Registry.DynData // Устарел.

};

foreach (RegistryKey regKey in regKeyArray)

{

Console.WriteLine("{0} - всего элементов: {1}.", regKey.Name, regKey.SubKeyCount);

}

Рефлексия

Рефлексия(отражение)этопроцесс,вовремякоторогопрограммаможетотслеживатьимодифицироватьсобственнуюструктуруиповедениево времявыполнения.

Парадигмапрограммирования,положеннаявосновуотражения,называетсярефлексивнымпрограммированием.Этоодинизвидов метапрограммирования.

Пример получения екзампляра Type

var my = new MyClass();

Type type;

// Способы получения экземрляра класса Type.

// 1.

type = my.GetType();

Console.WriteLine("1й способ: " + type);

// 2.

type = Type.GetType("TypeTest.MyClass"); // Полное квалифицированое имя типа в строковом представлении.

Console.WriteLine("2й способ: " + type);

// 3.

type = typeof(MyClass);

Console.WriteLine("3й способ: " + type);

Рефлексия

// Получаем разную информацию о Class1.

static void ListVariosStats(Class1 cl)

{

Console.WriteLine(new string('\_', 30) + " Информация о Class1" + "\n");

Type t = cl.GetType();

Console.WriteLine("Полное Имя: {0}", t.FullName);

Console.WriteLine("Базовый класс: {0}", t.BaseType);

Console.WriteLine("Абстрактный: {0}", t.IsAbstract);

Console.WriteLine("Это COM объект: {0}", t.IsCOMObject);

Console.WriteLine("Запрещено наследование: {0}", t.IsSealed);

Console.WriteLine("Это class: {0}", t.IsClass);

}

// Получаем информацию об Именах всех методов Class1.

static void ListMethods(Class1 cl)

{

Console.WriteLine(new string('\_', 30) + " Методы класса Class1" + "\n");

Type t = cl.GetType();

MethodInfo[] mi = t.GetMethods(BindingFlags.Instance

| BindingFlags.Static

| BindingFlags.Public

| BindingFlags.NonPublic | BindingFlags.DeclaredOnly);

foreach (MethodInfo m in mi)

Console.WriteLine("Method: {0}", m.Name);

}

// Получаем информацию об Именах полей Class1.

static void ListFields(Class1 cl)

{

Console.WriteLine(new string('\_', 30) + " Поля класса Class1" + "\n");

Type t = cl.GetType();

FieldInfo[] fi =

t.GetFields(BindingFlags.Instance

| BindingFlags.Static

| BindingFlags.Public

| BindingFlags.NonPublic);

foreach (FieldInfo f in fi)

Console.WriteLine("Field: {0}", f.Name);

}

// Получаем список всех Свойств Class1.

static void ListProps(Class1 cl)

{

Console.WriteLine(new string('\_', 30) + " Свойства класса Class1" + "\n");

Type t = cl.GetType();

PropertyInfo[] pi = t.GetProperties();

foreach (PropertyInfo p in pi)

Console.WriteLine("Свойство: {0}", p.Name);

}

// Получаем список всех Интерфейсов, поддерживаемых Class1.

static void ListInterfaces(Class1 cl)

{

Console.WriteLine(new string('\_', 30) + " Интерфейсы класса Class1" + "\n");

Type t = cl.GetType();

Type[] it = t.GetInterfaces();

foreach (Type i in it)

Console.WriteLine("Интерфейс: {0}", i.Name);

}

// Получаем информацию обо всех конструкторах Class1.

static void ListConstructors(Class1 cl)

{

Console.WriteLine(new string('\_', 30) + " Конструкторы класса Class1" + "\n");

Type t = cl.GetType();

ConstructorInfo[] ci = t.GetConstructors();

foreach (ConstructorInfo m in ci)

Console.WriteLine("Constructor: {0}", m.Name);

}

static void Main()

{

Console.SetWindowSize(80, 45);

var myClass = new Class1();

#region Вывод информации о типе

ListVariosStats(myClass); // Получаем разную информацию о Class1.

ListMethods(myClass); // Получаем информацию об Именах всех методов Class1.

ListFields(myClass); // Получаем информацию об Именах всех полей Class1.

ListProps(myClass); // Получаем список всех Свойств Class1.

ListInterfaces(myClass); // Получаем список всех Интерфейсов, поддерживаемых Class1.

ListConstructors(myClass); // Получаем информацию об Именах всех конструкторов Class1.

#endregion

#region Обращение к закрытым членам (Инкапсуляция... Что такое инкапсуляция? :)

Console.WriteLine(new string('-', 60));

Type type = myClass.GetType();

//Вызов private метода

MethodInfo methodC = type.GetMethod("MethodC", BindingFlags.Instance | BindingFlags.NonPublic);

methodC.Invoke(myClass, new object[] { "Hello", " world!" });

// Запись значения в private поле

FieldInfo mystring = type.GetField("mystring", BindingFlags.Instance | BindingFlags.NonPublic);

mystring.SetValue(myClass, "Привет Мир!");

Console.WriteLine(myClass.MyString);

#endregion

Assembly (сборка)

Ранее связывание

Чтобы подключить dll нужно выбрать references и в solution подключить ссылку на проект – это если проэкты находятся в одном солюшине.

Другой способ – добавить dll через browse. DLL будет находится в папке с проектом.

Позднее связывание

static void Main()

{

Console.SetWindowSize(80, 50);

// При помощи класса Assembly - можно динамически загружать сборки,

// обращаться к членам класса в процессе выполнения (ПОЗДНЕЕ СВЯЗЫВАНИЕ),

// а так же получать информацию о самой сборке.

Assembly assembly = null;

try

{

// Assembly.Load() - метод для загрузки сборки.

assembly = Assembly.Load("CarLibrary");

Console.WriteLine("Сборка CarLibrary - успешно загружена.");

}

catch (FileNotFoundException ex)

{

Console.WriteLine(ex.Message);

}

// Выводим информацию о всех типах в сборке.

ListAllTypes(assembly);

// Выводим информацию о всех членах в классе.

ListAllMembers(assembly);

// Выводим информацию о всех параметрах метода.

GetParams(assembly);

//Задержка.

Console.ReadKey();

}

// Метод для получения информации о всех типах в сборке.

private static void ListAllTypes(Assembly assembly)

{

Console.WriteLine(new string('\_', 80));

Console.WriteLine("\nТипы в: {0} \n", assembly.FullName);

Type[] types = assembly.GetTypes();

foreach (Type t in types)

Console.WriteLine("Тип: {0}", t);

}

// Метод для получения информации о членах класса.

private static void ListAllMembers(Assembly assembly)

{

Console.WriteLine(new string('\_', 80));

Type type = assembly.GetType("CarLibrary.MiniVan");

Console.WriteLine("\nЧлены класса: {0} \n", type);

MemberInfo[] members = type.GetMembers();

foreach (MemberInfo element in members)

Console.WriteLine("{0,-15}: {1}", element.MemberType, element);

}

// Получаем информацию о параметрах для метода TurboBoost()

private static void GetParams(Assembly assembly)

{

Console.WriteLine(new string('\_', 80));

Type type = assembly.GetType("CarLibrary.MiniVan");

MethodInfo method = type.GetMethod("Driver"); // Equals , Acceleration, Driver

// Вывод информации о количестве параметров.

Console.WriteLine("\nИнформация о параметрах для метода {0}", method.Name);

ParameterInfo[] parameters = method.GetParameters();

Console.WriteLine("Метод имеет " + parameters.Length + " параметров");

// Выводим некоторые характеристики каждого из параметров.

foreach (ParameterInfo parameter in parameters)

{

Console.WriteLine("Имя параметра: {0}", parameter.Name);

Console.WriteLine("Позиция в методе: {0}", parameter.Position);

Console.WriteLine("Тип параметра: {0}", parameter.ParameterType);

}

}

Вызов методов при позднем связывании

// Создаем объект выбранного нами типа "на лету".

// Позднее связывание - это технология которая позволяет обнаруживать типы,

// определять их имена и члены непосредственно в процессе выполнения.

// Раннее связывание - все указанные выше операции выполняются во время компиляции.

Assembly assembly = null;

try

{

assembly = Assembly.Load("CarLibrary");

}

catch (FileNotFoundException e)

{

Console.WriteLine(e.Message);

}

// Создание экземпляра класса MiniVan "на лету"

// При помощи класса Activator производится позднее связывание.

// Метод CreateInstance() - предназначен для создания экземпляра типа во время выполнения.

Type type = assembly.GetType("CarLibrary.MiniVan");

object instance = Activator.CreateInstance(type);

// Получаем экземпляр класса MethodInfo для метода Acceleration().

MethodInfo method = type.GetMethod("Acceleration");

// Вызов метода Acceleration().

// Первый параметр - ссылка на экземпляр для которого будет вызван метод Acceleration

// Второй параметр - массив аргументов метода Acceleration (в данном случае без параметров - null)

method.Invoke(instance, null);

// Получаем экземпляр класса MethodInfo для метода Driver().

method = type.GetMethod("Driver");

// Массив параметров для метода Driver("Shumaher", 36).

object[] parameters = { "Shumaher", 36 };

// Вызов метода Driver().

// Первый параметр - ссылка на экземпляр для которого будет вызван метод Acceleration

// Второй параметр - массив аргументов метода Acceleration (в данном случае - name:"Shumaher", age:36 )

method.Invoke(instance, parameters);

Dynamic рефлексия

Assembly assembly = null;

try

{

assembly = Assembly.Load("CarLibrary");

Type type = assembly.GetType("CarLibrary.MiniVan");

dynamic carInstance = Activator.CreateInstance(type);

carInstance.Acceleration();

carInstance.Driver("Shumaher", 26);

}

catch (FileNotFoundException e)

{

Console.WriteLine(e.Message);

}

Атрибуты

Существуетдватипаатрибутов:

Предопределенныеатрибуты(идутвпоставкеFCL),ипользовательскиеатрибуты, создаваемыепользователемдлядобавлениявкоддополнительныхсведений.Сточки зренияразработчикаобатипаимеютодинаковыйсинтаксис.

Пользовательскийатрибутпредставляетсобойобычныйкласс,унаследованныйоткласса Attribute.

Такойатрибутможетбытьиспользовандлялюбогометода, свойства,класса или сборки прииспользованииследующегосинтаксиса:

Правила создания пользовательского атрибута:

* + 1.Имя атрибутадолжно содержать суффикс Attribute *(Данное правило является рекомендацией, но его лучше придерживаться)*
  + 2.Класс-атрибут обязан наследоваться от системного класса System.Attribute

3.Класс-атрибут обязан быть декорирован атрибутом [AttributeUsageAttribute]

// Атрибуты. (Аспектно-Ориентированное Программирование)

// Атрибуты могут использоваться перед объявлениями типов, членов класса, членов интерфейса и перечислений.

// Виды атрибутов: Предопределенные и Пользовательские.

// [ ....... ] - секция атрибута.

// Объявление атрибута состоит из Имени атрибута и набора Параметров.

// Для атрибутов определяют два вида параметров - Позиционные и Именованные.

// Именованные параметры всегда располагаются после Позиционных.

namespace AttributeWork

{

// Для создания атрибута необходимо создать класс производный от класса : System.Attribute.

// Атрибут - [AttributeUsage] - задает свойства пользовательских атрибутов.

// Позиционный параметр - AttributeTargets, определяет элементы программы, для которых атрибут может быть применен.

// Параметр - AttributeTargets.All - позволяет использовать атрибут - MyAttribute, для любого элемента.

// Именованный параметр - AllowMultiple = false, определяет сколько раз к одному элементу можно применять атрибут.

[AttributeUsage(AttributeTargets.All, AllowMultiple = false)]

class MyAttribute : System.Attribute

{

private readonly string date;

public string Date

{

get { return date; }

}

// Позиционные параметры задаются формальными параметрами - public конструктора, класса атрибута.

public MyAttribute(string date)

{

this.date = date;

}

// Именованные параметры задаются открытыми нестатическими полями или свойствами, класса атрибута.

public int Number { get; set; }

}

}

[My("1/31/2008", Number = 1)]

public class MyClass

{

[MyAttribute("2/31/2007", Number = 2)]

public static void Method()

{

Console.WriteLine("MyClass.Method()\n");

}

}

class Program

{

static void Main()

{

MyClass my = new MyClass();

MyClass.Method();

Сериализация

Сериализацияпредставляетсобойпроцесспреобразованияобъектав потокбайтовсцельюсохраненияеговпамяти,вбазеданныхилив файле.

Основноеназначениесериализации—сохранитьсостояниеобъекта длятого,чтобыиметьвозможностьвоссоздатьеговслучае необходимости.

Процессобратныйсериализацииназываетсядесериализацией.

Serializable–отмечаетэлементдлясериализации.

NonSerializable–сериализациядляэлементавыполняться небудет.

OptionalField–используетсядлябезопаснойсериализации измененныхэлементов.

Для того, чтобы сериализоватьобъект в формате XML необходимо выполнитьследующие действия:

* + 1.Объявитькласскакоткрытый.
  + 2.Объявитьвсе членыкласса, которыенеобходимосериализовать, как открытые.
  + 3.Создатьконструктор не принимающийпараметров.

Сериализация XML

Сериализуются толбко public елементы. Нужно делать конструктор по умолчанию

// XmlRoot - Переименовывает рутовую ноду.

[XmlRoot("MyButton")]

public class MyClass

{

private string id = "button";

private int size = 10;

private Point position = new Point(20, 30);

private string password = "1234567890\_password";

private List<string> items = new List<string>();

// XML атрибут переименовываем и делаем атрибутом.

[XmlAttribute("SerialID")]

public string ID

{

get { return id; }

set { id = value; }

}

//XML атрибут.

[XmlAttribute("Length")]

public int Size

{

get { return size; }

set { size = value; }

}

//XML элемент.

[XmlElement("Pos")]

public Point Position

{

get { return position; }

set { position = value; }

}

// Исключаем свойство из процесса сериализации/десериализации.

[XmlIgnore]

public string Password

{

get { return password; }

set { password = value; }

}

// Характеристика массива.

[XmlArray("List")]

// Характеристика каждого элемента массива.

[XmlArrayItem("Element")]

public List<string> Items

{

get { return items; }

set { items = value; }

}

}

public Form1()

{

InitializeComponent();

}

// Тип для Сериализации и Десериализации.

readonly XmlSerializer serializer = new XmlSerializer(typeof(MyClass));

MyClass instance1 = new MyClass();

MyClass instance2;

// СЕРИАЛИЗАЦИЯ.

private void SerializeButton\_Click(object sender, EventArgs e)

{

for (int i = 0; i < 10; i++)

{

instance1.Items.Add("Element " + i); // Заполнение списка.

}

using (var stream = new FileStream("Serialization.xml", FileMode.Create, FileAccess.Write, FileShare.Read))

{

// Сохраняем объект в XML-файле на диске(СЕРИАЛИЗАЦИЯ).

serializer.Serialize(stream, instance1);

this.Text = "Объект сериализован!";

}

}

// ДЕСЕРИАЛИЗАЦИЯ.

private void DeserializeButton\_Click(object sender, EventArgs e)

{

try

{

using (var stream = new FileStream("Serialization.xml", FileMode.Open, FileAccess.Read, FileShare.Read))

{

// Восстанавливаем объект из XML-файла.

instance2 = serializer.Deserialize(stream) as MyClass;

this.Text = "Объект Десериализован!";

}

textBox.Text = "ID : " + instance2.ID + Environment.NewLine +

"Size : " + instance2.Size + Environment.NewLine +

"Position : " + instance2.Position.ToString() + Environment.NewLine +

"List : " + Environment.NewLine;

foreach (string item in instance2.Items)

{

textBox.Text += "\t" + item + Environment.NewLine;

}

textBox.Text += "Password: " + instance2.Password;

}

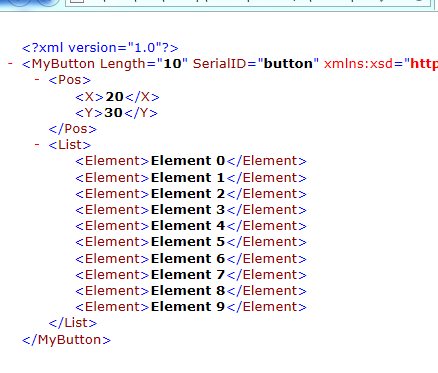
catch (Exception ex)

{

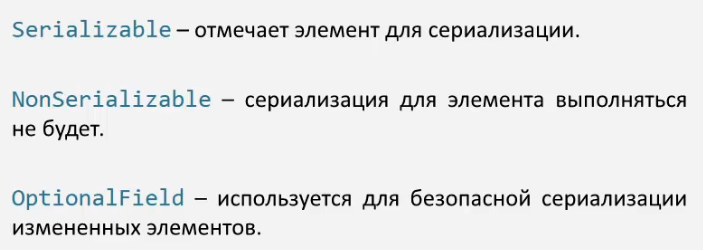
MessageBox.Show(ex.Message);

}

}



Сериализация в двоичный формат



public class Mersedes : Car

{

// Два режима работы - Спорт и Люкс.

protected Mode mode;

public Mersedes(string name, int speed, Mode mode)

: base(name, speed)

{

this.mode = mode;

}

public void SetMode(Mode mode)

{

this.mode = mode;

Console.WriteLine(this.mode);

}

public void ShowMode()

{

Console.WriteLine(this.mode);

}

}

// Класс Radio будет доступен для сериализации.

[Serializable]

public class Radio

{

[NonSerialized]

private int id = 9;

// Метод включения/выключения радио.

public void OnOff(bool state)

{

Console.WriteLine(state ? "Радио Включено." : "Радио Выключено.");

}

}

Mersedes auto = new Mersedes("G500", 250, Mode.Lux);

auto.TurnOnRadio(true);

auto.ShowMode();

FileStream stream = File.Create("CarData.dat");

// Помещаем объектный граф (для базовых типов) в поток в двоичном формате.

BinaryFormatter formatter = new BinaryFormatter();

// Cериализация.

formatter.Serialize(stream, auto);

stream.Close();

stream = File.OpenRead("CarData.dat");

// Десериализация.

auto = formatter.Deserialize(stream) as Mersedes;

Console.WriteLine("Имя : " + auto.Name);

Console.WriteLine("Скорость: " + auto.Speed);

auto.TurnOnRadio(false);

stream.Close();

\*\*\*

ISerializable

public class Car : ISerializable

{

public string name;

public int speed;

public Car(string name, int speed)

{

this.name = name;

this.speed = speed;

}

// Специальный вариант конструктора.

// SerializationInfo - объект в который помещаем все пары имя-значение представляющие состояние объекта.

// SerializationInfo - мешок со свойствами (property bag)

private Car(SerializationInfo propertyBag, StreamingContext context)

{

// Значение All перечисления StreamingContextState для свойства context.State, указывает,

// что данные могут быть переданы в любое место или получены из любого места.

Console.WriteLine("[ctor] ContextState: {0}", context.State.ToString());

// Из мешка со свойствами извлекаем значения свойств помещеенных ранее в методе GetObjectData()

name = propertyBag.GetString("name");

speed = propertyBag.GetInt32("speed");

}

// Метод ISerializable.GetObjectData() вызывается Formatter-ом

void ISerializable.GetObjectData(SerializationInfo propertyBag, StreamingContext context)

{

// Значение All перечисления StreamingContextState свойства context.State, указывает,

// что данные могут быть переданы в любое место или получены из любого места.

Console.WriteLine("[GetObjectData] ContextState: {0}", context.State.ToString());

// В мешок со свойствами добавляем два свойства и соответственно значения для них.

propertyBag.AddValue("name", name);

propertyBag.AddValue("speed", speed);

}

}

Car car = new Car("Mersedes-Benz", 250);

Stream stream = File.Create("CarData.dat");

BinaryFormatter formatter = new BinaryFormatter();

// Сериализация (Вызов метода ISerializable.GetObjectData()).

formatter.Serialize(stream, car);

stream.Close();

stream = File.OpenRead("CarData.dat");

// Десериализация (Вызов спецконструктора).

car = formatter.Deserialize(stream) as Car;

Console.WriteLine(car.name + "\n" + car.speed);

\*\*\*

DB

public partial class MainForm : Form

{

List<Car> cars = null;

public MainForm()

{

InitializeComponent();

CenterToScreen();

// Добавляем несколько объектов Car в массив

cars = new List<Car>()

{

new Car("Siddhartha", "BMW", "Silver"),

new Car("Chucky", "Caravan", "Green"),

new Car("Fred", "Audi TT", "Red")

};

// Обновляем DataGrid

UpdateGrid();

}

private void UpdateGrid()

{

if (cars != null)

{

// Создаем таблицу DataTable с именем Inventory

var table = new DataTable("Inventory");

// Создаем объекты DataColumn

var make = new DataColumn("Make");

var name = new DataColumn("Name");

var color = new DataColumn("Color");

// Добавляем объекты DataColumn в DataTable

table.Columns.Add(name);

table.Columns.Add(make);

table.Columns.Add(color);

// Для каждого элемента создаем строку в таблице

foreach (Car car in cars)

{

DataRow row = table.NewRow();

row["Name"] = car.name;

row["Make"] = car.make;

row["Color"] = car.color;

table.Rows.Add(row);

}

// Биндим DataTable к dataGridView

dataGridView.DataSource = table;

}

}

private void makeNewCarToolStripMenuItem\_Click(object sender, EventArgs e)

{

var dialog = new FormDialog();

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

{

cars.Add(dialog.car);

UpdateGrid();

}

}

private void saveCarFileToolStripMenuItem\_Click(object sender, EventArgs e)

{

// Настраиваем свойства диалогового окна для сохранения файлов

// SaveFileDialog mySaveFileDialog = new SaveFileDialog();

saveFileDialog.InitialDirectory = ".";

saveFileDialog.Filter = "car files (\*.car)|\*.car|All files(\*.\*)|\*.\*";

saveFileDialog.FilterIndex = 1;

saveFileDialog.RestoreDirectory = true;

saveFileDialog.FileName = "carDoc";

// Сохраняем объекты автомобилей

if (saveFileDialog.ShowDialog() == DialogResult.OK)

{

Stream myStream = saveFileDialog.OpenFile();

var serializer = new BinaryFormatter();

serializer.Serialize(myStream, cars);

myStream.Close();

}

}

private void openCarFileToolStripMenuItem\_Click(object sender, EventArgs e)

{

// Настраиваем свойства диалогового окна для открытия файлов

openFileDialog.InitialDirectory = ".";

openFileDialog.Filter = "car files (\*.car)|\*.car|All files(\*.\*)|\*.\*";

openFileDialog.FilterIndex = 1;

openFileDialog.RestoreDirectory = true;

// Восстанавливаем объекты автомобилей

if (openFileDialog.ShowDialog() == DialogResult.OK)

{

// Очищаем текущий массив

cars.Clear();

Stream stream = openFileDialog.OpenFile();

var deserializer = new BinaryFormatter();

cars = deserializer.Deserialize(stream) as List<Car>;

stream.Close();

UpdateGrid();

}

}

private void exitToolStripMenuItem\_Click(object sender, EventArgs e)

{

Application.Exit();

}

private void clearAllCarsToolStripMenuItem\_Click(object sender, EventArgs e)

{

cars.Clear();

UpdateGrid();

}

}

\*\*\*

IDeserializtionCallback

Поле тотал не сереализовалось, но при десериализации был вызван метод который его подсчитал

[Serializable]

class ShoppingCartItem : IDeserializationCallback

{

public int productId;

public decimal price;

public int quantity;

[NonSerialized]

public decimal total;

public ShoppingCartItem(int \_productId, decimal \_price, int \_quantity)

{

productId = \_productId;

price = \_price;

quantity = \_quantity;

//total = price \* quantity;

}

void IDeserializationCallback.OnDeserialization(object sender)

{

// После десериализации подсчитываем общую стоимость.

total = price \* quantity;

}

}

static class Program

{

static void Main()

{

var item = new ShoppingCartItem(2534681, 50000, 5);

#region Сериализация

var stream = new FileStream("SerializedCar.txt", FileMode.Create);

var formatter = new BinaryFormatter();

formatter.Serialize(stream, item);

stream.Close();

#endregion

#region Десериализвция

stream = new FileStream("SerializedCar.txt", FileMode.Open);

formatter = new BinaryFormatter();

item = (ShoppingCartItem)formatter.Deserialize(stream);

stream.Close();

// Отображаем десериализованную строку.

Console.WriteLine("Total : {0}", item.total);

#endregion

// Delay.

Console.ReadKey();

}

\*\*\*

OptionalField

При десериализации в объект с новыми свойствами инициализирует поле, чтобы не было exception

// Поле добавленное в класс в новой версии. Инициализируется значением по умолчанию.

[OptionalField]

public bool taxable;

\*\*\*

XmlSerializer

/\*

Чтобы создать класс, который можно сериализовать в формат XML, нужно выполнить следующее:

- объявить класс как открытый (public)

- объявить все члены, которые нужно сериализовать, как открытые (public)

- создать конструктор по умолчанию (без параметров)

\*/

namespace ClassSerializationXML

{

public class ShoppingCatItem

{

public Int32 productId;

public decimal price;

public Int32 quantity;

public decimal total;

}

class Program

{

static void Main()

{

var item = new ShoppingCatItem {productId = 22, price = 50000, quantity = 2};

item.total = item.quantity \* item.price;

// Создаем файл для сохранения данных

FileStream stream = new FileStream("SerializedClass.xml", FileMode.Create);

// Создаем объект XmlSerializer для выполнения сериализации

XmlSerializer serializer = new XmlSerializer(typeof(ShoppingCatItem));

// Используем объект XmlSerializer для сериализации данных в файл

serializer.Serialize(stream, item);

// Закрываем файл

stream.Close();

}

}

}

\*\*\*

SerizlizeDataSet

public static void SerializeDataSet(string fileName)

{

// Создаем DataSet; добавляем таблицу, столбец и десять строк.

DataSet dataSet = new DataSet("myDataSet");

DataTable table = new DataTable("table1");

DataColumn column = new DataColumn("thing");

table.Columns.Add(column);

dataSet.Tables.Add(table);

DataRow row;

for (int i = 0; i < 10; i++)

{

row = table.NewRow();

row[0] = "Thing " + i;

table.Rows.Add(row);

}

dataSet.WriteXml(fileName);

dataSet.WriteXmlSchema(Path.ChangeExtension(fileName, "xsd"));

}

static void Main()

{

SerializeDataSet("SerializedDataSet.xml");

}

\*\*\*

Пользовательскач сериализация

class ShoppingCartItem : ISerializable

{

public Int32 productId;

public decimal price;

public Int32 quantity;

[NonSerialized]

public decimal total;

// Конструктор, для начального создания экземпляра.

public ShoppingCartItem(int \_productId, decimal \_price, int \_quantity)

{

productId = \_productId;

price = \_price;

quantity = \_quantity;

total = price \* quantity;

}

// Специальный конструктор для десериализации.

protected ShoppingCartItem(SerializationInfo propertyBag, StreamingContext context)

{

productId = propertyBag.GetInt32("Product Id");

price = propertyBag.GetDecimal("Price");

quantity = propertyBag.GetInt32("Quantity");

total = price \* quantity;

}

// Метод, вызываемый во время сериализации.

public virtual void GetObjectData(SerializationInfo propertyBag, StreamingContext context)

{

propertyBag.AddValue("Product Id", productId);

propertyBag.AddValue("Price", price);

propertyBag.AddValue("Quantity", quantity);

}

public override string ToString()

{

return "id: " + productId + ": " + price + " x " + quantity + " = " + total;

}

}

class Program

{

static void Main()

{

FileStream stream;

BinaryFormatter formatter = new BinaryFormatter();

ShoppingCartItem item = new ShoppingCartItem(22, 50000, 2);

// Сериализация

stream = new FileStream("SerializedItem.txt", FileMode.Create);

formatter.Serialize(stream, item);

stream.Close();

// Десериализация

stream = new FileStream("SerializedItem.txt", FileMode.Open);

item = (ShoppingCartItem)formatter.Deserialize(stream);

stream.Close();

// Отображение значений полей.

Console.WriteLine(item);

// Delay.

Console.ReadKey();

}

\*\*\*

OnSerializing (До выполнения сериализации)

[Serializable]

class ShoppingCartItem

{

public Int32 productId;

public decimal price;

public Int32 quantity;

public decimal total;

public ShoppingCartItem(int productId, decimal price, int quantity)

{

this.productId = productId;

this.price = price;

this.quantity = quantity;

this.total = price \* quantity;

}

// Используйте OnSerializingAttribute, чтобы выполнять действия с объектом до выполнения сериализации.

// Чтобы использовать OnSerializingAttribute, метод должен содержать параметр StreamingContext.

[OnSerializing]

void CalculateTotal(StreamingContext context)

{

total = price \* quantity;

Console.WriteLine("Total = {0}", total);

}

// Используйте OnDeserializedAttribute, чтобы задать значения по умолчанию непосредственно перед десериализацией.

// Например, если десериализуемый тип не содержит конструктор, создайте метод,

// чтобы задать значения любых полей в экземпляре и применить атрибут к методу.

// Чтобы использовать OnDeserializedAttribute, метод должен содержать параметр StreamingContext.

[OnDeserialized]

void CheckTotal(StreamingContext context)

{

if (total == 0)

CalculateTotal(context);

else

Console.WriteLine("Total != 0");

}

public override string ToString()

{

return productId + ": " + price + " x " + quantity + " = " + total;

}

}

class Program

{

static void Main()

{

FileStream stream;

BinaryFormatter formatter = new BinaryFormatter();

ShoppingCartItem item = new ShoppingCartItem(22, 50000, 2);

// Сериализация

stream = new FileStream("SerializedItem.txt", FileMode.Create);

formatter.Serialize(stream, item);

stream.Close();

// Десериализация

stream = new FileStream("SerializedItem.txt", FileMode.Open);

item = (ShoppingCartItem)formatter.Deserialize(stream);

stream.Close();

// Отображение значений полей.

Console.WriteLine(item.ToString());

// Delay.

Console.ReadKey();

}

\*\*\*

Сборщик мусора

NextObjPtr – определяет где в куче будет выделена память для следуйщего объекта.

Есть две кучи – для мылых оьъектоа и больших.

Сборщик мусора проверяет 3 поколения.

В режиме роботы release при вызове метода Collect() оьъект будет удален, потому что он не видет дальнейшего использования ссылки

public class ResourceWrapper

{

~ResourceWrapper()

{

for (int i = 0; i < 1000; i++)

Console.Write(".");

}

}

class Program

{

static void Main()

{

ResourceWrapper resource = // Так как имеется сильная ссылка, сразу финализация не происходит.

new ResourceWrapper();

GC.Collect();

// resource = null;

Console.WriteLine("\n\nНажмите любую клавишу для завершения работы");

Console.WriteLine("и вызова (Деструктора) Finalize() сборщиком мусора");

Console.WriteLine("для объектов предусматривающих финализацию в AppDomain.");

// Задержка.

Console.ReadKey();

}

\*\*\*

static void Main()

{

var timer = new Timer(Method, "Hello", 0, 200);

Console.ReadLine();

//Продлить время роботы таймера

//timer.Dispose();

}

static void Method(object state)

{

Console.WriteLine(state);

GC.Collect();

}

}

\*\*\*

IDisposable

public class MyClass : IDisposable

{

// Пользователь объекта должен вызвать этот метод

// перед завершением работы с объектом.

public void Dispose()

{

// Освобождение неуправляемых ресурсов.

// Освобождение других содержащихся объектов.

// Например закрытие соединения с базой данных.

Console.WriteLine("Метод Dispose() отработал:"+this.GetHashCode());

}

// Деструктор.

~MyClass()

{

for (int i = 0; i < 10; i++)

Console.Write(".");

}

}

class Program

{

static void Main()

{

MyClass instance = new MyClass();

if (instance is IDisposable)

instance.Dispose();

instance.Dispose();

instance.Dispose(); instance.Dispose();

instance.Dispose();

instance.Dispose();

Console.WriteLine(new string('\_', 30));

// Dispose() вызывается автоматически при выходе за пределы области видимости using.

// Если возникает исключение внутри блока using то Dispose() гарантированно вызовется.

using (instance = new MyClass())

{

// Использование instance.

}

// Задержка.

Console.ReadKey();

}

}

\*\*\*

Паттерн финализации ресурсов

Паттерн предложеный microsoft

public class ResourceWrapper : IDisposable

{

// Флаг показывающий вызов метода Dispose()

private bool disposed = false;

public void Dispose()

{

// Вызов вспомогательного метода.

// Если true, значит очистку инициировал пользователь объекта.

CleanUp(true);

// SuppressFinalize() - устанавливает флаг запрещения завершения для объектов

// которые в противном случае могли бы быть завершены сборщиком мусора.

// Отменяет работу деструктора для данного класса.

GC.SuppressFinalize(this);

}

// Сборщик мусора вызывает Деструктор, если пользователь объекта забудет вызвать метод Dispose().

~ResourceWrapper()

{

Console.WriteLine("Finalise!!!!!!!!!!!");

CleanUp(false);

}

// Метод для избежания дублирования кода в Деструкторе и методе Dispose().

private void CleanUp(bool clean)

{

// Проверка, что ресурсы еще не освобождены.

if (!this.disposed)

{

// Если clean равно true, освободить все управляемые ресурсы.

if (clean)

{

Console.Write("Освобождение ресурсов");

for (int i = 0; i < 40; i++)

{

Console.Write("F");

}

}

Console.WriteLine("Finalized");

}

this.disposed = true;

}

}

using (var wrapper = new ResourceWrapper())

{

Console.WriteLine(wrapper);

}

Console.WriteLine(new string('-',20));

var wrapper2 = new ResourceWrapper();

Console.WriteLine(wrapper2);

wrapper2.Dispose();

wrapper2.Dispose();

wrapper2.Dispose();

wrapper2.Dispose();

var wrapper3 = new ResourceWrapper();

Console.ReadKey();

Console.WriteLine("Press any key to dispose");