**Переменные**

Переменная представляет именованную область памяти, в которой хранится значение определенного типа. Переменная имеет тип, имя и значение. Тип определяет, какого рода информацию может хранить переменная. Перед использованием любую переменную надо определить. Синтаксис определения переменной выглядит следующим образом:

*тип имя\_переменной*;

**Литералы**

Литералы представляют неизменяемые значения (иногда их еще называют константами). Литералы бывают логическими (true, false), целочисленными, вещественными, символьными и строчными. И отдельный литерал представляет ключевое слово null.

Специальную группу представляют *управляющие последовательности*, данная последовательность интерпретируется определенным образом. Наиболее часто используемые последовательности:

* '\n' - перевод строки
* '\t' - табуляция
* '\' - обратный слеш

**Типы данных**

* **bool:** хранит значение true или false (логические литералы).
* **byte**: хранит целое число от 0 до 255 и занимает 1 байт.
* **sbyte:** хранит целое число от -128 до 127 и занимает 1 байт.
* **short:** хранит целое число от -32768 до 32767 и занимает 2 байта.
* **ushort:** хранит целое число от 0 до 65535 и занимает 2 байта.
* **int:** хранит целое число от -2147483648 до 2147483647 и занимает 4 байта. Все целочисленные литералы по умолчанию представляют значения типа int:
* **uint:** хранит целое число от 0 до 4294967295 и занимает 4 байта.
* **long:** хранит целое число от –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807 и занимает 8 байт.
* **ulong:** хранит целое число от 0 до 18 446 744 073 709 551 615 и занимает 8 байт.
* **float:** хранит число с плавающей точкой от -3.4\*1038 до 3.4\*1038 и занимает 4 байта.
* **double:** хранит число с плавающей точкой от ±5.0\*10-324 до ±1.7\*10308 и занимает 8 байта.
* **decimal:** хранит десятичное дробное число. Если употребляется без десятичной запятой, имеет значение от ±1.0\*10-28 до ±7.9228\*1028, может хранить 28 знаков после запятой и занимает 16 байт.
* **char:** хранит одиночный символ в кодировке Unicode и занимает 2 байта.
* **string:** хранит набор символов Unicode.
* **object:** может хранить значение любого типа данных и занимает 4 байта на 32-разрядной платформе и 8 байт на 64-разрядной платформе.

Суффикс

Все вещественные литералы рассматриваются как значения типа double. И чтобы указать, что дробное число представляет тип float или тип decimal, необходимо к литералу добавлять суффикс: F/f - для float и M/m - для decimal.

*float a = 3.14F;*

*float b = 30.6f;*

*decimal c = 1005.8M;*

*decimal d = 334.8m;*

Все целочисленные литералы рассматриваются как значения типа int. Чтобы явным образом указать, что целочисленный литерал представляет значение типа uint, надо использовать суффикс U/u, для типа long - суффикс L/l, а для типа ulong - суффикс UL/ul:

*uint a = 10U;*

*long b = 20L;*

*ulong c = 30UL;*

**Неявная типизация**

Для неявной типизации вместо названия типа данных используется ключевое слово var. Нельзя сначала объявить неявно типизируемую переменную, а затем инициализировать. Нельзя указать в качестве значения неявно типизируемой переменной null.

**Арифметические операции**

Операции бывают унарными (выполняются над одним операндом), бинарными - над двумя операндами и тернарными - выполняются над тремя операндами.

Бинарные арифметические операции:

* + Операция сложения двух чисел:

*int x = 10;*

*int z = x + 12; // 22*

* - Операция вычитания двух чисел:

*int x = 10;*

*int z = x - 6; // 4*

* Операция умножения двух чисел:

*int x = 10;*

*int z = x \* 5; // 50*

* / операция деления двух чисел:

*int x = 10;*

*int z = x / 5; // 2*

*double z = 10 / 4; //результат равен 2*

*double z = 10.0 / 4.0; //результат равен 2.5*

* % Операция получение остатка от целочисленного деления двух чисел:

*double x = 10.0;*

*double z = x % 4.0; //результат равен 2*

Также есть ряд унарных операций, в которых принимает участие один операнд:

* ++ Операция инкремента.

Инкремент бывает префиксным: ++x - сначала значение переменной x увеличивается на 1, а потом ее значение возвращается в качестве результата операции. И также существует постфиксный инкремент: x++ - сначала значение переменной x возвращается в качестве результата операции, а затем к нему прибавляется 1.

* -- Операция декремента или уменьшения значения на единицу.

Также существует префиксная форма декремента (--x) и постфиксная (x--).

**Поразрядные операции**

***Логические операции***

* &(логическое умножение)

Умножение производится поразрядно, и если у обоих операндов значения разрядов равно 1, то операция возвращает 1, иначе возвращается число 0.

* | (логическое сложение)

Похоже на логическое умножение, операция также производится по двоичным разрядам, но теперь возвращается единица, если хотя бы у одного числа в данном разряде имеется единица.

* ^ (логическое исключающее ИЛИ)

Также эту операцию называют XOR, нередко ее применяют для простого шифрования:

Здесь опять же производятся поразрядные операции. Если у нас значения текущего разряда у обоих чисел разные, то возвращается 1, иначе возвращается 0.

* ~ (логическое отрицание или инверсия)

Еще одна поразрядная операция, которая инвертирует все разряды: если значение разряда равно 1, то оно становится равным нулю, и наоборот.

***Операции сдвига***

* x<<y - сдвигает число x влево на y разрядов.   
  Например, 4<<1 сдвигает число 4 (которое в двоичном представлении 100) на один разряд влево, то есть в итоге получается 1000 или число 8 в десятичном представлении.
* x>>y - сдвигает число x вправо на y разрядов.   
  Например, 16>>1 сдвигает число 16 (которое в двоичном представлении 10000) на один разряд вправо, то есть в итоге получается 1000 или число 8 в десятичном представлении.

Таким образом, если исходное число, которое надо сдвинуть в ту или другую строну, делится на два, то фактически получается умножение или деление на два. Поэтому подобную операцию можно использовать вместо непосредственного умножения или деления на два.

***Операции присваивания***

* +=: присваивание после сложения. A = A + B
* -=: присваивание после вычитания A = A - B
* \*=: присваивание после умножения. A = A \* B
* /=: присваивание после деления. A = A / B
* %=: присваивание после деления по модулю. A = A % B
* &=: присваивание после поразрядной конъюнкции. A = A & B
* |=: присваивание после поразрядной дизъюнкции. A = A | B
* ^=: присваивание после операции исключающего ИЛИ. A = A ^ B
* <<=: присваивание после сдвига разрядов влево A = A << B
* >>=: присваивание после сдвига разрядов вправо. A = A >> B

***Операции сравнения***

В операциях сравнения сравниваются два операнда и возвращается значение типа bool - true, если выражение верно, и false, если выражение неверно.

* == Сравнивает два операнда на равенство. Если они равны, то операция возвращает true, если не равны, то возвращается false:
* != Сравнивает два операнда и возвращает true, если операнды не равны, и false, если они равны.
* < Операция "меньше чем". Возвращает true, если первый операнд меньше второго, и false, если первый операнд больше второго:
* > Операция "больше чем". Сравнивает два операнда и возвращает true, если первый операнд больше второго, иначе возвращает false:
* <= Операция "меньше или равно". Сравнивает два операнда и возвращает true, если первый операнд меньше или равен второму. Иначе возвращает false.
* >= Операция "больше или равно". Сравнивает два операнда и возвращает true, если первый операнд больше или равен второму, иначе возвращается false:

Операции <, > <=, >= имеют больший приоритет, чем == и !=.

***Логические операции***

Также в C# определены логические операторы, которые также возвращают значение типа bool. В качестве операндов они принимают значения типа bool.

* | Операция логического сложения или логическое ИЛИ. Возвращает true, если хотя бы один из операндов возвращает true.
* & Операция логического умножения или логическое И. Возвращает true, если оба операнда одновременно равны true.
* || Операция логического сложения. Возвращает true, если хотя бы один из операндов возвращает true.
* && Операция логического умножения. Возвращает true, если оба операнда одновременно равны true.
* ! Операция логического отрицания. Производится над одним операндом и возвращает true, если операнд равен false. Если операнд равен true, то операция возвращает false:
* ^ Операция исключающего ИЛИ. Возвращает true, если либо первый, либо второй операнд (но не одновременно) равны true, иначе возвращает false

**Условные конструкции**

* **Конструкция if/else** проверяет истинность некоторого условия и в зависимости от результатов проверки выполняет определенный код:

*if(условие)*

*{…. }*

*else{….}*

* **Конструкция switch/case** аналогична конструкции if/else, так как позволяет обработать сразу несколько условий:

*switch (условие){*

*case "Y":*

*Console.WriteLine("Вы нажали букву Y");*

*break;*

*case "N":*

*Console.WriteLine("Вы нажали букву N");*

*break;*

*default:*

*Console.WriteLine("Вы нажали неизвестную букву");*

*break;*

*}*

В конце каждого блока сase должен ставиться один из операторов перехода: break, goto case, return или throw. Как правило, используется оператор break. При его применении другие блоки case выполняться не будут.

* **Тернарная операция** имеет следующий синтаксис: [первый операнд - условие] ? [второй операнд] : [третий операнд]. Здесь сразу три операнда. В зависимости от условия тернарная операция возвращает второй или третий операнд: если условие равно true, то возвращается второй операнд; если условие равно false, то третий.

*int z = selection=="+"? (x+y) : (x-y);*

**Циклы**

* **for**

*for ([инициализация счетчика]; [условие]; [изменение счетчика])*

*{*

*// действия*

*}*

*int i = 0;*

*for (; i<9;)*

*{*

*Console.WriteLine($"Квадрат числа {++i} равен {i \* i}");*

*}*

* **foreach** предназначен для перебора элементов в контейнерах, в том числе в массивах. Формальное объявление цикла foreach:

*foreach (тип\_данных название\_переменной in контейнер)*

*{*

*// действия*

*}*

Определенную сложность может представлять перебор многомерного массива. Прежде всего надо учитывать, что длина такого массива - это совокупное количество элементов*.*

*int[,] mas = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 }, { 10, 11, 12 } };*

*foreach (int i in mas)*

*Console.Write($"{i} ");*

*Console.WriteLine();*

* **while**

*int i = 6;*

*while (i > 0)*

*{*

*Console.WriteLine(i);*

*i--;*

*}*

* **do...while**

В цикле do сначала выполняется код цикла, а потом происходит проверка условия в инструкции while.

*int i = 6;*

*do*

*{*

*Console.WriteLine(i);*

*i--;*

*}*

*while (i > 0);*

***Операторы continue и break***

Иногда возникает ситуация, когда требуется выйти из цикла, не дожидаясь его завершения. В этом случае мы можем воспользоваться оператором *break*. Если мы хотим, чтобы при проверке цикл не завершался, а просто пропускал текущую итерацию можем воспользоваться оператором *continue*.

**Массивы**

Массив представляет набор однотипных данных. Объявление массива:

*тип\_переменной[] название\_массива;*

*int[] numbers;*

*int[] nums2 = new int[4] { 1, 2, 3, 5 };*

*int[] nums3 = new int[] { 1, 2, 3, 5 };*

*int[] nums4 = new[] { 1, 2, 3, 5 };*

*int[] nums5 = { 1, 2, 3, 5 };*

***Многомерные массивы***

*int[,] nums1;*

*int[,] nums2 = new int[2, 3];*

*int[,] nums3 = new int[2, 3] { { 0, 1, 2 }, { 3, 4, 5 } };*

*int[,] nums4 = new int[,] { { 0, 1, 2 }, { 3, 4, 5 } };*

*int[,] nums5 = new [,]{ { 0, 1, 2 }, { 3, 4, 5 } };*

*int[,] nums6 = { { 0, 1, 2 }, { 3, 4, 5 } };*

***Массив массивов***

*int[][] nums = new int[3][];*

*nums[0] = new int[2] { 1, 2 }; // выделяем память для первого подмассива*

*nums[1] = new int[3] { 1, 2, 3 }; // выделяем память для второго подмассива*

*nums[2] = new int[5] { 1, 2, 3, 4, 5 }; // выделяем память для третьего подмассива*

***Основные понятия массивов***

* Ранг (rank): количество измерений массива
* Длина измерения (dimension length): длина отдельного измерения массива
* Длина массива (array length): количество всех элементов массива

**Методы**

Метод - это именованный блок кода, который выполняет некоторые действия.

Общее определение методов выглядит следующим образом:

*[модификаторы] тип\_возвращаемого\_значения название\_метода ([параметры])*

*{*

*// тело метода*

*}*

Модификаторы и параметры необязательны. Консольная программа на языке C# должна содержать как минимум один метод - метод Main, который является точкой входа в приложение

*static void Main(string[] args)*

*{….}*

***Вызов методов***

Для вызова метода указывается его имя, после которого в скобках идут значения для его параметров (если метод принимает параметры).

*название\_метода (значения\_для\_параметров\_метода);*

***Возвращение значения***

Метод может возвращать значение, какой-либо результат. Методы с типом void не возвращают никакого значения. Они просто выполняют некоторые действия.

Если метод имеет любой другой тип, отличный от void, то такой метод обязан вернуть значение этого типа. Для этого применяется оператор return, после которого идет возвращаемое значение:

*return возвращаемое значение;*

Результат методов, который возвращают значение, мы можем присвоить переменным или использовать иным образом в программе*.*

***Выход из метода***

Мы можем использовать оператор return и в методах с типам void. В этом случае после оператора return не ставится никакого возвращаемого значения (ведь метод ничего не возвращает).

*static void SayHello()*

*{*

*int hour = 23;*

*if(hour > 22)*

*{*

*return;*

*}*

*else*

*{*

*Console.WriteLine("Hello");*

*}*

*}*

***Сокращенная запись методов***

*static void SayHello()*

Если метод в качестве тела определяет только одну инструкцию, то мы можем сократить определение метода.

*{*

*Console.WriteLine("Hello");*

*}*

*static void SayHello() => Console.WriteLine("Hello");*

*/\*---------------------------------------------------------------------\*/*

*static string GetHello()*

*{*

*return "hello";*

*}*

*static string GetHello() => "hello";*

***Параметры методов***

Параметры позволяют передать в метод некоторые входные данные. Формальные параметры - это собственно параметры метода (в данном случае x и y), а фактические параметры - значения, которые передаются формальным параметрам. То есть фактические параметры - это и есть аргументы метода.

***Необязательные параметры***

Для таких параметров нам необходимо объявить значение по умолчанию. Также следует учитывать, что после необязательных параметров все последующие параметры также должны быть необязательными:

*static int OptionalParam(int x, int y, int z=5, int s=4)*

*{*

*return x + y + z + s;*

*}*

***Именованные параметры***

В предыдущих примерах при вызове методов значения для параметров передавались в порядке объявления этих параметров в методе. Но мы можем нарушить подобный порядок, используя именованные параметры:

*OptionalParam(x:2, y:3);*

*OptionalParam(y:2, x:3, s:10);*

***Передача параметров по ссылке и значению.***

|  |  |
| --- | --- |
| Передача параметров по ссылке | Передача параметров по значению |
| При передаче параметров по ссылке перед параметрами используется модификатор ref. Модификатор ref указывается, как при объявлении метода, так и при его вызове | Это обычный способ передачи параметров |
| метод получает адрес переменной в памяти. И, таким образом, если в методе изменяется значение параметра, передаваемого по ссылке, то также изменяется и значение переменной, которая передается на его место. | метод получает не саму переменную, а ее копию. |
| *static void Addition(ref int x, int y)*  *{*  *x += y;*  *}*  *Addition(ref x, y);* | *static int Sum(int x, int y)*  *{*  *return x + y;*  *}*  *Sum(10, 15);* |

***Выходные параметры***

Чтобы сделать параметр выходным, перед ним ставится модификатор out. Здесь результат возвращается не через оператор return, а через выходной параметр. Ключевое слово out используется как при определении метода, так и при его вызове. Прелесть использования подобных параметров состоит в том, что по сути мы можем вернуть из метода не один вариант, а несколько. Методы, использующие такие параметры, обязательно должны присваивать им определенное значение.

*static void Sum(int x, int y, out int a)*

*{*

*a = x + y;*

*}*

***Входные параметры***

Чтобы сделать параметр входным, перед ним ставится модификатор in. Модификатор in указывает, что данный параметр будет передаваться в метод по ссылке, однако внутри метода его значение параметра нельзя будет изменить.

*static void GetData(in int x, int y, out int area, out int perim)*

*{*

*// x = x + 10; нельзя изменить значение параметра x*

*y = y + 10;*

*area = x \* y;*

*perim = (x + y) \* 2;*

*}*

|  |  |  |
| --- | --- | --- |
| ref | out | in |
| данный параметр будет передаваться в метод по ссылке | | |
| внутри метода его значение параметра можно изменить | внутри метода его значение параметра нужно изменить | внутри метода его значение параметра нельзя изменить |
| требуется инициализация | не требуется инициализация |  |

***Массив параметров и ключевое слово params***

|  |  |
| --- | --- |
| Массив | params |
| в качестве параметра передаётся массив. Массив может быть одномерным, двумерным, многомерным | можем передавать неопределенное количество параметров. При вызове метода на место параметра с модификатором params мы можем передать как отдельные значения, так и массив значений, либо вообще не передавать параметры. параметр с ключевым словом params при определении метода должен представлять одномерный массив того типа, данные которого мы собираемся использовать. |
| Если же нам надо передать какие- то другие параметры, то они могут указываться как удобно. | Если же нам надо передать какие- то другие параметры, то они должны указываться до параметра с ключевм словом params |

**Перечисления enum**

Перечисления представляют набор логически связанных констант. Объявление перечисления происходит с помощью оператора enum. Далее идет название перечисления, после которого указывается тип перечисления - он обязательно должен представлять целочисленный тип (byte, int, short, long). Если тип явным образом не указан, то по умолчанию используется тип int. Затем идет список элементов перечисления через запятую. Каждому элементу перечисления присваивается целочисленное значение, причем первый элемент будет иметь значение 0, второй - 1 и так далее. Мы можем также явным образом указать значения элементов, либо указав значение первого элемента. Каждое перечисление фактически определяет новый тип данных. Затем в программе мы можем определить переменную этого типа и использовать ее.

*enum Operation*

*{*

*Add = 2,*

*Subtract = 4,*

*Multiply = 8,*

*Divide = 16*

*}*

*enum Color*

*{*

*White = 1,*

*Black = 2,*

*Green = 2,*

*Blue = White // Blue = 1*

*}*

То есть несмотря на то, что каждая константа сопоставляется с определенным числом, мы не можем присвоить ей числовое значение, например, Operation op = 1;. И также если мы будем выводить на консоль значение этой переменной, то мы получим им константы, а не числовое значение. Если же необходимо получить числовое значение, то следует выполнить приведение к числовому типу.

**Кортежи**

Кортеж представляет набор значений, заключенных в круглые скобки:

*var tuple = (5, 10);*

*(string, int, double) person = ("Tom", 25, 81.23);*

Мы также можем дать названия полям кортежа:

*var tuple = (count:5, sum:10);*

*Console.WriteLine(tuple.count); // 5*

*Console.WriteLine(tuple.sum); // 10*

Кортежи могут передаваться в качестве параметров в метод, могут быть возвращаемым результатом функции, либо использоваться иным образом.

*private static (int, int) GetValues()*

*{*

*var result = (1, 3);*

*return result;*

*}*

**Строки**

В C# строки являются объектами. Следовательно, тип string относится к числу ссылочных. Самый простой способ построить символьную строку — воспользоваться строковым литералом.

*string str = "Пример строки";*

В данном случае переменная str инициализируется последовательностью символов "Пример строки". Объект типа string можно также создать из массива типа char.

*char[] chararray = {'e', 'x', 'a', 'm', 'p', 'l', 'e'};*

*string str = new string(chararray);*

Как ни странно, содержимое объекта типа string не подлежит изменению. Это означает, что однажды созданную последовательность символов изменить нельзя. Но данное ограничение способствует более эффективной реализации символьных строк. Поэтому этот, на первый взгляд, очевидный недостаток на самом деле превращается в преимущество. Так, если требуется строка в качестве разновидности уже имеющейся строки, то для этой цели следует создать новую строку, содержащую все необходимые изменения. А поскольку неиспользуемые строковые объекты автоматически собираются в "мусор", то о дальнейшей судьбе ненужных строк можно даже не беспокоиться.

Следует, однако, подчеркнуть, что переменные ссылки на строки (т.е. объекты типа string) подлежат изменению, а, следовательно, они могут ссылаться на другой объект. Но содержимое самого объекта типа string не меняется после его создания.

В классе String определено единственное поле:

*public static readonly string Empty;*

Поле Empty обозначает пустую строку, т.е. такую строку, которая не содержит символы. Этим оно отличается от пустой ссылки типа String, которая просто делается на несуществующий объект.

Помимо этого, в классе String определен единственный индексатор, доступный только для чтения:

*public char this[int index] { get; }*

И наконец, в классе String определено единственное свойство, доступное только для чтения:

*public int Length { get; }*

Свойство Length возвращает количество символов в строке.

когда оператор == применяется к ссылкам на объекты типа String, то на предмет равенства сравнивается содержимое самих строк. Это же относится и к оператору !=. Когда он применяется к ссылкам на объекты типа String, то на предмет неравенства сравнивается содержимое самих строк. В то же время другие операторы отношения, в том числе < и >=, сравнивают ссылки на объекты типа String таким же образом, как и на объекты других типов. А для того чтобы проверить, является ли одна строка больше другой, следует вызвать метод Compare(), определенный в классе String.

Основная функциональность класса String раскрывается через его методы, среди которых можно выделить следующие:

* Compare: сравнивает две строки с учетом текущей культуры (локали) пользователя
* CompareOrdinal: сравнивает две строки без учета локали
* Contains: определяет, содержится ли подстрока в строке
* Concat: соединяет строки
* CopyTo: копирует часть строки, начиная с определенного индекса в массив
* EndsWith: определяет, совпадает ли конец строки с подстрокой
* Format: форматирует строку
* IndexOf: находит индекс первого вхождения символа или подстроки в строке
* Insert: вставляет в строку подстроку
* Join: соединяет элементы массива строк
* LastIndexOf: находит индекс последнего вхождения символа или подстроки в строке
* Replace: замещает в строке символ или подстроку другим символом или подстрокой
* Split: разделяет одну строку на массив строк
* Substring: извлекает из строки подстроку, начиная с указанной позиции
* ToLower: переводит все символы строки в нижний регистр
* ToUpper: переводит все символы строки в верхний регистр
* Trim: удаляет начальные и конечные пробелы из строки

***StringBuilder***

Когда строка конструируется классом String, выделяется ровно столько памяти, сколько необходимо для хранения данной строки. Однако, в пространстве имен System.Text имеется класс StringBuilder, который поступает лучше и обычно выделяет больше памяти, чем нужно в данный момент. У вас, как разработчика, есть возможность указать, сколько именно памяти должен выделить StringBuilder, но если вы этого не сделаете, то будет выбран объем по умолчанию, который зависит от размера начального текста, инициализирующего экземпляр StringBuilder. Класс StringBuilder имеет два главных свойства:

Length, показывающее длину строки, содержащуюся в объекте в данный момент

Capacity, указывающее максимальную длину строки, которая может поместиться в выделенную для объекта память.

Любые модификации строки происходят внутри блока памяти, выделенного экземпляру StringBuilder. Это делает добавление подстрок и замену индивидуальных символов строки очень эффективными. Удаление или вставка подстрок неизбежно остаются менее эффективными, потому что при этих операциях приходится перемещать в памяти части строки. Выделять новую память и, возможно, полностью перемещать ее содержимое приходится только при выполнении ряда действий, которые приводят к превышению выделенной емкости строки. В дополнение к избыточной памяти, выделяемой изначально на основе экспериментов, StringBuilder имеет свойство удваивать свою емкость, когда происходит переполнение, а новое значение емкости не установлено явно.

|  |  |
| --- | --- |
| Метод | Назначение |
| Append() | Добавляет строку к текущей строке |
| AppendFormat() | Добавляет строку, сформированную в соответствии со спецификатором формата |
| Insert() | Вставляет подстроку в строку |
| Remove() | Удаляет символ из текущей строки |
| Replace() | Заменяет все вхождения символа другим символом или вхождения подстроки другой подстрокой |
| ToString() | Возвращает текущую строку в виде объекта System.String (переопределение метода класса System.Object) |

***Форматирование строк***

|  |  |  |  |
| --- | --- | --- | --- |
| Спецификатор | Применяется к | Значение | Пример |
| C | Числовым типам | Символ местной валюты | $835.50 (США)  £835.50 (Великобритания) 835.50р.(Россия) |
| D | Только к целочисленным типам | Обычное целое | 835 |
| E | Числовым типам | Экспоненциальная нотация | 8.35Е+002 |
| F | Числовым типам | С фиксированной десятичной точкой | 835.50 |
| G | Числовым типам | Обычные числа | 835.5 |
| N | Числовым типам | Формат чисел, принятый в данной местности | 4,384.50 (Великобритания/США)  4 384,50 (континентальная Европа) |
| P | Числовым типам | Процентная нотация | 835,000.00% |
| X | Только к целочисленным типам | Шестнадцатеричный формат | 1a1f |

***Регулярное выражение***

Регулярные выражения — это часть небольшой технологической области, невероятно широко используемой в огромном диапазоне программ. Регулярные выражения можно представить себе как мини-язык программирования, имеющий одно специфическое назначение: находить подстроки в больших строковых выражениях.

Главным преимуществом регулярных выражений является использование метасимволов — специальные символы, задающие команды, а также управляющие последовательности, которые работают подобно управляющим последовательностям C#. Это символы, предваренные знаком обратного слеша (\) и имеющие специальное назначение.

|  |  |  |  |
| --- | --- | --- | --- |
| Символ | Значение | Пример | Соответствует |
| Классы символов | | | |
| [...] | Любой из символов, указанных в скобках | [a-z] | В исходной строке может быть любой символ английского алфавита в нижнем регистре |
| [^...] | Любой из символов, не указанных в скобках | [^0-9] | В исходной строке может быть любой символ кроме цифр |
| . | Любой символ, кроме перевода строки или другого разделителя Unicode-строки |  |  |
| \w | Любой текстовый символ, не являющийся пробелом, символом табуляции и т.п. |  |  |
| \W | Любой символ, не являющийся текстовым символом |  |  |
| \s | Любой пробельный символ из набора Unicode |  |  |
| \S | Любой непробельный символ из набора Unicode. Обратите внимание, что символы \w и \S - это не одно и то же |  |  |
| \d | Любые ASCII-цифры. Эквивалентно [0-9] |  |  |
| \D | Любой символ, отличный от ASCII-цифр. Эквивалентно [^0-9] |  |  |
| Символы повторения | | | |
| {n,m} | Соответствует предшествующему шаблону, повторенному не менее n и не более m раз | s{2,4} | "Press", "ssl", "progressss" |
| {n,} | Соответствует предшествующему шаблону, повторенному n или более раз | s{1,} | "ssl" |
| {n} | Соответствует в точности n экземплярам предшествующего шаблона | s{2} | "Press", "ssl", но не "progressss" |
| ? | Соответствует нулю или одному экземпляру предшествующего шаблона; предшествующий шаблон является необязательным | Эквивалентно {0,1} |  |
| + | Соответствует одному или более экземплярам предшествующего шаблона | Эквивалентно {1,} |  |
| \* | Соответствует нулю или более экземплярам предшествующего шаблона | Эквивалентно {0,} |  |
| Символы регулярных выражений выбора | | | |
| | | Соответствует либо подвыражению слева, либо подвыражению справа (аналог логической операции ИЛИ). |  |  |
| (...) | Группировка. Группирует элементы в единое целое, которое может использоваться с символами \*, +, ?, | и т.п. Также запоминает символы, соответствующие этой группе для использования в последующих ссылках. |  |  |
| (?:...) | Только группировка. Группирует элементы в единое целое, но не запоминает символы, соответствующие этой группе. |  |  |
| Якорные символы регулярных выражений | | | |
| ^ | Соответствует началу строкового выражения или началу строки при многострочном поиске. | ^Hello | "Hello, world", но не "Ok, Hello world" т.к. в этой строке слово "Hello" находится не в начале |
| $ | Соответствует концу строкового выражения или концу строки при многострочном поиске. | Hello$ | "World, Hello" |
| \b | Соответствует границе слова, т.е. соответствует позиции между символом \w и символом \W или между символом \w и началом или концом строки. | \b(my)\b | В строке "Hello my world" выберет слово "my" |
| \B | Соответствует позиции, не являющейся границей слов. | \B(ld)\b | Соответствие найдется в слове "World", но не в слове "ld" |

**Классы**

Описанием объекта является класс, а объект представляет экземпляр этого класса. По сути класс представляет новый тип, который определяется пользователем. Класс определяется с помощью ключевого слова сlass:

*class Person*

*{*

*}*

Класс можно определять внутри пространства имен, вне пространства имен, внутри другого класса. Вся функциональность класса представлена его членами - полями (полями называются переменные класса), свойствами, методами, событиями.

***Конструкторы***

Конструкторы вызываются при создании нового объекта данного класса. Конструкторы выполняют инициализацию объекта.

* **Конструкторы по умолчанию.** Если в классе не определено ни одного конструктора, то для этого класса автоматически создается конструктор по умолчанию. Такой конструктор не имеет параметров и не имеет тела. Для создания объекта Person используется выражение new Person(). Оператор new выделяет память для объекта Person. И затем вызывается конструктор по умолчанию, который не принимает никаких параметров. В итоге после выполнения данного выражения в памяти будет выделен участок, где будут храниться все данные объекта Person. А переменная tom получит ссылку на созданный объект. Если конструктор не инициализирует значения переменных объекта, то они получают значения по умолчанию. Для переменных числовых типов это число 0, а для типа string и классов - это значение null.
* **Конструкторы без параметров.** Для класса Person

*public Person() { name = "Неизвестно"; age = 18; }*

*Person tom = new Person(); // вызов 1-ого конструктора без параметров*

* **Конструкторы с параметрами.** Для класса Person

*public Person(string n, int a) { name = n; age = a; }*

*Person sam = new Person("Sam", 25);*

***Ключевое слово this***

Ключевое слово this представляет ссылку на текущий экземпляр класса.

*class Person*

*{*

*public string name;*

*public int age;*

*public Person() : this("Неизвестно")*

*{*

*}*

*public Person(string name) : this(name, 18)*

*{*

*}*

*public Person(string name, int age)*

*{*

*this.name = name;*

*this.age = age;*

*}*

*public void GetInfo()*

*{*

*Console.WriteLine($"Имя: {name} Возраст: {age}");*

*}*

*}*

***Инициализаторы объектов***

Для инициализации объектов классов можно применять инициализаторы. Инициализаторы представляют передачу в фигурных скобках значений доступным полям и свойствам объекта:

*Person tom = new Person { name = "Tom", age=31 };*

При использовании инициализаторов следует учитывать следующие моменты:

* С помощью инициализатора мы можем установить значения только доступных из внешнего кода полей и свойств объекта. Например, в примере выше поля name и age имеют модификатор доступа public, поэтому они доступны из любой части программы.
* Инициализатор выполняется после конструктора, поэтому если и в конструкторе, и в инициализаторе устанавливаются значения одних и тех же полей и свойств, то значения, устанавливаемые в конструкторе, заменяются значениями из инициализатора.

***Типы значений и ссылочные типы***

Типы значений:

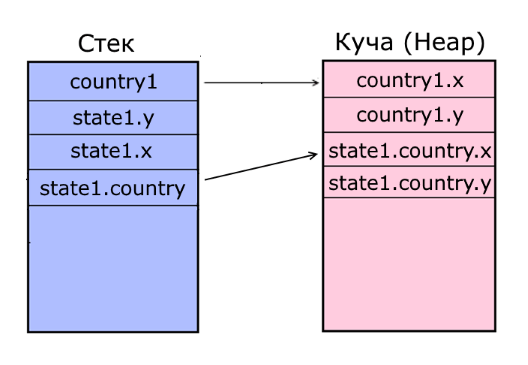
* Целочисленные типы (byte, sbyte, short, ushort, int, uint, long, ulong)
* Типы с плавающей запятой (float, double)
* Тип decimal
* Тип bool
* Тип char
* Перечисления enum
* Структуры (struct)

Ссылочные типы:

* Тип object
* Тип string
* Классы (class)
* Интерфейсы (interface)
* Делегаты (delegate)

Стек представляет собой структуру данных, которая растет снизу-вверх: каждый новый добавляемый элемент помещается поверх предыдущего. Параметры и переменные метода, которые представляют типы значений, размещают свое значение в стеке. Время жизни переменных таких типов ограничено их контекстом. Физически стек - это некоторая область памяти в адресном пространстве. Причем если параметр или переменная метода представляет тип значений, то в стеке будет храниться непосредственное значение этого параметра или переменной.

Ссылочные типы хранятся в куче или хипе, которую можно представить как неупорядоченный набор разнородных объектов. Физически это остальная часть памяти, которая доступна процессу. При создании объекта ссылочного типа в стеке помещается ссылка на адрес в куче (хипе). Когда объект ссылочного типа перестает использоваться, в дело вступает автоматический сборщик мусора: он видит, что на объект в хипе нету больше ссылок, условно удаляет этот объект и очищает память - фактически помечает, что данный сегмент памяти может быть использован для хранения других данных.



Где country это класс, а state это структура.

***Модификаторы доступа***

Все члены класса - поля, методы, свойства - все они имеют модификаторы доступа. Модификаторы доступа позволяют задать допустимую область видимости для членов класса. То есть модификаторы доступа определяют контекст, в котором можно употреблять данную переменную или метод.

В C# применяются следующие модификаторы доступа:

* **public**: публичный, общедоступный класс или член класса. Такой член класса доступен из любого места в коде, а также из других программ и сборок.
* **private**: закрытый класс или член класса. Представляет полную противоположность модификатору public. Такой закрытый класс или член класса доступен только из кода в том же классе или контексте.
* **protected:** такой член класса доступен из любого места в текущем классе или в производных классах. При этом производные классы могут располагаться в других сборках.
* **internal**: класс и члены класса с подобным модификатором доступны из любого места кода в той же сборке, однако он недоступен для других программ и сборок (как в случае с модификатором public).
* **protected internal**: совмещает функционал двух модификаторов. Классы и члены класса с таким модификатором доступны из текущей сборки и из производных классов.
* **private protected:** такой член класса доступен из любого места в текущем классе или в производных классах, которые определены в той же сборке.

Если для полей и методов не определен модификатор доступа, то по умолчанию для них применяется модификатор private. Классы и структуры, объявленные без модификатора, по умолчанию имеют доступ internal.

***Свойства***

Кроме обычных методов в языке C# предусмотрены специальные методы доступа, которые называют свойства. Они обеспечивают простой доступ к полям классов и структур, узнать их значение или выполнить их установку. Стандартное описание свойства имеет следующий синтаксис:

*[модификатор\_доступа] возвращаемый\_тип произвольное\_название*

*{*

*// код свойства*

*}*

*class Person*

*{*

*private string name;*

*public string Name*

*{*

*get*

*{ return name; }*

*set*

*{ name = value; }*

*}*

*}*

Но свойства позволяют вложить дополнительную логику, которая может быть необходима, например, при присвоении переменной класса какого-либо значения. Блоки set и get не обязательно одновременно должны присутствовать в свойстве. Если свойство определяют только блок get, то такое свойство доступно только для чтения - мы можем получить его значение, но не установить. И, наоборот, если свойство имеет только блок set, тогда это свойство доступно только для записи - можно только установить значение, но нельзя получить.

Автоматические свойства. если у нас с десяток и более полей, то определять каждое поле и писать для него однотипное свойство было бы утомительно. Поэтому в фреймворк .NET были добавлены автоматические свойства.

*class Person*

*{*

*public string Name { get; set; }*

*public int Age { get; set; }*

*public Person(string name, int age)*

*{*

*Name = name;*

*Age = age;*

*}*

*}*

***Сокращенная запись свойств***

Как и методы, мы можем сокращать свойства. Например:

*class Person*

*{*

*private string name;*

*// эквивалентно public string Name { get { return name; } }*

*public string Name => name;*

*}*

**Структура**

Один из способов создания собственных типов данных в C#.

*struct User*

*{*

*public string name;*

*public int age;*

*public void DisplayInfo()*

*{*

*Console.WriteLine($"Name: {name} Age: {age}");*

*}*

*}*

***Конструкторы структуры***

Как и класс, структура может определять конструкторы. Но в отличие от класса нам не обязательно вызывать конструктор для создания объекта структуры. Однако если мы таким образом создаем объект структуры, то обязательно надо проинициализировать все поля (глобальные переменные) структуры перед получением их значений или перед вызовом методов структуры. Также мы можем использовать для создания структуры конструктор без параметров, который есть в структуре по умолчанию и при вызове, которого полям структуры будет присвоено значение по умолчанию. Но в отличие от класса нельзя инициализировать поля структуры напрямую при их объявлении.

*struct User*

*{*

*public string name = "Sam"; // ! Ошибка*

*public int age = 23; // ! Ошибка*

*public void DisplayInfo()*

*{*

*Console.WriteLine($"Name: {name} Age: {age}");*

*}*

*}*

**Принципы ООП**

Объектно-ориентированное программирование (ООП) — методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.

***Принцип 1. Наследование***

Наследование — механизм, который позволяет описать новый класс на основе существующего (родительского). При этом свойства и функциональность родительского класса заимствуются новым классом.

***Принцип 2. Абстракция***

Абстракция означает выделение главных, наиболее значимых характеристик предмета и наоборот — отбрасывание второстепенных, незначительных.

***Принцип 3. Инкапсуляция***

Инкапсуляция – ограничение доступа к данным и возможностям их изменения. Инкапсуляция представляет собой способности языка скрывать излишние детали реализации от пользователя объекта. Основной единицей инкапсуляции в C# является класс, который определяет форму объекта. Очевидные примеры инкапсуляции — это модификаторы доступа (private, public и т.д.), а также геттеры-сеттеры.

***Принцип 4. Полиморфизм***

Полиморфизм — это возможность работать с несколькими типами так, будто это один и тот же тип. При этом поведение объектов будет разным в зависимости от типа, к которому они принадлежат. (Перегрузка, переопределение, интерфейсы, generics). В C# полиморфизм достигается за счет использования абстрактных и виртуальных методов.

**Наследование**

В языке C# класс, который наследуется, называется базовым, а класс, который наследует, — производным. Следовательно, производный класс представляет собой специализированный вариант базового класса. Он наследует все переменные, методы, свойства и индексаторы, определяемые в базовом классе, добавляя к ним свои собственные элементы. Поддержка наследования в C# состоит в том, что в объявление одного класса разрешается вводить другой класс. Для этого при объявлении производного класса указывается базовый класс.

Всякий раз, когда один класс наследует от другого, после имени производного класса указывается имя базового класса, отделяемое двоеточием.

*class Employee : Person*

*{*

*}*

Единственное, что не передается при наследовании, это конструкторы базового класса. ***Таким образом, наследование реализует отношение is-a (является).***

Все классы по умолчанию могут наследоваться. Однако здесь есть ряд ограничений:

* Не поддерживается множественное наследование, класс может наследоваться только от одного класса.
* При создании производного класса надо учитывать тип доступа к базовому классу - тип доступа к производному классу должен быть таким же, как и у базового класса, или более строгим. То есть, если базовый класс у нас имеет тип доступа internal, то производный класс может иметь тип доступа internal или private, но не public. Однако следует также учитывать, что если базовый и производный класс находятся в разных сборках (проектах), то в этом случае производный класс может наследовать только от класса, который имеет модификатор public.
* Если класс объявлен с модификатором sealed, то от этого класса нельзя наследовать и создавать производные классы.
* Нельзя унаследовать класс от статического класса.

***Доступ к членам базового класса из класса-наследника***

Производный класс может иметь доступ только к тем членам базового класса, которые определены с модификаторами *private protected*, *public, internal*, *protected* и *protected internal*.

***Ключевое слово base***

С помощью ключевого слова base мы можем обратиться к базовому классу. Конструкторы не передаются производному классу при наследовании. И если в базовом классе не определен конструктор по умолчанию без параметров, а только конструкторы с параметрами, то в производном классе мы обязательно должны вызвать один из этих конструкторов через ключевое слово base.

*class Person*

*{*

*public string Name { get; set; }*

*public Person(string name)*

*{*

*Name = name;*

*}*

*public void Display()*

*{*

*Console.WriteLine(Name);*

*}*

*}*

*class Employee : Person*

*{*

*public string Company { get; set; }*

*public Employee(string name, string company)*

*: base(name)*

*{*

*Company = company;*

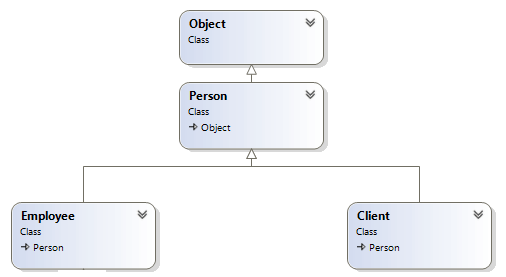
*}*

*}*

При вызове конструктора класса сначала отрабатывают конструкторы базовых классов и только затем конструкторы производных.

**Преобразование типов**

В этой иерархии классов мы можем проследить следующую цепь наследования: Object (все классы неявно наследуются от типа Object) -> Person -> Employee|Client.



Причем в этой иерархии классов базовые типы находятся вверху, а производные типы - внизу.

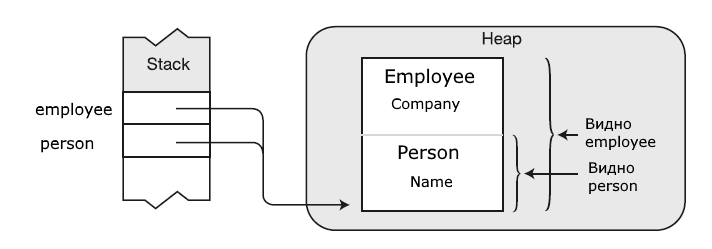
***Восходящие преобразования. Upcasting***

Объекты производного типа (который находится внизу иерархии) в то же время представляют и базовый тип.

*Employee employee = new Employee("Tom", "Microsoft");*

*Person person = employee; // преобразование от Employee к Person*

Переменные employee и person будут указывать на один и тот же объект в памяти, но переменной person будет доступна только та часть, которая представляет функционал типа Person.



***Нисходящие преобразования. Downcasting***

Для нисходящего преобразования необходимо применить явное преобразования, указав в скобках тип, к которому нужно выполнить преобразование.

*// Объект Employee также представляет тип object*

*object obj = new Employee("Bill", "Microsoft");*

*// чтобы обратиться к возможностям типа Employee, приводим объект к типу Employee*

*Employee emp = (Employee) obj;*

*// объект Client также представляет тип Person*

*Person person = new Client("Sam", "ContosoBank");*

*// преобразование от типа Person к Client*

*Client client = (Client)person;*

Если нам надо обратиться к каким-то отдельным свойствам или методам объекта, то нам необязательно присваивать преобразованный объект переменной:

*// Объект Employee также представляет тип object*

*object obj = new Employee("Bill", "Microsoft");*

*// преобразование к типу Person для вызова метода Display*

*((Person)obj).Display();*

*// либо так*

*// ((Employee)obj).Display();*

*// преобразование к типу Employee, чтобы получить свойство Company*

*string comp = ((Employee)obj).Company;*

***Способы преобразований***

1. Во-первых, можно использовать ключевое слово as. С помощью него программа пытается преобразовать выражение к определенному типу, при этом не выбрасывает исключение. В случае неудачного преобразования выражение будет содержать значение null.
2. Второй способ заключается в отлавливании исключения InvalidCastException, которое возникнет в результате преобразования.
3. Третий способ заключается в проверке допустимости преобразования с помощью ключевого слова is. Выражение person is Employee проверяет, является ли переменная person объектом типа Employee. Но так как в данном случае явно не является, то такая проверка вернет значение false, и преобразование не сработает.

**Перегрузка методов**

Возникает необходимость создать один и тот же метод, но с разным набором параметров. И в зависимости от имеющихся параметров применять определенную версию метода. Такая возможность еще называется перегрузкой методов (method overloading).

Сигнатура складывается из следующих аспектов:

* Имя метода
* Количество параметров
* Типы параметров
* Порядок параметров
* Модификаторы параметров

Перегрузка метода как раз заключается в том, что методы имеют разную сигнатуру, в которой совпадает только название метода. То есть методы должны отличаться по:

* Количеству параметров
* Типу параметров
* Порядку параметров
* Модификаторам параметров

*Add(int, int)*

*Add(int, int, int)*

*Add(int, int, int, int)*

*Add(double, double)*

Также перегружаемые методы могут отличаться по используемым модификаторам. А отличие методов по возвращаемому типу или по имени параметров не является основанием для перегрузки.

**Перегрузка операторов**

Наряду с методами мы можем также перегружать операторы. Перегрузка операторов заключается в определении в классе, для объектов которого мы хотим определить оператор, специального метода:

*public static возвращаемый\_тип operator оператор(параметры)*

*{ }*

Этот метод должен иметь модификаторы public static, так как перегружаемый оператор будет использоваться для всех объектов данного класса. Далее идет название возвращаемого типа. Возвращаемый тип представляет тот тип, объекты которого мы хотим получить. Затем вместо названия метода идет ключевое слово operator и собственно сам оператор. И далее в скобках перечисляются параметры. Бинарные операторы принимают два параметра, унарные - один параметр. И в любом случае один из параметров должен представлять тот тип - класс или структуру, в котором определяется оператор.

В частности, мы можем перегрузить следующие операторы:

* унарные операторы +, -, !, ~, ++, --
* бинарные операторы +, -, \*, /, %
* операции сравнения ==, !=, <, >, <=, >=
* логические операторы &&, ||
* Также стоит отметить, что мы можем переопределить операторы true и false.

И есть ряд операторов, которые нельзя перегрузить, например, операцию равенства = или тернарный оператор ?:, а также ряд других.

При перегрузке операторов также следует помнить, что мы не можем изменить приоритет оператора или его ассоциативность, мы не можем создать новый оператор или изменить логику операторов в типах, который есть по умолчанию в .NET.

**Значение null**

Если переменным ссылочного типа не присваивается значение, то им дается значение по умолчанию - значение null. Фактически оно говорит об отсутствии значения как такового. Но типы значений, например, int, decimal, double и т.д. не могут принимать значение null.

***Оператор ??***

Оператор ?? называется оператором null-объединения. Он применяется для установки значений по умолчанию для типов, которые допускают значение null. Оператор ?? возвращает левый операнд, если этот операнд не равен null. Иначе возвращается правый операнд. При этом левый операнд должен принимать null.

*object x = null;*

*object y = x ?? 100; // равно 100, так как x равен null*

*object z = 200;*

*object t = z ?? 44; // равно 200, так как z не равен null*

***Оператор условного null***

Иногда при работе с объектами, которые принимают значение null, мы можем столкнуться с ошибкой: мы пытаемся обратиться к объекту, а этот объект равен null. Чтобы избежать этой ошибки мы могли бы использовать условную конструкцию для проверки на null. Выражение ?. и представляет оператор условного null. Здесь последовательно проверяется равен ли объект user и вложенные объекты значению null. Если же на каком-то этапе один из объектов окажется равным null, то companyName будет иметь значение по умолчанию, то есть null.

*string companyName = user?.Phone?.Company?.Name;*

**Индексаторы**

Индексаторы позволяют индексировать объекты и обращаться к данным по индексу. Фактически с помощью индексаторов мы можем работать с объектами как с массивами. По форме они напоминают свойства со стандартными блоками get и set, которые возвращают и присваивают значение.

Формальное определение индексатора:

*возвращаемый\_тип this [Тип параметр1, ...]*

*{*

*get { ... }*

*set { ... }*

*}*

В отличие от свойств индексатор не имеет названия. Вместо него указывается ключевое слово this, после которого в квадратных скобках идут параметры. Индексатор должен иметь как минимум один параметр.

*class Person*

*{*

*public string Name { get; set; }*

*}*

*class People*

*{*

*Person[] data;*

*public People()*

*{*

*data = new Person[5];*

*}*

*// индексатор*

*public Person this[int index]*

*{*

*get*

*{*

*return data[index];*

*}*

*set*

*{*

*data[index] = value;*

*}*

*}*

*}*

***Перегрузка индексаторов***

Подобно методам индексаторы можно перегружать. В этом случае также индексаторы должны отличаться по количеству, типу или порядку используемых параметров.

class Person

{

public string Name { get; set; }

public int Age { get; set; }

}

class People

{

Person[] data;

public People ()

{

data = new Person [5];

}

public Person this [int index]

{

get {return data[index];}

set{data[index] = value;}

}

public Person this [string name]

{

get {

Person person = null;

foreach(var p in data)

{

if (p?.Name == name)

{person = p;

break;}}

return person;

}

}

}

**Виртуальные методы и свойства**

При наследовании нередко возникает необходимость изменить в классе-наследнике функционал метода, который был унаследован от базового класса. В этом случае класс-наследник может переопределять методы и свойства базового класса. Те методы и свойства, которые мы хотим сделать доступными для переопределения, в базовом классе помечается модификатором virtual. Такие методы и свойства называют виртуальными. А чтобы переопределить метод в классе-наследнике, этот метод определяется с модификатором override. Переопределенный метод в классе-наследнике должен иметь тот же набор параметров, что и виртуальный метод в базовом классе.

Чтобы сделать метод Display доступным для переопределения, этот метод определен с модификатором virtual. Поэтому мы можем переопределить этот метод, но можем и не переопределять. Допустим, нас устраивает реализация метода из базового класса. В этом случае объекты Employee будут использовать реализацию метода Display из класса Person. Но также можем переопределить виртуальный метод. Для этого в классе-наследнике определяется метод с модификатором override, который имеет то же самое имя и набор параметров.

При переопределении виртуальных методов следует учитывать ряд ограничений:

* Виртуальный и переопределенный методы должны иметь один и тот же модификатор доступа. То есть если виртуальный метод определен с помощью модификатора public, то и переопредленный метод также должен иметь модификатор public.
* Нельзя переопределить или объявить виртуальным статический метод.

Также, как и методы, можно переопределять свойства.

Кроме конструкторов, мы можем обратиться с помощью ключевого слова base к другим членам базового класса. В нашем случае вызов base.Display(); будет обращением к методу Display() в классе Person.

Также можно запретить переопределение методов и свойств. В этом случае их надо объявлять с модификатором sealed.

При создании методов с модификатором sealed надо учитывать, что sealed применяется в паре с override, то есть только в переопределяемых методах.

И в этом случае мы не сможем переопределить метод Display в классе, унаследованном от Employee.

**Сокрытие**

Сокрытие представляет определение в классе-наследнике метода или свойства, которые соответствует по имени и набору параметров методу или свойству базового класса. Для сокрытия членов класса применяется ключевое слово new. Если в базовом классе не является виртуальным, мы не можем его переопределить, но, допустим, нас не устраивает его реализация для производного класса, поэтому мы можем воспользоваться сокрытием, чтобы определить нужный нам функционал.

*class Person*

*{*

*protected string name;*

*public string Name*

*{*

*get { return name; }*

*set { name = value; }*

*}*

*}*

*class Employee : Person*

*{*

*public new string Name*

*{*

*get { return "Employee " + base.Name; }*

*set { name = value; }*

*}*

*}*

При этом если мы хотим обратиться именно к реализации свойства или метода в базовом классе, то опять же мы можем использовать ключевое слово base и через него обращаться к функциональности базового класса. Более того мы даже можем применять сокрытие к переменным и константам, также используя ключевое слово new.

|  |  |
| --- | --- |
| Переопределение | Сокрытие |
| Для работы с виртуальными методами компилятор формирует таблицу виртуальных методов (Virtual Method Table или VMT). В нее записывается адреса виртуальных методов. Для каждого класса создается своя таблица. Когда создается объект класса, то компилятор передает в конструктор объекта специальный код, который связывает объект и таблицу VMT. А при вызове виртуального метода из объекта берется адрес его таблицы VMT. Затем из VMT извлекается адрес метода и ему передается управление. То есть процесс выбора реализации метода производится во время выполнения программы. Следует учитывать, что так как среде выполнения вначале необходимо получить из таблицы VMT адрес нужного метода, то это немного замедляет выполнение программы. | Переменная tom представляет тип Person, но хранит ссылку на объект Employee. Однако при вызове метода Display будет выполняться та версия метода, которая определена именно в классе Person, а не в классе Employee. Почему? Класс Employee никак не переопределяет метод Display, унаследованный от базового класса, а фактически определяет новый метод. Поэтому при вызове tom.Display() вызывается метод Display из класса Person. |

**Абстрактные классы и члены классов**

Абстрактный класс похож на обычный класс. Он также может иметь переменные, методы, конструкторы, свойства. Единственное, что при определении абстрактных классов используется ключевое слово abstract:

*abstract class Human*

*{*

*public int Length { get; set; }*

*public double Weight { get; set; }*

*}*

Но главное отличие состоит в том, что мы не можем использовать конструктор абстрактного класса для создания его объекта. Если необходимо реализовать несколько классов, которые имеют что-то общее, то общую функциональность лучше вынести в какой-то отдельный класс и если не нужны объекты от общего класса, то его нужно сделать абстрактным.

*abstract class Person*

*{*

*public string Name { get; set; }*

*public Person(string name)*

*{*

*Name = name;*

*}*

*public void Display()*

*{*

*Console.WriteLine(Name);*

*}*

*}*

*class Client : Person*

*{*

*public int Sum { get; set; } // сумма на счету*

*public Client(string name, int sum)*

*: base(name)*

*{*

*Sum = sum;*

*}*

*}*

*class Employee : Person*

*{*

*public string Position { get; set; } // должность*

*public Employee(string name, string position)*

*: base(name)*

*{*

*Position = position;*

*}*

*}*

Однако несмотря на то, что напрямую мы не можем вызвать конструктор класса Person для создания объекта, тем не менее конструктор в абстрактных классах то же может играть важную роль, в частности, инициализировать некоторые общие для производных классов переменные и свойства, как в случае со свойством Name. И хотя в примере выше конструктор класса Person не вызывается, тем не менее производные классы Client и Employee могут обращаться к нему.

Кроме обычных свойств и методов абстрактный класс может иметь абстрактные члены классов, которые определяются с помощью ключевого слова abstract и не имеют никакого функционала. В частности, абстрактными могут быть:

* Методы
* Свойства
* Индексаторы
* События

Абстрактные члены классов не должны иметь модификатор private. При этом производный класс обязан переопределить и реализовать все абстрактные методы и свойства, которые имеются в базовом абстрактном классе. При переопределении в производном классе такой метод или свойство также объявляются с модификатором override (как и при обычном переопределении виртуальных методов и свойств). Также следует учесть, что если класс имеет хотя бы один абстрактный метод (или абстрактные свойство, индексатор, событие), то этот класс должен быть определен как абстрактный.

Абстрактные члены также, как и виртуальные, являются частью полиморфного интерфейса. Но если в случае с виртуальными методами мы говорим, что класс-наследник наследует реализацию, то в случае с абстрактными методами наследуется интерфейс, представленный этими методами.

Производный класс обязан реализовать все абстрактные члены базового класса. Однако мы можем отказаться от реализации, но в этом случае производный класс также должен быть определен как абстрактный:

abstract class Person

{

public abstract string Name { get; set; }

}

abstract class Manager : Person

{

}

**Отношения между классами и объектами**

Мы можем выделить несколько основных отношений: наследование, реализация, ассоциация, композиция и агрегация.

***Наследование***

Наследование является базовым принципом ООП и позволяет одному классу (наследнику) унаследовать функционал другого класса (родительского). Нередко отношения наследования еще называют генерализацией или обобщением. Наследование определяет отношение IS A, то есть "является".

***Реализация***

Реализация предполагает определение интерфейса и его реализация в классах. Например, имеется интерфейс IMovable с методом Move, который реализуется в классе Car:

*public interface IMovable*

*{*

*void Move();*

*}*

*public class Car : IMovable*

*{*

*public void Move()*

*{*

*Console.WriteLine("Машина едет");*

*}*

*}*

***Ассоциация***

Ассоциация - это отношение, при котором объекты одного типа неким образом связаны с объектами другого типа. Нередко при отношении ассоциации указывается кратность связей. В данном случае единица у Team и звездочка у Player на диаграмме отражает связь 1 ко многим. То есть одна команда будет соответствовать многим игрокам. Агрегация и композиция являются частными случаями ассоциации.

***Композиция***

Композиция определяет отношение HAS A, то есть отношение "имеет". Например, в класс автомобиля содержит объект класса электрического двигателя:

*public class ElectricEngine*

*{ }*

*public class Car*

*{*

*ElectricEngine engine;*

*public Car()*

*{*

*engine = new ElectricEngine();*

*}*

*}*

При этом класс автомобиля полностью управляет жизненным циклом объекта двигателя. При уничтожении объекта автомобиля в области памяти вместе с ним будет уничтожен и объект двигателя. И в этом плане объект автомобиля является главным, а объект двигателя - зависимой.

***Агрегация***

От композиции следует отличать агрегацию. Она также предполагает отношение HAS A, но реализуется она иначе:

*public abstract class Engine*

*{ }*

*public class Car*

*{*

*Engine engine;*

*public Car(Engine eng)*

*{*

*engine = eng;*

*}*

*}*

При агрегации реализуется слабая связь, то есть в данном случае объекты Car и Engine будут равноправны. В конструктор Car передается ссылка на уже имеющийся объект Engine. И, как правило, определяется ссылка не на конкретный класс, а на абстрактный класс или интерфейс, что увеличивает гибкость программы.

**Обработка исключений**

Иногда при выполнении программы возникают ошибки, которые трудно предусмотреть или предвидеть, а иногда и вовсе невозможно.

*try*

*{*

*}*

*catch*

*{*

*}*

*finally*

*{*

*}*

При использовании блока try...catch..finally вначале выполняются все инструкции в блоке try. Если в этом блоке не возникло исключений, то после его выполнения начинает выполняться блок finally. И затем конструкция try..catch..finally завершает свою работу.

Если же в блоке try вдруг возникает исключение, то обычный порядок выполнения останавливается, и среда CLR начинает искать блок catch, который может обработать данное исключение. Если нужный блок catch найден, то он выполняется, и после его завершения выполняется блок finally.

Если нужный блок catch не найден, то при возникновении исключения программа аварийно завершает свое выполнение.

С точки зрения производительности использование блоков try..catch более накладно, чем применение условных конструкций. Поэтому по возможности вместо try..catch лучше использовать условные конструкции на проверку исключительных ситуаций.

***Блок catch и фильтры исключений***

За обработку исключения отвечает блок catch, который может иметь следующие формы:

* *catch*

*{*

*// выполняемые инструкции*

*}*

Обрабатывает любое исключение, которое возникло в блоке try.

* *catch (тип\_исключения)*

*{*

*// выполняемые инструкции*

*}*

Обрабатывает только те исключения, которые соответствуют типу, указаному в скобках после оператора catch.

* *catch (тип\_исключения имя\_переменной)*

*{*

*// выполняемые инструкции*

*}*

Обрабатывает только те исключения, которые соответствуют типу, указаному в скобках после оператора catch. А вся информация об исключении помещается в переменную данного типа. Фактически этот случай аналогичен предыдущему за тем исключением, что здесь используется переменная. Если нам не нужна информация об исключении, то переменную можно не использовать как в предыдущем случае.

Фильтры исключений позволяют обрабатывать исключения в зависимости от определенных условий. Для их применения после выражения catch идет выражение when, после которого в скобках указывается условие:

*catch when(условие)*

*{…}*

***Класс Exception***

Базовым для всех типов исключений является тип Exception. Этот тип определяет ряд свойств, с помощью которых можно получить информацию об исключении.

* InnerException: хранит информацию об исключении, которое послужило причиной текущего исключения
* Message: хранит сообщение об исключении, текст ошибки
* Source: хранит имя объекта или сборки, которое вызвало исключение
* StackTrace: возвращает строковое представление стека вызывов, которые привели к возникновению исключения
* TargetSite: возвращает метод, в котором и было вызвано исключение

***Типы исключений.***

Но также есть более специализированные типы исключений, которые предназначены для обработки каких-то определенных видов исключений. Их довольно много, я приведу лишь некоторые:

* DivideByZeroException: представляет исключение, которое генерируется при делении на ноль
* ArgumentOutOfRangeException: генерируется, если значение аргумента находится вне диапазона допустимых значений
* ArgumentException: генерируется, если в метод для параметра передается некорректное значение
* IndexOutOfRangeException: генерируется, если индекс элемента массива или коллекции находится вне диапазона допустимых значений
* InvalidCastException: генерируется при попытке произвести недопустимые преобразования типов
* NullReferenceException: генерируется при попытке обращения к объекту, который равен null (то есть по сути неопределен)

При этом блоки catch для более общих, более базовых исключений следует помещать в конце - после блоков catch для более конкретный, специализированных типов. Так как CLR выбирает для обработки исключения первый блок catch, который соответствует типу сгенерированного исключения.

***Создание классов исключений***

Базовым классом для всех исключений является класс Exception, соответственно для создания своих типов мы можем унаследовать данный класс.

*class PersonException : Exception*

*{*

*public PersonException(string message)*

*: base(message)*

*{ }*

*}*

По сути класс кроме пустого конструктора ничего не имеет, и то в конструкторе мы просто обращаемся к конструктору базового класса Exception, передавая в него строку message. Но теперь мы можем изменить класс Person, чтобы он выбрасывал исключение именно этого типа и соответственно в основной программе обрабатывать это исключение.

Однако необязательно наследовать свой класс исключений именно от типа Exception, можно взять какой-нибудь другой производный тип. Каждый тип исключений может определять какие-то свои свойства.

*class PersonException : ArgumentException*

*{*

*public int Value { get;}*

*public PersonException(string message, int val)*

*: base(message)*

*{*

*Value = val;*

*}*

*}*

***Генерация исключения и оператор throw***

Язык C# также позволяет генерировать исключения вручную с помощью оператора throw. То есть с помощью этого оператора мы сами можем создать исключение и вызвать его в процессе выполнения. После оператора throw указывается объект исключения, через конструктор которого мы можем передать сообщение об ошибке. Естественно вместо типа Exception мы можем использовать объект любого другого типа исключений.

Затем в блоке catch сгенерированное нами исключение будет обработано.

Подобным образом мы можем генерировать исключения в любом месте программы. Но существует также и другая форма использования оператора throw, когда после данного оператора не указывается объект исключения. В подобном виде оператор throw может использоваться только в блоке catch

*try*

*{*

*try*

*{*

*Console.Write("Введите строку: ");*

*string message = Console.ReadLine();*

*if (message.Length > 6)*

*{*

*throw new Exception("Длина строки больше 6 символов");*

*}*

*}*

*catch*

*{*

*Console.WriteLine("Возникло исключение");*

*throw;*

*}*

*}*

*catch (Exception ex)*

*{*

*Console.WriteLine(ex.Message);*

*}*

В данном случае при вводе строки с длиной больше 6 символов возникнет исключение, которое будет обработано внутренним блоком catch. Однако поскольку в этом блоке используется оператор throw, то исключение будет передано дальше внешнему блоку catch.

***Проброска исключений***

**Интерфейсы**

Интерфейс представляет ссылочный тип, который может определять некоторый функционал - набор методов и свойств без реализации. Затем этот функционал реализуют классы и структуры, которые применяют данные интерфейсы. Для определения интерфейса используется ключевое слово interface. Как правило, названия интерфейсов в C# начинаются с заглавной буквы I.

В целом интерфейсы могут определять следующие сущности:

* Методы
* Свойства
* Индексаторы
* События
* Статические поля и константы (начиная с версии C# 8.0)

Однако интерфейсы не могут определять нестатические переменные.

*interface IMovable*

*{*

*// константа*

*const int minSpeed = 0; // минимальная скорость*

*// статическая переменная*

*static int maxSpeed = 60; // максимальная скорость*

*// метод*

*void Move(); // движение*

*// свойство*

*string Name { get; set; } // название*

*delegate void MoveHandler(string message); // определение делегата для события*

*// событие*

*event MoveHandler MoveEvent; // событие движения*

*}*

То есть интерфейс описывает некоторый функционал, который должен быть у движущегося объекта. Методы и свойства интерфейса могут не иметь реализации, в этом они сближаются с абстрактными методами и свойствами абстрактных классов. Еще один момент в объявлении интерфейса: если его члены - методы и свойства не имеют модификаторов доступа, но фактически по умолчанию доступ public, так как цель интерфейса - определение функционала для реализации его классом. Это касается также и констант, и статических переменных, которые в классах и структурах по умолчанию имееют модификатор private. В интерфейсах же они имеют по умолчанию модификатор public.

Но также, начиная с версии C# 8.0, мы можем явно указывать модификаторы доступа у компонентов интерфейса. Начиная с версии C# 8.0 интерфейсы поддерживают реализацию методов и свойств по умолчанию. Это значит, что мы можем определить в интерфейсах полноценные методы и свойства, которые имеют реализацию как в обычных классах и структурах. С реализацией свойств по умолчанию в интерфейсах дело обстоит несколько сложнее, поскольку мы не можем определять в интерфейсах нестатические переменные, соответственно в свойствах интерфейса мы не можем манипулировать состоянием объекта. Тем не менее реализацию по умолчанию для свойств мы тоже можем определять. Стоит отметить, что если интерфейс имеет приватные методы и свойства (то есть с модификатором private), то они должны иметь реализацию по умолчанию. То же самое относится к любым статическим методам и свойствам (не обязательно приватным).

Интерфейс представляет некое описание типа, набор компонентов, который должен иметь тип данных. И, собственно, мы не можем создавать объекты интерфейса напрямую с помощью конструктора. При применении интерфейса, как и при наследовании после имени класса или структуры указывается двоеточие и затем идут названия применяемых интерфейсов. При этом класс должен реализовать все методы и свойства применяемых интерфейсов, если эти методы и свойства не имеют реализации по умолчанию. Если методы и свойства интерфейса не имеют модификатора доступа, то по умолчанию они являются публичными, при реализации этих методов и свойств в классе и структуре к ним можно применять только модификатор public.

Допустим, у нас есть куча классов, которые реализуют некоторый интерфейс. Если мы добавим в этот интерфейс новый метод, то мы будем обязаны реализовать этот метод во всех классах, применяющих данный интерфейс. Иначе подобные классы просто не будут компилироваться. Теперь вместо реализации метода во всех классах нам достаточно определить его реализацию по умолчанию в интерфейсе. Если класс не реализует метод, будет применяться реализация по умолчанию.

Интерфейсы имеют еще одну важную функцию: в C# не поддерживается множественное наследование, то есть мы можем унаследовать класс только от одного класса. Интерфейсы позволяют частично обойти это ограничение, поскольку в C# класс может реализовать сразу несколько интерфейсов. Все реализуемые интерфейсы указываются через запятую:

*myClass: myInterface1, myInterface2, myInterface3, ...*

*{*

*}*

Преобразование от класса к его интерфейсу, как и преобразование от производного типа к базовому, выполняется автоматически. Обратное преобразование - от интерфейса к реализующему его классу будет аналогично преобразованию от базового класса к производному.

Интерфейсы, как и классы, могут наследоваться:

*interface IAction*

*{*

*void Move();*

*}*

*interface IRunAction : IAction*

*{*

*void Run();*

*}*

*class BaseAction : IRunAction*

*{*

*public void Move()*

*{*

*Console.WriteLine("Move");*

*}*

*public void Run()*

*{*

*Console.WriteLine("Run");*

*}*

*}*

Однако в отличие от классов мы не можем применять к интерфейсам модификатор sealed, чтобы запретить наследование интерфейсов. Также мы не можем применять к интерфейсам модификатор abstract, поскольку интерфейс фактически итак, как правило, предоставляет абстрактный функционал, который должен быть реализован в классе или структуре (за исключением методов и свойств с реализацией по умолчанию). Однако методы интерфейсов могут использовать ключевое слово new для сокрытия методов из базового интерфейса.

***Интерфейс ICloneable***

Поскольку классы представляют ссылочные типы, то это накладывает некоторые ограничения на их использование. Для копирования классов используется интерфейс ICloneable:

*public interface ICloneable*

*{*

*object Clone();*

*}*

Для сокращения кода копирования мы можем использовать специальный метод MemberwiseClone(), который возвращает копию объекта.

*class Person : ICloneable*

*{*

*public string Name { get; set; }*

*public int Age { get; set; }*

*public object Clone()*

*{*

*return this.MemberwiseClone();*

*}*

*}*

Этот метод реализует поверхностное (неглубокое) копирование. Однако данного копирования может быть недостаточно.

оверхностное копирование работает только для свойств, представляющих примитивные типы, но не для сложных объектов. И в этом случае надо применять глубокое копирование:

*class Person : ICloneable*

*{*

*public string Name { get; set; }*

*public int Age { get; set; }*

*public Company Work { get; set; }*

*public object Clone()*

*{*

*Company company = new Company { Name = this.Work.Name };*

*return new Person*

*{*

*Name = this.Name,*

*Age = this.Age,*

*Work = company*

*};*

*}*

*}*

*class Company*

*{*

*public string Name { get; set; }*

*}*

***Интерфейс IComparable***

Для сортировки наборов сложных объектов применяется интерфейс IComparable. Он имеет всего один метод:

*public interface IComparable*

*{*

*int CompareTo(object o);*

*}*

Метод CompareTo предназначен для сравнения текущего объекта с объектом, который передается в качестве параметра object o. На выходе он возвращает целое число, которое может иметь одно из трех значений:

* Меньше нуля. Значит, текущий объект должен находиться перед объектом, который передается в качестве параметра
* Равен нулю. Значит, оба объекта равны
* Больше нуля. Значит, текущий объект должен находиться после объекта, передаваемого в качестве параметра

Кроме интерфейса IComparable платформа .NET также предоставляет интерфейс IComparer:

*interface IComparer*

*{*

*int Compare(object o1, object o2);*

*}*

Метод Compare предназначен для сравнения двух объектов o1 и o2. Он также возвращает три значения, в зависимости от результата сравнения: если первый объект больше второго, то возвращается число больше 0, если меньше - то число меньше нуля; если оба объекта равны, возвращается ноль.

**Методы расширения**

Методы расширения (extension methods) позволяют добавлять новые методы в уже существующие типы без создания нового производного класса. Эта функциональность бывает особенно полезна, когда нам хочется добавить в некоторый тип новый метод, но сам тип (класс или структуру) мы изменить не можем, поскольку у нас нет доступа к исходному коду. Либо если мы не можем использовать стандартный механизм наследования, например, если классы определенны с модификатором sealed.

Для того, чтобы создать метод расширения, вначале надо создать статический класс, который и будет содержать этот метод. Затем объявляем статический метод. Собственно, метод расширения - это обычный статический метод, который в качестве первого параметра всегда принимает такую конструкцию: this имя\_типа название\_параметра.

Применение методов расширения очень удобно, но при этом надо помнить, что метод расширения никогда не будет вызван, если он имеет ту же сигнатуру, что и метод, изначально определенный в типе. Также следует учитывать, что методы расширения действуют на уровне пространства имен. То есть, если добавить в проект другое пространство имен, то метод не будет применяться к строкам, и в этом случае надо будет подключить пространство имен метода через директиву using.

*class Program*

*{*

*static void Main(string[] args)*

*{*

*string s = "Привет мир";*

*char c = 'и';*

*int i = s.CharCount(c);*

*Console.WriteLine(i);*

*Console.Read();*

*}*

*}*

*public static class StringExtension*

*{*

*public static int CharCount(this string str, char c)*

*{*

*int counter = 0;*

*for (int i = 0; i<str.Length; i++)*

*{*

*if (str[i] == c)*

*counter++;*

*}*

*return counter;*

*}*

*}*

**Обобщения**

Упаковка (boxing) предполагает преобразование объекта значимого типа (например, типа int) к типу object. При упаковке общеязыковая среда CLR обертывает значение в объект типа System.Object и сохраняет его в управляемой куче (хипе). Распаковка (unboxing), наоборот, предполагает преобразование объекта типа object к значимому типу. Упаковка и распаковка ведут к снижению производительности, так как системе надо осуществить необходимые преобразования. Эти проблемы были призваны устранить обобщенные типы (также часто называют универсальными типами). Обобщенные типы позволяют указать конкретный тип, который будет использоваться.

Поэтому определим класс Account как обобщённый:

*class Account<T>*

*{*

*public T Id { get; set; }*

*public int Sum { get; set; }*

*}*

Угловые скобки в описании class Account<T> указывают, что класс является обобщенным, а тип T, заключенный в угловые скобки, будет использоваться этим классом. Необязательно использовать именно букву T, это может быть и любая другая буква или набор символов. Причем сейчас нам неизвестно, что это будет за тип, это может быть любой тип. Поэтому параметр T в угловых скобках еще называется универсальным параметром, так как вместо него можно подставить любой тип. Поскольку класс Account является обобщенным, то при определении переменной после названия типа в угловых скобках необходимо указать тот тип, который будет использоваться вместо универсального параметра T.

***Значения по умолчанию***

Иногда возникает необходимость присвоить переменным универсальных параметров некоторое начальное значение, в том числе и null. Но напрямую мы его присвоить не можем:

*T id = null;*

В этом случае нам надо использовать оператор default(T). Он присваивает ссылочным типам в качестве значения null, а типам значений - значение 0:

*class Account<T>*

*{*

*T id = default(T);*

*}*

Обобщения могут использовать несколько универсальных параметров одновременно, которые могут представлять различные типы:

*class Transaction<U, V>*

*{*

*public U FromAccount { get; set; } // с какого счета перевод*

*public U ToAccount { get; set; } // на какой счет перевод*

*public V Code { get; set; } // код операции*

*public int Sum { get; set; } // сумма перевода*

*}*

Здесь класс Transaction использует два универсальных параметра. Применим данный класс:

*Account<int> acc1 = new Account<int> { Id = 1857, Sum = 4500 };*

*Account<int> acc2 = new Account<int> { Id = 3453, Sum = 5000 };*

*Transaction<Account<int>, string> transaction1 = new Transaction<Account<int>, string>*

*{*

*FromAccount = acc1,*

*ToAccount = acc2,*

*Code = "45478758",*

*Sum = 900*

*};*

Здесь объект Transaction типизируется типами Account<int> и string. То есть в качестве универсального параметра U используется класс Account<int>, а для параметра V - тип string. При этом, как можно заметить, класс, которым типизируется Transaction, сам является обобщенным.

Кроме обобщенных классов можно также создавать обобщенные методы, которые точно также будут использовать универсальные параметры.

*public static void Swap<T> (ref T x, ref T y)*

*{*

*T temp = x;*

*x = y;*

*y = temp;*

*}*

Здесь определен обощенный метод Swap, который принимает параметры по ссылке и меняет их значения. При этом в данном случае не важно, какой тип представляют эти параметры. В методе Main вызываем метод Swap, типизируем его определенным типом и передаем ему некоторые значения.

***Ограничения обобщений***

С помощью универсальных параметров мы можем типизировать обобщенные классы любым типом. Однако иногда возникает необходимость конкретизировать тип. С помощью выражения where T : Account мы указываем, что используемый тип T обязательно должен быть классом Account или его наследником. Благодаря подобному ограничению мы можем использовать внутри класса Transaction все объекты типа T именно как объекты Account и соответственно обращаться к их свойствам и методам. Следует учитывать, что только один класс может использоваться в качестве ограничения. В качестве ограничения также может выступать и обобщенный класс.

В качестве ограничений мы можем использовать следующие типы:

* Классы
* Интерфейсы
* class - универсальный параметр должен представлять класс
* struct - универсальный параметр должен представлять структуру
* new() - универсальный параметр должен представлять тип, который имеет общедоступный (public) конструктор без параметров

Есть ряд стандартных ограничений, которые мы можем использовать. В частности, можно указать ограничение, чтобы использовались только структуры или другие типы значений:

*class Account<T> where T : struct*

*{}*

При этом использовать в качестве ограничения конкретные структуры в отличие от классов нельзя.

Также можно задать в качестве ограничения ссылочные типы:

*class Transaction<T> where T : class*

*{}*

А также можно задать с помощью слова new в качестве ограничения класс или структуру, которые имеют общедоступный конструктор без параметров:

*class Transaction<T> where T : new()*

*{}*

Если для универсального параметра задано несколько ограничений, то они должны идти в определенном порядке:

* Название класса, class, struct. Причем мы можем одновременно определить только одно из этих ограничений
* Название интерфейса
* new()

Если класс использует несколько универсальных параметров, то последовательно можно задать ограничения к каждому из них:

*class Transaction<U, V>*

*where U : Account<int>*

*where V : struct*

*{*

*}*

Один обобщенный класс может быть унаследован от другого обобщенного. При этом можно использовать различные варианты наследования.

**Делегаты**

Делегаты представляют такие объекты, которые указывают на методы. То есть делегаты - это указатели на методы и с помощью делегатов мы можем вызвать данные методы.

Для объявления делегата используется ключевое слово delegate, после которого идет возвращаемый тип, название и параметры.

*delegate void Message();*

Делегат Message в качестве возвращаемого типа имеет тип void (то есть ничего не возвращает) и не принимает никаких параметров. Это значит, что этот делегат может указывать на любой метод, который не принимает никаких параметров и ничего не возвращает.

Для использования делегата объявляется переменная этого делегата:

*Message mes; // 2. Создаем переменную делегата*

*mes = GoodMorning; // 3. Присваиваем этой переменной адрес метода*

Затем через делегат вызываем метод, на который ссылается данный делегат:

*mes(); // 4. Вызываем метод*

Вызов делегата производится подобно вызову метода. Делегаты необязательно могут указывать только на методы, которые определены в том же классе, где определена переменная делегата. Это могут быть также методы из других классов и структур. Выше переменной делегата напрямую присваивался метод. Есть еще один способ - создание объекта делегата с помощью конструктора, в который передается нужный метод.

*Operation del = Add;*

*Operation del2 = new Operation(Add);*

Как было написано выше, методы соответствуют делегату, если они имеют один и тот же возвращаемый тип, и один и тот же набор параметров. Но надо учитывать, что во внимание также принимаются модификаторы ref и out.

Делегат может указывать на множество методов, которые имеют ту же сигнатуру и возвращаемые тип. Все методы в делегате попадают в специальный список - список вызова или invocation list. И при вызове делегата все методы из этого списка последовательно вызываются. И мы можем добавлять в этот спиок не один, а несколько методов. Для добавления делегатов применяется операция +=. Однако стоит отметить, что в реальности будет происходить создание нового объекта делегата, который получит методы старой копии делегата и новый метод, и новый созданный объект делеагата будет присвоен переменной mes1. При добавлении делегатов следует учитывать, что мы можем добавить ссылку на один и тот же метод несколько раз, и в списке вызова делегата тогда будет несколько ссылок на один и то же метод. Соответственно при вызове делегата добавленный метод будет вызываться столько раз, сколько он был добавлен.

Подобным образом мы можем удалять методы из делегата с помощью операции -= При удалении методов из делегата фактически будет создаватья новый делегат, который в списке вызова методов будет содержать на один метод меньше. При удалении следует учитывать, что если делегат содержит несколько ссылок на один и тот же метод, то операция -= начинает поиск с конца списка вызова делегата и удаляет только первое найденное вхождение. Если подобного метода в списке вызова делегата нет, то операция -= не имеет никакого эффекта.

Делегаты можно объединять в другие делегаты.

*Message mes1 = Hello;*

*Message mes2 = HowAreYou;*

*Message mes3 = mes1 + mes2; // объединяем делегаты*

*mes3(); // вызываются все методы из mes1 и mes2*

В данном случае объект mes3 представляет объединение делегатов mes1 и mes2. Объединение делегатов значит, что в список вызова делегата mes3 попадут все методы из делегатов mes1 и mes2. И при вызове делегата mes3 все эти методы одновременно будут вызваны.

Вызова делегата есть 2 способа:

1. Делегат вызывался как обычный метод. Если делегат принимал параметры, то при ее вызове для параметров передавались необходимые значения.  
   *op(3, 4);*
2. Вызов делегата представляет метод Invoke()   
   *op.Invoke(3, 4);*

Следует учитывать, что если делегат пуст, то есть в его списке вызова нет ссылок ни на один из методов (то есть делегат равен Null), то при вызове такого делегата мы получим исключение. Поэтому при вызове делегата всегда лучше проверять, не равен ли он null. Либо можно использовать метод Invoke и оператор условного null.

*mes?.Invoke();*

Если делегат возвращает некоторое значение, то возвращается значение последнего метода из списка вызова (если в списке вызова несколько методов).

Также делегаты могут быть параметрами методов.

*class Program*

*{*

*delegate void GetMessage();*

*static void Main(string[] args)*

*{*

*if (DateTime.Now.Hour < 12)*

*{*

*Show\_Message(GoodMorning);*

*}*

*else*

*{*

*Show\_Message(GoodEvening);*

*}*

*Console.ReadLine();*

*}*

*private static void Show\_Message(GetMessage \_del)*

*{*

*\_del?.Invoke();*

*}*

*private static void GoodMorning()*

*{*

*Console.WriteLine("Good Morning");*

*}*

*private static void GoodEvening()*

*{*

*Console.WriteLine("Good Evening");*

*}*

*}*

Делегаты могут быть обобщенными

*delegate T Operation<T, K>(K val);*

*class Program*

*{*

*static void Main(string[] args)*

*{*

*Operation<decimal, int> op = Square;*

*Console.WriteLine(op(5));*

*Console.Read();*

*}*

*static decimal Square(int n)*

*{*

*return n \* n;*

*}*

*}*

***Action***

Делегат Action является обобщенным, принимает параметры и возвращает значение void:

*public delegate void Action<T>(T obj)*

Данный делегат имеет ряд перегруженных версий. Каждая версия принимает разное число параметров: от Action<in T1> до Action<in T1, in T2,....in T16>. Таким образом можно передать до 16 значений в метод.

Как правило, этот делегат передается в качестве параметра метода и предусматривает вызов определенных действий в ответ на произошедшие действия.

***Predicate***

Делегат Predicate<T>, как правило, используется для сравнения, сопоставления некоторого объекта T определенному условию. В качестве выходного результата возвращается значение true, если условие соблюдено, и false, если не соблюдено:

*Predicate<int> isPositive = delegate (int x) { return x > 0; };*

*Console.WriteLine(isPositive(20));*

*Console.WriteLine(isPositive(-20));*

В данном случае возвращается true или false в зависимости от того, больше нуля число или нет.

***Func***

Еще одним распространенным делегатом является Func. Он возвращает результат действия и может принимать параметры. Он также имеет различные формы: от Func<out T>(), где T - тип возвращаемого значения, до Func<in T1, in T2,...in T16, out TResult>(), то есть может принимать до 16 параметров. Данный делегат также часто используется в качестве параметра в методах.

***Анонимные методы***

Анонимные методы используются для создания экземпляров делегатов.иОпределение анонимных методов начинается с ключевого слова delegate, после которого идет в скобках список параметров и тело метода в фигурных скобках:

*delegate(параметры)*

*{*

*// инструкции*

*}*

Анонимный метод не может существовать сам по себе, он используется для инициализации экземпляра делегата. И через эту переменную делегата можно вызвать данный анонимный метод.

Другой пример анонимных методов - передача в качестве аргумента для параметра, который представляет делегат. Если анонимный метод использует параметры, то они должны соответствовать параметрам делегата. Если для анонимного метода не требуется параметров, то скобки с параметрами опускаются. При этом даже если делегат принимает несколько параметров, то в анонимном методе можно вовсе опустить параметры.

*class Program*

*{*

*delegate void MessageHandler(string message);*

*static void Main(string[] args)*

*{*

*MessageHandler handler = delegate*

*{*

*Console.WriteLine("анонимный метод");*

*};*

*handler("hello world!"); // анонимный метод*

*Console.Read();*

*}*

*}*

То есть если анонимный метод содержит параметры, они обязательно должны соответствовать параметрам делегата. Либо анонимный метод вообще может не содержать никаких параметров, тогда он соответствует любому делегату, который имеет тот же тип возвращаемого значения.

При этом параметры анонимного метода не могут быть опущены, если один или несколько параметров определены с модификатором out.

***Лямбды***

Лямбда-выражения представляют упрощенную запись анонимных методов. Лямбда-выражения позволяют создать емкие лаконичные методы, которые могут возвращать некоторое значение и которые можно передать в качестве параметров в другие методы. Ламбда-выражения имеют следующий синтаксис: слева от лямбда-оператора => определяется список параметров, а справа блок выражений, использующий эти параметры: (список\_параметров) => выражение.

Код (x, y) => x + y; представляет лямбда-выражение, где x и y - это параметры, а x + y - выражение. При этом нам не надо указывать тип параметров, а при возвращении результата не надо использовать оператор return. При этом надо учитывать, что каждый параметр в лямбда-выражении неявно преобразуется в соответствующий параметр делегата, поэтому типы параметров должны быть одинаковыми. Кроме того, количество параметров должно быть таким же, как и у делегата. И возвращаемое значение лямбда-выражений должно быть тем же, что и у делегата.

Бывает, что параметров не требуется. В этом случае вместо параметра в лямбда-выражении используются пустые скобки. Также бывает, что лямбда-выражение не возвращает никакого значения.

*Hello hello1 = () => Console.WriteLine("Hello");*

Необязательно указывать тип параметров у лямбда-выражения. Однако, нам обязательно нужно указывать тип, если делегат, которому должно соответствовать лямбда-выражение, имеет параметры с модификаторами ref и out.

Лямбда-выражения можно передавать в качестве аргументов методу для тех параметров, которые представляют делегат, что довольно удобно

*int result1 = Sum(integers, x => x > 5);*

**События**

События сигнализируют системе о том, что произошло определенное действие. И если нам надо отследить эти действия, то как раз мы можем применять события. События объявляются в классе с помощью ключевого слова event, после которого указывается тип делегата, который представляет событие:

*delegate void AccountHandler(string message);*

*event AccountHandler Notify;*

Определив событие, мы можем его вызвать в программе как метод, используя имя события:

*Notify("Произошло действие");*

*Notify?.Invoke("Произошло действие");*

С событием может быть связан один или несколько обработчиков. Обработчики событий - это именно то, что выполняется при вызове событий. Нередко в качестве обработчиков событий применяются методы. Каждый обработчик событий по списку параметров и возвращаемому типу должен соответствовать делегату, который представляет событие. Для добавления обработчика события применяется операция +=. Для одного события можно установить несколько обработчиков и потом в любой момент времени их удалить. Для удаления обработчиков применяется операция -=. В качестве обработчиков могут использоваться не только обычные методы, но также делегаты, анонимные методы и лямбда-выражения.

С помощью специальных акссесоров add/remove мы можем управлять добавлением и удалением обработчиков. Как правило, подобная функциональность редко требуется, но тем не менее мы ее можем использовать.

*class Account*

*{*

*public delegate void AccountHandler(string message);*

*private event AccountHandler \_notify;*

*public event AccountHandler Notify*

*{*

*add*

*{*

*\_notify += value;*

*Console.WriteLine($"{value.Method.Name} добавлен");*

*}*

*remove*

*{*

*\_notify -= value;*

*Console.WriteLine($"{value.Method.Name} удален");*

*}*

*}*

*public Account(int sum)*

*{*

*Sum = sum;*

*}*

*public int Sum { get; private set;}*

*public void Put(int sum)*

*{*

*Sum += sum;*

*\_notify?.Invoke($"На счет поступило: {sum}");*

*}*

*public void Take(int sum)*

*{*

*if (Sum >= sum)*

*{*

*Sum -= sum;*

*\_notify?.Invoke($"Со счета снято: {sum}");*

*}*

*else*

*{*

*\_notify?.Invoke($"Недостаточно денег на счете. Текущий баланс: {Sum}"); ;*

*}*

*}*

*}*

Аксессор add вызывается при добавлении обработчика, то есть при операции +=. Добавляемый обработчик доступен через ключевое слово value. Здесь мы можем получить информацию об обработчике (например, имя метода через value.Method.Name) и определить некоторую логику. В данном случае для простоты просто выводится сообщение на консоль:

Блок remove вызывается при удалении обработчика. Аналогично здесь можно задать некоторую дополнительную логику. Внутри класса событие вызывается также через переменную \_notify. Но для добавления и удаления обработчиков в программе используется как раз Notify.

**LINQ**

LINQ (Language-Integrated Query) представляет простой и удобный язык запросов к источнику данных.

Существует несколько разновидностей LINQ:

* LINQ to Objects: применяется для работы с массивами и коллекциями
* LINQ to Entities: используется при обращении к базам данных через технологию Entity Framework
* LINQ to Sql: технология доступа к данным в MS SQL Server
* LINQ to XML: применяется при работе с файлами XML
* LINQ to DataSet: применяется при работе с объектом DataSet
* Parallel LINQ (PLINQ): используется для выполнения параллельных запросов

Чтобы использовать функциональность LINQ, убедимся, что в файле подключено пространство имен System.LINQ.

Простейшее определение запроса LINQ выглядит следующим образом:

*from переменная in набор\_объектов*

*select переменная;*

Выражение from t in teams проходит по всем элементам массива teams и определяет каждый элемент как t. Используя переменную t мы можем проводить над ней разные операции. Несмотря на то, что мы не указываем тип переменной t, выражения LINQ являются строго типизированными. То есть среда автоматически распознает, что набор teams состоит из объектов string, поэтому переменная t будет рассматриваться в качестве строки. Далее с помощью оператора where проводится фильтрация объектов, и если объект соответствует критерию (в данном случае начальная буква должна быть "Б"), то этот объект передается дальше. Оператор orderby упорядочивает по возрастанию, то есть сортирует выбранные объекты. Оператор select передает выбранные значения в результирующую выборку, которая возвращается LINQ-выражением. В данном случае результатом выражения LINQ является объект IEnumerable<T>. Нередко результирующая выборка определяется с помощью ключевого слова var, тогда компилятор на этапе компиляции сам выводит тип. Преимуществом подобных запросов также является и то, что они интуитивно похожи на запросы языка SQL, хотя и имеют некоторые отличия.

*Методы расширения LINQ*

Кроме стандартного синтаксиса from .. in .. select для создания запроса LINQ мы можем применять специальные методы расширения, которые определены для интерфейса IEnumerable. Как правило, эти методы реализуют ту же функциональность, что и операторы LINQ типа where или orderby.

Список используемых методов расширения LINQ

* Select: определяет проекцию выбранных значений
* Where: определяет фильтр выборки
* OrderBy: упорядочивает элементы по возрастанию
* OrderByDescending: упорядочивает элементы по убыванию
* ThenBy: задает дополнительные критерии для упорядочивания элементов возрастанию
* ThenByDescending: задает дополнительные критерии для упорядочивания элементов по убыванию
* Join: соединяет две коллекции по определенному признаку
* GroupBy: группирует элементы по ключу
* ToLookup: группирует элементы по ключу, при этом все элементы добавляются в словарь
* GroupJoin: выполняет одновременно соединение коллекций и группировку элементов по ключу
* Reverse: располагает элементы в обратном порядке
* All: определяет, все ли элементы коллекции удовлятворяют определенному условию
* Any: определяет, удовлетворяет хотя бы один элемент коллекции определенному условию
* Contains: определяет, содержит ли коллекция определенный элемент
* Distinct: удаляет дублирующиеся элементы из коллекции
* Except: возвращает разность двух коллекцию, то есть те элементы, которые создаются только в одной коллекции
* Union: объединяет две однородные коллекции
* Intersect: возвращает пересечение двух коллекций, то есть те элементы, которые встречаются в обоих коллекциях
* Count: подсчитывает количество элементов коллекции, которые удовлетворяют определенному условию
* Sum: подсчитывает сумму числовых значений в коллекции
* Average: подсчитывает cреднее значение числовых значений в коллекции
* Min: находит минимальное значение
* Max: находит максимальное значение
* Take: выбирает определенное количество элементов
* Skip: пропускает определенное количество элементов
* TakeWhile: возвращает цепочку элементов последовательности, до тех пор, пока условие истинно
* SkipWhile: пропускает элементы в последовательности, пока они удовлетворяют заданному условию, и затем возвращает оставшиеся элементы
* Concat: объединяет две коллекции
* Zip: объединяет две коллекции в соответствии с определенным условием
* First: выбирает первый элемент коллекции
* FirstOrDefault: выбирает первый элемент коллекции или возвращает значение по умолчанию
* Single: выбирает единственный элемент коллекции, если коллекция содердит больше или меньше одного элемента, то генерируется исключение
* SingleOrDefault: выбирает первый элемент коллекции или возвращает значение по умолчанию
* ElementAt: выбирает элемент последовательности по определенному индексу
* ElementAtOrDefault: выбирает элемент коллекции по определенному индексу или возвращает значение по умолчанию, если индекс вне допустимого диапазона
* Last: выбирает последний элемент коллекции
* LastOrDefault: выбирает последний элемент коллекции или возвращает значение по умолчанию

Есть два способа выполнения запроса LINQ: отложенное и немедленное выполнение.

При отложенном выполнении LINQ-выражение не выполняется, пока не будет произведена итерация или перебор по выборке.

Фактически LINQ-запрос разбивается на три этапа:

1. Получение источника данных
2. Создание запроса
3. Выполнение запроса и получение его результатов

С помощью ряда методов мы можем применить немедленное выполнение запроса. Это методы, которые возвращают одно атомарное значение или один элемент. Например, Count(), Average(), First() / FirstOrDefault(), Min(), Max() и т.д. Например, метод Count() возвращает числовое значение, которое представляет количество элементов в полученной последовательности. А метод First() возвращает первый элемент последовательности. Но чтобы выполнить эти методы, вначале надо получить саму последовательность, то есть результат запроса, и пройтись по ней циклом foreach, который вызывается неявно внутри структуры запроса.

Метод Aggregate выполняет общую агрегацию элементов коллекции в зависимости от указанного выражения. Например:

*int[] numbers = { 1, 2, 3, 4, 5};*

*int query = numbers.Aggregate((x,y)=> x - y);*

Переменная query будет представлять результат агрегации массива.

**Структура DateTime**

Для создания нового объекта DateTime также можно использовать конструктор. Пустой конструктор создает начальную дату:

*DateTime date1 = new DateTime();*

*Console.WriteLine(date1); // 01.01.0001 0:00:00*

То есть мы получим минимально возможное значение, которое также можно получить следующим образом:

*Console.WriteLine(DateTime.MinValue);*

Чтобы задать конкретную дату, нужно использовать один из конструкторов, принимающих параметры:

*DateTime date1 = new DateTime(2015, 7, 20); // год - месяц - день*

*Console.WriteLine(date1); // 20.07.2015 0:00:00*

Если необходимо получить текущую время и дату, то можно использовать ряд свойств DateTime:

*Console.WriteLine(DateTime.Now);*

*Console.WriteLine(DateTime.UtcNow);*

*Console.WriteLine(DateTime.Today);*

Свойство DateTime.Now берет текущую дату и время компьютера, DateTime.UtcNow - дата и время относительно времени по Гринвичу (GMT) и DateTime.Today - только текущая дата.

Основные операции со структурой DateTime связаны со сложением или вычитанием дат. Например, надо к некоторой дате прибавить или, наоборот, отнять несколько дней.

Для добавления дат используется ряд методов:

* Add(TimeSpan value): добавляет к дате значение TimeSpan
* AddDays(double value): добавляет к текущей дате несколько дней
* AddHours(double value): добавляет к текущей дате несколько часов
* AddMinutes(double value): добавляет к текущей дате несколько минут
* AddMonths(int value): добавляет к текущей дате несколько месяцев
* AddYears(int value): добавляет к текущей дате несколько лет

Для вычитания дат используется метод Substract(DateTime date):

Метод Substract не имеет возможностей для отдельного вычитания дней, часов и так далее. Но это и не надо, так как мы можем передавать в метод AddDays() и другие методы добавления отрицательные значения.

Кроме операций сложения и вычитания еще есть ряд методов форматирования дат:

*DateTime date1 = new DateTime(2015, 7, 20, 18, 30, 25);*

*Console.WriteLine(date1.ToLocalTime()); // 20.07.2015 21:30:25*

*Console.WriteLine(date1.ToUniversalTime()); // 20.07.2015 15:30:25*

*Console.WriteLine(date1.ToLongDateString()); // 20 июля 2015 г.*

*Console.WriteLine(date1.ToShortDateString()); // 20.07.2015*

*Console.WriteLine(date1.ToLongTimeString()); // 18:30:25*

*Console.WriteLine(date1.ToShortTimeString()); // 18:30*

Метод ToLocalTime() преобразует время UTC в локальное время, добавляя смещение относительно времени по Гринвичу. Метод ToUniversalTime(), наоборот, преобразует локальное время во время UTC, то есть вычитает смещение относительно времени по Гринвичу. Остальные методы преобразуют дату к определенному формату.

Для форматирования вывода дат и времени применяется ряд строковых форматов:

|  |  |
| --- | --- |
| D | Поный формат даты. Например, 17 июля 2015 г. |
| d | Краткий формат даты. Например, 17.07.2015 |
| F | Полный формат даты и времени. Например, 17 июля 2015 г. 17:04:43 |
| f | Полный формат даты и краткий формат времени. Например, 17 июля 2015 г. 17:04 |
| G | Краткий формат даты и полный формат времени. Например, 17.07.2015 17:04:43 |
| g | Краткий формат даты и времени. Например, 17.07.2015 17:04 |
| M, m | Формат дней месяца. Например, 17 июля |
| O, o | Формат обратного преобразования даты и времени. Вывод даты и времени в соответствии со стандартом ISO 8601 в формате "yyyy'-'MM'-'dd'T'HH':'mm':'ss'.'fffffffzzz". Например, 2015-07-17T17:04:43.4092892+03:00 |
| R, r | Время по Гринвичу. Например, Fri, 17 Jul 2015 17:04:43 GMT |
| s | Сортируемый формат даты и времени. Например, 2015-07-17T17:04:43 |
| T | Полный формат времени. Например, 17:04:43 |
| t | Краткий формат времени. Например, 17:04 |
| U | Полный универсальный полный формат даты и времени. Например, 17 июля 2015 г. 17:04:43 |
| u | Краткий универсальный полный формат даты и времени. Например, 2015-07-17 17:04:43Z |
| Y, y | Формат года. Например, Июль 2015 |

Не всегда удобно использование встроенных форматов даты и времени. Иногда бывает необходимо задать сообственную форму отображения объекта DateTime. В этом случае мы можем составить свой формат из описателей.

|  |  |
| --- | --- |
| d | Представляет день месяца от 1 до 31. Одноразрядные числа используются без нуля в начале |
| dd | Представляет день месяца от 1 до 31. К одноразрядным числам в начале добавляется ноль |
| ddd | Сокращенное название дня недели |
| dddd | Полное название дня недели |
| f / fffffff | Представляет миллисекунды. Количество символов f указывает на число разрядов в миллисекундах |
| g | Представляет период или эру (например, "н. э.") |
| h | Часы в виде от 1 до 12. Часы с одной цифрой не дополняются нулем |
| hh | Часы в виде от 01 до 12. Часы с одной цифрой дополняются нулем |
| H | Часы в виде от 0 до 23. Часы с одной цифрой не дополняются нулем |
| HH | Часы в виде от 0 до 23. Часы с одной цифрой дополняются нулем |
| K | Часовой пояс |
| m | Минуты от 0 до 59. Минуты с одной цифрой не дополняются начальным нулем |
| mm | Минуты от 0 до 59. Минуты с одной цифрой дополняются начальным нулем |
| M | Месяц в виде от 1 до 12 |
| MM | Месяц в виде от 1 до 12. Месяц с одной цифрой дополняется начальным нулем |
| MMM | Сокращенное название месяца |
| MMMM | Полное название месяца |
| s | Секунды в виде числа от 0 до 59. Секунды с одной цифрой не дополняются начальным нулем |
| ss | Секунды в виде числа от 0 до 59. Секунды с одной цифрой дополняются начальным нулем |
| t | Первые символы в обозначениях AM и PM |
| tt | AM или PM |
| y | Представляет год как число из одной или двух цифр. Если год имеет более двух цифр, то в результате отображаются только две младшие цифры |
| yy | Представляет год как число из одной или двух цифр. Если год имеет более двух цифр, то в результате отображаются только две младшие цифры. Если год имеет одну цифру, то он дополняется начальным нулем |
| yyy | Год из трех цифр |
| yyyy | Год из четырех цифр |
| yyyyy | Год из пяти цифр. Если в году меньше пяти цифр, то он дополняется начальными нулями |
| z | Представляет смецщение в часах относительно времени UTC |
| zz | Представляет смецщение в часах относительно времени UTC. Если смещение представляет одну цифру, то она дополняется начальным нулем. |

**Работа с дисками**

Работу с файловой системой начнем с самого верхнего уровня - дисков. Для представления диска в пространстве имен System.IO имеется класс DriveInfo.

Этот класс имеет статический метод GetDrives, который возвращает имена всех логических дисков компьютера. Также он предоставляет ряд полезных свойств:

* AvailableFreeSpace: указывает на объем доступного свободного места на диске в байтах
* DriveFormat: получает имя файловой системы
* DriveType: представляет тип диска
* IsReady: готов ли диск (например, DVD-диск может быть не вставлен в дисковод)
* Name: получает имя диска
* TotalFreeSpace: получает общий объем свободного места на диске в байтах
* TotalSize: общий размер диска в байтах
* VolumeLabel: получает или устанавливает метку тома

**Работа с каталогами**

Для работы с каталогами в пространстве имен System.IO предназначены сразу два класса: Directory и DirectoryInfo.

*Класс Directory*

Класс Directory предоставляет ряд статических методов для управления каталогами. Некоторые из этих методов:

* CreateDirectory(path): создает каталог по указанному пути path
* Delete(path): удаляет каталог по указанному пути path
* Exists(path): определяет, существует ли каталог по указанному пути path. Если существует, возвращается true, если не существует, то false
* GetDirectories(path): получает список каталогов в каталоге path
* GetFiles(path): получает список файлов в каталоге path
* Move(sourceDirName, destDirName): перемещает каталог
* GetParent(path): получение родительского каталога

*Класс DirectoryInfo*

Данный класс предоставляет функциональность для создания, удаления, перемещения и других операций с каталогами. Во многом он похож на Directory. Некоторые из его свойств и методов:

* Create(): создает каталог
* CreateSubdirectory(path): создает подкаталог по указанному пути path
* Delete(): удаляет каталог
* Свойство Exists: определяет, существует ли каталог
* GetDirectories(): получает список каталогов
* GetFiles(): получает список файлов
* MoveTo(destDirName): перемещает каталог
* Свойство Parent: получение родительского каталога
* Свойство Root: получение корневого каталога

Если мы просто применим метод Delete к непустой папке, в которой есть какие-нибудь файлы или подкаталоги, то приложение нам выбросит ошибку. Поэтому нам надо передать в метод Delete дополнительный параметр булевого типа, который укажет, что папку надо удалять со всем содержимым.

**Работа с файлами.**

для работы с файлами предназначена пара классов File и FileInfo. С их помощью мы можем создавать, удалять, перемещать файлы, получать их свойства и многое другое.

Некоторые полезные методы и свойства *класса FileInfo:*

* CopyTo(path): копирует файл в новое место по указанному пути path
* Create(): создает файл
* Delete(): удаляет файл
* MoveTo(destFileName): перемещает файл в новое место
* Свойство Directory: получает родительский каталог в виде объекта DirectoryInfo
* Свойство DirectoryName: получает полный путь к родительскому каталогу
* Свойство Exists: указывает, существует ли файл
* Свойство Length: получает размер файла
* Свойство Extension: получает расширение файла
* Свойство Name: получает имя файла
* Свойство FullName: получает полное имя файла

*Класс File* реализует похожую функциональность с помощью статических методов:

* Copy(): копирует файл в новое место
* Create(): создает файл
* Delete(): удаляет файл
* Move: перемещает файл в новое место
* Exists(file): определяет, существует ли файл

Метод CopyTo класса FileInfo принимает два параметра: путь, по которому файл будет копироваться, и булевое значение, которое указывает, надо ли при копировании перезаписывать файл (если true, как в случае выше, файл при копировании перезаписывается). Если же в качестве последнего параметра передать значение false, то если такой файл уже существует, приложение выдаст ошибку. Метод Copy класса File принимает три параметра: путь к исходному файлу, путь, по которому файл будет копироваться, и булевое значение, указывающее, будет ли файл перезаписываться.

***Чтение и запись файла***

Класс FileStream представляет возможности по считыванию из файла и записи в файл. Он позволяет работать как с текстовыми файлами, так и с бинарными.

Для создания объекта FileStream можно использовать как конструкторы этого класса, так и статические методы класса File. Конструктор FileStream имеет множество перегруженных версий, из которых отмечу лишь одну, самую простую и используемую:

*FileStream(string filename, FileMode mode)*

Здесь в конструктор передается два параметра: путь к файлу и перечисление FileMode. Данное перечисление указывает на режим доступа к файлу и может принимать следующие значения:

* Append: если файл существует, то текст добавляется в конец файл. Если файла нет, то он создается. Файл открывается только для записи.
* Create: создается новый файл. Если такой файл уже существует, то он перезаписывается
* CreateNew: создается новый файл. Если такой файл уже существует, то он приложение выбрасывает ошибку
* Open: открывает файл. Если файл не существует, выбрасывается исключение
* OpenOrCreate: если файл существует, он открывается, если нет - создается новый
* Truncate: если файл существует, то он перезаписывается. Файл открывается только для записи.

Другой способ создания объекта FileStream представляют статические методы класса File:

*FileStream File.Open(string file, FileMode mode);*

*FileStream File.OpenRead(string file);*

*FileStream File.OpenWrite(string file);*

Первый метод открывает файл с учетом объекта FileMode и возвращает файловой поток FileStream. У этого метода также есть несколько перегруженных версий. Второй метод открывает поток для чтения, а третий открывает поток для записи.

*Свойства и методы FileStream*

Рассмотрим наиболее важные его свойства и методы класса FileStream:

* Свойство Length: возвращает длину потока в байтах
* Свойство Position: возвращает текущую позицию в потоке
* void CopyTo(Stream destination): копирует данные из текущего потока в поток destination
* Task CopyToAsync(Stream destination): асинхронная версия метода CopyToAsync
* int Read(byte[] array, int offset, int count): считывает данные из файла в массив байтов и возвращает количество успешно считанных байтов. Принимает три параметра:

array - массив байтов, куда будут помещены считываемые из файла данные

offset представляет смещение в байтах в массиве array, в который считанные байты будут помещены

count - максимальное число байтов, предназначенных для чтения. Если в файле находится меньшее количество байтов, то все они будут считаны.

* Task<int> ReadAsync(byte[] array, int offset, int count): асинхронная версия метода Read
* long Seek(long offset, SeekOrigin origin): устанавливает позицию в потоке со смещением на количество байт, указанных в параметре offset.
* void Write(byte[] array, int offset, int count): записывает в файл данные из массива байтов. Принимает три параметра:

array - массив байтов, откуда данные будут записываться в файл

offset - смещение в байтах в массиве array, откуда начинается запись байтов в поток

count - максимальное число байтов, предназначенных для записи

* ValueTask WriteAsync(byte[] array, int offset, int count): асинхронная версия метода Write

FileStream представляет доступ к файлам на уровне байтов, поэтому, например, если вам надо считать или записать одну или несколько строк в текстовый файл, то массив байтов надо преобразовать в строки, используя специальные методы. Поэтому для работы с текстовыми файлами применяются другие классы. В то же время при работе с различными бинарными файлами, имеющими определенную структуру, FileStream может быть очень даже полезен для извлечения определенных порций информации и ее обработки.

Оператор using позволяет создавать объект в блоке кода, по завершению которого вызывается метод Dispose у этого объекта, и, таким образом, объект уничтожается. В данном случае в качестве такого объекта служит переменная fstream.

***Чтение и запись текстовых файлов. StreamReader и StreamWriter***

Класс FileStream не очень удобно применять для работы с текстовыми файлами. К тому же для этого в пространстве System.IO определены специальные классы: StreamReader и StreamWriter.

***Запись в файл и StreamWriter***

Для записи в текстовый файл используется класс StreamWriter. Некоторые из его конструкторов, которые могут применяться для создания объекта StreamWriter:

* StreamWriter(string path): через параметр path передается путь к файлу, который будет связан с потоком
* StreamWriter(string path, bool append): параметр append указывает, надо ли добавлять в конец файла данные или же перезаписывать файл. Если равно true, то новые данные добавляются в конец файла. Если равно false, то файл перезаписываетсяя заново
* StreamWriter(string path, bool append, System.Text.Encoding encoding): параметр encoding указывает на кодировку, которая будет применяться при записи

Свою функциональность StreamWriter реализует через следующие методы:

* int Close(): закрывает записываемый файл и освобождает все ресурсы
* void Flush(): записывает в файл оставшиеся в буфере данные и очищает буфер.
* Task FlushAsync(): асинхронная версия метода Flush
* void Write(string value): записывает в файл данные простейших типов, как int, double, char, string и т.д. Соответственно имеет ряд перегруженных версий для записи данных элементарных типов, например, Write(char value), Write(int value), Write(double value) и т.д.
* Task WriteAsync(string value): асинхронная версия метода Write
* void WriteLine(string value): также записывает данные, только после записи добавляет в файл символ окончания строки
* Task WriteLineAsync(string value): асинхронная версия метода WriteLine

Чтение из файла и StreamReader

***Класс StreamReader*** позволяет нам легко считывать весь текст или отдельные строки из текстового файла.

Некоторые из конструкторов класса StreamReader:

* StreamReader(string path): через параметр path передается путь к считываемому файлу
* StreamReader(string path, System.Text.Encoding encoding): параметр encoding задает кодировку для чтения файла

Среди методов StreamReader можно выделить следующие:

* void Close(): закрывает считываемый файл и освобождает все ресурсы
* int Peek(): возвращает следующий доступный символ, если символов больше нет, то возвращает -1
* int Read(): считывает и возвращает следующий символ в численном представлении. Имеет перегруженную версию: Read(char[] array, int index, int count), где array - массив, куда считываются символы, index - индекс в массиве array, начиная с которого записываются считываемые символы, и count - максимальное количество считываемых символов
* Task<int> ReadAsync(): асинхронная версия метода Read
* string ReadLine(): считывает одну строку в файле
* string ReadLineAsync(): асинхронная версия метода ReadLine
* string ReadToEnd(): считывает весь текст из файла
* string ReadToEndAsync(): асинхронная версия метода ReadToEnd

Для работы с бинарными файлами предназначена ***пара классов BinaryWriter и BinaryReader***. Эти классы позволяют читать и записывать данные в двоичном формате.

Основные метода класса BinaryWriter

* Close (): закрывает поток и освобождает ресурсы
* Flush (): очищает буфер, дописывая из него оставшиеся данные в файл
* Seek (): устанавливает позицию в потоке
* Write (): записывает данные в поток

Основные метода класса BinaryReader

* Close (): закрывает поток и освобождает ресурсы
* ReadBoolean (): считывает значение bool и перемещает указатель на один байт
* ReadByte (): считывает один байт и перемещает указатель на один байт
* ReadChar (): считывает значение char, то есть один символ, и перемещает указатель на столько байтов, сколько занимает символ в текущей кодировке
* ReadDecimal (): считывает значение decimal и перемещает указатель на 16 байт
* ReadDouble (): считывает значение double и перемещает указатель на 8 байт
* ReadInt16(): считывает значение short и перемещает указатель на 2 байта
* ReadInt32(): считывает значение int и перемещает указатель на 4 байта
* ReadInt64(): считывает значение long и перемещает указатель на 8 байт
* ReadSingle (): считывает значение float и перемещает указатель на 4 байта
* ReadString (): считывает значение string. Каждая строка предваряется значением длины строки, которое представляет 7-битное целое число

**Сериализация**

Термин сериализация описывает процесс сохранения (и, возможно, передачи) состояния объекта в потоке (т.е. файловом потоке и потоке в памяти). Последовательность сохраняемых данных содержит всю информацию, необходимую для реконструкции (или десериализации) состояния объекта с целью последующего использования. Применяя эту технологию, очень просто сохранять большие объемы данных (в различных форматах) с минимальными усилиями.

***Атрибут Serializable***

Чтобы объект определенного класса можно было сериализовать, надо этот класс пометить атрибутом Serializable. При отсутствии данного атрибута объект не сможет быть сериализован, и при попытке сериализация будет выброшено исключение SerializationException.

Сериализация применяется к свойствам и полям класса. Если мы не хотим, чтобы какое-то поле класса сериализовалось, то мы его помечаем атрибутом NonSerialized.

*[Serializable]*

*class Person*

*{*

*public string Name { get; set; }*

*public int Year { get; set; }*

*[NonSerialized]*

*public string accNumber;*

*public Person(string name, int year, string acc)*

*{*

*Name = name;*

*Year = year;*

*accNumber = acc;*

*}*

*}*

При наследовании подобного класса, следует учитывать, что атрибут Serializable автоматически не наследуется. И если мы хотим, чтобы производный класс также мог бы быть сериализован, то опять же мы применяем к нему атрибут.

Обратите внимание, что в каждом из этих классов поля данных определены как public, это сделано для упрощения примера. Конечно, приватные данные, представленные общедоступными свойствами, были бы более предпочтительны с точки зрения ООП. Если вы сохраняете состояние объекта, используя BinaryFormatter или SoapFormatter, то разницы никакой. Эти типы запрограммированы для сериализации всех сериализуемых полей типа, независимо от того, представлены они общедоступными полями, приватными полями или приватными полями с соответствующими общедоступными свойствами. Однако ситуация существенно меняется, если вы собираетесь применять тип XmlSerializer. Этот тип будет сериализовать только сериализуемые общедоступные поля данных или приватные поля, представленные общедоступными свойствами. Приватные данные, не представленные свойствами, просто игнорируются.

***Формат сериализации***

Хотя сериализация представляет собой преобразование объекта в некоторый набор байтов, но в действительности только бинарным форматом она не ограничивается. Итак, в .NET можно использовать следующие форматы:

* Бинарный. Тип BinaryFormatter сериализует состояние объекта в поток, используя компактный двоичный формат.
* SOAP. Тип SoapFormatter сохраняет состояние объекта в виде сообщения SOAP (стандартный XML-формат для передачи и приема сообщений от веб-служб).
* Xml. Для сохранения дерева объектов в документе XML имеется тип XmlSerializer.
* JSON. Для сохранения объектов в виде JSON используется DataContractJsonSerializer.

***IFormatter***

Независимо от того, какой форматер выбран, имейте в виду, каждый из них наследуется непосредственно от System.Object, так что они не разделяют общего набора членов от какого-то базового класса сериализации. Однако типы BinaryFormatter и SoapFormatter поддерживают общие члены через реализацию интерфейсов IFormatter и IRemotingFormatter (как ни странно, XmlSerializer не реализует ни одного из них).

Для классов BinaryFormatter и SoapFormatter сам функционал сериализации определен в интерфейсе IFormatter. IFormatter определены основные методы Serialize() и Deserialize(), которые выполняют черновую работу по перемещению графов объектов в определенный поток и обратно. Помимо этих членов в IFormatter определено несколько свойств, используемых "за кулисами" реализующим типом:

*public interface IFormatter*

*{*

*SerializationBinder Binder { get; set;}*

*StreamingContext Context { get; set;}*

*ISurrogateSelector SurrogateSelector { get; set;}*

*object Deserialize (Stream serializationStream);*

*void Serialize (Stream serializationStream, object graph);*

*}*

Для бинарной сериализации применяется класс BinaryFormatter.

Сериализация выполняется двумя ключевыми методами типа **BinaryFormatter**, о которых следует знать, являются Serialize() и Deserialize():

* Serialize() сохраняет граф объектов в указанный поток в виде последовательности байтов;
* Deserialize() преобразует сохраненную последовательность байт в граф объектов. Deserialize() возвращает объект общего типа System.Object, так что понадобится применить явное приведение.

**SoapFormatter**, сериализующий данные в подходящем конверте SOAP. Протокол SOAP (Simple Object Access Protocol — простой протокол доступа к объектам) определяет стандартный процесс вызова методов в независящей от платформы и операционной системы манере.

Как и в BinaryFormatter сериализация выполняется двумя ключевыми методами:

* Serialize() сохраняет граф объектов в указанный поток в виде последовательности байтов;
* Deserialize() преобразует сохраненную последовательность байт в граф объектов. Deserialize() возвращает объект общего типа System.Object, так что понадобится применить явное приведение.

В дополнение к двоичному форматеру и форматеру SOAP сборка System.Xml.dll предлагает третий класс форматера — System.Xml.Serialization.**XmlSerializer** — который может использоваться для сохранения общедоступного состояния заданного объекта в виде чистой XML-разметки, в противоположность данным XML внутри сообщения SOAP.

Тип XmlSerializer требует указания информации о типе, представляющей класс, который необходимо сериализовать.

Класс XmlSerializer требует, чтобы все сериализованные типы в графе объектов поддерживали конструктор по умолчанию (поэтому не забудьте его добавить, если определяли специальные конструкторы). Если этого не сделать, во время выполнения сгенерируется исключение InvalidOperationException.

По умолчанию класс XmlSerializer сериализует все общедоступные поля/свойства как элементы XML, а не как атрибуты XML. Чтобы управлять генерацией результирующего документа XML с помощью класса XmlSerializer, необходимо декорировать типы любым количеством дополнительных атрибутов из пространства имен System.Xml.Serialization. Ниже документированы некоторые (но не все) атрибуты, которые влияют на кодирование данных XML в потоке:

[XmlAttribute]

Поле или свойство будет сериализовано как атрибут XML (а не как подэлемент)

[XmlElement]

Поле или свойство будет сериализовано как элемент XML с указанным именем

[XmlEnum]

Этот атрибут предоставляет имя элемента, являющееся членом перечисления

[XmlRoot]

Этот атрибут управляет тем, как будет сконструирован корневой элемент (пространство имен и название элемента)

[XmlText]

Свойство или поле должно быть сериализовано как текст XML (т.е. содержимое, находящееся между начальным и конечным дескрипторами корневого элемента)

[XmlType]

Этот атрибут предоставляет имя и пространство имен типа XML

Метод Serialize добавляет данные в файл. А метод Deserialize извлекает их оттуда. Равным образом мы можем сериализовать массив или коллекцию объектов, но главное требование состоит в том, чтобы в них был определен стандартный конструктор.

Ключевым типом является класс **JsonSerializer,** который и позволяет сериализовать объект в json и, наоборот, десериализовать код json в объект C#. Для сохранения объекта в json в классе JsonSerializer определен статический метод Serialize(), который имеет ряд перегруженных версий. Затем обратно получаем из этой строки объект Person посредством метода JsonSerializer.Deserialize(). Объект, который подвергается десериализации, должен иметь конструктор без параметров. Сериализации подлежат только публичные свойства объекта (с модификатором public).

По умолчанию сериализации подлежат все публичные свойства. Кроме того, в выходном объекте json все названия свойств соответствуют названиям свойств объекта C#. Однако с помощью атрибутов JsonIgnore и JsonPropertyName. Атрибут JsonIgnore позволяет исключить из сериализации определенное свойство. А JsonPropertyName позволяет замещать оригинальное название свойства.

**Коллекции**

В C# коллекция представляет собой совокупность объектов.

Необобщенные коллекции - Реализуют ряд основных структур данных, включая динамический массив, стек, очередь, а также словари, в которых можно хранить пары "ключ-значение". В отношении необобщенных коллекций важно иметь в виду следующее: они оперируют данными типа object. Таким образом, необобщенные коллекции могут служить для хранения данных любого типа, причем в одной коллекции допускается наличие разнотипных данных. Очевидно, что такие коллекции не типизированы, поскольку в них хранятся ссылки на данные типа object. Классы и интерфейсы необобщенных коллекций находятся в пространстве имен System.Collections.

Специальные коллекции - Оперируют данными конкретного типа или же делают это каким-то особым образом. Например, имеются специальные коллекции для символьных строк, а также специальные коллекции, в которых используется однонаправленный список. Специальные коллекции объявляются в пространстве имен System.Collections.Specialized.

Обобщенные коллекции - Обеспечивают обобщенную реализацию нескольких стандартных структур данных, включая связные списки, стеки, очереди и словари. Такие коллекции являются типизированными в силу их обобщенного характера. Это означает, что в обобщенной коллекции могут храниться только такие элементы данных, которые совместимы по типу с данной коллекцией. Благодаря этому исключается случайное несовпадение типов. Обобщенные коллекции объявляются в пространстве имен System.Collections.Generic.

Основополагающим для всех коллекций является понятие перечислителя, который поддерживается в необобщенных интерфейсах IEnumerator и IEnumerable, а также в обобщенных интерфейсах IEnumerator<T> и IEnumerable<T>. Перечислитель обеспечивает стандартный способ поочередного доступа к элементам коллекции. Следовательно, он перечисляет содержимое коллекции. В каждой коллекции должна быть реализована обобщенная или необобщенная форма интерфейса IEnumerable, поэтому элементы любого класса коллекции должны быть доступны посредством методов, определенных в интерфейсе IEnumerator или IEnumerator<T>. Это означает, что, внеся минимальные изменения в код циклического обращения к коллекции одного типа, его можно использовать для аналогичного обращения к коллекции другого типа. Любопытно, что для поочередного обращения к содержимой коллекции в цикле foreach используется перечислитель.

|  |  |
| --- | --- |
| Необобщенный | Обобщенные |
| Структура DictionaryEntry  В пространстве имен System.Collections определена структура DictionaryEntry. Необобщенные коллекции пар "ключ-значение" сохраняют эти пары в объекте типа DictionaryEntry. В данной структуре определяются два следующих свойства:  *public object Key { get; set; }*  *public object Value { get; set; }*  Эти свойства служат для доступа к ключу или значению, связанному с элементом коллекции. Объект типа DictionaryEntry может быть сконструирован с помощью конструктора:  *public DictionaryEntry(object key, object value)*  где key обозначает ключ, a value — значение. | Структура KeyValuePair<TKey, TValue>  В пространстве имен System.Collections.Generic определена структура KeyValuePair<TKey, TValue> Она служит для хранения ключа и его значения и применяется в классах обобщенных коллекций, в которых хранятся пары "ключ-значение", как, например, в классе Dictionary<TKey, TValue> В этой структуре определяются два следующих свойства:  *public ТКеу Key { get; };*  *public TValue Value { get; };*  В этих свойствах хранятся ключ и значение соответствующего элемента коллекции. Для построения объекта типа KeyValuePair<TKey, TValue> служит конструктор:  *public KeyValuePair(ТКеу key, TValue value)*  где key обозначает ключ, a value — значение. |

***IEnumerator***

К элементам коллекции нередко приходится обращаться циклически, например, для отображения каждого элемента коллекции. С этой целью можно, с одной стороны, организовать цикл foreach, а с другой — воспользоваться перечислителем. Перечислитель — это объект, который реализует необобщенный интерфейс IEnumerator или обобщенный интерфейс IEnumerator<T>.

В интерфейсе IEnumerator определяется одно свойство, Current, необобщенная форма которого приведена ниже:

*object Current { get; }*

А в интерфейсе IEnumerator<T> объявляется следующая обобщенная форма свойства Current:

*Т Current { get; }*

В обеих формах свойства Current получается текущий перечисляемый элемент коллекции. Но поскольку свойство Current доступно только для чтения, то перечислитель может служить только для извлечения, но не видоизменения объектов в коллекции.

В интерфейсе IEnumerator определяются два метода. Первым из них является метод MoveNext() –при каждом вызове метода MoveNext() текущее положение перечислителя смещается к следующему элементу коллекции. Этот метод возвращает логическое значение true, если следующий элемент коллекции доступен, и логическое значение false, если достигнут конец коллекции. Перед первым вызовом метода MoveNext() значение свойства Current оказывается неопределенным.

Для установки перечислителя в исходное положение, соответствующее началу коллекции, вызывается приведенный ниже метод Reset() – после вызова метода Reset() перечисление вновь начинается с самого начала коллекции. Поэтому, прежде чем получить первый элемент коллекции, следует вызвать метод MoveNext().

В интерфейсе IEnumerator<T> методы MoveNext() и Reset() действуют по тому же самому принципу. Необходимо также обратить внимание на два следующих момента. Во-первых, перечислитель нельзя использовать для изменения содержимого перечисляемой с его помощью коллекции. Следовательно, перечислители действуют по отношению к коллекции как к доступной только для чтения. И во-вторых, любое изменение в перечисляемой коллекции делает перечислитель недействительным.

***Итератор***

Реализовать интерфейсы IEnumerator и IEnumerable нетрудно. Но еще проще воспользоваться итератором, который представляет собой метод, оператор или аксессор, возвращающий по очереди члены совокупности объектов от ее начала и до конца. Так, если некоторый массив состоит из пяти элементов, то итератор данного массива возвратит все эти элементы по очереди. Реализовав итератор, можно обращаться к объектам определяемого пользователем класса в цикле foreach. Обозначение yield служит в языке C# в качестве контекстного ключевого слова. Это означает, что оно имеет специальное назначение только в блоке итератора. А вне этого блока оно может быть использовано аналогично любому другому идентификатору. Следует особо подчеркнуть, что итератор не обязательно должен опираться на массив или коллекцию другого типа. Он должен просто возвращать следующий элемент из совокупности элементов.

Для преждевременного прерывания итератора служит следующая форма оператора yield:

*yield break;*

Когда этот оператор выполняется, итератор уведомляет о том, что достигнут конец коллекции. А это, по существу, останавливает сам итератор. В итераторе допускается применение нескольких операторов yield. Но каждый такой оператор должен возвращать следующий элемент в коллекции.

***Именованный итератор*** представляет собой метод, общая форма которого приведена ниже:

*public IEnumerable имя\_итератора(список\_параметров) {*

*// ...*

*yield return obj;*

*}*

где имя\_итератора обозначает конкретное имя метода; список\_параметров — от нуля до нескольких параметров, передаваемых методу итератора; obj — следующий объект, возвращаемый итератором. Как только именованный итератор будет создан, его можно использовать везде, где он требуется, например для управления циклом foreach.

Именованные итераторы оказываются весьма полезными в некоторых ситуациях, поскольку они позволяют передавать аргументы итератору, управляющему процессом получения конкретных элементов из коллекции. Например, итератору можно передать начальный и конечный пределы совокупности элементов, возвращаемых из коллекции итератором. Эту форму итератора можно перегрузить, расширив ее функциональные возможности.

***ArrayList***

ArrayList - Определяет динамический массив, т.е. такой массив, который может при необходимости увеличивать свой размер. Массив типа ArrayList создается с первоначальным размером. Если этот размер превышается, то массив автоматически расширяется. А при удалении объектов из такого массива он автоматически сокращается.

*public ArrayList()*

*public ArrayList(ICollection с)*

*public ArrayList(int capacity)*

Первый конструктор создает пустую коллекцию класса ArrayList с нулевой первоначальной емкостью. Второй конструктор создает коллекцию типа ArrayList с количеством инициализируемых элементов, которое определяется параметром c и равно первоначальной емкости массива. Третий конструктор создает коллекцию, имеющую указанную первоначальную емкость, определяемую параметром capacity.

Некоторые из наиболее часто используемых методов класса ArrayList перечислены ниже:

AddRange() - Добавляет элементы из коллекции в конец вызывающей коллекции типа ArrayList

BinarySearch() - Выполняет поиск в вызывающей коллекции значения. Возвращает индекс найденного элемента. Если искомое значение не найдено, возвращает отрицательное значение. Вызывающий список должен быть отсортирован

СоруТо() - Копирует содержимое вызывающей коллекции в массив, который должен быть одномерным и совместимым по типу с элементами коллекции

FixedSize() - Заключает коллекцию в оболочку типа ArrayList с фиксированным размером и возвращает результат. Возвращает часть вызывающей коллекции типа ArrayList. Часть возвращаемой коллекции начинается с элемента, указываемого по индексу, и включает количество элементов, определяемое параметром count. Возвращаемый объект ссылается на те же элементы, что и вызывающий объект.

IndexOf() - Возвращает индекс первого вхождения объекта в вызывающей коллекции. Если искомый объект не обнаружен, возвращает значение -1

InsertRange() - Вставляет элементы коллекции в вызывающую коллекцию, начиная с элемента, указываемого по индексу

Readonly() - Заключает коллекцию в оболочку типа ArrayList, доступную только для чтения, и возвращает результат

RemoveRange() - Удаляет часть вызывающей коллекции, начиная с элемента, указываемого по индексу index, и включая количество элементов, определяемое параметром count

Sort() - Сортирует вызывающую коллекцию по нарастающей

***List<T>***

Создает динамический массив. Обеспечивает такие же функциональные возможности, как и необобщенный класс ArrayList

***HashTable***

Hashtable - предназначен для создания коллекции, в которой для хранения ее элементов служит хеш-таблица. Информация сохраняется в хеш-таблице с помощью механизма, называемого хешированием. При хешировании для определения уникального значения, называемого хеш-кодом, используется информационное содержимое специального ключа. Полученный в итоге хеш-код служит в качестве индекса, по которому в таблице хранятся искомые данные, соответствующие заданному ключу. Преобразование ключа в хеш-код выполняется автоматически, и поэтому сам хеш-код вообще недоступен пользователю. Преимущество хеширования заключается в том, что оно обеспечивает постоянство времени выполнения операций поиска, извлечения и установки значений независимо от величины массивов данных.

*public Hashtable()*

*public Hashtable(IDictionary d)*

*public Hashtable(int capacity)*

*public Hashtable(int capacity, float loadFactor)*

В первой форме создается создаваемый по умолчанию объект класса Hashtable. Во второй форме создаваемый объект типа Hashtable инициализируется элементами из коллекции d. В третьей форме создаваемый объект типа Hashtable инициализируется, учитывая емкость коллекции, задаваемую параметром capacity. И в четвертой форме создаваемый объект типа Hashtable инициализируется, учитывая заданную емкость capacity и коэффициент заполнения loadFactor. Коэффициент заполнения, иногда еще называемый коэффициентом загрузки, должен находиться в пределах от 0.1 до 1.0. Он определяет степень заполнения хеш-таблицы до увеличения ее размера. В частности, таблица расширяется, если количество элементов оказывается больше емкости таблицы, умноженной на коэффициент заполнения. В тех конструкторах, которые не принимают коэффициент заполнения в качестве параметра, этот коэффициент по умолчанию выбирается равным 1.0.

ContainsKey() - Возвращает логическое значение true, если в вызывающей коллекции типа Hashtable содержится ключ, а иначе — логическое значение false

ContainsValue() - Возвращает логическое значение true, если в вызывающей коллекции типа Hashtable содержится значение, а иначе — логическое значение false

GetEnumerator() - Возвращает для вызывающей коллекции типа Hashtable перечислитель типа IDictionaryEnumerator

Synchronized() - Возвращает синхронизированный вариант коллекции типа Hashtable, передаваемой в качестве параметра

В классе Hashtable доступны также открытые свойства, определенные в тех интерфейсах, которые в нем реализуются. Особая роль принадлежит двум свойствам, Keys и Values, поскольку с их помощью можно получить ключи или значения из коллекции типа Hashtable.

***Dictionary<Tkey, TValue>***

Dictionary<Tkey, TValue> - Сохраняет пары "ключ-значение". Обеспечивает такие же функциональные возможности, как и необобщенный класс Hashtable

Словарь (dictionary) представляет собой сложную структуру данных, позволяющую обеспечить доступ к элементам по ключу. Главное свойство словарей — быстрый поиск на основе ключей. Можно также свободно добавлять и удалять элементы, подобно тому, как это делается в List<T>, но без накладных расходов производительности, связанных с необходимостью смещения последующих элементов в памяти.

Тип, используемый в качестве ключа словаря, должен переопределять метод GetHashCode() класса Object. Всякий раз, когда класс словаря должен найти местоположение элемента, он вызывает метод GetHashCode().

В классе Dictionary<TKey, TValue> предоставляется немало конструкторов. Ниже перечислены наиболее часто используемые из них:

*public Dictionary()*

*public Dictionary(IDictionary<TKey, TValue> dictionary)*

*public Dictionary(int capacity)*

В первом конструкторе создается пустой словарь с выбираемой по умолчанию первоначальной емкостью. Во втором конструкторе создается словарь с указанным количеством элементов dictionary. А в третьем конструкторе с помощью параметра capacity указывается емкость коллекции, создаваемой в виде словаря. Если размер словаря заранее известен, то, указав емкость создаваемой коллекции, можно исключить изменение размера словаря во время выполнения, что, как правило, требует дополнительных затрат вычислительных ресурсов.

В классе Dictionary<TKey, TValue> определяется также ряд методов:

Add() - Добавляет в словарь пару "ключ-значение", определяемую параметрами key и value. Если ключ key уже находится в словаре, то его значение не изменяется, и генерируется исключение ArgumentException

ContainsKey() - Возвращает логическое значение true, если вызывающий словарь содержит объект key в качестве ключа; а иначе — логическое значение false

ContainsValue() - Возвращает логическое значение true, если вызывающий словарь содержит значение value; в противном случае — логическое значение false

Remove() - Удаляет ключ key из словаря. При удачном исходе операции возвращается логическое значение true, а если ключ key отсутствует в словаре — логическое значение false

Кроме того, в классе Dictionary<TKey, TValue> определяются собственные свойства, помимо тех, что уже объявлены в интерфейсах, которые в нем реализуются. Эти свойства приведены ниже:

Comparer - Получает метод сравнения для вызывающего словаря

Keys - Получает коллекцию ключей

Values - Получает коллекцию значений

***Queue, Queue<T>***

Очередь (queue) — это коллекция, в которой элементы обрабатываются по схеме "первый вошел, первый вышел" (first in, first out — FIFO). Элемент, вставленный в очередь первым, первым же и читается.

Очередь (queue)

В классе Queue определяются приведенные ниже конструкторы:

*public Queue()*

*public Queue (int capacity)*

*public Queue (int capacity, float growFactor)*

*public Queue (ICollection col)*

В первой форме конструктора создается пустая очередь с выбираемыми по умолчанию емкостью и коэффициентом роста 2.0. Во второй форме создается пустая очередь, первоначальный размер которой определяет емкость, задаваемая параметром capacity, а коэффициент роста по умолчанию выбирается для нее равным 2.0. В третьей форме допускается указывать не только емкость (в качестве параметра capacity), но и коэффициент роста создаваемой очереди (в качестве параметра growFactor в пределах от 1.0 до 10.0). И в четвертой форме создается очередь, состоящая из элементов указываемой коллекции col. Ее первоначальная емкость равна количеству указанных элементов, а коэффициент роста по умолчанию выбирается для нее равным 2.0.

Члены класса:

Count - Свойство Count возвращает количество элементов в очереди.

Enqueue() - Метод Enqueue() добавляет элемент в конец очереди.

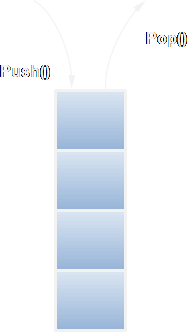
Dequeue() - Метод Dequeue() читает и удаляет элемент из головы очереди. Если на момент вызова метода Dequeue() элементов в очереди больше нет, генерируется исключение InvalidOperationException.

Peek() - Метод Peek() читает элемент из головы очереди, но не удаляет его.

TrimExcess() - Метод TrimExcess() изменяет емкость очереди. Метод Dequeue() удаляет элемент из очереди, но не изменяет ее емкости. TrimExcess() позволяет избавиться от пустых элементов в начале очереди.

***Stack, Stack<T>***

Stack - Стек (stack) — это контейнер, работающий по принципу "последний вошел, первый вышел" (last in, first out — LIFO). На рисунке показано представление стека, где метод Push() добавляет элемент, а метод Pop() — получает элемент, добавленный последним:



Этот класс создает динамическую коллекцию, которая расширяется по мере потребности хранить в ней вводимые элементы. Всякий раз, когда требуется расширить такую коллекцию, ее емкость увеличивается вдвое.

В классе Stack<T> непосредственно реализуются методы Clear(), Contains() и СоруТо(), определенные в интерфейсе ICollection<T>.

Count - Свойство Count возвращает количество элементов в стеке.

Метод Push() добавляет элемент в вершину стека.

Метод Pop() удаляет и возвращает элемент из вершины стека. Если стек пуст, генерируется исключение типа InvalidOperationException.

Метод Peek() возвращает элемент из вершины стека, не удаляя его при этом.

Метод Contains() проверяет наличие элемента в стеке и возвращает true в случае нахождения его там.

А если вызвать метод Pop() или Peek(), когда вызывающий стек пуст, то сгенерируется исключение InvalidOperationException.

***SortedList, SortedList<TKey, TValue>***

SortedList<TKey, TValue> - Создает отсортированный список из пар "ключ-значение". Обеспечивает такие же функциональные возможности, как и необобщенный класс SortedList

Если нужна коллекция, отсортированная по ключу, можно воспользоваться SortedList<TKey, TValue> Этот класс сортирует элементы на основе значения ключа. Можно использовать не только любой тип значения, но также и любой тип ключа.

*public SortedList()*

*public SortedList(IDictionary<TKey, TValue> dictionary)*

*public SortedList(int capacity)*

*public SortedList(IComparer<TK> comparer)*

В первой форме конструктора создается пустой список с выбираемой по умолчанию первоначальной емкостью. Во второй форме конструктора создается отсортированный список с указанным количеством элементов dictionary. В третьей форме конструктора с помощью параметра capacity задается емкость коллекции, создаваемой в виде отсортированного списка. Если размер списка заранее известен, то, указав емкость создаваемой коллекции, можно исключить изменение размера списка во время выполнения, что, как правило, требует дополнительных затрат вычислительных ресурсов. И в четвертой форме конструктора допускается указывать с помощью параметра comparer способ сравнения объектов, содержащихся в списке.

Некоторые из наиболее часто используемых методов этого класса перечислены ниже:

Add() - Добавляет в список пару "ключ-значение". Если ключ уже находится в списке, то его значение не изменяется, и генерируется исключение ArgumentException

ContainsKey() - Возвращает логическое значение true, если вызывающий список содержит объект key в качестве ключа; а иначе — логическое значение false

ContainsValue() - Возвращает логическое значение true, если вызывающий список содержит значение value; в противном случае — логическое значение false

GetEnumerator() - Возвращает перечислитель для вызывающего словаря

IndexOfKey(), IndexOfValue() - Возвращает индекс ключа или первого вхождения значения в вызывающем списке. Если искомый ключ или значение не обнаружены в списке, возвращается значение -1.

Remove() - Удаляет из списка пару "ключ-значение" по указанному ключу key. При удачном исходе операции возвращается логическое значение true, а если ключ key отсутствует в списке — логическое значение false.

TrimExcess() - Сокращает избыточную емкость вызывающей коллекции в виде отсортированного списка.

Кроме того, в классе SortedList<TK, TV> определяются собственные свойства, помимо тех, что уже объявлены в интерфейсах, которые в нем реализуются. Эти свойства приведены ниже:

Capacity - Получает или устанавливает емкость вызывающей коллекции в виде отсортированного списка

Comparer - Получает метод сравнения для вызывающего списка

Keys - Получает коллекцию ключей

Values - Получает коллекцию значений

И наконец, в классе SortedList<TKey, TValue> реализуется приведенный ниже индексатор, определенный в интерфейсе IDictionary<TKey, TValue>

*public TValue this[TKey key] { get; set; }*

Этот индексатор служит для получения и установки значения элемента коллекции, а также для добавления в коллекцию нового элемента. Но в данном случае в качестве индекса служит ключ элемента, а не сам индекс.

***SortedDictionary<TKey, TValue>***

SortedDictionary<TKey, TValue> - Создает отсортированный список из пар "ключ-значение" Класс SortedDictionary<TKey, Tvalue> представляет дерево бинарного поиска, в котором все элементы отсортированы на основе ключа. Тип ключа должен реализовать интерфейс IComparable<TKey>. Если тип ключа не сортируемый, компаратор можно также создать, реализовав IComparer<TKey> и указав его в качестве аргумента конструктора сортированного словаря.

Классы SortedDictionary<TKey, Tvalue> и SortedList<TKey, TValue> имеют схожую функциональность. Но поскольку SortedList<TKey, TValue> реализован в виде списка, основанного на массиве, a SortedDictionary<TKey, Tvalue> реализован как словарь, эти классы обладают разными характеристиками:

SortedList<TKey, TValue> использует меньше памяти, чем SortedDictionary<TKey, TValue>

SortedDictionary<TKey, TValue> быстрее вставляет и удаляет элементы.

При наполнении коллекции отсортированными данными SortedList<TKey,TValue> работает быстрее, если при этом не требуется изменение емкости.

В классе SortedDictionary<TKey, TValue> определен ряд методов. Некоторые наиболее часто используемые методы этого класса приведены ниже:

Add() - Добавляет в словарь пару "ключ-значение", определяемую параметрами key и value. Если ключ key уже находится в словаре, то его значение не изменяется, и генерируется исключение ArgumentException

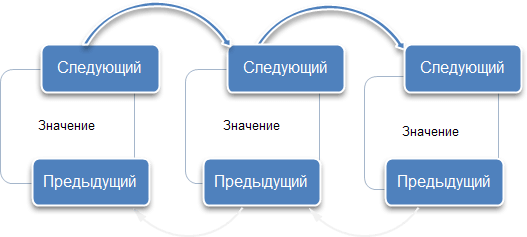
ContainsKey() - Возвращает логическое значение true, если вызывающий словарь содержит объект key в качестве ключа; в противном случае — логическое значение false

ContainsValue() - Возвращает логическое значение true, если вызывающий словарь содержит значение value, в противном случае — логическое значение false

Remove() - Удаляет ключ key из словаря. При удачном исходе операции возвращается логическое значение true, а если ключ key отсутствует в словаре — логическое значение false.

***LinkedList<T>***

LinkedList<T> представляет собой двухсвязный список, в котором каждый элемент ссылается на следующий и предыдущий, как показано на рисунке



Преимущество связного списка проявляется в том, что операция вставки элемента в середину выполняется очень быстро. При этом только ссылки Next (следующий) предыдущего элемента и Previous (предыдущий) следующего элемента должны быть изменены так, чтобы указывать на вставляемый элемент. В классе List<T> при вставке нового элемента все последующие должны быть сдвинуты.

Естественно, у связных списков есть и свои недостатки. Так, например, все элементы связных списков доступны лишь друг за другом. Поэтому для нахождения элемента, находящегося в середине или конце списка, требуется довольно много времени. Связный список не может просто хранить элементы внутри себя. Вместе с каждым из них ему необходимо иметь информацию о следующем и предыдущем элементах.

Наиболее часто используемые методы, определенные в классе LinkedList<T> представлены ниже:

AddAfter() - Добавляет в список узел со значением непосредственно после указанного узла. Указываемый узел не должен быть пустым (null). Метод возвращает ссылку на узел, содержащий значение.

AddBefore() - Добавляет в список узел со значением value непосредственно перед указанным узлом. Указываемый узел не должен быть пустым (null). Метод возвращает ссылку на узел, содержащий значение.

AddFirst(), AddLast() - Добавляют узел со значением в начало или в конец списка.

Find() - Возвращает ссылку на первый узел в списке, имеющий передаваемое значение. Если искомое значение отсутствует в списке, то возвращается пустое значение.

Remove() - Удаляет из списка первый узел, содержащий передаваемое значение. Возвращает логическое значение true, если узел удален, т.е. если узел со значением обнаружен в списке и удален; в противном случае возвращает логическое значение false.

***HashSet<T> и SortedSet<T>***

Коллекция, содержащая только отличающиеся элементы, называется множеством (set). В составе .NET 4 имеются два множества — HashSet<T> и SortedSet<T>. Оба они реализуют интерфейс ISet<T>. Класс HashSet<T> содержит неупорядоченный список различающихся элементов, а в SortedSet<T> элементы упорядочены.

|  |  |
| --- | --- |
| HashSet<T> - Сохраняет ряд уникальных значений, используя хештаблицу.  *public HashSet ()*  *public HashSet(IEnumerable<T> collection)*  *public HashSet(IEqualityCompare comparer)*  *public HashSet(IEnumerable<T> collection, IEqualityCompare comparer)*  В первой форме конструктора создается пустое множество, а во второй форме — множество, состоящее из элементов указываемой коллекции collection. В третьей форме конструктора допускается указывать способ сравнения с помощью параметра comparer. А в четвертой форме создается множество, состоящее из элементов указываемой коллекции collection, и используется заданный способ сравнения comparer. Имеется также пятая форма конструктора данного класса, в которой допускается инициализировать множество последовательно упорядоченными данными.  В этом классе предоставляется также метод RemoveWhere(), удаляющий из множества элементы, удовлетворяющие заданному условию, или предикату. Помимо свойств, определенных в интерфейсах, которые реализуются в классе HashSet<T>, в него введено дополнительное свойство Comparer, приведенное ниже:  public IEqualityComparer<T> Comparer { get; }  Оно позволяет получать метод сравнения для вызывающего хеш-множества. | SortedSet<T> - Создает отсортированное множество  *public SortedSet()*  *public SortedSet(IEnumerable<T> collection)*  *public SortedSet(IComparer comparer)*  *public SortedSet(IEnumerable<T> collection, IComparer comparer)*  В первой форме конструктора создается пустое множество, а во второй форме — множество, состоящее из элементов указываемой коллекции collection. В третьей форме конструктора допускается указывать способ сравнения с помощью параметра comparer. А в четвертой форме создается множество, состоящее из элементов указываемой коллекции collection, и используется заданный способ сравнения comparer. Имеется также пятая форма конструктора данного класса, в которой допускается инициализировать множество последовательно упорядоченными данными.  В этом классе предоставляется также метод GetViewBetween(), возвращающий часть множества в форме объекта типа SortedSet<T>, метод RemoveWhere(), удаляющий из множества элементы, не удовлетворяющие заданному условию, или предикату, а также метод Reverse(), возвращающий объект типа IEnumerable<T>, который циклически проходит множество в обратном порядке. |

В среде .NET Framework предусмотрен ряд специальных коллекций, оптимизированных для работы с данными конкретного типа или для их обработки особым образом. Классы этих необобщенных коллекций определены в пространстве имен System.Collections.Specialized и перечислены ниже:

|  |  |
| --- | --- |
| Класс специальной коллекции | Описание |
| CollectionsUtil | Содержит фабричные методы для создания коллекций |
| HybridDictionary | Предназначен для коллекций, в которых для хранения небольшого количества пар "ключ-значение" используется класс ListDictionary. При превышении коллекцией определенного размера автоматически используется класс Hashtable для хранения ее элементов |
| ListDictionary | Предназначен для коллекций, в которых для хранения пар "ключ-значение" используется связный список. Такие коллекции рекомендуются только для хранения небольшого количества элементов |
| NameValueCollection | Предназначен для отсортированных коллекций, в которых хранятся пары "ключ-значение", причем и ключ, и значение относятся к типу string |
| OrderedDictionary | Предназначен для коллекций, в которых хранятся индексируемые пары "ключ-значение" |
| StringCollection | Предназначен для коллекций, оптимизированных для хранения символьных строк |
| StringDictionary | Предназначен для хеш-таблиц, в которых хранятся пары "ключ-значение", причем и ключ, и значение относятся к типу string |

***CollectionBase***

Предоставляет абстрактный базовый класс для строго типизированной коллекции. этот базовый класс предоставляется, чтобы облегчить разработчикам создание строго типизированной настраиваемой коллекции. Разработчикам следует расширить этот базовый класс, а не создавать свой собственный.

Конструкторы

КОНСТРУКТОРЫ

CollectionBase() - Инициализирует новый экземпляр класса CollectionBase с начальной емкостью по умолчанию.

CollectionBase(Int32) - Инициализирует новый экземпляр класса CollectionBase с указанной емкостью.

*СВОЙСТВА*

Capacity - Возвращает или задает число элементов, которое может содержать список CollectionBase.

Count - Возвращает количество элементов, содержащихся в экземпляре CollectionBase. Это свойство нельзя переопределить.

InnerList - Возвращает объект ArrayList, в котором хранится список элементов экземпляра класса CollectionBase.

List - Возвращает объект IList, в котором хранится список элементов экземпляра класса CollectionBase.

*МЕТОДЫ*

Clear() - Удаляет все объекты из экземпляра класса CollectionBase. Этот метод не может быть переопределен.

Equals(Object) - Определяет, равен ли указанный объект текущему объекту.(Унаследовано от Object)

GetEnumerator() - Возвращает перечислитель, перебирающий элементы экземпляра класса CollectionBase.

GetHashCode() - Служит хэш-функцией по умолчанию.(Унаследовано от Object)

GetType() - Возвращает объект Type для текущего экземпляра.(Унаследовано от Object)

MemberwiseClone() - Создает неполную копию текущего объекта Object.(Унаследовано от Object)

OnClear() - Выполняет дополнительные пользовательские действия при очистке содержимого экземпляра CollectionBase.

OnClearComplete() - Осуществляет дополнительные пользовательские действия после удаления содержимого экземпляра класса CollectionBase.

OnInsert(Int32, Object) - Выполняет дополнительные пользовательские действия перед вставкой нового элемента в экземпляр класса CollectionBase.

OnInsertComplete(Int32, Object) - Выполняет дополнительные пользовательские действия после вставки нового элемента в экземпляр класса CollectionBase.

OnRemove(Int32, Object) - Осуществляет дополнительные пользовательские действия при удалении элемента из экземпляра класса CollectionBase.

OnRemoveComplete(Int32, Object) - Осуществляет дополнительные пользовательские действия после удаления элемента из экземпляра класса CollectionBase.

OnSet(Int32, Object, Object) - Выполняет дополнительные пользовательские действия перед заданием значения в экземпляре класса CollectionBase.

OnSetComplete(Int32, Object, Object) - Выполняет дополнительные пользовательские действия после задания значения в экземпляре класса CollectionBase.

OnValidate(Object) - Выполняет дополнительные пользовательские операции при проверке значения.

RemoveAt(Int32) - Удаляет элемент по указанному индексу в экземпляре класса CollectionBase. Этот метод нельзя переопределить.

ToString() - Возвращает строку, представляющую текущий объект.(Унаследовано от Object)