## Условные конструкции

**Оператор ветвления (условный оператор)** – конструкция языка программирования, обеспечивающая выполнение определенной команды или набора команд только при условии истинности некоторого логистического выражения.

ЕСЛИ: (Условие удовлетворяет истинности)

ТО: Выполнить эту серию инструкций

ИНАЧЕ: Выполнить эту серию инструкций

В C# есть 3 разновидности условных конструкций:

* Условный оператор: **if … else**
* Тернарный оператор: **…? … : …**
* Оператор многозначного набора: **switch – case**

namespace Условная\_конструкция\_if\_else\_с\_двумя\_ветвями

{

class Program

{

static void Main(string[] args)

{

int a = 3, b = 1;

if(a < b)

{

Console.WriteLine("a меньше b");

}

else

{

Console.WriteLine("a не меньше b");

}

Console.ReadKey();

}

}

}

Результат выполнения этой программы:

**a не меньше b**

Рассмотрим пример каскада оператора **if**.

class Program

{

static void Main(string[] args)

{

int a = 2, b = 2;

if (a < b)

{

Console.WriteLine("a < b");

}

else if (a > b)

{

Console.WriteLine("a > b");

}

else

{

Console.WriteLine("a == b");

}

Console.ReadKey();

}

}

}

Результат выполнения этой программы:

**a == b**

Рассмотрим пример использования оператора **if** для аутентификации.

class Program

{

static void Main(string[] args)

{

string login = "Admin";

string password = "Password";

Console.Write("Введите логин: ");

// Создаем строковую локальную переменную с именем

// и принимаем в нее ввод от пользователя

string usersLogin = Console.ReadLine();

if (login == usersLogin)

{

Console.Write("Введите пароль: ");

string usersPassword = Console.ReadLine();

if (password == usersPassword)

{

Console.WriteLine("Здравствуйте {0}, Вы вошли в систему", usersLogin);

}

else

{

Console.WriteLine("Вы ввели неверный пароль.");

}

}

else

{

Console.WriteLine("Нет пользователя с таким именем.");

}

Console.ReadKey();

}

}

}

Создаем строковые переменные login и password с значениями Admin и Password. Далее, предлагаем пользователю ввести логин, после чего строковой переменной usersLogin присваиваем значение, введенное пользователем. Здесь мы вызываем метод **ReadLine()**. С помощью оператора **if** проверяем, совпадает ли значение переменной login с значением, введенным пользователем. В случае совпадения мы оказываемся в теле условной конструкции и предлагаем пользователю ввести пароль. Далее, переменной password присваиваем значение, введенное пользователем. Если значение переменной password совпадает со значением переменной usersPassword (значение, введенное пользователем), тогда система приветствует нас, так как мы ввели верные логин и пароль. Если значения паролей не совпадают, выводится сообщение «Вы ввели неверный пароль». Если же мы ввели неверный логин, то система выдаст нам сообщение «Нет пользователя с таким именем».

Следующей условной конструкцией является **тернарный (третичный)** оператор. Он обозначается как **? :** . Если условие удовлетворяет истинности, мы выполняем одну серию команд. Если нет – другую. Рассмотрим следующий пример.

class Program

{

static void Main(string[] args)

{

int a = 1, b = 2, c = 0;

c = (a > b) ? a : b; // ЕСЛИ: а больше b ТО: вернуть а ИНАЧЕ: вернуть b;

Console.WriteLine(c);

Console.ReadKey();

}

}

}

В данном примере: если значение а больше значения b, то с равно а. Если же значение а меньше значения b, то c равно b. Знак вопроса заменяет слово ТО, а двоеточие заменяет слово ИНАЧЕ. Нужно обратить внимание, что тернарный оператор должен возвращать значение.

Рассмотрим еще один пример.

class Program

{

static void Main(string[] args)

{

int a = 1, b = 2, c = -5, max = 0;

max = (a > b) ? c = a : c = b;

Console.WriteLine(max);

Console.WriteLine(c);

Console.ReadKey();

}

}

}

В данном примере: если значение а больше значения b, то с присваивается значение а. После чего переменной max также присваивается значение а. Если же а меньше b, то с присваивается значение переменной b и переменной max также присваивается значение переменной b.

Рассмотрим еще один пример.

static void Main(string[] args)

{

string @string = "Hello ";

Console.Write("Введите свой логин: ");

string login = Console.ReadLine();

@string += login == "Admin" ? "Administrator" : "User";

Console.WriteLine(@string);

Console.ReadKey();

}

}

}

В данном примере:

**+=** - это конкатенация, что означает сцепление строк.

Создаем строковую переменную @string, которой присваивается значение «Hello ». Далее, предлагаем ввести логин. Используя конкатенацию, соединяем две переменные, причем если ввести логин “Admin”, то появится надпись «Hello Administrator». Если же ввести какой-то другой логин, то высветится надпись «Hello User».

Рассмотрим еще один пример.

class Program

{

static void Main(string[] args)

{

int a = 1;

double b = 0.0;

int max;

// Выражения [?] и [:] должны быть одного типа

max = (int)((a > b) ? a : b);

// ... или так

max = (a > b) ? a : (int)b;

Console.WriteLine(max);

Console.ReadKey();

}

}

}

В данном примере: если условие удовлетворяет истинности, мы переменной max присваиваем значение переменной а. Если же не удовлетворяет, то мы переменной max присваиваем значение переменной b, предварительно сконвертировав его в тип **int**. При этом дробная часть значения переменной b отсекается.

Рассмотрим еще один пример.

static void Main(string[] args)

{

// Определение квадранта в котором лежит заданная точка. Для упрощения примера условимся, что точка не будет лежать

// в центре координат и на координатных осях. Это значит, что x и y - не могут быть равны 0.

int x = 5, y = -5;

// Не рекомендуется вкладывать тернарные операторы, так как это ухудшает чтение кода.

string quadrant = (x > 0) ? ((y > 0) ? "I квадрант" : "IV квадрант") : ((y > 0) ? "II квадрант" : "III квадрант");

// ... или так

quadrant = x > 0 ? (y > 0 ? "I квадрант" : "IV квадрант") : (y > 0 ? "II квадрант" : "III квадрант");

// ... или вообще так

quadrant = x > 0 ? y > 0 ? "I квадрант" : "IV квадрант" : y > 0 ? "II квадрант" : "III квадрант";

Console.WriteLine(quadrant);

Console.ReadKey();

}

}

}

В данном примере:

Если значение x > 0 и значение y > 0, тогда выводим на экран «I квадрант». Если значение x > 0, а значение y < 0, тогда выводим на экран «IV квадрант». Если значение x < 0, а значение y > 0, тогда выводим на экран «II квадрант». Если же значение x < 0 и значение y < 0, тогда выводим на экран «III квадрант».

Рассмотрим еще один пример.

class Program

{

static void Main(string[] args)

{

double quantity = 10; // Количество единиц товара.

double price = 100; // Цена за единицу товара.

double discount = 0.75; // Размер скидки - 25%.

double cost; // общая стоимость.

// ЕСЛИ: купили 10 единиц товара и больше, ТО: предоставить скидку 25%.

// ИНАЧЕ: скидку не предоставлять.

cost = quantity >= 10 ? quantity \* price \* discount : quantity \* price;

Console.WriteLine("Общая стоимость товара составляет {0} USD", cost);

Console.ReadKey();

}

}

}

В данном примере:

Если количество товара превышает 10 единиц, то цена за общее количество купленного товара составит кол-во товара \* цену \* коэффициент скидки. Иначе цена за общее количество купленного товара составит кол-во товара \* цену.

Последняя условная конструкция, которая имеется в C# – это конструкция **switch-case**, которая также называется **оператором многозначного выбора (переключателем)**.

class Program

{

static void Main(string[] args)

{

string number = "1";

switch (number) // (number) - выражение-селектор.

{

case "1": // "1" - постоянное выражение.

{

Console.WriteLine("Один");

break;

}

case "2": // "2" - постоянное выражение.

{

Console.WriteLine("Два");

break;

}

}

Console.ReadKey();

}

}

}

Создаем строковую переменную number и присваиваем ей значение «1». Создаем конструкцию **switch** и в качестве селектора передаем значение переменной number. Select – это выбор. Нужно обратить внимание, что конструкция **switch** состоит из вложенных операторов **case** – случай. Значения возле **case** принято называть постоянными выражениями, так как они не меняются. Важно понимать, что если значение выражения селектора совпадает с одним из постоянных выражений операторов **case**, то выполняется тело того оператора **case**, с каким постоянным выражением совпадает значение выражения селектора.

Рассмотрим пример.

class Program

{

static void Main(string[] args)

{

// Предлагаем пользователю ввести число.

Console.WriteLine("Введите число 1 или 2: ");

// Принимаем ввод от пользователя.

string number = Console.ReadLine();

switch (number) // (number) - выражение-селектор.

{

case "1": // "1" - постоянное выражение.

{

Console.WriteLine("Один");

break; // Выход из переключателя.

}

case "2":

{

Console.WriteLine("Два");

break;

}

default:

{

Console.WriteLine("Вы ввели значение, которое отличается от 1 и 2.");

break;

}

}

Console.ReadKey();

}

}

}

В данном примере:

Предлагаем пользователю ввести значение «1» или «2». Далее, создаем переменную строчную переменную типа number и принимаем ввод от пользователя. После этого создаем конструкцию **switch**, которой в качестве выражения селектора передаем значение number. В теле конструкции **switch** у нас имеется несколько операторов **case**. Если введенное значение селектора совпадет с одним из постоянных выражений операторов **case**, которые имеются в **switch**, то сработает соответствующий кодовый блок. Если же не совпадет, тогда выполняется кодовый блок оператора **default**, который является необязательным. Однако он удобен тем, что пользователь будет уведомлен относительно ввода отличного от предлагаемых значений. Оператор **break** означает выход из переключателя.

Рассмотрим еще один пример.

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Введите число, соответствующее дню недели (от 1 до 7): ");

string myDay = Console.ReadLine();

int day = Convert.ToInt32(myDay);

switch(day)

{

case 1: Console.WriteLine("Понедельник");

break;

case 2: Console.WriteLine("Вторник");

break;

case 3:

Console.WriteLine("Среда");

break;

case 4:

Console.WriteLine("Четверг");

break;

case 5:

Console.WriteLine("Пятница");

break;

case 6:

Console.WriteLine("Суббота");

break;

case 7:

Console.WriteLine("Воскресенье");

break;

default:

Console.WriteLine("Вы ввели число, которое не соответствует ни одному из дней недели");

break;

}

Console.ReadKey();

}

}

}

В данном примере:

Задаем переменную строчную переменную myDay, которая будет принимать значение ввода от пользователя. Далее конвертируем ее в числовой тип **int** с помощью метода Covert.ToInt32 и присваиваем ее значение переменной day. Создаем конструкцию **switch**, которой в качестве выражения селектора передаем значение переменной day.

Рассмотрим еще один пример.

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Введите номер дня недели: 1-2-3-4-5-6-7: ");

string day = Console.ReadLine();

switch (day) {

case "1":

case "2":

case "3":

case "4":

case "5":

Console.WriteLine("Этот день недели - Рабочий.");

break;

case "6":

case "7":

Console.WriteLine("Этот день недели - Выходной.");

break;

default:

Console.WriteLine("Вы ввели несуществующий день недели");

break;

}

Console.ReadKey();

}

}

}

В данном примере:

Здесь мы демонстрируем работу оператора **switch** с возможностью проваливания. Предлагаем пользователю ввести день недели от 1 до 7. Создаем конструкцию-переключатель, которой присваиваем значение, введенное пользователем. Если мы введем значение от 1 до 5, то система выведет информацию о том, что этот день недели является рабочим. Если же мы введем значение 6 или 7, то увидим на экране информацию о том, что этот день недели является выходным. И, наконец, если мы введем какое-либо другое значение, то увидим на экране информацию, что ввели несуществующий день недели.

Рассмотрим еще один пример.

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Размеры стаканчиков кофе: 1 = Маленький, 2 = Средний, 3 = Большой");

Console.Write("Пожалуйста, сделайте свой выбор: ");

string Coffeesize = Console.ReadLine();

int cost = 0;

// Если оператор case имеет в своем теле код, то

// "проваливание" не поддерживается.

// Для организации "проваливания" можно использовать

// оператор безусловного перехода - goto

switch (Coffeesize)

{

case "1":

cost += 25;

break;

case "2":

cost += 25;

goto case "1";

case "3":

cost += 50;

goto case "1";

default:

Console.WriteLine("Неверный выбор. Пожалуйста, выберите 1, 2 или 3.");

break;

}

if (cost !=0)

{

Console.WriteLine("Внесите {0} копеек.", cost);

Console.WriteLine("Спасибо!");

}

else

{

Console.WriteLine("Пожалуйста, повторите выбор.");

}

Console.ReadKey();

}

}

}

Пользователю предлагают ввести число, соответствующее размеру стакана. Далее мы присваиваем введенное значение переменной Coffeesize. Создает переменную cost, которая будет обозначать стоимость напитка. Далее, создаем переключатель **switch**, которому в качестве выражения селектора мы передаем выбор пользователя. Например, введем 2. Cost += 25 – это присвоение со сложением. Переменной cost присваивается значение 25 и благодаря оператору **goto** мы отправляется в case “1” и добавляем к 25 еще 25. В итоге у нас получается стоимость 50.

## Логические конструкции

**Конъюнкция** (логическое умножение) – логическая операция, по своему применению максимально приближенная к союзу «и».

операнд1 && операнд2 = результат

Таблица истинности для операции конъюнкции двух логических выражений:

**true && true = true true && false = false**

**false && false = false false && true = false**

Побитовое И – это бинарная операция, действие которой эквивалентно применению логического И к каждой паре битов, которые стоят на одинаковых позициях в двоичных представлениях операндов.

Таблица истинности для операции побитового «И» (конъюнкции) значений

**1 & 1 = 1 1 & 0 = 0**

**0 & 0 = 0 0 & 1 = 0**

result = 255 & 1 = 1

1111 1111 255

& &

0000 0001 1

---------------- -----

0000 0001 1

**Дизъюнкция** (логическое сложение) – логическая операция, по своему применению максимально приближенная к союзу «или» в смысле «или то, или это, или оба сразу».

операнд1 || операнд2 = результат

Таблица истинности для операции дизъюнкции двух логических выражений:

**true || true = true true || false = true**

**false || false = false false || true = true**

Побитовое ИЛИ – это бинарная операция, действие которой эквивалентно применению логического ИЛИ к каждой паре битов, которые стоят на одинаковых позициях в двоичных представлениях операндов.

Таблица истинности для операции побитового «ИЛИ» (дизъюнкции) значений

**1 | 1 = 1 1 | 0 = 1**

**0 | 0 = 0 0 | 1 = 1**

result = 2 | 1 = 3

0000 0010 2

| |

0000 0001 1

---------------- -----

0000 0011 3

**Исключающее ИЛИ** (логическое сложение, строгая дизъюнкция) – булева функция и логическая операция. Результат выполнения операции является истинным только при условии, если является истинным в точности один из аргументов.

операнд1 ^ операнд 2 = результат

Таблица истинности для операции дизъюнкции двух логических выражений:

**true ^ true = false true ^ false = true**

**false ^ false = false false ^ true = true**

Побитовое исключающее ИЛИ – это бинарная операция, действие которой эквивалентно применению логического исключающего ИЛИ к каждой паре битов, которые стоят на одинаковых позициях в двоичных представлениях операндов.

Таблица истинности для операции побитового «Исключающего ИЛИ» значений

**1 ^ 1 = 0 1 ^ 0 = 1**

**0 ^ 0 = 0 0 ^ 1 = 1**

result = 3 | 1 = 2

0000 0011 3

^ ^

0000 0001 1

---------------- -----

0000 0010 2

**Отрицание в логике** – унарная операция над суждениями, результатом которой является суждение (в известном смысле) «противоположное исходному».

! операнд = результат

Таблица истинности для операции отрицания логического выражения:

**!false = true !true = false**

Отрицанием истины является ложь, а отрицанием лжи – истина.

Побитовое отрицание (НЕ) = это унарная операция, действие которой эквивалентно применению логического отрицания к каждому биту, в двоичном представлении операнда.

Таблица истинности для операции побитового «Отрицания НЕ» значений

**~ 0 = 1**

**~ 1 = 0**

~1 = -2

0000 0001 1

~ ---------------- ~ -----

1111 1110 -2

**Изменение знака.** Чтобы изменить знак числа, необходимо выполнить его отрицание, и полученный результат увеличить на 1.

Формула изменения знака числа

**~ ( + N ) + 1 = -N**

**~ ( - N ) + 1 = +N**

0000 0001 1

~ ~

1111 1110 -2

+ +

0000 0001 1

---------------- -----

1111 1111 -1

Рассмотрим пример.

class Program

{

static void Main(string[] args)

{

byte operand1 = 0, operand2 = 0;

int result;

/\* Конъюкнкция \*/

// Таблица истинности для операций Конъюнкции (И) - & - [AND]

// Если хоть один из операндов имеет значение 0 - вся конструкция

// имеет значение 0. Иначе - 1.

// 0 & 0 = 0 1 & 0 = 0

// 1 & 1 = 1 0 & 1 = 0

operand1 = 0xFF; // [1111 1111 Bin] = [FF Hex] = [255 Dec]

operand2 = 0x01; // [0000 0001 Bin] = [01 Hex] = [01 Dec]

result = operand1 & operand2; // [0000 0001 Bin] = [01 Hex] =

// = [01 Dec]

Console.WriteLine("{0} AND {1} = {2}", operand1, operand2, result);

/\* Дизъюнкция \*/

// Таблица истинности для Дизъюнкции (ИЛИ) - | - [OR]

// Если хоть один из операндов имеет значение 1 - вся конструкция

// имеет значение 1. Иначе - 0.

// 0 | 0 = 0 1 | 0 = 1

// 1 | 1 = 1 0 | 1 = 1

operand1 = 0x02; // [0000 0010 Bin] = [02 Hex] = [02 Dec]

operand2 = 0x01; // [0000 0001 Bin] = [01 Hex] = [01 Dec]

result = operand1 | operand2; // [0000 0011 Bin] = [03 Hex] =

// = [03 Dec]

Console.WriteLine("{0} OR {1} = {2}", operand1, operand2, result);

/\* Исключающее ИЛИ \*/

// Таблица истинности для исключающего (ИЛИ) - ^ - [XOR]

// Если операнды имеют одинаковое значение - результат

// операции 0. Иначе - 1.

// 0 ^ 0 = 0 1 ^ 0 = 1

// 1 ^ 1 = 0 0 ^ 1 = 1

operand1 = 0x03; // [0000 0011 Bin] = [03 Hex] = [03 Dec]

operand2 = 0x01; // [0000 0001 Bin] = [01 Hex] = [01 Dec]

result = operand1 ^ operand2; // [0000 0010 Bin] = [02 Hex] =

// = [02 Dec]

Console.WriteLine("{0} ^ {1} = {2}", operand1, operand2, result);

/\* Отрицание \*/

// Таблица истинности для отрицания (НЕТ) - ~ - [NOT]

// ~ 0 = 1 // [0000 0001 Bin] = [01 Dec]

// ~ 1 = 0 // [1111 1110 Bin] = [-2 Dec]

operand1 = 0x01;

result = ~operand1;

Console.WriteLine("NOT {0} = {1}", operand1, result);

/\* Изменение знака числа \*/

// Формула изменения знака числа с (+N) на (-N) или наоборот.

// Для того чтобы изменить знак числа, необходимо выполнить

// его отрицание, а результат увеличить на 1.

// ~ +N + 1 = -N

// ~ -N + 1 = +N

operand1 = 0x01; // [0000 0001 Bin]

result = ~operand1; // [1111 1110 Bin]

result++; // [1111 1111]

Console.WriteLine("~ {0} + 1 = {1}", operand1, result);

Console.ReadKey();

}

}

}

В данном примере:

Создали две переменные типа **byte** с именами operand1 и operand2 и значениями 0 и 0. Также создали целочисленную переменную result. Здесь у нас имеются блоки: конъюнкция, дизъюнкция, исключающее ИЛИ, отрицание и изменение знака числа. Далее, переменной operand1 присваиваем значение 0xFF. Переменной operand2 присваиваем значение 0x01 (соответствует десятичной 1). Переменной result присваиваем значение возвращаемое значение операции побитовой конъюнкции. После этого переменной operand1 присваиваем значение 0x02, переменной operand2 присваиваем 0x01, а переменной result присваиваем значение операции побитовой дизъюнкции, которая работает с двумя операндами. Далее, переменной operand1 присваиваем значение 0x03, переменной operand2 присваиваем 0x01, а переменной result присваиваем значение операции исключающее ИЛИ. На следующем этапе работаем с отрицанием: переменной operand1 присваиваем значение 1, после чего инвертируем это значение, заменив 1 на 0. В заключении мы записали формулу по изменению знака числа (образно говоря, она звучит «инверт плюс 1»). Переменной operand1 присваиваем значение 1 с дальнейшим его инвертированием и добавлением 1. В нашем случае 0000 0001 (соответствует единице) инвертировалось в 1111 1110 (соответствует -2). Добавляем к полученному значению 0000 0001 (соответствует единице) и получаем 1111 1111 (соответствует -1).

Рассмотрим следующий пример.

class Program

{

static void Main(string[] args)

{

// Например, мы имеем порт ввода/вывода или регистр с

// определенным значением в нем. Нам необходимо включить

// устройство управляемое первым битом, установив первый

// бит в 1.

// После нам потребуется выключить устройство, сбросив

// первый бит в 0.

// 1111 0000 - начальное значение порта ввода/вывода.

// 7654 3210 - нумерация битов управления устройствами.

byte port = 0xF0; // [1111 0000 Bin] [240 Dec]

// начальное значение порта ввода/вывода.

byte mask = 0x02; // [0000 0010 Bin] [02 Dec]

// маска включения устройства, управляемого первым битом.

Console.WriteLine("port = {0:X}", port); // ситуация с

// лампочкой в комнате.

port = (byte)(port | mask); // Включить устройство,

// управляемое первым битом.

Console.WriteLine("port = {0:X}", port);

mask = 0xFD; // [1111 1101 Bin] [253 Dec] - создаем

// маску выключения устройства, управляемого первым битом.

port = (byte)(port & mask); // выключить устройство,

// управляемое первым битом.

Console.WriteLine("port = {0:X}", port);

Console.ReadKey();

}

}

}

В данном примере:

Есть некий порт со значением 0xF0 (1111 0000). Создаем некую маску со значением 0x02 (0000 0010). Здесь в первом бите идет 1, а все остальные нули. Выводим на экран состояние порта. Далее, включаем этот порт через операцию дизъюнкции. В итоге значение 0xF0 (1111 0000) поменялось на 0xF2 (1111 0010). Теперь, чтобы выключить устройство, нужно поменять единицу обратно на нуль. Для этого необходима операция конъюнкции. Однако перед этим нам нужно задать маску, чтобы в результате операции у нас единица изменилась на нуль. Мы задали новую маску со значением 0xFD (1111 1101). В результате 0xF2 (1111 0010) изменилось на 0xF0 (1111 0000).

Рассмотрим еще один пример.

class Program

{

static void Main(string[] args)

{

// Используя операцию XOR, мы можем зашифровать сообщение.

// В таком виде шифрования используется один ключ, как для

// шифрования, так и для расшифровки.

// Криптостойкость такого ключа можно увеличить, если

// увеличить его длину.

ushort secretKey = 0x0088; // Секретный ключ (длина 16 bit).

char character = 'A'; // Исходный символ для шифрования.

Console.WriteLine("Исходный символ: {0}, его код в кодовой таблице: {1:X}", character, (byte)character);

Console.WriteLine("Размер символа: {0} = {1} бит", character, sizeof(char) \* 8);

// Зашифровываем символ.

character = (char)(character ^ secretKey);

Console.WriteLine("Зашифрованный символ: {0}, его код в кодовой таблице: {1:X}", character, (byte) character);

// Расшифровываем символ.

character = (char)(character ^ secretKey);

Console.WriteLine("Расшифрованный символ: {0}, его код в кодовой таблице: {1:X}", character, (byte) character);

Console.ReadKey();

}

}

}

В данном примере:

Код большого символа А равен 0x0041. Для его перекодировки нужно сформировать секретный ключ secretKey, имеющий значение 0x0088 или 1000 1000 в двоичном виде. При этом двоичный код символа А равен 0100 0001. Изначально мы выводим исходный символ и его код в кодовой таблице. Для просмотра его размера используем оператор **sizeof**. Передаем сюда тип **char**, который равен 2, так как это формат Юникод. 2 \* 8 = 16 бит (размер символа). Зашифровываем наше сообщение с помощью операции исключающее ИЛИ. 0100 0001 ^ 1000 1000 = 1100 1001 (С9). Далее, этой же операцией расшифровываем наше зашифрованное сообщение. Конъюнкция, дизъюнкция используются для работы с портами, для включения/выключения устройства или для установки/сброса флажка. Исключающее ИЛИ используется, например, для шифрования и расшифровки информации.

Рассмотрим следующий пример.

class Program

{

static void Main(string[] args)

{

bool operand1 = false, operand2 = false, result = false;

// Конъюнкция

operand1 = true;

operand2 = false;

result = operand1 & operand2;

Console.WriteLine("{0} AND {1} = {2}", operand1, operand2, result);

// Дизъюнкция

operand1 = true;

operand2 = false;

result = operand1 | operand2;

Console.WriteLine("{0} OR {1} = {2}", operand1, operand2, result);

// Исключающее ИЛИ

operand1 = true;

operand2 = false;

result = operand1 ^ operand2;

Console.WriteLine("{0} XOR {1} = {2}", operand1, operand2, result);

// Отрицание

operand1 = true;

result = !operand1;

Console.WriteLine("NOT {0} = {1}", operand1, result);

Console.ReadKey();

}

}

}

В данном примере:

Создаем булевы переменные operand1, operand2 и result. Помним, что логика условно разделяется на две части: работа с чистыми булевыми значениями и выполнение побитовых логических операций. Сейчас мы рассматриваем чистые логические булевы операции. Задаем переменным необходимые значения и проверяет результат конъюнкции, дизъюнкции, исключающего ИЛИ и отрицания. При отрицании задаем переменной operand1 значение true, а переменной result значение !operand1.

Рассмотрим следующий пример.

class Program

{

static void Main(string[] args)

{

// Чтобы проверить условие A < x < B, нельзя записать

// его в условном операторе непосредственно. Правильная

// форма записи: if (A < x && x < B)

int A = 0, B = 5, x = 3;

if (A < x && x < B)

{

Console.WriteLine("Число {0} находится в диапазоне чисел от {1} до {2}", x, A, B);

}

else

{

Console.WriteLine("Число {0} выходит за пределы диапазона чисел от {1} до {2}", x, A, B);

}

Console.ReadKey();

}

}

}

В данном примере:

В программировании математическую запись A < x < B нельзя записать в том виде, в котором она пишется на бумаге. Для этого нам необходимо использовать булеву операцию дизъюнкции с применением оператора **if**. Так, **if** (A < x && x < B).

При **логическом сдвиге** значение последнего бита по направлению сдвига теряется (копируясь в бит переноса), а первый бит приобретает нулевое значение.

253 << 2 = 244

1111 1101 253

<< << 2

xx1111 0100 244

xx – это биты, выходящие за пределы диапазона. Они теряются. При этом справа всегда заходят нули!

-127 >> 2 = -32

1000 0001 -127

>> >> 2

1110 0000xx -32

Слева – если число положительное, заходят нули, а если отрицательное – единицы.

class Program

{

static void Main(string[] args)

{

byte operand = 0x81; // [1000 0001]

Console.WriteLine("Число до сдвига: {0:X}", operand);

// Логический сдвиг числа влево

operand = (byte)(operand << 2); // 0000 0100

Console.WriteLine("Число после сдвига влево: {0:X}", operand);

// Логический сдвиг вправо

operand = (byte)(operand >> 1); // 0000 0010

Console.WriteLine("Число после сдвига вправо: {0:X}", operand);

Console.ReadKey();

}

}

}

Создаем переменную operand типа **byte** с значением 0x81 (1000 0001). Вывели ее значение на экран. Далее, пробуем сдвиг влево на 2. После этого у нас получилось 0000 0100 (4 Dec). После этого мы решили сдвинуть нашу четверку вправо на 1 бит. Сдвиг на 1 вправо равносилен делению числа на 2 (с условием, что не будет происходить потери части числа). В итоге у нас получилось 0000 0010.

Рассмотрим пример.

{

public partial class MainWindow: Window

{

public MainWindow()

{

InitializeComponent();

}

private void submit Click(object sender, RoutedEventArgs e)

{

tbResult.Text = "УРОВЕНЬ ВИЗУАЛИЗАЦИИ: \n";

// RenderCapability - Позволяет приложениям WPF запрашивать

// текущий уровень отрисовки.

// RenderCapability.Tier - Получает значение, указывающее

// уровень отрисовки текущего потока.

int renderingTier = RenderCapability.Tier >> 16;

// Значение Int32, старшее слово которого соответствует

// уровню отрисовки текущего потока.

// 16 и 17 биты.

// Возвращаемое || Уровень || Примечания

// значение || отрисовки ||

// ---------------------------------------------------

// 0x00000000 || 0 || Для приложения на этом устройстве аппаратное ускорение недоступно.

// 0x00010000 || 1 || Для данной видеокарты доступно частичное графическое аппаратное ускорение.

// 0х00020000 || 2 || Отрисовка уровня 2 означает, что большая часть графических возможностей

// || || WPF должна использовать аппаратное ускорение при условии, что необходимые системные

// || || ресурсы не исчерпаны. Это соответствует DirectX версии, которая больше или равна 9.0.

// Подробнее о том, какие возможности WPF снабжены аппаратным ускорением на перечисленных выше

// уровнях, смотрите http://msdn.microsoft.com/en-gb/library/ms742196.aspx

if (renderingTier == 0)

tbResult.Text += "Видеокарта не предоставляет аппаратного ускорения.";

if (renderingTier == 1)

tbResult.Text += "Видеокарта предоставляет частичное аппаратное ускорение.";

if (renderingTier == 2)

tbResult.Text += "Видеокарта предоставляет полное аппаратное ускорение";

}

}

}

В данном примере:

Мы имеем метод обработчик private void submit Click(object sender, RoutedEventArgs e) нажатия на кнопку. Если нажать на эту кнопку, программа проверит уровень визуализации видеокарты. Есть некий виртуальный порт, называемый Tier. Нужно обратить внимание, что у нас есть специальный объект, называемый RenderCapability. Он позволяет приложениям запрашивать текущий уровень визуализации. RenderCapability.Tier получает значение, указывающее уровень отрисовки текущего потока. В 16-ричном представлении Tier имеет 4 байта. Он типа int. Если в третьем байте у нас хранится 1 или 2, это говорит о том, что его биты что-то означают. Эти биты находятся в третьем байте. Значит это 17 и 18 биты. Нам нужно сдвинуть число вправо на 16. Создаем переменную типа **int** с именем renderingTier. Берем значение регистра, применяем к нему сдвиг вправо на 16. Слева у нас вводится 16 нулей. Что это за число? В данном случае это 2. Это число показывает уровень визуализации. Далее, мы сравниваем renderingTier = 0 – видеокарта не предоставляет аппаратного ускорения. Если 1, то видеокарта есть и не поддерживает девятую версию DirectX. Но если здесь оказалось число 2, то видеокарта предоставляет полное аппаратное ускорение. Число 20000 в десятеричном выражении 131072, а в двоичном – 100000000000000000. Сдвинем это число на 16. Получилось 2.

Рассмотрим следующий пример.

class Program

{

static void Main(string[] args)

{

/\* Короткозамкнутое вычисление - техника работающая

\* по следующему принципу:

\* Если значение первого операнда в операции AND (&&)

\* ложно, то второй операнд не вычисляется, потому что

\* полное выражение в любом случае будет ложным. \*/

int MIN\_VALUE = 1;

int denominator = 0;

int item = 2;

// Условие, которое работает с использованием техники

// Короткозамкнутого вычисления.

// Если бы это выражение вычислялось полностью, то

// операция деления во втором операнде генерерировала

// бы ошибку деления на 0.

if ((denominator != 0) && (item/denominator) > MIN\_VALUE)

{

Console.WriteLine("Мы в блоке if");

}

else

{

Console.WriteLine("Мы в блоке Else");

}

Console.ReadKey();

}

}

}

В данном примере:

Создаем переменные типа **int** MIN\_VALUE = 1, denominator = 0, item = 2. Деноминатор – это делитель. Создаем условную конструкцию **if**, в условии которой указано выражение ((denominator !=0) && (item / denominator) > MIN\_VALUE). Далее мы пытаемся поделить на ноль, а этого делать нельзя. Короткозамкнутое вычисление – техника, работающая по следующему принципу: если значение первого операнда в операции AND (&&) ложно, то второй операнд не вычисляется, потому что полное выражение в любом случае будет ложным. Короткозамкнутое вычисление работает только тогда, когда у нас используется двойной амперсанд (&&), а не одинарный (&). Если поменять местами (item / denominator) > MIN\_VALUE && (denominator !=0)), то мы не выполнимся, так как высветится ошибка. Это тоже необходимо учитывать.

Рассмотрим **теоремы Де Моргана**.

/\* Преобразования логических переменных в соответствии с теоремами

\* Де Моргана. Для применения теорем Де Моргана к логическому

\* оператору AND или OR и паре операндов требуется инвертировать

\* оба операнда, заменить AND на OR или OR на AND и инверитровать

\* все выражение полностью.

\* Исходное выражение Эквивалентное выражение

\* !A & !B = !(A | B)

\* !A & B = !(A | !B)

\* A & !B = !(!A | B)

\* !A | !B = !(A & B)

\* !A | B = !(A & !B)

\* A | !B = !(!A & B)

\* A | B = !(!A & !B)

\*/

namespace Теоремы\_Де\_Моргана

{

class Program

{

static void Main(string[] args)

{

bool A = true, B = false;

// Условие до применения теоремы Де Моргана

if (!A || !B)

Console.WriteLine("!A || !B = {0}", !A || !B);

else

Console.WriteLine("!A || !B = {0}", !A || !B);

// Условие после применения теоремы Де Моргана

if (!(A && B))

Console.WriteLine("!(A && B) = {0}", !(A && B));

else

Console.WriteLine("!(A && B) = {0}", !(A && B));

Console.ReadKey();

}

}

}

В данном примере:

Имеем некое исходное выражение. Например, НЕ А И НЕ В. Можно представить, что А и В – это истина или ложь. НЕ false – true. НЕ false – true. True И true даст нам true. Это выражение можно представить иначе. Мы можем создать эквивалентное выражение. Для применения теорем Де Моргана к логическому оператору AND или OR и паре операндов необходимо инвертировать оба операнда, заменить AND на OR или OR на AND и инвертировать все выражение полностью. Мы к отрицанию добавили еще одно отрицание. Отрицание на отрицание можно убрать. Все четные отрицания в логике убираются. НЕ правда = правда. !A & !B = !(A | B). **В каких-то случаях теоремы Де Моргана покажутся вам полезными просто лишь для удобства представления**. Просто имея такое выражение, можно вынести за скобки знак отрицания, поменять знак операции и сделать просто более читаемо. **Если поставить одинарные амперсанды и вертикальные слеши, у нас будут работать побитовые операции**.

## Циклические конструкции

**Цикл** – управляющая конструкция, предназначенная для организации многократного исполнения набора инструкций. Один проход цикла называется итерацией.

Циклом может называться любая многократно исполняемая последовательность инструкций, организованная любым способом (например, с помощью условного или безусловного перехода).

ПОКА: (Условие удовлетворяет истинности)

НАЧАЛО ЦИКЛА:

Выполнить эту серию инструкций;

КОНЕЦ ЦИКЛА:

**goto** (с англ. перейти к) – **оператор безусловного перехода** к определенной точке программы, обозначенной меткой перехода.

Label: // Метка перехода.

Console.WriteLine(“Hello!”);

**goto** Label;

Рассмотрим пример.

class Program

{

static void Main(string[] args)

{

// Label - метка, на которую будет произведен

// переход оператором goto.

Label: Console.WriteLine("Hello!");

goto Label;

}

}

}

В данном примере:

Создаем метку перехода с именем Label и пишем инструкцию языка C#. Главное, чтобы серия инструкций шла за меткой. Строка «Hello!» будет выводиться непрерывно.

Рассмотрим следующий пример.

class Program

{

static void Main(string[] args)

{

int counter = 0;

Label:

counter++;

Console.WriteLine("Counter = {0}", counter);

if (counter < 3)

{

goto Label;

}

Console.WriteLine("End");

Console.ReadKey();

}

}

}

В данном примере:

Создали переменную counter и присвоили ей значение 0. После этого создали метку перехода с именем Label. Она является лишь неким виртуальным адресом, на который нужно перейти. На следующую за меткой инструкцию будет производиться переход оператором **goto**. Далее, увеличиваем counter на 1 (counter++). Он был равен нулю, теперь он равен 1. Выводим новое значение на экран. После этого создаем условную конструкцию **if**, где counter < 3. Сейчас значение переменной равно 1, поэтому мы входим в условную конструкцию. После этого отправляемся в метку Label и переменной counter присваивается значение 2. Выводим его на экран и входим еще раз в условную конструкцию, которая вновь посылает нас в метку Label. В данном случае переменной counter присваивается значение 3, которое впоследствии отображается на экране. После этого мы не входим в условную конструкцию (counter = 3 является false), а выводим на экран надпись «End».

После оператора **goto** должно стоять какое-то завершающее условие, иначе он будет отсылать посылы к метке бесконечно. Он является спорным оператором, поэтому начинающие программисты должны использовать его крайне редко.

Рассмотрим следующий пример.

class Program

{

static void Main(string[] args)

{

bool condition = false;

if(condition == true)

{

goto Label;

}

Console.WriteLine("First line");

Label:

Console.WriteLine("Second Line");

Console.ReadKey();

}

}

}

В данном примере:

Создали булеву переменную с именем condition. В условной конструкции **if** указали выражение condition == true. Если условие удовлетворяет истинности, выполняется вход в тело условной конструкции и переход на метку Label. Нужно обратить внимание, что теперь метка перехода стоит ниже по коду в области старших адресов, потому что вверх идет область младших адресов памяти, а вниз – старших адресов. Из этого следует, что оператор перехода **goto** может переходить как в область младших адресов (в начало программы), так и в область старших адресов (конец программы). В данном случае у нас вывело на экран «First Line», а затем «Second Line».

**Цикл с предусловием** **while** – это цикл, который выполняется до тех пор, пока условие удовлетворяет истинности.

Рассмотрим пример.

class Program

{

static void Main(string[] args)

{

int counter = 0;

while(counter < 3)

{

counter++;

Console.WriteLine("Counter {0}", counter);

}

Console.WriteLine("Произведено {0} итераций.", counter);

Console.ReadKey();

}

}

}

В данном примере:

Создали переменную counter типа **int**. Эта переменная является счетчиком итераций цикла. Далее, создали циклическую конструкцию **while** с условием counter < 3. Если условие удовлетворяет действительности, входим в тело циклической конструкции **while** и выполняем все запланированные действия. Результаты выводим на экран. После того как переменной counter присвоится значение 3, происходит игнорирование цикла и вывод на экран количества итераций.

Рассмотрим еще один пример.

class Program

{

static void Main(string[] args)

{

// Циклическая конструкция while с досрочным выходом

// из цикла break.

int counter = 0;

while (counter < 3)

{

counter++;

Console.WriteLine("Counter {0}", counter);

break;

Console.WriteLine("Эта строка не выполнится");

}

Console.WriteLine("Произведено {0} итераций", counter);

Console.ReadKey();

}

}

}

В данном примере:

Все аналогично предыдущему примеру, только добавляем оператор **break**, который прекращает работу цикла после первой итерации. Стоит отметить, что оператор **break** рационально использовать только в связке с условными конструкциями.

Рассмотрим следующий пример.

class Program

{

static void Main(string[] args)

{

// Циклическая конструкция while с пропуском

// итерации continue.

int counter = 0;

while (counter < 3)

{

counter++;

Console.WriteLine("Counter {0}", counter);

continue;

Console.WriteLine("Эта строка не выполнится");

}

Console.WriteLine("Произведено {0} итераций", counter);

Console.ReadKey();

}

}

}

В данном примере:

Все аналогично предыдущему примеру, только вместо оператора **break** у нас стоит оператор **continue**. Данный оператор возвращает нас на начало цикла (на условие). Поэтому его рационально использовать только в связке с условными конструкциями.

Рассмотрим следующий пример.

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Угадайте задуманный цвет с пяти попыток.");

Console.WriteLine("Для выхода из программы введите 'exit'.");

const int maxAttempt = 5; // Допустимое количество попыток.

int attempt = 0; // Счетчик попыток.

string color = "red"; // Задуманный цвет.

while (attempt < maxAttempt)

{

attempt++;

Console.WriteLine("Попытка {0}:", attempt);

string value = Console.ReadLine();

if (value == "exit")

{

break;

}

if (value != color)

{

continue;

}

Console.WriteLine("Поздравляем, Вы угадали с {0} попытки!", attempt);

break;

}

Console.WriteLine("Конец игры!");

Console.ReadKey();

}

}

}

В данном примере:

Пользователю дается 5 попыток для того, чтобы угадать занесенный в программу цвет. Также мы предлагаем ему выйти из программы, если пользователь не будет заинтересован в угадывании цвета. Здесь мы создали константу типа **int** с именем maxAttempt и значением 5. Далее, установили счетчик попыток и установили задуманный цвет (red) с помощью переменной color типа **string**. После этого мы создали циклическую конструкцию **while**, в условии которой находится выражение attempt < maxAttempt. Показываем пользователю, сколько попыток он совершил. Присваиваем строковой переменной value предлагаемое пользователем значение цвета. Также с помощью условной конструкции **if** и оператора **break** предусматриваем выход из программы, если пользователь не желает угадывать цвет. Также создаем вторую условную конструкцию с условием value != color и оператором **continue**, чтобы продолжить итерации, если пользователь не угадает нужный цвет с первого раза. Если пользователь угадывает цвет, на экран выводится соответствующее уведомление.

**Цикл с постусловием** **do-while** – это цикл, в котором условие проверяется после выполнения тела цикла. Тело цикла **do-while** выполняется хотя бы один раз.

Рассмотрим пример.

class Program

{

static void Main(string[] args)

{

int counter = 0;

do

{

counter++;

Console.WriteLine("Counter {0}", counter);

}

while (counter < 3);

Console.WriteLine("Произведено {0} итераций.", counter);

Console.ReadKey();

}

}

}

В данном примере:

Создали счетчик итераций цикла благодаря переменной counter. После этого создали циклическую конструкцию **do-while**, в теле которой мы увеличиваем значение счетчика на 1. Выводим на экран значение счетчика итераций. Далее, проверяем условие counter < 3. Если условие удовлетворяет истинности, выполняется переход на начало цикла и выполнение тела заново. Если не удовлетворяет, показываем количество проведенных итераций.

Рассмотрим следующий пример.

class Program

{

static void Main(string[] args)

{

int counter = 0;

do

{

counter++;

Console.WriteLine("Counter {0}", counter);

break;

Console.WriteLine("Эта строка не выполнится.");

}

while (counter < 3);

Console.WriteLine("Произведено {0} итераций.", counter);

Console.ReadKey();

}

}

}

В данном примере:

Все аналогично предыдущему примеру, только добавился оператор **break**, поэтому цикл у нас прекращается после первой итерации.

Рассмотрим следующий пример.

class Program

{

static void Main(string[] args)

{

// Циклическая конструкция do-while с пропуском

// итерации continue.

int counter = 0;

do

{

counter++;

Console.WriteLine("Counter {0}", counter);

continue;

Console.WriteLine("Эта строка не выполнится.");

}

while (counter < 3);

Console.WriteLine("Произведено {0} итераций.", counter);

Console.ReadKey();

}

}

}

В данном примере:

Все как в предыдущем примере, только вместо оператора **break** появился оператор **continue**. Последний переводит нас не на начало цикла, а в условие. Данный оператор, как и предыдущий в предыдущем примере, является абсолютно бесполезным в данном случае.

*О факториале числа*.

N – это какое-то число и восклицательный знак – обозначает факториал числа. Например, факториал числа 4! вычисляется умножением 1 на 2\*3\*4. Берем натуральный ряд тех чисел, факториал которого мы хотим вычислить. Если число 4, то натуральный ряд чисел для него – 1,2,3,4. Т.е. 1\*2\*3\*4. Зачем он нужен? Это самая важная математическая операция в комбинаторике.

Важно запомнить, что факториал 0 равен 1!

Например, у нас заказали 4 книги люди из разных городов: Москва, Владивосток, Киев, Лондон. Нам необходимо знать количество вариантов маршрута развозки. Берем и 1\*2\*3\*4 = 24. В итоге получаются 24 варианта маршрута.

Рассмотрим следующий пример.

class Program

{

// Факториал числа n (обозначается n!, произносится -

// - эн факториал) - это произведение всех натуральных чисел,

// от 1 до n включительно:

// n! = 1\*2\*3\*...\*n

// 0! = 1

// В комбинаторике факториал натруального числа n интерпретируется

// как количество перестановок множества из n элементов.

// Например, для множества {A,B,C,D} из 4-х элементов существует:

// 4! = 1 \* 2 \* 3 \* 4 = 24 перестановки:

// ABCD BACD CABD DABC

// ABDC BADC CADB DACB

// ACBD BCAD CBAD DBAC

// ACDB BCDA CBDA DBCA

// ADBC BDAC CDAB DCAB

// ADCB BDCA CDBA DCBA

static void Main(string[] args)

{

int counter = 4;

int factorial = 1;

Console.WriteLine("Факториал числа: {0}! = ", counter);

do

{

// Сначала умножение, потом декремент.

factorial \*= counter--;

// Данная строка эквивалентна:

// factorial = factorial \* counter;

// counter = counter - 1;

}

while (counter > 0);

Console.WriteLine("{0}", factorial);

Console.ReadKey();

}

}

}

В данном примере:

Создали счетчик итераций цикла counter = 4, после чего создали переменную factorial и присвоили ей значение 1. Вывели на экран факториал этого числа. Нужно обратить внимание, что Console.Write – без перевода строки. Он остается в ожидании. После этого создали циклическую конструкцию **do-while**. Переменная factorial на первой итерации получает значение factorial \* counter. Только после этого выполняется декремент counter. То есть сначала 1 \* 4. Переменная factorial получила значение 4. При этом counter стал равен 3. Далее, factorial (равен 4) умножаем на counter (равен 3) и получаем 12. Переменная factorial равна 12. При этом counter уменьшился и стал 2. На следующей итерации 12 \* 2 = 24. При этом counter уменьшился до 1. Далее, 24 \* 1 = 24. При этом counter стал равен 0 и мы покинули циклическую конструкцию.

**Цикл со счетчиком for** – это цикл, в котором переменная – счетчик итераций цикла с определенным шагом, изменяет свое значение до заданного конечного значения.

Рассмотрим пример.

class Program

{

static void Main(string[] args)

{

// Построение квадрата из звездочек

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

{

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

{

Console.Write("\*");

}

Console.WriteLine();

}

Console.ReadKey();

}

}

}

В данном примере:

Строим квадрат из звездочек. Входим в циклическую конструкцию **for**, после чего входим во вложенную конструкцию **for**. В результате у нас рисуются десять звездочек подряд на каждой из десяти строк, после чего получаем визуальный квадрат.

Рассмотрим следующий пример. Цикл Дейкстры.

// При выполнении цикла Дейкстры в каждой итерации происходит вычисление

// охраняемых условий.

// Если хотя бы одно из них истинно, выполняется соответствующая охраняемая

// команда, после чего начинается новая итерация (если истинны несколько

// охраняемых условий, выполняется только одна охраняемая команда).

// Если все охраняемые условия ложны, цикл завершается.

// Цикл Дейкстры с одним охраняющим условием и одной охраняемой командой,

// по сути, представляет собой обычный цикл с предусловием (цикл "while").

// Классическое описание цикла Дейкстры.

// do

// P1 → S1,

// ...

// Pn → Sn

// od

// где:

// do - маркер начала конструкции цикла,

// od - маркер завершения конструкции цикла,

// Pi - i-тое охраняющее условие (логическое выражение, которое может иметь

// значение "истинно" или "ложно"),

// Si - i-я охраняемая команда.

// Цикл состоит из одной или нескольких ветвей (охраняемых выражений),

// каждая из которых представляет собой пару из охраняющего условия и

// охраняемой команды.

namespace Циклические\_конструкции.Цикл\_Дейкстры

{

class Program

{

static void Main(string[] args)

{

char character = '\0';

for (;;)

{

character = Console.ReadKey().KeyChar;

switch (character)

{

case 'L': // 'L' - охраняющее условие 1.

{

Console.WriteLine("Go left"); // охраняемая

// команда.

continue;

}

case 'R': // 'R' - охраняющее условие 2.

{

Console.WriteLine("Go right"); // охраняемая

// команда.

continue;

}

}

break; // Прерывание цикла.

}

Console.ReadKey();

}

}

}

Создали символьную переменную типа **character**. Далее, создали циклическую конструкцию **for**. Нужно обратить внимание, что она бесконечна. У нее нет условия и поэтому если мы не организуем правильную логику внутри этой конструкции, правильную логику выхода, то у нас будет бесконечный цикл. Предлагаем пользователю ввести значение. Создаем переключатель **switch**, который в качестве выражения селектора принимает введенное пользователем значение. **switch** имеет два **case** с двумя постоянными выражениями L и R. Если выражение селектора совпадает с L, то выполняется блок L. Далее выполняется переход на начало цикла. Если вводим R, выполняется блок R с последующим переходом на начало цикла. Если мы введем что-то отличное от L и R, осуществляется выход из конструкции **switch** и встреча с оператором **break**. В результате цикл прерывается. В этом и заключается суть Декстры: в охране определенных команд охраняющим условием. Если охраняемые условия ложны, происходит выход из программы.

Рассмотрим следующий пример. Цикл Паук.

// Цикл паук выполняется так:

// Вычисляются охраняющие условия.

// Если существует истинное охраняющее условие, выполняется соответсвущая

// охраняемая команда.

// Вычисляются условия выхода.

// Если существует истинное условие выхода, выполняется соответствующая

// команда завершения, после чего выполнение цикла заканчивается.

// Если все условия выхода ложны, начинается следующая итерация, но только

// в том случае, если в текущей итерации было истинным хотя бы одно из

// охраняющих условий.

// Если в данной итерации оказались ложными и все охраняющие условия, и

// все условия выхода, выполняется команда альтернативного завершения Е,

// после чего выполнение цикла прерывается.

namespace Циклический\_оператор\_for.Цикл\_Паук

{

class Program

{

static void Main(string[] args)

{

char character = '\0';

bool flag = default(bool);

for(;;)

{

Start:

flag = false;

character = Console.ReadKey().KeyChar;

switch(character)

{

case 'L': // 'L' - охраняющее условие 1.

{

Console.WriteLine("Go left");

flag = true;

break; // охраняемая команда.

}

case 'R': // 'R' - охраняющее условие 2.

{

Console.WriteLine("Go right");

flag = true;

break; // охраняемая команда.

}

}

switch(character)

{

case 'x': // 'x' - условие выхода 1.

{

Console.WriteLine("Exit");

goto End; // команда завершения.

}

case 'q': // 'q' - условие выхода 2.

{

Console.WriteLine("Quit");

goto End; // комадна завершения

}

}

// Ветвь альтернативного завершения.

if(flag)

{

goto Start;

}

Console.WriteLine("Alternative exit");

End:

break; // Прерывание цикла.

}

Console.ReadKey();

}

}

}

В данном примере:

Создали символьную переменную и некий флаг flag, который будет выступать в роли индикатора для некоторого состояния. Так мы назвали и переменную. **Флаг** в программировании – это либо один бит, либо один байт. Если флаг поднят – это единица. Если флаг опущен – это 0. Флаг является просто индикатором. Ключевое слово **default** имеет круглые скобки. Мы передаем ему тип **bool**. Default переводится как по умолчанию. Этот default имеет возвращаемое значение по умолчанию для каждого типа. Для булевых типов по умолчанию является значение false. Далее, мы создали бесконечный цикл for. Сбрасываем флаг и предлагаем пользователю ввести какое-то значение. Можно ввести L, R, x и q. Мы располагаем первым и вторым **switch**. Выводим на экран «Go left», флагу присваиваем значение true. Это означает, что у нас сработала охраняемая команда. **break** – выходим из **switch**. Если флаг равен true, то выполняется безусловный переход на метку Start. При необходимости выхода нужно ввести x или q. Вводим, например, x. На экран выводится exit и осуществляется переход в End, где стоит оператор **break**, выводящий из бесконечного цикла **for**.

Рассмотрим следующий пример. Упрощенный цикл Паук.

class Program

{

static void Main(string[] args)

{

char character = '\0';

for(;;)

{

character = Console.ReadKey().KeyChar;

switch(character)

{

case 'L': // 'L' - охраняющее условие 1.

{

Console.WriteLine("Go Left");

continue; // охраняемая команда.

}

case 'R': // 'R' - охраняющее условие 2.

{

Console.WriteLine("Go Right");

continue; // охраняемая команда

}

}

switch(character)

{

case 'x': // 'x' - условие выхода 1.

{

Console.WriteLine("Exit");

break; // команда завершения.

}

case 'q': // 'q' - условие выхода 2.

{

Console.WriteLine("Quit");

break; // комадна завершения.

}

default: // Ветвь альтернативного заверешния.

{

Console.WriteLine("Alternative exit");

break;

}

}

break;

}

Console.ReadKey();

}

}

}

В данном примере:

Создали символьную переменную **character** и бесконечный цикл **for.** Предлагаем ввести пользователю значение. Если ввести L или R, выполнится один из первых двух блоков. Если же ввести x или q, выполнится один из вторых двух блоков. В случае ввода отличного от указанных значения выполнится блок **default**. Он выбрасывает из **switch** и отправляет в тело цикла **for**, где встречается оператор **break**.

Рассмотрим следующий пример. Бесконечные циклы.

class Program

{

static void Main(string[] args)

{

// 1.

while(true)

{

}

// 2.

do

{

}

while (true);

// 3.

for(;;)

{

}

Console.ReadKey();

}

}

}

В данном примере:

Показаны варианты создания бесконечных циклов.

## Методы

**Метод** – это именованная часть программы, которая может вызываться из других частей программы столько раз, сколько необходимо.

byte Addition (byte argument)

{

// argument == 1

argument = argument + 1;

// argument == 2

return argument;

}

**return** – возвратить, вернуть.

**void** – пустота. Все методы должны быть вложены в более высокоуровневые контейнеры – классы.

string MyMethod (int arg)

{

Console.WriteLine(arg);

return “Hello!”;

}

где **string** – тип возвращаемого значения,

MyMethod – идентификатор (имя метода),

**int** arg – формальные аргументы (формальные параметры),

все, что в {…} – тело метода.

**Для создания метода необходимо:**

1. Указать тип возвращаемого значения, если метод ничего не возвращает указать **void**.
2. Выбрать имя метода в соответствии с действием, которое он выполняет.
3. Если метод принимает аргументы – обязательно указать их тип и имя, если нет – оставить аргументные скобки () пустыми.
4. Если метод имеет возвращаемое значение, обязательно в теле метода должно присутствовать ключевое слово **return**. Тип возвращаемого значения метода должен соответствовать типу значения, указанному после ключевого слова **return**.

**Для вызова метода необходимо:**

1. Написать имя метода.
2. Обязательно указать после имени аргументные скобки (), если метод принимает какие-то аргументы, передать соответствующее количество аргументов соответствующего типа.

Логически методы разделяются на Функции и Процедуры. **Функции** – это методы, имеющие возвращаемое значение. **Процедуры** – это методы, не возвращающие значения.

class Program

{

// Создаем метод с именем Procedure, который ничего не

// принимает и ничего не возвращает.

// В теле метода выводим на экран строку Hello!

static void Procedure()

{

Console.WriteLine("Hello!");

}

static void Main(string[] args)

{

// В теле метода Main вызываем метод Procedure.

Procedure();

Console.ReadKey();

}

}

}

В данном примере:

Создали метод с именем Procedure, который ничего не принимает и ничего не возвращает. Однако при этом он возвращает пустоту void. В теле метода мы выводим на экран строку «Hello!». В теле метода Main вызывается метод Procedure.

Рассмотрим следующий пример.

class Program

{

// Создаем метод с именем Function, который ничего

// не принимает и возвращает строковое значение.

// В теле метода, используя ключевое слово return,

// возвращаем строку Hello!

static string Function()

{

return "Hello!";

}

static void Main(string[] args)

{

// В теле метода Main создаем строковую

// локальную переменную с именем @string

// и присваиваем ей возвращаемое значение

// метода Function.

string @string = Function();

Console.WriteLine(@string);

Console.ReadKey();

}

}

}

В данном примере:

Создаем метод с именем Function (в предыдущем примере была Procedure, которая ничего не возвращала). Он ничего не принимает и возвращает строковое значение. В теле метода мы возвращаем значение «Hello!». Создаем метод Main. Он ничего не принимает и не возвращает. В его теле создаем строковую переменную @string и присваиваем ей возвращаемое значение Function(), которое представляет собой значение «Hello!». Далее, выводим на экран значение строковой локальной переменной @string.

Рассмотрим следующий пример.

class Program

{

static string Function()

{

string word = "Hello!";

return word;

}

static void Main(string[] args)

{

string word = Function();

Console.WriteLine(word);

Console.ReadKey();

}

}

}

В данном примере:

Создали метод с именем Function, который ничего не принимает и возвращает строковое значение. В теле метода создали строковую локальную переменную с именем word и присвоили ей значение «Hello!». Далее, возвращаем значение данной переменной с помощью **return**. В теле метода Main создали строковую локальную переменную с именем word, которой присвоили значение Function. Здесь не произойдет коллизии по причине одинаковых переменных, так как мы находимся в разных областях видимости. Выводим значение переменной word на экран.

Рассмотрим следующий пример.

class Program

{

// Создаем метод с именем Function, который принимает

// один строковый аргумент и возвращает строковое значение.

// В теле меода строковой локальной переменной sentence

// присваиваем конкатенацию строк и аргумента, используя

// ключевое слово return, возвращаем значение sentence.

static string Function(string name)

{

string sentence = "Hello" + name + "!";

return sentence;

}

static void Main(string[] args)

{

string sentence = Function("Alex");

Console.WriteLine(sentence);

Console.ReadKey();

}

}

}

В данном примере:

Создали метод с именем Function, который принимает один строковый аргумент с именем name. В теле метода создали строковую локальную переменную с именем sentence и присвоили ей конкатенацию двух строк и значения аргумента. С помощью **return** возвратили значение sentence. В теле метода Main создали строковую переменную sentence и присвоили ей значение Function, которому в качестве аргумента передаем строковое значение Alex. В результате метод вернет конкатенацию строки Hello аргумента Alex и восклицательного знака.

Рассмотрим следующий пример.

class Program

{

// Сложение двух целых чисел.

// Первое слагаемое.

// Второе слагаемое.

// Сумма.

static int Add(int summand1, int summand2)

{

return summand1 + summand2;

}

static void Main(string[] args)

{

int summand1 = 1, summand2 = 3;

int sum = Add(summand1, summand2);

Console.WriteLine("{0} + {1} = {2}", summand1, summand2, sum);

Console.ReadKey();

}

}

}

В данном примере:

Создали метод с именем Add, который принимает 2 целочисленных аргумента с именами summand1 и summand2. В теле метода мы возвращаем сумму аргументов. В теле метода Main создали две локальных целочисленных переменных summand1 и summand2. Присвоили им значения 1 и 3 соответственно. Создали переменную с именем sum и присвоили ей возвращаемое значение метода Add, которому в качестве аргументов передали значения переменных summand1 и summand2. В результате метод Add вернет нам сумму значений аргументов.

Рассмотрим следующий пример.

class Program

{

// Методы, возвращающие логическое значение, называют

// методами-предикатами.

static bool And(bool a, bool b)

{

return a && b;

}

static void Main(string[] args)

{

bool operand1 = true, operand2 = true;

bool result = And(operand1, operand2);

Console.WriteLine("{0} && {1} = {2}", operand1, operand2, result);

Console.ReadKey();

}

}

}

В данном примере:

Создали метод с именем And, который принимает 2 булевых аргумента и возвращает булево значение. Иногда методы возвращающие булево значение принято сленгово называть предикатами. Возвращаем возвращаемое значение а и b с помощью **return**. В теле метода Main создали 2 булевых операнда operand1 и operand2. Присвоили им значения true. Булевой переменной result присвоили значение And, которому в качестве аргументов передаем значение операндов operand1 и operand2. Метод And вернет значение true.

Рассмотрим следующий пример.

class Program

{

static string Compare(int value1, int value2)

{

if(value1 < value2)

{

return "Comparison Less Than";

}

else if(value1 > value2)

{

return "Comparison More Than";

}

return "Comparison Equal";

}

static void Main(string[] args)

{

int operand1 = 1, operand2 = 2;

string result = Compare(operand1, operand2);

Console.WriteLine(result);

Console.ReadKey();

}

}

}

В данном примере:

Создали метод Compare, который принимает два целочисленных аргумента value1 и value2. При этом метод возвращает строковое значение. В теле метода создали условную конструкцию **if**, в условии которой указываем выражение value1 и value2. Если это условие удовлетворяет истинности, то выполняется вход в тело конструкции и возврат строки «Comparison More Than». Если аргументы равны, то на экран выводится «Comparison Equal». Последнее условие должно стоять обязательно, так как у нас метод обязательно что-то возвращает. В теле метода Main создали два целочисленных операнда operand1 и operand2, и присвоили им значения 1 и 2. Создали строковую переменную result и присвоили ей возвращаемое значение метода Compare, которому в качестве аргументов присвоили значения наших операндов.

Рассмотрим следующий пример.

class Program

{

// Использование сторожевого оператора для защиты

// номинального варианта.

static string Function(string name)

{

if(name == "fool") // Сторожевой оператор

{

return "Вы использовали недопустимое слово";

}

// Код номинального варианта.

string sentence = "Hello" + name + "!";

return sentence;

}

static void Main(string[] args)

{

string sentence = Function("fool");

Console.WriteLine(sentence);

Console.ReadKey();

}

}

}

В данном примере:

Создали метод с именем Function, который принимает один строковый аргумент с именем name и возвращает строковое значение. Создали условную конструкцию **if**. Если значение аргумента name равно fool, то на экран выводится, что человек использовал недопустимое слово при вводе. Создали строковую переменную sentence, которая предполагает нормальный исход событий, то есть если человек вводит нормальное имя.

В информатике в методах существуют входящие **in** и исходящие параметры **out**. Microsoft разбивает последние на **out** и **ref**.

Рассмотрим следующий пример.

class Program

{

// Если в теле метода выполнится изменение значения

// переменной переданной по ссылке, ее значение

// будет измненено везде.

static int Method(ref int a)

{

int b = a \* 2;

a = 5;

return b;

}

static void Main(string[] args)

{

int operand = 2;

int result = Method(ref operand);

Console.WriteLine("{0}; {1};", operand, result);

Console.ReadKey();

}

}

}

В данном примере:

Создали метод с именем Method, который принимает один целочисленный аргумент с именем а, помеченный параметром **ref**. В теле метода создали локальную переменную b и присвоили ей произведение аргумента а на 2. На следующей строке значение аргумента изменилось на 5. В методе Main создали локальную переменную operand и присвоили ей значение 2. Далее, переменной result присвоили возвращаемое значение метода Method, при этом в качестве аргумента передали ему operand. Таким образом, b = 2 \* 2 = 4. При этом operand = 5, потому что мы указали перед аргументом ключевое слово **ref**.

Рассмотрим следующий пример.

class Program

{

static int Method(out int a)

{

// Выходные параметры должны быть изменены в теле

// метода, иначе будет ошибка.

a = 2;

return a \* 2;

}

static void Main(string[] args)

{

int operand;

// out позволяет передавать в метод

// неинициализированные переменные.

int result = Method(out operand);

Console.WriteLine("{0}; {1};", operand, result);

Console.ReadKey();

}

}

}

В данном примере:

Создали метод с именем Method, принимающий целочисленный аргумент, помеченный спецификатором out. Этот метод возвращает целочисленное значение. В его теле аргументу присвоено значение 2 и возвращаемое значение этого аргумента и цифры 2. Далее, создали локальную целочисленную переменную operand. У нее отсутствует инициализация, то есть она не имеет никакого значения. Создали целочисленную локальную переменную result, присвоили ей возвращаемое значение метода и в качестве аргумента метода передали ей переменную. Отличием **out** от **ref** является возможность использования первой неинициализированных переменных в качестве аргумента.

Рассмотрим следующий пример.

class Program

{

// Выходные параметры должны быть изменены в теле

// метода, иначе будет ошибка.

static int Method(out int a)

{

// Закомментировать!

a = 1;

return 2;

}

static void Main(string[] args)

{

int operand;

// out позволяет передавать в метод

// неинициализированные переменные.

int result = Method(out operand);

Console.WriteLine("{0}; {1};", operand, result);

Console.ReadKey();

}

}

}

В данном примере:

Создали метод с именем Method, который принимает аргумент а. Данный аргумент помечен спецификатором **out**. В теле метода аргументу а присвоено значение 1. В качестве возвращаемого значения присвоена 2. Создали переменную operand, которая не инициализирована. Создали целочисленную переменную result и присвоили ей возвращаемое значение метода Method.

Рассмотрим следующий пример.

class Program

{

// Передаваемый по значению параметр уничтожается

// при возврате значения методом.

static void AddTwo (int a)

{

a = a + 2;

Console.WriteLine("Значение int a = {0}", a);

}

static void Main(string[] args)

{

Console.WriteLine("Введите число: ");

// Принимаем ввод от пользователя и преобразуем

// его в целочисленное значение.

string number = Console.ReadLine();

int result = Int32.Parse(number);

Console.WriteLine("Значение result = {0}", result);

// В качестве аргумента передается не сама переменная

// result, а ее копия.

AddTwo(result);

Console.WriteLine("Значение result = {0}", result);

Console.ReadKey();

}

}

}

В данном примере:

Создали метод с именем AddTwo, который принимает один целочисленный аргумент и ничего не возвращает. В данном методе параметру а присвоили сумму этого параметра и 2. Поэтому метод и получил название «прибавить двойку». Вывели на экран значение а. Предложили пользователю ввести любое число. Далее, конвертировали строковое представление числа в целочисленный формат. **Parse** – преобразовать. Вывели на экран это число. Вызвали метод AddTwo и в качестве аргумента передали ему значение переменной result. После этого вывели его значение. Нужно обратить внимание, что сюда передается не сам аргумент, а его копия.

Рассмотрим следующий пример.

class Program

{

// Передается ссылка на исходные аргументы, копия

// не создается.

static void AddTwo(ref int a)

{

a = a + 2;

Console.WriteLine("Значение int a = {0}", a);

}

static void Main(string[] args)

{

Console.WriteLine("Введите число: ");

// Принимаем ввод от пользователя и преобразуем

// его в целочисленное значение.

string number = Console.ReadLine();

int result = Int32.Parse(number);

Console.WriteLine("Значение result = {0}", result);

// В качестве аргумента передается сама переменная

// result, а не ее копия.

AddTwo(ref result);

Console.WriteLine("Значение result = {0}", result);

Console.ReadKey();

}

}

}

В данном примере:

Аналогично предыдущему примеру, только в качестве аргумента передается сама переменная result, а не ее копия.

Рассмотрим следующий пример:

class Program

{

static int Add(int x, int y)

{

return x + y;

}

static void Main(string[] args)

{

Console.WriteLine("Введите первое число: ");

string operand1 = Console.ReadLine();

int summand1 = Int32.Parse(operand1);

Console.WriteLine("Введите второе число: ");

string operand2 = Console.ReadLine();

int summand2 = Int32.Parse(operand2);

int sum = Add(summand1, summand2);

Console.WriteLine("{0} + {1} = {2}", summand1, summand2, sum);

Console.ReadKey();

}

}

}

В данном примере:

Аналогично примеру со сложением аргументов, только здесь мы принимаем ввод от пользователя.

Рассмотрим следующий пример.

class Program

{

static void Add(int x, int y, out int sum)

{

sum = x + y;

}

static void Main(string[] args)

{

Console.WriteLine("Введите первое число: ");

string operand1 = Console.ReadLine();

int summand1 = Int32.Parse(operand1);

Console.WriteLine("Введите второе число: ");

string operand2 = Console.ReadLine();

int summand2 = Int32.Parse(operand2);

int sum;

Add(summand1, summand2, out sum);

Console.WriteLine("{0} + {1} = {2}", summand1, summand2, sum);

Console.ReadKey();

}

}

}

В данном примере:

Все аналогично предыдущему примеру, только, ко всему прочему, задается аргумент sum. При этом sum у нас не присваивается какое-либо значение.

## Методы. Рекурсия

**Перегрузка методов** – это создание нескольких методов (два и более) с одинаковым именем, но различным типом и/или количеством параметров. Наборы параметров перегруженных методов могут отличаться порядком следования, количеством, типом. Таким образом, перегрузка методов необходима для того, чтобы избежать создания нескольких разноименных методов, выполняющих сходные действия.

Рассмотрим пример.

class Program

{

// Перегруженные методы могут отличаться типом и количеством

// аргументов, а также ref и out параметрами.

static void Function() // 1-я перегрузка.

{

Console.WriteLine("Hello!");

}

static void Function(string s) // 2-я перегрузка.

{

Console.WriteLine(s);

}

static void Function(int i) // 3-я перегрузка.

{

Console.WriteLine(i);

}

static void Function(double d) // 4-я перегрузка.

{

Console.WriteLine(d);

}

static void Function(string s, int i) // 5-я перегрузка.

{

Console.WriteLine(s+i);

}

static void Function(int i, string s) // 6-я перегрузка.

{

Console.WriteLine(i+s);

}

static void Function(ref int i, string s) // 7-я перегрузка.

{

Console.WriteLine(i+s);

}

// Перегруженные методы не могут отличаться возвращаемыми

// значениями.

// static string Function(string s) // 8-я перегрузка.

// {

// return s;

// }

// Перегруженные методы не могут отличаться друг от друга

// только параметрами ref и out!

// static void Function(out int i, string s)

// {

// i = 5;

// Console.WriteLine(i+s); // 9-я перегрузка.

// }

static void Main(string[] args)

{

Function(); // 1-я перегрузка.

Function("A"); // 2-я перегрузка.

Function(1); // 3-я перегрузка.

Function(3.14); // 4-я перегрузка.

Function("B ", 2); // 5-я перегрузка.

Function(3, " C"); // 6-я перегрузка.

int variable = 5;

Function(ref variable, "D"); // 7-я перегрузка.

Console.ReadKey();

}

}

}

В данном примере:

Создали метод с именем Function, который ничего не принимает и ничего не возвращает. Он лишь выводит на экран строку. Далее, создали одноименный метод, который принимает строковый аргумент и ничего не возвращает. После этого создали еще один одноименный метод, который принимает **double** аргумент и ничего не возвращает. На следующей стадии создали еще один одноименный метод, который принимает два аргумента: первый строковый, второй целочисленный. После этого создали еще один одноименный метод, который принимает два параметра. Стоит обратить внимание, что мы уже имеем метод с такими параметрами, однако очередность этих параметров отличается. Если отличается очередность, то мы можем сказать, что по сути это разные методы. Далее, создали одноименный метод, который первый параметр принимает как **ref** параметр. Отличие между этим и предыдущим методом заключается в том, что в первом указан входной параметр **in**, а во втором выходной параметр **ref**. В C# перегружаемые методы не отличаются по типу возвращаемых значений. В языке C# перегрузки методов не могут отличаться возвращаемыми значениями. Также перегруженные методы не могут отличаться друг от друга параметрами **ref** и **out**.

**Разновидности аргументов**.

Аргументы (параметры) бывают: позиционные, именованные и опциональные.

При вызове **позиционного** параметра аргументы передаются в соответствии с заданной позицией и их типом. При этом количество аргументов должно совпадать.

void Method(int a, string b)

{

Console.WriteLine(a + b);

}

Вызов метода:

{

Method(5, "Hello");

}

При вызове **именованного** параметра аргументы передаются в соответствии с их именем и типом. При этом позиция не имеет значения, но количество аргументов должно совпадать. Если мы хотим поменять параметры метода местами, необходимо сначала указать имя аргумента, после чего поставить двоеточие и написать значение, передаваемое этому параметру.

void Method(int a, string b)

{

Console.WriteLine(a + b);

}

Вызов метода:

{

Method(b: "Hello", a: 5);

}

При вызове **опционального** параметра аргументам можно передавать значения в соответствии с их позицией, типом и именем. При этом те аргументы, которым не передано какое-то значение, уже имеют значение по умолчанию.

void Method(int a=1, string b="Ok")

{

Console.WriteLine(a + b);

}

Вызов метода:

{

Method(b: "Hello1");

Method(5);

Method();

}

Рассмотрим пример.

class Program

{

static int Difference(int height, int width)

{

return height - width;

}

static void Main(string[] args)

{

// Метод может быть вызван так:

int difference = Difference(6, 5);

// difference = Difference(5, 6);

Console.WriteLine("Разность равна: {0}", difference);

// А может быть вызван вот так:

difference = Difference(width: 5, height: 6);

Console.WriteLine("Разность равна: {0}", difference);

Console.ReadKey();

}

}

}

В данном примере:

Создали метод с именем Difference, который принимает аргументы height и width. В теле метода мы возвращаем разность аргументов. В методе Main переменной difference присвоили возвращаемое значение метода Difference. Передали значения 6 и 5. Если поменять местами значения, то получим результат -1. Далее вызвали метод Difference, но указали имя параметра, которому мы хотим присвоить определенное значение.

Рассмотрим следующий пример.

class Program

{

static void Operation()

{

Operation("val", 10, 12.2); // 1-я перегрузка.

}

static void Operation(string value1)

{

Operation(value1, 10, 12.2); // 2-я перегрузка.

}

static void Operation(string value1, int value2)

{

Operation(value1, value2, 12.2); // 3-я перегрузка.

}

static void Operation(string value1, int value2, double value3)

{

Console.WriteLine("{0}, {1}, {2}", value1, value2, value3); // 4-я перегрузка

}

static void Main(string[] args)

{

Operation(); // 1-я перегрузка

Operation("val"); // 2-я перегрузка

Operation("val", 10); // 3-я перегрузка

Operation("val", 10, 12.2); // 4-я перегрузка

Console.ReadKey();

}

}

}

В данном примере:

Идет работа с перегрузками методов и одновременно использованием именованных параметров. Имеются 4 перегрузки методов: Operation, Operation с одним аргументом, Operation с двумя аргументами и Operation с тремя аргументами. В теле первого метода Operation вызвали перегрузку четвертого метода Operation. Сюда мы передали три параметра. В теле второго метода также вызвали перегрузку четвертого метода. В качестве первого аргумента подставляем единственный аргумент этого метода. В теле третьего метода создали еще одну перегрузку. Здесь подставляются два аргумента этого метода и вызывается перегрузка четвертого метода.

Рассмотрим следующий пример.

class Program

{

// Метод вызывается так же, как если бы это были перегрузки.

static void Operation(string value1 = "val", int value2 = 10, double value3 = 12.2)

{

Console.WriteLine("{0}, {1}, {2}", value1, value2, value3);

}

static void Main(string[] args)

{

Operation(); // 1-я перегрузка

Operation("val"); // 2-я перегрузка

Operation("val", 10); // 3-я перегрузка

Operation("val", 10, 12.2); // 4-я перегрузка

// Нельзя поставить третий параметр value3 = 12.2 вместо второго параметра

// value2 = 10!

// 12.2 не может быть приведено к int - здесь C# пытается оставить по умолчанию

// третий параметр, а не второй.

// Operation("val", 12.2); // ОШИБКА!

// В данном случае возможно использование именованных параметров. Они состоят

// из указания имени параметра, двоеточия и значения, которое мы передаем.

Operation("val", value3: 12.2);

Operation(value2: 33, value3: 12.2);

Console.ReadKey();

}

}

}

В данном примере:

Создали метод Operation с тремя опциональными аргументами. Когда мы вызвали Operation и ничего не передали – это аналог первой перегрузки из предыдущего примера. Когда мы вызвали Operation и использовали один параметр – это аналогично второй перегрузке. При этом значения второго и третьего параметров остались по умолчанию. И так далее.

class Program

{

// Возврат значений из метода Main()

// По завершении программы есть возможность вернуть значение из

// метода Main() вызывающему процессу или операционной системе.

// Возвращаемое значение метода Main() может быть только типа int.

// Обычно возвращаемое методом Main() значение указывает на

// нормальное завершение программы или на аварийное ее завершение

// из-за сложившихся ненормальных условий выполнения.

// Условно нулевое возвращаемое значение обычно указывает на

// нормальное завершение программы, а все остальные значения

// обозначают тип возникшей ошибки.

static void Main(string[] args)

{

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

Console.ReadKey();

}

}

}

По завершении программы имеется возможность вернуть значение из метода Main вызывающему процессу или ОС. Процесс – это то, что у нас находится в диспетчере задач, т.е. запущенная программа. Программа на носителе – это приложение.

**Рекурсия** – это вызов методом самого себя непосредственно (простая рекурсия) или через другой метод (сложная рекурсия).

class Program

{

// Простая рекурсия.

// В теле метода Recursion рекурсивно вызывается метод Recursion.

// Простая рекурсия - вызов методом самого себя (самовызов). При

// каждом вызове строится новая копия метода.

static void Recursion(int counter)

{

counter--;

Console.WriteLine("Первая половина метода: {0}", counter);

if (counter != 0)

Recursion(counter);

Console.WriteLine("Вторая половина метода: {0}", counter);

}

static void Main(string[] args)

{

Recursion(3);

Console.ReadKey();

}

}

}

В данном примере:

Создали метод с именем Recursion, который принимает один целочисленный аргумент и в теле у нас организована какая-то логика. В теле метода Main мы вызвали метод Recursion, аргументу которого было присвоено значение 3. Переменная **counter** уменьшает значение на 1. Таким образом, при первой итерации ее значение равно 2. Заходим в условную конструкцию, так как у нас переменная не равна 0. Еще раз вызвали метод Recursion, который создался в виде копии. В копии у нас **counter** стала равняться 1 и создалась еще одна копия метода Recursion. Теперь в этой копии наша переменная стала равна 0.

**Сложная рекурсия**

class Program

{

// Сложная рекурсия - вызов методом себя через другой метод.

static void Recursion(int counter)

{

counter--;

Console.WriteLine("Первая половина метода Recursion: {0}", counter);

if (counter != 0)

Method(counter);

Console.WriteLine("Вторая половина метода Recursion: {0}", counter);

}

static void Method(int counter)

{

Console.WriteLine("Первая половина метода Method: {0}", counter);

Recursion(counter);

Console.WriteLine("Вторая половина метода Method: {0}", counter);

}

static void Main(string[] args)

{

Method(3);

Console.ReadKey();

}

}

}

В данном примере:

Создали метод Recursion и присвоили в качестве аргумента целочисленную переменную counter. Если она не равна нулю, то вызываем другой метод Method, который, в свою очередь, снова вызывает метод Recursion.

Рассмотрим пример:

class Program

{

// Вывод в двоичном формате числа, которое было передано в десятичном формате.

static void Converter(int n)

{

int temp;

temp = n % 2;

if (n >= 2)

Converter(n / 2);

Console.Write(temp);

}

static void Main(string[] args)

{

int n = 69;

Converter(n);

Console.ReadKey();

}

}

}

В данном примере:

С помощью рекурсии мы число 69 выводим в двоичном формате. В результате получается 1000101.

Рассмотрим еще один пример:

class Program

{

static int Calculate(int a, int b)

{

if (a % b == 0)

return b;

else

return Calculate(b, a % b);

}

static void Main(string[] args)

{

Console.WriteLine("Нахождение наибольшего общего делителя двух целых чисел");

int a = 15, b = 33;

Console.WriteLine("a = {0}, b = {1}, НОД = {2}", a, b, Calculate(a, b));

Console.ReadKey();

}

#region Второй вариант решения

static int Calculate2(int a, int b)

{

while (b != 0)

b = a % (a = b);

return a;

}

#endregion

}

}

В данном примере:

НОД = 3.

Рассмотрим еще один пример.

class Program

{

static int Factorial(int n)

{

if (n == 1)

return 1;

else

return n \* Factorial(n - 1);

}

static void Main(string[] args)

{

int factorial = Factorial(5);

Console.WriteLine(factorial);

Console.ReadKey();

}

}

}

В данном примере:

Factorial (5) = 5 \* 4 \* 3 \* 2 \* 1 = 120.

## Массивы

**Массив** – это именованный набор однотипных переменных, расположенных в памяти непосредственно друг за другом, доступ к которым осуществляется по индексу.

**byte**[] array = **new byte**[3];

Определенный участок памяти называется элементом массива. При этом целый массив имеет определенное имя, а каждый элемент – индекс.

**new** – это ключевое слово, которое ставится после знака = (знака присвоения).

**Стек** – это область памяти в ОЗУ для хранения адресов возврата для процедур.

**Куча (heap)** – это область памяти, где располагаются все важные, главные и живые объекты программы.

Массивы и являются такими большими и полноценными объектами, которые располагаются в куче. Для того чтобы массивы располагались в определенной области памяти, необходимо использовать ключевое слово **new.**

**Синтаксический сахар** – это те элементы языка, которые нужны человеку, но они абсолютно не нужны машине. Ключевое слово **new** предназначено для того, чтобы ОС выделила область памяти на куче, и в этой области памяти будет размещаться массив. Ключевое слово **new** пишется при создании массива, но не всегда.

Рассмотрим пример.

class Program

{

static void Main(string[] args)

{

// Создаем массив целочисленных элементов с именем array,

// размерностью в 5 элементов.

int[] array = new int[5];

// Элементам массива array по индексу 0,1,2 и т.д.

// присваиваем значение 10.

array[0] = 10;

array[1] = 20;

array[2] = 30;

array[3] = 40;

array[4] = 50;

// Выводим на экран значения элементов массива array.

Console.WriteLine(array[0]);

Console.WriteLine(array[1]);

Console.WriteLine(array[2]);

Console.WriteLine(array[3]);

Console.WriteLine(array[4]);

Console.ReadKey();

}

}

}

В данном примере:

Создали массив целочисленных элементов размерностью 5 элементов. Каждый элемент имеет индекс (0, 1, 2, 3, 4). При этом нумеровка элементов идет с 0. Также каждый проиндексированный элемент хранит определенное значение (10, 20, 30, 40, 50).

**Имя массива** – это целое число или значение типа (приводимого к целому), указывающее на конкретный элемент массива.

**Одномерный массив** – это массив, содержащий один индекс.

Четыре способа создания одномерных массивов:

**byte**[] array = **new byte**[3];

**byte**[] array = **new byte**[3] {1, 2, 3};

**byte**[] array = **new byte**[] {1, 2, 3};

**byte**[] array = {1, 2, 3};

Блок в скобках {1, 2, 3} – блок инициализатора, в котором указываются те значения, которые необходимо присвоить элементам массива. Если использовать блок инициализатора, можно не указывать выделяемую или желаемую размерность массива.

Рассмотрим следующий пример.

class Program

{

static void Main(string[] args)

{

// Одномерный массив.

int[] array = new int[5];

// Заполнение массива.

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

{

array[i] = i \* 2;

}

// Вывод на экран значений элементов массива.

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

{

Console.WriteLine(array[i]);

}

Console.ReadKey();

}

}

}

В данном примере:

В цикле **for** начинаем заполнять массив некими значениями. Результат: 0, 2, 4, 6, 8.

Рассмотрим следующий пример.

class Program

{

static void Main(string[] args)

{

// Создаем массив из заполняем его значениями в блоке инициализатора.

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

// Выводим на экран значения элементов массива.

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

{

Console.WriteLine(array[i]);

}

Console.ReadKey();

}

}

}

В данном примере:

Создали массив элементов размерностью 5 элементов и в блоке инициализатора заполнили этот массив значениями. Однако такой сценарий актуален исключительно для небольших массивов. Результат: 1, 2, 3, 4, 5.

Рассмотрим следующий пример.

class Program

{

static void Main(string[] args)

{

// Создаем массив из заполняем его значениями в блоке инициализатора.

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

// Выводим на экран значения элементов массива.

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

{

Console.WriteLine(array[i]);

}

Console.ReadKey();

}

}

}

Рассмотрим следующий пример.

class Program

{

static void Main(string[] args)

{

// Создаем массив из заполняем его значениями в блоке инициализатора.

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

// Выводим на экран значения элементов массива.

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

{

Console.WriteLine(array[i]);

}

Console.ReadKey();

}

}

}

Рассмотрим следующий пример.

class Program

{

static void Main(string[] args)

{

// Создаем массив из заполняем его значениями в блоке инициализатора.

int[] array = new int[5];

// Выводим на экран значения элементов массива.

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

{

Console.WriteLine(array[i]);

}

Console.ReadKey();

}

}

}

В данном примере:

Значение элементов в данном массиве равно 0, 0, 0, 0, 0, так как мы не присвоили этим элементам определенные значения.

**Многомерные массивы** – это массивы, имеющие более одного индекса. Многомерные массивы разделаются на прямоугольные и зубчатые.

**Прямоугольные массивы** – это массивы, которые содержат несколько измерений, где все строки имеют одинаковую длину.

**Зубчатые массивы** – это массивы, которые содержат некоторое количество внутренних массивов, каждый из которых может иметь собственный уникальный верхний предел.

**Двухмерный массив** – это прямоугольный массив, содержащий два индекса.

**int**[ , ] array = **new int**[3, 3];

Рассмотрим пример.

class Program

{

static void Main(string[] args)

{

// Массивы. Двухмерный массив.

int[,] array = new int[3, 3];

array[0, 0] = 1;

array[0, 1] = 2;

array[0, 2] = 3;

array[1, 0] = 4;

array[1, 1] = 5;

array[1, 2] = 6;

array[2, 0] = 7;

array[2, 1] = 8;

array[2, 2] = 9;

Console.WriteLine(array[0, 0]);

Console.WriteLine(array[0, 1]);

Console.WriteLine(array[0, 2]);

Console.Write("\n");

Console.WriteLine(array[1, 0]);

Console.WriteLine(array[1, 1]);

Console.WriteLine(array[1, 2]);

Console.Write("\n");

Console.WriteLine(array[2, 0]);

Console.WriteLine(array[2, 1]);

Console.WriteLine(array[2, 2]);

Console.ReadKey();

}

}

}

В данном примере:

Создали двухмерный массив целочисленных элементов с именем array и размерностью 3 на 3. Данный массив является матрицей третьего порядка. Заполнили данный массив значениями от 1 до 9.

Рассмотрим следующий пример.

class Program

{

static void Main(string[] args)

{

int[,] array = new int[3, 3];

// Заполнение массива.

for(int i = 0; i < 3; i++)

{

for(int j = 0; j < 3; j++)

{

array[i, j] = i \* j + 1;

}

}

for(int i = 0; i < 3; i++)

{

for(int j = 0; j < 3; j++)

{

Console.Write("{0}", array[i, j]);

}

Console.Write("\n");

}

Console.ReadKey();

}

}

}

В данном примере:

Создали двухмерный массив целочисленных элементов размерностью 3 на 3, который является матрицей третьего порядка. Заполнили данный массив элементами. Результат: 111, 123, 135.

Рассмотрим следующий пример.

class Program

{

static void Main(string[] args)

{

// Компилятор определяет размер массива на основании выражения инициализации.

int[,] array =

{

{1, 2, 3},

{4, 5, 6},

{7, 8, 9}

};

Console.WriteLine(array);

for(int i = 0; i < 3; i++)

{

for(int j = 0; j < 3; j++)

{

Console.Write("{0}", array[i, j]);

}

Console.Write("\n");

}

Console.ReadKey();

}

}

}

В данном примере:

Создали двухмерный массив с использованием блока инициализатора. Внутри этого блока инициализатора вложили еще 3 блока инициализатора. Результат:

System.Int32[ , ]

123

456

789

Рассмотрим следующий пример.

class Program

{

static void Main(string[] args)

{

// Массивы, состоящие из одного элемента.

int[] vector = new int[1];

vector[0] = 100;

Console.WriteLine(vector[0]);

int[,] matrix = new int[1, 1];

matrix[0, 0] = 300;

Console.WriteLine(matrix[0, 0]);

Console.ReadKey();

}

}

}

В данном примере:

Нужно обратить внимание, что массив может состоять из одного элемента. Здесь создали массив из целочисленных элементов vector размерностью один элемент. По индексу 0 присвоили значение 100. Одномерные массивы иногда называют вектором. Далее, создали двухмерный массив целочисленных элементов размерностью 1 на 1 с именем matrix. Для заполнения этого единственного элемента необходимо обратиться по его индексу. Данный массив имеет ячейку [0, 0]. Результат: 100, 300.

**Трехмерный массив** – это массив, имеющий три индекса.

**byte**[ , , ] array = **new byte**[2, 2, 2];

Рассмотрим пример.

class Program

{

static void Main(string[] args)

{

// Трехмерный массив.

int[,,] array = new int[3, 3, 3];

array[0, 0, 0] = 1;

array[0, 0, 1] = 2;

array[0, 0, 2] = 3;

array[0, 1, 0] = 4;

array[0, 1, 1] = 5;

array[0, 1, 2] = 6;

array[0, 2, 0] = 7;

array[0, 2, 1] = 8;

array[0, 2, 2] = 9;

array[1, 0, 0] = 1;

array[1, 0, 1] = 2;

array[1, 0, 2] = 3;

array[1, 1, 0] = 4;

array[1, 1, 1] = 5;

array[1, 1, 2] = 6;

array[1, 2, 0] = 7;

array[1, 2, 1] = 8;

array[1, 2, 2] = 9;

array[2, 0, 0] = 1;

array[2, 0, 1] = 2;

array[2, 0, 2] = 3;

array[2, 1, 0] = 4;

array[2, 1, 1] = 5;

array[2, 1, 2] = 6;

array[2, 2, 0] = 7;

array[2, 2, 1] = 8;

array[2, 2, 2] = 9;

for(int i = 0; i < 3; i++)

{

for(int j = 0; j < 3; j++)

{

for(int k = 0; k < 3; k++)

{

Console.Write("{0}", array[i, j, k]);

}

Console.Write("\n");

}

Console.Write("\n");

}

Console.ReadKey();

}

}

}

В данном примере:

Создали трехмерный массив целочисленных элементов с именем array размерности 3 на 3 на 3.

Рассмотрим следующий пример.

class Program

{

static void Main(string[] args)

{

int[,,] array =

{

{ {1, 2, 3}, {4, 5, 6}, {7, 8, 9} },

{ {1, 2, 3}, {4, 5, 6}, {7, 8, 9} },

{ {1, 2, 3}, {4, 5, 6}, {7, 8, 9} }

};

for(int i = 0; i < 3; i++)

{

for(int j = 0; j < 3; j++)

{

for(int k = 0; k < 3; k++)

{

Console.Write("{0}", array[i, j, k]);

}

Console.Write("\n");

}

Console.Write("\n");

}

Console.ReadKey();

}

}

}

В данном примере:

Создали трехмерный массив целочисленных элементов с помощью блока инициализатора.

Рассмотрим пример **четырехмерного** массива.

class Program

{

static void Main(string[] args)

{

int[,,,] array = new int[2, 2, 2, 2];

array[0, 0, 0, 0] = 0x0;

array[0, 0, 0, 1] = 0x1;

array[0, 0, 1, 0] = 0x2;

array[0, 0, 1, 1] = 0x3;

array[0, 1, 0, 0] = 0x4;

array[0, 1, 0, 1] = 0x5;

array[0, 1, 1, 0] = 0x6;

array[0, 1, 1, 1] = 0x7;

array[1, 0, 0, 0] = 0x8;

array[1, 0, 0, 1] = 0x9;

array[1, 0, 1, 0] = 0xA;

array[1, 0, 1, 1] = 0xB;

array[1, 1, 0, 0] = 0xC;

array[1, 1, 0, 1] = 0xD;

array[1, 1, 1, 0] = 0xE;

array[1, 1, 1, 1] = 0xF;

for(int i = 0; i < 2; i++)

{

for(int j = 0; j < 2; j++)

{

for(int k = 0; k < 2; k++)

{

for(int l = 0; l < 2; l++)

{

Console.Write("{0:X}", array[i, j, k, l]);

}

Console.Write("\n");

}

Console.Write("\n");

}

Console.Write("\n");

}

Console.ReadKey();

}

}

}

В данном пример:

Результат:

01

23

45

67…

Рассмотрим следующий пример.

class Program

{

static void Main(string[] args)

{

int[,,,] array =

{

{

{ {0x0, 0x1}, {0x2, 0x3} },

{ {0x4, 0x5}, {0x6, 0x7} },

},

{

{ {0x8, 0x9}, {0xA, 0xB} },

{ {0xC, 0xD}, {0xE, 0xF} }

}

};

for(int i = 0; i < 2; i++)

{

for(int j = 0; j < 2; j++)

{

for(int k = 0; k < 2; k++)

{

for(int l = 0; l < 2; l++)

{

Console.Write("{0}", array[i, j, k, l]);

}

Console.Write("\n");

}

Console.Write("\n");

}

Console.Write("\n");

}

Console.ReadKey();

}

}

}

**Зубчатые массивы** – это массивы, которые состоят из массивов разной длины.

Рассмотрим пример.

class Program

{

static void Main(string[] args)

{

int[][] jagged = new int[3][];

jagged[0] = new int[] {1, 2};

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

jagged[2] = new int[] {1, 2, 3};

// Во внешнем цикле выполняется проход по всем вложенным массивам.

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

{

// Во внутреннем цикле выполняется обращение к каждому элементу вложенного массива.

for(int j = 0; j < jagged[i].Length; ++j)

{

Console.Write("{0}", jagged[i][j]);

}

Console.Write("\n");

}

Console.ReadKey();

}

}

}

Создали зубчатый массив, имеющий 3 подмассива. Нулевому элементу присваивается массив из двух элементов, первому – из пяти элементов и второму – из трех элементов.

Рассмотрим следующий пример.

class Program

{

static void Main(string[] args)

{

int[][] jagged = new[]

{

new int[] {1, 2},

new int[] {1, 2, 3, 4, 5},

new int[] {1, 2, 3}

};

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

{

for(int j = 0; j < jagged[i].Length; ++j)

{

Console.Write("{0}", jagged[i][j]);

}

Console.Write("\n");

}

Console.ReadKey();

}

}

}

В данном примере:

Создали зубчатый массив с использованием блока инициализатора.

Рассмотрим следующий пример.

class Program

{

static int[] ModifyArray(int[] array, int modifier)

{

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

{

array[i] = array[i] \* modifier;

}

return array;

}

static void Main(string[] args)

{

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

myArray = ModifyArray(myArray, 5);

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

{

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

}

Console.ReadKey();

}

}

}

В данном примере:

Создали массив и передали его в качестве аргумента метода, а в методе находится некий механизм обработки этого массива. Результат: 510152025.

Рассмотрим следующий пример.

class Program

{

// Ключевое слово params позволяет определить параметр метода,

// принимающий переменное количество аргументов.

static void ShowArray(params int[] array)

{

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

{

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

}

}

static void Main(string[] args)

{

ShowArray(0, 1, 2, 3, 4, 5, 6, 7, 8, 9);

Console.ReadKey();

}

}

}

В данном примере:

Создали метод с именем ShowArray с аргументом типа массива элементов целочисленного типа. params – ключевое слово, которое позволяет передавать при вызове метода не массив, а какое-то переменное количество аргументов. Данные аргументы преобразовались в массив.

Рассмотрим следующий пример.

class Program

{

// Параметр с ключевым словом params должен быть только один и стоять

// последним в списке параметров метода.

static void ShowArray(string name, params int[] array)

{

Console.Write(name);

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

{

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

}

}

static void Main(string[] args)

{

ShowArray("Numbers: ", 0, 1, 2, 3, 4, 5, 6, 7, 8, 9);

Console.ReadKey();

}

}

}

В данном примере:

Создали метод ShowArray. Первый его аргумент строковый, а воторой – массив, помеченный params. Params может использоваться только один раз в списке аргументов.

Рассмотрим следующий пример.

class Program

{

static void Main(string[] args)

{

int[] array = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

for(int i = array.Length - 1; i >=0; i--)

{

Console.WriteLine(array[i]);

}

Console.ReadKey();

}

}

}

В данном примере:

Создали массив целочисленных элементов с помощью блока инициализатора. Результат: 9, 8, 7, 6, 5, 4, 3, 2, 1.

# Объектно-ориентированное программирование (ООП)

## Введение в ООП. Классы и объекты

Объектно-ориентированное программирование – парадигма программирования, в которой основными концепциями являются понятия объектов и классов.

**Класс** – это конструкция языка, состоящая из ключевого слова **class**, идентификатора (имени) и тела.

Класс может содержать в своем теле: поля, методы, свойства и события.

Все переменные, находящиеся в пределах класса, называются полями. Поля определяют состояние, а методы поведение будущего объекта.

class MyClass

{

public string field; // Поле

public void Method() // Метод

{

Console.WriteLine(field);

}

}

**Объект** – выделенная область памяти, в которой находятся поля и методы. Класс, оказавшийся в оперативной памяти, называется объектом.

class Butterfly

{

public string name;

public void Fly()

{

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

{

Console.WriteLine("Fly!");

}

}

}

class Program

{

static void Main(string[] args)

{

Butterfly mahaon = new Butterfly();

mahaon.name = "Admiral";

}

}

}

Объекты содержат в себе статические поля и все методы. Экземпляры содержат нестатические поля.

class MyClass

{

public int field;

public void Method()

{

Console.WriteLine(field);

}

}

class Program

{

static void Main(string[] args)

{

MyClass instance1 = new MyClass();

MyClass instance2 = new MyClass();

instance1.field = 3;

instance2.field = 5;

instance1.Method();

instance2.Method();

}

}

}

Создали экземпляры класса MyClass с именами instance1 и instance2.

**Модификаторы доступа** – **private** и **public** определяют видимость членов класса.

Никогда не следует делать поля открытыми – это плохой стиль. Для обращения к полю рекомендуется использовать методы доступа.

class MyClass

{

private string field = null;

public void SetField(string value) // Метод-мутатор - mutator(setter)

{

field = value;

}

public string GetField() // Метод-аксессор - accessor(getter)

{

return field;

}

}

class Program

{

static void Main(string[] args)

{

MyClass instance = new MyClass();

instance.SetField("Hello, World!"); // Метод-мутатор

string @string = instance.GetField(); // Метод-аксессор

Console.WriteLine(@string);

Console.ReadKey();

}

}

}

**Свойство** – интерфейс доступа к полю объекта. Свойства в C# – поля с логическим блоком, в которых есть ключевые слова **get** и **set** и являются суррогатом для замены методов доступа к полю. При обращении к свойству вызывается определенный метод, который выполняет определенные операции с объектом.

class MyClass

{

private string field = null;

public string Field

{

set // void FieldSet(string value) - метод-мутатор - mutator (setter)

{

field = value;

}

get // string GetField() - метод-аксессор - accessor (getter)

{

return field;

}

}

}

class Program

{

static void Main(string[] args)

{

MyClass instance = new MyClass();

instance.Field = "Hello, world!"; // Метод-мутатор

Console.WriteLine(instance.Field); // Метод-аксессор

Console.ReadKey();

}

}

}

class MyClass

{

private string field = null;

public string Field

{

set

{

if (value == "fool")

Console.WriteLine("Вы ввели недопустимое значение. Повторите попытку.");

else

field = value;

}

get

{

if (field == null)

return "В поле field отсутствуют данные.";

else if (field == "hello world")

return field.ToUpper() + "!";

else

return field;

}

}

}

class Program

{

static void Main(string[] args)

{

MyClass instance = new MyClass();

instance.Field = "fool";

Console.WriteLine(instance.Field);

Console.WriteLine(new string('-', 50));

instance.Field = "hello world";

Console.WriteLine(instance.Field);

Console.ReadKey();

}

}

}

class Constants

{

private double pi = 3.14D;

private double e = 2.71D;

// Свойство только для записи. - WriteOnly Property

public double Pi

{

set { pi = value; }

}

// Свойство только для чтения. - ReadOnly Property

public double E

{

get { return e; }

}

}

class Program

{

static void Main(string[] args)

{

Constants constans = new Constants();

constans.Pi = 3.14159265D;

// Console.WritLine(constans.Pi); // Недопустимо.

// constans.E = 3.71D; // Недопустимо.

Console.WriteLine("e = {0}", constans.E);

Console.ReadKey();

}

}

}

**Конструктор класса** – специальный метод, который вызывается во время построения класса.

Конструкторы бывают двух видов:

* конструкторы по умолчанию,
* пользовательские конструкторы.

Пример первого:

public MyClass()

{

}

Пример второго:

public MyClass(int arg)

{

}

Если в теле класса не определен явно ни один конструктор, то всегда используется "невидимый конструктор" по умолчанию.

Имя конструктора всегда совпадает с именем класса. Конструкторы не имеют возвращаемых значений.

**Задача конструктора по умолчанию** – инициализация полей значениями по умолчанию.

**Задача пользовательского конструктора** – инициализация полей предопределенными пользовательскими значениями.

Если в классе имеется пользовательский конструктор, и при этом требуется создавать экземпляры класса с использованием конструктора по умолчанию, то конструктор по умолчанию должен быть определен в теле класса явно, иначе возникнет ошибка на уровне компиляции.

class Point

{

// Поля.

private int x, y;

// Свойства.

public int X

{

get { return x; }

}

public int Y

{

get { return y; }

}

// Конструктор по умолчанию, инициализирующий поля значениями по умолчанию.

public Point()

{

Console.WriteLine("Конструктор по умолчанию");

}

// Пользовательский конструктор, инициализирующий поля заданными пользователем значениями.

public Point(int x, int y)

{

Console.WriteLine("Пользовательский конструктор!");

this.x = x;

this.y = y;

}

}

class Program

{

static void Main(string[] args)

{

// Применяем конструктор по умолчанию.

Point pointA = new Point();

Console.WriteLine("pointA.X = {0} pointA.Y = {1}", pointA.X, pointA.Y);

Console.WriteLine(new string('-', 30));

// Применяем конструктор с параметрами.

Point pointB = new Point(100, 200);

Console.WriteLine("pointB.X = {0} pointB.Y = {1}", pointB.X, pointB.Y);

Console.ReadKey();

}

}

}

Конструктор может вызывать в том же самом объекте другой конструктор с помощью ключевого слова **this**.

Один конструктор может вызывать другой конструктор того же класса, если после сигнатуры вызывающего конструктора поставить ключевое слово **this** и указать набор параметров, который должен совпадать по количеству и типу с набором параметров вызываемого конструктора.

Вызывающий конструктор:

public Point(string name)

: this(300, 400)

{

this.name = name;

}

Вызываемый конструктор:

public Point(int x, int y)

{

this.x = x;

this.y = y;

}

Попытка вызова конструктора с несуществующим набором параметров приведет к ошибке уровня компиляции.

class Point

{

// Поля.

private int x, y;

private string name;

// Свойства.

public int X

{

get { return x; }

}

public int Y

{

get { return y; }

}

public string Name

{

get { return name; }

}

// Конструкторы.

// Использование ключевого слова this в конструкторе с одним параметром

// приводит к вызову этого конструктора.

public Point(int x, int y)

{

Console.WriteLine("Конструктор с двумя параметрами.");

this.x = x;

this.y = y;

}

// Использование ключевого слова this в конструкторе приводит к вызову конструктора с двумя параметрами

public Point(string name)

: this(300, 400)

{

Console.WriteLine("Конструктор с одним параметром.");

this.name = name;

}

}

class Program

{

static void Main(string[] args)

{

Point point = new Point("A");

Console.WriteLine("{0}.X = {1}, {0}.Y = {2}", point.Name, point.X, point.Y);

Console.ReadKey();

}

}

}

class MyClass

{

public void Method()

{

Console.WriteLine("Вызван метод класса MyClass");

}

}

class MyClass2

{

// Создаем метод с именем CallMethod, который принимает один аргумент типа MyClass

// и ничего не возвращает

public void CallMethod(MyClass my)

{

// В теле метода CallMethod вызываем метод с именем Method аргумента my

my.Method();

}

}

class Program

{

static void Main(string[] args)

{

MyClass my = new MyClass();

MyClass2 my2 = new MyClass2();

my2.CallMethod(my);

Console.ReadKey();

}

}

}

Автоматически реализуемые свойства – это более лаконичная форма свойств. Смысл их использования актуален, когда в методах доступа (get и set) не требуется дополнительная логика. При создании автоматически реализуемых свойств компилятор создаст закрытое, анонимное резервное поле, которое будет доступно с помощью методов get и set свойства.

public class MyClass

{

public string Name { get; set; }

public string Book { get; set; }

}

class Program

{

public class Author

{

public string Name { get; set; }

public string Book { get; set; }

}

static void Main(string[] args)

{

Author author1 = new Author()

{

Name = "Jeffrey Richter", // Блок инициализатора.

Book = "CLR via C#"

};

Author author2 = new Author // ()

{

Name = "Steve McConnell", // Блок инициализатора.

Book = "Code Complete"

};

Console.WriteLine("Name: {0}, Book: {1}", author1.Name, author1.Book);

Console.WriteLine("Name: {0}, Book: {1}", author2.Name, author2.Book);

Console.ReadKey();

}

}

}

class MyClass

{

public void Method()

{

Console.WriteLine("Hello world!");

}

}

class Program

{

static void Main(string[] args)

{

new MyClass().Method();

Console.ReadKey();

}

}

}

Создали экземпляр по слабой ссылке.

**Инкапсуляция** (*инкапсуляция вариаций*) – техника сокрытия частей объектно-ориентированных программных систем.

## Классы и объекты. Диаграммы классов

В C# реализована возможность разделить создание класса или метода (структуры, интерфейса) между двумя или более исходными файлами или модулями. Каждый исходный файл содержит определение типа или метода, и все части объединяются при компиляции приложения.

Для разделения класса на несколько частей используется ключевое слово **partial**.

partial class PartialClass

{

public void MethodFromPart1()

{

}

}

partial class PartialClass

{

public void MethodFromPart2()

{

}

}

class Program

{

static void Main(string[] args)

{

PartialClass instance = new PartialClass();

instance.MethodFromPart1();

instance.MethodFromPart2();

}

}

}

**Частичные методы** – это методы, где "прототип" или сигнатура метода определена при создании частичного класса, а реализация выполняется в любой другой (только одной) части этого класса.

partial class PartialClass

{

partial void PartialMethod();

partial void MyMethod();

}

partial class PartialClass

{

partial void PartialMethod()

{

/\* ... \*/

}

public void CallPartialMethod()

{

PartialMethod();

}

}

class Program

{

static void Main(string[] args)

{

PartialClass instance = new PartialClass();

instance.CallPartialMethod();

}

}

}

**Правила** использования частичных методов:

* Частичные методы должны быть определены только в частичных классах.
* Частичные методы должны быть помечены ключевым словом **partial**.
* Частичные методы всегда являются **private**, попытка явного использования с ними модификатора доступа приведет к ошибке.
* Частичные методы должны возвращать **void**.
* Частичные методы могут быть нереализованными.

Объект не может скрывать (инкапсулировать) ничего от другого объекта того же класса.

class MyClass

{

MyClass my = null;

private void Method()

{

Console.WriteLine("Hello!");

}

public void CallMethod()

{

my = new MyClass();

// private метод виден на экземпляре!

my.Method();

}

}

class Program

{

static void Main(string[] args)

{

MyClass my = new MyClass();

my.CallMethod();

// my.Method(); // Недопустимо.

}

}

}

Если поле используется с модификатором **readonly**, то присвоение значений таким полям может происходить только при создании поля или в конструкторе того же класса.

public readonly string field = "Hello!";

**readonly** – это модификатор, который можно использовать только для полей.

class Program

{

// Поле только для чтения (устанавливается только конструктором)!

public readonly string field = "Hello";

// Конструктор.

public Program()

{

field = "Поле только для чтения";

field += "!";

}

static void Main(string[] args)

{

Program program = new Program();

Console.WriteLine(program.field);

// Ошибка компиляции.

// program.field = "Попытка записи в поле только для чтения.";

Console.ReadKey();

}

}

}

**UML** (Unified Modeling Language – унифицированный язык моделирования) – язык графического описания для объектного моделирования в области разработки ПО. Он был разработан в 1997 году.

UML был создан для того, чтобы участники процесса создания ПО могли строить модели для визуализации системы, определения ее структуры и поведения, сборки системы и документирования решений, принимаемых в процессе разработки.

Диаграммы классов используются для изображения классов, а также связей между ними. Самым важным является показ классов и связей между ними с различных сторон таким способом, чтобы передать наиболее важный смысл.

Диаграмма классов представляет собой статическую модель системы. Диаграмма классов не описывает поведение системы или то, как взаимодействуют экземпляры классов.

|  |  |
| --- | --- |
| **Program** | **Speaker** |
| -Attributes  +speaker: Speaker | -Attributes |
| -Operations  +Main(): void | -Operations  +SayHello(): void |

Основные элементы языка UML – это прямоугольник и линия.

Прямоугольник символизирует некоторый класс или структуру, или какой-либо другой стереотип. Линия представляет собой одну из нескольких связи отношений.

Прямоугольник принято называть классификатором.

Секции классификатора:

|  |
| --- |
| **MyClass** |
| field: int |
| Method(): int |
| Обязанности:  - Вернуть строку "Hello!"  - … |

Первая строка – Секция Имени

Вторая строка – Секция Атрибутов

Третья строка – Секция Операций

Четвертая строка – Секция Обязанностей (необязательная)

В UML поля называются Атрибутами.

*Для написания тасков в редакторе VS нужно указать* // todo: нужно сделать это, *а после перейти во вкладку* Task List *в ниженей части редактора.*

Стереотипы в Microsoft VS:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Стереотип** | **Пример** | **Стереотип** | **Пример** | **Стереотип** | **Пример** |
| Class | MyClass | Generic Class | MyGenericClass<Type> | Generic Delegate | MyDelegate<T, R> |
| Abstract Class | MyAbstractClass | Generic Abstract Class | MyAbstractClass<Type> | Delegate | MyDelegate |
| Interface | MyInterface | Generic Interface | MyInterface<Type> | Enum | MyEnum |
| Stuct | MyStructure | Generic Struct | MyStructure<Type> |  |  |

Использование техники фабричных методов.

// Использование техники фабричных методов.

class Product

{

public Product()

{

Console.WriteLine("Создан экземпляр класса Product");

}

}

class Factory

{

public Product FactoryMethod()

{

return new Product();

}

}

class Program

{

static void Main(string[] args)

{

Factory factory = new Factory();

Product product = factory.FactoryMethod();

Console.ReadKey();

}

}

}

Создание не фабричного метода (*не рекомендуется*).

class MyClass1

{

public MyClass1()

{

Console.WriteLine("Создан экземпляр класса MyClass1");

}

}

class MyClass2

{

private MyClass1 myObj = null;

// Данный метод не является фабричным.

public void Method()

{

myObj = new MyClass1();

}

}

class Program

{

static void Main(string[] args)

{

MyClass2 my = new MyClass2();

my.Method();

Console.ReadKey();

}

}

}

## Наследование и полиморфизм

**Наследование** – механизм объектно-ориентированного программирования (наряду с инкапсуляцией, полиморфизмом и абстракцией), позволяющий описать новый класс на основе уже существующего (родительского). При этом свойства и функциональность родительского класса заимствуются новым классом.

Базовый класс – Производный класс или

Супер класс – Подкласс или (сабкласс) или

Родительский класс – Дочерний класс или

Родитель - Потомок

class Basic

{

public int field1;

public void Method()

{

/\* ... \*/

}

}

class Derived : Basic

{

public int field2;

}

class Program

{

static void Main(string[] args)

{

Derived derived = new Derived();

derived.field1 = 5;

derived.field2 = 8;

derived.Method();

Console.ReadKey();

}

}

}

**Модификаторы доступа** – это ключевые слова, задающие доступность члена или типа. При помощи модификаторов доступа можно задавать уровни доступа к членам.

**public** – доступ к типу или члену возможен из любого другого кода в той же сборке или другой сборке, ссылающейся на него.

**protected** – доступ к типу или элементу можно получить только из кода в том же классе или структуре, либо в производном классе.

**private** – доступ к типу или члену можно получить только из кода в том же классе или структуре.

**Связанность** – мера зависимости одного объекта от другого.

**Связность** – мера самодостаточности объекта. Определяет функциональную полноту объекта.

// Наследование

class BaseClass

{

// Поля

public string publicField = "BaseClass.publicField";

private string privateField = "BaseClass.privateField";

protected string protectedField = "BaseClass.protectedField";

// Методы

public void Show()

{

Console.WriteLine(privateField);

}

}

class DerivedClass : BaseClass

{

// Конструктор.

public DerivedClass()

{

publicField = "DerivedClass.publicField";

protectedField = "DerivedClass.protectedField";

}

}

class Program

{

static void Main(string[] args)

{

DerivedClass instance = new DerivedClass();

Console.WriteLine(instance.publicField);

instance.Show();

Console.ReadKey();

}

}

}

class BaseClass

{

public int baseNumber;

// Конструктор по умолчанию.

public BaseClass()

{

}

// Пользовательский конструктор.

public BaseClass(int baseNumber)

{

this.baseNumber = baseNumber;

}

}

class DerivedClass : BaseClass

{

public int derivedField;

// Пользовательский конструктор.

// При создании объекта производного класса конструктор производного класса

// автоматически вызывает конструктор по умолчанию из базового класса.

// Конструктор базового класса присвоит всем данным какие-то свои безопасные значения.

// После этого начнет работу конструктор производного класса, который повторно будет

// определять значения для унаследованых членов. (ДВОЙНАЯ РАБОТА!)

public DerivedClass(int number1, int number2)

{

// Инициализируем поле базового класса.

baseNumber = number1;

// Инициализируем поле производного (данного) класса.

derivedField = number2;

}

}

class Program

{

static void Main(string[] args)

{

DerivedClass instance = new DerivedClass(1, 2);

Console.WriteLine(instance.baseNumber);

Console.WriteLine(instance.derivedField);

Console.ReadKey();

}

}

}

Вызов конструктора базового класса.

Использование ключевого слова **base** для вызова конструктора базового класса.

public class BaseClass

{

public BaseClass()

{

Console.WriteLine("Base");

}

}

public class DerivedClass : BaseClass

{

public DerivedClass()

: base()

{

Console.WriteLine("Derived");

}

}

Полиморфизм.

class BaseClass

{

public int field1;

public int field2;

public int field3;

}

class DerivedClass : BaseClass

{

public int field4;

public int field5;

}

class Program

{

static void Main(string[] args)

{

DerivedClass instance = new DerivedClass();

instance.field1 = 1;

instance.field2 = 2;

instance.field3 = 3;

instance.field4 = 4;

instance.field5 = 5;

// Приведение экземпляра класса DerivedClass к базовому типу BaseClass.

BaseClass newInstance = (BaseClass)instance; // Upcast

Console.WriteLine(newInstance.field1);

Console.WriteLine(newInstance.field2);

Console.WriteLine(newInstance.field3);

// Console.Writeline(newInstance.field4); // не отображаются

// Console.Writeline(newInstance.field5); // в объекте newInstance.

// Проверка.

Console.WriteLine("instance ID {0}", instance.GetHashCode());

Console.WriteLine("newInstance ID {0}", newInstance.GetHashCode());

Console.ReadKey();

}

}

Техника замещения методов.

class BaseClass

{

public void Method()

{

Console.WriteLine("Method from BaseClass");

}

}

class DerivedClass : BaseClass

{

public void Method()

{

// Замещение метода базового класса.

Console.WriteLine("Method from DerivedClass");

}

}

class Program

{

static void Main(string[] args)

{

DerivedClass instance = new DerivedClass();

instance.Method();

//UpCast

BaseClass instanceUp = instance;

instanceUp.Method();

//DownCast

DerivedClass instanceDown = (DerivedClass)instanceUp;

instanceDown.Method();

Console.ReadKey();

}

}

Приведение к базовому типу используется для сокрытия реализации членов производного класса.

BaseClass instance = new DerivedClass();

Переменная instance типа BaseClass хранит ссылку на экземпляр класса DerivedClass.

UpCast и DownCast

**UpCast** – приведение экземпляра производного класса к базовому типу.

BaseClass up = new DerivedClass();

**DownCast** – приведение экземпляра базового типа к производному типу.

DerivedClass down = (DerivedClass)up;

DownCast невозможен без предварительного UpCast.

**Полиморфизм** – возможность объектов с одинаковой спецификацией иметь различную реализацию.

Формы полиморфизма:

1. Ad-hoc полиморфизм.
2. Классический (принудительный) полиморфизм.
   * использование виртуальных членов (переопределение **virtial/override**).
   * приведение типов.

В случае одновременного использования двух форм классического полиморфизма первая форма нейтрализует вторую (доминирует над второй).

class BaseClass

{

public virtual void Method()

{

Console.WriteLine("Method from BaseClass");

}

public void NVMethod()

{

Console.WriteLine("NV BaseClass");

}

}

class DerivedClass : BaseClass

{

// Переопределение метода базового класса.

public override void Method()

{

Console.WriteLine("Method from DerivedClass");

}

public void NVMethod()

{

Console.WriteLine("NV DerivedClass");

}

}

class Program

{

static void Main(string[] args)

{

DerivedClass instance = new DerivedClass();

instance.Method();

instance.NVMethod();

// UpCast

BaseClass instanceUp = instance;

instanceUp.Method();

instanceUp.NVMethod();

// DownCast

DerivedClass instanceDown = (DerivedClass)instanceUp;

instanceDown.Method();

instanceDown.NVMethod();

Console.ReadKey();

}

}

Комбинирование виртуальных и не виртуальных методов. В случае с установкой ключевых слов **virtual/override** на экран три раза выводится содержимое находящегося в DerivedClass метода, а в случае с замещением методов при апкасте выводится содержимое метода базового класса.

class BaseClass

{

public virtual void Method()

{

Console.WriteLine("Method from BaseClass");

}

}

class DerivedClass : BaseClass

{

// Переопределение метода базового класса.

public override void Method()

{

// Вызов метода базового класса.

base.Method();

Console.WriteLine("Method from DerivedClass");

}

}

class Program

{

static void Main(string[] args)

{

DerivedClass instance = new DerivedClass();

instance.Method();

Console.ReadKey();

}

}

Вызов метода базового класса.

class ClassA { /\*...\*/}

class ClassB : ClassA { /\*...\*/}

class Program

{

static void Main(string[] args)

{

ClassB b = new ClassB();

ClassA a = null;

//-------------------------is--------------------------------------------

// Оператор is проверяет совместимость объекта с заданным типом.

// Если предоставленный объект может быть приведен к предоставленному типу,

// не вызывая исключение, выражение is принимает значение true.

// Например, в следующем коде определяется, является ли объект экземпляром

// типа А или типа, производного от А:

if(b is ClassA)

{

a = (ClassA)b;

}

else

{

a = null;

}

//------------------------as-----------------------------------------------

// Оператор as используется для выполнения преобразований между совместимыми

// ссылочными типами. Оператор as подобен оператору приведения. Однако, если

// преобразование невозможно, as возвращает значение null, а не вызывает исключение.

// В общем виде логика работы оператора as представляет собой механизм использования

// оператора is (пример - вышеуказанная условная конструкция) только в сокращенном виде.

a = b as ClassA;

Console.ReadKey();

}

}

class BaseClass

{

public virtual void Method()

{

Console.WriteLine("Method from BaseClass");

}

}

class DerivedClass : BaseClass

{

// Переопределение методов базового класса.

public override void Method()

{

Console.WriteLine("Method from DerivedClass");

}

}

class Program

{

static void Main(string[] args)

{

// Cast с использованием оператора as.

DerivedClass instance = new DerivedClass();

instance.Method();

// UpCast

BaseClass instanceUp = instance as BaseClass;

instanceUp.Method();

// DownCast

DerivedClass instanceDown = instanceUp as DerivedClass;

instanceDown.Method();

Console.ReadKey();

}

}

UpCast и DownCast с использованием ключевого слова **as**. Данное приведение переменной считается безопасным.

При применении к классу модификатор **sealed** запрещает другим классам наследоваться от этого класса.

Модификатор **sealed** можно использовать с методами или свойствами. Это позволяет запретить переопределять виртуальные методы или свойства в производных классах.

// Герметизированные классы.

sealed class SealedClass

{

public int x;

public int y;

}

// Попытка наследования от SealedClass приводит к ошибке компилятора.

class DerivedClass : //: SealedClass // Error

{

}

class Program

{

static void Main(string[] args)

{

SealedClass instance = new SealedClass();

instance.x = 100;

instance.y = 200;

Console.WriteLine("x = {0}, y = {1}", instance.x, instance.y);

Console.ReadKey();

}

}

// Sealed методы.

partial class ClassA

{

public virtual void Method1() { Console.WriteLine("ClassA.Method1"); }

public virtual void Method2() { Console.WriteLine("ClassA.Method2"); }

}

partial class ClassA

{

}

class ClassB : ClassA

{

sealed public override void Method1(){ Console.WriteLine("ClassB.Method1"); }

public override void Method2(){ Console.WriteLine("ClassB.Method2"); }

}

class ClassC: ClassB

{

// Попытка переопределить Method1 приводит к ошибке компилятора: CS0239.

// public override void Method() {Console.WriteLine("ClassC.Method1"); }

// Переопределение Method2 позволено.

public override void Method2() { Console.WriteLine("ClassC.Method2"); }

}

class BaseClass

{

}

class Derived1 : BaseClass

{

}

class Derived2 : BaseClass

{

}

class Container

{

public BaseClass field;

}

class Program

{

static void Main(string[] args)

{

Container instance = new Container();

instance.field = new Derived1();

Console.WriteLine(instance.field.GetHashCode());

instance.field = new Derived2();

Console.WriteLine(instance.field.GetHashCode());

Console.ReadKey();

}

}

Наглядный пример настоящего полиморфизма.

## Абстрактные классы и интерфейсы

**Абстракция** в ООП – это придание объекту характеристик, которые отличают его от всех других объектов, четко определяя его концептуальные границы.

**Абстрагирование** в ООП – это способ выделить набор значимых характеристик объекта, исключая из рассмотрения незначимые. Соответственно, абстракция – это набор всех таких характеристик.

**Абстрактный класс** в ООП – это базовый класс, который не предполагает создания экземпляров через вызов конструктора напрямую, но экземпляр абстрактного класса создается неявно при построении экземпляра производного конкретного класса.

abstract class MyClass

{

}

Ключевое слово **abstact** может использоваться с классами, методами, свойствами, индексаторами и событиями.

Возможности и ограничения абстрактных классов:

* Экземпляр абстрактного класса нельзя создать через вызов конструктора напрямую, но экземпляр абстрактного класса создается неявно при построении экземпляра производного конкретного класса.
* Абстрактные классы могут содержать как абстрактные, так и не абстрактные члены.
* Не абстрактный (конкретный) класс, являющийся производным от абстрактного, должен содержать фактические реализации всех наследуемых абстрактных членов.

// Абстрактный класс.

abstract class AbstractClass

{

public abstract void Method();

}

// Конкретный класс.

class ConcreteClass : AbstractClass

{

public override void Method()

{

Console.WriteLine("Implementation");

}

}

class Program

{

static void Main(string[] args)

{

AbstractClass instance = new ConcreteClass();

instance.Method();

Console.ReadKey();

}

}

// Абстрактный класс может быть унаследован от конкретного класса.

// Конкретный класс А.

class ConcreteClassA

{

public void Operation()

{

Console.WriteLine("ConcreteClassA.Operation");

}

}

// Абстрактный класс.

abstract class AbstractClass : ConcreteClassA

{

public abstract void Method();

}

// Конкретный класс В.

class ConcreteClassB : AbstractClass

{

public override void Method()

{

Console.WriteLine("ConcreteClassB.Method");

}

}

class Program

{

static void Main(string[] args)

{

AbstractClass instance = new ConcreteClassB();

instance.Method();

instance.Operation();

Console.ReadKey();

}

}

// Абстрактный класс может быть унаследован от абстрактного класса.

// Реализация абстрактного метода из базового абстрактного класса в производном абстрактном классе не обязательна.

// Абстрактный класс А.

abstract class AbstactClassA

{

public abstract void OperationA();

}

// Абстрактный класс В.

abstract class AbstactClassB : AbstactClassA

{

public abstract void OperationB();

}

// Конкретный класс.

class ConcreteClass : AbstactClassB

{

public override void OperationA()

{

Console.WriteLine("ConcreteClass.OperationA");

}

public override void OperationB()

{

Console.WriteLine("ConcreteClass.OperationB");

}

}

class Program

{

static void Main(string[] args)

{

AbstactClassA instance = new ConcreteClass();

instance.OperationA();

// instance.OperationB(); // данный метод недоступен, потому что мы привели объект к классу A,

// а класс B инкапсулировался.

Console.ReadKey();

}

}

abstract class AbstractBaseClass

{

//1. Обычный метод передается производному классу как при наследовании от конкретного класса.

public void SimpleMethod()

{

Console.WriteLine("AbstractBaseClass.SimpleMethod");

}

//2. Виртуальный метод передается производному классу как при наследовании от конкретного класса.

public virtual void VirtualMethod()

{

Console.WriteLine("AbstractBaseClass.VirtualMethod");

}

//3. Абстрактный метод - реализуется в производном классе.

public abstract void AbstractMethod();

}

class ConcreteDerivedClass : AbstractBaseClass

{

// Переопределяем виртуальный метод VirtualMethod() базового абстрактного класса.

// Если мы не переопределим виртуальный метод, то будет использован метод из базового класса.

public override void VirtualMethod()

{

Console.WriteLine("DerivedClass.VirtualMethod();");

}

// Реализуем абстрактный метод AbstractMethod() базового абстрактного класса.

public override void AbstractMethod()

{

Console.WriteLine("DerivedClass.AbstractMethod();");

}

}

class Program

{

static void Main(string[] args)

{

ConcreteDerivedClass instance = new ConcreteDerivedClass();

instance.SimpleMethod();

instance.VirtualMethod();

instance.AbstractMethod();

Console.ReadKey();

}

}

abstract class AbstractBaseClass

{

//1. Обычный метод передается производному классу как при наследовании от конкретного класса.

public void SimpleMethod()

{

Console.WriteLine("AbstractBaseClass.SimpleMethod");

}

//2. Виртуальный метод передается производному классу как при наследовании от конкретного класса.

public virtual void VirtualMethod()

{

Console.WriteLine("AbstractBaseClass.VirtualMethod");

}

//3. Абстрактный метод - реализуется в производном классе.

public abstract void AbstractMethod();

}

class ConcreteDerivedClass : AbstractBaseClass

{

// Переопределяем виртуальный метод VirtualMethod() базового абстрактного класса.

// Если мы не переопределим виртуальный метод, то будет использован метод из базового класса.

/\* public override void VirtualMethod()

{

Console.WriteLine("DerivedClass.VirtualMethod();");

} \*/

// Реализуем абстрактный метод AbstractMethod() базового абстрактного класса.

public override void AbstractMethod()

{

Console.WriteLine("DerivedClass.AbstractMethod();");

}

}

class Program

{

static void Main(string[] args)

{

ConcreteDerivedClass instance = new ConcreteDerivedClass();

instance.SimpleMethod();

instance.VirtualMethod();

instance.AbstractMethod();

Console.ReadKey();

}

}

abstract class AbstractClass

{

// Конструктор (отрабатывает первым).

public AbstractClass()

{

Console.WriteLine("1 AbstractClass()");

// Вызывается реализация метода из производного класса ConcreteClass.

this.AbstractMethod();

Console.WriteLine("2 AbstractClass()");

}

public abstract void AbstractMethod();

}

class ConcreteClass : AbstractClass

{

string s = "FIRST";

// Конструктор (отрабатывает вторым).

public ConcreteClass()

{

Console.WriteLine("3 ConcreteClass()");

s = "SECOND";

}

public override void AbstractMethod()

{

Console.WriteLine("Реализация метода AbstractMethod() в ConcreteClass {0}", s);

}

}

class Program

{

static void Main(string[] args)

{

AbstractClass instance = new ConcreteClass();

Console.WriteLine(new string('-', 55));

instance.AbstractMethod();

Console.ReadKey();

}

}

Возможности абстрактных методов:

* Абстрактный метод является неявным виртуальным методом.
* Создание абстрактных методов допускается только в абстрактных классах.
* Тело абстрактного метода отсутствует; создание метода просто заканчивается двоеточием, а после сигнатуры ставить фигурные скобки ({}) не нужно.
* Реализация предоставляется методом переопределения override, который является членом неабстрактного класса.

**Интерфейс** – семантическая и синтаксическая конструкция в коде программы, используемая для специфицирования услуг, предоставляемых классом или компонентом.

Интерфейс – стереотип, являющийся аналогом чистого абстрактного класса, в котором запрещена любая реализация.

Интерфейс – набор сигнатур методов. Сигнатура метода – имя метода и параметры, которые принимает метод.

Для имени интерфейса следует применять в качестве префикса букву "I". Это подсказывает, что данный тип является интерфейсом.

interface IMyInterface

{

}

Правила использовани интерфейсов:

* Невозможно создать экземпляр интерфейса.
* Интерфейсы и члены интерфейсов являются абстрактными. Интерфейсы не имеют реализации в C#.
* Интерфейс может содержать только абстрактные члены (методы, свойства, события и индексаторы).
* Члены интерфейсов автоматически являются открытыми, абстрактными, и они не могут иметь модификаторов доступа.
* Интерфейсы не могут содержать константы, поля, операторы, конструкторы экземпляров, деструкторы или вложенные типы (интерфейсы в том числе).
* Класс или структура, которые реализуют интерфейс, должны реализовать члены этого интерфейса, указанные при его создании.

interface IInterface

{

void Method();

}

class MyClass : IInterface

{

public void Method()

{

Console.WriteLine("Метод - реализация Интерфейса.");

}

}

class Program

{

static void Main(string[] args)

{

MyClass my = new MyClass();

my.Method();

Console.ReadKey();

}

}

interface Interface1

{

void Method1();

}

interface Interface2

{

void Method2();

}

class DerivedClass: Interface1, Interface2

{

public void Method1()

{

Console.WriteLine("Реализация метода Method1() из Interface1");

}

public void Method2()

{

Console.WriteLine("Реализация метода Method2() из Interface2");

}

}

class Program

{

static void Main(string[] args)

{

Interface1 instance1 = new DerivedClass();

Interface2 instance2 = new DerivedClass();

instance1.Method1();

instance2.Method2();

Console.ReadKey();

}

}

interface Interface1

{

void Method();

}

interface Interface2

{

void Method();

}

class DerivedClass : Interface1, Interface2

{

// На 10 строке реализуем метод с именем Method из базового интерфейса Interface1

// При реализации метода используем технику явного указания имени интерфейса

// в имени метода, которому принадлежит данный метод.

// По умолчанию одноименные методы являются private,

// но явно указывать модификаторы доступа недопустимо.

void Interface1.Method()

{

Console.WriteLine("Реализация метода Method() из Interface1");

}

void Interface2.Method()

{

Console.WriteLine("Реализация метода Method() из Interface2");

}

}

class Program

{

static void Main(string[] args)

{

DerivedClass instance = new DerivedClass();

// instance. -- // На экземпляре не видим методов интерфейсов.

// Приведем экземпляр класса DerivedClass - instance - к базовому интерфейсному типу Interface1

Interface1 instance1 = instance as Interface1;

instance1.Method();

Interface2 instance2 = instance as Interface2;

instance2.Method();

Console.ReadKey();

}

}

}

interface Interface1

{

void Method1();

}

interface Interface2

{

void Method2();

}

class BaseClass

{

public void Method()

{

Console.WriteLine("Метод класса - BaseClass.");

}

}

// В C# допустимо множественное наследование только от интерфейсов.

// Множественное наследование реализации (т.е. от двух и более классов или структур) недопустимо.

// Допустимо множественное наследование от одного класса и многих интерфейсов.

class DerivedClass : BaseClass, Interface1, Interface2

{

public void Method1()

{

Console.WriteLine("Реализация метода Method1() из Interface1");

}

public void Method2()

{

Console.WriteLine("Реализация метода Method2() из Interface2");

}

}

class Program

{

static void Main(string[] args)

{

DerivedClass instance = new DerivedClass();

instance.Method();

instance.Method1();

instance.Method2();

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

BaseClass instance0 = instance as BaseClass;

instance0.Method();

Interface1 instance1 = instance as Interface1;

instance1.Method1();

Interface2 instance2 = instance as Interface2;

instance2.Method2();

Console.ReadKey();

}

}

// Наследование интерфейса от интерфейса.

interface IInterface1

{

void Method1();

}

interface IInterface2 : IInterface1

{

void Method2();

}

class ConcreteClass : IInterface2

{

public void Method1()

{

Console.WriteLine("Method1 - реализация интерфейса IInterface1");

}

public void Method2()

{

Console.WriteLine("Method2 - реализация интерфейса IInterface2");

}

}

class Program

{

static void Main(string[] args)

{

ConcreteClass instance = new ConcreteClass();

instance.Method1();

instance.Method2();

IInterface1 instance1 = instance as IInterface1;

instance1.Method1();

IInterface2 instance2 = instance as IInterface2;

instance2.Method1();

instance2.Method2();

Console.ReadKey();

}

}

// Наследование интерфейса от интерфейса, у которых совпадают имена членов.

interface IInterface1

{

void Method();

}

interface IInterface2 : IInterface1

{

// Без new ошибки не будет, но будет предупреждение компилятора.

new void Method();

}

class ConcreteClass : IInterface2

{

void IInterface1.Method()

{

Console.WriteLine("Method - реализация интерфейса IInterface1");

}

void IInterface2.Method()

{

Console.WriteLine("Method - реализация интерфейса IInterface2");

}

}

class Program

{

static void Main(string[] args)

{

ConcreteClass instance = new ConcreteClass();

// instance.Method();

IInterface1 instance1 = instance as IInterface1;

instance1.Method();

IInterface2 instance2 = instance as IInterface2;

instance2.Method();

Console.ReadKey();

}

}

// Наследование от интерфейсов, у которых совпадают имена членов.

// Объединение реализации одноименных абстрактных членов.

interface IInterface1

{

void Method();

}

interface IInterface2

{

void Method();

}

class ConcreteClass: IInterface1, IInterface2

{

public void Method()

{

Console.WriteLine("Method - реализация интерфейса IInterface (1-2)");

}

}

class Program

{

static void Main(string[] args)

{

ConcreteClass instance = new ConcreteClass();

instance.Method();

IInterface1 instance1 = instance as IInterface1;

instance1.Method();

IInterface2 instance2 = instance as IInterface2;

instance2.Method();

Console.ReadKey();

}

}

// Наследование абстрактных классов от интерфейсов.

interface IInterface

{

void Method();

}

abstract class AbstractClass : IInterface

{

// Реализация абстрактного метода из интерфейса в абстрактном классе необязательна.

public void Method()

{

Console.WriteLine("Метод - реализация интерфейса в абстрактном классе.");

}

}

class ConcreteClass : AbstractClass

{

}

class Program

{

static void Main(string[] args)

{

ConcreteClass instance = new ConcreteClass();

instance.Method();

Console.ReadKey();

}

}

// Наследование абстрактных классов от интерфейсов.

interface IInterface

{

void Method();

}

abstract class AbstractClass : IInterface

{

// Замещение абстрактного метода из интерфейса в абстрактном классе.

public abstract void Method();

}

class ConcreteClass : AbstractClass

{

// Реализация абстрактного метода из абстраткного класса в конкретном классе обязательна.

public override void Method()

{

Console.WriteLine("Метод - реализация интерфейса в абстрактном классе");

}

}

class Program

{

static void Main(string[] args)

{

ConcreteClass instance = new ConcreteClass();

instance.Method();

Console.ReadKey();

}

}

// Наследование от класса и интерфейса, у которых совпадают сигнатуры членов.

interface IInterface

{

void Method();

}

class BaseClass

{

public void Method()

{

Console.WriteLine("BaseClass.Method()");

}

}

class DerivedClass : BaseClass, IInterface

{

// Реализация интерфейса не обязательна, т.к.

// сигнатуры методов в классе и интерфейсе совпадают.

}

class Program

{

static void Main(string[] args)

{

DerivedClass instance = new DerivedClass();

instance.Method();

IInterface instance1 = instance as IInterface;

instance1.Method();

Console.ReadKey();

}

}

Преимущества использования интерфейсов:

* Класс или структура может реализовать несколько интерфейсов (допустимо множественное наследование от интерфейсов).
* Если класс или структура реализует интерфейс, она получает только имена и сигнатуры метода.
* Интерфейсы определяют поведение экземпляров производных классов.
* Базовый класс может обладать ненужным функционалом, полученным от других его базовых классов, чего можно избежать, применяя интерфейсы.

// Пример использования интерфейса с однотипными и разнотипными объектами

interface IGoEater

{

void Go();

void Eat();

}

interface ISecurity : IGoEater

{

void Guard();

}

class Dog : IGoEater, ISecurity

{

public void Go()

{

Console.WriteLine("Dog is going");

}

public void Eat()

{

Console.WriteLine("Dog is eating");

}

public void Guard()

{

Console.WriteLine("Dog is guarding");

}

}

class Cat : IGoEater

{

public void Go()

{

Console.WriteLine("Cat is going");

}

public void Eat()

{

Console.WriteLine("Cat is eating");

}

}

class Program

{

static void Main(string[] args)

{

ISecurity dog = new Dog();

IGoEater cat = new Cat();

dog.Go();

dog.Eat();

dog.Guard();

Console.WriteLine(new string('-', 30));

cat.Go();

cat.Eat();

Console.ReadKey();

}

}

## Массивы и индексаторы

Типы массива являются ссылочными типами, производными от абстрактного базового класса Array.

Object Browser – ctrl + w + j

// Массивы (двумерный массив).

class Program

{

static void Main(string[] args)

{

Random random = new Random();

int[,] array = new int[3, 3];

// Заполнение массива случайными значениями.

for(int i = 0; i < 3; i++)

{

for(int j = 0; j < 3; j++)

{

array[i, j] = random.Next(0, 10);

}

}

for (int i = 0; i < 3; i++)

{

for (int j = 0; j < 3; j++)

{

Console.Write("{0} ", array[i, j]);

}

Console.Write("\n");

}

Console.ReadKey();

}

}

class Program

{

// Пример трехмерного массива.

static void Main(string[] args)

{

int[,,] threeDim =

{

{ {1, 2}, {3, 4} },

{ {5, 6}, {7, 8} }

};

// Получаем количество подмассивов в Массиве - GetLength(0). - 2 подмассива.

for(int i = 0; i < threeDim.GetLength(0); ++i)

{

// Получаем количество элементов в Подмассиве - GetLength(1). - 4 элемента в каждом подмассиве.

for(int j = 0; j < threeDim.GetLength(1); ++j)

{

for(int k = 0; k < threeDim.GetLength(2); k++)

{

Console.Write(threeDim[i, j, k] + ", ");

}

Console.WriteLine();

}

}

Console.WriteLine(threeDim.Length);

Console.ReadKey();

}

}

// Абстрактный класс Array.

// Все массивы являются производными от класса Array.

class Program

{

static void Main(string[] args)

{

int[] vector = { 1, 2, 3 };

Array array = vector as Array;

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

{

Console.WriteLine(vector[i]);

}

Console.ReadKey();

}

}

// Неявно типизированные массивы.

class Program

{

static void Main(string[] args)

{

// Массив Int32.

var array1 = new[] { 1, 2, 3, 4, 5 };

Console.WriteLine(array1.GetType());

// Массив Doubles.

var array2 = new[] {3.1415, 1, 6 };

Console.WriteLine(array2.GetType());

// Не компилируется. (Несовместимые типы)

// var array3 = new[] {1, "string" };

Console.ReadKey();

}

}

Результат:

System.Int32[]

System.Double[]

Массивы в C# ковариантные, но не контрвариантные.

Ковариантность массива – это неявный UpCast всех элементов массива.

Контрвариантность массива – это неявный DownCast всех элементов массива.

Массивы элементов ссылочных типов ковариантны, но не контрвариантны.

Массивы элементов структурных типов не ковариантны и не контрвариантны.

// Ковариантность массивов в C#.

public interface IAnimal

{

void Voice();

}

public class Dog : IAnimal

{

public void Voice()

{

Console.WriteLine("Voice");

}

public void Jump()

{

Console.WriteLine("Jump");

}

}

class Program

{

static void Main(string[] args)

{

Dog[] dogs = { new Dog(), new Dog(), new Dog() };

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

{

dogs[i].Voice();

dogs[i].Jump();

}

Console.WriteLine(new string('-', 10));

IAnimal[] animal = dogs; // Ковариантность.

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

{

animal[i].Voice();

// animal[i].Jump();

}

Console.WriteLine(new string ('-', 10));

dogs = (Dog[])animal; // Не является Контрвариантностью.

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

{

dogs[i].Voice();

dogs[i].Jump();

}

Console.ReadKey();

}

}

// Параметр с ключевым словом params должен быть только один и стоять последним в списке параметров метода.

class Program

{

static void ShowArray(string s, params int[] array)

{

Console.Write(s);

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

{

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

}

}

static void Main(string[] args)

{

ShowArray("Numbers: "0, 1, 2, 3, 4, 5, 6, 7, 8, 9);

Console.ReadKey();

}

}

Индексаторы позволяют индексировать экземпляры класса или структуры так же, как и массивы. Индексаторы напомниают свойства, но их методы доступа принимают параметры.

public int this[int index]

{

get { return array[index]; }

set { array[index] = value; }

}

Метод **set** автоматически срабатывает тогда, когда свойству пытаются присвоить значение. Это значение представлено ключевым словом **value**.

Метод **get** автоматически срабатывает тогда, когда мы пытаемся получить значение.

class MyClass

{

int[] array = new int[5];

// Индексатор.

public int this[int index]

{

get // Аксессор.

{

return array[index];

}

set // Мутатор.

{

array[index] = value;

}

}

}

class Program

{

static void Main(string[] args)

{

MyClass my = new MyClass();

my[0] = 1;

my[1] = 2;

my[2] = 3;

my[3] = 4;

my[4] = 5;

Console.WriteLine(my[0]);

Console.WriteLine(my[1]);

Console.WriteLine(my[2]);

Console.WriteLine(my[3]);

Console.WriteLine(my[4]);

Console.ReadKey();

}

}

class MyClass

{

string[] array = new string[5];

// Индексатор.

public string this[int index]

{

get // Аксессор.

{

if (index >= 0 && index < array.Length)

{

return array[index];

}

else return "Попытка обращения за пределы массива";

}

set // Мутатор.

{

if (index >= 0 && index < array.Length)

{

array[index] = value;

}

else Console.WriteLine("Попытка записи за пределами массива.");

}

}

}

class Program

{

static void Main(string[] args)

{

MyClass my = new MyClass();

my[0] = "string 1";

my[1] = "string 2";

my[2] = "string 3";

my[3] = "string 4";

my[4] = "string 5";

my[5] = "string 6";

Console.WriteLine(my[0]);

Console.WriteLine(my[1]);

Console.WriteLine(my[2]);

Console.WriteLine(my[3]);

Console.WriteLine(my[4]);

Console.WriteLine(my[5]);

}

}

// Работа с индексаторами в циклах.

class MyClass

{

string[] array = new string[5];

// Индексатор.

public string this[int index]

{

get // Аксессор.

{

if (index >= 0 && index < array.Length)

{

return array[index];

}

else return "Попытка обращения за пределы массива";

}

set // Мутатор.

{

if (index >= 0 && index < array.Length)

{

array[index] = value;

}

else Console.WriteLine("Попытка записи за пределами массива.");

}

}

}

class Program

{

static void Main(string[] args)

{

MyClass my = new MyClass();

for(int i = 0; i < 6; i++)

{

my[i] = string.Format("string {0}", i);

}

for(int i = 0; i < 6; i++)

{

Console.WriteLine(my[i]);

}

Console.ReadKey();

}

}

// Индексаторы и двумерный массив.

class MyClass

{

int[,] array = new int[3, 3];

public int this[int index1, int index2]

{

get

{

return array[index1, index2];

}

set

{

array[index1,index2] = value;

}

}

}

class Program

{

static void Main(string[] args)

{

MyClass my = new MyClass();

my[1, 1] = 2;

Console.WriteLine(my[1, 1]);

Console.WriteLine(my[0, 0]);

Console.ReadKey();

}

}

Индексаторы друг от друга могут отличаться только типом или количеством индексов.

class Dictionary

{

string[] key = new string[5];

string[] value = new string[5];

public Dictionary()

{

key[0] = "книга"; value[0] = "book";

key[1] = "ручка"; value[1] = "pen";

key[2] = "солнце"; value[2] = "sun";

key[3] = "яблоко"; value[3] = "apple";

key[4] = "стол"; value[4] = "table";

}

public string this[string index]

{

get

{

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

{

if(key[i] == index)

{

return key[i] + "-" + value[i];

}

}

return string.Format("{0} - нет перевода этого слова.", index);

}

}

public string this[int index]

{

get

{

if (index >= 0 && index < key.Length)

{

return key[index] + "-" + value[index];

}

else return "Попытка обращения за пределы массива.";

}

}

}

class Program

{

static void Main(string[] args)

{

Dictionary dictionary = new Dictionary();

Console.WriteLine(dictionary["книга"]);

Console.WriteLine(dictionary["дом"]);

Console.WriteLine(dictionary["ручка"]);

Console.WriteLine(dictionary["стол"]);

Console.WriteLine(dictionary["карандаш"]);

Console.WriteLine(dictionary["яблоко"]);

Console.WriteLine(dictionary["солнце"]);

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

for(int i = 0; i < 6; i++)

{

Console.WriteLine(dictionary[i]);

}

Console.ReadKey();

}

}

}

class BaseClass

{

private string[] baseArray = null;

// Конструктор.

public BaseClass()

{

baseArray = new string[3];

baseArray[0] = "Ноль";

baseArray[1] = "Один";

baseArray[2] = "Два";

}

// Виртуальный индексатор.

public virtual string this[int index]

{

get { return baseArray[index]; }

}

}

class DerivedClass : BaseClass

{

private string[] derivedArray = null;

// Конструктор.

public DerivedClass()

{

derivedArray = new string[3];

derivedArray[0] = "Zero!";

derivedArray[1] = "One!";

derivedArray[2] = "Two!";

}

// Переопределенный индексатор.

public override string this[int index]

{

get

{

{ return base[index] + " - " + derivedArray[index]; }

}

}

}

class Program

{

static void Main(string[] args)

{

DerivedClass instance = new DerivedClass();

BaseClass instance1 = instance;

for(int i = 0; i < 3; i++)

{

Console.WriteLine(instance1[i]);

}

Console.ReadKey();

}

}

* Индексаторы не обязаны использовать в качестве индекса целочисленное значение, конкретный механизм поиска определяет разработчик.
* Индексаторы можно перегружать.
* Индексаторы могут иметь более одного индекса, например, при доступе к двухмерному массиву.

## Статические и вложенные классы

**Статическая переменная** – это общая переменная для всех экземпляров класса, которая хранится в объекте.

Объекты содержат в себе поля и методы.

class NotStaticClass

{

private int Id;

public static int field;

// Конструктор.

public NotStaticClass(int Id)

{

this.Id = Id;

}

public void Method()

{

Console.WriteLine("Instance{0}.field = {1}", Id, field);

}

}

class Program

{

static void Main(string[] args)

{

NotStaticClass instance1 = new NotStaticClass(1);

NotStaticClass instance2 = new NotStaticClass(2);

instance1.Method();

instance2.Method();

// На классе-объекте NotStaticClass обращаемся к статическому полю - field

NotStaticClass.field = 1;

instance1.Method();

instance2.Method();

Console.ReadKey();

}

}

// В статических методах нельзя обращаться к нестатическим полям.

class NotStaticClass

{

private int id;

// Конструктор.

public NotStaticClass(int id)

{

this.id = id;

}

public static void Method()

{

// Console.WriteLine("Instance.id = {0}", id);

Console.WriteLine("В статических методах нельзя обращаться к нестатическим полям.");

}

}

// Константы.

class NotStaticClass

{

// Константны не могут быть статическими.

// public static const float e = 2.71828182845904523536f;

// Поле не может быть объявлено как static const, поле const по своему поведению уже является статическим.

// Поле const относится к типу, а не к экземплярам типа.

// Поэтому к полям const можно обращаться с использованием той же нотации ИмяКласса.ИмяЧлена,

// что и в используемой для статических полей.

public const float e = 2.71828182845904523536f;

}

class Program

{

static void Main(string[] args)

{

Console.WriteLine("e = {0}", NotStaticClass.e);

Console.ReadKey();

}

}

class NotStaticClass

{

// Статическое поле.

static int field;

// Статическое свойство.

public static int Property

{

get { return field; }

set { field = value; }

}

}

class Program

{

static void Main(string[] args)

{

NotStaticClass.Property = 33;

Console.WriteLine(NotStaticClass.Property);

Console.ReadKey();

}

}

// Статический конструктор.

// Статические конструкторы обладают следующими свойствами:

// 1. Статический конструктор не имеет модификаторов доступа и не принимает параметров.

// 2. Статический конструктор вызывается автоматически для инициализации класса перед созданием первого экземпляра

// или ссылкой на какие-либо статические члены.

// 3. Статический конструктор нельзя вызывать напрямую.

// 4. Пользователь не управляет тем, когда статический конструктор выполняется в программе.

// 5. Типичным использованием статических конструкторов является случай, когда класс использует файл журнала

// и конструктор применяется для добавления записей в этот файл.

class NotStaticClass

{

// Статические поля readonly должны быть инициализированы в конструкторе.

static readonly long readonlyField = 2;

// Статическое свойство только для чтения.

public static long ReadonlyField

{

get { return NotStaticClass.readonlyField; }

}

// Статический конструктор.

static NotStaticClass()

{

readonlyField = 1;

}

}

class Program

{

static void Main(string[] args)

{

Console.WriteLine(NotStaticClass.ReadonlyField);

Console.ReadKey();

}

}

class NotStaticClass

{

// Если класс содержит статические поля, должен быть предоставлен статический конструктор,

// который инициализирует эти поля при загрузке класса.

private static int field;

public static int Property

{

get { return field; }

set { field = value; }

}

// Статический конструктор.

// Единственное назначение статических конструкторов - присваивать исходные значения статическим переменным.

static NotStaticClass()

{

Console.WriteLine("Статический конструктор - NotStaticClass()");

field = 1;

}

// Статические методы могут быть перегружены.

public static void Method()

{

Console.WriteLine("Статический метод нестатического NotStaticClass");

}

// Статические методы могут быть перегружены.

public static void Method(int s)

{

Console.WriteLine("Перегруженный статический метод нестатического NotStaticClass {0}", s);

}

// Нестатический метод.

public void NotStaticMethod()

{

Console.WriteLine(field);

}

}

class Program

{

static void Main(string[] args)

{

NotStaticClass instance = new NotStaticClass();

instance.NotStaticMethod();

NotStaticClass.Property = 2;

Console.WriteLine(NotStaticClass.Property);

NotStaticClass.Method();

NotStaticClass.Method(3);

Console.ReadKey();

}

}

class NotStaticClass

{

// Конструктор экземпляра, вызывается только при создании экземпляра класса.

public NotStaticClass()

{

Console.WriteLine("Consturctor.");

}

// Статический конструктор вызывается в любом случае:

// как при обращении к статическому члену, так и при создании экземпляра класса.

static NotStaticClass()

{

Console.WriteLine("Static Constructor.");

}

// Статический метод.

public static void StaticMethod()

{

Console.WriteLine("StaticMethod");

}

public void NotStaticMethod()

{

Console.WriteLine("NotStaticMethod");

}

}

class Program

{

static void Main(string[] args)

{

// 1 Вариант. (Вызывается только статический констуктор.)

NotStaticClass.StaticMethod();

// 2 Вариант. (Вызываются оба конструктора.)

// new NotStaticClass().NotStaticMethod();

Console.ReadKey();

}

}

// Статические члены в абстрактных классах.

abstract class AbstractClass

{

// Статический фабричный метод.

public static AbstractClass CreateObject()

{

return new ConcreteClass();

}

public abstract void Method();

}

class ConcreteClass : AbstractClass

{

public override void Method()

{

Console.WriteLine("Hello world!");

}

}

class Program

{

static void Main(string[] args)

{

AbstractClass instance = AbstractClass.CreateObject();

instance.Method();

Console.ReadKey();

}

}

// Статические члены не могут быть виртуальными, переопределенными и абстрактными.

// Статические члены поддерживают замещение.

abstract class BaseClass

{

// public static virtual void StaticMethod()

// {

// Console.Writeline("BaseClass.StaticMethod");

// }

public static void StaticMethod()

{

Console.WriteLine("BaseClass.StaticMethod");

}

}

class DerivedClass : BaseClass

{

// public static override void StaticMethod()

// {

// Console.WriteLine("DerivedClass.StaticMethod");

// }

public static new void StaticMethod()

{

Console.WriteLine("DerivedClass.StaticMethod");

}

}

class Program

{

static void Main(string[] args)

{

BaseClass.StaticMethod();

DerivedClass.StaticMethod();

Console.ReadKey();

}

}

**Статический класс** – это контейнер, который содержит в себе только статические члены.

Доступ к членам статического класса осуществляется на **Классе-Объекте**

static class StaticClass

{

public static int item;

}

static void Main()

{

StaticClass.item = 10;

}

Если класс содержит статические поля, должен быть предоставлен статический конструктор, который инициализирует эти поля при загрузке класса.

**CTRL + W + J – открыть Object Browser.**

// Статический класс в C# выражает идею паттерна проектирования Singleton.

// Правила:

// 1. Экземпляр статического класса нельзя создать.

// 2. Static class всегда наследуется от Object (попытка наследоваться от чего-либо другого приводит к ошибке компиляции).

// 3. Static class не реализует интерфейсы. Попытка наследования от интерфейса приводит к ошибке уровня компиляции.

// 4. Содержит только статические члены (наличие в нем нестатического члена приведет к ошибке компиляции).

// 5. Статический класс не может содержать конструкторы экземпляра.

// 6. Статический класс закрыт для наследования от него. Попытка наследования от статического класса приводит к ошибке уровня компиляции.

class StaticClass // : object // Наследование только от Object (явно или неявно)

{

// Конструктор экземпляра недопустим.

// public StaticClass()

// {

// Console.WriteLine("Constructor.");

// }

// Статический конструктор.

static StaticClass()

{

Console.WriteLine("Static Constructor");

}

// Статический метод.

public static void StaticMethod()

{

Console.WriteLine("StaticMethod");

}

// Нестатический метод недопустим!

// public void NotStaticMethod()

// {

// Console.WriteLine("NotStaticMethod");

// }

}

class Program

{

static void Main(string[] args)

{

StaticClass.StaticMethod();

Console.ReadKey();

}

}

// Singleton

class Singleton

{

private static Singleton instance = null;

// Конструктор - "protected"

protected Singleton()

{

}

// Фабричный метод.

public static Singleton Instance()

{

// Если: объект еще не создан (1)

if(instance == null)

{

// То: создаем новый экземпляр (2)

instance = new Singleton();

}

// Иначе: возвращаем ссылку на существующий объект (3)

return instance;

}

}

class Program

{

static void Main(string[] args)

{

// Конструктор "protected" - невозможно использовать - new Singleton()

Singleton instance1 = Singleton.Instance();

Singleton instance2 = Singleton.Instance();

if (instance1 == instance2)

{

Console.WriteLine("Ссылки указывают на один экземпляр объекта.");

}

Console.ReadKey();

}

}

**Методы расширения** позволяют "добавлять" методы в существующие типы без создания нового производного типа, перекомпиляции или иного изменения исходного типа.

Расширяющие методы могут быть только статическими и создаваться только в статических классах.

static class MyClass

{

public static void Method(this string value)

{

Console.WriteLine(value);

}

}

class Program

{

static void Main(string[] args)

{

string text = "Тестовая строка";

MyClass.Method(text);

text.Method();

}

}

Аргумент расширения всегда должен быть только один и стоять первым в списке аргументов.

// Расширяющие методы (Extension methods).

// Расширяющие методы могут быть только статическими и создаваться только в статических классах.

static class ExtensionClass

{

// this - сообщает компилятору, что данный метод является расширяющим (Extension)!

public static void ExtensionMethod(this string value)

{

Console.WriteLine(value);

}

}

class Program

{

static void Main(string[] args)

{

string text = "Тестовая строка";

// Вызов метода, как статического.

ExtensionClass.ExtensionMethod(text);

// Вызов метода, как расширяющего.

text.ExtensionMethod();

Console.ReadKey();

}

}

// Расширяющие методы. (Extension methods)

// Аргумент расширения всегда должен быть только один и стоять первым в списке аргументов.

static class ExtensionClass

{

public static void ExtensionMethod(this string value1, string value2)

{

Console.WriteLine(value1 + value2);

}

}

class Program

{

static void Main(string[] args)

{

string text = "Hello";

text.ExtensionMethod(" world!");

Console.ReadKey();

}

}

// Расширяющие методы. (Extension methods)

// Расширяющие методы не могут иметь опциональных параметров, но могут быть перегружены.

static class ExtensionClass

{

public static void ExtensionMethod(this string value)

{

Console.WriteLine(value);

}

public static void ExtensionMethod(this string value1, string value2)

{

Console.WriteLine(value1 + value2);

}

public static void ExtensionMethod(this int value)

{

Console.WriteLine(value);

}

}

class Program

{

static void Main(string[] args)

{

string text = "Hello ";

text.ExtensionMethod("world!");

"Hello ".ExtensionMethod("world!");

2.ExtensionMethod();

Console.ReadKey();

}

}

// Расширяющие методы. (Extension methods)

// Рекурсия (простая).

static class ExtensionClass

{

public static void ExtensionMethod(this string value, int counter)

{

counter--;

Console.WriteLine(value+counter);

if(counter != 0)

{

value.ExtensionMethod(counter);

}

Console.WriteLine(value+counter);

}

}

class Program

{

static void Main(string[] args)

{

string text = "Hello ";

text.ExtensionMethod(3);

Console.ReadKey();

}

}

// Расширяющие методы. (Extension methods)

// Использование ref и out аргументов. (Аргумент расширения не может быть ни ref ни out)

static class ExtensionClass

{

public static void Add(this int summand1, ref int summand2, out int sum)

{

sum = summand1 + summand2;

Console.WriteLine("{0} + {1} = {2}", summand1, summand2, sum);

}

}

class Program

{

static void Main(string[] args)

{

int summand1 = 1, summand2 = 2, sum = 0;

summand1.Add(ref summand1, out sum);

Console.ReadKey();

}

}

}

**Вложенные классы** (Nested Classes)

Тип, определенный внутри класса или структуры, называется вложенным типом.

Экземпляр внутреннего класса не может существовать без привязки к включающему его классу верхнего уровня.

class Container

{

class Nested

{

int field;

}

}

Статические классы могут в себе содержать нестатические **Nested** классы.

// Nested classes.

class MyClass

{

public class Nested

{

public void Method()

{

Console.WriteLine("Метод из Nested класса");

}

}

}

class Program

{

static void Main(string[] args)

{

MyClass.Nested instance = new MyClass.Nested();

instance.Method();

Console.ReadKey();

}

}

}

// Nested classes.

class MyClass

{

private static int field = 0;

public class Nested

{

public void Method(int a)

{

field = a;

Console.WriteLine(field);

}

}

}

class Program

{

static void Main(string[] args)

{

MyClass.Nested instance = new MyClass.Nested();

instance.Method(1);

Console.ReadKey();

}

}

}

// Nested classes.

class MyClass

{

private int field = 0;

public class Nested

{

MyClass instance = new MyClass();

public void Method(int a)

{

instance.field = a;

Console.WriteLine(instance.field);

}

}

}

class Program

{

static void Main(string[] args)

{

MyClass.Nested instance = new MyClass.Nested();

instance.Method(1);

Console.ReadKey();

}

}

class BaseClass

{

public void MethodFromBase()

{

Console.WriteLine("Метод базового класса");

}

}

public class MyClass // Наследование от BaseClass не распространяется.

{

public class Nested : BaseClass

{

public void MethodFromNested()

{

Console.WriteLine("Метод Nested класса.");

}

}

}

class Program

{

static void Main(string[] args)

{

MyClass.Nested instance = new MyClass.Nested();

instance.MethodFromBase();

instance.MethodFromNested();

Console.ReadKey();

}

}

class BaseClass

{

public void MethodFromBase()

{

Console.WriteLine("Метод Базового класса.");

}

}

// Nested classes.

public class MyClass : BaseClass

{

public class Nested // Наследование от BaseClass не распространяется.

{

public void MethodFromNested()

{

Console.WriteLine("Метод Nested класса.");

}

}

}

class Program

{

static void Main(string[] args)

{

MyClass instance1 = new MyClass();

instance1.MethodFromBase();

MyClass.Nested instance2 = new MyClass.Nested();

instance2.MethodFromNested();

Console.ReadKey();

}

}

// Nested classes.

// Статические классы могут содержать в себе нестатические Nested классы.

public static class MyClass

{

static MyClass()

{

Console.WriteLine("Статический конструктор MyClass");

}

public static void StaticMethod()

{

Console.WriteLine("Статический метод класса MyClass");

}

public class Nested

{

public void MethodFromNested()

{

Console.WriteLine("Метод Nested класса.");

}

}

}

class Program

{

static void Main(string[] args)

{

MyClass.StaticMethod();

MyClass.Nested instance = new MyClass.Nested();

instance.MethodFromNested();

Console.ReadKey();

}

}

// Nested classes.

// Нестатические классы могут содержать в себе статические Nested классы.

public class MyClass

{

public static class Nested

{

public static void StaticMethodFromNested()

{

Console.WriteLine("Статический метод Nested класса.");

}

}

}

class Program

{

static void Main(string[] args)

{

MyClass.Nested.StaticMethodFromNested();

Console.ReadKey();

}

}

// Техника делегирования.

class A

{

public void DoSomething()

{

Console.WriteLine("Action");

}

}

class B

{

public void DoSomething()

{

A a = new A();

a.DoSomething();

}

}

class Program

{

static void Main(string[] args)

{

B b = new B();

b.DoSomething();

Console.ReadKey();

}

}

## Структуры и их разновидности

**Структура (классическая)** – конструкция языка, позволяющая содержать в себе набор полей различных типов.

struct Struct

{

int field;

static int second;

}

В структурах нельзя инициализировать поля непосредственно в месте создания. Инициализация статических полей необязательна.

Структуры хранятся в стеке и обычно используются для инкапсуляции небольших групп связанных переменных.

Структуры могут содержать конструкторы, константы, поля, методы, свойства, индексаторы, операторы, события и вложенные типы, однако, если требуется несколько таких членов, рекомендуется использовать тип **class**.

**Стек** – это специальная область памяти, в которой хранятся структурные типы.

// В структурах нельзя инициализировать поля непосредственно в месте создания.

struct MyStruct

{

public int field;

}

class Program

{

static void Main(string[] args)

{

// Создание экземпляра структурного типа, без вызова конструктора.

MyStruct instance;

instance.field = 1; // Закомментировать.

// Попытка вывода значения неинициализированного поля приведет к ошибке.

Console.WriteLine(instance.field);

Console.ReadKey();

}

}

// Если в структуре имеются члены, которые обращаются к полю и нет пользовательского конструктора,

// то требуется при создании экземпляра вызывать конструктор по умолчанию (иначе будет ошибка).

struct MyStruct

{

private int field;

public int Field

{

get { return field; }

set { field = value; }

}

public void Show()

{

Console.WriteLine(field);

}

}

class Program

{

static void Main(string[] args)

{

// Создание экземпляра структурного типа с вызовом конструктора по умолчанию.

MyStruct instance = new MyStruct();

instance.Field = 1;

Console.WriteLine(instance.Field);

Console.ReadKey();

}

}

struct MyStruct

{

public int field;

// Конструкторы по умолчанию нельзя задавать явно.

// public MyStruct()

// {

// }

// Если в структуре имеется пользовательский конструктор, то требуется в нем инициализировать все поля.

public MyStruct(int value)

{

Console.WriteLine("Constructor");

this.field = value;

}

}

class Program

{

static void Main(string[] args)

{

// Создание экземпляра структурного типа с вызовом конструктора по умолчанию.

MyStruct instance = new MyStruct();

Console.WriteLine(instance.field);

Console.ReadKey();

}

}

// Структуры.

struct MyStruct

{

public int field;

// Конструкторы по умолчанию нельзя задавать явно.

// public MyStruct()

// {

// }

// Если в структуре имеется пользовательский конструктор, то требуется в нем инициализировать все поля.

public MyStruct(int value)

{

Console.WriteLine("Constructor");

this.field = value;

}

}

class Program

{

static void Main(string[] args)

{

// Создание экземпляра структурного типа с вызовом пользовательского конструктора.

MyStruct instance = new MyStruct(1);

Console.WriteLine(instance.field);

Console.ReadKey();

}

}

// Структуры могут содержать статические члены.

// Статические структуры недопустимы.

struct MyStruct

{

public static int Field

{

get;

set;

}

public static void Show()

{

Console.WriteLine(Field);

}

}

class Program

{

static void Main(string[] args)

{

// Инициализация статических полей необязательна.

// MyStruct.Field = 1;

MyStruct.Show();

Console.ReadKey();

}

}

// Структуры. Статический конструктор.

struct MyStruct

{

public int field;

// Статический конструктор всегда отрабатывает первым.

static MyStruct()

{

Console.WriteLine("Static Constructor");

}

// Если в структуре имеется пользовательский конструктор, то в нем требуется инициализировать все поля.

public MyStruct(int value)

{

Console.WriteLine("Constructor");

this.field = value;

}

}

class Program

{

static void Main(string[] args)

{

// Создание экземпляра структурного типа с вызовом пользовательского конструктора.

MyStruct instance = new MyStruct { field = 0 };

Console.WriteLine(instance.field);

Console.ReadKey();

}

}

В структурах статический конструктор по умолчанию работает только в связке с пользовательским.

// Структуры. Конструкторы.

struct MyStruct

{

public int field;

// Пользовательский конструктор с параметрами.

public MyStruct(int value)

{

this.field = value;

Console.WriteLine(value);

}

}

class Program

{

static void Main(string[] args)

{

// Создание экземпляра структурного типа без вызова конструктора.

MyStruct instance;

// Нельзя использовать не инициализированную переменную.

// Так как конструктор не вызывался переменная field осталась не инициализированной.

// Console.WriteLine(instance.field); // Убрать комментарий.

Console.ReadKey();

}

}

// В структурах можно создавать автоматически реализуемые свойства,

// при этом требуется использовать конструктор при построении экземпляра.

struct MyStruct

{

public int Property

{

get;

set;

}

}

class Program

{

static void Main(string[] args)

{

MyStruct instance = new MyStruct();

instance.Property = 1;

Console.WriteLine(instance.Property);

Console.ReadKey();

}

}

// Пример плохого дизайна.

struct ZipCode

{

// Поля.

int fiveDigitCode;

int plusFourExtension;

// Свойства.

public int FiveDigitCode

{

get { return fiveDigitCode; }

set { fiveDigitCode = value; }

}

public int PlusFourExtension

{

get { return plusFourExtension; }

set { plusFourExtension = value; }

}

}

class Program

{

static void Main(string[] args)

{

ZipCode zipCode = new ZipCode();

zipCode.FiveDigitCode = 12345;

zipCode.PlusFourExtension = 1234;

Console.WriteLine(zipCode.FiveDigitCode);

Console.WriteLine(zipCode.PlusFourExtension);

Console.ReadKey();

}

}

// Пример хорошего дизайна.

struct ZipCode

{

// Поля.

int fiveDigitCode;

int plusFourExtension;

// Конструкторы.

public ZipCode(int fiveDigitCode, int plusFourExtension)

{

this.fiveDigitCode = fiveDigitCode;

this.plusFourExtension = plusFourExtension;

}

public ZipCode(int fiveDigitCode) : this(fiveDigitCode, 0)

{

}

// Свойства.

public int FiveDigitCode

{

get { return fiveDigitCode; }

}

public int PlusFourExtension

{

get { return plusFourExtension; }

}

}

class Program

{

static void Main(string[] args)

{

ZipCode zipCode = new ZipCode(12345, 1234);

Console.WriteLine(zipCode.FiveDigitCode);

Console.WriteLine(zipCode.PlusFourExtension);

Console.ReadKey();

}

}

Структуры могут реализовывать интерфейсы, но они не могут наследоваться от структур или классов, поскольку уже наследуются неявно от абстрактного класса ValueType.

В C# можно наследоваться только от одного класса и множества интерфейсов. Наследование от структур запрещено.

// Наследование структур разрешено только от интерфейсов.

// Наследование структур от классов и структур запрещено.

interface IInterface

{

void Method();

}

struct MyStruct : IInterface

{

public void Method()

{

Console.WriteLine("Method");

}

}

class Program

{

static void Main(string[] args)

{

MyStruct instance;

instance.Method();

Console.ReadKey();

}

}

// Все структуры неявно наследуются от абстрактного класса ValueType

struct MyStruct // : ValueType

{

public void Method()

{

Console.WriteLine("Method");

}

}

class Program

{

static void Main(string[] args)

{

MyStruct instance = new MyStruct();

ValueType valueType = instance as ValueType;

Console.WriteLine("instance = {0}", instance.GetHashCode());

Console.WriteLine("valueType = {0}", valueType.GetHashCode());

Console.ReadKey();

}

}

// От структур нельзя наследоваться.

struct MyStruct

{

// Структуры не могут иметь protected членов.

// protected int field;

}

class MyClass //: MyStruct

{

}

class Program

{

static void Main(string[] args)

{

Console.WriteLine("От структур нельзя наследоваться");

Console.ReadKey();

}

}

Структуры могут содержать в себе вложенные структуры и классы.

struct MyStruct

{

public struct Nested

{

public void Method()

{

Console.WriteLine("Nested");

}

}

}

class Program

{

static void Main(string[] args)

{

MyStruct.Nested instance = new MyStruct.Nested();

instance.Method();

Console.ReadKey();

}

}

}

// Структуры могут содержать вложенные классы.

struct MyStruct

{

public class Nested

{

public void Method()

{

Console.WriteLine("Nested");

}

}

}

class Program

{

static void Main(string[] args)

{

MyStruct.Nested instance = new MyStruct.Nested();

instance.Method();

Console.ReadKey();

}

}

**Поведение структур**:

* Структуры размещаются в стеке.
* При копировании структур создается отдельная копия объекта, которая живет "своей жизнью".
* От структур нельзя наследоваться.
* Структуры могут наследоваться только от интерфейсов.
* При передаче структуры как параметра передается вся структура.
* В структуре нельзя создать конструктор по умолчанию.

## Структуры и их разновидности. Перечисления

Стек – структура данных с методом доступа к элементам Last In - First Out "последним пришел – первым вышел".

Упаковка-преобразование (boxing) является неявным:

static void Main(string[] args)

{

int item = 10;

object o = item;

}

Распаковка-преобразование (unboxing) является явным:

static void Main(string[] args)

{

int item = 10;

object obj = item;

int item2 = (int)obj;

}

// Упаковка (boxing) - преобразование структурного типа (типа значения) в ссылочный тип

// (object или любой другой тип интерфейса, реализуемый этим типом значения).

// Когда тип значения упаковывается средой CLR, она создает программу-оболочку значения внутри

// System.Object и сохраняет ее в управляемой куче.

// Распаковка (unboxing) - преобразование ссылочного типа в структурный тип.

// Операция распаковки-преобразования извлекает тип значения из объекта.

// Перед распаковкой среда выполнения проверяет совместимость между объектом и структурой,

// в которую будет происходить распаковка.

class Program

{

static void Main(string[] args)

{

short a = 25;

// Упаковка переменной а (boxing).

object o = a;

// Распаковка объекта (unboxing).

short b = (short)o;

// Распаковка должна производиться только в тот тип, из которого производилась упаковка.

// byte s = (byte)o;

Console.ReadKey();

}

}

// Упаковка и распаковка

struct MyStruct //: ValueType

{

public void Method()

{

Console.WriteLine("Method");

}

}

class Program

{

static void Main(string[] args)

{

MyStruct my = new MyStruct();

// Упаковка (Boxing).

ValueType boxed = my;

// Распаковка объекта (Unboxing).

MyStruct unboxed = (MyStruct)boxed;

Console.ReadKey();

}

}

Ковариантность не применима к массивам элементов структурных типов.

public interface IAnimal

{

void Voice();

}

public struct Dog: IAnimal

{

public void Voice()

{

Console.WriteLine("Voice");

}

}

class Program

{

static void Main(string[] args)

{

// 1

Dog[] dogs = { new Dog(), new Dog(), new Dog() };

// IAnimal[] animal = dogs; // Ковариантность.

// dogs = array; // Контрвариантность.

ValueType i = new Int32() as ValueType;

// 2

int[] vector = new int[3] {1, 2, 3};

// object[] array = vector; // Ковариантность.

Console.ReadKey();

}

}

Структура **DateTime** представляет текущее время, обычно выраженное как дата и время суток.

Тип значения **DateTime** представляет дату и время в диапазоне от 00:00:00 1 января 0001 года (н.э.) и до 23:59:59 31 декабря 9999 года (н. э.).

DateTime.Now – возвращает объект System.DateTime, которому присвоены текущие дата и время суток данного компьютера.

class Program

{

static void Main(string[] args)

{

DateTime now = DateTime.Now;

Console.WriteLine("Текущая дата и время: {0}", now);

Console.WriteLine("Год: {0}", now.Year);

Console.WriteLine("Месяц: {0}", now.Month);

Console.WriteLine("День месяца: {0}", now.Day);

Console.WriteLine("Текущее время - {0}:{1}:{2}", now.Hour, now.Minute, now.Second);

Console.WriteLine("День недели: {0}", now.DayOfWeek);

Console.WriteLine("День года: {0}", now.DayOfYear);

// Получаем дату текущего компьютера и значение времени, равное полуночи (00:00:00).

Console.WriteLine(DateTime.Now.Date);

Console.ReadKey();

}

}

// DateTime представляет момент(значение) времени, тогда как TimeSpan представляет интервал(промежуток) времени.

// Это означает, что можно вычесть один экземпляр DateTime из другого для получения объекта TimeSpan,

// который представляе собой временной интервал между ними.

// Или можно прибавить положительное значение TimeSpan к текущему значению DateTime, чтобы получить значение

// DateTime, которое представляет собой будущую дату.

class Program

{

static void Main(string[] args)

{

// Создание Новой даты. DateTime(гг, мм, дд)

DateTime newYearDate = new DateTime(2017, 1, 1);

DateTime today = DateTime.Now;

// Представляет интервал времени.

TimeSpan left = newYearDate - today;

Console.WriteLine("До нового года осталось {0} дней", left.Days);

Console.WriteLine("До нового года осталось {0} часов", left.TotalHours);

// Создание Новой даты и времени. DateTime(гг, мм, дд, чч, мин, сек)

DateTime newDate = new DateTime(2012, 12, 05, 23, 11, 11);

Console.WriteLine(newDate); // Вывод значения даты и времени на экран.

Console.WriteLine(newDate.TimeOfDay); // Вывод значения времени, установленного

// пользователем на экран.

// Преобразует заданное строковое представление даты и времени в его эквивалент.

Console.WriteLine(DateTime.Parse("3/12/2012"));

Console.WriteLine(DateTime.Parse("05 march 2012")); // Месяц написать на локальном языке ОС.

Console.ReadKey();

}

}

// Форматированный вывод даты и времени.

class Program

{

static void Main(string[] args)

{

DateTime now = DateTime.Now;

Console.WriteLine("Дата месяц(прописью) год: {0:D}", now);

Console.WriteLine("Дата.месяц.год: {0:d}", now);

Console.WriteLine("Дата месяц(прописью) год время(чч:мм:сс): {0:F}", now);

Console.WriteLine("Дата месяц(прописью) год время(чч:мм): {0:f}", now);

Console.WriteLine("Дата.месяц.год время(чч:мм:сс): {0:G}", now);

Console.WriteLine("Дата.месяц.год время(чч:мм) : {0:g}", now);

Console.WriteLine("Текущий месяц и дата: {0:M}", now);

Console.WriteLine("Текущий месяц и год: {0:Y}", now);

Console.WriteLine("Время(чч:мм:сс): {0:T}", now);

Console.WriteLine("Время(чч:мм): {0:t}", now);

Console.ReadKey();

}

}

**Перечисление** – это конструкция языка, которая содержит в себе набор именованных констант, которые хранят в себе определенное значение.

Перечисления наследуются от Enum, который наследуется от ValueType, поэтому они относятся к категории структурных типов.

// Перечисление - это набор именованных констант, которыех хранят числовые значения.

// Перечисление определяет именованные константы, каждой из которых соответствует числовое значение.

// Все перечисления в C# происходят от единого Базового класса System.Enum

// При компиляции - компилятор подставляет вместо имен установленные им

// в соответствие числовые значения. [имя] = [число]

// По умолчанию типом данных констант перечисления будет int.

// Можно использовать любой целый тип данных C# (byte, sbyte, short, ushort, int, uint, long, ulong)

enum EnumType : byte // Явно указываем использование типа byte.

{

Zero = 0,

One = 1,

Two = 2,

Three = 3

}

class Program

{

static void Main(string[] args)

{

Console.WriteLine(EnumType.One);

Console.WriteLine((byte)EnumType.One);

EnumType digit = EnumType.Zero;

Console.WriteLine(digit);

Console.WriteLine((byte)digit);

Console.ReadKey();

}

}

// Все перечисления в C# происходят от единого Базового класса System.Enum

enum EnumType: byte

{

Zero = 0,

One = 1,

Two = 2,

Three = 3

}

class Program

{

static void Main(string[] args)

{

Enum one = EnumType.One;

Console.WriteLine(one);

EnumType digit = EnumType.Zero;

Enum zero = digit;

Console.WriteLine(zero);

Console.WriteLine(new string('-', 10));

for(EnumType number = EnumType.Zero; number <= EnumType.Three; number++)

{

Console.WriteLine(number); // Для вывода значений указываем Console.WriteLine((byte)number);

}

Console.ReadKey();

}

}

// Можно использовать псевдоним для любого целого типа данных C# (byte, sbyte, short, ushort, int,

// uint, long, ulong)

// Нельзя использовать любой системный целый тип данных С# (Byte, SByte, Int16, UInt16, Int32,

// UInt32, Int64, UInt64)

enum EnumType // : Int32 // Ошибка.

{

Zero = 0,

One = 1,

Two = 2,

Three = 3

}

class Program

{

static void Main(string[] args)

{

Console.WriteLine(EnumType.One);

Console.ReadKey();

}

}

enum EnumType

{

Zero, // = 0

One, // = 1

one = One,

Two = 2,

Three, // = 3

Four, // = 4

Five = 5,

Six = Nine,

// Seven,

Eight = 8, // Если не указать значение явно, то на 13-й строке будет ошибка!

Nine,

Ten = 10,

Infinite = 255

}

class Program

{

// Enum как аргумент метода

public static void MethodEnum(EnumType e)

{

switch (e)

{

case EnumType.Zero:

Console.WriteLine("Число 0");

break;

case EnumType.Two:

Console.WriteLine("Число 1");

break;

case EnumType.Five:

Console.WriteLine("Число 5");

break;

case EnumType.Ten:

Console.WriteLine("Число 10");

break;

default: break;

}

}

static void Main(string[] args)

{

MethodEnum(EnumType.Five);

EnumType digit = EnumType.Ten;

MethodEnum(digit);

int i = (int)(++digit);

Console.WriteLine(i);

Console.WriteLine(digit); // Переменная изменилась.

Console.WriteLine((int)EnumType.Ten); // Константа не изменилась.

digit++;

digit = digit + 5;

Console.ReadKey();

}

}

// Перечисления. Значение по умолчанию для перечислений.

enum EnumType : byte

{

Zero = 0,

One = 1,

Two = 2,

Three = 3

}

class Program

{

static void Main(string[] args)

{

// Переменной five типа EnumType может быть назначено любое значение, входящее в диапазон

// базового типа, значения не ограничены именованными константами.

EnumType five = (EnumType)5;

Console.WriteLine(five);

Console.ReadKey();

}

}

}

enum EnumType

{

Zero, // = 0

One = 1,

one = One,

Two = 2,

Three, // = 3

Four, // = 4

Five = 5,

//Six,

Seven,

Eight = 8,

Nine,

Ten = 10,

Infinite = 255

}

// Перечисления. Получение информации о типе элементов перечисления.

class Program

{

static void Main(string[] args)

{

EnumType digit = EnumType.Ten;

// Enum.GetUnderlyingType() показывает тип, который используется для представления

// числовых значений элементов перечисления.

Type @enum = digit.GetType();

Console.Write("1. Перечисление {0} состоит из элементов типа: ", @enum.Name);

Type @const = Enum.GetUnderlyingType(@enum);

Console.WriteLine(@const);

// оператор typeof возвращает экземпляр класса System.Type, который содержит информацию

// о том типе, который был передан в качестве аргумента данного оператора.

Console.Write("2. Перечисление EnumType состоит из элементов типа: ");

Console.WriteLine(Enum.GetUnderlyingType(typeof(EnumType)));

Console.ReadKey();

}

}

enum EnumType

{

Zero, // = 0

One = 1,

one = One,

Two = 2,

Three, // = 3

Four, // = 4

Five = 5,

//Six,

Seven,

Eight = 8,

Nine,

Ten = 10,

Infinite = 255

}

// Перечисления. Форматированный вывод элементов перечисления

class Program

{

static void Main(string[] args)

{

EnumType digit = EnumType.Ten;

Console.WriteLine("Число {0}", digit.ToString());

// Enum.Format() - Позволяет производить более точное форматирование за счет указания флага,

// а также получать имена элементов перечисления по их числовым значениям.

// Вывод в 16-ричном формате. Флаг "х" - hex (16-ричный формат)

Console.WriteLine("Hex значение {0}", Enum.Format(typeof(EnumType), EnumType.Ten, "x"));

// Вывод в 10-тичном формате. Флаг "D" - dec (10-тичный формат)

Console.WriteLine("Dec значение {0}", Enum.Format(typeof(EnumType), digit, "D"));

// Вывод в строковом формате. Флаг "G" - str (Строковый формат)

Console.WriteLine("Str значение {0}", Enum.Format(typeof(EnumType), 10, "G"));

Console.ReadKey();

}

}

enum EnumType

{

Zero, // = 0

One = 1,

one = One,

Two = 2,

Three, // = 3

Four, // = 4

Five = 5,

Six = Nine,

//Seven

Eight = 8, // Если не указать значение явно, то на 13-й строке будет ошибка!

Nine,

Ten = 10,

Infinite = 255

}

// Поиск элемента перечисления по имени константы.

class Program

{

static void Main(string[] args)

{

// Находим элемент перечисления по имени константы.

object element = Enum.Parse(typeof(EnumType), "Infinite");

EnumType number = (EnumType)element;

Console.WriteLine("Значение константы {0}: {1}", number, (byte)number);

// Enum.IsDefined() - Позволяет определить, является ли символьная строка элементом перечисления?

bool flag = Enum.IsDefined(typeof(EnumType), "one");

if(flag == true)

{

Console.WriteLine("Да, перечисление содержит элемент с таким именем.");

}

else

Console.WriteLine("Нет, перечисление не содержит элемент с таким именем.");

Console.ReadKey();

}

}

enum EnumType

{

Zero, // = 0

One = 1,

one = One,

Two = 2,

Three, // = 3

Four, // = 4

Five = 5,

// Six,

Seven,

Eight = 8,

Nine,

Ten = 10,

Infinite = 255

}

// Получение информации о количестве элементов перечисления. Получение всех элемнтов перечисления и их значений.

class Program

{

static void Main(string[] args)

{

// Enum.GetValues() - возвращает экземпляр System.Array, при этом каждому элементу массива

// будет соответствовать член указанного перечисления.

// Помещаем в массив элементы перечисления.

Array array = Enum.GetValues(typeof(EnumType));

// Получаем информацию о количестве элементов в массиве.

Console.WriteLine("Это перечисление содержит {0} членов \n", array.Length);

// Вывод на экран всех элементов перечисления

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

{

Console.WriteLine("Имя константы: {0}, значение {0:D}", array.GetValue(i));

}

Console.ReadKey();

}

}

enum EnumType

{

Zero, // = 0

One = 1,

one = 1,

Two = 2,

Three, // = 3

Four, // = 4

Five = 5,

// Six,

Seven,

Eight = 8,

Nine,

Ten = 10,

Infinite = 255

}

// Сравнение элементов перечисления.

class Program

{

static void Main(string[] args)

{

EnumType x = EnumType.Five;

EnumType y = EnumType.One;

// if(x < y) // Первый вариант сравнения.

if (EnumType.Five < EnumType.One) // Второй вариант сравнения.

{

Console.WriteLine("X = {0} (меньше чем) Y = {1}.", x, y);

}

else

Console.WriteLine("X = {0} (больше чем) Y = {1}.", y, x);

Console.ReadKey();

}

}

Использование перечислений позволяет сделать исходные коды программ более читаемыми, так как позволяют заменить "магические числа", кодирующие определенное значение, на читаемые имена.

## Делегаты

// Делегаты.

// Класс, метод которого будет сообщен с делегатом.

static class MyClass

{

// Создаем статический метод, который планируем сообщить с делегатом.

public static void Method()

{

Console.WriteLine("Строку вывел метод, сообщенный с делегатом.");

}

}

// Создаем класс делегата с именем MyDelegate.

// Метод, который будет сообщен с экземпляром данного класса-делегата,

// не будет ничего принимать и возвращать.

public delegate void MyDelegate(); // Создаем класс делегата. (1)

class Program

{

static void Main(string[] args)

{

MyDelegate myDelegate = new MyDelegate(MyClass.Method); // Создаем экземпляр делегата. (2)

myDelegate.Invoke(); // Вызываем метод, сообщенный с делегатом. (3)

myDelegate(); // Другой способ вызова метода, сообщенного с делегатом. (3\*)

Console.ReadKey();

}

}

// Делегаты.

// Класс, метод которого будет сообщен с делегатом.

class MyClass

{

// Создаем метод, который планируем сообщить с делегатом.

public void Method()

{

Console.WriteLine("Строку вывел метод, сообщенный с делегатом.");

}

}

// Создаем класс-делегат с именем MyDelegate.

// Метод, который будет сообщен с экземпляром данного класса-делегата,

// не будет ничего принимать и возвращать.

public delegate void MyDelegate(); // Создаем класс-делегат. (1)

class Program

{

static void Main(string[] args)

{

MyClass instance = new MyClass();

MyDelegate myDelegate = new MyDelegate(instance.Method); // Создаем экземпляр делегата. (2)

myDelegate.Invoke(); // Вызываем метод, сообщенный с делегатом. (3)

myDelegate(); // Другой способ вызова метода, сообщенного с делегатом. (3\*)

Console.ReadKey();

}

}

// Делегаты.

// Класс, метод которого будет сообщен делегатом.

class MyClass

{

// Создаем метод, который планируем сообщить с делегатом.

public string Method(string name)

{

return "Hello" + name;

}

}

// Создаем класс-делегат с именем MyDelegate.

// Метод, который будет сообщен с экземпляром данного класса-делегата,

// будет принимать один строковой аргумент и возвращать строкове значение.

public delegate string MyDelegate(string name); // Создаем класс-делегат. (1)

class Program

{

static void Main(string[] args)

{

MyClass instance = new MyClass();

MyDelegate myDelegate = new MyDelegate(instance.Method); // Создаем экземпляр делегата и сообщаем с ним метод. (2)

string greeting = myDelegate.Invoke("Jeffrey Richter"); // Вызываем метод, сообщенный с делегатом. (3)

Console.WriteLine(greeting);

greeting = myDelegate("Grady Booch"); // Другой способ вызова метода, сообщенного с делегатом. (3\*)

Console.WriteLine(greeting);

Console.ReadKey();

}

}

// Комбинированные (групповые) делегаты.

public delegate void MyDelegate();

class Program

{

// Методы.

public static void Method1()

{

Console.WriteLine("Method1");

}

public static void Method2()

{

Console.WriteLine("Method2");

}

public static void Method3()

{

Console.WriteLine("Method3");

}

static void Main(string[] args)

{

MyDelegate myDelegate = null;

MyDelegate myDelegate1 = new MyDelegate(Method1);

MyDelegate myDelegate2 = new MyDelegate(Method2);

MyDelegate myDelegate3 = new MyDelegate(Method3);

// Комбинируем делегаты.

myDelegate = myDelegate1 + myDelegate2 + myDelegate3;

Console.WriteLine("Введите число от 1 до 7");

string choice = Console.ReadLine();

switch (choice)

{

case "1": myDelegate1.Invoke();

break;

case "2": myDelegate2.Invoke();

break;

case "3": myDelegate3.Invoke();

break;

case "4": MyDelegate myDelegate4 = myDelegate - myDelegate1;

myDelegate4.Invoke();

break;

case "5": MyDelegate myDelegate5 = myDelegate - myDelegate2;

myDelegate5.Invoke();

break;

case "6": MyDelegate myDelegate6 = myDelegate - myDelegate3;

myDelegate6.Invoke();

break;

case "7": myDelegate.Invoke();

break;

default: Console.WriteLine("Вы ввели недопустимое значение.");

break;

}

Console.ReadKey();

}

}

// Анонимгные (лямбда) методы.

// Создаем класс-делегат.

public delegate void MyDelegate();

class Program

{

static void Main(string[] args)

{

// Создаем экземпляр класса-делегата MyDelegate и сообщаем с ним анонимный метод.

MyDelegate myDelegate = delegate { Console.WriteLine("Hello, world!"); };

// Вызов анонимного метода, сообщенного с делегатом.

myDelegate();

Console.ReadKey();

}

}

// Анонимные (лямбда) методы.

// Создаем класс-делегат.

public delegate int MyDelegate(int a, int b);

class Program

{

static void Main(string[] args)

{

int summand1 = 1, summand2 = 2, sum = 0;

MyDelegate myDelegate = delegate (int a, int b) { return a + b; };

sum = myDelegate(summand1, summand2);

Console.WriteLine("{0} + {1} = {2}", summand1, summand2, sum);

Console.ReadKey();

}

}

// Анонимные (лямбда) методы.

// Создаем класс-делегат.

public delegate void MyDelegate(ref int a, ref int b, out int c);

class Program

{

static void Main(string[] args)

{

int summand1 = 1, summand2 = 2, sum;

MyDelegate myDelegate = delegate (ref int a, ref int b, out int c) {a++; b++; c = a + b; };

myDelegate(ref summand1, ref summand2, out sum);

Console.WriteLine("{0} + {1} = {2}", summand1, summand2, sum);

Console.ReadKey();

}

}

// Анонимные (лямбда) методы.

// Создаем класс-делегат.

public delegate void MyDelegate(ref int a, ref int b, out int c);

class Program

{

static void Main(string[] args)

{

int summand1 = 1, summand2 = 2, sum;

MyDelegate myDelegate = delegate (ref int a, ref int b, out int c) {a++; b++; c = a + b; };

myDelegate(ref summand1, ref summand2, out sum);

Console.WriteLine("{0} + {1} = {2}", summand1, summand2, sum);

Console.ReadKey();

}

}

// Лямбда выражения и лямбда операторы.

public delegate int MyDelegate(int a);

class Program

{

static void Main(string[] args)

{

MyDelegate myDelegate;

myDelegate = delegate (int x) { return x \*2; }; // Лямбда-Метод.

myDelegate = (x) => { return x \* 2; }; // Лямбда-Оператор.

myDelegate = x => x \* 2; // Лямбда-Выражение.

int result = myDelegate(4);

Console.WriteLine(result);

Console.ReadKey();

}

}

// Лямбда выражения и лямбда операторы.

public delegate void MyDelegate();

class Program

{

static void Main(string[] args)

{

MyDelegate myDelegate;

myDelegate = delegate { Console.WriteLine("Hello 1"); }; // Лямбда-метод

myDelegate += () => { Console.WriteLine("Hello 2"); }; // Лямбда-оператор

myDelegate += () => { Console.WriteLine("Hello 3"); }; // Лямбда-выражение

myDelegate();

Console.ReadKey();

}

}

// Делегаты.

delegate Delegate2 Delegate1(); // 1.

delegate void Delegate2(); // 2.

class Program

{

public static Delegate2 Method1() // 1.

{

return new Delegate2(Method2);

}

public static void Method2() // 2.

{

Console.WriteLine("Hello world!");

}

static void Main(string[] args)

{

Delegate1 delegate1 = new Delegate1(Method1);

Delegate2 delegate2 = delegate1();

delegate2();

Console.ReadKey();

}

}

// Делегаты.

delegate Delegate3 Functional(Delegate1 delegate1, Delegate2 delegate2);

delegate string Delegate1();

delegate string Delegate2();

delegate string Delegate3();

class Program

{

public static Delegate3 MethodF(Delegate1 delegate1, Delegate2 delegate2)

{

return delegate { return delegate1.Invoke() + delegate2.Invoke(); };

}

public static string Method1() { return "Hello "; }

public static string Method2() { return "world!"; }

static void Main(string[] args)

{

Functional functional = new Functional(MethodF);

Delegate3 delegate3 = functional.Invoke(new Delegate1(Method1), new Delegate2(Method2));

Console.WriteLine(delegate3.Invoke());

Console.ReadKey();

}

}

// Делегаты.

delegate MyDelegate Functional(MyDelegate delegate1, MyDelegate delegate2);

delegate string MyDelegate();

class Program

{

static void Main(string[] args)

{

MyDelegate delegate1 = () => "Hello", delegate2 = () => "world!";

Functional functional = delegate (MyDelegate d1, MyDelegate d2) { return delegate () { return d1.Invoke() + d2.Invoke(); }; };

// Functional functional = delegate (MyDelegate d1, MyDelegate d2) { return () => d1.Invoke() + d2.Invoke(); };

// Functional functional = (MyDelegate d1, MyDelegate d2) => () => d1() + d2();

// Console.WriteLine((functional.Invoke(delegate1, delegate2)).Invoke());

Console.WriteLine(functional(delegate1, delegate2)());

Console.ReadKey();

}

}

// Делегаты.

delegate MyDelegate Functional(MyDelegate delegate1, MyDelegate delegate2);

delegate string MyDelegate();

class Program

{

static void Main(string[] args)

{

MyDelegate delegate1 = () => "Hello", delegate2 = () => "world!";

Functional functional = delegate (MyDelegate d1, MyDelegate d2) { return delegate () { return d1.Invoke() + d2.Invoke(); }; };

// Functional functional = delegate (MyDelegate d1, MyDelegate d2) { return () => d1.Invoke() + d2.Invoke(); };

// Functional functional = (MyDelegate d1, MyDelegate d2) => () => d1() + d2();

// Console.WriteLine((functional.Invoke(delegate1, delegate2)).Invoke());

Console.WriteLine(functional(delegate1, delegate2)());

Console.ReadKey();

}

}

// Рекурсия в лямбда операторах.

delegate void MyDelegate(int argument);

class Program

{

static void Main(string[] args)

{

MyDelegate my = null; // Требуется обязательно присвоить null.

// Требуется отдельное присвоение ссылки на делегат с сообщенным лямбда оператором

// в месте создания переменной. Недопустимо сразу создавать лямбда оператор.

my = (int i) =>

{

i--;

Console.WriteLine("Begin {0}", i);

if (i > 0)

{

my(i);

}

Console.WriteLine("End {0}", i);

};

my(3);

Console.ReadKey();

}

}

## Универсальные шаблоны

// Универсальные шаблоны.

// Создаем класс с именем MyClass, параметризированный Указателем Места Заполнения Типом - Т

class MyClass<T>

{

public T field;

public void Method()

{

Console.WriteLine(field.GetType());

}

}

class Program

{

static void Main(string[] args)

{

// Создаем экземпляр класса MyClass и в качестве параметра типа (тип MyClass) передаем тип int.

MyClass<int> instance1 = new MyClass<int>();

instance1.Method();

// Создаем экземпляр класса MyClass и в качестве параметра типа (тип MyClass) передаем тип long.

MyClass<long> instance2 = new MyClass<long>();

instance2.Method();

// Создаем экземпляр класса MyClass и в качестве параметра типа (тип MyClass) передаем тип string.

MyClass<string> instance3 = new MyClass<string>();

instance3.field = "ABC";

instance3.Method();

Console.ReadKey();

}

}

// Универсальные шаблоны.

class MyGenerics<TYPE1, TYPE2>

{

// Поля.

private TYPE1 variable1;

private TYPE2 variable2;

// Конструктор.

public MyGenerics(TYPE1 argument1, TYPE2 argument2)

{

this.variable1 = argument1;

this.variable2 = argument2;

}

// Свойства.

public TYPE1 Variable1

{

get { return this.variable1; }

set { variable1 = value; }

}

public TYPE2 Variable2

{

get { return this.variable2; }

set { variable2 = value; }

}

}

class Program

{

static void Main(string[] args)

{

MyGenerics<int, int> instance1 = new MyGenerics<int, int>(1, 2);

Console.WriteLine(instance1.Variable1 + instance1.Variable2);

MyGenerics<string, int> instance2 = new MyGenerics<string, int>("Number ", 1);

Console.WriteLine(instance2.Variable1 + instance2.Variable2);

MyGenerics<string, string> instance3 = new MyGenerics<string, string>("Hello ", "World");

Console.WriteLine(instance3.Variable1 + instance3.Variable2);

Console.ReadKey();

}

}

// Универсальные шаблоны. (Универсальный метод)

class MyClass

{

public void Method<T>(T argument)

{

T variable = argument;

Console.WriteLine(variable);

}

}

class Program

{

static void Main(string[] args)

{

MyClass instance = new MyClass();

instance.Method<string>("Hello world!");

instance.Method("Привет мир!");

Console.ReadKey();

}

}

// Универсальные шаблоны. (Универсальный делегат)

// Создаем класс-делегат с именем MyDelegate, параметризированный двумя указателями места заполнения типом.

// Методы, который будет сообщен с экземпляром данного класса-делегата, будет принимать один аргумент типа указателя места

// и возвращать значение типа указателя места заполнения типом - R.

delegate R MyDelegate<T, R>(T t);

class Program

{

public static int Add(int i)

{

return ++i;

}

public static string Concatenation(string s)

{

return "Hello " + s + "!";

}

static void Main(string[] args)

{

MyDelegate<int, int> myDelegate1 = new MyDelegate<int, int>(Add);

int i = myDelegate1.Invoke(1);

Console.WriteLine(i);

MyDelegate<string, string> myDelegate2 = new MyDelegate<string, string>(Concatenation);

string s = myDelegate2("Alex");

Console.WriteLine(s);

Console.ReadKey();

}

}

// Ковариантность обобщений.

public abstract class Shape { }

public class Circle : Shape { }

public interface IContainer<T>

{

T Figure { get; set; }

}

public class Container<T> : IContainer<T>

{

public T Figure { get; set; }

public Container(T figure)

{

this.Figure = figure;

}

}

class Program

{

static void Main(string[] args)

{

Circle circle = new Circle();

IContainer<Circle> container = new Container<Circle>(circle);

Console.WriteLine(container.Figure.ToString());

Console.ReadKey();

}

}

// Ковариантность обобщений.

public abstract class Shape { }

public class Circle : Shape { }

public interface IContainer<T>

{

T Figure { get; set; }

}

public class Container<T> : IContainer<T>

{

public T Figure { get; set; }

public Container(T figure)

{

this.Figure = figure;

}

}

class Program

{

static void Main(string[] args)

{

Circle circle = new Circle();

IContainer<Shape> container = new Container<Shape>(circle);

Console.WriteLine(container.Figure.ToString());

Console.ReadKey();

}

}

// Ковариантность обобщений.

// Ковариантность обобщений в C# 4.0 ограничена интерфейсами и делегатами.

public abstract class Shape { }

public class Circle : Shape { }

public interface IContainer<out T>

{

T Figure { get; }

}

public class Container<T> : IContainer<T>

{

private T figure;

public T Figure

{

get { return figure; }

}

public Container(T figure)

{

this.figure = figure;

}

}

class Program

{

static void Main(string[] args)

{

Circle circle = new Circle();

IContainer<Shape> container = new Container<Circle>(circle);

Console.WriteLine(container.Figure.ToString());

Console.ReadKey();

}

}

// Контрвариантность обобщений.

// Контрвариантность обобщений в C# 4.0 ограничена интерфейсами и делегатами.

public abstract class Shape { }

public class Circle : Shape { }

public interface IContainer<in T>

{

T Figure { set; }

}

public class Container<T> : IContainer<T>

{

private T figure;

public Container(T figure)

{

this.figure = figure;

}

public T Figure

{

set { figure = value; }

}

public override string ToString()

{

return figure.GetType().ToString();

}

}

class Program

{

static void Main(string[] args)

{

Shape shape = new Circle();

IContainer<Circle> container = new Container<Shape>(shape);

Console.WriteLine(container.ToString());

Console.ReadKey();

}

}

// Ковариантность обобщений.

// Ковариантность обобщений в C# 4.0 ограничена интерфейсами и делегатами.

class Animal { }

class Cat : Animal { }

class Program

{

delegate T MyDelegate<out T>(); // out - для возвращаемого значения.

public static Cat CatCreator()

{

return new Cat();

}

static void Main(string[] args)

{

MyDelegate<Cat> delegateCat = new MyDelegate<Cat>(CatCreator);

MyDelegate<Animal> delegateAnimal = delegateCat; // От производного к базовому.

Animal animal = delegateAnimal.Invoke();

Console.WriteLine(animal.GetType().Name);

Console.ReadKey();

}

}

// Контрвариантность обобщений.

// Контрвариантность обобщений в C# 4.0 ограничена делегатами.

class Animal { }

class Cat : Animal { }

class Program

{

delegate void MyDelegate<in T>(T a); // in - для аргумента.

public static void CatUser(Animal animal)

{

Console.WriteLine(animal.GetType().Name);

}

static void Main(string[] args)

{

MyDelegate<Animal> delegateAnimal = new MyDelegate<Animal>(CatUser);

MyDelegate<Cat> delegateCat = delegateAnimal; // От базового к производному.

delegateAnimal(new Animal());

delegateCat(new Cat());

// delegateCat(new Animal()); // Невозможно.

Console.ReadKey();

}

}

// Частичные классы и методы.

// Частичные методы не могут иметь out параметры.

public partial class MyClass<T>

{

partial void PartialMethod<T>(T a, ref T b);

}

public partial class MyClass<T>

{

partial void PartialMethod<T>(T a, ref T b)

{

b = default(T);

Console.WriteLine("{0}, {1}", a, b);

}

public void Proxy(T a, ref T b)

{

PartialMethod(a, ref b);

}

}

class Program

{

static void Main(string[] args)

{

MyClass<int> instance = new MyClass<int>();

int a = 5, b = 6, c = 7;

instance.Proxy(a, ref b);

Console.ReadKey();

}

}

// Плохой пример. Так делать не рекомендуется.

class MyClass<T>

{

public T Add(T a, T b)

{

if(typeof(T) == typeof(int))

{

return (T)(Object)((int)(object)a + (int)(object)b);

}

if (typeof(T) == typeof(double))

{

return (T)(Object)((double)(object)a + (double)(object)b);

}

return (T)(object)0;

}

}

class Program

{

static void Main(string[] args)

{

MyClass<int> my = new MyClass<int>();

int sum = my.Add(2, 3);

Console.WriteLine(sum);

Console.ReadKey();

}

}

Тип **Nullable<T>** представляет типы значений с пустыми (нулевыми) значениями.

int? a = null;

int? b = a + 4;

b = null

При сравнении операндов, один из которых null, результатом сравнения всегда будет false.

int? a = null;

int? b = -5;

if(a >= b) // - false

{Console.WriteLine("a >= b");}

else

{Console.WriteLine("a < b");}

class Program

{

static void Main(string[] args)

{

// a - содержит неизвестное значение.

int? a = null;

int? b = a + 4; // b = null

int? c = a \* 5; // c = null

Console.WriteLine("->{0}<-", a);

Console.ReadKey();

}

}

class Program

{

static void Main(string[] args)

{

// a - содержит неизвестное значение.

int? a = null;

int? b = -5; // b = -5

// При сравнении операндов, один из которых = null, результатом сравнения всегда будет false.

// Следовательно нельзя рассчитывать на истинность (правильность) результата.

if(a >= b)

{

Console.WriteLine("a >= b");

}

else

{

Console.WriteLine("a < b");

}

// Сравнивать операнды (Nullable) есть смысл только для проверки - оба ли содержат null?

// И если оба операнда содержат null, то результатом сравнения будет true.

b = null;

if(a == b)

{

Console.WriteLine("a == b");

}

else

{

Console.WriteLine("a != b");

}

Console.ReadKey();

}

}

// Операция поглощения - ??

class Program

{

static void Main(string[] args)

{

int? a = null;

int? b;

b = a ?? 10; // b = 10

//b = (a != null) ? a : 10;

Console.WriteLine(b);

a = 3;

b = a ?? 10; // b = 3

Console.WriteLine(b);

Console.ReadKey();

}

}

// Значение по умолчанию для NullableTypes == null.

class Program

{

static int? i;

static void Main(string[] args)

{

if(i == null)

{

Console.WriteLine("null");

}

else if(i == 0)

{

Console.WriteLine("0");

}

else

{

Console.WriteLine("Другое...");

}

Console.ReadKey();

}

}

class Program

{

static void Main(string[] args)

{

Nullable<int> a = 1;

if(a.HasValue == true)

{

Console.WriteLine("a is {0}.", a.Value);

}

// Короткая нотация Nullable типа - "?". Она доступна только для C#.

int? b = 1;

if(b.HasValue == true)

{

Console.WriteLine("b is {0}", b.Value);

}

// Неявно типизированная локальная переменная не может быть Nullable.

// var? c = null;

Console.ReadKey();

}

}

## Ограничения универсальных шаблонов

// Ограничения параметров типа.

// where T : class - аргумент типа должен иметь ссылочный тип. Это также распространяется на тип

// любого класса, интерфейса.

class MyClass<T> where T : class

{

public T variable;

}

// where T : struct - аргумент типа должен иметь тип значения. Допускается указание любого типа значения, кроме Nullable.

class MyClass2<T> where T : struct

{

public T variable;

}

class Program

{

static void Main(string[] args)

{

MyClass<string> instance1 = new MyClass<string>();

// MyClass<int> instance1 = new MyClass<string>(); // Ошибка. int - структурный тип.

MyClass2<int> instance2 = new MyClass2<int>();

// MyClass2<string> instance2 = new MyClass2<string>(); // Ошибка. string - ссылочный тип.

Console.ReadKey();

}

}

// Ограничения параметров типа

// where T : new() - аргумент типа должен иметь открытый конструктор без параметров.

// При использовании с другими ограничениями ограничение new() должно устанавливаться последним:

// class MyClass<T> where T : class, new() {/\* ... \*/}

class MyClass<T> where T : new()

{

public T instance = new T();

public void GetValues()

{

Console.WriteLine(instance.ToString());

}

}

class TestClass

{

public int MyIntProperty { get; set; }

public string MyStringProperty { get; set; }

public override string ToString()

{

return string.Format("{0} - {1}", MyIntProperty, MyStringProperty);

}

}

class Program

{

static void Main(string[] args)

{

MyClass<TestClass> foo = new MyClass<TestClass>();

foo.instance.MyIntProperty = 1;

foo.instance.MyStringProperty = "Hello World!";

foo.GetValues();

Console.ReadKey();

}

}

// Ограничения параметров типа

class Base { }

class Derived : Base { }

// where T : Base - аргумент типа должен являться или быть производным от указанного базового класса.

class MyClass<T> where T : Base { }

class Program

{

static void Main(string[] args)

{

MyClass<Base> mc1 = new MyClass<Base>();

MyClass<Derived> mc2 = new MyClass<Derived>();

// MyClass<string> mc3 = new MyClass<string>(); // Ошибка.

Console.ReadKey();

}

}

// Ограничения параметров типа

interface IInterface { /\* ... \*/}

interface IInterface<U> { /\* ... \*/}

class Derived : IInterface, IInterface<object> { /\* ... \*/}

// where T : IInterface, IInterface<object> - аргумент типа должен являться или реализовывать указанный интерфейс.

// Можно установить несколько ограничений интерфейса.

// Ограничивающий интерфейс также может быть универсальным.

class MyClass<T> where T : IInterface, IInterface<object> { /\* ... \*/}

class MyClass2<T> where T : IInterface { /\* ... \*/}

class MyClass3<T> where T : IInterface<object> { /\* ... \*/}

class Program

{

static void Main(string[] args)

{

// В качестве аргумента типа подходит Derived, так как он наследуется от обоих интерфейсов.

MyClass<Derived> my1 = new MyClass<Derived>();

// MyClass<IInterface> my1 = new MyClass<IInterface>(); // Ошибка

// MyClass<IInterface<object>> my1 = new MyClass<IInterface<object>>(); // Ошибка

MyClass2<IInterface> my2 = new MyClass2<IInterface>();

MyClass2<Derived> my3 = new MyClass2<Derived>();

MyClass3<IInterface<object>> my4 = new MyClass3<IInterface<object>>();

MyClass3<Derived> my5 = new MyClass3<Derived>();

Console.ReadKey();

}

}

// Ограничения параметров типа

interface IInterface { /\* ... \*/}

interface IInterface<U> : IInterface { /\* ... \*/}

class Derived : IInterface, IInterface<object> { /\* ... \*/}

class Derived2 : IInterface<object> { /\* ... \*/}

// where T : IInterface, IInterface<object> - аргумент типа должен являться или реализовывать указанный интерфейс.

// Можно установить несколько ограничений интерфейса.

// Ограничивающий интерфейс также может быть универсальным.

class MyClass<T> where T : IInterface, IInterface<object> { /\* ... \*/}

class MyClass2<T> where T : IInterface<object> { /\* ... \*/}

class Program

{

static void Main(string[] args)

{

// В качестве аргумента типа подходит Derived, так как он наследуется от обоих интерфейсов.

MyClass<Derived> my1 = new MyClass<Derived>();

// MyClass<IInterface> my1 = new MyClass<IInterface>(); // Ошибка

// Аргумент типа подходит, так как IInterface<object> наследуется от IInterface

MyClass2<IInterface<object>> my2 = new MyClass2<IInterface<object>>();

MyClass2<Derived> my3 = new MyClass2<Derived>();

MyClass2<Derived2> my4 = new MyClass2<Derived2>();

MyClass2 <IInterface<object>> my5 = new MyClass2<IInterface<object>>();

Console.ReadKey();

}

}

// Ограничения параметров типа - "naked"

// Аргумент типа, предчставляемый в качестве Т, должен совпадать с аргументом, представляемым в качестве U

// Это называется ограничением типа "Naked".

class MyClass<T, R, U> where T : U {}

class Program

{

static void Main(string[] args)

{

MyClass<int, Object, int> my1 = new MyClass<int, object, int>();

MyClass<string, Object, string> my2 = new MyClass<string, object, string>();

// Не совпадают первый и третий аргументы типов - T и U (string и int).

// MyClass<string, Object, int> my2 = new MyClass<string, object, int>(); // Ошибка!

Console.ReadKey();

class Program

{

static void Main(string[] args)

{

ArrayList arrayList = new ArrayList();

// Boxing.

arrayList.Add(1);

arrayList.Add((object)2);

// Unboxing.

int i1 = (int)arrayList[0];

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

{

Console.WriteLine((int)arrayList[i]);

}

Console.WriteLine(new string('-', 3));

List<int> list = new List<int>();

// Упаковки нет.

list.Add(3);

list.Add(4);

// Распаковки нет.

int i3 = list[0];

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

{

Console.WriteLine(list[i]);

}

Console.ReadKey();

}

}

// Словари (ключ/значение)

class Program

{

static void Main(string[] args)

{

Dictionary<int, string> dictionary = new Dictionary<int, string>();

dictionary.Add(0, "Ноль");

dictionary.Add(1, "Один");

dictionary.Add(2, "Два");

dictionary.Add(3, "Три");

Console.WriteLine(dictionary.ContainsValue("Ноль"));

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

{

Console.WriteLine(dictionary[i]);

}

Console.ReadKey();

}

}

## События

// События.

public delegate void EventDelegate();

public class MyClass

{

public event EventDelegate MyEvent = null;

public void InvokeEvent()

{

MyEvent.Invoke();

}

}

class Program

{

// Методы обработчики события.

static private void Handler1()

{

Console.WriteLine("Обработчик события 1");

}

static private void Handler2()

{

Console.WriteLine("Обработчик события 2");

}

static void Main(string[] args)

{

MyClass instance = new MyClass();

// Присоединение обработчиков событий. (Подписка на событие)

instance.MyEvent += new EventDelegate(Handler1);

instance.MyEvent += Handler2;

// Метод, который вызывает событие.

instance.InvokeEvent();

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

// Открепляем Handler2().

instance.MyEvent -= new EventDelegate(Handler2);

instance.InvokeEvent();

Console.ReadKey();

}

}

// События.

public delegate void EventDelegate();

public class MyClass

{

EventDelegate myEvent = null;

// Реализация методов доступа add и remove для события.

public event EventDelegate MyEvent

{

add { myEvent += value; }

remove { myEvent -= value; }

}

public void InvokeEvent()

{

myEvent.Invoke();

}

}

class Program

{

// Методы обработчики события.

static private void Handler1()

{

Console.WriteLine("Обработчик события 1");

}

static private void Handler2()

{

Console.WriteLine("Обработчик события 2");

}

static void Main(string[] args)

{

MyClass instance = new MyClass();

// Подписка на событие.

instance.MyEvent += Handler1;

instance.MyEvent += Handler2;

// Метод, который вызывает событие.

instance.InvokeEvent();

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

// Открепляем Handler2().

instance.MyEvent -= new EventDelegate(Handler2);

instance.InvokeEvent();

Console.ReadKey();

}

}

// События.

public delegate void PressKeyEventHandler();

public class Keyboard

{

public event PressKeyEventHandler PressKeyA = null;

public event PressKeyEventHandler PressKeyB = null;

public event PressKeyEventHandler PressKeyC = null;

// D...Z

public void PressKeyAEvent()

{

if (PressKeyA != null)

{

PressKeyA.Invoke(); // вызываем метод Invoke на событии.

}

}

public void PressKeyBEvent()

{

if(PressKeyB != null)

{

PressKeyB.Invoke();

}

}

public void PressKeyCEvent()

{

if (PressKeyC != null)

{

PressKeyC.Invoke();

}

}

public void Start()

{

while (true)

{

string s = Console.ReadLine();

switch (s)

{

case "a":

case "A":

PressKeyAEvent();

break;

case "b":

case "B":

PressKeyBEvent();

break;

case "c":

case "C":

PressKeyCEvent();

break;

case "exit":

goto Exit;

default:

Console.WriteLine("Нет обработчика нажатия на кнопку {0}", s);

break;

}

}

Exit:

Console.WriteLine("Exit!");

}

}

class Program

{

// Методы обработчики события.

static private void PressKeyA\_Handler()

{

Console.ForegroundColor = ConsoleColor.Blue;

Console.WriteLine();

Console.WriteLine(" X ");

Console.WriteLine(" X X ");

Console.WriteLine(" X X ");

Console.WriteLine(" XXXXXXX ");

Console.WriteLine("X X ");

Console.WriteLine();

Console.ForegroundColor = ConsoleColor.Gray;

}

static private void PressKeyB\_Handler()

{

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine();

Console.WriteLine("XXXXX ");

Console.WriteLine("X X ");

Console.WriteLine("XXXXXX ");

Console.WriteLine("X X");

Console.WriteLine("XXXXXX");

Console.WriteLine();

Console.ForegroundColor = ConsoleColor.Gray;

}

static private void PressKeyC\_Handler()

{

Console.ForegroundColor = ConsoleColor.DarkYellow;

Console.WriteLine();

Console.WriteLine("XXXXX ");

Console.WriteLine("X X");

Console.WriteLine("X ");

Console.WriteLine("X X");

Console.WriteLine("XXXXX ");

Console.WriteLine();

Console.ForegroundColor = ConsoleColor.Gray;

}

static void Main(string[] args)

{

Keyboard keyboard = new Keyboard();

keyboard.PressKeyA += new PressKeyEventHandler(PressKeyA\_Handler);

keyboard.PressKeyB += PressKeyB\_Handler;

keyboard.PressKeyC += PressKeyC\_Handler;

// Запуск метода, который будет следить за нажатием клавиш

keyboard.Start();

}

}

// События (abstract and virtual).

public delegate void EventDelegate();

interface IInterface

{

event EventDelegate MyEvent; // Абстрактное событие.

}

public class BaseClass : IInterface

{

EventDelegate myEvent = null;

public virtual event EventDelegate MyEvent // Виртуальное событие.

{

add { myEvent += value; }

remove { myEvent -= value; }

}

public void InvokeEvent()

{

myEvent.Invoke();

}

}

public class DerivedClass : BaseClass

{

public override event EventDelegate MyEvent // Переопределенное событие.

{

add

{

base.MyEvent += value;

Console.WriteLine("К событию базового класса был прикреплен обработчик - {0}", value.Method.Name);

}

remove

{

base.MyEvent -= value;

Console.WriteLine("От события базового класса был откреплен обработчик - {0}", value.Method.Name);

}

}

}

class Program

{

// Методы обработчики события.

static private void Handler1()

{

Console.WriteLine("Обработчик события 1");

}

static private void Handler2()

{

Console.WriteLine("Обработчик события 2");

}

static void Main(string[] args)

{

DerivedClass instance = new DerivedClass();

// Присоединение обработчиков событий.

instance.MyEvent += new EventDelegate(Handler1);

instance.MyEvent += Handler2;

// Метод, который вызывает событие.

instance.InvokeEvent();

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

// Открепляем Handler2.

instance.MyEvent -= new EventDelegate(Handler2);

instance.InvokeEvent();

Console.ReadKey();

}

}

// События (abstract and virtual).

public delegate void EventDelegate();

interface IInterface

{

event EventDelegate MyEvent;

void InvokeEvent();

}

public class BaseClass : IInterface

{

public virtual event EventDelegate MyEvent = null;

public virtual void InvokeEvent()

{

MyEvent.Invoke();

}

}

public class DerivedClass : BaseClass

{

public override event EventDelegate MyEvent = null;

public override void InvokeEvent()

{

MyEvent.Invoke();

}

}

class Program

{

static private void Handler1()

{

Console.WriteLine("Обработчик события 1");

}

static private void Handler2()

{

Console.WriteLine("Обработчик события 2");

}

static void Main(string[] args)

{

DerivedClass instance = new DerivedClass();

// Присоединение обработчиков событий.

instance.MyEvent += new EventDelegate(Handler1);

instance.MyEvent += Handler2;

// Метод, который вызывает событие.

instance.InvokeEvent();

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

// Открепляем Handler2

instance.MyEvent -= new EventDelegate(Handler2);

instance.InvokeEvent();

Console.ReadKey();

}

}

// События.

public delegate void EventDelegate();

public class MyClass

{

public event EventDelegate MyEvent = null;

public void InvokeEvent()

{

MyEvent.Invoke();

}

}

class Program

{

// Методы обработчики события.

static private void Handler1()

{

Console.WriteLine("Обработчик события 1");

}

static private void Handler2()

{

Console.WriteLine("Обработчик события 2");

}

static void Main(string[] args)

{

MyClass instance = new MyClass();

// Присоединение обработчиков событий.

instance.MyEvent += new EventDelegate(Handler1);

instance.MyEvent += new EventDelegate(Handler2);

instance.MyEvent += delegate { Console.WriteLine("Анонимный метод 1."); };

instance.InvokeEvent();

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

// Открепляем Handler2.

instance.MyEvent -= new EventDelegate(Handler2);

// Невозможно открепить ранее присоединенный анонимный метод.

instance.MyEvent -= delegate { Console.WriteLine("Анонимный метод 1."); };

instance.InvokeEvent();

Console.ReadKey();

}

}

## Потоки

**Многозадачность** (multitasking) – свойство операционной системы или среды программирования обеспечивать возможность параллельной (или псевдопараллельной) обработки нескольких процессов.

Многозадачность может быть на основе:

* процессов,
* потоков.

Многозадачность на основе процессов позволяет выполнить одновременно более одной программы в контексте операционной системы. При использовании многозадачности на основе процессов программа является наименьшей единицей кода, выполнение которой может контролировать планировщик задач.

Многозадачность на основе потоков означает параллельное выполнение отдельных частей программы. Это означает, что процесс может решать более одной задачи одновременно. У каждого процесса может быть один или более потоков.

**System.Threading** – пространство имен для работы с потоками. Для управления потоками оно содержит следующие классы: Thread, ThreadStart, ParameterizedThreadStart, Monitor.

// Потоки.

class Program

{

// Метод, который планируется выполнять в отдельном потоке.

static void WriteSecond()

{

while (true)

{

Console.WriteLine(new string(' ', 10) + "Secondary");

}

}

static void Main(string[] args)

{

ThreadStart writeSecond = new ThreadStart(WriteSecond);

Thread thread = new Thread(writeSecond);

thread.Start();

while (true)

{

Console.WriteLine("Primary");

}

Console.ReadKey();

}

}

// Потоки.

class Program

{

// Статический метод, который планируется выполнять одновременно в главном (первичном) и во вторичном потоках.

static void WriteSecond()

{

// CLR назначает каждому потоку свой стек и методы имеют свои собственные локальные переменные.

// Отдельный экзмепляр переменной counter создается в стеке каждого потока,

// поэтому для каждого потока выводятся свои значения counter - 0, 1, 2.

int counter = 0;

while(counter < 10)

{

Thread.Sleep(500);

Console.WriteLine("Thread ID {0}, counter = {1}", Thread.CurrentThread.GetHashCode(), counter);

}

}

static void Main(string[] args)

{

// Работа вторичного потока.

Thread thread = new Thread(WriteSecond);

thread.Start();

// Работа первичного потока.

WriteSecond();

Console.ReadKey();

}

}

// Потоки.

class Program

{

static void WriteSecond()

{

// Thread.CurrentThread - возвращает ссылку на экземпляр текущего потока.

Thread thread = Thread.CurrentThread;

// Присваиваем потоку имя.

thread.Name = "Secondary";

// Выводим на экран информацию о текущем потоке.

Console.WriteLine("ID потока {0}: {1}", thread.Name, thread.GetHashCode());

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

{

Console.WriteLine(new string(' ', 15) + thread.Name + " " + counter);

// Приостанавливаем выполнение текущего потока.

Thread.Sleep(1000);

}

}

static void Main(string[] args)

{

// Получаем ссылку на экземпляр текущего потока.

Thread primaryThread = Thread.CurrentThread;

// Присваиваем потоку имя.

primaryThread.Name = "Primary";

// Выводим на экран информацию о текущем потоке.

Console.WriteLine("ID потока {0}: {1}", primaryThread.Name, primaryThread.GetHashCode());

// Работа вторичного потока.

Thread secondaryThread = new Thread(WriteSecond);

secondaryThread.Start();

// Работа первичного потока.

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

{

Console.WriteLine(primaryThread.Name + " " + counter);

// Приостанавливаем выполнение текущего потока.

Thread.Sleep(1500);

}

Console.ReadKey();

}

}

// Потоки. Передача данных в поток.

class Program

{

// Метод, который планируется выполнять в отдельном потоке.

static void WriteSecond(object argument)

{

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

{

Console.WriteLine(argument);

Thread.Sleep(1000);

}

}

static void Main(string[] args)

{

ParameterizedThreadStart writeSecond = new ParameterizedThreadStart(WriteSecond);

Thread thread = new Thread(writeSecond);

thread.Start("Hello");

Thread.Sleep(500);

Console.ReadKey();

}

}

// Потоки. Анонимные методы.

class Program

{

static void Main(string[] args)

{

int counter = 0;

// ThreadStart

Thread thread = new Thread(delegate () { Console.WriteLine("1. counter = {0}", ++counter); });

thread.Start();

Thread.Sleep(100);

Console.WriteLine("2. counter = {0}", counter);

// ParameterizedThreadStart

thread = new Thread((object argument) => { Console.WriteLine("3. counter = {0}", (int)argument); });

thread.Start(counter);

Console.ReadKey();

}

}

// Основные и фоновые потоки. По умолчанию свойство IsBackground равно false.

class Program

{

// Метод, который планируется выполнять в отдельном потоке.

static void WriteSecond()

{

while (true)

{

Console.WriteLine(new string(' ', 15) + "Secondary");

Thread.Sleep(500);

}

}

static void Main(string[] args)

{

// Работа вторичного потока.

ThreadStart writeSecond = new ThreadStart(WriteSecond);

Thread thread = new Thread(writeSecond);

thread.Start();

// Работа первичного потока.

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

{

Console.WriteLine("Primary");

Thread.Sleep(500);

}

// Завершить работу вторичного потока.

// thread.IsBackground = true;

Console.ReadKey();

}

}

// Критическая секция (critical section).

// lock - блокирует блок кода так, что в каждый отдельный момент времени это блок кода

// сможет использовать только один поток. Все остальные потоки ждут, пока текущий поток закончит работу.

class MyCLass

{

object block = new object();

public void Method()

{

int hash = Thread.CurrentThread.GetHashCode();

lock (block) // Закомментировать lock.

{

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

{

Console.WriteLine("Поток # {0}: шаг {1}", hash, counter);

Thread.Sleep(200);

}

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

}

}

}

class Program

{

static void Main(string[] args)

{

Console.SetWindowSize(80, 40);

MyCLass instance = new MyCLass();

for(int i = 0; i < 3; i++)

{

new Thread(instance.Method).Start();

}

Thread.Sleep(500);

Console.ReadKey();

}

}

// Критическая секция (critical section).

class Program

{

static object locker = new object();

static void WriteSecond()

{

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

{

lock (locker)

{

Console.ForegroundColor = ConsoleColor.Yellow;

Console.WriteLine(new string('-', 10) + "Secondary");

Console.ForegroundColor = ConsoleColor.Gray;

Thread.Sleep(100);

}

}

}

static void Main(string[] args)

{

Console.SetWindowSize(80, 45);

ThreadStart writeSecond = new ThreadStart(WriteSecond);

Thread thread = new Thread(writeSecond);

thread.Start();

for(int i = 0; i < 20; i++)

{

lock (locker)

{

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine("Primary");

Console.ForegroundColor = ConsoleColor.Gray;

Thread.Sleep(100);

}

}

Console.ReadKey();

}

}

## Коллекции

// Экземпляры этого класса будет содержать коллекция UserCollection.

public class Element

{

// Поля.

private string name;

private int field1;

private int field2;

// Конструктор.

public Element(string s, int a, int b)

{

name = s;

field1 = a;

field2 = b;

}

// Свойства.

public int Field1

{

get { return field1; }

set { field1 = value; }

}

public int Field2

{

get { return field2; }

set { field2 = value; }

}

public string Name

{

get { return name; }

set { name = value; }

}

}

// Класс UserCollection коллекция (набор) объектов класса Element.

// Для применения foreach необходимо, чтобы класс реализовывал интерфейс IEnumerable.

public class UserCollection : IEnumerable, IEnumerator

{

Element[] elementsArray = null;

public UserCollection()

{

elementsArray = new Element[4];

elementsArray[0] = new Element("A", 1, 10);

elementsArray[1] = new Element("B", 2, 20);

elementsArray[2] = new Element("C", 3, 30);

elementsArray[3] = new Element("B", 4, 40);

}

// Указатель текущей позиции элемента в массиве.

int position = -1;

//-------------------------------------------------------------------------

// Реализация интерфейса IEnumerator.

// Передвинуть внутренний указатель (position) на одну позицию.

public bool MoveNext()

{

if(position < elementsArray.Length - 1)

{

position++;

return true;

}

else

{

return false;

}

}

// Установить указатель (position) перед началом набора.

public void Reset()

{

position = -1;

}

// Получить текущий элемент набора.

public object Current

{

get { return elementsArray[position]; }

}

//----------------------------------------------------------------------

// Реализация интерфейса IEnumerable.

IEnumerator IEnumerable.GetEnumerator()

{

return this as IEnumerator;

}

}

class Program

{

static void Main(string[] args)

{

UserCollection myCollection = new UserCollection();

// Используем foreach для обращения к каждому объекту Element внутри массива myCollection.

foreach(Element element in myCollection)

{

Console.WriteLine("Name : {0} Field1: {1} Field2: {2}", element.Name, element.Field1, element.Field2);

}

// myCollection.Reset(); // Убрать комментарий для проверки.

Console.WriteLine(new string ('-', 29) + "\n");

// Используем foreach для повторного обращения к каждому объекту Element внутри массива myCollection.

foreach(Element element in myCollection)

{

Console.WriteLine("Name : {0} Field1: {1} Field2: {2}", element.Name, element.Field1, element.Field2);

}

Console.WriteLine(new string('-', 29) + "\n");

//-------------------------------------------------------------------------------------------------

// Так работает foreach.

UserCollection myElementsCollection = new UserCollection();

// foreach - приводит коллекцию к интерфейсному типу IEnumerable.

IEnumerable enumerable = myCollection as IEnumerable;

// foreach - приводит коллекцию к интерфейсному типу, вызывая метод GetEnumerator().

IEnumerator enumerator = enumerable.GetEnumerator();

while (enumerator.MoveNext()) // Перемещаем курсор на один шаг вперед (с -1 на 0 и т.д.).

{

Element element = enumerator.Current as Element;

Console.WriteLine("Name: {0} FIeld1: {1} Field2: {2}", element.Name, element.Field1, element.Field2);

}

enumerator.Reset();

Console.ReadKey();

}

}

class UserCollection

{

public static IEnumerable Power()

{

yield return "Hello world!";

}

}

class Program

{

static void Main(string[] args)

{

foreach(string element in UserCollection.Power())

{

Console.WriteLine(element);

}

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

//-------------------------------------------------------

// Так работает foreach.

IEnumerable enumerable = UserCollection.Power();

IEnumerator enumerator = enumerable.GetEnumerator();

while (enumerator.MoveNext()) // Перемещаем курсор на один шаг вперед.

{

String element = enumerator.Current as String;

Console.WriteLine(element);

}

Console.ReadKey();

}

}

class UserCollection

{

public static IEnumerable Power()

{

while (true)

yield return "Hello world!";

}

}

class Program

{

static void Main(string[] args)

{

foreach(string element in UserCollection.Power())

{

Console.WriteLine(element);

}

Console.ReadKey();

}

}

// Экземпляры этого класса будет содержать коллекция UserCollection.

class Element

{

// Поля.

private string name;

private int field1;

private int field2;

// Конструктор.

public Element(string s, int a, int b)

{

name = s;

field1 = a;

field2 = b;

}

// Свойства.

public int Field1

{

get { return field1; }

set { field1 = value; }

}

public int Field2

{

get { return field2; }

set { field2 = value; }

}

public string Name

{

get { return name; }

set { name = value; }

}

}

class UserCollection : IEnumerable

{

Element[] elementsArray = null;

public UserCollection()

{

elementsArray = new Element[4];

elementsArray[0] = new Element("A", 1, 10);

elementsArray[1] = new Element("B", 2, 20);

elementsArray[2] = new Element("C", 3, 30);

elementsArray[3] = new Element("D", 4, 40);

}

// Указатель текущей позиции элемента в массиве.

int position = -1;

// Установить указатель (position) перед началом набора.

public void Reset()

{

position = -1;

}

//----------------------------------------------------------------------

// Реализация интерфейса IEnumerable.

public IEnumerator GetEnumerator()

{

while (true)

{

if(position < elementsArray.Length - 1)

{

position++;

yield return elementsArray[position];

}

else

{

Reset();

yield break; // Выход из цикла.

}

}

}

}

class Program

{

static void Main(string[] args)

{

UserCollection myCollection = new UserCollection();

// Используем foreach для обращения к каждому объекту Element внутри массива myCollection.

foreach (Element element in myCollection)

{

Console.WriteLine("Name : {0} Field1: {1} Field2: {2}", element.Name, element.Field1, element.Field2);

}

Console.ReadKey();

}

}

## Обработка исключений

// Обработка исключений.

class Program

{

static void Main(string[] args)

{

int a = 1, b = 2;

try

{

// Попытка деления на ноль.

a = a / (2 - b);

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

}

catch (Exception e)

{

Console.WriteLine("Обработка исключения.");

Console.WriteLine(e.Message);

}

Console.ReadKey();

}

}

// Обработка исключений.

class Program

{

static void Main(string[] args)

{

Exception ex = new Exception("Мое исключение");

try

{

throw ex;

}

catch(Exception e)

{

Console.WriteLine("Обработка исключения.");

Console.WriteLine(e.Message);

}

Console.ReadKey();

}

}

// Обработка исключений.

class Program

{

static void Main(string[] args)

{

try

{

throw new Exception("Мое исключение");

}

catch(Exception e)

{

Console.WriteLine("Обработка исключения.");

Console.WriteLine(e.Message);

}

Console.ReadKey();

}

}

// Обработка исключений.

class MyClass

{

public void MyMethod()

{

Exception exception = new Exception("Мое исключение");

exception.HelpLink = "http://MyCompany.com/ErrorService";

exception.Data.Add("Причина исключения: ", "Тестовое исключение");

exception.Data.Add("Время возникновения исключения: {0}", DateTime.Now);

throw exception;

}

}

class Program

{

static void Main(string[] args)

{

try

{

MyClass instance = new MyClass();

instance.MyMethod();

}

catch(Exception e)

{

Console.WriteLine("Имя члена: {0}", e.TargetSite);

Console.WriteLine("Класс, определяющий член: {0}", e.TargetSite.DeclaringType);

Console.WriteLine("Тип члена: {0}", e.TargetSite.MemberType);

Console.WriteLine("Message: {0}", e.Message);

Console.WriteLine("Source: {0}", e.Source);

Console.WriteLine("Help Link: {0}", e.HelpLink);

Console.WriteLine("Stack: {0}", e.StackTrace);

foreach(DictionaryEntry de in e.Data)

Console.WriteLine("{0} : {1}", de.Key, de.Value);

}

Console.ReadKey();

}

}

// Обработка исключений.

// Для создания пользовательского исключения трубется наследование от System.Exception.

class UserException : Exception

{

public void Method()

{

Console.WriteLine("Мое исключение!");

}

}

class Program

{

static void Main(string[] args)

{

try

{

throw new UserException();

}

catch(UserException e)

{

Console.WriteLine("Обработка исключения.");

e.Method();

}

Console.ReadKey();

}

}

// Обработка исключений.

// Для создания пользовательского исключения требуется наследование от System.Exception.

class UserException : Exception

{

public void Method()

{

Console.WriteLine("Мое исключение!");

}

}

class Program

{

static void Main(string[] args)

{

try

{

throw new UserException();

}

catch (UserException userException)

{

Console.WriteLine("Обработка исключения.");

userException.Method();

try

{

FileStream fs = File.Open(@"C:\NonExistentFile.log", FileMode.Open);

}

catch (UserException exception)

{

Console.WriteLine(exception.Message);

}

}

Console.ReadKey();

}

}

// Обработка исключений.

class Program

{

static void Main(string[] args)

{

int a = 1, n = 2;

try

{

Console.ForegroundColor = ConsoleColor.Yellow;

Console.WriteLine("Попытка деления на ноль.");

Console.WriteLine("a / (2 - n) = {0}", a / (2 - n));

}

catch(Exception e)

{

Console.ForegroundColor = ConsoleColor.White;

Console.BackgroundColor = ConsoleColor.Red;

Console.WriteLine("Обработка исключения.");

Console.WriteLine(e.Message);

}

finally

{

Console.ForegroundColor = ConsoleColor.Gray;

Console.BackgroundColor = ConsoleColor.Black;

}

Console.WriteLine("Press and key...");

Console.ReadKey();

}

}

// Обработка исключений.

// Для создания пользовательского исключения требуется наследование от System.Exception.

class UserException : Exception

{

public void Method()

{

Console.WriteLine("Мое Исключение!");

}

}

class Program

{

static void Main(string[] args)

{

try

{

throw new UserException();

}

catch (UserException userException)

{

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine("Обработка исключения 1:");

userException.Method();

try

{

throw userException;

}

catch(UserException exception)

{

Console.ForegroundColor = ConsoleColor.Yellow;

Console.WriteLine("Обработка исключения 2:");

exception.Method();

}

}

finally

{

Console.ForegroundColor = ConsoleColor.Gray;

}

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

Console.ReadKey();

}

}

// Обработка внутренних исключений.

public class ClassWithException

{

public void ThrowInner()

{

throw new Exception("Это внутреннее исключение!");

}

public void CatchInner()

{

try

{

this.ThrowInner();

}

catch(Exception e)

{

throw new Exception("Это внешннее исключение!", e);

}

}

}

class Program

{

static void Main(string[] args)

{

ClassWithException instance = new ClassWithException();

// instance.CatchInner(); // Попробовать вызвать.

try

{

instance.CatchInner();

}

catch (Exception exception)

{

Console.WriteLine("Exception caught: {0}", exception.Message);

Console.WriteLine("Inner Exception: {0}", exception.InnerException.Message);

}

Console.ReadKey();

}

}

// Обработка исключений.

// finally - не срабатывает в случае возникновения исключения StackOverflowException.

class Program

{

public static void Method()

{

int[] arr = new int[10];

Console.WriteLine(arr);

Method();

}

static void Main(string[] args)

{

try

{

Method();

}

catch (Exception ex)

{

Console.WriteLine(ex.Message);

}

finally

{

// finally - не сработает.

while(true)

Console.WriteLine("Finally");

}

Console.ReadKey();

}

}

// Обработка исключений.

// finally - срабатывает в случае возникновения исключения OutOfMemoryException.

class Program

{

public static void Method()

{

int[] arr = new int[100000000];

Console.WriteLine(arr);

Method();

}

static void Main(string[] args)

{

try

{

Method();

}

catch(Exception ex)

{

Console.WriteLine(ex.Message);

}

finally

{

// finally - сработает.

Console.WriteLine("Finally");

}

Console.ReadKey();

}

}

// Обработка исключений.

// finally - не срабатывает, если не завершается работа блока catch.

class Program

{

static void Main(string[] args)

{

try

{

throw new Exception();

}

catch (Exception)

{

throw new Exception();

// или так...

while (true)

Console.WriteLine("Catch");

}

finally

{

// finally - не сработает!

while (true)

Console.WriteLine("Finally");

}

Console.ReadKey();

}

}

// Обработка исключений.

class Program

{

static void Main(string[] args)

{

try

{

try

{

try

{

throw new Exception("Exception");

}

catch(Exception e)

{

Console.WriteLine("Catch 1: " + e.Message);

throw;

}

finally

{

Console.WriteLine("Finally 1:");

}

}

catch(Exception e)

{

Console.WriteLine("Catch 2: " + e.Message);

throw;

}

finally

{

Console.WriteLine("Finally 2: ");

throw new Exception("New Exception");

}

}

catch(Exception e)

{

Console.WriteLine("Catch 3: " + e.Message);

}

finally

{

Console.WriteLine("Finally 3: ");

}

Console.ReadKey();

}

}

// Обработка исключений.

class Program

{

static void Main(string[] args)

{

try

{

throw null;

}

catch(NullReferenceException ex)

{

Console.WriteLine(ex.Message);

}

Console.ReadKey();

}

}

// Обработка исключений. (Порядок обработки исключений)

class MyExceptionA : Exception

{

public MyExceptionA(string message) : base(message)

{

}

}

class MyExceptionB : MyExceptionA

{

public MyExceptionB(string message) : base(message)

{

}

}

class Program

{

static void Main(string[] args)

{

try

{

// throw new Exception("Exception");

throw new MyExceptionA("MyExceptionA");

// throw new MyExceptionB("MyExceptionB");

}

catch (MyExceptionB e)

{

Console.WriteLine(e.Message);

}

catch (MyExceptionA e)

{

Console.WriteLine(e.Message);

}

catch (Exception e)

{

Console.WriteLine(e.Message);

}

Console.ReadKey();

}

}

## Перегрузка операторов. Клонирование. Базовый тип Object

// Базовый класс Object.

class MyClassA : Object

{

}

class MyClassB : object

{

}

class Program

{

static void Main(string[] args)

{

Object instanceA = new MyClassA();

object instanceB = new MyClassB();

Console.ReadKey();

}

}

// Базовый класс Object.

class MyClass

{

public override string ToString()

{

return "Hello world!";

}

}

class Program

{

static void Main(string[] args)

{

MyClass instance = new MyClass();

Console.WriteLine(instance.ToString());

Console.ReadKey();

}

}

}

// Базовый класс Object.

class MyClass

{

public override int GetHashCode()

{

return 1234567890;

}

}

class Program

{

static void Main(string[] args)

{

MyClass instance = new MyClass();

Console.WriteLine(instance.GetHashCode());

Console.ReadKey();

}

}

// Базовый класс Object.

class Program

{

static void Main(string[] args)

{

Object obj1 = new Object();

Object obj2 = new Object();

Console.WriteLine(obj1.Equals(obj2));

obj1 = obj2;

Console.WriteLine(obj1.Equals(obj2));

}

}

// Базовый класс Object.

// Правило: Переопределяйте GetHashCode переопределяя Equals.

class Point : object

{

protected int x, y;

public Point(int xValue, int yValue)

{

x = xValue;

y = yValue;

}

public override bool Equals(Object obj)

{

if(obj == null || this.GetHashCode() != obj.GetHashCode())

{

return false;

}

Point p = (Point)obj;

return (x == p.x) && (y == p.y);

}

public override int GetHashCode()

{

return x ^ y;

}

}

class Program

{

static void Main(string[] args)

{

Point a = new Point(1, 2);

Point b = new Point(1, 2);

Point c = new Point(0, 0);

Console.WriteLine("a = b : {0}", a.Equals(b));

Console.WriteLine("a = c : {0}", a.Equals(c));

Console.ReadKey();

}

}

// Базовый класс Object.

class Program

{

static void Main(string[] args)

{

Object obj1 = new Object();

Object obj2 = new Object();

Console.WriteLine(ReferenceEquals(obj1, obj2));

obj1 = obj2;

Console.WriteLine(ReferenceEquals(obj1, obj2));

Console.ReadKey();

}

}

// Базовый класс Object.

class Program

{

static void Main(string[] args)

{

Object obj1 = new Object();

Object obj2 = new Object();

Console.WriteLine(Equals(obj1, obj2));

obj1 = obj2;

Console.WriteLine(Equals(obj1, obj2));

Console.ReadKey();

}

}

// Базовый класс Object.

// Правило: Переопределяйте GetHashCode переопределяя Equals.

class Point : Object

{

protected int x, y;

public Point(int xValue, int yValue)

{

x = xValue;

y = yValue;

}

public override bool Equals(Object obj)

{

if(obj == null || GetType() != obj.GetType())

{

return false;

}

Point p = (Point)obj;

return (x == p.x) && (y == p.y);

}

public override int GetHashCode()

{

return x ^ y;

}

}

class Program

{

static void Main(string[] args)

{

Point a = new Point(1, 2);

Point b = new Point(1, 2);

Point c = new Point(0, 0);

Console.WriteLine("a == b: {0}", a.Equals(b));

Console.WriteLine("a == c: {0}", a.Equals(c));

Console.WriteLine("a == b: {0}", Equals(a, b));

Console.WriteLine("a == c: {0}", Equals(a, c));

Console.ReadKey();

}

}

// Базовый класс Object.

class Program

{

static void Main(string[] args)

{

Object obj = new Object();

Type type = obj.GetType();

Console.WriteLine(type.ToString());

Console.ReadKey();

}

}

// Базовый класс Object.

class MyBaseClass

{

public static string CompanyName = "Microsoft";

public int age;

public string name;

}

class MyDerivedClass : MyBaseClass

{

static void Main()

{

MyDerivedClass original = new MyDerivedClass();

original.age = 42;

original.name = "Alex";

Console.WriteLine(original.age + " " + original.name + " " + MyDerivedClass.CompanyName);

// Клонирование.

MyDerivedClass clone = original.MemberwiseClone() as MyDerivedClass;

Console.WriteLine(clone.age + " " + clone.name + " " + MyDerivedClass.CompanyName + "\n");

// Проверка.

clone.age = 23;

clone.name = "Konstantin";

MyBaseClass.CompanyName = "CyberBionic Systematics";

Console.WriteLine(original.age + " " + original.name + " " + MyDerivedClass.CompanyName);

Console.WriteLine(clone.age + " " + clone.name + " " + MyDerivedClass.CompanyName);

}

}

// Базовый класс Object.

// Граф наследования клонируется глубоко.

class A { public int a = 1; }

class B : A { public int b = 2; }

class C : B { public int c = 3; }

class X : C { }

class Program : X

{

static void Main(string[] args)

{

Program original = new Program();

Console.WriteLine("Оригинал: " + original.a + " " + original.b + " " + original.c);

// Клонирование объекта.

Program clone = original.MemberwiseClone() as Program;

Console.WriteLine("Клон: " + clone.a + " " + clone.b + " " + clone.c + "\n");

// Проверка на глубокое клонирование.

clone.a = clone.b = clone.c = 7;

Console.WriteLine("Оригинал: " + original.a + " " + original.b + " " + original.c);

Console.WriteLine("Клон: " + clone.a + " " + clone.b + " " + clone.c);

Console.ReadKey();

}

}

// Базовый класс Object.

// Клонирование ассоциации происходнит поверхностно.

class A { public int a = 1; }

class B { public int b = 2; }

class C { public B B = new B(); }

class X

{

public A A = new A();

public C C = new C();

}

class Program : X

{

static void Main(string[] args)

{

Program original = new Program();

Console.WriteLine("Оригинал: " + original.A.a + " " + original.C.B.b);

// Клонирование объекта.

Program clone = original.MemberwiseClone() as Program;

Console.WriteLine("Клон: " + clone.A.a + " " + clone.C.B.b + "\n");

// Проверка на глубокое клонирование.

clone.A.a = clone.C.B.b = 7;

Console.WriteLine("Оригинал: " + original.A.a + " " + original.C.B.b);

Console.WriteLine("Клон: " + clone.A.a + " " + clone.C.B.b);

Console.ReadKey();

}

}

// Глубокое копирование (Deep copy)

public class Point : ICloneable

{

public int x, y;

public Point()

{

}

public Point(int x, int y)

{

this.x = x;

this.y = y;

}

// Реализация метода интерейса ICloneable

public object Clone()

{

return new Point(this.x, this.y) as object;

}

public override string ToString()

{

return "X: " + x + "Y: " + y;

}

}

class Program

{

static void Main(string[] args)

{

Point original = new Point(100, 100);

Point clone = original.Clone() as Point;

Console.WriteLine("Первая проверка");

Console.WriteLine(original);

Console.WriteLine(clone);

// Изменяем clone.x (при этом original.x не изменится)

clone.x = 0;

// Проверка.

Console.WriteLine("Вторая проверка после изменения");

Console.WriteLine(original);

Console.WriteLine(clone);

Console.ReadKey();

}

}

// Клонирование с использованием MemberwiseClone().

// Преимущество клонирования с использованием MemberwiseClone() в том, что

// при клонировании не вызывается конструктор. Оно происходит через

// копирование дампа памяти - тела оригинала.

class MyClass : ICloneable

{

int a, b;

public MyClass()

{

Thread.Sleep(1000);

a = new Random().Next(1, 100);

Thread.Sleep(1000);

b = new Random().Next(1, 100);

}

public object Clone()

{

return this.MemberwiseClone();

}

}

class Program

{

static void Main(string[] args)

{

Stopwatch timer = new Stopwatch();

// Замер времени построения оригинала.

timer.Start();

MyClass original = new MyClass();

timer.Stop();

Console.WriteLine("original построен за {0}", timer.Elapsed.Ticks);

timer.Reset();

// Замер времени построения клона.

timer.Start();

MyClass clone = original.Clone() as MyClass;

timer.Stop();

Console.WriteLine("clone построен за {0}", timer.Elapsed.Ticks);

Console.ReadKey();

}

}

// Клонирование с использованием конструктора.

class MyClass : ICloneable

{

int a, b;

public MyClass()

{

Thread.Sleep(1000);

a = new Random().Next(1, 100);

Thread.Sleep(1000);

b = new Random().Next(1, 100);

}

public object Clone()

{

return new MyClass();

}

}

class Program

{

static void Main(string[] args)

{

Stopwatch timer = new Stopwatch();

// Замер времени построения оригинала.

timer.Start();

MyClass original = new MyClass();

timer.Stop();

Console.WriteLine("original построен за {0}", timer.Elapsed.Ticks);

timer.Reset();

// Замер времени построения клона.

timer.Start();

MyClass clone = original.Clone() as MyClass;

timer.Stop();

Console.WriteLine("clone построен за {0}", timer.Elapsed.Ticks);

Console.ReadKey();

}

}

// Перегрузка операторов.

// Синтаксис перегрузки оператора.

// Point operator + (Point p1, Point p2) - метод, где:

// Point - тип возвращаемого методом значения,

// operator + - имя метода,

// (Point p1, Point p2) - аргументы метода.

public struct Point

{

// Координаты точки.

private int x, y;

public Point(int xPos, int yPos)

{

x = xPos;

y = yPos;

}

// Использовать ключевое слово operator можно только вместе с ключевым словом static!

// Перегрузка оператора +.

public static Point operator +(Point p1, Point p2)

{

return new Point(p1.x + p2.x, p1.y + p2.y);

}

public override string ToString()

{

return string.Format("[{0}, {1}]", this.x, this.y);

}

}

class Program

{

static void Main(string[] args)

{

Point a = new Point(1, 1);

Point b = new Point(2, 2);

Point c = a + b;

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

Console.ReadKey();

}

}

// Использовать ключевое слово operator можно только вместе с ключевым словом static.

public struct Point

{

// Координаты точки.

private int x, y;

public Point(int xPos, int yPos)

{

x = xPos;

y = yPos;

}

public static Point operator +(Point p1, Point p2)

{

return new Point(p1.x + p2.x, p1.y + p2.y);

}

public static Point operator -(Point p1, Point p2)

{

return new Point(p1.x - p2.x, p1.y - p2.y);

}

public static Point operator \*(Point p1, Point p2)

{

return new Point(p1.x \* p2.x, p1.y \* p2.y);

}

public static Point operator /(Point p1, Point p2)

{

return new Point(p1.x / p2.x, p1.y / p2.y);

}

public override string ToString()

{

return string.Format("[{0} {1}]", this.x, this.y);

}

}

class Program

{

static void Main(string[] args)

{

Point a = new Point(1, 1);

Point b = new Point(2, 2);

Console.WriteLine("a + b = {0}", a + b);

Console.WriteLine("a - b = {0}", a - b);

Console.WriteLine("a \* b = {0}", a \* b);

Console.WriteLine("a / b = {0}", a / b);

Point c = new Point(0, 0);

c += a;

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

Console.ReadKey();

}

}

// Перегрузка операторов (унарные операторы ++ / --).

// Использовать ключевое слово operator можно только вместе с ключевым словом static.

public struct Point

{

// Координаты точки.

private int x, y;

public Point(int xPos, int yPos)

{

x = xPos;

y = yPos;

}

// Перегруженный оператор ++.

public static Point operator ++(Point p1)

{

return new Point(p1.x + 1, p1.y + 1);

}

// Перегруженный поератор --.

public static Point operator --(Point p1)

{

return new Point(p1.x - 1, p1.y - 1);

}

public override string ToString()

{

return string.Format("[{0} {1}]", this.x, this.y);

}

}

class Program

{

static void Main(string[] args)

{

Point a = new Point(0, 0);

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

Console.WriteLine("a++ = {0}", a++);

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

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

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

Console.WriteLine("++a = {0}", ++a);

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

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

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

Console.ReadKey();

}

}

// Перегрузка операторов (бинарные операторы).

// Использовать ключевое слово operator можно только вместе с ключевым словом static.

public struct Point

{

// Координаты точки.

private int x, y;

public Point(int xPos, int yPos)

{

x = xPos;

y = yPos;

}

// Перегруженный оператор ==.

public static bool operator ==(Point p1, Point p2)

{

return p1.Equals(p2);

}

// Перегруженный поератор !=.

public static bool operator !=(Point p1, Point p2)

{

return !p1.Equals(p2);

}

public override string ToString()

{

return string.Format("[{0} {1}]", this.x, this.y);

}

// Переопределение Equals()

// В исходной реализации по умолчанию Equals() поддерживает сравенение только ссылочных типов.

// Переопределение его для сравнения структурных типов.

public override bool Equals(object obj)

{

if(obj is Point)

{

if (((Point)obj).x == this.x && ((Point)obj).y == this.y)

{

return true;

}

}

return false;

}

// Переопределение GetHashCode() - обязательно при переопределении Equals().

public override int GetHashCode()

{

return this.ToString().GetHashCode();

}

}

class Program

{

static void Main(string[] args)

{

Point a = new Point(1, 1);

Point b = new Point(2, 2);

Console.WriteLine("a == b = {0}", a == b);

Console.WriteLine("a != b = {0}", a != b);

Console.WriteLine(new string('-', 15));

Point c = new Point(1, 1);

Console.WriteLine("a == c = {0}", a == c);

Console.WriteLine("a != c = {0}", a != c);

Console.ReadKey();

}

}

// Перегрузка операторов.

// Использовать ключевое слово operator можно только вместе с ключевым словом static.

public struct Point : IComparable

{

// Координаты точки.

private int x, y;

public Point(int xPos, int yPos)

{

x = xPos;

y = yPos;

}

public static bool operator <(Point p1, Point p2)

{

return (p1.CompareTo(p2) < 0);

}

public static bool operator >(Point p1, Point p2)

{

return (p1.CompareTo(p2) > 0);

}

public static bool operator <=(Point p1, Point p2)

{

return (p1.CompareTo(p2) <= 0);

}

public static bool operator >=(Point p1, Point p2)

{

return (p1.CompareTo(p2) >= 0);

}

// Реализация интерфейса IComparable.

public int CompareTo(object obj)

{

if (obj is Point)

{

Point p = (Point)obj;

if (this.x > p.x && this.y > p.y)

{

return 1;

}

else if (this.x < p.x && this.y < p.y)

{

return -1;

}

else

{

return 0;

}

}

else

{

throw new ArgumentException();

}

}

}

class Program

{

static void Main(string[] args)

{

Point a = new Point(1, 1);

Point b = new Point(2, 2);

Console.WriteLine("a < b = {0}", a < b);

Console.WriteLine("a > b = {0}", a > b);

Console.WriteLine("a <= b = {0}", a <= b);

Console.WriteLine("a >= b = {0}", a >= b);

Console.WriteLine(new string('-', 15));

Point c = new Point(1, 1);

Console.WriteLine("a <= c = {0}", a <= c);

Console.WriteLine("a >= c = {0}", a >= c);

Console.ReadKey();

}

}

// Перегрузка операторов.

// Использовать ключевое слово operator можно только вместе с ключевым словом static.

public struct Point

{

// Координаты точки.

private int x, y;

public Point(int xPos, int yPos)

{

x = xPos;

y = yPos;

}

public static Point operator +(Point p1, Point p2)

{

return new Point(p1.x + p1.y, p2.x + p2.y);

}

public static Point operator -(Point p1, Point p2)

{

return new Point(p1.x - p1.y, p2.x - p2.y);

}

public static Point operator \*(Point p1, Point p2)

{

return new Point(p1.x \* p1.y, p2.x \* p2.y);

}

public static Point operator /(Point p1, Point p2)

{

return new Point(p1.x / p1.y, p2.x / p2.y);

}

// Методы.

public override string ToString()

{

return string.Format("[{0} {1}]", this.x, this.y);

}

// Operator + через Add().

public static Point Add(Point p1, Point p2)

{

return p1 + p2;

}

// Operator - через Substract.

public static Point Substract(Point p1, Point p2)

{

return p1 - p2;

}

}

class Program

{

static void Main(string[] args)

{

Point a = new Point(1, 1);

Point b = new Point(2, 2);

Console.WriteLine("Point.Add(a, b) = {0}", Point.Add(a, b));

Console.WriteLine("Point.Substract(a, b) = {0}", Point.Substract(a, b));

Console.ReadKey();

}

}

// Перегрузка оператора явного преобразования типа.

// Ключевое слово explicit служит для создания оператора явного преобразования типа.

struct Digit

{

public byte value;

// Конструктор.

Digit(byte value)

{

this.value = value;

}

// Оператор явного преобразования типа byte-to-Digit.

public static explicit operator Digit(byte argument)

{

Digit digit = new Digit(argument);

return digit;

}

public override string ToString()

{

return this.value.ToString();

}

}

class Program

{

static void Main(string[] args)

{

byte variable = 1;

// Явное преобразование byte-to-Digit.

Digit digit = (Digit)variable;

Console.WriteLine(digit);

Console.ReadKey();

}

}

// Перегрузка оператора неявного преобразования типа.

// Ключевое слово implicit служит для создания оператора неявного преобразования типа.

struct Digit

{

public byte value;

// Конструктор.

Digit(byte value)

{

this.value = value;

}

// Оператор неявного преобразования типа byte-to-Digit.

public static implicit operator Digit(byte argument)

{

Digit digit = new Digit(argument);

return digit;

}

public override string ToString()

{

return this.value.ToString();

}

}

class Program

{

static void Main(string[] args)

{

byte variable = 1;

// Неявное преобразование byte-to-Digit.

Digit digit = variable;

Console.WriteLine(digit);

Console.ReadKey();

}

}

## Анонимные и динамические типы. LINQ

// Анонимные типы.

// Анонимные типы предлагают удобный способ инкапсулирования набора свойств в один объект

// без необходимости предвирательного явного определения типа.

// Имя типа создается компилятором и недоступно на уровне исходного кода.

// Анонимные типы являются ссылочными типами, которые происходят непосредственно от класса object.

// Компилятор присваивает им имена, несмотря на то что эти имена недоступны для приложения.

class Program

{

static void Main(string[] args)

{

var instance = new { Name = "Alex", Age = 27 };

// instance.Name = "XXX"; // Свойства только для чтения.

Console.WriteLine("Name = {0}, Age = {1}", instance.Name, instance.Age);

Type type = instance.GetType();

Console.WriteLine(type.ToString());

Console.ReadKey();

}

}

// Анонимные типы.

class Program

{

static void Main(string[] args)

{

// Анонимные типы в анонимных типах.

var instance = new { Name = "Alex", Age = 27, Id = new { Number = 123 } };

Console.WriteLine("Name = {0}, Age = {1}, Id = {2}", instance.Name, instance.Age, instance.Id.Number);

Console.ReadKey();

}

}

// Анонимные типы (сильная ссылка).

class MyClass

{

public int field;

public void Method()

{

Console.WriteLine(field);

}

}

class Program

{

static void Main(string[] args)

{

var instance = new { My = new MyClass() };

instance.My.field = 1;

instance.My.Method();

Console.ReadKey();

}

}

// Анонимные типы (слабая ссылка).

class MyClass

{

public int field;

public void Method()

{

Console.WriteLine(field);

}

}

class Program

{

static void Main(string[] args)

{

new

{

My = new MyClass { field = 1 }

}.My.Method();

Console.ReadKey();

}

}

// Анонимные типы.

delegate void MyDelegate(string @string);

class Program

{

static void Main(string[] args)

{

var instance = new

{

MyDel = new MyDelegate((string @string) => Console.WriteLine(@string))

};

instance.MyDel("Hello world!");

Console.ReadKey();

}

}

// LINQ.

public class Employee

{

public string FirstName { get; set; }

public string LastName { get; set; }

public decimal Salary { get; set; }

public DateTime StartDate { get; set; }

}

class Program

{

static void Main(string[] args)

{

// База данных сотрудников.

var employees = new List<Employee>

{

new Employee

{

FirstName = "Ivan",

LastName = "Ivanov",

Salary = 94000,

StartDate = DateTime.Parse("1/4/1992")

},

new Employee

{

FirstName = "Petr",

LastName = "Petrov",

Salary = 123000,

StartDate = DateTime.Parse("12/3/1985")

},

new Employee

{

FirstName = "Andrew",

LastName = "Andreev",

Salary = 1000000,

StartDate = DateTime.Parse("1/12/2005")

}

};

// Выражение запроса.

var query = // query - переменная запроса.

from employee in employees // from - объявляет переменную диапазона employee.

where employee.Salary > 100000

orderby employee.LastName, employee.FirstName

select new

{

LastName = employee.LastName,

FirstName = employee.FirstName

};

Console.WriteLine("Высокооплачиваемые сотрудники: ");

foreach(var item in query)

Console.WriteLine("{0} {1}", item.LastName, item.FirstName);

Console.ReadKey();

}

}

// Динамические типы данных (локальные переменные).

class Program

{

static void Main(string[] args)

{

dynamic variable = 1;

Console.WriteLine(variable);

variable = "Hello world!";

Console.WriteLine(variable);

variable = DateTime.Now;

Console.WriteLine(variable);

Console.ReadKey();

}

}

// Динамические типы данных (статические поля).

class Program

{

static dynamic field = 1;

static void Main(string[] args)

{

Console.WriteLine(field);

field = "Hello world";

Console.WriteLine(field);

field = DateTime.Now;

Console.WriteLine(field);

Console.ReadKey();

}

}

// Динамические типы данных (нестатические поля).

class Program

{

dynamic field = 1, field2 = "Hello", field3 = true;

static void Main(string[] args)

{

dynamic instance = new Program();

Console.WriteLine(instance.field);

instance.field = "Hello world!";

Console.WriteLine(instance.field);

instance.field = DateTime.Now;

Console.WriteLine(instance.field);

Console.ReadKey();

}

}

// Динамические типы данных.

class Program

{

static void Main(string[] args)

{

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

{

Console.WriteLine("Hello world!");

}

Console.ReadKey();

}

}

// Динамические типы данных (динамические типы аргументов и возвращаемых значений методов).

class Program

{

static dynamic Method(dynamic argument)

{

return "Hello" + argument + "!";

}

static void Main(string[] args)

{

string @string = Method("friend");

Console.WriteLine(@string);

Console.ReadKey();

}

}

// Динамические типы данных.

class MyClass

{

dynamic field;

public MyClass(dynamic argument)

{

field = argument;

}

// Динамические свойства.

public dynamic MyAutoProperty { get; set; }

public dynamic Field

{

get { return field; }

set { field = value; }

}

// Динамические методы.

public dynamic Method(dynamic argument)

{

return argument;

}

// Динамические массивы и индексаторы.

dynamic[] array = new dynamic[3];

public dynamic this[dynamic index]

{

get { return array[index]; }

set { array[index] = value; }

}

}

class Program

{

static void Main(string[] args)

{

dynamic my = new MyClass("Hello");

Console.WriteLine(my.Field);

string variable = my.Method("World");

Console.WriteLine(variable);

my[0] = "Zero";

my[1] = "One";

my[2] = "Two";

for(dynamic i = 0; i < 3; i++)

{

Console.WriteLine(my[i]);

}

Console.ReadKey();

}

}

// Динамические типы данных (динамичекие типы в делегатах).

delegate dynamic MyDelegate(dynamic argument);

class Program

{

static dynamic Method(dynamic argument)

{

return argument;

}

static void Main(string[] args)

{

dynamic myDelegate = new MyDelegate(Method);

dynamic @string = myDelegate.Invoke("Hello world!");

Console.WriteLine(@string);

Console.ReadKey();

}

}

// Динамические типы данных (динамические типы в параметризированных делегатах).

delegate R MyDelegate<T, R>(T argument);

class Program

{

static dynamic Method(dynamic argument)

{

return argument;

}

static void Main(string[] args)

{

dynamic myDelegate = new MyDelegate<dynamic, dynamic>(Method);

dynamic @string = myDelegate("Hello world!");

Console.WriteLine(@string);

Console.ReadKey();

}

}

// Динамические типы данных (события).

delegate dynamic MyDelegate(dynamic sender, dynamic e);

class MyClass

{

dynamic myEvent;

public event MyDelegate MyEvent

{

add { myEvent += value; }

remove { myEvent -= value; }

}

public dynamic Method(dynamic sender, dynamic e)

{

myEvent.Invoke(sender, e);

return default(dynamic);

}

}

class Program

{

static dynamic Handler(dynamic sender, dynamic e)

{

Console.WriteLine("sender = {0}, e = {1}", sender, e);

return default(dynamic);

}

static void Main(string[] args)

{

}

}

// Динамические типы данных (события).

delegate dynamic MyDelegate(dynamic sender, dynamic e);

class MyClass

{

public dynamic MyEvent = default(dynamic); // Не является событием.

public dynamic Method(dynamic sender, dynamic e)

{

MyEvent.Invoke(sender, e);

return default(dynamic);

}

}

class Program

{

static dynamic Handler(dynamic sender, dynamic e)

{

Console.WriteLine("sender = {0}, e = {1}", sender, e);

return default(dynamic);

}

static void Main(string[] args)

{

dynamic my = new MyClass();

my.MyEvent += new MyDelegate(Handler);

my.Method("user", "mouse");

my.MyEvent.Invoke("user", "mouse");

Console.ReadKey();

}

}

}

// Динамические типы данных (наследование).

class Base

{

public dynamic field;

public dynamic Method()

{

return this.field;

}

}

class Derived : Base { }

class Program

{

static void Main(string[] args)

{

dynamic instance = new Derived();

// Динамические поля должны быть проинициализированы перед использованием.

instance.field = "Hello!";

Console.WriteLine(instance.Method());

Console.ReadKey();

}

}

// Динамические типы данных.

class MyClass { }

class Program

{

static dynamic FactoryMethod()

{

return new MyClass();

}

static void Main(string[] args)

{

dynamic instance = FactoryMethod() as dynamic;

if (false)

{

instance.Method(7); // Вызов метода.

instance.field = instance.Property; // Получение и установка значений полей и свойств.

instance["one"] = instance[2]; // Получение и установка значений через индексаторы.

dynamic variable = instance + 3; // Вызов операторов.

variable = instance(5, 7); // Вызов делегатов.

}

Console.ReadKey();

}

}

// Динамические типы данных (анонимные типы).

class Program

{

static void Main(string[] args)

{

dynamic instance = new { Name = "Alex", Age = 18 };

Console.WriteLine(instance.Name);

Console.WriteLine(instance.Age);

Console.ReadKey();

}

}

// Динамические типы данных (динамические типы аргументов и возвращаемых значений методов).

class Program

{

static dynamic Method(ref dynamic argument1, out dynamic argument2)

{

argument1 = ++argument1;

argument2 = 2;

return default(dynamic);

}

static void Main(string[] args)

{

dynamic variable1 = 0, variable2;

Method(ref variable1, out variable2);

Console.WriteLine(variable1);

Console.WriteLine(variable2);

Console.ReadKey();

}

}

// Динамические типы данных.

struct Point

{

dynamic x, y;

public Point(dynamic x, dynamic y)

{

this.x = x;

this.y = y;

}

// Один из параметров бинарного оператора должен иметь существующий тип.

// public static dynamic operator +(dynamic pointA, dynamic pointB) - так недопустимо.

public static dynamic operator +(Point pointA, dynamic pointB)

{

return new Point(pointA.x + pointB.x, pointA.y + pointB.y);

}

// В унарных операторах недопустимо использование динамических типов (вообще).

// public static dynamic operator ++(dynamic pointA) - так недопустимо.

public static Point operator ++(Point pointA)

{

return new Point(pointA.x + 1, pointA.y + 1);

}

public override string ToString()

{

return string.Format("x = {0}, y = {1}", x, y);

}

}

class Program

{

static void Main(string[] args)

{

}

}

// Динамические типы данных.

class Calculator

{

public dynamic Add(dynamic a, dynamic b)

{

return a + b;

}

}

class Program

{

static void Main(string[] args)

{

dynamic calculator = new Calculator();

Console.WriteLine(calculator.Add(2, 3));

Console.WriteLine(calculator.Add("1", 2));

Console.ReadKey();

}

}

## Пространства имен. Директивы препроцессора

// Пространства имен.

// Директива using импортирует пространство имен, избавляя от необходимости полной квалификации имен стереотипов.

using System;

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Hello world!");

}

}

// Пространства имен.

namespace Space1

{

class MyClass

{

public MyClass()

{

Console.WriteLine("Constructor - MyClass");

}

}

}

namespace Пространства\_имен.Пример\_4

{

class Program

{

static void Main(string[] args)

{

// Полная квалификация имени класса.

// Включает имя пространства имен, в котором находится класс.

Space1.MyClass my = new Space1.MyClass();

}

}

}

// Пространства имен.

namespace NamespaceA

{

namespace NamespaceB

{

namespace NamespaceC

{

class MyClass

{

public MyClass()

{

System.Console.WriteLine(this.GetType().Name);

}

}

}

}

}

namespace Пространства\_имен.Пример\_5

{

using NamespaceA.NamespaceB.NamespaceC;

class Program

{

static void Main(string[] args)

{

MyClass my = new MyClass();

}

}

}

// Пространства имен. Псевдонимы типов.

namespace NamespaceA

{

namespace NamespaceB

{

namespace NameSpaceC

{

class MyClassC

{

public void Method()

{

System.Console.WriteLine("Hello world");

}

}

}

class MyClassB { }

}

class MyClassA { }

}

namespace Пространства\_имен.Пример\_6

{

// Создание псевдонима MyClass для класса MyClassC из пространства имен NamespaceA.NamespaceB.NamespaceC.

using MyClass = NamespaceA.NamespaceB.NameSpaceC.MyClassC;

class Program

{

static void Main(string[] args)

{

MyClass my = new MyClass();

my.Method();

System.Console.WriteLine(my.GetType());

}

}

}

// Пространства имен.

using NameSpaceA;

using NameSpaceA.NameSpaceB;

using NameSpaceA.NameSpaceB.NameSpaceC;

namespace NameSpaceA

{

namespace NameSpaceB

{

namespace NameSpaceC

{

class MyClassC { }

}

class MyClassB { }

}

class MyClassA { }

}

// В данном месте импорт недопустим.

// Импорт допустим или в теле namespace, или перед всеми другими

// пространствами имен.

// using NameSpaceA;

// using NameSpaceA.NameSpaceB;

// using NameSpaceA.NameSpaceB.NameSpaceC; - здесь нельзя!

namespace Пространства\_имен.Пример\_7

{

// using NameSpaceA;

// using NameSpaceA.NameSpaceB;

// using NameSpaceA.NameSpaceB.NameSpaceC; - здесь можно!

class Program

{

static void Main(string[] args)

{

MyClassA myA = new MyClassA();

MyClassB myB = new MyClassB();

MyClassC myC = new MyClassC();

}

}

}

// Пространства имен.

// Видимость во вложенных пространствах имен при отсутствии импорта.

namespace NamespaceA

{

namespace NamespaceB

{

namespace NamespaceC

{

class MyClassC

{

MyClassA myA = new MyClassA();

MyClassB myB = new MyClassB();

MyClassC myC = new MyClassC();

}

}

class MyClassB

{

MyClassA myA = new MyClassA();

MyClassB myB = new MyClassB();

// MyClassC myC = new MyClassC();

}

}

class MyClassA

{

MyClassA myA = new MyClassA();

// MyClassB myB = new MyClassB();

// MyClassC myC = new MyClassC();

}

}

namespace Пространства\_имен.Пример\_8

{

class Program

{

static void Main(string[] args)

{

}

}

}

// Пространства имен.

namespace NamespaceA

{

class MyClassA

{

MyClassA my;

public MyClassA() // StackOverflowException

{

System.Console.WriteLine("Constructor MyClassA");

// Рефлексивная самоассоциация после первой попытки инстанцирования

// приводит к цикличному инстанцированию.

my = new MyClassA();

}

}

}

namespace Пространства\_имен.Пример\_9

{

using NamespaceA;

class Program

{

static void Main(string[] args)

{

// Попытка инстанцирования класса MyClassA.

MyClassA myA = new MyClassA();

}

}

}

// Пространства имен.

// Внимание! Распространенные ошибки рефлексивной самоассоциации после

// первой попытки инстанцирования.

using NamespaceA;

using NamespaceA.NamespaceB;

using NamespaceA.NamespaceB.NamespaceC;

namespace NamespaceA

{

namespace NamespaceB

{

namespace NamespaceC

{

class MyClassC

{

MyClassA myA = new MyClassA(); // StackOverflowException 3

MyClassB myB = new MyClassB(); // StackOverflowException 6

MyClassC myC = new MyClassC(); // StackOverflowException 7

}

}

class MyClassB

{

MyClassA myA = new MyClassA(); // StackOverflowException 4

MyClassB myB = new MyClassB(); // StackOverflowException 5

MyClassC myC = new MyClassC();

}

}

class MyClassA

{

MyClassA myA = new MyClassA(); // StackOverflowException 1

MyClassB myB = new MyClassB(); // StackOverflowException 2

MyClassC myC = new MyClassC();

}

}

namespace Пространства\_имен.Пример\_10

{

class Program

{

static void Main(string[] args)

{

MyClassA myA = new MyClassA();

MyClassB myB = new MyClassB();

MyClassC myC = new MyClassC();

}

}

// Пространства имен.

// Технически допускается создание нескольких пространств имен с одним именем.

// Логически несколько одноименных пространств имен объединяются в одно пространство имен.

namespace MyNamespace

{

class MyClass { }

}

namespace MyNamespace

{

// Недопустимо иметь в одноименных пространствах имен одноименные типы.

// class MyClass { } // Ошибка.

}

namespace MyNamespace

{

class Program

{

static void Main(string[] args)

{

MyNamespace.MyClass my = new MyNamespace.MyClass();

}

}

}

// Пространства имен.

namespace MyNamespace

{

class MyClass { }

namespace MyNamespace

{

// Допустимо иметь во вложенных пространствах имен одноименные стереотипы.

class MyClass { }

}

}

namespace Пространства\_имен.\_Пример\_12

{

class Program

{

static void Main(string[] args)

{

MyNamespace.MyClass my1 = new MyNamespace.MyClass();

MyNamespace.MyNamespace.MyClass my2 = new MyNamespace.MyNamespace.MyClass();

}

}

}

// Пространства имен.

namespace MyNamespace

{

class MyClass { }

namespace MyNamespace

{

// Допустимо иметь в вложенных пространствах имен одноименные стереотипы.

class MyClass { }

}

}

namespace Пространства\_имен.Пример\_13

{

using MyNamespace;

using MyNamespace.MyNamespace;

class Program

{

static void Main(string[] args)

{

// Если несколько импортируемых пространств имен содержат одноименные стереотипы,

// то инстанцирование требует полной квалификации имени стереотипа.

// MyClass my = new MyClass(); // Ошибка

MyNamespace.MyClass my1 = new MyNamespace.MyClass();

MyNamespace.MyNamespace.MyClass my2 = new MyNamespace.MyNamespace.MyClass();

}

}

}

// Пространства имен.

// Два пространства имен одного уровня вложенности не предоставляют доступа одно другому,

// к своим стереотипам без импорта.

namespace MyNamespace

{

namespace NamespaceA

{

class MyClassA

{

// MyClassB my = new MyClassB();

}

}

namespace NamespaceB

{

class MyClassB

{

// MyClassA my = new MyClassA();

}

}

}

namespace Пространства\_имен.Пример\_14

{

using MyNamespace.NamespaceA;

using MyNamespace.NamespaceB;

class Program

{

static void Main(string[] args)

{

MyClassA myA = new MyClassA();

MyClassB myB = new MyClassB();

}

}

}

// Пространства имен.

// Два пространства имен одного уровня вложенности не предоставляют доступа одно другому,

// к своим стереотипам без импорта.

namespace MyNamespace

{

namespace NamespaceA

{

class BaseClass { }

}

namespace NamespaceB

{

class DerivedClass : NamespaceA.BaseClass { }

}

}

namespace Пространства\_имен.Пример\_15

{

using MyNamespace.NamespaceA;

using MyNamespace.NamespaceB;

class Program

{

static void Main(string[] args)

{

BaseClass instance1 = new BaseClass();

DerivedClass instance2 = new DerivedClass();

}

}

}

// Пространства имен. Сокрытие имен стереотипов.

// В случае наличия одноименных стереотипов во вложенном и во внешнем пространствах имен

// происходит сокрытие имени стереотипа внешнего пространства имен.

// Обращение к одноименному стереотипу внешнего пространства имен требует полной

// квалификации имени стереотипа.

namespace NamespaceA

{

namespace NamespaceB

{

class MyClass { } // Одноименный стереотип.

class MyClassB

{

MyClass my1;

NamespaceA.MyClass my2; // Прямая видимость отсутствует.

MyClassA my3; // Прямая видимость имеется.

}

}

class MyClass { } // Одноименный стереотип.

class MyClassA { }

}

namespace Пространства\_имен.Пример\_16

{

using NamespaceA;

using NamespaceA.NamespaceB;

class Program

{

static void Main(string[] args)

{

}

}

}

// Пространства имен.

namespace NamespaceA

{

namespace NamespaceB

{

namespace NamespaceC

{

class MyClassC { }

}

class MyClassB { }

}

class MyClassA { }

}

// Повторное создание вложенного пространства имен.

namespace NamespaceA.NamespaceB.NamespaceC

{

class MyClassC2 { }

}

namespace Пространства\_имен.Пример\_17

{

using NamespaceA.NamespaceB.NamespaceC;

class Program

{

static void Main(string[] args)

{

MyClassC myC = new MyClassC();

MyClassC2 myC2 = new MyClassC2();

}

}

}

// Пространства имен.

namespace NamespaceA

{

namespace NamespaceB

{

namespace NamespaceC

{

class MyClassC { }

}

class MyClassB { }

}

class MyClassA { }

}

// Повторное создание вложенного пространства имен.

namespace NamespaceA.NamespaceB.NamespaceC

{

class MyClassC2

{

MyClassA myA = new MyClassA();

MyClassB myB = new MyClassB();

MyClassC myC = new MyClassC();

}

}

namespace Пространства\_имен.Пример\_18

{

using NamespaceA.NamespaceB.NamespaceC;

class Program

{

static void Main(string[] args)

{

}

}

}

// Пространства имен.

namespace NamespaceA

{

namespace NamespaceB

{

namespace NamespaceC

{

class MyClassC

{

MyClassA myA = new MyClassA();

MyClassB myB = new MyClassB();

MyClassB2 myB2 = new MyClassB2();

}

}

class MyClassB { }

}

class MyClassA { }

}

// Повторное создание вложенного пространства имен.

namespace NamespaceA.NamespaceB

{

class MyClassB2 { }

}

namespace Пространства\_имен.Пример\_19

{

using NamespaceA.NamespaceB;

class Program

{

static void Main(string[] args)

{

}

}

}

// Пространства имен.

// В случае отсутствия импорта пространства имен System полные имена

// базовых типов оказываются недоступными.

// Доступными оказываются лишь псевдонимы типов.

// using System;

namespace Пространства\_имен.Пример\_20

{

class Program

{

static void Main(string[] args)

{

// Int32 a = 1;

int b = 2;

}

}

}

// Создание DLL библиотек.

public class Class1

{

/// <summary>

/// This is my Method from Class1 for...

/// </summary>

public void Method()

{

Console.WriteLine("Hello from ClassLibrary 1");

}

}

public class Class2

{

/// <summary>

/// This is my Method from Class2 for...

/// </summary>

public void Method()

{

Console.WriteLine("Hello from ClassLibrary 2");

}

}

class Program

{

static void Main(string[] args)

{

Class1 instance = new Class1();

instance.Method();

Class2 inst = new Class2();

inst.Method();

Console.ReadKey();

}

}

namespace Library

{

public class MyClass

{

public void Method()

{

Console.WriteLine("Method from Library 2");

}

}

}

namespace Library

{

public class MyClass

{

public void Method()

{

Console.WriteLine("Method from Library 1");

}

}

}

// Ситуация:

// Совпадение имен двух пространств имен в разных сборках и совпадение имен стереотипов в этих пространствах имен.

// Для подключения алиаса необходимо выполнить следующие действия:

// 1. Добавьте в References необходимые сборки Library1 и Library2.

// 2. Откройте папку References.

// 3. Правой кнопкой мыши кликните по сборке Library1, откроется контекстное меню,

// в котором выберите пункт Properties.

// 4. В открывшемся окне свойств в свойсте Alliases замените значение global на L1.

// Те же действия выполните для сборки Library2.

extern alias L1;

extern alias L2;

namespace Пространства\_имен.Библиотеки\_DLL

{

class Program

{

static void Main(string[] args)

{

L1.Library.MyClass my1 = new L1.Library.MyClass();

L2.Library.MyClass my2 = new L2.Library.MyClass();

my1.Method();

my2.Method();

System.Console.WriteLine();

}

}

}

namespace LibraryA

{

// Ключевое слово internal является модификатором доступа для типов и членов типов.

// Внутренние (internal) типы или члены доступны только внутри этой сборки.

public class MyPublicClass

{

public void PublicMethod()

{

Console.WriteLine("PublicMethod");

}

// internal - доступ к типу или члену возможен из любого кода в этой сборке,

// но не из другой сборки.

internal void InternalMethod()

{

Console.WriteLine("InternalMethod");

}

// internal protected - доступ к типу или члену возможен из любого кода в этой сборке

// или из производного класса в другой сборке.

internal protected void InternalProtectedMethod()

{

Console.WriteLine("InternalProtectedMethod");

}

}

}

namespace LibraryA

{

// internal - доступ к типу или члену возможен из любого кода в той же сборке,

// но не из другой сборки.

internal class MyInternalClass

{

public MyInternalClass()

{

Console.WriteLine("Constructor - MyInternalClassA");

}

}

}

namespace Пространства\_имен.Internal

{

// Возвожмость обращения к internal protected методу базового класса

// из другой сборки.

class MyClass : MyPublicClass

{

public void MyMethod()

{

this.InternalProtectedMethod();

}

}

}

namespace Пространства\_имен.Internal

{

class Program

{

static void Main(string[] args)

{

// 1.

MyPublicClass instanceA = new MyPublicClass();

instanceA.PublicMethod();

// instanceA.InternalMethod(); // Недоступен.

// instanceA.InternalProtectedMethod(); // Недоступен.

// MyInternalClass instance = new MyInternalClass(); // Недоступен.

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

// 2.

MyClass instance = new MyClass();

instance.PublicMethod();

instance.MyMethod(); // Вызов InternalProtectedMethod().

// instance.InternalMethod(); // Недоступен.

Console.ReadKey();

}

}

}

namespace LibraryA

{

// Класс не может быть internal protected.

// internal protected class MyInternalProtectedClass

// {

// }

}

namespace Пространства\_имен.Internal\_2

{

class MyClass // : MyInternalProtectedCLass // Недоступен.

{

}

}

namespace Пространства\_имен.Internal\_2

{

class Program

{

static void Main(string[] args)

{

MyClass instance = new MyClass();

Console.ReadKey();

}

}

}

namespace Пространства\_имен.Директивы\_препроцессора.П1

{

class Program

{

static void Main(string[] args)

{

#region MyRegion

Console.WriteLine("Hello");

#endregion

Console.ReadKey();

}

}

}

// #define DEBUG

namespace Пространства\_имен.Директивы\_препроцессора.П2

{

class Program

{

static void Main(string[] args)

{

#if DEBUG

Console.WriteLine("Debug version");

#endif

// TODO: Посмотрите в Task List.

// HACK: Посмотрите в Task List.

Console.WriteLine("Release version");

Console.ReadKey();

}

}

}

// #define DEBUG

#define VC\_V7

using System;

namespace Пространства\_имен.Директивы\_препроцессора.П3

{

// Директивы препроцессора.

class Program

{

static void Main(string[] args)

{

#if(DEBUG && !VC\_V7)

Console.WriteLine("DEBUG is defined");

#elif (!DEBUG && VC\_V7)

Console.WriteLine("VC\_V7 is defined");

#elif (DEBUG && VC\_V7)

Console.WriteLine("DEBUG and VC\_V7 are defined");

#else

Console.WriteLine("DEBUG and VC\_V7 aren't defined");

#endif

Console.ReadKey();

}

}

}

namespace Пространства\_имен.Директивы\_препроцессора.П4

{

class Program

{

static void Main(string[] args)

{

// 1.

#line 123 // В окне ошибок показывает строку 123

Console.WriteLine(1); // Снять комментарий (строка с ошибкой).

// Console.WriteLine();

// 2.

Console.WriteLine("Строка #1.");

#line hidden // Скрывает строку от отладчика.

Console.WriteLine("Скрытая строка."); // Установить BreakPoint.

#line default

Console.WriteLine("Строка #2.");

Console.ReadKey();

}

}

}

namespace Пространства\_имен.Директивы\_препроцессора.П5

{

class Program

{

static void Main(string[] args)

{

// #warning позволяет создавать предупреждение первого уровня из опредлеенного

// места в коде.

// Посмотреть окно Ошибок, вкладка - Предупреждения.

#warning Пользовательское предупреждение.

Console.WriteLine("Hello world");

Console.ReadKey();

}

}

}

namespace Пространства\_имен.Директивы\_препроцессора.П6

{

class Program

{

static void Main(string[] args)

{

// #error - позволяет создавать ошибку первого уровня из определенного

// места в коде.

//#error Ошибка, определенная пользователем. // Снять комментарий.

Console.WriteLine(1);

Console.ReadKey();

}

}

}