**Платформа Microsoft .Net Framework**

Совокупность средств, с помощью которых программы пишутся, корректируются, преобразуются в машинные коды, отлаживаются и запускаются, называют *средой разработки* или *оболочкой*. *Платформа .Net* или *.Net Framework* – это больше чем просто *среда разработки* программ, это *объединение* ранее разрозненных технологий компанией Microsoft, которые позволяют разрабатывать разнотипные приложения на различных языках программирования под различные операционные системы.

.*NET* Framework является надстройкой над операционной системой, в качестве которой может выступать любая версия *Windows*, *Linux* и вообще любая ОС (по заверению разработчиков), и состоит из ряда компонентов. Так, .*NET* Framework включает в себя:

1. Официальные языки: С#, VB.NET, Managed C++, JScript .NET и др.
2. Общеязыковую объектно-ориентированную среду выполнения CLR (Common Language Runtime), совместно используемую этими языками для создания приложений.
3. Ряд связанных между собой библиотек классов под общим именем *FCL* (Framework *Class Library*).

Основным компонентом платформы .*NET* Framework является общеязыковая среда выполнения программ *CLR*. Название среды – "общеязыковая среда выполнения" - говорит само за себя: это исполняющая среда, которая подходит для различных языков программирования. К функциям *CLR* относятся:

1. двухшаговая компиляция: преобразование программы, написанной на одном из языков программирования в управляемый код на *промежуточном языке* (*Microsoft Intermediate Language*, *MSIL*, или просто IL), а затем преобразование IL-кода в машинный код конкретного процессора, который выполняется с помощью виртуальной машины или JIT-компилятора (Just In *Time compiler* - компилирование точно к нужному моменту);
2. управление кодом: загрузка и выполнение уже готового IL-кода с помощью JIT-компилятора;
3. осуществление доступа к метаданным с целью проверки *безопасности кода*;
4. управление памятью при размещении объектов с помощью сборщика мусора (*Garbage Collector*);
5. обработка исключений и исключительных ситуаций, включая межъязыковые исключения;
6. осуществление взаимодействия между управляемым кодом (код, созданный для СLR) и неуправляемым кодом;
7. поддержка сервисов для разработки разнотипных приложений.

Следующим компонентом .*Net* Framework является *FCL* – библиотека классов платформы. Эта библиотека разбита на несколько модулей таким образом, что имеется возможность использовать ту или иную ее часть в зависимости от требуемых результатов. Так, например, в одном из модулей содержатся "кирпичики", из которых можно построить *Windows*-приложения, в другом — "кирпичики", необходимые для организации работы в сети и т.д.

Часть *FCL* посвящена описанию *базисных типов*. Тип — это способ представления данных; *определение* наиболее фундаментальных из них облегчает совместное использование языков программирования с помощью .*NET* Framework. Все вместе это называется *Common Type System* (*CTS* — единая *система типов*).

Кроме того, *библиотека FCL* включает в себя Common *Language* *Specification* (*CLS* – общая языковая спецификация), которая устанавливает: основные правила языковой интеграции. Спецификация *CLS* определяет минимальные требования, предъявляемые к языку платформы .*NET*. Компиляторы, удовлетворяющие этой спецификации, создают объекты, способные взаимодействовать друг с другом. Поэтому любой язык, соответствующий требованиям *CLS*, может использовать все возможности библиотеки *FCL*.

Как уже отмечалось, основными языками, предназначенными для платформы .*NET* Framework, являются С#, VB.*NET*, Managed C++ и JScript .*NET*. Для данных языков Microsoft предлагает собственные компиляторы, переводящие программу в IL-код, который выполняется JIT-компилятором среды *CLR*. Кроме Microsoft, еще несколько компаний и академических организаций создали свои собственные компиляторы, генерирующие код, работающий в *CLR*. На сегодняшний момент известны компиляторы для *Pascal*, Cobol, *Lisp*, Perl, Prolog и т.д. Это означает, что можно написать программу, например, на языке *Pascal*, а затем, воспользовавшись соответствующим компилятором, создать *управляемый код*, который будет работать в среде *CLR*.

#### **Понятия приложения, проекта, решения**

.NET Framework не налагает никаких ограничений на возможные типы создаваемых приложений. Тем не менее, давайте рассмотрим некоторые наиболее часто встречающиеся типы приложений:

1. Консольные приложения позволяют выполнять вывод на "консоль", то есть в окно командного процессора.
2. Windows-приложения, использующие элементы интерфейса Windows, включая формы, кнопки, флажки и т.д.
3. Web-приложения представляют собой web-страницы, которые могут просматриваться любым web-браузером.
4. Web-сервисы представляют собой распределенные приложения, которые позволяют обмениваться по Интернету практически любыми данными с использованием единого синтаксиса независимо от того, какой язык программирования применялся при создании web-службы и на какой системе она размещена.

Приложение, находящееся в процессе разработки, называется проектом. Несколько приложений могут быть объединены в решение (solution).

Удобной средой разработки приложений является Visual Studio .Net.

### 

**Лекция 1: Технология объектно-ориентированного программирования**

В основе языка С# лежит технология объектно-ориентированного программирования (*ООП*). Все программы на языке С# в большей или меньшей степени являются объектно-ориентированными.

*ООП* основано на таких понятиях как "*класс*", "*объект*", "*интерфейс*", "*инкапсуляция*", "*наследование*", "*полиморфизм*", "событие".

*Объект* в программе *—* это *абстракция* реального объекта. *Объект* обладает атрибутами, поведением и индивидуальностью. Атрибуты определяют основные черты объекта, поведение — действия над объектом, индивидуальность — отличие одного объекта от другого с такими же атрибутами по их конкретным значениям.

*Класс* – это множество объектов с одинаковыми атрибутами и поведением, представляемое в языке программирования в виде абстрактного типа данных, который включает в себя члены класса. Рассмотрим некоторые из них:

* *Поля* – непосредственно данные определенного типа для описания атрибутов;
* *Методы* - функции, предназначенные для обработки внутренних данных объекта данного класса;
* *Свойства* – это специальные поля данных, с помощью которых, можно управлять поведением объектов данного класса.

*Класс* служит образцом для создания объектов или, другими словами, *объект* является экземпляром класса.

Важным свойством объекта является его обособленность. Детали реализации объекта, то есть внутренние структуры данных и алгоритмы их обработки, скрыты от пользователя и недоступны для непреднамеренного изменения. *Объект* используется через его *интерфейс* - совокупность правил доступа. Скрытие деталей реализации называется *инкапсуляцией*.

В *ООП* данные и методы одного класса могут передаваться другим классам с помощью механизма *наследования*. Порожденный *класс* (*потомок*), наследующий характеристики другого класса, обладает теми же возможностями, что и *класс* (предок), от которого он порожден. При этом *класс-предок* остается без изменения, а классу-потомку можно добавлять новые элементы (поля, методы, свойства) или изменять унаследованные методы. Благодаря этому *класс*-*потомок* обладает большими возможностями, чем предок. Так, например, все классы (а их очень много и с некоторыми из них мы познакомимся чуть позже) порождены от корневого класса ***System.Object***.

Классы-потомки некоторого класса являются разновидностями этого класса-предка. (Другими словами, *класс-предок* является обобщением своих потомков). Это означает, что к объектам классов-потомков можно обращаться с помощью одного и того же имени (но при этом могут выполняться различные действия) — что составляет суть *полиморфизма*. Чаще всего понятие полиморфизма связывают с механизмом виртуальных методов, который мы рассмотрим позднее.

Программу, построенную на принципах *ООП*, можно представить как совокупность взаимодействующих объектов. *Объект* А воздействует на *объект* Б, и для Б возникает *событие*, на которое Б отреагирует либо ответным воздействием на А, либо воздействием на *объект* В. Если А — внешний для системы *объект*, то Б — интерфейсный *объект* (отвечающий за взаимодействие системы с внешним миром).

*Операционная система* *Windows* — объектно-ориентированная система, в которой определены классы для производства объектов, обеспечивающих, в частности, *интерфейс* с пользователем. *Программа*, написанная под *Windows*, обращается к ней, командуя какой интерфейсный *объект* на каком именно месте создать — так строится внешний вид (*интерфейс*) программы. Эти интерфейсные объекты кажутся принадлежащими программе, но на самом деле они — часть *Windows*: именно она отвечает за их базовый внешний вид и поведение. Поэтому, когда *пользователь* воздействует на интерфейсный *объект* программы (нажимает кнопку, выбирает *пункт* *меню* и т.п.), для этого объекта происходит событие и *Windows* переводит это событие в сообщение для программы. При написании программы предусматривается однозначная *реакция* на это сообщение в виде метода, а в методе вызываются методы других объектов. Т.е. воздействие на интерфейсный *объект* приводит к появлению в недрах *Windows* сообщения, которое, приходя в программу, запускает цепочку взаимодействий внутренних для нее объектов. Таким образом, *ООП* *программа* фактически встраивается в ОС *Windows*.

Например, первая программа:

class Program //класс

{

static void Main () //метод класса

{

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

}

}

*Программа* содержит ***класс* Program** с единственным статическим (static) **методом Main**, что позволяет обращаться к данному методу класса без создания его экземпляра.

**Состав языка**

*Алфавит* – совокупность допустимых в языке символов. *Алфавит* языка С# включает:

1. прописные и строчные латинские буквы и буквы национальных алфавитов (включая кириллицу);
2. арабские цифры от 0 до 9, шестнадцатеричные цифры от A до F;
3. специальные знаки: " { } , | ; [ ] ( ) + - / % \* . \ ' : ? < = > ! & ~ ^ @ \_
4. пробельные символы: пробел, символ табуляции, символ перехода на новую строку.

Из символов алфавита формируются лексемы языка: идентификаторы, ключевые (зарезервированные) слова, знаки операций, константы, разделители (скобки, точка, запятая, пробельные символы).

*Идентификатор* – это имя программного элемента: константы, переменной, метки, типа, класса, объекта, метода и т.д. *Идентификатор* может включать латинские буквы и буквы национальных алфавитов, цифры и символ подчеркивания. Прописные и строчные буквы различаются, например, myname, myName и MyName — три различных имени. Первым символом идентификатора может быть буква или знак подчеркивания, но не цифра.

Пробелы внутри имен не допускаются. Язык С# не налагает никаких ограничений на длину имен, однако для удобства чтения и записи кода не стоит делать их слишком длинными.

Для улучшения читабельности кода программным элементам следует давать осмысленные имена, составленные в соответствии с определенными правилами.

*Ключевые слова* – это зарезервированные идентификаторы, которые имеют специальное *значение* для компилятора, например, *include*, main, *int* и т.д. Ключевые слова можно использовать только по прямому назначению. С ключевыми словами и их назначением можно ознакомиться в справочной системе С#.

**Типы данных**

С# является языком со *строгой типизацией*. В нем необходимо объявлять тип всех создаваемых программных элементов (например, переменных, объектов, окон, кнопок и т. д.), что позволяет компилятору предотвращать возникновение ошибок, следя за тем, чтобы объектам присваивались значения только разрешенного типа. Тип программного элемента сообщает компилятору о его размере (например, тип *int* показывает, что *объект* занимает 4 байта) и возможностях (например, кнопка может быть нарисована, нажата и т. д.).

В С# типы делятся на две группы: *базовые* типы, предлагаемые языком, и типы, *определяемые пользователем*. Кроме того, типы С# разбиваются на две другие категории: *размерные типы (типы по значению)* и *ссылочные типы*. Почти все базовые типы являются размерными *типами. Исключение* составляют типы Object и String. Все пользовательские типы, кроме структур, являются ссылочными. Дополнительно к упомянутым типам, язык С# поддерживает типы *указателей*, однако они используются только с неуправляемым кодом.

Принципиальное различие между размерными и ссылочными типами состоит в способе хранения их значений в памяти. В первом случае фактическое *значение* хранится в стеке (или как часть большого объекта ссылочного типа). *Адрес* переменной ссылочного типа тоже хранится в стеке, но сам *объект* хранится в куче.

**Замечание** *. Стек-* это структура, используемая для хранения элементов по *принципу первым пришел - последним ушел*. В данном случае под стеком понимается область памяти, обслуживаемая процессором, в которой хранятся значения локальных переменных. Куча – область памяти, которая используется для хранения данных, работа с которыми реализуется через указатели и ссылки. Память для размещения таких данных динамически выделяется или освобождается в куче неявно (средствами CLR) или явно (программистом).

Сборщик мусора уничтожает программные элементы в стеке через некоторое время после того, как закончит существование раздел стека, в котором они объявлены. В типичном случае раздел стека определяется методом. То есть, если в пределах метода объявлена *локальная переменная*, соответствующий программный элемент будет помечен для сборки мусора по окончании метода. *Объект* в куче подвергается сборке мусора через некоторое время после того, как уничтожена последняя *ссылка* на него.

Язык С# предлагает обычный набор базовых типов, каждому из них соответствует тип, поддерживаемый общеязыковой спецификацией .*NET* (*CLS*). Соответствие базовых типов языка С# и типов платформы .*NET* гарантирует, что объекты, созданные в С#, могут быть использованы на равных основаниях с объектами, созданными в любом другом языке, удовлетворяющем требованиям .*NET* *CLS* (например, в языке VB.*NET*).

|  |  |  |  |
| --- | --- | --- | --- |
| **Тип** | **Размер в байтах** | **Тип .NET** | **Описание** |
| **Базовый тип** | | | |
| object |  | Object | Может хранить все что угодно, т.к. является всеобщим предком |
| **Логический тип** | | | |
| bool | 1 | Boolean | true или false |
| **Целые типы** | | | |
| sbyte | 1 | SByte | Целое со знаком (от -128 до 127) |
| byte | 1 | Byte | Целое без знака (от 0 до 255) |
| short | 2 | Int16 | Целое со знаком (от -32768 до 32767) |
| ushort | 2 | UInt16 | Целое без знака (от 0 до 65535) |
| int | 4 | Int32 | Целое со знаком (от -2147483648 до 2147483647) |
| uint | 4 | UInt | Целое число без знака ( от 0 до 4 294 967 295) |
| long | 8 | Int64 | Целое со знаком (от -9223372036854775808 до 9223372036854775807) |
| ulong | 8 | UInt64 | Целое без знака (от 0 до 0fffffffffffffff) |
| **Вещественные типы** | | | |
| float | 4 | Single | Число с плавающей точкой двойной точности. Содержит значения приблизительно от -1.5\*10-45 до +3.4\*1038 c 7 значащими цифрами |
| double | 8 | Double | Число с плавающей точкой двойной точности. Содержит значения приблизительно от -5. 0\*10-324 до -1.7\*10308 c 15-16 значащими цифрами |
| **Символьный тип** | | | |
| char | 2 | Сhar | Символы Unicode |
| **Строковый тип** | | | |
| string |  | String | Строка из Unicode-символов |
| **Финансовый тип** | | | |
| decimal | 12 | Decimal | Число до 28 знаков с фиксированным положением десятичной точки. Обычно используется в финансовых расчетах. Требует суффикса <<m>> или <<М>> |

**Переменные и константы**

**Переменная** представляет собой типизированную область памяти. Программист создает переменную, объявляя ее тип и указывая имя. При объявлении переменной ее можно инициализировать (присвоить ей начальное *значение*), а затем в любой момент ей можно присвоить новое *значение*, которое заменит собой предыдущее.

static void Main()

{

int i=10; //объявление и инициализация целочисленной переменной i

Console.WriteLine(i); //просмотр значения переменной

i=100; //изменение значение переменной

Console.WriteLine(i);

}

В языке С# требуется, чтобы переменные были явно проинициализированы до их использования. Проверим этот факт на примере.

static void Main()

{

int i;

Console.WriteLine(i);

}

При попытке скомпилировать этот пример в списке ошибок будет выведено следующее сообщение: *Use of unassigned local variable 'i'* (используется неинициализированная *локальная переменная* i).

Однако инициализировать каждую переменную необязательно, но необходимо присвоить ей *значение* до того, как она будет использована.

**Константа** – это *переменная*, *значение* которой нельзя изменить. *Константы* бывают трех видов: *литералы, символические константы* и *перечисления*.

В операторе присваивания: x=32;

число 32 является литеральной константой. Его *значение* всегда равно 32 и его нельзя изменить.

*Символические константы* именуют постоянные значения. *Определение* *символической константы* происходит следующим образом:

const <тип> <идентификатор> = <значение>;

Рассмотрим пример:

static void Main()

{

const int i=10; //объявление целочисленной константы i

Console.WriteLine(i); //просмотр значения константы

i=100; //ошибка

Console.WriteLine(i);

}

### 

### Организация ввода-вывода данных. Форматирование

*Программа* при вводе данных и выводе результатов взаимодействует с внешними устройствами. Совокупность стандартных устройств ввода (клавиатура) и вывода (экран) называется консолью. В языке С# нет операторов ввода и вывода. Вместо них для обмена данными с внешними устройствами используются специальные объекты. В частности, для работы с консолью используется стандартный *класс* ***Console***, определенный в пространстве имен ***System.***

#### Вывод данных

В приведенных выше примерах мы уже рассматривали метод WriteLine, реализованный в классе Console, который позволяет организовывать вывод данных на экран. Однако существует несколько способов применения данного метода:

1. Console.WriteLine(x); //на экран выводится значение идентификатора х
2. Console.WriteLine("x=" + x +"y=" + y); /\* на экран выводится строка, образованная последовательным слиянием строки "x=", значения x, строки "у=" и значения у \*/
3. Console.WriteLine("x={0} y={1}", x, y); /\* на экран выводится строка, формат которой задан первым аргументом метода, при этом вместо параметра {0} выводится значение x, а вместо {1} – значение y\*/

**Замечание**.

Рассмотрим следующий фрагмент программы:

int i=3, j=4; Console.WriteLine("{0} {1}", i, j);

При обращении к методу WriteLine через запятую перечисляются три аргумента: "{0} {1}", i, j. Первый аргумент определяет формат выходной строки. Следующие аргументы нумеруются с нуля, так переменная i имеет номер 0, j – номер 1. Значение переменной i будет помещено в выходную строку на место {0}, а значение переменной j – на место {1}. В результате на экран будет выведена строка: 3 4. Если мы обратимся к методу WriteLine следующим образом Console.WriteLine("{0} {1} {2)", j, i, j), то на экран будет выведена строка: 4 3 4.

Последний вариант использования метода WriteLine является наиболее универсальным, потому что он позволяет не только выводить данные на экран, но и управлять форматом их вывода. Рассмотрим несколько примеров:

1. *Использование управляющих последовательностей*:

Управляющей последовательностью называют определенный символ, предваряемый обратной косой чертой. Данная совокупность символов интерпретируется как одиночный символ и используется для представления кодов символов, не имеющих графического обозначения (например, символа перевода курсора на новую строку) или символов, имеющих специальное обозначение в символьных и строковых константах (например, апостроф). Рассмотрим управляющие символы:

|  |  |
| --- | --- |
| **Вид** | **Наименование** |
| \a | Звуковой сигнал |
| \b | Возврат на шаг назад |
| \f | Перевод страницы |
| \n | Перевод строки |
| \r | Возврат каретки |
| \t | Горизонтальная табуляция |
| \v | Вертикальная табуляция |
| \\ | Обратная косая черта |
| \' | Апостроф |
| \" | Кавычки |

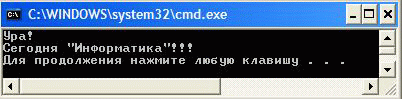
Пример:

static void Main()

{

Console.WriteLine("Ура!\nСегодня \"Информатика\"!!!");

}



1. *Управление размером поля вывода*:

Первым аргументом WriteLine указывается строка вида {n, m} – где n определяет номер идентификатора из списка аргументов метода WriteLine, а m – количество позиций (размер поля вывода), отводимых под значение данного идентификатора. При этом значение идентификатора выравнивается по правому краю. Если выделенных позиций для размещения значения идентификатора окажется недостаточно, то автоматически добавится необходимое количество позиций. Пример:

|  |  |
| --- | --- |
| static void Main()  {  double x= Math.E;  Console.WriteLine("E={0,20}", x);  Console.WriteLine("E={0,10}", x);  }  02_02 |  |

1. *Управление размещением вещественных данных*:

Первым аргументом WriteLine указывается строка вида {n: ##.###} – где n определяет номер идентификатора из списка аргументов метода WriteLine, а ##.### определяет *формат вывода* вещественного числа. В данном случае под целую часть числа отводится две позиции, под дробную – три. Если выделенных позиций для размещения целой части значения идентификатора окажется недостаточно, то автоматически добавиться необходимое количество позиций. Пример:

|  |  |
| --- | --- |
| static void Main()  {  double x= Math.E;  Console.WriteLine("E={0:##.###}", x);  Console.WriteLine("E={0:.####}", x);  }  02_03 |  |

1. *Управление форматом числовых данных*:

Первым аргументом WriteLine указывается строка вида {n: <спецификатор>m} – где n определяет номер идентификатора из списка аргументов метода WriteLine, <спецификатор> - определяет формат данных, а m – количество позиций для дробной части значения идентификатора. В качестве спецификаторов могут использоваться следующие значения:

|  |  |  |
| --- | --- | --- |
| **Параметр** | **Формат** | **Значение** |
| C или c | Денежный. По умолчанию ставит знак р. Изменить его можно с помощь объекта NumberFormatInfo | Задается количество десятичных разрядов. |
| D или d | Целочисленный (используется только с целыми числами) | Задается минимальное количество цифр. При необходимости результат дополняется начальными нулями |
| E или e | Экспоненциальное представление чисел | Задается количество символов после запятой. По умолчанию используется 6 |
| F или f | Представление чисел с фиксированной точкой | Задается количество символов после запятой |
| G или g | Общий формат (или экспоненциальный, или с фиксированной точкой) | Задается количество символов после запятой. По умолчанию выводится целая часть |
| N или n | Стандартное форматирование с использованием запятых и пробелов в качестве разделителей между разрядами | Задается количество символов после запятой. По умолчанию – 2, если число целое, то ставятся нули |
| X или x | Шестнадцатеричный формат |  |
| P или p | Процентный |  |

Пример:

static void Main()

{

Console.WriteLine("C Format:{0,14:C} \t{0:C2}", 12345.678);

Console.WriteLine("D Format:{0,14:D} \t{0:D6}", 123);

Console.WriteLine("E Format:{0,14:E} \t{0:E8}", 12345.6789);

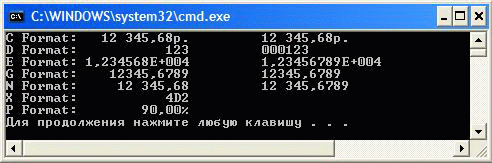
Console.WriteLine("G Format:{0,14:G} \t{0:G10}", 12345.6789);

Console.WriteLine("N Format:{0,14:N} \t{0:N4}", 12345.6789);

Console.WriteLine("X Format:{0,14:X} ", 1234);

Console.WriteLine("P Format:{0,14:P} ", 0.9);

}



#### Ввод данных

Для ввода данных обычно используется метод ReadLine, реализованный в классе Console. Особенностью данного метода является то, что в качестве результата он возвращает строку ( string ). Пример:

static void Main()

{

string s = Console.ReadLine();

Console.WriteLine(s);

}

Для того чтобы получить числовое значение необходимо воспользоваться преобразованием данных. Пример:

static void Main()

{

string s = Console.ReadLine();

int x = int.Parse(s); //преобразование строки в число

Console.WriteLine(x);

}

Или сокращенный вариант:

static void Main()

{

//преобразование введенной строки в число

int x = int.Parse(Console.ReadLine());

Console.WriteLine(x);

}

Для преобразования строкового представления целого числа в тип int мы используем метод int.Parse(), который реализован для всех числовых типов данных. Таким образом, если нам потребуется преобразовать строковое представление в вещественное, мы можем воспользоваться методом float.Parse() или double.Parse(). В случае, если соответствующее преобразование выполнить невозможно, то выполнение программы прерывается и генерируется исключение System.FormatExeption (*входная строка* имела неверный формат).

**Полный *список операций* языка С#**.

В данном разделе мы подробно рассмотрим только часть операций, остальные *операции* будут вводиться по мере необходимости.

1. **Инкремент (++) и декремент(--)**.

Эти операции имеют две формы записи - *префиксную*,когда операция записывается перед операндом, и *постфиксную* - операция записывается после операнда. *Префиксная операция* инкремента (декремента) увеличивает (уменьшает) свой операнд и возвращает измененное значение как результат. Постфиксные версии инкремента и декремента возвращают первоначальное значение операнда, а затем изменяют его.

Рассмотрим эти операции на примере.

|  |  |
| --- | --- |
| static void Main()  {  int i = 3, j = 4;  Console.WriteLine("{0} {1}", i, j);  Console.WriteLine("{0} {1}", ++i, --j);  Console.WriteLine("{0} {1}", i++, j--);  Console.WriteLine("{0} {1}", i, j);  } | *Результат работы программы*:  3 4  4 3  4 3  5 2 |

1. **Операция new**. Используется для создания нового объекта. С помощью ее можно создавать как объекты ссылочного типа, так и размерные, например:
2. object z=new object();

int i=new int(); // то же самое, что и int i =0;

1. **Отрицание**:
   * Арифметическое отрицание (-) - меняет знак операнда на противоположный.
   * Логическое отрицание (!) - определяет операцию инверсия для логического типа.

Рассмотрим эти операции на примере.

|  |  |
| --- | --- |
| static void Main()  {  int i = 3, j=-4;  bool a = true, b=false;  Console.WriteLine("{0} {1}", -i, -j);  Console.WriteLine("{0} {1}", !a, !b);  } | *Результат работы программы*:  -3 4  False True |

1. **Явное преобразование типа**.Используется для явного преобразования из одного типа в другой. Формат операции:

(тип) выражение;

Рассмотрим эту операцию на примере.

|  |  |
| --- | --- |
| static void Main()  {  int i = -4;  byte j = 4;  int a = (int)j; //преобразование без потери точности  byte b = (byte)i; //преобразование с потерей точности  Console.WriteLine("{0} {1}", a, b);  } | *Результат работы программы*:  4 252 |

1. **Умножение (\*), деление (/) и деление с остатком (%)**. Операции умножения и деления применимы для целочисленных и вещественных типов данных. Для других типов эти операции применимы, если для них возможно *неявное преобразование* к целым или вещественным типам. При этом тип результата равен "наибольшему" из типов операндов, но не менее int. Если оба операнда при делении целочисленные, то и результат тоже целочисленный.

Рассмотрим эти операции на примере.

static void Main()

{

int i = 100, j = 15;

double a = 14.2, b = 3.5;

Console.WriteLine("{0} {1} {2}", i\*j, i/j, i%j);

Console.WriteLine("{0} {1} {2}", a \* b, a / b, a % b);

}

*Результат работы программы*:

1500 6 10

49.7 4.05714285714286 0.1999999999999999

1. **Сложение (+) и вычитание (-)**. Операции сложения и вычитания применимы для целочисленных и вещественных типов данных. Для других типов эти операции применимы, если для них возможно *неявное преобразование* к целым или вещественным типам.
2. **Операции отношения ( <, <=, >, >=, ==, !=)**. Операции отношения сравнивают значения левого и правого операндов. Результат операции логического типа: true - если значения совпадают, false - в противном случае. Рассмотрим операции на примере:

|  |  |
| --- | --- |
| static void Main()  {  int i = 15, j = 15;  Console.WriteLine(i<j); //меньше  Console.WriteLine(i<=j); //меньше или равно  Console.WriteLine(i>j); //больше  Console.WriteLine(i>=j); //больше или равно  Console.WriteLine(i==j); //равно  Console.WriteLine(i!=j); //не равно  } | *Результат работы программы*:  False  True  False  True  True  False |

1. **Логические операции**: И ( && ), ИЛИ ( || ).

Логические операции применяются к операндам логического типа.

Результат логической операции И имеет значение истина тогда и только тогда, когда оба операнда принимают значение истина.

Результат логической операции ИЛИ имеет значение истина тогда и только тогда, когда хотя бы один из операндов принимает значение истина.

Рассмотрим операции на примере:

static void Main()

{

Console.WriteLine("x y x и y x или y");

Console.WriteLine("{0} {1} {2} {3}", false, false, false&&false, false||false);

Console.WriteLine("{0} {1} {2} {3}", false, true, false&&true, false||true);

Console.WriteLine("{0} {1} {2} {3}", true, false, true&&false, true||false);

Console.WriteLine("{0} {1} {2} {3}", true, true, true&&true, true||true);

}

*Результат работы программы*:

x y x и y x или y

False False False False

False True False True

True False False True

True True True True

1. **Условная операция**.

Формат: (<операнд1>)? <операнд2> : <операнд3> ;

Операнд1 - это логическое выражение, которое оценивается с точки зрения его эквивалентности константам true и false. Если результат вычисления операнда1 равен true, то результатом условной операции будет значение операнда2, иначе - операнда3. Фактически условная операция является сокращенной формой условного оператора if, который будет рассмотрен позже.

Пример использования условной операции:

static void Main()

{

int x=5; int y=10;

int max = (x > y) ? x : y;

Console.WriteLine(max);

}

1. **Операции присваивания**: =, +=, -= и т.д.

Формат операции *простого присваивания* ( = ):

операнд\_2 = операнд\_1;

В результате выполнения этой операции вычисляется значение операнда\_1, и результат записывается в операнд\_2. Возможно связать воедино сразу несколько операторов присваивания, записывая такие цепочки: a=b=c=100. Выражение такого вида выполняется справа налево: результатом выполнения c=100 является число 100, которое затем присваивается переменной b, результатом чего опять является 100, которое присваивается переменной a.

Кроме простой операции присваивания существуют *сложные операции присваивания*, например, умножение с присваиванием ( \*= ), деление с присваиванием ( /= ), остаток от деления с присваиванием ( % =), сложение с присваиванием ( += ), вычитание с присваиванием ( -= ) и т.д.

В сложных операциях присваивания, например, при *сложении с присваиванием*, к операнду\_2 прибавляется операнд\_1, и результат записывается в операнд\_2. То есть, выражение с += а является более компактной записью выражения с = с + а. Кроме того, сложные операции присваивания позволяют сгенерировать более эффективный код, за счет того, что в простой операции присваивания для хранения значения правого операнда создается временная переменная, а в сложных операциях присваивания значение правого операнда сразу записывается в левый операнд.

### Выражения и преобразование типов

*Выражение* - это синтаксическая *единица* языка, определяющая способ вычисления некоторого значения. Выражения состоят из операндов, операций и скобок. Каждый *операнд* является в свою *очередь* выражением или одним из его частных случаев - константой, переменной или функций.

**Лекция 2 Операторы языка C#.**

*Программа* на языке С# состоит из последовательности операторов, каждый из которых определяет законченное описание некоторого действия и заканчивается точкой с запятой. Все *операторы* можно разделить на 4 группы: *операторы* следования, *операторы* ветвления, *операторы цикла* и *операторы передачи управления*.

#### Операторы следования

Операторы следования выполняются компилятором в естественном порядке: начиная с первого до последнего. К операторам следования относятся: выражение и составной оператор.

Любое *выражение*, завершающееся точкой с запятой, рассматривается как оператор, выполнение которого заключается в вычислении значения выражения или выполнении законченного действия, например, вызова метода. Например:

++i; //оператор инкремента

x+=y; //оператор сложение с присваиванием

Console.WriteLine(x); //вызов метода

x=Math.Pow(a,b)+a\*b; //вычисление сложного выражения

Частным случаем оператора выражения является *пустой оператор* ; Он используется тогда, когда по синтаксису оператор требуется, а по смыслу - нет. В этом случае лишний символ ; является пустым оператором и вполне допустим, хотя и не всегда безопасен. Например, случайный символ ; после условия оператора while или if может совершенно поменять работу этого оператора.

*Составной оператор* или *блок* представляет собой последовательность операторов, заключенных в фигурные скобки {}. Блок обладает собственной *областью видимости*: объявленные внутри блока имена доступны только внутри данного блока или блоков, вложенных в него. Составные операторы применяются в случае, когда правила языка предусматривают наличие только одного оператора, а логика программы требует нескольких операторов. Например, тело цикла while должно состоять только из одного оператора. Если заключить несколько операторов в фигурные скобки, то получится блок, который будет рассматриваться компилятором как единый оператор.

#### Операторы ветвления

Операторы ветвления позволяют изменить порядок выполнения операторов в программе. К операторам ветвления относятся условный оператор if и оператор выбора switch.

##### Условный оператор if

Условный оператор *if* используется для разветвления процесса обработки данных на два направления. Он может иметь одну из форм: *сокращенную* или *полную*.

Форма *сокращенного оператора if*:

if (B) S;

где В - логическое или арифметическое выражение, истинность которого проверяется; S - оператор: простой или составной.

При выполнении сокращенной формы оператора if сначала вычисляется выражение B, затем проводится анализ его результата: если B истинно, то выполняется оператор S ; если B ложно, то оператор S пропускается. Таким образом, с помощью сокращенной формы оператора if можно либо выполнить оператор S, либо пропустить его.

Форма *полного оператора if*:

if (B) S1; else S2;

где B - логическое или арифметическое выражение, истинность которого проверяется; S1, S2 - оператор: простой или составной.

При выполнении полной формы оператора if сначала вычисляется выражение B, затем анализируется его результат: если B истинно, то выполняется оператор S1, а оператор S2 пропускается; если B ложно, то выполняется оператор S2, а S1 - пропускается. Таким образом, с помощью полной формы оператора if можно выбрать одно из двух альтернативных действий процесса обработки данных.

Рассмотрим несколько примеров записи условного оператора if:

if (a > 0) x=y; // Сокращенная форма c простым оператором

if (++i) {x=y; y=2\*z;} // Сокращенная форма c составным оператором

if (a > 0 || b<0) x=y; else x=z; // Полная форма с простым оператором

if (i+j-1) { x= 0; y= 1;} else {x=1; y:=0;} // Полная форма с составными операторами

Рассмотрим пример использования условного оператора.

static void Main()

{

Console.Write("x= ");

float x = float.Parse(Console.ReadLine());

Console.Write("y=");

float y = float.Parse(Console.ReadLine());

if (x < y ) Console.WriteLine("min= "+x);

else Console.WriteLine("min= "+y);

}

*Результат работы программы*:

x y min

0 0 0

1 -1 -1

-2 2 -2

##### Оператор выбора switch

Оператор выбора switch предназначен для разветвления процесса вычислений по нескольким направлениям. Формат оператора:

switch ( <выражение> )

{

case <константное\_выражение\_1>:

[<оператор 1>]; <оператор перехода>;

case <константное\_выражение\_2>:

[<оператор 2>]; <оператор перехода>;

...

case <константное\_выражение\_n>:

[<оператор n>]; <оператор перехода>;

[default: <оператор>; ]

}

**Пример**. По заданному виду арифметической операции (сложение, вычитание, умножение и деление) и двум операндам, вывести на экран результат применения данной операции к операндам.

static void Main()

{

Console.Write("OPER= ");

char oper=char.Parse(Console.ReadLine());

bool ok=true;

Console.Write("A= ");

int a=int.Parse(Console.ReadLine());

Console.Write("B= ");

int b=int.Parse(Console.ReadLine());

float res=0;

switch (oper)

{

case '+': res = a + b; break; //1

case '-': res = a - b; break;

case '\*': res = a \* b; break;

case ':': if (b != 0)

{

res = (float)a / b; break;

}

else goto default;

default: ok = false; break;

}

if (ok) Console.WriteLine("{0} {1} {2} = {3}", a, oper, b, res);

else Console.WriteLine("error");

}

*Результат выполнения программы*:

oper x y rez

+ 4 5 9

: 4 0 error

% 4 3 error

#### **Операторы цикла**

Операторы цикла используются для организации многократно повторяющихся вычислений. К операторам цикла относятся: *цикл с предусловием* while, *цикл с постусловием* do while, цикл с параметром for и цикл перебора foreach.

##### ***Цикл с предусловием while***

Оператор цикла while организует выполнение одного оператора (простого или составного) неизвестное заранее число раз. Формат цикла while:

while (B) S;

где B - выражение, истинность которого проверяется (условие завершения цикла); S - тело цикла - оператор (простой или составной).

Перед каждым выполнением тела цикла анализируется значение выражения В: если оно истинно, то выполняется тело цикла, и управление передается на повторную проверку условия В ; если значение В ложно - цикл завершается и управление передается на оператор, следующий за оператором S.

Если результат выражения B окажется ложным при первой проверке, то тело цикла не выполнится ни разу. Отметим, что если условие B во время работы цикла не будет изменяться, то возможна ситуация зацикливания, то есть невозможность выхода из цикла. Поэтому внутри тела должны находиться операторы, приводящие к изменению значения выражения B так, чтобы цикл мог корректно завершиться.

В качестве иллюстрации выполнения цикла while рассмотрим программу вывода на экран целых чисел из интервала от 1 до n.

static void Main()

{

Console.Write("N= ");

int n=int.Parse(Console.ReadLine());

int i = 1;

while (i <= n) //пока i меньше или равно n

Console.Write(" "+ i++ ); //выводим i на экран, затем увеличиваем его на 1

}

*Результаты работы программы*:

n ответ

10 1 2 3 4 5 6 7 8 9 10

##### ***Цикл с постусловием do while***

Оператор цикла do while также организует выполнение одного оператора (простого или составного) неизвестное заранее число раз. Однако в отличие от цикла while условие завершения цикла проверяется после выполнения тела цикла. Формат цикла do while:

do S while (B);

где В - выражение, истинность которого проверяется (условие завершения цикла); S - тело цикла - оператор (простой или блок).

Сначала выполняется оператор S, а затем анализируется значение выражения В: если оно истинно, то управление передается оператору S, если ложно - цикл завершается, и управление передается на оператор, следующий за условием B. Так как условие В проверяется после выполнения тела цикла, то в любом случае тело цикла выполнится хотя бы один раз.

В операторе do while, так же как и в операторе while, возможна ситуация зацикливания в случае, если условие В всегда будет оставаться истинным.

В качестве иллюстрации выполнения цикла do while рассмотрим программу вывода на экран целых чисел из интервала от 1 до n.

static void Main()

{

Console.Write("N= ");

int n=int.Parse(Console.ReadLine());

int i = 1;

do

Console.Write(" " + i++);

//выводим i на экран, затем увеличиваем его на 1

while (i <= n); //пока i меньше или равно n

}

##### ***Цикл с параметром for***

Цикл с параметром имеет следующую структуру:

for ( <инициализация>; <выражение>; <модификация>) <оператор>;

*Инициализация* используется для объявления и/или присвоения начальных значений величинам, используемым в цикле в качестве параметров (счетчиков). В этой части можно записать несколько операторов, разделенных запятой. Областью действия переменных, объявленных в части инициализации цикла, является цикл и *вложенные блоки*. Инициализация выполняется один раз в начале исполнения цикла.

*Выражение* определяет условие выполнения цикла: если его результат истинен, цикл выполняется. Истинность выражения проверяется перед каждым выполнением тела цикла, таким образом, цикл с параметром реализован как *цикл с предусловием*. В блоке выражение через запятую можно записать несколько логических выражений, тогда запятая равносильна операции логическое И ( && ).

*Модификация* выполняется после каждой итерации цикла и служит обычно для изменения параметров цикла. В части модификация можно записать несколько операторов через запятую.

*Оператор* (простой или составной) представляет собой тело цикла.

Любая из частей оператора *for* (инициализация, выражение, модификация, оператор) может отсутствовать, но точку с запятой, определяющую позицию пропускаемой части, надо оставить.

static void Main()

{

Console.Write("N= ");

int n=int.Parse(Console.ReadLine());

for (int i=1; i<=n;) //блок модификации пустой

Console.Write(" " + i++);

}

##### Вложенные циклы

Циклы могут быть простые или вложенные (кратные, циклы в цикле). Вложенными могут быть циклы любых типов: while, do while, for. Каждый внутренний цикл должен быть полностью вложен во все внешние циклы. "Пересечения" циклов не допускаются.

Рассмотрим пример использования вложенных циклов, который позволит вывести на экран числа следующим образом:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2 | 2 | 2 | 2 | 2 |
| 2 | 2 | 2 | 2 | 2 |
| 2 | 2 | 2 | 2 | 2 |
| 2 | 2 | 2 | 2 | 2 |

static void Main()

{

for (int i = 1; i <= 4; ++i, Console.WriteLine()) //1

for (int j=1; j<=5; ++j)

Console.Write(" " + 2);

}

***Оператор цикла foreach***

Это итерационный цикл (подобный for)для работы с элементами массива, рассмотри позже.

**Методы: основные понятия**

Метод – это функциональный элемент класса, который реализует вычисления или другие действия, выполняемые классом или его экземпляром (объектом). Метод представляет собой законченный фрагмент кода, к которому можно обратиться по имени. Он описывается один раз, а вызываться может многократно. Совокупность методов класса определяет, что конкретно может делать класс. Например, стандартный класс Math содержит методы, которые позволяют вычислять значения математических функций.

Синтаксис метода:

[атрибуты] [спецификторы] тип\_возвращаемого\_результата имя\_метода ([список\_параметров])

{

тело\_метода;

return значение

}

где:

1. Атрибуты и спецификторы являются необязательными элементами синтаксиса описания метода. На данном этапе атрибуты нами использоваться не будут, а из всех спецификаторов мы в обязательном порядке будем использовать спецификатор static, который позволит обращаться к методу класса без создания его экземпляра.

2. Тип\_возвращаемого\_результата определяет тип значения, возвращаемого методом. Это может быть любой тип, включая типы классов, создаваемые программистом. Если метод не возвращает никакого значения, необходимо указать тип void (в этом случае в теле метода отсутсвует оператор return ).

3. Имя\_метода – идентификатор, заданный программистом с учетом требований, накладываемыми на идентификаторы в С#, отличный от тех, которые уже использованы для других элементов программы в пределах текущей области видимости.

4. Список\_параметров представляет собой последовательность пар, состоящих из типа данных и идентификатора, разделенных запятыми. Параметры — это переменные или константы, которые получают значения, передаваемые методу при вызове. Если метод не имеет параметров, то список\_параметров остается пустым.

5. Значение определяет значение, возвращаемое методом. Тип значения должен соответствовать типу\_возвращаемого\_результата или приводится к нему.

Рассмотрим простейший пример метода:

class Program

{

static void Func() //дополнительный метод

{

Console.Write("x= ");

double x=double.Parse(Console.ReadLine());

double y = 1 / x;

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

}

static void Main() //точка входа в программу

{

Func(); //первый вызов метода Func

Func(); //второй вызов метода Func

}

}

В даном примере в метод Func не передаются никакие значения, поэтому список параметров пуст. Кроме того метод ничего не возвращает, поэтому тип возвращаемого значения void. В основном методе Main мы вызвали метод Func два раза. Если будет необходимо, то данный метод можно будет вызвать еще столько раз, сколько потребуется для решения задачи.

***Перегрузка методов***

Иногда бывает удобно, чтобы методы, реализующие один и тот же алгоритм для различных типов данных, имели одно и то же имя. Использование нескольких методов с одним и тем же именем, но различными типами и количеством параметров называется перегрузкой методов.Компилятор определяет, какой именно метод требуется вызвать, по типу и количеству фактических параметров.

Рассмотрим следующий пример:

class Program

{

static int max(int a) //первая версия метода max

{

int b = 0;

while (a > 0)

{

if (a % 10 > b) b = a % 10;

a /= 10;

}

return b;

}

static int max(int a, int b) //вторая версия метода max

{

if (a > b) return a;

else return b;

}

static int max(int a, int b, int c) //третья версия метода max

{

if (a > b && a > c) return a;

else if (b > c) return b;

else return c;

}

static void Main()

{

int a = 1283, b = 45, c = 35740;

Console.WriteLine(max(a));

Console.WriteLine(max(a, b));

Console.WriteLine(max(a, b, c));

}

}

При вызове метода max компилятор выбирает вариант, соответствующий типу и количеству передаваемых в метод аргументов. Если точного соответствия не найдено, выполняются неявные преобразования типов в соответствии с общими правилами. Если преобразование невозможно, выдается сообщение об ошибке. Если выбор перегруженного метода возможен более чем одним способом, то выбирается "лучший" из вариантов (вариант, содержащий меньшие количество и длину преобразований в соответствии с правилами преобразования типов). Если существует несколько вариантов, из которых невозможно выбрать лучший, выдается сообщение об ошибке.

Перегрузка методов является проявлением полиморфизма, одного из основных свойств ООП. Программисту гораздо удобнее помнить одно имя метода и использовать его для работы с различными типами данных, а решение о том, какой вариант метода вызвать, возложить на компилятор. Этот принцип широко используется в классах библиотеки .NET. Например, в стандартном классе Console метод WriteLine перегружен 19 раз для вывода величин разных типов.

***Рекурсивные методы***

Рекурсивным называют метод, если он вызывает сам себя в качестве вспомогательного. В основе рекурсивного метода лежит так называемое "рекурсивное определение" какого-либо понятия. Классическим примером рекурсивного метода является метод, вычисляющий факториал.

Из курса математики известно, что 0!=1!=1, n!=1\*2\*3…\*n. С другой стороны n!=(n-1)!\*n. Таким образом, известны два частных случая параметра n, а именно n= 0 и n=1, при которых мы без каких-либо дополнительных вычислений можем определить значение факториала. Во всех остальных случаях, то есть для n>1, значение факториала может быть вычислено через значение факториала для параметра n-1. Таким образом, рекурсивный метод будет иметь вид:

{

static long F(int n) //рекурсивный метод

{

if (n==0 || n==1)

return 1; //нерекурсивная ветвь

else return n\*F(n-1); //шаг рекурсии - повторный вызов метода с другим параметром

}

static void Main()

{

Console.Write("n=");

int n =int.Parse( Console.ReadLine());

long f=F(n); //нерекурсивный вызов метода F

Console.WriteLine("{0}!={1}",n, f);

}

}

Рассмотрим работу описанного выше рекурсивного метода для n=3.

Первый вызов метода осуществляется из метода Main, в нашем случае командой f=F(3). Этап вхождения в рекурсию обозначим жирными стрелками. Он продолжается до тех пор, пока значение переменной n не становится равной 1. После этого начинается выход из рекурсии (тонкие стрелки). В результате вычислений получается, что F(3)=3\*2\*1.

Рассмотренный вид рекурсии называют прямой. Метод с прямой рекурсией обычно содержит следующую структуру:

if (<условие>)

<оператор>;

else <вызов данного метода с другими параметрами>;

В качестве <условия> обычно записываются некоторые граничные случаи параметров, передаваемых рекурсивному методу, при которых результат его работы заранее известен, поэтому далее следует простой оператор или блок, а в ветви else происходит рекурсивный вызов данного метода с другими параметрами.

Что необходимо знать для реализации рекурсивного процесса? Со входом в рекурсию осуществляется вызов метода, а для выхода необходимо помнить точку возврата, т.е. то место программы откуда мы пришли и куда нам нужно будет возвратиться после завершения метода. Место хранения точек возврата называется стеком вызовов и для него выделяется определенная область оперативной памяти. В этом стеке запоминаются не только адреса точек возврата, но и копии значений всех параметров. По этим копиям восстанавливается при возврате вызывающий метод. При развертывании рекурсии за счет создания копий параметров возможно переполнение стека. Это является основным недостатком рекурсивного метода. С другой стороны, рекурсивные методы позволяют перейти к более компактной записи алгоритма.

Следует понимать, что любой рекурсивный метод можно преобразовать в обычный метод. И практически любой метод можно преобразовать в рекурсивный, если выявить рекуррентное соотношение между вычисляемыми в методе значениями.