Запуск скрипта

Если у вас не установлена платформа .NET 6, то нужно её установить. Запустить свой скрипт в интерпретаторе можно двумя способами.

1. При запуске интерпретатора из командной оболочки указать через пробел файл скрипта и аргументы, если надо (всё, как обычно в вашей ОС).
```
Пример запуска из cmd (Windows): C:\SomeDir\OverScript.exe C:\MyScripts\test.txt hello world
В Linux: dotnet /home/somedir/OverScript.dll /home/MyScripts/test.txt hello world
```
2. Запустить интерпретатор и после ввести файл и аргументы. При запуске интерпретатора появится предложение ввести файл скрипта. Формат для всех ОС один: если в пути файла или аргументе есть пробелы, то их нужно взять в одинарные кавычки (внутри не должно быть других одинарных кавычек), а если есть управляющие последовательности специальных символов (\t, \n \ и др.), то нужно взять в двойные кавычки (двойные кавычки внутри, как и др. символы, экранируются обратным слэшем: \").
```
Если пробелов и спец. символов нет, то: C:\MyScripts\test.txt hello world
Если есть пробелы: 'C:\MyScripts\test script.txt' hello world 'some arg with a spaces'
Если есть спец. символы: C:\MyScripts\test.txt "hello\tworld" (или так: C:\MyScripts\test.txt hello"\t"world)
```

Чтобы не писать путь файла скрипта, можно просто перетаскивать его в терминал (командную строку). Если вы вводите файл после запуска интерпретатора, то если путь содержит пробелы, нужно будет заменить добавленные автоматически двойные кавычки (в Windows) на одинарные, чтобы не происходила замена символов после обратного слэша.

Hello, world!

```
WriteLine("Hello, world!");
```

Прописывать главный (обёрточный) класс не нужно. Код автоматически считается находящимся в классе с именем приложения.

```
class AppName{
        WriteLine("Hello, world!");
}
// так делать не нужно!
```

Имя (по умолчанию - App), версия и прочая информация о программе указываются (опционально) в директиве #app, которая должна находится в начале программы (в первой строке до любых комментариев):

```
#app HelloWorld, Version="1.2.3.4", Author=John Doe, Description="This app outputs: Hello, world!"

WriteLine("Hello, world!");
//Данные из директивы #app можно получить функцией AppInfo:
WriteLine(AppInfo("Name")); //HelloWorld
WriteLine(AppInfo("Version")); //1.2.3.4
WriteLine(AppInfo("Author")); //John Doe
WriteLine(AppInfo("Description")); //This app outputs: Hello, world!
```

Как видите, брать значения в кавычки не обязательно. Значение Description взято в кавычки потому, что в нём есть запятая, которая не должна быть принята интерпретатором за разделитель параметров. Значение Version тоже нужно брать в кавычки, т.к. если написать Version=1.2.3.4, то будет ошибка потому, что интерпретатор примет 1.2.3.4 за числовой литерал и попробует перевести значение в тип double.
Задать текущий язык и региональные параметры можно параметром CurrentCulture:

```
#app HelloWorld, CurrentCulture=en-US
```

Пример, когда нужно задавать CurrentCulture:

```
double d:="2.51"; //FormatException: Input string was not in a correct format.
```

Если CurrentCulture по умолчанию ru-RU, значит в строковых представлениях чисел нужно использовать не точку, а запятую. Т.е. если заменить "2.51" на "2,51", то заработает. Либо нужно задать ту CurrentCulture, в которой используется точка, например en-GB.

```
#app HelloWorld, CurrentCulture=en-GB
double d:="2.51";
WriteLine(d); //2.51
```

Очень важно помнить, что если вы используете в коде определённый формат дат, чисел и т.п., то ваша программа может работать некорректно у пользователей с другими настройками языкового стандарта. Чтобы избежать этого, и нужно задавать

параметр CurrentCulture.
При загрузке программы выше, создаётся экземпляр класса HelloWorld (по имени, указанном в #app). В C# и других языках экземпляр главного класса не создаётся, а просто вызывается статический метод Main. Пример выше можно изменить, добавив конструктор:

```
New(){ //модификаторы для конструктора указывать не нужно.
        WriteLine("Hello, world!");
}
```

Код, который находится вне методов считается находящимся в методе Instance. Этот метод вызывается перед конструктором.

```
Instance(){
        WriteLine("Hello, world!");
}
// не нужно так делать потому, что Instance(){*****} добавляется автоматически.
```

Проверим, используя функцию CallChain, возвращающую цепочку вызовов.

```
WriteLine(CallChain()); // App.Instance()
```

В строке "App.Instance()" App - имя программы по умолчанию, а Instance() - метод, в который автоматически помещается код, находящийся вне методов текущего класса, в том числе объявление переменных класса. Объявление статических переменных помещается в метод Static().

```
static int x=5; //превращается в: Static(){int x=5;}
```

Типы данных и их преобразование

Типы данных: object, bool, byte, short, char, int, long, float, double, decimal, date, string. Все типы соответствуют типам C#. date - это DateTime.

```
string s="Hello"; // переменная строкового типа
s=null; // переменные ссылочных типов могут принимать значение null
int x=123; // переменная типа Int32
object o=12345.6789m; // в переменную типа object упаковывается значение типа decimal
o=null; // значение null говорит об отстутсвии значения как такового
// string, date и decimal можно создавать через new (используются стандартные конструкторы этих типов)
string str=new string('*', 5); // *****
date d=new date(2021, 2, 20); // 20.02.2021 0:00:00
decimal dcm=new decimal(1000000000, 0, 0, false, ToByte(0)); // 1000000000
// Конструкторы для других примитивных типов поддерживаются, но не имеют смысла
x = new int(); // в C# так тоже можно
```

В OverScript **все переменные при объявлении инициализируются значениями по умолчанию** (0 для числовых и null для ссылочных типов). Тип date (DateTime) инициализируется значением 01.01.0001 0:00:00.

```
int x;
WriteLine(x); //0
```

Декларировать одну и ту же переменную можно несколько раз, добавляя $ перед именем (тип должен быть один). Это сделано, чтобы случайно не перекрыть переменные, которые уже были объявлены ранее, ведь в OverScript переменные не имеют блочной области видимости (они как в JavaScript переменные, объявленные с помощью ключевого слова var).

```
long x=555;
long $x; //$ перед именем разрешает передекларировать переменную, но только с тем же типом
WriteLine(x); //0 - при объявлении переменной задаётся значение по умолчанию.
ReadKey();
```

Поддерживается вывод типов.

```
var s = "Hello, world!"; //s будет типа string
//s = 5; //ошибка!
```

Для вывода типов переменные в правой части должны быть задекларированы до var-переменной. Поэтому пример ниже выдаст ошибку при загрузке кода:

```
var x = @(int y = 5, y * 2); //Could not determine type of variable 'x'. Variable 'y' not found at 'App'.
//@(int y = 5, y * 2) - функция, которая выполняет аргументы и возвращает значение последнего (10 в данном случае). Хотя
она всегда возвращает object, интерпретатор для поиска функции всё равно должен сначала определить тип аргументов.
```

На момент определения типа `@(int y = 5, y * 2)` интерпретатор ещё не знает тип переменной y, т.к. он ищет декларирование переменных сверху вниз слева направо.

Преобразования типов происходят автоматически, если нет риска потери данных, или используется оператор := (присвоение с конвертированием).

```
double d=2.55;  // 2,55
byte b:=d; // 3 // оператор := используется для присваивания с конвертированием потому, что double больше byte и не может быть присвоен напрямую
float f=b; // 3 // здесь обычное присваивание, т.к. float больше byte
short s:="123"; // 123 //присваивание с конвертированием строки в число
string str:=d+f; // "5,55" //присваивание с конвертированием числа в строку
str=d+"123"; // "2,55123"
bool v:=1; // true
int i:=true; // 1
v:="True"; // true
date dt:="18.02.2021 13:45:30"; // 18.02.2021 13:45:30 // конвертирование строки в дату
// Даты преобразуются в строку и обратно по-разному с разными региональными параметрами
WriteLine(Now()); // 18.02.2021 14:04:15  // строковое представление текущей даты при региональных параметрах по умолчанию (ru-RU, например)
// Далее меняем текущие региональные параметры
object cultureInfo=typeof("System.Globalization.CultureInfo, System.Globalization"); // получаем тип CultureInfo
object culture=Create(cultureInfo,"en-US", false); // создаём объект CultureInfo с региональными параметрами en-US
cultureInfo->CurrentCulture=culture; // устанавливаем новые текущие региональные параметры en-US (-> - это синтаксический сахар для Reflection)
WriteLine(Now()); // 2/18/2021 2:04:15 PM // строковое представление текущей даты при новых параметрах en-US
```

Обратите внимание, что 2.55 (double) конвертируется в 3 (byte), а в C# при явном преобразовании было бы 2. Это потому, что преоразования типов производятся функциями класса Convert (byte b = Convert.ToByte(d)).
При работе с char-ами нужно помнить, что некоторые направления конвертирования не поддерживаются, например, char в double. В C# возможно такое приведение типа, но Convert.ToDouble такое преобразование не поддерживает. По этой причине все арифметические операции с char и числами (целыми и дробными) всегда возвращают целое, т.к. используются перегрузки для int или long.

```
int n='w'; //так можно, т.к. char это фактически как UInt16, который меньше int
WriteLine(n); //119
char c:=n; //как и в C#, напрямую присвоить нельзя, поэтому нужно либо присваивание с конвертированием, либо явное преобразование
WriteLine(c); //w
WriteLine(c+2.55); //122, а не 121,55, т.к. делается int + int (для char нет отдельных перегрузок оператора сложения, а с int подходит потому, что int может принять как char, так и double (нестрогая проверка)). double + double не подходит, т.к. только int и long параметры могут принимать char (потому, что char не конвертируется в double)
//c:=121.55; //выдаст ошибку: Can't convert type 'double' to type 'char'
//WriteLine(double\c); //выдаст ошибку: Can't cast object of type 'char' to type 'double'
WriteLine(double\int\c); //119 //можно: WriteLine((double)(int)c);
```

Литералы и интерполяция строк

С литералами всё как в C# (суффиксы l, f, m). Целые числа по умолчанию имеют тип int, а дробные double (5 - это литерал типа int, а 5.0 - double). Литералы типа char (символьный) берутся в одинарные кавычки, а строки - в двойные кавычки, как в C#. Если перед открывающей кавычкой поставить @, то управляющие последовательности, такие как \r, \n и другие, будут выведены как есть. Если @ не ставить, то управляющие последовательности специальных символов будут заменены на соответствующие символы.

Поддерживается простая интерполяция строк (в фигурных скобках должны быть переменные, а выражения пока не поддерживаются):

```
int x=2, y=3;
WriteLine($"x: {x}; y: {y}"); //x: 2; y: 3
//Интерпретатор видит предыдущую строку так:  WriteLine(@Format("x: {0}; y: {1}",x,y))
//т.е. интерполяция строк это синтаксический сахар для функции Format. @ перед именем функции указывает, что нужно вызвать именно базовую (встроенную) функцию.
//WriteLine($"x: {++x}; y: {y*2}"); //не сработает!
//WriteLine($"x: {x->ToString()}"); //не сработает!
WriteLine($"x: {x.ToString()}"); //сработает, если среди аргументов нет литералов. x.ToString() - это ToString(x).
//WriteLine($"x: {x.ToString(16)}"); //не сработает, т.к. 16 - литерал
//Можно задавать формат:
WriteLine($"x: {x,-15:N2};"); //x: 2,00           ;
//Подробнее про форматы - https://docs.microsoft.com/ru-ru/dotnet/api/system.string.format?view=net-5.0
```

Поясню, почему работает `x.ToString()`, а `x.ToString(16)` не работает (ToString(x, toBase) возвращает представление числа в указанной системе счисления). Когда интерпретатор превращает интерполяцию в функцию Format, он переносит код в фигурных скобках в аргументы функции как есть. `$"x: {x.ToString(16)}"` - это обычная строка, и интерпретатор не

заменяет в ней 16 на литерал (специальную строку с id литерала). Он также не заменяет операторы, синтаксический сахар и другие элементы кода, которые подлежат обработке для того, чтобы строки с ними можно было разложить в выражения. Поэтому, например, строка `$"x: {++x}; y: {y*2}"` превращается в `@Format("x: {0}; y: {1}",++x,y*2)`, в которой ++ и * для интерпретатора являются обычными символами, а не операторами (оператор после замены - это спец. строка с id оператора), и двойка не литерал. То же самое с `x.ToString(16)`, где 16 принимается интерпретатором за имя переменной, а не за число.

Использование символа @ перед именем функции

Кроме строковых литералов, у символа @ есть ещё 4 назначения.

1. Если в вызове функции перед её именем поставить @, то будет вызываться базовая, а не своя перегрузка (*базовая функция* это не функция в базовом классе, а встроенная функция OverScript).

```
WriteLine("test"); //вызов своей функции
void WriteLine(string str){
	//а теперь вызываем базовую функцию:
	@WriteLine("str: "+str); //str: test //без @ произойдёт зацикливание
}
```

2. Если вы вызываете нелокальную функцию (у экземпляра или статическую), то @@ указывает, что если в целевом классе нет нужной функции, то искать локальную, а если нет локальной, то базовую.

```
Foo f=new Foo();
f.@@Test("hello"); //hello // т.к. в Foo нет функции Test, то вызывается локальная Test
Foo.@@Test("world"); //world //то же самое для вызова статических функций
f.@@Test2("hello"); //Test2: hello // вызывается нестатическая Test2 в Foo
Foo.@@Test2("world"); //static Test2: world //вызывается статическая Test2 в Foo

Test(string s){WriteLine(s);}
Test2(string s){WriteLine(s);}

class Foo{
	public static Test2(string s){
		WriteLine("static Test2: "+s);
	}
	public Test2(string s){
		WriteLine("Test2: "+s);
	}
}
```

Вызов через @@ используется интерпретатором при перегрузке операторов для своих типов (*свои типы* - классы в вашем скрипте, а не типы .NET).

3. Если указать @@@, то если в целевом классе нет нужной функции, будет сразу искаться базовая (без поиска локальной). Вызов через @@@ используется интерпретатором при перегрузке операторов преобразования.

```
Bar b=new Bar();
Foo f=Foo\b; //Foo\b интерпретатор видит как Foo.@@@op_Casting(b). Можно писать: Foo f=(Foo)b;
//Если в Foo нет функции op_Casting, принимающей нужный тип, то вызывается встроенная функция op_Casting(Foo,b). Первым аргументом добавился путь-тип, хотя для всех других функций путь никогда не добавляется (op_Casting и op_ArrayCasting - исключения).
//Тут @@@ используются потому, что op_Casting должна находится в классе, в тип которого идёт преобразование, а перегрузка \ внешними функциями не поддерживается, в отличие от других операторов (чтобы не было неопределенности, к какому типу относится op_Casting).
```

4. При использовании стрелки @ указывает, что нужно искать член не у самого типа, а у типа типа (type.GetType()), т.е. у System.RuntimeType. Допустим, в DLL (C#) есть тип:

```
public class SimpleTest
{
    public static string ToString() => "Hello!";
}
```

Тогда в OverScript символ @ после стрелки (->) будет работать так:

```
const object SimpleTest=typeof("SimpleTestLib.dll", "SimpleTest");
WriteLine(SimpleTest->ToString()); //Hello!
WriteLine(SimpleTest->@ToString()); //SimpleTest
```

Т.е. без @ вызывается метод типа SimpleTest, а с @ метод типа typeof(SimpleTest).

Константы

Константы задаются на уровне класса или внутри метода, должны инициализироваться сразу после объявления.

```
const int X=10;
Test(); //10; 20

Test(){
	const int Y=20;
	WriteLine($"{X}; {Y}");
}
```

Константы можно объявлять в любом месте кода, т.к. строки объявлений вырезаются при загрузке скрипта. Далее обращения к константам заменяются на литералы.

```
const int X=10;
WriteLine(X);
//Код выше интерпретатор видит как:
//WriteLine(10);
```

В отличие от C# есть константы типа object.

```
const object X=typeof(int);  //typeof(int) - литерал
WriteLine(X); //System.Int32
```

В OverScript константу можно инициализировать не только литералом.

```
const object arr=new long[]{10L, 20L, 30L}; //значение после оператора присваивания вычислится при загрузке скрипта
WriteLine(arr+"; len: "+Length(arr)); //System.Int64[]; len: 3

const int x=5, y=2, z=x+y;
WriteLine(z); //7

const string str=ToUpper("test"); //можно использовать только встроенные функции, т.к. на момент вычисления констант свои функции ещё не загружены (вычисление констант происходит на раннем этапе загрузки кода)
WriteLine(str);//TEST
```

При инициализации константы можно ссылаться на константу в другом классе.

```
//const int X=Bar.X; //так не сработает, т.к. на момент вычисления констант унаследованные константы ещё недоступны
const int X=Foo.X;
WriteLine(X); //555

class Foo{
	public const int X=555;
}
class Bar:Foo{}
```

Инициализацию констант можно делать как оператором =, так и := (присваивание с конвертированием). Для констант они работают одинаково: если тип значения не совпадает с типом константы, то вычисляется новый литерал нужного типа.

```
const int A=123L; //из числа 123 типа long создастся новый литерал 123 типа int.
const long B=2+3; //создастся новый литерал 5 типа long.
const string C="test"; //обычное присваивание
const object D="test"; //создастся новый литерал типа object
WriteLine(object\C==D); //True - на самом деле значение обеих констант (обоих литералов) это один и тот же объект "test". object\C - то же, что и (object)C.
```

Операторы и управляющие конструкции

Операторы в порядке убывания приоритета: =, \, !, *, /, +, -, %, ~, &, ^, |, >, <, >>, <<, &&, ||, >=, <=, ++, --, &:=, ^:=, |:=, %:=, /:=, *:=, +:=, -:=, :=, &=, ^=, |=, %=, /=, *=, +=, ++, -=, --, !=, ==. Работают как в C#. Операторы являются функциями, которые можно перегружать (кроме присваивания и операторов с конвертированием). **Операторы выполняются строго по приоритету:** 9-2+3 будет 4, а не 10.
\ - в OverScript это оператор явного преобразования типов (пример: x = int \ y). Операторы, начинающиеся с двоеточия, делают автоматическое конвертирование (вместо x=int\y удобнее писать x:=y).

Как и в C#, операторы == и != могут сравнивать ссылки и значения.

```
object obj="hello";
string str="hello";
//WriteLine(obj==str); - так можно в C#, но нельзя в OverScript!
WriteLine(obj==object\str); //True //obj и str ссылаются на один объект-литерал
str="hel"+"lo"; //создаётся новый объект-строка
WriteLine(obj==object\str); //False //obj и str ссылаются на разные объекты
WriteLine(string\obj==str); //True //сравниваются строки, а не ссылки. string\obj - это как (string)obj в C#.
```

if-else

Оператор if проверяет условие и переводит выполнение на нужный блок.

```
if(5>4) Write("if true"); else Write("if false"); //if true
//интерпретатор при загрузке кода сам добавит фигурные скобки:
//if(5>4){Write("if true");}else{Write("if false");}
```

Есть else if:

```
int x=5;
if(x==1) Write("1");
else if(x==2) Write("2");
else if(x==5) Write("5"); //5
else Write("ELSE");
```

Примеры использования оператора !:

```
int x=0;
if(!x) WriteLine("if true"); else WriteLine("if false"); //if true //оператор ! для числа возвращает true, если число равно 0
string s="true";
if(!s) WriteLine("if true"); else WriteLine("if false"); //if false //оператор ! для строки возвращает true, если строка пустая или null
if(!!s) WriteLine("if true"); else WriteLine("if false"); //if true //два оператора ! - это так называемый оператор Bang Bang (Double Bang). Это именно два оператора !, а не один !!. Так можно проверить, что стока не пустая и не null.
```

switch

switch — это оператор выбора, который работает как в C#, но построен на словаре, в котором key - значения case, a value - строка кода, на которую переходить. Case-значения могут быть разных типов и при загрузке они конвертируются в тип значения в switch.

```
int x=5;
switch(x){
        case 2:
                WriteLine("case 2");
                break;
        case 5:
                WriteLine("case 5");
                break;
        default:
                WriteLine("default");
                break;
}

//Как сделать switch по датам:
date d:="05.10.2021"; //строка конвертируется в дату
switch(d){
        case "04.10.2021":
                WriteLine("case 04.10.2021");
                break;
        case "05.10.2021": //значения в case конвертируются в date
                WriteLine("case 05.10.2021"); //case 05.10.2021
                break;
        default:
                WriteLine("default");
                break;
}
```

switch может работать только с постоянными значениями (литералами/константами) и поддерживает выбор из типов:

```
object x=5L;
switch(GetType(x)){
	case int: // int, long и т.д. - постоянные значения (object-ы, хранящие тип)
		Write("Type is int");
		break;
	case long: //сработает этот case
		Write("Type is long");
		break;
	case typeof(byte): //так можно потому, что typeof(byte) - object-литерал
		Write("Type is byte");
		break;
}
```

for

Оператор for выполняет цикл, пока заданное условие равно true.

```
for(int i=0; i<5; i++) Write(i+" "); //0 1 2 3 4
```

В отличие от C#, for может содержать только три части: `for(счётчик; условие; изменение счётчика)`. Бесконечный цикл `for ( ; ; )` не поддерживается.

Для простых циклов лучше использовать foreach, т.к. в OverScript он быстрее.

foreach

Оператор foreach выполняет итерации по элементам массива или любой перечислимой коллекции.

```
int[] arr = new int[]{10, 20, 30};
foreach(int i in arr) Write(i+" "); //10 20 30 //i=555 не изменит значение в массиве
WriteLine();
//Использование с функцией Range:
foreach(int $i in Range(Length(arr))) Write(arr[i]+" "); //10 20 30
WriteLine();
//Пример перебора символов строки:
string s="Hello!";
foreach(char c in s) Write(c+" "); // H e l l o !
```

В отличие от C#, Dispose() для перечислителя не вызывается. Если нужно с вызовом Dispose(), то используйте функцию ForEach.

while, do-while

while и do-while работают как в C#.

```
int x=0;
while(x<5){
	Write(x+" ");
	x++;
}
//0 1 2 3 4
WriteLine();
x=0;
do{
	Write(x+" ");
	x++;
}while(x<5);
//0 1 2 3 4
```

Есть break и continue.

goto

Оператор goto переводит выполнение на указанную метку, которая может быть в любом месте функции (в C# более строго), поэтому использовать его нужно осторожно, чтобы не получить неожиданные ошибки.

```
for(int i=0; i<10; i++){
	for(int j=0; j<10; j++){
		if(i*j==15) goto end;
	}
}
end:
WriteLine(i+"; "+j); //3; 5
```

Можно делать переход на case:

```
int x=2;
switch(x){
        case 1:
                Write("1! ");
                break;
        case 2:
                Write("2! ");
                goto case 1;
        case 3:
                Write("3! ");
                break;
        default:
                WriteLine("default! ");
                //тут break; можно не писать, хотя в C# нужно
}
//2! 1!
```

lock

Как и в C#, инструкция lock получает взаимоисключающую блокировку заданного объекта перед выполнением определенных операторов, а затем снимает блокировку.

```
object obj=new object();
lock(obj){
        WriteLine("OK");
}
WriteLine("END");
```

using

Инструкция using вызывает Dispose() объекта IDisposable при выходе из блока.

```
using(Foo f=new Foo()){
        f.Test(); //Test!
} //Dispose!

ReadKey();

class Foo{
        public Test(){WriteLine("Test!");};
        Dispose(){WriteLine("Dispose!");}
}
```

Можно и так писать:

```
Foo f=new Foo();
using(f){
        f.Test();
}
```

Методы (функции)

Отличия от C#:

- Писать void необязательно.
- Есть модификаторы: public, private, static, fixed, exclusive, virtual, override, sealed. Все методы по умолчанию приватные (конструкторы публичные).
- Можно пропускать параметры, т.е. необязательные параметры могут быть не только в конце.

Есть перегрузка методов.

```
Test(1, , "hello"); // int, string, string: 1; foo; hello
Test(1, 2, 3); // int, int, int: 1; 2; 3
Test(1, , ); // int, string, string: 1; foo; bar
Test(1, 2); // int, int, string: 1; 2; test
Test(1, "7", ); // int, string, string: 1; 7; bar
Test( , , 9); // int, int, int: 10; 20; 9
Test(1); // int, string, string: 1; foo; bar
Test(); // int, string, string: 10; foo; bar
//Test("1", "2", "3"); // Function 'Test(string,string,string)' not found

Test(int x=10, int y=20, int z=30){
        WriteLine("int, int, int: "+x+"; "+y+"; "+z);
}
Test(int x=10, int y=20, string z="test"){
        WriteLine("int, int, string: "+x+"; "+y+"; "+z);
}
Test(int x=10, string y="foo", string z="bar"){
        WriteLine("int, string, string: "+x+"; "+y+"; "+z);
}
```

Для числовых типов делается нестрогая проверка типов.

```
Test(5L); //сработает, несмотря на то, что Test принимает тип short. Но если будет перегрузка с long, то будет вызвана она

Test(short v){WriteLine(v);}
```

Еще примеры:

```
Test(1, 2); // int, int: 1; 2
Test(1, 2.5); // int, float: 1; 2,5 //2.5 - это тип double, но сработает
Test(1, 2.5f); // int, float: 1; 2,5
Test(1.5, 2.5m); // int, float: 2; 2,5 //оба аргумента конвертированы в типы параметров (int и float)
//Test(long->MaxValue, 2.5m); //OverflowException: Value was either too large or too small for an Int32.

Test(int x, float y){
        WriteLine("int, float: "+x+"; "+y);
}
Test(int x, int y){
        WriteLine("int, int: "+x+"; "+y);
}
```

Параметры int и long типов могут принимать char (если нет перегрузки с char).

```
Test('w');

Test(int n){
        WriteLine(n); //119
}
```

Перегрузка работает и для методов принимающих экземпляры своих типов.

```
Foo f=new Foo();
Bar b=new Bar();
Baz bz=new Baz();

Test(f); // FOO!
Test(b); // BAR!
Test(bz); // FOO! //сработает, т.к. Baz наследует от Foo

Test(Foo f){WriteLine("FOO!");}
Test(Bar b){WriteLine("BAR!");}

class Foo{}
class Bar{}
class Baz:Foo{}
```

Аргументы можно передавать по ссылке (только **ref**, а out нет), но нельзя передавать элемент массива.

```
int x=5;
Test(ref x); //ref писать не обязательно
WriteLine(x); //555

//int[] arr=new int[]{10, 20, 30};
//Test(arr[1]); //не сработает!

Test(ref int v){
	v=555;
}
```

Значение по умолчанию параметра не обязательно должно быть константой.

```
Test();
Test(int x=TickCount()){
	WriteLine(x);
}
```

Можно делать несколько функций с одним именем и параметрами.

```
Foo.Test(); //Test 1

static class Foo{
	public static Test(){WriteLine("Test 1");}
	static Test(){WriteLine("Test 2");} //если добавить public, то будет вызываться именно эта функция (т.к. последняя)
}
```

Есть **params**:

```
Test("test", 1, 2, 3);
//s: test; values: 1 2 3
Test("test2", new int[]{1, 2, 3});
//s: test2; values: 1 2 3

Test(string s, params int[] values){
	WriteLine("s: "+s+"; values: "+Join(' ', values));
}
```

Method chaining

В современных объектно-ориентированных языках программирования есть возможность делать цепочки методов:

```
str=str.ToUpper().Trim().Replace("a","b");
```

Method chainig - общее название синтаксиса в ООП, в котором несколько методов вызываются один за другим. В OverScript **любой** метод можно использовать в цепочке методов. Значение слева от метода передаётся в метод первым аргуметом.

```
int x=10;
x=x.AddNum(5).AddNum(2);
WriteLine(x); // 17

int AddNum(int x, int y){
	return x+y;
}
```

Области видимости

Есть модификаторы public, private (только для функций) и static. Все переменные (поля), методы и вложенные классы являются приватными. Внутри методов переменные видны везде, а не только внутри цикла или условной конструкции, в которой они объявляются, и исчезают только после завершения работы метода (очищаются только ссылочные, а значимые потом перезаписываются новыми значениями).

```
for(int i=1; i<10; i++){int n=i;}
WriteLine(i+"; "+n); // 10; 9 //переменные остаются после выхода из цикла
```

Статические переменные можно объявлять как обычно (модификатором static), а можно объявлять в методе Static. Это статический конструктор, который вызывается при первом обращении к статическим членам.

```
SomeClass.Test();

class SomeClass
{
        Static(){ // статический конструктор. Модификаторы указывать не нужно.
                int x=5; //переменные указываются без static, но будут все статическими
        }
        public static Test(){
                WriteLine(x); // 5
        }
}
```

Через ссылку на экземпляр нельзя вызывать статические методы и обращаться к статическим полям. Обращаться к нестатическим членам из статических методов тоже нельзя. В статических классах могут быть только статические члены (методы и поля). Всё как в C#. Можно обращаться к статическим членам, которые находятся во внешних классах (в которые вложен текущий):

```
static string text="HELLO!";
Foo.Bar b = new Foo.Bar();
b.Test(); //HELLO!

class Foo{
        static PrintHello(){WriteLine(text);}

        public class Bar{
                public Test(){PrintHello();}
        }
}
```

Массивы

Массивы поддерживаются только одномерные. Массивы массивов не поддерживаются. Можно задавать содержимое массива используя инициализатор массива.

```
int[] arr = new int[3]; // создаётся int-массив с тремя элементами (0 - для чисел значение по умолчанию)
arr = new int[]{1, 2, 3}; // создаётся int-массив с элементами 1, 2, 3.
arr = {1, 2, 3}; // можно и так

byte[] arr2={1, 2, 3}; //числа int и они будут сконвертированы в byte
```

В массив типа object[] можно записывать любые значения, в том числе другие массивы.

```
object[] arr=new object[]{new int[]{1, 2, 3}, new string[]{"hello", "world"}};
int[] intArr:=arr[0]; //для того, чтобы присвоить переменной типа int[] значение типа object используется оператор :=
WriteLine(Join("; ",intArr)); // 1; 2; 3
string[] strArr:=arr[1];
WriteLine(Join("; ",strArr)); // hello; world
```

Создание двумерного массива, запись и чтение элементов через рефлекшн:

```
const object type=typeof("System.Int32[,]");
int w=3, h=2;
object arr=Create(type, h, w);
int v=0;
for(int row = 0; row<h; row++){
        for(int col = 0; col<w; col++){
                arr->SetValue(v++, row, col);
                WriteLine(row+"-"+col+": "+arr->GetValue(row, col));
        }
}
/*
0-0: 0
0-1: 1
0-2: 2
1-0: 3
1-1: 4
1-2: 5
*/
```

Классы

Классы создаются и используются как в C#. Только конструктор это New(), а финализатор - Finalize(), как в VB.

```
Car c = new Car();
c.PrintInfo(); // Model: Undefined; Seats: 0; Price: 0
c = new Car("BMW", 4, 12345.99);
c.PrintInfo(); // Model: BMW; Seats: 4; Price: 12345,99

Car[] cars = new Car[]{new Car("BMW", 4, 12345.99), new Car("Ford", 4, 9000.55), new Car("Ferrari", 2, 150579.79)};
WriteLine("All:\n"+cars); //при конвертировании массива cars в строку вызывается Car.ArrayToString(cars)
//All:
//Model: BMW; Seats: 4; Price: 12345,99
//Model: Ford; Seats: 4; Price: 9000,55
//Model: Ferrari; Seats: 2; Price: 150579,79

class Car
{
	string Model;
	int Seats;
	decimal Price;

	New() // конструктор по умолчанию (модификаторы не нужны)
	{
		Model="Undefined";
	}

	New(string model, int seats, decimal price) // конструктор
	{
		Model=model;
		Seats=seats;
		this.Price=price; //можно и this использовать
	}

	public PrintInfo(){ //модификатор public нужен, чтобы можно было вызвать извне, а void писать необязательно
		WriteLine(this); // при конвертировании this (текущий экземпляр класса) в строку будет вызвана ToString()
		//если ToString() нет, то строковое представление this будет: Instance of Car
	}

	Finalize() // финализатор (модификаторы не нужны)
	{
		//срабатывает при сборке мусора
	}

	//***** Функции ниже срабатывают автоматически, когда это необходимо *****

	string ToString(){ //используется при автоматическом конвертировании объекта в строку (модификатор public можно не
писать)
		return $"Model: {Model}; Seats: {Seats}; Price: {Price}";
	}

	//Функция ArrayToString используется при автоматическом конвертировании массива в строку
	static string ArrayToString(Car[] arr){ //в arr передаётся массив, который нужно конвертировать
		return Join("\n", arr); //в Join для каждого элемента вызывается ToString()
	}

	int GetHashCode(){ //возвращает хеш-кода объекта
		return 1234567890;
	}

	bool Equals(object obj){ //используется для проверки равенства объектов
		return obj==this;
	}
	//ToString, GetHashCode и Equals - в .NET это методы класса System.Object. Их можно переопределять в производных
классах. Одноимённые функции в OverScript именно это и делают.

	int CompareTo(object obj){ //эту функцию нужно прописывать, если предполагается использовать сортировку элементов
Car[], например, методом Sort(). Эта функция - реализация интерфейса IComparable.
		return 0; //возвращаемые значения: 0 - объекты равны, -1 - текщий экземпляр меньше, 1 - больше
	}

	Dispose(){ //срабатывает при освобождении ресурсов методом Dispose. Эта функция - реализация интерфейса
IDisposable.
		//код, выполняющий очистку используемых ресурсов.
	}
```

```
}
```

Методы ToString(), ArrayToString(), GetHashCode(), Equals(), CompareTo() и Dispose() при автоматическом вызове работают чуть медленнее (~15%), чем если бы они вызывались напрямую из-за создания нового колл-стека.

Инициализаторы объектов не поддерживаются.

```
Car c = new Car(){model='BMW', seats=4, price=12345.99}; // не сработает!
```

В OverScript оператором := можно присваивать переменной одного своего типа значение другого типа, но доступ тогда будет только к тем членам класса, которые есть в обоих классах. Для интерпретатора все экземпляры классов имеют тип CustomObject, а задание типа указывает интерпретатору, какие члены имеются у объекта.

```
Car c = new Car("BMW", 4, 20345.99);
Bike b = new Bike("Yamaha", 350, 15777.99);
WriteLine("Car: "+c.Model+"; "+c.Seats+"; "+c.Price); // Car: BMW; 4; 20345,99
WriteLine("Bike: "+b.Model+"; "+b.MaxSpeed+"; "+b.Price); // Bike: Yamaha; 350; 15777,99
b:=c; // так делать можно, но будут доступны только общие переменные
WriteLine("Bike-Car: "+b.Model+"; "+b.MaxSpeed+"; "+b.Price); // Bike-Car: BMW; 0; 20345,99
//WriteLine("Bike: "+b.Model+"; "+b.Seats+"; "+b.Price); //ScriptLoadingException: Variable 'Seats' not found at 'Bike'.

class Car
{
        public string Model; public int Seats; public decimal Price;
        New(string m, int s, decimal p){Model=m; Seats=s; Price=p;}
}

class Bike
{
        public string Model; public int MaxSpeed; public decimal Price;
        New(string m, int s, decimal p){Model=m; MaxSpeed=s; Price=p;}
}
```

В этом примере переменной типа Bike присваивается объект Car, у которого нет поля MaxSpeed. Поэтому значение MaxSpeed как бы равно 0, но если ранее переменной MaxSpeed в этой области видимости уже назначалось какое-то значение, то будет возвращено именно оно, а не 0 (значимые типы не очищаются сразу, а просто перезаписываются). Также возможно возникновение ошибки чтения переменной.

И ещё немного о технических нюансах: для каждого экземпляра своего класса создаётся объект типа ClassInstance, который хранит номер области видимости и ссылку на реализуемый класс, а значения переменных этой области видимости хранятся отдельно в общем хранилище (в массивах). Т.е. для CLR такие объекты не являются обычными, как например объект типа System.Drawing.Point, поэтому не нужно пытаться делать их сериализацию и десериализацию стандартными функциями .NET. Эти объекты существуют как абстракция в рамках OverScript. Их сериализация и десериализация (JSON) делается функциями ToJson, FromJson.

Наследование

Наследование в OverScript представляет собой копирование всех членов из указанных классов. Можно наследовать от нескольких классов. Копируются именно все, а не только публичные члены, как если бы они имели модификатор protected в C#. Унаследованные члены, в том числе статические методы (в отличие от C#), перекрывают ранее заданные. Конструкторы наследуются как обычные методы. Для вызова метода из базового класса нужно добавить $ перед именем метода.

```
Bar b=new Bar(5);
b.Test(); // Test of Bar: 5; 0
b=new Bar(5, 7);
b.Test(); // Test of Bar: 5; 7
b.$Test(); // Test of Foo: 5 // $ перед именем метода указывает, что нужно вызвать предыдущий метод, т.е. метод из класса Foo

class Foo{
        int x;
        New(int x){
                this.x=x;
        }
        public Test(){
                WriteLine("Test of Foo: "+x);
        }
}

class Bar:Foo{
        int y;
        New(int x, int y){
                this.x=x;
                this.y=y;
        }
        public Test(){
                WriteLine("Test of Bar: "+x+"; "+y);
        }
}
```

При наследовании сохраняются все версии методов. Для понимания, как работает вызов метода с $, рассмотрим простой пример:

```
Test(); // Test-3
$Test(); // Test-2
$$Test(); // Test-1

Test(){WriteLine("Test-1");}
Test(){WriteLine("Test-2");}
Test(){WriteLine("Test-3");}
```

Как видите, символ $ указывает, какой по порядку метод вызывать. Обычно вызывается последний, с $ - предпоследний, с $$ - предпредпоследний и т.д. Так и при наследовании, получается список методов, любой из которых можно вызывать, добавляя нужное кол-во $.

```
Baz bz=new Baz(2); // x=12
Bar b=new Bar(2); // x=4
Foo f=new Foo(2); // x=2

class Foo{
        New(int x){
                WriteLine("x="+x);
        }
}
class Bar:Foo{
        New(int x){
                $New(x*2); // вызов конструктора класса Foo
        }
}
class Baz:Bar{
        New(int x){
                $New(x*3); // вызов конструктора класса Bar
        }
}
```

Пример, в котором метод из базового класса вызывает метод из производного:

```
Bar b=new Bar();
b.Test(); // 2 // метод Test вызывает GetSomeVal из Bar
Foo f=new Foo();
f.Test(); // 1 // метод Test вызывает GetSomeVal из Foo

class Foo{
        public Test(){WriteLine(GetSomeVal());}
        int GetSomeVal(){return 1;}
}
class Bar:Foo{
        int GetSomeVal(){return 2;}
}
```

Переменные производного класса перекрывают переменные базового:

```
new Foo().Test(); //5+1+4: 10
new Bar().Test(); //7+2+9: 18

class Foo{
        int X=5;
        static int Y=1;
        const int Z=4;

        public Test(){
                WriteLine($"{X}+{Y}+{Z}: "+(X+Y+Z));
        }
}
class Bar:Foo{
        int X=7;
        static int Y=2;
        const int Z=9;
}
```

Такое перекрывание происходит потому, что при наследовании код функции копируется в класс, после чего функция собирается с текущим окружением. Чтобы этого не происходило, можно добавить модификатор **fixed**, и функция будет наследоваться как есть, без копирования. Это ускорит загрузку и уменьшит расход оперативной памяти.

```
new Foo().Test(); //X=5; Y=1; Z=4; FOO!
new Bar().Test(); //X=7; Y=1; Z=4; FOO! //без fixed было бы: X=7; Y=2; Z=9; BAR!
//константа, статическая переменная и функция GetSomeStr берутся из Foo, а не из Bar
//переменная X экземплярная, поэтому fixed её не касается

class Foo{
        int X=5;
        static int Y=1;
        const int Z=4;

        public fixed Test(){ //fixed указывает интерпретатору, что наследовать функцию нужно как есть, т.е. собранную в контексте Foo
                Write($"X={X}; Y={Y}; Z={Z}; ");
                WriteLine(GetSomeStr());
        }
        public string GetSomeStr(){
                return "FOO!";
        }
}
class Bar:Foo{
        int X=7;
        static int Y=2;
        const int Z=9;

        public string GetSomeStr(){
                return "BAR!";
        }
}
```

Пример множественного наследования (обратите внимание на порядок наследования, который не такой, как, например, в Python):

```
Baz bz=new Baz();
WriteLine(bz.X+"; "+bz.Y+"; "+bz.Z); // 1; 2; 3 // в Baz кроме Z есть X и Y из классов Foo и Bar.
bz.Test(); // Bar! // используется метод Test из Bar.
Qux q=new Qux();
q.Test(); // Foo! // используется метод Test из Foo.

class Foo{
        public int X=1;
        public Test(){WriteLine("Foo!");}
}
class Bar{
        public int Y=2;
        public Test(){WriteLine("Bar!");}
}
class Baz:Foo,Bar{ // Baz наследует сначала от Foo, а потом от Bar. Последним будет метод Test из Bar.
        public int Z=3;
}
class Qux:Bar,Foo{} // Qux наследует сначала от Bar, а потом от Foo. Последним будет метод Test из Foo.
```

Использовать множественное наследование не рекомендуется, т.к. интерпретатор пока никак не проверяет его на возможные накладки. Помните, что в OverScript наследование это простое копирование функций (переменные класса находятся в функциях Instance() и Static(), и копируются вместе с ними).

Запрет наследования метода делается модификатором **exclusive**:

```
Bar b=new Bar();
b.Test2(); //Test2 at Bar
//b.Test(); //ошибка: функция не найдена

class Foo{
    public exclusive Test(){WriteLine("Test at Foo");}
}
class Bar:Foo{
    public Test2(){WriteLine("Test2 at Bar");}
}
```

Можно делать **виртуальные методы**. Модификаторы те же, что и в C# (virtual, override, sealed):

```
Bar b=new Bar();
b.Test(); //At Bar
Foo f=b;
f.Test(); //At Bar

class Foo{
    public virtual Test(){WriteLine("At Foo");}
}

class Bar:Foo{
    public override Test(){WriteLine("At Bar");}
}
```

Обработка исключений

Обработка исключений работает как C#, но в catch нужно строкой указывать тип исключения. Имя исключения записывается в переменную exName, описание в exMessage, свой экземпляр в exObject (если в throw указать экземпляр своего типа), а само исключение в переменную exception. Декларировать эти переменные не нужно, т.к. они задекларированы по умолчанию (exName и exMessage - string, а exObject и exception - object).

```
try{
        int x=5/0;
}
catch("StackOverflowException"){ //"System." подставляется автоматически. Можно так:
catch(typeof("System.StackOverflowException"))
        WriteLine("Catch1: "+exMessage);
}
catch("DivideByZeroException"){
        WriteLine("Catch2: "+exMessage);
}
catch{
        WriteLine("Catch3: "+exMessage);
}
finally{
        WriteLine("Finally");
}

//Catch2: Attempted to divide by zero.
//Finally
```

То же самое, но с константами:

```
const object StackOverflowException=typeof("System.StackOverflowException");
const object DivideByZeroException=typeof("System.DivideByZeroException");

try{
        int x=5/0;
}
catch(StackOverflowException){
        WriteLine("Catch1: "+exMessage);
}
catch(DivideByZeroException){
        WriteLine("Catch2: "+exMessage);
}
catch{
        WriteLine("Catch3: "+exMessage);
}
finally{
        WriteLine("Finally");
}
```

Оператору throw можно передать экземпляр исключения.

```
try{
        throw(Create("System.Exception", "Hello World")); //можно без скобок: throw Create("System.Exception", "Hello
World");
}
catch("Exception"){ //перехват любых типов, наследующих от System.Exception.
        WriteLine(exName+"; "+exMessage); // Exception; Hello World
}
```

throw без операндов работает так же, как в C#, т.е. повторно создает исключение.

```
try{
        int x=5/0;
}
catch{
        throw;
}
//DivideByZeroException: Attempted to divide by zero.
```

Вместо передачи в throw объектов типа System.Exception и производных можно передать экземпляр своего класса (должен содержать string-переменные Name и Message).

```
try{
        throw new MyException("SomeError", "Some error occured");

}catch(MyException){
        WriteLine($"{exName}; {exMessage}; {exObject}"); //SomeError; Some error occured; Instance of MyException
        WriteLine((MyException\exObject).Name); //SomeError
        WriteLine(exception.GetType()); //OverScript.CustomThrownException //exception.GetType() - это GetType(exception)
        //Реальный тип исключения, если передаётся объект своего типа, всегда будет CustomThrownException
}
ReadKey();

class MyException{
        public string Name;
        public string Message;
        New(string name, string message){
                Name=name;
                Message=message;
        }
}
```

Можно не создавать экземпляр исключения, а просто указать имя и описание исключения. Будет создан экземпляр класса CustomThrownException с информацией о имени исключения и описании.

```
try{
        throw("SomeException", "Exception message"); //тут скобки обязательны
}
catch$("SomeException"){ //$ указывает, что нужно перехватить исключение по имени
        WriteLine(exName+"; "+exMessage); // SomeException; Exception message
}
```

Как видно из примера выше, исключения можно перехватывать по имени, причём любые исключения. Для своих типов исключений имя и описание берутся из переменных Name и Message. Для .NET-овских исключений имя - это имя типа (неполное). Перехват по имени производится, если не удалось перехватить по типу, т.е. сначала обрабатываются обычные catch (кроме catch{} по умолчанию), и только потом catch$.

```
try{
        int x=5/0;
}
catch$("DivideByZeroException"){ //этот сработает, если не будет блока catch("DivideByZeroException")
        WriteLine("Catched by name!");
}
catch("DivideByZeroException"){ //этот сработает, хоть и стоит ниже
        WriteLine("Catched by type!");
}
ReadKey();
```

Пример создания исключения с вложенным исключением, и последущим извлечением данных из него.

```
try{
        object innerException=Create("System.DivideByZeroException", "Нельзя делить на ноль!");
        object ex=Create("System.Exception", "Тест вложенного исключения", innerException);
        throw(ex);
}
catch{
        WriteLine(exception->Message); // "Тест вложенного исключения"
        innerException=exception->InnerException;
        WriteLine(innerException); // "System.DivideByZeroException: Нельзя делить на ноль!" - результат вызова
ToString().
        WriteLine(innerException->Message); // "Нельзя делить на ноль!"
}
```

Альтернативный способ перехвата исключений при вызове функции

После кода вызова любой функции можно добавить блок: `(значение_возвращаемое_при_исключении, условие_перехвата_исключения, кол-во_доп._попыток, условие_продолжения_доп._попыток, действие_перед_попыткой, ещё_действие, ещё_действие ...)`. Таких блоков может быть несколько. Назовём их *блоками значений при исключениях*. При возникновении исключения, как и при перехвате через try, данные об исключении записываются в переменные exName, exMessage, exObject, exception.

```
int x=GetSomeValue()(555); //если GetSomeValue() вызовет исключение, то вернуть 555.
x=GetSomeValue()(777, exName=="DivideByZeroException"); //если имя исключения равно "DivideByZeroException", то вернуть 777.
x=GetSomeValue()(777, exception.Is(typeof("System.DivideByZeroException"))); //с проверкой типа исключения. Можно и так: exception.Is("System.DivideByZeroException"), но лучше с typeof, чтобы не получать тип каждый раз
x=GetSomeValue()(0, exName=="DivideByZeroException", 2); //если имя "DivideByZeroException", то вернуть 0 после двух дополнительных попыток вызова GetSomeValue().
x=GetSomeValue()(777, exName=="DivideByZeroException")(555, exName=="IndexOutOfRangeException"); //если имя "DivideByZeroException", то вернуть 777, а если "IndexOutOfRangeException", то вернуть 555.
x=GetSomeValue()(777, exName=="DivideByZeroException")(555, exName=="IndexOutOfRangeException")(123); //вернуть 123, если имя исключения не "DivideByZeroException" или "IndexOutOfRangeException".
x=GetSomeValue()(555, , 2); //вернуть 555 после двух доп. попыток при любых исключениях
ReadKey();

int GetSomeValue(){
        return 5/0;
}
```

Если вы указали условие перехвата ошибки, а оно не выполняется, то возврата значения не произойдёт и перехвата ошибки не будет.

```
try{
        int x=GetSomeValue()(555, exName=="IndexOutOfRangeException"); // исключение будет перехваченно не блоком значения при исключении, а блоком try, как обычно
        WriteLine(x);
}
catch{
        WriteLine("Caught exception: "+exName); //Caught exception: DivideByZeroException
}
ReadKey();

int GetSomeValue(){
        return 5/0;
}
```

Пример, как делать доп. попытки пока выполняется некоторое условие. Кол-во попыток не указываем, а указываем условие n++ < 3. Если же указать кол-во попыток, то больше заданного кол-ва попытки делаться не будут.

```
int n=0;
int x=GetSomeValue()(555, , , n++ < 3, WriteLine($"Дополнительная попытка {n}..."));
WriteLine(x);
//Дополнительная попытка 1...
//Дополнительная попытка 2...
//Дополнительная попытка 3...
//555

ReadKey();

int GetSomeValue(){
        return 5/0;
}
```

Теперь пример, как спрашивать, делать ли дополнительные попытки.

```
int x=GetSomeValue()(555, , 3, ReadKey($"Ошибка '{exName}'! Попробовать ещё раз? (Y/N)",,true).ToString()=="Y",
WriteLine("Дополнительная попытка..."));
WriteLine(x);
//Ошибка 'DivideByZeroException'! Попробовать ещё раз? (Y/N)y
//Дополнительная попытка...
//Ошибка 'DivideByZeroException'! Попробовать ещё раз? (Y/N)y
//Дополнительная попытка...
//Ошибка 'DivideByZeroException'! Попробовать ещё раз? (Y/N)y
//Дополнительная попытка...
//555
ReadKey();

int GetSomeValue(){
        return 5/0;
}
```

Этот способ перехвата ошибок работает и для функций, которые ничего не возвращают (на самом деле, они всегда возвращают null).

```
object err=Test()(exception); //если Test() выдаст исключение, то в err будет записано исключение, а если нет, то null.
if(err!=null) WriteLine("An error occured: "+exMessage); //An error occured: Attempted to divide by zero.
ReadKey();

Test(){
        int x=5/0;
}
```

Т.к. все операторы (кроме = и :=) являются функциями то, к ним тоже можно применять такой перехват ошибок.

```
int x=(5/0)(0); //само выражение нужно взять в скобки. Интепретатор видит это так: op_Division(5,0)(0).
WriteLine(x); //0

x=((4+1)/(1-1))(10); //выражение может быть составным. Интепретатор видит это так: op_Division(4+1,1-1)(10).
WriteLine(x); //10

x=5/(2/0)(1); //2/0 вызывает исключение с перехватом и возвратом 1, и общее выражение получается 5/1
WriteLine(x); //5
```

Т.к. стрелочки (->) это синтаксический сахар для Reflection-функций, то для них тоже будет работать:

```
string s="test"->Substring(9)("error");
WriteLine(s); //error
```

```
const object Environment=typeof("System.Environment");
WriteLine((Environment->TickCount)(-1)); //будет работать
//Посмотрим, чем в действительности является Environment->TickCount
WriteLine(Expr((Environment->TickCount)(-1))); //(@TGetValue(#Int32 TickCount#,int))(-1) //это функция TGetValue, которая через рефлекшн-функцию PropertyInfo.GetValue() возвращает значение свойства TickCount, объект MemberInfo которого передаётся в TGetValue литералом (#Int32 TickCount# - это так выводится значение object-литерала: #литерал.ToString()#)

//А вот следующий код при загрузке выдаст ошибку
//WriteLine((int->MaxValue)(-1)); //ошибка потому, что int->MaxValue при загрузке заменяется в коде на литерал (2147483647), как и все константы, а применять блок значения при исключении можно только к функциям.
```

Вообще-то применять блок значения при исключении можно с любым выражением, если обернуть его в функцию _(выражение), которая просто возвращает значение первого аргумента (и вычисляет остальные, если есть). Пример получения значения из массива по индексу, выходящему за границы:

```
int[] arr={10, 20, 30};
//int x=arr[9](-1); //так нельзя, arr[9] не функция
int x=_(arr[9])(-1); //а так можно, т.к. _(arr[9]) это функция
WriteLine(x); //-1
//Посмотрим на сообщение об ошибке, перехваченной блоком (-1)
WriteLine(exMessage); //Array of size 3 does not contain element with index 9.
```

Если значение, которое нужно вернуть, пропустить, то будет выброшено исключение. Пример, после двух неуспешных дополнительных попыток выбросить исключение:

```
int x=GetSomeValue()(,, 2);
//GetSomeValue...
//GetSomeValue...
//GetSomeValue...
//DivideByZeroException: Attempted to divide by zero.

int GetSomeValue(){
        WriteLine("GetSomeValue...");
        return 5/0;
}
```

Исключение можно выбросить и так:

```
exception=null; //нужно обнулить потому, что в ней может храниться ранее выброшенное исключение, а блок значения при исключении не обнуляет exception, если исключения не выбрасывается
int x=(5/0)(-1);
if(exception!=null) throw;
//DivideByZeroException: Attempted to divide by zero.
WriteLine(x);
```

Теперь подробнее про то, как обрабатываются ошибки при доп. попытках, если есть несколько блоков значений при исключениях с разными условиями во втором аргументе. Когда при доп. попытке возникает ошибка, то поиск блока делается как и после первой (основной) попытки, т.е. начиная с первого блока. Если выбранный блок не тот же, что был до этого, то счётчик попыток обнуляется. При этом может возникнуть ситуация бесконечных попыток, если функция выдаёт каждый раз разные исключения:

```
int x=Test()(, exName=="EvenEx", 1)(, exName=="OddEx", 1);
WriteLine(x);

int n;
int Test(){
        n++;
        WriteLine("try "+n);
        if(n%2==0) throw("EvenEx", "Even number");
        else throw("OddEx", "Odd number");
}
```

Ограничить общее кол-во попыток можно так:

```
int tryes=5; //максимальное кол-во дополнительных попыток
int x=Test()(, exName=="EvenEx", 1, (tryes--)>0)(, exName=="OddEx", 1, (tryes--)>0); //вместо (tryes--)>0 можно написать --tryes>=0
WriteLine(x);
...
```

Вернуть значение, но с выводом сообщения об ошибке:

```
int x=(5/0)(_(-1, WriteLine("Error: "+exMessage)));
WriteLine(x);
//Error: Attempted to divide by zero.
//-1
```

Выбросить исключение функцией Throw:

```
int x=(5/0)(int\Throw("SomeError", "Error message")); // int\ нужно формально, чтобы при загрузке кода прошла проверка типа (возвращаемое значение должно соответствовать типу функции, к которой применяется блок значения при исключении). Приведение типа не будет производиться, т.к. до него будет выбрашено исключение. Нужна именно функция Throw, а не инструкция throw, т.к. выражения не должны содержать инструкций.
//SomeError: Error message
```

Блок значения при исключении не поддерживается при задании констант.

```
const int x=(5/0)(-1); //ошибка
```

Директивы препроцессора

#include "путь", **#include** <путь> - Указывают препроцессору включить содержимое указанного файла в точку, где находится директива. Форма в кавычках: препроцессор ищет включаемые файлы в том же каталоге, что и файл, содержащий #include инструкцию. Форма с угловыми скобками: препроцессор ищет включаемые файлы в папке modules, которая находится рядом с OverScript.dll.

#import "путь", **#import** <путь> - Указывают загрузить библиотеку (dll) и добавить из неё базовые функции. Можно задать префикс для добавляемых функций: **#import** "путь" as префикс. Без него функции будут добавляться со своими исходными названиями, и возможны накладки, если в разных библиотеках есть функции с одинаковыми именами. С префиксом функция будет доступна как: *префикс@имя_функции*.

Пример:

```
#import "Lib.dll" as MyLib
//из Lib.dll, которая лежит рядом со скриптом, добавляются все функции
//допустим, в Lib.dll есть функция SomeFunc(), тогда она будет доступна так:
MyLib@SomeFunc(); //MyLib - префикс, который добавляется к имени функции при импорте
```

Для того, чтобы префикс был без @, нужно в as дописать * после префикса: as MyLib_*:

```
#import "Lib.dll" as MyLib_*
MyLib_SomeFunc(); //MyLib_ - префикс, а @ писать не нужно
```

У этой директивы есть следующие параметры:

- override - указывает, можно ли перезаписывать ранее добавленные функции (override=true). По умолчанию false.
- from - указывает класс, в котором искать функцию импорта (from=MainClass). По умолчанию берётся класс с атрибутом [Import].

- funcs - позволяет задать имена функций для импорта, если не нужно импортировать все (funcs=SomeFunc|Test|Hello). Работает как фильтр, т.е. если функции нет, то ошибки не будет.
- call - задаёт имена функций импорта, которые нужно вызвать (call=AddFuncs|AddExtraFuncs). По умолчанию вызывается с атрибутом [Import].

Пример, загрузить из Lib.dll три функции (SomeFunc, SomeFunc2 и SomeFunc3) с разрешением на перезапись:

```
#import "Lib.dll" as MyLib, funcs=SomeFunc|SomeFunc2|SomeFunc3, override=true
```

Пример, в Lib.dll в классе Class1 вызовать методы импорта AddFuncs и AddFuncs2:

```
#import "Lib.dll" as MyLib, from=Class1, call=AddFuncs|AddFuncs2
```

Как происходит импорт, и как делать свои dll

При загрузке скрипта ищутся директивы #import, и для каждой загружается указанная dll, после чего в классе с атрибутом [Import] вызывается функция импорта с атрибутом [Import], которая добавляет функции из библиотеки в список всех базовых функций. Пример dll на C# (в проект добавлена ссылка на OverScript.dll):

```
using System;
using OverScript;

namespace OSLibExample
{
    [Import]
    public class MainClass
    {
        [Import]
        public static void AddFuncs(Action<string, BF> addFunc) //этод метод вызывается для импорта функций
        {
            //addFunc(имя_базовой_функции, объект_с_данными_о_функции) - добавляет вашу функцию к базовым функциям OverScript
            addFunc("Concat", new BF().Add(StrConcat_string_string, TypeID.String, TypeID.String) //добавление функции Concat к базовым функциям. new BF() - это создание новой базовой функции, а метод Add добавляет делегат с указанием типов параметров функции. Можно добавлять несколько перегрузок.
                                    .Add(IntConcat_int_int, TypeID.Int, TypeID.Int) //добавление к создаваемой базовой функции Concat перегрузки для сложения int-чисел
                                    );
            addFunc("PrintHelloWorld", new BF().Add(PrintHelloWorld)); //добавляем простой метод, который выводит "Hello, world!"

            //после вызова AddFuncs() директивой #import функции Concat(string, string) и PrintHelloWorld() будут доступны в вашем OverScript-приложении
        }
        //ниже сама функция, которую мы хотим добавить в OverScript. Имя может быть любое, а параметры именно такие
        static string StrConcat_string_string(EvalUnit[] fnArgs, int scope, ClassInstance srcInst, ClassInstance inst, CallStack cstack, EvalUnit csrc)
        {
            //fnArgs - передаваемые функции аргументы. Их значения ещё не вычислены.
            //scope - область видимости, в которой вызывается функция
            //srcInst - не используется
            //inst - экземпляр, из которого делается вызов
            //cstack - стек вызовов
            //csrc - выражение, из которого идёт вызов

            string lstr = fnArgs[0].EvalString(scope, inst, cstack); //вычисляем первый аргумент
            string rstr = fnArgs[1].EvalString(scope, inst, cstack); //вычисляем второй аргумент
            return lstr + rstr;
        }
        static int IntConcat_int_int(EvalUnit[] fnArgs, int scope, ClassInstance srcInst, ClassInstance inst, CallStack cstack, EvalUnit csrc)
        {
            int lval = fnArgs[0].EvalInt(scope, inst, cstack);
            int rval = fnArgs[1].EvalInt(scope, inst, cstack);
            return lval + rval;
        }
        //для методов, которые ничего не возвращают, нужно писать тип object, а не void, т.к. все методы должны соответствовать единому делегату FuncToCall<T>.
        static object PrintHelloWorld(EvalUnit[] fnArgs, int scope, ClassInstance srcInst, ClassInstance inst, CallStack cstack, EvalUnit csrc)
        {
            Console.WriteLine("Hello, world!");
            return null;
        }
    }
}
```

Положите dll рядом с вашим скриптом, в котором проверим импорт функций так (допустим, файл dll называется OSLibExample.dll):

```
#import "OSLibExample.dll"
WriteLine(Concat("foo", " bar")); //foo bar
WriteLine(Concat(2, 3)); //5
PrintHelloWorld(); //Hello, world!
```

#primebase "путь", **#primebase** <путь> - В пределах файла указывают препроцессору базовый путь для относительных путей. Прописывать директиву нужно вне функций. Если путь не указывать, то будет взят путь текущего файла (в котором находится директива).

#base "путь", **#base** <путь> - Как #primebase, но работает в пределах функции, в которой находится. Есть ещё функция SetBasePath(путь).

```
#primebase @"C:\docs\test"
WriteLine(BasePath()); //C:\docs\test
string text=ReadAllText("file.txt"); //считает данные из C:\docs\test\file.txt
Test();
ReadKey();

Test(){
        #base @"C:\docs\test\sub"
        //#base внутри функции переопределяет #primebase
        WriteLine(BasePath()); //C:\docs\test\sub
        string text=ReadAllText("file.txt"); //считает данные из C:\docs\test\sub\file.txt
}
```

В разработке: #using "путь", **#using** <путь> - Указывают препроцессору включить файл с кодом как класс на самый верхний уровень (где главный класс). Т.е. если включается файл, содержащий директиву "#app MyLib", то включенный класс будет MyLib, и будет два верхних класса: App и MyLib.

Явное преобразование типов

В OverScript преобразование типов делается оператором \ (обратный слеш). Работает как приведение типа в C#. Если применяется к object-у, то делается распаковка. По моему мнению, писать byte\x удобнее, чем (byte)x в C#. Такая форма легче читается в сложных выражениях. Например, в C# будет: ((Foo)obj).Test(), а в OverScript: (Foo\obj).Test(). Можно использовать и классический синтаксис.

```
WriteLine(byte\2.55); //2 // byte\2.55 - то же самое, что (byte)2.55 в C#.
WriteLine((byte)2.55); //то же самое, что и byte\2.55.

object obj=5; //упаковка int.
WriteLine(short\int\obj); //5 // в C# это (short)(int)obj, т.е. сначала распаковать значение int, а потом преобразовать в short. Чтобы сразу конвертировать то: ToShort(obj).

WriteLine(float\int\7.77); //7 //можно делать сразу несколько преобразований. В этом примере сначала 7.77 типа double конвертируется в int, а потом в float.

WriteLine(date\); //01.01.0001 0:00:00 // date\ - это как бы преобразование ничего в date, и возвращает значение по умолчанию (в C#: default(DateTime)).
WriteLine(byte\); //0
WriteLine(string\ == null); //True //строка это ссылочный тип, и будет null, а не "" (пустая строка).

Bar b=new Bar();
Foo f=Foo\b; //сработает потому, что Bar наследует от Foo.
WriteLine(GetType(f)); //App.Bar // реальный тип остаётся Bar.
//b=Bar\f; //при загрузке кода выдаст ошибку "Can't cast object...", т.к. Foo не наследует от Bar.
b=Bar\object\f; //а вот так сработает, т.к. проверка типа при загрузке делаться не будет, а в переменной f находится именно экземпляр Bar.

class Foo{}
class Bar:Foo{}
```

Работа с типами: typeof, GetType, Is, As

Получить экземпляр System.Type указанного типа (для базовых типов - int, string, byte[] и т.д.) можно оператором typeof, который во время загрузки кода заменяется на object-литерал. Функция GetType возвращает тип объекта.

```
WriteLine(typeof(int)); //System.Int32
WriteLine(typeof(int[])); //System.Int32[]
//Тип можно получить и так:
WriteLine(decimal); //System.Decimal //это уже не литерал
WriteLine(byte->MaxValue); //255 // -> это синтаксический сахар для получения свойства/поля через Reflection
```

Для своих типов возвращается не System.Type, а CustomType:

```
WriteLine(typeof(Foo)); //App.Foo //литерал
WriteLine(Foo); //App.Foo //то же самое, что и typeof(Foo), только для интерпретатора это не литерал, а просто постоянное значение
WriteLine(GetType(typeof(Foo))); //OverScript.CustomType
WriteLine(GetType(Foo)); //OverScript.CustomType

class Foo{}
```

Получить любой .NET-тип можно так:

```
WriteLine(typeof("System.Text.Encoding")); //System.Text.Encoding //в отличие от C# тип задаётся строкой, т.к. если задать typeof(System.Text.Encoding), то тип будет искаться как класс в вашем скрипте, а про то, что это .NET-тип интерпретатор не знает
WriteLine(typeof("System.Collections.Generic.List`1[System.String], System.Collections"));
//System.Collections.Generic.List`1[System.String]
WriteLine(typeof("System.Drawing.Point, System.Drawing")); //System.Drawing.Point //через запятую указывается, где искать тип
WriteLine(typeof("SomeLib.dll", "SomeLib.SomeClass"));//SomeLib.SomeClass //тип будет загружен из сборки SomeLib.dll (из папки, в которой находится скрипт)
WriteLine(GetTypeByName("System.Text.StringBuilder")); //System.Text.StringBuilder //так можно получить тип не при загрузке кода, а во время выполнения
```

О том, как задаются строки-имена типов, можно почитать в описании метода Type.GetType. Если у вас выдаёт ошибку, что тип не найден, то проверьте в C# свойство AssemblyQualifiedName нужного типа. Например: `typeof(System.Console).AssemblyQualifiedName` вернёт `"System.Console, System.Console, Version=5.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a"`. Это значит, что в OverScript получить этот тип можно так: `typeof("System.Console, System.Console")`.

Вместо операторов as и is есть функции As и Is:

```
object obj=5;
WriteLine(Is(obj, int)); //True
WriteLine(As(obj, short)); //5 // As в OverScript может конвертировать значимые типы

object x=As(obj, typeof("System.UInt64")); //преобразование в UInt64 (ulong); Можно и так: ChangeType(obj, typeof("System.UInt64"))
WriteLine(x+"; "+GetType(x)); //5; System.UInt64

obj=new Bar();
WriteLine(Is(obj, Bar)); //True
WriteLine(Is(obj, Foo)); //True
WriteLine(Is(obj, Baz)); //False

WriteLine(As(obj, Bar)!=null); //True
WriteLine(As(obj, Foo)!=null); //True
WriteLine(As(obj, Baz)!=null); //False

class Foo{}
class Bar:Foo{}
class Baz{}
```

Перегрузка операторов

Все операторы, кроме = и операторов с конвертированием (:=, *:=, +:= и др.), можно перегружать. Операторы с конвертированием перегружаются автоматически при перегрузке соответствующих обычных операторов.

Имена операторов

!	LogicalNot	&	BitwiseAnd
~	OnesComplement	^	ExclusiveOr
++	Increment	|	BitwiseOr
--	Decrement	<	LessThan
+	Addition	<=	LessThanOrEqual
-	Subtraction	>	GreaterThan
*	Multiply	>=	GreaterThanOrEqual
/	Division	==	Equality
%	Modulus	!=	Inequality
<<	LeftShift	&&	LogicalAnd
>>	RightShift	||	LogicalOr
??	Coalescing	\	Casting/ArrayCasting

Имя метода перегрузки это: "op_"+имя оператора.

```
WriteLine(2+3); //10 //при загрузке этот код превращается в: WriteLine(op_Addition(2, 3));

static int op_Addition(int x, int y){
	return @op_Addition(x, y)*2; //@ перед именем функции указывает, что нужно вызвать базовую функцию (для каждого оператора есть своя функция)
}
```

Пример перегрузки ++:

```
int v=2;
WriteLine(++v); //4
WriteLine(++v); //6

static int op_Increment(int x){
	return x+2;
}
```

Унарными операторами фактически являются только ++, -- и \. Поэтому, например, LogicalNot (!) можно использовать так:

```
WriteLine(5!2); //True
WriteLine(5!5); //False

static bool op_LogicalNot(int x, int y){
	return x!=y;
}
```

При перегрузке операторов для своих типов, интерпретатор сначала ищет нужную функцию в самом классе (в классе первого операнда):

```
Foo foo1=new Foo(2);
Foo foo2=new Foo(3);
Foo foo3=foo1+foo2; //foo1+foo2 конвертируется в Foo.@@op_Addition(foo1, foo2); //@@ указывает, что если в Foo нет функции op_Addition, то искать её, как если бы она была просто: op_Addition(foo1, foo2);

WriteLine(foo3); //5

class Foo{
	public int X;
	New(int x){X=x;}
	string ToString(){return ToString(X);}

	public static Foo op_Addition(Foo a, Foo b){
		return new Foo(a.X+b.X);
	}
}
```

Функция для оператора может быть и вне класса:

```
Foo foo1=new Foo(2);
Foo foo2=new Foo(3);
Foo foo3=foo1+foo2;
WriteLine(foo3); //5


static Foo op_Addition(Foo a, Foo b){
        return new Foo(a.X+b.X);
}

class Foo{
        public int X;
        New(int x){X=x;}
        string ToString(){return ToString(X);}
}
```

Функции операторов могут быть нестатическими (будут работать только внутри класса):

```
Foo f=new Foo(1);
WriteLine(f.Test(2, 3)); //5
f=new Foo(2);
WriteLine(f.Test(2, 3)); //10

class Foo{
        public int X;
        New(int x){X=x;}
        public int Test(int x, int y){return x+y;}

        int op_Addition(int x, int y){
                return @op_Addition(x, y)*X;
        }
}
```

Оператор преобразования (\) перегружается так:

```
object v=5;
WriteLine(int\v); //10

static int op_IntCasting(object x){ //к имени функции нужно добавлять имя типа, в который делается преобразование
        return ToInt(x)*2;
}
```

Перегрузка неявных преобразований пока не поддерживается.

А вот пример, как оператором \ конвертировать массив short[] в byte[]:

```
short[] arr=new short[]{10, 20, 30};
byte[] arr2=byte[]\arr;
WriteLine(Join(", ", arr2)); //10, 20, 30

static byte[] op_ByteArrayCasting(short[] arr){ //к Casting нужно добавлять в начало имя типа: ByteArray для byte[],
StringArray для string[] и т.д.
        return byte[]\ConvertArray(arr, byte);
        //а можно так:
        //return Select(arr, short v, ToByte(v)); //в C# было бы: arr.Select(v => Convert.ToByte(v)).ToArray()
}
```

Теперь пример преобразования из одного своего типа в другой:

```
Foo f=new Foo(5);
Bar b=Bar\f;
b.PrintY(); //5

class Foo{
        public int X;
        New(int x){X=x;}
        public PrintX(){WriteLine(X);}
}
class Bar{
        public int Y;
        New(int y){Y=y;}
        public PrintY(){WriteLine(Y);}

        public static Bar op_Casting(Foo f){ //имя функции - просто op_ + Casting
                return new Bar(f.X);
        }
}
```

Преобразование массива одного своего типа в другой:

```
Foo[] f=new Foo[]{new Foo(10), new Foo(20), new Foo(30)};
Bar[] b=Bar[]\f;
foreach(Bar item in b) item.PrintY();
//10
//20
//30

class Foo{
        public int X;
        New(int x){X=x;}
        public PrintX(){WriteLine(X);}
}
class Bar{
        public int Y;
        New(int y){Y=y;}
        public PrintY(){WriteLine(Y);}

        public static Bar[] op_ArrayCasting(Foo[] arr){ //имя функции - op_ + ArrayCasting
                return Select(arr, Foo f, new Bar(f.X));
        }
}
```

Функции преобразования (op_Casting и op_ArrayCasting) должны возвращать только экземпляры класса, в котором они находится. Т.е. эти функции должны быть прописаны в том классе, в тип которого происходит преобразование. Другими словами, тип должен знать, как преобразовывать в себя, а его преобразования в другие типы должны быть прописаны в соответствующих классах. Вот пример с двумя типами, каждый из которых умеет преобразовывать другой в себя:

```
Foo f=new Foo(123);
Bar b=Bar\f;
//Casting Foo to Bar
f=Foo\b;
//Casting Bar to Foo

class Foo{
	public int X;
	New(int x){X=x;}

	public static Foo op_Casting(Bar b){
		WriteLine("Casting Bar to Foo");
		return new Foo(b.Y);
	}
}
class Bar{
	public int Y;
	New(int y){Y=y;}

	public static Bar op_Casting(Foo f){
		WriteLine("Casting Foo to Bar");
		return new Bar(f.X);
	}
}
```

Базовую функцию преобразования можно вызывать так:

```
Foo f=op_Casting(Foo, obj); //первый аргумент - тип, в который делать преобразование.
//Если тип - массив, то:
Foo[] arr=op_ArrayCasting(Foo[], obj);
```

Потоки, задачи, таймеры

Потоки, задачи и таймеры могут быть локальными и обычными. Обычные создаются функции или из ссылки на функцию, а локальные прямо из выражения. Локальные могут работать с переменными из области создания. Простой пример для понимания разницы:

```
Test();
ReadKey();

Test(){
	string s="Hello!";
	object t=LocalThread(PrintStr(s));
	StartThread(t);
	//str: Hello!
	Wait(t); //если убрать ожидание завершения потока, то произойдёт выход из метода с очисткой всех его локальных переменных, и если поток не успеет до этого считать значение s, то он выведет "str: "
}

int PrintStr(string str){
	WriteLine("str: "+str);
}
```

Из примера выше видно, что локальный поток считывает переменные из той области, в которой он был создан. Теперь пример с обычным потоком, который создаётся из функции:

```
Test();
//str: Hello!
//Всё сработало как надо
ReadKey();

Test(){
	string s="Hello!";
	object t=Thread(PrintStr(""));
	StartThread(t, s); //запуск потока с передачей в него значения переменной s
	//Поток будет выполнен уже после выхода из Test()
}

int PrintStr(string str){
	WriteLine("str: "+str);
}
```

С Task-ами то же самое, только аргументы передаются при создании задачи:

```
Test();
//str: Hello!
ReadKey();

Test(){
	string s="Hello!";
	object t=Task(PrintStr(""), s);
	StartTask(t); //запуск задачи, в которой уже находится значение переменной s
}

int PrintStr(string str){
	WriteLine("str: "+str);
}
```

В локальных потоках, задачах и таймерах можно выполнять любые выражения:

```
string s;
object task=LocalTask(s="Hello!");
StartTask(task);
WriteLine(TaskResult(task)); //Hello! //это результат выражения s="Hello!"
WriteLine(s); //Hello! //задача изменила значение переменной s
ReadKey();
```

Вот простой пример одновременного выполнения двух выражений:

```
int x=2, y=5;
object task=LocalTask(x=SomeLongCalculation(x));
object task2=LocalTask(y=SomeLongCalculation(y));
StartTask(task);
StartTask(task2);
Wait(task);
Wait(task2);
//Вместо двух Wait-ов можно написать: WaitAll(new object[]{task, task2});

//Delay 1851 for 2
//Delay 1117 for 5
//Continuation for 5
//Continuation for 2

WriteLine(x+"; "+y);  //4; 10
ReadKey();

int SomeLongCalculation(int x){
	int delay=Rand(1000, 3000);
	WriteLine("Delay "+delay+" for "+x);
	Sleep(delay);
	WriteLine("Continuation for "+x);
	return x*2;
}
```

Если, например, в LocalTask нужно выполнить несколько выражений, то можно поместить их в @(выражение-1, выражение-2, выражение-3):

```
int a, b, c;
object task=LocalTask(@(a=2, b=a+3, c=a+b)); //функция @(...) выполняет по порядку все аргументы
StartTask(task);
Wait(task);
WriteLine($"a: {a}; b: {b}; c: {c}"); //a: 2; b: 5; c: 7
ReadKey();
```

Локальный таймер работает аналогично локальным потокам и задачам:

```
int x;
object t=LocalTimer(WriteLine(x++), 1000, true);
WriteLine("Нажмите любую клавишу, чтобы остановить таймер");
//Нажмите любую клавишу, чтобы остановить таймер
//0
//1
//2
ReadKey(true);
StopTimer(t);
WriteLine("x: "+x); //x: 3
ReadKey();
```

Reflection через стрелку

В OverScript стрелка (->) при загрузке кода заменяется на вызов reflection-функций.

```
WriteLine("test"->ToUpper()); //TEST
//Посмотрим, как видит стрелку интерпретатор:
WriteLine(Expr("test"->ToUpper())); //@TInvoke(#System.String ToUpper()#,string,"test")
```

Подробнее об использовании стрелки.