Быстрый старт

admin
December 2021 edited March 31

OverScript - статически типизированный язык с си-подобным синтаксисом, использующий стандартные .NET-типы и принципы классического ООП.

1. Инструкции должны заканчиваться точкой запятой.
2. Переменные нужно объявлять указывая тип. Есть вывод типов. Если значение не указывать, то переменная инициализируется значением по умолчанию. Поддерживаются одномерные массивы.

```
int a;
string b, c=new string('w', 3);
double d=2.5, e=4.99;
byte[] arr=new byte[]{10, 20, 30};
var x=5L; //mun x - long
```

4. Нет блочной области видимости переменных.
5. Функции НЕ являются объектами и привязаны к классу, в котором находятся. Есть возможность свободного вызова функции на экземплярах любого типа (CallOn). Поддерживается method chaining для любых функций:

```
WriteLine("test".ToUpper().Substring(2)); //ST
//строка выше то же, что и:
WriteLine(Substring(ToUpper("test"), 2)); //ST
```

6. Обращение к статическим и экземплярным членам делаются одинаково - точкой.

```
Foo f=new Foo();
int x=f.X;
int y=Foo.Y;
class Foo{
    public int X=123;
    public static int Y=777;
}
```

7. Значимые и ссылочные типы работают так же, как в C#. Все базовые типы соответствуют типам C# (являются ими). Тип date - это DateTime (значимый тип). Ссылочные только string, object и массивы.
8. Литералы поддерживаются следующих типов: object, string, char, int, float, double, decimal.

9. Поддерживаются константы любых типов. Если для константы задать не постоянное значение, а выражение, то оно будет вычислено при загрузке программы.

```
const string s="test".ToUpper()
```

10. Все операторы, кроме = (присваивание), являются функциями, поэтому их можно перегружать. Операторы выполняются строго по приоритету.

```
WriteLine(10-1+2); //7, а не 11, т.к. сложение выполняется всегда до вычитания
```

11. Интерпретатор делает проверку согласования типов. Чтобы присвоить переменной значение другого типа можно использовать оператор := (как в Паскале), который производит конвертирование, возможность которого проверяется на этапе загрузки.

```
int x:=2.51; //3 (с округлением)
```

12. Оператор явного преобразования это \ (обратный слэш), но можно использовать и традиционный синтаксис, т.е. скобки - ().

```
int x=int\2.99; //2
x=(int)2.99; //2
```

13. Вместо операторов as и is используются функции As, Is.

```
Bar b=new Bar();
WriteLine(b.Is(Foo)); //True
WriteLine(b.As(Foo)!=null); //True
ReadKey();

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

14. Наследование работает как простое копирование всех членов.

```
Bar b=new Bar();
b.PrintX(); //10
b.PrintXY(); //10; 20

ReadKey();

class Foo{
    int X=10;
    public PrintX(){
        WriteLine(X);
```

```
    }
}
class Bar:Foo{
    int Y=20;
    public PrintXY(){
        WriteLine(X+"; "+Y);
    }
}
```

15. Получить любой тип можно с помощью typeof(тип) или typeof("имя_типа"), а также функцией GetTypeByName.

```
WriteLine(typeof(Foo.Bar)); //App.Foo.Bar - свой тип (объект не Type, а CustomType)
WriteLine(typeof("System.Uri, System.Private.Uri")); //System.Uri - объект Type с .NET-типом.
WriteLine(GetTypeByName("System.Uri, System.Private.Uri")); //System.Uri
ReadKey();

class Foo{
    public class Bar{}
}
```

16. Экземпляры своих классов создаются через new, как в C#, а .NET типов - функцией Create.

```
object Uri=typeof("System.Uri, System.Private.Uri");
object u = Uri.Create("https://www.google.com/"); //Uri.Create("https://www.google.com/") то же, что и Create(Uri, "https://www.google.com/")
WriteLine(u.GetType()); //System.Uri //u.GetType() - это GetType(u)
WriteLine(u->Host); //www.google.com //u->Host - это доступ к свойству с помощью рефлекшн
```

17. Доступ к членам .NET-объектов осуществляется стрелкой, что показано в примере выше (u->Host). Интерпретатор заменяет стрелки вызовами reflection-функций.
18. Перехват ошибок отличается от C# тем, что в catch после типа не нужно писать переменную, а данные исключения помещаются в специальные переменные exName, exMessage, exObject, exception.

```
try{
    int x=5/0;
}
catch("DivideByZeroException"){
    WriteLine("Name: "+exName); //Name: DivideByZeroException
    WriteLine("Message: "+exMessage); //Message: Attempted to divide by zero.
    WriteLine("Is DivideByZeroException: "+exception.Is("System.DivideByZeroException")); //Is DivideByZeroException: True
}
```

Если перехватывается .NET-тип исключения, то его имя нужно указывать в кавычках. Можно перехватывать по имени исключения, а также свои типы исключений.
19. Для вызова функций (в том числе операторов) есть возможность задать, что вернуть в случае возникновения исключения. Есть полная форма с доп. параметрами.

```
int x=GetSomeValue()(-1);
WriteLine(x); //-1
x=(5/0)(555);
WriteLine(x); //555
ReadKey();

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

20. Можно создавать потоки, задачи, таймеры. Они могут быть локальными (могут работать с локальными переменными).

```
string s="Hello!";
object t=Task(WriteLine(""), s); //кроме функции задаётся аргумент, который ей передать
StartTask(t);
Wait(t);
//Hello!
t=LocalTask(WriteLine(s)); //при создании локальной задачи указывается выражение, которое выполнить
StartTask(t);
Wait(t);
//Hello!
ReadKey();
```

21. Можно создавать делегаты.

```
object d=Delegate(TestEvent(""), "System.Action`1[System.String]");
WriteLine(d.Raise("Hello!")); //Hello!
ReadKey();

TestEvent(string s){WriteLine(s);}
```

22. Встроенные функции можно импортировать из своих DLL.

```
#import "MyLib.dll" as MyLib

MyLib@SomeFunc(); //используется @ для отделения префикса MyLib
```