

About S#

[See Also](#)



S# is a weakly-typed dynamic language and runtime infrastructure to make your applications extendable, customizable and highly flexible. It allows introducing expressions and large code blocks evaluation within your applications in the similar way Microsoft Office deals with VBScript, gives you possibilities providing rich formula evaluation capabilities like it can be seen in MS Excel and other office applications.

The key principles of S# are: simplicity, efficiency, intuitive. The S# runtime has been designed to be easily hosted by applications. Minimum script execution scenario requires two lines of code! The important part of S# is its well-defined extendable runtime engine together with the application programming interface that allows full bi-directional communication between script and application code. In particular it is easy to extend S# by embedding external functions and functional objects, shared static or dynamic variables, operator handling and type filters from the host application. Moreover the execution semantics of some language constructs has extensibility mechanisms available externally. This enables developers to create user-friendly executable business/domain specific languages on S# basis.

S# can work in single-expression mode in order to execute string expressions to values. This is especially helpful when application should allow users executing only light-weight portions of the functionality. S# is a pure .NET interpreted language completely written in C#.

Currently S# is compatible and runs on top of the following platforms:

- Microsoft .NET 4
- Microsoft .NET 3.5 (SP1)

- Microsoft Silverlight > 3
- Microsoft .NET Compact Framework
- Microsoft XNA Framework
- MONO

This means that S# runtime can be hosted by applications based on .NET like Console, Windows Forms, ASP.NET, Silverlight 2 and 3, Windows Presentation Foundation (WPF), XNA (both PC and XBox scenarios) and MONO (Linux).

With respect to the DLR and languages like IronPython, S# is:

1. Designed to be easily embedded into applications (several lines of code required to evaluate script);
2. Highly extensible grammar and language (new functions, constants, variables can be added easily);
3. Rich and controlled communication between script and application code in both directions;
4. Sits on the top of .NET, full support for interaction with native .NET code;
5. Works in single-expression mode when executing string expressions to values;
6. No code emitting, CodeDom or background compilation, 100% interpreted language (own Virtual Machine and Debugging is in progress);
7. Totally in-memory execution, does not require temp files, local system access, etc;
8. C#-like language is designed to be familiar for .NET and Java developers; easy to study for non-developer users;
9. Various of platforms are supported (.NET/.NETCF/Mono; WinForms, ASP.NET, WPF, Silverlight, XNA);
10. Fully remotable (expressions and scripts can be sent across the wire and executed on a remote machine);

11. Weakly-typed as IronPython/IronRuby for .NET

S# was originally created by Petro Protsyk and subsequently adapted and improved by Denis Vuyka and Francois Vanderseypen.

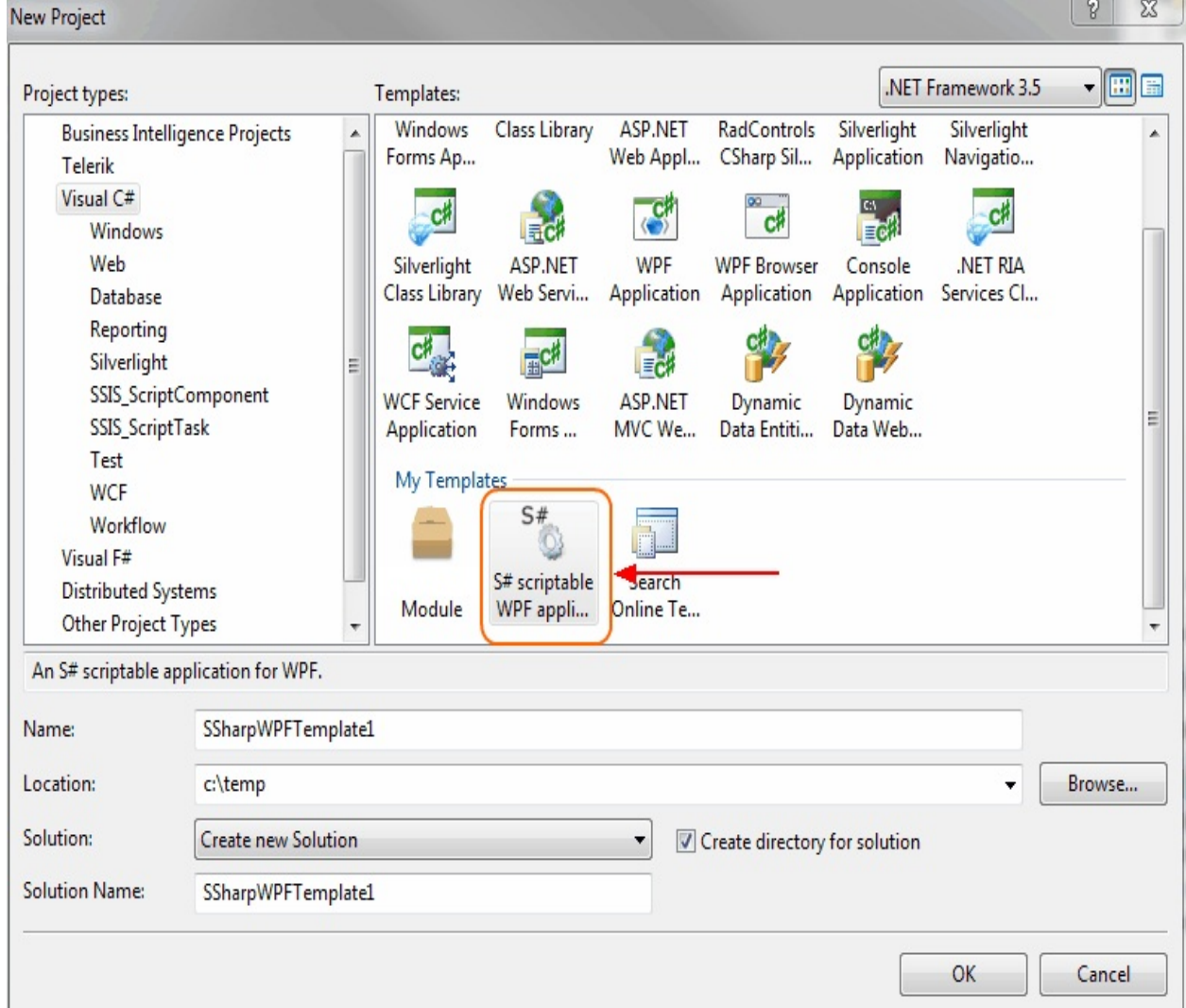
☐ [Collapse All](#) ▾ [Language Filter:](#)

S# API documentation

Downloads

[See Also](#)

- The [latest runtime](#) setup is available from our [S# product page](#). You can find an installation manual [here](#).
- A [Visual Studio 2008 template](#) to create quickly prototype of scriptable WPF applications is available. Add it to your \Documents\Visual Studio 2008\Templates\ProjectTemplates directory and you will see the template appear in your Visual C# templates



[\[-\] Collapse All](#) [\[+\] Language Filter:](#)

S# API documentation

What's New

[See Also](#)

Assembly version 3.0.0.0

- .NET 4 Support
- Error notification improved. Various types of exceptions were introduced.
- Notion of ContextEnabled events removed. All events are now ContextEnabled, which means event handler assigned within script will be executed in the context of this script.
- UnsubscribeAllEvents property removed from RuntimeHost. It is however possible manually clear all event subscriptions by using EventBroker.ClearAllSubscriptions method.

[☰ Collapse All](#) [▶ Language Filter:](#)

S# API documentation

Comments

[See Also](#)

Two forms of comments are supported: delimited comments and single-line comments.

A **delimited comment** is wrapped inside the `/* */` characters. Delimited comments can occupy a portion of a line, a single line, or multiple lines. The following example includes a delimited comment:

```
/* This is some sample multi-line comment.  
   Syntax and code will not be parsed inside this.  
   */
```

A **single-line comment** begins with characters `//` and extends to the end of the line. The following example includes a single-line comment:

```
a = 0; // initializing variable with 0  
b = 1; // initializing variable with 1
```

Comments don't nest. The character sequences `/*` and `*/` have no special meaning within a single-line comment, and the character sequences `//` and `/*` have no special meaning within a delimited comment. Comments are not processed within character and string literals.

Data types

[See Also](#)

It is possible to expose any .NET data type or instance to S#. There are however common .NET types available for script by default:

Type Name	.NET Analogue	Initializer
double	double	x = 1.23; x = 12d;
long	long	x = 3;
string	string	s='Hello World!'
bool	bool	b = true
array	object[]	A = [1, 1+2, 'Hello', s]

There is also an implicit type **object** which is basically alias for .NET **System.Object**. The list of base types may be changed through xml configuration. However, it does not mean that run-time can't access other .NET types. In contrast, base types are cached by alias and may be accessed faster than other types. Consider this if you want to improve performance of script execution.

Note: S# runtime infrastructure allows to filter types and whole assemblies available to the client scripts. It is also possible to mark members of a class definition with **[Promote(false)]** attribute to make them invisible to the default script binder.

Any type visible to the runtime can be accessed either by short name:

```
b = new StringBuilder();
```

or by its full name:

```
b = new System.Text.StringBuilder();
```

Besides the XML configuration, it is also possible to expose any .NET

Framework type to the S# programs and use it within a script via the `RuntimeHost.AddType()` method.

Note: If a type was not explicitly added, the S# runtime will search it in libraries loaded in the current application domain. Although it is possible to filter assemblies and types available to the script as well as completely override the assembly/type management through a custom implementation of the `IAssemblyManager` interface.

Type conversion

[See Also](#)

Even though S# is dynamically typed language there are cases when a value should be casted to a type explicitly. The type conversion operator is of the following form:

(Expression1) Expression2;

Where

- Expression1 - any valid expression which evaluates to .NET Type,
- Expression2 - any valid expression, value of which will be converted.

Examples:

```
(string)23;
```

```
s = string;
```

```
b = (s) 23; // b = "23";
```

Any type conversion during script execution is performed via the runtime Binder. The above shown examples are simply syntactic shortcuts for the following calls:

```
RuntimeHost.Binder.ConvertTo(23, string);
```

[\[-\] Collapse All](#) [\[+\] Language Filter:](#)

S# API documentation

Constants

[See Also](#)

Boolean = True | False

Object = null

String constants must be in ' ' or " " quotes, example:

```
s = 'Hello';
```

```
s = "Hello World!";
```

The S# parser supports following number formats:

//Hex

```
h = 0xAAFFAA;
```

//Unsigned int

```
u = 3u;
```

//long

```
l = 31231231278l;
```

//unsigned long

```
ul = 23423234548ul;
```

//Double

```
d = 3.2312d;
```

//Single, Float

```
f = 3424.123f;
```

//Decimal

```
m = 23123.25434543m;
```

//With exponent

```
n1 = 4e+3;
```

```
n2 = 6.32e-3;
```

Arrays in S# may be created in different ways. The most simple array constructor is:

[value_1, value_2, ..., value_n]

Arrays built using this constructor will have their type of object[].

Elements of array may be accessed by index:

array[index];

Example:

```
a = [1,2,3,4]; b = a[1];
```

This is equal to following code in C#:

```
object[] a = new object[] {1,2,3,4};  
object b = a[1];
```

There is also a custom function **array** which creates a typed array. There are two cases for usage array function:

- **Explicit type definition:**

```
a = array(string, 'alex', 'peter');
```

- **Implicit type inference:**

```
// a is of type string[]
```

```
a = array('alex', 'peter');
```

```
// b is of type object[]
```

```
b = array('alex', 1, 2);
```

☐ Collapse All ▶ Language Filter:

S# API documentation

Variables

[See Also](#)

All variables in S# are dynamically typed. Storage for variable values is called [scope](#).

Examples of valid variable names: `X`, `x1`, `name_of_var`.

Variables may be used in expressions, statements, function invocations, etc. The type information is computed at run-time basing on the value associated with a variable name.

See also:

- [Global and local variables](#)
- [Example of global variable in recursive function](#)
- [Scopes by example](#)

[Collapse All](#) [Language Filter:](#)

S# API documentation

Expressions

[See Also](#)

The syntax of expressions is very standard, examples are:

```
X = (y+4)*2;  
Y = a[5] + 8;  
Z = Math.Sqrt(256);  
P = new System.Drawing.Point(3,4);  
'this is string' is string
```

Expressions can be concatenated using the following operators:

`+, -, *, /, %, !, |, &, !=, >, <, is`

Code expression `<! program code !>` compiles a given code as S# function:

```
a = <! c++; b+=3; return 2; !>;  
c=2;  
b=5;  
f = a();
```

```
Test.AreEqual(2, f);  
Test.AreEqual(3, c);  
Test.AreEqual(8, b);
```

```
a = <! return a+b; !>;  
Test.AreEqual(19, a([a->9, b->10]));
```

There is also special operator **new** for creating instances of imported

types:

```
b = new TypeName(constructor arguments);
```

[\[-\] Collapse All](#) [\[+\] Language Filter:](#)

S# API documentation

Statements

[See Also](#)

A program in S# script is a sequence of statements. There are three common statement types supported: sequencing, looping and branching.

if ... then ... else

if (Expression) Statement **else** Statement

For example:

```
if (x>0) y = y + 1 ; else y = y - 1;  
if (x>0) message = 'X is positive';
```

for

for (Expression1;Expression2;Expression3) Statement

For example:

```
sum=0;  
for(i=0; i<10; i++) sum = sum + a[i];
```

while

while (Expression) Statement

For example:

```
while (i>0) i = i-1;
```

foreach ... in

foreach (Identifier **in** Expression) Statement

Note: The result of Expression calculation must implement IEnumerable. Expression evaluates only once, before loop starts. For example:

```
arr=![1,2,3,4,5]; sum = 0;  
foreach(i in arr ) sum = sum + i;
```

switch

```
switch (Expression)  
{  
    case expr1: Statement  
    ...  
    default: Statement  
}
```

For example:

```
switch (i)  
{  
    case 1 : MessageBox.Show('Hello!');  
    case 2 : MessageBox.Show('?');  
    default: MessageBox.Show('No way');  
}
```

using

```
using ( object or type )  
{  
    ...  
}
```

Example 1:

```
using (Math)  
{
```

```
    return Pow(2,10);  
}
```

Example 2:

```
a = new List<int>();  
using(a)  
{  
    Add(10);  
    Add(20);  
}  
return a[0];
```

break, continue

This has usual meaning and can be used only inside a loop.

return

Used only inside function calls.

See also:

- [Loops](#)
- [Using scope](#)

[\[-\] Collapse All](#) [\[+\] Language Filter:](#)

S# API documentation

Functions

[See Also](#)

Function can be created at run-time and added into execution scope by compiling a string containing its definition. Aside this, there are few other possibilities to introduce functions to script. Function could be native - compiled from source, or external - is .NET object implementing IInvokable interface.

Simple Definition

```
function NAME (id1, id2, ... , idn)
{
    Statement
}
```

With references to global variables

```
function NAME (id1, id2, ... , idn) global(id1,...,idk)
{
    Statement
}
```

For example:

```
y = 100;
function f() global(y)
{
    y = y - 1;
}
f();
```

After executing this script the variable y will have its value 99.

With contracts

```
function (id1, id2, ... , idn)
[
  pre(boolExpr);
  post(boolExpr);
  invariant(boolExpr);
]
{
  Statement
}
```

By function expression (this is usually called anonymous function)

```
NAME = function (id1, id2, ... , idn) { Statement };
```

For example:

```
helloFunction = function (name) { return string.Format("Hello {0}!", name); }
helloFunction('John');
```

All examples of syntactic function definitions are equal to the function expression. In fact when function a() {...} is written it actually executes as following assignment statement:

```
a = function () {};
```

By default after compilation all function expressions will be executed, so corresponding variables will appear in script scope. This is done by FunctionDeclarationVisitor post processing available in the library.

Note: The functional expression:

```
function (params) { body };
```

evaluates to an `IInvokable` object. During the execution of the function a local scope (Contract Scope) is created and all variables created within this scope will be removed at function return;

Example showing various ways of invoking function:

```
function fac(n){  
    if (n==1) return 1;  
    else return n*fac(n-1);  
}  
rez = fac(5);
```

```
//pointer to a function  
Func_pointer = fac;  
Func_pointer(4); //Call function using pointer
```

```
//Anonymous function  
aFunction = function(n){  
    if (n==1) return 1;  
    else return n*aFunction(n-1);  
};  
//Call function created as assignment  
aFunction(5);
```

```
//Using .NET Method  
sin = Math.Sin;  
v = sin(0.75);
```

See also:

- [Recursion](#)
- [Creating custom function](#)

[\[-\] Collapse All](#) [\[+\] Language Filter:](#)

S# API documentation

The IInvokable interface

[See Also](#)

Objects implementing the IInvokable interface will be treated as functions. Suppose there is a variable `f` in the function scope that implements IInvokable, then following S# code:

```
f(1,2,3);
```

will be interpreted as following in C# code:

```
IInvokable fi = f as IInvokable;  
if (fi == null || !fi.CanInvoke()) throw exception;  
return fi.Invoke(Context, new object[] {1,2,3});
```

See also:

[Tutorial 2. Creating custom function](#)

☐ [Collapse All](#) ▶ [Language Filter:](#)

S# API documentation

Reserved Functions

[See Also](#)

The following names are reserved words in the context of function definitions:

- **eval** evaluates value of an expression;

Example of eval function usage:

```
a = eval('2+3*4');
```

- **clear** clears all variables in context; (**Obsolete**)
- **array** - creates typed array of objects; (see, [Arrays](#) topic)

[\[-\] Collapse All](#) [\[+\] Language Filter:](#)

S# API documentation

Property bags

[See Also](#)

S# has internal notion of property bags. A property bag is a special kind of object which is composed of (name,value) pairs. Any property may be accessed via standard syntax **object.PropertyName**.

Syntactically property bag may be created as following:

```
vector3d = [  
    x -> 2,  
    y -> 3,  
    z -> -2  
];
```

After property bag was created it is possible to introduce new property to it:

```
vector3d.name = "my vector";
```

Underlying .NET property bag class should implement `IScriptable` interface. By default such class is called **Expando**, and specified by the following setting in runtime configuration:

```
<Item id="ScriptableObjectType"  
value="Scripting.SSharp.Runtime.Promotion.Expando" />
```

It is of course possible to replace default implementation by custom.

[\[-\] Collapse All](#) [\[+\] Language Filter:](#)

S# API documentation

IScriptable interface

[See Also](#)

Let's look at IScriptable definition:

```
/// <summary>
/// Expose dynamic members of an Instance to the script.
/// This require using of DefaultObjectBinder class as default object binder.
/// </summary>
public interface IScriptable
{
    /// <summary>
    /// Should return object wrapped by IScriptable or this
    /// </summary>
    [Promote(false)]
    object Instance { get; }

    /// <summary>
    /// Gets a binding to an instance's member (field, property)
    /// </summary>
    [Promote(false)]
    IMemberBinding GetMember(string name, params object[] arguments);

    /// <summary>
    /// Gets a binding to an instance's method
    /// </summary>
    [Promote(false)]
    IBinding GetMethod(string name, params object[] arguments);
}
```

Each class implementing this interface has special meaning in S#. Let's assume that variable **scriptable** associated with an instance implementing IScriptable interface. Then following examples are valid in S#:

a = scriptable.PropertyName;

//The same as following code:

// a = scriptable.GetMember("PropertyName").GetValue();

scriptable.PropertyName = 2;

//The same as following code:

//scriptable.GetMember("PropertyName").SetValue(2);

scriptable.MethodName(1, "test", 3);

//The same as following code:

// scriptable.GetMethod("MethodName").Invoke(currentContext, new object[]

[\[-\] Collapse All](#) [\[+\] Language Filter:](#)

S# API documentation

Implementing objects

[See Also](#)

S# is not object-oriented language, however it is possible to emulate classes using property bags.

Function definition is a first class object in S#, which means it may be associated with a variable name, thus allowing to introduce functional logic as a part of property bag. Consider following example:

```
v = [  
  x -> 4,  
  y -> 3,  
  
  length -> function() {  
    Math.Sqrt(me.x^2+me.y^2);  
  }  
];  
  
l = v.length();
```

In the given example variable `length` associated with anonymous function. This function calculates length of vector `v`. To get access to properties defined within current property bag, a function definition may use reserved name "**me**" (it is similar to C# **this**).

Here is yet more comprehensive example which shows implementation of 3x3 matrix class and matrix product function:

```
m = [  
  //Rows  
  r->3,  
  //Columns  
  c->3,
```

```

//Values
v->[1,2,3,
    4,5,6,
    7,8,9],

//Get value
cell->function(row,col){
    return me.v[me.c*row + col];
},

//Set value
set->function(row,col,val){
    me.v[me.c*row + col] = val;
}

];

function MatProduct(a,b){
    rez = [
        r->a.r,
        c->b.c,
        v->new double[a.r*b.c],
        cell->a.cell,
        set->a.set
    ];

    for(i=0; i<a.r; i++){
        for (j=0; j<b.c; j++){
            s = 0;
            for (v=0; v<a.c; v++){
                s+= a.cell(i,v)*b.cell(v,j);
            }

            rez.set(i,j,s);
        }
    }
}

```

```
return rez;  
}
```

```
r = MatProduct(m,m);
```

[-] [Collapse All](#) [▶ Language Filter:](#)

S# API documentation

Event handling

[See Also](#)

S# has limited capabilities of handling .NET events. However it is possible to subscribe a function defined in Script on .NET event.

Note: It is possible to subscribe on events having the following signature:

`public void Event(object sender, T EventArgs)` or which is the same `EventHandler<T>`.

Here's an example for running S# in a simple Windows Forms application:

```
function OnClick(s,e)
{
    MessageBox.Show('Hello');
}
button.Click += OnClick;
```

[You can also the sample from here.](#)

It is also possible to wire Routed Events with S# functions when running scripts within WPF or Silverlight applications.

☐ Collapse All ▶ Language Filter:

S# API documentation

Context bound events

[See Also](#)

All event subscriptions in S# are controlled by a specific component called event manager. It stores in-memory cache of all S# function subscriptions to .NET events. Each time script execute += operator new record will be added to event manager cache. Each event will be handled in the same context as a script containing this event function.

Here is an example which demonstrate this case (C#):

```
Script s = Script.Compile(  
    @"  
    invoked = false;  
    function handler(s,e) global(invoked) {  
        invoked = true;  
    }  
    test = new EventSource();  
    test.NameChanged += handler;  
    return test;  
    "  
);  
EventSource resultVal =(EventSource)s.Execute();  
Assert.IsFalse((bool)s.Context.GetItem("invoked", false));  
//At this point event will be executed in the script's context, i.e. s.Context  
resultVal.Name = "TestName";  
//Now, check that the value in context was changed  
Assert.IsTrue((bool)s.Context.GetItem("invoked", false));
```

Event Source class

```
public class EventSource  
{  
    string name;
```

```
public string Name
{
    get
    {
        return name;
    }
    set
    {
        name = value;
        if (NameChanged != null)
            NameChanged.Invoke(this, EventArgs.Empty);
    }
}
public event EventHandler<EventArgs> NameChanged;
}
```


[\[-\] Collapse All](#) [\[+\] Language Filter:](#)

S# API documentation

Generic parameters

[See Also](#)

You can use generics in S# through the syntax

MethodName<|TypeName|>(…)

For example if you have an `Attach<T>` method you can call it with the string type as follows:

Attach<|string|>(…)

and create instances of generic types:

list = new List<|string|>(…)

Example of using generic method:

C#:

```
public class TestGeneric
{
    public string GenericGet<T>(T input)
    {
        return input.ToString();
    }
}
```

S#:

```
a = new TestGeneric();
return a.GenericGet<|string|>('Hello World');
```

Script context

[See Also](#)

Script Context is a part of script that stores run-time information, such as: scopes, variables inside scopes, flags, functions, etc. With a help of Script Context host application may expose .NET objects for script. Script Context is a point of interoperability between runtime and host application.

Script runtime ensure that context is passed across all executable parts of script. For example, each user defined function implementing [IInvokable](#) interface will be supplied with an instance of context (as a first parameter) during its execution.

Context Switching¶

Run-time ensures that there is one-to-one relation between pre-compiled script instance and script context.

```
Script s1 = Script.Compile("return A;");
Script s2 = Script.Compile("return B;");

s1.Context = s2.Context;
//After this statement s2.Context will be equal to null
```

Following scenario is possible:

```
Script s = Script.Compile("return A;");

ScriptContext sc = new ScriptContext();
sc.SetItem("A", 1);
s.Context = sc;

ScriptContext sc1 = new ScriptContext();
sc1.SetItem("A", 10);
```

```
s.Context = sc1;
```

Note: during context switching all references created with Context.Ref method will be cleared. Context.Ref mainly used by runtime to cache references and improve performance. Avoid using this function.

[-] Collapse All [v] Language Filter:

S# API documentation

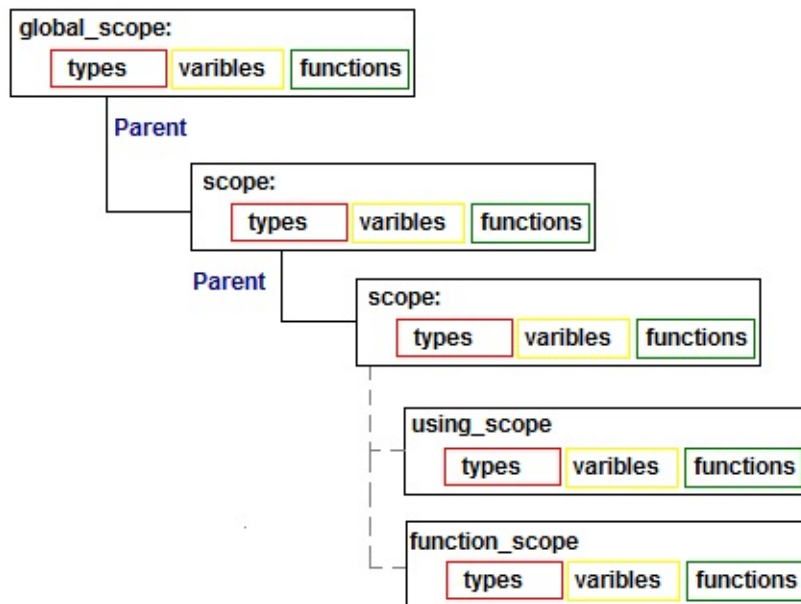
Script Scope

[See Also](#)

The purpose of the script Scope is to:

- Resolve variables: associate value with name, return value by given name;
- Create reference to variable for fast access to value;
- Associate name with `IInvokable` object ([function](#))

Scopes in S# forms a hierarchy:



There are different types of scopes: global, local, function scope, using statement scope, and events scope. All of them are used to resolve names: either variable's, type's, function's name or method's and properties' names in of certain object in case of using statement. It is possible to inherit and implement a `Scope` class to introduce custom behavior of name resolution. Before script execution the user can add objects,types and functions into Script's scope, so they will be available for script. For example:

```
List<int> vals = new List<int>();
vals.AddRange(new int[] { 1, 2, 3, 4 });
Script script = Script.Compile(@"
    rez = 0;
    foreach (number in numbers)
        rez += number;
");
//Adding variable to script's scope
script.Context.SetItem("numbers", vals);
object rez = script.Execute();
Console.WriteLine(rez);
```

[\[-\] Collapse All](#) [\[+\] Language Filter:](#)

S# API documentation

Global and local variables

[See Also](#)

Global variables are variables that are accessible in every scope. Such variables are used extensively to pass information between sections of code that don't share a caller/callee relation like functions.

Example:

```
// Creates a global variable "g_variable1" with value of 100  
g_variable1 = 100;  
// Creates a global variable "g_variable2" with value of 200  
g_variable2 = g_variable1 + 100;  
function MyFunction1() { ... }  
function MyFunction2() { ... }
```

Local variables are variables that are given local scope. Such variables are accessible only from the function or block in which it is declared. Local variables are contrasted with global variables.

Example:

```
// Creates a global variable  
my_variable = 100;  
function MyFunction()  
{  
  // Creates a local variable  
  my_variable = 200;  
}
```

To avoid naming conflicts when referencing local and global variables S# provides a special global: keyword that provides access to global scope

Example:

```
// Creates a global variable  
g_variable = 100;  
function MyFunction()  
{  
  // Creates a local variable  
  g_variable = 200;  
  Console.WriteLine("Local variable value: " + g_variable);  
  Console.WriteLine("Global variable value: " + global:g_variable);  
}  
// Invokes "MyFunction" function  
MyFunction();
```

When running the example above you will get the following output as a result:

Local variable value: 200

Global variable value: 100

Var Keyword

There is ability to specify that variable should be created in exactly that scope in which expression will be evaluated. This is the purpose of **var** keyword:

```
var a = 2;
```

[\[-\] Collapse All](#) [\[+\] Language Filter:](#)

S# API documentation

Merging Global and Local Scopes

[See Also](#)

S# also provides a possibility importing a set of global variables into the local scope of a function. This is achieved by means of `global()` statement used within function definition.

Example:

```
myVariable1 = 10;
myVariable2 = "hello";
function MyFunction() global (myVariable1, myVariable2)
{
    myVariable1++;
    myVariable2+= " world!";
    Console.WriteLine(myVariable1);
    Console.WriteLine(myVariable2);
}
```

The example above will provide the following output when executed:

```
11
hello world!
```

Please note that after variables from global scope are merged into the local one for "MyFunction" function it is not possible to declare local variables with the same names. It is recommended to use "global()" statement for [functions](#) which are intended to manipulate static (global) variables primarily.

[-] Collapse All [v] Language Filter:

S# API documentation

Scopes by example

[See Also](#)

Creating global variable

```
a = 1;
```

Creating global variable in local scope

```
//Global scope
{ //Local scope 1
  { //Local scope 2
    //Create global variable from local scope
    a = 4;
  }
}
// In this scope a = 4
return a;
```

Temporary local variables

```
//Global scope
{ //Local scope 1
  var a; //Create empty variable in local scope 1
  { //Local scope 2
    //This will set variable to top-most scope which contains
    //definition for variable, which is Local scope 1
    a = 4;
  }
}
//Global scope still empty
```

Variable resolving rules

```
//Global scope
{ //Local scope 1
  var a; //Create empty variable in local scope 1
```

```

    //Local scope 2
    //This will set variable to top-most scope which contains
    //definition for variable, which is Local scope 1
    a = 5;
    { //Local scope 3
      var a; //Create empty variable in local scope 3
      //This will set variable to top-most scope which contains
      //definition for variable, which is Local scope 1
      global:a = 4;
      a = 3; // Set local variable
    }
  }
  //Create variable in global scope equal to current value of a in this
  scope
  b = a;
}
//b = 4;
return b;

```

Temporary variables in for loop

```

var sum = 0;
for (var x=0; x<10; x++){
  var temp = x;
  sum += x;
}
//Here temp and x variables are absent
return sum;

```

☐ Collapse All ▶ Language Filter:

S# API documentation

.NET Integration

[See Also](#)

This topic describes various specific ways for interacting with .NET code.

1. Pointer to .NET Member (S#)

```
sin = Math.Sin;  
return sin(0.75);
```

```
a = DateTime.Now;  
b = a.ToString;  
return b();
```

2. Threading (S#)

```
function ThreadTest()  
{  
    return true;  
}
```

```
th = new Thread(new ParameterizedThreadStart(ThreadTest,  
ThreadTest.ThreadInvoke));  
th.Start(Context);
```

3. Interfaces

C#:

```
public interface ITest  
{
```

```
int Get();
}

public interface ITest1
{
    int Get();
}

public class TestInterface : ITest, ITest1
{
    #region ITest Members

    int ITest.Get()
    {
        return 2;
    }

    #endregion

    #region ITest1 Members

    public int Get()
    {
        return 15;
    }

    #endregion
}
```

S#:

```
a = new TestInterface();
return a.Get(); //Returns 15
```

```
a = new TestInterface();
```

```
i = new ExplicitInterface(a, ITest);  
return i.Get(); //Returns 2
```

4. Shadowed methods

S# will always access shadowed method/property.

C#:

```
public class TestShadowBase  
{  
    public string Name { get { return "Base"; } }  
  
    public virtual string Get(){ return "BaseGet"; }  
}  
  
public class TestShadow : TestShadowBase  
{  
    public new string Name { get { return "Shadow"; } }  
  
    public new string Get() { return "ShadowGet"; }  
}
```

S#:

```
a = new TestShadowBase();  
b = new TestShadow();  
  
return a.Name + b.Name + a.Get() + b.Get();  
//Returns "BaseShadowBaseGetShadowGet"
```

[\[-\] Collapse All](#) [\[+\] Language Filter:](#)

S# API documentation

Examples of scripts

[See Also](#)

All examples are given in S# language.

Test class has following implementation (C#):

```
public class Test
{
    public static void IsTrue(bool value)
    {
        if (!value) throw new TestException("Condition failed");
    }

    public static void AreEqual(object v1, object v2)
    {
        if (!object.Equals(v1, v2))
            throw new TestException("Equality condition failed, expected:
"
                + AsString(v1) + ", actual: " + AsString(v2));
    }

    private static string AsString(object v1)
    {
        if (v1 == null) return "null";

        return v1.ToString();
    }
}

public class TestException : Exception
{
    public TestException(string message)
```

```
    : base(message)
  {
  }
}
```

[\[-\] Collapse All](#) [\[+\] Language Filter:](#)

S# API documentation

Bubble Sort

[See Also](#)

```
a=[17, 0, 5, 3,1, 2, 55];

for (i=0; i < a.Length; i=i+1)
  for (j=i+1; j < a.Length; j=j+1)
    if (a[i] > a[j] )
      {
        temp = a[i];
        a[i] = a[j];
        a[j] = temp;
      }

s = "";
for (i=0; i < a.Length; i++)
  s = s + ',' + a[i];

Test.AreEqual('0,1,2,3,5,17,55',s);
```


[\[-\] Collapse All](#) [▶ Language Filter:](#)

S# API documentation

Quick Sort

[See Also](#)

```
function swap(array, a, b)
{
    tmp=array[a];
    array[a]=array[b];
    array[b]=tmp;
}
```

```
function partition(array, begin, end, pivot)
{
    piv=array[pivot];
    swap(array, pivot, end-1);
    store=begin;
    for(ix=begin; ix < end-1; ix++) {
        if(array[ix]<=piv) {
            swap(array, store, ix);
            store++;
        }
    }
    swap(array, end-1, store);
    return store;
}
```

```
function qsort(array, begin, end)
{
    if(end-1>begin) {
        pivot=begin+(end-begin) / 2;
        pivot=partition(array, begin, end, pivot);
    }
}
```

```
    qsort(array, begin, pivot);
    qsort(array, pivot+1, end);
  }
}
```

```
a = [1,2,10,0,12,34,5,3,3,4,1,23,4];
s1 = "";
for (i=0; i < a.Length; i++)
  s1 = s1+' '+a[i];
```

```
qsort(a, 0, a.Length);
s="";
for (i=0; i < a.Length; i++)
  s = s+' '+a[i];
```

```
Test.AreEqual('0 1 1 2 3 3 4 4 5 10 12 23 34', s);
```

☐ Collapse All ▶ Language Filter:

S# API documentation

Using Scope

[See Also](#)

```
using (Math)
{
    //Build-in Objects
    //using the Pow function from Math class
    a = Pow(2, 3);
}

Test.AreEqual(8d, a);
```

[\[-\] Collapse All](#) [\[+\] Language Filter:](#)

S# API documentation

Vector objects

[See Also](#)

```
function DotProduct(a,b){  
return a.x*b.x + a.y*b.y;  
}
```

```
function Scale(a, s){  
return [x->a.x*s, y->a.y*s];  
}
```

```
Test.AreEqual(19, DotProduct([x->1, y->3], [x->4, y->5]));  
Test.AreEqual(5.1, Scale([x->0.51, y->3], 10).x);
```

[\[-\] Collapse All](#) [\[+\] Language Filter:](#)

S# API documentation

Loops

[See Also](#)

```
a = [1,2,3,4,5];
```

```
//-----//
```

```
s = 0;
```

```
foreach (i in a)
```

```
s+=i;
```

```
Test.AreEqual(15, s);
```

```
//-----//
```

```
s = 0;
```

```
i = 0;
```

```
while (i < a.Length){
```

```
s += a[i];
```

```
i++;
```

```
}
```

```
Test.AreEqual(15, s);
```

```
//-----//
```

```
s = 0;
```

```
for(i=0;i<a.Length;i++){
```

```
s += a[i];
```

```
}
```

```
Test.AreEqual(15, s);
```

```
//-----//
```

[\[-\] Collapse All](#) [\[+\] Language Filter:](#)

S# API documentation

Generic List

[See Also](#)

```
a = new List<string>();
```

```
a.Add("Hello");
```

```
a.Add("World");
```

```
s = String.Join(" ", a.ToArray());
```

```
Test.AreEqual("Hello World", s);
```

[\[-\] Collapse All](#) [\[+\] Language Filter:](#)

S# API documentation

Code Objects

[See Also](#)

```
a = <!c++; b+=3; return 2;!>;
```

```
c=2;
```

```
b=5;
```

```
f = a();
```

```
Test.AreEqual(2, f);
```

```
Test.AreEqual(3, c);
```

```
Test.AreEqual(8, b);
```

```
Test.AreEqual(2, eval('1+1'));
```

[\[-\] Collapse All](#) [\[+\] Language Filter:](#)

S# API documentation

Operators

[See Also](#)

```
a = true;  
b = false;
```

```
c = a | b;
```

```
Test.IsTrue(c);
```

```
c = a & b;
```

```
Test.IsTrue(!c);
```

```
a = 2d;  
c = a is double;
```

```
Test.IsTrue(c);
```


[\[-\] Collapse All](#) [\[+\] Language Filter:](#)

S# API documentation

Arithmetic Expressions

[See Also](#)

```
a = 1.0;
b = 2.0;
c = 3.0;
d = 2.0;
e = 18.0;
f = 6.0;
```

```
p = 2.0; u = 3.0; v = 1.0; r = 2.0; s = 5.0; t = 12.0;
```

```
// r1 = 9
r1 = a + b + c*d;
Test.AreEqual(9d, r1);
```

```
// r2 = -2.5
r2 = a*(b - c/d) - e/f;
Test.AreEqual(-2.5, r2);
```

```
//r3 = -4.5
r3 = a*b*((c - d)*a - p*(u - v)*(r + s))/t;
Test.AreEqual(-4.5, r3);
```

```
//r4 = 65536
r4 = 2 * d^(c*5);
Test.AreEqual(65536d, r4);
```

```
//r5 = 2
r5 = 5 % 3;
Test.AreEqual(2, r5);
```

[\[-\] Collapse All](#) [\[+\] Language Filter:](#)

S# API documentation

Recursion

[See Also](#)

```
s = 0;
```

```
function Draw(level)
```

```
{
```

```
  global:s++;
```

```
  if (level>1)
```

```
  {
```

```
    level = level - 1;
```

```
    Draw(level);
```

```
    Draw(level);
```

```
  }
```

```
}
```

```
Draw(10);
```

```
Test.AreEqual((int)(2^10-1), s);
```

[\[-\] Collapse All](#) [\[+\] Language Filter:](#)

S# API documentation

Fibonacci Numbers

[See Also](#)

```
function fib(n){  
  if (n==1) return 1;  
  else if (n==2) return 2;  
  else return fib(n-1)+fib(n-2);  
}
```

[\[-\] Collapse All](#) [\[+\] Language Filter:](#)

S# API documentation

Factorial

[See Also](#)

```
function fac(n){  
  if (n==1) return 1;  
  else return n*fac(n-1);  
}
```

[\[-\] Collapse All](#) [\[+\] Language Filter:](#)

S# API documentation

GCD

[See Also](#)

```
function GCD(a,b){
  if (a>b) return GCD(a-b,b);
  else
    if (b>a) return GCD(a,b-a);
  else
    return a;
}
```

//Non recursive function

```
function GCD_fast(a,b){
  while ( a!=b )
  {
    if (a>b) a = a-b;
    else
      if (b>a) b = b-a;
  }
  return a;
}
```

☰ Collapse All ▶ Language Filter:

S# API documentation

Tutorial

[See Also](#)

- [Tutorial 1. Getting started](#)
- [Tutorial 2. Creating custom function](#)

[\[-\] Collapse All](#) [\[+\] Language Filter:](#)

S# API documentation

Tutorial 1. Getting started

[See Also](#)

For installation details and referencing S# assembly consider following topics:

- [Downloads](#)
- [Installation](#)

Minimum C# application:

```
using System;
using System.Diagnostics;
using System.IO;
using Scripting.SSharp;
using Scripting.SSharp.Runtime;

namespace Debug.Net
{
class Program
{
static void Main(string[] args)
{
    RuntimeHost.Initialize();
    Script s = Script.Compile("Console.WriteLine('Hello World');");
    s.Execute();
    s.Dispose();
    RuntimeHost.CleanUp();
}
}
}
```

[\[-\] Collapse All](#) [\[+\] Language Filter:](#)

S# API documentation

Tutorial 2. Creating custom function

[See Also](#)

It is possible to extend S# runtime with a custom function. For this purpose two requirements should be met:

- A class implementing `IInvokable` interface should be created;
- An instance of this class should be associated with a variable name, via script context.

Here is an example of custom function:

```
RuntimeHost.Initialize();
Script script = Script.Compile(@"
    return Test();
");
script.Context.SetItem("Test", new TestFunction());

object result = script.Execute();

Assert.AreEqual(10, result);
```

where:

```
public class TestFunction : IInvokable
{
    #region IInvokable Members

    public bool CanInvoke()
    {
        return true;
    }
}
```



```
public object Invoke(IScriptContext context, object[] args)
{
    return 10;
}

#endregion
}
```

[\[-\] Collapse All](#) [\[+\] Language Filter:](#)

S# API documentation

Tutorial 3. Passing objects between script and host

[See Also](#)

1. Create a new Visual Studio ... project