FastScript .NET Programmer Manual

Version 2025.1

General

FastScript .NET is a library for executing C# scripts. FastScript .NET is built using the "lexer-parser-interpreter" scheme, it does not use compilation into machine code. It can be used in environments where code generation is prohibited (Native AOT, iOS, WASM).

Supported language features

FastScript.Net supports the following language features. Think of it as it is mostly C# 1.0 compliant, with some great additions from later C# versions. Note if a feature is not listed here, then most likely it is not supported (more info in the sections below):

C# 1.0:

- Classes
- Structs
- Enums
- Interfaces
- Events
- Operator overloading
- User-defined conversion operators
- Properties
- Indexers
- Output parameters (out and ref)
- params arrays
- Delegates
- Operators and expressions
- Verbatim identifier

C# 2.0:

- Generics
- Partial types
- Nullable value types
- Getter/setter separate accessibility
- Static classes

C# 3.0:

- Auto-implemented properties
- Extension methods
- Implicitly typed local variables

C# 4.0:

- Optional arguments

C# 6.0:

- Auto-property initializers
- Expression bodied members
- Null propagator

C# 7.0:

- Out variables
- Local functions

C# 8.0:

- Readonly members
- Static local functions
- Null-coalescing assignment

C# 9.0:

- Top-level statements

Unsupported features

The following C# 1.0 features are not supported:

- Preprocessor directives (#if, #region and so on)
- Attributes
- Unmanaged code: pointers, unsafe keyword, P/Invoke
- checked, unchecked statements
- goto statement

Partially supported features

Here and below: "host" means the .NET application, "script" means something defined in a script.

Class inheritance

Base class can either be a script class or `System.Object`. You cannot inherit from a host class. Example:

```
class MyScriptClass: OtherScriptClass // ok
class MyScriptClass: Object // ok
class MyScriptClass // ok, same as Object
class MyScriptClass: ArrayList // error
```

Structs

Internally, struct is a class. FastScript uses special handling of struct instances (a copy of struct instance is created if you pass a struct to a method parameter, or assign it to a variable). Declaring a variable of struct type does not automatically create a struct instance:

```
MyStruct s; // s is null
s = new MyStruct(); // and must be initialized before use
```

Script vs host interop

Class defined in a script is visible to host as a `FastScript.Runtime.DataContext` instance.

You may override the following `System.Object` methods of a script class:

- `ToString`
- `Equals`
- `GetHashCode`

These overridden methods will also take an effect if used by host.

A script class may implement some of host interfaces, but it has effect in a script only. Passing such an instance to a host will not work, the host will not be able to use interface members implemented in a script.

Nullable types

Nullable types can use host types only.

Generic types and methods

You can use host types/methods only. You cannot define a generic type or method in a script.

Type forwarding

If a host type is marked as forwarded, it must be used directly by the host app in order to be available in a script. Example:

```
var list = new System.ComponentModel.BindingList<int>(); // error, BindingList does not exist
```

But if you add this line of code to your host app, the script will be compiled fine:

```
new System.ComponentModel.BindingList<int>();
```

Delegates

You can create delegates of any methods (script or host). Passing a delegate to host is not supported though. You also cannot create `Action<>` and `Func<>` delegates (these host classes require a native method with certain signature, which can't be done in a script).

Implicit and explicit conversion

User defined implicit and explicit conversion is limited to actually defined cases. If there is a conversion of type T to 'int', you may use it; however you can't convert T to 'float' if there is no T->float conversion defined. Consider the following example:

```
var m = new My();

int intValue = m; // ok
float floatValue = m; // error

int explicitIntValue = (int)m; // ok: explicit is not defined, but we have implicit one
float explicitFloatValue = (float)m; // error

floatValue = (int)m; // use this way

public class My
{
    private object _value;
    public static implicit operator int(My m) => (int)m._value;
}
```

Grammar (C#)

General

```
program
    : using_directive* top_level_statements? namespace_member_declaration* EOF

using_directive
    : USING namespace_name ';'

top_level_statements
    : statement_list

namespace_member_declaration
    : namespace_declaration
    | type_declaration

namespace_declaration
    : NAMESPACE namespace_name namespace_body ';'?

namespace_name
    : identifier ('.' identifier)*

namespace_body
    : '{' using_directive* namespace_member_declaration* '}'

identifier
    : IDENTIFIER
```

Simple types

```
namespace_or_type_name
    : name_part ('.' name_part)*

name_part
    : identifier type_argument_list?

type_
    : base_type '?'? rank_specifier*

type_no_rank
    : base_type '?'?

base_type
    : predefined_type
    | namespace_or_type_name

predefined_type
    : BOOL
    | BYTE
    | CHAR
    | DECIMAL
    | DOUBLE
    | FLOAT
    | INT
    | LONG
    | OBJECT
    | SBYTE
    | SHORT
    | STRING
    | UINT
    | ULONG
    | USHORT

type_argument_list
    : '<' type_ (',' type_)* '>'
```

Expressions

```
expression_list
    : expression (',' expression)*

expression
    : binary_expression (assignment | null_coalescing_expression | conditional_expression)?

assignment
    : assignment_operator expression

assignment_operator
    : '='
    | '+='
    | '-='
    | '*='
    | '/='
    | '%='
    | '&='
    | '|='
    | '^='
    | '??='

null_coalescing_expression
    : '??' expression

conditional_expression
    : '?' expression ':' expression
```

```
binary_expression
    : unary_expression (binary_operator unary_expression | is_operation | as_operation)*

binary_operator
    : '+'
    | '-'
    | '*'
    | '/'
    | '%'
    | '||'
    | '&&'
    | '=='
    | '!='
    | '<='
    | '>='
    | '|'
    | '&'
    | '^'
    | '<<'
    | '>>'
    | '<'
    | '>'

unary_expression
    : cast_expression
    | primary_expression
    | unary_operator unary_expression

unary_operator
    : '++'
    | '--'
    | '+'
    | '-'
    | '~'
    | '!'

cast_expression
    : '(' type_ ')' unary_expression

primary_expression
    : primary_expression_start bracket_expression* (
        (member_access | method_invocation | '++' | '- -') bracket_expression*
    )*

primary_expression_start
    : NUMBER | STRING
    | TRUE | FALSE | NULL
    | THIS | BASE
    | typeof_expression
    | name_part
    | '(' expression ')'
    | predefined_type
    | object_creation

typeof_expression
    : TYPEOF '(' type_ ')'

member_access
    : '?'? '.' name_part

bracket_expression
    : '?'? '[' expression_list ']'

is_operation
    : IS type_ identifier?

as_operation
    : AS type_
```

```
object_creation
    : NEW type_no_rank (method_invocation | array_creation)

method_invocation
    : '(' argument_list? ')'

argument_list
    : argument (',' argument)*

argument
    : out_argument
    | in_argument
    | ref_argument
    | expression

out_argument
    : OUT (embedded_variable_declaration | expression)

in_argument
    : IN expression

ref_argument
    : REF expression

array_creation
    : '[' expression_list? ']' rank_specifier* array_initializer?

array_initializer
    : '{' expression_list? ','? '}'
```

Statements

```
block
    : '{' statement_list? '}'

statement_list
    : statement+

statement
    : embedded_statement
    | declaration_statement

declaration_statement
    : local_variable_declaration ';'
    | local_constant_declaration ';'
    | local_function_declaration

embedded_statement
    : block
    | simple_embedded_statement

simple_embedded_statement
    : ';'
    | toplevel_statement
    | expression ';'

toplevel_statement
    : return_statement
    | if_statement
    | switch_statement
    | while_statement
    | do_statement
    | for_statement
    | foreach_statement
    | break_statement
```

```
    | break_statement
    | continue_statement
    | throw_statement
    | try_statement
    | using_statement

if_statement
    : IF '(' expression ')' embedded_statement (ELSE embedded_statement)?

switch_statement
    : SWITCH '(' expression ')' '{' switch_section* '}'

while_statement
    : WHILE '(' expression ')' embedded_statement

do_statement
    : DO embedded_statement WHILE '(' expression ')' ';'

for_statement
    : FOR '(' for_initializer? ';' expression? ';' for_iterator? ')' embedded_statement

foreach_statement
    : FOREACH '(' embedded_variable_declaration IN expression ')' embedded_statement

break_statement
    : BREAK ';'

continue_statement
    : CONTINUE ';'

return_statement
    : RETURN expression? ';'

throw_statement
    : THROW expression? ';'

try_statement
    : TRY block (catch_clauses finally_clause? | finally_clause)

using_statement
    : USING '(' resource_acquisition ')' embedded_statement

switch_section
    : switch_label+ statement_list

switch_label
    : CASE expression ':'
    | DEFAULT ':'

for_initializer
    : local_variable_declaration
    | expression (',' expression)*

for_iterator
    : expression (',' expression)*

catch_clauses
    : specific_catch_clause+ general_catch_clause?
    | general_catch_clause

specific_catch_clause
    : CATCH '(' base_type identifier? ')' block

general_catch_clause
    : CATCH block

finally_clause
    : FINALLY block

resource_acquisition
```

```
    resource_acquisition
        : local_variable_declaration
        | expression

    local_variable_declaration
        : local_variable_type variable_declarators

    local_variable_type
        : VAR
        | type_

    embedded_variable_declaration
        : local_variable_type identifier

    local_constant_declaration
        : CONST type_ constant_declarators

    local_function_declaration
        : STATIC? type_ method_declaration
```

Types

```
    type_declaration
        : all_member_modifier* (class_definition | struct_definition | interface_definition |
enum_definition | delegate_definition)

    class_definition
        : CLASS identifier base_classes? class_body ';'?

    base_classes
        : ':' base_type (',' base_type)*

    class_body
        : '{' class_member_declaration* '}'

    struct_definition
        : STRUCT identifier base_classes? class_body ';'?

    interface_definition
        : INTERFACE identifier base_classes? class_body ';'?

    enum_definition
        : ENUM identifier (':' type_)? enum_body

    enum_body
        : '{' enum_member_declarators '}'

    delegate_definition
        : DELEGATE type_ identifier '(' formal_parameter_list? ')' ';'
```

Members

```
    all_member_modifier
        : PUBLIC
        | PROTECTED
        | INTERNAL
        | PRIVATE
        | READONLY
        | STATIC
        | PARTIAL
        | ABSTRACT
        | VIRTUAL
        | OVERRIDE
```

```
      | NEW

  class_member_declaration
      : all_member_modifier* (constant_declaration | typed_member_declaration | constructor_declaration |
operator_declaration | event_declaration)
      | type_declaration

  enum_member_declarators
      : enum_member_declarator (',' enum_member_declarator)* ','?

  enum_member_declarator
      : identifier ('=' expression)?

  constant_declaration
      : CONST type_ constant_declarators ';'

  constant_declarators
      : constant_declarator (',' constant_declarator)*

  constant_declarator
      : identifier '=' expression

  constructor_declaration
      : identifier '(' formal_parameter_list? ')' constructor_initializer? method_body

  constructor_initializer
      : ':' (THIS | BASE) method_invocation

  operator_declaration
      : implicit_operator
      | explicit_operator
      | overload_operator

  implicit_operator
      : IMPLICIT OPERATOR type_ op_method_declaration

  explicit_operator
      : EXPLICIT OPERATOR type_ op_method_declaration

  overload_operator
      : type_ OPERATOR (binary_operator | unary_operator) op_method_declaration

  op_method_declaration
      : '(' formal_parameter_list? ')' method_body

  event_declaration
      : EVENT base_type explicit_interface? identifier (';' | event_accessor)

  event_accessor
      : '{' (event_add | event_remove)+ '}'

  event_add
      : ADD method_body

  event_remove
      : REMOVE method_body

  typed_member_declaration
      : type_ (method_declaration | property_declaration | field_declaration |
extension_method_declaration)

  method_declaration
      : explicit_interface? identifier '(' formal_parameter_list? ')' method_body

  explicit_interface
      : identifier '.'

  formal_parameter_list
      : parameter_array
```

```
    | fixed_parameters (',' parameter_array)?

fixed_parameters
    : fixed_parameter (',' fixed_parameter)*

fixed_parameter
    : parameter_modifier? type_ identifier ('=' expression)?

parameter_modifier
    : REF
    | OUT
    | IN

parameter_array
    : PARAMS array_type identifier

array_type
    : base_type rank_specifier+

rank_specifier
    : '[' ','* ']'

method_body
    : block
    | ';'
    | lambda_expression

lambda_expression
    : '=>' expression ';'

property_declaration
    : index_property_declaration
    | regular_property_declaration

index_property_declaration
    : explicit_interface? THIS '[' formal_parameter_list ']' property_accessor

regular_property_declaration
    : explicit_interface? identifier property_accessor

property_accessor
    : property_get_set ('=' expression ';')?
    | lambda_expression

property_get_set
    : '{' (property_getter | property_setter)* '}'

property_getter
    : all_member_modifier* GET (';' | method_body)

property_setter
    : all_member_modifier* SET (';' | method_body)

field_declaration
    : variable_declarators ';'

variable_declarators
    : variable_declarator (',' variable_declarator)*

variable_declarator
    : identifier ('=' expression)?

extension_method_declaration
    : identifier '(' THIS formal_parameter_list ')' method_body
```

Using FastScript .NET

To use FastScript .NET in your project, add the `FastScript` Nuget package (or `FastScript.Demo` if you are using the demo version).

Demo version limitations: when compiling a script, an exception with the text "FastScript: Demo version" is injected into script methods. This happens at random (rare) moments.

Quick start

Use the following code to run simple scripts:

```
using FastScript.CSharp;

var text =
@"
using System;

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

var script = new CSharpScript();

if (script.Compile(text))
{
    script.RunMain();
}
```

The example above uses a script with top level statements. If the compilation is successful, the first statement (Console.WriteLine) is run and the message `Hello!` is printed to the console.

Compiling the script

The `Script.Compile` method performs parsing of the source text and then its compilation. During the parsing process, an abstract syntax tree (AST) is created, which represents the source text as a tree data structure.

If no errors occur during the parsing stage, FastScript .NET performs compilation. During the compilation stage, the AST is parsed and memory structures that represent the types declared in the script are created. These types are available in the `Script.Types` property.

The compilation process does not perform machine code generation. This allows FastScript .NET to be used in environments where code generation is not possible (Native AOT, iOS, WASM). During operation, FastScript .NET does not create assemblies that remain sit in memory (like CodeDOM/Roslyn do). Allocated memory structures can be deleted by the garbage collector (GC) when you finish using the script instance.

Error handling

Errors may occur during the compilation of the script: parsing errors (syntax errors) and compilation errors (semantic errors).

Syntax errors occur when a script is not grammatically correct in terms of the C# language (for example, if a method call is missing a closing parenthesis). If syntax errors are present, further processing of the script (compilation) is not performed. When an error is encountered, an attempt is made to continue parsing the script further; as a result, multiple errors may be reported. In this regard, the behavior of FastScript is similar to that used in CodeDOM/Roslyn.

Semantic errors may occur at the compilation stage, for example, when calling a non-existent method.

Error information is contained in the `Script.Diagnostic.Errors` property. Use the following code to check for errors and print information to the console:

```
var script = new CSharpScript();

if (!script.Compile(script_text))
{
    foreach (var err in script.Diagnostic.Errors)
    {
        // basic error info
        Console.WriteLine($"{err.CompilationUnit.Name} ({err.Line},{err.Column}): error {err.Code}:
{err.Message}");
        // extended info
        Console.WriteLine(err.SourceCode());
    }

    return;
}
else
{
    script.RunMain();
}
```

In this case, extended error information is printed:

```
Sample.cs (4,21): error cs69: The name 'My' does not exist in the current context
var list = new List<My>();
                    ^^
```

Running the script

The execution of a script is initiated by the `Script.RunMain` method. This method scans the types declared in the script (the list of types is available in the `Script.Types` property). It looks for a type that has a static method `Main`, and passes control to this method.

In the example below, running the script using the `RunMain` method will find the `TestClass` type, which has a static `Main` method, and execute it:

```
using FastScript.CSharp;

var text =
@"
namespace Test
{
    public class TestClass
    {
        public static void Main()
        {
            System.Console.WriteLine(""Hello!"");
        }
    }
}
";

var script = new CSharpScript();

if (script.Compile(text))
{
    script.RunMain();
}
```

If the `Main` method being run has parameters, their values must be specified in the `RunMain` method, for example:

```
    // script method:
    // public static void Main(int id, string text)

    script.RunMain(1, "abc");
```

The `RunMain` method returns the value returned by the `Main` method declared in the script. If the `Main` method is declared as `void`, the return value is undefined.

If the script has top level statements, a special class is created for them that has a static `Main` method. When the script is run using the `RunMain` method, this method receives control.

The following is an example of a script similar to the one described above that uses top level statements:

```
using FastScript.CSharp;

var text =
@"
System.Console.WriteLine(""Hello!"");
";

var script = new CSharpScript();

if (script.Compile(text))
{
    script.RunMain();
}
```

Create script class instances

The `Script.RunMain` method is useful if the script has an explicitly or implicitly declared static `Main` method. Another scenario of using the script is as follows:

- an instance of the type declared in the script is created;
- its properties and methods are manipulated.

Consider the following example, which shows how to create an instance of the `MyClass` type and call its `Print` method:

```
using FastScript.CSharp;
using FastScript.Runtime.Types;

var text =
@"
using System;

namespace MyNamespace
{
    public class MyClass
    {
        public void Print(string message) => Console.WriteLine(message);
    }
}
";

var script = new CSharpScript();

if (script.Compile(text))
{
    // get MyClass type
    var myClass = script.Types["MyClass"] as ScriptTypeInfo;

    // make an instance of it (using default constructor)
    var myInstance = myClass.CreateInstance();

    // get Print method
    var printMethod = myClass.GetMethod("Print");

    // invoke it
    printMethod.Invoke(myInstance, new object[] { "Hello FastScript!" });
}
```

Explanations of the example:

- Types declared in the script are available in the `Script.Types` property.
- The script class is represented by the `FastScript.Runtime.Types.ScriptTypeInfo` class.
- The `ScriptTypeInfo.CreateInstance` method is used to create an instance of the class. In the method parameters, you can specify arguments for the class constructor (or omit them for the parameterless constructor).
- The `GetMethod` method is used to find the `Print` method. This API is similar to `System.Reflection` and includes the methods: `GetMember`, `GetMembers`, `GetMethod`, `GetMethods`, `GetConstructor`, `GetConstructors`, `GetField`, `GetFields`, `GetProperty`, `GetProperties`, `GetEvent`, `GetEvents`, `GetNestedType`, `GetNestedTypes`.
- The `Print` method is called using the `Invoke` method, similar to the `System.Reflection` API.

Using modules

The script text can be split into several files ("modules"). In this case, you need to use the `CompilationUnit` class and an overloaded version of the `Script.Compile` method:

```
public bool Compile(params CompilationUnit[] units)
```

Here is the example of using two compilation units:

```
using FastScript;
using FastScript.CSharp;

var text1 =
@"
using Test;

// top-level statements: can be used in one unit only
TestClass.Testing();
TestClass.Tested();
";

var text2 =
@"
using System;

namespace Test
{
    public class TestClass
    {
        public static void Testing()
        {
            Console.WriteLine(""Testing..."");
        }

        public static void Tested()
        {
            Console.WriteLine(""Tested"");
        }
    }
}
";

var unit1 = new CompilationUnit("unit1.cs", text1);
var unit2 = new CompilationUnit("unit2.cs", text2);
var script = new CSharpScript();

if (script.Compile(unit1, unit2))
{
    script.RunMain();
}
```

Restrict dangerous API

Any API available in your .NET application can be used in a script. If a script can be obtained from an untrusted source, this raises the security issue. FastScript .NET allows you to restrict the use of dangerous API, such as file system or network operations. You can restrict usage of entire assemblies, namespaces, or individual types.

Use the `Script.TypeProvider` property to control how .NET types are loaded. It has the following properties:

- `IncludeAssemblies`: a `string[]` property containing a list of assembly names. Types in these assemblies will be available in the script. If the list is empty, types in all assemblies loaded in your .NET application will be available, except those specified in the `ExcludeAssemblies` property.
- `ExcludeAssemblies`: a `string[]` property containing a list of assembly names. Types in these assemblies will not be available in the script.
- `ExcludeNamespaces`: a `string[]` property containing a list of namespaces. Types in these namespaces (and their child namespaces) will not be available in the script.

Let's look at example that restricts types in assemblies and the `System.IO` namespace:

```
using FastScript.CSharp;

var text =
@"
using System;

Console.WriteLine(""OK"");

// this will fail with API restriction
var dir = System.IO.Directory.GetCurrentDirectory();
Console.WriteLine(""Current dir: "" + dir);
";

var script = new CSharpScript();

// use types defined in these two assemblies only
script.TypeProvider.IncludeAssemblies = ["System.Private.Corelib", "System.Console"];

// restrict usage of System.IO namespace and its types
script.TypeProvider.ExcludeNamespaces = ["System.IO"];

if (!script.Compile(text))
{
    foreach (var err in script.Diagnostic.Errors)
    {
        Console.WriteLine($"({err.Line},{err.Column}): error {err.Code}: {err.Message}");
        Console.WriteLine(err.SourceCode());
    }
    return;
}

script.RunMain();
```

When running this example you will get a compilation error:

```
(7,17): error cs69: The name 'IO' does not exist in the current context
var dir = System.IO.Directory.GetCurrentDirectory();
                ^^^
```

To restrict usage of particular classes, you may use the following technique.

Suppose you want to disable the `System.IO.File` class, but leave the ability to use other classes in the `System.IO` namespace. In this case, using the `ExcludeNamespaces` property will not help.

Add the following code to your script (this can be done by adding the code to the end of the script, or by placing it in a separate module - compilation unit):

```
namespace System.IO
{
    public class File { }
}
```

This script will replace the original `System.IO.File` class with a new, empty class. Attempting to use the methods or properties of the original class will result in a compile-time error.

Using Native AOT

In a script, you may use only classes available in the host app. Native AOT compiled host app may not contain a class (or a member of class) that you want to use in a script because the class/member was trimmed.

Another problem is generics (types/methods). In Native AOT, you cannot create closed generic of any type; the generic you want to construct must present in a compiled host app. For example, your host app uses `List<int>` but does not use `List<double>`. In this case, you will be able to use `List<int>` in a script, but constructing a `List<double>` will throw an error.

So it's your duty to ensure classes/methods are available in the compiled host app to use them in a script. Various techniques may be used to do this, for example, constructing instances of types, using attributes:

```
[DynamicDependency(DynamicallyAccessedMemberTypes.All, typeof(List<>))]
public void EnsureAOTVisible()
{
    var list = new List<int>();
}
```

Note that generic parameters of reference types can be used if an open generic type is available in a host app. For example, having a `List<>` type available in your app, you may construct `List<object>` or `List<List<...>>` or `List<Dictionary<...>>`.