Scripts Classes Reference

Organize reusable classes for scripts.

ScriptsClasses (Reference) provide a repository of reusable code libraries, methods, and functions accessible throughout your FrameworX solution. The platform automatically manages these classes as static objects with built-in exception handling.

ScriptClasses enable code reuse across:

- Script Tasks
- Display Code Behind
- Other ScriptClasses
- Expressions throughout the solution

The platform provides automatic:

- Static object instantiation
- Exception protection
- Cross-language interoperability (C#/Python/VB.NET)

Creating ScriptClasses
Code Structure Requirements
ScriptClass Types
Instances and Allocation
Invocation Syntax by Caller Domain
Built-in Classes
Library Import/Export
Cross-Language Interoperability
Accessing Solution Objects
External References
Compilation and Build Order
Best Practices Checklist
Troubleshooting

Creating ScriptClasses

Access Scripts Classes

Click the **plus icon** to create a new class. Three options are available:

1. **Class with Methods** - Standard reusable code library
2. **Full Namespace** - Advanced .NET namespace (Visual Studio-like)
3. **MCP Tool** - AI model integration via Model Context Protocol

Configuration Properties

Property	Description	Required
Name	Class identifier used in code references	Yes
Code	Programming language (C#, Python, VB.NET)	Yes
Domain	Execution location: Server (global) or Client (local)	Yes
ClassContent	Type: Methods or Namespace	No
Edit Security	Permission level required to modify	No
Build Order	Compilation sequence when multiple classes exist	No
Build Status	Green check (success) or Red X (errors)	No
Build Errors	Compilation error details	No
Description	Documentation for the class purpose	No

Code Structure Requirements

CRITICAL: Do not include class declarations or namespace definitions in ScriptClasses

When creating a ScriptClass with Methods or MCP Tool:

- DO NOT include: `public class MyClass { }`
- DO NOT include: `namespace MyNamespace { }`
- DO NOT include: `using` statements directly in code
- DO include: Method definitions only
- DO use: Namespace Declarations button for `using` statements

The platform automatically wraps your code in the appropriate class structure using the Name from the configuration table.

Correct ScriptClass Format

csharp

```
// Methods directly - NO class wrapper, NO using statements
public string ProcessData(double value)
{
    return $"Processed: {value:F2}";
}

public int Calculate(int a, int b)
{
    return a + b;
}
```

Adding Namespace Declarations

To add `using` statements (C#) or `Import` statements (VB.NET):

1. Open the Code Editor
2. Click the **Namespace Declarations** button in toolbar
3. Add required namespaces in the dialog
4. Never put `using` statements directly in the code

ScriptClass Types

1. Class with Methods (Standard)

Most common type for reusable code:

- Platform manages as static object
- Automatic exception handling
- No instantiation required
- Access via: `@Script.Class.ClassName.MethodName()`

Example:

csharp

```
// In ScriptClass named "Calculations"
public double CalculateEfficiency(double actual, double target)
{
    if (target == 0) return 0;
    return (actual / target) * 100;
}

// Usage anywhere in solution:
// double eff = @Script.Class.Calculations.CalculateEfficiency(95, 100);
```

2. MCP Tool

Special class type for AI integration:

- Exposes methods to external Language Models
- Requires decorator attributes
- Methods without decorators behave as regular methods
- Available in all editions including EdgeConnect

Example:

csharp

```
[MCPMethod(Description = "Get tank level")]
public double GetTankLevel(
    [MCPParameter(Description = "Tank ID")] string tankId)
{
    return @Tag[$"Tank_{tankId}_Level"];
}
```

3. Full Namespace (Advanced)

Advanced Feature Warning: Full Namespace is for experienced developers only. Most applications should use standard Class with Methods. This option requires manual object instantiation and exception handling like Visual Studio development.

Complete .NET namespace implementation:

- Requires full namespace and class declarations
- Manual object instantiation required
- No automatic exception protection
- Full control over implementation

Example:

csharp

```csharp
namespace MyCompany.Utilities
{
    public class AdvancedProcessor
    {
        private int counter = 0;

        public void Process()
        {
            // Implementation
        }
    }
}

// Usage requires instantiation:
// var processor = new MyCompany.Utilities.AdvancedProcessor();
// processor.Process();
```

Instances and Allocation

Where a method can be called from starts with how the system allocates the code you write.

Tasks vs Classes

A **Task** is the execution of a method: the system owns its lifecycle and you only write the method body and configure **when** it runs (a trigger condition or a periodic schedule) — see Scripts Tasks Reference. A **ScriptClass** is reusable code that Tasks, Displays, expressions, and other classes call into — and how it is instantiated depends on its **ClassContent**.

Class with Methods — system-allocated

The system owns the class wrapper; you only write the methods inside it. Because it knows the class (its name is the row's *Name*), it instantiates it once with `new`, exposes its methods in IntelliSense, and keeps that single instance allocated for the whole solution. Every Task, Display, and other class calls it through `@Script.Class.<Name>.<Method>(...)` — you never write `new`.

Full Namespace — you allocate

You declare the `namespace` and the class or classes inside it. Since there can be **multiple** .NET classes, the system cannot know at design time how many exist or which one to allocate, so it cannot list them in IntelliSense. By default it pre-allocates (`new`) the **first class in declaration order** and exposes that one instance through `@Script.Class.<Name>`, available system-wide including every Task. For any other class — or for full control — you instantiate it yourself with `new Namespace.Class()`.

Calling a Namespace class through @Script.Class

The `@Script.Class.<Name>` shortcut targets that single pre-allocated instance — the first class in declaration order — so you call its methods without writing `new` yourself. Two consequences follow:

- Only that first class is reachable through `@Script.Class.<Name>`; any other class in the namespace requires `new Namespace.Class()`.
- Abstract class definitions are ignored, and only **non-static** methods are reachable (a pre-allocated instance exposes instance members only).

Instantiating with `new Namespace.Class()` gives full access to every class in the namespace, and is the recommended approach for anything beyond the simplest calls.

Concurrent Calls to the Shared Instance

Each Script Class exists as one shared instance within its Domain, not one instance per caller. Periodic Script Tasks, expressions, events and other classes all call members on the same object, and the platform does not serialize the method bodies. Two callers run the same method at the same time when their calls overlap, and the method applies its work twice. The symptom is a wrong calculated value, for example a KPI averaged over a doubled set of rows, rather than a timing error.

Guard shared state inside your own class body: use `lock` in C# or `SyncLock` in VB.NET, or an atomic once-flag such as `Interlocked.CompareExchange`. Example in C#:

```
private readonly object sync = new object();
private bool loaded;

public void EnsureLoaded()
{
    lock (sync)
    {
        if (loaded)
            return;
        // Load the shared data once
        loaded = true;
    }
}
```

On a license other than Enterprise, or when multi-threading is disabled, periodic Script Tasks are queued instead of dispatched concurrently, so two periodic Tasks do not overlap. This queue does not cover expression, event or class-to-class callers, and it does not serialize the method body, so guard shared state in every mode. This behavior applies to FrameworX 10.1.5 and 10.1.6.

Invocation Syntax by Caller Domain

Where a ScriptClass method can be called from depends on two things: the class's **ClassContent** (Class with Methods vs Full Namespace) and its **Domain** (Server or Client).

ClassContent	Domain	Callable from	Notes
Class with Methods	Server	Server-domain scripts; client Tasks (WPF only); Display code-behind on any engine (WPF, Portable, HTML5)	On **Portable / HTML5** the call must be `await`-ed — the method executes on the server and the client waits for the result.
Class with Methods	Client	Client Tasks (WPF only); Display code-behind on any engine	Executes client-side.
Full Namespace	Client	Client Tasks (WPF only); Display code-behind on any engine	Instantiate with `new Namespace.Class()`.
Full Namespace	Server	**Server-domain scripts — called directly** (server Tasks, server ScriptsClasses, server expressions). **Not directly from a client.** Use a server-domain *Class with Methods* as a gateway (see below).	

Client Tasks run on the WPF rich client only. Portable and HTML5 clients have no client-domain Task execution — their client-side logic lives in Display code-behind, and server-domain methods are reached with `await`.

Gateway pattern (Client server-domain Namespace)

To consume a server-domain Namespace class from a client, expose a server-domain *Class with Methods* as a gateway: the client calls the gateway method, and the gateway method — running on the server — calls into the server-domain Namespace class. On Portable / HTML5, `await` the gateway call.

Standard **Class with Methods** ScriptsClasses are unaffected by this restriction: `@Script.Class.ClassName.MethodName(...)` resolves from both domains because the platform pre-instantiates them under a known static path.

Why server-domain Namespace classes can't run client-side. When you instantiate a Namespace class, the object is created **on the client** and all its methods run **client-side**. A server-domain Namespace method may depend on server-local resources — e.g. reading a file on the server machine's disk. Executing it on a remote WPF client or an HTML5 client would fail, because that machine has no access to those resources. Blocking client-side execution of server-domain Namespace methods prevents that whole class of runtime error; the gateway pattern is the supported bridge.

Built-in Classes

All solutions include:

ServerMain

Global methods library executed on server:

- Available to all server-side scripts
- Runs in TServer main thread
- Pre-instantiated by platform
- Special entry points for server initialization

ServerMain has special integration with TServer process. Additional documentation on advanced ServerMain features will be provided in future updates.

ClientMain

Local methods library executed on each client:

- Runs independently per client
- Pre-instantiated for each client connection
- Access to local display context
- Ideal for UI-specific operations

Library Import/Export

ScriptClasses can be shared between solutions using the Library feature:

Export to Library

1. Select ScriptClass in table
2. Click **Export to Library**
3. Class is saved to central `Library.dbsln` repository

Import from Library

1. When creating new class, select **Get from Library**
2. Browse available classes in Library.dbsln
3. Select and import desired class

The Library serves as a centralized repository for reusable components across all your solutions.

Cross-Language Interoperability

FrameworX enables calling between languages transparently:

csharp

```csharp
// C# ScriptClass calling Python method
double result = @Script.Class.PythonClass.calculate_value(10, 20);
```

python

```python
# Python ScriptClass calling C# method
efficiency = TK.Script.Class.CSharpClass.CalculateEfficiency(95.5, 100.0)
```

For detailed Python/.NET integration including type conversions and limitations, see [Python.NET Integration (Reference)]

Accessing Solution Objects

ScriptClasses can directly access all solution namespaces:

csharp

```csharp
public void UpdateProduction()
{
    // Access Tags
    double rate = @Tag.ProductionRate;

    // Access Alarms
    bool hasAlarm = @Alarm.HasActive("Line1");

    // Access Historian
    var history = @Historian.GetValues("Temperature", 100);

    // Access other Script Classes
    var result = @Script.Class.OtherClass.Method();

    // Access Script Tasks
    @Script.Task.MyTask.Run();
}
```

External References

To use external .NET assemblies:

1. **Add Assembly Reference**

- Go to Scripts References
- Add external DLL location
- See [Scripts References (Reference)] for details

2. **Add Namespace Declaration**
   - Open Code Editor
   - Click Namespace Declarations button
   - Add namespaces from referenced assembly
3. **Use in Code**
   - Call methods from external assembly
   - No `using` statements in code itself

Compilation and Build Order

Build Process

1. Classes compile in Build Order sequence (lowest first)
2. Cached compilation for unchanged classes
3. Full rebuild clears cache

Circular References

Avoid circular references between classes. While not blocked by the system, they cause:

- Compilation failures during full builds
- Maintenance difficulties
- Debugging complications

If circular references exist, temporarily comment one reference to allow compilation.

Best Practices Checklist

- **Use descriptive names** - Class and method names should indicate purpose
- **Add XML comments** - Document methods for IntelliSense
- **Handle exceptions** - Even with automatic protection, validate inputs
- **Avoid circular references** - Design clear dependency hierarchy
- **Use standard classes** - Only use Full Namespace when absolutely necessary
- **Test incrementally** - Verify Build Status after each change
- **Use Namespace Declarations** - Never put `using` statements in code

---

Troubleshooting

Red X Build Status

- Double-click to see specific errors
- Check for missing semicolons or braces
- Verify all referenced tags/objects exist

Methods not accessible

- Confirm Build Status is green
- Check Domain (Server/Client) matches usage location
- Verify syntax: `@Script.Class.ClassName.MethodName()`

Class wrapper errors

- Remove any `public class` declarations
- Remove namespace definitions (unless Full Namespace type)
- Remove `using` statements from code (use Namespace Declarations button)

Cannot find external types

- Add assembly reference in Scripts References
- Add namespace via Namespace Declarations button
- Verify DLL compatibility with target framework

In this section...
