



# TJScript

**Scripting Language Specification v3.0**

Tommy van der Vorst ([tommy@mycms.nl](mailto:tommy@mycms.nl)), 2006-2008

# Contents

- Language Specification ..... 3**
  - Why A New Language?..... 3
  - Engine Internals..... 3
  - Script Anatomy ..... 4
  - Language Data Types..... 4
  - Method call syntax ..... 4
  - Static method calls ..... 5
  - Branching Statements ..... 5
  - Variables ..... 5
  - Functions ..... 5
  - Scopes..... 5
  - Delegates ..... 6
  - Iterators..... 6
  - Reflection ..... 6

## Language Specification

TJScript is a complete scripting environment that makes scripting of application fast and easy. Scripts are compiled to a bytecode language, allowing for efficient execution. The API is as clean as possible. The syntax is probably most like JavaScript, although greatly simplified. The scripting language is mainly focused on getting things done in the native environment; Logic should be in native code, and this scripting language is designed to glue those native parts (or components) together. To reduce complexity of the language and parser/compiler, some limitations do apply:

- Methods and fields are essentially the same. Calling `name()` is the same as calling `name` without parentheses. Setting the name is done by calling `set(name=...)` (by convention). Other constructs, like the `index[]`-operator or iterators in `for`-constructs, are also based on simple method calls. This allows for a very simple native interface. It allows generalization of many things in the scripting engine, so many things we're made very easy and fast by this decision.
- TJScript doesn't provide any standard classes other than iterator and value classes (string, integer, double, boolean, null). Value classes can only be instantiated literally. It is up to the embedding application to provide base classes. The TJScript library provides some base classes, but it is up to the application to use them or not
- TJScript's compiler and `ScriptContext` are threadsafe, so you can use them from any thread. Scripts can run in a separate thread, which implies that all classes and API's should be threadsafe. `ScriptContext/VM` can only execute one script at a time per context. Scriptable objects may be shared by different threads (even between scripts that are executing at the same time), but only if they are threadsafe.
- TJScript is not platform dependant in any way, except for Standard C and STL libraries. The library used for parsing is Spirit, which is available on a large number of platforms. TJScript might also use some functions from `TJShared`.

### Why A New Language?

One of the main reasons for choosing to develop a new language for scripting is the complete control we have over it; should we want to implement special syntax for show control (maybe for describing tracks inline in the language), we could add this relatively easy. Other engines we looked at either had some licensing issues or were just too heavy for this purpose.

TJScript consists of roughly 5000 lines of code, which is very small compared to other engines. It integrates tightly with `TJShared` memory management and internal objects and can therefore perform better than alternatives (at least in theory) when used by applications that also use `TJShared` memory management.

### Engine Internals

Although you will probably never need to know anything about how the scripting engine works, it is useful to know the global composition of it. First of all, there is the language parser. When a script is

executed, the first step the scripting engine takes is converting the human-readable script into an intermediate language, called 'bytecode'.

This bytecode is then stored in a separate object (CompiledScript). If you want to execute the same script multiple times, it is faster to execute the compiled script several times than parsing and executing the script each time.

The bytecode is read and executed by the Virtual Machine (VM). The VM holds all variables and stacks needed throughout script execution. The VM decides which objects are available to the script.

The scripting engine always throws exceptions as ScriptException to the host application (TJShow). Errors can occur during parsing (parse-time: syntax errors or other mistakes in the code) and during execution (run-time: usage of inexistent or unavailable objects or methods and such).

### **Script Anatomy**

Each script consists of one or more statements, always followed by a semicolon. Statements may be on the same line or each on a different line. Literals must be on the same line.

### **Language Data Types**

The data types provided by the language are only the most basic ones. The scripting language is designed around the philosophy that the host application will implement all needed additional types (for example for dates, times, complex numbers) through classes. The built-in types are:

- Integer (literally represented as 100 or -100), representing a real, positive or negative number
- Boolean (literally represented as true or false)
- Double (100.055 or -100.055) decimal real number
- String (literals like "everything enclosed between these")

The basic types can be converted into each other when needed. This happens automatically, but could cause errors (for example, adding "1.0" + 1.0 will not always give the desired result).

### **Method call syntax**

When 'test' is an object instance in the current scope, any method or member of test can be called with the following syntaxes:

```
test.someMember;  
test.someMethod();  
test.someMethod(parameterA = "valueB", parameterB = 1.0, parameterC =  
false);
```

Chaining method calls is possible too:

```
test.someMethod().someOtherMethod();
```

Where someOtherMethod is called on the result of test.someMethod() (if any). Method names are case-sensitive by default, however the host application may drop this restriction if desired (TJShow methods are case-sensitive by default).

## Static method calls

TJScript also supports static method calls. Static methods are provided by types. A type can be identified using angle brackets ('<Type>' syntax). A very useful type is the built-in 'Math' type. You can use it to perform various calculations:

```
var value = <Math>.sin(0.2323);
```

Will calculate the sine of 0.2323 and store it in the variable 'value'.

## Branching Statements

Branching can be done like this:

```
if(somevar == "somevalue") {  
    ...  
}  
else {  
    ...  
}
```

The following operators usually work on basic data types: ==, !=, >, <, !, &&, ||, ^, +, -, /, \*

## Variables

Variables can be created (in the current scope only) using the following syntax:

```
var x = "somevalue";
```

The created variable will be accessible from the current scope and nested scopes. To store persistent values, use an interface provided by TJShow's API.

## Functions

Functions can be defined using the following syntax:

```
function multiply(a,b) {  
    return a*b;  
};  
multiply(a=100, b=20);
```

Or, in delegate syntax:

```
var multiply = function(a,b) {  
    return a*b;  
}
```

Functions can be passed as a parameter to another function or stored in a variable. This allows for a construct called 'delegate'. Although this works perfectly, you cannot use function-delegates for native functions, as functions can never be called from outside the script. This is because a function might use some other function defined in the script. Therefore, there is a separate 'delegate' syntax, which creates true delegates (see below).

## Scopes

Using the always available functions *delete*("varname") and *exists*("varname") you can delete variables or check if they are set.

## Delegates

Delegates are blocks of code that can be called from native code when needed, even when the script where the delegate was defined is not running anymore. Delegates can be used to handle certain native events, for example. Delegates cannot reference variables defined in the script, but do have access to all objects normal scripts have access to. A delegate is defined as follows:

```
var del = delegate {
    alert("hello");
};
```

The variable 'del' now contains a delegate, which can be used as a normal variable. To use it, pass 'del' to a native function that accepts and uses delegates. For example, some implementations provide a 'defer' method, that executes the delegate asynchronously in another thread. You can use this with the delegate defined above:

```
defer(del);
```

## Iterators

Some methods return a list of objects, which you want to iterate over in your script. For example, TJSHow provides a list of tracks in a timeline. To iterate over them, you can use the iterator syntax, which is a really easy way to do iteration:

```
for(var track: timeline.tracks) {
    alert(track.name);
}
```

This will show the name of each track in the main timeline. Another useful class is the 'Range' class, with which you can iterate over a range of numeric values:

```
var n = 0;
for(var i: new Range(0,100)) {
    n = n + i;
}
alert(n);
```

The variable 'i' will take the values 0, 1, 2.....100 (so the range is 'inclusive'). In this example, the variable 'n' will contain the sum of 0+1+2+...+100 (which is 5050).

## Reflection

Most objects and types support a method 'class', which returns a string explaining the class name of the object or type. An example:

```
alert(network.class);    shows 'class tj::show::script::NetworkScriptable'
alert(<Math>.class);    shows 'class tj::script::ScriptMathType'
```

Some objects support a method called 'members' that returns an iterator, which you can use to list members of an object:

```
for(var m: <Math>.members) {
    log(m);
}
```

```
}
```

This little script will print a list of all members of the <Math> type to the log window.