weroSoft
Consulting, Development & Education

Expression Library

Programmer's Guide

2017

Part 1

Part 2

Part 3

List of Tables

List of Figures

List of Code

Terms and Abbreviations

Terms / Abbreviation	Description
2D Graph	A component of the expression library, which allows to draw mathematical expressions to a graph in a 2-dimensional Cartesian Coordinate System.
Constant	A value which is immutable.
Core Library	The assembly containing the basic functionality of the programming environment of the company weroSoft AG.
Data Factory	The data factory is a component of the expression library. It creates and populates real objects based on string data.
Expression	A string containing constant data or variable references combined with some functionality calls and concatenations using some operators
Expression Manager	The most important type which is responsible for solving expressions.
Expression Library	The assembly containing the code for solving expressions.
Field	A data element in a type.
HPL	Heron Programming Language. The language developed by weroSoft AG.
Literal	An immutable data element representing a concrete information like a character, a string, a number or other discrete data.
Operand	A term representing data used for working with an operator.
Operator	A discrete functionality represented by a few characters like +, %, is, && and more.
Operator Library	The assembly containing the most of the code implementing the operators within the Expression Library.
Property	A data element in a type.
Term	Generic name of the elements within an expression. Each term is a part which is parsed and interpreted. A term can be an operator, an operand, a constant value or a method call.
Type	A type implements the logic of particular functionality. It is a basic construct of the .NET Programming System.
Type Manager	The responsible part of the Expression Library for resolving types.
Variable	An object storing a value of a particular type.
Variable Manager	The responsible part of the Expression Library for resolving variables-
weroSoft	The company responsible for providing HPL and the Expression Library.

Part 1

Introduction

Heron is the common Server-Framework of the company weroSoft AG for building client server applications. The framework offers default functionality for easy creating own applications and using already existing modules (called engines) as well.

One of such an existing module is the workflow engine which offers the possibility to define own workflows using the Heron Programming Language (HPL). Beside the HPL we have added the «Expression Library» which introduces the possibility to execute «Expressions» based on strings for controlling data content by algorithms. The «Expression Library» can also be used without Heron or any other programming element of the company weroSoft AG. It is a library which is independent of all other elements (except the base library of weroSoft).

This documentation introduces the library by explaining «Expressions» and how they can be used. The usage of the data factory and the usage of the two dimensional simple graphics creation are explained as well.

What an «Expression» is

An «Expression» in the sense of HPL is a string containing constant data or variable references combined with some functionality calls and concatenations using some operators. The following table shows some samples:

Expression	Explanation
42	A simple constant integer value
34.78E-2	A constant real value using the science notification
“My demo text”	A simple constant string value
true	A simple constant Boolean value
40 + 2L	A simple mathematical calculation using two constant values
variable = 42	A simple assignment of a constant value to a variable
variable = Math.Sin(xVariable)	A trigonometric function using a variable with assignment of the result to another variable
variable == 5.45	Check of a variable to equality to a constant value
(2+3) * -4	Calculation of a value controlling the precedence and using a negative value for multiplication.

Table 1 Samples of expressions

There are only logical restrictions to combine the different elements. The Expression system is based on the definitions of the programming language C#.NET. The logic of the expressions is based on the appropriate operators and precedence rules of the formal definition of C#. To become familiar with all these rules, it is not necessary to study the appropriate technical documentations of C#.NET.

All the necessary information for working with the weroSoft «Expression Library» is provided within this documentation including all the details to programmatically use the Library.

What the «Expression Library» is for

The «Expression Library» is introduced to bring the world of «Code Execution» and the «Free Definition of Formulas in an existing application» together. So, the primary goal of this component is to allow the applications user expressing any formula of mathematics, string check or dynamic data definition. This shall be possible without programing any checks or computations of the formula programmed by the developer.

Performance

Due to the fact, that an expression basically is a string, which must be parsed, checked, transformed and executed, the solving of expressions is not as fast as precompiled code is. Depending on the first or second execution of the same expression and on the complexity of the expression the Expression Library uses 1 Microsecond to 1 Millisecond for executing one expression term.

To get an estimation of speed you simply count the amount of terms in your expression and this is a brief indication of the time used for evaluation in milliseconds. Due to internal caching mechanism, this may differ in different scenarios.

Formats in this document

This document uses the weroSoft Formats for documentations in the following way.

- Here is a rule
- Here is another rule

- Here is a definition
- Here is another definition

- Here is an annotation
- Here is another annotation

If you have any remarks or feedbacks to us, either for this documentation or the implementation of the Expression Library and its possibilities please contact us using one of the following possibilities:

Type of Contact	Contact
Support	support@weroSoft.net
Licensing	support@weroSoft.net
Other	info@weroSoft.net

Table 2 Contact to weroSoft AG

Part 2

Expressions

An Expression is always a string containing symbols which may be resolved to a data elements. The string is parsed and afterward interpreted to either a useful result or an error. Although the «Expression Library» supports a wide range of possibilities, not all the details of C#-Expressions are supported. This chapter introduces the formal definition of expressions in the sense of the weroSoft «Expression Library».

Formal definition of Expressions

The formal definition of an expression is based on the definition of expressions in C#. The Expressions are constructed from operands and operators. The operators of an expression indicate which operations to apply to the operands. Examples of operators are «+, -, *, /, and new». Examples of operands are literals, fields, local variables, and expressions.

The definition above results in the formal definition as shown below:

Part		Explanation

`Expression`	`:=`	`{ Literal |`
		`  Field |`
		`  Property |`
		`  Operator |`
		`  Method |`
		`  Expression }`

`Literal`	`:=`	`StringConstant |`
		`CharacterConstant |`
		`NumericConstant |`
		`BoolConstant |`
		`BracedConstant`

`Field`	`:=`	`(* Field definition like Object.Field *)`
`Property`	`:=`	`(* Property definition like Object.Property *)`

`Operator`	`:=`	`(* An operator like it is defined in the section`
		`   «oprators» *)`

`Method`	`:=`	`(* A method call to an object like Object.Method() *)`

Figure 1 Formal Definition of an expression

Supported Elements in Expressions

The section before shows the possible content in a very formal definition. Due to the big amount of different details, the formal description is not complete. To give a more complete impression about the possibilities of the Expression Library, this chapter describes the elements in a more common way in the following sections.

Literals

The Expression Library allows the usage of literal data. The literals are typed and must be interpreted as shown in the table below. Note that the third column of the table represents the effective types how the literal is interpreted in an expression.

Literal type	Description	Type	Sample
Single Character	Represents a character	char	'a'
String	Represents a string	string	"This is a sample text"
Boolean	Represents a Boolean value	bool	true, false
Integer number	Represents an integer number	int	42
Real number	Represents a real number	double	3.142 3.142e-10 3.142e10

Date	Represents a date, a time or a time stamp in different representations.	DateTime	{15-04-24D}
			{2015-04-24D}
			{2015-04-24 12:55D}
			{2015-04-24 12:55:33D}
			{2015-04-24 12:55:33.125D}
			or
			{12:55D}
			{12:55:33D}
			{12:55:33.125D}

Timespan	Represents a timespan in different representations	TimeSpan	{5.04:10:10.234T}
			{5.04:10:10T}
			{04:10:10.234T}
			{04:10:10T}

Guid	The GUID modifier is taken from registry version of printed GUID.	Guid	{706D9474-AFA0-4D81-9A7C-61EDAA924B19}

Table 3 Definition of Literals

- The maximal and minimal values of the read types like numeric or date time values are kept like the definitions made by .NET
- The way how literals of date, time and Guid are expressed differs from the C# Standard. It is only used in the weroSoft «Expression Library».

Escaping and Modifiers of Literals

In some cases, it is necessary to escape or modify literals to get a correct result.

Escape Definitions

An escaped character is a character which is prefixed by a backslash character. Note that escape sequences are allowed in strings and characters as well. The Expression Library supports the following escape sequences.

Escape	Description	Sample
\’	Single Quote character	`'\''` → a single quote as character `"\'"` → a single quote as string
\”	Quote character	`"\""` → a quote as string `'\"'` → a quote as character
\\	Backslash character	`"\\\\Server\\Share"` → A share definition as string. This sample uses first two backslashes and second one backslash.
\a	Matches a bell (alarm) character, \u0007.	`"Test: \a"`
\t	Matches a tab, \u0009.	`"Test: \t"`
\r	Matches a carriage return, \u000D. Note that \r is not equivalent to the newline character, \n.	`"Test: \r"`
\n	Matches a new line, \u000A.	`"Test: \n"`
\v	Matches a vertical tab, \u000B.	`"Test: \v"`
\f	Matches a form feed, \u000C.	`"Test: \f"`
\xnn	Matches an ASCII character, where nn is a two-digit hexadecimal character code.	`"Test: \x41"` → Results in a A-character
\unnnn	Matches a UTF-16 code unit whose value is nnnn hexadecimal.	`"Test: \u0041"` → Results in a A-character

Table 4 Supported Escape characters in literals

Modifiers of literals

A modifier is a character sequence which is post-fixing a numerical or a braced expression. Note that modifier sequences are only allowed to be used in strings or braced constants. The Expression Library supports the following modifiers.

Modifier	Description	Sample
Byte	The byte modifier is used to push an integer literal to be interpreted as a byte value.	42B
Signed Byte	The signed byte modifier is used to push an integer literal to be interpreted as a signed byte value.	42SB
Short	The short modifier is used to push an integer literal to be interpreted as a short value.	42S
Unsigned Short	The unsigned short modifier is used to push an integer literal to be interpreted as an unsigned short value.	42US
Unsigned Integer	The unsigned modifier is used to explicitly express an integer value as unsigned.	42U
Long	The long modifier is used to push an integer literal to be interpreted as a long value.	42L
Unsigned Long	The long modifier is used to push an integer literal to be interpreted as a long value.	42UL
Float	The float modifier is used to push an integer or real literal to be interpreted as a float value.	42.42F
Decimal	The decimal modifier is used to push an integer or real literal to be interpreted as a decimal value.	42.42M
Exponential	The exponential modifier is used for number scientific usage.	0.3E10
DateTime	The DateTime modifier is used to push a time literal to be interpreted as a DateTime value.	{12:33:56D}
TimeSpan	The TimeSpan modifier is used to push a time literal to be interpreted as a TimeSpan value.	{12:33:56T}
Hexadecimal notation	In some rare cases, we like to use hexadecimal notation which are recognized according the samples. Note that the character 'x' may be capitalized too.	0XA1 0X00A1 0X00A100A2

Table 5 Modifier of Literals

- Note that the expression library always uses capital letters. This is different to C# which does not know as many modifiers but support small and capital letters as well.

Operators

The Expression Library supports a lot of operators. This chapter defines the supported operators and their precedence ordered by main precedence groups.

Primary Operators

Operator	Name	Remarks
x.y	Member access	The member «y» is accessed in the object «x».
f(x)	Method invocation	The method «f» is evaluated.
a[x]	Indexed access	The element with index «x» of the array «a» is accessed.
x++	Post-increment	The variable «x» is incremented after evaluation
x--	Post-decrement	The variable «x» is decremented after evaluation
new T(…)	Object creation	An object of the type «T» gets created
new T[…]	Array creation	An array of the type «T» gest created

Table 6 Primary operators

Unary Operators

Operator	Name	Remarks
+x	Plus sign	The value «x» is explicitly positive signed
-x	Mathematical negotiation	The value «x» gets mathematically negotiated
!x	Logical negotiation	The value «x» gets logically negotiated
~x	Bitwise negation	The value «x» gets bitwise negotiated
++x	Pre-increment	The variable «x» is incremented before evaluation
--x	Pre-decrement	The variable «x» is decremented before evaluation
(T)x	Cast	Explicitly converts the type of «x» to the type «T»

Table 7 Unary operators

Multiplicative Operators

Operator	Name	Remarks
x * y	Multiplication	«x» and «y» gets multiplied
x / y	Division	«x» and «y» gets divided
x % y	Reminder	The reminder of the division «x / y» gets evaluated

Table 8 Multiplicative operators

Additive Operators

Operator	Name	Remarks
x + y	Addition, string concat-enation	«x» and «y» gets added if they are numeric values, and concatenated if they are two strings.
x - y	Subtraction	«x» and «y» are subtracted

Table 9 Additive operators

Shift Operators

Operator	Name	Remarks
x << y	Shift left	The bits of the value «x» are left shifted by the amount of «y» bits
x >> y	Shift right	The bits of the value «x» are right shifted by the amount of «y» bits

Table 10 Shift operators

Relational and Type Operators

Operator	Name	Remarks
x < y	Less than	True if «x» is less than «y»; otherwise false
x > y	Greater than	True if «x» is greater than «y»; otherwise false
x <= y	Less or equal than	True if «x» is less or equal than «y»; otherwise false
x >= y	Greater or equal than	True if «x» is greater or equal than «y»; otherwise false
x is T	is of type	True if «x» is of Type «T»; otherwise false
x as T	conditional cast	returns «x» as Type of «T» or null if this is not possible

Table 11 Relational and Type operators

Equality Operators

Operator	Name	Remarks
x == y	Equal	True if «x» is equal as «y»; otherwise false
x != y	Not equal	True if «x» is not equal as «y»; otherwise false

Table 12 Equality Operators

Logical, Conditional, and Null Operators

Operator	Name	Remarks
x & y	Logical AND	Integer bitwise AND, Boolean logical AND
x ^ y	Logical XOR	Integer bitwise XOR, Boolean logical XOR
x | y	Logical OR	Integer bitwise OR, Boolean logical OR
x && y	Conditional AND	Evaluates «y» only if «x» is true
x || y	Conditional OR	Evaluates «y» only if «x» is false
x ?? y	Null coalescing	Evaluates to «y» if «x» is «null», to «x» otherwise
x ? y : z	Conditional	Evaluates to «y» if «x» is true, otherwise to «z»

Table 13 Logical, Conditional, and Null Operators

Assignment Operators

Operator	Name	Remarks
x = y	Assignment	Assigns the value «y» to the variable «x»
x += y	Addition Assignment	Equivalent to «x = x + y»
x -= y	Subtraction Assignment	Equivalent to «x = x - y»
x *= y	Multiplication Assignment	Equivalent to «x = x * y»
x /= y	Division Assignment	Equivalent to «x = x / y»
x %= y	Reminder Assignment	Equivalent to «x = x % y»
x &= y	AND Assignment	Equivalent to «x = x & y». Performs a bitwise logical AND operation on integral operands and logical AND on bool operands
x |= y	OR Assignment	Equivalent to «x = x | y». Performs a bitwise logical OR operation on integral operands and logical OR on bool operands
x ^= y	OR Assignment	Equivalent to «x = x ^ y».
x <<= y	Shift left Assignment	Equivalent to «x = x << y»
x >>= y	Shift right Assignment	Equivalent to «x = x >> y»

Table 14 Assignment Operators

Expression Library Components

As already mentioned, the «Expression Library» is intended to define an expression by a string and afterward interpret this string to get a result. Due to the expression, may contain .NET types and variables the «Expression Library» must support these two components too.

Result of the basic requirements above, the «Expression Library» consists of a few components working together. The figure below visualizes the block diagram of the used components in the «Expression Library» and the next sections explain what you need to know about the defined libraries.

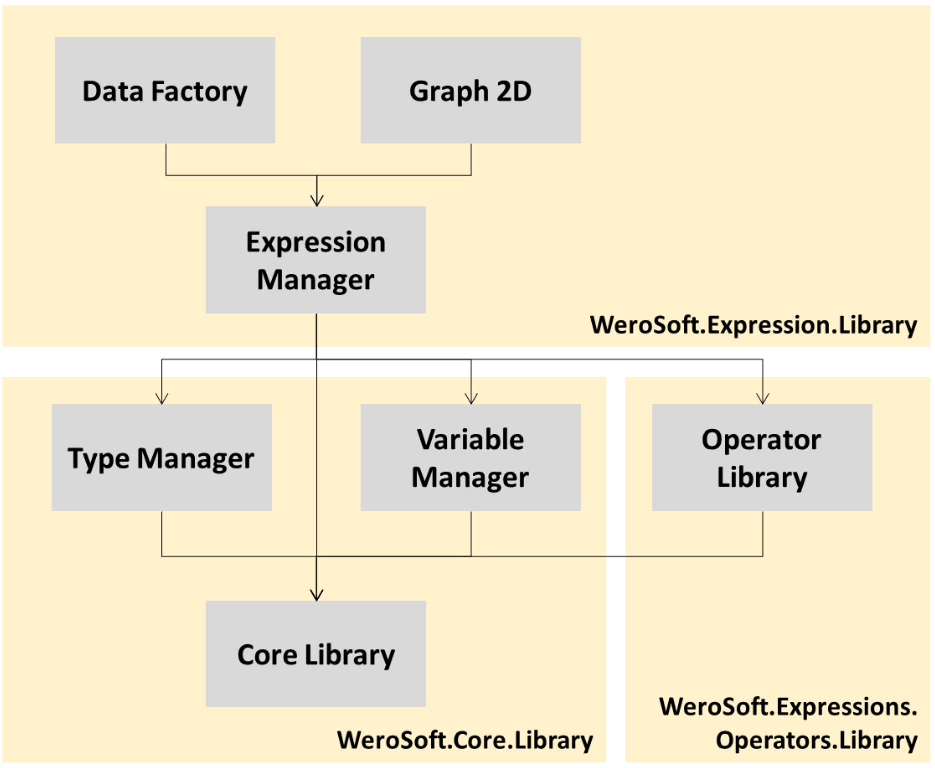


Figure 2 Block diagram of the used components

If you are using the expression library, you need a couple of assemblies. Please add the references to your project as shown below:

- WeroSoft.Core.Library
- WeroSoft.Expressions.Library
- WeroSoft.Expressions.Operators.Library

- The Operators library is basically loosely coupled. This means that if it missed, no error happens on loading the code, but your expressions may always result in an syntax error because the operators are missed.

Expression Manager

The Expression Manager is responsible for tokenizing, parsing and evaluating a given expression. Internally the Expression Manager delegates these tasks to a couple of components as shown in the diagram below.

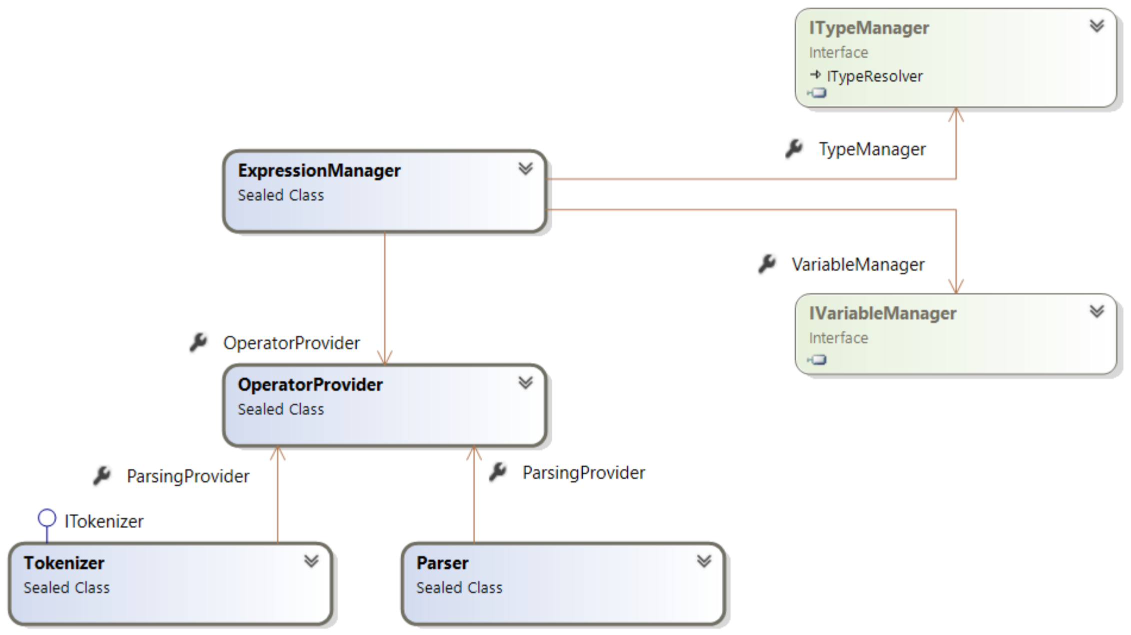


Figure 3 Class diagram of the Main components

The responsibilities of the different components are defined as follows:

Component	Responsibility
Expression Manager	Main object to work with. The Expression Manager coordinates all the work with the other components and manages the expression cache of already parsed and evaluated expressions to speed up repetitive work.
Operator Provider	Self-configuring component knowing all available operators and parser used during parsing and evaluating the expression.
Tokenizer	Implements the first step of the parsing process and reduces a given expression string to tokens. The content of the tokens is already in a form to support the parsing algorithm.
Parser	Implements the second step of parsing and builds an expression tree, which can directly be evaluated by the Expression Manager.
Type Manager	The Type Manager is responsible for configuration of all .NET types. Read more detailed information section «Type Manager».
Variable Manager	The Variable Manager is responsible for handling the known variables usable in expressions. Read more detailed information section «Variable Manager».

Figure 4 Responsibilities of the main components

The behavior of the Expression Manager and its subordinated components is defined to follow a couple of rules. Ensure following these rules to successfully use the Expression Library.

Basic rules of the Expression Manager

- The Expression Manager can be instantiated as often as needed
- The Expression Manager follows an easy to use paradigm which allows to evaluate an Expression using as less code as possible
- For more complex scenarios, the Expression Manager allows configuring the Type Manager and or the Variable Manager individually
- The Expression Manager uses the Operator Library to be fully operable of all documented operators according chapter
- Operators are not over-writable, but can be extended by new ones

Basic rules for Expressions

- An Expression is always defined by a string
- The Expression is only evaluated if it is syntax error free
- An Expression can integrate all supported elements
- Names of variables and .NET Elements like types and Elements are case sensitive
- An Expression has no length restriction
- The nesting of an Expression may be as complex as needed
- Expressions may be cached to support a higher performance on evaluation

Basic rules for the Type Manager

- .NET Types and their elements are resolved by the Type Manager
- In the standard configuration, the Type Manager supports not all .NET types
- The Type Manager may be configured to allow using all needed types
- The Type Manager supports a simple default configuration for daily work
- The Type Manager can be replaced by a customer one

Basic rules for the Variable Manager

- Expressions allows using variables
- Variables are always resolved using the Variable Manager
- Variables are handled type safe, as long as you are using the weroSoft Library possibilities
- The Variable Manager can be replaced by a customer one

Type Manager

The Type Manager is a component which allows configuring the usage of .NET types which may be used by the Expression Manager. In addition to that, it provides all needed methods for finding a particular type or a member of a type including fields, properties or methods.

Default configuration

The Type Manager has a default configuration which supports resolving types in the assemblies below:

- mscorlib.dll
- system.dll

By default, not all contained types in the two assemblies above are resolved. The Type Manager enables only the types belonging to the namespaces below:

- System
- System.Collection
- System.Collections.Concurrent
- System.Collection.Generics

HPL Type support

In addition to the direct usage of the types of .NET, the Type Manager allows using aliased types within expressions. The most of these aliased types are the same as in C#, but there are also some additional types defined and supported as shown in the table below.

- The types shown in the table below are part of HPL and fully compatible to the appropriate .NET types

Type keyword	Aliased .NET Type	Type keyword	Aliased .NET Type
object	System.Object	ulong	System.UInt64
bool	System.Boolean	float	System.Single
byte	System.Byte	double	System.Double
sbyte	System.SByte	decimal	System.Decimal
short	System.Int16	char	System.Char
ushort	System.UInt16	string	System.String
int	System.Int32	guid	System.Guid
uint	System.UInt32	datetime	System.DateTime
long	System.Int64	timespan	System.TimeSpan

Table 15 HPL alias types supported by the Type Manager

Replacing the Type Manager

Basically, the type manager is designed to be replaceable by a Customer Type Manager. For doing that, you must implement the appropriate interfaces as shown in Figure 5.

- Note that implementing these interfaces targeting an own type manager results in extensive work, if you like to fully support Type Access of .NET.
- The better way would be to implement the interface in a new class and route the requests to an aggregated usage of the standard Type Manager.
- This would allow combining an own Type Manager with the one of our library.

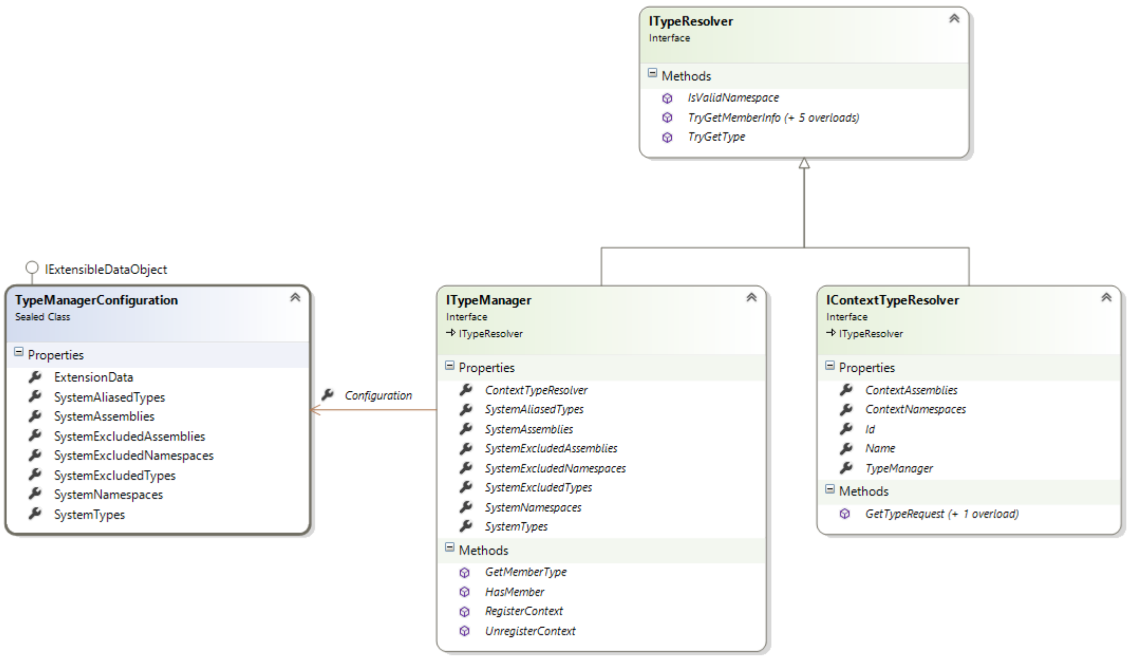


Figure 5 Base interfaces and types of the Type Manager

Variable Manager

The Variable Manager is intended to allow using variables in the expressions. Each variable which shall be used within an expression must be registered to the Variable Manager. Note that the variables may consists of any type as long as the Type Manager allows resolving the needed type.

The Variable Manager consists of an implementation of the appropriate interfaces as shown below. The Variable Manager offers three methods, for registering, accessing and removing a variable.

Note that as easy as using the Variable Manager is, as easy is it to define own variables in your code, to use the manager. The figure below shows on the right-hand side the default implementation to express a variable usable in HPL or on using the Expression Library as well.

A variable is defined by its type, its name and its modifier which is defining whether a variable can normally be used or whether its value is constant or read-only.

- A normal variable can be set, and read as often as needed
- A constant variable can only be set once directly on the variable definition
- A read-only variable can be set once in its life cycle, but must not be initialized directly on its definition
- Variable of any types are initialized automatically according their defaults. Numeric variables are defined as '0', any reference types are initialized as 'null' and so on.
- Controlling the behavior according the modifier of the variable is not part of the Expression Library or the Expression Manager. This must be performed by the Variable itself.

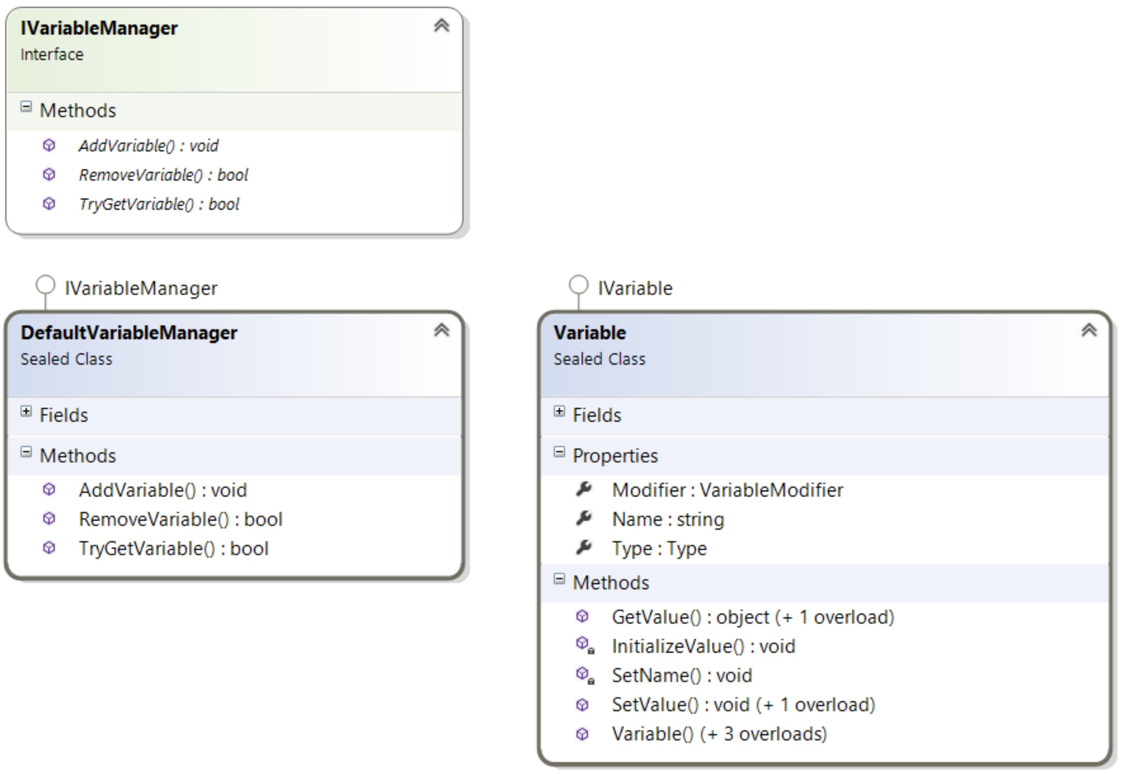


Figure 6 Variable Manager Definition and default implementation

Part 3

Using the Expression Library

The programmatic usage of the Expression Library is very simple. This chapter shows how to use the Expression Library in different scenarios and shows also a few samples of expressions.

Basic scenario using Expression

The basic scenario of usage an expression introduces the minimal components used to resolve an expression. Use the following steps to get a simple expression successfully resolved:

- Create an instance of the Expression Manager
- Create an expression object using the constructor defining the expression string
- Call the method EvaluateExpression() and use the Expression object as a parameter
- Get the result out from the Expression object by accessing the property «Value»

The code below shows how the steps above looks like in reality. The Expression is initialized by the string “2 + 3 * 4” which represents a simple mathematical operation.

```
// Define a default instance of the Expression Manager. This
// instance provides a Type Manager initialized by the defaults
// and a Variable Manager which allows handling variables.
ExpressionManager expressionManager = new ExpressionManager();

// Define an expression by a constant string.
IExpression expression = new Expression("2 + 3 * 4");

// Evaluate the expression.
expressionManager.EvaluateExpression(expression);

// Get the result.
System.Console.WriteLine("{0} = {1}", expression.ToString(), expression.Value);
```

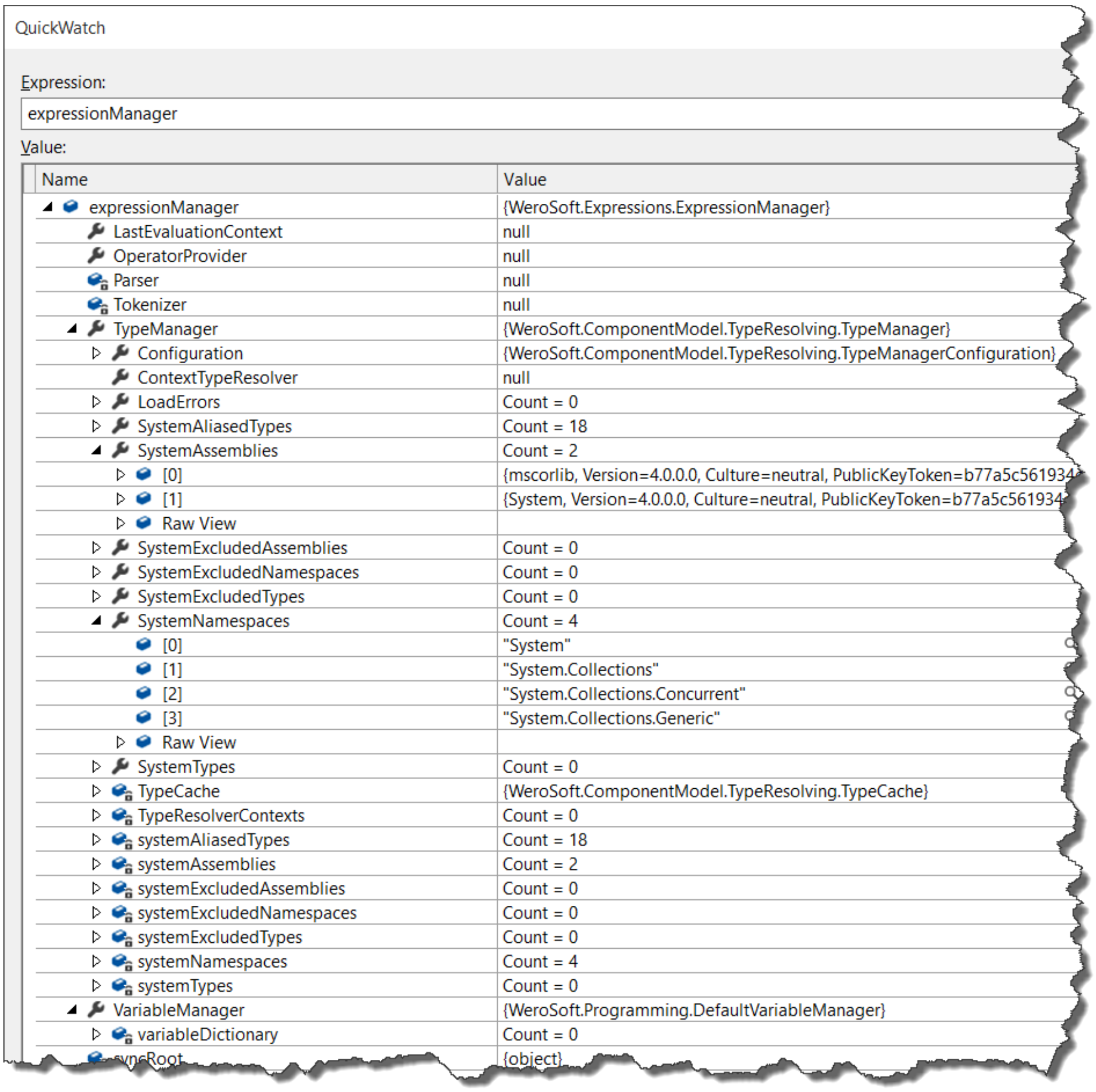
Code 1 Basic scenario for usage the expression library

The call of the method «EvaluateExpression(…)» parses and directly evaluates the expression if it is syntax error free.

- The basic scenario uses a default configuration of the Expression Manager, the Type Manager and the Variable Manager
- There is only a limited access to the Type Manager configured
- The basic scenario allows full featured access including all Operators

Expression Manager

The instance of Expression Manager in this scenario uses the default configuration of the Type Manager and the default Variable Manger as well. These elements are initialized as needed. The figure below shows the content of the Expression Manager directly after the instantiation. Internally in the two types you can recognize some details about the known assemblies, types or variables as well.



QuickWatch

Expression:

expressionManager

Value:

Name	Value
expressionManager	{WeroSoft.Expressions.ExpressionManager}
LastEvaluationContext	null
OperatorProvider	null
Parser	null
Tokenizer	null
TypeManager	{WeroSoft.ComponentModel.TypeResolving.TypeManager}
Configuration	{WeroSoft.ComponentModel.TypeResolving.TypeManagerConfiguration}
ContextTypeResolver	null
LoadErrors	Count = 0
SystemAliasedTypes	Count = 18
SystemAssemblies	Count = 2
[0]	{mscorlib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934
[1]	{System, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934
Raw View	
SystemExcludedAssemblies	Count = 0
SystemExcludedNamespaces	Count = 0
SystemExcludedTypes	Count = 0
SystemNamespaces	Count = 4
[0]	"System"
[1]	"System.Collections"
[2]	"System.Collections.Concurrent"
[3]	"System.Collections.Generic"
Raw View	
SystemTypes	Count = 0
TypeCache	{WeroSoft.ComponentModel.TypeResolving.TypeCache}
TypeResolverContexts	Count = 0
systemAliasedTypes	Count = 18
systemAssemblies	Count = 2
systemExcludedAssemblies	Count = 0
systemExcludedNamespaces	Count = 0
systemExcludedTypes	Count = 0
systemNamespaces	Count = 4
systemTypes	Count = 0
VariableManager	{WeroSoft.Programming.DefaultVariableManager}
variableDictionary	Count = 0
syncRoot	{object}

Figure 7 The internals of the Expression Manager after a default creation

Expression Type

As shown in the introduction of the basic scenario, the Expression Library uses the type «Expres-sion» for handling the data. The expression type is based on two interfaces (see also Figure 9); the common base interface and its inherited generic type.

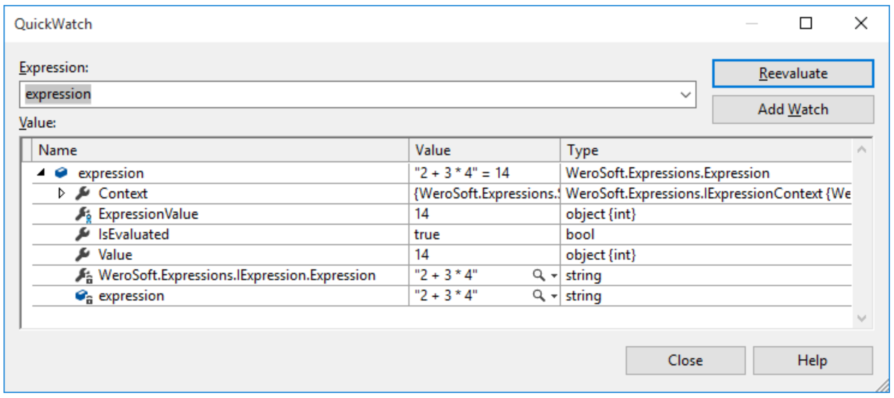
The type is responsible for the following information

- Keeping the expression itself
- Providing a value indicating that evaluation was successful or not
- Providing the result of the evaluation
- Providing a context

The figure below visualizes the data represented by the Expression object how it is used in the basic scenario code.

- If an expression cannot be evaluated due to a syntax error, the expressions property «IsEvaluated» is kept false
- In case of an error the ExpressionManager throws an «SyntaxException»
- Get more information about errors in expression in the «Erroneous Scenario»

The figure below shows the expression of the basic scenario within the «QuickWatch» of the Visual Studio Debugger after the expression has been evaluated.


QuickWatch

Expression: expression

Value:

Name	Value	Type
expression	"2 + 3 * 4" = 14	WeroSoft.Expressions.Expression
Context	{WeroSoft.Expressions.!	WeroSoft.Expressions.IExpressionContext {We
ExpressionValue	14	object {int}
IsEvaluated	true	bool
Value	14	object {int}
WeroSoft.Expressions.IExpression.Expression	"2 + 3 * 4"	string
expression	"2 + 3 * 4"	string

Figure 8 An expression object in the debugger

As mentioned before, the figure below introduces the two interfaces and their concrete implementation for defining expressions. We recommend to use the generic interface in clear distinctive cases of resulting types only.

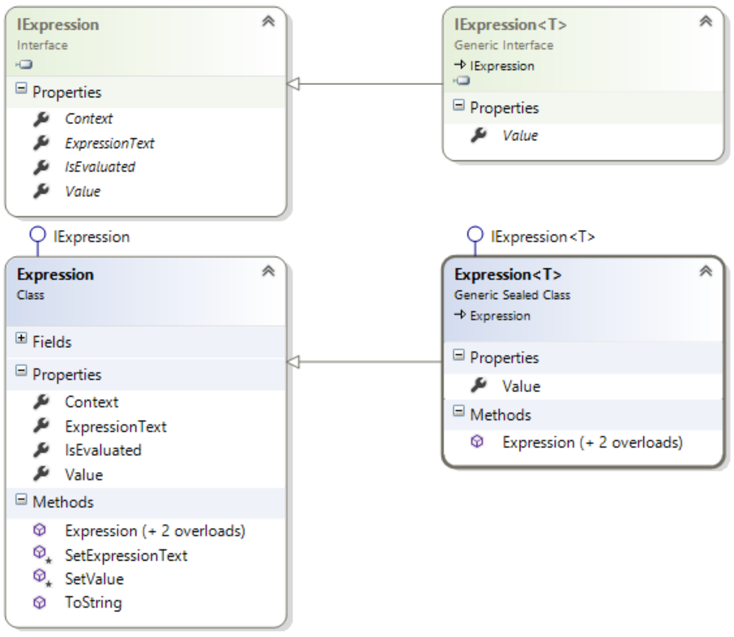


Figure 9 Type Expression Types

Calculating scenario using variables

The calculating scenario introduces a more sophisticated usage of the Expression Library. The sample creates an image containing the graph of a damped vibration using the formula as shown in the figure below.

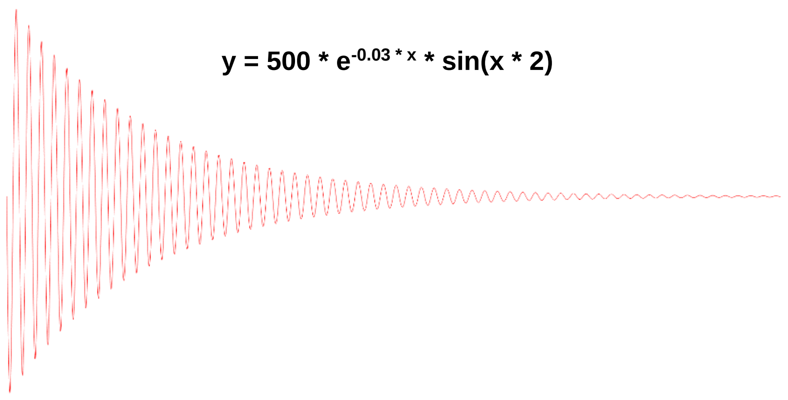


Figure 10 The resulting image of the calculation

Same as the basic scenario, this calculating scenario uses the default initiated Expression Manager and the discrete version of the expression, which is initialized by the appropriate formula as shown in the Figure 10 (see «Code 2» comment C1).

- Note that for calculating of the mathematics the class Math is used
- The class Math is in the namespace System, implemented in the assembly mscorlib.dll
- These elements are loaded automatically by the default configuration of the Expression Manager

The second part of the code (see «Code 2» comment C2) creates two variables named «x» and «y» of the type «double». The variables are instantiated by using the type «Variable» which is part of the weroSoft core library. You'll find the type in the namespace `weroSoft.Programming`. After creating the two variables, they are registered on the Variable Manager which is part of the Expression Manager instance. After registration of the variables they can be used by solving any expression.

The next part of the code (see «Code 2» comment C3) is responsible to prepare the bitmap in the size of a full HD image. In this bitmap, later the graph as shown in Figure 10 is created. After the preparation of the bitmap the loop for calculating the damped vibration is performed (see «Code 2» comment C4).

The loop control is built by a «for-statement» which is based on the usage of the variable known by C# and the Expression Manager as well.

- To keep the example simpler, the loop calculates the amount of points of the horizontal resolution of the image and not on the mathematical angle as expected of the function sinus.
- The steps taken by the calculation is 0.1 unit. This results in 1920 rounds of calculation.
- To successfully compile the appropriate, sample your project must have the assembly «System.Drawing.dll» referenced.

To complete the creation of the image, the first step in the loop is the calculation of the value «y» based on the defined expression (see «Code 2» comment C5). Due to the expressions mathematics, this evaluation results in an assignment of the variable «y».

The last step before drawing is to get the value out of the variables and preparing them in the way that the drawing methods can work (see «Code 2» comment C6).

```
// C1: Use standard expression manager.
ExpressionManager expressionManager = new ExpressionManager();
IExpression expression = new Expression(
    "y = 500 * Math.Pow(Math.E, -0.03 * x) * Math.Sin(x * 2)");

// C2: Define two variables to be used on working with the graphic.
IVariable x = new Variable(typeof(double), "x");
IVariable y = new Variable(typeof(double), "y");
expressionManager.VariableManager.AddVariable(x);
expressionManager.VariableManager.AddVariable(y);

// C3: Create a bitmap using the simple fashion graphics programming interface
// and initialize its basic values.
Bitmap bitmap = new Bitmap(1920, 1080);
Graphics context = Graphics.FromImage(bitmap);
context.SmoothingMode = System.Drawing.Drawing2D.SmoothingMode.AntiAlias;
context.InterpolationMode =
    System.Drawing.Drawing2D.InterpolationMode.HighQualityBicubic;
context.TranslateTransform(0f, 540f);
context.Clear(Color.White);
float x1 = 0f;
float y1 = 0f;
float x2 = 0f;
float y2 = 0f;

// C4: Create the drawing by using the of the maximal size.
for (x.SetValue(0.0); x.GetValue<double>()<192; x.SetValue(x.GetValue<double>()+0.1))
{
    // C5: Evaluate the expression accessing the variables.
    expressionManager.EvaluateExpression(expression);

    // C6: Get the values and converts their values to meet
    // the coordinate system of GDI+.
    x2 = Convert.ToSingle(x.GetValue());
    y2 = Convert.ToSingle(y.GetValue());

    // Draw the line
    context.DrawLine(Pens.Red, x1 * 10, y1, x2 * 10, y2);
    // Prepare the next calculation item and count the amount of loops.
    x1 = x2;
    y1 = y2;
}

// Save the image and dispose the used resources.
bitmap.Save("PerformanceTest.png", ImageFormat.Png);
context.Dispose();
bitmap.Dispose();
```

Code 2 Calculating scenario source

Using the Data Factory

The data factory is a component in the expression engine. It allows creating data object and initialize their values based on strings. Internally the data factory uses the expression engine for transforming and assigning the values given by a data dictionary.

The figure below shows the principle. On the left-hand side of the image is the type to be factored and the data describing the properties and their values. The data is defined in a key-value based collection. Thereby the key defines the name of the property whose value shall be set or the path starting from the given data object to be accessed to set the value. The value defines an expression being computed during the creation and whose value is assigned to the defined property. Finally, the output of the «Data Factory» is the specific data object with its initialized properties.

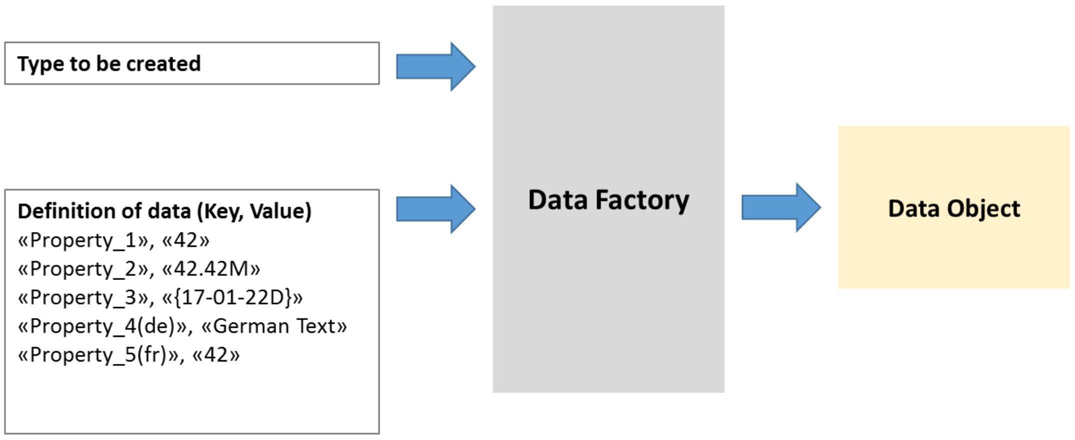


Figure 11 Principle of the usage of the data factory.

The code sample below gives an impression how the data factory is working. After creating and initializing a dictionary of data defining the property name (key) and the expression used to initialize the data object (value), the data factory gets created and afterward used for creating the instance of the type «DataFactoryTestType». Note that this sample only shows one of multiple overloads to create the data object.

```
// Prepare the data dictionary to use.
Dictionary<string, string> data = new Dictionary<string, string>();
data.Add("IntegerProperty", "42");
data.Add("StringProperty", "Test string");
data.Add("DecimalProperty", "42.42M");
```

```
// Create a default instance of the data factory and create the object afterwards.
DataFactory dataFactory = CreateDataFactory();
DataFactoryTestType dataObject = dataFactory.CreateInstance(
    typeof(DataFactoryTestType).FullName, data) as DataFactoryTestType;
```

Code 3 Defining a simple data object using the data factory

The «Data Factory» supports a wide range of possibilities to create the correct data even if the type of the property represents a more complex element like an array, a dictionary another complex type. To address the right behavior, you should respect the possible syntaxes used for expressing the property or path to be assigned.

Syntax rules of the properties

Basically, the properties are addressed using the real name of the property in the target type. The table below shows the supported possibilities.

Syntax	Behavior
Property	A normal text defines a property of data in the created object. If the given name does not exist as a property, the data is read-in but not applied to the business object.
Property(language)	Use this syntax for defining a specific language entry on a property supporting multi language data (localized property or localized string[1]). The language shall be defined by the RFC 4646 (e.g. 'de' for German, 'fr' for French, 'it' for Italian or 'en' for English.
Property.Property. …	The dot is used for cascading properties. A cascaded property originates in an object which is the property itself. The data factory mechanism automatically creates the cascaded objects and assigns the value to the appropriate property. Note that cascading is defined open without any boundaries or restrictions.
Property[index]	Using a square bracket on the property allows accessing or defining a collection. The data factory mechanism defines the collection behind based on the target type. The following types are supported: • Any array type • IEnumerables<T> using a list • ICollection<T> using a list • IList<T> using a list • IDictionary using a dictionary Note that the index itself may be an expression too!

Table 16 Description of property syntax in the data population dictionary

- The automatic dynamic creation of cascaded objects is only possible if the appropriate classes are supporting default construction.

[1] Localized Property or Localized Strings are special types of the weroSoft Core Library to support multiple languages in the same string.

Syntax rules of the values

Because the values are always interpreted as expressions it is basically easy to get them written. But on a more detailed view, it is not easy to work with expression addressing the target type «String», because every value is expressed as a string. This leads to the problem how a string shall be expressed to distinguish a constant string from a string to be interpreted as an expression?

The following code sample explains the problem by example:

- The string "42" is interpreted by the expression engine as value 42 of the type integer.
- The string "Environment.MachineName" is interpreted as property MachineName of the type Evnvironment. The result of this is a string containing the name of the machine executing the code.
- "Text" is tried to be interpreted as name of something → Commonly would result in an exception because "Test" would not exist.
- "\"Test\"" is kept as a string and therefore interpreted as string.
- "\"42\"" is kept as a string and therefore interpreted as string having two characters.

The table below makes a more formal definition. To allow recognizing strings better, the basic string is written in «». This allows easier to detect inner string characters.

Syntax	Behavior of the data factory
«Text»	If the target type is not a string, this form is interpreted as an expression. If the target type is a string, this form is interpreted as a string too!
«"Text"»	If the target type is a string, this form is interpreted as a string. If the target type is any other type this results in an exception.
«[[Text]]»	If the target type is a string, this form is interpreted as an expression. If the target type is not a string this form makes no sense.

Table 17 Description of value syntax in the data population dictionary

Some more complex assignments

The code below shows some more complex scenarios. The scenario is based on a class describing a person and its address. The scenario first defines some discrete string contents for the property «Name» and «First name», afterwards the «Data of birth» is defined as an expression using a braced value and the «Gender» is also set using an expression. The gender is expressed by a value of the enumeration Gender behind the scene.

The second part of the code fragment below defines the data of an address object which is linked using the property «DefaultPostalAddress». This can be recognized on the syntax using a dot in the name. The first part represents the property in the persons' object, the second part of the dotted name represents the property in the address type (e.g. Street). In addition to that we can also recognize that the City is using a «localized property» which allows storing multiply languages of the same information simultaneously.

```
// Define basic values first
Dictionary<string, string> data = new Dictionary<string, string>();
data.Add("Name", "TestName");
data.Add("FirstName", "TestFirstName");
data.Add("DateOfBirth", "{1987-06-28D}");
data.Add("Gender", "Female");

// Define the postal address content
data.Add("DefaultPostalAddress.Street", "Rue da la Gare");
data.Add("DefaultPostalAddress.StreetNumber", "45g");
data.Add("DefaultPostalAddress.City(de)", "Biel");
data.Add("DefaultPostalAddress.City(fr)", "Bienne");
```

Code 4 Defining a person object using the data factory

The next few lines show how to define collection based elements. The type used proposes a collection based property with the name «ListOfSlavesProperty». The collection finally will have 4 elements. Note that the order of assigning details to an object in the collection is not relevant. The index is uniquely defining access to the specific object. The two last elements are again addressing a path containing more than one level of object.

```
Dictionary<string, string> data = new Dictionary<string, string>();
data.Add("ListOfSlavesProperty[0].StringProperty", "String");
data.Add("ListOfSlavesProperty[2].IntegerProperty", "2");
data.Add("ListOfSlavesProperty[3].IntegerProperty", "3");
data.Add("ListOfSlavesProperty[1].IntegerProperty", "1");
data.Add("ListOfSlavesProperty[0].DateTimeProperty", "DateTime.Now");
data.Add("ListOfSlavesProperty[0].TimeSpanProperty", "TimeSpan.FromHours(1)");
data.Add("ListOfSlavesProperty[0].CascadedSlaveProperty.StringProperty", "String");
data.Add("ListOfSlavesProperty[0].CascadedSlaveProperty.DateTimeProperty",
    "DateTime.Now");
data.Add("ListOfSlavesProperty[0].CascadedSlaveProperty.TimeSpanProperty",
    "TimeSpan.FromHours(1)");
```

Code 5 Defining complex collection based and cascaded objects

Using the Graph 2D

Starting with this release the expression engine supports a simple 2D graph for drawing mathematical expression into a «Cartesian Coordinate System. The principle to use this part of the expression engine is kept very simple:

1. Create and initialize an object of the type «MathGraphicsData2D» describing the mathematical definitions of the graph.
2. Create the graph object by instantiating the type «MathGraphics2D»
3. Call the method CreateGraph()
4. Save the image using the method SaveAs on the MathGraphics2D object or get the image using the method GetImage().

The code below shows the usage on a sample creating a graph containing a formula for a vibration.

```
// Arrange test data.
string fileName = "TestImageVibration.png";
MathGraphicsData2D imageData = new MathGraphicsData2D();
imageData.Formula = "y = 100 * Math.Pow(Math.E, -0.015 * x) * Math.Sin(x * 1)";
imageData.XMinimum = -200;
imageData.XMaximum = 400;
imageData.YMinimum = -1000;
imageData.YMaximum = 1000;
MathGraphics2D graphics = new MathGraphics2D(imageData);

// Create the graphics to file.
graphics.CreateGraph();
graphics.SaveAs(fileName, ImageFormat.Png);

// Create the image to stream.
Stream stream = graphics.GetImage(ImageFormat.Png);
```

- The formula above uses the two variables «x» and «y». According the standard usage of these names in the Cartesian Coordinate System, the grapher uses these two names by default as variable names for the calculation.
- You can change these names in your formula, but in this case, you have to define the new names using the property «NameX» and «NameY» too.
- All mathematical definitions for the graph are done in RAD. The current implementation does not support change of this behavior.

Note that saving the image is not mandatory, you can directly get the image as a byte array or alternatively as a WPF «ImageSource» to directly map the image to the screen in an WPF application.

Configuring Type Manager Scenario

Since the Expression Library uses the Type Manager to resolve Types embedded in the expression, it is important to know the types which may be used by default and how to configure the Type Manager to resolve additional types.

As already defined in part 2, the Type Manager resolves only a few basic types by default. All of them are configured in the core assembly (mscorlib.dll) and the base system library (system.dll).

The sample code below shows how the type manager can be configured to allow resolving special types found in the «`System.Net`» namespace located in the appropriate assembly. Note that the difference to the other scenarios basically is to create a Type Manager Configuration first. On this configuration, all used namespaces and assemblies must be configured.

```
// Creates a type manager configuration.
TypeManagerConfiguration configuration = new TypeManagerConfiguration();
configuration.SystemAssemblies = new System.Collections.Generic.List<string>();
configuration.SystemAssemblies.Add(typeof(System.Net.Dns).Assembly.FullName);
configuration.SystemNamespaces = new System.Collections.Generic.List<string>();
configuration.SystemNamespaces.Add("System");
configuration.SystemNamespaces.Add("System.Net");

// Create the type manager and the expression manager.
ITypeManager typeManager = new TypeManager(configuration);
ExpressionManager expressionManager = new ExpressionManager(typeManager, null);

// Register a variable to get the result into the code.
IVariable result = new Variable(typeof(string), "result");
expressionManager.VariableManager.AddVariable(result);

// Define and evaluate the expression.
string test = System.Net.Dns.GetHostEntry("www.weroSoft.net")
       .AddressList[0].ToString();
IExpression expression = new Expression(
    "result = System.Net.Dns.GetHostEntry(\"www.weroSoft.net\")" +
    ".AddressList[0].ToString()");
expressionManager.EvaluateExpression(expression);

// Check the value.
Assert.AreEqual<string>("217.168.47.10", result.GetValue<string>(),
    "The expression was not evaluated as expected.");
```

Code 6 Configuring Type Manager Scenario

Erroneous Scenario

The Expression Manager checks the syntax of a given expression. This happens either if an expression is parsed only or if it is evaluated. If no syntax error can be experienced, all is working as expected. If any error happens or actively can be detected, the Expression Manager returns the errors either by an exception or by a result list.

Syntax check

To check the syntax of an expression without evaluating it, use the method CheckSyntax() of the Expression Manager. The code below shows first the syntax check of an expression without an error and afterwards the check of an expression containing an error.

- The result of a syntax check is an object of the type CheckResult. In case of a positive check (no errors) the object contains an empty error list and the level is none

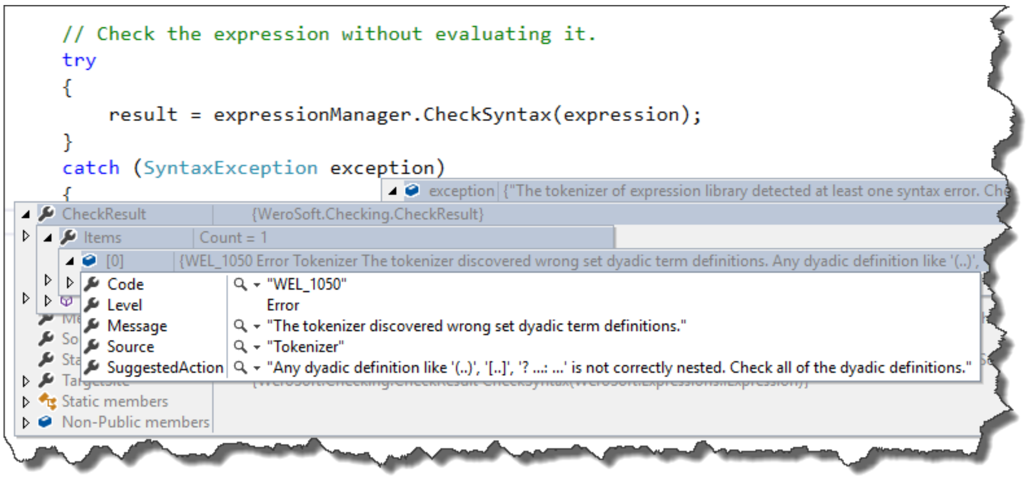
```
try
{
    // Defines a simple expression without an error.
    ExpressionManager expressionManager = new ExpressionManager();
    IExpression expression = new Expression("2 + 3 * 4");

    // Check the expression without evaluating it.
    CheckResult result = expressionManager.CheckSyntax(expression);

    // Defines a simple expression with a syntax error.
    expression = new Expression("2 + 3 * (4");

    // Check the expression without evaluating it.
    expressionManager.CheckSyntax(expression);
}
catch (ExpressionSyntaxException exception)
{
    Assert.AreEqual<int>(1, exception.CheckResult.Count(),
        "The caught exception reflects not the correct amount of errors.");
}
```

Code 7 Erroneous scenario using syntax check

In case of a detected syntax error, the method `CheckSyntax()` raises an `ExpressionSyntaxException`. This special exception type contains a property of the type `CheckResult`.

- The CheckResult contains for each detected error an entry
- Each entry describes the error by:
  - A unique code per error type
  - A level the error is classified in (Warning or Error)
  - The effective error message
  - The source which was responsible for the error detection (Tokenizer, Parser, Expression Manager)
  - Suggested action to correct the detected error

The image below gives the real impression about the caught exception and its description of the error according the sample.



```
// Check the expression without evaluating it.
try
{
    result = expressionManager.CheckSyntax(expression);
}
catch (SyntaxException exception)
{
```

```
exception | {"The tokenizer of expression library detected at least one syntax error. Che
CheckResult        {WeroSoft.Checking.CheckResult}
  Items            Count = 1
    [0]            {WEL_1050 Error Tokenizer The tokenizer discovered wrong set dyadic term definitions. Any dyadic definition like '(..)',
      Code            "WEL_1050"
      Level           Error
      Message         "The tokenizer discovered wrong set dyadic term definitions."
      Source          "Tokenizer"
      SuggestedAction "Any dyadic definition like '(..)', '[..]', '? ...: ...' is not correctly nested. Check all of the dyadic definitions."
Static members
Non-Public members
```

Figure 12 Result embedded in the syntax exception

Evaluation exception

Basically, the evaluation of an expression works the same way as checking the syntax of it. Additionally, to the proactive checks the evaluation also catches the exceptions which may happen during runtime caused on the life system. These two situations are reflected by the two possible exception types which may be raised:

- ExpressionEvaluationException
- ExpressionSyntaxException

The code sample below show some handling of evaluation exception:

```
try
{
    // Defines a simple expression without an error.
    ExpressionManager expressionManager = new ExpressionManager();
    IExpression expression = new Expression("2 + 3 * 4");

    // Evaluates the expression.
    expression = expressionManager.EvaluateExpression(expression);

    // Defines a simple expression with a syntax error.
    expression = new Expression("2 + 3 * (4");

    // Evaluates the expression.
    expression = expressionManager.EvaluateExpression(expression);
```

… continuation on next page

```
}
catch (ExpressionSyntaxException exception)
{
    Assert.AreEqual<int>(1, exception.CheckResult.Count(),
        "The caught exception reflects not the correct amount of errors.");
}
catch (ExpressionEvaluationException exception)
{
    Assert.AreEqual<int>(1, exception.CheckResult.Count(),
        "The caught exception reflects not the correct amount of errors.");
}
```

Code 8 Erroneous scenario using evaluation

About this document

Responsibilities

Responsibility	Name	Organization
Author	Rolf Wenger	weroSoft AG
Reviewer	Patrick Arpagaus	weroSoft AG
Distribution	Rolf Wenger	weroSoft AG

Distribution

Level	Name	Organization
To attention	All interested developers	Any
To note		

References

Nr.	Document	Author	Version / Date / ISBN …
[1]			

Name of the document

D-02-01 Expression Library Programmers Guide.docx

Copyright

History

Version	Date	Author	Description
1	2015-09-29	RoWe	Creation of the documentation
2	2015-11-29	RoWe	Finished first draft of documentation without chapter about Extending the Expression Library.
3	2016-02-28	RoWe	Finalized documentation for first releasing.
4	2017-03-30	RoWe	Corrections of typos. Added usage of Data Factory. Added usage of Graph 2D.