JewelScript Reference Manual

Version 1.2, June 2014

www.jewe.org

Table of Contents – TOC

Language Documentation

This chapter covers a detailed discussion of the JewelScript syntax and language features.

Lexical structure

Identifiers

Identifiers are used to associate a language entity with a user-defined name. Identifiers are used for variable names, function names, class names, and the likes. JewelScript identifiers pretty much are the same as in most other languages. They can consist of any combination of capital and small letters, digits, and the underscore. The first character of the identifier, however, can not be a digit, so it can be distinguished from a numerical literal.

```
identifier := [A-Za-z_][0-9A-Za-z_]*
```

All characters of the identifiers are significant, there is no length limit for identifiers in JewelScript. Furthermore, the language is *case sensitive.* This means that the identifiers 'Foo', 'foo' and 'FOO' are 3 different identifiers.

Keywords

Keywords are words that have a special meaning for the language and therefore cannot be used as identifier names. JewelScript recognizes the following keywords, discussed in detail later in this document:

`__brk`	`__selftest`	`accessor`	`alias`	`and`	`break`
`case`	`class`	`clause`	`cofunction`	`const`	`continue`
`default`	`delegate`	`do`	`else`	`explicit`	`extern`
`false`	`for`	`function`	`goto`	`hybrid`	`if`
`import`	`interface`	`method`	`namespace`	`native`	`new`
`not`	`null`	`option`	`or`	`return`	`sameref`
`strict`	`switch`	`this`	`throw`	`true`	`typeof`
`using`	`weak`	`while`	`yield`		

The keyword `extern` is reserved for future implementation, it is not yet used by the language.

Operators

The following is a list of operators that are supported by JewelScript:

`!`	`!=`	`%`	`%=`	`&`	`&&`	`&=`	`()`
`(cast)`	`*`	`*=`	`+`	`++expr`	`+=`	`-`	`--expr`
`-=`	`-expr`	`.`	`/`	`/=`	`<`	`<<`	`<<=`
`<=`	`=`	`==`	`=>`	`>`	`>=`	`>>`	`>>=`
`?`	`[]`	`^`	`^=`	`and`	`expr++`	`expr--`	`new`
`not`	`or`	`sameref`	`typeof`	`|`	`|=`	`||`	`~`

Other tokens

Other tokens recognized by JewelScript are:

`"`	`"/`	`#`	`'`	`,`	`/"`	`0b`	`0o`
`0x`	`:`	`::`	`;`	`@"`	`{`	`}`	

Literals

Literals are the simplest form of specifying data in an expression. JewelScript recognizes integer numbers, floating point numbers and alphanumerical strings as literals.

Example	*Interpreted as*	
`12345`	const int	Integer number in decimal format
`012345`	const int	Integer number in decimal format
`0x12345`	const int	Integer number in hexadecimal format
`0o12345`	const int	Integer number in octal format
`0b100111000111`	const int	Integer number in binary format
`1234.5`	const float	Floating point number
`1.234e-5`	const float	Float number in scientific notation
`'a', 'ab', 'abc', 'abcd'`	const int	Integer number as a character constant
`true`	const int	Syntactical sugar for the value '1'.
`false`	const int	Syntactical sugar for the value '0'.
`"foo"`	const string	String literal
`/"foo"/`	const string	Verbatim string literal (no processing of ANSI escape characters)
`@"foo"`	const string	Verbatim string literal

Comments

Comments are annotations that developers can embed into the source code in order to explain code in their own, human language. The JewelScript compiler completely ignores comments, they are treated as if they weren't present in the source code.

Like most C-style script languages, JewelScript supports two forms of comments, the *single-line comment* (adopted from C++), and the *multi-line comment.* Here is an illustration of using both forms of comments in source code:

```
(code) // This is a single-line comment.
(code continues)
```

The two forward slashes designate all characters between them and the end of the line as a comment. This is considered more convenient since the developer does not need to explicitly specify an end for the comment, as opposed to the multi-line comment:

```
(code) /* This is a multi-line comment.
It can span as many code lines as we want. */ (code continues)
```

The multi-line comment is often used to "comment out" parts of code.

Since version 0.8, the 'hash' (#) symbol is supported as an alternative single-line comment. This has been added to allow a better (command-line) integration in Linux/Unix/BSD systems.

Built-in data types

JewelScript has a number of built-in data types that allow basic operation, even if the runtime is used "as-is", i.e. without any additional native bindings.

While the primitive data types *int* and *float* are not treated as classes by the language, the data types *string*, *array*, *list, iterator* and *table* are classes and in fact regular native type libraries, registered to the runtime upon initialization.

It is not necessary to import the built-in types, since all of them except the *runtime* class are automatically imported when the compiler is initialized.

null

The null value is used by the language to indicate whether a variable's value or reference is undefined. The language allows to compare values of any type against the value null, compare the type of any value against the type of null, and to assign the value null to reference variables of any type. Apart from that no other operation is valid on the value null.

```
Foo foo = null;   // class Foo reference variable foo is undefined
```

int

An int is a signed 32-bit integer data type. We can only assign non-fractional numbers in the range from -2147483648 to +2147483647 to a variable of type int.

```
int a = 4711;          // an integer literal in decimal format
int b = 0x1267;        // an integer literal in hexadecimal format
int c = 0o11147;       // an integer literal in octal format
int d = 'abcd';        // a character constant with up to 4 digits
```

The language allows implicit conversion from an int to a float.

float

A float is a 64-bit floating point data type, unless the runtime was compiled for 32-bit float support. The language allows implicit conversion from a float value to an int value, even though we loose precision and the value might get truncated from 64 to 32 bits. While this is convenient in real life programming, it is also critical and prone to programming errors. Therefore the compiler will issue a compile-time warning when a float gets implicitly converted to an int.

```
float a = 0.001;       // 1/1000 as a floating point number
float b = 1e-3;        // 1/1000 in scientifc notation
```

string

A string is a sequence of alphanumerical characters. The JewelScript string type is a class that offers a set of methods in order to manipulate a string's characters. A string in JewelScript therefore is an *instance* of this class, and thus a dynamically allocated *object*. There is no explicit limit for the maximum length of a string in JewelScript. Internally, the length of a string is represented as a signed 32-bit value, so it is implicitly limited to a length of 2 GB of memory. In JewelScript, characters are unsigned 8 bit values.

When specifying a string literal enclosed in regular double quotation marks ("), the JewelScript compiler will recognize all *escape sequences* defined by the ANSI C standard. For example, the literal "\n" will result in a *newline character* in the string, not the characters "\" and "n".

While processing ANSI escape sequences is often the only way to enter special characters or byte values in a string literal, it is also often "in the way", making it hard to enter path names or regular expressions.

To remedy this, JewelScript also allows us to specify "verbatim string literals". Within these string literals, no ANSI escape sequences are processed, so path names and regular expressions can be entered verbatim into the literal.

```
// match expression (hello) or ("hello")
string reg1 = "\\([a-z\"]*\\)";  // normal literal
string reg2 = /"\([a-z"]*\)"/;   // verbatim literal
string reg3 = @"\([a-z\x22]*\)"; // C# style verbatim literal
```

In JewelScript's verbatim string literal, the quotation mark can be entered verbatim. In the C# style literal, entering a quotation mark is currently not possible. The \x22 will be processed by the regular expression, not the string literal.

Similar to C and C++, the JewelScript compiler will automatically concatenate two or more string literals if there is only white-space between them. This is useful if we want to break up a long string literal and want it to span multiple lines of code:

```
string myStr = "This "
    "is "
    "one string.";
```

Will be the same as:

```
string myStr = "This is one string.";
```

Of course this is also possible using the verbatim string literal. Furthermore, it is possible to concatenate literals of both types to each other this way.

To append a string to another string at runtime, operators + or += can be used.

The string class has constructors to construct a string from an int or float value, allowing convenient auto-conversion of a number to a string.

```
string s = 3.1;       // convert 3.1 to string
```

The string class also has convertor methods to convert a string into a numerical value. However, these are declared using the *explicit* keyword and thus require us to use a typecast operator to explicitly confirm that a conversion is wanted.

```
float z = (float) s;  // convert string to float
```

array

An array is a sequence of values of any type, addressed by a common identifier name and an integer index number that specifies the position of an element in the array. Array index numbers in JewelScript range from 0 up to, but not including, the total number of elements in the array. An array in JewelScript is a dynamically allocated object.

The JewelScript array has the ability to dynamically grow; based on the index used to access elements from it. There is no explicit limit for the maximum length of an array in JewelScript. Internally, the array index is represented as a signed 32-bit integer, so implicitly an array is limited to a length of about 2.1 billion elements.

To append elements to an array at runtime, operators + or += can be used. If the right-hand operand is an array too, all elements from the right-hand array will be added to the left-hand array.

It is possible to construct an array during runtime and on-the-fly by assigning a comma separated list of expressions enclosed in curly braces to the variable:

```
array a = { x, y, z };
print( a[2] );    // prints z
```

An array can be conveniently converted to a string by using the typecast operator. The string convertor method of the array class will write all elements in ascending index order into the string. This is a recursive operation, meaning sub-arrays in the array will be written into the string as well.

```
string name = "Ned Baker";
float a = 3.1;
string s = (string) {"Hello ",name,"! 'a' is currently ",a,"\n"};
```

Type-less arrays

JewelScript supports type-less arrays, meaning array elements do not have any type. We can store values of any type in the array.

The disadvantage of type-less arrays is, that the language does not know the type of values in the array at compile-time, and thus cannot assist the developer in ensuring type safety. Therefore, when using type-less arrays, programmers must ensure type safety of the program on their own.

There are more disadvantages when using type-less arrays. See also type *→var.*

Type-safe arrays

Version 0.9 of the language introduced the possibility to declare mono-type arrays. The advantage of these arrays is that they remedy all of the problems that type-less arrays suffer from – like for example ensuring type-safety, allowing automatic type conversion, access to members of objects in an array, and so forth.

The downside to these arrays is that they can only store values of the same type, of course.

A mono-type array is declared by prefixing the element's type name to an array declaration statement:

```
int array myArray = {10, 11, 12};
string array myStrings = {"Hello", "World", "Foo"};
CFoo array myFooArray = new array();
```

An empty pair of square brackets can be used instead of the array keyword to declare an array:

```
int[] myIntArray;
string[] myStringArray;
CFoo[] myFooArray;
```

Multi-dimensional arrays

In order to create arrays with more than one dimension, we can place an array into an array, and so forth, up to the desired number of dimensions. Here is a simple example of how to create a two dimensional array of 3 by 3 elements:

```
int[] matrix = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };
```

Accessing this 2-dimensional array would then look like this:

```
int val = matrix[2,1];     // val will be set to 8
matrix[2,1] = 10;          // set element to 10
```

To create arrays with a larger number of elements and more dimensions, it is recommended to use operator *new* to construct the multi-dimensional array. This example will create a 3 dimensional array and fill it with 1000 integer values:

```
int[] a = new array(10, 10, 10);
```

When instantiating an array using operator new, the array is automatically filled with values, if the array's element type is a value type (int or float). For reference types, the array is filled with null-references.

list

A list is a container class that allows adding and removing data very quickly.

When a program needs to deal with data more dynamically, specifically if it must be able to quickly add and remove data to a collection, using the built-in list class is preferable over trying to achieve that with an array.

The JewelScript list is a *double chained list* implementation. List items are objects itself, even though we cannot access them directly. They are private to the list implementation. A list item is defined as a *key / value* pair, where the key represents an associative criteria under which the developer wants to put a certain value into the list, or retrieve a certain value from the list.

The list offers a set of methods to add, remove, sort and find items in the list by their key. The list restricts the type of an item's key to int, float and string, because it must be able to use the key to look up items. An item's value can be of any type, since the list does not look at the value at all; it just stores and retrieves it for the developer.

```
list l = new list();       // create a new list
l.add( "foo", x );         // add a value under the key "foo"
l.add( "bar", y );         // add another value under "bar"
print( l.value("bar") );   // prints y
```

iterator

The built-in iterator type is strongly related to the list. Iterators are used to crawl over lists in order to read or modify items in that list. If a program needs to continuously process a large collection of list items, using an iterator is much more efficient than looking up every item by their key.

The JewelScript iterator offers methods to read the current item's key and value, methods to navigate in the list, as well as inserting and deleting list items.

In order to associate an iterator to a list, we construct the iterator and pass a reference to the list to the constructor. This happens automatically if we *initialize* an iterator variable with a list:

```
list l = new list();       // create a new list
l.add( "foo", x );         // add a value under the key "foo"
l.add( "bar", y );         // add another value under "bar"
iterator i = l;            // create an iterator for this list
print( i.value );          // prints x
i.next();                  // goto next item
print( i.value );          // prints y
```

table

This is a hash-table class that can store and retrieve any type of value by a given string argument, the *hash key*. Like lists, hash-tables are containers of data, and allow us easy access to every element stored within them.

However, when looking up a value in the container, hash-tables are much faster than lists. This is because lists have to search through all elements stored in them in order to find an element, while hash-tables use a different and more direct mechanism to access the element.

Because of this, developers often use tables if they need to store and retrieve values by their “name”, especially if there can be many thousand values stored in the container.

There are some downsides to tables, however. First of all, due to the way they work, hash-tables use way more memory to store an element than lists. Second, there is no real way to iterate over a table[1], hence it is only possible to retrieve an element, if its hash key is known.

Because of these differences, one should not generally prefer tables over lists, or vice-versa. Both container types have their strong points and disadvantages, so the key is to find out which one is more efficient and suitable for the actual problem at hand.

var

In JewelScript, the keyword *var* can be used to declare a variable as being *type-less*. That means there is no specific type information associated with the variable, it can refer to data of any type. Even though JewelScript is not a dynamically typed language, the virtual machine runtime is dynamically typed, and the var keyword lets us utilize this fact to a certain extent.

However, some caution should be taken when using type-less variables. First of all, using them has the disadvantage that the compiler cannot assist developers in detecting type mismatches and other programming errors during compile-time. That means, if we *assume* that our type-less variable currently contains a number and want to multiply it, but in fact it contains an instance of class Foo, then this will only be detected during runtime, producing a runtime error that in many cases prevents the code from being executed to the end.

1) In JewelScript 1.0, the table::enumerate() method allows to “iterate” over all items in a table, by calling a delegate for each item stored in the table.

Another disadvantage is, due to the fact that the compiler does not know the type of the data, we cannot easily access members from objects through such a variable. In this case we need to explicitly tell the compiler what type we *think* is currently referenced by the variable. And again, if our assumption was wrong, a runtime error will be likely to occur.

```
var foo = new Foo();       // allocate a Foo instance
foo = 2 * foo;             // Error: Foo cannot be multiplied, but compiler
                           // cannot detect this!
foo.Run();                 // Error: Compiler does not know if 'foo' has a
                           // member Run(), it doesn't even know if 'foo'
                           // is an instance of a class right now!
foo.Foo::Run();            // Alright, this will work, provided 'foo' really
                           // contains an instance of class Foo.
```

Using type-less variables will cause the compiler to not being able to perform automatic type conversion. Because in order to convert a value to a specific type, the compiler would need to know the current type of the value. Consequently, type-less values will always be passed around "as-is" by JewelScript, they never get automatically converted.

Moreover, since the compiler cannot know whether it is a value type or reference type, it will always pass type-less values around as references.

Expressions

JewelScript supports most of the expressions known from C#, C++, Java and other C-style script languages. This section will discuss expressions in more detail.

Operators

Assignment

Assignment operators are used to set the variable left from the operator to a new value.

```
a = 10;       // a simple assignment
a += 10;      // simplified a = a + 10
a++;          // simplified a = a + 1;
```

The result of an assignment operator is always the left hand operand *after* performing the operation. This allows us to assign the result of an assignment to a variable, assign that result to another variable and so on:

```
a = b = c = 0;    // set a, b and c to 0
```

Further available assignment operators are:

```
-=   *=   /=   %=
&=   |=   ^=   <<= >>=
```

++lval, --lval operators

These operators are called pre-increment and pre-decrement operators, because they will actually increment or decrement the left-hand value *lval* **before** it is returned.

```
int a = 1;
int b = ++a; // a incremented, then b set to 2
```

lval++, lval-- operators

These operators are called post-increment and post-decrement operators, because they will actually increment or decrement the left-hand value *lval* **after** it is returned.

```
int a = 1;
int b = a++; // b set to 1, then a incremented
```

Arithmetic

Of course JewelScript can also evaluate arithmetic expressions. It supports the arithmetic operators + (addition), - (subtraction), * (multiplication), / (division) and % (modulo).

These operators can be used for the numerical data types int and float. Operator + can also be used with the string and array types.

The result of an arithmetic operator is always a new value. The result is always non-constant regardless whether the operands are constants or not.

If an expression uses mixed int and float values, the int values are converted to float where necessary and the result is converted back to int if necessary.

```
int val = 10 * 3.5 + 44;
```

Note that if the second operand to a division or modulo operator is zero, the virtual machine will generate a "DIVISION BY ZERO" exception, which will, if not handled by a C/C++ callback function, abort byte-code execution and return an error code to the application.

Relational

Relational operators allow us to compare values to each other. JewelScript supports the relational operators == (is equal), != (is not equal), < (is less than), <= (is less than or equal), > (is greater than), >= (is greater than or equal).

These operators can be used for the numerical data types int and float.

The operators == and != can also be used to compare strings, and to compare values against the null value. To compare two values by reference, operator sameref() can be used.

The result of a relational operator is always an int value. It is 1 if the expression resolves to *true* and 0 (zero) if it resolves to *false*. The runtime treats the int value 0 (zero) as *false*, and any other int value as *true*.

Logical

Logical operators allow us to express a boolean relationship between two boolean values. Logical operators supported by JewelScript are && (and), || (or) and ! (not). JewelScript offers the keywords *and, or, not* as syntactical sugar for these operators.

operand1 && operand2

The && operator will return true if *operand1* resolves to true, and *operand2* also resolves to true. It will return false in any other case. The && operator will only evaluate the second operand, if the first operand resolves to true.

operand1 || operand2

The || operator will return true if *operand1* resolves to true, or if *operand2* resolves to true, or if both resolve to true. It will return false if none of them resolve to true. The || operator will only evaluate the second operand, if the first operand resolves to false.

! operand

The ! operator will return true if *operand* resolves to false. It will return false if *operand* resolves to true. Thus, it returns the (boolean) complement of the operand's value.

Bitwise

JewelScript allows operations on bit level. The bitwise operators available are & (bitwise and), | (bitwise or), ^ (bitwise xor), ~ (bitwise not), << (unsigned shift left), >> (unsigned shift right).

These operators can only be used with the int data type. The result of these operators will always be a new int value. The result of these operators is always non-constant, regardless whether their operands are constants or not.

The left-shift and right-shift operators will treat the first operand's value as an unsigned integer, making it suitable for logical bit-manipulation, but not for arithmetic calculations.

Operator [ ]

The square brackets allow to access an element of an array by it's array index. The result of this operator is a reference to the value addressed by the index.

```
indices := expr [',' indices]
brackets := '[' indices ']' [brackets]
expr := identifier brackets
```

In the case of a type-less array, the resulting type of this operator is *var*.

In the case of a mono-type array, the resulting type of this operator is the element's type.

When creating multi-dimensional arrays, the runtime will actually nest array objects (put an array into an array, and so forth). So when using this operator with a multi-dimensional array, the resulting reference can actually be an array, if specifying less indices than dimensions in the array:

```
int[] a = new array(10, 10); // create 2-dimensional int array
int[] b = a[0];              // return sub-array
```

To access elements from arrays with more than one dimension, it is possible to concatenate multiple operator [ ], up to the number of dimensions:

```
int[] a = new array(10, 10, 10);
int i = a[0][0][0];
```

Another, more convenient possibility is to specify a comma separated list of indices to operator [ ]:

```
int i = a[0, 0, 0];
```

If one is so inclined, one can also mix these two styles:

```
int i = a[0][0, 0];
int j = a[0, 0][0];
```

Operator ? (ternary operator)

```
expr := expr1 '?' expr2 ':' expr3
```

As the name suggests, the ternary operator consists of three expressions.

The first expression defines a boolean condition.

If expr1 resolves to true, the ternary operator returns expr2.

If expr1 resolves to false, the operator returns expr3.

```
max = str.length < max ? str.length : max;
```

Operator => (lambda operator)

```
args := identifier [',' args]
expr1 := '(' [args] ')' '=>' expr
expr2 := '(' [args] ')' '=>' '{' [statements] '}'
```

The lambda operator defines a function literal. It is syntactical sugar for the anonymous function or anonymous method literal, respectively.

```
f = (x, y) => x * x + y;
```

This is equivalent to:

```
f = function(x, y) { return x * x + y; };
```

Or, if the expression is used within an instance member function (method):

```
f = method(x, y) { return x * x + y; }
```

Operator sameref()

This operator allows us to test whether two variables refer to the same value. The result of this operator is returned as a new int value (true or false).

```
expr := 'sameref' '(' expr1 ',' expr2 ')'
```

If expr1 refers to the same value as expr2, the result is *true*.

If expr1 and expr2 refer to different values, the result is *false*.

If one of the expressions is a weak reference, taken from the other expression, the result will be *false.*

```
string a = "Hello";
string b = "Hello";
print( sameref(a, b) ); // will be false
string c = a;
print( sameref(c, a) ); // will be true
print( sameref(c, b) ); // will be false
weak string d = a;
print( sameref(d, a) ); // will be false
print( sameref(d, b) ); // will be false
```

Operator typeof()

The typeof operator returns the type identifier number of a given literal, variable, expression or type name. The type identifier number is an int value that is unique to every type in JewelScript.

```
expr := 'typeof' '(' expr ')'
```

The language will first attempt to resolve the type of the given operand during compile-time. If this is not possible, because the operand depends on runtime data, it will place a VM instruction to determine the type identifier number at runtime.

The typeof operator is especially useful when using type-less variables. We can check the actual type of a type-less variable during runtime using this operator.

```
if( typeof(myVar) == typeof(string) ) { ... }
```

Operator new

The new-operator returns a new instance of the specified data type.

```
expressions := expr [',' expressions]
long_typename := typename ['::' long_typename]
expr := 'new' long_typename '(' [expressions] ')' [block-statement]
```

If *typename* is the name of a class, a new instance of that class is created and the constructor matching the given arguments is called.

As of JewelScript 1.0.3.13, an *initializer block-statement* can be appended to the expression. The code is compiled as if it were part of the class of the newly instantiated object and allows convenient access to it's member variables and methods.

```
CFoo foo1 = new CFoo();          // default constructor
CFoo foo2 = new CFoo(27);        // construct from int value
Cfoo foo3 = new CFoo(){ value = 27; }; // construct and set member variable
```

If *typename* is the name of an interface, this will *factorize* the interface. Factorization means creating an instance of all known implementers of the given interface and returning them in an array. The interface must define a constructor for this to work.

```
IPerson[] factory = new IPerson("Sally", 29);
```

If *typename* is the name of the built-in types *int* or *float,* the language supports using operator new to copy-construct a value of that type:

```
int l = new int(17);
float f = new float(27.5)
```

Of course the built-in classes *string, array, list, iterator* and *table* define a copy-constructor as well.

If *typename* is the name of the built-in types *int, float, string* or *array*, the language supports using operator new to default-construct a value of that type.

```
int l = new int();         // create new int value ( 0 )
float f = new float();     // create new float value ( 0.0 )
string s = new string();   // create new, empty string ( "" )
array a = new array();     // create new, empty array ( {} )
```

Finally, the new-operator allows to allocate an array of a given size:

```
array b = new array(x);          // pre-allocate 'x' elements
```

Specifying a size for an array is not necessary, since an array can dynamically grow. However, in order to have the runtime automatically create multi-dimensional arrays, it is necessary to specify the sizes of each dimension.

```
array c = new array(10, 10, 10); // 3-dimensional array of 1000 elements
```

When using operator *new* to create a multi-dimensional array for a mono-type array, the operator will automatically fill the array with instances, if the type is int or float. For reference types, the array will be filled with references to the *null* value.

Typecast operator

The typecast operator allows developers to explicitly convert a value of one type into a value of another compatible or convertible type.

```
long_typename := typename ['::' long_typename]
expr := '(' long_typename ')' expr
```

JewelScript allows to declare *type-less variables*, which have an effect on automatic type conversion: If a variable is *type-less*, the compiler cannot do auto-conversion. Using the typecast operator, we can specify explicitly, which way of type conversion is wanted. (See also: →*var*)

The typecast operator also allows us to convert a reference from an interface type down to it's actual class type. To ensure type-safety, this will generate an instruction that ensures at runtime, that the conversion is really possible.

The typecast operator is also required, when a constructor or convertor method of a class has been declared *explicit*. The explicit modifier tells the compiler that it *must not* call constructors or convertor methods automatically for type conversion. Instead, an explicit typecast is required to "acknowledge" the conversion. Classes use the explicit modifier typically for conversions that are non-trivial and can potentially fail, to have the developer confirm that they know what they are doing and that the conversion is really what they want.

The typecast operator does not directly allow the developer to cast away the constancy of a value. However, the result of a typecast operator with a constant operand *can be* non-constant, if the typecast results in the construction of a new value due to the conversion.

The operator also does not allow to reinterpret a value's type to another type. Trying to convert a type into another type that is not compatible, will result in a compile-time error.

Operator precedence

When using expressions that combine multiple operators, the order of which operator precedes which is important. Normally, the language will evaluate an expression from the left to the right, just like we (in the western world) usually read it. But there are exceptions.

The following chart shows the various operators and their precedence. The closer to the top of the chart an operator is, the higher is it's precedence. It will precede all operators on lower lines in the table, but not those that are on the same line or higher in the table.

Type	*Operator*	*Execution*
Primary operators	`( ) [ ] . ++ -- sameref`	left-to-right
Unary operators	`-expr ! ~ not new (cast) typeof =>`	right-to-left

Binary operators	`* / %`
	`+ -`
	`>> <<`
	`< > <= >=`
	`== !=`
	`&`
	`^`
	`|`
	`&& and`
	`|| or`
		left-to-right

Ternary operator	`?`	left-to-right
Assignment operators	`= += -= *= /= %= >>= <<= &= ^= |=`	right-to-left

Array constructor

The array constructor expression lets us create and fill an array on-the-fly during runtime. This is a powerful feature that also makes it simple to dynamically pass a variable number of arguments to a function, or return a variable number of result values.

```
expressions := expr [',' expressions]
expr := '{' [expressions] '}'
```

Here are a few code examples using the array constructor:

```
array a = {};     // create an empty array

// create an array containing 3 int values:
array b = { 10, 20, 30 };
print( b[2] );        // prints 30
```

```
// construct an array on-the-fly and pass to function:
print( { "a is ", a, ", b is ", b, " and c is ", c, "\n" } );

// return multiple result values:
return { a, b, c };
```

Function call

The function call expression executes the code of a global function, global class member function, method, or delegate. If the function to call expects any arguments, the function call expression must specify these in the form of a comma separated list of expressions.

```
arguments := expr [',' arguments]
long_identifier := identifier ['::' long_identifier]
expr := long_identifier '(' [arguments] ')'
```

When calling a global class member function from outside of the class's scope, the class name of the class, followed by the scope-operator needs to be specified. Unless the function has been mapped to the global scope by the using-statement.

The result of a function call depends on the function's declaration. If no result type has been declared at all, the function call expression does not return a value, and thus cannot be used as a right-hand value in an assignment, as an operand to an operator, or as an argument to a function call.

```
function foo(int);

int a = foo(10);      // not possible since foo() has no result
```

For more information on function calls, see the →*Functions* section of this document.

Anonymous function / method literal

Defines an anonymous function or method. This requires that the value left of the assignment is a delegate type.

```
args := identifier [',' args]
expr := 'function|method' ['(' [args] ')'] '{' [statements] '}'
```

The following example selectively prints elements from a string array, if a condition is met. The condition is given as a function literal.

```
delegate int Condition(string);

method Select(Condition c) {
    for(int i = 0; i < Strings.length; i++)
        if( c(Strings[i]) )
            println(Strings[i]);
}

Select( function(str) { return str == "hello"; } );

Select( (str) => str == "hello" ); // lambda equivalent
```

Co-function instantiation

Co-functions are JewelScript's implementation of the concept of *co-routines*. In a nutshell, they are functions that run in their own thread context and can be resumed and suspended by script code.

A co-function is instantiated by operator new, which allocates a new thread object, initializes it with the given arguments and sets the thread's program counter to the beginning of the specified co-function.

```
arguments := expr [',' arguments]
expr := 'new' [ [typename] '::' ] identifier '(' [arguments] ')'
```

The instantiation of a co-function does *not* execute the co-function directly. Instead, it returns a reference to a thread object that can be stored in a variable. The thread variable can then be used to resume the thread.

```
Generator MyThread = new Generator(0, 10);
```

For more details about co-functions, please see the →*Co-functions* section of this document.

Co-function resume

The purpose of this expression is to resume the thread of a co-function.

```
expr := 'identifier' '(' ')'
```

The compiler expects 'identifier' to be a variable that holds a reference to a co-function thread object. No arguments can be specified between the parentheses, since the thread is already running and cannot receive any additional arguments on it's stack.

```
int n = MyThread();
```

For more details about co-functions, please see the →*Co-functions* section of this document.

Statements

Statements are the actual *building blocks* of program code. Statements are used to control the program flow, define loops, declare variables, classes and functions, and so on. This section will discuss the statements available in JewelScript in detail.

In JewelScript, a statement must end with a semicolon:

```
statements := statement ';' [statements]
```

Exceptions to this rule are the *block-statement* and the *switch-statement.*

Block

The block-statement can contain multiple statements itself. It can be used anywhere where a single statement is expected, allowing us to specify more than a single statement. Since a block is a statement and can contain statements, blocks can be nested.

A block-statement begins with an open curly brace, followed by any amount of statements, and closed by a closing curly brace:

```
block-statement := '{' [statements] '}'
```

It is important to know that variables declared inside a block-statement will only 'live' until the end of the block-statement is reached. After the execution has left a block-statement's closing brace, all local variables declared in that block are automatically freed.

```
{
     int a = 27;
     string b = "hello";
     {
          int x = 0;
          float y = 27.5;
     }    // 'x' and 'y' automatically freed
}    // 'a' and 'b' automatically freed
```

Note that unlike other languages, JewelScript does not allow a variable name to be redeclared in a nested block, if it already has been declared in a parent block:

```
{
     int a = 0;
     {
          int a = 0;
     }
}
```

Redeclaring a variable name in a new block after the previous block is out of scope is of course OK:

```
{
     {
          int a = 0;
     }
     {
          int a = 0;
     }
}
```

Control Flow

Control flow statements are used to, like the name already suggests, control the flow of the program. This means that they control which parts of the code will be executed under which condition.

if / else / else-if

Tests a given expression and conditionally executes another statement.

```
statement := 'if' '(' expr ')' statement1 ['else' statement2]
```

The if-statement first evaluates the expression *expr*, which must result in an int value. If the result is true, *statement1* will be executed. After this, execution will continue beyond the if-statement.

If the result was false, and no *else* branch has been specified, *statement1* will be skipped and execution will continue beyond the if-statement.

If the result was false and an *else* branch has been specified, *statement2* will be executed. After that, execution will continue beyond the if-statement.

```
if( a == 1 )              // if a is equal to 1,
    text = "a is 1";      //    then this gets executed.
else                      // otherwise,
    text = "a is not 1"; //    this gets executed.
```

An if-statement can immediately follow an else keyword from a previous if-statement, forming an *else-if-statement*:

```
if( a == 1 )              // if a is equal to 1,
    text = "a is 1";      //    then this gets executed.
else if( a == 2 )         // otherwise, if a is equal to 2,
    text = "a is 2";      //    then this gets executed.
else                      // otherwise,
    text = "a is neither 1"
         " nor is it 2";  //    this gets executed.
```

switch

The switch-statement allows to delegate execution to any amount of code fragments depending on the value of a single expression.

```
case_tag := 'case' const_expr ':' [statement]
case_tags := case_tag [case_tags]
statement := 'switch' '(' expr ')'
'{'
    case_tags
    ['default' ':' [statement]]
'}'
```

The switch-statement will first evaluate the expression *expr*, which must result in an int or string value. After that, the statement will compare the result to each of the constant expressions *const_expr* specified to the case tags.

If a matching case tag is found, the statement associated with that case tag is executed. If no matching case tag is found, the statement associated with the default tag is executed. If no default tag is present, execution is continued beyond the switch-statement. The default tag must be the last tag in the switch-statement, if specified.

```
switch( a )
{
    case 1:                     // if a is equal to 1
        text = "a is 1";        // this will be executed,
        break;                  // and this will leave the switch.
    case 2:                     // if a is equal to 2
        text = "a is 2";        // this will be executed,
        break;                  // and this will leave the switch.
    default:                    // otherwise,
        text = "a is neither 1 nor 2"; // we get here.
        break;                  // break not necessary in default
}
```

'fall through' and break

If a case tag's statement is executed, execution will *not* automatically stop at a following case tag. Instead, execution will 'fall through' into the next case tag. To prevent this, a case tag must be concluded with a *break*-statement, which will cause execution to continue beyond the switch-statement.

```
switch( a )
{
    case 1:
        text = "a is 1";
    case 2:
        text = "a is 2";
        break;
}
```

As we can see in above example, no break has been specified for the case tag labelled with '1'. This will cause code execution to fall through into the following case tag if 'a' is equal to 1. Thus, 'text' will contain the string "a is 2" if 'a' is 1 or 2. Of course, this is a bug in almost all cases. But there are also situations where we can utilize fall through. Namely in situations where we want to handle multiple cases with a single statement or block:

```
switch( a )
{
    case 1:
    case 2:
        text = "a is 1 or 2";
        break;
    case 3:
    case 5:
    case 8:
        text = "a is 3, 5 or 8"
        break;
    default:
        text = "a is something else";
}
```

The compiler will automatically put all statements between a case tag and it's concluding break statement into it's own stack context. This means that it is possible to declare the same variable names in each case block, without having to explicitly use a block statement.

```
switch( a )
{
    case 1:
        int b = 2;
        print( b );
        break;
    case 2:
        int b = 4;
        print( b );
        break;
}
```

clause / goto

The clause-statement is a special kind of block statement that can be tagged with a label and a result variable. The goto-statement can be used to jump out of a clause to another clause.

```
vardecl := typename identifier ['=' expr]
statement1 := 'clause' '(' vardecl ')' block-statement
statement2 := 'clause' identifier block-statement [statement2]
statement := statement1 [statement2]
```

A clause-statement can consist of one or more consecutive clause blocks. The first clause block must define a result variable. Any type can be used, including interface, class or delegate references.

All following clause blocks can only specify a branch label. The label must be unique for all clause blocks in the same scope.

```
clause (int err)
{
    clause (string str)
    {
        /* do something */
        if (/* condition */)
            goto Exit("An exit argument");
        if (/* condition */)
            goto Error(-10);
    }
    clause Exit
    {
        println(str);
    }
}
clause Error
{
    println("Error #" + err);
}
```

Execution will enter the first clause block, but will not fall through into following clause blocks. The clauses *Exit* and *Error* in the example above will never be entered, unless a goto explicitly jumps into them.

The goto-statement passes a result value to the clause-statement's variable. This can be used to carry an error or result value over from one clause block to another. The expression specified to the goto-statement must match the type of the clause block's result variable.

Clause-statements can be nested. It is possible to jump from an inner clause block to an outer clause block of any level.

When nesting clause blocks, labels used for an inner statement will hide those of an outer statement. This means we may reuse the label *Exit* for an inner block, even if the label has already been used for an outer block. In that case, the outer clause block will become unreachable.

break

The break-statement can be used to jump out of a while-, do-while-, for-, or switch-statement.

```
statement := 'break'
```

When using break in nested loops or switch-statements, the break-statement always directly corresponds to the scope of the statement where it is used, for example:

```
for( int i = 0; i < 10; i++ )
{
    for( int j = 0; ; j++ )
    {
        if( j == 5 )
            break;
    }
}
```

The break-statement belongs to the scope of the inner for-statement and will jump out of it if 'j' is equal to 5. It does *not* belong to the scope of the outer for-statement, thus the outer loop will not be interrupted by the break-statement.

continue

The continue-statement can be used to jump to the start, respectively end of a for-, while- or do-while statement.

```
statement := 'continue'
```

In a for-statement, continue will jump to the increment statement for the iterator variable.

In a while-statement, continue will jump to the top of the loop, to the loop's expression.

In a do-while-statement, continue will jump to the end of the loop, to the loop's expression.

return

The return-statement ends execution of a function and causes execution to return to the caller of the function. If a result has been declared for the function, an expression must be specified to the return-statement. If no result has been declared for the function, the function cannot return a result and the return-statement cannot specify an expression.

```
statement := 'return' [expr]
```

The return-statement will cause the function to go out of scope, no matter where in the function it was encountered. This means that all variables defined locally in the function will be automatically freed.

yield

The yield-statement suspends execution of a co-function and passes a return value to the parent thread. If a result has been declared for the co-function, an expression must be specified to the yield-statement. If no result has been declared for the co-function, the co-function cannot return a result and the yield-statement cannot specify an expression.

```
statement := 'yield' [expr]
```

throw

The throw statement allows us to throw an instance of an exception class from anywhere in the function's body, if a critical situation is detected and we need to end script execution immediately.

```
statement := 'throw' expr
```

The throw statement expects 'expr' to be a type that implements the built-in *exception* interface. At present, exceptions can only be handled in native C/C++ code.

For more details about exceptions, please see the →*Exceptions* section of this document.

Loops

Loops are statements that allow a portion of code to be repeatedly executed. Most loops use an expression to define a condition under which a loop should continue, or when the loop should be exited. However, we can also define unconditional loops, which will run infinitely.

while

The while-statement defines a simple loop that performs a conditional check before entering the loop. Thus, the loop's body may never be executed.

```
statement := 'while' '(' expr ')' statement
```

The while-statement will first evaluate the given expression *expr*, which must result in an int value.

If the result is true, *statement* will be executed. After this, execution will return to the given expression *expr* and will evaluate it again. This will repeat for as long as *expr* will result to true.

If *expr* results to false, *statement* will be skipped and execution will continue beyond the while-statement.

A while-statement can run infinitely if the given expression always results to true:

```
while( true )
    print("Hello!");  // infinitely print "Hello!"
```

To repeat more than a single statement, use a block-statement:

```
int a = 0;
while( a < 10 )
{
    print("Hello!");  // print "Hello!"
    a++;
}
```

To jump out of a while-loop, we can use the break-statement:

```
int a = 0;
while( true )
{
    print("Hello!");  // print "Hello!"
    a++;
    if( a == 10 )     // only do this 10 times
        break;
}
```

do-while

The do-while-statement only differs from the while-statement in that the conditional check is performed at the end of the loop. Thus, the loop's body will be executed at least once.

```
statement := 'do' statement 'while' '(' expr ')'
```

The do-while-statement will first execute *statement*. After this, it will evaluate *expr*, which must result in an int value.

If the result is true, execution will return to the start of the loop and *statement* will be executed again.

If *expr* results to false, execution will continue beyond the do-while-statement.

for

The for-statement repeats a statement for as long as a given expression results to true.

```
statement := 'for' '(' [statement1] ';' [expr1] ';' [expr2] ')' statement2
```

First, the for-statement will execute *statement1*. This is also called the *initializer statement* because it is used to initialize an iterator variable. If this statement is used to declare an iterator variable, this variable will be created within the scope of the for-statement. As a result, when the for-statement is left, the iterator variable is freed and its name can be reused.

Next, the statement will evaluate *expr1*, which must result in an int value. If the result is true, *statement2* will be executed, which is the loop's body.

After this, the for-statement will evaluate *expr2*, but the result of this expression will simply be ignored. This is because e*xpr2* is intended to increment an iterator variable at the end of an iteration. Finally, execution will return to the start of the loop by evaluating *expr1* again.

If the result of *expr1* is false, the loop is left and execution continues beyond the for-statement.

Usually, the for-statement is used like this:

Print 10 times "Hello!":

```
for( int i = 0; i < 10; i++ )
    print( "Hello!" );
```

Count from 1 to 100 in steps of 10:

```
for( int i = 1; i <= 100; i += 10 )
    print( i );
```

Count from 10 to 0 backwards:

```
for( int i = 10; i >= 0; i-- )
    print( i );
```

Note that *statement1*, *expr1* and *expr2* can be omitted, which has the following consequences:

If *statement1* is not specified, no code will be executed within the for-statement's scope that will initialize the loop's iterator variable. This can be useful if we want the iterator variable to exist outside of the for-statement's scope.

```
int i = 0;
for( ; i < 10; i++ )
{
    print( "Hello! );
    if( i == 5 )
        break;
}
print( i );       // 'i' is still valid
```

If *expr1* is not specified, the loop will simply run infinitely, since no exit-condition has been defined. This can be useful if we need to check for special exit conditions inside the loop's body, or if we want to create an infinite loop.

```
for( int i = 0; ; i++ )
    print( i );       // count infinitely
```

If *expr2* is not specified, no code is executed before the loop returns to the beginning. This can be useful if we don't want to increase the iterator variable in all cases, but only in certain cases that are handled in the loop's body.

If neither *statement1*, *expr1* nor *expr2* are specified, this creates the simplest form of an infinite loop:

```
for( ; ; )
    print( "Hello!" );    // print infinitely
```

Note that this loop is even simpler than a *while(true)* loop, because that would actually cause the constant '1' to be tested for every iteration of the loop. *for(; ;)* will only result in a branch back to the start of the loop's body.

import

The import-statement allows us to *modularize* a program. Meaning, we can put every class into a separate file and import them where needed, making it possible to share and reuse classes between projects. The import-statement can only be used in the *global scope*, that means outside of all other statements.

```
identifiers := identifier ['.|::' identifiers]
statement := 'import' identifiers
```

The statement will first check if *identifier* is the class name of a *native type library*, which is a class written in C / C++ that has been registered to the runtime. If true, the class declaration of this native type library is compiled and the class can be used.

If the identifier is not a native type library, the import-statement will check the local file system's current path for a script file with that name. If a file is found, it is loaded, compiled and can be used. Note that importing files from the local file system can be disabled by the "*file-import*" compiler option (see →*option*). Furthermore, the runtime can be compiled with disabled file import.

When specifying multiple identifier names separated by dots, the import-statement will interpret all identifiers except the last one as subdirectory names, relative to the current path in the local file system:

```
import classes.graphics.gui.button;
```

This will try to import the file "button.jc" from the path "classes / graphics / gui", if such a path is found under the current working directory.

Note that JewelScript assumes that the file name of a script file is the same as the class name defined in that file, plus the file extension for script files, which is usually "jc".

Since not all applications embedding JewelScript probably would want to use the standard extension, a compiler option is available that allows developers to override the file extension assumed by the import-statement.

The statement automatically keeps track of classes that have already been imported. If a class has already been imported, the import-statement will silently ignore all further attempts to import that class.

There is no need to import the built-in classes. These are automatically imported when the JewelScript compiler is initialized and thus are always defined.

Import paths

JewelScript 1.1 introduced a new feature called “import paths”. It’s purpose is to make building libraries of shared script code easier.

By using the API function JCLAddImportPath(), application developers can define any amount of standard paths on the local file-system, where the import-statement should look for script files.

Each import path must be defined with a unique keyword. If this keyword is the first identifier to an import-statement, the path associated with that keyword is used as the base directory.

For example:

```
JCLAddImportPath(pVM, "System", "C:\\Program Files\\My Application\\Scripts");
```

This would allow us to use the following statement in scripts:

```
import System.Console.Application;
```

Which would load and compile the script file “C: \ Program Files \ My Application \ Scripts \ Console \ Application.jc”.

import all

To make HTML documentation generation for an application easier, the keyword ‘all’ can be used to import all native types registered to the runtime at once.

```
import all; // imports all known native types!
```

However, this may cause a performance hit and use a lot of memory with applications that register many classes. For this reason, even though it may seem convenient, it is not recommended to use this feature for normal programming purposes.

using

The purpose of the using-statement is to declare a namespace usage in order to simplify access to global class members. The using statement is valid in the global scope, inside a namespace and inside a class declaration.

```
long_identifier := identifier ['::' long_identifier]
identifiers := long_identifier [',' identifiers]
statement := 'using' identifiers
```

When a class has global functions, global constants or member types, JewelScript requires the developer to specify the *full-qualified name* in order to access the member. This is because more than one class can define a member with that name. In order to distinguish them, their full-qualified name must be specified:

```
CFoo::FooFunction();  // call FooFunction from class CFoo
CBar::FooFunction();  // call FooFunction from class CBar
```

With long class names and members that are often used, specifying the full-qualified name can become tedious. Therefore it is possible to tell the compiler to assume certain namespaces when accessing functions, constants or member types.

```
import CFoo;
using CFoo;

function test()
{
    FooFunction();
}
```

In this example, we tell the compiler to additionally search in class CFoo when searching for global functions, constants or member types. Thus, we can omit the class name when specifying a call to FooFunction().

However, this will only work if no other FooFunction() is already accessible. If a FooFunction() is already defined, either because a global function with that name already exists, or because another using-statement has mapped it from another class, then an ambiguous access error will occur:

```
import CFoo;
import CBar;
using CFoo, CBar;

function test()
{
    FooFunction();              // ambiguous access error!
}

function test2()
{
    CFoo::FooFunction();        // ambiguous access resolved
}
```

As the example shows, such a situation can always be avoided by specifying the full-qualified name of the function that we wanted to call.

If a name conflict exists between a global function and a static class member function, and the global function should be called, this can be resolved by prefixing the function name with the scope operator:

```
import CFoo;
using CFoo;

function test()
{
    ::FooFunction(); // call FooFunction(), not CFoo::FooFunction()
}
```

As of Version 1.2 of the JewelScript language, 'using' can also be used to access member types defined in classes and namespaces. There are generally two strategies for this:

1. Specify the full-qualified name of the type to the using statement:

```
class System::Containers::Array
{
    delegate var Processor(var);
    function Array Process(Array, Processor);
}

using System::Containers::Array;

function main()
{
    Array a = new Array();
    Processor fn = function(e){ ... };
    a = Process(a, fn);
}
```

The above example illustrates an Array class inside a namespace. It has a member delegate and a global member function. To simplify access to the Array class, we specify it's full-qualified name to the using-statement.

This will expose all global member functions, global constants and member types of the given class. While this can make accessing the class very convenient, it is also prone to ambiguous access errors. If any of the class's member names is also accessible from another class or the global space, then this will likely lead to a compile-time error.

For example, a List class could also have a Processor delegate as a member. If we had specified the List to the using statement before, then accessing 'Processor' would be ambiguous.

2. Specify a type's parent-namespace to the using-statement:

```
using System::Containers;

function main()
{
    Array a = new Array();
    Array::Processor fn = function(e){ ... };
    a = Array::Process(a, fn);
}
```

Specifying a type's parent-namespace is a compromise. Instead of exposing all members of the type, only access to the type itself is simplified. The type's name still has to be specified when accessing any of it's members, making it less convenient than the first variant.

This strategy can be safer, because even if the namespace contains a List class that also has a 'Processor' delegate, this is not a problem. Since we have to access the member by 'List::Processor' or 'Array::Processor', no ambiguous situation is produced.

On the other hand, specifying the parent-namespace to the using statement will expose all member classes and interfaces of that namespace. In our second example, the using-statement will expose all type names from 'System::Containers'. This in itself can lead to ambiguous situations.

Consequently, neither solution 1 nor solution 2 is perfect. Which solution is best depends on the actual situation.

Regardless whether using solution 1 or 2, it is always preferable to make the using-statement as local as feasible. Specify it only in the class that requires access to another class, or in the namespace that contains classes that require access to another namespace.

A safer way to simplify access to member types is using the alias-statement.

Unlike most other language constructs dealing with namespaces, the name specified to the using-statement is always *absolute*. Meaning, even if we specify a using-statement inside a namespace, we still have to specify the full-qualified name of the type:

```
namespace System::Containers
{
    class Array
    {
        using System::Containers::List;
    }
}
```

alias

The *alias* statement allows developers to define alias names for existing types. The alias statement can be used in the global space, in a namespace or in a class declaration.

```
long_identifier := identifier ['::' long_identifier]
statement := 'alias' long_identifier identifier
```

By using the alias statement, it is possible to use the type names known from other languages in JewelScript, for example C++.

```
alias float  double;      // make a 'double' type
alias int    long;        // make a 'long' type
```

An alias is not a new type. Both the alias and the original type share the same type-id, and they are always implicitly convertible to each other.

When the JewelScript compiler is initialized, it will automatically define the type aliases *bool* and *char* as alias for the type *int*. Using these aliases may make code look a bit more clear.

```
method bool IsDigit(char c)
{
    return (c >= 48 && c <= 57);
}
```

Aliases are especially useful when we want to simplify access to member types of a class:

```
class System::Containers::Array
{
    delegate var Processor(var);
    function Array Process(Processor);
}

alias System::Containers::Array MyArray;

function Test()
{
    MyArray a = new MyArray();
    MyArray::Processor fn = function { ... };
    a = MyArray::Process(a, fn);
}
```

Accessing member types and functions from a class that is deep in a namespace can be cumbersome. While we can make this a lot easier by employing the using-statement, there is always the risk that this creates ambiguous situations.

An ambiguous situation occurs when multiple classes define a member type of the same name and the compiler cannot determine which type the user wants to access.

However, unlike the using-statement, the alias statement lets us shorten long class names without creating ambiguous situations.

It is preferable to declare aliases locally in a class where we need simplified access to other classes. That way other classes are not affected by the alias, and they can reuse the aliases name.

namespace

This statement allows to declare one or more namespaces in the current scope. The statement can be used at global scope or inside another namespace-statement.

```
long_identifier := identifier ['::' long_identifier]
statement := 'namespace' long_identifier '{' statements '}'
```

Valid statements in a namespace statement are *namespace*, *class*, *interface*, *using*, *delegate* and *alias*.

It is also possible to implement functions and methods in a namespace, if they have been declared in a member class of that namespace. It is not possible to define global functions or constants directly in a namespace though.

The purpose of the statement is to avoid name conflicts between types. For example, a Console library could define a Rect class which is based on character coordinates. If another library would then try to define a Rect class for graphical coordinates, this would result in a name conflict.

```
namespace Console
{
    class Rect // define Console::Rect
    {
    }
}
namespace Graphics
{
    class Rect // define Graphics::Rect
    {
    }
}
```

In general, the namespace name of any member type does not have to be specified inside of that namespace.

So in the above example, class Rect can be used without the prefix "Console::" in all classes that are members of the Console namespace. Likewise, we can use Rect without the prefix "Graphics::" for all classes in the Graphics namespace.

Namespaces are also created if we define a member type inside of a class:

```
class Console
{
    class Rect // define Console::Rect
    {
    }
}
```

The same effect can be produced by declaring the Rect class outside of class Console:

```
class Console
{
}
class Console::Rect // define Console::Rect
{
}
```

To simplify access to namespace members, outside of that namespace, the using- and alias-statements can be used. However, alias can only be used with a discrete type, not with a pure namespace name.

option

The option-statement allows developers to change certain compiler options, such as warning level, optimization level, and so on. The option-statement can only be used in the global scope.

```
statement := 'option' string_literal
```

The language uses a global set of compiler-options that can be defined by the application embedding JewelScript.

In addition, the option-statement can be used to specify options which are only valid within the file that contains the option-statement. When importing a file using the import-statement, the global set of options will be copied for that newly imported file, but can be altered within that file by the option-statement. This means that options specified in one file will generally *not* be carried over into another file.

The string literal *string_literal* is assumed to specify a comma separated list of *name* = *value* pairs. Values can be boolean, integers or strings, depending on the option. For switches, the words *true, false, yes* and *no* can be used as an alternative to the integer values 1 and 0.

```
option "warning-level=2, verbose=yes";
```

The following list explains all supported compiler options in detail. Defaults specified refer to the release build of the library.

verbose

```
verbose = 0|1 (default: 0)
```

Enables / disables additional compiler output, for example number of allocated and freed blocks, number of errors and warnings, optimizations successfully performed, generated code size, etc.

warning-level

```
warning-level = 0...4 (default: 3)
```

Set the filter level for the output of warnings. Warning level 0 disables all warnings, 1 outputs only critical warnings, 2 outputs critical and important warnings, 3 outputs critical, important and unimportant warnings, 4 outputs all warnings.

stack-locals

```
stack-locals = 0|1 (default: 1)
```

This option specifies whether local variables will be held in virtual machine registers or on the stack. For full byte code optimization, this needs to be enabled.

In general, real life performance tests have shown that using registers for local variables does not significantly increase the performance over using local variables on the stack.

optimize

```
optimize = 0...3 (default: 3)
```

This option will set the optimization level to use. A value of 0 disables all optimization, 1 will only perform basic optimizations, 2 will perform advanced optimizations and 3 will perform full code optimization. Note that the level 3 optimizations require that the “stack-locals” option is enabled. Therefore, when setting this option to 3, the option “stack-locals” is automatically set to 1.

use-rtchk

```
use-rtchk = 0|1 (default: 1)
```

This option enables or disables the placement of the rtchk instruction whenever a type-less variable gets converted to a distinct type. The rtchk instruction uses the runtime type information of the virtual machine to verify that the object in the variable is really of the type that the code is trying to convert to, and if not, will generate a virtual machine exception.

file-ext

```
file-ext = string (default: "jc")
```

This option changes the file extension for script files the import-statement assumes. If an application wants to import external files, but requires them to have an extension different from "jc", developers can use this option to specify the extension to use. The specified string must not include the dot character, or any other invalid characters. Valid characters are:

```
fileext := [0-9A-Za-z_]+
```

file-import

```
file-import = 0|1 (default: 1)
```

In certain applications that embed the JewelScript runtime, it might be unwanted that script developers can access the local file system to import other script files. This option allows to disable loading and compiling of additional script files. Native types registered to the runtime are not affected by this and can still be imported. Note that there also is a preprocessor macro in "jilplatform.h" that allows to build the runtime library without support for file import.

error-format

```
error-format = default|ms (default: default)
```

Allows to change the format in which error and warning messages are presented. "default" is JewelScript's format, where the file name, line and column occur last in the message. Developers using Microsoft® development tools can switch to "ms" format, where file name and line occur first in the message. This allows them to double click on an error in the output window, which directly opens the file in the IDE and sets the cursor to the error line.

log-garbage

```
log-garbage = none|brief|all (default: none)
```

If this option is enabled, detailed information about each value reclaimed by the garbage collector is written to the log output callback of the runtime. This allows script programmers to find and optimize reference cycles in their program.

Setting ‘brief’ will only list values directly destroyed by the garbage collector, this is the most useful option. Setting ‘all’ will list all values, directly or indirectly destroyed by GC. This may be useful to investigate in more detail, what kind of values have leaked.

This option is **global**, it is not applied on a per-file basis. If multiple files use this option, the value from the last compiled file is used.

data-stack-size, call-stack-size, stack-size

```
stack-size = n (default: application defined)
```

These options set the sizes for the data stack, the call stack, or both of them respectively. This allows applications to waste less memory, by specifying a smaller default stack size when initializing the runtime. Instead of picking a "worst case" stack size, it allows to use a smaller one that is suitable for most common scripting needs. If a particular script file should require a bigger stack, for example due to heavily use of recursions, it can use these options to request larger stacks.

The data stack size specifies the maximum amount of local variables and function arguments the virtual machine can handle at any point during runtime. The minimum data stack size is 128. The call stack size specifies the maximum amount of nested function calls the virtual machine can handle. Obviously, this is usually a smaller figure than the space needed for variables on the stack. Thus, it makes sense to specify data stack size and call stack size separately. The minimum call stack size is 32. If setting the size of both stacks using the 'stack-size' option, the call stack size will be set to one quarter of the data stack size.

Note that these options can only be used when the virtual machine has not yet been initialized, which means, when it has not yet executed code. Once code has been executed, the stacks are created and the options cannot be specified any more, should the application continue to compile code.

Scripts that use a lot of co-function threads can optimize memory usage by tweaking the stack sizes. However, too small stack sizes carry the risk of stack overruns, which will lead to memory corruption and crashes.

These options are **global**, they are not applied on a per-file basis. If multiple files use these options, the value from the last compiled file is used.

document

```
document = user|builtin|all (default: user)
```

This option controls which types will be documented when invoking the HTML documentation generator.

Setting ‘user’ will only document user classes, meaning everything but the runtime’s built-in types. Setting ‘builtin’ will only document the built-in types. Setting ‘all’ will document all types.

All settings require that the respective types have been imported or declared when the HTML documentation generator is invoked. The *import all* statement may be used to simplify documenting all classes.

This option is **global**, it is not applied on a per-file basis. If multiple files use this option, the value from the last compiled file is used.

delegate

The delegate statement defines a new delegate type. It associates a function signature with a type name, which can then be used to declare delegate variables. The statement can be used at the global scope and inside a class declaration.

```
typedecl := ['const'] ['weak'] typename ['[]']
arguments := typedecl [identifier] [',' arguments]
statement := 'delegate' typedecl identifier '(' arguments ')'
```

All delegate types that share the same function signature also share the same type-id, so they are interchangeable.

```
delegate int Comparer(string a, string b);
delegate int Verifier(string a, string b);

Comparer fn = function { return a == b; };
Verifier ve = fn; // OK since signatures match
```

For more information, see the →*Delegates* section of this document.

__brk

The __brk statement allows to directly place a BRK instruction in a JewelScript function. This can be used to debug script code.

```
statement := '__brk'
```

Upon execution, the BRK instruction will generate a break exception, which can be caught and handled by a C / C++ callback function registered to the virtual machine. The main purpose of the BRK instruction is of course providing a break-point functionality for debugging purposes.

It can, however, also be used for other purposes, depending on the application and what C / C++ handler it registers for this exception.

Variable declaration, definition

In JewelScript, a variable needs to be declared and defined, before it can be used. A variable is declared by a variable declaration statement:

```
typedecl := ['const'] ['weak'] typename ['[]']
identifiers := identifier ['=' expr] [',' identifiers]
statement := typedecl identifiers
```

As an example, this statement declares a floating-point variable with the name 'foo':

```
float foo;
```

It is possible to declare more than one variable in a single declaration statement:

```
float foo, bar, x, y;
```

When declaring a variable without immediately assigning an initial value, the compiler will automatically initialize the variable with a *default value*. In the case of a numerical type, this default value is 0. For class types, the compiler will try to construct an instance by calling the *default constructor*. If the class does not have a default constructor, or it has been declared *explicit,* a compile-time error is produced.

For an in-depth discussion of variables, please refer to the section →*Variables* of this document.

Function declaration, definition

Similar to variables, functions need to be declared before they can be used. Functions can be declared by a function declaration statement:

```
typedecl := ['const'] ['weak'] typename ['[]']
arguments := typedecl [identifier] [',' arguments]
statement := 'function' [typedecl] identifier '(' [arguments] ')'
```

Here are a few examples of function declarations:

```
function foo1();                        // no result, no arguments
function foo2(int);                     // no result, one int argument
function int foo3(int, int);            // int result, two int arguments
function string foo4(const CFoo);       // string reference result,
                                        // const CFoo reference argument
```

After it's declaration, the function's body is not yet defined. When compiling code, JewelScript allows a function to be called that has been declared, but not yet defined. A function's body can be defined by a function definition statement:

```
arguments := typedecl identifier [',' arguments]
statement :=
    'function' [typedecl] identifier '(' [arguments] ')' block-statement
```

Note that declaring a function explicitly before defining it is not required. If a function is defined by a function definition statement that has not yet been declared, it will automatically be declared, then defined.

For a simple function that calculates the Fibonacci sequence, the declaration and definition statements would look like this:

```
function int fib(int);     // declaration

function int fib(int n)    // definition
{
    if( n < 2 )
        return 1;
    return fib(n - 1) + fib(n - 2);
}
```

For an in-depth discussion of functions, please refer to the section →*Functions* of this document.

Co-function declaration, definition

Co-functions are declared by a co-function declaration statement:

```
typedecl := ['const'] ['weak'] typename ['[]']
arguments := typedecl [identifier] [',' arguments]
statement := 'cofunction' [typedecl] identifier '(' [arguments] ')'
```

An example declaring a simple co-function:

```
cofunction int Generator(int, int);
```

The same rules that apply to defining a regular function apply when defining a co-function. After it's declaration, the co-function's body is undefined and needs to be defined by a co-function definition statement:

```
arguments := typedecl identifier [',' arguments]
statement :=
    'cofunction' [typedecl] identifier '(' [arguments] ')' block-statement
```

Inside the co-function's body, the same statements can be used as for normal functions, except for the return-statement, which cannot be used since a co-function is a thread. Instead, a co-function can be suspended by the yield-statement.

```
cofunction int Generator(int min, int max)
{
    for( int i = min; i < max; i++ )
        yield i;
}
```

For more detailed information about co-functions, please refer to the →*Co-functions* section of this document.

Class declaration, definition

Classes in JewelScript work very similar to Java and C#. A class can be declared by a *class declaration statement*. A class declaration statement has the purpose of making the class name, it's methods, member variables, global functions and global constants known to the language. A class declaration could look like this:

```
class Foo
{
     method Foo();               // a constructor, recognized by the name
     method DoSomething();       // an instance member function
     function SomeFunc();        // a global member function
     string m_Str;               // a member variable (instance data)
     const int kConst = 11;      // a global class member constant
}
```

After having declared a class, the class can be instantiated and all members of it can be accessed. In order to define any methods or global functions that have been declared in the class, a function or method definition statement can be used:

```
method Foo::Foo()
{
     m_Str = "Hello";
}
method Foo::DoSomething()
{
     m_Str += "World!";
}
function Foo::SomeFunc()
{
     print( "Hello World!\n" );
}
```

It is, however, also possible to define the method and function bodies *inline*, meaning directly in the class declaration statement:

```
class Bar
{
     method Bar()
     {
          m_Str = "Hello";
     }
     string m_Str;
}
```

Under certain circumstances we need to *forward declare* a class in order to use the type name, before we can actually declare the class. We can forward declare a class by a class forward declaration statement:

```
class Foo;
```

After having forward declared a class, its name is known to the language and can be used to declare variables and function arguments of that type. It is, however, not possible to instantiate a class that has only been forward declared, or access any members of it.

Classes are discussed in more detail in the →*Classes* section of this document.

Interface declaration

Interfaces are *pure abstract* class declarations. They can be implemented by classes. An interface is declared by an interface declaration statement:

```
interface Foo
{
     method string GetName();
     method SetName(const string);
}
```

A class that would implement interface 'Foo' would *at least* need to implement two methods, GetName() and SetName(), and at least one constructor, as mentioned earlier. A class declaration that implements interface 'Foo' might look like this:

```
class Bar : Foo   // class Bar implements Foo
{
    method Bar() { m_Name = "Unnamed"; }
    method string GetName() { return m_Name; }
    method SetName(const string name) { m_Name = name; }
    string m_Name;
}
```

Since class constructors in JewelScript are regular methods, they can be declared in interfaces as well. The consequence of declaring a constructor in an interface, is that all implementing classes must implement a matching constructor.

```
interface Foo
{
    method Foo(const string);
    method string GetName();
    method SetName(const string);
}
class Bar : Foo
{
    method Bar(const string name) { m_Name = name; }
    method string GetName() { return m_Name; }
    method SetName(const string name) { m_Name = name; }

    string m_Name;
}
```

In the above example, class 'Bar' derives the constructor from interface 'Foo' and therefore must implement a similar constructor, otherwise this will result in a compile-time error.

To learn more about interfaces, please refer to the →*Interfaces* section of this document.

Expression statement

The language allows to specify a single expression anywhere where a statement is expected. In this case, the result of the expression is thrown away.

```
statement := expr
```

While this might at first seem odd to some readers, the expression statement is in fact one of the most often used statements while programming in any C-style language. This is because a function call is an expression, as well as an assignment to a variable:

```
c = 27;               // an expression statement
FooFunction( 27 );    // another expression statement
d++;                  // and another one
```

Note that JewelScript is a bit more restrictive than other C-style languages when it comes to expression statements. More specifically, it does *not* allow multiple expressions combined by *binary operators* to be used as an expression statement.

```
1 + 2;   // not allowed
c << 1;  // not allowed
```

The general rule of thumb is that the language disallows developers to create "useless code", code which might affect performance, even though the result of it will be thrown away.

It might seem weird that JewelScript allows to specify a *literal constant* as an expression statement:

```
1000;
"Hello";
3.141;
```

This is because doing this will result in *absolutely no code*. The compiler is smart enough to know that these values are never used and just ignores them.

Unlike C, C++ and other C-style languages, JewelScript does not support multiple expression statements delimited by operator comma. In JewelScript the comma is not an operator at all, it is just a token recognized to delimit function arguments.

Empty statement

C and C++ support an empty statement to be placed, meaning a statement that contains no code. One reason for this is the C / C++ preprocessor, which allows to define macros, which in turn can result in source code being removed from the file before the compiler 'sees' it.

Even though JewelScript does not have a preprocessor, it allows an empty statement to be specified as well.

```
statement :=
```

This is generally of little use when coding in JewelScript. One area where it probably could be useful is as a *place holder* in if-statements:

```
if( condition == 27 )
    /* TODO: implement condition! */;
else
    Dispatch( condition );       // normal case already implemented
```

Variables

In general, a variable is a symbol denoting a quantity or value. More technically spoken, a variable is a user-defined identifier that the language associates with a location in memory where a certain value is stored.

In JewelScript, a variable is actually a *reference* to a value, it does not directly store the value. Instead, it stores the *address* in memory, where the value has been put. However, when using value types (as opposed to *reference types*, discussed later in this document), this is totally transparent to the developer, meaning value type variables appear to the developer as if they directly store a certain value.

Variables in JewelScript are bound to a distinct data type. This means that a variable can only store values of the type the variable is bound to. If the language detects that a value of a different type is assigned to a variable, this is considered to be a programming error, unless the language knows of a way to convert the value to the type of the variable.

Older versions of JewelScript distinguished between declaration, initialization and assignment of a variable. However, this has been simplified, and at present a variable's declaration implicitly is it's definition. In addition, initialization and assignment are treated equally.

Declaration / Definition

Before we can use a new variable, we need to declare it. Declaring a variable has the purpose of making the variable known to the language. This includes making the variable's name, the data type bound to the variable, and the variable's visibility and life-time known.

The following statement declares the floating-point variables 'x' and 'y':

```
float x, y;
```

Note that the location where we put a variable declaration is of importance, since it determines the visibility and life-time of the variable (also termed the *scope* of the variable). If we place a variable declaration in the *global scope*, then this variable will be a global variable, meaning it's life-time spans the whole life-time of the program and it is visible from anywhere in the program.

If we place a variable declaration in a function definition, then this variable's life-time only spans the time the function is being executed. Once the function returns, the variable does not exist any more. Furthermore, the variable is only visible in that function, which means that other functions cannot refer to the variable.

When a global or local variable is declared, but no immediate initialization value is assigned, the compiler will automatically initialize the variable with a *default value*. For the numerical types float and int this default value is 0. For class types, the compiler will try to create an instance by calling the *default constructor* of the class. If the class does not have a default constructor, or it has been declared *explicit,* this will result in a compile-time error.

When a class member variable is declared, it's value is initially *null*, until it gets initialized in the class constructor. The compiler will generate a compile-time error when a constructor does not initialize all member variables.

Initialization / Assignment

The initialization of a variable is defined as *the first assignment* of a value to it. To initialize a global or local variable with a value other than the default value, we need to immediately assign this value when the variable is declared:

```
float x = 3.141;
float y = 27.5;
```

For convenience, we can use a single statement to initialize both variables:

```
float x = 3.141, y = 27.5;
```

Note that special rules apply when it comes to initializing variables declared in classes (class member variables), these variables can *not* be immediately initialized in the declaration statement, but must be initialized in the constructors of the class instead.

When initializing a value type variable, the right-hand value gets copied. When initializing a reference variable, a reference to the right-hand value is taken.

Any further assignment to a variable after it's initialization is handled exactly like the initialization.

References

Reference variables allow us to *share* the same value between multiple variables. Unlike value type variables, the language will *not* copy the right-hand value when initializing a reference variable, but instead will create a new reference to the right-hand value. This means that two or more variables can actually refer to the *same object* in memory.

The purpose of using reference variables is to avoid excessive and unnecessary copying of data, which can severely reduce performance. If our program for example creates a large structure of deeply nested objects and wants to pass this as an argument to a function, then this structure is passed by reference, which is significantly faster than copying all this and pass the copy to our function.

However, passing values by reference has some important consequences: If two variables share the same object in memory, and we *change* this object using one variable, then this change will also be visible through the other variable. Sometimes this *side effect* is explicitly wanted by the developer, but sometimes it can also be the cause of a programming error.

In order to explicitly assign a copy of an object to a reference variable, we have to create a new instance using the type's copy-constructor.

```
string s = "hello";
string t = s;
s += " world";    // side-effect: 's' and 't' are now "hello world"

string s = "hello";
string t = new string(s);
s += " world";    // no side-effect: only 's' has changed
```

Declaration / Definition

All types except the built-in types int and float are implicitly considered reference types. Thus, declaring a variable with a reference type will declare a reference variable:

```
Foo g = new Foo();    // declare reference to a new Foo instance
```

If a reference variable is not initialized explicitly by the developer, the language will try to automatically initialize it by calling the type's default constructor.

```
Foo f;                // declare reference to a new Foo instance
```

When assigning an object that is constant, the language will automatically create a copy and assign a reference to the copy instead.

```
string s = "Hello";   // assigns a copy of the constant "Hello"
```

It is not mandatory to declare function arguments constant. However, it has the advantage that we don't have to take care of copying arguments inside the function.

```
class Rect
{
    method Rect(const Point topLeft, const Point bottomRight)
    {
        this.TopLeft = topLeft;         // argument is copied
        this.BottomRight = bottomRight; // argument is copied
    }
}
```

Another solution is copy-constructing the values instead of making the arguments constant.

```
class Rect
{
    method Rect(Point topLeft, Point bottomRight)
    {
        this.TopLeft = new Point(topLeft);
        this.BottomRight = new Point(bottomRight);
    }
}
```

Assignment

The assignment to a reference variable is handled exactly like the initialization. After the assignment, the variable will refer to a new value in memory.

However, if the variable is declared *const*, then it is not possible to assign a new reference to that variable. Declaring a variable as constant means that both the value and the reference to it are meant to remain constant.

Reinitialization

Former versions of JewelScript required a special operator to re-initialize a reference variable. This operator is no longer required. We can just assign new values to reference variables.

```
string s = "hello"; // init with copy of "hello"
s = "world";        // assign copy of "world"
s = null;           // assign null reference
```

Weak references

The weak reference is a special kind of reference that slightly alters the way the language treats referenced values. We can imagine a weak reference as a reference that is not counted. Although this is not the way they are implemented in the virtual machine, their effect is like that. A variable can be declared as a weak reference, by prefixing the modifier keyword *weak* to a reference variable declaration.

```
weak Foo wFooRef = myFoo; // take weak reference from 'myFoo'
```

Introducing the concept of weak references into the language is a mixed blessing. On one hand, they help developers to avoid *reference cycles,* a problem that can occur in reference counted systems when a reference counted object directly or indirectly references itself. This is explained more detailed in the next section.

On the other hand, weak references have an effect on the safety of a program, since they circumvent two important features of the language:

- Freeing the developer from thinking about the life-time of objects. When not using weak references, the language can guarantee that an object referenced by a variable always exists. An object cannot be destroyed until there is no variable any more that uses it. Weak references do not provide this safety. Since their reference to the object is not counted, the object might be gone before the weak reference variable is destroyed. The weak reference would then reference a *zombie:* an object that doesn't exist any more. If the zombie's memory area has been reused by the application for other values in the meantime, modifying such a zombie object will overwrite these values, which will sooner or later lead to an application crash.
- Freeing the developer from the risk of creating memory leaks. When not using weak references, the language can make sure that all dynamically allocated objects are freed automatically as soon as they are no longer in use, provided the developer did not create the aforementioned reference cycle. When using weak references, it is theoretically possible to create a memory leak, if no other variable exists that keeps a counted reference to the same object. Even though the language tries to detect such situations, for example when developers try to directly allocate a new object into a weak reference variable, there are still ways of producing such a situation that the language cannot detect.

Because of these disadvantages, weak references should only be used if there is a problem that cannot be solved otherwise. In other words, if the program does not have a reference cycle problem, then there is no reason to use weak references.

Yet, if it turns out that a program has a reference cycle problem, using weak references to solve the issue should be the *last* option on the list. Reference cycles can very often be avoided by a "proper" design of the relationships between objects.

For example, the simple addition of a “dispose()” method to our class, that we call when we don't need the object any more, can help us avoid reference cycles, if this dispose method sets all it's member references to null.

The following sections will explain the reference cycle problem in more detail and give some pointers on how they can be avoided or at least broken.

Reference cycles

Reference counted garbage collection could probably be the perfect solution to all garbage collection problems, if there wasn't the reference cycle problem. A reference cycle can be created, when two or more ref-counted objects reference each other. Another way to put it, is that a garbage cycle is created when a ref-counted object directly or indirectly references itself.

This can be best explained by a simple A / B example:

```
class Object
{
    method Object() { ref = null; }
    Object ref;
}


function Test()
{
    Object A, B;      // default construct two objects
    A.ref = B;        // have A reference B
    B.ref = A;        // have B reference A
}
```

In the above example code, we create two reference counted objects, `A` and `B`, of a type called Object. The type doesn't really matter, the important thing is that this type allows the object to keep a reference to another object.

After the code in function test has run through, and just before the function returns, both objects have a reference count of 2. One reference is from the other object, the second is from the Test function's stack.

Now we leave the function, which will free the function's stack, which in turn will release it's references to the variables `A` and `B`. Hence the reference counts of both objects are decreased to 1. That's all, nothing else happens. Both `A` and `B` will not be destroyed and have 'leaked'.

Since object `A` still has a reference from `B.ref`, the runtime cannot destroy it, it must assume `A` is still in use. Likewise, since object `B` still has a reference from `A.ref`, the runtime cannot destroy it, it must assume `B` is still in use.

In real programming life, reference cycles can be much, much more complex. Imagine a chain of hundreds of Object instances, each one referencing another instance (A → B → C → D), and the last one referencing back to A. This cycle would be just as unbreakable as our small example.

Most reference counted languages therefore employ an additional *garbage collector* that periodically searches the heap for leaked objects due to reference cycles and frees them. JewelScript offers an optional “mark and sweep” garbage collection mechanism for this purpose. This garbage collector can be run periodically by the application to make sure that reference cycles are found and freed.

Avoiding reference cycles

When designing code and classes, probably the most important thing is to think about the relationships between objects, and how they should interact.

Typical questions we need to ask ourselves are usually what objects need to *collaborate* with each other, and what objects have an *ownership* relation to other objects.

We can define a *collaboration* as a relation where two or more objects work together very closely, and therefore need to "know" each other. A pointer or reference is used as the "link" between these objects. Often, there is a rather flat hierarchy between these objects.

An *ownership* relation is much more strict, hierarchical, and mutually exclusive. An object that owns another object can not be owned by that object at the same time. One object always is the owner, the other object is the "owned". The owned can never directly or indirectly own its owner.

While these relationships are typically expressed by using pointers in C/C++, they can only be expressed using references in JewelScript.

Developers that have knowledge in C or C++ might be inclined to understand JewelScript references as "something like pointers". However, this way of thinking about references might be misleading and leading to code design that would work well with pointers in C, but not necessarily using references in JewelScript.

Pointers in C/C++ define the relationship between objects rather loosely. A pointer can represent an ownership relation, a collaboration, or anything in between. This is completely different with references in JewelScript.

Due to the mechanics of the virtual machine and its reference counting, the relationship between objects expressed with references can almost always be seen as an ownership. So whenever we add a reference variable to a class definition, we say "This class owns an instance of that type."

```
class Employee;
class Company
{
    Employee worker;  // Company owns Employee
}
```

While designing collaborations in JewelScript can be tricky and will almost always require us to use weak references, expressing ownership relations works quite well using references. So if possible, when thinking about their class design, developers should choose a hierarchical model using ownership relations, rather than a flat collaboration model.

As an example, lets try to find a class model of a company with a CEO, managers, project leaders, and several employees. Here is our first attempt:

```
class Company
{
    CEO                 theChief;
    Manager             humanResources;
    Manager             production;
    ProjectLeader       newCoolProduct;
    Employee[]          employees;
}
class CEO
{
    Company             myCompany;
}
class Manager
{
    Company             myCompany;
}
class ProjectLeader
{
    Company             myCompany;
}
class Employee
{
    Company             myCompany;
}
```

Now we have a Company class that owns a CEO, two managers, a project leader and several employees. Each of the other classes have (next to other data that I omitted to keep things compact) a reference back to the company they work for.

This is necessary, because the CEO needs to give orders to the management, the management to the project leader, and the project leader to the employees. So the easiest thing is to give them all a reference back to the main company object.

Its not hard to guess – this design is really, really bad. And it will cause nothing but problems in JewelScript. More specifically, it will cause the reference cycle that has been mentioned earlier.

If we stick to regarding references as "is owned by", we will actually notice the mistake right away. The company can own the employees, but can the employees own the company as well?

References back to something are almost always a bad idea, or if we can't do without them, almost always a candidate for a weak reference. Because references back to something usually indicate that we are probably working on a collaboration design, rather than an ownership design.

Finding a better class design

We can stick to a hierarchical ownership design and avoid using weak references, if we redesign our classes.

```
class Company
{
    CEO               theChief;
}
class CEO
{
    Manager           humanResources;
    Manager           production;
}
class Manager
{
    ProjectLeader     theLeader;
}
class ProjectLeader
{
    Employee[]        employees;
}
class Employee
{
}
```

Now the CEO has access and can give orders to anyone in the company, while each manager has access to his / her project leader and employee teams. One problem, however, remains unsolved: How can the employee, project leader, manager report back to their higher instance?

If our design really requires this form of access, we probably need to use a weak reference. We could add a weak reference to every class that references back to the company object for this purpose. However, that's not a must – there are other solutions.

Make one reference direction temporary

If there are only a few methods in each class that occasionally need access to the company object, we could pass the company object by reference as an argument to the method that needs it. This way of referencing back will not cause a reference cycle, because the reference back to the company object then is a local variable, and thus has a well defined life span. It will only exist until the method call has returned.

For example, the human resources manager most likely would need access to all of the company, including other managers and the CEO, in order to pay the monthly salaries. So the PaySalaries() method could get passed a reference to the company object.

```
method Manager::PaySalaries(Company theCompany)
{
}
```

Help the runtime determine the life-time

As already mentioned earlier, we can easily circumvent reference cycles by defining the life-time of objects ourselves, by adding a method that we call when the object is no longer needed.

If we would add a “dispose” method to all our example classes above, which is called when the company object is no longer used, then we could simply set all our references to null after propagating the dispose call on to our child objects.

That way, the reference cycle would be broken once the main object gets disposed, and therefore we would not have any memory leaks.

Use weak references for one direction

If none of the above methods to avoid reference cycles seem feasible, then we'll have to use a weak reference. Using a weak reference is safe, if we can guarantee that:

- There is also a counted reference to the main object somewhere in the program.
- The life span of the main object is at least as long as the lifespan of the objects that own the weak reference.

Employ the garbage collector

Of course it can also be a solution to simply accept that reference cycles are created and have the garbage collector periodically search for them and free them.

Weak references and co-functions

Just like regular global functions, co-functions can be declared in classes, making them class member co-functions. However, having a co-function in a class can lead to a reference cycle problem: In order to access instance members of the class, we would need to pass a reference to the instance (the *this* reference) to our co-function.

```
class Foo
{
     method Foo()
     {
          m_CoFunc = new MyCoFunc(this); // pass this to cofunction
     }
     cofunction MyCoFunc(Foo self)
     {
     }
     MyCoFunc m_CoFunc;
}
```

This little example class illustrates the problem: Class Foo owns a reference to a thread object that gets initialized with a reference to an instance of class Foo.

This will create a reference cycle, because the Foo instance owns a thread object, the thread object owns a stack, and the stack owns a reference to our Foo instance. Thus, we have a Foo instance indirectly referencing itself.

Because of this, a co-function that is a member of a class and should have access to instance members of the class, must use a weak reference.

```
class Foo
{
     method Foo()
     {
          m_CoFunc = new MyCoFunc(this);
     }
     cofunction MyCoFunc(weak Foo self)
     {
     }
     MyCoFunc m_CoFunc;
}
```

There is no problem in doing this, since in this case it is guaranteed that:

- There is always a counted reference to the Foo instance (when instantiating the class).
- The Foo instances life span is as long as the thread object's life span.

Constants

Constants, as opposed to variables, are values that remain unchanged during the whole life-time of a program. The simplest form of constants are literals:

```
0x7fff
3.14159
"Hello World!"
```

However, JewelScript also supports symbolic constants, which associate an identifier name with a type and a value. This is done by adding the 'const' modifier keyword to a variable declaration statement:

```
statement := 'const' ['weak'] typename ['[]'] identifier '=' expr
```

Note that immediately assigning an initialization value is mandatory for symbolic constants, whereas it is optional for variables.

Some languages treat symbolic constants as if they were literals, meaning if a symbolic constant 'kFoo' is encountered while compiling an expression, the compiler will simply replace that with the value defined for kFoo and compile that value as if it was specified literally. The result of this is that symbolic constants do not take up any space in classes in these languages, which is an advantage. However, it also means that the actual value of the constant needs to be already known during compile-time, so in these languages a constant can only be initialized by a constant expression, meaning an expression that does not depend on runtime data.

In JewelScript, symbolic constants are treated in exactly the same way as variables. The only difference is that symbolic constants can not be changed after they have been initialized. Any further assignment to such a constant after it's initialization will result in a compile-time error. In a way one can see a symbolic constant as a variable that is “write once”.

Of course a disadvantage of this approach is, that JewelScript constants take up memory space in the program. The advantage is, that JewelScript allows a fully dynamic initialization of symbolic constants. Actually, any way of initializing a variable is also valid for symbolic constants.

```
function string MakeRandomName(int length)
{
    string result = "";
    for( int i = 0; i < length; i++ )
        result += string::ascii( rand(65, 90) );
    return result;
}

const string kFileName = MakeRandomName(8) + ".TXT";
```

The above example defines a function that randomly generates a string of a given length. Later, a global constant is defined that calls the function to generate a random file name and add the extension ".txt" to it.

The ability to dynamically initialize constants can also be used to automatically generate enumerated constants. Developers who are missing the enum-statement in JewelScript can use the language's ability to dynamically initialize constants to come by this missing feature. Here is an example of how to generate automatically enumerated constants in JewelScript:

```
int g_Enum = 0;
function int enu(int v) { return g_Enum = v; }
function int enu() { return ++g_Enum; }


// enumerated constants
const int kAlignLeft = enu(0);
const int kAlignRight = enu();
const int kAlignCenter = enu();
const int kNumAlignModes = enu();
```

```
// enumerated flags
const int kWantDraw = 1 << enu(0);
const int kWantClick = 1 << enu();
const int kWantDrag = 1 << enu();
const int kNumFlags = enu();
```

Of course this solution does not save developers any typing effort, so having a real enum-statement available would still be preferable.

Constant references

The ability to declare constants is especially important when it comes to passing values by reference to functions. If a function argument is a constant reference, any value we pass to the function is automatically copied, if it gets assigned to a non-constant variable inside the function.

Note that declaring a reference as constant means that both, the reference *and* it's value are meant to remain constant. That means, neither are we allowed to change the value, nor are we allowed to change the reference.

```
const string k = "hello";
k += "world";              // error: cannot change k's value
k = "world";               // error: cannot change reference
```

Constant arrays

When declaring an array using the const modifier, the language will treat the array variable, as well as each array element, as a constant.

This means that we are neither allowed to change the array itself, nor any elements of it.

```
const int array arr = {10, 12, 14};
arr = {20, 22, 24};        // error: cannot assign a new array
arr += 26;                 // error: cannot add new element
arr[0] = 20;               // error: cannot change element
```

Constant objects

When declaring a reference to an object as constant, the language will treat the reference variable, as well as all members of the object as constants.

```
function Test(const CFoo foo)
{
    foo = new Foo();       // error: cannot assign a new object
    foo.value = 27;        // error: cannot change members
}
```

Constant methods

In it's current state, the language does not support declaring methods with the const modifier. That means, if a reference variable is constant, the language will simply ignore the constancy of the object when calling methods.

```
function Test(const CFoo foo)
{
    foo.value = 7;         // error: cannot change member variable
    foo.SetValue(7);       // works even though 'foo' might get changed
}
```

It is safer to use set-accessors instead of methods to modify an object's properties, because the compiler can properly check if calling set-accessors would modify a constant object.

The ability to declare constant methods might be supported by a future version of the language.

Functions

In JewelScript, the term 'function' always refers to a *global function*, while the term 'method' always refers to an instance member function.

JewelScript functions can be declared in the *global scope*, which means outside of any class declaration, or inside of a class declaration, making it a global class member function. They can not be declared directly in namespaces.

Functions and methods can be used as first class values, meaning references to them can be passed around as values.

Functions and methods can also be nested by defining local anonymous delegates. The following sections generally apply to functions and methods, even though only functions are mentioned.

Declaration

Functions need to be declared before they can be used. Declaring a function has the purpose of making the function known to the language. When a function has been declared, the compiler knows it's name, scope, argument types and result type. However, it doesn't know anything about what the function *does*, the actual code of the function, the function's body, is still undefined.

The following statement declares a function 'fib', that returns an int value and accepts an int value as an argument:

```
function int fib(int);
```

While compiling source code, JewelScript allows a function to be called that has been declared, but not yet defined. However, before any code can be executed, the *link process* will make sure that every function body has been defined.

JewelScript does not require the developer to use function declaration statements. If a function has not yet been declared, a function definition statement will automatically declare it, then define it. However, there are situations where a function declaration statement can be useful to *forward declare* a function, to make it known and to be able to call it, but implement it later, maybe even in another source file.

It is possible to declare multiple functions with the same name, but different result or argument types. This can create some ambiguous situations, though. When a call to a function is compiled, and multiple functions with that name have been declared, JewelScript will take the number of function arguments, their types and the result type into account in order to find the correct variant to call.

When declaring a function, the compiler will check the new function against existing functions in order to detect ambiguous declarations, and generate a compile-time error if a problem is detected. Ambiguous declarations can occur, when for example using the 'const' or 'weak' modifiers inconsequently:

```
function foo( const float );
function foo( float );          // error in redeclaration

function foo( weak string );
function foo( string );         // error in redeclaration
```

They can also occur in combination with *type-less* arguments or result types. If, for instance, one function 'foo' accepts an int as an argument, and another function accepts a type-less argument, then this will be ambiguous, because the compiler cannot know which variant of the function the developer is going to call. Both would be suitable when calling foo with an integer value as argument:

```
function foo(int n);  // foo accepting an int
function foo(var n);  // foo accepting anything produces conflict
```

Definition

Functions do not need to be explicitly declared. If the function definition is located in the same file, then this will both declare and define the function.

JewelScript uses a two-pass compilation scheme to eliminate the need to forward declare functions in a file, a problem that can occur in C or C++, if a function is called that is located later in the file, below the function that is currently compiled:

```
function main()
{
    foo();   // call foo
}

function foo()
{
}
```

In C / C++ the above example would be a problem, unless we have declared function 'foo' before using it in function main. This problem is the reason why we need include files in C / C++ and place all function declarations there. The two-pass compilation scheme in JewelScript makes forward declarations (and include files) unnecessary.

In the first pass, the compiler will declare all functions in a file, no matter if they appear in the form of a function declaration or definition statement. The second pass will compile the bodies of all function definition statements. That way, all functions in a file are always known before any code is compiled.

Note that the two-pass compilation scheme is applied on a per-file basis. That means, if the code of a program is modularized across multiple files, then it might be necessary to explicitly forward declare a function from another file, before it can be used. This depends on the order in which files get compiled.

The compiler API function JCLCompile() treats each portion of source code passed to it for compilation as an individual file. So in order to utilize two-pass compilation to the maximum, developers could read all their script files into one big string and have JCLCompile() compile that, instead of calling it individually for each script file.

Function call

A function call can be used to call a global function, an instance member function (method), or a delegate. In general, function calls are pretty much the same as in any other C-style language.

This section will explain how the language will resolve a function call. It will also explain how the language determines which function to call when multiple functions have the same name, and the conflicts this can produce.

When the compiler recognizes a function call expression, it tries to look up the function's declaration by the specified function name. This is done as follows:

If the function call occurs from a member function of a class, the compiler will search the class for a member function of the specified name. If the function call occurs from the global scope, then this step is skipped.

If the function was not found, the compiler will search the global scope for a function with the specified name.

If the function was again not found, the compiler will search in all classes specified by the using-statement for a *global* member function of the specified name. If this fails too, then this will produce a compile-time error.

When multiple functions with the same name exist, the number of arguments will determine the function to call.

```
function foo(int);
function foo(int, int);

foo( 10 );        // calls foo(int)
foo( 27, 10 );    // calls foo(int, int)
```

If multiple functions with the same name and number of arguments exist, the types of the specified expressions will determine the function to call.

```
function foo(int);
function foo(float);
```

```
foo( 10 );        // calls foo(int)
foo( 10.5 );      // calls foo(float)
```

In general, the compiler will always choose the variant of the function that will require *the least* amount of type conversions in order to call the function. This allows developers to 'specialize' a method or function that, for some of the calls in a program, would require a costly type-conversion of an argument, if a variant can be implemented that accepts the type directly and allows to do the job more efficiently.

If an argument passed to the function call is *type-less*, this can lead to more than one variant of the function to match the call. Since the compiler then does not know which variant the developer intended to call, an ambiguous function call error is produced.

```
var v = 10;
foo( v );         // error: both foo() variants match!
```

If two functions have the same name and number of arguments, and all argument types are the same, then the expected result type (the left-hand value's type) will determine the function to call.

```
function int foo(int);
function string foo(int);

int a = foo( 10 );    // calls int foo(int)
string b = foo( 10 ); // calls string foo(int)
```

Note that there are limits to resolving calls to functions that only differ by return type. If the left-hand value's type is *type-less* or no left-hand value is specified at all, then the compiler will produce an ambiguous function call error, since the compiler cannot really know which variant of the function the developer wanted to call:

```
function int foo(int);
function string foo(int);

var a = foo( 10 );    // ambiguous function call
foo( 10 );            // same
```

Resolving ambiguous function calls

In certain cases, the language allows developers to clarify, which variant of a function should be called by specifying the full-qualified name when the function is called.

For example, if two functions with the same name and signatures exist in different class namespaces, this can produce a conflict when both of the classes have been mapped to the global namespace by the *using* statement.

When we try to call one of the conflicting functions, the compiler will detect the name conflict and issue a compile-time error. However, we can use the scope operator to resolve such a conflict, explicitly telling the compiler which function we want to call.

If a class name, followed by the scope operator, has been specified to a function call, then the compiler will only search in the declaration of the specified class for the specified function.

If only the scope operator, but no class name has been specified to a function call, then the compiler will only search in the global scope for the specified function.

```
Foo::Bar(27);     // explicitly call 'Bar' from class Foo
::Bar(27);        // explicitly call 'Bar' from the global scope
```

See also →*using* keyword.

Co-functions

As previously mentioned, co-functions are JewelScript's implementation of the concept of *co-routines*. Their purpose is to run script code in multiple cooperative threads.

There is no "scheduler" or any other means of automatic thread switching in the runtime. Instead, script code must be written to explicitly resume and suspend thread execution, hence the word *cooperative* multi-threading.

Of course this makes co-function threads unsuitable to execute different threads of script code "simultaneously" and in parallel. But that's not the purpose of co-functions anyway.

Co-functions can be extremely useful in cases where we need to implement complex behaviour in an environment where we cannot stay in an infinite loop, but must eventually return.

Application event handlers are usually an example for this kind of environment. In such an event handler, we are supposed to respond to the event and then return. We cannot stay in the event handler infinitely, since it would block the application's thread and it wouldn't be able to respond to other events any more.

In C++, the only way to implement complex behaviour in such an event handler would be to create a state machine. We would remember the state our event handler had after the last call in a set of member variables and use a switch to decide what step to take when our event handler gets called next time.

```
method OnClick(int x, int y)
{
    switch(m_CurrentState)
    {
        case kCollapsed:
            Expand();
            m_CurrentState = kExpanded;
            break;
        case kExpanded:
            SelectItem();
            m_CurrentState = kSelected;
            break;
        case kSelected:
            Collapse();
            m_CurrentState = kCollapsed;
            break;
    }
}
```

The above example illustrates an application control's *OnClick* event handler. The behaviour of the control is defined as follows: If the control is currently collapsed, a click will expand it to show a list of options. If the list is shown, a click will select an item from the list. If the control is clicked again, the control is collapsed.

A behaviour like that would be much easier implemented, if we could simply do it in a loop, where we perform one step of the procedure each time the event handler is called.

With co-functions we can do exactly that. They allow us to implement state machines in a loop, rather than a switch.

Definition

Defining a co-function is very similar to defining a regular function. Like normal functions, co-functions are *global functions* that can be defined at global scope or in the scope of a class. Syntactically, there are only two differences to regular functions worth mentioning:

First, co-functions are defined using the *cofunction* keyword instead of the function keyword.

Second, the return-statement cannot be used directly in a co-function's body, because technically, a co-function never returns. The co-function's thread remains valid and can be resumed for as long as any thread variable keeps a reference to it, thus a return-statement cannot end a thread like one would probably expect. Instead, the *yield*-statement can be used to suspend the co-function's thread execution and return a value to the parent thread.

Apart from these syntactical differences it is important to know that a co-function, as opposed to a regular function, cannot be directly called. This is because a co-function needs a *thread context* to run within. A thread context contains all the fundamental variables required by the virtual machine in order to execute instructions, such as a stack, registers, program counter, and so on.

Consequently, a co-function is not only a piece of code, it is also an object. Therefore declaring a co-function also declares a new type, which can then be used to create an instance of the co-function.

Instantiation

A co-function instantiation expression looks very much like a regular class instantiation. The only difference is that it returns a reference to a thread object, which can then be stored in a variable of the co-function's type. We can then use that variable to resume the thread, which actually will execute the co-function's code.

Here is a very simple example that illustrates co-function instantiation:

```
cofunction string Hello() // a simple cofunction
{
    yield "Hello World!";
    yield "How are you?";
    yield "Bye!";
}

Hello hello = new Hello(); // instantiate into a global variable
```

It is important to understand that the co-function instantiation expression does *not* start to execute the co-function's code. It only allocates and initializes a new thread context and sets it's program counter to the beginning of the specified co-function. To start execution, we must resume the thread.

Resuming a co-function

Resuming a co-function is quite simple. We just 'call' the thread variable by appending empty parentheses to the variable name.

```
function main()
{
    print( hello() ); // Prints "Hello World!"
    print( hello() ); // Prints "How are you?"
    print( hello() ); // Prints "Bye!"
}
```

It is not possible to specify any arguments to this 'call' expression though, because our variable refers to a different thread that has it's own stack. We cannot put new arguments onto a foreign thread's stack when it is already running – this would just mess up the stack and cause a crash.

A co-function can be resumed from anywhere where access to the thread variable is given – a global function, a method or another co-function. It is even possible to recursively instantiate and resume a co-function. But one should be aware that instantiating a new context for every recursion can consume a lot of memory. Roughly, one can calculate the thread context size in bytes by taking 8 times the stack size used when initializing the runtime, plus two additional K for good measure.

When resuming a co-function's thread, the thread will keep executing until it encounters a yield-statement. Hence, the resume expression will not return until the next yield-statement in the resumed thread. When the co-function's thread is resumed again, execution will continue after the yield-statement, resuming where the thread was left the last time.

As we can see in the example, this leads to the first resume returning the string "Hello World!", while the second will return "How are you?", and the third "Bye!".

If the co-function has fully run through and an unconditional yield-statement specifying a return value was encountered, resuming the co-function will immediately and infinitely return the last return value (without re-evaluating the yield-statement's expression). Since this is the case in our example, continuing to resume hello() in function main would always return "Bye!".

If no unconditional yield-statement was encountered before it's end, resuming the co-function will immediately and infinitely return 'null'.

Performance considerations

This article has been moved to the developer blog at:

http://blog.jewe.org/?p=1549

Classes

JewelScript supports classes written in both script code and native C/C++ code.

They can be derived from an interface in order to have multiple classes implement the same set of functionality, or to ensure that a class implements all methods expected by an application that embeds the runtime.

Next to member variables and member functions (methods), the language allows to define global functions and global constants as class members. Other valid member types are co-functions, delegates, aliases, member interfaces and member classes.

This section discusses using JewelScript classes in more detail.

Forward declaration

Sometimes it can become necessary to *forward declare* a class name. Forward declaring means making the name of a class known to the language, without having to fully declare the class, and thus without having to declare all of it's class members.

The following statement forward declares the class *Foo*:

```
class Foo;
```

Forward declaring a class can become necessary, if we need to use the type name before being able to fully declare the class:

```
class Foo;        // forward declare 'Foo'
class Bar         // declare 'Bar'
{
    Foo m_Foo;
}
class Foo         // declare 'Foo'
{
    Bar m_Bar;
}
```

In the above example, both class declarations depend on each other. In order to be able to declare one of them, we need to forward declare the other.

The language will allow multiple forward declarations of the same class. It is also allowed to forward declare a class, but never fully declare and define it, as long as the class is never used.

When forward declaring a class, everything except the name of the class is still unknown. Trying to access any member variable or function of such a class will produce a compile-time error. This includes trying to instantiate the class, because it implies access to a constructor.

Declaration

In order to instantiate a class and access any members of it, a full class declaration statement is required. The purpose of a class declaration statement is to make the class name and all it's properties known to the language. We can declare global functions, global constants, instance member functions (methods), instance member variables and co-functions in classes.

As an example, a class declaration statement could look like this:

```
class Foo
{
    method Foo();
    method string GetName();
    method SetName(const string);
    string m_Name;
}
```

This example shows a complete class declaration of a class called 'Foo'. It has a single constructor taking no arguments (the *default constructor*) and the methods (instance member functions) GetName() and SetName(). Furthermore, it has one member variable, a string named 'm_Name'.

Definition

Defining a class means specifying the actual code of it's constructor(s) and, if present, any member functions. There are several options how to define a class in JewelScript. One of them is to define the class *inline*, which means directly in the class declaration statement:

```
class Foo
{
     method Foo() { m_Name = "Unnamed"; }
     method string GetName() { return m_Name; }
     method SetName(const string name) { m_Name = name; }
     string m_Name;
}
```

For small or medium sized classes, this form of declaring and defining a class in one statement can be convenient. However, it tends to make the code harder to read if the class gets bigger. Therefore, the language also allows to define methods outside of the class declaration, in a way similar to C++:

```
class Foo                                       // declaration
{
     method Foo();
     method string GetName();
     method SetName(const string);
     string m_Name;
}
method Foo::Foo()                               // defining the constructor
{
     m_Name = "Unnamed";
}
method string Foo::GetName()                    // defining GetName()
{
     return m_Name;
}
method Foo::SetName(const string name)          // defining SetName()
{
     m_Name = name;
}
```

Of course it is possible to mix these two styles of defining class member functions. Part of it can be put into the declaration (especially useful for "one-liners"), while the rest of it can be defined outside of the class declaration.

Constructors

Constructors are class member functions that are used by the language for a specific purpose, namely to initialize a new instance of a class. Whenever we create an instance of a class, the language will call the appropriate constructor in order to initialize the newly created object.

A constructor in JewelScript is a regular method, except it cannot return a value and it's name is always the name of the class. JewelScript allows to define any amount of constructors to a class.

If a class contains only global functions and no member variables, no constructor needs to be defined. If member variables or methods are present in the class, the language will enforce that at least one constructor is defined, though.

When compiling a class, JewelScript will verify that each constructor specified to a class initializes all instance member variables declared in the class. This is done to ensure that the program is not being run with objects that contain uninitialized variables. The constructor must initialize the member variables *directly*, meaning not by calling another method or function that does so.

Constructors also play an important role in automatic type conversion. If a class has a matching constructor, the language can automatically convert any value to the type of that class by constructing a new instance. For more information on constructors and automatic type conversion, see →*Automatic type conversion*.

Default constructor

A default constructor is a special type of constructor that needs no additional arguments in order to be invoked.

When declaring local or global variables of a class, the language will try to automatically create an instance of that class and call it's default constructor, if the variable has not been explicitly initialized by the developer.

```
class Foo
{
    method Foo()      // default constructor
    {
        m_Str = "Hello!";
    }
    string m_Str;
}

Foo f;                // implicitly create instance by calling default ctor
Foo g = new Foo();    // explicitly call default ctor
```

JewelScript does not automatically define a default constructor. If the developer does not specify one, the class doesn't have one.

Copy constructor

A copy constructor is a special type of constructor that takes a reference to an instance of the same class as the only argument. It's purpose is to initialize a new instance of a class with the values copied from another instance of that class. This allows the class to define how it's instances should be copied.

If a class written in script code does not define a copy-constructor, the language will automatically copy the object in cases where a copy is required. In that case member variables of the copy will be initialized with references to the members of the source object.

If copying references is unwanted, the class must implement a copy-constructor and create real copies of the source object's members.

Defining a copy-constructor may have an impact on performance, because the runtime will invoke the copy-constructor in **all cases** where the script object is copied. So with complex structures of many objects, this can become a time consuming recursive operation.

If a native class does not define a copy-constructor, then it means that copying the object is forbidden. In this case the compiler will issue a compile-time error if code is produced that requires the object to be copied.

```
class Foo
{
    method Foo()
    {
        m_Str = "Hello!";
    }
    method Foo(const Foo src)    // copy constructor
    {
        m_Str = src.m_Str;       // copies m_Str since it is 'const'
    }
    string m_Str;
}

Foo f;
const Foo k;
Foo g = new Foo(f);   // explicitly call copy ctor
Foo h = k;            // implicitly call copy ctor
```

Instance member functions (methods)

Every instance of a class basically consists of two parts: Code and data. The code part is represented in the form of instance member functions, or methods as they are called in JewelScript.

Methods are declared / defined by using the *method* keyword. In general, everything said in the →*Functions* section of this document is also true for methods.

The only noteworthy thing to add about methods is that the language provides a reference to the instance of the class through the *this* reference. We can use the *this* reference to access member variables or methods, or to pass a reference to our instance to other functions that are not methods in our class.

```
method Foo::SetNumber(int number)
{
     this.number = number;
     Update( this );
}
function Update(Foo f)
{
     print( "number=" + f.number );
}
```

Convertor methods

Convertor methods are a special kind of method that can be used to implement a conversion of a class instance to another type. Like constructors, they play an important role in automatic type conversion.

If a class implements a convertor method for a specific type, the language will automatically call this method to convert an instance to that type.

```
class Color
{
     method Color(int r, int g, int b)
     {
          this.r = r;
          this.g = g;
          this.b = b;
     }
     method int convertor()       // pack RGB color into an int
     {
          return r << 16 | g << 8 | b;
     }
     int r, g, b;
}

Color pink = new Color(255, 0, 255);
int colorkey = pink;  // convert 'pink' to int
```

A valid convertor method must return a value and must not accept any arguments, otherwise a compile-time error will occur:

```
'method' typename 'convertor' '(' ')'
```

Note that the language allows to declare a convertor to *type-less,* even though this doesn't make much sense, since the compiler will never perform automatic type conversion if the left hand value is type-less.

```
method var convertor();    // conversion to type-less never called!
```

For more information on convertor methods, see →*Automatic type conversion*.

Accessor methods

A common practice to control access to member variables in C++ is to declare member variables as 'protected' or 'private' to disallow direct access to these variables from the outside of the class. Then usually accessor methods are implemented, usually a *getter* and a *setter* method, to allow other classes access to the variable in a controlled manner.

JewelScript employs the *accessor* keyword in order to allow developers to restrict and control access to member variables.

By using this keyword, it is possible to declare a special kind of method, that can be either used to read a class property, or write it.

The difference to C++ is however, that using accessors does not change the semantics of the code. In other words, we can easily "redirect" direct access to a member variable to a method call, without having to change all the code lines that formerly accessed the variable. Here is an example:

```
class Foo
{
    method Foo(int v)
    {
        value = v;
    }
    accessor int value()  // 'get' accessor for value
    {
        return value;
    }
    accessor value(int v) // 'set' accessor for value;
    {
        value = v;
    }
    int value;            // our member variable
}

Foo f = new Foo( 27 );    // create an instance of class Foo
print( f.value );         // get value
f.value = 12;             // set value
```

In this example, we define a simple class *Foo* that has an integer number as a member variable. To prevent developers from accessing *value* directly, we define the two accessor methods that have the same name as the variable – and thus hide the member variable from the outside of the class.

By doing this, we instruct the language to 'redirect' any accesses to the variable to either the get accessor method, or the set accessor method. Whenever the variable should be read, the language will compile a call to the get accessor. Whenever it should be written, it will compile a call to the set accessor.

This is not only done for simple assignments like in the above example, but for all operators. For example, if we use the operator += to add to *value*, the language will call the get accessor, perform the add, and call the set accessor.

Restricting access

Inside the accessor methods, we have full control over how the variable is read or written. However, in certain cases it might be more useful to disallow reading or setting of a member variable completely.

We can achieve this in JewelScript as well. For example, to only allow reading of class properties, but disallow writing them, we can implement a read accessor, but choose to not implement a write accessor.

As soon as a get accessor has been declared, the language will interpret the lack of a matching set accessor as our wish to protect the class property from being written. Trying to assign or modify the member variable from outside of the class will then result in a compile-time error.

Likewise, if only a set accessor has been declared, it will interpret the lack of a matching get accessor as our wish to protect the class property from being read. Trying to read the member variable from outside of the class will again result in a compile-time error.

The accessor methods are only called when the property is accessed from outside of the class. If a member function of the class itself accesses the property, this will be compiled into a regular variable access and not a call to the set or get accessor.

If we need to call the set or get accessor from within the class itself, we can use a regular function call expression.

```
method Foo::AddTen()
{
     int i = value();
     i += 10;
     value( i );
}
```

Abstract class properties

Accessor methods can not only be used to hide member variables from being accessed from the outside of the class. It is not mandatory to use the name of an existing member variable for the name of an accessor method, any name is valid.

This lets us add abstract properties to the class, properties that are not simple variables and have no real storage location in the class. This is especially useful in native types. For example, the *length* property of the built-in string class is a read accessor.

Furthermore, since accessors are methods, they can also be declared in interfaces. This lets us define class properties as a part of a common interface for deriving classes: Every class implementing our interface would need to implement these properties.

Prototyping

Accessor methods are restricted to two specific prototypes.

A set accessor must not return a value, and must receive a single argument:

```
'accessor' identifier '(' typename identifier ')'
```

A get accessor must return a value, and must not receive any arguments:

```
'accessor' typename identifier '(' ')'
```

Static member functions

Static member functions are global functions that are member of a class and thus are protected by the class namespace. We can declare a global function to a class by using the *function* keyword.

Since these functions are not methods, they don't require an instance of their class in order to be called. For the same reason, accessing instance members and using the *this* reference is not possible in global class member functions.

```
class Foo
{
     method Foo()
     {
          m_Val = 0;
     }
     function Foo Create() // static member function
     {
          return new Foo();
     }
     int m_Val;
}

Foo f = Foo::Create();     // create an instance
```

By default, JewelScript requires developers to specify the *full qualified function name* when calling a static class member function, because these functions are protected by the class namespace.

We could have many classes that implement a function with the name 'Create'. To guarantee that no name conflicts occur and the language always knows which function to call, we need to use the full qualified name.

However, with long class- and function names, specifying the full qualified function name can become quite tedious. Therefore it is possible to have the language assume certain class namespaces by default, by specifying them to the →*using* statement.

```
using Foo;   // by default, we use global functions from class Foo

Foo f = Create();
```

Instance member variables

As already said, every instance of a class basically consists of two parts: Code and data. Instance member variables (usually only called member variables) make up the data part of an instance.

Since member variables have already been discussed to a certain degree in the →*Variables* section of this document, this section will only focus on the differences to local and global variables.

Declaration

A variable is a member variable, if it is declared within a class declaration statement. The scope (visibility) and life time of a member variable is bound to an instance of the class. This means that the member variable is visible whenever we have access to an instance of the class, and it exists for as long as the instance of the class exists.

```
class Foo
{
    string myMember;  // declare a member variable
}
```

If our class declares many member variables and methods, their position is of no importance. Though it is considered good programming style to group all method declarations together at the beginning, and all member variables at the end of the class declaration.

Initialization

While local and global variables can be initialized directly in the declaration statement, this is not possible with member variables.

```
class Foo
{
    string myMember = "Hello";   // this does not work
}
```

To initialize a member variable, we have to define a constructor for our class.

```
class Foo
{
    string myMember;      // declare member variable
    method Foo()          // initialize it in our constructor
    {
        myMember = "Hello";
    }
}
```

Access

Member variables can only be accessed through a reference to an instance. While executing a method of the class, the reference to the instance is implied. This means the language will look up variables and methods in our class automatically as we use them, we don't have to explicitly specify a reference to our class instance. However, we can use the *this* reference to resolve name conflicts between a member variable and a local or global variable.

```
class Point
{
    method Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
    int x, y;
}
```

JewelScript generally allows all class member variables to be read and written from outside of the class, as long as we have a reference to an instance of the class.

```
Point p = new Point(0, 0);
p.x = 17;
p.y = 19;
```

It is possible to 'hide' a member variable from outside of the class by declaring an accessor method with the variable's name.

Static member constants

By convention, the language treats constants declared inside a class as *globals*. Like normal global variables and constants outside of a class, those declared inside of a class have to be initialized immediately in their declaration statement.

```
class Color
{
    method Color(int r, int g, int b)
    {
        this.r = r;
        this.g = g;
        this.b = b;
    }
    int r, g, b;

    const Color kBlack = new Color(0, 0, 0);
    const Color kWhite = new Color(255, 255, 255);
}

SetBackColor( Color::kWhite );
SetFontColor( Color::kBlack );
```

JewelScript 1.0.3.13 extended the argument-scope when calling a member function of a class. The compiler will first try to find a given constant in the class that is being called, then in the class that is performing the call, and finally in the global space.

```
class CharColor
{
    const int Red = 0;
    const int Blue = 1;
    const int Green = 2;

    method CharColor(int foreground, int background)
    {
        Foreground = foreground;
        Background = background;
    }
    int Foreground;
    int Background;
}

function main()
{
    CharColor c = new CharColor(Red, Black);
}
```

As of Version 1.2, the *using* statement also provides easy access to global constants declared in classes.

Class instantiation

Instantiating a class means creating an instance of a class by allocating memory for its member variables, and initializing them by calling one of the class constructors. In general, there are three ways to instantiate a class in JewelScript:

1. Explicit instantiation by →*Operator new*

2. Implicit instantiation by automatic variable initialization
3. Implicit instantiation by automatic type conversion

Here is an example that illustrates these three ways:

```
class Foo
{
    method Foo()
    {
        m_Val = 0;
    }
    method Foo(int v)
    {
        m_Val = v;
    }
    int m_Val;
}

Foo f1 = new Foo(12); // 1. explicit instantiation
Foo f2;               // 2. instantiation by auto initialization
Foo f3 = 27;          // 3. instantiation by auto type conversion
```

Automatic type conversion

A language based on static typing would be next to useless if there was no way to convert values of one type to another type. Especially when the language supports the definition of classes, the developer must be able to define a way to convert values of one type to another.

When assigning a value to a variable of a type that is not compatible to the value, JewelScript can automatically convert the value to the type of the variable, if it knows of a suitable conversion method.

In the case of the primitive data types *int* and *float*, the language has built-in functionality to convert between these types.

In the case of classes, the developer can define a custom way of converting any type of value to an instance of the class, and an instance of the class to any type of value. JewelScript does this by automatically calling a matching class constructor, respectively a matching convertor method.

The language will try to perform automatic type conversion, if the operand-type left from an assignment is incompatible with the operand-type right from the assignment. This will be done as follows:

If the source operand (the r-value) of the assignment is of a class type, the language will try to find a convertor method in the source operand's class that returns the destination operand's type (the l-value's type).

If the source operand is not a class type, or it doesn't define a suitable convertor method, the language will try to find a constructor in the destination operand's class that allows to construct a new object from the source operand's type.

If no appropriate constructor is available, the types are not convertible and the language will produce a compile-time error.

This does not only apply to assignments that use the = operator. In fact, when passing an argument to a function call, or specifying an argument to the *return* statement, this is an assignment as well, and thus auto conversion will be performed if necessary.

Conversion by convertor methods

In order to define a custom conversion to another type, we can add a convertor method to our class declaration. Lets say we have defined a *Color* class that allows us to store RGB colors. A very basic example could look like this:

```
class Color
{
    method Color(int r, int g, int b)  // constructor
    {
        this.r = r;
        this.g = g;
        this.b = b;
    }
    int r, g, b;
}
```

This basic class serves our goal to store RGB values. However, colors are represented in many different formats, and certain functions we want to call might expect them in a different format. Some functions might expect the three channels packed into a single integer value, for example.

```
function SetColorKey( int color );
```

So in order to call these functions, we would have to manually pack the red, green and blue components into an integer before calling the function.

```
Color c = new Color(255, 0, 255);       // create a RGB color object
int ck = c.r << 16 | c.g << 8 | c.b;    // pack into an int
SetColorKey( ck );                      // set color key
```

While this works, it can become really tedious if our program is complex and we have to call SetColorKey() or other similar functions very often. A better solution would be to define the packing of the color channels as a conversion and implement a convertor method for it.

```
class Color
{
    method Color(int r, int g, int b)  // constructor
    {
        this.r = r;
        this.g = g;
        this.b = b;
    }
    method int convertor()             // convert to integer
    {
        return r << 16 | g << 8 | b;
    }
    int r, g, b;
}
```

Now the language can automatically convert our RGB color to an integer whenever a conversion is needed.

```
Color c = new Color(255, 0, 255);       // create a RGB color object
SetColorKey( c );                       // set color key
```

Conversion by construction

Now we can convert our RGB value into a packed integer. However, it would also be useful to convert a packed integer into an RGB value. Some functions might return colors in packed integer format.

```
function int GetColorKey();
```

By adding a constructor to the class that allows JewelScript to automatically create an instance from an integer, we can achieve this.

```
class Color
{
    method Color(int r, int g, int b) { ... } // constructor
    method int convertor() { ... }            // convert to integer

    method Color(int color)                   // construct from integer
    {
        this.r = (color >> 16) & 255;
        this.g = (color >> 8 ) & 255;
        this.b = color & 255;
    }
    int r, g, b;
}
```

Now we can conveniently convert a color packed into an integer value to an object of our Color class.

```
Color c = GetColorKey();
```

Keyword *explicit*

Sometimes it might be unwanted that certain constructors or convertor methods are automatically called. For example, some constructors or convertor methods may cause a performance hit, or can potentially fail. In these cases it might be a good idea to have the developer explicitly acknowledge that a conversion is wanted.

The keyword *explicit* can be used to declare a constructor or convertor method that should **never** be called by automatic type conversion. Instead, the developer must explicitly acknowledge that a conversion is wanted by using the →*Typecast operator*.

The language will notify the developer with a compile-time error if a statement is using a conversion that requires a typecast operator.

As an example, the built-in *string* class has convertor methods to integer and floating point values that are declared explicit. This has been done to ensure that developers are notified when they create code that might unintentionally convert numerical values to strings and back, which can cause a significant performance hit and loss of precision.

```
class string
{
    method string();
    method string(int i);
    method string(float f);
    explicit int convertor();
    explicit float convertor();
}
```

Member types

It is possible to create member types inside a class, which are protected by the namespace of the class.

When declaring a co-function or delegate inside a class, this actually creates a type that is local to that class.

```
class Foo
{
    delegate int processor(int chr);   // declare local type 'processor'

    method Process(processor fn);      // use local type processor
}
```

The full qualified name of the *processor* delegate above is *Foo::processor*. This is important, so multiple classes can define a delegate named *processor* without producing name conflicts.

However, we don't need to use the full qualified type name when using the type inside of class Foo. Instead, it is sufficient to simply use the type name *processor*. The compiler will look up type names locally in the class first, before searching for them globally.

In addition, JewelScript 1.2 added the possibility to have member classes and interfaces in a class. As described above, the compiler will look up these member types locally in the class first, before searching for them in all parent namespaces and the global scope.

As of Version 1.2, it is also possible to use local type names from outside of the class directly, by specifying their full qualified names.

```
class Foo
{
     class Bar
     {
          method Bar()
          {
          }
     }

     method Foo()
     {
          bar = new Bar();
     }
     Bar bar;
}

function main()
{
     Foo::Bar bar = new Foo::Bar();
}
```

Member aliases

Aliases can be defined in the namespace of a class. This allows us to *reuse* local type names from other classes locally in our class. Like local type names, local aliases will be defined using their full qualified names.

```
class NewFoo
{
     alias Foo::processor processor;     // results in NewFoo::processor

     method Process(processor fn);       // use NewFoo::processor
}
```

See also: →*alias* and →*using.*

Interfaces

JewelScript supports the declaration of interfaces and implementing them in classes. An interface in JewelScript is always *abstract*, meaning it can only consist of method declarations. It cannot contain static member functions or member variables, and methods cannot be implemented directly in the interface. Instead, they can only be implemented by having a class implement the interface.

When a class implements an interface, all methods from that interface must be implemented in the class. If a method derived from an interface is not implemented, the language will create a default implementation for that method, or issue a compile-time error if the interface was declared using the *strict* modifier keyword. (See →*Strict functions, classes and interfaces*)

The purpose of using interfaces in JewelScript is to ensure that script programmers implement all the methods in their script class that the native application expects it to have. Having script programmers implement an interface is also a good way for the native application to ensure that the *order* of the methods in an object is as assumed by the application.

```
interface Control
{
    method   Control(NativeControl nc);
    method   OnClick(int x, int y);
    method   OnKey(int key);
    method   OnIdle();
}
```

The above statement declares an interface *Control* which gets passed a reference to a native control object in it's constructor and declares the event handler methods *OnClick*, *OnKey* and *OnIdle*.

By having script programmers implement this interface, our native application allows them to implement additional GUI controls in JewelScript.

The interface is a way for the application to make three important things sure:

- All event handler methods that our native application wants to call are actually present in the classes that script developers create. This saves the application from having to check that the class implementation is complete.
- All event handler methods have been implemented *correctly*, meaning the argument list and result type of the methods are as the native application expects them to be.
- The *order* in which methods are implemented is fixed (because it's defined by the interface), so the native application can rely on fixed method indexes. For example, *OnKey* is always at index 2 in the object.

To summarize, the benefit of using an interface is that the native application doesn't have to spend any effort to check that a script object correctly interfaces with it. By declaring an interface the application can leave that checking entirely up to the JewelScript compiler.

Finally, interfaces can not only be implemented by script classes. It is also possible to have native types implement them, so native objects and script objects can be handled by the same code, because they share a common base class interface.

Up and down cast of references

When a class implements an interface, references to instances of that class can be passed around using the type name of the interface. This allows to pass references to different types of objects to functions, as long as they share a common base class interface.

```
class TextControl : Control { ... }
class Button : Control { ... }
function AddControl(Control cntl) { ... }

AddControl( new Button() );
AddControl( new TextControl() );
```

Converting a reference to a class to it's interface type is called up-casting because the reference is converted from the sub-class up to the super-class (which can only be an interface in JewelScript).

References to objects can always be implicitly converted to their interface type in JewelScript, there is no additional code required for the up-cast, and it is completely safe to do so.

However, sometimes it is necessary to reverse this conversion, and cast the reference back down to it's actual class type, in order to access methods or member variables that are not defined as part of the common interface.

```
function SetText(Control cntl, string msg)
{
    cntl.TextControl::SetText(msg); // call method not in interface
}
```

This type of conversion is critical, because the language has no knowledge if the value in the variable is really of the type we try to cast to. Of course many classes can implement the Control interface, and to stick to our example, there's no guarantee the caller of this function really has passed a TextControl instance to the function, it could also be a Button.

Therefore the compiler will generate a special instruction that verifies during runtime, that the type of the object in 'cntl' really is a TextControl. If it is not, a type mismatch exception will occur that aborts script code execution, to prevent the following code from causing a crash.

Down-casting an interface reference to access member variables works the same way. This too, will generate the instruction to ensure the correct type of “cntl” at runtime.

```
cntl.TextControl::m_Text = "Test";
```

Starting with JewelScript 1.0.3.13, it is also possible to use the →*Typecast operator* to cast down a reference to it’s actual type. To ensure type-safety, this will also generate the instruction to ensure the correct type of the value at runtime.

```
function AddButton(Control btn)
{
    Button button = (Button)btn;
}
```

Strict functions, classes and interfaces

By default, when a function was declared, but never was implemented, JewelScript will detect this during the linking phase and notify the user with a warning. After that, the language will create a *default function implementation,* just in case the native application is going to call the function.

For functions that declare a result, the default function implementation will return null. For functions that do not declare a result, the default implementation simply returns.

Script programmers can utilize this behaviour when implementing an interface. It allows them to only implement those methods defined in the interface, that need implementation, and omit implementation of those that are not needed, leaving the generation of a default implementation to the compiler.

However, there may be cases where a default implementation of a method is not sufficient. Therefore it is possible to declare functions and methods using the strict modifier keyword.

```
strict function int MyFunc(int);
```

When a function or method has been declared strict, it means that generating a default function implementation for this function or method is not allowed. Consequently, the compiler will issue a compile-time error instead of generating the default implementation.

It is also possible to use the strict modifier keyword when declaring interfaces and classes. When making an interface strict, it means that all of the methods declared in the interface are strict. If a class implements the interface, it inherits the strict modifier from the interface methods, but any additional methods in that class are not affected.

```
strict interface IPerson
{
    method string GetName();
    method SetName(string);
}

class Person : IPerson
{
    method Person();            // constructors always strict
    method string GetName();    // inherited strict
    method SetName(string);     // inherited strict
    method int GetAge();        // not strict
    method SetAge(int);         // not strict
}
```

This example shows a class implementing a strict interface. If none of the methods declared in this class were implemented, the compiler would default implement the GetAge() and SetAge() methods, but would issue an error for the constructor, GetName() and SetName().

JewelScript never default implements class constructors, they are always implicitly considered 'strict'. This is because class constructors are supposed to initialize member variables, so a default implementation for them would not be sufficient.

If we want to make all of the functions in the Person class strict, we need to declare the Person class strict as well.

Delegates

Delegates are references to functions and methods in JewelScript. References can be taken from any function, regardless whether it is a global or instance member function, and regardless whether the class is implemented in script or native code.

Delegates can be assigned to variables, passed to functions as arguments or returned by functions as result. They can be put into lists or arrays as well. In short, delegates can be used like any other value in the language.

To ensure type safety, delegates have to be declared using the *delegate* keyword, which will create a new delegate type. A delegate type is a type that simply describes the signature of a function or method. It specifies the result type, number of arguments and argument types.

```
delegate int Comparator(string, string);
```

This statement declares a delegate type for functions and methods that return an int and take two string objects as arguments. Using this new type we can now declare a variable and assign a function or method to it that matches the delegate's signature.

```
function int CompareLengths(string a, string b)
{
    return a.length == b.length;
}

function Test(string s, string t, Comparator func)
{
    if( func(s, t) )
    {
        print("The comparator returned true");
    }
}

function Main1()
{
    Test("Hello", "World", CompareLengths);   // pass function to Test()
}
```

When taking a reference from an instance member function, the language will store both the reference to the function *and* the reference to the object in the delegate variable.

```
class Foo
{
    method int Compare(string a, string b) { ... }
}

function Main2()
{
    Foo foo = new Foo();
    Test("Hello", "World", foo.Compare);      // pass foo method to Test()
}
```

Delegate arrays

Since declaring a delegate creates a regular type, it is possible to create arrays of delegates. This can be useful for random access to functions (“jump table”) or a function stack. We can also use it to maintain a list of event handler methods.

```
class Control
{
     delegate ClickFn(Point pos);
     method Control();
     method OnClick(Point pos);
     method AddListener(Control ctrl)
     {
          m_Listeners += ctrl.OnClick;   // add another listener
     }
     ClickFn[] m_Listeners;
}
```

In order to call all the event handlers in such an array, we can use the array::enumerate() method, and pass a delegate that performs the actual call to each element.

```
method Control::OnClick(Point pos)
{
    m_Listeners.enumerate(CallOnClick, pos);
}

function CallOnClick(var element, var args)
{
     ((ClickFn) element)(args);
}
```

Furthermore, delegate arrays can be a good alternative to large switch-statements, and can actually be faster than a switch.

```
const int kInit = 0;
const int kWelcome = 1;
const int kMain = 2;
const int kGameOver = 3;
const int kNextLevel = 4;

delegate GameState();

GameState[] State = { OnInit, OnWelcome, Main, GameOver, NextLevel };

int CurrentState = kInit;

function RenderFrame()
{
    State[CurrentState]();
}
```

Local functions / anonymous delegates

When using delegates a lot, having to define a delegate type and a regular function or method can become quite an effort. Sometimes, we just want to pass a piece of code as a delegate to a function, without having to create a function for the piece of code, because we are certain that we don't need the code anywhere else.

For these cases, JewelScript allows us to pass an anonymous function or method as a literal to a function, or assign it to a variable. To make this as easy as possible, the language will derive the function's signature from the left-hand type of the assignment.

```
delegate ClickFn(Point pos);

ClickFn clickHandler = function
{
    print("Click at " + pos.x + ", " + pos.y + "\n");
};
```

In this example we define a variable *clickHandler,* and assign a reference to an anonymous function. The language will determine the signature of the function from the value left from the assignment, which is a delegate of type *ClickFn*, which defines no result and an argument of type *Point*.

Note, that in this case it is mandatory to specify names for the arguments in the delegate declaration, but it is optional if we don't use the delegate for anonymous functions.

Of course this method of determining the signature of the function by looking at the left hand value, implies that the left hand value's type must be a delegate. We cannot use automatic type conversion when assigning anonymous functions, and no type-less variables can be used left from the assignment.

Furthermore, note the semicolon at the end of the assignment statement. Unlike 'regular' named functions and methods, which are (block-) statements that don't require a closing semicolon, an anonymous function is actually an *expression* and thus the assignment statement requires us to complete it with a semicolon, just like any other assignment.

Anonymous delegates are mostly useful to pass small pieces of code to functions that expect a delegate as an argument, code that is so small, that defining a new function for it isn't really worth the effort. Like for instance, the *CallOnClick()* function shown in the example above.

```
method Control::OnClick(Point pos)
{
    m_Listeners.enumerate(function { ((ClickFn) element)(args); }, pos);
}
```

Local methods

Inside instance member functions of a class, it might be preferable to use an anonymous method instead of an anonymous function, so we have access to our class instance from within the anonymous method.

```
method Control::OnClick(Point pos)
{
    m_Listeners.enumerate(method { ((ClickFn) element)(args); }, pos);
}
```

Of course this example is a bit pointless, because the anonymous method does not access any members of our Control class, but at least it shows how anonymous methods can be defined locally in a method.

Nested local functions

Since anonymous functions are expressions that can be used in functions, they can be nested.

```
delegate VoidFunction();
function main()
{
    VoidFunction subFunc = function
    {
        VoidFunction subSubFunc = function
        {
            print("This is subSubFunc()\n");
        };
        print("This is subFunc()\n");
        subSubFunc();
    };
    subFunc();
}
```

Functions as literals

Finally, anonymous functions and methods can be used as literals to initialize an array.

```
VoidFunction[] State =
{
    function { print("This is state function 1.\n"); },
    function { print("This is state function 2.\n"); },
    function { print("This is state function 3.\n"); }
};

function RenderFrame()
{
    State[CurrentState]();
}
```

Specifying argument names

In certain cases it can become necessary to specify names for the delegate's arguments. The reason for this is that all delegates which share the same signature, also share the same type-id. This is done so that delegates with matching signatures are interchangeable.

```
delegate int Comparer(int x, int y);
delegete int Verifier(int a, int b);

Comparer fn = function { return x == y; };
Verifier ve = fn; // same signatures, OK
```

The example illustrates two delegates with the same signature: *int (int, int)*. JewelScript allows us to move the value from variable *fn* to *ve*, because both delegate types are actually just aliases for the signature *int (int, int)*.

However, this can sometimes lead to the case that the compiler doesn't know the argument names of the delegate type. The compiler only memorizes argument names for the first delegate type of any given signature. As we can see in the example, the compiler has memorized 'x' and 'y' from delegate 'Comparer'.

If we would try to use arguments 'a' and 'b' from the 'Verifier' delegate, this would fail.

```
Verifier ve = function { return a != b; }; // error 'a' and 'b' unknown!
```

Because of this, we sometimes must provide explicit argument names for the delegate.

```
Verifier ve = function(i, j) { return i != j; };
```

Specifying the names is also convenient if we can't remember the argument names the delegate type was declared with.

Hybrid classes

The JewelScript compiler supports a *pseudo inheritance* from another class by automatically constructing delegates from the specified base class.

When constructing a hybrid class, the new class has access to all the global functions and instance member functions of the base class, but it is not compatible or convertible to it's base class. Hybrid classes are especially useful to integrate native functions into a class that is written in script code.

Imagine an application that allows their users to write additional GUI controls in JewelScript. To provide the script programmer with the required native functionality, the application passes a NativeControl object into the script control's constructor.

```
class Control : IControl
{
    method Control(NativeControl nc);
    method OnClick(int x, int y);
    method OnKey(int key);
    method OnDraw();

    NativeControl m_NativeControl;
}
```

This NativeControl object allows the script control to access certain important functions, for example the drawing functions DrawRect() and DrawText(). Normally, a script programmer would have to store the reference to the NativeControl object in a member variable of the class, and use the member variable to access the native functions.

```
method Control::OnDraw()
{
    int w = m_NativeControl.Width();
    int h = m_NativeControl.Height();
    m_NativeControl.DrawRect(0, 0, w, h);
    m_NativeControl.DrawText(4, 4, "Hello");
}
```

However, if there is really a lot of collaboration between the Control class and the NativeControl class, it makes sense to integrate both classes into one. This can be done by creating delegates for all functions from NativeControl in the Control class.

We could do this by manually declaring the required delegate types, adding the required delegate member variables to our class, and initializing them in our class constructor. However, we can also have the compiler do that work for us by using the *hybrid* keyword.

```
class Control : IControl hybrid NativeControl
{
    method Control(NativeControl nc) hybrid (nc);
    method OnClick(int x, int y);
    method OnKey(int key);
    method OnDraw();
}
```

By adding the term *hybrid basetype* to a class declaration, we actually instruct the compiler to automatically create the required delegates for all suitable functions defined in *basetype*. Suitable functions are all global member functions and instance member functions of the specified base class, but not constructors, convertors, accessors, or co-functions.

By adding the term *hybrid (expr)* to every constructor of the class, we specify the instance of *basetype* that the compiler should use to obtain instance member functions from. The compiler will use this instance to generate code in our constructor that initializes our delegate variables.

```
method Control::Control(NativeControl nc) hybrid (nc)
{
    // compiler constructs all delegates from 'nc'
}
```

After this, all functions and methods defined in NativeControl are integrated into the Control class, and can be used like any other class member function.

```
method Control::OnDraw()
{
    DrawRect(0, 0, Width(), Height());
    DrawText(4, 4, "Hello");
}
```

As can be seen in the examples above, a class can implement an interface and at the same time inherit a hybrid base class. Doing this can have profound consequences.

hybrid and interface – unrelated

If the interface and the hybrid base class are not related to each other, then hybrid will simply inherit all suitable methods and make the resulting class compatible to the interface.

This can be useful to create *wrapper classes* for different types of objects that share a common base class interface.

```
interface IControl
{
    method OnClick(int x, int y);
}

class TextControl : IControl { ... }
class Button : IControl { ... }

class ControlWrapper hybrid IControl
{
    method ControlWrapper(IControl cntl) hybrid (cntl) {}
}

function Test()
{
    ControlWrapper w1 = new ControlWrapper(new TextControl());
    ControlWrapper w2 = new ControlWrapper(new Button());

    w1.OnClick(10, 10);
    w2.OnClick(15, 15);
}
```

The example above illustrates two script classes *TextControl* and *Button* that share a common base class interface *IControl*. Instead of calling the script classes *OnClick* methods directly, we create a wrapper class for the interface and indirectly call the methods through the instance of the wrapper class.

Of course there is no real benefit doing that in this example, because the wrapper class does not add any extra functionality. However, we could add more methods and data members to the wrapper class to extend the original functionality of the IControl class.

Additionally, all methods in the wrapper class are actually delegates that can be selectively reassigned with a new implementation.

hybrid and interface – related

A special case arises when we implement an interface and use the same interface – or any class implementing that interface – for the hybrid declaration.

```
class Control : IControl hybrid TextControl   // duplicate symbols!
{
}
```

Prior to JewelScript 1.1, this case generated a compile-time error, because it leads to duplicate symbols in the new class. Obviously we would derive method OnClick from IControl and – as a delegate – from TextControl.

JewelScript 1.1 solves this problem by generating code that implements method OnClick for the interface, which in turn calls the delegate OnClick created from the hybrid base class. That way, we can derive methods from the hybrid base class and still make our new class compatible to the interface.

Another special case is handling of accessors from the hybrid base class. The compiler cannot create delegates from accessors, therefore they aren't normally considered as 'suitable' methods when using hybrid inheritance.

However, accessors can be declared in interfaces. So if the compiler detects matching accessors in the interface and the hybrid base class, it will again generate code automatically. The compiler will implement accessors derived from the interface by calling their counterparts defined in the hybrid base class.

This automatic code generation will be performed during the linking process, and only if the script developer has chosen not to implement the method or accessor in the new class. That means, it is possible to selectively reimplement methods and accessors in the new class.

The compiler also automatically creates the variable *base* in the new class. This is a reference to the instance of our hybrid base class. We can use this to override and call base class.

```
strict interface IPerson
{
    accessor string Name();
    accessor Name(string name);
}

class Person : IPerson
{
    method Person() { name = "Peter"; }
    accessor string Name() { return name; }
    accessor Name(string s) { name = s; }
    string name;
}

class NewPerson : IPerson hybrid Person
{
    method NewPerson() hybrid (new Person())
    {
    }
    accessor Name(string s)
    {
        println("NewPerson::SetName = " + s);
        base.Name(s); // this can be used to call base class
    }
}
```

To summarize: By declaring methods and accessors in a common interface and having classes implement that interface, hybrid inheritance allows us almost true sub-classing.

The *NewPerson* class above can again be used as a base class for a new class *AnotherPerson*, in which we would have the option of either inheriting base functionality, or selectively reimplementing methods and accessors.

strict interfaces

If the interface our new class implements has been declared strict, JewelScript will regard existing counterparts for the interface's methods and accessors in the hybrid base class as sufficient.

That means, there will be no link-time error if methods or accessors derived from a strict interface are not implemented in the new class. The fact that a base class implementation exists is satisfactory for the linking process.

Exceptions

At present, JewelScript allows users to define exception classes and to throw exception instances from within script code. Exceptions currently can only be handled in native C/C++ code, though. A try / catch statement for handling exceptions in script code is planned for a later version of the language.

In order to throw an exception, we need to define an exception class that implements the built-in exception interface. The exception interface is defined as follows:

```
strict interface exception
{
    method int getError();
    method string getMessage();
}
```

By implementing this interface, we can make sure that the runtime, as well as the native application can obtain an error code from the exception object, regardless of what other data members are added to the exception object.

```
class WrongTypeException : exception
{
    method WrongTypeException(string s)   { _s = s; }
    method int getError()                 { return typeof(this); }
    method string getMessage()            { return _s; }
    string _s;
}

function var Incrementor(var element, var args)
{
    if( typeof(element) != typeof(int) )
        throw new WrongTypeException("in Incrementor(), variable 'element'");
    return element + 1;
}
```

The above example shows a custom exception class that allows the script programmer to throw an arbitrary string, and an Incrementor() delegate, that throws the custom exception, if it detects that the wrong type of element has been passed to it.

If an exception is thrown, the whole stack is automatically unrolled to the most recent native entry-point. The JILCallFunction() API function will return the handle to the exception object instead of the result of the script function that has been called.

Native code should use the NTLHandleToError() API function to test, whether the result of a script function call is an exception object or a regular function result. The function will actually call the getError() method of the exception object and return the result as the error code.

Therefore, the getError() method of an exception object should never return 0. Apart from that, the native application is free to define any value and meaning of error codes that can be thrown. The example above just returns the type-id of the exception class as the error code.

The API function NTLHandleToErrorMessage() can be called to retrieve the exception's error message. The API function will call the exception object's getMessage() method to retrieve the string.

Of course the application can also check the type of the returned exception object, and dependent on the type, call any additional member functions of the exception object to retrieve additional information about the error.

The purpose of throwing exceptions from script code is to instantly end a call from the native application to a script function, and pass an error object, no matter how deep inside nested function calls the virtual machine is currently running.

This also works with multiple nested native → script → native → script calls, as long as the involved native code properly propagates the error code up to their caller. Therefore throwing exceptions is also useful to exit delegates that are executed from native types.