# Simple Objects

Sascha Demetrio

July 23, 2005

**Abstract**

This documents defines the properties of so called simple objects and how these objects are serialized. A simple object is completely self contained with respect to data. The semantics of simple object methods are up to the application handling simple objects.

Technically, a simple object is a container for simple structured data consisting of numeric values, arrays (may be associative), UNICODE strings, opaque binary objects, external variable references and programmatic expressions. There are two serialization forms defined for simple objects: a human readable and self documenting text serialization and a compact binary serialization.

Simple objects may be used for data storage and exchange between applications. A typical application would be a remote procedure call / method invocation (RPC/RMI) protocol based on a simple object serialization.

Simple objects may come handy where object representations based on XML are too much. An application may use both, simple objects and objects represented as instanced of XML schemas.

A reference implementation of a programming library offering access to simple object serializations can be found on

`http://www.wizard-labs.org/sobject/`

The reference implementation is available under a BSD style license an may be used free of charge for commercial and non-commercial applications. For details, see the `LICENSE` file included in the distribution.

# Contents

# 1 Overview

Simple objects are objects with a defined serialization that may be accessed from different runtime environments. The following requirements were considered in the design:

- The most common basic data types shall be supported. That is a **NIL** object, booleans, integers, floats, strings, lists/arrays/dictionaries.
- The string type shall support UNICODE characters.
- It shall be possible to store opaque binary data.
- It shall provide a method-call data type suitable for remote method invocation (RMI).
- An efficient binary serialization shall be defined.
- A human readable text serialization shall be defined.
- A standard representation of addresses in simple objects shall be defined.

To keep the design simple, the following limitations are accepted:

- There is no support for a reference or pointer type.
- The serializations are not optimized for fast random access.

This specifcation consits of a *core* part and an *appendix*. The core part contains everything that is required for serialization and deserialization of simple objects. The specifcation of *addresses* is also part of the core specifcation. An address is an expression selecting a subobject of a simple object (e.g. an object contained in an array or part of an expression). The following sections will cover the details of the simple object structure, serializations, and addresses. Section 2 covers the structure of simple objects, sections 3.1 and 4 define the text and binary serializations, section 5 covers addresses.

Semantics of classes or expressions is *not* part of the core specifcation. Specific application programming interfaces (APIs) are also *not* covered by this specifcation. However, some semantics are very commonplace (like arithmetic operation semantics on numeric types), as well as some object classes and APIs. The *appendix* covers some of these aspects. Appendix A lists some examples of text and binary serializations of simple objects, appendix B defines standard expression semantics, appendix C defines some common object classes along with representation and semantics, appendix D defines a language independed simple object API which should inspire actual simple objects programming libraries.

# 2 Structure

A simple object is a (*type*, *class*, *data*) triple. The *type* is one of the following: `nil`, `bool`, `int`, `float`, `string`, `binary`, `array` (representing a simple array or an associative array), and `expr` (representing a programmatic expression). The object's class is either null (not present) or a UNICODE string identifying the object's class. The semeantics of the object class is up to the application, however some object classes are defined in the appendix (which is not part of the core specification). The *data* field depends on the object type.

## 2.1 Object Types

The *type* field of a simple object tuple has one of the following values:

`nil`
: This type is reserved for the **NIL** object. The **NIL** object typically represents an unspecified or undefined value. No *data* field is associated with this type.

`bool`
: An object of this type holds the truth values **TRUE** or **FALSE** in its data field.

`int`
: An object of this type represents an integer value. The *data* field holds a 64 bit signed integer in the range $-2^{63} \ldots (2^{63}-1)$.

`float`
: An object of this type represents a floating point value. The *data* field holds a 64 bit IEEE floating point number. This is typically equivalent to the ISO C type `double`.

`string`
: An object of type string represents a sequence of UNICODE characters and escape sequences. The *data* field holds a null-terminated sequence of bytes encoding the UNICODE characters and escape sequences using the UTF-8 scheme. UNICODE strings with escape sequences are called extended strings and are covered in section 2.3.

`binary`
: An object of this type represents an opaque byte array, associated with a simple object describing its type. The *data* field is a tuple (*id*, *body*), where *id* is an arbitrary simple object and *body* is a byte array.

`array`
: An object of type `array` is a (possibly empty) sequence of tuples of arbitrary simple objects. The *data* field as a list of simple object tuples (*key*, *value*). For a typical array (or list), the *key* field of all tuples holds the object **NIL**. For an associative array (or dictionary), the *key* field holds the associated key. Note that the key is not necessarily unique. If multiple values share the same key, the last value with that specific key is the value bound to the key.

`expr`
: Objects of type `expr` (short for "expression") represent a programmatic expression. An expression combines one or more objects with an operator. Section 2.4 covers expressions in detail.

`vref`
: Objects of type `vref` represent a variable reference. The *data* field of a variable reference holds an extended UNICODE string (see 2.3. The semantics of a variable reference are defined by the application).

## 2.2 Object Classes

The *class* field of a simple object is optional. We'll say the class field is null if it is omitted. If present, the class field is a simple UNICODE string (i.e. *not* an extended UNICODE string as defined in section 2.3 – class names may *not* contain variable references).

A programming library may support callback hooks for serialization and deserialization of objects belonging to a specific class. For example, time information may be stored in ISO notation and translated to an epoch representation when deserialized.

A set of standard object classes (and semantics) is defined in appendix C.

## 2.3 Extended Strings

An *extended string* is a sequence of UNICODE characters and escape sequences. An escape sequence either represents the character ESC itself or a variable reference. An escape sequence is introduced by the ASCII character ESC (code point `0x1b`).

The simple case is a self quoting escape sequence. This is the character ESC followed by another ESC character. This sequence represents a single ESC character.

A variable referencing escape sequence is an ESC followed by an STX and a reference string. The reference string is terminated by the sequence ESC–ETX (ASCII STX has the code point `0x2`, ASCII ETX has the code point `0x3`) or by the end of the string. The reference string itself may contain escape sequences, so the terminating ESC–ETX must match the initial ESC–STX.

An unrecognized ESC sequence or an ESC–ETX sequence outside of an escape sequence is an error and the semantics are undefined.

The semantics of variable references are completely up to the application. A programming library implementing access to simple objects would typically handle variable references in extended strings transparently through a callback mechanism or virtual method.

## 2.4 Expression Objects

An expression object represents a programmatic expression containing simple objects (operands) linked with operators. We'll call operators combining two or more expressions infix operators, since these operators are typically written infix (between the operands) when the expression is written down.

The following operators are defined for simple object expressions:

Plus, minus, times, divide, modulo, concat.
: The plus/minus/times/divide/modulo/concat operator represents an abstract addition/subtraction/multiplication/division/modulo/concatenation of two simple object expressions.

Approximate less-than, less-or-equal, greater-than, greater-or-equal.
: These operators represent a symbolic approximate ordered comparison of two simple object expressions using a specified fuzz value. This is a three operand expression where the first two operands are objects being compared and the third operand is the fuzz value.

Less-than, less-or-equal, greater-than, greater-or-equal.
: These operators represent a symbolic ordered comparison of two simple object expressions.

Approximate equal, not-equal.
: These operators represent a symbolic application comparison for equality of two simple object expressions using a specified fuzz value. This is a three operand expression where the first two operands are objects being compared and the third operand is the fuzz value.

Equal, not-equal.
: These operators represent a symbolic comparison for equality of two simple object expressions.

And, or.
: These operators represent a symbolic logical AND/OR combination of two simple object expressions.

Not, negate, positive.
: This operator represents a symbolic logical/numerical negation/numerical positive of a simple object expression.

Conditional.
: This operator represents a conditional combination of three simple object expressions. The first operand represents the condition selecting either the second or the third operand.

Sequential combination.
: The sequential combination operator combines two simple object expressions to a symbolic sequence.

Selection.
: The selection operator combines two simple object expressions to a symbolic selection expression. The first operand represents the object being selected from and the second operand represents the selector.

Index, call.
: An index/call operation combines two expressions to a symbolic index/method call expression. The first operand represents the object being indicated/called and the second operand represents the index/argument list. The second operand of an index/call operator is always an array object.

The semantics of expressions are defined by the application, the simple objects only take care of storing the expression. A programming library implementing access to simple object serializations may or may not offer support in evaluating expression objects.

Appendix B suggests expression semantics for all operators except for the call operator. A programming library for simple objects should support for evaluating expressions conforming to these semantics.

## 2.5 Comparing Simple Objects

Simple objects may be compared for equality. The core specification does not define the semantics of an ordered comparison on simple objects, ordered comparisons are covered in appendix B (Expression Semantics). The equality of two simple objects may depend on the application context (see below).

Simple objects of different type (*type* field) or class (*class* field) are never equal wrt. this definition. If both the *type* and *class* of two simple objects match, these objects are compared according to the following rules:

`nil`
: Two objects of type `nil` are considered equal.

`bool`, `integer`, `float`
: Two objects of type `bool`, `integer`, or `float` are considered equal, if their numeric values are equal. For values of type `float` this mean *exact* identity. Note that an integer value is always *not equal* to the corresponding (numerically equal) floating point number.

`string`
: Two strings are considered equal, if the UTF-8 encoding of the resolved strings (i.e. after preforming a substitution of all variable references in the strings) are byte-per-byte equal.

*Implementation Note:* If an application wants to compare two strings as-is, it may create an environment where variable references are not substituted. A programming library may provide a "raw" variant of object comparison as a convenience feature.

`binary`
: Two binary objects are considered equal if the type ID objects are equal and the data parts are byte-per-byte equal.

`array`
: Two arrays are considered equal if they contain the same number of (*key*, *value*) pairs the corresponding *key* and *value* objects (at the respective same index) compare equal.

  **Note:** If arrays are used as dictionaries, the order of the keys is significant. If an application wants to perform a true dictionary comparison, it should normalize (i.e. sort) the arrays in an appropriate way.

`expr`
: Two expressions are considered equal if they use the same operator, the same number of operands, and if the respective operands of both expressions compare equal.

# 3 Text Serialization

The text serialization is intended for human readable object storage and for objects specified by a user of an application (e.g. in a configuration file). The text serialization syntax can be used for user specified values in a configuration file or even for the entire configuration file.

The text serialization used for storage utilizes a subset of the available syntax and uses ASCII characters only. In this section we'll first describe the full syntax of simple object text serializations. The ASCII only serializations are covered in section 3.2.

## 3.1 Text Serialization Syntax

A text serialization of a simple object is a sequence of byte encoded characters. For objects nested within a simple object, the serialization of the nested object is contained within the serialization of the entire object as a substring. The syntax of the simple object text serialization mimics the typical programming language syntax of constant expressions.

### 3.1.1 Serialization Contexts

A text serialization is always interpreted relative to a context type (called the serialization context, or context for short). An entire object is typically interpreted relative to the `general` context or `expression` context, but may also be interpreted relative to another context. When a text serialization is read, the application specifies a serialization context. A serialization context is one of the following:

`general`
: The general context is the typical top-level context for text serializations.

`selection`
: The `selection` context is used when the specified object acts as a key in an associative array or selector in a selection expression. Every valid serialization in the `general` context is also valid in the `selection` context. The only difference is that keywords are interpreted as strings in the selection context.

`array`
: In the array context, the text serialization is interpreted as a list of simple objects (values) or tuples of simple objects (key/value pairs). Values and key/value pairs may be mixed.

`expression`
: In the `expression` context, the text serialization is interpreted as a hierarchy of objects connected with operators.

`string`
: In the `string` context, the text serialization is interpreted as a string.

### 3.1.2 Comments

A text serialization of a simple object may contain textual comments. Comments may appear in all places where insignificant whitespace may be used. Note that comments may not be used when a serialization is parsed in the `string` context.

Two forms of comments are supported: script-style comments and C style comments. Script style comments are introduced by a hash character "#" and are terminated by the next line separator (either ASCII CR or ASCII LF) or by the end of the serialization. C style comments are introduced by the character sequence "`/*`" and must be terminated by the character sequence "`*/`". C style comments may not be nested.

### 3.1.3 Keywords

Some objects (**NIL**, booleans and out-of-range floating point numbers) are represented by keywords. In a simple object text serialization, the following keywords are recognized: `nil`, `true`, `false`, `nan`, `inf`, `-inf`. Keywords are matched case insensitive, e.g. `true`, `TRUE`, `TrUe` are all valid notations for the keyword `true`.

### 3.1.4 The `general` Context

The `general` context is the default context for the text encoding. Leading and trailing whitespace is ignored (skipped).

A serialization in `general` context may start with an (optional) class specifier. A class specifier is a class name enclosed in curly braces. The name enclosed in braces specifies the *class* field of the simple object. The class name may be encoded as a UTF-8 string and may contain character escape sequences introduced by a backslash as described in the section "Quoted Strings" (3.1.11, page 16). Whitespace following the class specifier is ignored (skipped).

In addition to the class specifier, the following lexical structures are recognized:

- The keyword `nil` represents the **NIL** object.
- The keyword `true` represents an object of type `bool` with the truth value **TRUE**.
- The keyword `false` represents an object of type `bool` with the truth value **FALSE**.
- The keywords `nan`, `inf`, `-inf`. These keywords represent the IEEE floating point values NAN (not a number), INF (infinity), and -INF (negative infinity). On platforms not supporting a quiet NaN[1], infinity is used instead of NaN. On systems not supporting a signed infinity, an unsigned infinity is used for both positive and negative infinity.
- Integer values in C notation (without suffixes). These values represent objects of type int (64 bit integer). The value associated with notations describing a value that is out of bounds is undefined.
- Floating point values in C notation (including ISO C hex floats). These values represent objects of type float (64 bit floating point, equivalent to a C `double` in most C implementations).
- Binary objects. A binary object is enclosed in a pair of single ASCII percent signs "%" or double ASCII percent signs "%%". Text serializations of binary objects are covered in section 3.1.9.
- *Unquoted strings*. An unquoted string is a sequence of ASCII alphanumerics, underscores, and hyphens. An unquoted string may not start with a digit and must contain at least one character that is not a hyphen. If an unquoted string starts with a hyphen, its second character may not be a digit.
- *Quoted strings*. A quoted string is a sequence of characters enclosed in quoting characters (single quotes or double quotes). Quoted strings are covered in section 3.1.11.
- Arrays. An array serialization is enclosed in a matching pair of brackets (ASCII [ and ASCII ]). The character sequence enclosed is interpreted relative to the array context.
- Expressions. An expression serialization is enclosed in parentheses (ASCII ( and ASCII )). The character sequence enclosed is interpreted relative to the expression context.
- Variable references. A variable reference is introduced by an ASCII dollar sign. Variable references are covered in section 3.1.10.

[1] A quiet NaN is a not-a-numner value that does not rise a floating point exception.

### 3.1.5 The `selection` Context

The `selection` context can be considered a variant of the `general` context. Every object serialization that is valid in the `general` context is also valid in the `selection` context, and vice versa. However, in the `selection` context, all keywords are recognized as strings. When a keyword is recognized as a string, its case *is* significant.

To encode a keyword represented object in selection context, the object can be written as an expression (see section 3.1.7). This is done by putting the keyword in parentheses.

### 3.1.6 The `array` Context

In the `array` context, the serialized object is interpreted as an array or associative array of simple objects, that is, a sequence of values (simple objects interpreted in `general` context) and key/value pairs.

A key/value pair is represented by two simple object serializations separated by an ASCII equal sign "`=`" or ASCII colon "`:`". The first object represents the key and is interpreted in `selection` context, the second object represents the value and is interpreted in `general` context.

The values and key/value pairs in the array are separated by whitespace and/or commas. More precisely, in array context, commas are treated as whitespace.

### 3.1.7 The `expression` Context

The `expression` context is used to interpret a serialization of a simple object expression. The `expression` context offers a syntactical superset of the `general` context. Every valid and complete serialization is also valid in the `expression` context (*complete* means that the serialization is terminated by the end of the string).

In the `expression` context, simple object serializations may be combined using the following operator sings: "`+`" (plus, positive), "`-`" (minus, negate), "`*`" (multiply), "`/`" (divide), "`%`" (modulo), `~` (concatenation), "`?:`" (conditional), "`&&`" (logical AND), "`||`" (logical OR), "`!`" (logical NOT), "`<`" (less than), "`<=`" (less or equal), "`>`" (greater than), "`>=`" (greater or equal), "`==`" (equal), "`!=`" (not equal), "`,`" (sequence), "`.`" (selection), "`[]`" (index), "`()`" (method call).

If multiple operators are present, the expression nesting using the following precedence rules:

1. The operators with the highest precedence are nested first. The operator precedence values are defined below.
2. For unary operators with equal precedence, the inner expressions are nested first.
3. For infix operators with equal precedence, the expressions are nested from the left to the right.

The list below specifies the operator syntax in detail. For every operator, the operator precedence is specified. For every operand, the context is specified. The specified context applies only if the operand is not combined by an operator precedence rule.

Selection operator "`.`" (precedence 9)
: Syntax: *object* `.` *selector*
This represents a selection expression. The *selector* string is interpreted in `selection` context and the *object* string is interpreted in `general` context.

Index operator "`[]`" (precedence 9)
: Syntax: *object* `[` *index* `]`
This represents an index expression. The *object* string is interpreted in `selection` context and the *index* string is interpreted in `array` context.

Method call operator "`()`" (precedence 9)

Syntax: *object* `(` *arguments* `)`

This represents a method call expression. The *object* string is interpreted in `selection` context and the *arguments* string is interpreted in `array` context.

Logical NOT operator "`!`" (precedence 8)

Syntax: `!` *object*

This represents a logical NOT expression. The *object* string is interpreted in `general` context.

Arithmetic negate operator "`-`" (precedence 8)

Syntax: `-` *object*

This represents an arithmetic negation expression. The *object* string is interpreted in `general` context.

Arithmetic positive operator "`+`" (precedence 8)

Syntax: `+` *object*

This represents an arithmetic positive expression. The *object* string is interpreted in `general` context.

Concatenation operator ~ (precedence 7)

Syntax: $operand_1$ ~ $operand_2$

This represents a concatenation expression. Both operands $operand_1$ and $operand_2$ are interpreted in `general` context.

Multiply/divide/modulo operators "`*`", "`/`", "`%`" (precedence 7)

Syntax:

$operand_1$ `*` $operand_2$
$operand_1$ `/` $operand_2$
$operand_1$ `%` $operand_2$

This represents a multiplication/division/modulo expression. Both operands $operand_1$ and $operand_2$ are interpreted in `general` context.

Plus/minus operators "`+`", "`-`" (precedence 6)

Syntax:

$operand_1$ `+` $operand_2$
$operand_1$ `-` $operand_2$

This represents a plus/minus expression. Both operands $operand_1$ and $operand_2$ are interpreted in `general` context.

Approximate ordered comparison operators "`< +-`", "`<= +-`", "`> +-`", "`>= +-`" (precedence 5)

Syntax:

$operand_1$ `<` $operand_2$ `+-` $operand_3$
$operand_1$ `<=` $operand_2$ `+-` $operand_3$
$operand_1$ `>` $operand_2$ `+-` $operand_3$
$operand_1$ `>=` $operand_2$ `+-` $operand_3$

This represents an approximate ordered comparison expression. All operands are interpreted in `general` context.

Ordered comparison operators "`<`", "`<=`", "`>`", "`>=`" (precedence 5)

Syntax:

$operand_1$ `<` $operand_2$
$operand_1$ `<=` $operand_2$
$operand_1$ `>` $operand_2$
$operand_1$ `>=` $operand_2$

This represents an ordered comparison expression. Both operands $operand_1$ and $operand_2$ are interpreted in `general` context.

Approximate equality comparison operators approx-equal/approx-not equal "`== +-`", "`!= +-`" (precedence 5)

Syntax:

*operand*$_1$ `==` *operand*$_2$ `+-` *operand*$_3$

*operand*$_1$ `!=` *operand*$_2$ `+-` *operand*$_3$

This represents an approximate equality comparison expression (compare for approximate equality, approximate non-equality). All operands are interpreted in `general` context.

Equality comparison operators equal/not-equal "`==`", "`!=`" (precedence 5)

Syntax:

*operand*$_1$ `==` *operand*$_2$

*operand*$_1$ `!=` *operand*$_2$

This represents an equality comparison expression (compare for equality, non-equality). Both operands *operand*$_1$ and *operand*$_2$ are interpreted in `general` context.

Logical AND operator "`&&`" (precedence 4)

Syntax: *operand*$_1$ `&&` *operand*$_2$

This represents a logical AND expression. Both operands *operand*$_1$ and *operand*$_2$ are interpreted in `general` context.

Logical OR operator "`||`" (precedence 3)

Syntax: *operand*$_1$ `||` *operand*$_2$

This represents a logical OR expression. Both operands *operand*$_1$ and *operand*$_2$ are interpreted in `general` context.

Sequential operator "`,`" (precedence 2)

Syntax: *expression*$_1$ `,` *expression*$_2$

This represents a sequential expression. Both operands *expression*$_1$ and *expression*$_2$ are interpreted in `general` context.

Conditional operator "`?:`" (precedence 1)

Syntax: *condition* `?` *expression*$_1$ `:` *expression*$_2$

This represents a conditional expression. All operands *condition*, *expression*$_1$, and *expression*$_2$ are interpreted in `general` context.

### 3.1.8 The `string` Context

The `string` context is used if the serialization is to be interpreted as a string. In `string` context, leading whitespace is skipped. If the first non-whitespace character is a quoting character (single quote or double quote), the string is interpreted as if it was parsed in `general` context.

If the first non-whitespace character is not a quoting character, the entire serialization starting with the first non-whitespace character is interpreted as a string. Escape sequences (introduced by a backslash character) and variable references are accepted as if the string was enclosed in a pair of quoting characters. If the string contains unquoted quoting characters, these quoting characters are interpreted literately.

### 3.1.9 Binary Objects

Binary objects are serialized either as text containing escape sequences for non-ASCII characters or as base 64 encoded string. The text variant has the advantage of being human readable but has the disadvantage that line breaks or other whitespace characters may be disturbed when the serialization is stored or transferred. The text variant is typically used for fragments of script code that are in itself resistant to whitespace/line break modifications. The base 64 encoding is completely 8 bit clean but takes a little bit of extra storage/bandwidth.

**Base 64 Encoding**
The base 64 encoding of binary objects is introduced by a single ASCII percent sign "%", followed by a simple object interpreted in `selection` context[2] representing the *id* field of the binary object, followed by a single ASCII colon character "`:`", followed by the base 64 encoded binary *data* field of the binary object (called the *data section*, and a terminating single ASCII percent sign. If the *id* of the binary object is itself a binary object (i.e. the serialization starts with a percent sign), it has to be separated from the initial percent sign by at least one whitespace character. Whitespace characters preceding the colon and whitespace characters in the data section are ignored.

**Text Encoding**
The serialization as text is introduced by double ASCII percent signs "%%", followed by the *id* represented by a serialization of a simple object interpreted in `selection` context, followed by a single ASCII colon "`:`", followed by a the *data* represented as ASCII text (called the *data section*), followed by terminating double ASCII percent signs. In the data section, leading whitespace up to (and including) the first line break[3] is ignored. If the leading whitespace does not contain a line break, the entire leading whitespace is ignored.

Inside the binary section, special characters can be escaped using the sequence "`\X`", followed by a single non-alphanumeric character (which is replaced by that character) or two hex digits (specifying the byte value). "`\X`" sequences are substituted only if the number of unescaped backslashes preceding the "`X`" is odd. A trailing "`\X`" sequence (i.e. preceding the terminating double ASCII percent signs) is skipped (ignored).

Examples for escape sequences in the data section:

`X\\X`
: Not processed, since the number of unescaped backslashes preceding the second "`X`" is even.

`X\X\\X1b`
: The sequence "`\X\`" is replaced by a single backslash character, the sequence "`\X1b`" is replaced by an escape character (ASCII ESC). Note that the backslash character preceding the sequence "`\X1b`" is escaped, hence the number of unescaped backslashes preceding the "`X1b`" is odd.

`X\X%%`
: The sequence "`\X`" is skipped. The "`%%`" marks the end of the data section. If you wish to quote double ASCII percent signs, you can use the sequence "`%\X%`".

### 3.1.10 Variable References

Variable references are introduced by a dollar sign and are followed by a *reference string*. When a variable reference is resolved by a simple objects programming library, the reference string is passed to the application which in turn generates a replacement object. The variable reference syntax offers three syntax variants for specifying the reference string:

Simple Reference
: The reference string is specified as a sequence of ASCII alphanumerics and underscores. The sequence has contain at least one character. The sequence is terminated by a character that is not an ASCII alphanumerics and not an underscore or by the end of the serialization.

Quoted Reference
: The reference string is enclosed in double angle brackets (i.e. double less-than "`<<`" and double greater-than "`>>`". The reference string may contain variable references and character escape sequences (introduced by a backslash). If the reference string ends with a single backslash, the

[2] See section 3.1.1 on page 10.
[3] ASCII LF, ASCII CR, or ASCII CR–LF.

trailing backslash is ignored (this feature is required to encode a reference string ending with a greater-than sign). (Note that if the reference string contains a double greater-than sign, it has to be quoted as "`>\>`".)

Grouped Reference
: The reference string is delimited by a matching pair of grouping characters. The following characters are grouping characters: "`(`", "`)`", "`[`", "`]`", "`{`", "`}`". The delimiting pair may itself contain matching pairs of quoting characters. Quoted grouping characters and characters contained in a nested quoted reference are not considered for matching. The delimiting pair of grouping characters is part of the reference string.

No whitespace is allowed between the dollar sign and the reference string.

Variable references using a quoted reference or a grouped reference may contain variable references. When a variable reference is resolved, the variable references contained in the reference string have to be resolved first. An application or library program resolving variable references should provide some sort of loop detection.

### 3.1.11 Quoted Strings

A *quoted string* is a sequence of characters enclosed in a pair of quoting characters. If the character sequence contains non-ASCII characters, these characters are interpreted as UTF-8 encoded UNICODE. Invalid UTF-8 sequences or invalid (unpaired) UNICODE surrogate characters are silently discarded.

If a string is enclosed in double quotes variable references are recognized. A variable references within a string is syntactically similar to the serializations of a variable referencing simple object as described in section 3.1.10. Variable referencing in quoted strings is covered in paragraph 3.1.11 below.

**Escape Sequences**
To represent special characters, quoted strings may contain escape sequences similar to those available in ISO C. The following escape sequences are recognized:

`\a`
: Bell (ASCII BEL), code point `0x07`.

`\b`
: Backspace (ASCII BS), code point `0x08`.

`\e`
: Escape (ASCII ESC), code point `0x1b`.

`\E`
: The same as "`\e`".

`\f`
: Formfeed (ASCII FF), code point `0x0c`.

`\n`
: Newline (ASCII LF, Linefeed), code point `0x0a`.

`\r`
: Carriage return (ASCII CR), code point `0x0d`.

`\s`
: A literal ASCII SPACE, code point `0x20`.

`\t`
: Horizontal TAB (ASCII HT), code point `0x08`.

\v
: Vertical TAB (ASCII VT), code point 0x0b.

\LF, \CR, \CR LF
: A skipped line break can be inserted by preceding the line break (either a UNIX style LF, a Mac-style single CR, or a DOS/Windows style CR–LF) with an unquoted backslash character. A skipped line break is ignored.

\"
: A literal double quote, code point 0x22.

\'
: A literal single quote, code point 0x27.

\\
: A literal backslash, code point 0x5c.

\$
: A literal dollar sign, code point 0x24. A quoted dollar sign is not recogized as an introducing character for variable references (see below).

\(
: A literal unmatched opening parenthesis, code point 0x28.

\)
: A literal unmatched closing parenthesis, code point 0x29.

\[
: A literal unmatched opening bracket, code point 0x5b.

\]
: A literal unmatched closing bracket, code point 0x5d.

\{
: A literal unmatched opening brace, code point 0x7b.

\}
: A literal unmatched closing brace, code point 0x7d.

\>
: A literal greater-than sign, code point 0x3e.

\x*N*
: Hex escape. The "\x" is followed by two hex digits specifying an 8 bit character code. Note that "\x00" is illegal, since strings may not contain null-characters.

*N*
: Octal escape. The "\" is followed by 1 to 3 octal characters. If an octal escape is followed by an octal digit, it has to be padded with leading zeroes to make it 3 digits long. The digits represent an 8 bit character code. Note that "\000" is illegal, since strings may not contain null-characters.

\u*N*
: Unicode escape (Java style). The "\u" is followed by 4 hex digits specifying a 16 bit character code. The use of "\u" escapes to form surrogate pairs is discouraged, use "\U" to specify code points that can't be represented as a 16 bit value.

\U*N*
: UTF-32 Unicode Escape. The "\U" is followed by 8 hex digits specifying a 32 bit character code. The use of "\U" escapes to form surrogate pairs is discouraged, these code points should be represented by a single "\U" escape.

`\&R;`
: Character reference. These references work similar to HTML/XML character references. All symbolic character references of HTML 4.0 are supported. Numeric character references "`\&#N;`" are also supported. Note that numeric character references are encoded in decimal, not in hex.

**Variable References in Quoted Strings**
Quoted strings enclosed in double quotes may contain variable references similar to the variable reference serializations described in section 3.1.10.

A variable reference is introduced by an unquoted dollar sign[4]. The dollar sign is followed by a *reference string*. The syntax for the reference string is the same as for variable reference serializations described in section 3.1.10. No whitespace is allowed between the dollar sign and the reference string.

Variable references using a grouped reference may contain variable references. When a variable reference is resolved, the variable references contained in the reference string have to be resolved first.

**Note:** Quoted strings enclosed in single quotes can *not* contain variable references.

**UTF-8 in Quoted Strings**
Quoted strings may contain UNICODE UTF-8 sequences. Any sequences of characters in the range `0x80` – `0xff` (i.e non-ASCII characters) that is not a valid UTF-8 sequences will be discarded. If the string contains UNICODE surrogate character pairs, these pairs are replaced by the represented character. Unpaired surrogate characters are discarded.

## 3.2 ASCII Serialization

Every simple object can be serialized as text using ASCII characters only. The only object type with a text serialization that can contain non-ASCII characters is the string object (string serializations may contain UTF-8 sequences). However, every non-ASCII UNICODE character represented by a UTF-8 sequence may be represented by an escape sequence as described in section 3.1.11.

[4] A literal dollar sign can be represented by quoting the dollar sign with a backslash "`\$`".

# 4 Binary Serialization

The binary serialization is interpreted free of context. The length of the serialization may be determined reliably from the serialization itself. Every object nested inside the represented simple object has a binary serialization that is a substring of the serialization of the entire object.

Every binary serialization of a simple object is composed of a *type byte*, an optional class name, and a (possibly empty) sequence of *data bytes*.

## 4.1 The Type Byte

The type bytes indicated the object *type* (as defined in section 2.1) and a storage size indicator.

| Bits | Description |
|---|---|
| 7 | 1 |
| 6 | Class name present (1) |
| 5 … 3 | Object type (2) |
| 2 … 0 | Storage size (3) |

(1) Class name present.
If this bit is set, the type byte is followed by a null-terminated UTF-8 string indicating the class name of the object (*class* field). If this bit is clear, no class name is associated with the object (i.e. the class name is null).

(2) Object type.
The following list defines the type values for all object types:

| Value | Type | Value | Type |
|---|---|---|---|
| 0 | `nil`/`bool` | 4 | `binary` |
| 1 | `int` | 5 | `array` |
| 2 | `float` | 6 | `expr` |
| 3 | `string` | 7 | `vref` |

(3) Storage size.
The semantics of the storage size depends on the object type. The storage size is determined from the encoded value through the following table:

| Value | Storage Size | Value | Storage Size |
|---|---|---|---|
| 0 | 0 | 4 | 64 |
| 1 | 8 | 5 | 96 |
| 2 | 16 | 6 | 128 |
| 3 | 32 | 7 | extended |

The storage sizes 96, 128, and “extended” are not used by the current version of this specification. The storage sizes 96 and 128 are intended for encoding higher precision numeric values, especially 96 and 128 bit floating point numbers. The special value “extended” might be used by a future version for encoding arbitrary precision integer and floating point values.

## 4.2 The Class Name

If (and only if) bit 6 of the type byte is set, the type byte is followed by a null-terminated, UTF-8 encoded string indicating the class name (*class* field) of the simple object.

## 4.3 The Data Bytes

The format of the data contained in the data bytes of a binary serialization depends on the object type and storage size defined in the type byte. For all data types, the storage size 0 indicates that no data bytes are present in the serialization of the object. Such an object typically represents a suitable null-value.

### 4.3.1 The Types `nil` and `bool`

The type `nil` is always of storage size 0. For objects of type `bool`, the truth value is encoded into the storage size defined in the type byte. The storage size 8 indicates the value **FALSE**, the storage size 16 represents the value **TRUE**. Other storage sizes are not valid.

### 4.3.2 The Type `int`

The data bytes contain the value of an object of type `int`, encoded as a 2-complement signed integer stored in big-endian byte order. The storage size indicates the bit width (and hence the number of data bytes). The following storage sizes are allowed: 0, 8, 16, 32, and 64. If the storage size is 0, the represented integer value is 0.

### 4.3.3 The Type `float`

For objects of type `float`, both the size and the format of the data bytes depend on the storage size.

| Storage Size | Value Encoding |
|---|---|
| 0 | No encoding. The represented value is 0. |
| 8 | The value is encoding as an 8 bit signed fixed point number, 2-complement. The object value 1 is represented by the encoded value 10. |
| 16 | The value is encoded as a 16 bit signed fixed point number, 2-complement. The object value 1 is represented by the encoded value 100. |
| 32 | The value is encoded as a 32 bit ANSI/IEEE 754-1985 floating point number. |
| 64 | The value is encoded as a 64 bit ANSI/IEEE 754-1985 floating point number. |

For storage size 8, the encodable values are in the range $-12.8 \ldots 12.7$ (in steps of 0.1). For storage size 16, the encodable values are in the range $-327.68 \ldots 327.67$ (in steps of 0.01).

### 4.3.4 The Type `string`

The type `string` represents an extended UNICODE string as defined in section 2.3 (page 7). The string is stored as a (*length*, *cdata*) tuple, where *length* indicates the length of the character data *cdata*. *cdata* is a sequence of *length* bytes holding the string using the UNICODE UTF-8 encoding.

The *length* is stored as a big-endian unsigned integer. The width (number of bits) of that integer is indicated by the storage size, which may be 0, 8, 16, 32, or 64. Other storage sizes are not valid. The storage size 0 indicates an empty string.

### 4.3.5 The Type `binary`

Objects of type `binary` are stored as a triple (*id*, *length*, *bdata*).

*id*
: This is the serialization of an arbitrary simple object representing the *id* field of the binary object.

*length*
: This is the length of the binary data, in bytes, stored as a big-endian unsigned integer. The width (number of bits) of that integer is indicated by the storage size, which may be 0, 8, 16, 32, or 64. Other storage sizes are not valid. The storage size 0 indicates a length of 0 bytes.

*bdata*
: *length* bytes of binary data.

### 4.3.6 The Type `array`

The serialization of an `array` object is stored as an alternating sequence of keys and values, preceded by the length of the array (i.e. the number of values).

$$(length, key_0, value_0, ..., key_{length-1}, value_{length-1})$$

The keys and values are stored as binary serializations of simple objects. *length* is stored as a big-endian, unsigned integer. The width (number of bits) of that integer is indicated by the storage size, which may be 0, 8, 16, 32, or 64. Other storage sizes are not valid. The storage size 0 indicates an empty array.

### 4.3.7 The Type `expr`

An expression object is serialized to a expression control byte and a number of operand objects. The expression control byte determines the type of expression and the number of operands.

| Bits | Description |
|---|---|
| 7 | Reserved, must be 0 |
| 6 … 1 | The expression type, see below |
| 1, 0 | The number of operands minus 1 |

The encoding used for the expression types is shown in table 1.

The number of operands specified in the expression control byte must match a defined simple object expression type. Note that the number of operands *minus one* is stored in the low two bits of the expression control byte, i.e. the value 0 indicates 1 operand, the value 2 indicates 3 operands.

The storage size of an expression object is always 0. Other storage sizes are not valid.

**Index and Method Call Operands**
The right operands of an index or method call operand is always required to by of type `array`, as defined in section 2.4 (page 7). If the operand is an array containing exactly one element, the element *value* is not of type `array`, and the element *key* is **NIL**, then that element may be encoded directly instead of encoding and array containing that element.

| Value | Operator Sign | Expression Type |
|---|---|---|
| 0 | `+` | Plus (2 operands) or unary positive (1 operand) |
| 1 | `-` | Minus (2 operands) or negation (1 operand) |
| 2 | `*` | Multiplication (2 operands) |
| 3 | `/` | Division (2 operands) |
| 4 | `%` | Modulo (2 operands) |
| 5 | `<` | Ordered comparison: less-than (2 operands) or approximate-less-than (3 operands) |
| 6 | `<=` | Ordered comparison: less-or-equal (2 operands) or approximate-less-or-equal (3 operands) |
| 7 | `>` | Ordered comparison: greater-than,(2 operands) or approximate-greater-than (3 operands) |
| 8 | `>=` | Ordered comparison: greater-or-equal (2 operands) or approximate-greater-or-equal (3 operands) |
| 9 | `==` | Equality comparison (2 operands) or approximate equality comparison (3 operands) |
| 10 | `!=, !` | Non-equality comparison (2 operands) or approximate non-equality comparison (3 operands) or logical negation (1 operand) |
| 11 | `&&` | Logical AND (2 operands) |
| 12 | `\|\|` | Logical OR (2 operands) |
| 13 | `?:` | Conditional (3 operands) |
| 14 | `,` | Sequence (2 operands) |
| 15 | `.` | Selection (2 operands) |
| 16 | `[]` | Index (2 operands) |
| 17 | `()` | Method call (2 operands) |
| 18 | `~` | Concatenation (2 operands) |

Table 1: Expression Type Encoding

#### 4.3.8 The Type `vref`

An object of type `vref` is serialized to a tuple (*length*, *vdata*). The *vdata* holds *length* bytes containing an extended UNICODE string (see section 2.3 on page 7). The string represents the reference string of the variable reference. Nested variable references are represented by variable references in the extended string.

The *length* is stored as a big-endian unsigned integer. The width (number of bits) of that integer is indicated by the storage size, which may be 0, 8, 16, 32, or 64. Other storage sizes are not valid.

# 5 Addresses

An *address* is a simple object (or serialization of a simple object) representing a path to a subobject nested within a simple object. An address is an arbitrary simple object.

## 5.1 Address Resolution

Address objects are applied to simple objects to select a subobject of that object. Such a subobject may either be a propper subobject or a symbolic subobject.

A *propper subobject* of a simple object is an object that is completely contained within that object. A propper subobject is actually a part of the containing object.

A *symbolic subobject* is an object containing a propper subobject of an object. A symbolic subobject is typically the result of a resolution process where some parts of the address could not be resolved.

### 5.1.1 Address Resolution Operations

The following operations may be performed in an address resolution process:

Self Substitution
: The address object not substituted (i.e. replaced by itself).

Object Substitution
: The address object is substituted by the entire object the address is being applied to.

Quoted Substitution
: A *quoted substitution* is performed on a *operand* object. The address is resolved to the operand.

Selection
: A *selection operation* is performed on a *dictionary* object using a *selector* object. If the dictionary is of type `array`, the elements of the array are processed in *reverse order* until an element is found whose key compares equal[5]. If a matching key was found, the address is resolved to associated value. If the key is not found or if the dictionary is not of type `array`, then the address is resolved to a selection expression object where the first operand is the dictionary and the second operand is the selector.

Index
: An *index operation* is performed on a *sequence* object using an *index*. If the index $I$ is of type `int`, it is used to select $(I+1)$th element from the sequence. If $I$ is negative, then the index is relative to the end of the sequence, i.e. it addresses the $(L+I+1)$th element from the sequence, where $L$ is the length of the sequence. An index substitution is performed if and only if the index is of type `int` and the sequence is of type `array`, `string`, or `expr` (expression).

  The length $L$ of the sequence and the exact definition of an *element* of the sequence depends on the sequence type:

  `array`
  : The length $L$ of an array is the number of values held in the array. Here, an *element* of the array is a single value (i.e. not the key/value pair). The element keys are ignored in an index operation.

  `string`
  : The length of a string is number of UNICODE characters and variable references in the extended string (a variable references is treated like a single UNICODE character). An element is a single UNICODE character or a variable reference, which is represented by a

[5] The comparison of simple objects is covered in section 2.5, page 8.

single ASCII character, a UTF-8 character sequence, or an extended string escape sequence. In an index operation, a single UNICODE character or variable reference is represented by a string object holding that character.

`expr`
: The length of an expression is the number of operands. An element of an expression is a single operand.

The value $E = I$ for $I \geq 0$ or $E = L + I$ for $I < 0$ is called the *effective index* $E$. If $E$ is out of bounds (i.e. either $E < 0$ or $E \geq L$), then the address is resolved to **NIL**. If the sequence is not of a valid sequence type (`array`, `string`, `expr`) or the index is not of type `index`, then the address is resolved to an index expression where the first operand is the sequence and the second operand is a one element array containing the index.

Slice
: The *slice operation* is similar to the index operation. A slice operation is performed on a *sequence* using a pair of bounds called the *upper bound* and the *lower bound*. A slicing subsection is performed if both bounds are of type `int` and the sequence is of type `array` or `string`.

The length $L$ of the sequence and the exact definition of an *element* of the sequence depends on the sequence type:

`array`
: The length $L$ of an array is the number of values held in the array. Here, an *element* of the array is a single value (i.e. not the key/value pair). The element keys are ignored in an index operation.

`string`
: The length of a string is number of UNICODE characters and variable references in the extended string (a variable reference is treated like a single UNICODE character). An element is a single UNICODE character or a variable reference, which is represented by a single ASCII character, a UTF-8 character sequence, or an extended string escape sequence. In an index operation, a single UNICODE character or variable reference is represented by a string object holding that character or variable reference.

Let $I$ be the lower bound and $J$ the upper bound. The *effective lower bound* $E$ and *effective upper bound* $F$ are defined as:

$$E = \begin{cases} 0 & \text{if } I < -L-1 \\ L+1+I & \text{if } -L-1 <= I < 0 \\ I & \text{if } 0 <= I <= L \\ L & \text{if } I > L \end{cases} \qquad F = \begin{cases} 0 & \text{if } J < -L-1 \\ L+1+J & \text{if } -L-1 <= J < 0 \\ J & \text{if } 0 <= J <= L \\ L & \text{if } J > L \end{cases}$$

The resolved object depends on the sequence type:

`array`
: If the sequence is of type `array`, the address is resolved to an array containing all elements from the index $E$ (inclusive) to the index $F$ (exclusive). If $E \geq F$, the address is resolved to an empty array.

`string`
: If the sequence is of type `string`, the address is resolved to a string containing all UNICODE characters and variable references from the index $E$ (inclusive) to the index $F$ (exclusive). If $E \geq F$, the address is resolved to an empty string.

If the sequence is not of type `array` or `string` or if at least one of the bounds is not of type `int`, the address is resolved to an index expression where the first operand is the sequence and the second operand is an array containing the lower bound at index 0 and the upper bound at index 1.

### 5.1.2 The Resolution Process

The address resolution is a recursive process, performing the following steps:

1. If the address is a variable reference (i.e. an object of type `vref`), then a variable substitution step is performed. If the substitution succeeds, the address is replaced by the resulting object. If the substitution fails, the `vref` object itself is used as the address.
2. If the address is an object of type `expr` (expression), then the resolution function is applied to all operands of type `expr`.
3. The resolution function (see below) is applied to the result of step 2.

### 5.1.3 The Resolution Function

The *resolution function* itself is described by the following set of rules, matched in the specified order:

Object Substitution
: If the address is **NIL**, it is resolved to the entire object *X*. resolved to the address object itself. This resolution is called an *object substitution*.

Object Selection
: An *object selection substitution* is performed if the address is a selection expression where first operand is **NIL**. A *selection operation* is performed (as defined in section 5.1.1). The *dictionary* is the entire object the address is applied to and the *selector* is the second operand of the selection expression.

Object Index
: An *object index substitution* is performed if the address is an index expression where the first operand is **NIL**, the second operand is an array holding a single element, and the key of the array element is **NIL**. An *index operation* is performed (as defined in section 5.1.1). The *sequence* is the entire object the address is applied to and the *index* is the value of the array element.

Object Slice
: An *object index substitution* is performed if the address is an index expression where the first operand is **NIL**, the second operand is an array holding exactly two elements, and the key of both array elements is **NIL**. A *slice operation* is performed (as defined in section 5.1.1). The *sequence* is the entire object the address is applied to, the *lower bound* is the value of the first array element, and the *upper bound* is the value of the second array element.

Object Append
: An *append substitution* is performed of the address is an index expression where the first operand is **NIL** and the second operand is an empty array. A *slice operation* is performed (as defined in section 5.1.1). The *sequence* is the entire object the address is applied to, the *lower bound* is the length of the sequence and the *upper bound* is equal to the *lower bound* (effectively specifying an empty slice at the end of the sequence).

Combined Selection
: A *combined selection* is performed if the address is a selection expression where the first operand is *not* **NIL**. A *selection operation* is performed (as defined in section 5.1.1). The *dictionary* is first operand and the *selector* is the second operand of the selection expression.

Combined Index
: A *combined index selection* is performed if the address is an index expression where the first operand is *not* **NIL**, the second operand is an array holding a single element, and the key of the array element is **NIL**. An *index operation* is performed (as defined in section 5.1.1). The *sequence* is first operand and the *index* is the value of the array element of the second operand.

Combined Slice
: A *combined slice selection* is performed if the address is an index expression where the first operand is *not* **NIL**, the second operand is an array holding exactly two elements, and the key of both array elements is **NIL**. A *slice operation* is performed (as defined in section 5.1.1). The *sequence* is first operand of the index expression, the *lower bound* is the value of the first array element of the second operand, and the *upper bound* is the value of the second array element of the second operand.

Quoted Selection
: A *quoted selection* is performed if the address is an arithmetic positive expression (i.e. an operand combined with a unary plus operator). A *quoted substitution* is performed as defined in second 5.1.1. The *operand* is the operand of the expression.

Self Selection
: A *self selection* is performed if the address matches none of the patterns listed above. A *self substitution operation* is performed as described in section 5.1.1.

## 5.2 Pure Addresses

A *pure address* is an address that is one of the following:

1. An object of type `nil`.
2. A selection expression where the first operand is a pure address and the second operand is not a selection or index expression.
3. An index operation where the first operand is a pure address and the second operand is an array holding one or two object of type `int`.

A *pure address resolution* is a restricted address resolution process that yields either a propper subobject of the object it is applied to or an error indication[6].

## 5.3 Considerations and Examples

Addresses are typically applied to an object specified elsewhere. A typical application of addresses is an API[7] of a programming library providing access to objects nested in other objects. Such an API might define a method of the class representing a simple object which accepts a text serialization of an address (using the `expression` context). For example:

```
SObject x, y;

x = obtain_an_instance_of_SObject();
y = x.get("NIL.prefs.(${ENV}.USER).editor");
```

We'll assume that the application has set up a variable reference substitution mechanism that substitutes "`${ENV}`" with an (associative) array object representing the process environment. In fact, the `get()` method could automatically prepend the string `"NIL"` if the argument starts with a dot or bracket. In this case, the address could be abbreviated to

```
y = x.get(".prefs.(${ENV}.USER).editor");
```

To address an element value from an associative array where the element key is itself an address, the selection key has to be quoted. The following address resolves to the string `"FOO"` (represented as a text serialization in `expression` context):

[6] What that error indication looks like depends on the application or programming library handling the restricted address resolution.
[7] Application Programming Interface.

```
[(NIL.key1) = FOO (NIL[42]) = BAR].(+ NIL.key1)
```

This example also demonstrates that the very left operand of an address does not have to be **NIL**. In fact, the **NIL** reference to the object the address is being applied may appear multiple times in the address, or not at all. Here's an example where the **NIL** reference appears somewhere nested in the address object (again written in Java style pseudo code):

```
x = new SObject(1);
y = x.get("[FOO, BAR, FIZZLE][NIL]");
```

The value of `y` will be an object representing the string `"BAR"`.

## 5.4 UNICODE Normalization

Some simple objects hold UNICODE strings. These are all simple objects of type `strings` as well as all objects associated with a class name. These UNICODE strings are stored as sequences of numbers resembling UNICODE code points. These code points are *not* limited to 16 bits.

### 5.4.1 Surrogate Pairs

UNICODE code points beyond `0xffff` can be represented as pairs of code points below `0xffff`. The code points making up such pairs are called *surrogates*, the pairs are called *surrogate pairs*. The first surrogate in a pair is called the *high surrogate*, the second is called the *low surrogate*. Surrogate pairs are used in the UTF-16 encoding to represent code points that could otherwise not be represented in 16 bits.

A simple object *never* contains surrogates. All matching pairs of surrogates are resolved and all unmatched surrogates are silently discarded.

### 5.4.2 Normalization

Some characters have different UNICODE representations (e.g. accented characters). Simple objects representing the same character string with different character representations will *not* compare equal. To avoid this problem, an application may

a) use a normalization convention. That means that the application makes sure that all strings are normalized whenever a simple object is created.

b) install a normalization callback. A programming library should provide a hook for installing a string normalization callback function.

The UNICODE standard defines sevaral string normalizations.

# A Serialization Examples

This section contains a collection of serialization examples for text and binary serialization. Binary data will be represented commented as hexdumps.

## A.1 Text Serialization Examples

## A.2 Binary Serialization Examples

# B Expression Semantics

This section defines the expression semantics implemented by the default environment. The expression semantics resembles the typical operator semantics of programming languages like C or Java.

Some operands examine the truth value of an operand. The truth value is defined as follows:

**FALSE**
: If the object is **NIL**, **FALSE**, `0` (zero of type `int`), `0.0` (zero of type `float`), an empty string, a binary object (type `binary`) with a data body length of 0, or en empty array (type `array`).

Undefined
: If the object is of type `expr` or `vref`.

**TRUE**
: For all other objects.

The expression semantics defined in this section are implemented in the default class environment of the standard environment (see section C.1 by overriding the `ClassEnv::eval()` method (or by implementing the `eval()` callback function, if the programming environment does not support inheritance). As a consequence, the expression semantics below are applicable only to expression objects where the class name of the first operand is associated with the default class (i.e. there's no specific class for the class name or the object has no class name).

Note that the class environment used for evaluating an expression object is derived from the class name of the first operand, not the class name of the expression itself. If the expression object has a class name, that class name is applied to the object resulting from the evaluation.

## B.1 Arithmetic Semantics

For arithmetic expressions the `Env::eval()` method performs a recursive evaluation of all operands before the operator is examined. Arithmetic expressions are expressions of the following types: `EXPR_POS`, `EXPR_NEG`, `EXPR_NOT`, `EXPR_PLUS`, `EXPR_MINUS`, `EXPR_MUL`, `EXPR_DIV`, `EXPR_MOD`, `EXPR_CAT`, `EXPR_LS`, `EXPR_LE`, `EXPR_GT`, `EXPR_GE`, `EXPR_EQ`, `EXPR_EQ_APPROX`, `EXPR_NE`, `EXPR_NE_APPROX`, `EXPR_AND`, `EXPR_OR`.

### B.1.1 `EXPR_POS` (Unary positive operator)

If the operand is of type `bool`, it is promoted to type `int` where **FALSE** is mapped to `0` and **TRUE** is mapped to `1`.

### B.1.2 `EXPR_NEG` (Unary negation operator)

If the operand is a number or a boolean (i.e. of type `bool`, `int`, or `float`), it is negated numerically. For booleans **FALSE** is replaced by `0` and **TRUE** is replaced by `-1`.

### B.1.3 `EXPR_NOT` (Unary logical negation operator)

If the operand is not of type `expr` or `vref`, the expression is replaced by the negation of the truth value of the operand.

### B.1.4 `EXPR_PLUS` (Binary plus operator)

The expression is evaluated if one of the following conditions holds:

1. If both operands are numbers (i.e. of type `int` or `float`), the expression is replaced by the sum of both numbers. If both operands are of type `int`, the result will be of type `int`, else the result is of type `float`.
2. Both operands are of type `string`. The expression is replaced by the sequential concatenation of both strings. Note that variable reference are resolved when both operands are evaluated recursively.
3. The first operand is of type `array`, the second operand is not of type `array`. The expression is replaced by an array derived from the first operand, where every value is replaced by the evaluated `EXPR_PLUS`-expression created from the original array element value and the second expression operand.
   Example:

   `([2, key: a] + 1)` is evaluated to `[3, key: (a + 1)]`
4. The first operand is not of type `array`, the second operand is of type `array`. The expression is replaced by an array derived from the second operand, where every value is replaced by the evaluated `EXPR_PLUS`-expression created from the first expression operand and the original array element value.
   Example:

   `(1 + [key: 2, a])` is evaluated to `[key: 3, (1 + a)]`
5. Both operands are arrays of equal length. The expression is replaced by an array derived from the first operand where all keys are taken from the first operand and the values are replaced by evaluated `EXPR_PLUS`-expressions created from positionally corresponding element values.
   Example:

   ```
   ([key1: 1, 2, a, b] + [key2: 3, c, 4, d])
   ```

   is evaluated to

   ```
   [key1: 4, (2 + c), (a + 4), bd]
   ```

Note that the array evaluations are preformed recursively, using the class environment matching the respective expression. An array may contain element values with a class matching a class environment implementing different evaluation semantics.

#### B.1.5 `EXPR_MINUS` (Binary minus operator)

The expression is evaluated if one of the following conditions holds:

1. If both operands are numbers (i.e. of type `int` or `float`), the expression is replaced by the subtraction of both numbers. If both operands are of type `int`, the result will be of type `int`, else the result is of type `float`.
2. The first operand is of type `array`, the second operand is not of type `array`. The expression is replaced by an array derived from the first operand, where every value is replaced by the evaluated `EXPR_MINUS`-expression created from the original array element value and the second expression operand.
   Example:

   `([2, key: a] - 1)` is evaluated to `[1, key: (a - 1)]`
3. The first operand is not of type `array`, the second operand is of type `array`. The expression is replaced by an array derived from the second operand, where every value is replaced by the evaluated `EXPR_MINUS`-expression created from the first expression operand and the original array element value.
   Example:

`(1 - [key: 2, a])` is evaluated to `[key: -1, (1 - a)]`

4. Both operands are arrays of equal length. The expression is replaced by an array derived from the first operand where all keys are taken from the first operand and the values are replaced by evaluated `EXPR_MINUS`-expressions created from positionally corresponding element values. Example:

   ```
   ([key1: 1, 2, a, b] - [key2: 3, c, 4, d])
   ```

   is evaluated to

   ```
   [ key1: -2, (2 - c), (a - 4), (b - d) ]
   ```

Note that the array evaluations are preformed recursively, using the class environment matching the respective expression. An array may contain element values with a class matching a class environment implementing different evaluation semantics.

### B.1.6 `EXPR_MUL` (Multiplication operator)

The expression is evaluated if one of the following conditions holds:

1. If both operands are numbers (i.e. of type `int` or `float`), the expression is replaced by the subtraction of both numbers. If both operands are of type `int`, the result will be of type `int`, else the result is of type `float`.
2. The first operand is of type `array`, the second operand is not of type `array`. The expression is replaced by an array derived from the first operand, where every value is replaced by the evaluated `EXPR_MUL`-expression created from the original array element value and the second expression operand.
3. The first operand is not of type `array`, the second operand is of type `array`. The expression is replaced by an array derived from the second operand, where every value is replaced by the evaluated `EXPR_MUL`-expression created from the first expression operand and the original array element value.
4. Both operands are arrays of equal length. The expression is replaced by an array derived from the first operand where all keys are taken from the first operand and the values are replaced by evaluated `EXPR_MUL`-expressions created from positionally corresponding element values.

See section B.1.5 for examples.

### B.1.7 `EXPR_DIV` (Division operator)

The expression is evaluated if one of the following conditions holds:

1. If both operands are numbers (i.e. of type `int` or `float`), the expression is replaced by the quotient (division) of both numbers. If both operands are of type `int` and the second operand is not `0`, the result will be of type `int`, else the result is of type `float`.
2. The first operand is of type `array`, the second operand is not of type `array`. The expression is replaced by an array derived from the first operand, where every value is replaced by the evaluated `EXPR_DIV`-expression created from the original array element value and the second expression operand.
3. The first operand is not of type `array`, the second operand is of type `array`. The expression is replaced by an array derived from the second operand, where every value is replaced by the evaluated `EXPR_DIV`-expression created from the first expression operand and the original array element value.

4. Both operands are arrays of equal length. The expression is replaced by an array derived from the first operand where all keys are taken from the first operand and the values are replaced by evaluated `EXPR_DIV`-expressions created from positionally corresponding element values.

See section B.1.5 for examples.

#### B.1.8 `EXPR_MOD` (Modulo operator)

The expression is evaluated if one of the following conditions holds:

1. If both operands are numbers (i.e. of type `int` or `float`), the expression is replaced by the division residue of the first number divided by the second. Note that floating point numbers are allowed for this operation. If both operands are of type `int` and the second operand is not `0`, the result will be of type `int`, else the result is of type `float`. If the second operand is `0` or `0.0`, the result will be `nan` (floating point NAN, “Not A Number”).
2. The first operand is of type `array`, the second operand is not of type `array`. The expression is replaced by an array derived from the first operand, where every value is replaced by the evaluated `EXPR_MOD`-expression created from the original array element value and the second expression operand.
3. The first operand is not of type `array`, the second operand is of type `array`. The expression is replaced by an array derived from the second operand, where every value is replaced by the evaluated `EXPR_MOD`-expression created from the first expression operand and the original array element value.
4. Both operands are arrays of equal length. The expression is replaced by an array derived from the first operand where all keys are taken from the first operand and the values are replaced by evaluated `EXPR_MOD`-expressions created from positionally corresponding element values.

See section B.1.5 for examples.

#### B.1.9 `EXPR_CAT` (Concatenation operator)

The expression is evaluated if one of the following conditions holds:

1. Both operands are of type `array`. The expression is replaced by the concatenation of both arrays.
2. Both operands are strings. The expression is replaced by the concatenation of both strings.
3. One of both operands is **NIL**. The expression is replaced by the operand that is not **NIL**.

#### B.1.10 `EXPR_LS` (Less-than comparison operator)

The expression is evaluated if one of the following conditions holds: If both operands are comparable (see section B.1.20), the expression is replaced by a boolean resulting from a “less-than” comparison.

#### B.1.11 `EXPR_LE` (Less-or-equal comparison operator)

If both operands are comparable objects (see section B.1.20), the expression is replaced by a boolean resulting from a “less-or-equal” comparison.

#### B.1.12 `EXPR_GT` (Greater-than comparison operator)

If both operands are comparable objects (see section B.1.20), the expression is replaced by a boolean resulting from a “greater-than” comparison.

#### B.1.13 `EXPR_GE` (Greater-or-equal comparison operator)

If both operands are comparable objects (see section B.1.20), the expression is replaced by a boolean resulting from a “greater-or-equal” comparison.

#### B.1.14 `EXPR_EQ` (Equals comparison operator)

If both operands are comparable objects (see section B.1.20), the expression is replaced by a boolean resulting from a “equal” comparison.

#### B.1.15 `EXPR_EQ_APPROX` (Equals approximate comparison operator)

If the first two operands are comparable objects (see section B.1.20) and the third operand is a number, the expression is replaced by a boolean resulting from a “approximate-equal” comparison (see section B.1.21 on approximate comparisons).

#### B.1.16 `EXPR_NE` (Not-equal comparison operator)

If both operands are comparable objects (see section B.1.20), the expression is replaced by a boolean resulting from a “not-equal” comparison.

#### B.1.17 `EXPR_NE_APPROX` (Not-equal approximate comparison operator)

If the first two operands are comparable objects (see section B.1.20) and the third operand is a number, the expression is replaced by a boolean resulting from a “not-approximate-equal” comparison (see section B.1.21 on approximate comparisons).

#### B.1.18 `EXPR_AND` (Logical AND operator)

If both operands have a defined truth value (i.e. are not of type `expr` or `vref`), the expression is replaced by the logical AND of the truth values of both operands. In contrast to most programming languages, this operator has no shortcut semantics, i.e. both operands are evaluated *before* the operation is performed.

#### B.1.19 `EXPR_OR` (Logical OR operator)

If both operands have a defined truth value (i.e. are not of type `expr` or `vref`), the expression is replaced by the logical OR of the truth values of both operands. In contrast to most programming languages, this operator has no shortcut semantics, i.e. both operands are evaluated *before* the operation is performed.

#### B.1.20 Comparable Objects

In an ordered comparison (“less-than”, “less-or-equal”, “greater-than”, “greater-or-equal”), two objects are comparable if

- both objects are numbers (i.e. of type `integer` or `float`),

- both objects are of type `string`, or
- both objects are arrays of equal length where all keys are **NIL** and all corresponding pairs of elements are comparable.

In an ordered approximate comparison (“approximate-less-than”, “approximate-less-or-equal”, “approximate-greater-than”, “approximate-greater-or-equal”) or an approximate equality comparison (“approximate-equal”, “not-approximate-equal”), two objects are comparable if

- both objects are numbers (i.e. of type `integer` or `float`),
- both objects are array where all keys are **NIL** and all corresponding pairs of elements are comparable.

In an equality comparison (“equal”, “not-equal”), any two objects are comparable.

For ordered comparisons, numbers are compared by their numerical value, strings are compared lexically by the numerical order of the UNICODE code points, arrays are compared recursively and the result is the logical AND of the results of the element comparisons[8].

#### B.1.21 Approximate Comparisons

An approximate comparisons is an operation with three operands. The first and second operands are compared using a fuzz value specified in the third operand. The fuzz value causes the result to be **TRUE**, even if the tested condition is missed by an amount smaller or equal to the specified fuzz value. (Note that the `EXPR_NE_APPROX` is the exact opposite of `EXPR_EQ_APPROX`, that’s why it is called “not-approximate-equal” instead of “approximate-not-equal”.) Approximate comparisons can not be applied to strings.

## B.2 Programmatic Semantics

The following expression types are *programmatic expressions*. For programmatic expressions, the evaluation of operands may be deferred or skipped depending on other operands.

`EXPR_COND` (Conditional operator)
: A conditional expression is always replaced by an evaluated expression. The first operand of a conditional expression is interpreted as a truth value. If the truth value of the first operand is undefined (i.e. the operand is of type `vref` or `expr`), then the operand is evaluated once and checked again. Depending on the truth value, one of the following actions is taken:

  **TRUE**
  : The second operand of the conditional expression is evaluated and the expression is replaced by the result of that evaluation. The third operand is discarded without being evaluated.

  **FALSE**
  : The third operand of the conditional expression is evaluated and the expression is replaced by the result of that evaluation. The second operand is discarded without being evaluated.

  Undefined
  : The conditional expression is replaced with a conditional expression where the first operand is replaced with the evaluated first operand and the second and third operand are kept. The class of the expression (if present) is kept.

[8] As ca consequence, the result of a comparison of two empty arrays is always **TRUE**.

`EXPR_SEQ` (Sequence operator)

A sequence expression is always replaced by an evaluated expression. The first operand of the sequence expression is evaluated and the result is discarded[9]. The second operand is also evaluated and the sequence expression is replaced with the result of that evaluation.

`EXPR_SEL` (Selection operator)

A selection expression is always replaced by an evaluated expression. The selection operator is evaluated by performing a selection operation as defined in section 5.1.1 “Address Resolution Operations” on page 23. The *dictionary* is the result of the evaluation of the first operand, the *selector* is the result of the evaluation of the second operand. The selection expression is replaced with the result of the selection operation.[10]

`EXPR_INDEX` (Index operator)

An index expression is evaluated only if the second operand is an array with all keys **NIL** holding exactly one or exactly two elements.

If the second operand is a one element array, the index expression is evaluated by performing an index operation as defined in section 5.1.1 “Address Resolution Operations”. The *sequence* is the result of the evaluation of the first operand expression is replaced with the result of the index operation, the *index* is the result of the evaluation of the array element of the second operand. The index expression is replaced with the result of the index operation.

If the section operand is a two element array, the index expression is evaluated by performing a slice operation as defined in section 5.1.1 “Address Resolution Operations”. The *sequence* is the result of the evaluation of the first operand, the *lower bound* is result of the evaluation of the first array element of the second operand, the *upper bound* is the result of the evaluation of the second element of the second operand. The index expression is replaced with the result of the slice operation.

`EXPR_CALL` (Call operator)

A call expression is evaluated only if the first operand is a selection expression and the second operand is an array.[11] A call operation is performed *without* prior evaluation of the second operand.

The result of the evaluation of the first operand of the first operand of the call expression (i.e. the first operand of the selection expression) is called the *target* of the call expression; the result of the evaluation of the second operand of selection expression is called the *selector* if the call expression; the (unevaluated) second operand of the call expression is called the *argument array* of the call expression.

The call expression is evaluated using the `ClassEnv::call()` method of the class environment associated with the target of the call expression. If the target has no class name or no class environment matches the class name of the target and the environment object provides a default class environment, that default class environment is used to evaluate the call expression. If no class environment matches the target and no default class environment is available, the call expression is replaced by call expression where the first operand of the selection expression is replaced with the target.

[9] Note that the evaluation may have a side effect.

[10] If the selection itself should be an expression, the `quote` method from the standard default class (see section C.1) can be used to quote the expression.

[11] By definition, the second operand of a call expression is always of type `array`.

# C Standard Object Classes

The standard environment provides class environments for a set of class names and a default class environment.

All of the class environments below implement the `ClassEnv::call()` method. Every call method dispatches a number of method names. If the `call()` method is called with a non-string selector or an unrecognized selector, or if the argument array does not match the formal parameter definition of the dispatched method, the `call()` method returns a call expression equivalent to the call expression being evaluated, but with a clear class name.[12] For every class, a list of method prototypes is defined.

## C.1 The Standard Default Class

The standard default class environment implements the `ClassEnv::eval()` and `ClassEnv::call()` methods. The `eval()` method implements the standard expression semantics defined in appendix B.

The standard default class environment defines the following methods:

quote( *object* )
: The method returns the specified parameter *object* as is. The called object is not evaluated and is typically specified as **NIL**. The purpose of this method is to prevent an expression object from being evaluated (for example for specifying an expression as a selector in a selection operation). Example (in `expression` context):

```
$(exprdict).(nil.quote(($a + $b)))
```

The expression `($a + $b)` is used as the selector in the selection operation performed when the selection expression is evaluated.

## C.2 The `time` Class Environment

The `time` class represents an absolute time and date information. Instances of the `time` class use the ISO 8601 format for representing a time in a serialized form. For living objects, either a string representing (in ISO format) or an epoch representation is used.

Epoch Representation
: The time is represented as the number of milliseconds since the epoch date January 1st 1970, 00:00 UTC.[13]

ISO 8601 Representation
: The time is represented as a string in ISO 8601 format. An exact definition of the time format is given in section C.2.1.

The `ClassEnv::pack()` method converts time objects in epoch representation to ISO format.[14] The `ClassEnv::unpack()` method converts an ISO time representation as defined in section C.2.1 to epoch representation. The `ClassEnv::eval()` method is overridden to implement simple arithmetic on time objects (as defined in section C.2.4).

[12] If the call expression being evaluated is associated with a class name, that class name will be attached to the resulting object. Hence, if the call can not be dispatched, a call expression equal to the expression being evaluated is created by the evaluation.

[13] This is equivalent to the Java time representation.

[14] Complete calendar date representation as defined in section 5.4.1 a) of the ISO 8601:2000 standard, using UTC representation (`Z` suffix).

### C.2.1 The ISO 8601 Time and Data Format

The string representation for dates used by the `time` class environment uses a subset of the ISO 8601:2000 standard. An instance of a time object is encoded as a combination of an ISO date and an ISO time of day as defined in section 5.4 of the standard. This definition makes no assumptions about extra agreements about the time representation. Truncated and expanded representations are not valid. Decimal fraction notations for hours, minutes, or seconds are allowed. The fraction should be separated by a comma, but a period must also be accepted by the decoder.

Encoding the ISO Format

The encoder of the ISO format (as implemented by the `ClassEnv::pack()` method) must generate a *extended complete* calendar date representation as defined in section 5.4.1 a) of the ISO 8601:2000 standard, using the UTC time. If the time can not be represented with 1 second precision, the decimal fraction part of the second is separated by a comma. For example, the 27th of August 2002, 16:47:00 and 834 milliseconds UTC is represented as:

```
2002-08-27T16:47:00,834Z
```

The same time without the milliseconds fraction is represented as:

```
2002-08-27T16:47:00Z
```

Decoding the ISO Format

The decoder must be capable of decoding a time and date string conforming to the ISO 8601:2000 standard, section 5.4 to an epoch representation, excluding truncated time and/or date representations (as defined in sections 5.2.3.3 and 5.3.1.4 of the standard) and expanded date representations (as defined in section 5.2.3.4 of the standard). Note that time of day representations with reduced precision and local time representations and decimal fraction representations must be decoded as defined in the standard.

Examples:

| | |
|---|---|
| `20020827T1647Z` | Basic representation with reduced precision in the time of day. |
| `1985102T1015` | Local time in ordinal representation (calendar day 102 of the year 1985). |
| `1985-W15-1T10:15+04` | Extended week date (monday of calendar week 15), time of day with reduced precision, UTC + 4 hours. |

### C.2.2 The Epoch Representation

The epoch representation encodes the (positive or negative) number of milliseconds since the 1st of January 1970, 00:00 midnight UTC. The calculation is done on a formula ignoring leap seconds, leap years are accounted for as defined in official standards. (Note that including leap seconds in the conversion folmula is not feasable, since it is hardly possible to predict for which years the IERS (International Earth Rotation Service) will announce a leap second.)

Note that implementations based on a POSIX.1 compliant implementation of the standard C library functions `mktime()`, `gmtime()`, and `localtime()` will not handle leap years correctly for dates before the year 1901 and after the year 2099.

### C.2.3 Methods of the `time` Class

The `time` class environment provides methods for creating, converting, and evaluating time objects. Some of the methods use an array representation of a time. This representation is inspired by (but not compatible to) the `struct tm` of the standard C library. This representation is a (classless) array with the following fields:

`year`
: The year number. This should be an integer holding the year.[15]

`month`
: The month, represented as an integer number in the range [1..12] (January is represented by the number 1, December is represented by the number 12). The number 0 represents an undefined month.

`week`
: The calendar week. The first calendar week (week number 1) is the week containing the 4th of January of the specified year.[16] The number 0 represents an undefined week.

`yday`
: The day of the year. The 1st of January is day 1. The number 0 represents an undefined day of the year.

`mday`
: The day of the month, represented an integer in the range [1..31]. The number 0 represents an undefined day of the month.

`wday`
: The day of the week, represented as an integer in the range [1..7]. Monday is day 1 and Sunday is day 7.[17] The number 0 represents an undefined day of the week.

`hour`
: The hour of the day. This is an integer in the range [0..24].[18]

`min`
: The minute of the hour. This is an integer in the range [0..59].

`sec`
: The second of the minute. This is an integer in the range [0..60].[19]

`msec`
: The millisecond of the second. This is an integer in the range [0..999].

`tz`
: The local timezone represented as a difference to UTC, specified in hours. The value **NIL** represents local time.

`tzmin`
: The local timezone represented as an additional offset to UTC, specified in minutes. This is an integer in the range [0..59].[20]

Note that representations of week dates and calendar/ordinal dates are not compatible. The days 1st of January through 3rd of January may belong to the last calendar week of the previous year, so the week date and the calendar date representation of the day may disagree on the year number. Hence, a time array may either represent a calendar/ordinal date or a week date. If the field `week` is defined (non zero), then the fields `month`, `mday`, and `yday` *must* be set to undefined (value 0).

The following methods are provided by the `time` class environment:

[15] Note that in contrast to `struct tm` of the standard C library, the specified number is *not* interpreted relative to 1900.
[16] A week starts with Monday and ends with Sunday.
[17] Note that in contrast to `struct tm` of the standard C library, the value 0 is *not* an alternate representation for Sunday.
[18] `hour=24`, `min=0`, `sec=0`, `msec=0` is a valid encoding for midnight.
[19] The value `sec=60` is required to encode leap seconds.
[20] The ISO 8601:2000 standard allow timezone offsets with minute accuracy.

create( *tm* )
: This method discards the called object (typically `{time}nil`) and interprets the parameter *tm* as a time array. The time array specified through the *tm* parameter should hold a combination of elements uniquely specifying a point in time. If any of the fields `hour`, `min`, `sec`, `msec`, `tz`, or `tzmin` are missing, they are considered to have the integer value `0`. Of the other fields, one of the following combinations must be present:

  - `year`, `month`, `mday` (calendar date representation).
  - `year`, `yday` (ordinal date representation).
  - `year`, `week`, `wday` (week date representation).

  The fields are checked in the order listed above. If unneeded fields are specified, these fields are ignored. For example, if the `year`, `yday`, `wday`, and `week` fields are present, the `wday` and `week` fields are ignored, since the ordinal representation takes precedence over the week date representation.

  The method returns a time object in epoch representation.

split( *tz* `=` **`NIL`**, *weekdate* `=` **`FALSE`**)
: Split a time object to array representation. The parameter *tz* is a time array containing the requested time zone (`tz` and/or `tzmin` fields). If the parameter *tz* is **NIL**, local time is assumed.

  The parameter *weekdate* is interpreted as a truth value (as defined in appendix B). If *weekdate* is **TRUE**, the date is decoded to a week date; if *weekdate* is **FALSE**, the date is decoded to a calendar/ordinal date. The field `wday` is set independed from the *weekdate* parameter.

  The function returns a complete classless time array as defined above.

encode( *tz* `=` **`NIL`**)
: Encode a time object using the ISO representation. The parameter *tz* is a time array containing the requested time zone (`tz` and/or `tzmin` fields). If the parameter *tz* is **NIL**, local time is assumed.

decode()
: Decode a time object from ISO representation to epoch representation.

### C.2.4 Time Expression Evaluation

Expressions including time objects can be used to perform simple arithmetic on time objects (addition and subtraction of deltas, computation of time deltas, ordered and equality comparisons of time objects). Whenever a time delta is computed, it is specified as a floating point number of class `time delta`.

For time arithmetic, a time delta may be specified as a number (measured in seconds) or a string. If a delta is specified as a floating point number, the number is rounded to millisecond precision. If the delta is specified as a string, it is interpreted as follows:

1. A delta specification is a (possibly empty) list of time delta elements, separated by whitespace and/or commas.

2. A time delta elements is an integer number followed by an optional unit specifier. The integer number may be written in decimal (starting with a non-zero digit), octal (starting with a zero digit), or hex (starting with the character sequence `0x` or `0X`).

Unit specifiers are case-insensitive, i.e. “`s`” is equivalent to “`S`”. The following unit specifiers are recogized:

`ms` The delta is specified in milliseconds.

`s` The delta is specified in seconds.

`m` The delta is specified in minutes.

`h` The delta is specified in hours.

`d` The delta is specified in days.

If a delta element does not contain a unit specifier, seconds are assumed.

The following expression types are recogized by the time evaluation:

`EXPR_PLUS` (Time and delta addition)
: The evaluated second operand is interpreted as a time delta specification, either as a number or as a string. The result is a time object representing the result of the time plus delta addition. If the delta is specified as a number, it may be negative. If the second operand is not a number or string holding a valid time delta specification, the expression is not evaluated.

`EXPR_MINUS` (Time and delta subtraction, time delta)
: If the evaluated second operand is a number or string not of class `time`, the result is a time object representing the result of the time minus delta subtraction. If the second operand is of class `time`, the difference between the two time objects is computed an returned as a floating point number of class `time delta` (measured in seconds, up to a precision of milliseconds).

`EXPR_LS` (Time comparison earlier-than)
: If both operands are of type `time`, the expression is evaluated to a boolean representin the result of an “earlier-than” comparison.

`EXPR_LE` (Time comparison earlier-or-equal)
: If both operands are of type `time`, the expression is evaluated to a boolean representin the result of an “earlier-or-equal” comparison.

`EXPR_GT` (Time comparison later-than)
: If both operands are of type `time`, the expression is evaluated to a boolean representin the result of an “later-than” comparison.

`EXPR_GE` (Time comparison later-or-equal)
: If both operands are of type `time`, the expression is evaluated to a boolean representin the result of an “later-or-equal” comparison.

`EXPR_EQ` (Time comparison equal)
: If both operands are of type `time`, the expression is evaluated to a boolean representin the result of an “equal” comparison.

`EXPR_NE` (Time comparison not-equal)
: If both operands are of type `time`, the expression is evaluated to a boolean representin the result of an “not-equal” comparison.

Note that both operands are evaluated for all operations on `time` objects.

# D Simple Objects API

This section defines a language neutral inteface for simple object APIs. The APIs are defined in a syntax simple to that of the *Java* programming language. The API consits of two parts, a set of functions/methods operating on simple object serializations and a set of methods operating on instances of a simple object class, holding an in-memory representation of a simple object. Instances of such a simple object class are called *runtime objects*.

A programming library typically provides a set of hooks for callback functions/methods. These callback hooks should not be held in a global and static location. Instead, objects instanciated from an environment class should be used to hold all these callbacks. Such objects are called *environment objects*.

Every runtime object should reference such an environment object. All methods operating on runtime objects and accessing the environment should be provided in two variants: one using the environment object referenced by the runtime object and one using an environment object specified by the caller. In languages supporting default values for method parameters, these two variants may be implemented using a defaulted parameter.

## D.1 The Environment Class

The *environment class* is the base class for environment objects. In an object-oriented environment, the environment object would be an instance of a class derived from the environment class. The environment class provided by the simple objects library would provide default implementations of the callbacks methods.

The environment class provided by the API should be equivalent to the following class `Env`:

```
class Env
{
  static final int MSG_ERR = 1;
  static final int MSG_WARN = 2;
  static final int MSG_INFO = 3;
  static final int MSG_DEBUG = 4;

  bool error;

  int message(int msgType, string msgID,
      Dictionary msgArgs, string message) { return -1; }

  string str(SObject object) { return null; }

  SObject vref(string ref) { return null; }

  string vref_str(string ref) {
    SObject object = vref(ref);
    if (object != null) return str(object); else return null;
  }

  Dictionary classEnv;

  // ...

  final void add_class(ClassEnv classEnv);
  final ClassEnv find_class(string name);
};
```

The methods implemented in the actual environment class derived from `Env` should implement the following functionality:

**int** message(**int** *msgType*, **string** *msgID*, **Dictionary** *msgArgs*, **string** *message*)

Process a message of the specified type *msgType*. The message should be sent to a logging facility and/or terminal.
Parameters:

**int** *msgType*

The message type. This is one of the following constants:

`MSG_ERR`
: An error message. An operation failed due to an error condition.

`MSG_WARN`
: A warning message.

`MSG_INFO`
: An informational message. Info messages may be discarded in a production environment.

`MSG_DEBUG`
: A debugging message. Debug messages should be discarded in a production environment.

**string** *msgID*

A message identification string. A list of message identification strings is given in section D.5. Message IDs are language neutral and may be used for internationalization.

**Dictionary** *msgArgs*

A set of key/value pairs holding additional information about the event. The keys are all plain ASCII strings and the values are serializations of simple objects.

**string** *message*

A short message text in English. This message text should contain all relevant information about the event.

The return value indicates if a default message processing should be performed. The default message processing will write error (`MSG_ERR`) and warning (`MSG_WARN`) messages to the standard error of the process and discard all other messages.
Return values:

-1
: Perform a default messages processing.

0
: Message processed, OK. (No default processing is performed.)

All other return values are reserved and should not be returned by the implementation of `message()`.

**string** str(**SObject** *object*)

Create a UNICODE UTF-8 string representation of a runtime object (the returned string should *not* be an extended UNICODE string). The return value `null` indicates, that a default string representation should be used.

**SObject** vref(**string** *ref*)

Resolve a variable reference. The parameter *ref* specifies the reference string. The return value `null` indicates that the variable reference should not be substituted, i.e. the variable reference is kept unchanged.

**string** vref_str(**string** *ref*)

Resolve a variable reference in an UTF-8 encoded UNICODE string (*not* an extended string). The method implementations defaults to a combination of the `vref()` and `str()` methods.

The following fields are defined in the environment class:

`MSG_ERR`, `MSG_WARN`, `MSG_INFO`, `MSG_DEBUG`
: Constants for the *msgType* parameter of the `message()` method (see above).

**bool** *error*
: This flag is set when the `message()` method is called with the *msgType* set to `MSG_ERR`.

**Dictionary** *classEnv*
: If not `null`, the *classEnv* dictionary holds a set of class environment objects (see section D.2). The dictionary maps the class names (*class* field of a simple object) to an instance of a class derived from `ClassEnv`.

  **Note:** This field should be treated as a read-only instance variable, even if a specific implementation allows modifications to *classEnv*.

The class environments of an environment object are managed using the following methods. These methods are provided by the simple object implementation and must not be overridden in derived classes.

**void** add_class(**ClassEnv** *classEnv*)
: Add a class environment to an environment object. If the class name of the specified class environment is not set or empty, the specified class environment is installed as the default class environment.

  **ClassEnv** *classEnv*
  : The class environment to be added to the environment object.

**ClassEnv** find_class(**String** *name*)
: Find the class environment associated with the specified class name. If no class environment with the specified name is found and if a default class environment is installed, the default class environment is returned. The method returns `null` if the name is not bound and no default class environment is installed.

  **String** *name*
  : The class name of the requested class environment.

In a programming environment not supporting classes and inheritance, the environment objects should be implemented as a collection of function objects (e.g. function pointers in C) with a defined null-value indicating the default behaviour.

## D.2 The Class Environment Class

The *class environment class* is the base class for objects defining simple object classes. These instances are referenced by the *classEnv* field of an environment object.

The class environment class provided by a simple objects API should be equivalent to the following class:

```
class ClassEnv
{
  string name;

  SObject pack(SObject object, Env env = null) { return object; }

  SObject unpack(SObject object, Env env = null) { return object; }

  SObject call(SObject object, SObject method,
      SObject args, Env env = null) { return SObject.NIL(); }

  SObject eval(SObject expression) { return null; }
};
```

The field named *name* holds the name of the class, or `null` if the class environment applies to all objects without a matching class environment. The methods implemented in the actual environment class derived from `ClassEnv` should implement the following functionality:

**SObject** pack(**SObject** *object*, **Env** *env* `= null`)
This method is called before the object is serialized.
Parameters:

**SObject** *object*
The simple object being serialized.

**Env** *env* (default value `null`)
The environment object for the operation. This environment object should be passed to all operations performed on the object. A null-pointer (value `null`) indicates that the default environment should be used.

The return value is the object transformed for serialization. If no transformation is required, the function should return the object *object* (this is the default behaviour for all objects).

**SObject** unpack(**SObject** *object*, **Env** *env* `= null`)
This method is called after the object is deserialized.
Parameters:

**SObject** *object*
The simple object being deserialized.

**Env** *env* (default value `null`)
The environment object for the operation. This environment object should be passed to all operations performed on the object. A null-pointer (value `null`) indicates that the default environment should be used.

The return value is the object transformed after deserialization. If no transformation is required, the function should return the object *object* (this is the default behaviour for all objects).

**SObject** call(**SObject** *object*, **string** *method*, **SObject** *args*, **Env** *env* `= null`)
This method implements method calls on simple objects of the defined simple object class.
Parameters:

**SObject** *object*
The simple object instance the method is called on.

**SObject** *method*
The method identifier.

**SObject** *args*
The arguments passed to the method call. A null-pointer indicates that the method was called without arguments. If *args* is not a null-pointer, it is always of type `array`.

**Env** *env* (default value `null`)
The environment object for the operation. This environment object should be passed to all operations performed on the object. A null-pointer (value `null`) indicates that the default environment should be used.

If the method call succeeds, the method should return a simple object representing the return value of the method call. If the method call failed, the method call should return a null-pointer.

If a null-pointer is returned (error indication), the *error* flag of the corresponding environment object is set. The method is responsible for propper error reporting (e.g. by calling the `message()` method from the environment object).

**SObject** eval(**SObject** *expression*, **Env** *env* `= null`)

This method implements the expression evaluation function for the object class.
Parameters:

**SObject** *expression*
: The expression object to be evaluated. This is always an object of type `expr`. The first operand of the expression is the evaluated operand of the original expression.[21]

**Env** *env* (default value `null`)
: The environment object for the operation. This environment object should be passed to all operations performed on the object. A null-pointer indicates that the default environment should be used.

The method should return the evaluated object. If the expression can not and should not be evaluated, the method should return the parameter *expression* as is.

If the method returns `null`, an evaluation is performed on all subexpressions.

**Note:** The `eval()` method of the class environment is responsible for performing recursive evaluation of subexpressions[22].

### D.2.1 The Default Class Environment

A class environment object without a class name (i.e. the *name* field is `null`) is called the *default class environment*. If a default class environment is present, it is used as the class environment for all classes without a matching class environment.

**Note:** For objects without an associated class name, the default class environment is used when present in the environment object.

## D.3 Runtime Objects

Runtime objects are represented by instances of the API class `SObject`. Instances of the runtime object class `SObject` are immutable. That means, that the value of a simple object never changes after the object is created. The API should be designed in a way that avoids the creation of unnecesseary temporary objects.

### D.3.1 Factories

Objects of all types may be created using the following static factory methods:

```
class SObject
{
  // ...
  static final int EXPR_POS = 1;
  static final int EXPR_NEG = 2;
  static final int EXPR_NOT = 3;
  static final int EXPR_PLUS = 4;
  static final int EXPR_MINUS = 5;
  static final int EXPR_MUL = 6;
  static final int EXPR_DIV = 7;
  static final int EXPR_MOD = 8;
  static final int EXPR_CAT = 9;
  static final int EXPR_LS = 10;
  static final int EXPR_LE = 12;
```

[21] The simple objects library needs to evaluate the first operand anyway to dispatch the evaluation to the correct class environment.
[22] This enables the `eval()` method to perform conditional evaluation of subexpressions.

```
  static final int EXPR_GT = 14;
  static final int EXPR_GE = 16;
  static final int EXPR_EQ = 18;
  static final int EXPR_EQ_APPROX = 19;
  static final int EXPR_NE = 20;
  static final int EXPR_NE_APPROX = 21;
  static final int EXPR_AND = 22;
  static final int EXPR_OR = 23;
  static final int EXPR_COND = 24;
  static final int EXPR_SEQ = 25;
  static final int EXPR_SEL = 26;
  static final int EXPR_INDEX = 27;
  static final int EXPR_CALL = 28;
  // ...
  static SObject NIL(string cn = null);
  static SObject BOOL(bool value, string cn = null);
  static SObject INT(long value, string cn = null);
  static SObject FLOAT(double value, string cn = null);
  static SObject STRING(string value, string cn = null);
  static SObject STRING_raw(string value, string cn = null);
  static SObject STRING_ucx(byte[] value, string cn = null);
  static SObject BINARY(SObject id, byte[] body, string cn = null);
  static SObject ARRAY(SObject[] key, SObject[] value, string cn = null);
  static SObject VREF(string ref, string cn = null);
  static SObject VREF_raw(string ref, string cn = null);
  static SObject VREF_ucx(byte[] ref, string cn = null);
  static SObject EXPR(int opType, SObject op1,
      SObject op2 = null, SObject op3 = null,
      string cn = null);
  // ...
};
```

static **SObject** NIL(**string** *cn* `= null`)

Create a runtime object representing a **NIL** object (a simple object of type `nil`).
Parameters:

**string** *cn* (default value `null`)
If not `null`, the parameter *cn* specifies the class name of the new object.

static **SObject** BOOL(**bool** *value*, **string** *cn* `= null`)

Create a runtime object representing a simple object of type `bool`.
Parameters:

**bool** *value*
The value.

**string** *cn* (default value `null`)
If not `null`, the parameter *cn* specifies the class name of the new object.

static **SObject** INT(**long** *value*, **string** *cn* `= null`)

Create a runtime object representing a simple object of type `int`.
Parameters:

**long** *value*
The numeric value.

**string** *cn* (default value `null`)
If not `null`, the parameter *cn* specifies the class name of the new object.

static **SObject** FLOAT(**double** *value*, **string** *cn* `= null`)
Create a runtime object representing a simple object of type `float`.
Parameters:

**double** *value*
The numeric value.

**string** *cn* (default value `null`)
If not `null`, the parameter *cn* specifies the class name of the new object.

static **SObject** STRING(**string** *value*, **string** *cn* `= null`, **Env** *env* `= null`)
Create a runtime object representing a simple object of type `string`.
Parameters:

**string** *value*
The string value. The value is interpreted as a text serialization in `string` context.

**string** *cn* (default value `null`)
If not `null`, the parameter *cn* specifies the class name of the new object.

**Env** *env* (default value `null`)
The environment object. The special value `null` represents the default environment.

static **SObject** STRING_raw(**string** *value*, **string** *cn* `= null`)
Create a runtime object representing a simple object of type `string`.
Parameters:

**string** *value*
The string value. The value is interpreted as a UNICODE string.

**string** *cn* (default value `null`)
If not `null`, the parameter *cn* specifies the class name of the new object.

**Note:** String objects created with this factory can not contain variable references.

static **SObject** STRING_ucx(**byte[]** *value*, **string** *cn* `= null`)
Create a runtime object representing a simple object of type `string`.
Parameters:

**byte[]** *value*
An extended string value. The value is interpreted as an extended UNICODE string as defined in section 2.3 on page 7. If the specified string is not a valid extended UNICODE string, it is interpreted as a regular UNICODE string.

**string** *cn* (default value `null`)
If not `null`, the parameter *cn* specifies the class name of the new object.

static **SObject** BINARY(**SObject** *id*, **byte[]** *body*, **string** *cn* `= null`)
Create a runtime object representing a simple object of type `array`. Parameters:

**SObject** *id*
The type ID object of the binary object (the *id* field of the simple object *data*).

**byte[]** *body*
The *body* field of the simple object *data*. The value `null` is equivalent to an empty body.

**string** *cn* (default value `null`)
If not `null`, the parameter *cn* specifies the class name of the new object.

static **SObject** ARRAY(**SObject[]** *key*, **SObject[]** *value*, **string** *cn* `= null`)
Create a runtime object representing a simple object of type `array`. The array object created will be empty (i.e. not contain any array elements).
Parameters:

**SObject[]** *key*
The array of element keys. The special value `null` indicates that all keys are **NIL**.

**SObject[]** *value*
The array of values. The special values `null` indicates that am empty array should be created.

**string** *cn* (default value `null`)
If not `null`, the parameter *cn* specifies the class name of the new object.

The length of the resulting array depends *only* on the *value* parameter. If the *key* array is shorter than the *value* array, the extra elements in the *value* array are associated with a **NIL** key. If the *key* array is longer than the *value* array, the extra elements in the *key* array are ignored.

static **SObject** VREF(**string** *ref*, **string** *cn* `= null`)
Create a runtime object representing a simple object of type `vref`.
Parameters:

**string** *ref*
The reference string of the variable reference. The specified string is converted to UTF8 and interpreted as a text serialization in `string` context. The represented string is taken as the reference string of the new object.

**string** *cn* (default value `null`)
If not `null`, the parameter *cn* specifies the class name of the new object.

static **SObject** VREF_raw(**string** *ref*, **string** *cn* `= null`)
Create a runtime object representing a simple object of type `vref`.
Parameters:

**string** *ref*
The reference string of the variable reference. The specified string is interpreted as a raw UNICODE string (i.e. not containing nested variable references).

**string** *cn* (default value `null`)
If not `null`, the parameter *cn* specifies the class name of the new object.

static **SObject** VREF_ucx(**byte[]** *ref*, **string** *cn* `= null`)
Create a runtime object representing a simple object of type `vref`.
Parameters:

**byte[]** *ref*
The reference string of the variable reference. The specified string is interpreted as an extended UNICODE string as defined in section 2.3 on page 7. If the specified string is not a valid extended UNICODE string, it is interpreted as a regular UNICODE string.

**string** *cn* (default value `null`)
If not `null`, the parameter *cn* specifies the class name of the new object.

static **SObject** EXPR(**int** *opType*, **SObject** *op1*, **SObject** *op2* `= null`, **SObject** *op3* `= null`, **string** *cn* `= null`)
Create a runtime object representing a simple object of type `expr`.
Parameters:

**int** *opType*
The expression type. The constants representing expression types are summarized in table 2.

**SObject** *op1*
The first operand of the expression.

**SObject** *op2* (default value `null`)
The second operand of the expression. If the expression type requires only one operand, this parameter *must* be `null`.

| Constant | Operands | Expression Type |
|---|---|---|
| EXPR_POS | 1 | Unary positive operator |
| EXPR_NEG | 1 | Unary negation operator |
| EXPR_NOT | 1 | Unary logical negation operator (NOT) |
| EXPR_PLUS | 2 | Binary plus operator |
| EXPR_MINUS | 2 | Binary minus operator |
| EXPR_MUL | 2 | Multiplication operator |
| EXPR_DIV | 2 | Division operator |
| EXPR_MOD | 2 | Modulo operator |
| EXPR_CAT | 2 | Concatenation operator |
| EXPR_LS | 2 | Less-than comparison operator |
| EXPR_LE | 2 | Less-or-equal comparison operator |
| EXPR_GT | 2 | Greater-than comparison operator |
| EXPR_GE | 2 | Greater-or-equal comparison operator |
| EXPR_EQ | 2 | Equals comparison operator |
| EXPR_EQ_APPROX | 3 | Equals approximate comparison operator |
| EXPR_NE | 2 | Not-equal comparison operator |
| EXPR_NE_APPROX | 3 | Not-equal approximate comparison operator |
| EXPR_AND | 2 | Logical AND operator |
| EXPR_OR | 2 | Logical OR operator |
| EXPR_COND | 3 | Conditional operator |
| EXPR_SEQ | 2 | Sequence operator |
| EXPR_SEL | 2 | Selection operator |
| EXPR_INDEX | 2 | Index operator |
| EXPR_CALL | 2 | Call operator |

Table 2: Expression Type Constants in Class `SObject`

**SObject** *op3* (default value `null`)
The third operand of the expression. If the expression type requires only one or two operands, this parameter *must* be `null`.

**string** *cn* (default value `null`)
If not `null`, the parameter *cn* specifies the class name of the new object.

### D.3.2 Object Modification

All so called *object modification* methods are in fact factories, since instances of `SObject` are immutable. E.g. removing an element from an array creates a new simple object representing a flat copy of the original array with the specified element removed.

**Note:** The objects created by the object modification methods below preserve the class name of the called object.

The following object modification methods are provided by the `SObject` class:

```
class SObject
{
  // ...
  SObject set_classname(string cn);
```

```
  SObject insert(int position, SObject key, SObject value);
  SObject insert(int position, SObject[] key, SObject[] value);
  SObject append(SObject key, SObject value);
  SObject append(SObject[] key, SObject[] value);
  SObject remove(int position, int count = 1);
  SObject remove(SObject key);
  SObject replace(int position, SObject key, SObject value);
  SObject put(SObject key, SObject value, Env env = null);
  SObject concat(SObject operand);

  SObject set(string address, SObject value,
      bool slice = false, Env env = null);
  SObject set(SObject address, SObject value,
      bool slice = false, Env env = null);

  SObject string_insert(int position, string str);
  SObject string_append(string str);
  SObject string_replace(int position, int length, string str);
  // ...
};
```

**SObject** set_classname(**string** *cn*)

Change the class name of an object. Parameters:

**string** *cn*
: The class name of the new object. If this is `null`, no class name will be associated with the new object.

An object identical to the called object except for the class name. The class name is substituted with the specified class name *cn*.

**SObject** insert(**int** *position*, **SObject** *key*, **SObject** *value*)

Insert an element (i.e. key/value pair) into an array object. It is a fatal runtime error if the called object is not of type `array`.
Parameters:

**int** *position*
: The index position for inserting the element. Negative values are counted from the end if the array. If *index* is equal to the length of the array, the element is appended to the array. It is a fatal runtime error if the index is out of bounds.

**SObject** *key*
: The key of the key/value pair. The value `null` is equivalent to a **NIL** object.

**SObject** *value*
: The value of the key/value pair. The value `null` is equivalent to a **NIL** object.

The method returns an array object with the specified element inserted.

**SObject** insert(**int** *position*, **SObject[]** *key*, **SObject[]** *value*)

Insert a sequence of elements (key/value pairs) into an array object. It is a fatal runtime error if the called object is not of type `array`.
Parameters:

**int** *position*
: The index position for inserting the sequence of elements. If *position* is equal to the length of the array, the elements are appended to the array. It is a fatal runtime error if the index is out of bounds.

**SObject[]** *key*
Array of element keys. The special value `null` indicates that all element keys are **NIL**.

**SObject[]** *value*
Array of values. The length of the array *value* indicates the number of elements being inserted. If the *value* array is longer than the *key* array, the unspecified keys are set to **NIL**. If the *value* array is shorter than the *key* array, the extra key values in *key* are ignored.

If the specified element sequence is empty, the called object remains unchanged. The method returns an array object with the specified elements inserted.

**SObject** append(**SObject** *key*, **SObject** *value*)
Append an element (i.e. key/value pair) to an array object. It is a fatal runtime error if the called object is not of type `array`.
Parameters:

**SObject** *key*
The key of the key/value pair. The value `null` is equivalent to a **NIL** object.

**SObject** *value*
The value of the key/value pair. The value `null` is equivalent to a **NIL** object.

The method returns an array object with the specified element appended.

**SObject** append(**SObject[]** *key*, **SObject[]** *value*)
Append a sequence of elements (key/value pairs) to an array object. It is a fatal runtime error if the called object is not of type `array`.
Parameters:

**SObject[]** *key*
Array of element keys. The special value `null` indicates that all element keys are **NIL**.

**SObject[]** *value*
Array of values. The length of the array *value* indicates the number of elements being appended. If the *value* array is longer than the *key* array, the unspecified keys are set to **NIL**. If the *value* array is shorter than the *key* array, the extra key values in *key* are ignored.

If the specified element sequence is empty, the called object remains unchanged. The method returns an array object with the specified elements appended.

**SObject** remove(**int** *position*, **int** *count* `= 1`)
Remove a number of elements from an array object. It is a fatal runtime error if the called object is not of type `array`.
Parameters:

**int** *position*
The inserted position of the first element being removed. Negative values are counted from the end of the array. It is a fatal runtime error if the index is out of bounds.

**int** *count* (default value `1`)
The number of elements being removed. This value must be positive or zero. If *count* is zero, the array object remains unchanged.

The method returns an array object with the specified elements removed.

**SObject** remove(**SObject** *key*, **Env** *env* `= null`)
Remove the last element whose key matches the specified *key*. It is a fatal runtime error if the specified object is not of type `array`.
Parameters:

**SObject** *key*
The specified element key.

**Env** *env* (default value `null`)
The environment object used for resolving variable references in the element keys of type `string`. If this is `null`, the default environment is used.

The array object remains unchanged if the specified key is not found. The method returns an array object with the specified element removed.

**Note:** Only the last element with a matching key is removed. The array might contain multiple elements with a matching key.

**SObject** replace(**int** *position*, **SObject** *key*, **SObject** *value*)
Replace an element in an array object. It is a fatal error if the called object is not of type `array`.
Parameters:

**int** *position*
The index position of the element being replaced. Negative values are counted from the end of the array. It is a fatal runtime error if the index is out of bounds.

**SObject** *key*
The replacement key of the specified element. The special value `null` indicates that the key should remain unchanged.

**SObject** *value*
The replacement value of the specified element. The special value `null` indicates that the value should remain unchanged.

The method returns an array object with the specified element altered.

**Note:** The special value `null` does *not* represent a **NIL** value.

**SObject** put(**SObject** *key*, **SObject** *value*, **Env** *env* `= null`)
Alter or add an element in/to an array object. It is a fatal error if the called object is not of type `array`.
Parameters:

**SObject** *key*
The key of the array element that should be altered. The last element in the array with a matching key is altered. If the specified key is not found in the array, the specified key/value pair is appended to the array.

**SObject** *value*
The value of the specified element. The special value `null` indicates a **NIL** value.

**Env** *env* (default value `null`)
The environment object used for resolving variable references in the address and in element keys of type `string`. If this is `null`, the default environment is used.

The method returns an array object with the specified element altered or added.

**SObject** concat(**SObject** *operand*)
Append an array object to an array object or a string object to a string object. It is a fatal runtime error if the called object is not if type `array` or type `string`.
Parameters:

**SObject** *operand*
The array object being appended. It is a fatal runtime error if the parameter *operand* is not of the same type as the called object (either `array` or `string`).

The method returns the concatenation of the called object and the operand.

The `set()` methods below replace a subobject of a specified simple object with another object. The position of the subobject is specified as an address (see section 5 on page 23 for a description of addresses). The address with the last operation stripped is called the *path* and the last operation is called the *selection*.

The path is resolved using *pure address resolution* (see section 5.2). Cases where the *path* can not be resolved are handled using the following set of rules:

1. If an index or selection operation is applied to a **NIL** object, that object is transparently replaced by an empty array object.
2. If a selection operation references an undefined key, that key is appended to the array and associated with the value **NIL**.
3. A slice operation selecting an empty slice from an array causes the key/value pair **NIL**/**NIL** to be inserted at the position of the *lower bound* of the slice. If the *lower bound* is equal to the length of the array, the **NIL**/**NIL** element is appended to the array. The selected object is the inserted/appended element.

If path can't be resolved directly or by using the rules above, the `set()` methods below will return `null`.

The selection is used to identity the subobject that should be replaced. For this operation, array indices and slices are handled in a special way: If the *slice* flag (passed to all variants of the `set()` method) is clear, the selected element or slice is replaced by a single element consiting of the key **NIL** and the specified value. If the *slice* flag is set, the selected element or slice is replaced by a sequence of elements specified as an object of type `array`. In this case the special *value* `null` is interpreted as an empty array (causing the selected element or slice to be removed).

If an array object is created transparently in the process of address resolution, these objects will *not* be associated with a class name.

**SObject** set(**string** *address*, **SObject** *value*, **bool** *slice* `= false`, **Env** *env* `= null`)

Substitute a subobject at the specified address.

**string** *address*

The parameter *string* is encoded as an UTF-8 byte array and interpreted as a text serialization of a simple object. If the first non-whitespace character in *string* is a dot (.) or an opening bracket ([), the string `"NIL"` (without the quotes) is prepended to the parameter. The represented simple object is interpreted as an address selecting the object to be substituted.

**SObject** *value*

The value to be substituted in place of the selected subobject.

**bool** *slice* (default value `false`)

Flag indicating if an element or slice addressed by an index or slice selection should be replaced by a single value or a sequence of elements (see above).

**Env** *env* (default value `null`)

The environment object used for resolving variable references in the address and in element keys of type `string`. The special value `null` causes the default environment to be used.

If the address can't be resolved, the function returns `null`. On succeeds, an object with the specified subobject replaced is returned.

**SObject** set(**SObject** *address*, **SObject** *value*, **bool** *slice* `= false`, **Env** *env* `= null`)

Substitute a subobject at the specified address.

**SObject** *address*

The address selecting the simple object to be substituted.

**SObject** *value*
The value to be substituted in place of the selected subobject.

**bool** *slice* (default value `false`)
Flag indicating if an element or slice addressed by an index or slice selection should be replaced by a single value or a sequence of elements (see above).

**Env** *env* (default value `null`)
The environment object used for resolving variable references in the address and in element keys of type `string`. The special value `null` causes the default environment to be used.

If the address can't be resolved, the function returns `null`. On succeeds, an object with the specified subobject replaced is returned.

The following string manipulation methods operate on simple objects of type `string`. The language type **string** should be a string type capable of storing UNICODE strings. Whenever a position with in string is specified through a *position* parameter, the parameter indicates the index position of a UNICODE character. Note that some characters may be represented by a sequence of UNICODE characters (e.g. accented characters), so the indicated position depends on how such characters are represented. See section 5.4 on page 27 for details about string normalization.

String objects represented as simple object instances are *extended UNICODE strings* (see section 2.3 on page 7). An extended strings may contain variable references. The string manipulation methods below treat variable references like a single UNICODE character.

**SObject** string_insert(**int** *position*, **string** *str*)
Insert a string into a string object. It is a fatal runtime error if the called object is not of type `string`.
Parameters:

**int** *position*
The character position for inserting the specified string. If the position is `-1` or points beyond the end of the string, the specified string *str* is appended to the called object. It is a fatal runtime error to pass a negative value smaller than `-1`.

**string** *str*
The string to be inserted.

The method returns a string object with the specified string inserted at the specified position.

**SObject** string_append(**string** *str*)
Append a string to the end of a string object. It is a fatal runtime error if the specified object is not of type `string`.
Parameters:

**string** *str*
The string to be appended.

The method returns a string object with the specified string appended.

**SObject** string_replace(**int** *position*, **int** *length*, **string** *str*)
Replace a substring of the specified string object. It is a fatal runtime error of the specified object is not of type `string`.
Parameters:

**int** *position*
The character position for replacing the specified string. If the position is `-1` or points beyond the end of the string, the specified string *str* is appended to the called object.

**int** *length*
The length of the substring to be replaced. For *length* == 0 this is equivalent to the `string_insert()` method. If *length* is larger than the rest of the string or `-1`, the entire rest of the string is replaced. It is a fatal runtime error to pass a negative value smaller than `-1`.

**string** *str*
The replacement string.

The method returns a string object with the specified substring replaced.

### D.3.3 Subobject Access

Subobjects of a simple object may be accessed as instances of the class `SObject`. The following subobject access methods are defined for the `SObject` class:

```
class SObject
{
  // ...
  SObject index(int index);
  SObject slice(int start, int end);
  SObject select(SObject key, Env env = null);
  SObject lookup(SObject key, Env env = null);
  SObject call(SObject args, Env env = null);

  int index_of(SObject key, Env env = null);

  SObject get(string address, bool pure = false, Env env = null);
  SObject get(SObject address, bool pure = false, Env env = null);

  string substring(int position, int length, Env env = null);
  // ...
};
```

**SObject** index(**int** *index*)
Perform an *index operation* as described in section 5.1.1, page 23. The called object has to be a valid *sequence* object for the operation. For an index operation, this is one of `array`, `string`, `expr`. If the called object is not a valid sequence, a classless index expression is created where the first operand is the called object and the second operand (the index array) holds a single integer object (type `int`) representing the value of the parameter *index*.

**int** *index*
The *index* in the index operation.

The method returns the result of the index operation.[23]

**SObject** slice(**int** *start*, **int** *end*)
Perform a *slice operation* as described in section 5.1.1, page 23. The called object has to be a valid *sequence* object for the operation. For a slice operation, this is either `array` or `string`. If the called object is not a valid sequence, a classless index expression is created where the first operand is the called object and the second operand (the index array) holds two integer objects (type `int`) representing the value of the parameters *start* and *end*.

**int** *start*
The (inclusive) *lower bound* of the slice operation.

**int** *end*
The (exclusive) *upper bound* of the slice operation.

[23] The result of the index operation will be **NIL** if the index is out of bounds.

The method returns the result of the slice operation.

**SObject** select(**SObject** *key*, **Env** *env* `= null`)
Perform a *selection operation* as described in section 5.1.1, page 23. The called object is the *dictionary* of the operation and has to be of type `array`. If the dictionary is not of type `array` or the dictionary does not contain the specified key, the object returned is a (classless) selection expression where the first operand is the called object and the second operand is the selector.

**SObject** *key*
The *selector* of the selection operation.

**Env** *env* (default value `null`)
The environment object used for comparison of the keys with the specified selector. The special value `null` represents the default environment.

The method returns the result of the selection operation.

**SObject** lookup(**SObject** *key*, **Env** *env* `= null`)
Perform a dictionary lookup operation. It is a fatal runtime error of the called object is not of type `array`.

**SObject** *key*
The key of the desired element.

**Env** *env* (default value `null`)
The environment object used for comparison of the keys with the specified selector. The special value `null` represents the default environment.

The method returns the value bound to the specified key or `null` if the specified key is not bound.

**SObject** call(**SObject** *args*, **Env** *env* `= null`)
Perform a method call operation on the called object. The called object should be a selection expression. The selector of that expression (the second operand) is interpreted as a method selector. If a class environment object (see section D.2 on page 43) is associated with the first operand of the called expression object, the `call` method of the class environment is called.

**Note:** It is a fatal runtime error if the parameter *args* is not of type `array`.

**SObject** *args*
The arguments operand of the call. This is an object of type `array` holding the call arguments (positional or named or both). It is a fatal runtime error if the value of *args* is not of type `array`.

**Env** *env* (default value `null`)
The environment object passed the to `call` method of the class environment. The special value `null` represents the default environment.

If the call operation is handled by the `call` method of a class environment, the return value of that call is passed on to the caller. If no class environment is associated with the called object or the called‘ object is not a selection expression, the return value is a call expression object where the first operand is the called object an the second operand is the value of the *args* parameter.

**int** index_of(**SObject** *key*, **Env** *env* `= null`)
Return the index of the element associated with the specified key. It is a fatal runtime error if the called object is not of type `array`.

**SObject** *key*
The key to look for.

**Env** *env* (default value `null`)
The environment object used for comparing the keys. The value `null` represents the default environment.

The method returns the index of the *last* element in the array matching the specified key or `-1` if the specified key was not found.

The following `get()` methods use addresses to select a subobject from a simple object. Addresses and the address resolution process is defined in section 5 (page 23). The `get()` methods offer an optional *pure* flag indicating *pure address resolution*. Pure addresses and the pure address resolution process are defined in section 5.2 (page 26).

If the address objects passed to the `get()` methods contains method calls and if the *pure* flag is clear, the method call subexpressions are substituted using the `call` method of the called object.

**SObject** get(**string** *address*, **bool** *pure* `= false`, **Env** *env* `= null`)
Get a subobject of the called object.

**string** *address*
The address of the subobject. The string *address* is converted to UTF-8 and interpreted as a text serialization of a binary object (see section 3.1). If the first non-whitespace character of *address* is a dot (.) or opening bracket ([), the string "`NIL`" is prepended to the address.

**bool** *pure* (default value `false`)
Flag indicating that pure address resolution should be used.

**Env** *env* (default value `null`)
The environment object used. The value `null` represents the default environment.

The method returns the specified subobject. The function returns `null` if the *pure* flag was selected an the address does not resolve to a subobject of the called object.

**SObject** get(**SObject** *address*, **bool** *pure* `= false`, **Env** *env* `= null`)
Get a subobject of the called object.

**SObject** *address*
The address of the subobject.

**bool** *pure* (default value `false`)
Flag indicating that pure address resolution should be used.

**Env** *env* (default value `null`)
The environment object used. The value `null` represents the default environment.

The method returns the specified subobject. The function returns `null` if the *pure* flag was selected an the address does not resolve to a subobject of the called object.

The following string selection methods operate on simple objects of type `string`. The language type **string** should be a string type capable of storing UNICODE strings. Whenever a position with in string is specified through a *position* parameter, the parameter indicates the index position of a UNICODE character. Note that some characters may be represented by a sequence of UNICODE characters (e.g. accented characters), so the indicated position depends on how such characters are represented. See section 5.4 on page 27 for details about string normalization.

**string** substring(**int** *position*, **int** *length*, **Env** *env* `= null`)
Get a substring from the called object. The called object has to be of type `string`. It is a fatal runtime error if the called object in not of type `string`.

**int** *position*
The position of the first character of the substring. Negative values are counted from the end of the string (e.g. the value `-1` selects the last character of the string. It is a fatal runtime error if the position is out of bounds.

**int** *length*
The length of the selected substring (number of UNICODE characters). If the value is `-1` or larger that the number of UNICODE characters following the specified position, the entire rest of the string is selected. Values smaller than `-1` cause a fatal runtime error.

**Env** *env* (default value `null`)
The environment object used for resolving variable references in the string. The value `null` represents the default environment.

The method returns the selected substring.

### D.3.4 Object Evaluation

```
class SObject
{
  // ...
  static final int TYPE_NIL = 0;
  static final int TYPE_BOOL = 1;
  static final int TYPE_INT = 2;
  static final int TYPE_FLOAT = 3;
  static final int TYPE_STRING = 4;
  static final int TYPE_BINARY = 5;
  static final int TYPE_ARRAY = 6;
  static final int TYPE_EXPR = 7;
  static final int TYPE_VREF = 8;
  // ...
  int type(void);
  string classname(void);
  bool to_bool(void);
  int to_int(void);
  double to_double(void);
  string to_string(void);

  string str(Env env = null);

  SObject binary_type_id(void);
  byte[] binary_data(void);
  int array_count(void);
  SObject[] array_values(void);
  SObject[] array_keys(void);
  string vref_ref(void);
  int expr_type(void);
  SObject[] expr_op(void);

  SObject eval(Env env = null);
  SObject resolve(bool recursive = true, Env env = null);
  int compare(SObject operand, Env env = null);
  // ...
};
```

**int** type(void)
Return the object type. This is one of the constants `TYPE_NIL` (**NIL** object, type `nil`), `TYPE_BOOL` (type `bool`), `TYPE_INT` (type `int`), `TYPE_FLOAT` (type `float`), `TYPE_STRING` (type `string`), `TYPE_BINARY` (type `binary`), `TYPE_ARRAY` (type `array`), `TYPE_EXPR` (type `expr`), `TYPE_VREF` (type `vref`).

**string** classname(void)
Return the class name. If the object is not associated with a type, `null` is returned.

The following set of methods converts simple object to native values of the programming environment. All method names are of the form `to_`*typename*, where *typename* is the name of a native type. As a consequence, the method names of these conversion functions depend on the programming environment. It is possible to have multiple conversion methods for the same simple object type, e.g. in C there might be conversion methods named `to_int`, `to_long`, and `to_long_long`.

The methods will perform implicit type conversion using the following rules:

1. Objects of type `nil` are converted to type `bool`, value **FALSE**, the integer or floating point value 0 (`int` or `float`), or to an empty string (type `string`).
2. Objects of type `bool` are converted to type `int`, `float`, or to type `string`. **FALSE** maps to 0 (`int` or `float`) or "`false`" (`string`), **TRUE** maps to 1 (`int` or `float`) or "`true`" (`string`).
3. Objects of type `int` are converted to type `float` or type `string`. If the object is converted to type `float`, the numerical value is retained as accurately as possible. If the object is converted to `string`, the shortest decimal representation using ASCII digits is generated.
4. Objects of type `array` are converted to type `string` by converting the values of all array elements to type `string` and concatenating these string objects. No separators are inserted between the array elements. The element keys are ignored.
5. All objects not convertable to type `string` using one of the rules above are converted to type `string` by putting the type name in angle brackets (i.e. < and >). E.g. an object of type `vref` is converted to the string "`<vref>`".

If a type conversion is not possible, a fatal runtime error is generated.

**bool** to_bool(**bool** &*error*)
: Return the boolean value of the object. If the object is not of type `bool`, it is converted to type `bool` using the ruleset above. If the object can not be converted to type `bool`, **FALSE** is returned and a conversion error is indicated.

  **bool** &*error*
  : In case of a conversion error, *error* is set to **TRUE**.

**int** to_int(**bool** &*error*)
: Return the integer value of the object. If the object is not of type `int`, it is converted to type `int` using the ruleset above. If the object can not be converted to type `int`, `0` is returned and a conversion error is indicated. If the resulting intger value can not be converted to the native **int** type (overflow), an error is indicated and the minimum/maximum representatable value is returned (e.g. in C these will be `INT_MIN` and `INT_MAX`).

  **bool** &*error*
  : In case of a conversion error, *error* is set to **TRUE**.

**double** to_double(**bool** &*error*)
: Return the floating point value of the object. If the object is not of type `float`, it is converted to type `float` using the ruleset above. If the object can not be converted to type `float`, `0.0` is returned and a conversion error is indicated. If the resulting floating point value is too large for the native **double** type, infinity (or negative infinity, if appropriate and available) is returned *without* a conversion error indication.

  **bool** &*error*
  : In case of a conversion error, *error* is set to **TRUE**.

**string** to_string()
Return the string value of the object. If the object is not of type `string`, it is converted to type `string` using the ruleset above. Variable references embedded in the string value are *not* resolved.

**Note:** Rule 5 in the ruleset above makes sure that *all* objects can be converted to `string`.

**string** str(**Env** *env* `= null`)
Return the string representation of the simple object. The string representation is created using the `str()` method from the environment object. If the `str()` method returns `null`, the `to_string()` method is used to create the string representation.

**Env** *env* (default value `null`)
The environment object. The special value `null` represents the default environment.

The following methods can be used to examine simple objects that can't be mapped to a simple base type. No type conversion is done. If an object is not of the expected type, a fatal runtime error is generated.

**SObject** binary_type_id(void)
Return the type ID object of a binary object (i.e. the *id* field of the *data* field, see section 2.1). It is a fatal runtime error if the called object is not of type `binary`.

**byte[]** binary_data(void)
Return the data held by the *body* of a binary object. It is a fatal runtime error if the called object is not of type `binary`.

**int** array_count(void)
Return the number of elements stored in an array object. It is a fatal runtime error if the called object is not of type `array`.

**SObject[]** array_values(void)
Return an array holding the values stored in an array object. It is a fatal runtime error if the called object is not of type `array`.

**SObject[]** array_keys(void)
Return an array holding the keys stored in an array object. It is a fatal runtime error if the called object is not of type `array`.

**string** vref_ref(void)
Return the reference string of a variable reference as an extended UNICODE string as defined in section 2.3.[24] It is a fatal runtime error if the called object is not of type `vref`.

**int** expr_type(void)
Return the operator type of an expression object. This is one of the constants listed in table 2 on page 49. It is a fatal runtime error if the called object is not of type `expr`.

**SObject[]** expr_op(void)
Return an array of expression operands. It is a fatal runtime error if the called object is not of type `expr`.

**SObject** eval(**Env** *env* `= null`)
Perform an object evaluation. The semantics of an evaluation operation depends on the object type:

`string`
If the string object contains variable references, these variable references are resolved.

[24] An extended UNICODE string can be resolved by creating a `string` object with `SObject::STRING_ucx()` and calling `SObject::resolve()`.

`vref`
: The variable reference is resolved. If the resulting object is of type `string` or `expression`, the `eval()` method is called again on the result.

`expr`
: If a class environment is associated with the first operand of the expression, the `eval()` method of that environment is called to perform the evaluation (see section D.2). If the expression object itself is associated with a class name, that class name is applied to the resulting object.[25]

all other types
: The called object is returned.

Note that the `eval()` method calles itself recursively only if an object of type `vref` resolves to an object of type `string` or `expr`. As a consequence, at most one recursion is performed.
Parameters:

**Env** *env* (default value `null`)
: The environment used for evaluation. The value `null` represents the default environment.

The method returns the result of the evaluation or `null` if the evaluation failed.

**SObject** resolve(**bool** *recursive* `= true`, **Env** *env* `= null`)
: Resolve variable references. The method may operate recursively on all contained objects. An object of type `string` is replaced by a string with all variable references resolved. An object of type `vref` is resolved. All other objects are kept unchanged.

**bool** *recursive* (default value `true`)
: Flag indicating if the method should operate recursively. If this flag is set, the method operates on all objects contained in the called object. If the flag is clear, the method operates only on the called object itself.

**Env** *env* (default value `null`)
: The environment used for variable resolution. If this is `null`, the environment of the called object is used.

The method returns the resolved object. If no resolution was performed, the returned object is the called object itself. The caller may compare the called object to the object returned to find out if a variable resolution was done.

**Note:** The resolution of variable references may yield more variable references. If this is the case, the variable references returned by a variable lookup are *not* resolved.

**int** compare(**SObject** *operand*, **Env** *env* `= null`)
: Perform an ordered comparison of the called object with the specified object *operand*. Variable references in strings are resolved before the objects are compared. Objects of type `vref` are *not* resolved.
Parameters:

**SObject** *operand*
: The operand compared with the called object.

**Env** *env* (default value `null`)
: The environment object. The special value `null` represents the default environment.

**Note:** This comparison method implies a total order on simple object values. This order depends on the variable resolution method `vref()` or `vref_str()` of the environment object.

The method returns:

[25] Applying a class name to an expression object can be interpreted as a cast operation.

`-1` if the called object is smaller than the operand,

`0` if the called object is equal to the operand (after resolving variable references in strings),

`1` if the called object is greater than the operand.

### D.3.5 Serialization and Deserialization

```
class SObject
{
  // ...
  static final int CTX_GENERAL = 1;
  static final int CTX_SELECTION = 2;
  static final int CTX_ARRAY = 3;
  static final int CTX_EXPRESSION = 4;
  static final int CTX_STRING = 5;

  static final int SER_BINARY = 1;
  static final int SER_TEXT = 2;
  static final int SER_TEXT_COMPACT = 3;
  static final int SER_TEXT_PRETTY = 4;
  // ...
  static SObject unpack(byte[] ser,
int context = SObject.CTX_GENERAL,
Env env = null, bool envall = false);
  static SObject unpack(string ser,
int context = SObject.CTX_GENERAL,
Env env = null, bool envall = false);
  byte[] pack(int mode = SObject.SER_TEXT);
  // ...
};
```

Serialization and deserialization is performed by the object methods `pack()` (serialization) and `unpack()` (deserialization). The serialization contexts are represented by the `CTX_` constants:

`CTX_GENERAL`
: `general` context.

`CTX_SELECTION`
: `selection` context.

`CTX_ARRAY`
: `array` context.

`CTX_EXPRESSION`
: `expression` context.

`CTX_STRING`
: `string` context.

The deserialization method (`unpack()`) accepts both serialization forms, binary and text serialization. The binary serialization is recogized if the specified context is not `CTX_STRING` and the high bit of the first byte of the serialization is set.

When the deserialization is done, the specified environment object (parameter *env*) is bound to all object instances created that are not of type `nil`, `bool`, `int`, `float`. The flag *envall* forces the specified environment to be attached to *all* objects.

The serialization method `pack()` can be used to create the following serialization variants:

`SER_BINARY`
: A binary serialization is created.

`SER_TEXT`
: A standard text serialization is created. The serialization will contain whitespace characters for better readability.

`SER_TEXT_COMPACT`
: A compact text serialization is created. The serialization will be as dense as possible.

`SER_TEXT_PRETTY`
: A pretty printed text serialization is created. The serialization will contain a consiterable amount of whitespace.

Note that the text serialization variants `SER_TEXT`, `SER_TEXT_COMPACT`, and `SER_TEXT_PRETTY` may produce the same output for some implementations.

static **SObject** unpack( **byte[]** *ser*, **int** *context* `= SObject.CTX_GENERAL`, **Env** *env* `= null`)
: Deserialize a simple object. The serialized representation may be text or binary (resolved automatically). After deserialization, the `unpack()` method from the appropriate class environment is called.

  **byte[]** *ser*
  : The serialization of the simple object.

  **int** *context* (default value `SObject.CTX_GENERAL`)
  : The serialization context (see section 3.1.1 on page 10). If *ser* is a binary serialization, the *context* parameter is ignored.

  **Env** *env* (default value `null`)
  : The environment object. The special value `null` represents the default environment.

  On success, the deserialized object instance is returned. On error, `null` is returned and the error indicatior of the environment object *env* is set.

static **SObject** unpack( **string** *ser*, **int** *context* `= SObject.CTX_GENERAL`, **Env** *env* `= null`)
: Deserialize a simple object. This is a variant of the `unpack()` method above, acceping a string holding the serialization. The string is converted to UTF-8 before deserialization.

  **int** *context* (default value `SObject.CTX_GENERAL`)
  : The serialization context (see section 3.1.1 on page 10). If *ser* is a binary serialization, the *context* parameter is ignored.

  **string** *ser*
  : The serialization of the simple object, represented as a native `string`. This string is converted to UTF-8 before deserialization.

  **Env** *env* (default value `null`)
  : The environment object. The special value `null` represents the default environment.

  On success, the deserialized object instance is returned. On error, `null` is returned and the error indicatior of the environment object *env* is set.

**byte[]** pack(**int** *mode* `= SObject.SER_TEXT`, **Env** *env* `= null`)
: Create a serialization from a simple object instance. Before serialization, the `pack()` method of the class environment is called.

  **int** *mode* (default value `SObject.SER_TEXT`)
  : This parameter specifies the serialization variant. This is one of `SER_BINARY`, `SER_TEXT`, `SER_TEXT_COMPACT`, `SER_TEXT_PRETTY`. Specify `SER_BINARY` for a binary serialization and `SER_TEXT` for a default text serialization.

**Env** *env* (default value `null`)
The environment object. The value `null` represents the default environment.

The method returns a native `byte`-array holding the serialization of the called object.

### D.3.6 The Default Environment

The default environment can be obtained and defined using the following static methods from the `Env` class:

```
class Env
{
  // ...
  static Env get_default(void);
  static void set_default(Env env);
  // ...
};
```

static **Env** get_default(void)
Return the current default environment. The method returns the environment itself, not a clone.

static **void** set_default(**Env** *env*)
Set the default environment.

**Env** *env*
The new default environment object.

## D.4 The Standard Environment

A programming library may support the standard expression semantics described in appendix B (Expression Semantics) and/or the standard object classes described in appendix C (Standard Object Classes). If these standard expression semantics are supported, they should be available through a factory method of the environment class.

```
class Env
{
  // ...
  static Env STANDARD(void);
  // ...
};
```

static **Env** STANDARD(void)
Return an instance of the standard environment. Note that a new instance is returned for every call to the method, so the caller may modify the environment.

For object-oriented programming environments, the standard environment should be an instance of a class derived from the class `Env` called `StdEnv`:

```
class StdEnv extends Env
{
  // ...
};
```

## D.5 Message Identifiers

An error condition encountered by a simple objects implementation is communicated to the application using message identifiers. A message identifiers is a sequence of ASCII alphanumerics and underscores. Every message identifier is associated with a set of named formal parameters.

An application willing to handle error message identifiers typically declares a class derived from the standard environment class `StdEnv` implementing the `Env::message()` method.[26]

*XXX a list of message identifiers and parameters goes here. should use separate subsubsections so the message identifiers appear in the TOC.*

[26] An application written in C or another non-object-oriented language would could a standard environment instance directly.