Version: 7/11/2002 3:40 PM

FARGOS/VISTA
Object Implementation Language 2
Reference

FARGOS/VISTA Object Implementation Language 2 Reference

FARGOS Development, LLC
757 Delano Road
Yorktown Heights, NY 10598
http://www.fargos.net
mailto:support@fargos.net

Abbreviations

FARGOS Development, LLC is a Limited Liability Company registered with the State of New York. It is required to identify itself as such in its name, hence the ", LLC" suffix. For purposes of readability in this document, the ", LLC" suffix is sometimes dropped. The phrase "FARGOS Development" always denotes "FARGOS Development, LLC" and is not intended to suggest any alternate form of organization.

Contents

1. Introduction

Object Implementation Language 2 (hereafter referred to as OIL2) is the implementation language of choice for FARGOS/VISTA-based applications. OIL2 is a high-level object-oriented programming language designed for the writing of distributed applications.

Programs that are written in OIL2 can be compiled to an architecture neutral format object-code (OIL2 ANF) that can be interpreted by facilities contained with the FARGOS/VISTA Object Management. This is a convenient form for development because compile/load/execute operations can be performed in under a second. It also enables developers to distribute a single object file that can be executed on any platform supported by FARGOS/VISTA. The architecture-neutral object code is always dynamically loaded into a FARGOS/VISTA Object Management Environment.

OIL2 can also be compiled to an architecture-neutral C++ source file, which is then compiled into the native object code for a particular CPU architecture/operating system combination (e.g., Sun Solaris on SPARCs, Microsoft Windows NT on Intel Pentiums) using a C++ compiler (such as g++ or Microsoft's Visual C++). The primary benefit of native object code is increased performance. On all platforms, native object code can be statically linked into the respective FARGOS/VISTA executable. If supported by the underlying host operating system, native object code can also be dynamically loaded into a FARGOS/VISTA Object Management Environment in the same fashion as the architecture neutral object code.

OIL2 is an easy-to-learn language. Historical results with its predecessor (OIL) demonstrated that programmers with little prior experience with C were able to pick it up and start writing surprisingly sophisticated (e.g., fault-tolerant, distributed) applications within an afternoon. Novice programmers may find the tutorial document *An Introduction to Programming using OIL2* of interest. Many illustrative classes are found in *FARGOS/VISTA Examples*.

OIL2 is a language intended to enhance programmer productivity. Towards that end, it is concise without being obscure and shares a syntax and control structures that will be familiar to C programmers. While suitable for building conventional applications, its inherent power is most evident when building distributed applications. The design of OIL2 also pays attention to the elimination of much of minutiae of common programming tasks, which are all too easy for a programmer to get wrong. For example, advanced native data structures, such as sparse arrays, associative arrays and sets, eliminate the need for many table maintenance routines. All storage is dynamically allocated as needed and freed when no longer referenced without requiring explicit direction by the programmer. This eliminates common problems, such as buffer overflow and memory leaks, that plague many programs and contribute unreliable and insecure operation.

All OIL2 data is also tagged, enabling a program to inquire as to the type of a variable at runtime and take appropriate action. Coupled with other facilities in the FARGOS/VISTA Object Management Environment, this provides OIL2 programs with a self-describing environment.

2. The OIL2 Object Model

Strictly speaking, the name Object Implementation Language 2 is a bit of a misnomer: programmers actually write class definitions and their associated methods whereas an object is an instance of such a class. A more proper name would be Object-oriented Implementation Language 2. Luckily, that too would be called OIL2.

Object-oriented programming languages have been available since the late 1960's, but widespread recognition of their benefits did not take hold until the mid 1980's. As a rule, object-oriented models provide a separation between the interface to data structures and the implementation of the algorithms that manipulate that data. Individual object-oriented programming languages enforce this separation to various degrees by the constructs provided in the language. Some languages are very lax and provide the opportunity for object-oriented programming but do not require its use (C++ is a good example since it is, in practice, a superset of ANSI C). At the other extreme, some object-oriented languages enforce a pure, pristine object-oriented model.

OIL2 strives to implement an object model that is firmly rooted in theory, with the result being simple, consistent and yet powerful. Many of the rules that define the foundation of the OIL2 object model will be familiar to programmers who have previously been exposed to other object-oriented languages.

- A class defines both the variables that represent the state of a particular object (these are called instance variables) and the operations that can be performed against objects of a particular class (these operations are called methods).
- A class is uniquely named by three elements: a name space, the class name and a version Id.
- A class can inherit from one or more classes. The classes from which it inherits are called its *base* classes. From the perspective of its individual base classes, it is considered a *derived* class. The terminology of *super*- and *sub*-class is also commonly used; however, this document will use the terms *base* and *derived* as an aid to clarity since the similarity of the prefixes super and sub can create confusion when read quickly.
- A class must inherit from the base class **Object**. This can either be explicitly stated or implied as a property of inheriting from another base class that in turn eventually inherits from the class **Object**. Note that this requirement means that the implementation of the class **Object** cannot be expressed in OIL2 (nor can the class **Thread**).
- Every class must have both a **create** and a **delete** method. While most classes have more methods, it is entirely possible to have a useful class that only implements these two methods.
- An object is said to be an instance of a class. It is a distinct collection of variables as defined by the class definition.

The rules above pertain to the static nature of class definitions. The OIL2 object model also includes operational aspects, some of which are unconventional:

- Every object is identified by a globally unique identifier. This unique identifier is referred to as an object Id. Globally unique means across all machines, not just the address space into which the object is born.
- Two objects can only interact through the sending of a message. OIL2 does not permit the use of pointers and thus the direct manipulation of another object's instance variables. In OIL2, a message is sent using the **send** statement.
- In general, when a message is received for an object, the indicated method is executed. This process is called a method invocation. The indicated method may have a null body, which means that no code is to be executed. The **delete** method of many classes has this characteristic.

- If an object's method body is not null, then its execution is performed by a separate thread. This means that the runtime environment of OIL2 objects is one in which parallelism is supported at a fine level of granularity, namely that of a method invocation. If compared to conventional environments, this would correspond to a separate thread of execution being spawned for each function call.
- By default, only one method can be active on an object at a time. This restriction enforces safe behavior by default and prevents race conditions, a common issue in multi-threaded environments. Except in very complex cases, programmers need take no action to disable the default behavior, but this capability is available.
- If more than one method is active against an object, the other method cannot proceed until the currently active method is suspended. Again, the default is to enforce safe behavior and prevent race conditions, but this can be overridden.

For any language, actual programs execute within an environment that provides facilities beyond those declared by the language. For example, OIL2 programs run within a FARGOS/VISTA Object Management Environment and can take advantage of features such as reflection and persistence.

3. Design Philosophy

The design of OIL2 was influenced by experience gained from its predecessor; it retains the goals of that language. One of those goals was to permit independent programmers to cooperate in the building of complex distributed systems without requiring close synchronization of their efforts. Another is to provide a significant boost in programmer productivity (6 to 10-fold improvements were seen with OIL). One means to that end is to assist in the construction of robust systems and the elimination of details that often, while trivial, are overlooked or incorrectly implemented[1]. Like OIL, the most important philosophical design decision enforced in OIL2 is that methods are the only external interfaces to an object.

This has far-reaching implications. One is that a derived class does not have direct access to the instance variables of a base class[2]. An immediate benefit from this is the avoidance of the fragile base class syndrome, in which derived classes that make use of a base class must be recompiled after any non-trivial modification is made to

[1] For example, a common cause of program crashes or opportunity for system penetration by hackers is the use of inherent predefined limits on the size of an array or a string buffer. Often in languages like C/C++, the size of an array is fixed at compile time to be of a size that the programmer "knows" is large enough. Then when the program encounters a situation not anticipated by the programmer, it crashes or the overflowed buffer can be used to insert new instructions on the thread’s stack. OIL2 does not have limits on arrays, strings, etc. declared, so these sorts of issues cannot occur. Another common case is the issue of dynamically allocating and freeing memory. Within an OIL2 method body, memory is implicitly allocated as needed and the freeing of storage is performed automatically. Consequently, OIL2 has no equivalent to C++'s **new** and **delete** operators.

[2] In C++, this would be the equivalent of saying that all member variables are declared to be **private**. After many years of use, direct access to specific variables of a base class was added to OIL. This clear violation of a governing design principle was made for performance reasons. OIL2 currently remains "pure" in this aspect because the sophistication of the underlying FARGOS/VISTA runtime avoids much of the overhead that was circumvented by this extension to OIL.

the source of the base class. Another is that a cooperating group of programmers does not need to exchange header files, much less keep them synchronized.

4. Language Elements

Before the OIL2 grammar is described, it is useful to have an understanding of some of the runtime environment issues.

Native Data Types

OIL2 has several native data types that are used to hold data in global, local and instance variables. The type keywords and a description of each native type appear in Table 1.

Table 1

Keyword	Type	Comments
nil	A null value	A special value to indicate no value
int **int32**	32-bit integer	A normal integer; int and int32 are synonymous.
int64	64-bit integer	Sometimes, 32 bits are not enough.
float	32-bit floating point	Single precision floating point.
double	64-bit floating point	Double precision floating point.
fixed	Fixed point decimal	Arbitrary precision arithmetic, useful for currency.
string	Octet string	Can contain embedded nulls; also keeps track of character set (e.g., ASCII, EBCDIC, Unicode, binary); the reference counted implementation in FARGOS/VISTA makes it very inexpensive to pass strings around since no data copying occurs.
oid	Object Id	Reference to an object.
array	Sparse array	Can be subscripted by non-contiguous int32 values; reference counted implementation in FARGOS/VISTA.
assoc	Associative array	Subscripted by strings (binary data OK); reference counted implementation in FARGOS/VISTA.
set	Set of elements	Really a list that preserves order; reference counted implementation in FARGOS/VISTA.
nlm	Native Language Message	Internationalized and machine-readable messages; referenced-counted implementation in FARGOS/VISTA.
any	Any type	Any of the above types can be used.

A collection of related variables are normally placed together in a class; in simple situations, they may be bound together in a structure:

```
struct [structTypeName] { variableDeclarationList } [identifierList];
```

The individual elements of a structure are selected using a "." operator:

```
struct example {
   int   var1;
   float var2;
} s;
int      i;
float    j;

i = s.var1;
J = s.var2;
```

Class Namespaces and Versions

All OIL2 classes are uniquely identified by three elements: a namespace, a class name and a version Id. A namespace is a somewhat arbitrary text name and is used to prevent name collisions between classes. There are a small number of predefined name spaces:

Namespace	Description
Standard	Reserved for FARGOS/VISTA classes in the Object Management Environment core.
Local	Suggested default name space for OIL2 classes.

An organization can keep classes that it develops within their own namespace and thus prevent unintended collisions with classes developed by other organizations. Exploitation of this feature can also permit a site to deploy classes that effectively replace the implementation of standard classes. It is recommended that when developing production-quality code, programmers should explicitly specify a namespace instead of using the default.

OIL2 also provides support for environments that utilize persistent objects by recognizing that class implementations may need to be altered over time while old data is retained. This capability is supported in part using version Ids. Each class implementation has a unique version Id associated with it. The FARGOS/VISTA Object Management Environment permits more than one implementation of a given name space/class name to simultaneously exist. If a version Id is not explicitly specified in a class definition, the OIL2 compiler generates one automatically. If a class may be used by objects that will be persistent, it is a good idea to explicitly specify the class version Id rather than let the OIL2 compiler create one automatically.

Global Variables

OIL2 supports global variables that are local to a FARGOS/VISTA Object Management Environment process. These can be used to achieve effects similar to variables declared as static in C++ classes. Normally, global variables are visible only to the methods declared in a single source file; however, a collection of variables can be assigned a name and subsequently be visible to methods in more than one file. This is similar to named COMMON sections in Fortran. It should be noted, however, that explicitly named sections incur more overhead than unnamed sections and should be used only when necessary.

External Functions

OIL2 permits functions written in C++ to be called from OIL2 method bodies. Such functions need to be declared before they are referenced. OIL2 supports two calling conventions: the conventional form handles a fixed number of arguments and the other permits the passing of a variable number of arguments. This is discussed in more detail below (see External Function Declarations).

Include Files

The OIL2 compiler can include files using the **%include** directive. A **%include** directive must appear at the beginning of a source code line. It intentionally has the same form as the C preprocessor **#include** command:

```
   %include <fileSuffix>   // searches for file using a predefined list of
directories
   %include "fileName"     // uses the specified file name as-is
```

The standard directories searched by a file included using the "**<***fileSuffix***>**" form are:

- The list of directories specified by the OIL2_INCLUDE_PATH environment variable.
- The current working directory
- The directory specified by $VISTA_ROOT/oil2Include, where $VISTA_ROOT is root of the distribution tree which is defined by the value of the VISTA_ROOT environment variable.

5. The OIL2 Grammar

Comments

As in C++, comments can be placed in OIL2 source code by two means. The first is by prefixing the comment with the character sequence "**//**". The remainder of the line is treated as a comment. The second approach is a block comment and is introduced with the character sequence "**/***". The end of the block comment is indicated by the character sequence "***/**".

Constants

OIL2 supports several kinds of constants.

Table 2

Type	Specification
base 10 integer	A sequence of digits
base 16 integer	0x followed by a sequence of hexadecimal digits
floating point	An optional sequence of digits, then a period followed by a sequence of digits
fixed point	A "$" followed by an optional sequence of digits, then a period followed by a sequence of digits

Type	Specification
string	a double quote mark, any number of characters, another double quote mark. A double quote can be escaped by a backslash. Strings may contain escaped characters.
integer	a single quote, a character, another single quote. The character may be escaped.

Escaped Characters

The OIL2 compiler recognizes several escaped characters within the body of a string or character constant. These are detailed in Table 3.

Table 3

Escape Sequence	Meaning
\n	Line Feed
\r	Carriage Return
\t	Tab
\f	Page Feed
\”	Double quote mark
\’	A single quote mark

Documentation

In 1992, the OIL grammar provided explicit support for inline documentation of classes. The original OIL compiler output documentation in a variety of formats: Unix man pages, IBM’s BookMaster SGML, Adobe’s FrameMaker MIF, Microsoft’s Rich Text Format, and Hyper Text Markup Language (HTML). At the current time, support for HTML has been widely implemented in most publishing applications, so it has been adopted as the internal markup language for OIL2 documentation. Documentation blocks in OIL2 source code are introduced by the three-character sequence “**/*!**”and terminated by a corresponding reversed character sequence of “**!*/**”. One documentation block can appear per class definition or method.

Identifier Names

OIL2 identifier names are constructed as a sequence of one or more alphanumeric characters. The initial character must be either a letter (uppercase or lowercase) or one of the three special characters "_", "$" or "@". Subsequent characters can also make use of the digits 0-9. OIL2 does not prescribe any limit on the length of an identifier, but individual compiler implementations will often impose a practical limit, such as 16383 characters (which is over 200 screen lines of solid text and the difficulties in locating a typographical error in a variable name containing more than 16,000 characters should be readily apparent).

File Structure

An OIL2 source file is composed of zero or more blocks. Blocks can appear in any order, however, a block must appear before its contents are referenced. The various blocks are, in suggested order of appearance:

Block Type	Description
External	Declares external functions (e.g., that are written in C++)
Global	Defines a collection of global variables and constants
Implicit	Defines a set of variables and constants that should be implicitly defined for every method body
Function	Defines a function written in OIL2
Class	Defines a class
Method	Defines a method of a class

Each of the blocks is described below.

External Function Declarations

OIL2-callable functions that are external to the source file are declared in an **external** block. Most functions take a fixed number of arguments and they are defined as illustrated by the prototype below:

external *typeName* functionName **(**[*typeName* [parameterName] [**,** *typeName* [parameterName]] …**)**;

The specification of a parameter name is optional; however, an OIL2 compiler can use this information to provide error messages that are easier to understand. Unlike many languages, OIL2 makes it very easy to write methods that take a variable number of arguments. It also supports functions that do the same. Functions that take a variable number of arguments are defined in a fashion similar to that of their fixed position brethren, with the exception that the parameter list is an ellipsis (a sequence of three periods):

external *typeName* functionName **(…)**;

The **external** keyword may also be abbreviated as **extern**, which is identical to the C/C++ keyword. Some examples appear below:

```
external int typeOf(any)
extern int registerService(string serviceName, oid object, int exportable);
external string makeAsString(...);
```

Global Variable Definitions

Variables that are to be global to the methods in the source file are defined in a **global** block. Normally, global blocks are unnamed and the global variables defined are visible only within the source file. The result is very similar in effect to variables declared as **static** in the C programming language. OIL2 also make provision for a block to be given a name and thus be visible to code from multiple source files. Used in this fashion, the result is much like named COMMON blocks in Fortran. The syntax is illustrated below:

global [globalBlockName] **{** variableAndConstantDeclarations **}**;

An example **global** declaration appears below:

```
global {
   const string srcID = "$Id$";
   int   totalCount;
};
```

Declaration of Implicit Variables

For convenience, variables can be defined as **implicit**. This enables a programmer to avoid the need to explicitly define commonly used variables in the body of each method. This capability is only useful if a consistent coding convention is followed. The syntax is similar to that of a **global** block, with the exception that no name for the block is permitted.

implicit { constantAndVariableDeclarations **}**;

An example follows:

```
implicit {
   int   rc;
};
```

OIL2-implemented Functions

Functions that are implemented in OIL2 instead of C++ are described in a function block. NOTE: the current release level of the OIL2 compiler does not support this feature. Use a method body instead and **call** it.

Named Constant Definitions

A constant can be assigned a name by declaring it within a **global** block, **implicit** block or method body. Most constants are declared using a **const** statement:

const identifierName **=** constantExpression**;**

A special form of constant is an enumerated list, which is defined by an **enum** statement:

enum [enumName] **{** identifier1 [**=** integerExpression] [**,** identfier2 [**=** integerExpr] …] **};**

By default, the first identifier in the list is assigned a value of 1; each subsequent identifier is assigned a value one greater than its predecessor. Any given element can be assigned a specific value using the optional "**=** *constantExpression*" clause. There is a second form of the **enum** statement that is convenient for declaring bit masks:

enum set enumName **{** identifier1 [**,** identifier2 …] **};**

Each identifier will automatically be assigned a value that corresponds to a distinct bit position.

Class Definitions

A class is defined in a class block. The information provided by a class definition includes the name of the class, the classes from which it inherits and a description of the instance variables for each object of the class. It is also possible to define class-specific constants that are only visible within the scope of the methods of the class. A class definition is illustrated below:

[**unique**] **class** [nameSpace **.**] className [**(**versionID**)**] **{**
constantAndVariableDeclarations

} inherits from [nameSpace1 **.**] class1[**(**versionID**)**] [**,** [nameSpace2 **.**] class2[**(**versionID**)**]] … ;

While not mandatory, it is recommended that programmers provide the desired name space of the new class. If not specified, the OIL2 compiler will generate a default name space, which is typically *Local*. The version of the new class can also be specified. This is very useful for classes that might be used with persistent objects (e.g., see class **PersistentObject**). If a version Id is not specified, one is automatically generated.

The **inherits from** clause specifies the classes from which a new class inherits. At least one class must be specified; usually this is **Object**. An inherited class can be qualified with a name space prefix and a version Id suffix. If the name space is not specified, the name space will be determined at runtime. If a version Id is not specified, the most recent version of a class will be used.

The class definition can be prefix with the keyword **unique**. This declares that a unique set of instance variables should be maintained for each derived class that inherits this class. This only becomes an issue with multiple inheritance and can be illustrated by an example. Suppose class B inherits from A; class C also inherits from A; and new class D inherits from both B and C. The question is: should D have one or two copies of the instance variables associated with class A?

The default answer is one and the resulting inheritance graph would visually appear as a diamond, as illustrated in Figure 1.[3]

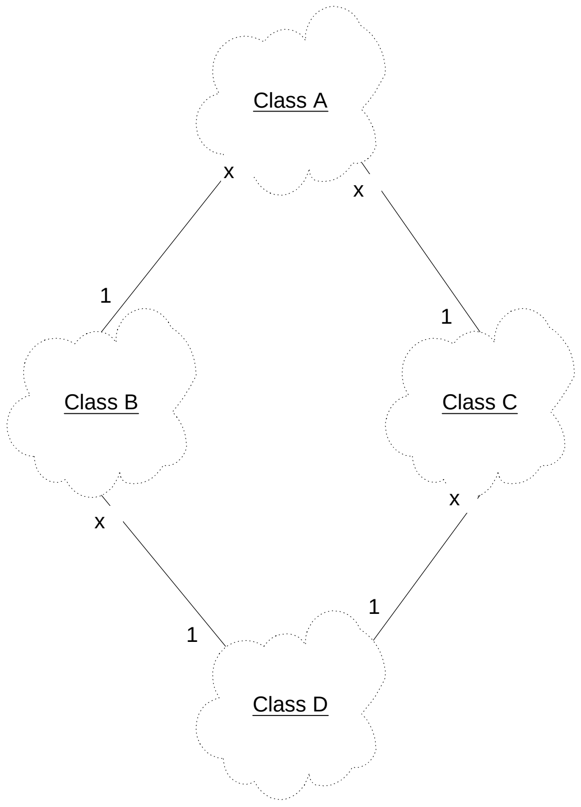

Figure 1

[3] For C++ programmers, this is equivalent to having all base classes declared as virtual.

Most applications will work fine with this compression. If, however, one attempts to write a class that offers services to a derived class and assumes that it will be only used by one such class, it will fail when two or more classes within the same inheritance tree attempt to make use of the service. A trivial illustration would be a class that implemented a link in a chain of items. If written so that it only participated in one chain, having two derived classes try to use it at the same time would be a disaster. This demonstrates the need for one to sometimes to have a unique copy of a base class for the derived classes that use it. This is enabled by the use of the **unique** keyword and the resulting inheritance graph looks like a wedge, as illustrated in Figure 2.[4]

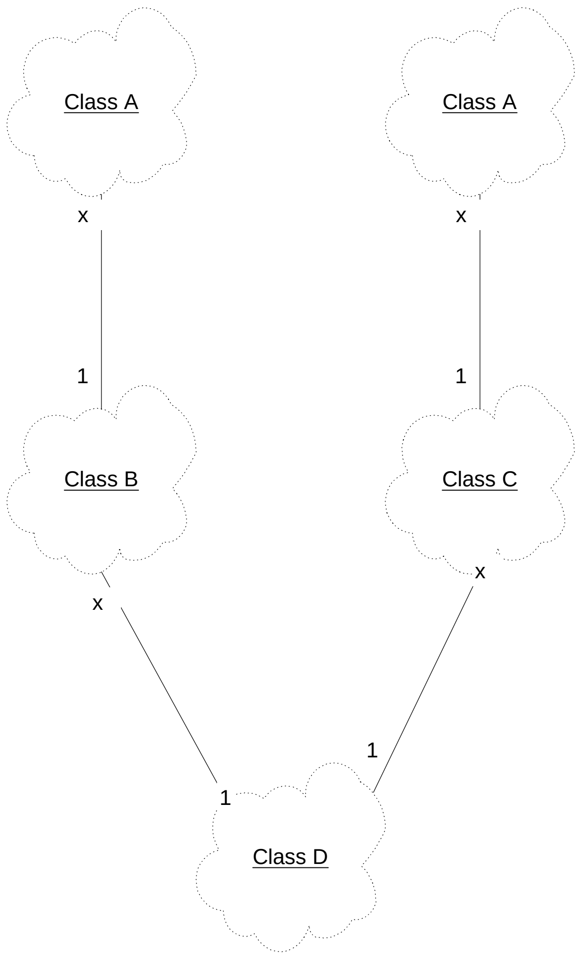

Figure 2

It is worth nothing that the specification of **unique** is provided by the class implementer, not the user of a class. This is one aspect of isolation of the implementation details of a class from the users of said class.

If the name space for the class is not defined, it defaults to a value defined by the OIL2 compiler. Typically, this is "*Local*"; however, it is quite reasonable for an OIL2

[4] This is a case where OIL2 is 180-degrees opposite from C++. In C++, you have to explicitly denote that a base class should have only one instance in the inheritance hierarchy; by default, you would get unique copies for each derived class that made use of it.

compiler to permit the default value to be overridden by a command line argument or via a site-specific profile.

A complete, albeit imaginary, example appears below:

```
class Local . Demo (1) {
   const string serviceName = "/SomeService";
   const float PI = 3.14;
   int   count;
   oid   clientObj;
} inherits from AnotherClass(1), Object;
```

Method Implementation

Each method implementation appears in a method block. A method of a given class must appear in the file at some point after the declaration of the class. Method blocks are one of the most complicated constructs because their primary purpose is to contain executable statements that detail the logic of an application. The presence of such executable statements is a significant way in which OIL2 is distinguished from the plethora of Interface Definition Languages (IDLs)—the method itself is defined, not just a prototype.

A method block appears similar to the following:

[nameSpace **.**] className [**(**version**)**] **:** methodName **(**argumentListPrototype**)**
[**unique**] **{** methodBody **}**

In practice, the *nameSpace* and class *version* Id parameters are left off and the compiler uses the class name alone to determine to which class the method belongs; the name space and version Id of the most recently defined class of the indicated name in the source file are used as implied values. Full qualification of the class name would be used in a situation where more than one class with the exact same name is implemented in the same source file. Such a scenario would be rare indeed: most developers, if forced to maintain two versions of the same class, would tend to work with two separate source files.

The argument list prototype is a comma-separated list of type name/parameter name pairs:

[[**optional**] *typeName* [parameterName] [**,** [**optional**] *typeName*
[parameterName]] …

Here, OIL2 differs from most languages in that all of the arguments to a method are always made available via two predefined variables. The integer variable *argc* contains a count of the arguments passed to the method. The arguments themselves are available as elements of the array variable *argv*, starting with subscript 0. Providing the type of a method argument instructs the compiler as to what is expected and adding a name for the argument allows it to be conveniently referenced as a variable by code within the body of the method. A type declaration can be prefixed with the keyword **optional**, which serves as a hint to both the compiler and users that the indicated parameter is not always passed. The name of a parameter is really an alias for the corresponding subscripted element of the *argv* array.

An example appears below:

```
Demo:method1(int arg1, string arg2, optional float arg3)
{
   int   j, k;
   // variable declarations here…
   // body of method here…
   j = arg1;
   k = argv[0];
   // j and k reference the same data
}
```

Method Overloading

Normally, an OIL2 class has only one implementation corresponding to a particular method name. This is in keeping with the prior practice of OIL. OIL2, however, also supports overloading of methods that is common in other object-oriented languages that do not have the ability to determine the type of an argument at runtime. If method overloading is used, the actual method to be invoked will depend not just on its name, but also on the type of the arguments passed at runtime. Method overloading is selected by appending the keyword **unique** after the method's argument list.

Method Aliases

Sometimes a method's implementation is identical or nearly identical to an existing method. Rather than implement a duplicate method body, a new method can be declared an **alias for** an existing method of the class. This not only reduces the programmer's effort required to develop and maintain an extra method, but it also reduces the size of the object file since only one copy of the function code is needed.

[nameSpace **.**] className [**(**version**)**] **:** methodName **(**argumentListPrototype**)**
[**unique**] **alias for** originalMethodName **;**

The usage is illustrated below:

```
Demo:method2(int arg1, string arg2) alias for method1;
```

Note: the code within a method body can examine the thread context variable *thisMethod* to determine the name by which the method was invoked.

Method Bodies

OIL2 method bodies are composed of a set of statements, which can be either declarative or executable. Declarative statements declare things, such as the type and name of a variable or the value of a constant. Executable statements specify an action to be taken, such as the computation of a result or a transfer of the flow of execution. Declarative statements can appear in the bodies of many of the blocks described that have been described above, but executable statements can normally only appear within the body of a method or function[5]. A semicolon always separates one statement from another. Braces are used to enclose statement blocks and introduce a new naming scope.

Executable Statements

Executable statements in OIL fall into several broad groups:

- the computation of results, more formally referred to as expressions

[5] The exception is a constant expression: the value of a **const** declaration can be a simple expression.

- looping constructs, such as the various forms of the **for**-statement, the **while**- and **do**-statements and support statements like **break** and **continue**.
- conditional execution, best illustrated by the **if-else** statement.
- inter-object communication performed by the various forms of the **send**-statement.

In OIL2, an expression alone can sometimes be a statement but a statement is never an expression. A common example of an expression-as-a-statement would be a function call where the returned result is ignored. The operation of assignment, which is treated as a statement in many languages, is treated in OIL2 as an expression. Like in C, this permits usage such as:

```
    a = b = c = 0;
```

Technically, this means that the common construct:

```
    j = 0;
```

can also be viewed as an expression whose result is ignored.

Thread Context

In the OIL2 object model, every method executes as a separate thread. Each thread is provided with several predefined variables, which are detailed in Table 4.

Table 4

Variable Name	Type	Description
argc	int	The number of arguments passed to method, zero indicates none were passed.
argv	array	Array of arguments to method, first parameter in subscript 0.
thisMethod	string	The name of the executing method.
threadContext	assoc	Associative array of environment variables
userInfo	assoc	Id of user
thisObject	oid	Object Id of the object upon which the method is active
fromObject	any	Object Id or service name of the object from which the message was sent; nil if no reply is desired
thisThread	oid	Object Id of the active thread
threadErrorCode	any	Thread-specific error information
$replyResult	any	Refers to the result from an RPC-style method invocations; not normally referenced directly.
ObjectCreator	oid	Object Id of the **ObjectCreator** object

The arguments to each thread are always made available as the predefined variables *argc* and *argv*.

The OIL2 compiler also predefines some constants:

Constant Name	type	Description

emptyAssoc	assoc	an empty associative array
emptyArray	array	an empty sparse array
emptySet	set	an empty set
nil	nil	a nil value

In addition, the various type names, which are keywords, can also be used wherever an integer constant is valid. These are most often used when comparing the result of the **typeOf()** function. For example:

```
    if (typeOf(x) == string) { // then it's a string
          display("It's a string\n");
    else if (typeOf(x) == float) {
          display("It's a floating point number\n");
    }
```

OIL2 Statement Reference

send

The **send** statement is used to send a message to another object. Normally, this results in a method being invoked on the target (this might not happen for a few reasons, including a null method body—nothing interesting to do—or the indicated method does not exist). There are two forms of the **send** statement, the basic primitive and an RPC-style invocation.

send methodName[**(**[argumentList]**)**] **to** destinationObject [**from** fromObj] [**in** timeoutSeconds]

There are two mandatory elements of the **send** statement. The first is the method name and the second is the destination object. Both of these are expressions that can be computed at runtime. This has far-reaching implications and yields unprecedented flexibility. While in practice the vast majority of method names are specified as constants, an application can generate the name of the method it wants to invoke by performing any arbitrary computations it needs.

While optional, most method invocations will include some data as arguments. The argument list is computed at runtime as a set expression using the union operator (|) between each comma-separated expression. This novel characteristic makes it trivial to manipulate and construct argument lists at runtime. This is quite different from many programming languages. While many programmers might never have written a function capable of taking a variable number of arguments and their sole experience with variable argument functions might be only the printf* family of functions, OIL2 programmers make use of variable argument lists frequently because of their ease of use.

Note: if an argument list expression is a set, it is expanded into its individual elements due to the use of the union operator. Thus, if a set needs to be passed as a positional argument, it should be first encapsulated in an empty set using the addition operator:

```
    set   argList, s1;
    s1 += 1;
    s1 += 2;
    argList = emptySet + s1; // encapsulate set
    send “doSomething”(argList) to destObj;
```

The destination object may be specified using an object Id or it can be a string that corresponds to the name of a registered service.

A method invocation can be made to appear as if it came from another object using the **from** clause. By convention, if one does not want a result returned from a method, the **from** clause should be used with a value of **nil** specified as the *fromObj*. This is not always necessary, as some methods are written to not return results, but it is good form. When using the simple form above, if the **from** clause is not used, the value of *fromObj* defaults to that of *thisObject*.

There is also a second form of the **send** statement that is used to make Remote Procedure Call-style method invocations. It is almost identical in appearance:

lval **= send** methodName[**(**argumentList**)**] **to** destinationObject [**from** fromObj] [**in** timeoutSeconds]

The significant difference is that there is an assignment into a variable specified. With this form, the method invocation takes places as normal, but the thread is put to sleep and waits for a result to be returned. After the result is obtained, the thread is woken up and the value is stored in the location indicated by *lval*. While the **from** clause is supported in this RPC-style form, its use will provide correct results only under very special conditions created by expert programmers. When an RPC-style form of the **send** statement is used, the value of *fromObj* defaults to that of *thisThread* instead of *thisObject*.

NOTE: while recognized and used by the OIL2 compiler, support for the timeout parameter is currently not implemented by the FARGOS/VISTA Object Management Environment.

An RPC-style **send** performs functionality that is logically equivalent to the following sequence of statements:

```
    sleepingThread = thisThread;
    send methodName(argumentList) to destinationObject from thisThread;
    send “suspendThread” to thisThread;
    sval = $replyResult;
```

The class **Thread** implements a highly optimized implementation of the *reply* method that performs functionality similar to:

```
Thread:reply(any result)
{
    $replyResult = result;
    send “resumeThread” to sleepingThread;
}
```

break

The **break** statement forces an immediate exit from the innermost **for**/**do**/**while** loop.

continue

The **continue** statement cause an immediate jump to the top of the innermost **for**/**do**/**while** loop. It is used to immediately start the next pass through a loop.

exit

The **exit** statement causes an immediate termination of the current thread. An implicit **exit** takes place whenever the flow of execution reaches the bottom of a method. No value is return from the method—if this is desired, a **return** statement must be used.

return

The **return** statement returns a value from a method and terminates the thread. It takes the following form:

return (expression **)**

Note: the parentheses are mandatory, unlike C/C++. If a method is to not return a value, the **exit** statement should be used instead.

The actual behavior of this statement differs based on whether or not the method was invoked by a **send** statement or called as a function by a **call** statement. If invoked via a **send**, then the **return()** statement essentially performs the following:

```
    if (fromObject != nil) send "reply"(expression) to fromObject;
    exit;
```

Note: if the value of *fromObject* is **nil**, then the **send** of the *reply* method is not performed.

If the method was executed as the result of a **call**, the behavior is more like that of a conventional return from a function: the result to be returned is placed at the appropriate place on the calling thread's stack and the flow of execution is returned to the caller. In this case, the active thread is not terminated because the flow of execution needs to continue in the context of the caller.

do-while

The **do-while** statement is a looping construct that makes one pass through the statements to be executed and then evaluates a condition that determines if another pass should be performed.

do statement **while (**expression**)**

Note: The parentheses around the conditional expression are mandatory. If the conditional expression evaluates to a non-zero value, the statement will be executed again. If a **break** statement is encountered, the loop will be exited. If a **continue** statement is encountered, the flow of control will pass to the bottom of the loop, where the conditional expression is reevaluated.

while

The **while** statement is a looping construct that first evaluates a condition that determines if a pass should be performed through a block of statements.

while (expression**)** statement

Note: the parentheses around the conditional expression are mandatory. If the conditional expression evaluates to a non-zero value, the statement will be executed again. If a **break** statement is encountered, the loop will be exited. If a **continue** statement is encountered, the flow of control will pass to the top of the loop, where the conditional expression is reevaluated.

if-else

Statements can be conditionally executed with the **if-else** statement.

if (conditionalExpression**)** statement1 [**else** statement2]

Note: the parentheses around the conditional expression are mandatory.

for-in-do

The **for-in-do** statement performs iteration over a set in OIL2.

for lval **in** setExpression **do** statement

The variable indicated by *lval* is set to the first element of the set and the statement block is executed. Subsequent passes are made with the variable having been set to the value of the second, third, etc. element in the set. When either all of the elements in the set have been processed or a **break** statement is encountered, the flow of execution proceeds to the next statement. A **continue** statement will stop execution of the current pass, retrieve the next variable and start the next pass.

for

This form of the **for** statement is a convenient way to express many looping situations and will be familiar to C/C++ programmers.

for(initialStatement**;**conditionalExpression**;**afterEachPassStatement**)** statement

The *initialStatement* is always executed once, then the *conditionalExpression* is evaluted. If the result is non-zero, then the *statement* block is executed. When this is complete, the *afterEachPassStatement* is executed and the flow of control passes back to the top of the loop where the *conditionalExpression* will be reevaluated.

If a **break** statement is encountered, the loop will be exited. If a **continue** statement is encountered, the flow of control will pass to the *afterEachPassStatemen*t and then back to the top of the loop.

Simple Expressions

Simple expressions are represented by constants and the result of operators, such as addition and subtraction. The full lists of operators are provided in Table 5 and Table 6.

Table 5

Operator	Meaning	Comments
+	Addition	String concatenation, set insertion
-	Subtraction	Set removal
*	Multiplication	
/	Division	Set removal
%	Remainder module N	
mod	Remainder modulo N	Same as “%”
&	Logical AND	Remove elements not in set
|	Logical OR	Set union
^	Exclusive OR	
&&	Conditional AND	
and	Conditional AND	Same as “&&”
||	Conditional OR	
or	Conditional OR	Same as “||”
<	Less than	
>	Greater Than	
==	Equal to	
!=	Not equal	
<=	Less than or equal to	
>=	Greater than or equal to	
!	Unary negation	
not	Unary negation	Same as “!”

Assignment expressions

Beyond the simple assignment operator, several shorthand operators update a variable in place.

Table 6

Operator	Meaning
=	Assignment
+=	Add and assign
–=	Subtract and assign
*=	Multiply and assign

Operator	Meaning
/=	Divide and assign
%=	Modulo remainder and assign
&=	Logical AND and assign
|=	Logical OR and assign
^=	Logical XOR and assign

Function call expressions

OIL2 supports two calling conventions for external functions. One style takes a fixed number of arguments. In this conventional style, each comma-separated argument must correspond to a declared parameter of the function. The alternate style permits the use of a variable number of arguments. While normally not viewed any differently than the fixed number of argument style, argument lists that support a variable number of elements can be built up programmatically using sets. The semantics for a function taking a variable number of arguments is identical to that of the argument lists for the **send** and **call** statements.

Call method expressions

One interesting aspect of OIL2 methods is that they can also be called as functions. This is done using a **call** statement.

call methodName [**(**argumentList**)**]

As in the **send** statement, the *methodName* is an expression, which allows an arbitrary method to be called at runtime. The argument list is a set expression, which again permits its computation at runtime.

```
    var = call "method1"(1, 2, 3);
```

6. Reserved Words

alias	else	mod
and	extern	nlm
any	external	not
array	exit	oid
assoc	for	optional
break	fixed	or
call	float	return
case	from	set
class	global	send
const	if	string
continue	implicit	switch
default	in	to
do	int	unique
double	int32	while
	int64	

7. Index