

An Overview of Objective Modula-2

Benjamin Kowarsch

Sunrise Telephone Systems KK

<http://www.sunrisetel.co.jp>

Abstract

This document provides an overview of *Objective Modula-2*, a reflective, object oriented programming language which features both static and dynamic typing.

Objective Modula-2 is a hybrid of Modula-2 and Smalltalk, where Objective-C served as a design blueprint. It retains most of Modula-2's features and syntax, most importantly data encapsulation via modules, the use of explicit import lists and strong type checking for statically defined data objects, while all of its object oriented features are derived from Objective-C, a hybrid of C and Smalltalk. With the exception of the Smalltalk derived message passing syntax, which has been incorporated into *Objective Modula-2* as is, all other Objective-C derived language features have been cast into a Modula-2 like syntax.

Objective Modula-2 is designed to use the Objective-C runtime and it has all the capabilities of Objective-C. Classes written in *Objective Modula-2* can be used within Objective-C and vice versa. Yet, by virtue of its core language, Modula-2 being safer than Objective-C's core language, C, it can be considered a safer language than Objective-C.

Objective Modula-2 is predominantly aimed at Cocoa and GNUstep development and in particular at software developers who would like to make use of the Cocoa and GNUstep APIs from within a type-safe core language in general or from within a language of the Pascal family in particular.

Language Basics

Objective Modula-2 is a "thin" layer on top of a subset of Modula-2 as defined in Wirth's third edition of *Programming in Modula-2*. Features omitted in this subset are local modules, variant records, no subrange types, no subrange array indices and no `WITH DO` statement, all of which have also been omitted in Oberon, a language which Wirth designed as a successor to Modula-2.

Language extensions for object-oriented features, including message passing are derived from Objective-C, which itself derived its object system and associated message passing syntax from Smalltalk.

In addition to Modula-2 and Objective-C/Smalltalk derived features and syntax, *Objective Modula-2* has several other, minor language extensions:

- Built-in base type UNICHAR for unicode characters
- Pragma delimiters < * and * > as in ISO Modula-2
- Concatenation of string literals as in ISO Modula-2
- Single character string literals are compatible with CHAR
- C-style /* and */ comments and BCPL style // comments
- Postfix ++ and -- operators may be used as a shorthand for INC and DEC
- String literals may contain escaped characters \0, \n, \r, \t, \\, \' and \"
- Variables are exported immutable and parameters may be declared immutable
- Elements of enumeration types are referenced by qualified identifiers
- Enumeration types may be declared as a union of other enumeration types

Unicode Characters

Objective Modula-2 has a built-in base type UNICHAR for unicode characters:

```
VAR uc : UNICHAR; (* Unichar variable declaration *)
uc := 0B3F4U; (* Unichar literals are designated by U *)
```

Listing 1: unicode character type and literal

Immutable Export of Variables

Variables declared within a definition module are exported as immutable objects. An imported variable may not be assigned to within the scope of an importing module¹.

Immutable Procedure Parameters

Formal parameters may be declared immutable in the header of a procedure. An immutable parameter may not be assigned to within the scope of the procedure.

```
PROCEDURE Foobar (IMMUTABLE foo : Foo; IMMUTABLE VAR bar : Bar)
```

Listing 2: immutable procedure parameters

Enumeration Type Elements

In Modula-2, name conflicts can occur when importing enumeration types. To avoid this, *Objective Modula-2* requires the elements of enumeration types to be qualified².

¹ A compiler pragma EXPORT_RW is provided to export variables as mutable objects.

² A compiler pragma IMPORT_ENUMS_UQ is provided to use unqualified notation within an importing module.

Given the following enumeration type declaration:

```
TYPE Status = ( failure, success );
```

the elements of enumeration type `Status` are referenced as follows:

```
foo := Status.failure; bar := Status.success;
```

Enumeration Type Unions

Enumeration types may be declared as a union of other enumeration types.

```
TYPE
  Foo = ( foo, bar, baz );
  Bar = ( bam, boo );
  Baz = ( jar, jam );
  FooBarBaz = UNION OF Foo, Bar, Baz;
```

Listing 3: declaring an enumeration type as a union of other enumeration types

Message Passing

Objective Modula-2 uses the Objective-C object model, which is based on the Smalltalk object model and thus distinct from the object model of Simula, used by C++ and other programming languages. This distinction is semantically important. The main differences are that instead of "calling a method", one "sends a message" and parameters are interleaved within the method name. An object called `obj` whose class has a method `doSomething` implemented is said to "respond" to the message `doSomething`. The Objective-C based syntax for sending a `doSomething` message to `obj` is `[obj doSomething]`; As an alternative, *Objective Modula-2* also provides a more Smalltalk-like syntax which does not use brackets:

```
foo := [[FooClass alloc] init]; (* Objective-C style syntax *)
foo := `FooClass alloc init; (* Smalltalk style alternative *)
```

Listing 4: message passing syntax

Dynamic Typing

A major difference to statically typed languages such as C++ and Java is that in *Objective Modula-2* it is possible to send messages to objects that do "not" respond to them. This is because the object oriented part of *Objective Modula-2* is dynamically typed, just like Objective-C. This means that it is possible to send a message to an object which does not have a method specified in its interface to respond to that message. This may seem like a bad idea, but in fact this allows for a great level of

flexibility - in *Objective Modula-2* an object can "capture" this message, and depending on the object, it can pass the message on to a different object which can respond to the message correctly and appropriately, or pass the message on yet again. In Objective-C parlance this is called delegation, also referred to as "message forwarding". An error handler can be used in case the message cannot be forwarded. However if the object does not respond to the message, nor forward it, nor handle the error, then a runtime error occurs. An example of a dynamically typed object in *Objective Modula-2* is shown below:

```
VAR anObject : OBJECT;
```

Listing 5: declaring a dynamically typed object

The type `OBJECT` in *Objective Modula-2* is the equivalent of type `id` in Objective-C. Like other dynamically typed languages, there is the potential problem of an endless stream of runtime errors that come from sending the wrong message to the wrong object. However, *Objective Modula-2* allows the programmer to optionally specify the class of an object, and in such cases the compiler will treat the object as statically typed. An example of a statically typed object in *Objective Modula-2* is shown below:

```
VAR aString : NSString;
```

Listing 6: declaring a statically typed object

Unlike Objective-C where the declaration of a variable of a class type requires the class identifier to be referenced with a preceding `*` operator, in *Objective Modula-2* the class identifier is not preceded by `POINTER TO`.

Class Interfaces and Implementations

In *Objective Modula-2* the facility of a module is used to define and implement classes. A class' interface is defined by a definition module and its implementation by an implementation module. The Modula-2 derived syntax for definition and implementation modules has been augmented accordingly. Unlike Objective-C where the interface and implementation of a class can be placed within the same file, in *Objective Modula-2* they are required to be in separate compilation units.

Class Interfaces

The interface of a class is represented by a definition module. An example of a class definition module defining a class called `FooBar` as a descendant of superclass `NSObject` is shown below:

```
(* Define class FooBar as a subclass of NSObject *)
DEFINITION MODULE FooBar : NSObject;

  (* instance variables *)
  VAR
    foo : Foo;
    bar : Bar;

  (* constructor and initialiser *)
  CLASS METHOD newWithFoo: (foo : Foo)
    andBar: (bar : Bar) : OBJECT;

  (* accessor for foo *)
  INSTANCE METHOD foo : Foo;

  (* mutator for foo *)
  INSTANCE METHOD setFoo: (foo : Foo);

END FooBar.
```

Listing 7: class definition module

Class Implementations

While the class definition module only declares instance variables and prototypes of the class' methods, the implementation of methods is placed in the class implementation module. An example of a corresponding class implementation module for class FooBar is shown below:

```
IMPLEMENTATION MODULE FooBar : NSObject;

(* constructor and initialiser *)
CLASS METHOD newWithFoo: (foo : Foo)
                    andBar: (bar : Bar) : OBJECT;
VAR
    thisInstance : FooBar;
BEGIN
    thisInstance := [[FooBar alloc] init];
    thisInstance^.foo := foo;
    thisInstance^.bar := bar;
    RETURN thisInstance;
END newWithFoo;

(* accessor for foo *)
INSTANCE METHOD foo : Foo;
BEGIN
    RETURN SELF^.foo;
END foo;

(* mutator for foo *)
INSTANCE METHOD setFoo: (foo : Foo);
BEGIN
    SELF^.foo := foo;
END setFoo;

END FooBar.
```

Listing 8: class implementation module

Methods are written in a different way than Modula-2 procedures. A procedure in both Modula-2 and *Objective Modula-2* follows this general form:

```
PROCEDURE ProcedureName (parameter : FormalType) : ReturnedType;
  VAR
    (* local variables *)
  BEGIN
    (* instructions *)
    RETURN result;
  END;
```

Listing 9: procedure

However, the syntax for procedure headers cannot be used for methods because of the way in which the Smalltalk derived Objective-C object model and its message passing works. Therefore, additional syntax for declaring class and instance methods has been added to the language as shown in the FooBar class implementation example. This new syntax allows to define methods which follow the Smalltalk derived notation for sending messages. An example of how the methods in the FooBar class are used is shown below:

```
MODULE UseFoobar;
IMPORT FooBar;
CONST
  someFoo = 1; someBar = 2;
VAR
  foobar : FooBar;
BEGIN
  (* allocate and initialise a new instance of FooBar *)
  foobar := [FooBar newWithFoo:someFoo andBar:someBar];

  (* sending a foo message to foobar *)
  someFoo := [foobar foo];

  (* sending a setFoo: message to foobar *)
  [foobar setFoo:someFoo];
END UseFoobar.
```

Listing 10: module using a class and its methods

Refining Classes

The Objective-C object model defines an instrument to refine an existing class by overloading existing methods or adding additional methods. In Objective-C parlance this is called a category. The methods within a category are added to a class at run time. Thus, categories permit the programmer to modify or add methods to an existing class without the need to recompile that class or even have access to its source code. For example, if an underlying object system does not contain a spell checker in its String implementation, then it can be added without modifying the String source code. Methods within categories become indistinguishable from the methods in a class when the program is run. A category has full access to all of the instance variables within the class, including private variables.

Categories provide an elegant solution to the fragile base class problem for methods. If a method declaration in a category has the same method signature as an existing method in a class, the category's method is adopted. Thus categories can not only add methods to a class, but also replace existing methods. This feature can be used to fix bugs in other classes by rewriting their methods, or to cause a global change to a class's behavior within a program. If two categories have methods with the same method signature, then it is undefined which category's method is adopted.

To allow the declaration and implementation of categories, additional syntax has been added in *Objective Modula-2*. An example of a class definition module which defines the interface of a category called `AdditionsToNSString` targeting the `NSString` class is shown below:

```
DEFINITION MODULE AdditionsToNSString REFINES NSString;
  (* declare a new method to be added to the target class *)
  INSTANCE METHOD stringByCollapsingWhitespace : NSString;
END AdditionsToNSString.
```

Listing 11: category definition module

An example of a corresponding implementation module for the category `AdditionsToNSString` is shown below:

```
IMPLEMENTATION MODULE AdditionsToNSString REFINES NSString;
  INSTANCE METHOD stringByCollapsingWhitespace : NSString;
    (* code to implement the method *)
  END stringByCollapsingWhitespace;
END AdditionsToNSString.
```

Listing 12: category implementation module

Multiple Inheritance

The Objective-C object model defines an instrument for multiple inheritance of specification (but not implementation). In Objective-C parlance this is called a protocol. There are two types of protocols: ad-hoc protocols, called *informal protocols*, and compiler enforced protocols called *formal protocols*.

An informal protocol is a list of methods that a class may implement. It is specified in the documentation, since it has no presence in the language. Informal protocols often include optional methods, where implementing the method can change the behavior of a class. For example, a text field class might have a delegate that should implement an informal protocol with an optional autocomplete method. The text field discovers whether the delegate implements that method (via reflection), and if so, calls it to support autocomplete. The methods suggested by an informal protocol are often implemented using categories.

A formal protocol is a collection of method declarations that any given class may adopt by implementing the methods declared by the protocol. A class which adopts a protocol must implement all methods declared by that protocol. To allow the declaration of formal protocols, additional syntax has been added in *Objective Modula-2*. An example of a protocol definition module which declares a formal protocol called `Foobaring` is shown below:

```

PROTOCOL MODULE Foobaring;
OPTIONAL
(* method declarations for optional methods *)
REQUIRED
(* method declarations for required methods *)
END Foobaring.
```

Listing 13: protocol module

To allow class modules to be declared to adopt formal protocols, further syntax has been added to *Objective Modula-2*. An example of a class definition module declaring a class `Blob` which adopts the `Foobaring` protocol is shown below:

```

DEFINITION MODULE Blob : NSObject;
IMPORT PROTOCOL Foobaring;
(* instance variables *)
(* method declarations *)
END Blob.
```

Listing 14: class definition module adopting a protocol

An example for the corresponding implementation module of class `Blob` adopting protocol `Foobaring` is shown below:

```
IMPLEMENTATION MODULE Blob : NSObject;
IMPORT PROTOCOL Foobaring;
(* method implementations *)
(* implementations of methods declared by protocol Foobaring *)
END Blob.
```

Listing 15: class implementation module adopting a protocol

Instance Variable Access Modes

By default, the instance variables of a class are visible and accessible to classes residing within the same link image. This is not always desirable.

The class itself and its categories always need to have direct access to the instance variables, but outside of the class and its categories, it is often desirable that they should only be accessible via accessor and mutator methods. In some cases it may however, be desirable to make them public to all.

In order to accommodate the afore mentioned scenarios, different access modes for instance variables are provided.

In *Objective Modula-2* there are at present four access modes³ for instance variables: public, link-image, protected and private. The default access mode is link-image.

Access Mode	Qualifier	Visibility
Public	PUBLIC	anywhere
Link Image (aka Package)	none (default)	within the class itself, its categories, subclasses and any classes linked into the same image file
Protected	PROTECTED	within the class itself, its categories and subclasses
Private	PRIVATE	within the class itself and its categories

³ The equivalent access modes in Objective-C are called `@public`, `@package`, `@protected` and `@private`. However, the `@package` access mode was only introduced with Objective-C 2.0. Instance variables declared with access mode `@package` will instead have access mode `@public` when their classes are used from classes compiled under Objective-C prior to version 2.0.

An example of instance variables with different access modes is shown below:

```
DEFINTION MODULE FooBar : NSObject;  
PUBLIC VAR  
    foo : Foo; (* Any class can access foo *)  
PROTECTED VAR  
    bar : Bar; (* Subclasses of FooBar can access bar *)  
PRIVATE VAR  
    baz : Baz; (* Only FooBar and its categories can access baz *)
```

Listing 16: instance variables with different access modes

Objective-C Runtime Exception Handling

Further syntax has been added in *Objective Modula-2* in order to support Objective-C runtime exception handling. An example is shown below:

```
TRY  
    (* statements during which a runtime exception may occur *)  
ON exception DO  
    (* statements to be executed if specified exception occurs *)  
CONTINUE  
    (* statements to be executed in any event *)  
END;
```

Listing 17: exception handling

Synchronising Thread Execution

Further syntax has been added in *Objective Modula-2* in order to support Objective-C runtime compatible thread synchronisation. An example is shown below:

```
INSTANCE METHOD doSomeCriticalStuff : Foo;  
    (* non-critical code *)  
    CRITICAL ( SELF )  
        (* critical code *)  
    END;  
    (* non-critical code *)  
END doSomeCriticalStuff;
```

Listing 18: synchronising a thread

Appendix A: Outlook on Future Additions

Further language additions are being considered, most notably conditional compilation, per-object garbage collection, name space compatibility with the Objective-C runtime environment, and qualified enumeration type members.

Conditional compilation

Conditional compilation *will* be introduced into *Objective Modula-2* through using compiler pragmas as shown in the example below:

```
CONST
  <* IF foo > bar *>
    foobarbaz = foo;
  <* ELSIF foo = bar *>
    foobarbaz = baz;
  <* ELSE *>
    foobarbaz = bar;
  <* ENDIF *>
```

Listing 19: conditional compilation

In addition to conditional compilation pragmas, a built-in ternary function procedure, similar to C's ternary operator but with a Modula-2 like syntax *may* also be introduced. An example how this might look is shown below:

```
foobar := TEVAL(foo > bar, baz, bam);
```

Listing 20: ternary function

which would result in compile time replacement of the right hand expression with either baz or bam, depending on the evaluation of the expression `foo > bar`.

Automatic Garbage Collection

It is envisaged to support automatic garbage collection through adding methods and classes, similar to the way autorelease pools are implemented in Objective-C. This would allow the programmer to choose a memory management scheme on a per object basis. Objects could be manually managed, autoreleased or garbage collected simply by sending a message. An example how objects could be declared using different coexisting memory management schemes is shown below:

```
VAR
  (* declaring a garbage collected object *)
```

```
garbageCollectedString : NSString :=
    [[NSString alloc] init] gc];
(* declaring a an autorelease pool object *)
autoreleaseString : NSString :=
    [[NSString alloc] init] autorelease];
```

Listing 21: possible syntax for garbage collection

With Objective-C 2.0, Apple Inc. has introduced automatic garbage collection into its Objective-C runtime and it is possible to mix reference counted memory management and automatic garbage collection, but per-object selection requires further study.

Name Space Compatibility with Objective-C

In Modula-2, name conflicts are avoided by qualified import of objects with identical names, which are then referenced by qualified identifier. An example is shown below:

```
IMPORT FooLib, BarLib, BazLib;
FooLib.write(); BarLib.write(); BazLib.write();
```

Listing 22: qualified import in Modula-2

In *Objective Modula-2* classes are modules and therefore, the class identifier is not qualified as it represents the module itself:

```
IMPORT FoobarClass;
VAR
    foobar : FoobarClass := [[FoobarClass alloc] init];
```

Listing 23: import of classes in Objective Modula-2

In order to implement name spaces for classes in *Objective Modula-2*, classes would need to be defined within a library module, for example:

```
DEFINITION MODULE ClassLib;
    CLASS FooClass : NSObject;
        (* instance variable and method declarations *)
    END FooClass;
    CLASS BarClass : NSObject;
        (* instance variable and method declarations *)
    END BarClass;
END ClassLib.
```

Listing 24: class declarations within a library module

in which case the classes could be imported from the library module using qualified import and consequently they would then be referenced by qualified identifiers:

```
IMPORT ClassLib;

VAR
    foo : ClassLib.FooClass := [[ClassLib.FooClass alloc] init];
    bar : ClassLib.BarClass := [[ClassLib.BarClass alloc] init];
```

Listing 25: qualified class identifiers

Unfortunately, there is a compatibility issue with qualified class identifiers and the Objective-C runtime. Compilers for languages which support name spaces generally combine the name of the compilation unit with the name of the object, a process called name mangling. For example, the actual symbol name for class `FooClass` might be `ClassLib$FooClass` or similar. However, compilers for languages which do not support name spaces, do not understand mangled names.

Thus, if classes were made definable within library modules and importable via qualified import in *Objective Modula-2*, then the resulting qualified class identifiers could not be easily referenced from classes written in Objective-C. Any such reference to the class within Objective-C code would need to use the mangled class name verbatim, for example:

```
import "ClassLib.h"

ClassLib$FooClass *foo = [[ClassLib$FooClass alloc] init];
ClassLib$BarClass *bar = [[ClassLib$BarClass alloc] init];
```

Listing 26: inability of Objective-C to unmangle name-mangled symbols

In principle it would be possible to define a better readable alias name by using C preprocessor macros and use that instead of the mangled name within Objective-C source code. However, the translation scheme for converting qualified identifiers into name mangled symbols will need to be very carefully selected because mangled names must be legal within the naming conventions of the Objective-C runtime and they must not clash with any internal symbols or reserved names in use by the implementors of the runtime. This will require further study.

However, subject to resolving this name compatibility issue, it is *envisaged* that name spaces for classes *will* be introduced into *Objective Modula-2* in the future on the basis of class declarations placed within library modules. The following section is a proposal how the classes may in future be declarable within library modules.

Class Libraries Proposal

It is proposed to allow classes to be bundled into class libraries in *Objective Modula-2* by placing class definitions and implementations inside a library module. The example below shows how a class cluster might be defined using such a facility:

```

DEFINITION MODULE Blobs;
(* the base class of this class cluster *)
CLASS Blob : NSObject;
// instance variables
// method declarations
END Blob;
(* the mutable variant of the base class *)
CLASS MutableBlob : Blob;
// instance variables
// method declarations
END MutableBlob;
(* the attributed variant of the base class *)
CLASS AttributedBlob : Blob;
// instance variables
// method declarations
END AttributedBlob;
(* the mutable variant of the attributed variant *)
CLASS MutableAttributedBlob : AttributedBlob;
// instance variables
// method declarations
END MutableAttributedBlob;
END Blobs.

```

Listing 27: library definition module defining a class cluster

Classes within module Blobs could then be imported qualified:

```
IMPORT Blobs; (* to be referenced as Blobs.Blob etc *)
```

and also unqualified:

```
FROM Blobs IMPORT Blob; (* to be referenced as Blob *)
```

However, from within Objective-C code, the classes could only be reliably referenced by a macro to translate the mangled class names into a C compliant identifier.

An example of the corresponding implementation module for Blobs is shown below:

```
IMPLEMENTATION MODULE Blobs;
(* the base class of this class library *)
CLASS Blob : NSObject;
// method implementations
END Blob;
(* the mutable variant of the base class *)
CLASS MutableBlob : Blob;
// method implementations
END MutableBlob;
(* the attributed variant of the base class *)
CLASS AttributedBlob : Blob;
// method implementations
END AttributedBlob;
(* the mutable variant of the attributed variant *)
CLASS MutableAttributedBlob : AttributedBlob;
// method implementations
END MutableAttributedBlob;
END Blobs.
```

Listing 28: library implementation module implementing a class cluster

Lowercase Synonyms for Reserved Words

Specifically for the convenience of Pascal programmers, lowercase synonyms for reserved words *may* be added in the future. When introduced, the facility may be switched-off via pragma to allow the use of legacy Modula-2 source code.

Extensible Record Types

Extensible record types as found in Oberon *may* be added in the future.

Appendix B: Project Related Information

Reference Compiler

An experimental compiler for *Objective Modula-2* for the purpose of testing language design concepts has been under development since 2006. As the language definition has matured, a new compiler, derived from the earlier experimental compiler, is being developed with the aim to produce a reference implementation. The compiler uses a recursive descent parser and initially it will generate Objective-C source code.

Source code for the compiler is being made available under a peer-review license until the reference implementation is complete, at which point the source code will be relicensed under a proper open source software license.

Further Reading

<http://www.sunrisetel.net/software/devtools/objective-modula-2.shtml>

License

This document is released under the Creative Commons License.