A

Objective C Language Summary

Objective C adds a small number of constructs to the C language and defines a handful of conventions for effectively interacting with the run-time system. This appendix lists all the additions to the language, but doesn't go into great detail. For more information, see Chapters 2 and 3 of this manual. For a more formal presentation of Objective C syntax, see Appendix B, ªReference Manual for the Objective C Language,º which follows this summary.

Messages

Message expressions are enclosed in square brackets:

[*receiver message***]**

The *receiver* can be:

- A variable or expression that evaluates to an object (including the variable **self**)
- A class name (indicating the class object)
- **super** (indicating an alternative search for the method implementation)

The *message* is the name of a method plus any arguments passed to it.

Defined Types

The principal types used in Objective C are defined in **objc/objc.h**. They are:

id	An object (a pointer to its data structure)
Class	A class object (a pointer to the class data structure)
SEL	A selector, a compiler-assigned code that identifies a method name
IMP	A pointer to a method implementation that returns an **id**
BOOL	A boolean value, either YES or NO

id can be used to type any kind of object, class or instance. In addition, class names can be used as type names to statically type instances of a class. A statically typed instance is declared to be a pointer to its class or to any class it inherits from.

The **objc.h** header file also defines these useful terms:

nil	A null object pointer, (**id**)0
Nil	A null class pointer, (Class)0

Preprocessor Directives

The preprocessor understands these new notations:

#import	Imports a header file. This directive is identical to **#include**, except that it won't include the same file more than once.
//	Begins a comment that continues to the end of the line.

Compiler Directives

Directives to the compiler begin with ª@º. The following directives are used to declare and define classes, categories, and protocols:

@interface	Begins the declaration of a class or category interface
@implementation	Begins the definition of a class or category
@protocol	Begins the declaration of a formal protocol
@end	Ends the declaration/definition of a class, category, or protocol

The following mutually-exclusive directives specify the visibility of instance variables:

@private	Limits the scope of an instance variable to the class that declares it
@protected	Limits instance variable scope to declaring and inheriting classes
@public	Removes restrictions on the scope of instance variables

The default is **@protected**.

In addition, there are directives for these particular purposes:

@class	Declares the names of classes defined elsewhere
@selector(*method*)	Returns the compiled selector that identifies *method*
@protocol(*name*)	Returns the *name* protocol (an instance of the Protocol class)
@encode(*spec*)	Yields a character string that encodes the type structure of *spec*
@defs(*classname*)	Yields the internal data structure of *classname* instances

Classes

A new class is declared with the **@interface** directive. It imports the interface file for its superclass:

#import "*ItsSuperclass***.h"**

@interface *ClassName* **:** *ItsSuperclass* **<** *protocol list* **>**
{
    instance variable declarations
}
method declarations
@end

Everything but the compiler directives and class name is optional. If the colon and superclass name are omitted, the class is declared to be a new root class. If any protocols are listed, the header files where they're declared must also be imported.

A class definition imports its own interface:

#import "*ClassName***.h"**

@implementation *ClassName*
method definitions
@end

Categories

A category is declared in much the same way as a class. It imports the interface file that declares the class:

#import "*ClassName***.h"**

@interface *ClassName* **(** *CategoryName* **)** **<** *protocol list* **>**
method declarations
@end

The protocol list and method declarations are optional. If any protocols are listed, the header files where they're declared must also be imported.

Like a class definition, a category definition imports its own interface:

#import "*CategoryName***.h"**

@implementation *ClassName* **(** *CategoryName* **)**
method definitions
@end

Formal Protocols

Formal protocols are declared using the **@protocol** directive:

@protocol *ProtocolName* **<** *protocol list* **>**
method declarations
@end

The list of incorporated protocols and the method declarations are optional. The protocol must import the header files that declare any protocols it incorporates.

Within source code, protocols are referred to using the similar **@protocol()** directive, where the parentheses enclose the protocol name.

Protocol names listed within angle brackets (<...>) are used to do three different things:

- In a protocol declaration, to incorporate other protocols (as shown above)
- In a class or category declaration, to adopt the protocol (as shown under ªClassesº and ªCategoriesº above)
- In a type specification, to limit the type to objects that conform to the protocol

Within protocol declarations, these type qualifiers support remote messaging:

oneway	The method is for asynchronous messages and has no valid return.

in	The argument passes information to the remote receiver.
out	The argument gets information returned by reference.
inout	The argument both passes information and gets information.
bycopy	A copy of the object, not a proxy, should be passed or returned.

Method Declarations

The following conventions are used in method declarations:

- A ª+º precedes declarations of class methods.
- A ª-º precedes declarations of instance methods.
- Arguments are declared after colons (**:**). Typically, a label describing the argument precedes the colon. Both labels and colons are considered part of the method name.
- Argument and return types are declared using the C syntax for type casting.
- The default return and argument type for methods is **id**, not **int** as it is for functions. (However, the modifier **unsigned** when used without a following type always means **unsigned int**)

Method Implementations

Each method implementation is passed two hidden arguments:

- The receiving object (**self**)
- The selector for the method (**_cmd**)

Within the implementation, both **self** and **super** refer to the receiving object. **super** replaces **self** as the receiver of a message to indicate that only methods inherited by the implementation should be performed in response to the message.

Methods with no other valid return typically return **self**.

Naming Conventions

The names of files that contain Objective C source code have a ª.mº extension. Files that declare class and category interfaces or that declare protocols have the ª.hº extension typical of header files.

Class, category, and protocol names generally begin with an uppercase letter; the names of methods and instance variables typically begin with a lowercase letter. The names of variables that hold instances usually also begin with lowercase letters.

In Objective C, identical names that serve different purposes don't clash. Within a class, names can be freely assigned:

- A class can declare methods with the same names as methods in other classes.
- A class can declare instance variables with the same names as variables in other classes.
- An instance method can have the same name as a class method.
- A method can have the same name as an instance variable.

Likewise, protocols and categories of the same class have protected name spaces:

- A protocol can have the same name as a class, a category, or anything else.
- A category of one class can have the same name as a category of another class.

However, class names are in the same name space as variables and defined types. A program can't have a global variable with the same name as a class.