Objective-C

Objective-C is a general-purpose, object-oriented programming language that adds Smalltalk-style messaging to the C programming language. It is the main programming language used by Apple for the OS X and iOS operating systems, and their respective application programming interfaces (APIs), Cocoa and Cocoa Touch.

The programming language Objective-C was originally developed in the early 1980s. It was selected as the main language used by NeXT for its NeXTSTEP operating system, from which OS X and iOS are derived.[2] Generic Objective-C programs that do not use the Cocoa or Cocoa Touch libraries, or those using parts that may be ported or reimplemented for other systems, can also be compiled for any system supported by GCC or Clang.

Objective-C	
Paradigm(s)	Reflective, class-based object-oriented
Designed by	Brad Cox and Tom Love
Developer	Apple Inc.
Appeared in	1983
Stable release	2.0
Typing discipline	static, dynamic, weak
Major implementations	Clang, GCC
Influenced by	C, Smalltalk
Influenced	Java, Nu, Objective-J, TOM, Swift[1]
OS	Cross-platform
Filename extension(s)	.h, .m, .mm
Website	developer.apple.com/library/mac/navigation (http://developer.apple.com/library/mac/navigation)
Objective-C Programming at Wikibooks	

Objective-C source code program files usually have `.m` filename extensions, while Objective-C header files have `.h` extensions, the same as for C header files. Objective-C++ files are denoted with a `.mm` file extension.

Contents

History

Objective-C was created primarily by Brad Cox and Tom Love in the early 1980s at their company Stepstone.[3] Both had been introduced to Smalltalk while at ITT Corporation's Programming Technology Center in 1981. The earliest work on Objective-C traces back to around that time.[4] Cox was intrigued by problems of true reusability in software design and programming. He realized that a language like Smalltalk would be invaluable in building development environments for system developers at ITT. However, he and Tom Love also recognized that backward compatibility with C was critically important in ITT's telecom engineering milieu.[5]

Cox began writing a pre-processor for C to add some of the capabilities of Smalltalk. He soon had a working implementation of an object-oriented extension to the C language, which he called "OOPC" for Object-Oriented Pre-Compiler.[6] Love was hired by Schlumberger Research in 1982 and had the opportunity to acquire the first commercial copy of Smalltalk-80, which further influenced the development of their brainchild.

In order to demonstrate that real progress could be made, Cox showed that making interchangeable software components really needed only a few practical changes to existing tools. Specifically, they needed to support objects in a flexible manner, come supplied with a usable set of libraries, and allow for the code (and any resources needed by the code) to be bundled into a single cross-platform format.

Love and Cox eventually formed a new venture, Productivity Products International (PPI), to commercialize their product, which coupled an Objective-C compiler with class libraries. In 1986, Cox published the main description of Objective-C in its original form in the book *Object-Oriented Programming, An Evolutionary Approach*. Although he was careful to point out that there is more to the problem of reusability than just the language, Objective-C often found itself compared feature for feature with other languages.

Popularization through NeXT

In 1988, NeXT licensed Objective-C from StepStone (the new name of PPI, the owner of the Objective-C trademark) and extended the GCC compiler to support Objective-C. NeXT developed the AppKit and Foundation Kit libraries on which the NeXTstep user interface and Interface Builder were based. While the NeXT workstations failed to make a great impact in the marketplace, the tools were widely lauded in the industry. This led NeXT to drop hardware production and focus on software tools, selling NeXTstep (and OpenStep) as a platform for custom programming.

In order to circumvent the terms of the GPL, NeXT had originally intended to ship the Objective-C frontend separately, allowing the user to link it with GCC to produce the compiler executable. After being initially accepted by Richard M. Stallman, this plan was rejected after Stallman consulted with GNU's lawyers and NeXT agreed to make Objective-C part of GCC.[7]

The work to extend GCC was led by Steve Naroff, who joined NeXT from StepStone. The compiler changes were made available as per GPL license terms, but the runtime libraries were not, rendering the open source contribution unusable to the general public. This led to other parties developing such runtime libraries under open source license. Later, Steve Naroff was also principal contributor to work at Apple to build the Objective-C frontend to Clang.

The GNU project started work on its free software implementation of Cocoa, named GNUstep, based on the OpenStep standard.[8] Dennis Glatting wrote the first GNU Objective-C runtime in 1992. The GNU Objective-C runtime, which has been in use since 1993, is the one developed by Kresten Krab Thorup when he was a university student in Denmark. Thorup also worked at NeXT from 1993 to 1996.

Apple development and Swift

After acquiring NeXT in 1996, Apple Computer used OpenStep in its new operating system, Mac OS X. This included Objective-C and NeXT's Objective-C based developer tool, Project Builder (now called Xcode), as well as its interface design tool, Interface Builder. Most of Apple's present-day Cocoa API is based on OpenStep interface objects, and is the most significant Objective-C environment being used for active development.

At WWDC 2014, Apple announced plans to replace Objective-C for Cocoa development with its new Swift language, which it characterized as "Objective-C without the C".

Syntax

Objective-C is a thin layer on top of C, and moreover is a *strict superset* of C; it is possible to compile any C program with an Objective-C compiler, and to freely include C code within an Objective-C class.[9][10][11][12][13][14]

Objective-C derives its object syntax from Smalltalk. All of the syntax for non-object-oriented operations (including primitive variables, pre-processing, expressions, function declarations, and function calls) is identical to that of C, while the syntax for object-oriented features is an implementation of Smalltalk-style messaging.

Messages

The Objective-C model of object-oriented programming is based on message passing to object instances. In Objective-C one does not *call a method*; one *sends a message*. This is unlike the Simula-style programming model used by C++. The difference between these two concepts is in how the code referenced by the method or message name is executed. In a Simula-style language, the method name is in most cases bound to a section of code in the target class by the compiler. In Smalltalk and Objective-C, the target of a message is resolved at runtime, with the receiving object itself interpreting the message. A method is identified by a *selector* or `SEL` — a `NUL`-terminated string representing its name — and resolved to a C method pointer implementing it: an `IMP`.[15] A consequence of this is that the message-passing system has no type checking. The object to which the message is directed — the *receiver* — is not guaranteed to respond to a message, and if it does not, it simply raises an exception.[16]

Sending the message `method` to the object pointed to by the pointer `obj` would require the following code in C++:

```
obj->method(argument);
```

In Objective-C, this is written as follows:

```
[obj method:argument];
```

Both styles of programming have their strengths and weaknesses. Object-oriented programming in the Simula (C++) style allows multiple inheritance and faster execution by using compile-time binding whenever possible, but it does not support dynamic binding by default. It also forces all methods to have a corresponding implementation unless they are abstract. The Smalltalk-style programming as used in Objective-C allows messages to go unimplemented, with the method resolved to its implementation at runtime. For example, a message may be sent to a collection of objects, to which only some will be expected to respond, without fear of producing runtime errors. Message passing also does not require that an object be defined at compile time. An implementation is still required for the method to be called in the derived object. (See the dynamic typing section below for more advantages of dynamic (late) binding.)

Interfaces and implementations

Objective-C requires that the interface and implementation of a class be in separately declared code blocks. By convention, developers place the interface in a header file and the implementation in a code file. The header files, normally suffixed .h, are similar to C header files while the implementation (method) files, normally suffixed .m, can be very similar to C code files.

Interface

In other programming languages, this is called a "class declaration".

The interface of a class is usually defined in a header file. A common convention is to name the header file after the name of the class, e.g. `Ball.h` would contain the interface for the class `Ball`.

An interface declaration takes the form:

```
@interface classname : superclassname {
  // instance variables
}
+ classMethod1;
+ (return_type)classMethod2;
+ (return_type)classMethod3:(param1_type)param1_varName;

- (return_type)instanceMethod1With1Parameter:(param1_type)param1_varName;
- (return_type)instanceMethod2With2Parameters:(param1_type)param1_varName param2_callName:(param2_type)pa
@end
```

In the above, plus signs denote class methods, or methods that can be called on the class itself (not on an instance), and minus signs denote instance methods, which can only be called on a particular instance of the class. Class methods also have no access to instance variables.

The code above is roughly equivalent to the following C++ interface:

```
class classname : public superclassname {
 protected:
  // instance variables

 public:
  // Class (static) functions
  static void * classMethod1();
  static return_type classMethod2();
  static return_type classMethod3(param1_type param1_varName);
```

```
// Instance (member) functions
return_type instanceMethod1With1Parameter (param1_type param1_varName);
return_type instanceMethod2With2Parameters (param1_type param1_varName, param2_type param2_varName=defau
};
```

Note that `instanceMethod2With2Parameters:param2_callName:` demonstrates the interleaving of selector segments with argument expressions, for which there is no direct equivalent in C/C++.

Return types can be any standard C type, a pointer to a generic Objective-C object, a pointer to a specific type of object such as NSArray *, NSImage *, or NSString *, or a pointer to the class to which the method belongs (instancetype). The default return type is the generic Objective-C type `id`.

Method arguments begin with a name labeling the argument that is part of the method name, followed by a colon followed by the expected argument type in parentheses and the argument name. The label can be omitted.

```
- (void)setRangeStart:(int)start end:(int)end;
- (void)importDocumentWithName:(NSString *)name withSpecifiedPreferences:
(Preferences *)prefs beforePage:(int)insertPage;
```

Implementation

The interface only declares the class interface and not the methods themselves: the actual code is written in the implementation file. Implementation (method) files normally have the file extension `.m`, which originally signified "messages".[17]

```
@implementation classname
+ (return_type)classMethod
{
 // implementation
}
- (return_type)instanceMethod
{
 // implementation
}
@end
```

Methods are written using their interface declarations. Comparing Objective-C and C:

```
- (int)method:(int)i
{
return [self square_root:i];
}
```

```
int function (int i)
{
return square_root(i);
}
```

The syntax allows pseudo-naming of arguments.

```
- (int)changeColorToRed:(float)red green:(float)green blue:(float)blue;

[myColor changeColorToRed:5.0 green:2.0 blue:6.0];
```

Internal representations of a method vary between different implementations of Objective-C. If myColor is of the class `Color`, instance method `–changeColorToRed:green:blue:` might be internally labeled `_i_Color_changeColorToRed_green_blue`. The `i` is to refer to an instance method, with the class and then method names appended and colons changed to underscores. As the order of parameters is part of the method name, it cannot be changed to suit coding style or expression as with true named parameters.

However, internal names of the function are rarely used directly. Generally, messages are converted to function calls defined in the Objective-C runtime library. It is not necessarily known at link time which method will be called because the class of the receiver (the object being sent the message) need not be known until runtime.

Instantiation

Once an Objective-C class is written, it can be instantiated. This is done by first allocating an uninitialized instance of the class (an object) and then by initializing it. An object is not fully functional until both steps have been completed. These steps should be accomplished with a single line of code so that there is never an allocated object that hasn't undergone initialization (and because it is not advisable to keep the intermediate result since `–init` can return a different object than that which it is called on).

Instantiation with the default, no-parameter initializer:

```
MyObject *o = [[MyObject alloc] init];
```

Instantiation with a custom initializer:

```
MyObject *o = [[MyObject alloc] initWithString:myString];
```

In the case where no custom initialization is being performed, the "new" method can often be used in place of the alloc-init messages:

```
MyObject *o = [MyObject new];
```

Also, some classes implement class method initializers. Like +new, they combine +alloc and –init, but unlike +new, they return an autoreleased instance. Some class method initializers take parameters:

```
MyObject *o = [MyObject object];
MyObject *o2 = [MyObject objectWithString:myString];
```

The *alloc* message allocates enough memory to hold all the instance variables for an object, sets all the instance variables to zero values, and turns the memory into an instance of the class; at no point during the initialization is the memory an instance of the superclass.

The *init* message performs the set-up of the instance upon creation. The *init* method is often written as follows:

```
- (id)init {
    self = [super init];
    if (self) {
        // perform initialization of object here
    }
    return self;
}
```

In the above example, notice the `id` return type. This type stands for "pointer to any object" in Objective-C (See the Dynamic typing section).

The initializer pattern is used to assure that the object is properly initialized by its superclass before the init method performs its initialization. It performs the following actions:

1. self = [super init]

   Sends the superclass instance an *init* message and assigns the result to *self* (pointer to the current object).

2. if (self)

   Checks if the returned object pointer is valid before performing any initialization.

3. return self

   Returns the value of self to the caller.

A non-valid object pointer has the value *nil*; conditional statements like "if" treat nil like a null pointer, so the initialization code will not be executed if [super init] returned nil. If there is an error in initialization the init method should perform any necessary cleanup, including sending a "release" message to self, and return *nil* to indicate that initialization failed. Any checking for such errors must only be performed after having called the superclass initialization to ensure that destroying the object will be done correctly.

If a class has more than one initialization method, only one of them (the "designated initializer") needs to follow this pattern; others should call the designated initializer instead of the superclass initializer.

Protocols

In other programming languages, these are called "interfaces".

Objective-C was extended at NeXT to introduce the concept of multiple inheritance of specification, but not implementation, through the introduction of protocols. This is a pattern achievable either as an abstract multiple inherited base class in C++, or as an "interface" (as in Java and C#). Objective-C makes use of ad hoc protocols called *informal protocols* and compiler-enforced protocols called *formal protocols*.

An informal protocol is a list of methods that a class can opt to implement. It is specified in the documentation, since it has no presence in the language. Informal protocols are implemented as a category (see below) on NSObject and often include optional methods, which, if implemented, can change the behavior of a class. For example, a text field class might have a delegate that implements an informal protocol with an optional method for performing auto-completion of user-typed text. The text field discovers whether the delegate implements that method (via reflection) and, if so, calls the delegate's method to support the auto-complete feature.

A formal protocol is similar to an interface in Java, C#, and Ada 2005. It is a list of methods that any class can declare itself to implement. Versions of Objective-C before 2.0 required that a class must implement all methods in a protocol it declares itself as adopting; the compiler will emit an error if the class does not implement every method from its declared protocols. Objective-C 2.0 added support for marking certain methods in a protocol optional, and the compiler will not enforce implementation of optional methods.

A class must be declared to implement that protocol to be said to conform to it. This is detectable at runtime. Formal protocols cannot provide any implementations; they simply assure callers that classes that conform to the protocol will provide implementations. In the NeXT/Apple library, protocols are frequently used by the Distributed Objects system to represent the capabilities of an object executing on a remote system.

The syntax

```
@protocol NSLocking
- (void)lock;
- (void)unlock;
@end
```

denotes that there is the abstract idea of locking. By stating in the class definition that the protocol is implemented,

```
@interface NSLock : NSObject <NSLocking>
//...
@end
```

instances of NSLock claim that they will provide an implementation for the two instance methods.

Dynamic typing

Objective-C, like Smalltalk, can use dynamic typing: an object can be sent a message that is not specified in its interface. This can allow for increased flexibility, as it allows an object to "capture" a message and send the message to a different object that can respond to the message appropriately, or likewise send the message on to another object. This behavior is known as *message forwarding* or *delegation* (see below). Alternatively, an error handler can be used in case the message cannot be forwarded. If an object does not forward a message, respond to it, or handle an error, then the system will generate a runtime exception.[18] If messages are sent to *nil* (the null object pointer), they will be silently ignored or raise a generic exception, depending on compiler options.

Static typing information may also optionally be added to variables. This information is then checked at compile time. In the following four statements, increasingly specific type information is provided. The statements are equivalent at runtime, but the additional information allows the compiler to warn the programmer if the passed argument does not match the type specified.

```
- (void)setMyValue:(id)foo;
```

In the above statement, *foo* may be of any class.

```
- (void)setMyValue:(id<NSCopying>)foo;
```

In the above statement, *foo* may be an instance of any class that conforms to the `NSCopying` protocol.

```
- (void)setMyValue:(NSNumber *)foo;
```

In the above statement, *foo* must be an instance of the *NSNumber* class.

```
- (void)setMyValue:(NSNumber<NSCopying> *)foo;
```

In the above statement, *foo* must be an instance of the *NSNumber* class, and it must conform to the `NSCopying` protocol.

Forwarding

Objective-C permits the sending of a message to an object that may not respond. Rather than responding or simply dropping the message, an object can forward the message to an object that can respond. Forwarding can be used to simplify implementation of certain design patterns, such as the observer pattern or the proxy pattern.

The Objective-C runtime specifies a pair of methods in `Object`

- forwarding methods:

```
- (retval_t)forward:(SEL)sel args:(arglist_t)args; // with GCC
- (id)forward:(SEL)sel args:(marg_list)args; // with NeXT/Apple systems
```

- action methods:

```
- (retval_t)performv:(SEL)sel args:(arglist_t)args; // with GCC
- (id)performv:(SEL)sel args:(marg_list)args; // with NeXT/Apple systems
```

An object wishing to implement forwarding needs only to override the forwarding method with a new method to define the forwarding behavior. The action method performv:: need not be overridden, as this method merely performs an action based on the selector and arguments. Notice the SEL type, which is the type of messages in Objective-C.

Note: in OpenStep, Cocoa, and GNUstep, the commonly used frameworks of Objective-C, one does not use the Object class. The - (void)forwardInvocation:(NSInvocation *)anInvocation method of the NSObject class is used to do forwarding.

Example

Here is an example of a program that demonstrates the basics of forwarding.

Forwarder.h

```objc
# import <objc/Object.h>

@interface Forwarder : Object {
 id recipient; //The object we want to forward the message to.
}

//Accessor methods.
- (id)recipient;
- (id)setRecipient:(id)_recipient;

@end
```

Forwarder.m

```objc
# import "Forwarder.h"

@implementation Forwarder

- (retval_t)forward:(SEL)sel args:(arglist_t) args {
/*
* Check whether the recipient actually responds to the message.
* This may or may not be desirable, for example, if a recipient
* in turn does not respond to the message, it might do forwarding
* itself.
*/
```

```
if([recipient respondsToSelector:sel]) {
 return [recipient performv:sel args:args];
} else {
 return [self error:"Recipient does not respond"];
}
}

- (id)setRecipient:(id)_recipient {
[recipient autorelease];
recipient = [_recipient retain];
return self;
}

- (id) recipient {
return recipient;
}
@end
```

Recipient.h

```
# import <objc/Object.h>

// A simple Recipient object.
@interface Recipient : Object
- (id)hello;
@end
```

Recipient.m

```
# import "Recipient.h"

@implementation Recipient

- (id)hello {
printf("Recipient says hello!\n");

return self;
}

@end
```

main.m

```
# import "Forwarder.h"
# import "Recipient.h"

int main(void) {
Forwarder *forwarder = [Forwarder new];
Recipient *recipient = [Recipient new];

[forwarder setRecipient:recipient]; //Set the recipient.
/*
* Observe forwarder does not respond to a hello message! It will
* be forwarded. All unrecognized methods will be forwarded to
* the recipient
* (if the recipient responds to them, as written in the Forwarder)
*/
[forwarder hello];

[recipient release];
[forwarder release];
```

```
    return 0;
}
```

Notes

When compiled using gcc, the compiler reports:

```
$ gcc -x objective-c -Wno-import Forwarder.m Recipient.m main.m -lobjc
main.m: In function `main':
main.m:12: warning: `Forwarder' does not respond to `hello'
$
```

The compiler is reporting the point made earlier, that `Forwarder` does not respond to hello messages. In this circumstance, it is safe to ignore the warning since forwarding was implemented. Running the program produces this output:

```
$ ./a.out
Recipient says hello!
```

Categories

During the design of Objective-C, one of the main concerns was the maintainability of large code bases. Experience from the structured programming world had shown that one of the main ways to improve code was to break it down into smaller pieces. Objective-C borrowed and extended the concept of *categories* from Smalltalk implementations to help with this process.[19]

Furthermore, the methods within a category are added to a class at run-time. Thus, categories permit the programmer to add methods to an existing class without the need to recompile that class or even have access to its source code. For example, if a system does not contain a spell checker in its String implementation, it could 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.

If a category declares a method with 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 name (not to be confused with method signature), it is undefined which category's method is adopted.

Other languages have attempted to add this feature in a variety of ways. TOM took the Objective-C system a step further and allowed for the addition of variables as well. Other languages have used prototype oriented solutions instead, with the most notable being Self.

The C# and Visual Basic.NET languages implement superficially similar functionality in the form of extension methods, but these do not have access to the private variables of the class.[20] Ruby and several other dynamic programming languages refer to the technique as "monkey patching".

Logtalk implements a concept of categories (as first-class entities) that subsumes Objective-C categories functionality (Logtalk categories can also be used as fine-grained units of composition when defining e.g. new classes or prototypes; in particular, a Logtalk category can be virtually imported by any number of classes and prototypes).

Example usage of categories

This example builds up an `Integer` class, by defining first a basic class with only accessor methods implemented, and adding two categories, `Arithmetic` and `Display`, which extend the basic class. While categories can access the base class's private data members, it is often good practice to access these private data members through the accessor methods, which helps keep categories more independent from the base class. Implementing such accessors is one typical usage of categories. Another is to use categories to add methods to the base class. However, it is not regarded as good practice to use categories for subclass overriding, also known as monkey patching. Informal protocols are implemented as a category on the base `NSObject` class. By convention, files containing categories that extend base classes will take the name BaseClass+ExtensionClass.h.

Integer.h

```
# import <objc/Object.h>

@interface Integer : Object {
 int integer;
}

- (int) integer;
- (id) integer: (int) _integer;
@end
```

Integer.m

```
# import "Integer.h"

@implementation Integer
- (int) integer {
 return integer;
}

- (id) integer: (int) _integer {
 integer = _integer;

 return self;
}
@end
```

Integer+Arithmetic.h

```
# import "Integer.h"

@interface Integer (Arithmetic)
- (id) add: (Integer *) addend;
- (id) sub: (Integer *) subtrahend;
@end
```

Integer+Arithmetic.m

```objc
# import "Integer+Arithmetic.h"

@implementation Integer (Arithmetic)
- (id) add: (Integer *) addend {
  return [self integer: [self integer] + [addend integer]];
}

- (id) sub: (Integer *) subtrahend {
  return [self integer: [self integer] - [subtrahend integer]];
}
@end
```

Integer+Display.h

```objc
# import "Integer.h"

@interface Integer (Display)
- (id) showstars;
- (id) showint;
@end
```

Integer+Display.m

```objc
# import "Integer+Display.h"

@implementation Integer (Display)
- (id) showstars {
  int i, x = [self integer];
  for (i = 0; i < x; i++) {
    printf("*");
  }
  printf("\n");

  return self;
}

- (id) showint {
  printf("%d\n", [self integer]);

  return self;
}
@end
```

main.m

```objc
# import "Integer.h"
# import "Integer+Arithmetic.h"
# import "Integer+Display.h"

int main(void) {
  Integer *num1 = [Integer new], *num2 = [Integer new];
  int x;

  printf("Enter an integer: ");
  scanf("%d", &x);
```

```
    [num1 integer:x];
    [num1 showstars];

    printf("Enter an integer: ");
    scanf("%d", &x);

    [num2 integer:x];
    [num2 showstars];

    [num1 add:num2];
    [num1 showint];

    return 0;
}
```

Notes

Compilation is performed, for example, by:

```
gcc -x objective-c main.m Integer.m Integer+Arithmetic.m Integer+Display.m -lobjc
```

One can experiment by leaving out the `#import "Integer+Arithmetic.h"` and `[num1 add:num2]` lines and omitting `Integer+Arithmetic.m` in compilation. The program will still run. This means that it is possible to "mix-and-match" added categories if necessary; if one does not need to have some capability provided in a category, one can simply not compile it in.

Posing

Objective-C permits a class to wholly replace another class within a program. The replacing class is said to "pose as" the target class.

Note: Class posing was declared deprecated with Mac OS X v10.5, and is unavailable in the 64-bit runtime. Similar functionality can be achieved by using method swizzling in categories, that swaps one method's implementation with another's that have a same signature.

For the versions still supporting posing, all messages sent to the target class are instead received by the posing class. There are several restrictions:

- A class may only pose as one of its direct or indirect superclasses.
- The posing class must not define any new instance variables that are absent from the target class (though it may define or override methods).
- The target class may not have received any messages prior to the posing.

Posing, similarly with categories, allows global augmentation of existing classes. Posing permits two features absent from categories:

- A posing class can call overridden methods through super, thus incorporating the implementation of the target class.
- A posing class can override methods defined in categories.

For example,

```
@interface CustomNSApplication : NSApplication
@end

@implementation CustomNSApplication
- (void) setMainMenu: (NSMenu*) menu {
 // do something with menu
}
@end

class_poseAs ([CustomNSApplication class], [NSApplication class]);
```

This intercepts every invocation of setMainMenu to NSApplication.

#import

In the C language, the `#include` pre-compile directive always causes a file's contents to be inserted into the source at that point. Objective-C has the equivalent `#import` directive except each file is included only once per compilation unit, obviating the need for include guards.

Other features

Objective-C's features often allow for flexible, and often easy, solutions to programming issues.

- Delegating methods to other objects and remote invocation can be easily implemented using categories and message forwarding.
- Swizzling of the *isa* pointer allows for classes to change at runtime. Typically used for debugging where freed objects are swizzled into zombie objects whose only purpose is to report an error when someone calls them. Swizzling was also used in Enterprise Objects Framework to create database faults. Swizzling is used today by Apple's Foundation Framework to implement Key-Value Observing.

Language variants

Objective-C++

Objective-C++ is a language variant accepted by the front-end to the GNU Compiler Collection and Clang, which can compile source files that use a combination of C++ and Objective-C syntax. Objective-C++ adds to C++ the extensions that Objective-C adds to C. As nothing is done to unify the semantics behind the various language features, certain restrictions apply:

- A C++ class cannot derive from an Objective-C class and vice versa.
- C++ namespaces cannot be declared inside an Objective-C declaration.
- Objective-C declarations may appear only in global scope, not inside a C++ namespace
- Objective-C classes cannot have instance variables of C++ classes that do not have a default constructor or that have one or more virtual methods, but pointers to C++ objects can be used as instance variables without restriction (allocate them with new in the -init method).

- C++ "by value" semantics cannot be applied to Objective-C objects, which are only accessible through pointers.
- An Objective-C declaration cannot be within a C++ template declaration and vice versa. However, Objective-C types, (e.g., Classname *) can be used as C++ template parameters.
- Objective-C and C++ exception handling is distinct; the handlers of each cannot handle exceptions of the other type. This is mitigated in recent runtimes as Objective-C exceptions are either replaced by C++ exceptions completely (Apple runtime), or partly when Objective-C++ library is linked (GNUstep libobjc2).
- Care must be taken since the destructor calling conventions of Objective-C and C++'s exception run-time models do not match (i.e., a C++ destructor will not be called when an Objective-C exception exits the C++ object's scope). The new 64-bit runtime resolves this by introducing interoperability with C++ exceptions in this sense.[21]
- Objective-C blocks and C++11 lambdas are distinct entities, however a block is transparently generated on Mac OS X when passing a lambda where a block is expected.[22]

Objective-C 2.0

At the 2006 Worldwide Developers Conference, Apple announced the release of "Objective-C 2.0," a revision of the Objective-C language to include "modern garbage collection, syntax enhancements,[23] runtime performance improvements,[24] and 64-bit support". Mac OS X v10.5, released in October 2007, included an Objective-C 2.0 compiler. GCC 4.6 supports many new Objective-C features, such as declared and synthesized properties, dot syntax, fast enumeration, optional protocol methods, method/protocol/class attributes, class extensions and a new GNUnn Objective-C runtime API.[25]

Garbage collection

Objective-C 2.0 provided an optional conservative, generational garbage collector. When run in backwards-compatible mode, the runtime turned reference counting operations such as "retain" and "release" into no-ops. All objects were subject to garbage collection when garbage collection was enabled. Regular C pointers could be qualified with "__strong" to also trigger the underlying write-barrier compiler intercepts and thus participate in garbage collection.[26] A zero-ing weak subsystem was also provided such that pointers marked as "__weak" are set to zero when the object (or more simply, GC memory) is collected. The garbage collector does not exist on the iOS implementation of Objective-C 2.0.[27] Garbage collection in Objective-C runs on a low-priority background thread, and can halt on user events, with the intention of keeping the user experience responsive.[28]

Garbage collection was deprecated in OS X v10.8 in favor of Automatic Reference Counting (ARC).[29] Objective-C on iOS 7 running on ARM64 uses 19 bits out of a 64-bit word to store the reference count, as a form of tagged pointers.[30][31]

Properties

Objective-C 2.0 introduces a new syntax to declare instance variables as properties, with optional attributes to configure the generation of accessor methods. Properties are, in a sense, public instance variables; that is, declaring an instance variable as a property provides external classes with access (possibly limited, e.g. read only) to that property. A property may be declared as "readonly", and may be provided with storage semantics such as "assign", "copy" or "retain". By default, properties are considered atomic, which results in a lock preventing multiple threads from accessing them at the same time. A property can be declared as "nonatomic", which removes this lock.

```objc
@interface Person : NSObject {
@public
NSString *name;
@private
int age;
}

@property(copy) NSString *name;
@property(readonly) int age;

-(id)initWithAge:(int)age;
@end
```

Properties are implemented by way of the @synthesize keyword, which generates getter (and setter, if not read-only) methods according to the property declaration. Alternatively, the getter and setter methods must be implemented explicitly, or the @dynamic keyword can be used to indicate that accessor methods will be provided by other means. When compiled using clang 3.1 or higher, all properties which are not explicitly declared with `@dynamic`, marked `readonly` or have complete user-implemented getter and setter will be automatically implicitly `@synthesize`'d.

```objc
@implementation Person
@synthesize name;

-(id)initWithAge:(int)initAge {
self = [super init];
if (self) {
age = initAge; // NOTE: direct instance variable assignment, not property setter
}
return self;
}

-(int)age {
return age;
}
@end
```

Properties can be accessed using the traditional message passing syntax, dot notation, or, in Key-Value Coding, by name via the "valueForKey:"/"setValue:forKey:" methods.

```objc
Person *aPerson = [[Person alloc] initWithAge: 53];
aPerson.name = @"Steve"; // NOTE: dot notation, uses synthesized setter,
 // equivalent to [aPerson setName: @"Steve"];
NSLog(@"Access by message (%@), dot notation(%@),
property name(%@) and direct instance variable access (%@)",
 [aPerson name], aPerson.name, [aPerson valueForKey:@"name"], aPerson->name);
```

In order to use dot notation to invoke property accessors within an instance method, the "self" keyword should be used:

```objc
-(void) introduceMyselfWithProperties:(BOOL)useGetter {
  NSLog(@"Hi, my name is %@.", (useGetter ? self.name : name));
  // NOTE: getter vs. ivar access
}
```

A class or protocol's properties may be dynamically introspected.

```objc
int i;
int propertyCount = 0;
objc_property_t *propertyList = class_copyPropertyList([aPerson class], &propertyCount);

for (i = 0; i < propertyCount; i++) {
  objc_property_t *thisProperty = propertyList + i;
  const char* propertyName = property_getName(*thisProperty);
  NSLog(@"Person has a property: '%s'", propertyName);
}
```

Non-fragile instance variables

Objective-C 2.0 provides non-fragile instance variables where supported by the runtime (i.e. when building 64-bit Mac OS X code as well as all iOS code). Under the modern runtime, an extra layer of indirection is added to instance variable access, allowing the dynamic linker to adjust instance layout at runtime. This feature allows for two important improvements to Objective-C code:

- This eliminates the fragile binary interface problem—superclasses can change sizes without affecting binary compatibility.
- This allows instance variables that provide the backing for properties to be synthesized at runtime without them being declared in the class's interface.

Fast enumeration

Instead of using an NSEnumerator object or indices to iterate through a collection, Objective-C 2.0 offers the fast enumeration syntax. In Objective-C 2.0, the following loops are functionally equivalent, but have different performance characteristics.

```objc
// Using NSEnumerator
NSEnumerator *enumerator = [thePeople objectEnumerator];
Person *p;

while ((p = [enumerator nextObject]) != nil) {
  NSLog(@"%@ is %i years old.", [p name], [p age]);
}
```

```objc
// Using indexes
for (int i = 0; i < [thePeople count]; i++) {
```

```
  Person *p = [thePeople objectAtIndex:i];
  NSLog(@"%@ is %i years old.", [p name], [p age]);
}


// Using fast enumeration
for (Person *p in thePeople) {
  NSLog(@"%@ is %i years old.", [p name], [p age]);
}
```

Fast enumeration generates more efficient code than standard enumeration because method calls to enumerate over objects are replaced by pointer arithmetic using the NSFastEnumeration protocol.[32]

Class Extensions

A class extension has the same syntax as a category declaration with no category name, and the methods and properties declared in it are added directly to the main class. It is mostly used as an alternative to a category to add methods to a class without advertising them in the public headers, with the advantage that for class extensions the compiler checks that all the privately declared methods are actually implemented.[33]

Implications for Cocoa development

All Objective-C applications developed for Mac OS X that make use of the above improvements for Objective-C 2.0 are incompatible with all operating systems prior to 10.5 (Leopard). Since fast enumeration does not generate exactly the same binaries as standard enumeration, its use will cause an application to crash on OS X version 10.4 or earlier.

Blocks

Blocks is a nonstandard extension for Objective-C (as well as C and C++) that uses special syntax to create closures. Blocks are only supported in Mac OS X 10.6 "Snow Leopard" or later and iOS 4 or later, as well as GNUstep with libobjc2 1.7 and compiling with clang 3.1 or later.[34]

```
#include <stdio.h>
#include <Block.h>
typedef int (^IntBlock)();

IntBlock MakeCounter(int start, int increment) {
    __block int i = start;

    return Block_copy( ^ {
        int ret = i;
        i += increment;
        return ret;
    });

}

int main(void) {
    IntBlock mycounter = MakeCounter(5, 2);
    printf("First call: %d\n", mycounter());
    printf("Second call: %d\n", mycounter());
    printf("Third call: %d\n", mycounter());

    /* because it was copied, it must also be released */
```

```
    Block_release(mycounter);

    return 0;
}
/* Output:
    First call: 5
    Second call: 7
    Third call: 9
*/
```

Modern Objective-C

Automatic Reference Counting

Automatic Reference Counting (ARC) is a compile-time feature that eliminates the need for programmers to manually manage retain counts using `retain` and `release`.[35] Unlike garbage collection, which occurs at run time, ARC eliminates the overhead of a separate process managing retain counts. ARC and manual memory management are not mutually exclusive; programmers can continue to use non-ARC code in ARC-enabled projects by disabling ARC for individual code files. XCode can also attempt to automatically upgrade a project to ARC.

Literals

NeXT and Apple Obj-C runtimes have long included a short-form way to create new strings, using the literal syntax `@"a new string"`, or drop to CoreFoundation constants `kCFBooleanTrue` and `kCFBooleanFalse` for `NSNumber` with Boolean values. Using this format saves the programmer from having to use the longer `initWithString` or similar methods when doing certain operations.

When using Apple LLVM compiler 4.0 or later, arrays, dictionaries, and numbers (`NSArray`, `NSDictionary`, `NSNumber` classes) can also be created using literal syntax instead of methods.[36] Literal syntax uses the `@` symbol combined with `[]`, `{}`, `()`, to create the classes mentioned above, respectively.[37]

Example without literals:

```
NSArray *myArray = [NSArray arrayWithObjects:object1,object2,object3,nil];
NSDictionary *myDictionary1 = [NSDictionary dictionaryWithObject:someObject forKey:@"key"];
NSDictionary *myDictionary2 = [NSDictionary dictionaryWithObjectsAndKeys:object1, key1, object2, key2, nil
NSNumber *myNumber = [NSNumber numberWithInt:myInt];
NSNumber *mySumNumber = [NSNumber numberWithInt:(2 + 3)];
NSNumber *myBoolNumber = [NSNumber numberWithBool:YES];
```

Example with literals:

```
NSArray *myArray = @[ object1, object2, object3 ];
NSDictionary *myDictionary1 = @{ @"key" : someObject };
NSDictionary *myDictionary2 = @{ key1: object1, key2: object2 };
NSNumber *myNumber = @(myInt);
NSNumber *mySumNumber = @(2+3);
NSNumber *myBoolNumber = @YES;
```

However, different from string literals, which compile to actual constants in the executable, these literals compile to code equivalent to the above method calls. In particular, under manually reference-counted memory management, these objects are autoreleased, which requires additional care when e.g. used with function-static variables or other kinds of globals.

Subscripting

When using Apple LLVM compiler 4.0 or later, arrays and dictionaries (`NSArray` and `NSDictionary` classes) can be manipulated using subscripting.[36] Subscripting can be used to retrieve values from indexes (array) or keys (dictionary), and with mutable objects, can also be used to set objects to indexes or keys. In code, subscripting is represented using brackets `[ ]`.[37]

Example without subscripting:

```
id object1 = [someArray objectAtIndex:0];
id object2 = [someDictionary objectForKey:@"key"];
[someMutableArray replaceObjectAtIndex:0 withObject:object3];
[someMutableDictionary setObject:object4 forKey:@"key"];
```

Example with subscripting:

```
id object1 = someArray[0];
id object2 = someDictionary[@"key"];
someMutableArray[0] = object3;
someMutableDictionary[@"key"] = object4;
```

"Modern" Objective-C syntax (1997)

After the purchase of NeXT by Apple, attempts were made to make the language more acceptable to programmers more familiar with Java than Smalltalk. One of these attempts was the introduction of what was dubbed "Modern Syntax" for Objective-C at the time[38] (as opposed to the current, "classic" syntax). There was no change in actual behaviour, this was merely an alternative syntax. Instead of writing a method invocation like

```
object = [[MyClass alloc] init];
[object firstLabel: param1 secondLabel: param2];
```

It was instead written as

```
object = (MyClass.alloc).init;
object.firstLabel ( param1, param2 );
```

Similarly, declarations went from the form

```
-(void) firstLabel: (int)param1 secondLabel: (int)param2;
```

to

```
-(void) firstLabel ( int param1, int param2 );
```

This "modern" syntax is no longer supported in current dialects of the Objective-C language.

Portable Object Compiler

Besides the GCC/NeXT/Apple implementation, which added several extensions to the original Stepstone implementation, another free, open-source Objective-C implementation called the Portable Object Compiler[39] also exists. The set of extensions implemented by the Portable Object Compiler differs from the GCC/NeXT/Apple implementation; in particular, it includes Smalltalk-like blocks for Objective-C, while it lacks protocols and categories, two features used extensively in OpenStep and its derivatives and relatives. Overall, POC represents an older, pre-NeXT stage in the language's evolution, roughly conformant to Brad Cox's 1991 book.

It also includes a runtime library called ObjectPak, which is based on Cox's original ICPak101 library (which in turn derives from the Smalltalk-80 class library), and is quite radically different from the OpenStep FoundationKit.

GEOS Objective-C

The PC GEOS system used a programming language known as **GEOS Objective-C** or **goc**;[40] despite the name similarity, the two languages are similar only in overall concept and the use of keywords prefixed with an @ sign.

Clang

The Clang compiler suite, part of the LLVM project, implements Objective-C as well as other languages.

Library use

Objective-C today is often used in tandem with a fixed library of standard objects (often known as a "kit" or "framework"), such as Cocoa, GNUstep or ObjFW. These libraries often come with the operating system: the GNUstep libraries often come with GNU/Linux based distributions and Cocoa comes with OS X. The programmer is not forced to inherit functionality from the existing base class (NSObject / OFObject). Objective-C allows for the declaration of new root classes that do not inherit any existing functionality. Originally, Objective-C based programming environments typically offered an Object class as the base class from which almost all other classes inherited. With the introduction of OpenStep, NeXT created a new base class named NSObject, which offered additional features over Object (an emphasis on using object references and reference counting instead of raw pointers, for example). Almost all classes in Cocoa inherit from NSObject.

Not only did the renaming serve to differentiate the new default behavior of classes within the OpenStep API, but it allowed code that used Object—the original base class used on NeXTSTEP (and, more or less, other Objective-C class libraries)—to co-exist in the same runtime with code that used NSObject

(with some limitations). The introduction of the two letter prefix also became a simplistic form of namespaces, which Objective-C lacks. Using a prefix to create an informal packaging identifier became an informal coding standard in the Objective-C community, and continues to this day.

More recently, package managers have started appearing, such as CocoaPods, which aims to be both a package manager and a repository of packages. A lot of open-source Objective-C code that was written in the last few years can now be installed using CocoaPods.

Analysis of the language

Objective-C implementations use a thin runtime system written in C, which adds little to the size of the application. In contrast, most object-oriented systems at the time that it was created used large virtual machine runtimes. Programs written in Objective-C tend to be not much larger than the size of their code and that of the libraries (which generally do not need to be included in the software distribution), in contrast to Smalltalk systems where a large amount of memory was used just to open a window. Objective-C applications tend to be larger than similar C or C++ applications because Objective-C dynamic typing does not allow methods to be stripped or inlined. Since the programmer has such freedom to delegate, forward calls, build selectors on the fly and pass them to the runtime system, the Objective-C compiler cannot assume it is safe to remove unused methods or to inline calls.

Likewise, the language can be implemented on top of existing C compilers (in GCC, first as a preprocessor, then as a module) rather than as a new compiler. This allows Objective-C to leverage the huge existing collection of C code, libraries, tools, etc. Existing C libraries can be wrapped in Objective-C wrappers to provide an OO-style interface. In this aspect, it is similar to GObject library and Vala language, which are widely used in development of GTK applications.

All of these practical changes lowered the barrier to entry, likely the biggest problem for the widespread acceptance of Smalltalk in the 1980s.

The first versions of Objective-C did not support garbage collection. At the time this decision was a matter of some debate, and many people considered long "dead times" (when Smalltalk performed collection) to render the entire system unusable. Some 3rd party implementations have added this feature (most notably GNUstep) and Apple has implemented it as of Mac OS X v10.5.[41] However in more recent versions of Mac OS X and iOS, garbage collection has been deprecated in favor of ARC (see below).

Another common criticism is that Objective-C does not have language support for namespaces. Instead, programmers are forced to add prefixes to their class names, which are traditionally shorter than namespace names and thus more prone to collisions. As of 2007, all Mac OS X classes and functions in the Cocoa programming environment are prefixed with "NS" (e.g. NSObject, NSButton) to identify them as belonging to the Mac OS X or iOS core; the "NS" derives from the names of the classes as defined during the development of NeXTstep.

Since Objective-C is a strict superset of C, it does not treat C primitive types as first-class objects.

Unlike C++, Objective-C does not support operator overloading. Also unlike C++, Objective-C allows an object to directly inherit only from one class (forbidding multiple inheritance). However, in most cases, categories and protocols may be used as alternative ways to achieve the same results.

Because Objective-C uses dynamic runtime typing and because all method calls are function calls (or, in some cases, syscalls), many common performance optimizations cannot be applied to Objective-C methods (for example: inlining, constant propagation, interprocedural optimizations, and scalar replacement of aggregates). This limits the performance of Objective-C abstractions relative to similar abstractions in languages such as C++ where such optimizations are possible.

Apple introduced Automatic Reference Counting (ARC) in 2011 as an alternative to garbage collection. With ARC, the compiler inserts retain and release calls automatically into Objective-C code based on static code analysis. The automation relieves the programmer of having to write in memory management code. ARC also adds weak references to the Objective-C language.[42]

Philosophical differences between Objective-C and C++

The design and implementation of C++ and Objective-C represent fundamentally different approaches to extending C.

In addition to C's style of procedural programming, C++ directly supports certain forms of object-oriented programming, generic programming, and metaprogramming. C++ also comes with a large standard library that includes several container classes. Similarly, Objective-C adds object-oriented programming, dynamic typing, and reflection to C. Objective-C does not provide a standard library *per se*, but in most places where Objective-C is used, it is used with an OpenStep-like library such as OPENSTEP, Cocoa, or GNUstep, which provides functionality similar to C++'s standard library.

One notable difference is that Objective-C provides runtime support for reflective features, whereas C++ adds only a small amount of runtime support to C. In Objective-C, an object can be queried about its own properties, e.g., whether it will respond to a certain message. In C++, this is not possible without the use of external libraries.

The use of reflection is part of the wider distinction between dynamic (run-time) features and static (compile-time) features of a language. Although Objective-C and C++ each employ a mix of both features, Objective-C is decidedly geared toward run-time decisions while C++ is geared toward compile-time decisions. The tension between dynamic and static programming involves many of the classic trade-offs in programming: dynamic features add flexibility, static features add speed and type checking.

Generic programming and metaprogramming can be implemented in both languages using runtime polymorphism; in C++ this takes the form of virtual functions and runtime type identification, while Objective-C offers dynamic typing and reflection. Objective-C lacks compile-time polymorphism (generics) entirely, while C++ supports it via function overloading and templates.

See also

- C (programming language)
- C++
- Comparison of programming languages
- Swift
- Xcode

References

1. ^ Lattner, Chris (2014-06-03). "Chris Lattner's Homepage" (http://nondot.org/sabre/). Chris Lattner. Retrieved 2014-06-03. "The Swift language is the product of tireless effort from a team of language experts, documentation gurus, compiler optimization ninjas, and an incredibly important internal dogfooding group who provided feedback to help refine and battle-test ideas. Of course, it also greatly benefited from the experiences hard-won by many other languages in the field, drawing ideas from Objective-C, Rust, Haskell, Ruby, Python, C#, CLU, and far too many others to list."

2. ^ Singh, Amit (December 2003). "A Brief History of Mac OS X" (http://osxbook.com/book/bonus/ancient/whatismacosx/history.html). Mac OS X Internals. Retrieved 11 June 2012.

3. ^ Garling, Caleb. "iPhone Coding Language Now World's Third Most Popular" (http://www.wired.com/wiredenterprise/2012/07/apple-objective-c/). Wired. Retrieved May 20, 2013.

4. ^ Wentk, Richard (2009). *Cocoa: Volume 5 of Developer Reference Apple Developer Series* (http://books.google.com/books?id=vBglDfsAjVQC&pg=PT23&dq=tom.love+objective.c+cocoa&hl=en&ei=TrFbTemfAo-0vgPiiK3UDA&sa=X&oi=book_result&ct=book-preview-link&resnum=1&ved=0CCkQuwUwAA#v=onepage&q=tom.love%20objective.c%20cocoa&f=false). John Wiley and Sons,. ISBN 0-470-49589-8.

5. ^ Biancuzzi, Federico; Warden, Shane (2009). *Masterminds of Programming* (http://books.google.com/books?id=yB1WwURwBUQC&pg=PA242&dq=Engineering.objective.C+tom.love&hl=en&ei=T-NbTem3Fof2vwPRvenzDA&sa=X&oi=book_result&ct=result&resnum=1&ved=0CCoQ6AEwAA#v=onepage&q=Engineering.objective.C%20tom.love&f=false). O'Reilly Media, Inc. pp. 242–246. ISBN 0-596-51517-0.

6. ^ Cox, Brad (1983). "The object oriented pre-compiler: programming Smalltalk 80 methods in C language" (http://portal.acm.org/citation.cfm?id=948095). *ACM SIGPLAN Notices* (New York, NY: ACM) **18** (1). doi:10.1145/948093.948095 (http://dx.doi.org/10.1145%2F948093.948095). Retrieved 2011-02-17.

7. ^ "Common Lisp and Readline" (http://clisp.cvs.sourceforge.net/viewvc/clisp/clisp/doc/Why-CLISP-is-under-GPL). "The issue first arose when NeXT proposed to distribute a modified GCC in two parts and let the user link them. Jobs asked me whether this was lawful. It seemed to me at the time that it was, following reasoning like what you are using; but since the result was very undesirable for free software, I said I would have to ask the lawyer. What the lawyer said surprised me; he said that judges would consider such schemes to be "subterfuges" and would be very harsh toward them. He said a judge would ask whether it is "really" one program, rather than how it is labeled. So I went back to Jobs and said we believed his plan was not allowed by the GPL. The direct result of this is that we now have an Objective C front end. They had wanted to distribute the Objective C parser as a separate proprietary package to link with the GCC back end, but since I didn't agree this was allowed, they made it free."

8. ^ "GNUstep: Introduction" (http://www.gnustep.org/information/aboutGNUstep.html). GNUstep developers/GNU Project. Retrieved 2012-07-29.

9. ^ "Write Objective-C Code" (https://developer.apple.com/library/mac/referencelibrary/GettingStarted/RoadMapOSX/books/WriteObjective-CCode/WriteObjective-CCode/WriteObjective-CCode.html). apple.com. 2013-04-23. Retrieved 2013-12-22.

10. ^ "Objective-C Boot Camp" (http://www.informit.com/articles/article.aspx?p=1765122). "Objective-C is a strict superset of ANSI C"

11. ^ "Examining Objective-C" (http://www.drdobbs.com/examining-objective-c/184402055). "Objective-C is

11. Examining Objective-C" (http://www.drdobbs.com/examining-objective-c/184402033). "Objective-C is an object-oriented strict superset of C"

12. ^ "Pro Objective-C By Keith Lee" (http://books.google.com/books?id=ec6zAAAAQBAJ&lpg=PA381&ots=NoFaE3OEX0&pg=PA381#v=onepage&q&f=false).

13. ^ "Tags for Objective-C Headers" (http://web.mit.edu/darwin/src/modules/headerdoc/Documentation/ObjCTags.html). "Objective-C is a superset of C"

14. ^ "AppScan Source 8.7 now available" (http://www-01.ibm.com/support/docview.wss?uid=swg24034606). "The Objective-C programming language is a superset of the C programming language"

15. ^ Apple, Inc. (19 October 2009). "Dynamic Method Resolution" (http://developer.apple.com/mac/library/documentation/Cocoa/Conceptual/ObjCRuntimeGuide/Articles/ocrtDynamicResolution.html). *Objective-C Runtime Programming Guide*. Retrieved 25 November 2014.

16. ^ Apple, Inc. (19 October 2009). "Avoiding Messaging Errors" (https://web.archive.org/web/20100908132046/http://developer.apple.com/mac/library/documentation/Cocoa/Conceptual/ObjectiveC/Articles/ocSelectors.html). *The Objective-C Programming Language*. Archived from the original (http://developer.apple.com/mac/library/documentation/cocoa/conceptual/ObjectiveC/Articles/ocSelectors.html#//apple_ref/doc/uid/TP30001163-CH23-89447) on 8 September 2010.

17. ^ Dalrymple, Mark; Knaster, Scott. *Learn Objective-C on the Mac*. p. 9. "The .m extension originally stood for "messages" when Objective-C was first introduced, referring to a central feature of Objective-C"

18. ^ "Objective-C Runtime Programming Guide" (https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/ObjCRuntimeGuide/Articles/ocrtForwarding.html#//apple_ref/doc/uid/TP40008048-CH105-SW1). Apple Inc.

19. ^ Example of *categories* concept (http://video.google.com/videoplay?docid=-7466310348707586940&ei=0dr7SIe6L46qrgLk7dHsDg&q=Smalltalk-80)

20. ^ "Extension Methods (C# Programming Guide)" (http://msdn.microsoft.com/en-us/library/bb383977.aspx). Microsoft. October 2010. Retrieved 2011-07-10.

21. ^ Using C++ With Objective-C (http://developer.apple.com/mac/library/documentation/Cocoa/Conceptual/ObjectiveC/Articles/ocCPlusPlus.html#//apple_ref/doc/uid/TP30001163-CH10-SW1) in Mac OS X Reference Library, last retrieved in 2010-02-10.

22. ^ "Clang Language Extensions — Clang 3.5 documentation" (http://clang.llvm.org/docs/LanguageExtensions.html#interoperability-with-c-11-lambdas). Clang.llvm.org. Retrieved 2014-04-16.

23. ^ "Objective-C 2.0: more clues" (http://lists.apple.com/archives/Objc-language/2006/Aug/msg00039.html). Lists.apple.com. 2006-08-10. Retrieved 2010-05-30.

24. ^ "Re: Objective-C 2.0" (http://lists.apple.com/archives/Objc-language/2006/Aug/msg00018.html). Lists.apple.com. Retrieved 2010-05-30.

25. ^ "GCC 4.6 Release Series: Changes, New Features, and Fixes" (http://gcc.gnu.org/gcc-4.6/changes.html).

26. ^ Garbage Collection Programming Guide: Garbage Collection API (http://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/GarbageCollection/Articles/gcAPI.html) (Apple developer website - search for "__strong")

27. ^ "Garbage Collection Programming Guide: Introduction to Garbage Collection" (https://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/GarbageCollection/Introduction.html). Apple Inc. 2011-10-03.

28. **^** "Leopard Technology Series for Developers: Objective-C 2.0 Overview"
    (http://web.archive.org/web/20100724195423/http://developer.apple.com/leopard/overview/objectivec2.html
). Apple Inc. 2007-11-06. Archived from the original
    (http://developer.apple.com/leopard/overview/objectivec2.html) on July 24, 2010. Retrieved 2010-05-30.
29. **^** "Transitioning to ARC Release Notes"
    (http://developer.apple.com/library/mac/#releasenotes/ObjectiveC/RN-
    TransitioningToARC/Introduction/Introduction.html). Apple Inc. 2012-07-17. Retrieved 2012-08-26.
30. **^** Mike Ash. "Friday Q&A 2013-09-27: ARM64 and You" (https://www.mikeash.com/pyblog/friday-qa-
    2013-09-27-arm64-and-you.html). mikeash.com. Retrieved 2014-04-27.
31. **^** "Hamster Emporium: [objc explain]: Non-pointer isa"
    (http://www.sealiesoftware.com/blog/archive/2013/09/24/objc_explain_Non-pointer_isa.html).
    Sealiesoftware.com. 2013-09-24. Retrieved 2014-04-27.
32. **^** Apple, Inc. (2009). "Fast Enumeration"
    (http://developer.apple.com/mac/library/documentation/Cocoa/Conceptual/ObjectiveC/Articles/ocFastEnume
    ration.html). apple.com. Retrieved 2009-12-31.
33. **^** Free Software Foundation, Inc. (2011). "GCC 4.6 Release Series – Changes, New Features, and Fixes"
    (http://gcc.gnu.org/gcc-4.6/changes.html). http://gcc.gnu.org. Retrieved 2013-11-27.
34. **^** "Blocks Programming Topics – Mac Developer Library"
    (http://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/Blocks/Articles/00_Introduction.
    html). Apple Inc. March 8, 2011. Retrieved November 28, 2012.
35. **^** "Transitioning to ARC" (https://developer.apple.com/library/ios/#releasenotes/ObjectiveC/RN-
    TransitioningToARC/Introduction/Introduction.html). Apple Inc. Retrieved 8 October 2012.
36. **^** *a* *b* "Programming with Objective-C: Values and Collections"
    (https://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Fo
    undationTypesandCollections/FoundationTypesandCollections.html). Apple Inc. Retrieved 8 October 2012.
37. **^** *a* *b* "Objective-C Literals — Clang 3.5 documentation"
    (http://clang.llvm.org/docs/ObjectiveCLiterals.html). Clang.llvm.org. Retrieved 2014-04-16.
38. **^** *Rhapsody Developer's Guide*, AP Professional, 1997
39. **^** "Portable Object Compiler" (http://users.pandora.be/stes/compiler.html). Users.pandora.be. 1970-01-01.
    Retrieved 2010-05-30.
40. **^** "Breadbox Computer Company LLC homepage" (http://www.breadbox.com/downloads.asp?
    id=54&category=GeosSDK&maincategory=SDK). Retrieved 8 December 2010.
41. **^** Apple, Inc. (August 22, 2006). "Mac OS X Leopard – Xcode 3.0"
    (http://www.apple.com/macosx/developertools/xcode.html). apple.com. Retrieved 2006-08-22.
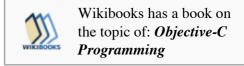42. **^** "Transitioning to ARC Release Notes"
    (http://developer.apple.com/library/ios/#releasenotes/ObjectiveC/RN-
    TransitioningToARC/Introduction/Introduction.html). *iOS Developer Library*. Developer.apple.com.
    Retrieved 2014-04-16.

Further reading

- Cox, Brad J. (1991). *Object Oriented Programming: An Evolutionary Approach*. Addison Wesley. ISBN 0-
  201-54834-8.

External links

- Programming with Objective-C

  (http://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/), from Apple (2012-12-13)
- *The Objective-C 2.0 Programming Language*

  (https://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/ObjectiveC/Introduction/introObjectiveC.html), from Apple (2011-10-11)
- *Objective-C Runtime Programming Guide*

  (https://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/ObjCRuntimeGuide/Introduction/Introduction.html), from Apple (2009-10-19)
- Objective-C GNUstep Base Programming Manual

  (http://www.gnu.org/software/gnustep/resources/documentation/Developer/Base/ProgrammingManual/manual_toc.html)
- Objective-C by Brad Cox (http://virtualschool.edu/objectivec/)
- Objective-C FAQ (http://www.faqs.org/faqs/computer-lang/Objective-C/faq/)

Categories: C programming language family | Object-oriented programming languages | Dynamically typed programming languages | Class-based programming languages | NeXT | GNUstep | Objective-C | Programming languages created in 1986