From C++ to Objective-C

version 2.1 en

Pierre Chatelier
e-mail: pierre.chatelier@club-internet.fr

English adaptation : Aaron Vegh

Document revisions available at :
`http://pierre.chachatelier.fr/programmation/objective-c.php`

This document is also available in french
Ce document est aussi disponible en français

With special thanks to: For their attentive reading and many helpful comments, I would like to thank Pascal Bleuyard, Jérôme Cornet, François Delobel and Jean-Daniel Dupas, whose help was important in making this work the best possible. Jack Nutting, Ben Rimmington and Mattias Arrelid have also provided many feedback. Jonathon Mah has been particularly implied in bringing a lot of very judicious corrections.
They are not responsible of any mistake I could add after their reviewing.

Contents

Introduction

This document is designed to act as a bridge between C++ and Objective-C. Many texts exist to teach the object model through Objective-C, but to my knowledge, none target advanced C++ developers wanting to compare these concepts with their own knowledge. At first, the Objective-C language seems to be an obstacle rather than a boost for **Cocoa** programming (cf. section 1 on the following page): it was so different that I couldn't get into it. It took me some time to appreciate its challenges, and understand the many helpful concepts it supplies. This document is not a tutorial, but a quick reference, to these concepts. I hope that it will be useful in preventing a developer from either abandoning Objective-C or misusing its features, because of a misunderstanding of the language. This document does not claim to be a *full* reference, but a *quick* one. For detailed explanations of a concept, be sure to read a specialized Objective-C manual [4].

Making comparisons with C# would require another document, because that language is much closer to Objective-C than C++ can be. Thus, a C# developer would certainly learn Objective-C faster. According to me, C# is, despite a bunch of advanced concepts, far less interesting than Objective-C, because it gives hard access to simple Objective-C features, and the `Cocoa` API quality is miles ahead of `.NET`. This personal opinion is not the subject of the present document.

1 Objective-C and Cocoa

An initial distinction should be made : **Objective-C** is a language, while **Cocoa** is a set of classes that contribute to native MacOS X programming. Theoretically, it is possible to use Objective-C without Cocoa: there is a gcc front-end. But under MacOS X, both are almost inseparable, as most of the classes supplied by the language are part of Cocoa.

More precisely, Cocoa is the implementation by Apple, for MacOS X, of the OpenStep standard, originally published in 1994. It consists of a developer framework based upon Objective-C. The GNUstep project [6] is another implementation, which is free. Its goal is to be as portable as possible on most Unix systems, and is still under development.

1.1 A short history of Objective-C

It is hard to give a precise date of birth for a language, owing to the fact that there is some time between first steps, improvements, standardisation and official announcement. However, a rough history is given in Figure 1 to get a quick look at Objective-C amongst its ancestors and "challengers".

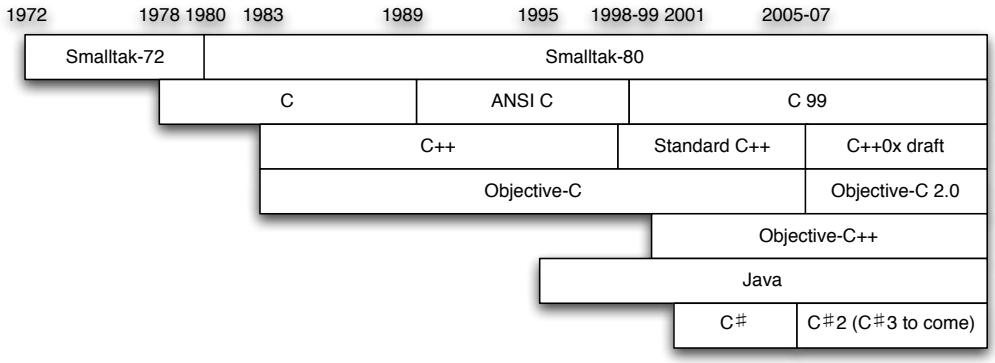

Figure 1: Timeline of Java, C, C#, C++ and Objective-C

Smalltalk-80 is one the first "real" object languages. C++ and Objective-C are two different branches that build a superset of the C language. Objective-C is very close to Smalltalk in terms of syntax and dynamism, while C++ is much more static, with the goal of having better run-time performance. Java targets a C++ audience, but is also very inspired by Smalltalk for its object model. That's why, despite this document's title, many references are made to Java. The C# language, developed by Microsoft, is a direct challenger to Objective-C.

Objective-C++ is a kind of merge between Objective-C and C++. It is already usable, but some behaviours are still not perfect. The goal of Objective-C++ is to mix up the syntaxes of Objective-C and C++ to benefit from the best features of both worlds (cf. section 14 on page 64).

1.2 Objective-C 2.0

The present document has been updated to take in account the new features of Objective-C 2.0, which has been released alongside MacOS X10.5. Those features are deep technical improvements, but the high-level modifications for the developers are easily enumerable. They can now use:

- a garbage-collector : cf. section 6.6 on page 48;
- properties : cf. section 12.2 on page 58;
- fast enumeration : cf. section 11.2.2 on page 55;
- new keywords `@optional` and `@required` for protocols : cf. section 4.4 on page 22;
- updated run-time Objective-C library features : cf. section 13.2 on page 63.

Each one is detailed in a specific section.

2 Syntax overview

2.1 Keywords

Objective-C is a superset of the C language. Like with C++, a well-written C program should be compile-able as Objective-C, as long as it is not using some of the bad practices allowed by C. Objective-C has only added some concepts and their associated keywords. To avoid conflicts, these keywords begin with the @ (at) character. Here is the (short) exhaustive list: `@class`, `@interface`, `@implementation`, `@public`, `@private`, `@protected`, `@try`, `@catch`, `@throw`, `@finally`, `@end`, `@protocol`, `@selector`, `@synchronized`, `@encode`, `@defs` (no more documented in [4]). Objective-C 2.0 (cf. 1.2 on the preceding page) has added `@optional`, `@required`, `@property`, `@dynamic`, `@synthesize`. Let us alo quote the values `nil` et `Nil`, the types `id`, `SEL` and `BOOL`, the boolean values being `YES` et `NO`. At last, a few kewords are available in particular contexts, and are not reserved outside: `in`, `out`, `inout`, `bycopy`, `byref`, `oneway` (they can be met when defining protocols : cf. section 4.4.5 on page 24) and `getter`, `setter`, `readwrite`, `readonly`, `assign`, `retain`, `copy`, `nonatomic` (they can be met when defining properties : cf. section 12.2 on page 58).

There is an easy confusion between the language keywords and some methods inherited from the root class `NSObject` (the mother of every class, cf. section 3.1 on page 10). For instance, the similar-looking "keywords" for memory managament, named `alloc`, `retain`, `release` and `autorelease`, are in fact methods of `NSObject`. The words `super` and `self` (cf. section 3.3.1 on page 12), could also be understood as keywords, but `self` is in fact a hidden parameter to each method, and `super` an instruction asking the compiler to use `self` differently. However, the confusion between these false keywords and the true ones will not likely prove problematic in normal use.

2.2 Comments

The comments `/* ... */` and `//` are allowed.

2.3 Mixing up code and declarations

Like in C++, it is possible to insert the declaration of a variable in the middle of a block of instructions.

2.4 New types and values

2.4.1 BOOL, YES, NO

In C++, the boolean type is `bool`. In Objective-C, it is `BOOL`, which can be set to `YES` or `NO`.

2.4.2 `nil`, `Nil` and `id`

These three keywords are explained later in the document, but briefly:

- Every object is of type `id`. This is a tool for weak-typing;
- `nil` is the equivalent of `NULL` for a pointer to an object. `nil` and `NULL` should not be interchangeable.
- `Nil` is the equivalent of `nil` for a class pointer. In Objective-C, a class is an object (it is the instance of a meta-class).

2.4.3 `SEL`

The `SEL` type can store *selectors* values, which are method identifiers unrelated to any class instance object. These values can be computed by a call to `@selector`. A selector can be used as a kind of pointer to a method, even if it is not technically a real pointer to a function. See section 3.3.5 on page 15 for more details.

2.4.4 `@encode`

For the purpose of interoperability, teh data types in Objective-C, even custom types, and prototypes of functions or methods, can be ASCII-encoded, according to a documented format [4]. A call to `@encode(a type)` returns a C string (`char*`) representing the type.

2.5 Organization of source code: *.h* and *.m* files, inclusion

Like in C++, it is useful to split the code between interface and implementation for each class. Objective-C uses *.h* files for headers, and *.m* files for the code; *.mm* files are used for Objective-C++ (see Section 14 on page 64). Objective-C introduces the `#import` directive to replace `#include`. Indeed, every C header should use compilation guards to prevent multiple inclusions. This is automatic when using `#import`. Below is a typical interface/implementation example. The Objective-C syntax is explained later.

C++

```
//In file Foo.h

#ifndef __FOO_H__ //compilation guard
#define __FOO_H__ //

class Foo
{
...
};

#endif
```

```
//In file Foo.cpp

#include "Foo.h"

...
```

Objective-C

```
//In file Foo.h

//class declaration, different from
//the "interface" Java keyword
@interface Foo : NSObject
{
...
}
@end
```

```
//In file Foo.m

#import "Foo.h"

@implementation Foo
...
@end
```

2.6 Class names: why NS?

In this document, almost all class names begin with *NS*, like *NSObject* or *NSString*. The reason is simple: they are Cocoa classes, and most Cocoa classes begin with *NS* since they were initiated under NeXTStep.

It is a common practice to use a prefix to identify the origin of a class.

2.7 Differencing functions and methods

Objective-C is not a language with "function calls using square brackets". This would be a legitimate thought when observing code like that :

```
[object doSomething];
```

instead of

```
object.doSomething();
```

But in fact, Objective-C is a superset of C, so that *functions* match the same syntax and semantics as C for declaration, implementation and call. On the contrary, *methods*, which do not exist in C, have a special syntax, which includes square brackets. Moreover, the difference is not only in the syntax, but also the meaning. This is detailed further in Section 3.2 on the next page: this is not a method call, this is sending a message. This is not just a simple academic distinction; it has implications on the mechanism of Objective-C. Even if it is the same regarding the source code organization, this mechanism allows much more dynamism. For instance, it is compatible with adding a method at run-time (cf. section 13.2 on page 63). The syntax is also more readable, especially with nested calls (cf. section 3.3.1 on page 12).

3 Classes and objects

Objective-C is an object-oriented language: it manages classes and objects. Objective-C uses a strict object model, unlike C++ which has many discrepancies against the ideal object model. For instance, in Objective-C, classes are objects and can be dynamically managed: it is possible to add classes at run-time, create instances based on the name of the class, ask a class for its methods, and so on. This is much more powerful than C++ RTTI (cf. section 13.1 on page 62), which have been added to a very "static" language. Discouraging RTTI use is common since the results may depend on the compiler itself and lack portability.

3.1 Root class, type `id`, `nil` and `Nil` values

In an object-oriented language, each program makes use of a set of classes. Unlike C++, Objective-C defines a root class. Every new class should be a descendant of the root class. In Cocoa, that class is `NSObject`, and it provides a huge number of facilities for the run-time system. The root class notion is not specific to Objective-C; it's related to the object model. Smalltalk and Java make use of a root class, while C++ does not.

Strictly speaking, every object should be of type `NSObject`, and every pointer to an object could be declared as `NSObject*`. In fact, one can use the type `id` instead. This is a short and handy way to declare a pointer to any object, and provides dynamic type-checking instead of static type-checking. It is very useful for some weak typing on generic methods. Please note that a null pointer to an object should be set to `nil`, not `NULL`. These values are not interchangeable. A normal C pointer can be set to NULL, but `nil` was introduced in Objective-C for pointers to objects. In Objective-C, classes are also objects (meta-class instances), and it is possible to declare a pointer to a class. Their null value is `Nil`.

3.2 Class declaration

It is hard to show with a single example all the differences between Objective-C and C++ for class declaration and implementation. Syntax and concepts are interleaved and require explanation. In the following, the differences are exposed sequentially and specifically.

3.2.1 Attributes and methods

In Objective-C, attributes are called *instance data*, and member functions are called *methods*.

C++

```
class Foo
{
  double x;

  public:
    int   f(int x);
    float g(int x, int y);
};



int   Foo::f(int x) {...}
float Foo::g(int x, int y) {...}
```

Objective-C

```
@interface Foo : NSObject
{
  double x;
}

-(int)   f:(int)x;
-(float) g:(int)x :(int)y;
@end

@implementation Foo
 -(int)   f:(int)x {...}
 -(float) g:(int)x :(int)y {...}
@end
```

In C++, attributes and methods are declared together inside the braces of the class. Method implementation syntax is similar to C, with the addition of the scope resolution operator (`Foo::`).

In Objective-C, attributes and methods cannot be mixed. The attributes are declared in braces, the methods follow. Their implementation lies in an `@implementation` block.

This is a major difference with C++, since some methods can be implemented without being exposed in the interface. This is detailed later. Briefly, this is a way to clean up header files by removing unnecessary declarations ("private" methods, and silently redefined virtual methods like destructors). Please see Section 4.3.2 on page 21 for further explanations.

Instance methods are prefixed by the minus symbol "-", and class methods by the plus symbol "+"(cf. section 3.3.9 on page 18); this symbol has nothing to do with the UML notation and the meaning *public* or *private*. The type of the parameters are enclosed in parenthesis, and the parameters are separated by the symbol ":". Please see Section 3.3.1 on the next page for further explanations on the syntax of prototypes.

In Objective-C, there is no need for a semi-colon at the end of a class declaration. Also note that the keyword to declare a class is `@interface` and not `@class`. The keyword `@class` is only used in forward declarations (cf. section 3.2.2 on the current page). Finally, if there is no instance data in a class, the braces, which would enclose nothing, can be ommitted.

3.2.2 Forward declarations: `@class`, `@protocol`

To avoid cyclic dependencies in header files, the C language supports the *forward declaration*, that allows the coder to declare a class when the only required knowledge is its existence and not its structure. In C++, the keyword `class` is used; in Objective-C, it is `@class`. The keyword `@protocol` can also be used to anticipate the declaration of a protocol (cf. section 4.4 on page 22).

C++

```
//In file Foo.h

#ifndef __FOO_H__
#define __FOO_H__

class Bar; //forward declaration
class Foo
{
  Bar* bar;

  public:
    void useBar(void);
};

#endif
```

```
//In file Foo.cpp

#include "Foo.h"
#include "Bar.h"

void Foo::useBar(void)
{
 ...
}
```

Objective-C

```
//In file Foo.h

@class Bar; //forward declaration
@interface Foo : NSObject
{
  Bar* bar;
}

-(void) useBar;

@end
```

```
//In file Foo.m

#import "Foo.h"
#import "Bar.h"

@implementation Foo

-(void) useBar
{
 ...
}

@end
```

3.2.3 `public`, `private`, `protected`

One major feature of the object model is data encapsulation, which limits the visibility of data to some parts of the code, in order to ensure its integrity.

C++	Objective-C

`class Foo`
`{`
`  public:`
`  int x;`
`  int apple();`
 	
`  protected:`
`  int y;`
`  int pear();`
 	
`  private:`
`  int z;`
`  int banana();`
`};`
	`@interface Foo : NSObject`
	`{`
	`  @public`
	`  int x;`
	
	`  @protected:`
	`  int y;`
	
	`  @private:`
	`  int z;`
	`}`
	
	`-(int) apple;`
	`-(int) pear;`
	`-(int) banana;`
	`@end`

In C++, attributes and methods can belong to a `public`, `protected` or `private` scope. The default mode is `private`.

In Objective-C, only the instance data can be `public`, `protected` or `private`, and the default visibility is `protected`. Methods can only be public. However, it is possible to mimic the private mode, by implementing some methods in the `@implementation`, without declaring them in the `@interface`, or using the notion of class category (cf. section 4.5 on page 25). It does not prevent methods from being called, but they are less exposed. Implementing a method without a previous declaration is a special property of Objective-C and has a specific purpose, as explained in Section 4.3.2 on page 21.

Inheritance cannot be tagged `public`, `protected` or `private`. The only way is `public`. Inheritance in Objective-C looks more like Java than C++ (section 4 on page 21).

3.2.4 `static` attributes

It's not possible in Objective-C to declare a class data attribute (`static` as in C++). However, it is possible to do the same thing in a different way: use a global variable in the implementation file (optionally with the `static` C keyword to limit the scope). The class can then use accessors on it (with class methods or normal methods), and its initialization can be done in the `initialize` method of the class (cf. section 5.1.10 on page 34).

3.3 Methods

The syntax for a method in Objective-C is different from the syntax for common C functions. This section aims to describe this syntax and adds some information on the underlying message sending principle.

3.3.1 Prototype and call, instance methods, class methods

- A method is prefixed by "−" if it is an instance method (common case), or by "+" if it is a class method (`static` in C++). This symbol is in no way linked to the UML notation for *public* or *private*. Methods are always public in Objective-C;

- the types for the return value, or for the parameters, are enclosed by parentheses;
- the parameters are separated by a colon ":";
- the parameters can be associated with a **label**, a name specified just before ":"; **the label is then part of the name of the method and modifies it**. That makes calls to this function particularly readable. In fact, the label usage should be systematic. Note that the first parameter cannot have a label – indeed, its label is already the name of the method;
- a method name can be the same as an attribute name, without conflict. This is very useful for getters (cf. section 6.4.8 on page 46).

C++

```
//prototype
void Array::insertObject(void *anObject, unsigned int atIndex)

//use with a "shelf" instance of the Array class and a "book" object
shelf.insertObject(book, 2);
```

Objective-C

Without label (direct translation from C++)

```
//prototype
//The method is named "insertObject::", the colon being used to separate
//the parameters (it is not a scope resolution operator like in C++)
-(void) insertObject:(id)anObject:(unsigned int)index

//use with a "shelf" instance of the Array class and a "book" object
[shelf insertObject:book:2];
```

With label

```
//prototype. The "index" parameter is now labeled "atIndex"
//The method is now named "insertObject:atIndex:"
//The call can easily be read like a sentence
-(void) insertObject:(id)anObject atIndex:(unsigned int)index

//use with a "shelf" instance of the Array class and a "book" object
[shelf insertObject:book:2];         //Error !
[shelf insertObject:book atIndex:2]; //OK
```

Please note that the syntax with square brackets should not be read as *calling the method* `insertObject` of object "shelf" but rather *sending the message* `insertObject` to object "shelf". This is what makes Objective-C. One can send any message to any target. If it is not able to handle the message, it can ignore it (an exception is raised but the program does not abort). If, at the time it receives the message, a target is able to handle it, then the matching method will be called. The compiler is able to raise a warning if a message is sent to a class for which it is known that no matching method exists. This is not considered as an error thanks to the *forwarding* facility (cf. section 3.4.3 on page 19). If a target is only known under the type `id`, no warning will be made at compile time; a potential error would be raised at run-time.

3.3.2 `this`, `self` and `super`

There are two particular targets for a message: `self` and `super`. `self` is the current object (like `this` in C++), `super` is the parent class. The keyword `this` does not exist in Objective-C. It is replaced by `self`.

NB: `self` is not a real keyword, it is a hidden parameter that each method receives, the value of which is the current object. Its value can be changed, unlike C++'s `this` keyword. However, this is only useful in constructors (cf. section 5.1 on page 27).

3.3.3 Accessing instance variables inside a method

Like in C++, an Objective-C method can access the instance variables of its current object. The optional `this->` in C++ can be written `self->`.

C++	Objective-C

`class Foo`
`{`
`  int x;`
`  int y;`
`  void  f(void);`
`};`
 	
`void Foo::f(void)`
`{`
`  x = 1;`
`  int y; //making ambiguity with this->y`
`  y = 2; //using local y`
`  this->y = 3; //disambiguating`
`}`
	`@interface Foo : NSObject`
	`{`
	`  int x;`
	`  int y;`
	`}`
	`-(void) f;`
	`@end`
	
	`@implementation Foo`
	`-(void) f`
	`{`
	`  x = 1;`
	`  int y; //making ambiguity with this->y`
	`  y = 2; //using local y`
	`  self->y = 3; //disambiguating`
	`}`
	`@end`

3.3.4 Prototype id and signature, overloading

A function is a part of code that can be referenced, for instance to use as function pointers, or functors. Moreover, even if the name of the function is a good candidate for a unique id, one must take care when using overloading. C++ and Objective-C use opposite ways to differentiate prototypes. The first is based on parameters types, while the second is based on parameter labels.

In C++, two functions can have the same name as long as their parameters have different types. When using methods, the `const` option is also discriminative.

C++

```
int f(int);
int f(float); //OK, float is different from int

class Foo
{
  public:
    int g(int);
    int g(float); //OK, float is different from int
    int g(float) const; //OK, const is discriminative
};

class Bar
{
  public:
    int g(int); //OK, we are in Bar::, different from Foo::
}
```

In Objective-C, all functions are C functions: they cannot be overloaded (unless the compiler can be told to use C99; like `gcc` does). However, methods use a different syntax, and can be differentiated thanks to parameter labels.

Objective-C

```
int f(int);
int f(float); //Error : C functions cannot be overloaded

@interface Foo : NSObject
{
}

-(int) g:(int) x;
-(int) g:(float) x; //Error : this method is not different
                    // from the previous one (no label)
-(int) g:(int) x :(int) y;   //OK : two anonymous labels
-(int) g:(int) x :(float) y; //Error : not different from the
                             //previous one
-(int) g:(int) x andY:(int) y;   //OK : second label is "andY"
-(int) g:(int) x andY:(float) y; //Error : not different from the
                                 //previous one
-(int) g:(int) x andAlsoY:(int) y; //OK : second label is
                                   //"andAlsoY", different from "andY"
@end
```

The label-based identifying method is useful for expressing the exact "name" of the function, as stated below.

```
@interface Foo : NSObject {}

//The method name is "g"
-(int) g;

//The method name is "g:"
-(int) g:(float) x;

//The method name is "g::"
-(int) g:(float) x :(float) y;

//The method name is "g:andY:"
-(int) g:(float) x andY:(float) y;

//The method name is "g:andZ:"
-(int) g:(float) x andZ:(float) z
@end
```

Clearly, two Objective-C methods are differentiated by *labels* instead of *types*. This is a way to replace "pointers on member functions", expressed by *selectors*, as explained in section 3.3.5 on the current page.

3.3.5 Pointer to member function: Selector

In Objective-C, the *methods* have a particular syntax using parentheses and labels. It is not possible to declare *functions* with this syntax. The notion of pointer to *functions* is the same in C and Objective-C. It is only for pointers on *methods* where a difference occurs.

In C++, the syntax is hard but consistent with the C langage : the pointer focus is on types.

C++

```
class Foo
{
  public:
    int f(float x) {...}
};

Foo bar
int (Foo::*p_f)(float) = &Foo::f; //Pointer to Foo::f

(bar.*p_f)(1.2345); //calling bar.f(1.2345);
```

In Objective-C, a new type has been introduced. Such a "pointer to a method" is called a *selector*. Its type is `SEL` and its value is computed using `@selector` on the exact name of the method (with labels of parameters). Calling the method can be done using the class `NSInvocation`. Most of the time, utility methods of the `performSelector:` family (inherited from NSObject) are handier, but a little restrictive. The simplest three ones are:

```
-(id) performSelector:(SEL)aSelector;
-(id) performSelector:(SEL)aSelector withObject:(id)anObjectAsParameter;
-(id) performSelector:(SEL)aSelector withObject:(id)anObjectAsParameter
                                    withObject:(id)anotherObjectAsParameter;
```

The returned value is the same as that of the called method. For methods which have parameters that are not objects, one should generally use wrapper classes like `NSNumber`, that can provide `float`, `int`, and so on. The `NSInvocation` class, which is more general and powerful, can also be used (see the documentation).

According to what has been said, nothing prevents a method from being called on an object, even if the object does not implement it. In fact, the method is effectively triggered if the message is accepted. But an exception, that can be caught, is raised when the object does not know the method; the application is not interrupted. Moreover, it is possible to check that an object can trigger a method by calling `respondsToSelector:`.

Finally, the value of `@selector()` is computed at compile-time, which does not slow down the code.

Objective-C

```
@interface Slave : NSObject {}
  -(void) readDocumentation:(Document*)document;
@end

//Let us suppose an array "array[]" of 10 slaves,
//and a document "document"

//Normal method call
for(i=0 ; i<10 ; ++i)
  [array[i] readDocumentation:document];

//Just for the example, try to use performSelector: instead
for(i=0 ; i<10 ; ++i)
  [array[i] performSelector:@selector(readDocumentation:)
                withObject:document];

//The type of a selector is SEL
//The following version is not more efficient than the previous one,
//since @selector() is evaluated at compile-time
SEL methodSelector = @selector(readDocumentation:);
for(i=0 ; i<10 ; ++i)
  [slaves[i] performSelector:methodSelector withObject:document];

//for an object "foo" which type is not known (id)
//the test is not mandatory, but prevents an exception to be raised
//if the object has no readDocumentation: method.
if ([foo respondsToSelector:@selector(readDocumentation:)])
  [foo performSelector:@selector(readDocumentation:) withObject:document];
```

A selector can thus be used as a very simple function parameter. Generic algorithms, like sorting, may be easily specialized that way (cf. 11.3 on page 55).

A selector is not, strictly speaking, a pointer to a function; its real underlying type is a C string, registered by the run-time as a method identifier. When a class is loaded, its methods are automatically registered in a table, so that `@selector()` can work as expected. This way, for instance, the equality of two selectors can be done on adresses with `==` rather than reyling on a string comparison.

The real adress of a method, seen as a C function, can be obtained with a different notion, and the type `IMP`, that is briefly presented in section 11.3.2 on page 55. It is rarely used but for optimization. Virtual calls, for instance, are handled by selectors but not by `IMP`. The Objective-C equivalent for C++ method pointers is definitely in selectors.

Finally, you may remember that `self` is in Objective-C, like `this` in C++, a hidden parameter of every method that stores the current object. You can also note that there is a second hidden parameter, `_cmd`, which is the current method .

Objective-C

```
@implementation Foo

-(void) f:(id)parameter //equivalent to a C function of type
                        //"void f(id self, SEL _cmd, id parameter)"
{
  id currentObject = self;
  SEL currentMethod = _cmd;
  [currentObject performSelector:currentMethod
                      withObject:parameter]; //recursive call
  [self performSelector:_cmd withObject:parameter]; //idem
}

@end
```

3.3.6 Default values of parameters

Objective-C does not allow a default value to be specified for the parameters of functions or methods. One should create as much functions as necessary when some parameters could be optional. In the case of constructors, one should use the notion of *designated initializer* (section 5.1.7 on page 32).

3.3.7 Variable number of arguments

Objective-C allows the use of methods with a variable number of arguments. Like in C, the syntax relies on using "`...`" as the last argument. This is rarely useful, even if many Cocoa methods are doing so. More details can be found in the Objective-C documentation.

3.3.8 Anonymous arguments

In C++, it is possible to give no name to a parameter in a prototype, since the type is sufficient to characterize the signature of a function. This is not possible in Objective-C.

3.3.9 Prototype modifiers (`const`, `static`, `virtual`, `"= 0"`, `friend`, `throw`)

In C++, a few modifiers can be added to the prototype of a function. None of them exist in Objective-C. Here is the list:

- `const`: a method cannot be set `const`. As a consequence, the `mutable` keyword cannot exist;
- `static`: making the difference between an instance method and a class method is done through the use of "-" or "+" in front of the prototype;
- `virtual`: Objective-C methods are virtual, so the keyword is useless. **Pure** virtual methods are implemented with a formal protocol (cf. section 4.4 on page 22);
- `friend`: there is no "friend" notion in Objective-C;
- `throw`: in C++, one can restrict a method to transmit some exceptions only. This is not possible in Objective-C.

3.4 Messages and transmission

3.4.1 Sending a message to `nil`

By default, it is legal to send a message (call a method) to `nil`. The message is just ignored. The code can be greatly simplified by reducing the number of tests usually made with the null pointer. GCC has an option to disable that handy behaviour, for extra optimizations.

3.4.2 Delegating a message to an unknown object

Delegation is common with user interface elements in Cocoa (tables, outlines...), taking advantage of the ability to send a message to an unknown object. An object can, for example, delegate some tasks to an assistant.

```
//This is a function to define an assistant
-(void) setAssistant:(id)slave
{
  [assistant autorelease]; //see the section about memory management
  assistant = [slave retain];
}
```

```
//the method performHardWork can use delegation
-(void) performHardWork:(id)task
{
  //the assistant is not known
  //we check that it can handle the message
  if ([assistant respondsToSelector:@selector(performHardWork:)])
    [assistant performHardWork:task];
  else
    [self findAnotherAssistant];
}
```

3.4.3 Forwarding: handling an unknown message

In C++, code cannot be compiled if a method is called on an object that does not implement it. In Objective-C, there's a difference: one can always send a message to an object. If it can't be handled at run-time, it will be ignored (and raise an exception); moreover, instead of ignoring it, it can forward the message to another object.

When the compiler is told the type of an object, it can detect if a message sending – method call – will fail, and raise a warning. However, this is not an error, since in this case, an alternative is available. This second chance is represented by a call to the `forwardInvocation:` method, which can be redefined to redirect the message at the very last moment. This is obviously a method of `NSObject`, which does nothing by default. Here is another way to manage assistant objects.

```
-(void) forwardInvocation:(NSInvocation*)anInvocation
{
  //if we are here, that is because the object cannot handle
  //the message of the invocation
  //the bad selector can be obtained by sending "selector"
  //to the object "anInvocation"
  if ([anotherObject respondsToSelector:[anInvocation selector]])
    [anInvocation invokeWithTarget:anotherObject];
  else  //do not forget to try with the superclass
    [super forwardInvocation:anInvocation];
}
```

Even if a message can be handled, at last, in a `forwardInvocation:`, and only there, a check based on `respondsToSelector:` will still return `NO`. Indeed, the `respondsToSelector:` mechanism is not designed to guess whether `forwardInvocation:` will work or not.

Using the *forward invocation* can be thought as a bad practice, because it is triggering some code while an error should occur. In fact , very good uses can be made of that mechanism, like in the implementation of the Cocoa's `NSUndoManager`. It allows an exceptionnaly pleasant syntax : the Undo manager can records calls to methods, though it is not itself the target of those calls.

3.4.4 Downcasting

Downcasting is needed in C++ to call methods on a subclass, when only a parent class pointer is known, this is made possible using `dynamic_cast`. In Objective-C, this practice is not necessary, since a message can be sent to an object event if it seems that the object cannot handle it.

However, to avoid a compiler warning, one can simply *cast* the type of the object; there is no explicit *downcasting* operator in Objective-C, the traditionnal *cast* syntax of the C language can be used.

```
//NSMutableString is a subclass of NSString (string of characters)
//that allows mutating operations
//the "appendString:" method only exists in NSMutableString
NSMutableString* mutableString = ...initializing a mutable string...
NSString* string = mutableString;//storing in an NSString pointer

//those different calls are all valid
[string                   appendString:@"foo"];//compiler warning
[(NSMutableString*)string appendString:@"foo"];//no warning
[(id)string               appendString:@"foo"];//no warning
```

4 Inheritance

4.1 Simple inheritance

Objective-C obviously implements the notion of inheritance, but does not support multiple inheritance. This limitation is addressed by other concepts (protocols, class categories) that are explained later in this document (sections 4.4 on the following page, 4.5 on page 25).

C++	Objective-C

`class Foo : public Bar,`
`protected Wiz`
`{`
`}`
	`@interface Foo : Bar //single inheritance`
	`//An alternative technique must be`
	`//used to also "inherit" from Wiz`
	`{`
	`}`
	`@end`

In C++, a class can be derived from one or several other classes, using `public`, `protected` or `private` mode. In the methods, one can reference a superclass using the scope resolution operator `::` (`Bar`::, `Wiz`::).

In Objective-C, one can derive from one class only, using `public` mode. A method can reference the superclass with the (false) keyword `super`, like in Java.

4.2 Multiple inheritance

Objective-C does not implement multiple inheritance, but brings other concepts, the *protocols* (cf. 4.4 on the following page) and the *categories* (cf. 4.5 on page 25).

4.3 Virtuality

4.3.1 Virtual methods

In Objective-C, all methods are virtual. Hence, the *virtual* keyword does not exist and has no equivalent.

4.3.2 Silent redefinition of virtual methods

It is possible in Objective-C to implement a method that has not been declared in the *interface* section. This feature does not replace the `@private` notion for the methods (even if it can be used to "hide" methods) : they can still be called; but it does lighten the interface declarations.

This is not a bad practice: the methods one uses with this technique are often "well-known" methods of the super-classes. Many methods of the root-class `NSObject` are silently redefined. One can quote the constructor `init` (cf. section 5.1 on page 27), the destructor `dealloc` (cf. section 5.2 on page 34), the drawing method `drawRect:` of views, and so on.

The interfaces are then lighter and easier to read, even if it is more difficult to see what can be redefined, implying a regular reading of the super class documentation.

The notion of pure virtual methods (a method that must be redefined in sub-classes), is addressed by the concept of *formal protocols* (cf. section 4.4.1 on the following page on the next page).

4.3.3 Virtual inheritance

Virtual inheritance is not relevant in Objective-C, since inheritance cannot be multiple and has none of the associated problems.

4.4 Protocols

Java and C# fix the lack of multiple inheritance by the notion of *interface*. In Objective-C, the same notion is used, and is called a *protocol*. In C++, this would be an abstract class. A protocol is not a real class: it can only declare methods and cannot hold any data. There are two types of protocols : *formal* and *informal*.

4.4.1 Formal protocol

A formal protocol is a set of methods that must be implemented in any conforming class. This can be seen as a certification regarding a class, ensuring that it is able to handle everything that is necessary for a given service. A class can conform to an unlimited number of protocols.

C++

```
class MouseListener
{
  public:
    virtual bool mousePressed(void) = 0; //pure virtual method
    virtual bool mouseClicked(void) = 0; //pure virtual method
};

class KeyboardListener
{
  public:
    virtual bool keyPressed(void) = 0; //pure virtual method
};

class Foo : public MouseListener,  public KeyboardListener {...}

//Foo MUST implement mousePressed, mouseClicked and keyPressed
//It can then be used as an event listener for the mouse and the keyboard
```

Objective-C

```
@protocol MouseListener
  -(BOOL) mousePressed;
  -(BOOL) mouseClicked;
@end

@protocol KeyboardListener
  -(BOOL) keyPressed;
@end

@interface Foo : NSObject  <MouseListener,  KeyboardListener>
{
...
}
@end

//Foo MUST implement mousePressed, mouseClicked and keyPressed
//It can then be used as an event listener for the mouse and the keyboard
```

In C++, a protocol is implemented by an abstract class and pure virtual methods. The abstract class in C++ is more powerful than the Objective-C protocol since it can contain data.

In Objective-C, the protocol is a specific concept. The syntax using angular brackets <...> is not linked to the C++ templates, a feature that does not exist in Objective-C.

A class can implement all the methods of a protocol without declaring its conformance. In this case, the `conformsToProtocol`: method returns `NO`. For efficiency reasons, `conformsToProtocol`: does not check, method-by-method, the conformance to a protocol, but is based on the explicit declaration from the developer. However, the negative answer to `conformsToProtocol:` does not prevent the program from behaving correctly if the methods of the protocol are called. Here is the prototype of `conformsToProtocol`:

```
-(BOOL) conformsToProtocol:(Protocol*)protocol
//a Protocol object is returned by a call to @protocol(protocol name)
```

The type of an object that conforms to a formal protocol can be added the name of the protocol itself, between angular braces. This is useful for assertions. For example :

```
//the following standard Cocoa method takes one parameter which is of
//any type (id), but must conform to the protocol NSDraggingInfo
-(NSDragOperation) draggingEntered:(id <NSDraggingInfo>)sender;
```

4.4.2 Optional methods

It may be desirable that a class conforms to a protocol, to show that it can handle a particular service, but without forcing it to conform to the *whole* protocol. For instance, in Cocoa, the notion of a *delegate* object is widely used: an object can be given an assistant, to handle some tasks, but not all of them.

An immediate solution would be to split a formal protocol into multiple ones, and then make the class conform to a subset of these protocols. This is not very handy. Cocoa brings a solution in the notion of *informal protocols*. With Objective-C 1.0, the *informal protocols* coudl be used (cf. section 4.4.3). With Objective-C 2.0, the new keywords `@optional` and `required` can make the difference between optional and required methods.

```
@protocol Slave

@required //required part
-(void) makeCoffee;
-(void) duplicateDocument:(Document*)document count:(int)count;

@optional //optional part
-(void) sweep;

@required //you can split required/optional sections
-(void) bringCoffee;

@end
```

4.4.3 Informal protocol

The informal protocol i not really a “protocol” : it creates no constraint upon the code. But it is “informal” by nature and targets it to the code auto-documentation.

An informal protocol lets a developer group methods by application field, so that it can organize its classes consistently.

So, it is not so surprising that an informal protocol is not declared with the relaxation of a formal protocol. Another concept is used : the *class category* (cf. section 4.5 on page 25).

Let us imagine a service called “document managing”. Assume there is a difference between green, blue and red documents. Even if the class can only handle blue documents, a `Slave` class is preferred to using three formal protocols: `manageGreenDocuments`, `manageBlueDocuments`

and `manageRedDocuments`. To the `Slave` class is rather added a *category* `DocumentsManaging`, declaring the methods for the tasks it is able to accomplish. The name of the category is specified in parenthesis (more explanations are given in Section 4.5 on the following page):

```
@interface Slave (DocumentsManaging)
-(void) manageBlueDocuments:(BlueDocument*)document;
-(void) trashBlueDocuments:(BlueDocument*)document;
@end
```

Any class can use the `DocumentsManaging` category, to declare methods that are related to that service.

```
@interface PremiumSlave (DocumentsManaging)
-(void) manageBlueDocuments:(BlueDocument*)document;
-(void) manageRedDocuments:(RedDocument*)document;
@end
```

A developer can then browse the code and see the `DocumentsManaging` category. Hence, he can suppose that the class is useful for some tasks, and he can check which ones exactly by consulting the documentation. Even if he does not check the source code, a run-time request is still possible:

```
if ([mySlave respondsToSelector:@selector(manageBlueDocuments:)])
  [mySlave manageBlueDocuments:document];
```

Strictly speaking, apart from the knowledge of the prototypes, the informal protocol is useless to the compiler, it does not restrict the use of objects. However, it is vital for self-documenting code, making APIs more readable.

4.4.4 Object of type `Protocol`

At run-time, a protocol is like a class represented by an object, and is typed `Protocol*`. Such an object can be used as a parameter of a method like `conformsToProtocol:` (cf. section 13.1.2 on page 62).

The keyword `@protocol`, that is used to declare protocols, is also used to build a `Protocol*` object from its name:
`Protocol* myProtocol = @protocol(`*protocol name*`)`.

4.4.5 Message qualifiers for distant objects

Thanks to the dynamism of Objective-C, distant objects can communicate easily. They can belong to distinct programs, on different machines, but can delegate some tasks and exchange some information. Now, formal protocols are a perfect way to ensure that an object conforms to a given service, wherever it comes from. The formal protocol concept has been given some extra keywords to allow for more efficient communication between distant objects.

These keywords are `in`, `out`, `inout`, `bycopy`, `byref` and `oneway`. It is only applicable to distributed objects, and outside a protocol definition, they are not reserved keywords and can be freely used.

These keywords are inserted inside the prototypes of the methods declared inside a formal protocol, to add extra information regarding their behaviour. They can be used to specify which parameters are *input* parameters, and which are *ouput* results; it is also possible to tell whether they are to be used by copy or by reference; and the methods can be made synchronous or not. Here are the different meanings:

- an `in` parameter is an input variable;
- an `out` parameter is an output variable;
- an `inout` parameter can be used in both ways (input and output);

- a `bycopy` parameter is transmitted by copy ;
- a `byref` parameter is transmitted by reference (without copy) ;
- a `oneway` method is asynchronous (the result is not immediately expected) - hence it must return `void`.

For instance, this is an asynchronous method that returns an object:

```
-(oneway void) giveMeAnObjectWhenAvailable:(bycopy out id *)anObject;
```

By default, parameters are considered to be `inout`, except `const` pointers, that are supposed to be `in`. Choosing `in` or `out` instead of `inout` is an optimization. The default mode to transmit the parameters is `byref`, and the methods are synchronous by default (without `oneway`).

For parameters transmitted by value, like non-pointers variables, `out` and `inout` make no sense, only `in` is correct.

4.5 Class categories

Creating *categories* for a class is a way to split its implementation into several parts. Each *category* is a part of the class. A class can use an arbitrary number of categories, but none can add instance data. It brings the following benefits:

- For the meticulous developer, it is possible to make groups of methods. For a very rich class, its different roles can be cleanly separated;
- It is possible to compile them separately, making collaborative work possible on the same class;
- If the interface of a category and its implementation are present in some implementation file (*.m* file), it is very easy to define *private* methods, which are only visible inside the file (even if anybody who knows the prototype can use them, there is no calling restriction). An appropriate name for such a category could be *FooPrivateAPI*;
- A class can be extended differently in different applications, without duplicating the common code. Any class can be extended, even existing Cocoa classes.

The last point is important: each developer would have their preference to extend a standard class with the methods that are useful for them. This is not a real problem: inheritance is a way to do that. However, in the context of simple inheritance, this can lead to a heavy subclassing scheme. Moreover, it can seem a disproportional effort to make a subclass for a single method. The class categories are an elegant solution to this problem.

C++	Objective-C

`class MyString : public string`
`{`
`  public:`
`  //counts the vowels`
`  int vowelCount(void);`
`};`
 	
`int MyString::vowelCount(void)`
`{`
`...`
`}`
	`@interface NSString (VowelsCounting)`
	`//please note that no brackets {} are used`
	`-(int) vowelCount; //counts the vowels`
	`@end`
	
	`@implementation NSString (VowelsCounting)`
	`-(int) vowelCount`
	`{`
	`...`
	`}`
	`@end`

In C++, the new class is usable without restrictions.

In Objective-C, the NSString class (a standard Cocoa class) is given an extension that is usable inside the whole program. No new class is created. Every NSString object benefits from the extension (even constant strings, cf. section 9.1 on page 52). But no instance variable can be added inside a category, so there is no *{...}* block.

A category can even be anonymous, which is perfect for a "private" one.

```
@interface NSString ()
//please note that no brackets {} are used
-(int) myPrivateMethod;
@end

@implementation NSString ()
-(int) myPrivateMethod
{
...
}
@end
```

4.6 Joint use of protocols, categories, subclassing:

The only restriction in the joint specification of protocols, categories and derivation, is that a subclass and a category cannot be declared concurrently; two steps are needed.

```
@interface Foo1 : SuperClass <Protocol1, Protocol2, ... > //ok
@end

@interface Foo2 (Category)  <Protocol1, Protocol2, ... > //ok
@end

//below : compilation error
@interface Foo3 (Category)  : SuperClass <Protocol1, Protocol2, ... >
@end

//a solution :
@interface Foo3 : SuperClass <Protocol1, Protocol2, ... > //step 1
@end
@interface Foo3 (Category) //step 2
@end
```

5 Instantiation

The instantiation of a class leads to two problems: how are the notions of a constructor/ destructor/ copy operator implemented, and how are they managed in memory ?

First, an important point: in the C and C++ languages, variables are said to be "automatic" by default: unless they are declared `static`, they only exist inside their definition block. Only dynamically allocated memory is usable beyond, until the matching `free()` or `delete` method is called. Objects follow this rule in C++.

However, in Objective-C, **all objects are dynamically allocated**. This is rather logical, C++ being very static, and Objective-C being very dynamic. Objective-C's dynamism wouldn't be that rich if objects weren't created at run-time.

Please see the section 6 on page 39 for a detailed explanation about the way to retain or release objects.

5.1 Constructors, initializers

5.1.1 Distinction between *allocation* and *initialization*

In C++, allocation and initialization of an object are mixed when calling the constructor. In Objective-C, they are two different methods.

Allocation is handled by the **class method** `alloc`, which also initializes all instance data. Instance data are set to 0, except the `isa` (is-a) pointer of `NSObject`, which value describes the exact type of the newly created object at run-time. Should instance data be set to a particular value, depending on the parameters of construction, then the corresponding code is deported inside an **instance method**. Its name usually begins with *init*. Therefore, the construction is clearly divided into two steps : allocation and initialization. The `alloc` message is sent to the *class*, and the `init...` message is sent to the *object* newly instantiated by `alloc`.

The initialization step is not optional, and `alloc` should always be followed by `init`; with successive calls to superclass initializers, this must end in calling the `init` method of `NSObject`, which performs important work.

In C++, the name of the constructor cannot be chosen. In Objective-C, the initializer is a method like any other, and the `init` prefix is traditional but not mandatory. However, you are strongly encouraged to follow a rule: **the name of an initialization method must begin with "init"**.

5.1.2 Using `alloc` and `init`

A call to `alloc` returns a new object that must be sent `init` . The call to `init` also returns an object. Most of the time, this will be the initial object. Sometimes, when one is using a singleton (an object for which a single instance is allowed), `init` could substitute another return value. So, the returned value of `init` should not be ignored. Usually, `alloc` and `init` are called on the same line.

C++

```
Foo* foo = new Foo;
```

Objective-C

```
Foo* foo1 = [Foo alloc];
[foo1 init]; //bad : the returned value should be used

Foo* foo2 = [Foo alloc];
foo2 = [foo2 init]; //ok, but not handy

Foo* foo3 = [[Foo alloc] init]; //ok, usual way to do
```

To know whether an object was created or not, C++ requires either exception catching, or a test against 0 (if `new(nothrow)` was used). With Objective-C, a test against `nil` is sufficient.

5.1.3 Example of a correct initializer

The constraints for a correct initializer are

- its name begins with *init*;
- it returns the object to use;
- it calls some `init` method of the superclass, so that `init` of `NSObject` will finally be called;
- it takes in account the value returned by `[super init...]`;
- it handles correctly the construction errors, whether they are voluntary or inherited.

Here is an example of object instanciation, in C++ and in Objective-C.

C++

```
class Point2D
{
  public:
    Point2D(int x, int y);
  private:
    int x;
    int y;
};
Point2D::Point2D(int anX, int anY) {x = anX; y = anY;}
...
Point2D  p1(3,4);
Point2D* p2 = new Point2D(5, 6);
```

Objective-C

```
@interface Point2D : NSObject
{
    int x;
    int y;
}
//Note : id is somewhat similar to void* in Objective-C.
//(id) is the most "general" type for an object
-(id) initWithX:(int)anX andY:(int)anY;
@end

@implementation Point2D

-(id) initWithX:(int)anX andY:(int)anY
{
  //an initializer of the superclass must be called
  if (!(self = [super init])) //if the superclass is NSObject, this must be init
    return nil; //in case of super-class failure, return nil to propagate
  //in case of success, make additional initializations
  self->x = anX;
  self->y = anY;
  return self; //and return the object itself
}
@end
...
Point2D* p1 = [[Point2D alloc] initWithX:3 andY:4];
```

5.1.4 `self = [super init...]`

The most surprising syntax in the construction is `self = [super init...]`. Let us recall that `self` is a hidden argument given to every method, representing the current object. Thus, it is a local variable; so why do we have to change its value ? Will Shipley[9] tried to show in a very interesting document that it was useless. Its arguments were relying on hypothesis on the Objective-C run-time that turned out to be false; indeed, `self` must be modified, but explanations are welcome.

It can happen that `[super init]` returns another object than the current one. A singleton would be such a case, but it is a counter-argument of [9] : il is illogic to call twice `init` for a singleton; being there reveals a conception failure on top of that.

However, an API can substitute, to a newly-allocated object, another object. Core Data[1] is doing that, to handle a special treatment of instance data, which are then linked to the fields of a database. When sub-classing the `NSManagedObject` class provided by Cocoa, it is mandatory to take care of that substitution.

In such a case, `self` will hold successively two different values : the first is the one returned by `alloc`, the second is the one returned by `[super init...]`. Modifying the value of `self` has border effects : every access to a data member is using it implicitely, as shown in the code below.

```
@interface B : A
{
int i;
}
@end

@implementation B

-(id) init
{
  //at that step, the value of self is the one returned by alloc

  //let us suppose that A is performing subsitution, returning a different "self"
  id newSelf = [super init];
  NSLog(@"%d", i);//prints the value of self->i
  self = nouveauSelf; //one could think "i" is untouched, but...
  NSLog(@"%d", i);//prints the value of self->i, therefore newSelf->i,
                  //not necessarily the same as previously
  return self;
}

@end

...
B* b = [[B alloc] init];
```

The concise form `self = [super init]` is the most simple to avoid introducing bugs afterwards. However, it is legible to worry about the destiny ot the object pointed by the "old" `self` : indeed, it must be freed.

The first rule is simple : handling the old `self` must be done by the one who substituted it (if it happened); here, it is `[super init...]` who must perform the deallocation. For instance, if you *subclass*`NSManagedObject` (a Cocoa class which performs substitution), you do not have to care about the old `self`. On the contrary, the developer of `NSManagedObject` had to handle it properly.

[1] Core Data is a Cocoa API provided by Apple

Thus, if you happen to develop class that performs substitution, you should know how to deallocate an object during its initialization. This problem is the same as handling errors : what should we do if we want to make the construction fail (invalid parameters, unavailable resources...) ? This is answered in Section 5.1.5.

5.1.5 Initialization failure

When constructing an object (initializing it fact), an error may happen, or being triggered, in three different places :

- 1: before the call to `[super init...]` : if the construction parameters are considered invalid, the initialization must be stopped as soon as possible;
- 2: with the call to `[super init...]` : if the super-class fails, we should abort the current one;
- 3: after the call to `[super init...]` : if an additional resource allocation fails, for instance.

In every case, `nil` must be returned, and deallocating the current object must be done by the one who triggered the error. Here, we are responsible of cases 1 and 3, but not of case 2. To deallocate the current object, one just have to call `[self release]`, which is very natural (cf. section 6 on page 39 about memory management, wich explains the `release`).

The destruction of the object will end up by calling `dealloc` (cf section 5.2 on page 34 on destructors); thus the implementation of that method must be compatible with partially-initialized objects. The fact that all instance data are initialized to 0 by `alloc` is rather useful for that.

```
@interface A : NSObject {
  unsigned int n;
}
-(id) initWithN:(unsigned int)value;
@end

@implementation A

-(id) initWithN:(unsigned int)value
{
  //case #1 (is the construction legible ?)
  if (value == 0) //here, we want a strictly positive value
  {
    [self release];
    return nil;
  }

  //case #2 (is the super-class ok ?)
  if (!(self = [super init])) //even is self is substitued, it's the super-class
    return nil; //who is reponsible, if an error occurs, for releasing "self"

  //case #3 (can the initialization be complete ?)
  n = (int)log(value);
  void* p = malloc(n); //trying to allocate a resource
  if (!p) //in case of failure, we want it to be an error
  {
    [self release];
    return nil;
  }
}

@end
```

5.1.6 "Splitting" construction into `alloc`+`init`

The successive application of `alloc` and `init` can sound laborious in some cases. Fortunately, it can be shortened as a *convenience constructor*. The exact knowledge of such a constructor relies on notions of memory management in Objective-C. So, the exact explanations are given further in this document in Section 6.4.6 on page 42. Briefly, such a constructor, the name of which should be prefixed by the name of the class, behaves like an `init` method, but it performs the `alloc` itself. However, the object is registered in the autorelease pool (cf. section 6.4 on page 39) and will have a limited life cycle if it is not sent a `retain` message. Here is an example :

```
//laborious
NSNumber* tmp1 = [[NSNumber alloc] initWithFloat:0.0f];
...
[tmp1 release];

//handier
NSNumber* tmp2 = [NSNumber numberWithFloat:0.0f];
...
//no need to release
```

5.1.7 Default constructor : designated initializer

The notion of default constructor has no real meaning in Objective-C. Because all objects are allocated dynamically, their construction is always explicit. However, a preferred constructor can be used to factorize some code. Indeed, a correct initializer is most of the time similar to:

```
if (!(self = [super init])) //"init" or another appropriated
  return nil;               //initializer of the superclass

//in case of success, add some code...
return self;
```

Since code redundancy is a bad practice, it seems heavy-handed to repeat that scheme in each possible initializer. The best solution is to polish up the most essential initializer with this code. Then the other initializers will call this "preferred" initializer, known as the *designated initializer*. Logically, the designated initializer is the one with the greatest number of parameters, since it is not possible in Objective-C to give default values to these parameters.

```
-(id) initWithX:(int)x
{
  return [self initWithX:x andY:0 andZ:0];
}

-(id) initWithX:(int)x andY:(int)y
{
  return [self initWithX:x andY:y andZ:0];
}

//designated initializer
-(id) initWithX:(int)x andY:(int)y andZ:(int)z
{
  if (!(self = [super init]))
    return nil;
  self->x = x;
  self->y = y;
  self->z = z;
  return self;
}
```

If the designated initializer is not the one with the greatest number of parameters, then it's not very useful:

```
//the following code is not handy
-(id) initWithX:(int)x //designated initializer
{
  if (!(self = [super init]))
    return nil;
  self->x = x;
  return self;
}
-(id) initWithX:(int)x andY:(int)y
{
  if (![self initWithX:x])
    return nil;
  self->y = y;
  return self;
}
-(id) initWithX:(int)x andY:(int)y andZ:(int)z
{
  if (![self initWithX:x])
    return nil;
  self->y = y;
  self->z = z;
  return self;
}
```

5.1.8 List of initialization and default value of instance data

The idea of *list of initialization* for C++ constructors does not exist in Objective-C. However, it is remarkable that, unlike the C++ behaviour, the `alloc` method of Objective-C initializes all the bits of instance data to 0, so the pointers are set to `nil`. This would be a problem in C++ for objects as attributes, but in Objective-C, the objects are always represented by pointers.

5.1.9 Virtual constructor

It is possible in Objective-C to get real virtual constructors. For more details, please see Section 6.4.6 on page 42, after the introduction to the memory management (Section 6 on page 39).

5.1.10 Class constructors

In Objective-C, since the classes are themselves objects, they supply a constructor that can be redefined. it is obvioulsy a class method, inherited from NSObject; its prototype is `+(void) initialize;`

This method is automatically called when using the class or one of its sub-classes for the first time. However, it is not true that this method is called only once for a given class; indeed, if a sub-class does not redefine `+(void) initialize`, the Objective-C mechanism calls `+(void) initialize` from the mother class.

5.2 Destructors

In C++, the destructor, like the constructor, is a specific method that can be redefined. In Objective-C, it is an instance method named `dealloc`.

In C++, the destructor is called automatically when an object is freed; it is the same with Objective-C; only the way to release the object is different (cf. Section 6 on page 39).

The destructor should never be called explicitly. In fact, in C++ there is one case where it can be : when the developer himself is managing the memory pool used for the allocation. But in Objective-C, no case justifies an explicit call to `dealloc`. One can use custom memory zones in Cocoa, but their use does not influence common allocation/deallocation practices (cf. Section 5.3 on the following page).

C++

```
class Point2D
{
  public:
    ~Point2D();
};

Point2D::~Point2D() {}
```

Objective-C

```
@interface Point2D : NSObject
-(void) dealloc; //this method can be redefined
@end

@implementation Point2D
//in this example the redefinition was not necessary
-(void) dealloc
{
  [super dealloc]; //do not forget to transmit to the superclass
}
@end
```

5.3 Copy operators

5.3.1 Classical cloning, `copy`, `copyWithZone:`, `NSCopyObject()`

In C++, it is important to define a coherent implementation of the copy constructor and the affectation operator. In Objective-C, operator overloading is impossible; one must only ensure that the cloning method is correct.

Cloning in Cocoa is associated with a protocol (cf. Section 4.4 on page 22) named `NSCopying`, requesting the implementation of the method

```
-(id) copyWithZone:(NSZone*)zone;
```

Its argument is a memory zone in which the clone should be allocated. Cocoa allows the use of different custom zones: some methods take such a zone as an argument. Most of the times, the default zone is perfect, and there is no need to specify it each time. Fortunately, `NSObject` supplies a method

```
-(id) copy;
```

which encapsulates a call to `copyWithZone:` with the default zone as parameter. But it is the `copyWithZone:` version which is required by `NSCopying`. Finally, the utility function `NSCopyObject()` provides a slightly different approach, that can be simpler but also requires caution. First, the code is presented without taking `NSCopyObject(...)` into account, for which explanations are given in section 5.3.2 on the following page.

The implementation of `copyWithZone:` for some Foo class would look like:

```
//if the superclass does not implement copyWithZone:, and that NSCopyObject()
//is not used
-(id) copyWithZone:(NSZone*)zone
{
  //a new object must be created
  Foo* clone = [[Foo allocWithZone:zone] init];
  //instance data must be manually copied
  clone->integer = self->integer; //"integer" is here of type "int"
  //trigger the same mechanism for sub-objects to clone
  clone->objectToClone = [self->objectToClone copyWithZone:zone];
  //some sub-objects may not be cloned but shared
  clone->objectToShare = [self->objectToShare retain]; //cf. memory management
  //if there is a mutator, it can be used in both cases
  [clone setObject:self->object];
  return clone;
}
```

Note the use of `allocWithZone:`, instead of `alloc`, to handle the zone parameter. `alloc` encapsulates a call to `allocWithZone:` with the default zone. To learn more about Cocoa's custom zone management, refer to the Cocoa documentation.

However, one must take care of the possible implementation of `copyWithZone:` in the superclass.

```
//if the superclass implements copyWithZone:, and that NSCopyObject() is not used
-(id) copyWithZone:(NSZone*)zone
{
  Foo* clone = [super copyWithZone:zone]; //creates the new object
  //you must clone instance data specific to the current sub-class
  clone->integer = self->integer; //"integer" is here of type "int"
  //trigger the same mechanism for sub-objects to clone
  clone->objectToClone = [self->objectToClone copyWithZone:zone];
  //some sub-objects may not be cloned but shared
  clone->objectToShare = [self->objectToShare retain]; //cf. memory management
  //if there is a mutator, it can be used in both cases
  [clone setObject:self->object];
  return clone;
}
```

5.3.2 NSCopyObject()

The NSObject class does not implement the NSCopying protocol, that is why a direct sub-class cannot benefit from a call to [super copy...], and must use a standard initialization based on the [[... alloc] init] scheme.

The utility function NSCopyObject() can be used to write simpler code, but requires some caution with pointer members (including objects). This function creates a binary copy of an object and its prototype is :

```
//the extra bytes is usually 0, but can be used for the extra space of
//indexed instance data
id <NSObject> NSCopyObject(id <NSObject> anObject,
                           unsigned int extraBytes, NSZone *zone)
```

The binary copy can automate the copy of instance data that are non-pointers; but for a pointer member (including objects), one must keep in mind that it silently creates an additional reference to the pointed data. The usual practice is to reset the pointers afterwards with values coming from correct clonings.

```
//if the superclass does not implement copyWithZone:
-(id) copyWithZone:(NSZone*)zone
{
  Foo* clone = NSCopyObject(self, 0, zone);//binary copy of data
  //clone->integer = self->integer; //useless : binary data already copied

  //a sub-object to clone must be really cloned
  clone->objectToClone = [self->objectToClone copyWithZone:zone];
  //a sub-object to share must only register the new reference
  [clone->objectToShare retain]; //cf. memory management

  //The mutator is likely to release clone->object. This is undesirable,
  //because of the binary copy of the pointer value
  //Therefore, before using the mutator, the pointer is reset
  clone->object = nil;
  [clone setObject:self->object];
  return clone;
}
```

```
//if the superclass implements copyWithZone:
-(id) copyWithZone:(NSZone*)zone
{
  Foo* clone = [super copyWithZone:zone];
  //does a superclass implement NSCopyObject() ? This is important
  //to know what remains to be done

  clone->integer = self->integer;//only if NSCopyObject() has not been used
  //In case of doubt, it can be done systematically

  //a sub-object to clone must still be really cloned
  clone->objectToClone = [self->objectToClone copyWithZone:zone];

  //NSCopyObject() or not, a retain must be done (cf. memory management)
  clone->objectToShare = [self->objectToShare retain];

  clone->object = nil; //in case of doubt, it's better to reset
  [clone setObject:self->object];

  return clone;
}
```

5.3.3 Dummy-cloning, mutability, `mutableCopy` and `mutableCopyWithZone:`

When cloning an object that cannot change, a fundamental optimization is to pretend that it is cloned; instead of duplicating it, a reference to it can be returned. Starting from that, we can distinguish the notion of an *immutable* and *mutable* object.

An immutable object cannot have any of its instance data changed; only the initializer gives it a valid state. In this case, it can safely be "pseudo-cloned" by returning only a reference to itself. Since neither it nor its clone can be modified, none of them can be undesirably affected by a modification of the other. A very efficient `copyWithZone:` implementation can be proposed in that case:

```
-(id) copyWithZone:(NSZone*)zone
{
  //the object returns itself, counting one more reference
  return [self retain]; //see the section about memory management
}
```

The use of `retain` comes from the memory management in Objective-C (cf. Section 6 on page 39). The reference counter is incremented by 1 to make the existence of the clone "official" so that deleting that clone will not destroy the original object.

"Pseudo-cloning" is not a marginal optimization. Creating an object requests a memory allocation, which is a "long" process that should be avoided when possible. That is why it is interesting to identify two kind of objects: immutable ones, for which cloning can be fictive, and the others. Making the distinction is as simple as creating "immutable" classes and optionally subclassing them into "mutable" versions, adding methods to change their data instance. For instance, in Cocoa, `NSMutableString` is a subclass of `NSString`, `NSMutableArray` is a subclass of `NSArray`, `NSMutableData` is a subclass of `NSData`, and so on.

However, with the techniques presented here, it seems impossible to get a real clone, safely mutable, from an immutable object which would only know how to "pseudo-clone" itself. Such a limitation would significantly decrease the usefulness of immutable objects, isolating them from the "external world".

In addition to the `NSCopying` protocol, there is another protocol (cf. Section 4.4 on page 22) named `NSMutableCopying`, requesting the implementation of

```
-(id) mutableCopyWithZone:(NSZone*)zone;
```

The `mutableCopyWithZone:` method must return a mutable clone, where modifications would not apply to the original object. Similarly to the `copy` method, there is a `mutableCopy` method which automatically calls `mutableCopyWithZone:` by giving it the default zone. The implementation of `mutableCopyWithZone:` looks like the following, similar to the classical copy previously presented:

```
//if the superclass does not implement mutableCopyWithZone:
-(id) mutableCopyWithZone:(NSZone*)zone
{
  Foo* clone = [[Foo allocWithZone:zone] init]; //or NSCopyObject() if possible
  clone->integer = self->integer;
  //Like with copyWithZone:, some sub-objects can be cloned, some others shared
  //A mutable sub-object can be cloned with a call to mutableCopyWithZone:
  //...
  return clone;
}
```

Do not forget to use the possible `mutableCopyWithZone:` of the superclass:

```
//if the superclass implements mutableCopyWithZone:
-(id) mutableCopyWithZone:(NSZone*)zone
{
  Foo* clone = [super mutableCopyWithZone:zone];
  //...
  return clone;
}
```

6 Memory management

6.1 `new` and `delete`

The C++ keywords `new` and `delete` do not exist in Objective-C (`new` exists as a method, but it is just a deprecated shortcut for `alloc`+`init`). They are respectively replaced by calls to `alloc` (cf. Section 5.1 on page 27) and `release` (cf. Section 6.2 on the current page).

6.2 Reference counting

Memory management in Objective-C is one of the most important parts of the language. In C or C++, a memory area is allocated once and freed once. It can be referenced with as many pointers as desired, but only one pointer will be given the `delete` call.

On the other hand, Objective-C implements a reference counting scheme. An object knows how many times it is referenced. This can be explained by the analogy of dogs and leashes (an analogy directly taken from *Cocoa Programming for MacOS X* [7]). If an object is a dog, everyone can ask for a leash to hold it. If someones does not care about the dog anymore, it can drop its leash. While the dog has at least one leash, it must stay there. But as soon as the number of leashes falls to 0, the dog is free!

More technically, the reference counter of a newly created object is set to 1. If a part of the code needs to reference that object, it can send it a `retain` message, which will increase the counter by one. When a part of the code does not need the object any more, it can send it a `release` message, that will decrease the counter by 1.

An object can receive as many `retain` and `release` messages as needed, as long as the reference counter has a positive value. As soon as it falls to 0, the destructor `dealloc` is automatically called. Sending `release` again to the address of the object, which is now invalid, triggers a memory fault.

This technique is not equivalent to the `auto_ptr` from the C++ STL. On the contrary, the Boost library [5] supplies an encapsulation of pointers into a `shared_ptr` class, which implements the reference counting scheme. But it is not part of the standard library.

6.3 `alloc`, `copy`, `mutableCopy`, `retain`, `release`

The understanding of memory management does not fully explain how it is used. The goal of this section is to give some rules. The keyword `autorelease` is left aside for now, since it is more difficult to understand.

The basic rule to apply is **Everything that increases the reference counter with `alloc`, `[mutable]copy[WithZone:]` or `retain` is in charge of the corresponding `[auto]release`**. Indeed, these are the three ways to increment the reference counter. It also means that you should take care of releasing an object in only a limited number of cases:

- when you explicitly instantiate an object with `alloc`;
- when you explicitly clone the object with `copy[WithZone:]` or `mutableCopy[WithZone:]` (whatever the copy is : a real or a pseudo-clone. This should not have any importance, cf. section 5.3.3 on page 37);
- when you explicitly use `retain`.

Please remember that by default, it is legal to send a message (like `release`) to `nil`, without any consequence (see Section 3.4.1 on page 19).

6.4 `autorelease`

6.4.1 Precious `autorelease`

The rule stated in the previous section is so important that it bears repeating: **Everything that increases the reference counter with `alloc`, `[mutable]copy[WithZone:]` or `retain` is in charge of the corresponding `[auto]release`**.

In fact, with the mere `alloc`, `retain` and `release`, this rule could not be applied. Indeed, there are some methods that are not constructors, but that are designed to create objects : for instance a binary addition operator in C++ (`obj3 operator+(obj1, obj2)`). In C++, the returned object would be allocated on the stack and automatically destroyed when leaving its scope. But in Objective-C, such objects do not exist. The function has necessarily used `alloc`, but cannot release the object before returning it on the stack ! Here are given some illustrations of what is going wrong:

```
-(Point2D*) add:(Point2D*)p1 and:(Point2D*)p2
{
  Point2D* result = [[Point2D alloc] initWithX:([p1 getX] + [p2 getX])
                                          andY:([p1 getY] + [p2 getY])];
  return result;
}
//ERROR : the function performs "alloc", so, it is creating
//an object with a reference counter of 1. According
//to the rule, it should destroy the object.

//This can lead to a memory leak when summing three points :
[calculator add:[calculator add:p1 and:p2] and:p3];

//The result of the first addition is anonymous
//and nobody can release it. It is a memory leak.
```

```
-(Point2D*) add:(Point2D*)p1 and:(Point2D*)p2
{
  return [[Point2D alloc] initWithX:([p1 getX] + [p2 getX])
                               andY:([p1 getY] + [p2 getY])];
}
//ERROR : This is exactly the same code as above. The fact that
//no intermediate variable is used does not change anything.
```

```
-(Point2D*) add:(Point2D*)p1 and:(Point2D*)p2
{
  Point2D* result = [[Point2D alloc] initWithX:([p1 getX] + [p2 getX])
                                          andY:([p1 getY] + [p2 getY])];
  [result release];
  return result;
}
//ERROR : obviously, it is nonsense to destroy the object after creating it
```

The problem seems intractable. It would be, if `autorelease` was not there. To simplify, let us say that sending `autorelease` to an object means that it is sent a `release` that will be performed "later". But "later" does not mean "at any time"; this is detailed in Section 6.4.2 on the following page. For a first approach, here is the only possible solution:

```
-(Point2D*) add:(Point2D*)p1 and:(Point2D*)p2
{
  Point2D* result = [[Point2D alloc] initWithX:([p1 getX] + [p2 getX])
                                          andY:([p1 getY] + [p2 getY])];
  [result autorelease];
  return result;        //a shorter writing is "return [result autorelease]"
}
//CORRECT : "result" will be automatically released later,
//after being used in the calling code
```

6.4.2 The autorelease pool

In the previous section, `autorelease` has been presented as a kind of magical `release` that is automatically applied at the right moment. But it would make no sense to let the compiler guess what the right moment is. In this case, a garbage collector would be more useful. To explain how it works, more details must be given about `autorelease`.

Each time an object receives `autorelease`, it is only registered into an "autorelease pool". When the pool is destroyed, the object receives a real `release`. The problem has moved : how is this pool handled ?

There is not a single answer : if you use Cocoa for an application with a graphical interface, most of the time there is nothing to do. Otherwise, you would have to create and destroy the pool yourself.

An application with a graphical interface generally uses an *event loop*. Such a loop waits for actions from the user, then wakes up the program to perform the action, then goes back to sleep until the next event. When you create a graphical application with Cocoa, an autorelease pool is automatically created at the beginning of the loop and destroyed at the end. This is logical: generally, a user's action triggers a cascade of tasks. Temporary objects are created, then destroyed, since they do not have to be kept for the next event. If some of them must be persistent, the developer must use `retain` as necessary.

On the other hand, when there is no graphical interface, you have to create an autorelease pool around the code that is needing it. When an object receives `autorelease`, it knows how to find the closest autorelease pool. Then, when it is time to empty the pool, you can just destroy it with a simple `release`. Typically, a command-line Cocoa program contains the following code:

```
int main(int argc, char* argv[])
{
  NSAutoreleasePool* pool = [[NSAutoreleasePool alloc] init];
  //...
  [pool release];
  return 0;
}
```

Please note that MacOS X10.5 has added the `drain` method to the class `NSAutoreleasePool`. This metid is equivalent to `release` if the garbage collector is enabled, and triggers a run of garbage collection otherwise (cf. 6.6 on page 48). This is useful to write code which behaves the same in both cases.

6.4.3 Using several autorelease pools

It is possible, and sometimes useful, to have more than one autorelease pool in the program. An object that receives `autorelease` will register in the closest pool. Therefore, if a function creates and uses a large number of temporary objects, increased performance can be obtained by creating a local autorelease pool. That way, the crowd of temporary objects will be destroyed as soon as possible and will not clutter the memory after the function has returned.

6.4.4 Caution with `autorelease`

The fact that `autorelease` is handy should not lead to its misuse.

- First, sending more `autorelease` calls than necessary is similar to sending too many `release` calls: it triggers a memory fault when emptying the pool;
- Then, even if it is true that any `release` message can be replaced by `autorelease`, this would be a performance hit, since the autorelease pool is more work than a normal `release`. Moreover, delaying all deallocations can lead to useless and irrelevant peaks of memory consumption.

6.4.5 `autorelease` and `retain`

Thanks to `autorelease`, a method can create an object that can plan its own releasing. However, it is common that the object must be kept longer. In this case, one has to send a `retain` on it, and plan for a release. Then there are two points of view regarding the lifetime of this object :

- from the point of view of the developer of the function, the object is created and its release is planned;
- from the point of view of the caller of the function, the lifetime is increased with the `retain` (the `autorelease` planned by the function will not decrease the reference counter down to 0), but since there was a 1 increment, te caller is now responsible for releasing later.

6.4.6 Convenience constructor, virtual constructor

The successive applications of `alloc` and `init` can sound laborious in some cases. Fortunately, it can be shortened by the notion of *convenience constructor*. Such a constructor, the name of which should be prefixed by the name of the class, behaves like an `init` method, but it performs the `alloc` itself. However, the returned object is registered inside an autorelease pool, and will be temporary if it is not sent any `retain`. For example:

```
//laborious
NSNumber* zero_a = [[NSNumber alloc] initWithFloat:0.0f];
...
[zero_a release];

...

//handier
NSNumber* zero_b = [NSNumber numberWithFloat:0.0f];
...
//no need of release
```

With help from the section about memory management (Section 6 on page 39), it's obvious that such a constructor relies on `autorelease`. The underlying code is not that obvious anyway, since it requires the correct use of `self`. Indeed, a convenience constructor is a *class method*, so that `self` refers to an object of type `Class`, which is a *meta-class* instance. In an *initializer*, which is an *instance* method, `self` is an *instance* of the class, and refers to a "normal" object.

It is easy to write a bad convenience constructor. Let us assume a class `Vehicle` holding a color, and supplying a convenience constructor.

```
//The Vehicle class
@interface Vehicle : NSObject
{
  NSColor* color;
}
-(void) setColor:(NSColor*)color;

//convenience constructor
+(id) vehicleWithColor:(NSColor*)color;
@end
```

The implementation of the convenience constructor is somewhat subtle.

```
//bad convenience constructor
+(Vehicle*) vehicleWithColor:(NSColor*)color
{
  //the value of "self" should not change here
  self = [[self alloc] init]; // ERROR !
  [self setColor:color];
  return [self autorelease];
}
```

`self` in this class method refers to the *class*. It cannot be set to be the *instance*.

```
//Almost perfect constructor
+(id) vehicleWithColor:(NSColor*)color
{
  id newInstance = [[Vehicle alloc] init]; // OK, but ignores potential
                                           // sub-classes
  [newInstance setColor:color];
  return [newInstance autorelease];
}
```

We can still improve this. In Objective-C one can get the behaviour of a virtual constructor. The constructor just needs to perform an introspection to know which is the real class of the object performing the method. Then it can directly produce an object of the right subclass. The false keyword `class` can be used; this is a method of `NSObject` that returns the current object's class object (a meta-class instance).

```
@implementation Vehicle
+(id) vehicleWithColor:(NSColor*)color
{
  id newInstance = [[[self class] alloc] init]; // PERFECT, the class is
                                                // dynamically identified
  [newInstance setColor:color];
  return [newInstance autorelease];
}
@end

@interface Car : Vehicle {...}
@end

...

//produces a (red) car
id car = [Car vehicleWithColor:[NSColor redColor]];
```

Similar to the rule about the **init** prefix for initializers, you are strongly encouraged to prefix a convenience constructor with the name of the class. There are very few cases where this is not the case, like `[NSColor redColor]` in the previous code, which should have been written `[NSColor colorRed]`.

Finally, let us repeat the rule: **Everything that increases the reference counter with `alloc`, `[mutable]copy[WithZone:]` or `retain` is in charge of the corresponding `[auto]release`.** When **calling** a convenience constructor, you are not explicitly calling `alloc`, so you are not in charge of the `release`. However, when `creating` such a constructor, you are writing `alloc`, and you must not forget the `autorelease`.

6.4.7 Setter

A setter (mutating accessor) is a typical example of something that is difficult to write without the knowledge of memory management in Objective-C. Let us assume a class encapsulating an `NSString` named *title*, and let us suppose that we want to change the value of this string. This very simple example raises the main problem related to setters : how is the parameter supposd to be used ? Unlike C++, only one prototype is legible (an object can only be used through pointers), but several implementations can be found. It can be an *assignation*, an *assignation with retain*, or a `copy`. Each one has a specific meaning regarding the data model chosen by the developer. Moreover, in each case, the old resources must be released first to avoid a memory leak.

assignation (incomplete code)
The outer object is just referenced weakly, without any `retain`. If the outer object is modified, it is visible from the current class. If the outer object happened to be deallocated without the present reference being set to `nil` before, it would be an invalid reference.

```
-(void) setString:(NSString*)newString
{
   ... memory management to be detailed later
  self->string = newString; //assignation
}
```

assignation with retain (incomplete code)
The outer object is referenced, and the reference counter is increased by 1 thanks to a `retain`. If the outer object is modified, this is visible from the current class. The outer object cannot be deallocated as long as the current reference is not released.

```
-(void) setString:(NSString*)newString
{
   ... memory management to be detailed later
  self-> string = [newString retain]; //assignation with retain
}
```

copy (incomplete code)
The outer object is not references : a clone is created instead. If the oter object is modified, this is not visible on the clone. Logically, the clone is handled by the current owner object, and its lifetime should not be greater.

```
-(void) setString:(NSString*)newString
{
   ... memory management to be detailed later
  self->string = [newString copy]; //cloning;
                                   //the NSCopying protocol is used
}
```

To complete the code, the previous state of the object should be considered : in each case, the mutator would have to release the old reference (if any) before setting a new one. This part of the code is tricky.

assignation (complete code)
The simplest case. The old reference can be overwritten.

```
-(void) setString:(NSString*)newString
{
  //no strong link : the old reference can be overwritten
  self->string = newString; //assignation

}
```

assignation with retain (complete code)
In this case, the old reference should be released, unless it is the same as the new one.

```
//Bad codes

-(void) setString:(NSString*)newString
{
  self->string = [newString retain];
  //ERROR : memoy leak : the old "string" is no more referenced
}

-(void) setString:(NSString*)newString
{
  [self->string release];
  self->string = [newString retain];
  //ERROR : if newString == string,  (it can happen)
  //and that the reference counter of newString was 1,
  //the it is invalid to use newString (string) after
  //[self->string release], because it has been dealocated at this point
}

-(void) setString:(NSString*)newString
{
  if (self->string != newString)
    [self->string release]; //ok: it is safe to send release even to nil
  self->string = [newString retain];  //ERROR : should be in the "if";
  //because if string == newString,
  //the counter should not be incremented
}
```

```
//Correct codes

//Practice "Check before change"
//the most intuitive for a C++ developer
-(void) setString:(NSString*)newString
{
  //avoid degenerated case where there is nothing to do
  if (self->string != newString)
  {
    [self->string release]; //release the old one
    self->string = [newString retain]; //retain the new one
  }
}

//Practice "Autorelease the old value"
-(void) setString:(NSString*)newString
{
  [self->string autorelease]; //even if string == newString,
                        //it's correct, because release is delayed
  self->string = [newString retain];
   //... and thus this retain happens before
}

//Practice "retain, then release"
-(void) setString:(NSString*)newString
{
  [self->newString retain]; //the reference counter is increased by 1 (except on nil)
  [self->string release]; //...so that is does not reach 0 here
  self->string = newString; //but no "retain" is added here !
}
```

copy (complete code)
Regarding the typical errors or the good solutions, this case is almost identical to the assignation with retain, where `retain` is replaced by `copy`.

pseudo-clonage
Please note that the copy can be a "dummy-cloning" under the hood (cf. section 5.3.3 on page 37), without any consequences.

6.4.8 Getters

With the Objective-C language, all objects are dynamically allocated. They are referenced and encapsulated as pointers. Typically, getters are only returning the pointer's values, and do not copy the object on the fly. The name of a getter is usually the same as the data member, this is possible in Objective-C and does not create a conflict. In the case of a boolean value, the name might begin with an `is`, in order to be read as a predicate.

```
@interface Button
{
  NSString* label;
  BOOL      pressed;
}
-(NSString*) label;
-(void) setLabel:(NSString*)newLabel;
-(BOOL) isPressed;
@end

@implementation Button
-(NSString*) label
{
  return label;
}

-(BOOL) isPressed
{
  return pressed;
}

-(void) setLabel:(NSString*)newLabel {...}
@end
```

When returning the pointers of instance data, it is easy to modify them if they are mutable. It can be undesirable to allow such modifications to outer objects, since data encapsulation should not be infringed.

```
@interface Button
{
  NSMutableString* label;
}
-(NSString*) label;
@end

@implementation Button
-(NSString*) label
{
  return label; //OK, but a well-informed user could downcast
  //the result to NSMutableString, and thus modify the string
}

-(NSString*) label
{
  //solution 1 :
  return [NSString stringWithString:label];
  //OK : a new, immutable string, is returned

  //solution 2 :
  return [[label copy] autorelease];
  //OK, using copy (and not mutableCopy) on an NSMutableString will return
  //an NSString
}
@end
```

6.5 Retain cycles

A retain cycle is something that must be avoided. If an object A retains an object B, and that B and C are mutually retaining each other, B and C are forming a retain cycle.

$$A \rightarrow B \rightleftarrows C$$

If A releases B, B will not be deallocated since it is still retained by C. And C cannot be deallocated since it is retained by B. However, A was the only reference to that cycle, so it is no more eachable. Retain cycles usually lead to memory leaks. This is why, in a tree structure for instance, a node usually retains its children, but a child usually does not retain its parent.

6.6 Garbage collector

Objective-C 2.0 (cf. Section 1.2 on page 6) implements a garbage collector. In other words, you can delegate all the memory management and do not care about *retain* and *release* any more. A great choice of Objective-C was to make the garbage collector an optional feature : you can decide if you want to precisely control the life-time of objects, or if you want to make less bug-prone code. The garbage collector is enabled or disabled for the whole program.

If the garbage collector is enabled, `retain`, `release` and `autorelease` are redefined to do nothing. Thus, code that was written without garbage collection can theoretically be recompiled easily for the garbage collector. "Theoretically" means that many subtleties should be handled regarding resource liberation. Thus, it is not easy to write unified code for both cases, and developers are even advised again that practice. Those subtleties are detailed in Apple's documentation [2]. The next four sections are quoting some of these difficulties, to highlight the most important points that should be properly studied.

6.6.1 `finalize`

In a garbage-collected environment, the non-deterministic order of the destruction of the objects is does not suit very well the use of `dealloc`. A `finalize` method has been added to `NSObject` to split the destruction into two steps : resources release and effective deallocation. But a "good" `finalize` method is subtle and tricky. Some conception constraints should be taken in account [2].

6.6.2 `weak`, `strong`

It is not very common to see *__weak* and *__strong* explicitely used in a declaration. However, their knowledge helps understanding some new difficulties related to the garbage collector.

By default, a pointer to an object is using the *__strong* attribute : this is a **strong** reference. It means that the object cannot be destroyed as long as this reference remains. This is the expected behaviour : when all (strong) references have disappeared, the object can be collected and released. In some cases, it is useful to disable that behaviour : some collections should not increase the lifetime of the objects they are holding, because it would prevent these object to be ever destroyed. In this case, these collections are using **weak** references, with the *__weak* keyword. `NSHashTable` is an example (cf. section 11.1 on page 54). A *__weak* reference is automaticaly nullified (set to `nil`) when the referenced object has disappeared.

A very relevant example is the *Notification Center* of Cocoa, which is out of the scope of pure Objective-C, and is not detailed in the present document.

6.6.3 NSMakeCollectable()

`Cocoa` is not the only API of MacOS X. `Core Foundation` is another one; they are compatible and can share data and objects, but `Core Foudation` is a procedural API written in C only. At first sight, the garbage collector cannot work with `Core Foundation` pointers. This problem has been addressed, and it is possible to use `Core Foundation` without memory leaks in a garbage-collected environment. The `NSMakeCollectable` documentation is a good starting point to know the tricks.

6.6.4 AutoZone

The Objective-C garbage collector created by Apple is called `AutoZone`. It has been publicly released as Open-Source [1]. Evolutions are planned for MacOS X10.6.

7 Exceptions

Exception handling in Objective-C is closer to Java than to C++, particularly because of the `@finally` keyword. `finally` is known in Java but not in C++. It is an additional (but optional) part of a `try()...catch()` block, containing code that will always be executed, whether an exception is caught or not. This is useful to write short and clean code that properly frees resources.

Apart from that, the behaviour of `@try...@catch...@finally` in Objective-C is very classical; however, only objects can be thrown (unlike C++). A simple example with and without `@finally` is shown.

Without finally	With finally

`BOOL problem = YES;`
`@try{`
`  dangerousAction();`
`  problem = NO;`
`}`
`@catch (MyException* e){`
`  doSomething();`
`  cleanup();`
`}`
`@catch (NSException* e){`
`  doSomethingElse();`
`  cleanup();`
`  //here is the exception re-thrown`
`  @throw`
`}`
`if (!problem)`
`  cleanup();`
	`@try{`
	`  dangerousAction();`
	`}`
	`@catch (MyException* e){`
	`  doSomething();`
	`}`
	`@catch (NSException* e){`
	`  doSomethingElse();`
	`  @throw //here is the exception re-thrown`
	`}`
	`@finally{`
	`  cleanup();`
	`}`

Strictly speaking, `@finally` is not essential, but is a useful tool for handling exceptions properly. As shown in the previous example, it is also handling well the case of a re-thrown exception in a `@catch`. In fact, the `@finally` is triggered when quitting the `@try` scope. An illustration is given below.

```
int f(void)
{
  printf("f: 1-you see me\n");
  //See the section about strings to understand the "@" syntax
  @throw [NSException exceptionWithName:@"panic"
                                 reason:@"you don't really want to known"
                                 userInfo:nil];
  printf("f: 2-you never see me\n");
}

int g(void)
{
  printf("g: 1-you see me\n");
  @try {
    f();
    printf("g: 2-you do not see me (in this example)\n");
  }
  @catch(NSException* e) {
    printf("g: 3-you see me\n");
    @throw;
    printf("g: 4-you never see me\n");
  }
  @finally {
    printf("g: 5-you see me\n");
  }
  printf("g: 6-you do not see me (in this example)\n");
}
```

A last point: the `catch(...)` of C++ (with suspension points), that can catch anything, does not exist in Objective-C. Indeed, since only objects can be thrown, it is always possible to catch them with the type `id` (cf. Section 3.1 on page 10).

Please note that an `NSException` class exists in Cocoa, and that you are encouraged to use it as mother class of what you throw. Thus, a `catch(NSException* e)` should be equivalent to `catch(...)`.

8 Multithreading

8.1 Thread-safety

It is perfectly legible for a program written in Objective-C to use POSIX APIs[2] in order to realize multi-threading. Cocoa provides its own classes to manage concurrent threads. the caveats are identical : one must take care that simultaneous access on several threads to some parts of the code will not lead to unpredictable results by operating on the same memory space.

The POSIX APIs, as well as Cocoa, are implementing lock and mutex objects. Objective-C is providing a keyword, `@synchronized`, equivalent to the identically-named Java keyword.

8.2 `@synchronized`

A block of code enclosed in a `@synchronized(...)` section is automatically locked to be used by one thread only at once. This is not always the best solution to manage concurrency, but this is a simple, light and concise method for most critical sections.

`@synchonized` requires an object as parameter (any object, for instance `self`) to be used as lock.

```
@implementation MyClass

-(void) criticalMethod:(id) anObject
{
  @synchronized(self)
  {
    //this part of the code is exclusive to any block @synchronized(self)
    //(with the same "self"...)
  }
  @synchronized(anObject)
  {
    //this part of the code is exclusive to any block @synchronized(anObject)
    //(with the same "anObject"...)
  }
}

@end
```

[2]Application Programming Interface

9 Strings in Objective-C

9.1 The only static objects in Objective-C

In the C language, strings are arrays of characters, or `char*` pointers. Handling such data is difficult and causes many bugs. The `string` class of C++ is a relief. In Objective-C, it has been explained in Section 5 on page 27 that objects cannnot be automatic and must be allocated at run-time. This is rather incompatible with the use of static strings. This could lead to a situation where static C strings are used as construction parameters of `NSString` objects; this would be unhandy and a waste of memory.

Fortunately, static Objective-C strings do exist. These are simple C strings between quotation marks, with an additional `@` prefix.

```
NSString* notHandy = [[NSString alloc] initWithUTF8String:"helloWorld"];
NSString* stillNotHandy = //initWithFormat is a kind of sprintf()
   [[NSString alloc] initWithFormat:@"%s", "helloWorld"];
NSString* handy = @"hello world";
```

Moreover, a static string can be sent methods like a regular object.

```
int size = [@"hello" length];
NSString* uppercaseHello = [@"hello" uppercaseString];
```

9.2 `NSString` and encodings

NSString objects are very useful, since in addition to a large number of handy methods, they also support different encodings (ASCII, Unicode, ISO Latin 1...). Translation and localization of applications using such strings is really simple.

9.3 Description of an object, `%@` format extension, NSString to C string

In Java, every object inherits from `Object` and benefits from a `toString` method, that can describe the object as a string, which is useful for debugging. In Objective-C, this method is named `description` and returns an `NSString`.

The `printf` function of the C language has not been extended to support `NSString` objects. `NSLog` can be used instead, or any other function that accepts a formatted-string like `printf`. For an `NSString` object, the format to use is not "%s" but "%@". And more generally, "%@" can be used on any object because it is in fact using the returned value of a call to `-(NSString*) description`.

Moreover, an `NSString` can be converted into a C-like string with the `UTF8String` method (previously `cString`).

```
char* name = "Spot";
NSString* action1 = @"running";
printf("My name is %s, I like %s, and %s...\n",
       name, [action1 UTF8String], [@"running again" UTF8String]);
NSLog(@"My name is %s, I like %@ and %@\n",
        name, action1, @"running again");
```

10 C++ specific features

So far, you've seen that object-oriented concepts of C++ are present in Objective-C. However, some other features are not. It does not concern object-oriented concepts, but some coding facilities.

10.1 References

References (&) do not exist in Objective-C. Memory management using reference counting and `autorelease` makes it less useful. Since the objects are always dynamically created, they are only referenced as pointers.

10.2 Inlining

Objective-C does not implement inlining. This is rather logical for the methods, because the dynamism of Objective-C cannot afford to "freeze" some code. However, inlining would be very useful for utility functions written in simple C, like `max()`, `min()`... This is a problem that Objective-C++ can address.

Nevertheless, note that the GCC compiler supplies the non-standard keyword `__inline` or `__inline__` to use inlining in C, and so in Objective-C. Moreover, GCC is also able to compile C99 code, a revision of C that supports inlining with the keyword `inline` (standard in this case). Thus, Objective-C based on C99 code benefits from inlining.

This is not inlining, but for performance, you can also consider using IMP-caching (cf. section 11.3.2 on page 55).

10.3 Templates

Templates are a kind of alternative to inheritance and virtual methods, designed for efficiency, but clearly far from pure object model principles. (Did you know that a subtle use of templates can give `public` access to `private` members?). Objective-C does not implement templates, and it would be very difficult because of the rules about method names overloading and selectors.

10.4 Operators overloading

Objective-C does not implement operator overloading.

10.5 Friends

There is no *friend* notion in Objective-C. Indeed, this is rather useful in C++ for the efficiency of overloaded operators, that are not present in Objective-C. The notion of packages in Java, that is somewhat close to friends, can be addressed most of the time in Objective-C with class categories (cf. Section 4.5 on page 25).

10.6 `const` methods

Methods cannot be declared `const` in Objective-C. Therefore, the `mutable` keyword does not exist.

10.7 List of initialization in the constructor

Lists of initialization at construction does not exist in Objective-C.

11 STL and Cocoa

The standard C++ library is one of its strengths. Even if it has some gaps (for instance in the functors; gaps usually filled with the SGI STL [8]), it is very complete. This is not a real part of the language, since it is only an extension, but it is common to find such equivalents in other languages. In Objective-C, you have to look at Cocoa to find containers, iterators and other ready-to-use algorithms.

11.1 Containers

Obviously, Cocoa does things in the object-oriented way: a container is not a template, it can only contain objects. At the time of writing, the available containers are:

- `NSArray` and `NSMutableArray` for ordered collections;
- `NSSet` and `NSMutableSet` for unordered collections;
- `NSDictionary` and `NSMutableDictionary` for key/value associating collections;
- `NSHashTable` as hash table using weak references (Objective-C 2.0 only, see below).

You might notice the lack of `NSList` or `NSQueue`. Indeed, it seems that these concepts should be implemented as mere `NSArray` objects.

Unlike a `vector<T>` in C++, an `NSArray` of Objective-C really hides its internal implementation and only exposes its content through the accessors. Therefore, an `NSArray` has no obligation to organize its content as contiguous memory cells. The implementors of `NSArray` have certainly chosen a compromise, letting `NSArray` be used efficiently like an array or like a list. Since in Objective-C a container can only hold pointers to objects, the manipulation of the cells can be very efficient.

`NSHashTable` is equivalent to `NSSet`, but is using weak references (cf. section 6.6.2 on page 48), which is useful in a garbage collected environment (section 6.6 on page 48).

11.2 Iterators

11.2.1 Classical enumeration

The pure object-oriented approach makes the notion of iterator more flexible in Objective-C than in C++. The `NSEnumerator` is designed so that:

```
NSArray* array = [NSArray arrayWithObjects:object1, object2, object3, nil];
NSEnumerator* enumerator = [array objectEnumerator];
NSString* aString = @"foo";
id anObject = [enumerator nextObject];
while (anObject != nil)
{
  [anObject doSomethingWithString:aString];
  anObject = [enumerator nextObject];
}
```

A method of the container (`objectEnumerator`) returns an iterator, which is then able to move by itself (`nextObject`). The behaviour is closer to Java than to C++. When the iterator reaches the end of the container, `nextObject` returns `nil`.

There is a more common syntax to use iterators, that is based only on the C ability to shorten things.

```
NSArray* array = [NSArray arrayWithObjects:object1, object2, object3, nil];
NSEnumerator* enumerator = [array objectEnumerator];
NSString* aString = @"foo";
id anObject = nil;
while ((anObject = [enumerator nextObject])) {
  [anObject doSomethingWithString:aString];
}
//double parenthesis suppress a gcc warning here
```

11.2.2 Fast enumeration

Objective-C 2.0 (cf. Section 1.2 on page 6) has introduced a new syntax to enumerate a container, so that the `NSEnumerator` is implicit (besides, it is no more an `NSEnumerator`). The syntax is of the form:

```
NSArray* someContainer = ...;
for(id object in someContainer) { //here, each object is typed "id"
  ...
}
for(NSString* object in someContainer) { //here, each object is typed "NSString*"
  ...//if an object is not an NSString*, it's up to the developer to handle that
}
```

11.3 Functors (function objects)

11.3.1 Using selectors

The power of the `selector` in Objective-C makes functors less useful. Indeed, weak typing allows the user to send a message without really taking care of the ability of the receiver to handle it. For instance, here is a code equivalent to the previous one using iterators:

```
NSArray* array = [NSArray arrayWithObjects:object1, object2, object3, nil];
NSString* aString = @"foo";
[array makeObjectsPerformSelector:@selector(doSomethingWithString:)
                       withObject:aString];
```

in this case, the objects don't have to be of the same kind of class, and they don't even have to implement a `doSomethingWithString:` method (it would just raise an exception "selector not recognized") !

11.3.2 IMP caching

This won't be detailed here, but it is possible to get the address of a C function which represents a method into memory. This can be used to optimize multiple calls to the same `selector` on a bunch of objects, by doing the method lookup only once. This is called "IMP caching" since `IMP` is in Objective-C the data type of a method implementation.

A call to `class_getMethodImplementation()` (cf. section 13.2 on page 63) may be used for instance to get such a pointer. Please note, however, that it is a real pointer to the implemented method : it will not resolve virtual calls. Its use is most of the time related to optimization and must be done with care.

11.4 Algorithms

The large set of generic algorithms of the STL really have no equivalent in Cocoa. Instead, you should look at the methods supplied by each container.

12 Implicit code

This section gathers two features that allow code simplification. Their goal are different : the *Key-value coding* (section 12.1) can resolve an indirect method call by selecting the first valid matching implementation, while properties (cf. section 12.2 on page 58) can let the compiler generate some "glue", boring code.

The *Key-value coding* is in fact a facility offered by Cocoa, while the notion of *property* is a part of the language itself, added in Objective-C 2.0.

12.1 Key-value coding

12.1.1 Principle

The *Key-value coding* is the name given to the practice of accessing the value of a data member by its name. This is kind of similar to the an associative array (`NSDictionary`, cf. section 11.1 on page 54), where the name of the data member is the key. The class `NSObject` provides methods entitled `valueForKey:` and `setValue:forKey:`. If the data members are objects themselves, the exploration can be done in depth; in this case the key must be a "keypath", the components being dot-separated. The methods to use are `valueForKeyPath:` and `setValue:forKeyPath:`.

```
@interface A {
  NSString* foo;
}

... //some methods must be implemented for that code to be complete

@end

@interface B {
  NSString* bar;
  A* myA;
}

... //some methods must be implemented for that code to be complete

@end

@implementation B
...
//Let us assume an object a of type A and an object b of type B
B* a = ...;
B* b = ...;
NSString* s1 = [a valueForKey:@"foo"]; //ok
NSString* s2 = [b valueForKey:@"bar"]; //ok
NSString* s3 = [b valueForKey:@"myA"]; //ok
NSString* s4 = [b valueForKeyPath:@"myA.foo"]; //ok
NSString* s5 = [b valueForKey:@"myA.foo"]; //erreur !
NSString* s6 = [b valueForKeyPath:@"bar"]; //ok, why not
...
@end
```

Thanks to that syntax, it is possible to use the same code to manage some objects of different classes that are using the same names for their instance data.

The best use case is the ability to bind a data (its name) to some triggers (especially method calls), like in the *Key-Value Observing* (KVO), which is not detailed here.

12.1.2 Interception

Accessing a data through a call to `valueForKey:` or `setValue:forKey:` is not an atomic operation. This access conforms to a calling convention procedure. In fact , this access is only possible if some methods have been implemented (such methods may be automatically generated when using *properties*, cf. section 12.2 on the next page), or if a direct access to the instance data has been explicitely allowed.

The Apple documentation describes precisely the behaviour of `valueForKey:` and `setValue:forKey:` [3]. For a call to `valueForKey:@"foo"`

- if it exists, call the method `getFoo`;
- otherwise, if it exists, call the method `foo` (most common case);
- otherwise, if it exists, call the method `isFoo` (common for boolean values);
- otherwise, if the class returns `YES` for the method `accessInstanceVariablesDirectly`, try to read the data member (if it exists) `_foo`, otherwise `_isFoo`, otherwise `foo`, otherwise `isFoo`;
- in case of success for a previous step, return the matching value;
- in case of failure, the method `valueForUndefinedKey:` is called; there is a default implementation in `NSObject` that throws an exception.

For a call to `setValue:..forKey:@"foo"`

- if it exists, call the method `setFoo:`;
- otherwise, if the class returns `YES` for the method `accessInstanceVariablesDirectly`, try to write the data member (if it exists) `_foo`, otherwise `_isFoo`, otherwise `foo`, otherwise `isFoo`;
- in case of failure, the method `setValue:forUndefinedKey:` is called; there is a default implementation in `NSObject` that throws an exception.

Please note that **A call to `valueForKey:` or `setValue:forKey:` can be used to trigger any compatible method ; there may be no data member underneath, it can be "dummy"**. For instance, calling `valueForKey:@"length"` on a string is semantically equivalent to calling directly the method `length`, since it is the first one which will be found when resolving the KVC.

However, the performance of KVC is obviously not as good as a direct method call, and must be advisely used.

12.1.3 Prototypes

Using the KVC requires to conform to the expected prototypes of the methods that are to be called : getters have no parameters and return an object, setters have one object as parameter and return nothing. The exact type of the parameter has no real importance in the prototype since it is of type `id` anyway.

Please note that structures and native types (`int`, `float`...) are supported : the Objective-C run-time Objective-C is able to perform an automatic *boxing* withing an `NSNumber` or an `NSValue` object. Thus, the value returned by `valueForKey:` is always an object.

The special case of the value `nil` given to `setValue:forKey:` is handled by the method `setNilValueForKey:`.

12.1.4 Advanced features

There are some details that should be considered, even if they are not detailed here.

- The first one is about *keypaths* that can include special treatments, like a sum calculation, a mean, a max or a min... The `@` character is the distinguishing mark.

- The second one is about the consistency between a call to `valueForKey:` or `setValue:forKey:` regarding the methods `objectForKey:` and `setObject:forKey:` provided by collections like associative arrays (cf. section 11.1 on page 54). Here again, the `@` is used to solve some ambiguities.

12.2 Properties

12.2.1 Use of properties

The notion of *property* can be met when defining classes. The keyword `@property` (and some attributes, cf. section 12.2.3 on the next page) can be associated to a data member, to tell how the accessors can be automatically generated by the compiler. It aims at writing less code and save some development time.

Moreover, the syntax used to access properties is simpler than a method call, so it can be handy to use properties even if we eventually have to write the code we want behind. The performance of properties are identical to a method call, because the identification of underlying method calls is made at compile time.

Most of the time, a property is bound to a data member. But if the accessors are redefined, nothing prevents the property to be "dummy"; in other words, it can *look like* an attribute from outside the object, and cover a behaviour much more complex than a simple value management from inside.

12.2.2 Description of properties

Describing a property means to tell the compiler how the accessors should be implemented :

- is it a read-only property from the outside ?
- if the data member is a native type, there are few variants, but if it is an oject, should it be encapsulated by copy, by strong reference, or by weak reference ? (this is related to memory management, cf. section 6.4.7 on page 44) ;
- shoudl it be *thread-safe* (cf. section 8.1 on page 51) ?
- what are the names of the accessors ?
- which data member should it be bound to ?
- which accessors should be automatically generated, and which one are left the the developer ?

Answering those questions is made in two steps :

- in the `@interface` block of a class, properties are declared with the appropriate attributes (cf. section 12.2.3 on the following page);
- in the `@implementation` block of that class, the accessors are qualified as implicit, or they are given an implementation (cf section 12.2.4 on page 60).

The prototype of the accessors is strict : for a getter, the expected type (or compatible) must be returned, and for a setter, `void` is returned and only one parameter of the expected type (or compatible) must be set.

The name of the accessors is also codified : for a `foo` data, names are `foo` for the getter, and `setFoo:` for the setter. It is allowed to customized the names. But keep in mind that unlike *Key-Value Coding* (section 12.1.2 on the preceding page), the name must be known **at compile time**, because the use of *properties* is designed to be as fast as calling the methods directly. Hence, no *boxing* is applied to the parameters which would be of incomatible types.

Here is an example with few explanations, but that acts as an quick preview of the global behaviour. The following subsections give the details required for a full understanding.

```
@interface class Car : NSObject
{
  NSString* registration;
  Person* driver;
}

//The registration is a read-only field, set by copy
@property NSString*  (readonly, copy) registration;

//the driver is a weak reference (no retain), and can be modified
@property Person* (assign) driver;

@end

...

@implementation

//let the compiler generate the code for "registration" if the
//developer does not do that himself
@synthesize registration;

//the developer is providing the implementation of getters/setters for "driver"
@dynamic driver;

//this method will match the getter for @dynamic driver
-(Person*) driver  {
 ...
}

//this method will match the setter for @dynamic driver
-(void) setDriver:(Person*)value {
 ...
}

@end
```

12.2.3 Properties attributes

A property is declared according to the following template :
`@property` *type name*;
or
`@property(`*attributes*`)` *type name*;

If they are not given, the attributes have a default value; otherwise, they can be redefined to answer the questions stated in the previous section. They can be :

- `readwrite` (default) or `readonly` to tell if the property should have both getter/setter or only the getter;
- `assign` (default), `retain` or `copy`, to tell how the value is stored internally;
- `nonatomic` to prevent thread-safety guards to be generated. They are generated by default. (There is no `atomic` keyword);
- `getter=..., setter=...` to change the default name of the accessors.

Inside the setter, the behaviours `assign`, `retain` or `copy` are affecting the way the data member is modified.

In a `-(void) setFoo:(Foo*)value` method, the three ways are :

```
self->foo = value ; //simple assignation
self->foo = [value retain]; //assignation with reference counter increment
self->foo = [value copy]; //object is copied, it must conform to the protocol
                          //NSCopying (cf. section 5.3.1 on page 35
```

In a garbage-collected environment (cf. section 6.6 on page 48), `retain` is not different from `assign`. But in that case, the attributes `__weak` and `__strong` can be added.

```
@property(copy,getter=getS,setter=setF:) __weak NSString* s; //complex declaration
```

(please note the "setF:" syntax **with the colon**)

12.2.4 Custom implementation of properties

The code snippet of section 12.2.2 on page 58 states that the implementation relies on two keywords only : `@synthesize` and `@dynamic`.

`@dynamic` means that it is up to the developer to provide the expected implementations (a mere setter if *read-only* was specified when declaring the property; getter and setter otherwise).

`@synthesize` means that, *unless the developer already did it*, the compiler should generate the accessors itself, conforming to the constraints that were used in the property declaration. Thus, in the example that was given, if the developer had implemented a `-(NSString*)registration` method, the compiler would have chosen it instead of generating a new one. Hence, we can deduce that one accessor out of two can be generated automatically, the other being provided by the developer.

At last, if an accessor was not found at compile time, and has not been created by the compiler by `@synthesize`, it can be added at run-time (cf. section 13.2 on page 63). This is valid to access the property. But in this case, the name of the expected accessor is decided at compile-time.

If, at run-time, no accessor is found, an exception is raised, but the program won't stop; it is the same behaviour as for a missing method.

When using `@synthesize`, the compiler can be asked to bind the property to a particular data member, which has not necessarily the same name.

```
@interface A : NSObject {
  int _foo;
}
@property int foo;
@end

@implementation A
@synthesize foo=_foo; //bind to "_foo" rather than "foo"
                      //(which does not even exist here)
@end
```

12.2.5 Syntax to access properties

To get or set the value of a property, the syntax to use is the dot : it is the same syntax as a simple C `struct`, and is consistent with the *keypath* principle (cf. section 12.1.1 on page 56). The performance is identical to a direct call to the underlying methods.

```
@interface A : NSObject {
  int i;
}
@property int i;
@end

@interface B : NSObject {
  A* myA;
}
@property(retain) A* a;
@end

...
A* a = ...
B* b = ...;
a.i = 1; //equivalent to [a setI:1];
b.myA.i = 1;//equivalent to [[b myA] setI:1];
```

Please note that inside the class `A` of the above example, the difference would be huge between `self->i` and `self.i`. Indeed, `self->i` is a direct access to the data member, while `self.i` triggers the property mechanism, and is a method call.

12.2.6 Advanced details

The documentation of properties [4] says that for a 64-bits compilation, the Objective-C run-time has a few discrepancies compared to 32-bits mode. Instance data related to some `@property` declarations can be omitted, for instance, since they can be implicit. The Apple documentation remains a reference that must be read to get all information.

13 Dynamism

13.1 RTTI (Run-Time Type Information)

The C++ language might be said to be a "false" object-oriented language. Compared to Objective-C, it is very static. This is a deliberate choice in favour of best performances at run-time. The run-time information that C++ can supply, through the *typeinfo* library, cannot be safely used because they depend on the compiler. Asking for the class of an object is a rare request because of strong typing; but it can occur when browsing a container. The `dynamic_cast` operator and, sometimes, `typeid`, can be used, but the interaction with the user of the program is limited. How to test that an object is an instance of a class known by its name ?

Objective-C is naturally designed for such questions. The classes being objects, they inherit their behaviour.

Downcasting is detailed in section 3.4.4 on page 20 since it is a particular use of `dynamic_cast`.

13.1.1 `class`, `superclass`, `isMemberOfClass`, `isKindOfClass`

The ability for an object to know its own type at run-time is called *introspection*, and can be handled with several methods.

`isMemberOfClass:` is a method that answers the question: "Am I an instance of a given class (leaving aside inheritance)" ?, while `isKindOfClass:` answers "Am I an instance of a given class or one of its subclasses ?".

Using these methods requires the false keyword: `class` (and not `@class`, which is used for forward declarations). Indeed, `class` is a method of NSObject, and returns a `Class` object, which is an instance of meta-class. Please note that the nil value for a class pointer is not `nil` but `Nil`.

```
BOOL test = [self isKindOfClass:[Foo class]];
if (test)
  printf("I am an instance of the Foo class\n");
```

Please also note that there is a way to get a mother-class class pointer: you can use the method `superclass`.

13.1.2 `conformsToProtocol`

This method has been explained in the section detailing protocols (section 4.4 on page 22). It is useful to know whether an object conforms to a protocol or not. This is not really dynamic, since the compiler only checks for explicit conformance rather than checking every methods. If an object implements all the methods of a given protocol, but does not explicitly say that it conforms to that protocol, the program is correct but `conformsToProtocol:` returns `NO`.

13.1.3 `respondsToSelector`, `instancesRespondToSelector`

`respondsToSelector:` is a instance method, inherited from `NSObject`. It is able to check whether an object implements a given method or not. The notion of *selector* is used (cf. section 3.3.4 on page 14). For example:

```
if ( [self respondsToSelector:@selector(work)] )
{
  printf("I am not lazy.\n");
  [self work];
}
```

To know whether a class implements a given method, without checking for inherited methods, one can use the *class* method `instancesRespondToSelector:`.

```
if ([[self class] instancesRespondToSelector:@selector(findWork)])
{
    printf("I can find a job without the help of my mother\n");
}
```

Please note that a call to `respondsToSelector:` cannot determine whether a class can handle a message through *forwarding* (cf. section 3.4.3 on page 19).

13.1.4 Strong typing or weak typing with `id`

C++ uses strong typing: one can use an object according to its apparent type only, otherwise the program cannot be compiled. In Objective-C, the constraint is more flexible if an object **whose type is explicitly known** is the target of a message that it cannot handle *at first sight*, then the compiler outputs a warning, but the program will run. The message will be lost (raising an exception), except if *forwarding* is triggered (cf. section 3.4.3 on page 19). If it is what the developer had in mind, the warning is redundant; in this case, typing the target to `id` instead of its real type removes the warning. In fact, any object is of type `id`, and can be the target of any message.

This weak-typing property is essential when using delegation: the delegate object does not have to be known to be used. Here is an example:

```
-(void) setAssistant:(id)anObject
{
  [assistant autorelease];
  assistant = [anObject retain];
}

-(void) manageDocument:(Document*)document
{
  if ([assistant respondToSelector:@(manageDocument:)])
    [assistant manageDocument:document];
  else
    printf("Did you fill the blue form ?\n");
}
```

Delegations are very common in Cocoa when a graphical interface is used, so controls can transmit actions from the user to worker objects.

13.2 Manipulating Objective-C classes at run-time

By including the header `<objc/objc-runtime.h>`, one can call several utility functions related to class information, and can add methods or instance variables, at run-time.

Objective-C 2.0 has introduced new functions, handier than the ones of Objective-C 1.0 (like `class_addMethod(...)` instead of `class_addMethods(...)`), deprecating most of Objective-C 1.0 functions.

This, it is remarkably easy to modify classes at run-time. Even if it is not so common, there are many cases where this feature is a real asset.

14 Objective-C++

Objective-C++ is still being developed. It is usable, but still has some gaps. Objective-C++ allows the concurrent use of Objective-C and C++, to take the best of both.

When mixing C++ and Objective-C exceptions, or when using C++ objects as instance data of Objective-Cobjects, refer to the documentation in order to understand the expected behaviour. However, it is definitely possible to write programs in Objective-C++. The implementation files have the extension *.mm*.

Since MacOS X 10.4, a C++ object can be used as instance data of an Objective-C class, and benefit from automatic construction/destruction [3].

15 The future of Objective-C

I have no information about future developments of Objective-C, like a version 3.0 or something else. However, Objective-C automatically benefits from the improvements of the C language. The projects GCC[4] and LLVM[5] are very interesting.

GCC, to follow the standards, is continuously improving its support of C++0x. Objective-C++will certainly benefit from that. Moreover, LLVM has been recently enriched by *closures* through the notion of `blocks` (cf. 15.1). This is a major C improvement, and Objective-C can use it.

So, the language is still under development, directly or indirectly, and we can assume that it ill be even richer in the future.

15.1 The *blocks*

15.1.1 Support and use cases

Since MacOS X 10.6, native APIs, like Grand Central Dispatch, may use *blocks*, with a syntax looking like pointers to function, where the star is replaced by a circumflex. An example :

```
void dispatch_apply(
  size_t iterations, dispatch_queue_t queue, void (^block)(size_t));
```

The last parameter is not a pointer to a function, but a *block*.

A *block* is a concept that is linked neither to Cocoa nor to Objective-C. It is a language C feature, not standard, not supported by every compiler. The LLVM compiler distributed with MacOS X 10.6 does support it.

For the ones who know, it is the C implementation of *closures*, well-known in Javascript. To simplify, it can be seen as anonymous functions, built on the fly, capturing their environment (variable values).

Blocks are very useful to replace callbacks, and more generally to reduce the code that must be written for delayed or asynchronous function calls.

[3] `http://developer.apple.com/releasenotes/Cocoa/RN-ObjectiveC/index.html`
[4] `http://gcc.gnu.org/`
[5] `http://llvm.org/`

15.1.2 Syntax

A block is created by the syntax

```
^{...some code...}
```

or

```
^(paramer1,[other parameters...]){...some code...}
```

It can be stored in a variable like a pointer to function, where the circumflex is replacing the star.

```
typedef void (^b1_t)(void);
b1_t block1 = ^{printf("hello\n");};
block1(); //print hello

typedef void (^b2_t)(int i);
b2_t block2 = ^(int i){printf("%d\n", i);};
block2(3);//print  3
```

15.1.3 Capturing the environment

A block can reference the variables that exist when it is created. It "captures" the variables state, and can thus use them, **read-only**. Modifying a variable which is not part of the block requires specific instructions (cf. section 15.1.4).

```
//example 1
int i = 1;
b1_t block1 = ^{printf("%d", i);};
block1();//display 1
i = 2;
block1();//still display 1 ! The value is the captured one.

//example 2
int i = 1;
b1_t block1 = ^{i=2;}//invalid : cannot compile.
                     //The variable cannot be modified like this.

//exemple 3
typedef void (^b1_t)(void);
b1_t getBlock() //this function returns a block
{
  int i = 1; //this variable is local to the function
  b1_t block1 = ^{printf("%d", i);};
  return block1;
}

b1_t block1 = getBlock();
block1(); //display 1:the block captured the local variable when it was accessible
```

15.1.4 `__block` variables

The fact that a block can use a variable which is no more in the current scope seems trivial : it has just created a copy. That is why to avoid confusion, you cannot modify such a variable in the block.

However, this can be changed when using a variable with the attribute `__block`. Then, it can be modified by a block. This is more subtle, because the captured variable must still be accessible when the block is called.

Please note that an `extern` or a `static` variable, since it is not an `auto` one (the default scope for variables in C), can be modified by a block.

```
//exemple 1
__block int i = 1;
b1_t block1 = ^{printf("%d", i);};
block1();//display 1
i = 2;
block1();//display 2

//exemple 2
__block int i = 1;
b1_t block1 = ^{printf("++i = %d", ++i);};
block1();//display 2
block1();//display 3

//exemple 3
typedef void (^b1_t)(void);
b1_t getBlock() //this function returns a block
{
  __block int i = 1; //this variable is local to the function, the returned block
                     //won't be valid outside getBlock()
  b1_t block1 = ^{printf("%d", i);};
  return block1;
}

b1_t block1 = getBlock();
printf("Caution: the following call is invalid\n");
block1(); //probably Segmentation Fault because of the "i" variable
```

Conclusion

This document is a quick look at many aspects of Objective-C, compared to C++ notions. It is useful as a quick reference for advanced developers willing to learn the language. I hope that the goal has been reached, but the reader is still welcome to provide any feedback to improve it again.

References

[1] Apple Computer, Inc. Autozone. `http://www.opensource.apple.com/darwinsource/10.5.5/autozone-77.1/README.html`.

[2] Apple Computer, Inc., Developer documentation. Garbage Collection Programming Guide. `http://developer.apple.com/documentation/Cocoa/Conceptual/GarbageCollection.pdf`.

[3] Apple Computer, Inc., Developer documentation. Key-Value Coding Programming Guide. `http://developer.apple.com/documentation/Cocoa/Conceptual/KeyValueCoding/KeyValueCoding.pdf`.

[4] Apple Computer, Inc., Developer documentation. The Objective-C 2.0 Programming Language. `http://developer.apple.com/documentation/Cocoa/Conceptual/ObjectiveC/ObjC.pdf`.

[5] Boost. `http://www.boost.org/`.

[6] GNUstep. `http://www.gnustep.org/`.

[7] Aaron Hillegass. *Cocoa Programming for MacOS X, 2nd edition.* Addison-Wesley, 2004.

[8] SGI. Standard template library programmer's guide. `http://www.sgi.com/tech/stl`.

[9] Will Shipley. `self = [supid init]`. `http://wilshipley.com/blog/2005/07/self-stupid-init.html`.

Document revisions

version 2.1

- added explanations on blocks.

version 2.0

- update for Objective-C 2.0;
- little fixes.

version 1.11

- fixed typos.

version 1.10

- English adaptation by Aaron Vegh;
- the section about cloning has been rewritten;
- added a section about IMP caching;
- precisions on `initialize`;
- precisions on Objective-C++;
- fixed error about `superclass`;
- precisions about “static” instance data;
- precisions on the labels of method parameters;
- fixed typos about virtual constructors;
- fixed mistakes about `self` and `super`;
- precisions about `in, out, byref...`
- precisions about Objective-C 2.0;
- precisions about static strings;
- fixed minor typos.

version 1.9

- minor changes.

version 1.8

- first English version.

Index