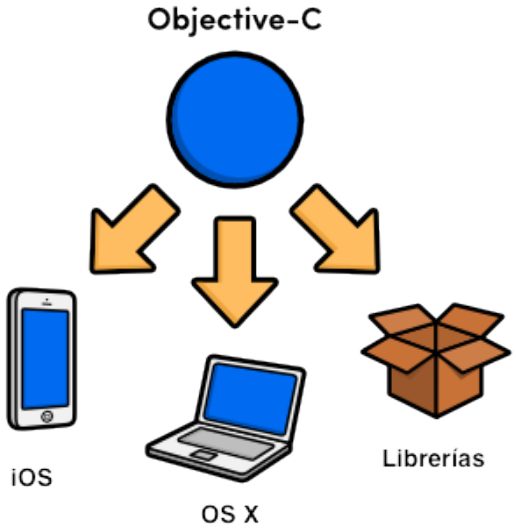
Introducción

Objective-C es el lenguaje de programación nativo para los sistemas iOS y OSX de Apple, es un lenguaje compilado de propósito general que puede ser utilizado para crear utilidades de terminal, hasta interfaces aplicaciones con interfaces gráficas animadas para dispositivos móviles.



Al igual que C++, Objective-C fue diseñado para agregar propiedades orientadas a objetos a C, pero ambos lenguajes lo logran con filosofías fundamentalmente distintas. Objective-C es mas dinámico, y relega muchas de sus decisiones al tiempo de ejecución en vez de tomarlas en tiempo de compilación, esto se ve reflejado en muchos de los patrones de diseño que se utilizan en el desarrollo de aplicaciones para iOS y OS X.

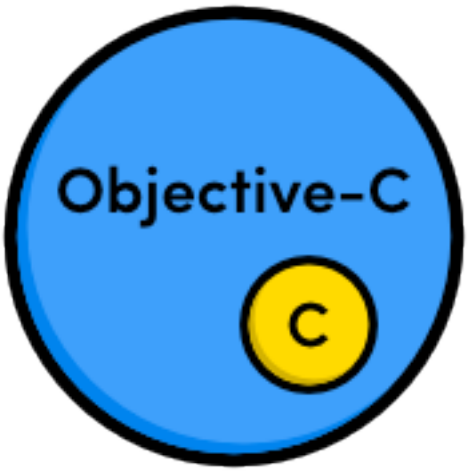
Objective-C también es conocido por la manera tan explícita en la que se denominan variables y métodos, resultando en un código que es virtualmente imposible de malinterpretar. A continuación se muestra un ejemplo de esto en una comparación de C# con Objective-C.

```
// C#
gabriel.Manejar("Ford Fiesta", "Oficina");
// Objective-C
[gabriel manejarElCarro:@"Ford Fiesta" alDestino:@"Oficina"];
```

Como se puede ver, los métodos escritos en Objective-C tienen la particularidad de incluir los parámetros dentro del nombre del método, al igual que se tiende a nunca abreviar nombres ni variables, gracias a esto, todos los métodos se leen mucho mas parecido al lenguaje natural. Una vez que un programador se acostumbra a esto, es mucho mas fácil retomar código escrito hace algún tiempo, trabajar en grandes equipos, o reutilizar código hecho por terceros.

Conceptos básicos de C

Objective-C es un súper grupo de C, lo que significa que es posible implementar ambos lenguajes transparentemente en el mismo código, de hecho Objective-C emplea C para manejar la mayoría de sus operaciones básicas, por lo que es importante tener una buena base en C antes de intentar aplicar aspectos avanzados del lenguaje.



Relación entre C y Objective-C

En esta sección se proveerá un repaso conciso del lenguaje de programación C, desde realización de comentarios, variables, operadores y punteros, esto sentara una base solida para todos los temas que se tocarán en este tutorial.

Comentarios

Hay dos maneras de incluir comentarios en un programa hecho en C. Comentarios de linea comienzan con un doble slash y terminan al final de la linea. Comentarios de bloque permiten comentar varias lineas de código simultáneamente, esto se logra rodeando las lineas que se quieren comentar con /* al inicio y */ al final del comentario.

Por ejemplo:

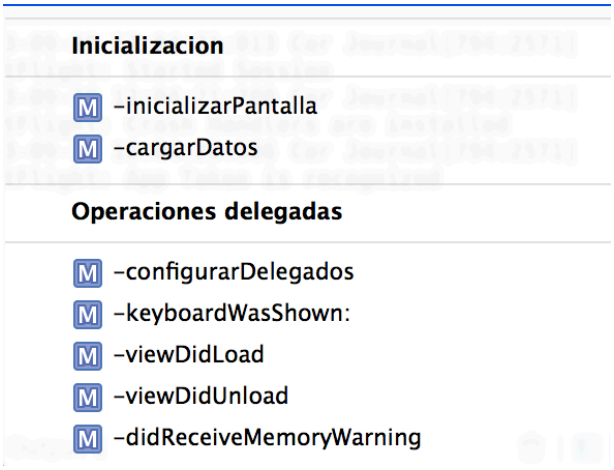
```
// Esto es un comentario de linea

/* Esto es un comentario de bloque.
   Se pueden comentar varias lineas. */
```

Aunque los Pragma Mark no son una funcionalidad de C, la incluyo en esta sección ya que están incluidos en XCode y ayudan a comentar y organizar el código. Ellos son el equivalente mas cercano a los Códigos de bloque encontrados en .Net.

Por ejemplo, al utilizar el siguiente código, se agrupan todos los métodos que estén por debajo de el hasta el siguiente Pragma Mark.

```
#pragma mark -
#pragma mark Inicializacion
#pragma mark -
```

Con esto podemos lograr el siguiente resultado en el navegador de XCode



Variables

Las variables son contenedores que pueden almacenar diferentes valores. En C, las variables son fuertemente tipadas, lo que significa que se debe especificar el tipo de valores que va a contener. Para declarar una variable se utiliza la sintaxis <tipo> <nombre>, y para asignarle un valor se utiliza el operador =. Es posible interpretar una variable con otro tipo de datos agregándole como prefijo el nuevo tipo de datos entre paréntesis, esto también es conocido como “Casteo”.

Toda la implementación de variables es demostrada en el siguiente ejemplo:

```
double costoProducto = 9200.8;
int costoProductoComoEntero = (int) costoProducto;
NSLog(@"El producto cuesta %.1f bolívares", costoProducto);  // 9200.8
NSLog(@"El producto cuesta %d bolívares", costoProductoComoEntero); // 9200
```

Ademas de los tipos de dato int y double, C define una serie de variables primitivas, las cuales serán explicadas con detalle en el modulo de Variables Primitivas, al igual que los parámetros de formato %.1f y %d utilizados en el ejemplo.

Constantes

El modificador const puede ser utilizado para decirle al compilador que una variable nunca puede cambiar, por ejemplo:

```
double const pi = 3.14159;
pi = 42001.0;            // Resulta en un error del compilador
```

Operadores aritméticos

Los operadores +, -, *, /, % son los símbolos utilizados para las operaciones aritméticas básicas.

```
NSLog(@"6 + 2 = %d", 6 + 2);    // 8
NSLog(@"6 - 2 = %d", 6 - 2);    // 4
NSLog(@"6 * 2 = %d", 6 * 2);    // 12
NSLog(@"6 / 2 = %d", 6 / 2);    // 3
NSLog(@"6 %% 2 = %d", 6 % 2);    // 0
```

Otros operadores que son comúnmente utilizados dentro de los ciclos son el de incrementar un entero (++) y el de reducirlo (--).

Condicionales

C provee la sintaxis estándar de los if que se puede encontrar en casi cualquier lenguaje de programación, la cual es demostrada en el ejemplo a continuación:

```
int modeloDelCarro = 1990;
if (modeloDelCarro < 1967) {
    NSLog(@"El carro es antiguo!!!");
} else if (modeloDelCarro <= 1991) {
    NSLog(@"El carro es clásico!");
} else if (modeloDelCarro == 2013) {
    NSLog(@"Es un carro nuevo!");
} else
    NSLog(@"No hay nada de especial en este carro.");
```

Operador	Descripción
a == b	Igual que
a != b	Diferente que
a > b	Mayor que
a >= b	Mayor o igual que
a < b	Menor que
a <= b	Menor o igual que
!a	Negación lógica
a && b	"Y" lógico
a || b	"O" lógico

C también incluye el operador switch, sin embargo este solo funciona con números enteros, lo que lo hace inflexible comparado con el switch de otros lenguajes o el if descrito anteriormente. Por Ejemplo:

```
switch (modeloDelCarro) {
   case 1987:
      NSLog(@"Tu carro es de 1987.");
      break;
   case 1988:
      NSLog(@"Tu carro es del 1988.");
      break;
   case 1989:
   case 1990:
      NSLog(@"Tu carro es de 1989 o de 1990.");
      break;
   default:
      NSLog(@"No tengo idea de cuando fabricaron tu carro.");
      break;
}
```

Ciclos

Existen los ciclos iterativos while y for, los cuales nos permiten recorrer arreglos o listas, también poseen los operadores break y continue, los cuales nos permiten terminar prematuramente la ejecución de un ciclo o saltar una iteración.

```
int modeloDelCarro = 1990;
// Ciclo While
int i = 0;
while (i<5)
{
   .....
   i++;
}
// Ciclo For
for (int i=0; i<5; i++)
{
   .....
}
```

Aunque técnicamente el ciclo for-in no pertenece a C, será incluido en esta sección ya que nos permite iterar sobre listas y arreglos de forma mas eficiente.

```
// Ciclo For-in (Especifico a Objective-C)
NSArray *modelos = @[@"Ford", @"Honda", @"Nissan", @"Porsche"];
for (id modelo in modelos)
{
   NSLog(@"%@", modelo);
}
```

Punteros en Objective-C

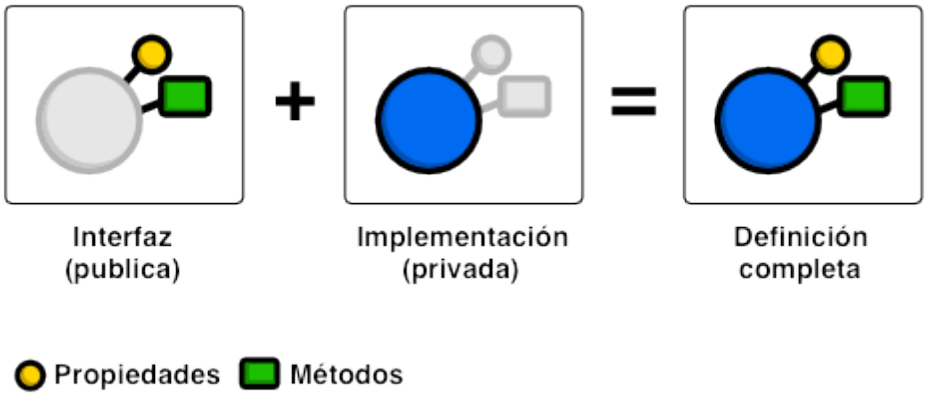
No es necesario ir muy profundo en teoría de punteros al trabajar en Objective-C, lo mas importante que se debe saber es que todos los objetos complejos deben ser tratados como punteros, por ejemplo un objeto del tipo NSSTring es referido con un puntero y no una variable normal:

```
NSString *marca = @"Honda";
```

Cuando queremos referirnos a punteros nulos, hay una pequeña diferencia entre Objective-C y C. Cuando en C utilizamos NULL, Objective-C define su propio nombre para objetos nulos y lo denomina nil. Una buena regla es utilizar nil para variables de Objective-C y utilizar NULL cuando se trabaje con punteros de C.

Clases

El manejo de clases en Objective-C es similar al de C++ en el sentido de que separa la interfaz de una clase de su implementación. Una interfaz declara propiedades y métodos públicos de una clase, y la implementación define el código que realmente hace que los métodos funcionen.



La interfaz e implementación de una clase.

Creando Clases

Como ejemplo utilizaremos una clase llamada Carro, cuya interfaz (también llamada cabecera) se encuentra en un archivo llamado Carro.h y su implementación en otro llamado Carro.m. Estas son las extensiones de archivo estándar en Objective-C, la interfaz de una clase es lo que otras clases utilizan cuando necesitan interactuar con ella, y su implementación solo es utilizada por el compilador.

Interfaces

```
// Carro.h
#import <Foundation/Foundation.h>

@interface Carro : NSObject

@property (copy) NSString *modelo;
- (void) manejar;

@end
```

Una interfaz es declarada con la palabra clave @interface, después del cual deben venir el nombre de la clase y de la súper clase separados por dos puntos (:).

La palabra clave @property declara una propiedad publica y el atributo (copy) define el comportamiento del manejador de memoria para esa propiedad. En este caso, el valor asignado a modelo, va a ser guardado como una copia, y no como un puntero directo (En el modulo de propiedades se discutirá esto con mas detalle). Luego viene el tipo de datos de la propiedad y su nombre, igual que una declaración normal de variable.

La linea -(void) manejar declara un método llamado manejar, que no recibe parámetros, y (void) define el tipo de datos que retorna. El signo menos (o guión) lo define como un método de instancia, por lo que es necesario crear un objeto Carro para poder invocar este método, si por el contrario se utiliza un signo mas (+), el método puede ser invocado directamente desde la clase, sin necesidad de instanciar un objeto (Esto se conoce como un método estático en otros lenguajes).

Implementaciones

Lo primero que cualquier implementación de clase debe importar es su interfaz correspondiente. La palabra clave @implementation es similar a @interface, excepto que no es necesario incluir la súper clase. Es posible declarar una interfaz privada dentro de la implementación, para contener ahí propiedades privadas que solo son disponibles para la implementación, así como también se pueden almacenar variables privadas entre llaves, después del nombre de la clase:

```
// Carro.m
#import "Carro.h"
@interface Carro()

@property (nonatomic, copy) Conductor *conductor;
@end

@implementation Carro
{
   // variables privadas
   double _kilometraje;
}

@synthesize modelo = _modelo;
@synthesize conductor = _conductor;

- (void) manejar
{
   NSLog(@"Manejando un %@!!", self.modelo);
}
@end
```

@synthesize es una directiva que automáticamente genera los métodos que permiten acceder a los datos de una variable. Por defecto, el getter es simplemente el nombre de la propiedad (modelo) y el setter es el nombre de la propiedad con la primera letra en mayúscula, con el prefijo set (setModelo). Esto es mucho mas fácil que crear manualmente estos métodos para cada propiedad.

La implementación de manejar debe tener el mismo nombre que en la interfaz, pero es seguido del código que debe ser ejecutado cuando el método es llamado. Hay que notar como se accede al valor de modelo a través de self.modelo y no directamente con la propiedad _modelo. El único lugar donde se debe acceder a las propiedades directamente es dentro de los métodos init y dealloc.

La palabra clave self se refiere a la instancia actual que está llamando al método (igual que lo hace this en C++, C# y Java).

Instanciar y utilizar clases

Cualquier clase que quiera interactuar con la clase Carro que acabamos de crear, debe importar su interfaz o cabecera (Carro.h). Nunca se debe intentar acceder directamente a la implementación. El uso completo de la clase carro se puede ver en el código a continuación.

```
// main.m
#import <Foundation/Foundation.h>
#import "Carro.h"

int main(int argc, const char * argv[])
{
    Carro *ford = [[Carro alloc] init];

    [ford setModelo:@"Ford Fiesta"];
    NSLog(@"Creado un carro modelo %@", [ford modelo]);

    toyota.model = @"Ford Explorer";
    NSLog(@"Cambié el modelo del carro a una %@", ford.modelo);

    [ford manejar];
    return 0;
}
```

Luego de que se importa la interfaz con el #import, se pueden instanciar objetos con la sentencia alloc/init mostrada en el código. Instanciar un objeto es un proceso que consta de dos pasos, primero se debe reservar memoria para el objeto utilizando el método alloc, para luego inicializarlo con el método init. Nunca se debe utilizar un objeto sin inicializar.

No está de mas recordar, que todos los objetos deben ser creados como punteros, por esto es que se utiliza Carro *ford y no Carro ford al declarar la variable.

Para llamar un método de un objeto en Objective-C, se deben colocar la instancia del objeto y el método, separados por un espacio entre corchetes. Los argumentos se pasan luego del nombre del método precedidos por dos puntos (:). Así que para

personas que tengan experiencia en C++, C#, Java, o Python, la sentencia [ford setModelo:@"Ford Fiesta"] se traduciría a:

```
ford.setModelo("Ford Fiesta");
```

Este ejemplo también muestra ambas maneras de trabajar con las propiedades de in objeto, se puede utilizar los métodos generados por el @synthesize (modelo y setModelo), o se puede utilizar el punto, lo que es una sintaxis mas familiar para los desarrolladores.

Métodos de clase (estáticos) y variables

El código previo define propiedades y métodos de instancia (necesitan un objeto instanciado para utilizarlos), pero también es posible definir propiedades y métodos de la clase. Estos son llamados comúnmente estáticos en otros lenguajes de programación, pero no se deben confundir con la palabra clave static (en Objective-C un método static, es un método privado que solo puede ser invocado desde la misma clase).

Los métodos de clase se definen exactamente igual que los de instancia, con la diferencia que están precedidos de un signo mas (+) en vez de un signo menos o guión (-). Por ejemplo, vamos a añadir el siguiente método de clase a Carro.h:

```
// Carro.h
+ (void)setModeloPorDefecto:(NSString *)unModelo;
```

Mientras que técnicamente no existe una variable de clase en Objective-C, se puede simular el comportamiento, declarando una variable estática antes de definir la implementación:

```
// Carro.m
```

```
#import "Carro.h"

static NSString *_modeloPorDefecto;

@implementation Carro {
...

+ (void) setModeloPorDefecto:(NSString *) unModelo
{
   _modeloPorDefecto = [unModelo copy];
}

@end
```

La llamada [unModelo copy] crea una copia del parámetro en vez de asignarlo directamente, este paso se puede saltar si utilizamos el atributo (copy) en la propiedad modelo.

Los métodos de clase utilizan la misma sintaxis de corchetes que los métodos de instancia, pero deben ser llamados directamente en la clase como se muestra a continuación. Estos métodos no pueden ser llamados desde una instancia de la clase ([ford setModeloPorDefecto:@"Modelo T"] arrojará un error).

```
// main.m
[Carro setModeloPorDefecto:@"Nissan Vera"];
```

Constructores

En Objective-C no existen los constructores tal y como los conocemos, en vez de ellos, un objeto es inicializado al llamar el método init inmediatamente después de llamar al método alloc, por eso es que instanciar un objeto es un proceso que consta de dos pasos.

init es el método de inicialización por defecto, pero se pueden definir versiones propias que acepten parámetros. Estos son métodos normales de instancia, pero siempre deben comenzar su nombre con init. Por ejemplo:

```
// Carro.h
- (id)initConModelo:(NSString *)unModelo;
```

La implementación de este método se puede ver a continuación.

```
// Carro.m
- (id) initConModelo:(NSString *) unModelo
  {
        self = [super init];
         if (self){
           _model = [unModelo copy];
           _kilometraje = 0;
         }
   return self;
}

- (id)init {
   // Asi se puede asegurar que el objeto carro siempre va a tener una configuración base
   return [self initConModelo:_modeloPorDefecto];
}
```

Los métodos de inicialización siempre deben retornar una referencia al propio objeto, y si este no puede ser inicializado, deben retornar nil. Por esto es que siempre hay que verificar que self exista antes de intentar utilizarlo.

Inicializacion de clases

El método initialize es el equivalente a init al nivel de clases. Este método permite personalizar la clase antes de que nadie la utilice. Por ejemplo lo podemos utilizar para llenar el valor de la variable _modeloPorDefecto:

```
// Carro.m
+ (void)initialize {
   if (self == [Carro class])
  {
      _modeloPorDefecto = @"Ford Fiesta";
   }
}
```

Este método es llamado una vez por cada clase y sub clase, antes que la clase sea utilizada. Debido a esto, es una buena practica utilizar el condicional self == [Carro class] para asegurarnos que el código no se ejecute con las subclases de Carro.

En Objective-C no es necesario decir explícitamente cuando un método esta siendo sobrescrito y aunque init e initialize están ambos definidos en la súper clase NSObject, el compilador no detectara ningún error al definirlos en Carro.m.

A continuación podemos ver los los métodos inicializadores en acción.

```
Carro *fordFiesta = [[Carro alloc] init];
NSLog(@"Creado un %@", [fordFiesta modelo]);
```

```
Carro *toyotaMeru = [[Carro alloc] initConModelo:@"Toyota Meru"];
NSLog(@"Creado una %@.", toyotaMeru.modelo);
```

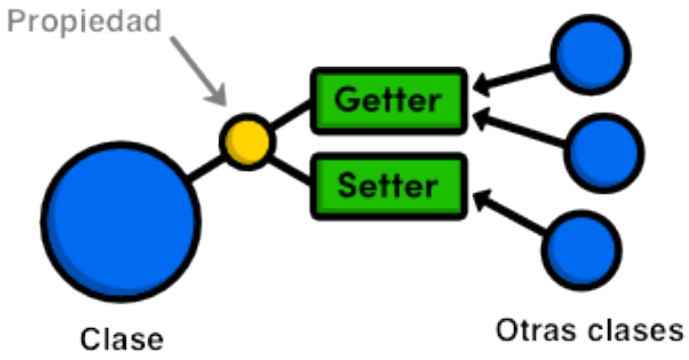
Tipado de clases

Ya que las clases en Objective-C también son definidas como objetos, es posible consultar a que clase pertenece un objeto, abriendo las posibilidades de crear métodos que reciban objetos genéricos.

```
Carro *fordEcoSport = [[Carro alloc] initConModelo:@"Ford EcoSport"];

// Verifica que un objeto pertenezca a una clase o una de sus subclases
if ([fordEcoSport isKindOfClass:[NSObject class]])
    NSLog(@"%@ es una instancia de NSObject o una de sus subclases",
        [fordEcoSport modelo]);

// Verifica que un objeto pertenezca a una clase, pero no a una de sus subclases
if ([fordEcoSport isMemberOfClass:[Carro class]]) {
    NSLog(@"%@ es una instancia de Carro", [fordEcoSport modelo]);
```

Propiedades

Las propiedades de un objeto permiten que otros objetos interactuen con el, pero en una aplicación orientada a objetos bien diseñada, no es posible acceder a las propiedades internas de los objetos. En vez de esto se utilizan los getters y los setters como una abstracción que nos permite controlar la manera en la que otras clases interactúan con las propiedades de nuestra clase



La meta de la directiva @property es facilitar la creación y configuración de los getters y setters, ya que ésta los implementa automáticamente con la nomenclatura estándar de Objective-C. A continuación un ejemplo del uso del a directiva @property:

```
@property (nonatomic, strong) Carro *carro;
```

Esto se debería utilizar en alguna clase que desee utilizar un objeto del tipo Carro, pero es buena practica sobreescribir el comportamiento del getter para propiedades de objetos, al sobreescribirlo podemos asegurarnos que el getter nunca retorne un objeto nil, evitando así problemas mas adelante.

Para sobreescribir el getter solo debemos declarar un método con el mismo nombre de la nomenclatura estándar de Objective-C y el compilador se encarga de hacer la sobreescritura.

```
- (Carro *) carro
{
   if(!_carro)
   {
      _carro = [[Carro alloc] init];
   }

   return _carro;
}
```

Atributos de las propiedades

Al declarar una propiedad es posible agregarles atributos o modificadores que influyen en la manera de comportarse del objeto, a continuación una lista de los posibles atributos y su significado.

Atributo	**Descripción**
readonly	No genera el método setter.
nonatomic	Sobreescribe el comportamiento por defecto de las propiedades, que bloquea la propiedad para que nadie mas pueda accesarla o modificarla mientras se está ejecutando una operación sobre ella.*
strong	Crea una relación fuerte con el valor asignado. (Este es el comportamiento por defecto).
weak	Crea una relación débil con el valor asignado. Principalmente se utiliza para evitar ciclos de propiedad.
copy	Crea una copia del valor asignado en vez de referenciar la instancia existente.

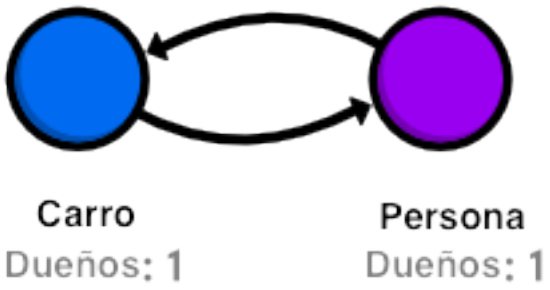
*Una propiedad atómica es útil en ambientes de múltiples hilos, pero ya que en la mayoría de las aplicaciones de iOS, el procesamiento se realiza en el hilo principal (el de la interfaz), se suele utilizar la propiedad nonatomic.

Manejo de memoria

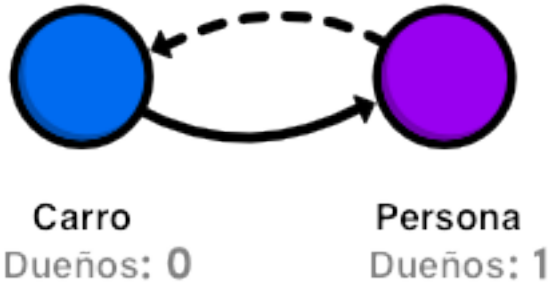
En cualquier lenguaje orientado a objetos, los objetos “viven” en la memoria de la computadora y en todos los sistemas y especialmente en teléfonos celulares, este es un recurso escaso. La meta del manejo de memoria es asegurarse que el programa no consuma mas memoria de la que es necesaria creando y eliminando objetos de forma eficiente.

Muchos lenguajes logran esto con el recolector de basura, pero Objective-C usa una alternativa mucho mas eficiente referida como adueñamiento de objetos. Cuando se empieza a interactuar con un objeto, se dice que eres dueño de él, lo que garantiza que el objeto exista mientras lo estés usando. Cuando ya no lo necesitas utilizar, cedes la propiedad del objeto y si el objeto no tiene mas dueños, el sistema operativo lo destruye y libera la memoria asociada a el. Esta tarea es llevada a cabo automáticamente en Objective-C si se activa el Automatic Reference Count o ARC.

Algo que hay que tener presente es que existen casos donde un objeto pueda necesitar saber quien lo esta utilizando, generando así un ciclo, el cual si no tenemos cuidado podría resultar en un ciclo de propiedad, en donde ninguno de los dos objetos será eliminado nunca por el ARC, ya que siempre va a haber otro objeto que lo referencia.



Este problema lo podemos evitar utilizando la propiedad `weak` en el objeto secundario, ésta propiedad genera un vinculo sin colocar al objeto como dueño del otro, lo cual evita el ciclo de propiedad. Sin embargo, utilizando el ejemplo, puede que el objeto carro sea eliminado mientras que persona sigue estando instanciado y si intentamos acceder a el, la propiedad `weak nos` retornará `nil` para evitar un puntero perdido.



Excepciones y Errores

Dos tipos de problemas pueden suceder al ejecutar una aplicación en iOS o en OSX. Excepciones representan errores del nivel del programador, como tratar de acceder a una posición de una lista que no existe. Ellos están diseñados para informar al desarrollador que una algo inesperado sucedió y ya que por lo general ellas causan que el programa se detenga, una excepción no debería ocurrir en producción.

Por el contrario, los errores son problemas del nivel del usuario, como cargar un archivo que no existe. Porque los errores son algo esperado durante la ejecución normal del programa, se debe buscar este tipo de condiciones e informar al usuario cuando ocurren. En la mayoría de los casos, los errores no deben detener tu aplicación.

Excepciones

Las excepciones son representadas por la clase NSException. Está diseñada para ser una manera universal de encapsular los datos de una excepción. Las tres propiedades principales son las siguientes:

Propiedad	Descripción
name	Un NSString que identifica la excepción.
reason	Un NSString que contiene la descripción en lenguaje natural de la excepción.
userInfo	Un NSDictionary (Hash) cuyos pares llave-valor contiene información adicional acerca de la excepción.

Es importante entender que las excepciones solo deben ser utilizadas para problemas serios en la programación y la idea es que el desarrollador sepa que algo está mal temprano en el ciclo del desarrollo y se espera que el problema sea arreglado para que no vuelva a suceder.

El manejo de excepciones no varía mucho con respecto a otros lenguajes, ya que se utilizan las mismas tres sentencias básicas: @try, @catch y @finally.

```
@try {
        ....
    } @catch(NSException *excepcion) {
      NSLog(@"Ocurrió un error: %@", excepcion.name);
      NSLog(@"Detalle del error: %@", excepcion.reason);
    } @finally {
      NSLog(@"Ejecutando el finally");
    }
```

Errores

Los errores representan problemas esperados y existen varios tipos de operaciones que pueden fallar sin causar que el programa se detenga. Al contrario que las excepciones, el verificar por errores es un aspecto normal de la producción de código de calidad.

La clase NSError encapsula los detalles de una operación fallida, sus propiedades principales son similares a las de NSException.

Propiedad	Descripción
domain	Un NSString que contiene el dominio del error. Esto es utilizado para organizar los errores en una jerarquía y asegurar que los códigos de error no entren en conflicto.
code	Un NSInteger que representa el ID del error. Cada error en el mismo dominio debe tener un código único.
userInfo	Un NSDictionary (Hash) cuyos pares llave-valor contiene información adicional acerca del error.

El diccionario userInfo por lo general contiene mas información que la versión de NSException. Algunas de las llaves predefinidas son las siguientes:

Llave	Valor
NSLocalizedDescriptionKey	Un NSString que representa la descripción completa del error. Por lo general también incluye la razón.
NSLocalizedFailureReasonErrorKey	Un breve NSString con la razón del fallo.
NSUnderlyingErrorKey	Una referencia a otro objeto NSError que representa el error en el dominio superior.

Dependiendo del error, éste diccionario también contendrá mas información específica al dominio. Por ejemplo, errores de carga de archivo tendrán una llave llamada NSFilePathErrorKey que contiene la ruta al archivo solicitado.