

Tutorial Objective-C

Andrés Mauricio Rondón Patiño
Osman David Jiménez Gutiérrez
Manuel Alejandro Vergara Díaz

1 de Mayo de 2016

Índice

1. Instalación	2
1.1. Windows (probado en Windows 7)	2
1.2. Mac OS X	6
1.3. Linux - Ubuntu (Probado en Ubuntu 14.04 y Linux Mint 17.2)	10
2. Primeros pasos con Objective-C	13
2.1. Tipos de datos	15
2.1.1. Tipos básicos	15
2.1.2. Tipos enumerados	17
2.1.3. Tipo vacío	17
2.1.4. Tipos derivados	17
2.2. Operadores	23
2.2.1. Operadores aritméticos	23
2.2.2. Operadores relacionales	23
2.2.3. Operadores lógicos	24
2.2.4. Operadores bit a bit	24
2.2.5. Operadores de asignación	26
2.2.6. Otros operadores	26
2.3. Condicionales	26
2.3.1. Sentencia if	27
2.3.2. Sentencia if-else	27
2.3.3. Sentencia switch	28
2.4. Bucles	29
2.4.1. Bucle While	29

2.4.2.	Bucle Do-While	29
2.4.3.	Bucle For	30
2.4.4.	Declaraciones para el control de bucles	31
3.	Particularidades de Objective-C	31
3.1.	Clase envolvente (Números)	31
3.2.	Cadenas	33
3.3.	Funciones	34
3.4.	Colecciones	36
3.4.1.	Arreglos - Listas	36
3.4.2.	Sets	38
3.4.3.	Diccionarios	41
4.	Programación Orientada a Objetos	43
4.1.	Clases	43
4.1.1.	Definiendo clases	44
4.1.2.	Herencia	48
4.1.3.	Polimorfismo	51
4.2.	Encapsulamiento	54
4.3.	Protocolos	55
5.	Ejemplos	56
5.1.	Recorrido por el lenguaje	56
5.2.	Clases	56

1. Instalación

Escoja el tutorial según su sistema operativo.

1.1. Windows (probado en Windows 7)

Para la instalación en Windows hay que dirigirse a la siguiente url <http://www.gnustep.org/windows/installer.html> y descargar e instalar los paquetes binarios "MSYS system", Corez "Devel".

Package	Required?	Stable	Unstable	Notes
GNUstep MSYS System	Required	0.30.0	-	MSYS/MinGW System
GNUstep Core	Required	0.34.0	-	GNUstep Core
GNUstep Devel	Optional	1.4.0	-	Developer Tools
GNUstep Cairo	Optional	0.34.0	-	Cairo Backend
ProjectCenter	Optional	0.6.1-2	-	IDE (Like Xcode, but not as complex)
Gorm	Optional	1.2.20-2	-	Interface Builder (Like Xcode NIB builder)

Figura 1: Instaladores Objective-C en Windows

Se continúa con el proceso de instalación para los 3 paquetes como se hace normalmente en Windows. Es importante instalarlos en el orden que se indica en la Figura 1 para evitar problemas de configuración.

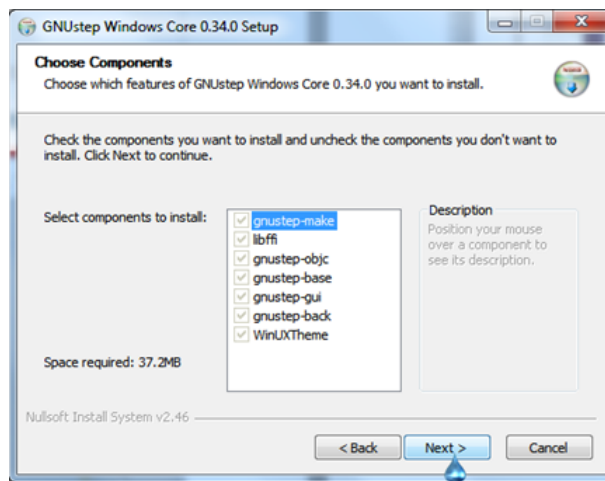


Figura 2: Instalación Paquete Core

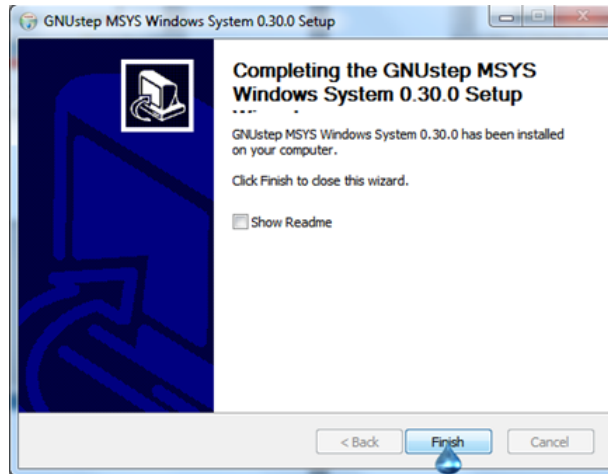


Figura 3: Instalación Paquete MSYS

La instalación incluye una Shell propia para compilar y ejecutar en Objective-C (algo similar al cmd de Windows):

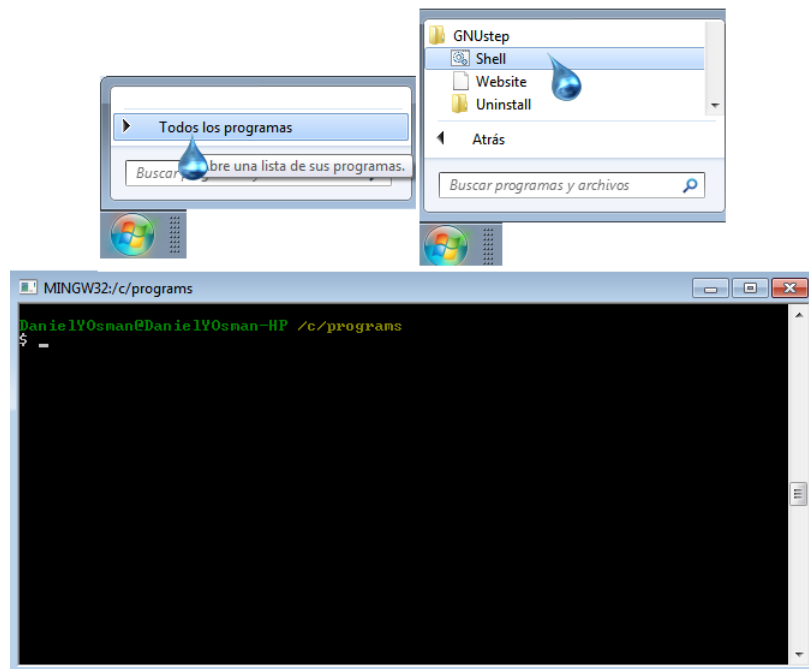
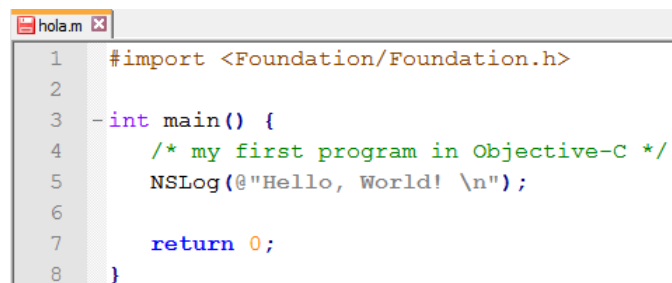


Figura 4: Shell de Objective-C en Windows

Para probar el correcto funcionamiento del lenguaje, se puede hacer una prueba básica (un "holamundo"). Para esto se debe crear un archivo que tenga el código a probar, y se guarda con la extensión **.m**, para este caso se va a crear un archivo llamado **helloworld.m** (revisar la página) y lo vamos a compilar y ejecutar.



```
1  #import <Foundation/Foundation.h>
2
3  -int main() {
4      /* my first program in Objective-C */
5      NSLog(@"Hello, World! \n");
6
7      return 0;
8  }
```

Figura 5: Código fuente de un holamundo

Para compilarlo, se abre el shell y nos movemos entre las carpetas a donde está el archivo que se quiere compilar, para esto usamos el comando **cd nombre_carpetas**. Una vez situados en la carpeta se escribe el siguiente comando:

```
gcc 'gnustep-config --objc-flags' -L /GNUstep/System/Library
/Libraries helloworld.m -o helloworld -lgnustep-base -lobjc
```

Básicamente lo que hace este comando es ubicar las librerías de Objective-C, compilar el archivo con extensión **.m** y crear el archivo **.exe** con el nombre que ponemos después del **-o**. Una vez que se ejecuta este comando, en la carpeta se generan dos archivos, uno con extensión **.exe** y otro con extensión **.d**. El archivo que usaremos para ejecutar el código, es el de extensión **.exe** con el siguiente comando:

```
./helloworld.exe
```

Y nos debería salir algo como lo siguiente:

```
MINGW32:/c/Programs
DanielYOsman@DanielYOsman-HP ~
$ cd C:/
DanielYOsman@DanielYOsman-HP /c
$ cd Programs
DanielYOsman@DanielYOsman-HP /c/Programs
$ dir
helloworld.m
DanielYOsman@DanielYOsman-HP /c/Programs
$ gcc `gnustep-config --objc-flags` -L /GNUstep/System/Library/Libraries hellow
orld.m -o helloworld -lgnustep-base -lobjc
DanielYOsman@DanielYOsman-HP /c/Programs
$ dir
helloworld.d helloworld.exe helloworld.m
DanielYOsman@DanielYOsman-HP /c/Programs
$ ./helloworld.exe
2016-05-01 14:42:00.046 helloworld[9340] Hello, World!
DanielYOsman@DanielYOsman-HP /c/Programs
$
```

Figura 6: Compilación y ejecución de un holamundo

1.2. Mac OS X

Para la instalación de Objective-C lo primero que debemos hacer es abrir el App Store de Mac, y allí buscar el IDE Xcode. Cuando lo hayamos encontrado damos clic en "*install*", como se muestra en la *figura 7*, y esperamos que se descargue e instale.

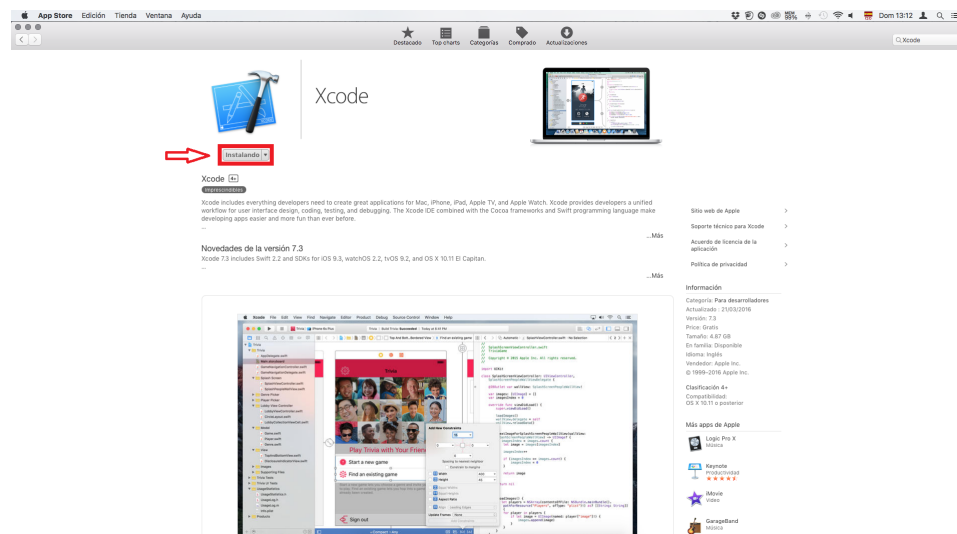


Figura 7: IDE Xcode en el App Store de MAC OS X

Cuando tengamos ya el IDE instalado, lo ejecutamos y nos aparecerá una ventana como la que se muestra en la *figura 8*, damos clic en "Create a new Xcode project" para crear un nuevo proyecto.

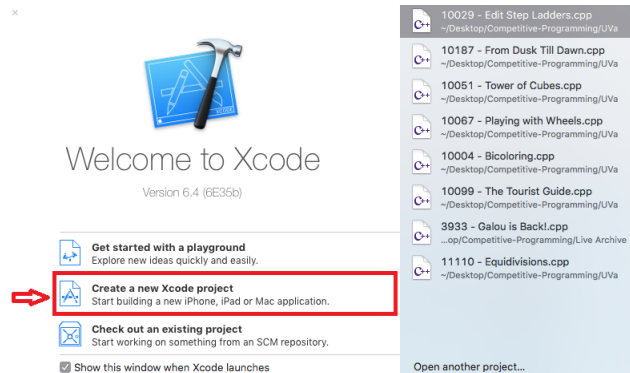


Figura 8: Creación de un proyecto en Xcode

Seleccionamos en el menú izquierdo la opción "Application" del submenú de "OS X", luego de esto se mostrarán unas nuevas opciones en la parte derecha de la ventana, allí seleccionamos "Command Line Toolz" damos clic en el botón "next" para continuar. Ver la *figura 9* para una mejor claridad.

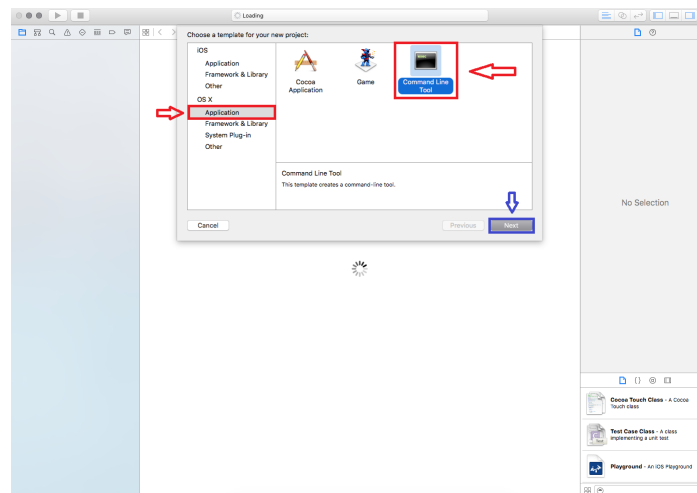


Figura 9: Plantilla para el nuevo proyecto

Ahora nos saldrá una nueva ventana que nos permitirá nombrar nuestro proyecto, allí lo más importante es seleccionar en el item de "Language" la

opción de "Objective-C", damos en "next" para continuar. Nos pedirá que escojamos en donde deseamos guardar el proyecto antes de terminar.

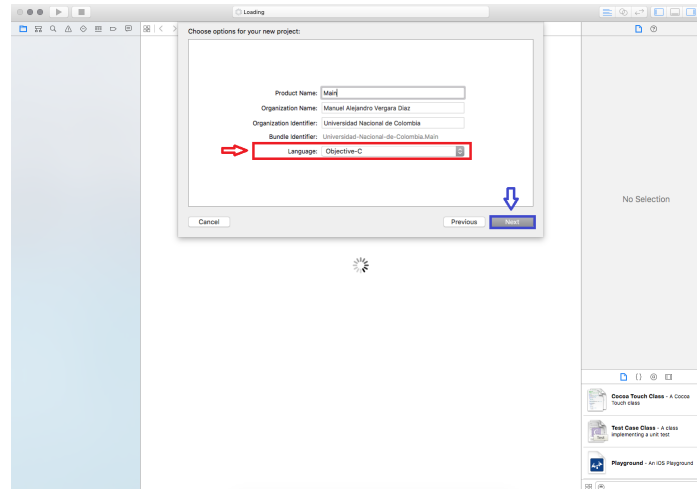


Figura 10: Opciones del nuevo proyecto

Después de creado el proyecto con toda la configuración previa, se desplegará una ventana como la que se muestra en la *figura 11*, seleccionamos en el menú de la parte izquierda el archivo "main.m" para abrirlo(la extensión .m es la que identificará los archivos de Objective-C).

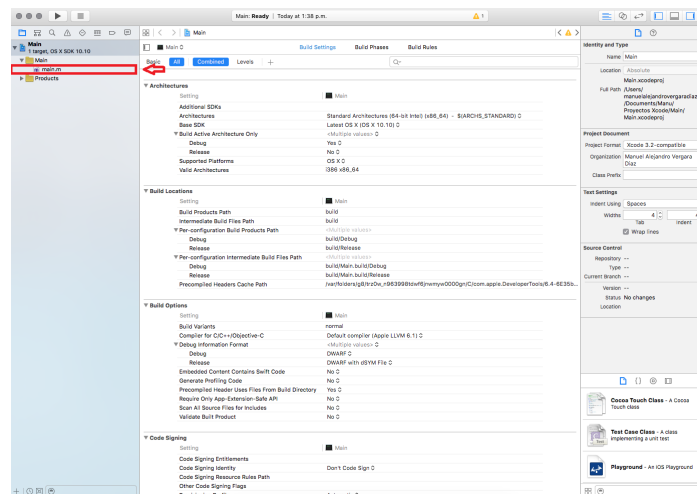


Figura 11: Ventana después de haber creado y configurado el nuevo proyecto

Ahora podremos ver un pequeño código que nos carga por defecto Xcode, el clásico *Hello World*, para ejecutarlo en la parte superior izquierda del editor damos clic en el botón de *play* para compilar y poder ver el resultado del programa que hemos hecho. Ver la *figura 12* para mejor claridad.

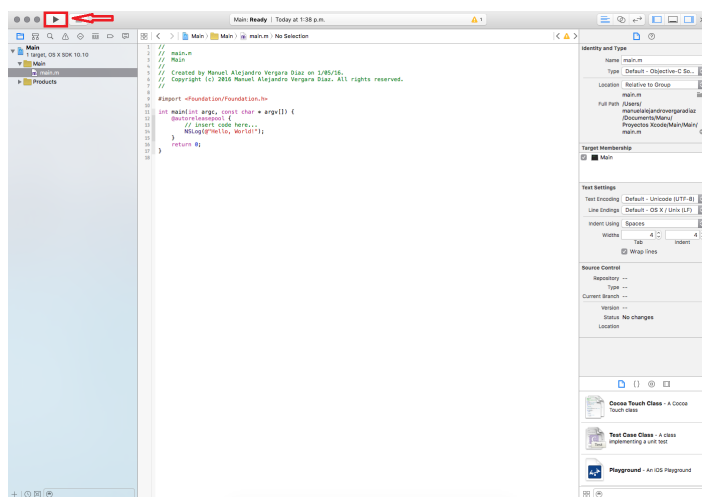


Figura 12: Ejecución de un programa en Objective-C

Finalmente en la parte inferior aparecerán unas ventanas nuevas, una de ellas es la consola donde podremos interactuar directamente con el programa que hemos ejecutado anteriormente.

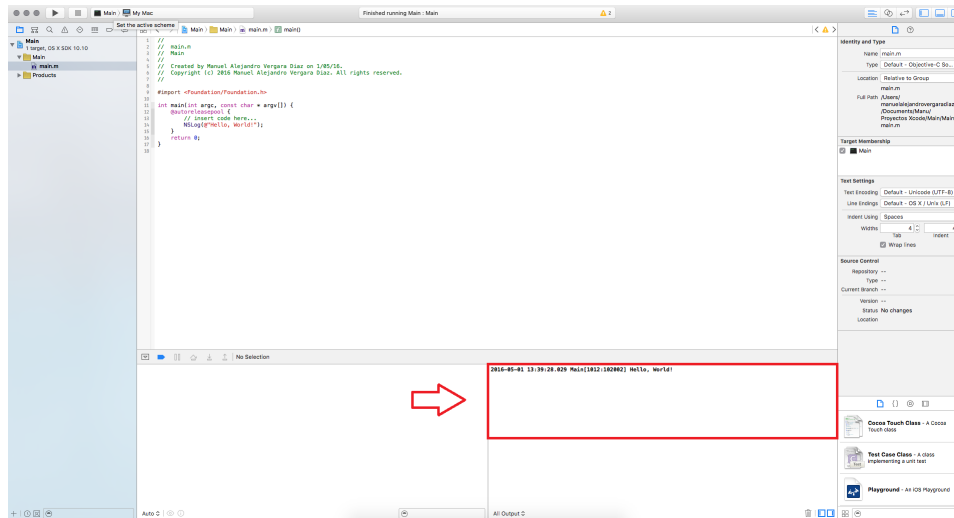


Figura 13: Resultados de la ejecución del programa

1.3. Linux - Ubuntu (Probado en Ubuntu 14.04 y Linux Mint 17.2)

La instalación se hará por medio del gestor de paquetes predeterminado de Ubuntu y sus distribuciones derivadas (apt-get).

Lo primero que debemos hacer es actualizar el gestor de paquetes con el comando

```
sudo apt-get update
```

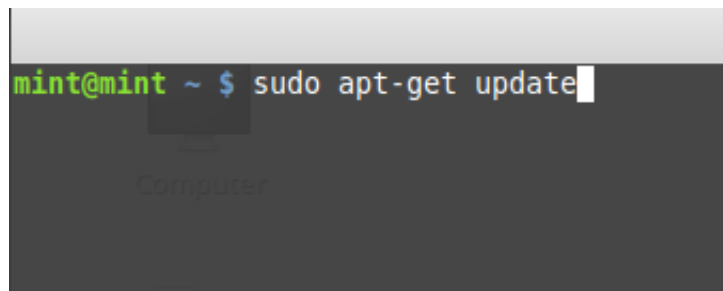
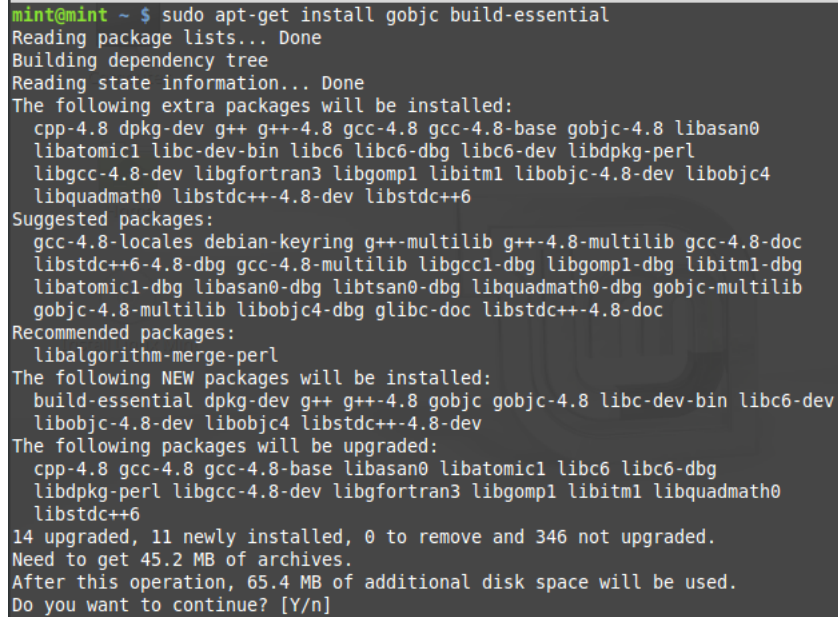


Figura 14: Comando para actualizar el gestor de paquetes apt-get

Descargamos los paquetes que se listan a continuación con el comando

```
sudo apt-get install gobjc build-essential
```

Con el comando anterior estaremos descargando los componentes esenciales para que el compilador gcc pueda compilar nuestro código Objective-C



```
mint@mint ~ $ sudo apt-get install gobjc build-essential
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following extra packages will be installed:
  cpp-4.8 dpkg-dev g++ g++-4.8 gcc-4.8 gcc-4.8-base gobjc-4.8 libasan0
  libatomic1 libc-dev-bin libc6 libc6-dbg libc6-dev libdpkg-perl
  libgcc-4.8-dev libgfortran3 libgomp1 libitm1 libobjc-4.8-dev libobjc4
  libquadmath0 libstdc++-4.8-dev libstdc++6
Suggested packages:
  gcc-4.8-locales debian-keyring g++-multilib g++-4.8-multilib gcc-4.8-doc
  libstdc++6-4.8-dbg gcc-4.8-multilib libgcc1-dbg libgomp1-dbg libitm1-dbg
  libatomic1-dbg libasan0-dbg libtsan0-dbg libquadmath0-dbg gobjc-multilib
  gobjc-4.8-multilib libobjc4-dbg glibc-doc libstdc++-4.8-doc
Recommended packages:
  libalgorithm-merge-perl
The following NEW packages will be installed:
  build-essential dpkg-dev g++ g++-4.8 gobjc gobjc-4.8 libc-dev-bin libc6-dev
  libobjc-4.8-dev libobjc4 libstdc++-4.8-dev
The following packages will be upgraded:
  cpp-4.8 gcc-4.8 gcc-4.8-base libasan0 libatomic1 libc6 libc6-dbg
  libdpkg-perl libgcc-4.8-dev libgfortran3 libgomp1 libitm1 libquadmath0
  libstdc++6
14 upgraded, 11 newly installed, 0 to remove and 346 not upgraded.
Need to get 45.2 MB of archives.
After this operation, 65.4 MB of additional disk space will be used.
Do you want to continue? [Y/n]
```

Figura 15: Comando para instalar los componentes esenciales del compilador gcc para que pueda compilar Objective-C

Le damos que si (Y) y continuará con la instalación.

Ahora, instalamos las librerías propias de Objective-C con el siguiente comando.

```
sudo apt-get install gnustep-devel
```

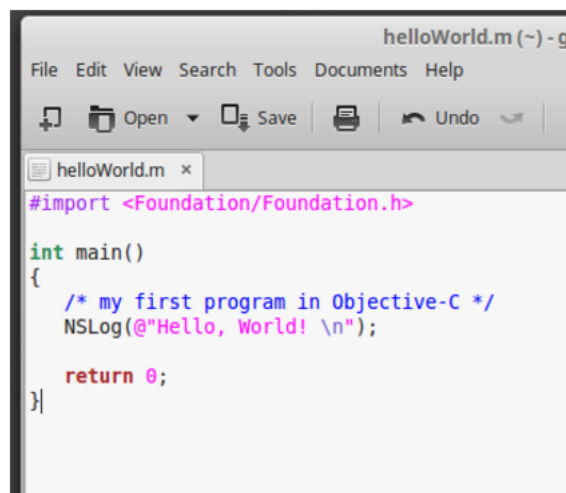
```

mint@mint ~ $ sudo apt-get install gnustep-devel
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following extra packages will be installed:
  autotools-dev gnustep-back-common gnustep-back0.22 gnustep-back0.22-art
  gnustep-base-common gnustep-base-runtime gnustep-common gnustep-core-devel
  gnustep-gpbs gnustep-gui-common gnustep-gui-runtime gnustep-make gorm.app
  imagemagick-common libao-common libao4 libgnustep-base-dev
  libgnustep-base1.24 libgnustep-gui-dev libgnustep-gui0.22 liblqr-1-0
  libmagickcore5 mknfonts.tool projectcenter.app
Suggested packages:
  gnustep-base-doc gnustep-base-examples steptalk libpantomim1.2-dev
  libpopplerkit-dev libnetclasses-dev libaddresses-dev libaddressview-dev
  librsskit-dev gnustep-dl2 gnustep-make-doc libesd0 libesd-alsa0
  gnustep-gui-doc libmagickcore5-extra
Recommended packages:
  gnustep-core-doc librenaissance0-dev
The following NEW packages will be installed:
  autotools-dev gnustep-back-common gnustep-back0.22 gnustep-back0.22-art
  gnustep-base-common gnustep-base-runtime gnustep-common gnustep-core-devel
  gnustep-devel gnustep-gpbs gnustep-gui-common gnustep-gui-runtime
  gnustep-make gorm.app imagemagick-common libao-common libao4
  libgnustep-base-dev libgnustep-base1.24 libgnustep-gui-dev

```

Figura 16: Comando para instalar las librerías propias del lenguaje Objective-C

Probamos nuestro compilador con el archivo helloWorld.m (incluido en la página)



```

helloWorld.m (~) - g
File Edit View Search Tools Documents Help
Open Save Undo
helloWorld.m x
#import <Foundation/Foundation.h>

int main()
{
    /* my first program in Objective-C */
    NSLog(@"Hello, World! \n");

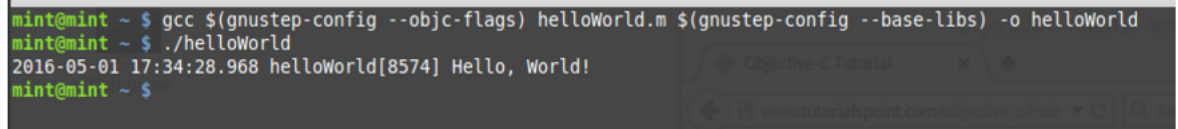
    return 0;
}

```

Figura 17: Código fuente del programa de prueba

Para compilar el código fuente usamos el siguiente comando.

```
gcc $(gnustep-config --objc-flags) helloWorld.m
$(gnustep-config --base-libs) -o helloWorld
```

A terminal window with a dark background. The prompt is 'mint@mint ~ \$'. The first command is 'gcc \$(gnustep-config --objc-flags) helloWorld.m \$(gnustep-config --base-libs) -o helloWorld'. The second command is './helloWorld'. The output is '2016-05-01 17:34:28.968 helloWorld[8574] Hello, World!'. The prompt returns to 'mint@mint ~ \$'.

```
mint@mint ~ $ gcc $(gnustep-config --objc-flags) helloWorld.m $(gnustep-config --base-libs) -o helloWorld
mint@mint ~ $ ./helloWorld
2016-05-01 17:34:28.968 helloWorld[8574] Hello, World!
mint@mint ~ $
```

Figura 18: Comandos para compilar y ejecutar el programa.

La variable de entorno `$(gnustep-config --objc-flags)` tiene las banderas adecuadas para que el compilador gcc tome el código de entrada como código fuente de Objective-C y la variable de entorno `$(gnustep-config --base-libs)` contiene los comandos específicos para que el compilador haga el linkeado de nuestro programa con las librerías de Objective-C. Por último, el comando `-o helloWorld` especifica el nombre del ejecutable.

Para ejecutar el programa, basta con el comando

```
./helloWorld
```

En la figura 14, se puede ver el resultado de la ejecución.

2. Primeros pasos con Objective-C

En su editor de texto favorito, crearemos un archivo llamado `helloWorld.m`. En él, escribiremos el código fuente de nuestro primer programa, será un programa muy modesto pero bastante ilustrativo de algunos aspectos claves del lenguaje. Se trata de un programa que muestra por pantalla "Hello World!"

Copiaremos el siguiente código en nuestro editor y lo guardamos en `helloWorld.m`

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

int main()
{
    /* my first program in Objective-C */
    NSLog(@"Hello, World! \n");
}
```

```
    return 0;
}
```

El siguiente paso, es compilar el código fuente como se indicó en la sección de instalación y según nuestro sistema operativo, una vez hecho esto, ejecutamos nuestro programa y el resultado deberá ser muy similar al siguiente

```
2016-04-29 14:28:42.176 HelloWorld[12242] Hello, World!
```

Ahora, explicaremos que significan cada una de las componentes del código fuente del programa.

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

La primera instrucción con la que nos encontramos es una sentencia de tipo *import*. Esta sentencia incluirá en nuestro código fuente, todo el código que se encuentre en el archivo *Foundation.h* es decir, esa línea de código, se reemplazará por las muchas líneas que pueda tener el archivo *Foundation.h* y si a su vez este tuviera más sentencias *import* (efectivamente así es) se reemplazarán de nuevo por el código de los correspondientes archivos referenciados.

El archivo *Foundation.h* define la capa base de las clases de Objective-C. Además, provee un conjunto de objetos y clases primitivas útiles y define funcionalidades y paradigmas que no están cubiertas por el lenguaje en sí.

```
int main() {}
```

El programa comienza a ejecutarse desde la primera instrucción que se encuentre dentro del cuerpo de la función *main*. Es decir, esta instrucción define el punto de entrada al programa y este se ejecutará desde ahí línea a línea de arriba a abajo.

```
/* my first program in Objective-C */
```

Esta línea contiene lo que se denomina un comentario. Un comentario es una serie de caracteres que no serán tomados en cuenta a la hora de la compilación, es decir, son simplemente ignorados. Estos comentarios sirven para aclarar qué es lo que está haciendo el código si este es muy complejo y difícil de entender.

En Objective-C un comentario se puede hacer colocándolo entre */** y **/*

```
NSLog(@"Hello, World! \n");
```

Con la instrucción anterior, le estamos diciendo al computador que queremos que muestre por pantalla la cadena de texto “Hello World!” un espacio y un salto de línea

Es necesario que coloquemos una @ (arroba) antes de una cadena de texto, para convertirla al tipo NSString (el tipo de cadena predeterminado de Objective-C)

Esa instrucción es la responsable que el mensaje ”Hello World!” se muestre por pantalla

```
return 0;
```

Por último, la instrucción anterior, le informa al sistema operativo que el programa ha finalizado correctamente y regresa el control al programa padre. En este caso, la consola del sistema operativo.

2.1. Tipos de datos

Objective-C cuenta con bastantes tipos de datos que pueden ser utilizados pero se pueden clasificar básicamente en 4 categorías.

2.1.1. Tipos básicos

Son tipos aritméticos y se componen de dos tipos: enteros y flotantes. Para los enteros se tiene:

Tipo	Tamaño	Rango de valores
char	1 byte	-128 - 127 o 0 - 255
unsigned char	1 byte	0 - 255
signed char	1 byte	-128 - 127
int	2 or 4 bytes	-32,768 - 32,767 o -2,147,483,648 - 2,147,483,647
unsigned int	2 or 4 bytes	0 - 65,535 o 0 - 4,294,967,295
short	2 bytes	-32,768 - 32,767
unsigned short	2 bytes	0 - 65,535
long	4 bytes	-2,147,483,648 - 2,147,483,647
unsigned long	4 bytes	0 - 4,294,967,295

Figura 19: Tabla de tipos enteros

Para los flotantes se tiene:

Tipo	Tamaño	Rango de valores	Precision
float	4 byte	1.2E-38 - 3.4E+38	6 lugares decimales
double	8 byte	2.3E-308 - 1.7E+308	15 lugares decimales
long double	10 byte	3.4E-4932 - 1.1E+4932	19 lugares decimales

Figura 20: Tabla de tipos flotantes

La sintaxis para declarar una variable de estos tipos es ejemplificada a continuación:

```
int a;
char c = 'a';
float d, b, i = 2.0;
```


2.1.2. Tipos enumerados

También son tipos aritméticos y se utilizan para definir las variables a las que sólo se pueden asignar determinados valores enteros discretos en todo el programa.

```
enum direction {north, south = 5, west, east};
int cur = south;
NSLog(@"%d ", cur);
NSLog(@"%d ", north);
NSLog(@"%d", west);
```

Básicamente lo que se hace es que se crean unas variables north, south, west, east, que están almacenadas en direction (no es necesario declararlas antes), e inicialmente tienen un valor equivalente a la posición en la que estén, por ejemplo north tendrá un valor de 0, también se les puede asignar valores, como en el caso de south. Para el caso de west como no se le asigna un valor directamente, se le asigna el valor de la última variable asignada (que es south) y se le suma 1, es decir queda con un valor de 6. También se puede acceder al valor que tiene una variable por medio de la variable de tipo **enum**. Entonces para este caso en una variable cur, se guardara el valor de south. Los valores que se almacenan en las variables enumeradas, solo pueden ser enteros. La salida del programa anterior seria *5 0 6*.

2.1.3. Tipo vacío

El tipo vacío especifica que no hay un valor disponible. Se usa en 2 situaciones.

- Argumentos de una función

```
int suma(void);
```

- Funciones que no retornan algo (Procedimientos)

```
void suma(int a, int b);
```

2.1.4. Tipos derivados

Son principalmente los siguientes:

- Punteros:

Un puntero es una variable cuyo valor es la dirección de otra variable, es decir, la dirección directa de la posición de memoria. Al igual que cualquier variable o constante, debe declarar un puntero (*) antes de poder utilizarlo para almacenar cualquier dirección variable. La forma general de una declaración de variable puntero es:

```
tipo *nombre-variable;
```

A continuación se muestra un ejemplo del uso de punteros.

```
#import <Foundation/Foundation.h>
int main ()
{
    int var = 20;    /* declaracion de una variable */
    int *ip;         /* declaracion de un puntero */

    ip = &var;      /*Guardar direccion de var en ip*/

    NSLog(@"Direccion de memoria de var: %x\n", &var );
    NSLog(@"Direccion guardada en ip: %x\n", ip );
    /* Acceder al valor usando el puntero ip */
    NSLog(@"Value of *ip variable: %d\n", *ip );

    return 0;
}
```

El resultado sería el siguiente:

```
2016-05-01 19:30:33.170 helloworld[10264] Direccion de memoria de var: 22ff1c
2016-05-01 19:30:33.170 helloworld[10264] Direccion guardada en ip: 22ff1c
2016-05-01 19:30:33.170 helloworld[10264] Value of *ip variable: 20
```

Figura 21: Salida ejemplo de punteros

- Arreglos:

Es una estructura de datos que nos permite almacenar un tamaño fijo de elementos del mismo tipo.

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

int main(int argc, const char * argv[]) {
```

```

    int n[ ] = { 11, 22, 33, 44 }; /* n is an array of 4 integers with the values 11, 22, 33, 44 */

    /* initialize elements of array n to 0 */
    for( int i = 0; i < 4; i++ ) {
        n[ i ] = n[ i ]+100; /* set element at location i to n[ i ] + 100 */
    }

    /* output each array element's value */
    for( int j = 0; j < 4; j++ ) {
        NSLog(@"Element[%d] = %d\n", j, n[j] );
    }

    return 0;
}

```

Salida:

```

2016-05-01 23:49:50.430 Main[1138:208420] Element[0] = 111
2016-05-01 23:49:50.431 Main[1138:208420] Element[1] = 122
2016-05-01 23:49:50.431 Main[1138:208420] Element[2] = 133
2016-05-01 23:49:50.431 Main[1138:208420] Element[3] = 144

```

Otra cosa a tener en cuenta y de gran importancia es la manera en que podemos enviar arreglos como parámetros de una función y así mismo poder retornar un arreglo. Para lo anterior la manera en que le indicaremos al programa que reciba un arreglo o retorne un arreglo es:

```
( type * )
```

A continuación se especifica un ejemplo, en donde se recibe como parámetro un arreglo y se retorna otro arreglo que equivale al original ordenado.

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

```

@interface Utility:NSObject
+ ( void ) swap: ( int * ) a B: ( int * ) b;
+ ( int * ) sort_array: ( int * ) arr Size: ( int ) size;
@end

```

```

@implementation Utility
+ ( void ) swap: ( int * ) a B: ( int * ) b {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
+ ( int * ) sort_array: ( int * ) arr Size: ( int ) size {
    int ret[ size ];
    int *p = ret;
    for( int i = 0; i < size; i++ )
        *( p+i ) = *( arr+i );
    for( int i = 0; i < size; i++ ) {
        for( int j = i+1; j < size; j++ ) {
            if( *( p+i ) > *( p+j ) ) {
                [ Utility swap: (p+i) B: ( p+j ) ];
            }
        }
    }
    return p;
}
@end

int main(int argc, const char * argv[]) {

    @autoreleasepool {
        int size = 5;
        int arr[ ] = { 5, 1, 3, 7, 4 };
        int *ans = [ Utility sort_array: arr Size: size ];
        NSLog( @"The original array is:\n" );
        for( int i = 0; i < size; i++ ) {
            NSLog( @"arr[ %d ] = %d\n", i, *( arr+i ) );
        }
        NSLog( @"The sorted array is:\n" );
        for( int i = 0; i < size; i++ ) {
            NSLog( @"ans[ %d ] = %d\n", i, *( ans+i ) );
        }
    }

    return 0;
}

```

```
}
```

Obtenemos como salida:

```
2016-05-02 01:23:40.598 Main[2165:722155] The original array is:
2016-05-02 01:23:40.599 Main[2165:722155] arr[ 0 ] = 5
2016-05-02 01:23:40.599 Main[2165:722155] arr[ 1 ] = 1
2016-05-02 01:23:40.599 Main[2165:722155] arr[ 2 ] = 3
2016-05-02 01:23:40.599 Main[2165:722155] arr[ 3 ] = 7
2016-05-02 01:23:40.599 Main[2165:722155] arr[ 4 ] = 4
2016-05-02 01:23:40.599 Main[2165:722155] The array sorted is:
2016-05-02 01:23:40.599 Main[2165:722155] ans[ 0 ] = 1
2016-05-02 01:23:40.599 Main[2165:722155] ans[ 1 ] = 3
2016-05-02 01:23:40.599 Main[2165:722155] ans[ 2 ] = 4
2016-05-02 01:23:40.599 Main[2165:722155] ans[ 3 ] = 5
2016-05-02 01:23:40.599 Main[2165:722155] ans[ 4 ] = 7
```

- Estructuras:

Una estructura es otro tipo de datos definido por el usuario, que permite combinar distintos tipos de datos en uno solo. A continuación se muestra un ejemplo de cómo se declara y se manejan las estructuras.

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

struct date {int month; int day; int year;};

int main() {
    struct date tomorrow = {6, 22, 2016};
    struct date today;

    today.month = 5;
    today.day = 22;
    today.year = 2016;

    NSLog(@"Today's date is %i/%i/%i", today.month,
          today.day, today.year);
    NSLog(@"Tomorrow's date is %i/%i/%i", tomorrow.month,
          tomorrow.day, tomorrow.year);

    return 0;
}
```

En una estructura se pueden almacenar todo tipo de datos, incluso otro tipo estructura. Esto se asemeja un poco al concepto de clases, pero se diferencian en bastantes cosas, por ejemplo el hecho que una estructura no tiene un constructor. El resultado del ejemplo anterior es el siguiente:

```
$ ./helloworld.exe
2016-05-01 18:49:25.016 helloworld[8248] Today's date is 5/22/2016
2016-05-01 18:49:25.112 helloworld[8248] Tomorrow's date is 6/22/2016
```

Figura 22: Salida ejemplo de estructuras

- Uniones:

Las uniones son una estructura de datos que le permiten almacenar más de un tipo de dato en el mismo campo de memoria. Las uniones son similares a las estructuras, con diferencia que solo manejan un campo de memoria, es decir si una variable puede tener 3 posibles valores, no usara 3 campos como una estructura, sino 1 solo. La variable que sea de tipo union solo puede almacenar 1 de los 3 valores al tiempo, es por eso que cuando se le asigna un valor, se desecha el anterior que tenía.

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

union mixed {int i; float f; char c;};

int main() {

    union mixed x;
    x.i = 5;
    NSLog(@"%d", x.i);
    x.f = 2.5;
    NSLog(@"%f", x.f);
    x.c = 'A';
    NSLog(@"%c", x.c);

    return 0;
}
```

Si después de asignar un valor, intentamos consultar un valor diferente al que asignamos, nos saldrá un valor sin sentido. La salida para este ejemplo seria la siguiente:

```
$ ./helloworld.exe
2016-05-01 18:08:38.375 helloworld[5468] 5
2016-05-01 18:08:38.375 helloworld[5468] 2.500000
2016-05-01 18:08:38.395 helloworld[5468] a
```

Figura 23: Ejemplo de salida union

2.2. Operadores

Objective-C cuenta con una gran variedad de operadores que permiten llevar a cabo determinadas manipulaciones lógicas o matemáticas. Podemos distinguir 6 categorías.

2.2.1. Operadores aritméticos

Operador	Descripción
+	Suma dos operandos
-	Resta el segundo operando del primero
*	Multiplica dos operandos
/	Divide el primer operando entre el segundo
%	Arroja el residuo de una división entre enteros
++	Incrementa en uno una variable entera
--	Decrementa en uno una variable entera

Figura 24: Tabla de operadores aritméticos

2.2.2. Operadores relacionales

Los operadores relacionales permiten comparar variables retornando 'true' en caso de que la condición se cumpla, 'false' de lo contrario. En general cualquier valor distinto a 0 puede considerarse como un valor verdadero.

Operador	Descripción
&&	Chequea si dos operandos son iguales
	Chequea si dos operandos son diferentes
!	Chequea si el primer operando es mayor que el segundo
<	Chequea si el primer operando es menor que el segundo
>=	Chequea si el primer operando es mayor o igual que el segundo
<=	Chequea si el primer operando es menor o igual que el segundo

Figura 25: Tabla de operadores relacionales

2.2.3. Operadores lógicos

Operador	Descripción
&&	Si ambos operandos son distintos de cero(falso), retorna verdadero
	Si alguno de los operandos es distinto de cero(falso), retorna verdadero
!	Niega una condición lógica

Figura 26: Tabla de operadores lógicos

2.2.4. Operadores bit a bit

Los operadores bit a bit nos permiten interactuar con la representación interna de un número que usa el computador.

Operador	Descripción
&	Arroja un '1' en el resultado si los dos operandos están prendidos
	Arroja un '1' en el resultado si los alguno de los operandos está encendido
^	Arroja un '1' si los dos operandos son bits distintos
~	Arroja la negación de un bit
<<	Mueve los bits del primer operando tantas posiciones a la izquierda como lo indique el segundo operando
>>	Mueve los bits del primer operando tantas posiciones a la derecha como lo indique el segundo operando

Figura 27: Tabla de operadores bit a bit

2.2.5. Operadores de asignación

Operador	Descripción
=	Asigna los valores del lado derecho al lado izquierdo
+=	Agrega los valores del lado derecho al lado izquierdo y el resultado se lo asigna
-=	Resta los valores del lado derecho al lado izquierdo y el resultado se lo asigna
*=	Multiplica los valores del lado derecho al lado izquierdo y el resultado se lo asigna
/=	Divide los valores del lado derecho al lado izquierdo y el resultado se lo asigna
%=	Halla el residuo entre los valores del lado izquierdo y los del derecho, el resultado es asignado al operando del lado izquierdo
<<=	Ejecuta el corrimiento a la izquierda y se lo asigna al operando
>>=	Ejecuta el corrimiento a la derecha y se lo asigna al operando
&=	Ejecuta el AND bit a bit y se lo asigna al operando
^=	Ejecuta el XOR bit a bit y se lo asigna al operando
=	Ejecuta el OR bit a bit y se lo asigna al operando

Figura 28: Tabla de operadores de asignación

2.2.6. Otros operadores

Operador	Descripción
&	Retorna la dirección de la variable
*	Apuntador a una variable

Figura 29: Tabla de otros operadores

2.3. Condicionales

Son sentencias que de acuerdo al valor de verdad que pueda tomar una condición permitirán ejecutar un bloque de código.

2.3.1. Sentencia if

Consiste en una condición que si toma el valor de verdad verdadero ejecutará un bloque de código.

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

int main(int argc, const char * argv[]) {
    /* local variable definition */
    int a = 10;

    /* check the boolean condition using if statement */
    if( a < 20 ) {
        /* if condition is true then print the following */
        NSLog(@"a = %d is less than 20\n", a);
    }
    return 0;
}
```

Al ejecutar el código anterior obtenemos:

```
2016-05-01 23:18:49.524 Main[811:85183] a = 10 is less than 20
```

2.3.2. Sentencia if-else

Consiste en una condición que al ser evaluada si es verdadera ejecuta un bloque de código específico de lo contrario ejecutará otro bloque de código, nunca se ejecutan ambos.

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

int main(int argc, const char * argv[]) {
    /* local variable definition */
    int a = 100;

    /* check the boolean condition using if statement */
    if( a < 20 ) {
        /* if condition is true then print the following */
        NSLog(@"a = %d is less than 20\n", a);
    }
    else {
```

```

        /* if condition is false then print the following */
        NSLog(@"a = %d is not less than 20\n", a );
    }
    return 0;
}

```

Al ejecutarlo podemos ver:

```
2016-05-01 23:25:47.678 Main[883:105048] a = 100 is not less than 20
```

2.3.3. Sentencia switch

La sentencia switch permite tomar una variable y testearla con unos valores especificados llamados casos, que cuando coincidan se ejecutará de ahí hasta que encuentre una sentencia **break** o el fin de la sentencia switch.

```

#import <Foundation/Foundation.h>

int main(int argc, const char * argv[]) {
    /* local variable definition */
    char grade = 'B';
    switch( grade ) {
        case 'A':
            NSLog(@"Excellent!\n" );
            break;
        case 'B':
        case 'C':
            NSLog(@"Well done\n" );
            break;
        case 'D':
            NSLog(@"You passed\n" );
            break;
        case 'F':
            NSLog(@"Better try again\n" );
            break;
        default:
            NSLog(@"Invalid grade\n" );
    }
    NSLog(@"Your grade is %c\n", grade );
    return 0;
}

```

Obtenemos

```
2016-05-01 23:27:32.501 Main[912:115675] Well done
2016-05-01 23:27:32.501 Main[912:115675] Your grade is B
```

2.4. Bucles

Los bucles son estructuras de control que permiten ejecutar varias veces un bloque de código secuencialmente, hasta que una determinada condición no se cumpla.

2.4.1. Bucle While

Repite la ejecución de un bloque de código hasta que la condición sea falsa. Antes de poder ejecutar el bloque es necesario haber evaluado la condición.

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

int main( int argc, const char * argv[ ] ) {
    /* local variable definition */
    int a = 18;

    /* while loop execution */
    while( a < 20 ) {
        NSLog(@"value of a: %d\n", a);
        a++;
    }

    return 0;
}
```

Obtenemos:

```
2016-05-01 22:51:17.691 Main[521:14014] value of a: 18
2016-05-01 22:51:17.692 Main[521:14014] value of a: 19
```

2.4.2. Bucle Do-While

La particularidad de este bucle es que siempre ejecuta por lo menos una vez el bloque de código que contiene.

```

#import <Foundation/Foundation.h>

int main( int argc, const char * argv[ ] )
{
    /* local variable definition */
    int a = 20;

    /* do loop execution */
    do {
        NSLog(@"value of a: %d\n", a);
        a = a+1;
    }while( a < 20 );

    return 0;
}

```

Obtenemos:

```
2016-05-01 22:55:22.823 Main[564:16709] value of a: 20
```

2.4.3. Bucle For

El bucle for permite escribir eficientemente un ciclo que se necesita que se ejecute un número específico de veces.

```

#import <Foundation/Foundation.h>

int main( ) {
    /* for loop execution */
    for( int a = 15; a < 20; a++ ) {
        NSLog(@"value of a: %d\n", a);
    }
    return 0;
}

```

Obtenemos:

```

2016-05-01 23:00:09.147 Main[611:18792] value of a: 15
2016-05-01 23:00:09.148 Main[611:18792] value of a: 16
2016-05-01 23:00:09.148 Main[611:18792] value of a: 17
2016-05-01 23:00:09.148 Main[611:18792] value of a: 18
2016-05-01 23:00:09.148 Main[611:18792] value of a: 19

```

2.4.4. Declaraciones para el control de bucles

- Break:
Termina la ejecución de un bucle o switch, y se traslada directamente a la siguiente declaración después del ciclo.
- Continue:
Descarta el resto del bloque del ciclo y se dirige a la cabecera del ciclo para testear de nuevo la condición.

3. Particularidades de Objective-C

3.1. Clase envolvente (Números)

NSNumber es una clase envolvente dirigida a objetos, que se encarga de los datos primitivos. Su función principal es almacenar los valores primitivos como si fueran objetos y para esto tiene un método para cada tipo primitivo.

```
NSNumber *A = [NSNumber numberWithInt:NO];  
NSNumber *B = [NSNumber numberWithChar:'z'];  
NSNumber *C = [NSNumber numberWithUnsignedChar:255];  
NSNumber *D = [NSNumber numberWithShort:32767];  
NSNumber *E = [NSNumber numberWithUnsignedShort:65535];  
NSNumber *F = [NSNumber numberWithInt:2147483647];  
NSNumber *G = [NSNumber numberWithUnsignedInt:4294967295];  
NSNumber *H = [NSNumber numberWithLong:9223372036854775807];  
NSNumber *I = [NSNumber numberWithUnsignedLong:18446744073709551615];  
NSNumber *J = [NSNumber numberWithFloat:26.99f];  
NSNumber *K = [NSNumber numberWithDouble:26.99];
```

Puede parecer redundante tener una versión orientada a objetos de todos los datos primitivos, pero es necesario ya que si uno quiere almacenar valores numéricos en un NSArray, NSDictionary, o en alguno de las colecciones que existen, necesariamente deben ser objetos, estas colecciones no saben cómo manejar datos primitivos. Pero también tiene algunas ventajas. Una de estas es que es más fácil representar un número en forma de NSString (o castear de un tipo a otro) usando la representación del valor:

```
NSNumber *comoInt = [NSNumber numberWithInt:42];  
float comoFloat = [comoInt floatValue];  
NSLog(@"%.2f", comoFloat);
```

```
NSString *comoString = [comoInt stringValue];
NSLog(@"%@", comoString);
```

Existe otra manera de asignar valores a un NSNumber y es la manera literal, que es presentada a continuación (U para unsigned, L para long, F para float):

```
NSNumber *A = @0;
NSNumber *B = @'z';
NSNumber *C = @2147483647;
NSNumber *D = @4294967295U;
NSNumber *E = @9223372036854775807L;
NSNumber *F = @26.99F;
NSNumber *G = @26.99;
double x = 24.0;
NSNumber *result = @(x * .15);
NSLog(@"%.2f", [result doubleValue]);
```

Es importante entender que NSNumber es **INMUTABLE** es decir que no se puede cambiar su valor. En este sentido, esta clase actúa como los datos primitivos, ya que si se desea asignar un nuevo valor, internamente se está creando una nueva instancia. Para comparar dos números se usa isEqualToNumber el cual es un método que devuelve unos valores específicos como resultado.

```
NSNumber *anInt = @27;
NSNumber *sameInt = @27U;
// Comparacion de direcciones
if (anInt == sameInt) {
    NSLog(@"Son los mismos objetos");
}
// Comparacion de valores
if ([anInt isEqualToNumber:sameInt]) {
    NSLog(@"Tienen el mismo valor");
}
```

La función isEqualToNumber devuelve 3 tipos de valores, NSOrderedAscending en caso de que el primer argumento sea menor que el segundo, NSOrderedSame, en caso de que sean iguales y NSOrderedDescending en caso que el primer argumento sea mayor al segundo.


```

NSNumber *anInt = @27;
NSNumber *anotherInt = @42;
NSComparisonResult result = [anInt compare:anotherInt];
if (result == NSOrderedAscending) {
    NSLog(@"27 < 42");
} else if (result == NSOrderedSame) {
    NSLog(@"27 == 42");
} else if (result == NSOrderedDescending) {
    NSLog(@"27 > 42");
}

```

3.2. Cadenas

La clase NSString es la clase básica para representar texto. NSString ofrece varios métodos para la manipulación de su contenido. NSString también es un tipo **INMUTABLE**. Pero existe una contraparte llamada NSMutableString, la cual es evidentemente mutable. La forma más común de crear un string es usando la sintaxis @.Alguna cadena". Pero stringWithFormat es útil también para generar cadenas, sobre todo para cadenas que están compuestas por valores de otras variables. Puede tomar el mismo formato que se usa para NSLog().

```

NSString *make = @"Porsche";
NSString *model = @"911";
int year = 1968;
NSString *message = [NSString stringWithFormat:
    @"That's a %@ %@ from %d!", make, model, year];
NSLog(@"%@", message);

```

Los dos principales métodos de NSString son **length** y **characterAtIndex**, los cuales retornan el número de caracteres en la cadena, y un carácter en la posición de la cadena, respectivamente.

```

NSString *make = @"Porsche";
for (int i=0; i<[make length]; i++) {
    char letter = [make characterAtIndex:i];
    NSLog(@"%d: %c", i, letter);
}

```

Comparaciones con NSString tiene los mismos problemas que NSNumber. Siempre se debe usar isEqualToString para comparar los valores de 2

cadenas. El siguiente ejemplo muestra el uso de este, también se muestra el uso de **hasPrefix** y **hasSuffix** para comparaciones parciales, o por pedazos.

```
NSString *car = @"Porsche Boxster";
if ([car isEqualToString:@"Porsche Boxster"]) {
    NSLog(@"El carro es un Porsche Boxster");
}
if ([car hasPrefix:@"Porsche"]) {
    NSLog(@"El carro comienza con Porsche");
}
if ([car hasSuffix:@"Carrera"]) {
    NSLog(@"El carro termina en Carrera");
}
```

Para la comparación de menor, mayor o igual, se hace de la misma manera que se hizo con `NSNumber` en la sección anterior.

3.3. Funciones

Las funciones son un conjunto de sentencias que unidas permiten llevar a cabo una tarea en específico. Para funciones fuera de objetos la declaración y llamado es como en C:

```
//Function declaration
return_type function_name( argument_type_1 argument_name_1, ... ,
                           argument_type_n argument_name_n ) {
    body of the function
}
//Function call
function_name( arg1, ... , argn );
```

Pero cuando nos referimos a funciones que corresponden a los comportamientos de un objeto(en realidad para este caso se conocen como métodos) es diferente su declaración. A continuación se especifica:

```
+/- (return_type) method_name:( argumentType1 )argumentName1
    joiningArgument2:( argumentType2 )argumentName2 ...
    joiningArgumentn:( argumentTypen )argumentNamen {
    body of the function
}
```

Ahora vamos a ver en específico cada parte para la declaración de un método:

- `+/-`:
El `+` nos especificará un método estático, que es cuando tal comportamiento es común para todos los elementos de la clase y por lo tanto para llamarlo no necesitaremos un objeto, mientras que cuando especificamos `-` estamos diciendo que para utilizar ese método sólo se puede declarando un objeto de la clase.
- Tipo de retorno:
Aquí especificaremos que va a retornar nuestra función, en caso de que no vamos a retornar nada especificamos la palabra reservada *void*.
- Argumento de unión:
Nos facilita la lectura del código y adicionalmente cuando vayamos a llamar el método va a ser más claro que argumentos debemos enviar. Es como una etiqueta de cada argumento.

A continuación ilustramos estos conceptos con un pequeño ejemplo:

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

@interface Math:NSObject
+( int ) max: ( int ) a B: ( int ) b;
@end

@implementation Math
+( int ) max: ( int ) a B: ( int ) b {
    int result = a;
    if( b > a )
        result = b;
    return result;
}
@end

int main(int argc, const char * argv[]) {

    @autoreleasepool {
        int a = 10, b = 120;
        NSLog( @"Max value between ( %d, %d ) is: %d\n",
```

```

        a, b, [ Math max: a B: b ] );
    }

    return 0;
}

```

El resultado que obtenemos es:

```

2016-05-02 03:01:59.121 Main[3251:1015437]
    Max value between ( 10, 120 ) is: 120

```

3.4. Colecciones

3.4.1. Arreglos - Listas

Hay dos tipos de listas en Objective-C: inmutables y mutables. El contenido de una lista *Immutable* no puede cambiar en tiempo de ejecución. Las listas inmutables se crean como instancia de la clase **NSArray**. Las listas *Mutable* se crean mediante la clase **NSMutableArray** (una subclase de **NSArray**) y puede ser modificada después de haber sido creado e inicializado.

```

NSArray *arr = [NSArray arrayWithObjects: @"Colombia", @"1",
        @"Lenguajes", nil];
int i, size = [arr count];

for(i = 0; i < size; i++) {
    NSLog(@"posicion %d = %@ ", i, [arr objectAtIndex: i]);
}

```

Cuando se crea una lista es necesario crearla como un apuntador (*). Para calcular el tamaño del arreglo se usa el método **count**, para llamarlo se encierra entre [] y se pone el nombre del arreglo y el nombre del método, en caso que el método reciba argumentos, se pone dos puntos después del método y se le manda el argumento correspondiente. Los **NSMutableArray** funcionan como un arreglo con tamaño dinámico, es decir que se pueden agregar y borrar elementos.

Para inicializar una lista vacía se hace de la siguiente manera:

```

NSArray *arr = [NSArray new]

```

Si se quiere inicializar una lista explícitamente con unos valores se puede hacer de la siguiente forma:

```
NSArray *arr = [NSArray arrayWithObjects: @"Colombia", @"1",
                                             [NSNumber numberWithInt:22], @"Lenguajes", nil];
```

Es importante cuando se inicializa una lista explícitamente poner **nil** en una posición, para indicar en donde termina la lista. Como se dijo anteriormente las listas NSArray son inmutables, por lo cual no se puede modificar su contenido, al contrario de NSMutableArray la cual si se puede. Para este último existen varios métodos que ayudan a modificar los valores de la lista, entre los cuales se muestra **removeObject**, **removeObjectAtIndex** y **addObject**

```
[myArray removeObject: @"1"];
[myArray removeObjectAtIndex: 0];
[myArray addObject: @"Hola"];
```

Para comparar si dos listas son iguales, basta con usar la función **isEqualToArray** la cual recibe dos parámetros (las dos listas). Esta función compara el tamaño de ambas listas, y los elementos que contiene cada una de las dos listas, comparando las posiciones 0 con 0, 1 con 1, y así sucesivamente.

```
if ([arr isEqualToArray:myArray]) {
    NSLog(@"Wow, ambos arreglos son identicos");
}
```

A continuación se muestra un código simple en el que se hace uso de las listas.

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

int main() {
    NSAutoreleasePool *pool = [NSAutoreleasePool new];

    NSArray *arr = [NSArray arrayWithObjects: @"Colombia", @"1",
                                             [NSNumber numberWithInt:22], @"Lenguajes", nil];
    NSMutableArray *myArray = [NSMutableArray new];
    int i, size = [arr count];

    for(i = 0; i < size; i++) {
        NSLog(@"posicion %d = %@", i, [arr objectAtIndex: i]);
        [myArray addObject: [arr objectAtIndex: i]];
    }
}
```

```

[myArray removeObject: @"1"];
[myArray removeObjectAtIndex: 0];

size = [myArray count];
NSLog(@"\n");
for(i = 0; i < size; i++) {
    NSLog(@"posicion %d = %@", i, [myArray objectAtIndex: i]);
}

if ([arr isEqualToArray:myArray]) {
    NSLog(@"Wow, ambos arreglos son identicos");
}
[pool drain];
return 0;
}

```

El resultado de ejecutar este programa seria el siguiente:

```

2016-05-01 23:41:23.003 helloworld[10856] posicion 0 = Colombia
2016-05-01 23:41:23.006 helloworld[10856] posicion 1 = 1
2016-05-01 23:41:23.006 helloworld[10856] posicion 2 = 22
2016-05-01 23:41:23.006 helloworld[10856] posicion 3 = Lenguajes
2016-05-01 23:41:23.006 helloworld[10856]
2016-05-01 23:41:23.006 helloworld[10856] posicion 0 = 22
2016-05-01 23:41:23.006 helloworld[10856] posicion 1 = Lenguajes

```

Figura 30: Ejemplo de salida NSMutableArray

3.4.2. Sets

Un NSSet es otra de las colecciones de Objective-C que representa un conjunto no ordenado de distintos elementos. Los NSSet son usados más que todo para chequear si un objeto está en un conjunto con unas características, cosa que no puede hacerse con un NSArray eficientemente. Esta colección es **IMUTABLE** pero permite modificar los elementos que almacena.

Para crear un NSSet se puede inicializar con el método setWithObjects en el cual se ponen explícitamente los elementos, y al final se pone **nil**, o bien se puede recibir como argumento un NSArray para iniciarlo con los valores de este.

```

NSSet *americanMakes = [NSSet setWithObjects:@"Chrysler", @"Ford",
    @"General Motors", nil];
NSArray *japaneseMakes = @[@"Honda", @"Mazda", @"Mitsubishi", @"Honda"];

```

```
// Toma los valores repetidos y los deja 1 sola vez
NSSet *uniqueMakes = [NSSet setWithArray:japaneseMakes];
```

La manera más fácil para recorrer un set es haciendo uso del for-each, lo único que hacemos es iterar por cada uno de los elementos del set, y ponemos un tipo genérico **id** como tipo de dato que contiene el set, a continuación se muestra un ejemplo:

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

int main() {
    NSAutoreleasePool *pool = [NSAutoreleasePool new];
    NSSet *models = [NSSet setWithObjects:@"Civic", @"Accord", @"Odyssey",
        @"Pilot", @"Fit", nil];
    NSLog(@"El set tiene %li elementos", [models count]);
    for (id item in models) {
        NSLog(@"%@", item);
    }

    [pool drain];
    return 0;
}
```

Y la salida seria la siguiente:

```
2016-05-02 04:38:15.373 helloworld[8920] El set tiene 5 elementos
2016-05-02 04:38:15.376 helloworld[8920] Accord
2016-05-02 04:38:15.377 helloworld[8920] Civic
2016-05-02 04:38:15.378 helloworld[8920] Fit
2016-05-02 04:38:15.378 helloworld[8920] Pilot
2016-05-02 04:38:15.379 helloworld[8920] Odyssey
```

Figura 31: Ejemplo de salida NSSet

Hay varios tipos de comparaciones que se pueden hacer con los NSSet, las cuales se muestran a continuación:

```
NSSet *japaneseMakes = [NSSet setWithObjects:@"Honda", @"Nissan",
    @"Mitsubishi", @"Toyota", nil];
NSSet *johnsFavoriteMakes = [NSSet setWithObjects:@"Honda", nil];
NSSet *marysFavoriteMakes = [NSSet setWithObjects:@"Toyota",
    @"Alfa Romeo", nil];

if ([johnsFavoriteMakes isEqualToSet:japaneseMakes]) {
```

```

        // Se compara si el contenido de John es igual al de los japoneses
        NSLog(@"John comparte todo lo de los japoneses");
    }
    if ([johnsFavoriteMakes intersectsSet:japaneseMakes]) {
        // Existe un elemento en común entre los japoneses y John
        NSLog(@"John tiene algo en común con los japoneses");
    }
    if ([johnsFavoriteMakes isSubsetOfSet:japaneseMakes]) {
        // Se mira si John es un subconjunto de los japoneses
        NSLog(@"john está incluido en los japoneses");
    }
    if ([marysFavoriteMakes isSubsetOfSet:japaneseMakes]) {
        // Se mira si Mary es un subconjunto de los japoneses
        NSLog(@"Mary está incluida en los japoneses");
    }
}

```

Ahora para revisar si un elemento hace parte de un set, se puede usar el método `containsObject` el cual revisa en el set si un objeto especificado se encuentra allí. O bien se puede usar el método `member`, la diferencia es que `member` me devuelve un objeto como tal, puede ser `nil`, y `containsObject`, me devuelve un valor booleano.

```

NSSet *selectedMakes = [NSSet setWithObjects:@"Maserati", @"Porsche", nil];
if ([selectedMakes containsObject:@"Maserati"]) {
    NSLog(@"Maserati esta en selectedMakes");
}
NSString *result = [selectedMakes member:@"Maserati"];
if (result != nil) {
    NSLog(@"%@ es uno de los SelectedMakes", result);
}

```

Otra función útil de los sets, es unir dos conjuntos, combinar la información. Esto se hace mediante el uso del método `setByAddingObjectsFromSet`, a continuación un ejemplo

```

NSSet *affordableMakes = [NSSet setWithObjects:@"Ford", @"Honda",
    @"Nissan", @"Toyota", nil];
NSSet *fancyMakes = [NSSet setWithObjects:@"Ferrari", @"Maserati",
    @"Porsche", nil];
NSSet *allMakes = [affordableMakes setByAddingObjectsFromSet:fancyMakes];
NSLog(@"%@", allMakes);

```


y la salida seria la siguiente:

```
2016-05-02 05:13:58.009 helloworld[13204] <Nissan, Porsche, Maserati, Honda, Ferrari, Toyota, Ford>
```

Figura 32: Ejemplo unión de dos sets

Como se había dicho NSMutableSet es mutable, pero existe NSMutableSet que es una subclase de la clase NSMutableSet, y si permite agregar y borrar elementos de un set, usando los métodos addObject y removeObject, similares a los que usamos con NSMutableArray

3.4.3. Diccionarios

Parecido al NSMutableSet, el NSDictionary es una colección no ordenada, que asocia un valor con una clave. Es útil para manejar relaciones de 2 objetos. NSDictionary es **INMUTABLE** pero existe NSMutableDictionary que si es mutable.

Hay varias maneras de crear un NSDictionary, a continuación se mostraran varias.

```
// Valores y claves como argumentos
NSDictionary *inventory = [NSDictionary dictionaryWithObjectsAndKeys:
    [NSNumber numberWithInt:13], @"Mercedes-Benz SLK250",
    [NSNumber numberWithInt:22], @"Mercedes-Benz E350",
    [NSNumber numberWithInt:19], @"BMW M3 Coupe",
    [NSNumber numberWithInt:16], @"BMW X6", nil];

// Valores y claves como arreglos
NSArray *models = @[@"Mercedes-Benz SLK250", @"Mercedes-Benz E350",
    @"BMW M3 Coupe", @"BMW X6"];
NSArray *stock = @[NSNumber numberWithInt:13,
    [NSNumber numberWithInt:22],
    [NSNumber numberWithInt:19],
    [NSNumber numberWithInt:16]];

inventory = [NSDictionary dictionaryWithObjects:stock forKey:models];
NSLog(@"%@", inventory);
```

Hay que ser cuidadosos al usar dictionaryWithObjects:ForKey: ya que se debe estar seguro que ambos arreglos tiene el mismo tamaño. Para acceder a un valor se debe usar el método objectForKey que se muestra a continuación.

```
[nombre_diccionario objectForKey:clave]);
[inventory objectForKey:@"Mercedes-Benz E350"]]);
```

Para recorrer un NSDictionary se puede usar un for-each, de la siguiente manera:

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

int main() {
    NSAutoreleasePool *pool = [NSAutoreleasePool new];
    NSDictionary *inventory = [NSDictionary dictionaryWithObjectsAndKeys:
        [NSNumber numberWithInt:13], @"Mercedes-Benz SLK250",
        [NSNumber numberWithInt:22], @"Mercedes-Benz E350",
        [NSNumber numberWithInt:19], @"BMW M3 Coupe",
        [NSNumber numberWithInt:16], @"BMW X6", nil];
    NSLog(@"We currently have %ld models available", [inventory count]);
    for (id key in inventory) {
        NSLog(@"There are %@ %@'s in stock", [inventory objectForKey:key]
    }
    [pool drain];
    return 0;
}
```

La salida se muestra a continuación:

```
2016-05-02 05:36:53.046 helloworld[1208] We currently have 4 models available
2016-05-02 05:36:53.054 helloworld[1208] There are 16 BMW X6's in stock
2016-05-02 05:36:53.071 helloworld[1208] There are 19 BMW M3 Coupe's in stock
2016-05-02 05:36:53.071 helloworld[1208] There are 13 Mercedes-Benz SLK250's in
stock
2016-05-02 05:36:53.091 helloworld[1208] There are 22 Mercedes-Benz E350's in st
ock
```

Figura 33: Ejemplo recorrer un diccionario

Para comparar dos diccionarios solo basta con usar el método isEqualToDictionary:

```
if ([diccionario1 isEqualToDictionary:diccionario2]) {
    NSLog(@"Ambos diccionarios son iguales");
}
```

Las siguientes son las operaciones para modificar, agregar y borrar llaves de un diccionario (debe ser mutable):

```
// Modifica un valor ya existente
[diccionario setObject:@"Mary" forKey:@"Audi TT"];
// Borra un elemento del diccionario
[diccionario removeObjectForKey:@"Audi A7"];
// Agregar una nueva llave y un valor al diccionario
diccionario[@"Audi R8 GT"] = @"Jack";
```

4. Programación Orientada a Objetos

4.1. Clases

Como muchos otros lenguajes de programación orientados a objetos, las clases de Objective C definen una plantilla para crear objetos. La ventaja que proveen las clases es que tenemos asociadas toda una serie de datos y acciones que juntas forman un solo paquete que se denomina Objeto. Lo cual permite tener un código fuente más estructurado y organizado.

Objective C es similar a C++ en cuanto a que ambos abstraen la interfaz de una implementación concreta.

Este concepto se puede entender mejor con la siguiente figura.

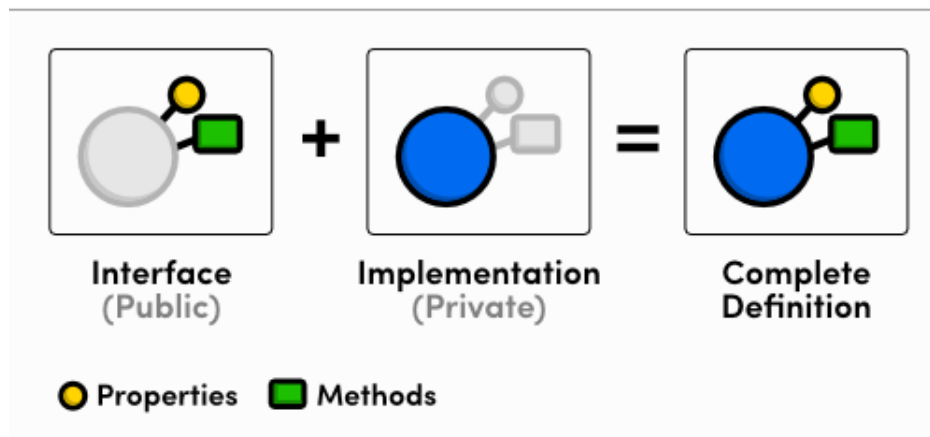


Figura 34: Diferencia entre clase (implementación) e interfaz.

Como se puede ver en la imagen, una interfaz provee el punto de entrada del mundo externo a un objeto. Es decir, la interfaz define una serie de métodos y propiedades que son públicas y cualquier objeto puede acceder mientras que la implementación define propiedades no accesibles para

otro objeto es decir, son privadas. Finalmente, si juntamos ambos conceptos obtenemos la definición completa de una clase.

4.1.1. Definiendo clases

Trabajaremos con una clase llamada *Box* cuya interfaz reside en un archivo llamado *Box.h* y su implementación en un archivo llamado *Box.m*. Esto quiere decir que para crear una clase, es conveniente tener en archivos separados su interfaz y su implementación. La interfaz se crea el archivo *.h* y la implementación en el archivo *.m*

Como primer paso, definimos la interfaz. Es decir, en nuestro editor creamos un archivo *Box.h* cuyo contenido es el siguiente.

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

@interface Box:NSObject
{
    double length;    // Length of a box
    double breadth;   // Breadth of a box
    double height;    // Height of a box
}
@property(n nonatomic, readwrite) double height; // Property

-(double) volume;

@end
```

Una interfaz es creada por la directiva *@interface* seguido del nombre de la clase y la superclase. Podemos incluir variables protegidas dentro de los corchetes.

La siguiente línea

```
@property(n nonatomic, readwrite) double height; // Property
```

Esta especificando que un objeto de la clase *Box* tendrá una propiedad de tipo *double* en el cual se va a almacenar la altura de la caja como número real. La directiva *@property* Generará automáticamente getters y/o setters para la propiedad *height*, y la especificación *nonatomic*, nos dice que a la hora de acceder a la propiedad, no realizará ningún tipo de bloqueo en caso de que haya más hilos que quieran acceder a él. Si eso pasa, los hilos

entrarán como si de una sección de código normal se tratara. Por último, y la especificación *readwrite* nos dice que creará tanto getters (read) como setters (write).

```
-(double) volume;
```

Con la anterior línea, se especifica que la clase Box tiene un método llamado volume que no recibe parámetros y retorna *double*. El signo menos (-) al comienzo de la declaración del método quiere decir que es un método de instancia. Es decir, que pertenece a una instancia en particular en lugar de a la clase. Por ejemplo, el modificador static en Java y C++ hace que el método en cuestión sea de clase y no de instancia mientras que en Objective C esto se logra anteponiendo un signo más (+) a la declaración del método o de la propiedad.

Pasamos a la implementación.

```
//Box.m

# import "Box.h"

@implementation Box

@synthesize height;

-(id)init
{
    self = [super init];
    length = 1.0;
    breadth = 1.0;
    return self;
}

-(double) volume
{
    return length*breadth*height;
}

@end
```

Las implementaciones comienzan por la directiva *@implementation* y terminan con la directiva *@end*.

La siguiente línea

```
@synthesize height;
```

Concretiza los getters y setters que fueron pensados en la interfaz para la propiedad height.

Ahora veremos cómo se definen los métodos en Objective-c

```
-(id)init  
{  
    self = [super init];  
    length = 1.0;  
    breadth = 1.0;  
    return self;  
}
```

El método init, aun cuando no ha sido declarado en la interfaz, se sobre escribe para hacer las veces de constructor. Esta función inicializa los parámetros del objeto y retorna la dirección de memoria donde el objeto ha sido creado.

```
self = [super init];
```

Esta línea se encarga de llamar al constructor de la súper clase, que en este caso es *NSObject* (similar a *Object* de Java), es necesaria para no retornar un puntero nulo, es decir, crear correctamente el objeto.

```
return self;
```

Esta línea retorna la dirección de memoria donde el objeto ha sido creado.

```
-(double) volume  
{  
    return length*breadth*height;  
}
```

Esta es la definición del método volume la cual retorna el resultado de una operación aritmética.

Tiempo de pasar al programa principal

```

//BoxMain.m

# import "Box.h"

int main( )
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    Box *box1 = [[Box alloc]init];    // Create box1 object of type Box
    Box *box2 = [[Box alloc]init];    // Create box2 object of type Box
    Box *box3 = [[Box alloc]init];    // Create box3 object of type Box

    double volume = 0.0;    // Store the volume of a box here

    // box 1 specification
    box1.height = 5.0;

    // box 2 specification
    box2.height = 10.0;

    // box 3 specification
    box3.height = 111.0;

    // volume of box 1
    volume = [box1 volume];
    NSLog(@"Volume of Box1 : %f", volume);
    // volume of box 2
    volume = [box2 volume];
    NSLog(@"Volume of Box2 : %f", volume);

    volume = [box3 volume];
    NSLog(@"Volume of Box3 : %f", volume);

    [pool drain];
    return 0;
}

```

Con su respectiva salida.

```

2016-05-01 18:06:00.217 BoxMain[15189] Volume of Box1 : 5.000000
2016-05-01 18:06:00.219 BoxMain[15189] Volume of Box2 : 10.000000
2016-05-01 18:06:00.219 BoxMain[15189] Volume of Box3 : 111.000000

```

Para ayudar al programador con el manejo de memoria, creamos un objeto de tipo *NSAutoreleasePool* el cual, al finalizar el programa liberará la memoria usada por los objetos creados.

```
NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
```

Ahora realizamos 3 instancias de la clase Box, una clase se instancia de la siguiente manera

```
Box *box1 = [[Box alloc] init];    // Create box1
```

Note que un objeto es creado solamente sobre un puntero, a diferencia de C++, que sí permite crear objetos sin apuntadores.

El llamado a un método de una clase se hace de la siguiente manera.

```
//volume of box 1  
volume = [box1 volume];
```

Para liberar la memoria utilizada escribimos

```
[pool drain];
```

Note que esto es un llamado al método drain del objeto pool.

Para compilar una clase (o varias), listamos todas las implementaciones (archivos .m) de estas en el comando y adicional mente compilamos el programa principal que también debe estar en un archivo .m

```
gcc $(gnustep-config --objc-flags)  
Box.m BoxMain.m $(gnustep-config --base-libs) -o BoxMain
```

4.1.2. Herencia

Objective-C soporta la herencia de varios niveles, pero una clase solo puede tener una superclase. Es decir, se pueden hacer varios niveles de jerarquías de clases pero solo se permite la herencia simple.

En Objective C todas las clases son hijas, directas o indirectas de NSObject, la cual es la superclase general y básica cuya definición se encuentra en Foundation.h

La sintaxis para aplicar la herencia en una clase es

```
@interface Clase : SuperClase
```


El siguiente ejemplo muestra el uso de la herencia con clases creadas por el programador.

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

@interface Person : NSObject

{
    NSString *personName;
    NSInteger personAge;
}

- (id)initWithName:(NSString *)name andAge:(NSInteger)age;
- (void)print;
@end

@implementation Person

- (id)initWithName:(NSString *)name andAge:(NSInteger)age{
    personName = name;
    personAge = age;
    return self;
}

- (void)print{
    NSLog(@"Name: %@", personName);
    NSLog(@"Age: %ld", personAge);
}

@end

@interface Employee : Person

{
    NSString *employeeEducation;
}

- (id)initWithName:(NSString *)name andAge:(NSInteger)age
andEducation:(NSString *)education;
- (void)print;
```

```
@end
```

```
@implementation Employee
```

```
- (id)initWithName:(NSString *)name andAge:(NSInteger)age
andEducation: (NSString *)education
{
    personName = name;
    personAge = age;
    employeeEducation = education;
    return self;
}

- (void)print
{
    [super print];
    NSLog(@"Education: %@", employeeEducation);
}
```

```
@end
```

```
int main(int argc, const char * argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    NSLog(@"Base class Person Object");
    Person *person = [[Person alloc] initWithName:@"Raj" andAge:5];
    [person print];
    NSLog(@"Inherited Class Employee Object");
    Employee *employee = [[Employee alloc] initWithName:@"Raj"
andAge:5 andEducation:@"MBA"];
    [employee print];
    [pool drain];
    return 0;
}
```

Inicialmente, tenemos una clase Person (Persona) que hereda de NSObject. Se definen los atributos de nombre y edad así como el constructor.

Luego, se crea una clase Employee (Empleado) que es una subclase de Person que a su vez es subclase de NSObject
la línea clave que indica esto es

```
@interface Employee : Person
```

Es de notar también cómo se re usa código en el método print de Employee. Nosotros sabemos que print de Person muestra el nombre y la edad, entonces basta con llamar al método print de la superclase y luego agregar en lo que se ha especializado la nuestra

```
- (void)print
{
    [super print];
    NSLog(@"Education: %@", employeeEducation);
}
```

El resultado de ejecutar el código es el siguiente

```
2016-05-01 20:37:40.520 Person.m[16723] Base class Person Object
2016-05-01 20:37:40.522 Person.m[16723] Name: Raj
2016-05-01 20:37:40.522 Person.m[16723] Age: 5
2016-05-01 20:37:40.522 Person.m[16723] Inherited Class Employee Object
2016-05-01 20:37:40.522 Person.m[16723] Name: Raj
2016-05-01 20:37:40.522 Person.m[16723] Age: 5
2016-05-01 20:37:40.522 Person.m[16723] Education: MBA
```

4.1.3. Polimorfismo

Polimorfismo significa que el resultado de llamar un método puede variar dependiendo del tipo objeto que lo llame. En el siguiente ejemplo veremos cómo teniendo referencias a objetos de tipo básico (Shape), sus áreas son calculadas de una manera totalmente distinta porque son dos tipos de figuras distintas.

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

```
@interface Shape : NSObject
```

```
{
    CGFloat area;
```

```

}

- (void)printArea;
- (void)calculateArea;
@end

@implementation Shape

- (void)printArea{
    NSLog(@"The area is %f", area);
}

- (void)calculateArea{

}

@end

@interface Square : Shape
{
    CGFloat length;
}

- (id)initWithSide:(CGFloat)side;

- (void)calculateArea;

@end

@implementation Square

- (id)initWithSide:(CGFloat)side{
    length = side;
    return self;
}

- (void)calculateArea{
    area = length * length;
}

```

```

- (void)printArea{
    NSLog(@"The area of square is %f", area);
}

@end

@interface Rectangle : Shape
{
    CGFloat length;
    CGFloat breadth;
}

- (id)initWithLength:(CGFloat)rLength andBreadth:(CGFloat)rBreadth;

@end

@implementation Rectangle

- (id)initWithLength:(CGFloat)rLength andBreadth:(CGFloat)rBreadth{
    length = rLength;
    breadth = rBreadth;
    return self;
}

- (void)calculateArea{
    area = length * breadth;
}

@end

int main(int argc, const char * argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    Shape *square = [[Square alloc] initWithSide:10.0];
    [square calculateArea];
    [square printArea];
    Shape *rect = [[Rectangle alloc]

```

```

initWithLength:10.0 andBreadth:5.0];
[rect calculateArea];
[rect printArea];
[pool drain];
return 0;
}

```

El resultado de la ejecución del programa es la siguiente

```

2016-05-01 21:14:52.056 Shape[17113] The area of square is 100.000000
2016-05-01 21:14:52.058 Shape[17113] The area is 50.000000

```

Se puede ver que aparentemente la llamada al mismo método (*calculateArea*) en realidad son dos llamadas a métodos de distintas clases.

4.2. Encapsulamiento

La encapsulación de datos es un mecanismo para ocultar la información de un objeto y evitar interferencias externas. Por lo tanto, desde afuera de la clase no podremos acceder a sus atributos pues estos son privados.

```

#import <Foundation/Foundation.h>

@interface Adder : NSObject
{
    NSInteger total;
}

- (id)initWithInitialNumber:(NSInteger)initialNumber;

- (void)addNumber:(NSInteger)newNumber;

- (NSInteger)getTotal;

@end

@implementation Adder

- (id)initWithInitialNumber:(NSInteger)initialNumber{
    total = initialNumber;
    return self;
}

```

```

}

- (void)addNumber:(NSInteger)newNumber{
    total = total + newNumber;
}

- (NSInteger)getTotal{
    return total;
}

@end

int main(int argc, const char * argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    Adder *adder = [[Adder alloc] initWithInitialNumber:10];
    [adder addNumber:5];
    [adder addNumber:4];
    NSLog(@"The total is %ld",[adder getTotal]);
    [pool drain];
    return 0;
}

```

En el ejemplo anterior la única manera de modificar la variable total perteneciente a la clase Adder es por medio de los métodos de la clase, si queremos saber su valor usamos el método *getTotal*, y si queremos modificarle su valor lo hacemos mediante *addNumber*.

4.3. Protocolos

Los protocolos declaran métodos que se prevé que se van a utilizar en una situación particular. Para declararlos se hace de la siguiente manera:

```

@protocol ProtocolName
@required
// list of required methods
@optional
// list of optional methods
@end

```

Ahora si queremos que una clase implemente determinados protocolos se hace de la siguiente manera:

```
@interface MyClass : NSObject <MyProtocol1>, ... , <MyProtocol2>
...
@end
```

Un ejemplo de lo anterior lo podemos encontrar en la página, el archivo con el nombre de **protocolos.m**.

5. Ejemplos

5.1. Recorrido por el lenguaje

En la página web se pueden encontrar ejemplos básicos, subidos en ideone para que esté al alcance de cualquier persona.

5.2. Clases

En la página web se pueden encontrar los archivos correspondientes al primer ejemplo (Box.h Box.m BoxMain.m).

Adicionalmente. Se adjuntan los archivos (Car.h Car.m CarMain.m) donde se muestra un ejemplo manejando una clase Car (Carro) con variable miembro brand (marca) , se realizan tres instancias de esta clase, y se invoca un método para cada instancia.

Referencias

[Ry's Objective 2014] <http://rypress.com/tutorials/objective-c/index> consultado el 1/05/2016.

[TutorialsPoint] http://www.tutorialspoint.com/objective_c/. consultado el 1/05/2016.