Desarrollo de Aplicaciones iOS

Sesión 2: Introducción a Objective-C

Puntos a tratar

- Tipos de datos
- Paso de mensajes
- Manejo de cadenas
- Clases y objetos
- Gestión de la memoria
- Clases útiles

Objective-C

- Extensión del lenguaje C orientada a objetos
  - Ficheros con extensión `.m` en lugar de `.c`
  - Utilizaremos la API Cocoa Touch

- Diferente a C++
  - Se basa en paso de mensajes en lugar de llamadas a métodos

- Podemos utilizar cualquier elemento de C estándar

- Existe la variante Objective-C++
  - Permite utilizar código C++ dentro de Objective-C
  - Ficheros con extensión `.mm`

Paso de mensajes

- En Objective-C se habla de paso de mensajes
  - Similar a lenguajes como Smalltalk
  - La dirección a llamar se resuelve en tiempo de ejecución

- Obtener longitud de una cadena

```
NSString* cadena = @"cadena-de-prueba";
int tam = [cadena length];
```

- El método puede no existir en el objeto receptor del mensaje

```
NSString* cadena = @"cadena-de-prueba";
[cadena metodoInexistente]; // Produce warning, pero compila
```

Puntero genérico a objeto

```
id cadena = @"cadena-de-prueba";
[cadena metodoInexistente]; // Solo da error de ejecucion
```

Cadenas

- Las cadenas en Objective-C son objetos de clase `NSString`
  - No confundir con las cadenas de C (`char *`)
  - Los literales de `NSString` se crean con el prefijo `@`

```
NSString *cadena = @"Cadena";
```

- Crear una cadena con formato

```
NSString *nombre = @"Pepe";
int edad = 20;

NSString *cadena =
  [NSString stringWithFormat: @"Nombre: %@ (edad %d)", nombre, edad];
```

- Impresión de *logs*

```
NSLog(@"i = %d, obj = %@", i, obj);
```

  - Aparecen en la consola de depuración

Localización de cadenas

- Por defecto se extraen a un fichero `Localizable.strings`
  - Creamos el fichero con *New File* > *iOS* > *Resource* > *Strings File*

- Formato del fichero

```
"identificador" = "cadena a mostrar";
"Titulo" = "Moviles UA";
```

- Acceso a la cadena localizada desde el código

```
NSString *cadenaLocalizada = NSLocalizedString(@"Titulo", @"Mobile UA");
```

Tipos de datos básicos

- Contamos con todos los tipos básicos de C
  - `char, short, int, long, float, double,` ...

- Cocoa Touch define tipos básicos adicionales
  - Se adaptan a la arquitectura (32 ó 64 bits)
    `NSInteger` (entero con signo)
    `NSUInteger` (entero sin signo)
    `CGFloat` (flotante)

- Tipo booleano
  - Puede tomar como valor las constantes `YES` o `NO`

```
BOOL b = YES;
```

Enumeraciones

- Se definen igual que en C
- Cada elemento tiene asignado un valor entero (incremental)

```
typedef enum {
  UATipoAsignaturaOptativa,
  UATipoAsignaturaObligatoria,
  UATipoAsignaturaTroncal
} UATipoAsignatura;
```

- Podemos asignar manualmente los valores

```
typedef enum {
  UATipoAsignaturaOptativa = 0,
  UATipoAsignaturaObligatoria = 1,
  UATipoAsignaturaTroncal = 2
} UATipoAsignatura;
```

Estructuras de datos

- También se definen igual que en C

```
struct CGPoint {
  CGFloat x;
  CGFloat y;
};
typedef struct CGPoint CGPoint;
```

- En Cocoa Touch hay funciones para inicializar y gestionar las estructuras que define

```
CGPoint punto = CGPointMake(x,y);
```

Directivas

- `#import`
  - Importa fichero de cabecera `.h`
  - A diferencia de `#include`, evita inclusiones cíclicas
- `#define`
  - Define macros de preprocesamiento
  - Se pueden comprobar los símbolos definidos

```
#ifdef DEBUG
  NSLog(@"Texto del log");
#endif
```

- `#pragma mark`
  - Permite etiquetar segmentos de código

```
#pragma mark Constructores
// Código de los constructores

#pragma mark Eventos del ciclo de vida
// Código de los manejadores de eventos
```

Constantes

- Se definen mediante el modificados `const`
- Afecta al elemento justo a su izquierda

```objc
// Puntero variable a objeto NSString constante (MAL)
const NSString * UATitulo = @"Menu";

// Equivalente al anterior (MAL)
NSString const * UATitulo = @"Menu";

// Puntero constante a objeto NSString (BIEN)
NSString * const UATitulo = @"Menu";
```

- Es preferible definir las constantes con `const` en lugar de `#define`

Definición de una clase (.h)

Nombre de la clase

Hereda de `NSObject`

Variables de instancia (protegidas)

Métodos de clase

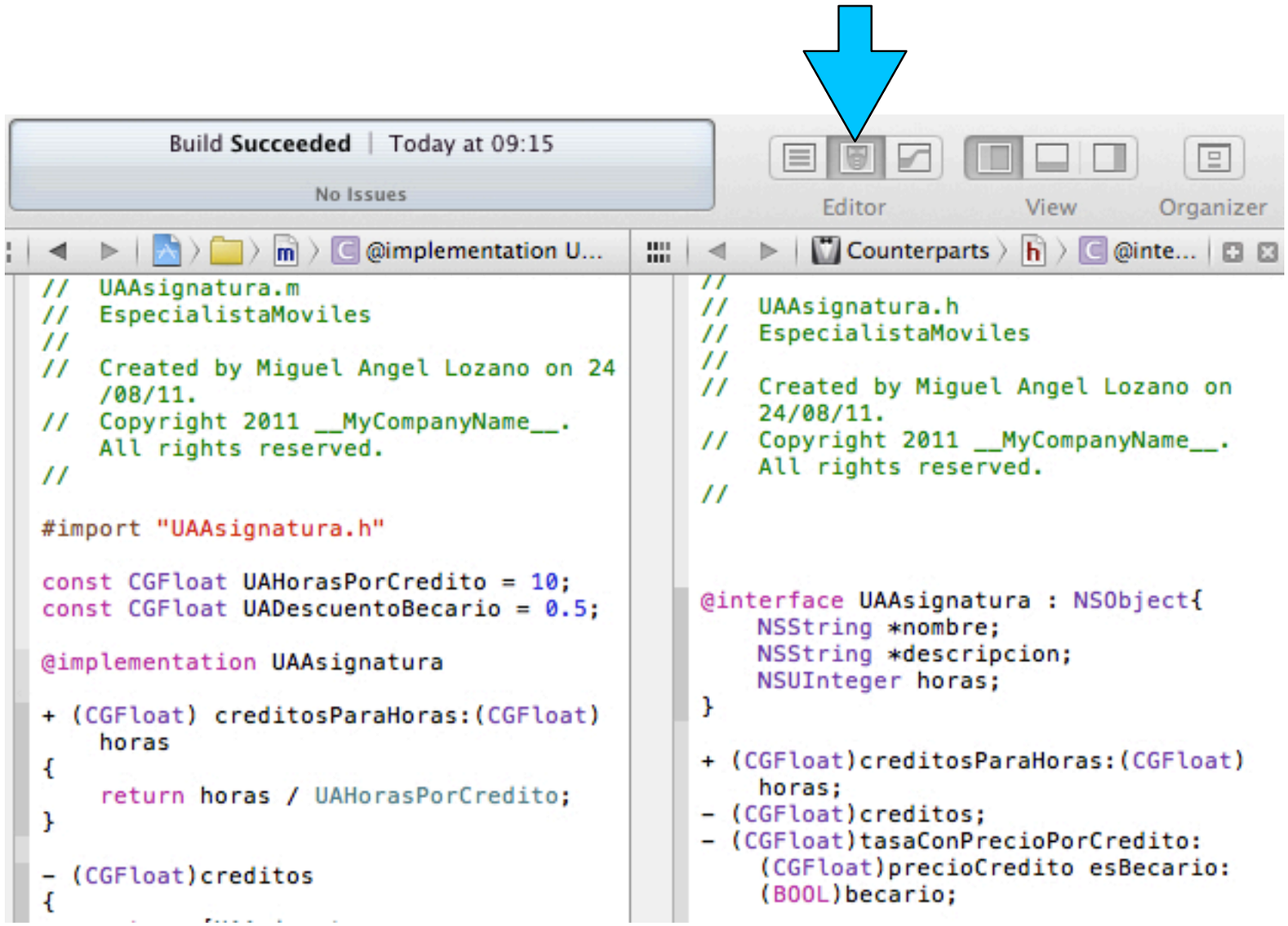
Métodos de instancia

```
@interface UAAsignatura : NSObject {
    NSString *_nombre;
    NSString *_descripcion;
    NSUInteger _horas;
}

+ (CGFloat)creditosParaHoras:(CGFloat)horas;
- (CGFloat)creditos;
- (CGFloat)tasaConPrecioPorCredito:(CGFloat)precioCredito
                        esBecario:(BOOL)becario;

@end
```

Implementación de la clase (.m)

```
#import "UAAsignatura.h"

const CGFloat UAHorasPorCredito = 10;
const CGFloat UADescuentoBecario = 0.5;

@implementation UAAsignatura

+ (CGFloat) creditosParaHoras:(CGFloat)horas {
    return horas / UAHorasPorCredito;
}

- (CGFloat)creditos {
    return [UAAsignatura creditosParaHoras: _horas];
}

- (CGFloat)tasaConPrecioPorCredito:(CGFloat)precioCredito esBecario:(BOOL)becario {
    CGFloat precio = [self creditos] * precioCredito;
    if(becario) {
        precio = precio * UADescuentoBecario;
    }
    return precio;
}

@end
```

Vista de asistente



Creación e inicialización

- Reservamos memoria para un objeto con `alloc`
- Inicializamos el objeto con un inicializador `init`

```
NSString *cadVacia = [[NSString alloc] init];
NSString *cadFormato = [[NSString alloc] initWithFormat: @"Numero %d", 5];
```

- Inicialización mediante métodos factoría
  - Son métodos de clase
  - No hace falta que reservemos memoria nosotros
  - Su nombre comienza por el del objeto que van a crear

```
NSString *cadVacia = [NSString string];
NSString *cadFormato = [NSString stringWithFormat: @"Numero %d", 5];
```

Implementación de inicializadores

Devuelven siempre el tipo `id`

Inicializa la superclase y asigna el objeto resultante al objeto actual (`self`)

Si la superclase no ha devuelto un puntero a nulo, inicializa variables de instancia propias

```
- (id)initWithNombre:(NSString*)nombre
         descripcion:(NSString*)descripcion
               horas:(NSUInteger)horas
{
    self = [super init];
    if(self != nil) {
        _nombre = nombre;
        _descripcion = descripcion;
        _horas = horas;
    }
    return self;
}
```

Devuelve el objeto inicializado (`self`)

Inicializador designado

- Debe ser invocado por el resto de inicializadores
- Suele ser el que lleva un mayor número de parámetros

```
- (id)init;
- (id)initWithNombre:(NSString*)nombre;
- (id)initWithNombre:(NSString*)nombre
         descripcion:(NSString*)descripcion
               horas:(NSUInteger)horas;
```

```
- (id)init
{
    return [self initWithNombre: @"Sin nombre"];
}

- (id)initWithNombre:(NSString *)nombre
{
    return [self initWithNombre:nombre
                    descripcion:@"Sin descripcion"
                          horas:-1];
}
```

Gestión de la memoria

- La gestión se hace contando referencias
  - Cuando se reserva con `alloc` el número de referencias es 1
  - Podemos incrementar el número de referencias con `retain`
  - Podemos decrementarlo con `release`
  - Cuando las referencias llegan a 0, se libera la memoria
- Regla de oro
  - El objeto que retiene (`alloc-retain`), debe liberar (`release`)

Retención y liberación

- En el constructor retenemos variables de instancia

```
- (id)initWithNombre:(NSString*)nombre
         descripcion:(NSString*)descripcion
               horas:(NSUInteger)horas {
    self = [super init];
    if(self != nil) {
        _nombre = [nombre retain];
        _descripcion = [descripcion retain];
        _horas = horas;
    }
    return self;
}
```

- Al liberarse un objeto, se ejecuta su método `dealloc`
  - Liberar variables retenidas por el objeto
  - Llamar a `dealloc` en la superclase

```
- (void)dealloc {
    [_nombre release];
    [_descripcion release];
    [super dealloc];
}
```

Gestión en métodos factoría

- El propio método debe liberar lo que ha retenido
  - El objeto debe estar disponible en memoria al menos hasta que lo recoja quien llamó al método

- Utilizamos `autorelease`
  - Guarda la liberación como pendiente en un *autorelease pool*, que se llevará a cabo cuando termine la pila de llamadas

```
+ (id)asignaturaWithNombre:(NSString*)nombre
                descripcion:(NSString*)descripcion
                      horas:(NSUInteger)horas
{
    return [[[UAAsignatura alloc] initWithNombre:nombre
                                     descripcion:descripcion
                                           horas:horas] autorelease];
}
```

Clase `NSObject`

- Casi todos los objetos heredan de ella en última instancia
  - Podemos no heredar de nadie para crear una estructura de datos

- Podemos sobrescribir una serie de métodos
  - `isEqual`

    Comprueba si dos objetos son iguales internamente

    Devuelve `YES` o `NO`
  - `description`

    Devuelve la descripción del objeto en forma de cadena

    Es lo que se obtiene al imprimir el objeto con `%@`
  - `hash`

    Código *hash* para indexar el objeto

    Debe ser coherente con `isEqual`

Copia de objetos

- Algunos objetos pueden copiarse con el método `copy`
  - Al crear un objeto como copia su contador de referencias es 1

- Objetos mutables e inmutables
  - Algunos objetos existen en las dos modalidades
    `NSString` y `NSMutableString`

- Podemos obtener copias mutables de objetos inmutables con `mutableCopy`

```
NSString *cadena = @"Mi cadena";

NSString *copiaInmutable = [cadena copy];

NSMutableString *copiaMutable = [cadena mutableCopy];
```

Gestión de fechas

- Se gestionan mediante la clase `NSDate`

```
NSDate *fecha = [NSDate date];
```

- Podemos obtener los componentes de una fecha

```
NSCalendar *calendario = [NSCalendar currentCalendar];
NSDateComponents *componentes = [calendario
    components:(NSDayCalendarUnit | NSMonthCalendarUnit | NSYearCalendarUnit)
      fromDate:fecha];

NSInteger dia = [componentes day];
NSInteger mes = [componentes month];
NSInteger anyo = [componentes year];
```

- O una fecha a partir de sus componentes

```
NSDateComponents *componentes = [[NSDateComponents alloc] init];
[componentes setDay: dia];
[componentes setMonth: mes];
[componentes setYear: anyo];

NSDate *fecha = [calendario dateFromComponents: componentes];
[componentes release];
```

Tratamiento de errores

- Se pueden utilizar excepciones

```
@try
    // Codigo
@catch(NSException *ex) {
    // Codigo tratamiento excepcion
}
@finally {
    // Codigo de finalización
}
```

  - Son siempre *unchecked*
- Para errores de tipo *checked* suele utilizarse `NSError`

```
NSError *error;
NSString *contenido = [NSString
                stringWithContentsOfFile: @"texto.txt"
                                encoding: NSASCIIStringEncoding
                                   error: &error];
NSString *motivo = [error localizedFailureReason];
```

Ámbito de las variables

- **Global**
  - Se declaran fuera de cualquier método. Para que se pueda acceder desde otros ficheros deben aparecer declaradas con `extern` en algún fichero de cabecera.

- **Fichero**
  - Se declaran fuera de cualquier método con modificador `static`. Accesible sólo desde el fichero en el que se define.

- **Local**
  - Se declaran dentro de un método. Si lleva modificador `static` sólo se instancia la primera vez que se ejecuta el método.

Patrón singleton

- Podemos utilizar variables de tipo `static` para implementar el patrón *singleton*

- Normalmente crearemos un método de clase con prefijo `shared` que nos dará acceso a la instacia única

```
+ (UADatosCompartidos) sharedDatosCompartidos {
    static DatosCompartidos *datos = nil;
    if(nil == datos) {
        datos = [[DatosCompartidos alloc] init];
    }
    return datos;
}
```

¿Preguntas...?