Desarrollo de Aplicaciones iOS

Sesión 3: Propiedades y colecciones

© 2012 Depto. Ciencia de la Computación e IA

Puntos a tratar

- Propiedades de los objetos
- Gestión de la memoria con y sin ARC
- Colecciones de datos
- KVC
- Programación de eventos
- Ciclo de vida de la aplicación

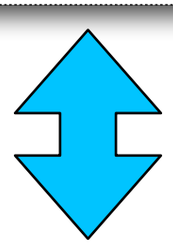
Propiedades

- Las variables de instancia por defecto son protegidas
  - Podemos incluir modificadores de acceso

    `@private, @protected, @public`
  - Lo habitual es dejar el valor por defecto y definir *getters* y *setters*

- Propiedades
  - Información a la que se accede mediante *getters* y *setters*

```
@property(nonatomic,retain) NSString *nombre;
```



```
- (NSString *)nombre;
- (void)setNombre: (NSString *)nombre;
```

Definir y sintetizar propiedades

- Se definen en la interfaz

```
@interface UAAsignatura : NSObject
...
@property(nonatomic,retain) NSString *nombre;
@property(nonatomic,assign) NSUInteger horas;

@end
```

Equivale a declarar

```
-(NSString*)nombre
-(void)setNombre:
            (NSString*)
```

- Se sintetizan en la implementación

```
@implementation UAAsignatura

@synthesize nombre = _nombre;
@synthesize horas = _horas;
...
@end
```

Crea variable de instancia `_nombre` de forma automática asociada a la propiedad

Equivale a implementar `nombre` y `setNombre`

Acceso a las propiedades

- Mediante paso de mensajes a los *getters* y *setters*

```
[asignatura setNombre: @"Plataforma iOS"];
NSLog(@"Nombre: %@", [asignatura nombre]);
```

- Mediante el operador .

```
asignatura.nombre = @"Plataforma iOS";
NSLog(@"Nombre: %@", asignatura.nombre);
```

- Equivalente a llamar al *getter* o *setter*
- Se puede utilizar para acceder a cualquier método
- No es recomendable abusar de él (sólo usar con propiedades)

Modificadores de las propiedades

- Permiten afinar la forma en la que se definen los *getters* y *setters*
  - `nonatomic`

    No sincroniza el acceso a la propiedad. Normalmente utilizaremos `nonatomic` para optimizar.
  - `readonly`

    Sólo genera el *getter*, para que la propiedad no se pueda modificar
  - `readwrite`

    Comportamiento por defecto, genera *getter* y *setter*
  - `getter=nombre_getter`

    Permite especificar el nombre que tendrá el *getter*
  - `setter=nombre_setter`

    Permite especificar el nombre que tendrá el *setter*

Gestión de la memoria

- Utilizamos modificadores para indicar cómo se gestiona la memoria al asignar valores a las propiedades
  - `assign`

    Sólo asigna el valor de la propiedad, sin retenerla (`retain`)
  - `retain`

    Al asignar un valor libera el anterior (`release`) y retiene el nuevo (`retain`)
  - `copy`

    Crea una copia del objeto al asignar (`copy`), liberando el valor anterior. Deberá implementar el protocolo `NSCopying`

- Si utilizamos `retain` o `copy` deberemos liberar las variables en `dealloc`

Automatic Reference Counting (ARC)

- Nueva característica de Xcode 4.2
- El compilador se encarga de realizar la gestión de la memoria
- Debemos seguir una serie de reglas:
  - Nunca llamar a `retain`, `release` o `autorelease`
  - No llamar a `[super dealloc]`, el compilador se encarga de ello
  - No es necesario definir `dealloc`
  - No hacer referencias a objetos Objective-C desde estructuras C
  - No hacer *cast* entre `(id)` y `(void *)`
  - Los *autorelease pools* se deben definir mediante la etiqueta:

```
@autoreleasepool {
    ...
}
```

Propiedades con ARC

- Ahora hablamos de referencias fuertes y débiles
  - `strong`

    Referencia fuerte, equivale a `retain`
  - `weak`

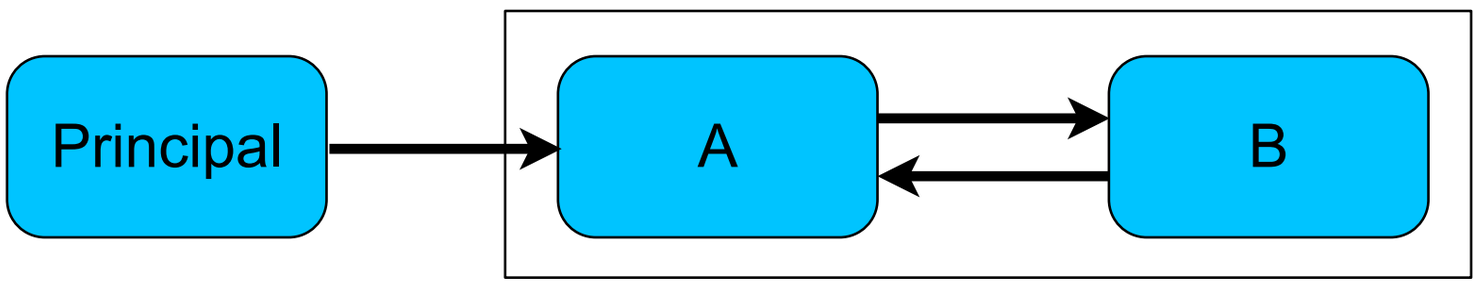
    Referencia débil. Cuando el objeto es liberado de memoria la referencia se pone a `nil` automáticamente. Sólo funciona con iOS 5.
- `unsafe_unretained`

  Referencia débil equivalente a `assign`. Se utiliza para punteros a objetos. Al ser liberados podríamos tener un error en el acceso.
- `assign`

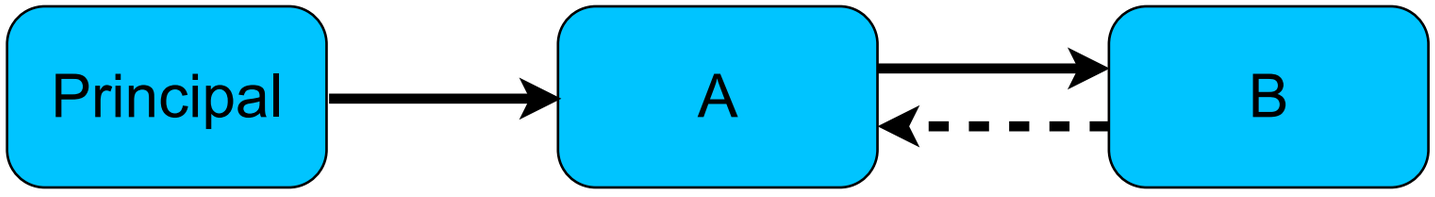
  Se utiliza para tipos de datos básicos.

Ciclo de retenciones

- Debemos seleccionar con cuidado las referencias fuertes y débiles
  - Un ciclo de referencias fuertes provoca una fuga de memoria



- Como regla general, las referencias a clases en niveles superiores de la jerarquía deben ser débiles



Colecciones de datos

- Colecciones genéricas de objetos
  - Listas

```
NSArray *lista = [NSArray arrayWithObjects: obj1, obj2, obj3, nil];
```

  - Diccionarios

```
NSDictionary *diccionario =
    [NSDictionary dictionaryWithObjectsAndKeys:
        obj1, @"clave1", obj2, @"clave2", obj3, @"clave3", nil];
```

  - Conjuntos

```
NSSet *conjunto = [NSSet setWithObjects: obj1, obj2, obj3, nil];
```

- Existen versiones mutables e inmutables de cada una
  - `NSMutableArray`, `NSMutableDictionary`, `NSMutableSet`

Tipos básicos en las colecciones

- Valor nulo

```
[NSNull null]
```

- Números

```
NSNumber *booleano = [NSNumber numberWithBool: YES];
NSNumber *entero = [NSNumber numberWithInt: 10];
NSNumber *flotante = [NSNumber numberWithFloat: 2.5];
...
BOOL valorBool = [booleano boolValue];
int valorEntero = [entero intValue];
float valorFlotante = [flotante floatValue];
```

- Otros tipos

```
typedef struct {
    int x;
    int y;
} Punto;
...
Punto p;
NSValue *valorPunto = [NSValue valueWithBytes:&p objCType:@encode(Punto)];
```

Acceso a los elementos de las listas

- Número de elementos

```
NSUInteger numElementos = [lista count];
```

  - Cada elemento de la lista está en un índice, de `0` a `count-1`
- Acceso a un elemento determinado

```
id primerObjeto = [lista objectAtIndex: 0];
```

- Recorrer la lista

```
for(id obj in lista) {
    NSLog(@"Obtenido el objeto %@", obj);
}
```

  - Si todos los elementos de la lista son de un tipo, podemos utilizar

```
for(NSString *cadena in lista) {
    NSLog(@"Obtenida la cadena %@", cadena);
}
```

Listas mutables

- Podemos modificar los elementos que contienen

```
NSMutableArray *listaMutable = [NSMutableArray arrayWithCapacity: 100];
```

- Operaciones

```
// A partir del indice 5 se mueven a la siguiente posición
[listaMutable insertObject:obj atIndex:5];

// Lo añade al final de la lista
[listaMutable addObject:obj];

// A partir del indice 5 se mueven a la anterior posición
[listaMutable removeObjectAtIndex:5];

// Es más eficiente, con coste constante
[listaMutable removeLastObject];

[listaMutable replaceObjectAtIndex:5 withObject:obj];
```

Diccionarios

- Contienen parejas *(clave, valor)*
  - Obtención del valor asociado a una clave

```
id obj = [diccionario objectForKey:@"clave1"];
```

  - Listas de claves y valores

```
NSArray *claves = [diccionario allKeys];
NSArray *valores = [diccionario allValues];
```

- Diccionarios mutables

```
NSMutableDictionary *diccionarioMutable =
    [NSMutableDictionary dictionaryWithCapacity: 100];
```

  - Establecer valor para una clave o eliminarla

```
[diccionario setObject:obj forKey:@"clave1"];
[diccionario removeObjectForKey:@"clave1"];
```

Almacenamiento de colecciones

- Sólo podemos guardar en los directorios de la aplicación
- Existen varios directorios donde almacenar datos
- Uno de ellos es el directorio de documentos

```
NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
                                                     NSUserDomainMask,
                                                     YES);
NSString *directory = [paths objectAtIndex:0];
NSString *filename = [directory stringByAppendingPathComponent:@"coleccion.plist"];
```

- Podemos guardar la colección directamente en el fichero

```
BOOL guardado = [coleccion writeToFile:filename atomically:YES];
```

- Podemos leer la colección mediante el siguiente inicializador

```
coleccion = [NSArray arrayWithContentsOfFile:filename];
```

Tipos de datos soportados

- En la colección sólo podemos tener
  - `NSString`
  - `NSData`
  - `NSDate`
  - `NSNumber`
  - `NSArray`
  - `NSDictionary`

- Para soportar otros tipos podemos adoptar `NSCoding`

```
@interface UAAsignatura : NSObject<NSCoding>
    ...
@end
```

Implementación de NSCoding

- Nos obliga a definir métodos de serialización y deserialización

```
- (id)initWithCoder:(NSCoder *)aDecoder {
    self = [super init];
    if(self!=nil) {
        self.nombre = [aDecoder decodeObjectForKey:@"nombre"];
        self.descripcion = [aDecoder decodeObjectForKey:@"descripcion"];
        self.horas = [aDecoder decodeIntegerForKey:@"horas"];
    }
    return self;
}

- (void)encodeWithCoder:(NSCoder *)aCoder {
    [aCoder encodeObject:self.nombre forKey:@"nombre"];
    [aCoder encodeObject:self.descripcion forKey:@"descripcion"];
    [aCoder encodeInteger:self.horas forKey:@"horas"];
}
```

Carga y almacenamiento

- Utilizamos las clases `NSKeyedArchiver` y `NSKeyedUnarchiver` para leer y almacenar objetos que adopten `NSCoding`
  - Las colecciones adoptan dicho protocolo, por lo que es aplicable

```
+ (UAAsignatura *) load {

    NSArray *dirs = NSSearchPathForDirectoriesInDomains(
                        NSDocumentDirectory, NSUserDomainMask, YES);
    NSString *filename = [[dirs objectAtIndex:0]
                          stringByAppendingPathComponent: @"datos"];

    UAAsignatura *datos = [NSKeyedUnarchiver unarchiveObjectWithFile: filename];
    return datos;
}

- (BOOL) save {
    NSArray *dirs = NSSearchPathForDirectoriesInDomains(
                        NSDocumentDirectory, NSUserDomainMask, YES);
    NSString *filename = [[dirs objectAtIndex:0]
                          stringByAppendingPathComponent: @"datos"];

    return [NSKeyedArchiver archiveRootObject: self toFile: filename];
}
```

Key Value Coding (KVC)

- Permite acceder a las propiedades de los objetos como si fueran entradas de un diccionario

```
NSString *nombre = [asignatura valueForKey: @"nombre"];
NSNumber *horas = [asignatura valueForKey: @"horas"];

[asignatura setValue:@"Proyecto iOS" forKey:@"nombre"];
[asignatura setValue:[NSNumber numberWithInteger:30] forKey:@"horas"];
```

- Permite acceder a las variables aunque sean privadas
- Si en un diccionario definimos las claves como cadenas podemos acceder de la misma forma
- Podemos indicar una ruta de propiedades en objetos anidados

```
NSString *nombreCoordinador =
        [asignatura valueForKeyPath:@"coordinador.nombre"];
```

Programación de eventos

- Una forma habitual de definir *callbacks* es mediante el patrón observador
  - Registramos un método definido por nosotros como observador para un determinado tipo de eventos
  - Cuando se produce el evento se invoca a nuestro método para que realice las tareas oportunas

- En Objective-C el observador se especifica mediante

  target

  Objeto al que se le pasará el mensaje

  selector

  Método al que irá dirigido el mensaje

Selectores

- El selector consiste en el nombre completo del método sin indicar tipos de parámetros (sólo aparece `:` donde va cada parámetro)

```
creditos                                  (Método sin parámetros)
creditosParaHoras:                          (1 parámetro)
tasaConPrecioPorCredito:esBecario:          (2 parámetros)
```

  - En un objeto no puede haber dos métodos con el mismo selector (no existe sobrecarga en Objective-C)
- Se representan mediante el tipo `SEL`
  - Podemos obtener un dato de tipo `SEL` con `@selector`

```
SEL miSel = @selector(tasaConPrecioPorCredito:esBecario:);
```

  - Ejecutar selectores

```
[asignatura performSelector:miSel
                 withObject:[NSNumber numberWithFloat: 60.0]
                 withObject:[NSNumber numberWithBool: YES]];
```

Ejemplo: Temporizadores

- Podemos programar temporizadores con `NSTimer`

```
NSTimer *temporizador = [NSTimer
    scheduledTimerWithTimeInterval:5.0
                            target: self
                          selector: @selector(tick:)
                          userInfo: nil
                           repeats: YES];
```

- Cuando se dispare el temporizador se pasará un mensaje a nuestro método `tick:`

```
- (void) tick: (NSTimer*)temporizador;
```

  - Los *callbacks* suelen tomar como parámetro el objeto que produjo el evento (en el caso anterior el temporizador)

Notificaciones

- Permiten comunicar objetos lejanos en el diagrama de clases
- Centro de notificaciones común definido como *singleton*

```
[NSNotificationCenter defaultCenter]
```

- Registrarnos como observadores de una notificación

```
[[NSNotificationCenter defaultCenter]
     addObserver:self
        selector:@selector(sincronizado:)
            name:@"SincronizacionCompletada"
          object:nil];
```

- Enviar la notificación

```
[[NSNotificationCenter defaultCenter]
    postNotificationName:@"SincronizacionCompletada" object:self];
```

Objetos delegados

- Un objeto delega en otro para realizar su tarea
- Patrón muy utilizado en la API de Cocoa Touch, por ejemplo
  - `UIApplication` delega la programación de su ciclo de vida
- Implementación
  - El objeto delegado se asigna a una propiedad `delegate`
  - Cuando se debe realizar una tarea que se quiera delegar
    Comprueba si hay un objeto delegado definido
    Comprueba si el delegado implementa el método que realiza dicha tarea
  - Los métodos que debe definir el delegado suelen definirse en protocolos

Protocolos

- Similares a las interfaces en Java
- Pueden definir métodos requeridos u opcionales

```
@protocol MiProtocolo

- (void)metodoObligatorio;

@optional
- (void)metodoOpcional;
- (void)otroMetodoOpcional;

@required
- (void)otroMetodoObligatorio;

@end
```

- Se implementan de la siguiente forma:

```
@interface MiClase : NSObject<MiProtocolo>
    ...
@end
```

Introspección

- Algunos delegados no requieren protocolos
- Algunos protocolos tienen métodos opcionales
- Debemos poder saber si un objeto implementa un método

```
if([asignatura respondsToSelector:@selector(creditos)]) { ... }
```

- Podemos también saber el tipo de la clase o los protocolos que implementa

```
[asignatura isMemberOfClass:[UAAsignatura class]]     // YES
[asignatura isMemberOfClass:[NSObject class]]         // NO
[asignatura isKindOfClass:[UAAsignatura class]]       // YES
[asignatura isKindOfClass:[NSObject class]]           // YES

[asignatura conformsToProtocol:@protocol(NSCopying)]
```

Ciclo de vida de las aplicaciones

- Gestionado por un *app delegate* definido por nosotros
  - Implementa el protocolo `UIApplicationDelegate`

- Debemos iniciar la aplicación en

```
- (BOOL)application:(UIApplication *)application
  didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window.rootViewController = self.navigationController;
    [self.window makeKeyAndVisible];

    return YES;
}
```

- Tenemos otros métodos para controlar multitarea, paso a segundo plano, finalización de la aplicación o falta de memoria

¿Preguntas...?