Objective C

Luis Montesano & Ana C. Murillo

Algunos conceptos de OOP

- Clase
- Instancia u objeto
- Mensaje
- Método
- Variable instancia (instance variable o ivar)

Conceptos avanzados

- Encapsulación
- Herencia
- Polimorfismo
- Sistema dinámico de tipos

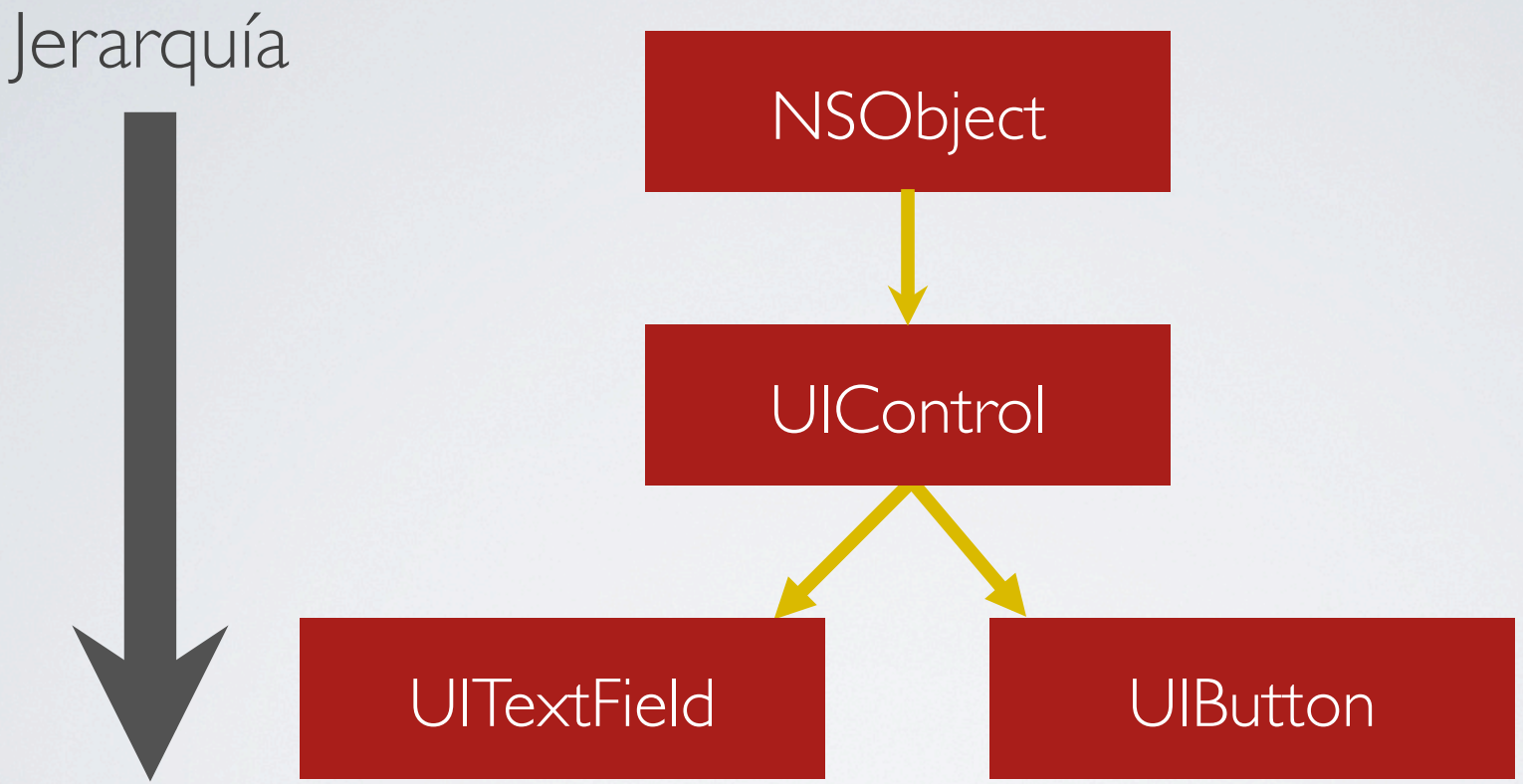
Encapsulación

- Un módulo es una unidad del programa que se puede desarrollar de forma independiente.
- Un módulo bien diseñado debe permitir la conexión con otros módulos de manera simple pero opaca.
  - Proceso por el cual se separa el que del como
  - Posibilita la conexión con otros módulos
- La estrategia para conseguir encapsular suele basarse en:
  - Parte de los componentes puedan exportarse.
  - Los otros permanezcan completamente ocultos en el interior del módulo.

herencia

- Organización jerárquica
- Compartir y reutilizar código
- Encapsular
- Permite la especialización y la ampliación de comportamientos de las super clases

herencia



- La súper clase provee comportamientos que son heredados por las clases descendientes (e.g. manejo de memoria, eventos)
- Las subclases modifican, extienden y especializan estos comportamientos

polimorfismo

- Un sistema monomórfico es aquel en que cada variable tiene un unico tipo
- Un sistema polimórfico es aquel en el que un mismo interfaz puede manejar distintos tipos
  - Función polimórfica: acepta distintos tipos como argumento
- Polimorfismo en programación orientada a objetos resulta de la estructura jerárquica de clases:
  - Vector de objetos (NSArray)
  - Re-escritura de métodos (distinto de sobrecarga)

Sistema de tipos dinámico

- Sistema de tipos estático
  - Cada variable tiene un tipo
  - En el proceso de compilación se pueden verificar la mayor parte de las comprobaciones de tipos
- Sistema de tipos dinámico
  - Los valores tienen tipos, las variables no
  - El chequeo de tipos ser realiza durante la ejecución
  - Mayor flexibilidad (permite implementar *delegation)*

Objective C

- Super conjunto de C
- Tiene algunas extensiones respecto a C, muy sencillas
  - Clases y mensajes
  - Algunos tipos nuevos (anónimo, selectores)
- Sigue la sintaxis de mensajes de smalltalk
- Herencia simple. Cada clase hereda de una y solo una súper clase
- Permite la definición de protocolos que definen comportamientos comúnes para clases distintas
- Permite usar un sistema de tipos dinámico, aunque también se pueden declarar los tipos

Durante la ejecución

- Los objetos son creados siempre en memoria dinámica
- No hay objetos en la pila (los no informáticos, deberíais preguntarme ahora!!!)
- Se entregan los mensajes usando `objc_msgSend()`
  - Argumentos: el objeto, los selectores (nombre del método) y los argumentos
- Se puede obtener mucha información del propio objeto (introspection) como la clase, la existencia de métodos, su relación jerárquica con otras clases
  - Necesario para manejar un sistema dinámico

Clases, objetos e instancias

- Clases e instancias son ambos objetos
- La clase es un objeto que define el patrón para crear instancias (u objetos) de esa clase
- Normalmente se compone de

1. Datos para representar y mantener un estado
2. Métodos que implementan el comportamiento del objeto basado en el estado

Instancias

- En Objective C todo son objetos (excepto por su compatibilidad con C)
- Los datos son por tanto instancias, llamadas variables instancia
- Una variable instancia se puede definir como:

```
@public: // NO ES RECOMENDABLE !!!
UITextField *textField;
@protected:
UILabel *label;
NSString *string;
```

  - Las variables instancia son normalmente (y por defecto) privadas
  - Se acceden mediante los métodos de lectura (*getter*) y escritura (*setter*)

METODOS

- Existen dos tipos de métodos:

1. De Clase

- `+ (id)alloc;`
- `+ (id)identifier;`

1. De Instancia

- `- (id)init;`
- `- (float)height;`
- `- (void)walk;`

Metodos de Clase y de instancia

- Los métodos de instancia:
  - Se declaran con -
  - Son los más habituales
  - Pueden acceder en la implementación a las variables instancia definidas en la clase
- Los métodos de clase:
  - se declaran con +
  - implementan creación de objetos
  - información compartida entre instancias o creación de instancias compartidas
  - Son de la clase, no pueden acceder a las variables instancia de la clase porque no existen. Llamar a self, implica llamar a la clase!

Sintaxis de mensajes

```
[receiver message];
[receiver message: arg1];
[receiver message: arg1 andArg: arg2 ...];
```

- receiver puede ser self (el propio objeto)
- también puede ser super (la clase padre)
- Los mensajes se pueden anidar

```
[self setMyViewController:aViewController];
```

```
MyViewController *aViewController = [[MyViewController alloc]
initWithNibName:@"MyViewController" bundle:[NSBundle mainBundle]];
```

Nombres de métodos

- Objective-C methods are composed of a few different components. I'll list the components here, examples follow:

1. If there is a return value, the method should begin with the property name of the return value (for accessor methods), or the class of the return value (for factory methods).

2. A verb describing either an action that changes the state of the receiver or a secondary action the receiver may take. Methods that don't result in a change of state to the receiver often omit this part.

3. A noun if the first verb acts on a direct object (for example a property of the receiver) and that property is not explicit in the name of the first parameter.

4. If the primary parameter is an indirect object to the main verb of the method, then a preposition (i.e. "by", "with", "from") is used. This preposition partly serves to indicate the manner in which the parameter is used but mostly serves to make the sentence more legible.

5. If a preposition was used and the first verb doesn't apply to the primary parameter or isn't present then another verb describing direct action involving the primary parameter may be used.

6. A noun description (often a class or class-like noun) of the primary parameter, if this is not explicit in one of the verbs.

7. Subsequent parameter names are noun descriptions of those parameters. Prepositions, conjunctions or secondary verbs may precede the name of a subsequent parameter but only where the subsequent parameter is of critical importance to the method. These extra terms are a way to highlight importance of secondary parameters. In some rarer cases secondary parameter names may be a preposition without a noun to indicate a source or destination.

Properties (propiedades)

- Para cada variable instancia que queremos que sea visible, se debe proporcionar los métodos de escritura y lectura (*getter* and *setter):*

- `-(void) setAge(NSNumber *) myAge;`

- `-(id) age;`

- Convención de nombres

- Objective C proporciona un mecanismo para la definición de los métodos de acceso a las variables instancia

- `@property(readonly) NSNumber *age;`

- Reemplaza la definición en el interfaz de los métodos de escritura y lectura

PROPERTIES

- Objective C también proporciona un mecanismo para la implementación automática de los métodos de acceso
- `@synthesize age;`
- También permite usar la notación con punto
- `self.age=[NSNumber numberWithFloat:3.0];`
- `NSNumber newAge = self.age;`

PROPERTIES

```
@synthesize maximumSpeed;

...

-(int)maximumSpeed{

   if (self.maximumSpeed > 65)

   {

      return 65;

    }

  else {

      return maximumSpeed;

  }

}
```

tipos dinámicos

- Tipos estáticos
  - `Alumno *unAlumno;`
- Tipos dinámicos
- `id unAlumno;`
- No se utiliza `id *, id` ya es un puntero, `id *` es un puntero a puntero
- El compilador chequeará la primera definición, pero no la segunda
- En el segundo caso, el compilador acepta cualquier mensaje que conozca (definido en alguna clase)
- El programa fallará si el objeto llamado no posee ese método (aunque el compilador no se queja)
- Se puede forzar una conversión de tipos

Que hemos entendido?

```
@interface Ship : Vehicle

- (void)shoot;

@end
```

```
Ship *s = [[Ship alloc] init];

[s shoot]; // ???
```

```
Vehicle *v = s;

[v shoot];          // ???

Ship *castedVehicle = (Ship *)someVehicle;

[castedVehicle shoot]; // ??
```

```
id obj = ...;

[obj shoot]; // ???

[obj lskdfjslkfjslfkj]; // ???

[(id)someVehicle shoot]; // ???
```

```
Vehicle *tank = [[Tank alloc] init];

[(Ship *)tank shoot];     // ???
```

Que hemos entendido?

```
@interface Ship : Vehicle

- (void)shoot;

@end
```

```
Ship *s = [[Ship alloc] init];

[s shoot]; //
```

```
Vehicle *v = s;

[v shoot];          // Avisa, no hay fallo

Ship *castedVehicle = (Ship *)someVehicle;

[castedVehicle shoot]; // Avisa, no hay fallo
```

```
id obj = ...;

[obj shoot]; // El compilador

[obj lskdfjslkfjslfkj]; // El compilador avisa, fallo

[(id)someVehicle shoot]; // no avisa, puede fallar
```

```
Vehicle *tank = [[Tank alloc] init];

[(Ship *)tank shoot; // No avisa, fallo en ejecución
```

Nil

- Valor de una variable instancia que no apunta a ningun objeto (puede ser de tipo `id` o `Alumno *)`
- Es realmente 0
- Es el valor que NSObject pone en la inicialización de los objetos
- Un mensaje a `nil` devuelve normalmente 0 y es seguro
  - Existen casos en que por compatibilidad con C puede fallar (e.g. devolución de una estructura, veremos por que se usan estructuras más adelante)
- Se puede usar directamente en un `if`

BOOL

- Objective C definió su propio tipo booleano
- Se puede usar `false` y `true`, pero lo mas común es `YES` y `NO`
- Como en C, 0 y 1. También se chequea igual en un condicional

Introspección

- La capacidad de buscar información de un objeto sobre si mismo
- Es importante debido a la naturaleza dinámica de las asociaciones de Objective C
- Permite conocer la clase, la existencia de métodos ....

introspección

- Recuperar la clase de un objeto

```
Class myClass = [myObject class];

NSLog(@"My class is %@", [myObject className])
```

- Chequear si un objeto pertenece a una clase (incluyendo subclases):

```
if ([myObject isKindOfClass:[UIControl class]]) {   // something}
```

- O por una clase en particular (sin subclases):

```
if ([myObject isMemberOfClass:[NSString class]])

{ // something string specific}
```

INTROSPECCIÓN

- Verificar si un objeto responde a un método. Utiliza un token especial como argumento (selector):

```
@selector(shoot)  or @selector(foo:bar:)
```

```
if ([obj respondsToSelector:@selector(shoot)]) {
```

- El tipo de un selector es SEL, e.g., -

```
(void)performSelector:(SEL)action
```

- Esto es importante para enlazar una acción de respuesta a un evento (target action) programando en lugar de con Interface Builder

```
[btn addTarget:self action:@selector(shoot) ...]
```

- Enviar un mensaje a un objeto NSObject con un selector

```
[obj performSelector:@selector(shoot)]
```

```
[obj performSelector:@selector(foo:) withObject:x]
```

Utilizar Introspección

```
id anObject = ...;

SEL aSelector = @selector(someMethod:);

if ([anObject respondsToSelector:aSelector]) {

    [anObject performSelector:aSelector withObject:self];

}
```

Comparación de objetos

- Todo son referencias (punteros)!
- Diferencia entre identidad e igualdad?

Comparación de objetos

```
if (object1 == object2) {

 // Exactamente el mismo objeto!

}
```

```
if ([object1 isEqual: object2]) {

 // Logicamente iguales, pero pueden ser distinto objeto

}
```

Escribir mensajes de log

```
NSLog((NSSting *) string)
```

- Especificadores de formato como en C

```
NSString *myString = [NSString stringWithFotrmat:@"%d", 3);
```

- Se puede escribir directamente un objeto
- NSObjeto implementa el método descripción

```
NSLog([self description])
```

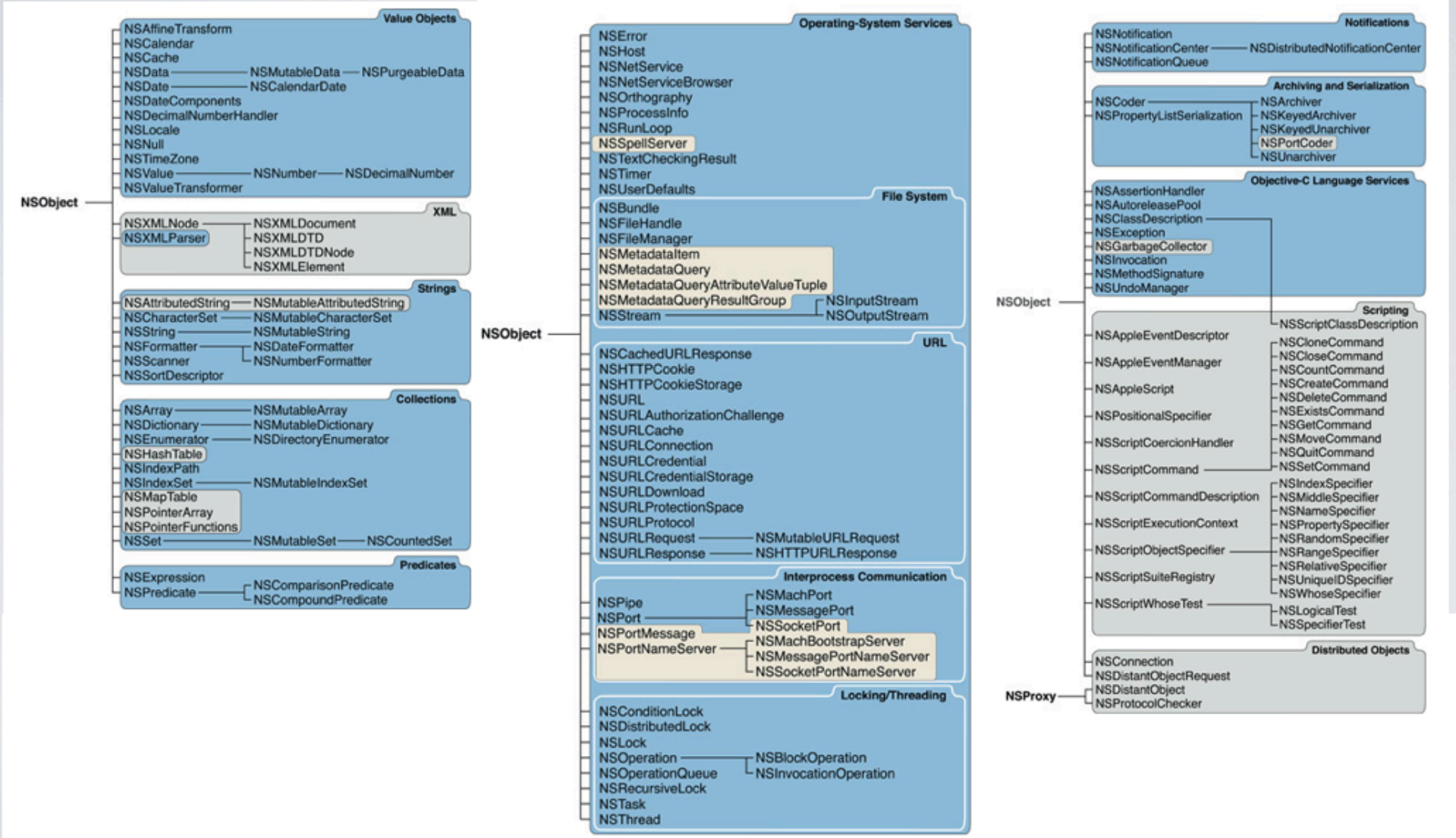
```
NSString *myString = [NSString stringWithFotrmat:@"%@", [self description]);
```

- El método se puede sobreescribir para dar información extra de las subclases

Foundation Classes

- El entorno *Foundation* es una jerarquía de clases
- Esta clases definen una capa base de funcionalidades y paradigmas sobre Objective C que facilitan y simplifican el desarrollo de aplicaciones
- Vamos a ver algunos ejemplos muy útiles

Una vista global



NSArray

- NSArray: Colección ordenada de objetos (vector)
  - No se puede modificar
  - Métodos de creación e inicialización

```
+ (id)array

- (id)initWithObjects:(id)firstObj, ...
```

  - Métodos de ordenación, consulta

```
- (id)objectAtIndex:(NSUInteger)index

- (NSUInteger)count
```

- NSMutableArray: Subclase de NSArray
  - Se puede modificar
  - Añade metodos para añadir elementos, modificarlos ...

NSArray

- NSArray: Colección ordenada de objetos (vector)
  - No se puede modificar
  - Métodos de creación e inicialización

```
+ (id)array

- (id)initWithObjects:(id)firstObj, ...
```

  - Métodos de ordenación, consulta

```
- (id)objectAtIndex:(NSUInteger)index

- (NSUInteger)count
```

- NSMutableArray: Subclase de NSArray
  - Se puede modificar
  - Añade metodos para añadir elementos, modificarlos ...

VER DOCUMENTACIÓN

NSSTRING

- NSString es una cadena de caracteres.
  - No se puede modificar
  - Tiene su versión modificable NSMutableString

NSDIctionary

- NSDictionary: Clase que define conjuntos de pares clave-valor

- La clase de la clave debe implementar una función hash `- (NSUInteger)hash` y una comparación `(BOOL)isEqual:(NSObject *)obj`

Los objetos de la clase NSString son los más utilizados como claves (ya que tienen implementados ambos métodos).

- Métodos importantes:

```
- (int)count
- (id)objectForKey:(id)key
- (NSArray *)allKeys
- (NSArray *)allValues
```

- Versión modificable: NSMutableDictionary

```
- (void)setObject:(id)object forKey:(id)key
- (void)removeObjectForKey:(id)key
- (void)addEntriesFromDictionary:(NSDictionary *)dictionary
```

NSSet

- NSSet: Conjunto de objetos no ordenados sin repetición
  - Versión modificable: NSMutableSet

OTras clases

- NSUserDefaults
- NSDate
- NSFileManager
- NSThread
- ....

ITERADORES-ENUMERADORES

- NSEnumerator
  - Clase abstracta. Se crea a partir de objetos que representan colecciones

```
NSArray *anArray = // ... ;

NSEnumerator *enumerator = [anArray objectEnumerator];
id object;

while ((object = [enumerator nextObject])) {
    // do something with object...
}
```

- Objective-C implementa enumeración rápida (*fast enumeration*)
  - Más rápido (?), seguro (no se puede modificar) y conciso

```
for (type myVariable in expression) { ... }

Type existingItem;
for (existingItem in expression) { ... }
```

ENUMERADORES

```
NSArray *array = [NSArray arrayWithObjects:

        @"One", @"Two", @"Three", @"Four", nil];

for (NSString *element in array) {
    NSLog(@"element: %@", element);
}

NSDictionary *dictionary = [NSDictionary
dictionaryWithObjectsAndKeys:
    @"quattuor", @"four", @"quinque", @"five", @"sex",
@"six", nil];

NSString *key;
for (key in dictionary) {
    NSLog(@"English: %@, Latin: %@", key, [dictionary
valueForKey:key]);
}
```

Protocolos

- Similar a un interfaz de JAVA
  - Declara métodos que pueden implementar cualquier clase

```
@protocol MyProtocolName
//Method declarations go here
-(void)doAction;
@optional
-(void)doSthElse
@required
-(int)getValue;
@end
```

- Una clase declara que implementa un protocolo

```
@interface MyClass : NSObject <foo>

//Method decla
```

Protocolo

- Una variable o un argumento también pueden especificar la necesidad de cumplir con un protocolo:

```
id <MyProtocol> obj = [[MyClass alloc] init]
```

```
-(NSNumber *) getNumber:(id <MyProtocol>)objWithMyProtocol;
```

- El compilador comprueba:
  - Si una clase declara que cumple un protocolo y no implementa los métodos obligatorios
  - Si se asigna un objeto a una variable y el objeto no cumple el protocolo
  - Si se pasa un objeto a un método y no cumple con el protocolo especificado

Delegados

- Es un patrón de diseño común en programación orientada a objetos
- Un objeto, en lugar de implementar un conjunto de acciones, las delega
- Para implementar este tipo de diseños se usa frecuentemente protocolos
  - Una clase que quiere delegar, define un protocolo y tiene una *property* que sera el objeto delegado
  - La clase delegado implementa esos métodos

EJEMPLO DELEGADO-PROTOCOLO

```
@interface TCScrollView : NSView {

   id delegate; // A delegate that wants to act on events in this view
}

-(IBAction)scrollToCenter:(id)sender; // A method that can be bound to a button in
the UI
-(void)scrollToPoint:(NSPoint)to;

// Accessors. Implementation not shown.
@property (nonatomic, assign) id <MyProtocol> delegate;

@end


@protocol MyProtocol

@optional
-(BOOL)scrollView:(TCScrollView*)scrollView shouldScrollToPoint:(NSPoint)newPoint;

@end
```

EJEMPLO DELEGADO-PROTOCOLO

- `@implementation TCScrollView`

```
-(IBAction)scrollToCenter:(id)sender; { [self scrollToPoint:NSPointMake(0,0)];}

-(void)scrollToPoint:(NSPoint)to {

   BOOL shouldScroll = YES;

   // Comprobar que hay un delegado y que implementa el protocolo
   if(delegate && [delegate respondsToSelector:@selector
       (scrollView:shouldScrollToPoint:)])
     // ask it if it's okay to scroll to this point.
     shouldScroll = [delegate scrollView:self shouldScrollToPoint:to];

   // If not, ignore the scroll request.
   if(!shouldScroll)
      return;

   // Scrolling code
}

@end
```

ejemplo delegado-protocolo

```
@interface MyCoolAppController : NSObject <TCScrollViewDelegate> {


    IBOutlet TCScrollView* scrollView;
}

@end
```

EJEMPLO PROTOCOLO-DELEGADO

```
@implementation MyCoolAppController

-(void)awakeFromNib {

  [scrollView setDelegate:self];
}

-(BOOL)scrollView:(TCScrollView*)scrollView shouldScrollToPoint:
(NSPoint)newPoint {

  if(newPoint.x > 0 && newPoint.y > 0)
    return YES;

  return NO;
}

@end
```

ciclo de vida de los objetos
gestión de memoria

Creación de un objeto

- Hemos visto varios ejemplos de creación de objetos

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

```
Student *MyStudent = [[Student alloc] init];
```

- Método general: reservar memoria e inicializar

  Reservar: `+(id)alloc`

  Inicializar: `-(id)init`

- La clase NSObject define un método de inicialización por defecto
- Pone todas las variables instancia a 0
- Se pueden crear nuevos inicializadores

  •Normalmente un inicializador con pocos parámetros usa parámetros por defecto (y puede llamar a un inicializador completo)

```
Student *MyStudent = [[Student alloc] initWithName:name age:Years];
```

```
Student *MyStudent = [[Student alloc] initWithName:name]; // Años por defecto
```

Creación de objetos

- Cuando implementamos nuestro propio método de inicialización debemos inicializar la super clase

```
-(id)init{

   // Inicializar la superclase primero
  if(self = [super init]){
      // Inicializaciones propias de la clase
  }

  return self;
}
```

Manejo de memoria

- Recordad: los objetos viven todos en memoria dinámica
- ¿Que ocurre cuando terminamos de usar un objeto?
  - alloc y dealloc
  - Como en C, deben ser balanceadas para evitar *memory leaks o terminaciones abruptas*
  - Nunca llamamos a dealloc
  - En el iPhone no hay recolección de basura (garbage collection)

Contando referencias

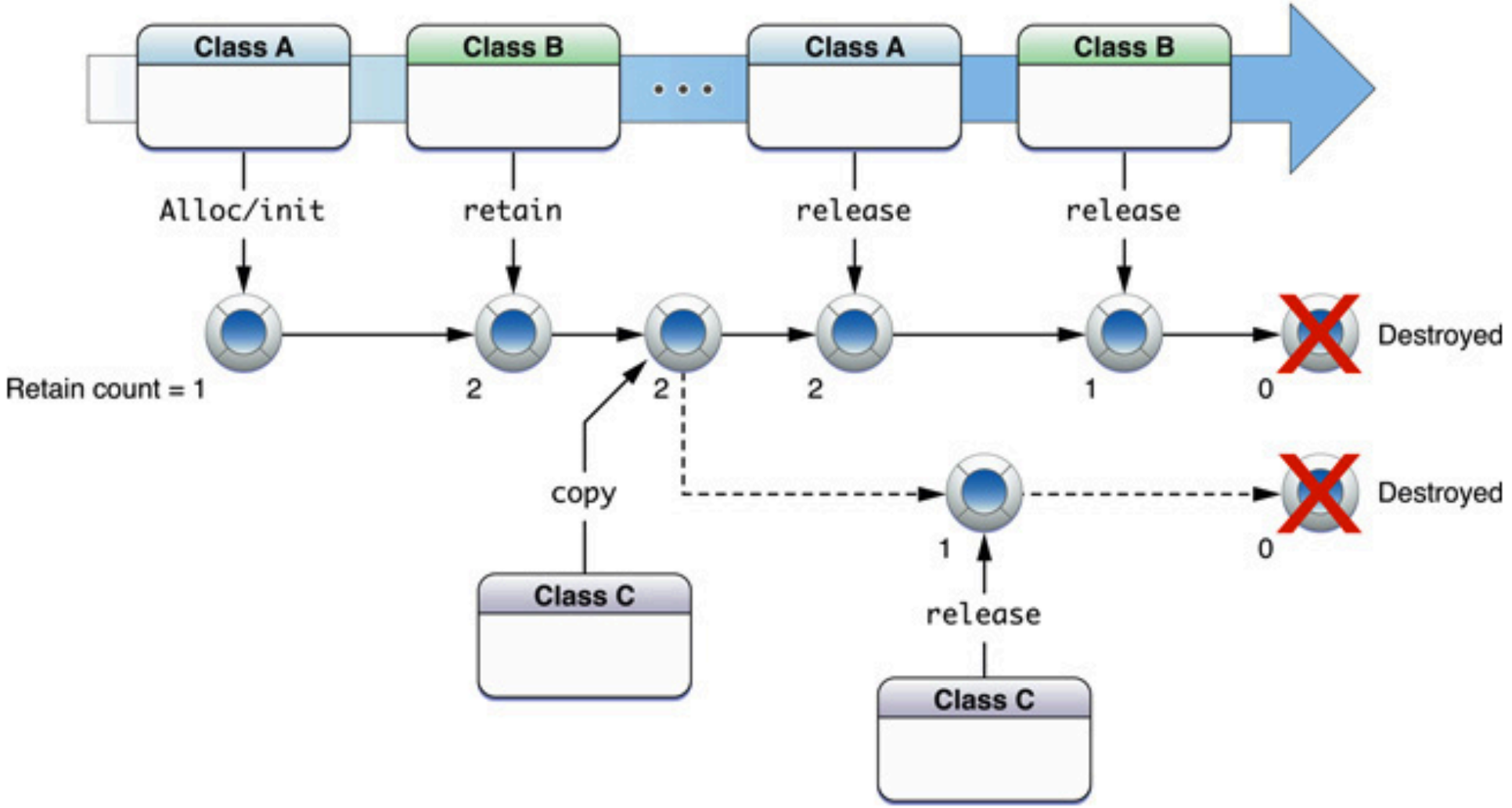
- Cada objeto tiene un contador de referencias
  - Definido en NSObject
  - Mientras es contador sea mayor que 0, el objeto será valido
  - Cuando el contador llegue a 0 el objeto será destruido
  - Incrementar el contador:
    - alloc, copy, retain
  - Decrementar el contador:
    - release
  - Cuando el contador llega a 0, el método dealloc se llama automáticamente

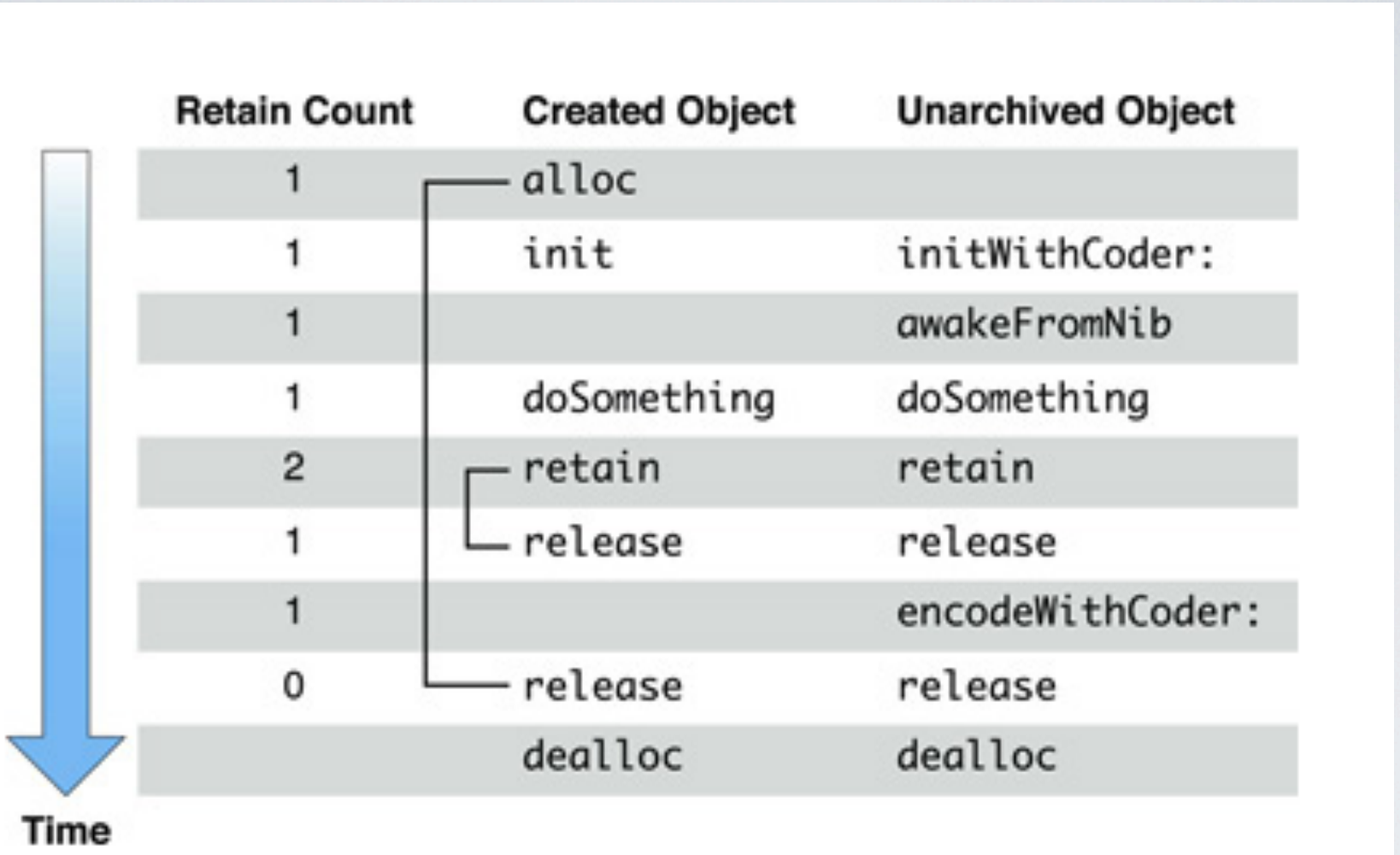
RETAIN

- Un objeto puede ser dueño de otro objeto
  - Cuando hacemos alloc, el objeto que hace alloc es dueño del objeto
  - Cuando enviamos el mensaje retain a un objeto, nos hacemos dueños del objeto
  - Cuando pedimos a otra clase que cree un objeto para nosotros, no somos dueños del objeto
    - Podemos usar el objeto dentro del método en el que hemos pedido que se cree el objeto, pero no más tarde
    - Si queremos usarlo más tarde, tendremos que enviarle el mensaje retain

Release

- Cuando hemos terminado con un objeto debemos hacer release (decrementar el numero de referencias a ese objeto)
  - Si el número de referencias es 0 (último dueño del objeto), el objeto se eliminara
  - A partir de este momento no se puede enviar mensajes a este objeto. Abortará la aplicación
  - Es un proceso sin vuelta atrás


Class A
Class B
Class A
Class B
Alloc/init
retain
release
release
Retain count = 1
2
2
2
1
0
Destroyed
copy
Class C
1
0
Destroyed
release
Class C


Retain Count
Created Object
Unarchived Object
1
alloc
1
init
initWithCoder:
1
awakeFromNib
1
doSomething
doSomething
2
retain
retain
1
release
release
1
encodeWithCoder:
0
release
release
dealloc
dealloc
Time

Autorelease

- ¿Como se pueden compartir objetos?
- E.g. 1: devolver un objeto que hemos creado
- E.g. 2: Funciones que crean objetos y no son alloc ni copy

```
[NSString stringWithFormat:@"%f",3.0]
```

- Usamos autorelease en lugar de release antes de devolver el objeto
- UIKit se encargara de enviar el release en algún momento en el futuro (siempre seguro para poder recuperar el objeto).

AUTORELEASE

```
-(Student *)registerWithName:(NSString *)name {

    Student *newStudent = [[Student alloc]
initWithName:name];

    [newStudent autorelease];

     return newStudent;

  // Tambien vale return [newStudent autorelease]

}
```

Autorelease

- Con NSString y NSMutableString

AUTORELEASE

- Properties

Objetos desde un NIB

- Wake UP
  - -(void) awakeFromNib (método de NSObject
  - -(void)viewDidLoad (para subclases de UIViewController)