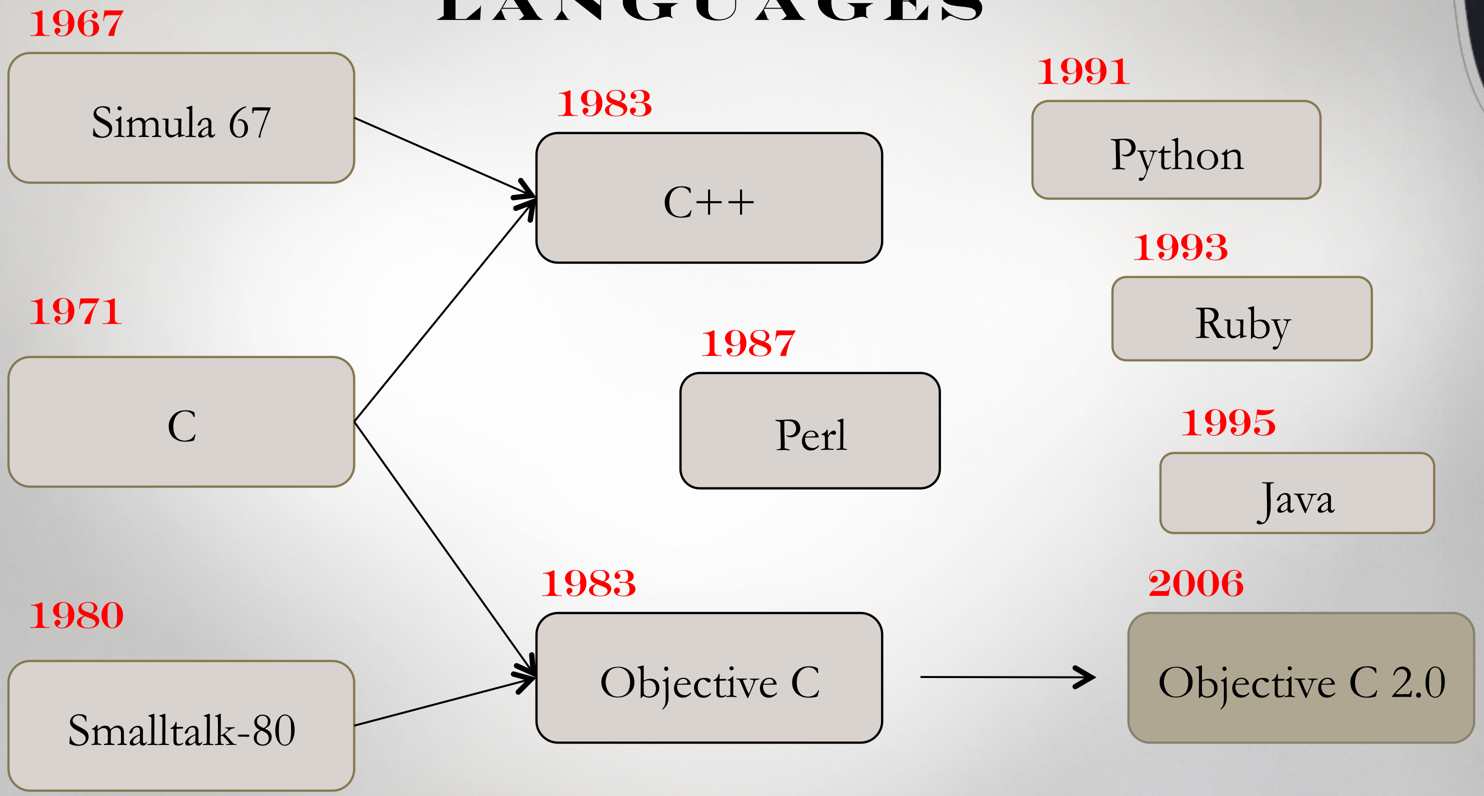
OBJECTIVE-C

A modern-person's guide to iPhone code development:

YE OLDE PROGRAMMING LANGUAGES



1967

Simula 67

1971

C

1980

Smalltalk-80

1983

C++

1987

Perl

1983

Objective C

1991

Python

1993

Ruby

1995

Java

2006

Objective C 2.0

*Source: Computer Languages Timeline * http://www.levenez.com/lang/*

WHAT IS OBJECTIVE-C?

- An object oriented language which lies on top of the C language .
- Its primary use in modern computing is on Mac OS X as a desktop language and also on iPhone OS (or as it is now called: iOS).
- It was originally the main language for NeXTSTEP OS, also known as the operating system Apple bought and descended Mac OS X from, which explains why its primary home today lies on Apple's operating systems.
- Because any compiler of Objective-C will also compile any straight C code passed into it, we have all the power of C along with the power of objects provided by Objective-C.

PRIMITIVES

- ❖ The usual C Types
  - int, float, ...
- ❖ It's own boolean *(ObjC forked before C99)*
  - BOOL
  - *Takes values* ***NO=0 and YES=1***
- ❖ Some special types
  - id, Class, SEL, IMP
  - ***nil is used instead of null.***

STRINGS

- *Always use (NSString *) instead of C Strings*
  - Inline :

    ```
    @"This is an inline string";
    ```

  - Assigned:

    ```
    NSString *str = @"This is assigned to a variable";
    ```

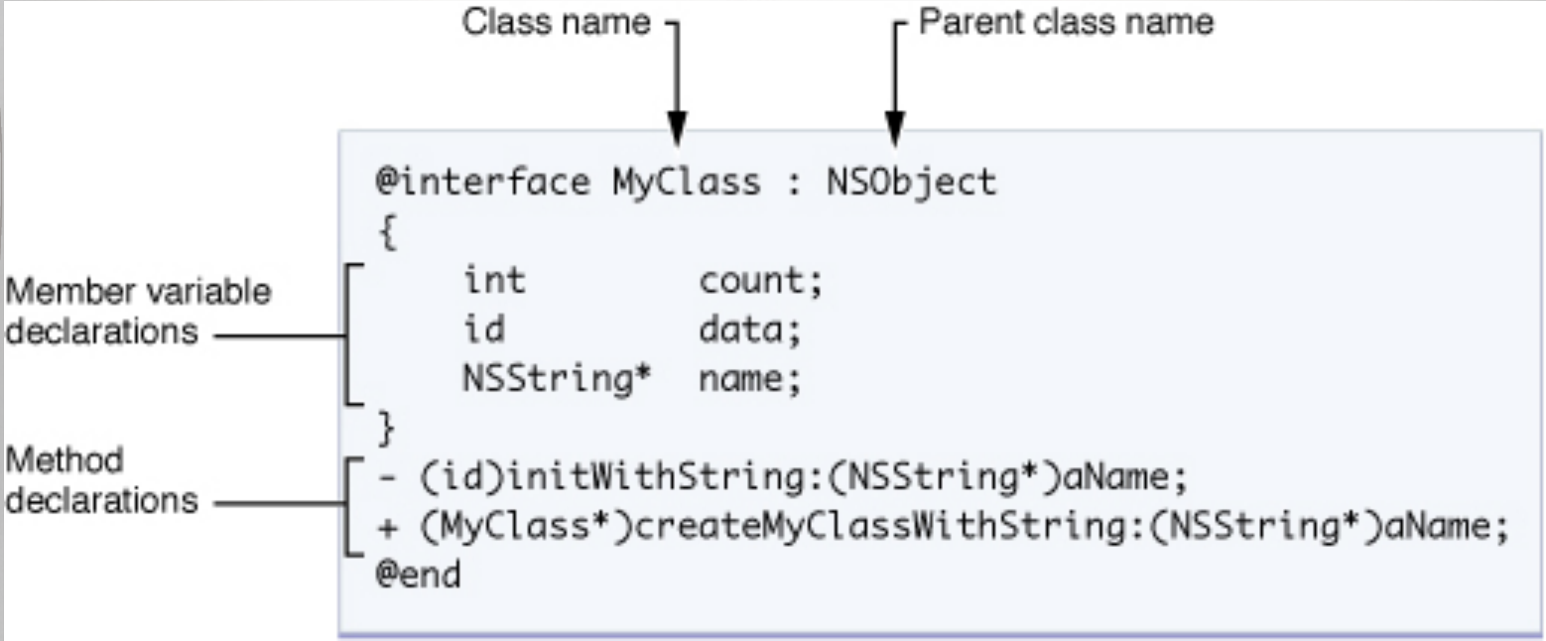
You must define constant strings this way, lest you incite a programmic crash

- *leaving out the @, causes a crash!*
- *Objective C Pointers aren't abstracted, like java is. Look at this, in the notes!*

INTERFACE AND IMPLEMENTATION

- A simple class in Objective-C , by default, has two files:
  - The **implementation** file which is a file that ends with a suffix of **.m**
  - The **interface** file which is a file that ends with a suffix of **.h**.

CLASS DECLARATION



```
@interface MyClass : NSObject
{
    int         count;
    id          data;
    NSString*   name;
}
- (id)initWithString:(NSString*)aName;
+ (MyClass*)createMyClassWithString:(NSString*)aName;
@end
```

INTERFACE

```
#import <COCOA Cocoa.h>

    @interface Person : NSObject {

        //This is where attributes go
       NSString *name;
       NSNumber *age;
       NSString  *address;

    }

    //This is where methods go
    - (void)updageAddress;

    @end
```

A system allowing for the declaration of clases and methods

IMPORTING THE INTERFACE

- ❖ The interface file must be included in any source module that depends on the class interface
- ❖ The interface is usually included with the `#import` directive.

REFERRING TO OTHER CLASSES

- An interface file declares a class and, by importing its superclass, implicitly contains declarations for all inherited classes, from NSObject on down through its superclass.
- If the interface mentions classes not in this hierarchy, it must import them explicitly or declare them with the `@class` directive:

```
@class SyFy, FlyingMachine;
```

IMPLEMENTATION

```
     #import “Person.h"

     @implementation Person

     -(void) updateAddress {
// code goes here to add gas
     }

     @end
```

MESSAGES

- Method Calling v. Message Passing
- In Objective-C, we call object methods by passing messages.
- A message is sent to the instance
- The message is the method we want to apply.
- Programmatically it looks like this:

```
[recipient message];
```

- With *No* arguments

```
          [object message];
          [aPerson init];
```

- With *1* Argument

```
          [object  message:value];
          [aPerson initWithLast:@”Smith”];
```

- With *2* arguments

```
          [object  message:value1  arg2:value2];

[aPerson initWithLast:@”Smith” andFirst:@”John”];
```

MORE ON MESSAGES

❖ Messages can be sent to **classes:**

```
[Person alloc];
```

❖ Messages can be nested:

```
Person* p = [[Person alloc]
initWithName:@”John”];
```

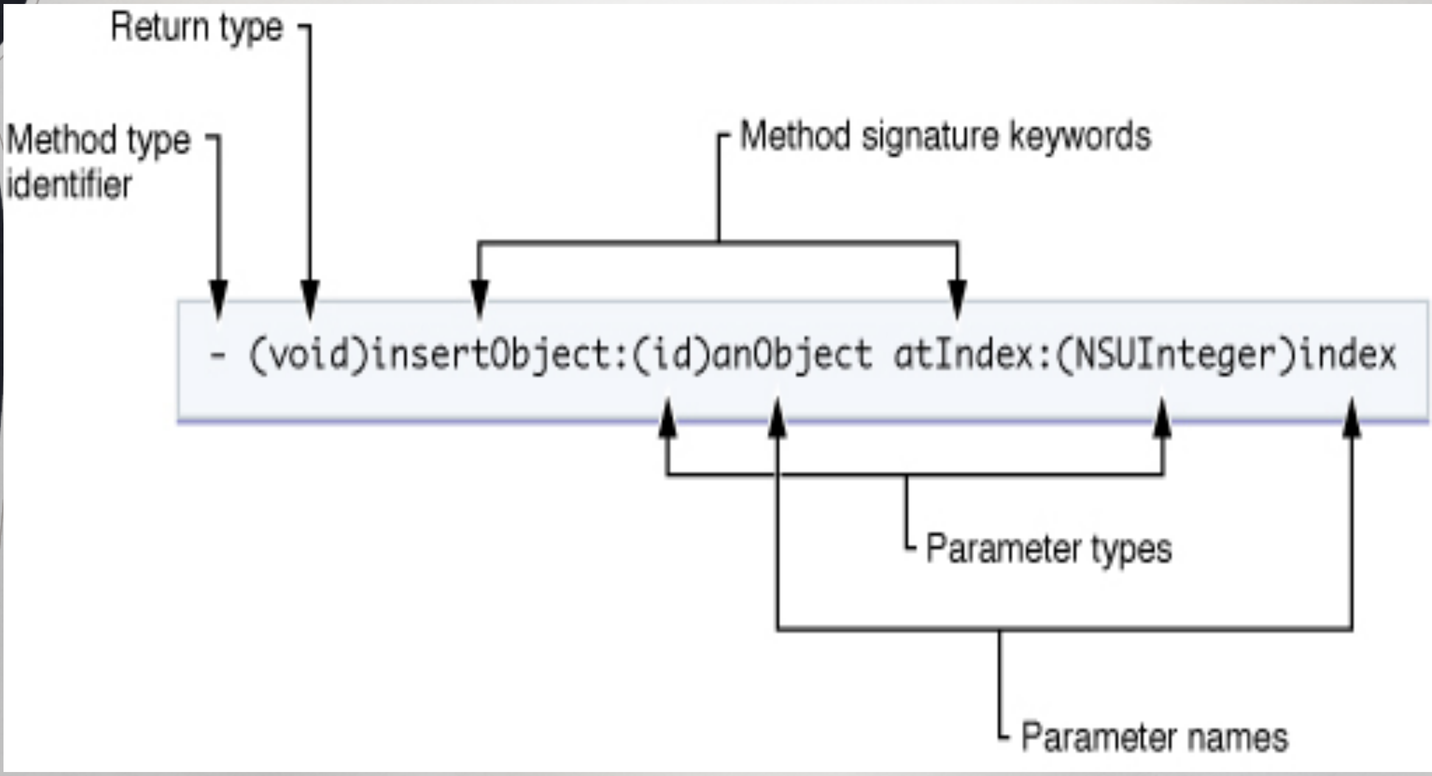
- Equal to:

```
  Person* p = [Person alloc];
  [p initWithName:@”John”];
```

❖ A crucial difference between function calls and messages is that a function and its arguments are joined together in the compiled code, but a message and a receiving object aren’t united until the program is running and the message is sent.

❖ Therefore, the exact method that’s invoked to respond to a message can only be determined at runtime, not when the code is compiled.

METHOD DECLARATION



```
- (void)insertObject:(id)anObject atIndex:(NSUInteger)index
```

METHODS

❖ Define a method:

```
- (id)initWithFirst:(NSString*)firstName
                                                  andLast:
(NSString*)lastName;
```

- Call a method:

```
          [aPerson initWithFirst:@”John”
                                        andLast:@”Smith”];
```

CLASS EXAMPLES

- **NSString** is a string of text that is immutable.
- **NSMutableString** is a string of text that is mutable.
- **NSArray** is an array of objects that is immutable.
- **NSMutableArray** is an array of objects that is mutable.
- **NSNumber** holds a numeric value.
- If an object is **immutable** that means when we create the object and assign a value then it is static. The value can not be changed.
- If an object is **mutable** then it is dynamic, meaning the value can be changed after creation.

Objective-C does not have syntax to mark classes as abstract, nor does it prevent you from creating an instance of an abstract class.

OBJECT TYPING

- Every object is of type

```
id
```

- General type for any kind of object regardless of class
- Can be used for both instances of a class and class objects themselves.
- As a pointer to the instance data of the object.

```
id person; (dynamic typing)
```

- Can declare a more specific type:

```
Person*  person;  (static typing)
```

TYPE INTROSPECTION

- Instances can reveal their types at runtime.
- *isMemberOfClass*: defined in the NSObject class, checks whether the receiver is an instance of a particular class.

```
        if ( [anObject  isMemberOfClass:someClass] )
```

- *isKindOfClass*: checks more generally whether the receiver inherits from or is a member of a particular class (whether it has the class in its inheritance path):

```
        if ( [anObject  isKindOfClass:someClass] )
```

DYNAMIC TYPING

- ❖ The *id* type is completely nonrestrictive. By itself, it yields no information about an object, except that it is an object.
- ❖ A program typically needs to find more specific information about the objects it contains.
- ❖ Each object has to be able to supply it at runtime.
- ❖ The *isa* instance variable identifies the object's **class**—what kind of object it is.
- ❖ Objects with the same behavior (methods) and the same kinds of data (instance variables) are members of the same class.

EQUIVALENT STATEMENTS

- `Person *p = [[Person alloc] init];`
- `id p= [[Person alloc] init];`

POINTERS AND INITIALIZATION

```
int main (int argc, const char * argv[]) {
        NSString *testString;
        testString = [[NSString alloc] init];
        testString = @“This is a test string !";
        NSLog(@"testString: %@", testString);
        return 0;
}
```

CREATING INSTANCES

❖Objective-C has a lot of conventions that are not enforced by the compiler:

❖Allocates memory and returns a pointer.

```
+(id)alloc;
```

❖Initializes the newly allocated object.

```
-(id)init;
```

The alloc method dynamically allocates memory for the new object's instance variables and initializes them all to 0—all, that is, except the isa variable that connects the new instance to its class.

NSLOG COMMON STRING FORMAT SPECIFIERS

%@	Objective-C object using the description or descriptionWithLocale: results
%%	The “%” literal character %d Signed integer (32-bit)
%u	Unsigned integer (32-bit)
%f	Floating-point (64-bit)
%e	Floating-point printed using exponential (scientific) notation (64-bit)
%c	Unsigned char (8-bit)
%C	Unicode char (16-bit)
%s	Null-terminated char array (string, 8-bit)
%S	Null-terminated Unicode char array (16-bit)
%p	Pointer address using lowercase hex output, with a leading 0x
%x	Lowercase unsigned hex (32-bit)
%X	Uppercase unsigned hex (32-bit)

THE SCOPE OF INSTANCE VARIABLES

- ❖ @private
    - The instance variable is accessible only within the class that declares it.
- ❖ @protected – (default)
    - The instance variable is accessible within the class that declares it and within classes that inherit it.
- ❖ @public
    - The instance variable is accessible everywhere.
- ❖ @package
    - Using the modern runtime, an @package instance variable acts like @public inside the image that implements the class, but @private outside.
- ❖ By default, all unmarked instance variables are @protected.

INITIALIZING A CLASS OBJECT

❖If a class makes use of static or global variables, the ***initialize*** method is a good place to set their initial values.

```
    + (void)initialize {
  if (self == [ThisClass class]) {
      // Perform initialization here. ...
  }
}
```

Because of inheritance, an initialize message sent to a class that doesn’t implement the initialize method is forwarded to the superclass, even though the superclass has already received the initialize message.

CLASS NAMES IN SOURCE CODE

❖ The class name can be used as a type name for a kind of object

```
Rectangle *anObject;
```

❖ As the receiver in a message expression, the class name refers to the class object

```
if ( [anObject isKindOfClass:[Rectangle class]] )
```

❖ If you don’t know the class name at compile time but have it as a string at runtime, you can use NSClassFromString to return the class object:

```
NSString *className; ...
if ( [anObject isKindOfClass:NSClassFromString(className)] )
```

❖ You can test two class objects for equality using a direct pointer comparison.

```
if ([objectA class] == [objectB class]) { //...
```

VARIABLES AND CLASS OBJECTS

- ❖ For all the instances of a class to share data:
- ❖ The simplest way to do this is to declare a variable in the class implementation file:

```
int MCLSGlobalVariable;
@implementation MyClass
```

- ❖ In a more sophisticated implementation, declare a variable to be static, and provide class methods to manage it.
- ❖ Declaring a variable static limits its scope to just the class—and to just the part of the class that's implemented in the file.
- ❖ Unlike instance variables, static variables cannot be inherited by, or directly manipulated by, subclasses.

PROPERTY AND SYNTHESIZE

```
@interface Person : NSObject {

           //This is where attributes go
           NSString *name;
           NSNumber *age;
           NSString  *address;

        }
        @end
```

These all need
Setters and Getters

```
@interface Person : NSObject {

               //This is where attributes go
              NSString *name;
              NSNumber *age;
              NSString  *address;

            }
            @property(readwrite, retain) NSString* name;
            @property(readwrite, retain) NSString* address
            @property(readwrite, retain) NSNumber* age;

            @end
```

❖ Think of a property as a compiler macro that generate the getter and setter for you.

```
@implementation Person
@synthesize name,age, address;
  …
}
@end
```

❖@property replaces all of the interface method declarations for getters and setters,

❖@synthesize replaces the actual methods themselves.

PROPERTY ATTRIBUTES

```
@property (readonly) int key;
@property (nonatomic, retain) NSString *title;
@property (nonatomic, copy) NSString *first_name;
```

Format:
@property (**attributes)** type name**;**

Writability
readwrite (default)
readonly

Setter Semantics
assign (default)
retain
copy
Atomicity
nonatomic
(no "atomic" attribute
but this is the default)

Source:

http://developer.apple.com/documentation/Cocoa/Conceptual/ObjectiveC/Articles/chapter_5_section_3.html

CALLING PROPERTIES

```
@property (nonatomic, copy) NSString*name;
```

Use “dot notation” :

```
person.name = @”John Smith";
a = person.name;
```

Or Message passing:

```
[person setName: @”John Smith"];
a = [person getName];
```

USING DOT SYNTAX

- Use **dot syntax** to invoke accessor methods using the same pattern as accessing structure elements.

```
myInstance.value = 10;
```

  - which is equivalent to:

```
[myInstance setValue:10];
```

- You can read and write properties using the dot (.) operator
- Accessing a property *property* calls the get method associated with the property (by default, *property*)
- Setting it calls the set method associated with the property (by default, **set***Property***:**).
- An advantage of the dot syntax is that the compiler can signal an error when it detects a write to a read-only property, whereas at best it can only generate an undeclared method warning that you invoked a non-existent set*Property*: method, which will fail at runtime.
- There is one case where properties cannot be used:

```
id y; x = y.z; // z is an undeclared property
```

- If you want to access a property of self using accessor methods, you must explicitly call out self as illustrated in this example:

```
self.age = 10;
```

- If you do not use self., you access the instance variable directly.
- In the following example, the set accessor method for the age property is *not* invoked:

```
age = 10;
```

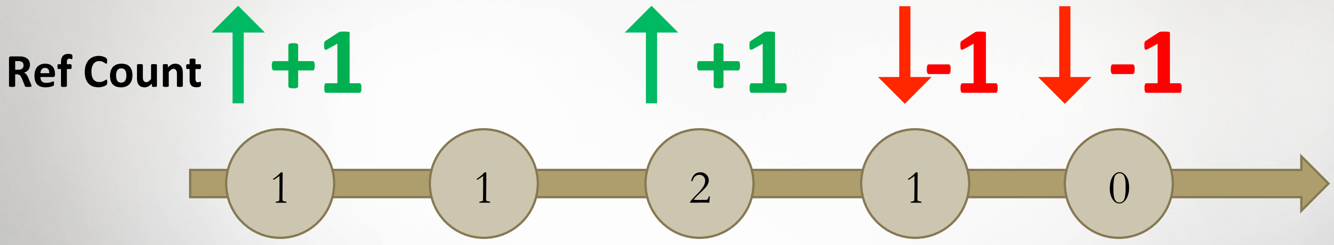
- If a nil value is encountered during property traversal, the result is the same as sending the equivalent message to nil.

MEMORY MANAGEMENT

❖ In Objective-C there are two methods for managing memory:

- Reference counting -- Manual–
  - depends on code added by the programmer
- Garbage collection. -- Automatic –
  - system automatically managing the memory. –
  - Not available on iPhone.

OBJECT LIFECYCLE



Op:	+ alloc	- init	- retain	- release	- release
main()	Create array	initialize			Release from use
my_func()			Retain for use	Release from use	

AUTORELEASE AND AUTORELEASE POOLS

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

- When the autorelease message is sent to an object, that object is then added to the inner most auto release pool
- When the pool is sent the drain (i.e. release) message, then all the objects sent the autorelease message are released
- Autorelease defers the release until later.
- You may nest as many autorelease pools as you need.
- The inner autorelease pool has absolutely no effect on the outer autorelease pool

RETAINCOUNT

- A method we can use to see how many references an object has.
- Can be used as follows:

```
NSLog(@"retainCount for person: %d", [person retainCount]);
```

- No need to pay too much attention to retainCount, best practice:
  - When you want an object, **retain (or alloc) it.**
  - When you are done with an object, **release it.**

WHEN IS AN OBJECT DESTROYED?

- *When it's retain count reaches 0,*
  then the deconstructor - *dealloc* is called
- *Never call dealloc yourself -- this is always called automatically for you.*
- *(Except when you're calling [super dealloc] from within your dealloc implementation)*

```
-(void)dealloc {
        [super dealloc];
        [name release];
         [address release];
}
```

PROTOCOLS

- A protocol declares methods that can be implemented by any class.
- Protocols are not classes themselves.
- They simply define an interface that other objects are responsible for implementing.
- When you implement the methods of a protocol in one of your classes, your class is said to conform to that protocol.

```
@protocol MyProtocol
                    - (void)myProtocolMethod;
@end
```

- protocols do not have a parent class
- they do not define instance variables

```
@interface MyClass : NSObject <UIApplicationDelegate, MyProtocol > {
     }
@end
```

- Protocol methods can be marked as optional using the @optional keyword. Or required using @required keyword to formally denote the semantics of the default behavior.
- The default is @required, if no keyword is specified.

CATEGORIES

- Allow us to add methods to an existing class, so that all instances of that class in the application gain the added functionality.
- They are different from subclassing
- Categories don't allow you to use instance variables.
- it is possible to overwrite a method already in place

CATEGORY SYNTAX

```
@interface ClassNameHere (category)
        // method declaration(s)

@end
```

The implementation looks like this;

```
@implementation ClassNameHere (category)

        // method implementation(s)

@end
```

File naming convention following the pattern of the name of the class we are adding a category to, a plus sign, and the name of our category.

CATEGORY EXAMPLE

```
@interface NSString (reverse)

        -(NSString *) reverseString;
@end
```

For the implementation:

```
@implementation NSString (reverse)

-(NSString *)reverseString {

}
@end
```

The two files created are named: *NSString+reverse.h* (interface)
And *NSString+reverse.m* (implementation).

EXTENSIONS

❖ Class extensions are like “anonymous” categories, except that the methods they declare must be implemented in the main `@implementation` block for the corresponding class.

```
@interface MyObject ()
- (void)setNumber:(NSNumber *)newNumber;
@end
```

CREATING A SINGLETON

```
static MySingleton* sharedSingleton = nil;
      + (MySingleton *) sharedSingleton {
                    if (sharedSingleton == nil) {
                              sharedSingleton=[[superallocWithZone:NULL]init];
                    }
          return sharedSingleton ;
          }
          + (id)allocWithZone:(NSZone *)zone { return [[self
sharedSingleton ] retain]; }
           - (id)copyWithZone:(NSZone *)zone { return self; }
          - (id)retain { return self; }
          - (NSUInteger)retainCount {
                    return NSUIntegerMax; //denotes an object that cannot be
released
          }
          - (void)release { //do nothing }
          - (id)autorelease { return self; }
```

FAST ENUMERATION

❖ Fast enumeration is a language feature that allows you to efficiently and safely enumerate over the contents of a collection using a concise syntax.

❖ The syntax is defined as follows:

for (*Type newVariable* in *expression*) { *statements* }

❖ or

Type existingItem;
for (*existingItem* in *expression*) { *statements* }
Ex:

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

SELECTORS

❖ Has two meanings:

- Used to refer to the name of a method when it's used in a source-code message to an object.
- Also used to refer to the unique identifier that replaces the name when the source code is compiled.
- Compiled selectors are of type SEL.
- All methods with the same name have the same selector.
- You can use a selector to invoke a method on an object
- this provides the basis for the implementation of the target-action design pattern in Cocoa.

- Instead of using full ASCII names , the compiler writes each method name into a table, then pairs the name with a unique identifier that represents the method at runtime.
- The runtime system makes sure each identifier is unique: No two selectors are the same, and all methods with the same name have the same selector.
- Compiled selectors are assigned to a special type, SEL, to distinguish them from other data

```
SEL setWidthHeight;
setWidthHeight = @selector(setWidth:height:);
```

- The performSelector:, performSelector:withObject:, and performSelector:withObject:withObject: methods, defined in the NSObject protocol, take SEL identifiers as their initial arguments
- setWidthHeight = NSSelectorFromString(aBuffer);

  NSString *method;
- method = NSStringFromSelector(setWidthHeight);

EXCEPTION HANDLING

- ❖Objective-C’s exception support revolves around four compiler directives: @try, @catch, @throw, and @finally:
- ❖Can re-throw the caught exception using the @throw directive without an argument inside a @catch() block
- ❖can throw any Objective-C object as an exception object.
- ❖The NSException class provides methods that help in exception processing, but it can be customized to implement your own if you so desire.

SYNCHRONIZING THREAD EXECUTION

- Objective-C supports multithreading in applications.
- Two threads can try to modify the same object at the same time, a situation that can cause serious problems in a program.
- To protect sections of code from being executed by more than one thread at a time, Objective-C provides the @synchronized() directive.

```
- (void)criticalMethod {
        @synchronized(self) {
                    // Critical code. ...
       }
   }
```

- The @synchronized() directive takes as its only argument any Objective-C object, including self
- This object is known as a *mutual exclusion* semaphore or *mutex*.

❖ The Objective-C synchronization feature supports recursive and reentrant code

❖ A thread can use a single semaphore several times in a recursive manner; other threads are blocked from using it until the thread releases all the locks obtained with it

❖ When code in an @synchronized() block throws an exception, the Objective-C runtime catches the exception, releases the semaphore (so that the protected code can be executed by other threads), and re-throws the exception to the next exception handler.

IMPORTANT FOUNDATION CLASSES

- **Strings**
  - The NSString class
- **Numbers and Dates**
  - Unlike standard C floats, integers, and so forth, these elements are all objects
- **Collections**
  - arrays, dictionaries, and sets.

- NSString *myString = @"A string constant";
- NSString *myString = [NSString stringWithFormat: @"The number is %d", 5];
- NSLog(@"%@", [myString stringByAppendingString:@"22"]);
- NSLog(@"%@", [myString stringByAppendingFormat:@"%d", 22]);
- NSLog(@"%d", myString.length); //length
- printf("%c", [myString characterAtIndex:2]);
- Convert to C-String:
    - printf("%s\n", [myString UTF8String]); printf("%s\n",
    - [myString cStringUsingEncoding: NSUTF8StringEncoding]);
- Convert to NSString:
    - [NSString stringWithCString:"Hello World" encoding: NSUTF8StringEncoding]

❖ Write a string to a file:

- [myString writeToFile:path atomically:YES encoding:NSUTF8StringEncoding error:&error]

❖ Reading a string from a file:

- NSString *inString = [NSString stringWithContentsOfFile:path encoding:NSUTF8StringEncoding error:&error];

❖ Split a String:

- NSString *myString = @"One Two Three Four Five Six Seven"; NSArray *wordArray = [myString componentsSeparatedByString: @" "];

❖ Substrings

- NSString *sub1 = [myString substringToIndex:7];
- NSString *sub2 = [myString substringFromIndex:4];

- Substring using a range

```
NSRange r;
r.location = 4;
r.length = 2;
 NSString *sub3 = [myString substringWithRange:r];
```

- search a string for a substring returns a range

```
NSRange searchRange = [myString rangeOfString:@"Five"];
if (searchRange.location != NSNotFound)
NSLog(@"Range location: %d, length: %d", searchRange.location,
    searchRange.length);
```

- replace a subrange with a new string
  - `NSString *replaced = [myString stringByReplacingOccurrencesOfString: @" " withString: @" * "];`

❖ Change string case

- [myString uppercaseString];
- [myString lowercaseString]);
- NSLog(@"%@",[myString   capitalizedString]);

❖ compare and test strings

- [s1 isEqualToString:s2] , [s1 hasPrefix:@"Hello"]
- [s1 hasSuffix:@"Hello"]

❖ Convert strings into numbers by using a value method

- [s1 intValue], [s1 boolValue]); NSLog(@"%f", [s1 floatValue]); NSLog(@"%f", [s1 doubleValue]);

❖ Mutable string

- NSMutableString *myString = [NSMutableString stringWithString: @"Hello World. "];
- [myString appendFormat:@"The results are %@ now.", @"in"];

NUMBERS AND DATES

- NSNumber class
  - NSNumber *number = [NSNumber numberWithFloat:3.141592];
  - NSLog(@"%d", [number intValue]); NSLog(@"%@", [number stringValue]);
- Also:
  - numberWithInt, numberWithFloat:, numberWithBool

WORKING WITH DATES

- NSDate objects - use number of seconds since an epoch (midnight on January 1, 2001.)(Unix epoch is midnight on January 1, 1970).
- Current time
  - NSDate *date = [NSDate date]; /
- Time relative to current (10 sec from now)
  - date = [NSDate dateWithTimeIntervalSinceNow:10.0f];
- Show Date
  - NSLog(@"%@" [date description]);
  - Use NSDateFormatter class

TIMERS

- ❖ NSTimer class
  - [NSTimer scheduledTimerWithTimeInterval: 1.0f target: self selector: @selector(handleTimer:) userInfo: nil repeats: YES];
- ❖ Disable timer
  - send it the invalidate message
    - [timer invalidate];

INDEX PATHS

- NSIndexPath Class
- Stores the section and row number for a user selection in tables
- indexPath.row and indexPath.section properties

ARRAYS

- NSArray, NSMutableArray classes that hold any type of object
  - NSArray *array = [NSArray arrayWithObjects:@"One", @"Two", @"Three", nil];
- array.count, [array objectAtIndex:0], arrayWithArray: addObject: removeObjectAtIndex:, arrayWithArray: arrayByAddingObjectsFromArray: containsObject:,
- Convert an array to string
  - [array componentsJoinedByString:@" "]);

DICTIONARIES

- NSDictionary, NSMutableDictionary, classes
- Create a dictionary

```
NSMutableDictionary *dict = [NSMutableDictionary dictionary];
 [dict setObject:@"1" forKey:@"A"];
```

- Search a dictionary

```
[dict objectForKey:@"A“];
```

- Remove objects

```
[dict removeObjectForKey:@"B"];
```

- List keys
  - [dict allKeys];

SET OBJECTS

- NSSet class
- Access set objects:
    - [aSet allObjects];

- ❖ Arrays, sets and dictionaries automatically retain objects when they are added
- ❖ And release those objects when they are removed from the collection.
- ❖ Releases are also sent when the collection is deallocated.
- ❖ Collections do not copy objects.
- ❖ They rely on retain counts to hold onto objects and use them as needed.

- Write collections to files (NSArray, NSDictionary)
  - *writeToFile: atomically:*
  - Objects must be of type: NSData, NSDate, NSNumber, NSString, NSArray, and NSDictionary
- To recover an array or dictionary from file
  - *NSArray *newArray = [NSArray arrayWithContentsOfFile:path];*
  - And *dictionaryWithContentsOfFile:*
- NSURL objects point to resources.
  - These resources can refer to both local files and to URLs on the Web.
  - NSURL *url1 = [NSURL fileURLWithPath:path]; for a file
    - NSString *path = [NSHomeDirectory() stringByAppendingPathComponent:@"Documents/foo.txt"];
  - NSURL *url2 = [NSURL URLWithString:urlpath]; for the web
    - NSString *urlpath = @"http://neu.edu";

- ❖ NSData objects correspond to buffers.
- ❖ NSData provides data objects that store and manage bytes.
  - *NSData *data = [NSData dataWithContentsOfURL:url2];*