## Programming in

## Objective-C

Contents:

1. Introduction to Mac OS 3
2. [MacBook](http://www.apple.com/macbook/) Built-in Apps 5
3. Introduction to Objective-C 10
4. Objective-C Variables 11
5. Objective C DATA Types 14
6. Control Statements 17
7. Interface 22
8. Implementation 23
9. Constructor Methods 24
10. Classes 26
11. Method overriding 31
12. Method overloading 33
13. The id Type 34
14. Objective C NSArray 35
15. Objective C NSMutableArray 39
16. NSDictionary 42
17. NSString 50
18. NSMutableString 55
19. NSNumber
20. Protocols 58
21. Selector 60
22. Properties 62
23. The copy Attribute 64
24. Memory Management in Obj-C 66

**Introduction to Mac OS**

* **Mac OS** is a series of [graphical user interface](https://en.wikipedia.org/wiki/Graphical_user_interface)–based [operating systems](https://en.wikipedia.org/wiki/Operating_system) developed by [Apple Inc.](https://en.wikipedia.org/wiki/Apple_Inc.) for their [Macintosh](https://en.wikipedia.org/wiki/Macintosh) line of [computer systems](https://en.wikipedia.org/wiki/Computer_system).
* The [original operating system](https://en.wikipedia.org/wiki/System_1) was first introduced in 1984 as being integral to the [original Macintosh](https://en.wikipedia.org/wiki/Macintosh_128K), and referred to as the "System".
* Referred to by its major revision starting with "System 6" and "System 7", Apple rebranded version 7.6 as "Mac OS" as part of their [Macintosh clone](https://en.wikipedia.org/wiki/Macintosh_clone) program in 1996.
* The Macintosh, specifically its system software, is credited with having popularized the early [graphical user interface](https://en.wikipedia.org/wiki/Graphical_user_interface) concept.
* Mac OS is the computer [operating system](http://searchcio-midmarket.techtarget.com/definition/operating-system) for Apple Computer's [Macintosh](http://searchcio-midmarket.techtarget.com/definition/Macintosh) line of personal computers and workstations.
* A popular feature of its latest version, Mac [OS X](http://whatis.techtarget.com/definition/OS-X) , is a [desktop](http://searchwinit.techtarget.com/definition/desktop) interface with some [3-D](http://whatis.techtarget.com/definition/3-D-three-dimensions-or-three-dimensional) appearance characteristics.
* OS X has a modular design intended to make it easier to add new features to the operating system in the future. It runs [UNIX](http://searchenterpriselinux.techtarget.com/definition/Unix) applications as well as older Mac applications.
* Mac OS comes with Apple Computer's [iMac](http://whatis.techtarget.com/definition/iMac) and Power Macintosh line of computers.

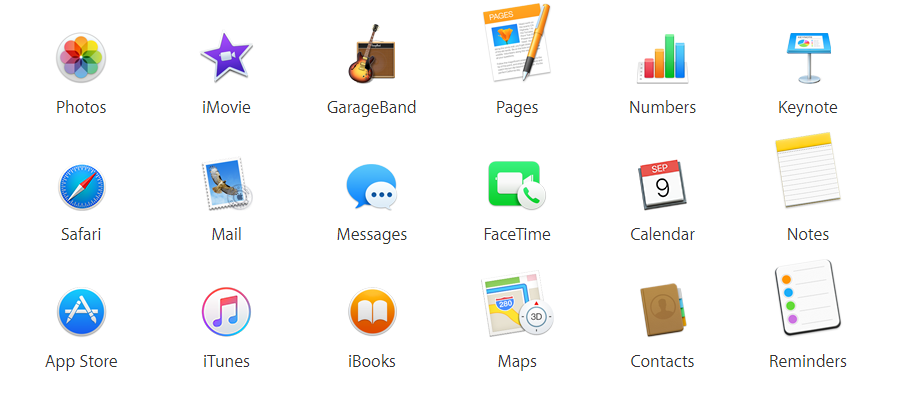


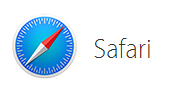
|  |  |
| --- | --- |
| [**Developer**](https://en.wikipedia.org/wiki/Software_developer) | [Apple](https://en.wikipedia.org/wiki/Apple_Inc.), [NeXT](https://en.wikipedia.org/wiki/NeXT) |
| **OS family** | [Classic Mac OS](https://en.wikipedia.org/wiki/Classic_Mac_OS) (System 1–7, Mac OS 8–9) [Unix](https://en.wikipedia.org/wiki/Unix), Mach and [FreeBSD](https://en.wikipedia.org/wiki/FreeBSD) ([OS X](https://en.wikipedia.org/wiki/OS_X))[[1]](https://en.wikipedia.org/wiki/Mac_OS#cite_note-leopard_unix_cert-1)[[2]](https://en.wikipedia.org/wiki/Mac_OS#cite_note-2)[[3]](https://en.wikipedia.org/wiki/Mac_OS#cite_note-3) |
| **Working state** | Publicly released |
| **Source model** | **Classic Mac OS:** [Closed source](https://en.wikipedia.org/wiki/Closed_source) **OS X:** [Closed source](https://en.wikipedia.org/wiki/Closed_source) (with [open-source](https://en.wikipedia.org/wiki/Open-source) components) |
| **Initial release** | **Classic Mac OS:** January 24, 1984; 32 years ago **OS X:** September 13, 2000; 15 years ago (as Mac OS X) |
| [**Latest release**](https://en.wikipedia.org/wiki/Software_release_life_cycle) | 10.11.3 (Build 15D21) / January 19, 2016; 23 days ago |
| [**Latest preview**](https://en.wikipedia.org/wiki/Software_release_life_cycle) | 10.11.4 Beta (Build 15E27e) / January 11, 2016; 31 days ago |
| [**Kernel**](https://en.wikipedia.org/wiki/Kernel_(computing))**type** | **Classic Mac OS:** [Monolithic](https://en.wikipedia.org/wiki/Monolithic_kernel) **OS X:** [Hybrid](https://en.wikipedia.org/wiki/Hybrid_kernel) ([XNU](https://en.wikipedia.org/wiki/XNU)) |
| [**License**](https://en.wikipedia.org/wiki/Software_license) | [Proprietary](https://en.wikipedia.org/wiki/Proprietary_software) |
| **Official website** | [apple.com/osx/](http://apple.com/osx/) |

[MacBook](http://www.apple.com/macbook/) Built-in Apps

# One of the best things about a Mac is everything you can do with it.

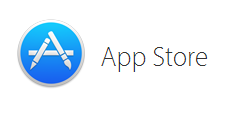
* Every Mac comes with a collection of great apps for things you do every day, like Safari for surﬁng the web, Photos for managing your photos and videos, Pages for creating documents, Numbers for making spreadsheets, and Keynote for preparing presentations.
* There are apps for sending email and text messages, taking notes, and staying up to date with your contacts and calendar. It even comes with the Mac App Store for finding new apps. Your Mac is more than fully featured, it’s fully loaded.





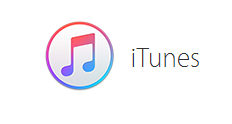
Safari is the best way to see the sites.

* Safari is the best, fastest, and most secure way to surf the web on your Mac, with all kinds of innovative features that make your browsing more enjoyable.
* Keep favorite websites open and accessible with Pinned Sites.
* Quickly mute audio without hunting for the tab it’s coming from.
* Use AirPlay to stream video from a web page to your HDTV with Apple TV.
* It’s also easy to share the cool pages you find.
* Just click the Share button to tweet, post to Facebook, or send to a friend.
* Energy-saving technologies let you surf longer. And built-in privacy features protect you from being tracked as you go.



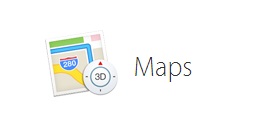
## Get apps for your Mac from your Mac.

* The Mac App Store lets you browse, purchase, and install apps, widgets, and extensions on your Mac.
* New apps install in one step to Launchpad.
* If you get a new Mac, you can quickly re download all the apps you own.
* The Mac App Store can automatically update your apps and even OS X for you, so you always have the latest versions.
* You can browse Mac apps by category, such as games, productivity, music, and more. Or do a quick search for something specific.
* Read descriptions and customer reviews.
* Flip through screenshots.
* When you find an app you like, click to buy it.
* The Mac App Store has apps for just about everything and everyone.



## Your music, movies, and TV shows take center stage.

* iTunes is home to a universe of entertainment right on Mac.
* It’s the best way to organize and enjoy the music, movies, and TV shows you already have, and shop for the content you want to get.
* Beats 1 — is free Apple Music worldwide radio station — for the latest music, interviews, and culture, 24/7.
* With an Apple Music membership, you can tune in to on-demand music stations with handpicked songs and artists based on what you listen to and like, and explore artists, albums, and genres you never knew you’d love.
* And membership gives you unlimited access to the more than 30 million songs in the Apple Music library.



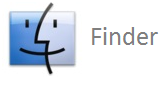
## A destination unto itself.

* By using Maps you will get the whole world on your desktop.
* Get information on local points of interest like restaurants and hotels, with phone numbers, photos, and Yelp reviews.
* You can look up transit, driving, or walking directions on your Mac and send them to your iPhone for voice navigation on the way.
* Navigating with Maps is incredibly smooth and responsive, and you get gorgeous views using Flyover, a photo-realistic, interactive 3D experience that lets you soar high above select cities.



## The last word in word processing.

* It is used make beautiful documents in Pages.
* Paint a picture with words. Or add a picture, movie, shape, or gorgeous chart.
* Tools for whatever you’re doing appear right when you need them in the context-sensitive format panel and when you’re ready to share your Pages document, you can simply send a link that anyone can open from any modern web browser on a Mac or PC.
* Writing has never been so effortless.



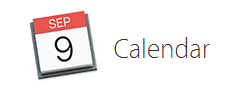
# The Finder organizes all of our files

* The Finder in OS X provides access to the files, folders, and drives on your Mac, and helps you keep them organized.
* The Finder is the app that helps you navigate all of the files and folders on Mac.
* The Finder lets you browse your apps, disks, files, and folders in a variety of ways.
* You can use the Finder to organize these items the way you want.
* You can also use the Finder to search for items, delete files you no longer want, and more.

## C:\Users\rnadaf\Desktop\Maps.jpg Spotlight

# Spotlight helps you find what you‘re looking for

* Spotlight helps you quickly locate files on your Mac, and more.
* Use Spotlight to search for things like apps, documents, images and other files.
* In OS X Yosemite, Spotlight suggestions offer additional results like Wikipedia, news sites, Maps, iTunes, movie listings, and more.
* To open Spotlight, click the magnifying glass icon in the upper-right corner of the menu bar, or press Command-Space from any app.
* Spotlight appears front and center when you open it.
* To search for something, type words related to it like an app name, a word contained in a document, or [a tag](http://support.apple.com/kb/HT202754) you've attached to a file.
* You can also search for items like a Wikipedia article, or the location of your nearest Apple Store.
* As you type, Spotlight auto completes the search field with suggestions. You don't have to finish typing to open a suggested item, just type the first few letters and press Return. It's a convenient way to quickly open your favorite apps.



## An app you’ll appreciate day after day

* Keep track of our busy schedule and share it with others using Calendar.
* Create separate calendars — one for home, another for school, a third for work, and so on. See all your calendars in a single window or choose to see only the calendars you want.
* The inspector lets you add a location to your event, suggests locations as you type, provides maps and weather forecasts, and automatically alerts you when it’s time to leave.
* And when you use iCloud, every Calendar change you make on your Mac automatically appears on your iOS devices and Apple Watch.3 So no dinner dates are missed, no meeting goes unattended, and no anniversary is forgotten.

Introduction to Objective-C

* Objective-C is the general purpose programming language that is used for developing Apple’s iOS and OS X operating systems and reusable Libraries.
* Objective-C is developed on top of C Programming language by adding features of Small Talk programming language making it an object-oriented language.
* Initially, Objective-C was developed by NeXT for its NeXTSTEP OS from whom it was taken over by Apple for its iOS and Mac OS X.
* The below snippet shows an example for iOS programming.

Example: #import <Foundation/Foundation.h>

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

{

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

NSLog (@"Hello world");

**Foundation Framework**

Foundation Framework provides large set of features and they are listed below.

* It includes a list of extended data types like NSArray, NSDictionary, NSSet and so on.
* It consists of a rich set of functions manipulating files, strings, etc.
* It provides features for URL handling, utilities like date formatting, data handling, error handling, etc.

Objective-C Variables:

* Variable is nothing but a name given to a storage area that our programs can manipulate.
* Each variable in Objective-C has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.
* The name of a variable can be composed of letters, digits, and the underscore character.
* It must begin with either a letter or an underscore.
* Upper and lowercase letters are distinct because Objective-C is case sensitive.
* The following are basic variable types:

|  |  |
| --- | --- |
| **Type** | **Description** |
| char | Typically a single octet (one byte). This is an integer type. |
| int | The most natural size of integer for the machine. |
| float | A single-precision floating point value. |
| double | A double-precision floating point value |
| void | Represents the absence of type. |

* Objective-C programming language also allows defining various other types of variables, like Enumeration, Pointer, Array, Structure, Union, etc.

**Variable Definition in Objective-C:**

* A variable definition means to tell the compiler where and how much to create the storage for the variable.
* A variable definition specifies a data type and contains a list of one or more variables of that type as follows:

**Syntax**: type variable list;

* Here, type must be a valid Objective-C data type including char, int, float, double, bool or any userdefined object, etc., and variable list may consist of one or more identifier names separated by commas.
* Some valid declarations are shown below:

int i, j, k;

char c, ch;

float f, salary;

double d;

* The line int i, j, k; both declares and defines the variables i, j and k; which instructs the compiler to create variables named i, j and k of type int.
* Variables can be initialized (assigned an initial value) in their declaration. The initializer consists of an equal sign followed by a constant expression as follows:

**Syntax**: type variable name = value;

**Examples:**

//defining & initializing Boolean type data type

BOOL checkBooleanOrNot = YES;

//defing and initializing the char val of letter m

char definingCharacter = 'm';

//Defining & initializing Single pricision val to variable

float aFloatingPointVal = 73.956;

//Double pricession floating point

double storingDoubleVal = -21.847;

//Defining & initializing an intiger val to anIntegerTypeOfVal variable

int anIntegerTypeOfVal = -1234678353;

//defining & storing an long type of val

long longTypeOfVal = -46476432131709874;

//Defing a short type of val & storing it into storingShortTypeVal Variable

short storingShortTypeVal = -1453;

Objective C DATA TYPES:

* In the Objective-C programming language, data types refer to declaring variables or functions of different types.
* The type of a variable determines how much space it occupies in storage and how the bit pattern stored is interpreted.

The types in Objective-C can be classified as follows:

**Integer Types**

Following table gives the details about standard integer types with its storage sizes and value ranges:

|  |  |  |
| --- | --- | --- |
| **Type** | **Storage size** | **Value range** |
| char | 1 byte | -128 to 127 or 0 to 255 |
| unsigned char | 1 byte | 0 to 255 |
| signed char | 1 byte | -128 to 127 |
| Int | 4 bytes | 32,768 to 32,767 or -2,147,483,648 to 2,147,483,647 |
| unsigned int | 4 bytes | 0 to 65,535 or 0 to 4,294,967,295 |
| short | 2 bytes | -32,768 to 32,767 |
| unsigned short | 2 bytes | 0 to 65,535 |
| long | 4 bytes | -2,147,483,648 to 2,147,483,647 |
| unsigned long | 4 bytes | 0 to 4,294,967,295 |
| long long | 8 bytes | A double long |
| BOOL | 1 byte | Boolean (signed char YES or NO) |

**Floating-Point Types:**

Following table gives you details about standard float-point types with storage sizes and value ranges and their precision:

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Storage size** | **Value range** | **Precision** |
| float | 4 byte | 1.2E-38 to 3.4E+38 | 6 decimal places |
| double | 8 byte | 2.3E-308 to 1.7E+308 | 15 decimal places |
| long double | 10 byte | 3.4E-4932 to 1.1E+4932 | 19 decimal places |

**Basic Data Types**

Some of the more common data types we use in Objective-C include:

* **int** – An integer value, i.e. a whole number (no decimals) that includes zero and negative numbers.
* **float** – A floating point value that includes as many decimal places as it can hold. Because the decimal place can change, or float, it’s important to know that these values may technically be imprecise. When precise decimals are needed, like for currency, we should use the NSDecimalNumber data type.
* **BOOL** – Short for “boolean”, this is a 1-bit “true” or “false” value that can only be in one of those states. The C language (and hence, Objective-C) treat 0 as “false” and 1 as “true”. As such, the following keywords can be used to represent true/false values: YES/NO, TRUE/FALSE, true/false, 1,0.
* **char** – A single character, such as the letter A or the symbol “#”. Note that lowercase and uppercase characters are different, so “a” and “A” are two different characters.
* **NSString** – String data is a bunch of characters strung together to make text, like a banner strung up at a party.
* **NSNumber** – This class is a lightweight “wrapper” class that gives object-oriented features to the primitive number types mentioned above (among others).

**Defining Constants in Objective-C**

There are the following two simple ways in Objective-C to define constants:

* Using #define preprocessor
* Using const keyword

**Objective-C The #define Preprocessor**

Following is the syntax to use #defines preprocessor to define a constant:

Syntax:

#define identifier value

Let's look at the following example:

Objective-C Example Source Code:

#import <Foundation/Foundation.h>

#define LENGTH 5

#define WIDTH 10

#define NEWLINE '\n'

int main()

{

int area;

area = LENGTH \* WIDTH;

NSLog(@"value of area : %d", area);

NSLog(@"%c", NEWLINE);

return 0;

}

Control Statements:

* 1. **if / else :**

“if statement” is used to perform conditional checks.

**a.**

if (condition)

{

Statement 1

}

Statement X;

In the 1st scenario first condition will be checked. If the condition is true the statement 1 will be executed and then control will be transferred to statement X. If condition is false then control will be transferred to statement X without executing statement **1**.

**b.**

if (condition)

{

Statement 1

}

else

{

Statement 2

}

Statement X;

In the 2nd scenario if the condition is true then statement 1 will be executed and the control will be transferred to statement X. If the condition is false then statement 2 will be executed and control will be transferred to statement X. Here either if or else condition will execute and both of them will never execute together.

**c.**

if (condition)

{

Statement 1

}

else if (condition)

{

Statement 2

}

else

{

Statement 3

}

Statement X;

In the 3rd scenario if the condition is true then statement 1 will be executed and then control will be transferred to statement X. If the condition is false it will come to else if part and checks the condition. If true then Statement 2 will be executed and control will be transferred to statement X. If false then statement 3 will be executed and control will be transferred to statement X.

Ex: //Defining & initializing personAge variable to 20

int personAge = 20;

//Checking weather personAge variable contains less than 10 value

if(personAge < 10)

{

NSLog(@"It's a kids Age");

}

//Checking weather personAge variable contains less than or equal to 20 value

else if(personAge <= 20)

{

NSLog(@"It's students age");

}

//Checking weather personAge variable contains greather than or equal to 25 value

else if(personAge >= 25)

{

NSLog(@"It's Workers Age");

}

//If personAge variable's value not satisfyies the above conditions

//It prints this message

else

{

NSLog(@"There is nothing special about this age");

}

* 1. **for**
* for statement is used to execute a set of statements multiple times.

1 2 4

for (initialization;condition;increment/decrement)

{ 3

Statement 1;

}

* It is alternative to while loop. In while loop initialization, condition, statements and increment/decrement are not defined in single statement.
* When for statement is encountered 1st initialization statement will be executed and then condition will be verified. If condition is true then statements inside for block will be executed and then increment or decrement operation will be executed. After increment or decrement operation again condition will be verified. If the condition is true then again for block will be executed and same process will be repeated till condition becomes false. If the condition is false then control will come out the loop and executes statement X
  1. **switch / case :**
* Switch statement is used to execute a particular set of statements among multiple conditions.
* It is alternative to complicated if else if ladder conditions.
* The expression type in the switch must be any of the following data type.
* break is optional. Every case must end with break statement which will help to terminate and transfer the control outside of switch block. If no break is used then execution will continue to next case.
* default is optional. If present it will execute only if the value of the expression does not match with any of the case and then control goes to statement X. If not present then control will exit switch statement and goes to statement X.
  1. **while and do/while :**

**while:**

* while is an entry controlled loop. In while statement 1st condition will be verified. If the condition is true loop will be repeated .If the condition is false then control will come out of the loop.

**do/while:**

* do/while is exit controlled loop. In do while first all the statements inside the block will be executed and then condition will be verified. If the condition is true then loop will be repeated. If condition is false then control will come out of the loop and execute statement X.
* In while when the condition is false for the first time then the statement inside the block will be executed for 0 times, whereas in do while statement will be surely executed for one time.
  1. **break :**
* It helps to transfer control to another part of the program.
* It can be used in switch statement or in do, while and for loops. It also can be used in labeled blocks.
* It has the following uses.
  + It helps in terminating a switch statement.
  + It is used to exit a loop or force immediate termination of a loop bypassing the conditional expression and other remaining code in the loop.
  1. **continue :**
* It is used to move the control to the beginning of the loop.
* When continue is used in while or do-while loop then control is directly transferred to the test condition that controls the loop.
* When continue is used in for loop control is transferred to iteration portion first then to the test condition.

**Note:** break is used to move the control to the end of the loop but continue is used to move the control to the beginning of the loop.

* 1. **return :**
* return keyword terminates the execution in a method and returns the control to the caller method.

Interface

* The interface of a class is usually defined in a header file. A common convention is to name the header file after the name of the class. E.g. Ball.h would contain the interface for the class Ball.

An interface ClassName : Super Class Name

{

//instance variable

}

+ Class Method1;

+ (return\_type) Class Method2;

+ (return\_type) ClassMethod3 : (param1\_type)param1\_var name;

- (return\_type) instanceMethod1 : (param1\_type)param1\_varName:(param2\_type)

Param2\_var name;

- (return\_type)instanceMethod2 With Parameter:(param1\_type)param1\_var name and other Parameter : (parameter2\_type)param2\_var\_name;

@end

* ‘+’ sign denotes class methods or methods that can be called without an instance of class.
* ‘-’sign denotes instance methods, which can only be called within a particular instance of the class.
* Class methods also have no access to instance variables.

Implementation

* The interface only declares the class interface and not the method themselves the actual code is written in the implementation file
* Implementation (method) files normally have the file extension .m, which originally signified “messages”.

@implementation ClassName

+ Class Method

{

//implementation

}

* Instance Method

{

//implementation

}

“Constructor” Methods

* There are no constructor methods in Objective-C.
* In Objective C, an object is **initialized** by calling the init method immediately after it’s allocated.
* This is why instantiation is always a two-step process: allocate, and then initialize.
* There is also a class-level initialization method in Objective-C.
* In Objective-C init is the default initialization method.
* In Objective-C you can also define your own versions of Constructors to accept configuration parameters.
* There’s nothing special about custom initialization methods—they’re just normal instance methods, except the method name should always begin with init.

**Example:**

// Car.h

- (id)initWithModel:(NSString \*)aModel;

* To implement this method, you should follow the canonical initialization pattern shown in initWithModel: below.

**Example:**

// Car.m

- (id)initWithModel:(NSString \*)aModel {

self = [super init];

if (self) {

// Any custom setup work goes here

\_model = [aModel copy];

\_odometer = 0;

}

return self;

}

* The super keyword refers to the parent class, and again, the self keyword refers to the instance calling the method.
* Initialization methods should always return a reference to the object itself, and if it cannot be initialized, it should return nil.
* This is why we need to check if self exists before trying to use it

Classes

* As in many other object-oriented programming languages, Objective-C classes provide the blueprint for creating objects.
* First, you define a reusable set of properties and behaviors inside of a class then, you instantiate objects from that class to interact with those properties and behaviors.
* Objective-C is abstracts a class’s interface from its implementation.
* An **interface** declares the public properties and methods of a class, and the corresponding **implementation** defines the code that actually makes these properties and methods work.
* In object-oriented programming terms, an object is an instance of a class.
* A class is used to specify the form of an object and it combines data representation and methods for manipulating that data into one neat package.
* The data and methods within a class are called members of the class.

## Objective-C characteristics:

* The class is defined in two different sections namely **@interface** and **@implementation**.
* Almost everything is in form of objects.
* Objects receive messages and objects are often referred as receivers.
* Objects contain instance variables.
* Objects and instance variables have scope.
* Classes hide an object's implementation.
* Properties are used to provide access to class instance variables in other classes.

## Interfaces:

* An interface is created with the @interface directive, after which come the class and the super class name, separated by a colon.
* Protected variables can be defined inside of the curly braces, but most developers treat instance variables as implementation details and prefer to store them in the .m file instead of the interface.
* The @property directive declares a public property, and the (copy) attribute defines its memory management behavior.
* In this case, the value assigned to model will be stored as a copy instead of a direct pointer.
* The - (void) call line declares a method called drive that takes no parameters, and the (void) portion defines its return type.
* The minus sign indicates the method marks it as an instance method (opposed to a [class method](http://rypress.com/tutorials/objective-c/classes#class-methods-and-variables)).

**Example:**

// Mobile.h

#import <Foundation/Foundation.h>

@interface Mobile : NSObject {

// protected instance variables (not recommended)

}

@property (copy) NSString \*model;

- (void)call;

@end

Mobile.h contains some template code; this declares a property called model and a method called drive.

## Implementations

* The first thing any class implementation needs to do is import its corresponding interface.
* The @implementation directive is similar to @interface, except you don’t need to include the super class.
* Private instance variables can be stored between curly braces after the class name:

**Example:**

// Mobile.m

#import "Mobile.h"

@implementation Mobile {

// Private instance variables

double \_callTimer;

}

@synthesize model = \_model; // Optional for Xcode 4.4+

- (void) call {

NSLog(@"Calling to a number using %@ model", self.model);

}

@end

* @synthesize is a convenience directive that automatically generates accessor methods for the property.
* By default, the getter is simply the property name (model), and the setter is the capitalized name with the set prefix (setModel).
* This is much easier than manually creating accessor for every property.
* The \_model portion of the synthesize statement defines the private instance variable name to use for the property.

## Instantiation and Usage:

* A class provides the blueprints for objects, so basically an object is created from a class.
* Any files that need access to a class must import its header file (Mobile.h)—they should never, ever try to access the implementation file directly.

**Example:**

// main.m

#import <Foundation/Foundation.h>

#import "Mobile.h"

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

@autoreleasepool {

Mobile \*motorola = [[Mobile alloc] init];

[motorola setModel:@"motorola Corolla"];

NSLog(@"Created a %@", [motorola model]);

motorola.model = @"motorola Camry";

NSLog(@"Changed the Mobile to a %@", motorola.model);

[motorola call];

}

return 0;

}

* After the interface has been imported with the #import directive, you can instantiate objects with the alloc/init pattern shown above.
* As you can see, instantiation is a two-step process: first you must allocate some memory for the object by calling the alloc method, then you need to initialize it so it’s ready to use.
* You should never use an uninitialized object.
* To call a method on an Objective-C object, you place the instance and the method in square brackets, separated by a space.
* Arguments are passed after the method name, preceded by a colon.

**Example:**

motorola.setModel("3rd Gen");

## Class Methods and Variables:

* In Objective C it’s also possible to define class-level Methods and Variables.
* These are commonly called “static” methods/properties in other programming languages.
* Class method declarations look just like instance methods, except they are prefixed with a plus sign instead of a minus sign.

**Example:**

Let’s add the following class-level method to Mobile.h:

// Mobile.h

+ (void)setDefaultModel:(NSString \*)aModel;

* Similarly, a class method implementation is also preceded by a plus sign.

**Example:**

// Mobile.m

#import "Mobile.h"

static NSString \*\_defaultModel;

@implementation Mobile {

...

+ (void)setDefaultModel:(NSString \*)aModel {

\_defaultModel = [aModel copy];

}

@end

* The [aModel copy] call creates a copy of the parameter instead of assigning it directly.
* Class methods use the same square-bracket syntax as instance methods, but they must be called directly on the class, as shown below.

**Example:**

// Mobile.m

[Car setDefaultModel:@"Motorola 2nd Gen"];

Method overriding

* Method overriding is a language feature in which a class can provide an implementation of a method that is already provided by one of its parent classes.
* The implementation in this sub-class replaces (that is, overrides) the implementation in the parent class.
* When you define a method with the same name as that of a parent class, that new method replaces the inherited definition
* The new method must have the same return type and take the same number and type of parameters as the method you are overriding. Here’s an example:

**Example:**

@interface MyClass : NSObject {

}

- (int)myNumber;

@end

@implementation MyClass : NSObject {

}

- (int)myNumber {

return 1;

}

@end

@interface MySubclass : MyClass {

}

- (int)myNumber;

@end

@implementation MySubclass

- (int)myNumber {

return 2;

}

@end

* If you create an instance of MyClass and send it a myNumber message, it returns 1. If you create an instance of MySubclass and send it a myNumber message, it returns 2.
* The subclass’s method must have the same name and parameter list as the superclass's overridden method.
* In addition to completely replacing an existing implementation, you might want to extend a superclass’s implementation. To do this, you can invoke the superclass’s implementation using the superkeyword.

**Example:**

@implementation MySubclass

- (int)myNumber {

int subclassNumber = [super myNumber] + 1;

return subclassNumber;

}

@end

* Within a method definition, super refers to the parent class (the superclass) of the current object.
* You send a message to super to execute the superclass’s implementation of a method.
* In the above example, the implementation of myNumber by MySubclass simply adds 1 to whatever value is returned by the implementation of MyClass.

Method overloading

* Method overloading is not possible in Objective-C.
* In objective C you cannot overload the method by using different arguments as shown in example below.
* When you overload the method in interface class it will give the compile time error.

**Example:**

-(int) addTwoNumber : (int)firstNumber : (int) secondNumber; //Line number 1

-(float) addTwoNumber : (float)firstNumber : (float) secondNumber; //Line Number 2 **error.**

* In the above example at second line it gives an error of declaration of duplicate methods.
* But you can declare a method like as give below example.

**Example:**

-(int) addTwoNumber : (int)firstNumber : (int) secondNumber; //Line number 1

-(int) addTwoNumber : (int)firstNumber : (int) secondNumber : (int) thirdNumber; //Line number 2

* In the above example 1 , when you pass the parameter as,

[FirtsClass addTwoNumber : 2 : 3]; it will execute the first method from Line number 1.

* In the above example 2 , when you pass the parameter as,

[FirtsClass addTwoNumber : 2 : 3 : 7]; it will execute the second method from Line number 2.

The id Type

* The id type is the generic type for all Objective-C objects.
* We can think of it as the object-oriented version of C’s void pointer.
* And, like a void pointer, it can store a reference to any type of object.
* The following example uses the same id variable to hold a string and a dictionary.

**Example:**

id mysteryObject = @"An NSString object";

NSLog(@"%@", [mysteryObject description]);

mysteryObject = @{@"model": @"Ford", @"year": @1967};

NSLog(@"%@", [mysteryObject description]);

* id is a generic type. This means that the compiler will expect any **data** type there, and will not enforce restrictions.
* It can be useful if you're expecting to use more than one class of objects there.
* You can then use introspection to find out which class it is.
* id automatically assumes a pointer, as all objects in Objective-C are passed as pointers/references
* That is, id can be used for storing objects that belong to any class.

Objective C NSArray

* Data storage and its retrieval is one the most important in any program.
* NSArray is Objective-C’s general-purpose array type.
* **NSArray** and its subclass [**NSMutableArray**](https://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/Classes/NSMutableArray_Class/index.html#//apple_ref/occ/cl/NSMutableArray) manage ordered collections of objects called **arrays**.
* NSArray creates static arrays, and NSMutableArray creates dynamic arrays. You can use arrays when you need an ordered collection of objects
* The NSArray class is used for managing an ordered collection of objects.
* An ordered collection of objects is a grouping of objects that is expected to be maintained in the order in which they were stored.
* Typically, an ordered collection of objects is accessed either through enumeration or by index.
* NSArray is used to hold an immutable array of objects.
* Mutability helps to change the array in runtime a pre allocated array.
* If we use NSArray, we only replace the existing array and cannot change the contents of the existing array.
* NSArray is immutable, so you cannot dynamically add or remove items.
* The NSArray class is immutable — once it is created, you cannot modify its contents.

However, because Objective-C provides no mechanism for ensuring the immutability of the objects inside an array, if you access an element of an array, those objects can be modified.

* NSArray as well as all collections in Objective-C is zero based. This means the first element starts at index 0, and the last element has an index of one less than the length of the array.
* If you try to access an element outside of these index boundaries, you will get an exception.

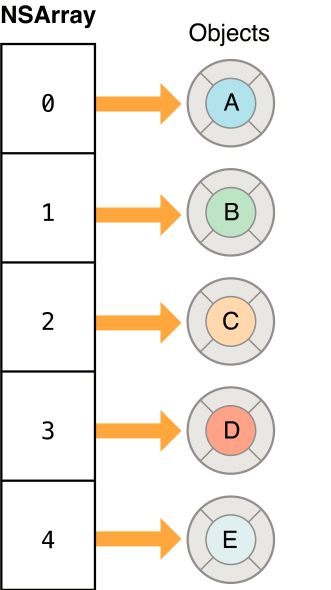


Fig a): NSArray collection classes of the Foundation Framework.

Courtesy: Apple Documentation

**Creating Arrays:**

Immutable arrays can be defined as literals using the @[] syntax.

**Example:**

NSArray \*germanMakes = @[@"Mercedes-Benz", @"BMW", @"Porsche",@"Opel", @"Volkswagen", @"Audi"];

NSArray \*ukMakes = [NSArray arrayWithObjects:@"Aston Martin",@"Lotus", @"Jaguar", @"Bentley", nil];

NSLog(@"First german make: %@", germanMakes[0]);

NSLog(@"First U.K. make: %@", [ukMakes objectAtIndex:0]);

## Enumerating Arrays:

Fast-enumeration is the most efficient way to iterate over an NSArray, and its contents are guaranteed to appear in the correct order. It’s also possible to use the count method with a traditional for-loop to step through each element in the array:

**Example:**

NSArray \*germanMakes = @[@"Mercedes-Benz", @"BMW", @"Porsche", @"Opel", @"Volkswagen", @"Audi"];

**// With fast-enumeration**

for (NSString \*item in germanMakes) {

NSLog(@"%@", item);

}

**// With a traditional for loop**

for (int i=0; i<[germanMakes count]; i++) {

NSLog(@"%d: %@", i, germanMakes[i]);

}

**Important methods of NSArray are as follows**

* alloc/initWithObjects: Used to initialize an array with objects.
* objectAtIndex: Returns the object at specific index.
* count: Returns the number of objects
* lastObject: which returns the last element of the array. To find the index of a specific element, you can also use the method
* isEqual: message to each of the elements of the array, and returns the first element which returns YES.

## Comparing Arrays:

Arrays can be compared for equality with the aptly namedisEqualToArray: method, which returns YES when both arrays have the same number of elements and every pair pass an isEqual:comparison. NSArray does not offer the same subset and intersection comparisons as NSSet.

NSArray \*germanMakes = @[@"Mercedes-Benz", @"BMW", @"Porsche", @"Opel", @"Volkswagen", @"Audi"];

NSArray \*sameGermanMakes = [NSArray arrayWithObjects:@"Mercedes-Benz" @"BMW", @"Porsche", @"Opel” @"Volkswagen", @"Audi", nil];

if ([germanMakes isEqualToArray:sameGermanMakes]) {

NSLog(@"Oh good, literal arrays are the same as NSArrays");

}

**Inherits From:**

[NSObject](https://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/Classes/NSObject_Class/index.html#//apple_ref/occ/cl/NSObject)

NSArray

[NSMutableArray](https://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/Classes/NSMutableArray_Class/index.html#//apple_ref/occ/cl/NSMutableArray)

**Import Statement:**

* @import Foundation;

**Availability:**

* Available in OS X v10.0 and later

**Advantages:**

* It represents an ordered collection of objects, and it provides a high-level interface for sorting and otherwise manipulating lists of data.

**Disadvantage:**

* The NSArray class is immutable — once it is created, you cannot modify its contents.

Objective C NSMutableArray

* NSMutableArray is Objective-C’s general-purpose array type.
* NSMutableArray is subclass of NSArray.
* NSMutableArray is inherited from NSArray and hence all instance methods of NSArray is available in NSMutableArray.
* NSMutableArray creates dynamic arrays.
* The NSMutableArray class lets you dynamically add or remove items from arbitrary locations in the collection.
* Mutability helps to change the array in runtime a pre allocated array.
* If we use NSMutableArray, we can change the contents of the existing array.
* NSMutableArray are often used to represent the state of a system, but the fact that NSMutableArray records the order of its elements opens up new modeling opportunities.
* NSMutableArray can record the order in which they should be fixed.

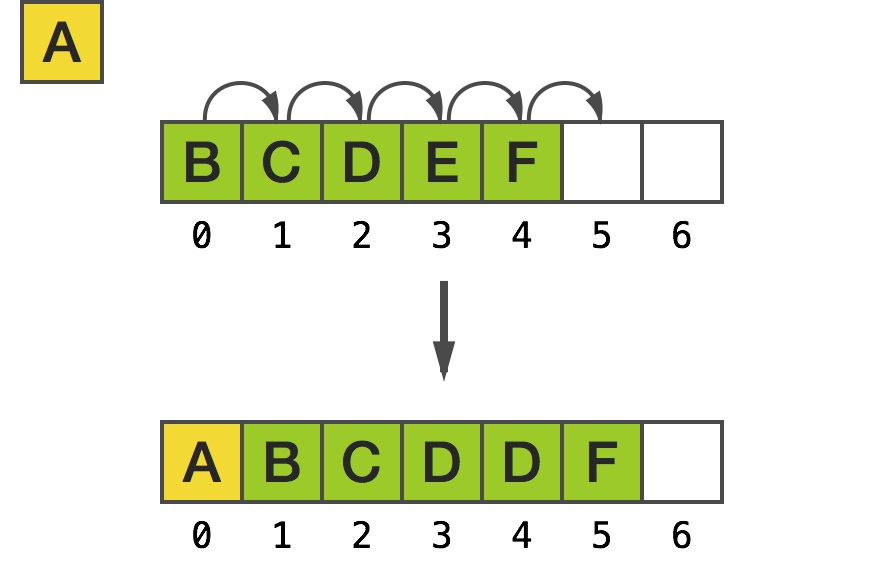


Fig a): NSMutableArray  collection classes of the Foundation Framework.

**Important methods of NSMutableArray are as follows:**

* removeAllObjects: Empties the array.
* addObject: Inserts a given object at the end of the array.
* removeObjectAtIndex: This is used to remove objectAt a specific index
* exchangeObjectAtIndex:withObjectAtIndex: Exchanges the objects in the array at given indices.
* replaceObjectAtIndex:withObject: Replaces the object at index with an Object.

## Creating Mutable Arrays:

**Example:**

NSMutableArray \*brokenCars = [NSMutableArray arrayWithObjects:@"Audi A6", @"BMW Z3", @"Audi Quattro", @"Audi TT", nil];

## Adding and Removing Objects:

The two basic methods for manipulating the contents of an array are the addObject: and removeLastObject methods. The former adds an object to the end of the array, and the latter is pretty self-documenting. Note that these are also useful methods for treating an NSArray as a stack.

**Example:**

NSMutableArray \*brokenCars = [NSMutableArray arrayWithObjects: @"Audi A6", @"BMW Z3", @"Audi Quattro", @"Audi TT", nil];

[brokenCars addObject:@"BMW F25"];

NSLog(@"%@", brokenCars); // BMW F25 added to end

[brokenCars removeLastObject];

NSLog(@"%@", brokenCars); // BMW F25 removed from end

**Inherits From:**

[NSObject](https://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/Classes/NSObject_Class/index.html#//apple_ref/occ/cl/NSObject)

NSArray

[NSMutableArray](https://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/Classes/NSMutableArray_Class/index.html#//apple_ref/occ/cl/NSMutableArray)

**Import Statement:**

* @import Foundation;

**Availability:**

* Available in OS X v10.0 and later

**Advantages of NSMutableArray over NSArray:**

* Changing how NSMutableArray stores the elements of its collection. You might do this for performance reasons or for better compatibility with legacy code.
* Acquiring more information about what is happening to the collection (for example, statistics gathering).

**Disadvantages:**

* In a subclass, you must override all the methods. You must also override the primitive methods of the [NSArray](https://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/Classes/NSArray_Class/index.html#//apple_ref/occ/cl/NSArray) class.

NSDictionary

* The NSDictionary class declares the programmatic interface to objects that manage immutable associations of keys and values.
* NSDictionary is used to hold an immutable dictionary of objects.
* NSDictionary is immutable.
* NSDictionary creates static dictionaries.
* The NSDictionary class represents an unordered collection of objects.
* The term **dictionary** refers to any instance of one of these classes without specifying its exact class membership.
* A key-value pair within a dictionary is called an entry.
* Each entry consists of one object that represents the key and a second object that is that key’s value.
* Within a dictionary, the keys are unique.
* No two keys in a single dictionary are equal (as determined by [isEqual:](https://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/Protocols/NSObject_Protocol/index.html#//apple_ref/occ/intfm/NSObject/isEqual:)).
* In general, a key can be any object, but note that when using key-value coding the key must be a string.
* Neither a key nor a value can be nil; if you need to represent a null value in a dictionary, you should use [NSNull](https://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/Classes/NSNull_Class/index.html#//apple_ref/occ/cl/NSNull).

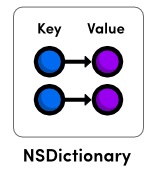


Fig a): NSDictionary collection classes of the Foundation Framework.

Courtesy: <http://rypress.com/>

**Important methods of NSDictionary are as follows**

* alloc/initWithObjectsAndKeys: Initializes a newly allocated dictionary with entries constructed from the specified set of values and keys.
* valueForKey: Returns the value associated with a given key.
* count: Returns the number of entries in the dictionary.

## Creating Dictionaries

Immutable dictionaries can be defined using the literal @{} syntax.

**Example:**

// Literal syntax

NSDictionary \*inventory = @{ @"Mercedes-Benz SLK250" : [NSNumber numberWithInt:13],@"Mercedes-Benz E350" : [NSNumber numberWithInt:22],@"BMW M3 Coupe" : [NSNumber numberWithInt:19], @"BMW X6" : [NSNumber numberWithInt:16],

};

// Values and keys as arguments

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];

// Values and keys as arrays

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 forKeys:models];

NSLog(@"%@", inventory);

## Accessing Values and Keys:

You can use the same subscripting syntax as arrays (someDict[key]) to access the value for a particular key. The objectForKey: method is the other common way to access values.

NSDictionary \*inventory = @{

@"Mercedes-Benz SLK250" : [NSNumber numberWithInt:13],

@"Mercedes-Benz E350" : [NSNumber numberWithInt:22],

@"BMW M3 Coupe" : [NSNumber numberWithInt:19],

@"BMW X6" : [NSNumber numberWithInt:16],

};

NSLog(@"There are %@ X6's in stock", inventory[@"BMW X6"]);

NSLog(@"There are %@ E350's in stock",

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

## Enumerating Dictionaries:

As with sets and arrays, fast-enumeration is the most efficient way to enumerate a dictionary, and it loops through the keys (not the values).NSDictionary also defines a count method, which returns the number of entries in the collection.

NSDictionary \*inventory = @{

@"Mercedes-Benz SLK250" : [NSNumber numberWithInt:13],

@"Mercedes-Benz E350" : [NSNumber numberWithInt:22],

@"BMW M3 Coupe" : [NSNumber numberWithInt:19],

@"BMW X6" : [NSNumber numberWithInt:16],

};

NSLog(@"We currently have %ld models available", [inventory count]);

for (id key in inventory) {

NSLog(@"There are %@ %@'s in stock", inventory[key], key);

}

**Inherits From:**

[NSObject](https://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/Classes/NSObject_Class/index.html#//apple_ref/occ/cl/NSObject)

NSDictionary

[NSMutableDictionary](https://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/Classes/NSMutableDictionary_Class/index.html#//apple_ref/occ/cl/NSMutableDictionary)

**Import Statement:**

* @import Foundation;

**Availability:**

* Available in OS X v10.0 and later

**Advantages:**

* It stores two objects one is key and another one is value.
* A dictionary is also a form of hash table.

**Disadvantage:**

* It can store neither a key nor a value can be nil.

NSMutableDictionary

* The NSMutableDictionary class declares the programmatic interface to objects that manage mutable associations of keys and values.
* It is subclass of NSDictionary.
* It adds modification operations to the basic operations it inherits from [NSDictionary](https://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/Classes/NSDictionary_Class/index.html#//apple_ref/occ/cl/NSDictionary).
* [NSMutableDictionary](http://rypress.com/tutorials/objective-c/data-types/nsdictionary#nsmutabledictionary) data structure lets you dynamically add and remove entries as necessary.
* The NSMutableDictionary class lets you add new key-value pairs dynamically.
* Mutable collections in general lend themselves to representing system states, and mutable dictionaries are no different.
* NSMutableDictionary is inherited from NSDictionary and hence all instance methods of NSDictionary are available in NSMutableDictionary.

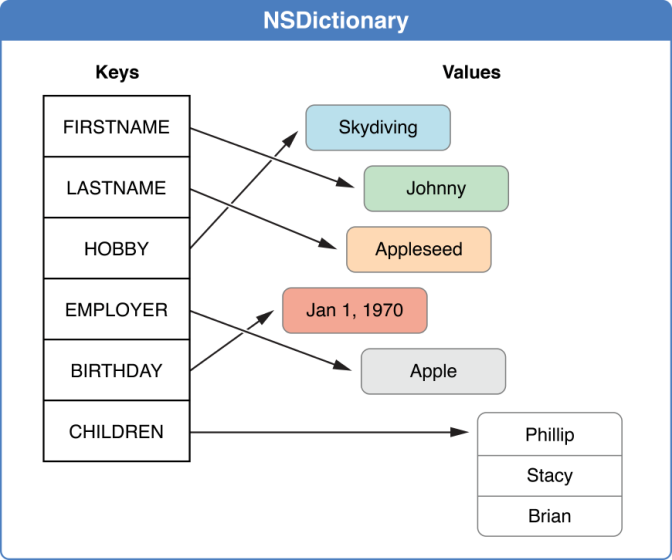


Fig a): NSDictionary collection classes of the Foundation Framework.

Courtesy: Apple Documentation

**Important methods of NSMutableDictionary are as follows:**

* removeAllObjects: Empties the dictionary of its entries.
* removeObjectForKey: Removes a given key and its associated value from the dictionary.
* setValue:forKey: Adds a given key-value pair to the dictionary.

## Creating Mutable Dictionaries:

Mutable dictionaries can be created by calling any of the factory methods defined by NSDictionary on the NSMutableDictionary class.

NSMutableDictionary \*jobs = [NSMutableDictionary

dictionaryWithDictionary:@{

@"Audi TT" : @"John",

@"Audi Quattro (Black)" : @"Mary",

@"Audi Quattro (Silver)" : @"Bill",

@"Audi A7" : @"Bill"

}];

NSLog(@"%@", jobs);

## Adding and Removing Entries:

The setObject:forKey: and removeObjectForKey: methods are the significant additions contributed by NSMutableDictionary.

NSMutableDictionary \*jobs = [NSMutableDictionary

dictionaryWithDictionary:@{

@"Audi TT" : @"John",

@"Audi Quattro (Black)" : @"Mary",

@"Audi Quattro (Silver)" : @"Bill",

@"Audi A7" : @"Bill"

}];

// Transfer an existing job to Mary

[jobs setObject:@"Mary" forKey:@"Audi TT"];

// Finish a job

[jobs removeObjectForKey:@"Audi A7"];

// Add a new job

jobs[@"Audi R8 GT"] = @"Jack";

## Enumeration Considerations:

* Dictionaries should not be mutated while you’re iterating over them.
* dictionaryWithDictionary: class method to create a shallow copy of the entire dictionary.

**Inherits From:**

[NSObject](https://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/Classes/NSObject_Class/index.html#//apple_ref/occ/cl/NSObject)

NSDictionary

[NSMutableDictionary](https://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/Classes/NSMutableDictionary_Class/index.html#//apple_ref/occ/cl/NSMutableDictionary)

**Import Statement:**

* @import Foundation;

**Availability:**

* Available in OS X v10.0 and later

**Advantages:**

* It stores two objects one is key and another one is value dynamically.
* It is better choice than mutable arrays.
* A dictionary is also a form of hash table.

**Disadvantage:**

* In a subclass, you must override both of its primitive methods:
* setObject:forKey
* removeObjectForKey
* You must also override the primitive methods of the [NSDictionary](https://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/Classes/NSDictionary_Class/index.html#//apple_ref/occ/cl/NSDictionary) class.

NSString

* The NSString classes represent text strings and provide methods for searching, combining, and comparing strings.
* NSString is one the most commonly used classes that is used for storing strings and texts.
* The NSString class and its mutable subclass, [NSMutableString](https://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/Classes/NSMutableString_Class/index.html#//apple_ref/occ/cl/NSMutableString), provide an extensive set of APIs for working with strings.
* The [NSString](https://developer.apple.com/library/mac/#documentation/Cocoa/Reference/Foundation/Classes/NSString_Class/Reference/NSString.html) class is the basic tool for representing text in an Objective-C application.
* NSString is an immutable type, so you cannot change it after it’s been instantiated.
* The string in Objective-C programming language is represented using NSString and its subclass NSMutableString provides several ways for creating string objects.
* An NSString object encodes a Unicode-compliant text string, represented as a sequence of UTF–16 code units.
* NSString objects are used extensively throughout Foundation and other Cocoa frameworks, serving as the basis for all textual and linguistic functionality on the platform
* The simplest way to create a string object is to use the Objective-C @"..." construct:
* An NSString object encodes a Unicode-compliant text string, represented as a sequence of UTF–16 code units
* All lengths, character indexes, and ranges are expressed in terms of 16-bit platform-Endean values, with index values starting at 0.
* An immutable string is a text string that is defined when it is created and subsequently cannot be changed. An immutable string is implemented as an array of Unicode characters (in other words, a text string).
* To create and manage an immutable string, use the NSString class.
* The objects you create using NSString and NSMutableString are referred to as string objects
* The term C string refers to the standard char \* type.
* string object’s class is private, its interface is public, as declared by these abstract super classes, NSString and NSMutableString.
* The string classes adopt the [NSCopying](https://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/Protocols/NSCopying_Protocol/index.html#//apple_ref/occ/intf/NSCopying) and [NSMutableCopying](https://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/Protocols/NSMutableCopying_Protocol/index.html#//apple_ref/occ/intf/NSMutableCopying) protocols, making it convenient to convert a string of one type to the other .

**Objective-C supports a wide range of methods for manipulate strings:**

|  |  |
| --- | --- |
| **S.N.** | **Method & Purpose** |
| 1 | **- (NSString \*)capitalizedString;**  Returns a capitalized representation of the receiver. |
| 2 | **- (unichar)characterAtIndex:(NSUInteger)index;**  Returns the character at a given array position. |
| 3 | **- (double)doubleValue;**  Returns the floating-point value of the receiver’s text as a double. |
| 4 | **- (BOOL)hasPrefix:(NSString \*)aString;**  Returns a Boolean value that indicates whether a given string matches the beginning characters of the receiver. |
| 5 | **- (BOOL)hasSuffix:(NSString \*)aString;**  Returns a Boolean value that indicates whether a given string matches the ending characters of the receiver. |
| 6 | **- (id)initWithFormat:(NSString \*)format ...;**  Returns an NSString object initialized by using a given format string as a template into which the remaining argument values are substituted. |
| 7 | **- (NSInteger)integerValue;**  Returns the NSInteger value of the receiver’s text. |
| 8 | **- (BOOL)isEqualToString:(NSString \*)aString;**  Returns a Boolean value that indicates whether a given string is equal to the receiver using a literal Unicode-based comparison. |
| 9 | **- (NSUInteger)length;**  Returns the number of Unicode characters in the receiver. |
| 10 | **- (NSString \*)lowercaseString;**  Returns lowercased representation of the receiver. |
| 11 | **- (NSRange)rangeOfString:(NSString \*)aString;**  Finds and returns the range of the first occurrence of a given string within the receiver. |
| 12 | **- (NSString \*)stringByAppendingFormat:(NSString \*)format ...;**  Returns a string made by appending to the receiver a string constructed from a given format string and the following arguments. |
| 13 | **- (NSString \*)stringByTrimmingCharactersInSet:(NSCharacterSet \*)set;**  Returns a new string made by removing from both ends of the receiver characters contained in a given character set. |
| 14 | **- (NSString \*)substringFromIndex:(NSUInteger)anIndex;**  Returns a new string containing the characters of the receiver from the one at a given index to the end. |

## Creating Strings:

The most common way to create strings is using the literal @"Some String" syntax, but the stringWithFormat: class method is also useful for generating strings that are composed of variable values. It takes the same kind of format string as NSLog():

NSString \*make = @"Porsche";

NSString \*model = @"911";

int year = 1968;

NSString \*message = [NSString stringWithFormat:@"That's a %@ %@ from %d!",

make, model, year];

NSLog(@"%@", message);

Notice that we used the @"%@" format specifies in the NSLog() call instead of passing the string directly with NSLog(message).

## Enumerating Strings:

The two most basic NSString methods are length andcharacterAtIndex:, which return the number of characters in the string and the character at a given index, respectively

NSString \*make = @"Porsche";

for (int i=0; i<[make length]; i++) {

unichar letter = [make characterAtIndex:i];

NSLog(@"%d: %hu", i, letter);

}

## Comparing Strings:

String comparisons present the same issues as NSNumber comparisons. Instead of comparing pointers with the == operator, you should always use the isEqualToString: method for a more robust valuecomparison. The following example shows you how this works, along with the useful hasPrefix: and hasSuffix: methods for partial comparisons.

NSString \*car = @"Porsche Boxster";

if ([car isEqualToString:@"Porsche Boxster"]) {

NSLog(@"That car is a Porsche Boxster");

}

if ([car hasPrefix:@"Porsche"]) {

NSLog(@"That car is a Porsche of some sort");

}

if ([car hasSuffix:@"Carrera"]) {

// this won't execute

NSLog(@"That car is a Carrera");

}

**Inherits From:**

[NSObject](https://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/Classes/NSObject_Class/index.html#//apple_ref/occ/cl/NSObject)

NSString

[NSMutableString](https://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/Classes/NSMutableString_Class/index.html#//apple_ref/occ/cl/NSMutableString)

**Import Statement:**

* @import Foundation;

**Availability:**

* Available in OS X v10.0 and later

**Advantages:**

* It stores any type of character.
* It has many built in methods.

**Disadvantage:**

* It is immutable.

NSNumber

* The NSNumber Class in Objective-C programming language provides the ability to save the basic data types like int, float, and Boolean in the form of object.
* The NSNumber class is a lightweight, object-oriented wrapper around C’s numeric primitives.
* We convert primitive type to Object because to get built in functionality and when we want to store values in NSArray and Dictionary it mandatory to convert primitive data type to object type.
* Like NSArray and other Classes like NSDictionary these classes will not know how to interact with primitive type data types so these classes must want the data in the form of objects.
* NSNumber allows us the use of Selectors.
* It’s main job is to store and retrieve primitive values, and it comes with dedicated methods for each data type:

**Example:**

NSNumber \*aBoolValue = [NSNumber numberWithBool:NO];

NSNumber \*aCharValue= [NSNumber numberWithChar:'z'];

NSNumber \*aUCharValue = [NSNumber numberWithUnsignedChar:255];

NSNumber \*aShortValue = [NSNumber numberWithShort:32767];

NSNumber \*aUShortValue = [NSNumber numberWithUnsignedShort:65535];

NSNumber \*anIntValue= [NSNumber numberWithInt:2147483647];

NSNumber \*aUIntValue = [NSNumber numberWithUnsignedInt:4294967295];

NSNumber \*aLongValue = [NSNumber numberWithLong:9223372036854775807];

NSNumber \*aULongValue = [NSNumber numberWithUnsignedLong:18446744073709551615];

NSNumber \*aFloatValue = [NSNumber numberWithFloat:26.99f];

NSNumber \*aDoubleValue = [NSNumber numberWithDouble:26.99];

NSLog(@"%@", [aBool boolValue] ? @"YES" : @"NO");

NSLog(@"%c", [aChar charValue]);

NSLog(@"%hhu", [aUChar unsignedCharValue]);

NSLog(@"%hi", [aShort shortValue]);

NSLog(@"%hu", [aUShort unsignedShortValue]);

NSLog(@"%i", [anInt intValue]);

NSLog(@"%u", [aUInt unsignedIntValue]);

NSLog(@"%li", [aLong longValue]);

NSLog(@"%lu", [aULong unsignedLongValue]);

NSLog(@"%f", [aFloat floatValue]);

NSLog(@"%f", [aDouble doubleValue]);

* NSNumber provide a straight forward approach to convert primitive type to String type

**Example:**

NSNumber \*aUCharValue = [NSNumber numberWithUnsignedChar:255];

NSNumber \*anIntValue = [NSNumber numberWithInt:2147483647];

NSNumber \*aFloatValue = [NSNumber numberWithFloat:26.99f];

NSLog(@"%@", aUCharValue);

NSLog(@"%@", anIntValue);

NSLog(@"%@", aFloatValue);

Numeric Literals

* In Xcode 4.4 introduced numeric literals, which offer a much more convenient alternative to the above methods.
* The NSNumberversion of BOOL’s, char’s, int’s and double’s can all be created by simply prefixing the corresponding primitive type with the @ symbol and for unsigned int’s, long’s, and float’s must be appended with the U, L, or F modifiers, as shown below.

**Example:**

NSNumber \*aBool = @NO;

NSNumber \*aCharValue= @'z';

NSNumber \*anIntValue = @2147483647;

NSNumber \*aUIntValue = @4294967295U;

NSNumber \*aLongValue = @9223372036854775807L;

NSNumber \*aFloatValue = @26.99F;

NSNumber \*aDoubleValue = @26.99;

Immutability

* NSNumber is immutable i.e once we created NSNumber objects it’s not possible to change its value.
* The NSNumber instance acts exactly like a primitive value when we need a new double value, we need to create a new literal and it’s not possible to change an existing one
* From a practical standpoint, the above sentence means we need to create a new NSNumber object every time when we want to change its value.

**Example:**

NSNumber \*counter = @0;

for (int i=0; i<10; i++) {

counter = @([counter intValue] + 1);

NSLog(@"%@", counter);

}

Comparing Numbers

* By using NSNumber Class it’s possible to directly compare NSNumber pointers by using the method isEqualToNumber this is the much more robust way to check for equality.
* The isEqualToNumber method guarantees that two values will compare equal, even if they are stored in different objects.

**Example:**

NSNumber \*anInt = @27;

NSNumber \*sameInt = @27U;

// Pointer comparison (fails)

if (anInt == sameInt) {

NSLog(@"They are the same object");

}

// Value comparison (succeeds)

if ([anInt isEqualToNumber:sameInt]) {

NSLog(@"They are the same value");

}

**If we need to check for inequalities, we can use the related** **compare methods described below**

|  |  |
| --- | --- |
| **Return Value** | **Description** |
| NSOrderedAscending | receiver < argument |
| NSOrderedSame | receiver == argument |
| NSOrderedDescending | receiver > argument |

**Example:**

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");

NSMutableString

* The [NSMutableString](https://developer.apple.com/library/mac/#documentation/Cocoa/Reference/Foundation/Classes/NSMutableString_Class/Reference/Reference.html) class is a mutable version of NSString.
* Unlike immutable strings, it’s possible to alter individual characters of a mutable string without creating a brand new object.
* The NSString class and its mutable subclass, [NSMutableString](https://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/Classes/NSMutableString_Class/index.html#//apple_ref/occ/cl/NSMutableString), provide an extensive set of APIs for working with strings, including methods for comparing, searching, and modifying strings.
* This is preferred data structure when you’re performing several small edits on the same string.
* NSMutableString inherits from NSString, so aside from the ability to manipulate it in place, you can use a mutable string just like you would an immutable string.
* NSMutableString inherits from NSString, so aside from the ability to manipulate it in place, you can use a mutable string just like you would an immutable string.
* The fundamental workflow for mutable strings is different than that of immutable ones.
* Instead of creating a new object and replacing the old value, NSMutableString methods operate directly on the existing instance.
* Over distributed-object connections, mutable string objects are passed by-reference and immutable string objects are passed by-copy
* To construct and manage a string that can be changed after it has been created, use [NSMutableString](https://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/Classes/NSMutableString_Class/index.html#//apple_ref/occ/cl/NSMutableString).

## Creating Mutable Strings:

Mutable strings can be created through the stringWithString: class method, which turns a literal string or an existing NSString object into a mutable one:

NSMutableString \*car = [NSMutableString stringWithString:@"Porsche 911"];

After you’ve created a mutable string, the setString: method lets you assign a new value to the instance:

[car setString:@"Porsche Boxster"];

Compare this to NSString, where you re-assign a new value to the variable. With mutable strings, we don’t change the instance reference, but rather manipulate its contents through the mutable API.

## Replacing/Deleting Substrings

It’s possible to replace or delete substrings via thereplaceCharactersInRange:withString: anddeleteCharactersInRange: methods, as shown below.

NSMutableString \*car = [NSMutableString stringWithCapacity:20];

[car setString:@"Lotus Elise"];

[car replaceCharactersInRange:NSMakeRange(6, 5)

withString:@"Exige"];

NSLog(@"%@", car); // Lotus Exige

[car deleteCharactersInRange:NSMakeRange(5, 6)];

NSLog(@"%@", car);

## When to Use Mutable Strings:

* A good rule of thumb is to use a mutable string whenever you’re running any kind of algorithm that edits or assembles a string in several passes and to use an immutable string for everything else.
* This also applies to [sets](http://rypress.com/tutorials/objective-c/data-types/nsset.html), [arrays](http://rypress.com/tutorials/objective-c/data-types/nsarray.html), and [dictionaries](http://rypress.com/tutorials/objective-c/data-types/nsdictionary.html).

**Inherits From:**

[NSObject](https://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/Classes/NSObject_Class/index.html#//apple_ref/occ/cl/NSObject)

NSString

[NSMutableString](https://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/Classes/NSMutableString_Class/index.html#//apple_ref/occ/cl/NSMutableString)

**Import Statement:**

* @import Foundation;

**Availability:**

* Available in OS X v10.0 and later

**Advantages:**

* It stores any type of character.
* It has many built in methods.

**Disadvantage:**

* Any subclass of NSString must override the primitive instance methods [length](https://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/Classes/NSString_Class/index.html#//apple_ref/occ/instp/NSString/length) and [characterAtIndex:](https://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/Classes/NSString_Class/index.html#//apple_ref/occ/instm/NSString/characterAtIndex:)

Protocols

**Protocols:**

* Objective C was extended at NEXT to introduce the concept of multiple inheritances of specification, not implementation, through the introduction of protocols.
* This is a pattern achievable either as an abstract multiply-inherited base class in C++ or as an interface (as in java and C#).
* Objective C makes use of ad-hoc protocols, called informal protocols and compiler enforced protocols, called formal protocols.
* Informal protocol is a list of methods, which a class can opt to implement. It's specified in the documentation, since it has no presence in language.
* Informal protocols often include optional methods, where implementing the method can change the behavior of class.
* For Ex: a text field class might have a delegate which should implement an informal protocol with an optional auto complete method. The text field discovers whether the delegate implements that method (via reflection) and if so, calls it to support auto complete.
* Formal protocol is similar to an interface in java or C#. It's a list of methods, which any class can declare itself to implement.
* Versions of Objective C before 2.0 required that a class must implement all methods in a protocol, it declares itself as adopting, the compiler will send an error if the class does not implement every method of its declared protocols.
* Objective C 2.0 added support for making certain methods in a protocol optional, and the compiler will not enforce implementation of optional methods.
* The Objective C concept of protocols is different from the java or C# concept of interfaces in which a class may implement that protocol without being declared to implement that protocol.
* The difference is not detectable from outside code.
* Formal protocols cannot provide any implementation; they simply assure caller that classes which conform to the protocol will provide implementation.

**Defining a protocol**

@protocol MyProtocolName <NSObject>

//Methods go here

@ends

Replace "MyProtocolName" with name of your choice .There are no curly Braces.

That is because variables go in curly braces, and protocols have Variables associated with them "<NSObject>" means that the amount of protocol is derivation of the NSObject Protocol.

There are Both NSObject class and NSObject Protocol. Pointed brackets are associated with protocols.

**using the Protocol:**

In Java we specify that a class implements an interface with the "implements" keyword. In Objective C we use pointy brackets in the interface declaration.

(In Objective C "interface” means part of the class in the header file or "interface"

In java also same), following the class you extend.

E.x: we usually declare class like this

@interface CustomView:UIView

To specify that it implements a protocol, simply change it to this:

@ interface CustomView:UIView <MyProtocol Name>.

**Protocol as variables**:

Here is where it differs from java the most. In java when declaring a variable, you would use an interface name just you would a class.

In Objective -C you declare a variable this way:

id<My Protocol Name> myNewVariable

So the new type is "id<MyProtocolName>", id is the generic object even though it’s a pointer to an object, it does not have an asterisks it's assumed.

You can also use the notation when defining methods

Ex:

-(void)dosomethingWithThisObject :(id<My Protocol Name>)

Selector

* Selectors are Objective-C’s internal representation of a method name.
* A selector is the name used to select a method to execute for an object, or the unique identifier that replaces the name when the source code is compiled.
* A selector by itself doesn’t do anything.
* It simply identifies a method.
* The only thing that makes the selector method name different from a plain string is that the compiler makes sure that selectors are unique
* What makes a selector useful is that (in conjunction with the runtime) it acts like a dynamic function pointer that, for a given name, automatically points to the implementation of a method appropriate for whichever class it’s used with
* Suppose you had a selector for the method run, and classes Dog, Athlete, and ComputerSimulation (each of which implemented a method run). The selector could be used with an instance of each of the classes to invoke its run method—even though the implementation might be different for each.

**Note:** You use this technique in special situations, such as when you implement an object that uses the target-action design pattern. Normally, you simply invoke the method directly.

**Getting a Selector**

* There are two ways to get the selector for a method name.
* The @selector()directive lets you convert a source-code method name to a selector, and the NSSelectorFromString() function lets you convert a string to a selector (the latter is not as efficient).
* Both of these return a special data type for selectors called SEL.
* You can use SEL the exact same way as BOOL, int, or any other data type.

Compiled selectors are of type SEL. There are two common ways to get a selector:

At compile time, you use the compiler directive @selector.

**SEL aSelector = @selector(methodName);**

At runtime, you use the NSSelectorFromString function, where the string is the name of the method:

**SEL aSelector = NSSelectorFromString(@"methodName");**

You use a selector created from a string when you want your code to send a message whose name you may not know until runtime.

**Examples:**

//Using Function NSSelectorFromString

SEL stepOne = NSSelectorFromString(@"startEngine");

//Using directive @selector

SEL stepTwo = @selector(driveForDistance:);

//Using directive @selector

SEL stepThree = @selector(turnByAngle:quickly:);a

Properties

* Property is a robust way to handle Object’s data.
* We need to define a property if you want single piece of data (or a class object member) to be visible to other classes.

**Definition of simple Property**

**@**interface SimpleProperties : UIViewController

{

int count;

}

@property (read write) int count;

@end

In the class interface we need two things

1. Define a class variable as we normally do.
2. Define a property using the directive @property.

* In the above example ‘readwrite’ is property attribute.
* Property attribute you define decides how the property behaves.

After you define the property on the class interface, you need to also define the property implementation.

@implementation SimpleProperties

@synthesize count

@end

@synthesize will create automatically setter & getter methods for the property.

**Property Declarations:**

* Atomic: Means blocking access to accessors a single access both.

If two threads try to modify the value of property, their access to property is not so simultaneous. Using this behavior helps to avoid potential problems (Ex Values not expected to pass coming two setters).

On the other hand non atomic accessor render much factor, but make no warranty as to the simultaneous access of threads to access property.

Both are similar and use multi-threading .In this case non has been selected for factor access and atomic for safer and robust access.

Non atomic is not thread safe and multi-tasking is allowed.

**Strong:** Class/Object values.

Reference count of object is always increasing i.e. memory is modified.

**Weak:** Primitive Values.

Reference count of object is same i.e. memory remains unchanged.

**Copy:** Duplicates values at initialization.

**Assign:** Primitive types.

In .h file

@property (non atomic, Strong )NSString \*str;

In .m file

@implemetation

@synthesize str; // Compile time feature generate setter/getter methods.

The copy Attribute

* The copy attribute is an alternative to strong.
* Instead of taking ownership of the existing object, it creates a copy of whatever you assign to the property, then takes ownership of that
* Only objects that conform to the [NSCopying protocol](https://developer.apple.com/library/ios/documentation/cocoa/Reference/Foundation/Protocols/NSCopying_Protocol/Reference/Reference.html#//apple_ref/occ/intf/NSCopying) can use this attribute.
* Properties that represent values (opposed to connections or relationships) are good candidates for copying. For example, developers usually copy NSString properties instead of strongly reference them:

// Car.h

@property (nonatomic, copy) NSString \*model;

* Now, Car will store a brand new instance of whatever value we assign to model.
* If you’re working with mutable values, this has the added perk of freezing the object at whatever value it had when it was assigned. This is demonstrated below:

// main.m

#import <Foundation/Foundation.h>

#import "Car.h"

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

@autoreleasepool {

Car \*honda = [[Car alloc] init];

NSMutableString \*model = [NSMutableString stringWithString:@"Honda Civic"];

honda.model = model;

NSLog(@"%@", honda.model);

[model setString:@"Nissa Versa"];

NSLog(@"%@", honda.model); // Still "Honda Civic"

}

return 0;

}

Summary

The goal of all these attributes is to help us to focus on what data needs to be recorded by letting the compiler automatically determine how it’s represented. They are summarized below.

| **Attribute** | **Description** |
| --- | --- |
| getter= | Use a custom name for the getter method. |
| setter= | Use a custom name for the setter method. |
| readonly | Don’t synthesize a setter method. |
| nonatomic | Don’t guarantee the integrity of accessors in a multi-threaded environment. This is more efficient than the default atomic behavior. |
| strong | Create an owning relationship between the property and the assigned value. This is the default for object properties. |
| weak | Create a non-owning relationship between the property and the assigned value. Use this to prevent retain cycles. |
| copy | Create a copy of the assigned value instead of referencing the existing instance. |

Memory Management in Objective C

* In any programming language memory management is one of the important processes to make the effective usage of memory.
* It is the process of allocating memory when we require it, and de allocating the memory after using it.
* The goal of any memory management system is to reduce the memory wastage in program by controlling the lifetime of all its objects.
* In iOS and OS X applications this can be achieved through **object ownership**, which makes sure objects exist as long as they in use and it can de referenced after use.
* Object-ownership scheme is implemented through a **reference-counting system** that internally tracks how many owners that each object is having.
* When you claim ownership of an object then the reference count is increased by 1 and when you release the ownership of an object then reference count is decreased by 1.
* When the reference count of an object is one then it is guaranteed that the object is still exist and if the reference count of an object is 0 then the operating system will destroy that object.
* Memory management is the matter of performance of application, In application we have to free the unused object to increase the performance of the application.

**In Objective-C Memory management is classified into two types**

1. "Manual Retain-Release" or MRR
2. "Automatic Reference Counting" or ARC

**Manual Retain-Release" or MRR**

* In olden days, developers manually controlled an object’s reference count by calling special memory-management methods defined by the [NSObject protocol](https://developer.apple.com/library/mac/documentation/cocoa/reference/foundation/Protocols/NSObject_Protocol/Reference/NSObject.html). This is called **Manual Retain Release (MRR)**.
* In **Manual Retain Release developer has to take care of claiming the ownership to an object and releasing the ownership of an object after using it.**
* **The Manual Retain Release can be ensured by calling the special methods described below**

|  |  |
| --- | --- |
| **Method** | **Behavior** |
| alloc | Create an object and claim ownership of it. |
| retain | Claim ownership of an existing object. |
| copy | Copy an object and claim ownership of it. |
| release | Relinquish ownership of an object and destroy it immediately. |
| autorelease | Relinquish ownership of an object but defer its destruction. |

* If you not free the unused object in MRR it results the memory leak.
* Small leakage in memory will not make that much effect in memory, but if application consists of much memory leakage then your application will crash.
* When you try to access the de referenced object then also your program will end up with crash.

**Automatic Reference Counting (ARC)**

* In Xcode 4.2 **Automatic Reference Counting (ARC) is** introduced, which automatically inserts all of the methods of **Manual Retain Release (MRR)** for us during compile time.
* All modern applications should always use Automatic **Reference Counting (ARC) because it is more reliable** and lets us to focus on only app’s features instead of its memory management.
* **The Automatic Reference Counting works exactly same as Manual Retain Release, but it is automatically inserts the corresponding methods during compilation.**
* **In Automatic Reference Developer need not to take care of memory management and also how it works because this all happens implicitly for us**