**Objective C**

**Introduction:**

* Objective-C is general-purpose language that is developed on top of C Programming language by adding features of Small Talk programming language making it an object-oriented language.
* It is primarily used in developing iOS and Mac OS X operating systems as well as.
* Initially, Objective-C was developed by NeXT for its Next STEP OS from whom it was taken over by Apple for its iOS and Mac OS X.
* Cocoa is Apple’s native object-oriented application program interface (API) for the [OS X](https://en.wikipedia.org/wiki/OS_X) [operating system](https://en.wikipedia.org/wiki/Operating_system). For iOS, there is a similar API called [Cocoa Touch](https://en.wikipedia.org/wiki/Cocoa_Touch).

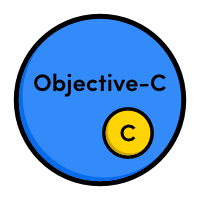


fig: relation between Objective c and c from rypress.com

**Objective-C Data Types**

|  |  |
| --- | --- |
| S.L.No | Description |
| 1 | **Basic Types:**  They are arithmetic types and consist of the two types:  (a) integer types  (b) floating-point types |
| 2 | **The type void:**  The *void* indicates that no value is available |
| 3 | **Derived types:** (a) Pointer types, (b) Array types, (c) Structure types, (d) Union types and (e) Function types |

**Integer Types**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Type** |  | **Storage size** |  | **Value range** | |
| char  unsigned char |  | 1 byte  1 byte |  | -128 to 127 or 0 to255  0 to 255 | |
| signed char |  | 1 byte |  | -128 to 127 | |
|  |  |  |  |  | |
|  |  |  |  |  | |
| int  unsigned int |  | 2 or 4 bytes  2 or 4 bytes |  | -32,768 to 32,767  0 to 65,535 | |
| short |  | 2 bytes |  | -32,768 to 32,767 | |
| unsigned short |  | 2 bytes |  | 0 to 65,535 | |
|  |  |  |  |  | |
|  |  |  |  |  | |
| long | 4 | bytes |  |  | |
|  |  |  |  |  | |
| unsigned long |  | 4 bytes |  | 0 to 4,294,967,295 | |
| E.g: integer datatype:  #import <Foundation/Foundation.h>  int main()  {  NSLog(@"Storage size for int : %d \n", sizeof(int));  return 0;  } | | | | |

**Floating Point types:**

In floating point there are different types :

1)single Precision floating point

2)double precision floating point

3) extended precision floating point

Objective C table:

|  |  |  |
| --- | --- | --- |
| type | storage size | value range |
| float  double  Long double | 4 byte  8 byte  10 byte | 1.2E-38 to 3.4E+38  2.3E-308 to 1.7E+308  3.4E-4932 to 1.1E+4932 |

**Eg:**

#import <Foundation/Foundation.h>

int main()

{

NSLog(@"Storage size for float : %d \n", sizeof(float));

return 0;

}

**Variables:**

syntax:

<type><name> and we have to assign a value to it you use=operator(assignment operators).

e.g.: int anint=65353;

**Void types:**

Means it does not return any value to the main. it used with function.

Void Types: There consists following types.

|  |  |
| --- | --- |
| type | Description |
| Function returns as void | There are various functions in Objective-C which do not return value or you can say they return void. A function with no return value has the return type as void. For example, void exit |
| Function arguments as void | There are various functions in Objective-C which do not accept any parameter. A function with no parameter can accept as a void. For example, int rand(void); |

**constants**: The constants refer to fixed values that the program may not alter during its execution

It is used to tell the compiler that variable not allowed to change.

Once we defined it cannot be changed.

**Operations:**

**Arithmetic Operators:**

Arithmetic Operations are used for doing mathematical operations. When we combine with two or more operands then it is called as expression.

The result of an arithmetic expression integer or floating value.

**Relational Operators:**

All the relational operators are binary operators

|  |  |
| --- | --- |
| operators | Description |
| a==b  a!=b  a>b    a>=b  a<b  a<=b      !a    a&&b      a||b | Equal to  Not Equal to  Greater than  Greater than equal to  Less than  Less than equal to  Logical negation    Logical and  Logical or |

E.g:

#import <Foundation/Foundation.h>

main()

{

int a = 21;

int b = 10;

int c ;

if ( a == b )

{

NSLog (@"1 - a is equal to b\n" );

}

else

{

NSLog (@"1 - a is not equal to b\n" );

}

if ( a < b )

{

NSLog (@"2 - a is less than b\n" );

}

else

{

NSLog (@"2 - a is not less than b\n" );

}

if ( a > b )

{

NSLog (@"3 - a is greater than b\n" );

}

else

{

NSLog (@"3 - a is not greater than b\n" );

}

a = 5;

b = 20;

if ( a <= b )

{

NSLog (@"4 - a is either less than or equal to b\n" );

}

if ( b >= a )

{

NSLog (@"5 - b is either greater than or equal to b\n" );

}

Output:

1= a is not equal to b

2= a is not less than b

3= a is greater than b

4= a is either less than or equal to b

5= b is either greater than or equal to b

**Loops:**

|  |  |
| --- | --- |
| **Type** | **Description** |
| For loop | Execute a sequence of statements multiple times there we use . |
| while loop | Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body. |
| do...while  loop | Like a while statement, except that it tests the condition at the end of the loop body. |
| |  |  | | --- | --- | | nested loops |  | | You can use one or more loops inside any another while, for or do..while loop. |

**Syntax: for loop:**

for ( initialization; condition checker; incrimination/decrimentation )

{

Statements;

}

Eg: for(int i=0;i<5;i++)

{

NSLog(@"the value of i is %d",i);

}

* **Syntax: while loop**

while (condition)

{

statement(s);

}

Eg:

#import <Foundation/Foundation.h>

// main method starts execution//

int main()

{

// declare variable

int a = 10;

// while loop execution it will check condition //

while( a < 20 )

{

NSLog(@"values of a : %d\n", a);

//increment //

a++;

}

return 0;

}

Output:

values of a : 10

values of a : 11

values of a : 12

values of a : 13

values of a : 14

values of a : 15

values of a : 16

values of a : 17

values of a : 18

values of a : 19

* **Syntax :do while:**

do

{

statement(s);

}while( condition );

**Eg:**

#import <Foundation/Foundation.h>

{

int main()

{

int a=10;

do

{ NSLog(@"the value of a is %d",a);

a=a+1;

}

while(a<20);

NSLog(@"the value of a is %d",a);

}

Output: Value of a is:10

Value of a is:11

**Classes and Objects**

**Classes:-**

* Objective-C classes provide the blueprint for creating objects.
* A class describes the behavior and properties common to any particular type of object.
* Class have reusable set of properties and behaviors inside of a class.
* In object-oriented programming terms, an object is an instance of a class.
* The class is defined in three different sections namely
* **@interface**
* **@implementation**
* **Program Section**
* **@ interface**:
* class definition starts with the keyword **@interface** followed by the interface(class) name; and the class body, enclosed by a pair of curly braces.
* An **interface** declares the public properties and methods of a class.
* interface resides in a (class name).h file.
* interface includes the list of messages that the class can receive
* interface we only declare the behaviour.
* **implementation:**
* **It** defines the code that actually makes these properties and methods work.
* implementation resides in a (class name).m file.
* Means (classname.h) whose implementation resides in classname.m.
* The first thing any class implementation needs to do is import its corresponding interface.
* **Program section**:
* In program section we write the program. we create object of a class and executing the program.

**Create the interface for Class:**

@interface Classname: NSObject

-(void)methodname;

@end;

**create the implementation For that class**:

@implementation Classname

-void methodname

{

NSLog(@" ");

}

**In main method we create Objects for Class:**

* Classname \*varname=[[Class name alloc] init];
* alloc it will allocate the memory for particular Class Object.
* init method used for initialization.

[variable name methodname];

**Objective C Data Types**

**1)NSString:**

* NSString: The string in Objective-C programming language is represented using NSString.
* It is class in Objective C. It is used to represent string value.
* NSString class is immutable.
* If you create an object whose contents cannot be changed. This is referred to as an immutable object.
* Means its contents are set at creation and cannot later be changed. If you want to represent a different string, you must create a new string object.
* NSObject is Super class of all class .
* NSString Does not have alloc method.
* NSString is used when if the value is fixed do not be changed or modified

in that scenario we go for NSString. IF you might want to delete some characters from a string or perform a search-and-replace operation on a string.

These types of strings are handled through the NSMutableString class.

**when we use NSString ?**

* + - NSString is used when if the value is fixed do not be changed or modifiedin that scenario we go for NSString.

**Creating a String:**

**NSString \*somestr = @”this is a string”;**

E.g: #import <Foundation/Foundation.h>

int main () {

NSString \*string = @"welcome";

NSLog(@"The string message is: %@\n", string);

return 0

}

Output: The string message is: welcome

**NSString Methods:**

**1) - (NSString \*)capitalizedString;**

* Returns a capitalized representation of the receiver.

2) **- (unichar)characterAtIndex:(NSUInteger)index;**

* Returns the character at a given array position.

3) **- (BOOL)hasPrefix:(NSString \*)aString;**

* Returns a Boolean value that indicates whether a given string matches the beginning characters of the receiver.

4) **- (BOOL)hasSuffix:(NSString \*)aString;**

* Returns a Boolean value that indicates whether a given string matches the ending characters of the receiver.

5) **- (NSString \*)lowercaseString;**

* Returns lowercased representation of the receiver.

**6) - (NSRange)rangeOfString:(NSString \*)aString;**

* Finds and returns the range of the first occurrence of a given string within the receiver.

**7)- (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.

8) **- (NSString \*)stringByTrimmingCharactersInSet:(NSCharacterSet \*)set;**

* Returns a new string made by removing from both ends of the receiver characters contained in a given character set.

9) **- (NSString \*)substringFromIndex:(NSUInteger)anIndex**

* Returns a new string containing the characters of the receiver from the one at a given index to the end.

10) **- (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.

**11)- (NSUInteger)length;**

Returns the number of Unicode characters in the receiver.

**2) NSMutableString:**

* NSMutableString is a class.
* it is subclass of NSString.
* If you want to modify or alter the content of string ,then we use NSMutableString. it will affect in same memory location no need to create a new object.
* We can do the operation like append, concant, lower to uppercase and upper to lower case these all operation can be done.
* NSMutableString add or remove in want to in that scenario we go for NSMutableString.

**when we use NSMutableString?**

If you want to modify or alter the content of string ,then we use NSMutableString.

**E.g:**

#import <Foundation/Foundation.h>

//main method starts execution //

int main ()

{

// declare string variables //

NSString \*str1 = @"Hello";

NSString \*str2 = @"World";

NSString \*str3;

int len ;

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

/ / uppercase string //

str3 = [str2 uppercaseString];

NSLog(@"Uppercase String : %@\n", str3 );

// concatenates str1 and str2 //

str3 = [str1 stringByAppendingFormat:@"World"];

NSLog(@"Concatenated string: %@\n", str3 );

// total length of str3 after concatenation //

len = [str3 length];

NSLog(@"Length of Str3 : %d\n", len );

/\* InitWithFormat \*/

str3 = [[NSString alloc] initWithFormat:@"%@ %@",str1,str2];

NSLog(@"Using initWithFormat: %@\n", str3 );

[pool drain];

return 0;

}

**Output:**

Uppercase String : WORLD

Concatenated string: Hello World

Length of Str3 : 10

Using initWithFormat: Hello World.

**NSArray:**

* NSArray is Objective-C’s general-purpose array type.
* NSArray it can work with all datatype primitive aswell as objects.
* Immutable Array are handled by NSArray class.
* NSArray index it cannot grow and shrink.
* Means only similar type of elements are allowed.
* once index has been created we cannot modify or alter the size. if you try to modify New Object is created.
* NSArray, we only replace the existing array. cannot change the contents of the existing array.
* NSArray is subclass of NSObject because it is base class of all class.
* If we do not want to modify the index of an array that time you go for NSArray.

**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.

**Creating a NSArray**

**NSArray \*array=[NSArray arraywithObjects:@"Hi,@Hello,@How,@are,@you];**

**Basic Example of NSArray**

#import <Foundation/Foundation.h>

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

{

int i;

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

// Create an array to contain the month names//

NSArray \*monthNames = [NSArray arrayWithObjects:

@"January", @"February", @"March", @"April",

@"May", @"June", @"July", @"August", @"September",

@"October", @"November", @"December", nil ];

// Now list all the elements in the array//

NSLog (@"Month Name");

NSLog (@"===== ====");

for (i = 0; i < 12; ++i)

NSLog (@" %2i %@", i + 1, [monthNames objectAtIndex: i]);

[pool drain];

return 0;

**}**

**Output:**

Month Name

===== ====

1 January

2 February

3 March

4 April

5 May

6 June

7 July

8 August

9 September

10 October

11 November

12 December

**NSArray Different Methods:**

**ComparingArray:**

* If u want to compare to array we use **isEqulaToArray** method.
* It returns YES when both arrays have the same number of elements and every pair pass an isEqual:comparison method.

eg: NSArray \*arr1=[[ NSArray alloc]initwithobjects:@"aa",@bb,@cc,nil];

NSArray \*arr2=[[ NSArray alloc]initwithobjects:@"bb",@cc,@aa,nil];

if([arr1 isEqualToarray:arr2])

{

NSLog(@"EQUAL");

}

else

{ NSLog(@"Not equal");

}

**Sorting Arrays:**

* Sorting is one of the main advantages of arrays.
* One of the most flexible ways to sort an array is with the **sortedArrayUsingComparator**: method.
* This accepts an NSComparisonResult(id obj1, id obj2) block, which should return one of the **following enumerators depending on the relationship between obj1 and obj2:**

| **Return Value** | **Description** |
| --- | --- |
| * NSOrderedAscending | obj1 comes before obj2 |
| * NSOrderedSame | obj1 and obj2 have no order |
| * NSOrderedDescending | obj1 comes after obj2 |

**String Conversion:**

* The **componentsJoinedByString**: method concatenates each element of the array into a string, separating them by the specified symbol(s).
* This can be useful for regular expression generation, file path manipulation, and rudimentary CSV processing.

eg:

NSArray \*arr=[[ NSArray alloc]initwithobjects:@"aa",@bb,@cc,nil];

NSLog**(@"%@",[**arr **componentsJoinedByString:@"," ])**

**Subdiving Arrays:**

* Subdividing an array is essentially the same as extracting substrings from an NSString, but instead of  substringWithRange: you use subarrayWithRange.

Eg:

NSArray \*arr=[[ NSArray alloc]initwithobjects:@"aa",@bb,@cc,dd,ee ,nil];

Nsarray \*lastarry=[ arr **subarrayWithRange**:NSmakeRange(2,4 )];

**NSMutableArray:**

* NSMutableArray is a Class.
* NSMutableArray is used to hold an mutable array of objects.
* It is inherited from NSArray. Means all the instances and methods are used to access in NSMutableArray.
* NSMutable Array it can grow and shrink .means index it can increase or decrease .
* NSMutable Array we can modify the content of existing array.
* But by using NSArray we cannot modify the contents.
* NSMutableArray we can add and remove the objects.

**Methods of NSMutableArray:**

* remove All Objects: Empties the array.
* add Object: Inserts a given object at the end of the array.
* removeObjectAtIndex: This is used to remove object At a specific index.
* exchangeObjectAtIndex:withObjectAtIndex: Exchanges the objects in the array at given indices.

**creation of NSMutable Array:**

NSMutableArray\* trucksArray =[NSMutableArray arrayWithObjects: @[<#objects, ...#>] nil];

* You can create empty mutable arrays using the array or  **arrayWithCapacity.**

**when to use NSArray and when to use NSMutableArray ?**

* If we do not want to modify or remove the index of an array that time you go for NSArray.
* NSMutable Array we can add or remove the content of existing array.

that time you go with NSMutable Array.

## NSDictionary:

## NSDictionary is a classes used in objective C.

## It is used to work with value and key pair.

* The keys in a dictionary must be unique, and they can be of any

object type, although they are typically **Strings**.

* The value associated with the key can also be of any object type, but it cannot be nil.

## NSDictionary is used to hold an immutable class means (we cannot modify the contents NSDictionary objects.)

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

* Dictionaries can be searched based on a particular key, and their contents

## can be enumerated.

## Accessing Values and Keys:

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

**Methods of NSDictionary:**

* 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.

**Create a NSDictionary:**

NSArray \*keys = [NSArray arrayWithObjects:@"key1", @"key2", @"key3", nil];

NSArray \*objs = [NSArray arrayWithObjects:@"obj1", @"obj2", @"obj3", nil];

NSDictionary \*dict = [NSDictionary dictionaryWithObjects:objs forKeys:keys];

// Iterate it

for (id key in dict)

{

    NSLog(@"key: %@   value:%@", key, [dict objectForKey:key]);

}

**By using Argument Method**

NSDictionary \*wordDictionary = [NSDictionary dictionaryWithObjectsAndKeys:

@"A friendly mammal commonly kept as a pet", @"Cat",

@"An awesome smartphone", @"iPhone",nil];

**Comparing Dictionaries:**

* Comparing dictionaries works the same as comparing arrays.
* The **isEqualToDictionary method** returns YES when both dictionaries contain the same key-value pairs.

**Eg:** NSDictionary \* dict1=@{

@"Om":[NSNumber numberwithInt:1],

@"ram":[ NSNumber numberwithInt:2],

@"jai":[ NSNumber numberwithInt:3]

};

NSDictionary \* dict2=@{

@"Om":[NSNumber numberwithInt:1],

@"ram":[ NSNumber numberwithInt:2],

@"jai":[ NSNumber numberwithInt:3]

}

if([dict1 **isEqualToDictionary:** dict2])

{

NSLog(@"Equal");

}

**else**

**{**

NSLog(@" not Equal");

**}**

**Sorting Dictionary Keys:**

* Dictionaries can’t be directly sorted into a new NSDictionary instance.
* But it is possible to sort the keys of the dictionary with **keysSortedByValueUsingComparator** method.
* Which accepts a block that should return one of the NSComparisonResult enumerators described in the [NSArray](http://rypress.com/tutorials/objective-c/data-types/nsarray.html#sorting-arrays) module.

|  |
| --- |
|  |

**NSMutableDictionary:**

* NSMutableDictionary is a class used in objective C.
* NSMutableDictionary means Dynamically it can be added or removed.

## NSMutableDictionary is used to hold an mutable dictionary means (we can modify the contents NSMutableDictionary objects.)

## It is used to work with Key value pair.

* You can add key-value pairs to the dictionary using the setObject:forKey: method. After the dictionary has been constructed, you can retrieve the value for a given key using the objectForKey.

**create a Mutable Dictionary:**

* NSMutableDictionary\*mutabledict=[NSMutableDictionary dictionary];

**Methods of NSMutableDictionary:**

* 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.

## Adding and Removing Entries:

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

## Combining Dictionaries:

* Mutable dictionaries can be expanded by adding the contents of another dictionary to its collection via the **addEntriesFromDictionary** method.

When to use **NSDictionary** and when to use **NSMutableDictionary ?**

## NSDictionary is used to hold an immutable class means we cannot modify the contents NSDictionary objects once created.

## we cannot add or remove the once the NSDictionary object is created in that scenario we use NSDictionary class.

**NSMutableDictionary**

* NSMutableDictionary is used to hold an mutable dictionary class means we can modify the contents.
* Once Object has been created we can add or remove the contents for key value pair in that scenario we use NSMutableDictonary.

id Datatype:-

* A variable is dynamically typed when the type of the object it points to is not checked at compile time.
* Objective-C uses the id data type to represent a variable that is an object without specifying what sort of object it is. This is referred to as *dynamic typing*.
* id data type and noted that it is a generic object type.
* id can be used for storing objects that belong to any class.
* This data type is exploited when it’s used this way to store different types of objects in a variable. during the execution of a program.
* id datatype it can used to achive dynamic polymorphism.
* 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.
* Dynamic typing is about using the *id* type to provide a variable that can be used to store objects of any type during program execution.
* *Dynamic binding* takes this one step further by allowing methods on an object stored in an *id* variable to be called without prior knowledge of the type of object currently assigned to the variable.

**Eg**:-

-(id)add:(int)a:(int)b

// we can change the implementation //

double result;

result=a+b;

return result;

**Method Overloading:**

* **What is Method Overloading** ?
* Method Overloading is a feature that allows a class to have two or more methods having same name, if their argument lists are different.
* Method Overloading can be done in single class.
* Argument differ in:

1.Number of parameters.

2. Data type of parameters.

3. Sequence of Data type of parameters.

* In Objective-C does not support Method overloading. why because it can allow only integer type as an argument.
* Suppose we are passing two or three argument as float it will show compilation Error.
* So the Objective C language does not supported method overloading concept.

**Method Overriding**

* **What is Method Overriding?**
* In Super class and subclass method having same name, same parameter and same return type is called Method Overriding.
* Method Overriding done between super class and sub class.
* What is the use of Method Overriding?
* Method overriding is used to provide specific implementation of a method that is already provided by its super class.
* **Rules For Method Overriding:**
* method must have same name as in the parent class.
* method must have same parameter as in the parent class.
* must be IS-A relationship (inheritance).
* **When we do method overriding** ?
* Whenever we want to add or change functionality in super class method that time we go for method overriding.

**Eg**:

@interface MyClass: NSObject

-(int)mynumber;

@end;

@implementation MyClass:NSObject

-(int)mynumber

{

return 1;

}@end

@interface MySubClass: MyClass

-(int)mynumber;

@end;

@implementation MyClass:NSObject

-(int)mynumber

{

return 2;

}

@end

**Selector**

* A selector is a message that can be sent to a class of an object.
* It can be used to refer simply to the name of a method when its used in a source-code message to an object.
* It also refers to unique identifier that replaces the name when source code is complied. Compiled selectors are of type SEL. All the methods with same name have the same selector.
* You can use a selector to invoke a method of an object.

**Methods and selectors**

* For efficiency full ascii names are not used as method selectors in compiled code.
* Instead, the compiler writes each method name into table, then pairs the name with a unique identifier that represents the method at runtime.
* The runtime system makes our each identifier is unique. No two selectors are same, and all methods with same name have the same selector.

**Types Of Selector:**

**1)SEL**

**2) @selector**

* Compiled selector are assigned to a special type, SEL to distinguish them from another data.
* Valid selectors are never 0.
* You must let the system assign SEL identifiers to methods; its pointless to assign them arbitrally
* The @selector directive lets you refer to the compiled selector, rather than to the full method name. Here, the selector for setWidth:Height: is assigned to the setWidthHeight variable.

**Example:**

SEL setWidthHeight;

setWidthHeight=@selector(setWidth:height);

this is the most efficient way to assign values to the SEL variables at compile time with the @selector directive.

* However in some cases, you may need to convert a character string to a selector at runtime. You can do this with the NSSelectorFromString function.

setWidthHeight= NSSelectorFromString(aBuffer);

* conversion in the opposite direction is also possible. The NSStringFromSelector function returns a method name for selector.

NSString \*method;

Method=NSStringFromSelector(setWidthHeight);

* Compiled selectors identify method names, not method implementations

The display method for one class, for example, has the same selector as display methods difined in other classes.

* This is essential for polymorphism and dynamic binding; it lets you send the same message to receivers belonging to different class.
* If there were one selector per method implementation, a message would be no different than a function call.
* A class method and an instance method with same name are assigned with same selector. However because of their separate domains, there is no confusion between two. A class could define a display class method in addition to display instance method.

**return type and argument type of methods:**

* The messaging routine has access to method implementations only through selector, so it treats all methods with selectors alike. It discovers the return type of a method and data type of its arguments, from the selector.
* Therefore, except for the message sent to statically typed receivers, dynamic binding requires all implementations of identically named methods to have the same return type and the same arguments type.
* Statically typed receivers are an exception to this rule, since the compiler can learn about the method implementation from the class type.
* Although identically named class methods and instance methods are represented by the name selector, they can have different argument and return type.

**Properties and Synthesize**

* @property is an objective C directive that allows to generate accessors.
* Here we can specify the name and type of the property.
* @property generates prototypes for getter and setter methods.
* We declare in an @interface block which is itself in a .h file.
* **@synthesize**: We Declare directive automatically generates setters and getters for us.
* the two properties are of different data types (double and long) it was necessary to declare them on separate *@property* lines. they been of the same data type we could have placed them on the same line, separated by commas:

@property int a, int b;

next step declare that synthesized accessor methods are required. This is performed in the *@implementation* section.

* data types of the properties involved, the variable names may be placed on a single *@synthesize* line.
* In assigning a name to a synthesized accessor method, Objective-C takes the name of the instance variable (for example *accountBalance*), capitalizes the first letter (*AccountBalance*) and then pre-fixes it with *set*(*setAccountBalance*).

**Definition of simple Property**

**@**interface SimpleProperties : NSObject

{

int count;

}

@property (read write) int count;

@end

@implementation SimpleProperties

@synthesize count

@end

* 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.

**Property Declarations Different Attributes:**

**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:** For strong we give Class/Object values.

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

* **Weak:** For weak we give Primitive Values.

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

* **Copy:** Duplicates values at initialization.
* **Assign:** Primitive types.

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>)

CATEGORIES

* Categories collect method implementations into separate files.
* The programmer can place groups of related methods into a category to make them more reliable.
* For instances one could create a “Reverse String” category “on” String object, while collecting all of the methods related to Reverse String.
* It contains .h and .m files.
* The methods within a category are added to a class at runtime. Thus, categories permit the programmer to add method to an existing class without the need to recompile that or even have access to its source code.
* When we write Object Oriented Programs, we will often want to add some behavior to an existing class. There are always new hoops for objects to jump through.
* For ex: We might have designed a new kind of tire, so we would subclass Tire & add the new behavior. When we want to add behavior to an existing class, we usually create a sub class.
* But sometimes sub classing isn't convenient. For ex: we may want to add new behaviors to NSString, but we realize that NSString is really the front end for a class cluster, and So it's difficult to sub class.
* In other cases, you might be able to make a subclass, but you are using a toolkit or library that won't be able to handle objects of the new class.
* For Ex: SubClass of NSString won't be returned when we create a new string with the string with format class method. The dynamic run-time dispatch mechanism employed by the Objective-C term for those new methods is 'categories'.
* crearing a category..

A category is a way to add new method to existinfg classes thus can be done to any class, even classes we don't have the source code for.

Let us say we are writing a cross word puzzle App that takes a series of strings, determines the length of each string, them puts those length into an NSArray or NSDictionary.

NSNumber \*number;

number = [NSNumber numberWithUnsignedInt:[Stringlength]];

//do something with number

@interface

//The declaration of a category looks a lot like the declaration for a class.

@interface NSString(NumberConvenience)

-(NSNumber \*) lengthAsNumber;

@end//Number Convenience

First, an existing class is mentioned, followed by a new name in parenthesis.

This means that the category is called "NumberConvenience", and it adds methods to NSString. Another way to say this is wwe are adding a category

onto NSString called NumberConvenience. You can add as many categories to a class as you want as long as the category names are unique.

We indicate the class you are putting the category onto (NSString),and the name of the category(Number convince),and list the methods you are adding ,following by @end.

Now instance variables cant be added ,so there is no instance variable section as there is with a class declaration .

@implementation

There is an @implementation companion to @interface

@implementation NSString (Number convince )

-(NSNumber \* )lengthAs Number

{

unsigned int length =[self length];

return ([NSNumber number withUnsigned int:length]);

}

@end

@implementation has the names of the class the category, along with the bodies of the new methods.

The lenghtAsNumber method gets the length of the string by calling [self length].This well be string to which you send the length AsNumber. then a new NSNumber is added with length.

numberWithUnsignedInt is not ' alloc','copy',or' new'method.

The NSNumber Object we create will get cleaned up when the currently active auto release pool is destroyed.

**Limitations Of Categories:**

* New instance variable to a class can't created
* The second limitations concerns name collosion ,in which one the category methods has the name as an existing method.when names collide the category
* Category methods will completely replace the original method ,with no way of getting the original back.