Objective C

**Introduction**

Objective-C is a programming language used to develop Apple’s ios and MAC(OS X) operating system. It has verbose naming convention so it prevents misunderstanding of the code. It contains identifiers, character set, variables like other object oriented language.

Objective-C is a super-set of C, object oriented features are added to C language to achieve oops concept.

**Variables**

Variables are named memory locations, variables are used to hold the values depending on the data type we have used. It follows the common rules or convention to declare a variable.

**Data type**

Data type specifies the type of the data it stores in the variable. Objective C supports all data types of C language and also it has its own set of data types like BOOL etc…

Eg: void, short, int, long, float, double, long double, long long

**Declaring variable**

**Syntax:**

**Data type variable\_name;**

**Short**

It is used to store whole number but the range is less compare to integer data type.

Format specifier of signed short is **%hd** for unsigned integer **%hu**.

**Integer**

Integer is used to store the whole number values. If we want to store whole numbers like 2,10,15… we will choose integer as a data type.

Format specifier of signed int is **%d** for unsigned integer **%u**.

**Float**

It is used to store the fractional values (single precession). If we want to store numbers like 2.2, 10.5, 15.7… we will choose float as a data type and we should use f as a suffix for each and every number.

Eg: float radius = 5.5f;

Format specifier is **%f** and if the representation is in exponential form then the format specifier is %e.

**Double**

It is also used to store the fractional values (double precession). If we want to store numbers like 2.234, 10.542, 15.75… we will choose double as a data type and we should give f as a suffix for each and every number we assigned to the variable.

Format specifier is **%8.2f**

Here 8.2f (modifier) specifies the number of digits before and after decimal point.

**BOOL**

BOOL is a data type it takes only YES or NO values. It is used to check whether true or false. YES and NO are keywords in objective-C.

**Long**

It is used to store long range of values. Storage size will be more compare to integer.

Format specifier of signed long is **%ld** for unsigned long **%lu**.

**Long long**

It is used to store long range of values. Storage size will be more compare to long.

Format specifier of signed long is **%lld** for unsigned long **%llu**.

**Comment:**

Objective –C also follows same convention like other language.

1. // is used for single line comment.
2. /\* and \*/ used for multi line comments

**Control statements**

Control statement alters the flow of execution of a program. If the block of code should be executed or not is decided by the condition specified in the control stalemates.

There are different types of control statements.

1. Selection statement.
2. Looping statement.
3. Branching statement.

Selection statement

If else statement

If the given condition is true then if block will be executed, if condition is false then false block will be executed.

Syntax:

if(condition)

{

//true block

Statements;

}

Else

{

//false block;

Statement;

}

Eg :

if(age>=18)

{

//if age is greater then 18 following things will execute

NSLog(@"eligible for voting");

}

else

{

//executes when age is below 18

NSLog(@"not eligible for voting");

}

**If else if condition**

It is used when we want to specify more than one condition in a program then if else if will be used.

Syntax:

if(condition)

{

Statements;

}

else if(conditon)

{

Statement;

}

else

{

Statements;

}

Eg :

if(percentage>=80 && percentage<=100)

{

NSLog(@"FCD");

}

else if(percentage>=60 && percentage<80)

{

NSLog(@"FC");

}

else if(percentage>=50 && percentage<60)

{

NSLog(@"SC");

}

else if(percentage>=35 && percentage<50)

{

NSLog(@"Pass");

}

else if(percentage >=1 && percentage<35)

{

NSLog(@"fail");

}

else

NSLog(@"invalid");

**Operators**

These are the operators used with control statements.

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

**Looping statements**

**For loop:**

If you want to execute group of statements repeatedly then we can use for loop. Group of statements will be executed until the specified condition becomes false.

**Syntax:**

For(initialization;condition;increment/decrement)

{

//Statements;

}

**NSString**

NSString is immutable class used to represent text in Objective C application.

It’s not only provides object-oriented wrapper for strings, NSString provides many utility methods for searching and manipulating contents.

Once NSString object is gets instantiated then the content of it cannot be changed because it is of type immutable.

**Creating NSString**

The most common way of creating string is using literal @”some string” syntax, but stringWithFormat: is a method of class NSString also used for generating strings. It takes same kind of format string as NSLog.

Eg:

NSString \*name=@”Bhaskar”;

NSString \*message=[NSString stringWithFormat: @”my name is %@”,name];

NSLog (@”%@”message);

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

* **-(NSUInteger)length** - returns number of characters in the string.
* **-(unichar)characterAtIndex**:(NSUInteger)theIndex - returns the character at the index.

These two methods make it possible to iterate through individual characters in a string.

For example:

#import <Foundation/Foundation.h>

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

{

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

int i;

NSLog (@"hello world");

NSString \*name=@"bhaskar";

for(i=0;i<[name length];i++)

{

NSLog(@"%c",[name characterAtIndex:i]);

}

[pool drain];

return 0;

}

Yet the real power of NSString comes in its higher-level functionality. Some of the most common methods are described in the following list, but keep in mind that this list is far from complete.

* +(id)stringWithFormat:(NSString \*)format ... - Create a string using the same placeholder format as NSLog().

**stringByAppendingString:**

* -(NSString \*)stringByAppendingString:(NSString \*)aString - Append a string to the receiving object.

#import <Foundation/Foundation.h>

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

{

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

int i;

NSLog (@"hello world");

NSString \*firstName=@"bhaskar";

NSString \*lastName=@"NS";

NSLog(@"%@",[firstName stringByAppendingString:lastName]);

[pool drain];

return 0;

}

* -(NSString \*)stringByApp stringByAppendingString:endingFormat:(NSString \*)format ... - Append a string using the same placeholder format as NSLog().
* -(NSString \*)lowercaseString - Return the lowercase representation of the receiving string.

**substringWithRange:**

* -(NSString \*)substringWithRange:(NSRange)aRange - Return a substring residing in aRange.

#import <Foundation/Foundation.h>

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

{

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

NSLog (@"hello world");

NSString \*firstName=@"bhaskar";

NSRange range=NSMakeRange(1, 4);

NSString \*subString=[firstName substringWithRange:range];

NSLog(@"%@",subString);

[pool drain];

return 0;

}

* -(NSRange)rangeOfString:(NSString \*)aString - Search for aString in the receiving string and return the location and length of the result as an NSRange(see following example for usage).
* -(NSString \*)stringByReplacingOccurancesOfString:(NSString \*)target withString:(NSString \*)replacement - Replace all occurrences of targetwith replacement.

**NSMutableString**

NSMutableString is a one that allows us to change individual characters without generating an entirely new string. If you are doing any small changes to mutable string then it changes the character in place. On other hand immutable string allocates or creates new string for each change.

NSMutableString is implemented as a sub class of NSString, so you can access to all methods of NSString class and can also have additional methods of NSMutableString for manipulating the characters array at place.

* -(void)appendString:(NSString \*)aString - Append aString to the end of the receiving string.
* -(void)append Format:(NSString \*)format ... - Append a string using the same placeholder format as NSLog().
* -(void)insertString:(NSString \*)aString atIndex (NSUInteger)anIndex - Insert a string into the specified index.
* -(void)deleteCharactersInRange:(NSRange)aRange - Remove characters from the receiving string.
* -(void)replaceCharactersInRange:(NSRange)aRange withString:(NSString \*)aString – Replace

Note that these methods all have void return types, whereas the corresponding NSString methods return NSString objects. This is indicative of the behavior of mutable strings: nothing needs to be returned, because the characters are manipulated in place.

**NSArray**

Arrays are ordered collection of objects used to maintain and sort list of data. Like NSString, NSArray is immutable, so its content cannot change dynamically without creating entirely new array.

**Creating array**

Immutable arrays can be defined as literal using @[] syntax, it can also create using factory method that is **arrayWithObjects**. Creation of arrays with different methods are demonstrated below.

**Eg:**

NSArray \*ios=@[@”harisha” ,@”bhaskar”, @”bhagyashree”,@”nadaf”, @”raghavendra”];

NSArray \*android=[NSArray arrayWithObjects:@”mahima”, @”sameer”, @”khadeer”, nil];

NSLog(@”first person in ios: %@”,ios[0]);

NSLog(@”first person in android: %@”,[android objectAtIndex:0]);

Few methods of NSArray object

* +(id)arrayWithObjects:(id)firstObject - Create a new array by passing in a list of objects.
* -(NSUInteger)count - Return the number of elements in the array.
* -(id)objectAtIndex:(NSUInteger)anIndex - Return the element in the array at index anIndex.
* -(BOOL)containsObject:(id)anObject - Return whether or not anObject is an element of the array.
* -(NSUInteger)indexOfObject:(id)anObject - Return the index of the first occurrence of anObject in the array. If the object is not in the array, return theNSNotFound constant.
* -(NSArray \*)sortedArrayUsingFunction:(NSInteger (\*)(id, id, void \*))compareFunction context:(void \*)context - Sort an array by comparing objects with a user-defined function (see the second example that follows for usage).

In NSArray all most all methods uses generic type **id** for an argument. NSArray accepts only objects, it cannot be used with primitive data types. If you want to work with primitive data type like int, float , char with NSArray, it can be facilitate by wrapping them into an object-oriented container.

Eg:

NSNumber \*n1 = [NSNumber numberWithFloat:22.5f];

NSNumber \*n2 = [NSNumber numberWithFloat:8.0f];

NSNumber \*n3 = [NSNumber numberWithFloat:-2.9f];

NSNumber \*n4 = [NSNumber numberWithFloat:13.1f];

NSArray \*numbers = [NSArray arrayWithObjects:n1, n2, n3, n4, nil];

NSLog(@"%@", numbers);

**NSMutableArray**

The NSMutableArray class allows us to dynamically add or remove items from arbitrary locations in the collection. Compare to set or dictionary insertion and deletion of elements from mutable array is slow. It is possible to change items after array as been allocated and it is possible to extend or shrink the number of elements.

NSMutableArray provides some extra methods to manipulating the contents of an array.

* +(id)arrayWithCapacity:(NSUInteger)numItems - Create an empty mutable array. The numItems argument is used as a size hint, so it should be roughly the number of initial items you plan to store.
* -(void)addObject:(id)anObject - Add the given object to the end of the existing array.
* -(void)insertObject:(id)anObject atIndex:(NSUInteger)anIndex - Insert the given object into the specified index.
* -(void)removeObjectAtIndex:(NSUInteger)anIndex - Remove the object at the specified index.
* -(void)removeAllObjects - Clear the array.
* -(void)replaceObjectAtIndex:(NSUInteger)anIndex withObject:(id)anObject- Overwrite the object at anIndex with anObject.
* -(void)exchangeObjectAtIndex:(NSUInteger)index1 withObjectAtIndex:(NSUInteger)index2 - Swap the locations of two objects in the array.

Most of the methods in mutable array are essentially “write ” methods and methods in the NSArray are “read” methods. These differences are same as NSString and NSMutableString.

**NSDictionary and NSMutableDictionary**

[Dictionaries](https://developer.apple.com/library/mac/#documentation/Cocoa/Reference/Foundation/Classes/NSDictionary_Class/Reference/Reference.html), also called associative arrays, are unordered associations of key-value pairs. It's possible to use any object as a key or a value, so dictionaries can be used for anything from dynamically assigning roles to objects to mapping string commands to functions.

**Creating dictionaries**

Immutable dictionaries can be defined using literal constants @{} syntax like array literals but you can also create using factory method i.e., dictionaryWithObjectsAndKeys.

**//using literal syntax**

NSDictionary \*student=@{@”harisha”:[NSNumber numberWithInt:01],

”bhagyashree”:[NSNumber numberWithInt:02],

”nadaf”:[NSNumber numberWithInt:03],

”raghavendra”:[NSNumber numberWithInt:04],

”bhaskar”:[NSNumber numberWithInt:05]

};

**//using factory method (dictionaryWithObjectsAndKeys)**

Student=[NSDictionary dictionaryWithObjectsAndKeys : [NSNumber numberWithInt:01],@”harisha”,

[NSNumber numberWithInt:02],@” bhagyashree”, [NSNumber numberWithInt:03],@”nadaf”, [NSNumber numberWithInt:04],@” raghavendra”, [NSNumber numberWithInt:05],@”bhaskar”, nil];

The dictionaryWithObjectsAndKeys: method treats its argument list as value-key pairs, so every two parameters define a single entry. This ordering is somewhat counterintuitive, so make sure that the value always comes before its associated key. The dictionaryWithObjects:ForKeys: method is a little bit more straightforward, but you should be careful to ensure that the key array is the same length as the value array.

Like string, array and set there is a immutable and mutable version. Some of the most common methods of NSDictionary are

* +(id)dictionaryWithObjectsAndKeys:(id)firstValue, (id)firstKey, ... - Create a dictionary by passing key-value pairs as parameters. Every two objects in the parameter list define a pair, and the first object defines the value, while the second object defines the key for that value (see next example for usage).
* -(NSUInteger)count - Return the number of entries in the dictionary.
* -(id)objectForKey:(id)aKey - Return the object (value) associated withaKey, or nil if there is no entry for aKey.
* -(NSArray \*)allKeys - Return a new array containing all of the keys in the dictionary.
* -(NSArray \*)allValues - Return a new array containing all of the values in the dictionary.
* -(NSArray \*)allKeysForObject:(id)anObject - Return a new array containing all of the keys associated with anObject. Note that it's possible to have multiple keys associated with a single object, so keys must be returned as an array, not a single object.

The two core methods of NSMutableDictionary are listed below, again note that these methods are “write” for associated and “read” methods of NSDictionary.

* -(void)setObject:)(id)anObject forKey forKey:i(id<NSCopying>)aKey - Add a new key-value pair to the dictionary.
* -(void)removeObjectForKey:(id)aKey - Remove the entry using aKey as its key.

**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 \*employees =@{@”harisha”:[NSNumber numberWithInt:01],

”bhagyashree”:[NSNumber numberWithInt:02],

”nadaf”:[NSNumber numberWithInt:03],

”raghavendra”:[NSNumber numberWithInt:04],

”bhaskar”:[NSNumber numberWithInt:05]

};

NSLog(@”Number of students in ios are %ld”, [student count]);

for(id i in employees)

{

NSLog(“%@”,employees[i]);

}

You can isolate a dictionary’s keys/values with the allKeys/allValuesmethods, which return an NSArray of each key/value in the collection, respectively. Note that there is no guarantee that these methods will return keys and values in the same order.

NSLog(“%@”,[employees allValues]);

NSLog(“%@”,[employees allKeys]);

## 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 withkeysSortedByValueUsingComparator:, 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. The following example sorts the models from most expensive to most affordable.

NSDictionary \*employees =@{@”harisha”:[NSNumber numberWithInt:01],

”bhagyashree”:[NSNumber numberWithInt:02],

”nadaf”:[NSNumber numberWithInt:03],

”raghavendra”:[NSNumber numberWithInt:04],

”bhaskar”:[NSNumber numberWithInt:05]

};

NSArray \*sorted Keys=[employees keysSortedByValueUsingComparator:

^NSComparisonResult(id obj1,id obj2){

Return [obj2 compare:obj1];

}];

NSLog(@”%@”,sortedKeys);

**NSMutableDictionary**

The NSMutableDictionary class lets you add new key-value pairs dynamically. Mutable dictionaries provide similar performance to mutable sets when it comes to inserting and deleting entries, and remember that both of these are a better choice than mutable arrays if you need to constantly alter the collection.

**Creating NSMutableDictionaries**

Mutable dictionaries can be created by calling any of the factory methods defined by NSDictionary on the NSMutableDictionary class. But, since many of these methods aren’t always the most intuitive to work with, you might find it useful to convert a literal dictionary to a mutable one using dictionaryWithDictionary:

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. The former can be used to either replace existing keys or add new ones to the collection. As an alternative, you can assign values to keys using the dictionary subscripting syntax, also shown below.

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";

## Combining Dictionaries

Mutable dictionaries can be expanded by adding the contents of another dictionary to its collection via the addEntriesFromDictionary:method. This can be used, for example, to combine jobs from two auto shop locations:

NSMutableDictionary \*jobs = [NSMutableDictionary

dictionaryWithDictionary:@{

@"Audi TT" : @"John",

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

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

@"Audi A7" : @"Bill"

}];

NSDictionary \*bakerStreetJobs = @{

@"BMW 640i" : @"Dick",

@"BMW X5" : @"Brad"

};

[jobs addEntriesFromDictionary:bakerStreetJobs];

This method also presents another option for creating mutable dictionaries:

// Create an empty mutable dictionary

NSMutableDictionary \*jobs = [NSMutableDictionary dictionary];

// Populate it with initial key-value pairs

[jobs addEntriesFromDictionary:@{

@"Audi TT" : @"John",

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

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

@"Audi A7" : @"Bill"

}];

**Id Data Type**

Id is a part of foundation framework.

Id data type is a generic object data type; it can hold a pointer to any Objective-C object, regardless of its class.

This makes it possible to store different kinds of objects in a single variable, opening the door to dynamic programming.

For example:

id lets you store an NSNumber, an NSDecimalNumber or an NSString in the same variable.

id mysteryObject = [NSNumber numberWithInt:5];

NSLog(@"%@", mysteryObject);

mysteryObject = [NSDecimalNumber decimalNumberWithString:@"5.1"];

NSLog(@"%@", mysteryObject);

mysteryObject = @"5.2";

NSLog(@"%@", mysteryObject);

Note that id implies that the value will be a pointer, so variable declarations don't require an asterisk before the variable name. In other words, variables should always be declared as id mysteryObject, not id \*mysteryObject.

Since an id variable doesn't check what kind of object it contains, it's the *programmer's* responsibility to makes sure he or she doesn't call methods or access properties that aren't defined on the object (e.g., don't try to call stringValue when the variable contains an NSString instance.

**Objective C class and objects**

**Class**

Objective-C classes provide the blueprint for creating objects. First, you define a reusable set of properties and behaviors inside of a class. You can instantiate object from that class to interact with those properties and behaviors.

When you create a Person class in Xcode, you will find two new classes **Person.h** and **Person.m**. .h file contains interface, it includes only method declarations, .m file contains the corresponding method implementation for the methods present in the .h file.

Separating a class' interface from its implementation makes it possible to hide implementation details from third-party objects. Other files that need to interact with the class import the *header* file-never the *implementation* file. This provides the abstract definition necessary to call methods and access properties while being completely independent of the class' implementation.

**Components of a class**

If you create a class Person, you will find the following Objective-C code.

**.h file(header file)**

#import <Foundation/Foundation.h>

@interface Person : NSObject

@end

The #import directive includes another file in the current context. Including a header file gives us access to all of the classes and functions it defines. In this case, we included the Foundation framework. The Foundation framework defines the basic constructs of the Objective-C language-things like strings, arrays, dictionaries, etc.-so it's a necessary part of virtually every Objective-C program.

The @interface directive begins an interface for a class. Next comes the class name, Person, followed by a colon and the parent class, NSObject. As noted earlier, NSObject is the top-level object in Objective-C. It contains the necessary methods for creating and destroying instances, along with some other useful functionality shared by all objects.

**.m file (implementation file)**

#import "Person.h"

@implementation Person

@end

Implementation files *must* include their associated header, otherwise they won't be able to find the class that they're trying to implement.

Also notice that this #import directive uses quotation marks instead of angled brackets. Quotation marks should be used to import *local* headers, while brackets indicate *global* headers. Global headers reside outside of the project and are linked to the compiler during the build process. Apple's standard frameworks are always included in angled brackets, whereas your project files should be imported with quotation marks.

And of course, the **.m** file uses the @implementation directive instead of @interface. Note that you *don't* have to specify the parent class here, since this information is already contained in the header.

**Instantiating Objects**

#import <Foundation/Foundation.h>

#import "Person.h"

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

    @autoreleasepool {

        Person \*somePerson = [[Person alloc] init];

    }

    return 0;

}

The Person \*somePerson expression declares a variable called somePerson and tells the compiler that it's going to hold an instance of the Person class. The asterisk next to the variable name indicates that it's a **pointer**, which is the most common way to reference objects in Objective-C.

**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 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:

**Super Class**

SuperClass.h

@interface SuperClass: NSObject

{

-(void) methodA;

}

@end

superClass.m

@implementation SuperClass

-()void method{

NSLog(@”methodA of SuperClass”);

}

@end

**Sub Class**

SubClass.h

@interface SubClass: SuperClass

{

-(void) methodA;

}

@end

subClass.m

@implementation SubClass

-()void method{

NSLog(@”methodA of SuperClass”);

}

@end

main.m

#import<Foundation/foundation>

#import “SubClass.h”

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

@ autoreleasepool{

SubClass \*sc=[[SubClass alloc]init];

[sc methodA];

}

return 0;

}

If you want to change or add implementation to super class methods then you will go for method overriding. The criteria for overriding is the signature of a method must be same in the sub class ie,

* return type of a method
* name of the method
* type and number of parameters

If you create a instance of sub class and send message to method A then on in the subclass implementation gets executed.

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 super keyword.

**Method overloading**

Method overloading is nothing but having the same name of methods with different functionality is called function overloading.

Objective-C does not support method overloading, because it depends on statically typed (Java, C#) and dynamically typed (Ruby, Python, Objective-C). In a dynamically typed language, type information is very often not known until runtime. At runtime, all objects are statically typed as id in Objective-C.

Additionally, a core idea in dynamically typed OO languages is that you should not care what type an object is as long as it responds to the messages you want to send. So overloading based on type would fly right in the face of that.

But in Objective C allows method overloading only if change in number of parameters.

For example

-(void) add: (int) firstNumber: (int) secondNumber;

-(void) add: (int) firstNumber: (int) secondNumber: (int) thirdNumber;

If we have change in number of parameters in a method then we can overload method in objective C or else the compiler will not allow to do .

**@property directive**

An object’s properties let other objects inspect or change its state. But, in a well-designed object-oriented program, it’s not possible to directly access the internal state of an object. Instead, **accessor methods**(getters and setters) are used as an abstraction for interacting with the object’s underlying data.

The goal of the @property directive is to make it easy to create and configure properties by automatically generating these accessor methods. It allows you to specify the behavior of a public property on a semantic level, and it takes care of the implementation details for you.

**Declaring properties**

Properties can be declared in an interface using the **@property** directive. For example lets create a class called Person. Properties are declared in @interface using property directive.

#import <Foundation/Foundation.h>

@interfacePerson : NSObject

@property NSString\*name;

@end

This section declares a property called name of type NSString.

**Implementing properties**

Here we use @synthesize directive to automatically create getter and setter methods. Remember that default getter method is simply a name of property and setter is just set is prefixed to the name of the property. The setter method for the above declared property is setName:

#import "Person.h"

@implementationPerson

@synthesizename = \_name;

@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.

**SEL and @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.

Constructors

Objective-C enables user to define constructor with the help of self and super keyword.

It has a parent class and a programmer can access its constructor by statement [super init], this statement returns an instance of parent class, which we assign to the “self” keyword. Actually “self” plays the same role as “this” keyword in java statement.

The default constructor is “-(id) init” statement

If(self) is used to check the condition self!=nil to confirm that parent class has returned a new object successfully.

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.

**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 aceess 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

}

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>)

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.