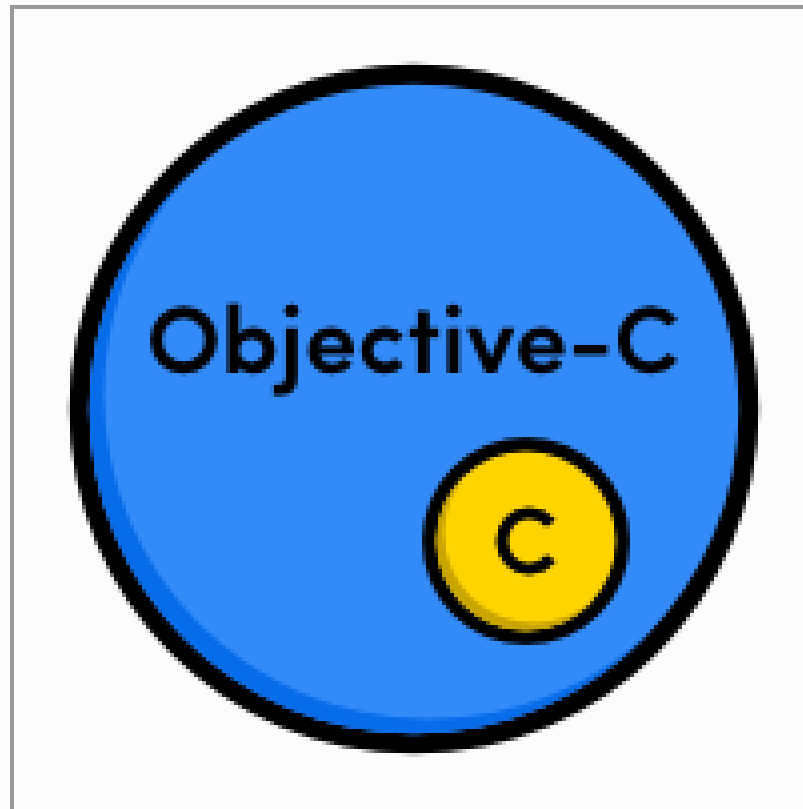


Unit – III – Objective C



Unit – III Objective-C Programming

- 1. Introduction to Objective-C**
- 2. Data Types and Expressions**
- 3. Decision Making and Looping**
- 4. Objects and Classes**
- 5. Property**
- 6. Messaging**
- 7. Category**
- 8. Extensions**
- 9. Fast Enumeration – NSArray and NSDictionary**
- 10. Methods and Selectors**
- 11. Static and Dynamic Objects**
- 12. Exception Handling**
- 13. Memory Management**
- 14. Required Tools – Xcode, iOS Simulator, Instruments, ARC and Frameworks**

1. Introduction

- Objective-C is a **general-purpose, object-oriented programming language** that adds **Smalltalk-style** messaging to the C programming language.
- This is the main programming language used by Apple for the **OS X and iOS operating systems** and their respective APIs, **Cocoa and Cocoa Touch**.
- 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**.

1.1 Objective-C Program Structure

- Preprocessor Commands
- Interface
- Implementation
- Method
- Variables
- Statements & Expressions
- Comments

Example

```
#import <Foundation/Foundation.h>
@interface SampleClass:NSObject
- (void)sampleMethod;
@end
```

```
@implementation SampleClass
- (void)sampleMethod {
    NSLog(@"Hello, World! \n");
}
@end
```

Example (con...)

```
int main() {  
    /* my first program in Objective-C */  
    SampleClass *sampleClass = [[SampleClass  
    alloc]init];  
    [sampleClass sampleMethod];  
    return 0;  
}
```

1.2 Various parts of the program

- The first line of the program *#import <Foundation/Foundation.h>* is a preprocessor command, which tells a Objective-C compiler to include *Foundation.h* file before going to actual compilation.
- The next line *@interface SampleClass:NSObject* shows how to create an interface. It inherits NSObject, which is the base class of all objects.
- The next line - *-(void)sampleMethod;* shows how to declare a method.
- The next line *@end* marks the end of an interface.

Various parts of the program (con...)

- The next line *@implementation* *SampleClass* shows how to implement the interface *SampleClass*.
- The next line - *(void)sampleMethod{}* shows the implementation of the *sampleMethod*.
- The next line *@end* marks the end of an implementation.
- The next line *int main()* is the main function where program execution begins.

Various parts of the program (con...)

- The next line `/*...*/` will be ignored by the compiler and it has been put to add additional comments in the program. So such lines are called **comments** in the program.
- The next line `NSLog(...)` is another function available in Objective-C which causes the message "Hello, World!" to be displayed on the screen.
- The next line `return 0;` terminates `main()` function and returns the value 0.

1.3 Token

- A Objective-C program consists of various **tokens** and a token is either,
 - a keyword
 - an identifier
 - a constant
 - a string literal, or a symbol.
- The **semicolon** is a statement terminator.

Token (con...)

```
NSLog(@"Hello, World! \n");
```

(or)

```
NSLog  
(  
@  
"Hello, World! \n"  
)  
;
```

2. Data types classified

- **Basic Types**
 - They are arithmetic types and consist of the two types: (a) integer types and (b) floating-point types.
- **Enumerated types**
 - They are again arithmetic types and they are used to define variables that can only be assigned certain discrete integer values throughout the program.
- **The void type**
 - The type specifier *void* indicates that no value is available.
- **Derived types**
 - They include (a) Pointer types, (b) Array types, (c) Structure types, (d) Union types and (e) Function types.

2.1 Data types

- Integer Types
 - char - 1 byte
 - unsigned char - 1 byte
 - signed char - 1 byte
 - int - 2 or 4 bytes
 - unsigned int - 2 or 4 bytes
 - short - 2 bytes
 - unsigned short - 2 bytes
 - long - 4 bytes
 - unsigned long - 4 bytes

Data types (con...)

- **Floating-Point Types**
 - float - 4 byte
 - double - 8 byte
 - long double - 10 byte
- **The void Type**
 - Function returns as void
 - Function arguments as void

2.2 Variables

- A 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
 - the **set of operations** that can be applied to the variable.

2.3 Basic variable types

- **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.

Example

```
#import <Foundation/Foundation.h>  
// Variable declaration:  
extern int a, b;  
extern int c;  
extern float f;
```

Example (con...)

```
int main () {  
    /* variable definition: */  
    int a, b; int c; float f;  
    /* actual initialization */  
    a = 10; b = 20; c = a + b;  
    NSLog(@"value of c : %d \n", c);  
    f = 70.0/3.0;  
    NSLog(@"value of f : %f \n", f);  
    return 0;  
}
```

2.4 Constants

- The constants refer to **fixed values** that the program may not alter during its execution.
- Constants can be of any of the **basic data types** like
 - *integer constant*
 - *floating constant*
 - *character constant*
 - *string literal*
 - *enumeration constant*

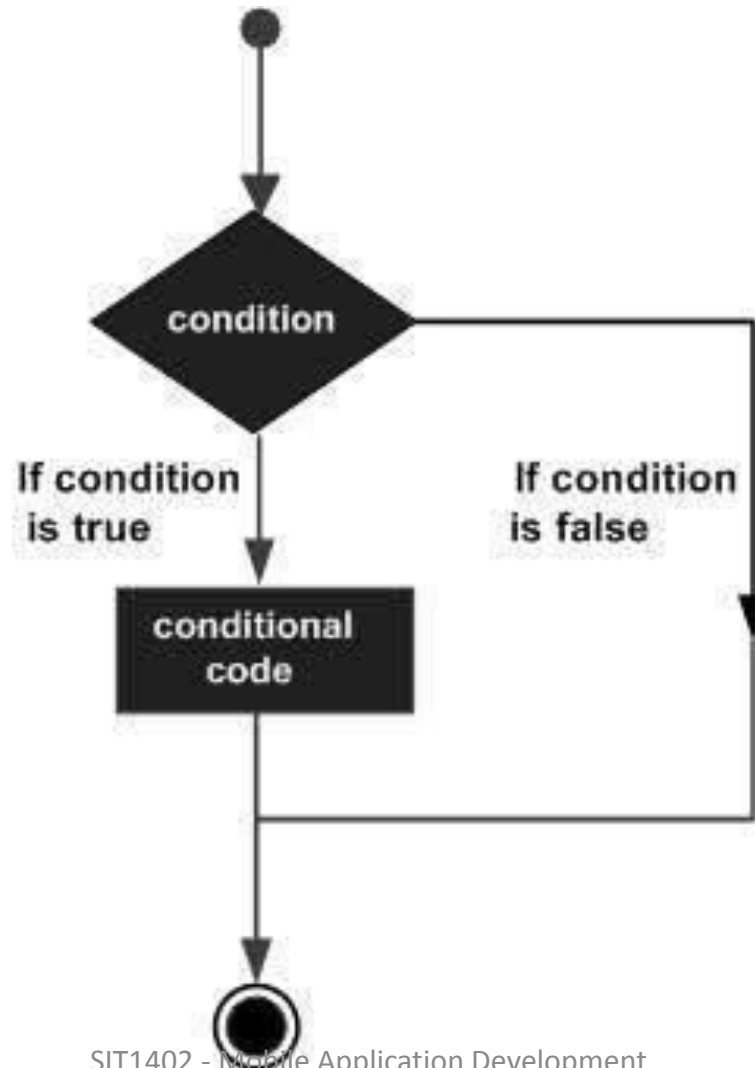
2.5 Operators

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Misc Operators
 - sizeof()
 - & operator
 - * operator
 - ?: operator

3 Decision Making - Branching

- Decision making structures require that the programmer specify **one or more conditions** to be evaluated or tested by the program, along with a statement or **statements to be executed if the condition is determined to be true**, and optionally, **other statements to be executed if the condition is determined to be false**.

Diagram



3.1 Types of decision making statements

- **If statement**

```
if(boolean_expression) {  
    /* statement(s) will execute if the boolean expression is true */  
}
```

- **If else statement**

```
if(boolean_expression) {  
    /* statement(s) will execute if the boolean expression is true */  
}  
else {  
    /* statement(s) will execute if the boolean expression is false */  
}
```

Branching (con...)

- Nested if statement

```
if( boolean_expression 1) {  
    /* Executes when the boolean expression 1 is  
    true */  
    if(boolean_expression 2) {  
        /* Executes when the boolean expression 2 is  
        true */  
    }  
}
```

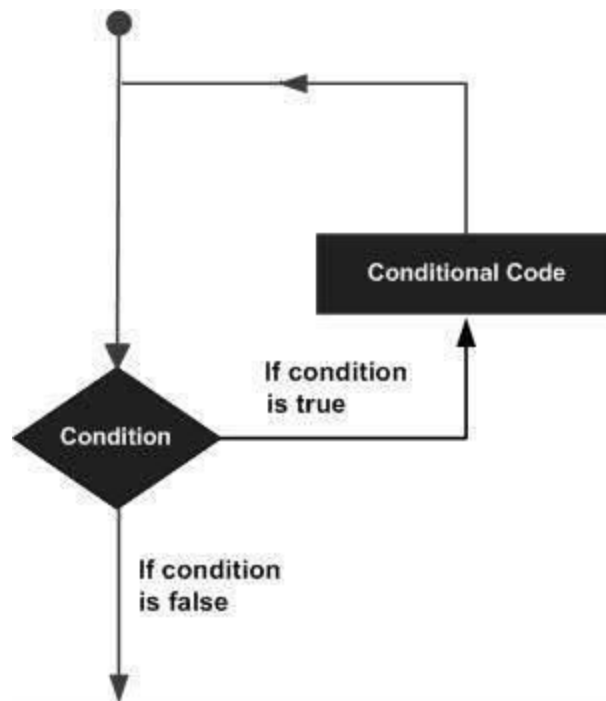

Branching (con...)

- Switch statement

```
switch(expression){  
    case constant-expression :  
        statement(s);  
        break;  
    /* you can have any number of case statements */  
    default :  
        /* optional */  
        statement(s);  
        break;  
}
```

3.2 Decision making - Looping

- A looping statement allows us to execute a **statement or group of statements multiple times.**



Looping statement

- While loop

```
while(condition) {  
    statement(s);  
}
```
- For loop

```
for ( init; condition;  
increment ) {  
    statement(s);  
}
```
- Do while loop

```
do {  
    statement(s);  
}while( condition );
```

4. Classes & Objects

- The main purpose of Objective-C programming language is **to add object orientation** to the C programming language.
- Classes are the central feature of Objective-C that support **object-oriented programming** and are often called user-defined types.
- 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**.

4.1 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.

4.2 Class Definitions

- Define a **blueprint** for a data type.
 - Define what the **class name** means?
 - What an **object** of the class will consist?
 - What **operations** can be performed on such an object?
- A class definition starts with the keyword **@interface** followed by the interface(class) name; and the class body, enclosed by a pair of curly braces.
- In Objective-C, all classes are derived from the base class called **NSObject**. It is the **super class** of all Objective-C classes. It provides basic methods like **memory allocation** and **initialization**.

Example

```
@interface Box:NSObject {  
    //Instance variables  
    double length;  // Length of a box  
    double breadth; // Breadth of a box  
}  
  
    // Property  
    @property(nonatomic, readwrite) double height;  
@end
```

4.3 Allocating and initializing Objects

- A class provides the blueprints for objects, so basically an **object is created from a class**.
- We **declare objects of a class with exactly the same sort of declaration** that we declare variables of basic types.
- **Example**

```
Box box1 = [[Box alloc] init]; // Create box1 object of type Box
```

```
Box box2 = [[Box alloc] init]; // Create box2 object of type Box
```


4.4 Accessing the Data Members

- The properties of objects of a class can be accessed using the **direct member access operator (.)**
- **Example**

```
#import <Foundation/Foundation.h>
```

```
@interface Box:NSObject {
```

```
    double length;
```

```
    double breadth;
```

```
    double height; }
```

```
    @property(nonatomic, readwrite) double height;
```

```
    -(double) volume;
```

```
@end
```

Accessing the Data Members (con...)

@implementation Box

@synthesize height;

-(id)init {

self = [super init];

length = 1.0;

breadth = 1.0;

return self; }

-(double) volume { return length*breadth*height; }

@end

Accessing the Data Members (con...)

```
int main( ) {  
    Box *box1 = [[Box alloc]init];  
    Box *box2 = [[Box alloc]init];  
    double volume = 0.0;  
    box1.height = 5.0;  
    box2.height = 10.0;  
    volume = [box1 volume];  
    NSLog(@"Volume of Box1 : %f", volume);  
    volume = [box2 volume];  
    NSLog(@"Volume of Box2 : %f", volume);  
    return 0; }
```

4.5 Function

- A function is a **group of statements** that together perform a task.
- A **function declaration** tells the compiler about a function's name, return type, and parameters.
- A **function definition** provides the actual body of the function.
- **Call the function** as method

4.6 Defining a Method

- **Syntax:**
 - (return_type) method_name:(argumentType1
argumentName1 joiningArgument2:(
argumentType2)argumentName2 ...
joiningArgumentN:(argumentTypeN
)argumentNameN {
body of the function
}

Example

```
/* function returning the max between two numbers */  
- (int) max:(int) num1 Num2:(int) num2 {  
    int result;  
    if (num1 > num2) {  
        result = num1;  
    }  
    else {  
        result = num2;  
    }  
    return result;  
}
```

4.7 Method Declaration

- **Syntax:**

- (return_type) function_name:(argumentType1
argumentName1 joiningArgument2:(
argumentType2)argumentName2 ...
joiningArgumentN:(argumentTypeN
)argumentNameN;

- **Example**

- (int) max:(int)num1 andNum2:(int)num2;

Example

```
#import <Foundation/Foundation.h>
@interface SampleClass:NSObject
    /* method declaration */
    - (int)max:(int)num1 andNum2:(int)num2;
@end
```


Example (con...)

```
@implementation SampleClass
/* method returning the max between two numbers */
- (int)max:(int)num1 andNum2:(int)num2{
    /* local variable declaration */
    int result;
    if (num1 > num2)
        result = num1;
    else
        result = num2;
    return result;
}
@end
```

Example (con...)

```
int main () {  
    /* local variable definition */  
    int a = 100;  
    int b = 200;  
    int ret;  
    SampleClass *sampleClass = [[SampleClass alloc] init];  
    /* calling a method to get max value */  
    ret = [sampleClass max:a andNum2:b];  
    NSLog(@"Max value is : %d\n", ret );  
    return 0;  
}
```

4.8 Log Handling

- NSLog method
 - To print logs, we use the **NSLog** method
 - **Syntax:**

```
NSLog(@"String");
```

- **Example**

```
#import <Foundation/Foundation.h>
int main() {
    NSLog(@"Hello, World! \n");
    return 0;
}
```

Example

- DebugLog Method

- To print logs in a live build.

```
#import <Foundation/Foundation.h>
```

```
#if DEBUG == 0
```

```
    #define DebugLog(...)
```

```
#elif DEBUG == 1
```

```
    #define DebugLog(...) NSLog(__VA_ARGS__)
```

```
#endif
```

Example (con...)

```
int main() {  
    DebugLog(@"Debug log, our custom addition gets \  
    printed during debug only" );  
    NSLog(@"NSLog gets printed always" );  
    return 0;  
}
```

Output

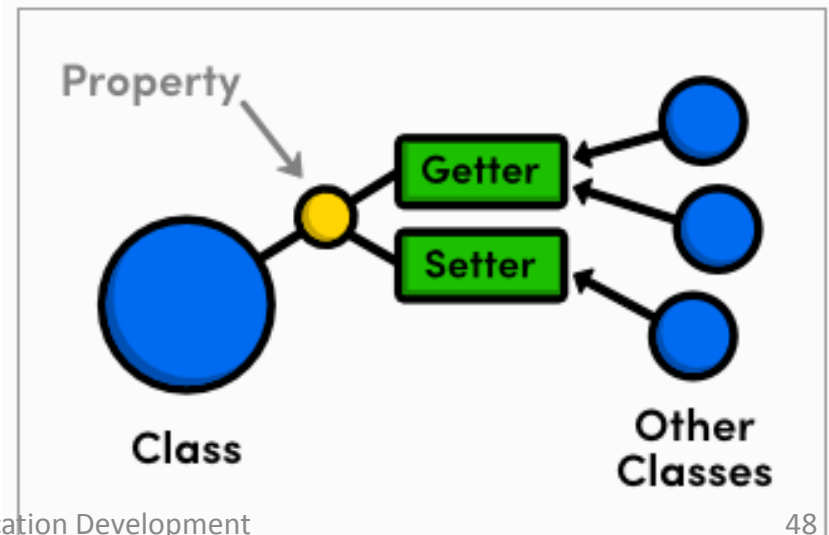
- We compile and run the program in **debug mode**, the output is,
 - Debug log, our custom addition gets printed during debug only
 - NSLog gets printed always
- We compile and run the program in **release mode**, the output is,
 - NSLog gets printed always

5. Property

- To ensure that the instance variable of the class can be **accessed outside the class**.
- The various **parts are the property** declaration are as follows
 - Properties begin with **@property**, which is a keyword
 - **Access specifiers** (atomic, nonatomic, readwrite, readonly, strong, weak)
 - This is followed by the **data-type** of the variable.
 - Finally, we have the **property name** terminated by a semicolon.
 - We can add **synthesize statement** in the implementation class.

Property (con...)

- Properties let other objects inspect or **change its state**
- A well-designed object-oriented program, it's not possible to directly access the **internal state of an object**
- **Accessor methods** are used
 - Setters
 - Getters



6. Messaging

- In Objective-C, messages **aren't bound** to method implementations until runtime.
- The compiler **converts a message expression**, **[receiver message]** into a call on a messaging function, **objc_msgSend**.
- This function takes the receiver and the name of the method mentioned in the message—that is, the method selector—as its two principal parameters:

objc_msgSend(receiver, selector)

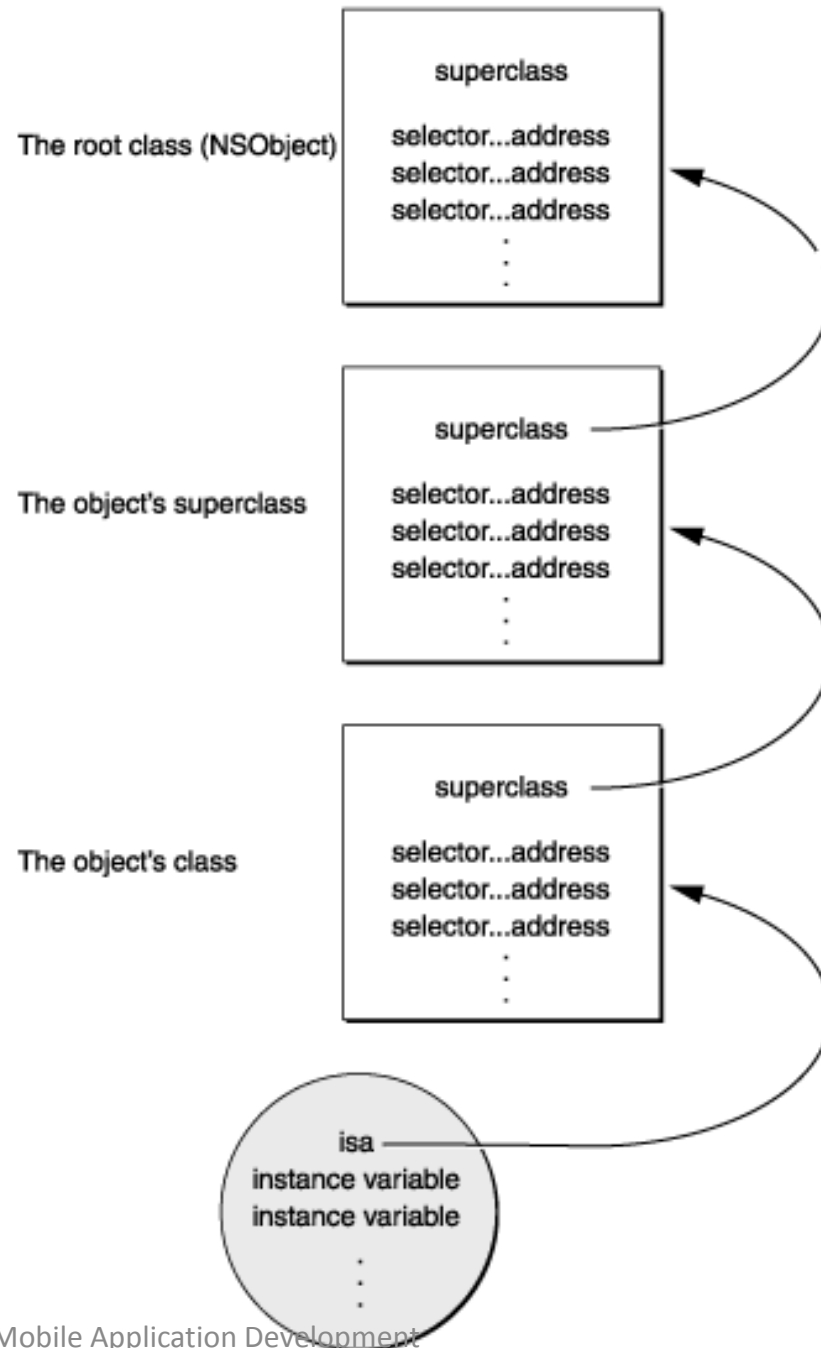
- Any arguments passed in the message are also handed to objc_msgSend:

objc_msgSend(receiver, selector, arg1, arg2, ...)

Messaging (con...)

- The **messaging function** does everything necessary for dynamic binding:
 - It first **finds the procedure** (method implementation) that the selector refers to. Since the same method can be implemented differently by separate classes, the precise procedure that it finds depends on the class of the receiver.
 - It then **calls the procedure**, passing it the receiving object (a pointer to its data), along with any arguments that were specified for the method.
 - Finally, **it passes on the return value** of the procedure as its own return value.

Messaging Framework



Messaging (con...)

- When a message is sent to an object, the messaging function follows the object's **isa pointer** to the class structure where it looks up the method selector in the dispatch table.
- If it can't find the selector here, **objc_msgSend follows the pointer to the superclass** and tries to find the selector in its dispatch table.
- Successive failures cause **objc_msgSend to climb the class hierarchy** until it reaches the NSObject class.
- Once it locates the selector, **the function calls the method entered** in the table and passes it the receiving object's data structure.

Messaging (con...)

- To speed the messaging process, the **runtime system caches the selectors and addresses** of methods as they are used.
- There's a **separate cache for each class**, and it can contain selectors for inherited methods as well as for methods defined in the class.
- Before searching the dispatch tables, the messaging routine first **checks the cache of the receiving object's class**.
- If the method selector is in the cache, messaging is only slightly slower than a function call.
- Caches grow dynamically to accommodate new messages as the program runs.

7. Categories

- To extend an **existing class by adding behavior** that is useful only in certain situations.
- If you need to add a method to an existing class, the easiest way is to **use a category**.
- To declare a category uses the **@interface** keyword.
- Syntax

```
@interface ClassName (CategoryName)
```

```
-----
```

```
@end
```

7.1 Characteristics of category

- A category can be declared for any class, even if you don't have the original implementation source code.
- Any methods that you declare in a category will be available to all instances of the original class, as well as any subclasses of the original class.
- At runtime, there's no difference between a method added by a category and one that is implemented by the original class.

Person.h

```
@interface Person : NSObject
    @property (readonly) NSMutableArray* friends;
    @property (copy) NSString* name;
    - (void)sayHello;
    - (void)sayGoodbye;
@end
```


Person.m

```
#import "Person.h"

@implementation Person

@synthesize name = _name;
@synthesize friends = _friends;

-(id)init{
    self = [super init];
    if(self){
        _friends = [[NSMutableArray alloc] init];
    }
    return self;
}

- (void)sayHello {
    NSLog(@"Hello, says %@.", _name);
}

- (void)sayGoodbye {
    NSLog(@"Goodbye, says %@.", _name);
}

@end
```

Person+Relations.h

```
#import <Foundation/Foundation.h>
```

```
#import "Person.h"
```

```
@interface Person (Relations)
```

- (void)addFriend:(Person *)aFriend;
- (void)removeFriend:(Person *)aFriend;
- (void)sayHelloToFriends;

```
@end
```

Person+Relations.m

```
#import "Person+Relations.h"
```

```
@implementation Person (Relations)
```

```
- (void)addFriend:(Person *)aFriend {  
    [[self friends] addObject:aFriend];  
}
```

```
- (void)removeFriend:(Person *)aFriend {  
    [[self friends] removeObject:aFriend];  
}
```

```
- (void)sayHelloToFriends {  
    for (Person *friend in [self friends]) {  
        NSLog(@"Hello there, %@!", [friend name]);  
    }  
}
```

```
@end
```

main.m

```
#import <Foundation/Foundation.h>
#import "Person.h"
#import "Person+Relations.h"

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        Person *joe = [[Person alloc] init];
        joe.name = @"Joe";
        Person *bill = [[Person alloc] init];
        bill.name = @"Bill";
        Person *mary = [[Person alloc] init];
        mary.name = @"Mary";

        [joe sayHello];
        [joe addFriend:bill];
        [joe addFriend:mary];
        [joe sayHelloToFriends];
    }
    return 0;
}
```

8. Extensions

- A class extension is **similar to a category**, but it can only be added to a class for which you have the source code at compile time.
- The methods declared by a class extension are implemented in the **implementation block for the original class**.
- Extensions are actually categories **without the category name**. It's often referred as **anonymous categories**.
- The syntax to declare a extension uses the **@interface** keyword.

- **Syntax**

```
@interface ClassName ()
```

```
-----
```

```
@end
```

8.1 Characteristics of extensions

- An extension **cannot be declared for any class**, only for the classes that we have original implementation of source code.
- An extension is **adding private methods and private variables** that are only specific to the class.
- Any method or variable declared inside the extensions is **not accessible even to the inherited classes**.

Example

```
@interface SampleClass : NSObject {  
    NSString *name;  
}
```

```
- (void)setInternalID;  
- (NSString *)getExternalID;
```

```
@end
```

```
@interface SampleClass() {  
    NSString *internalID;  
}
```

```
@end
```

Example (con...)

@implementation SampleClass

- (void)setInternalID {

internalID = [NSString stringWithFormat:

@\"UNIQUEINTERNALKEY%dUNIQUEINTERNALKEY\",arc4r
andom()%100];

}

- (NSString *)getExternalID {

return [internalID stringByReplacingOccurrencesOfString:

@\"UNIQUEINTERNALKEY\" withString:@\" \"];

}

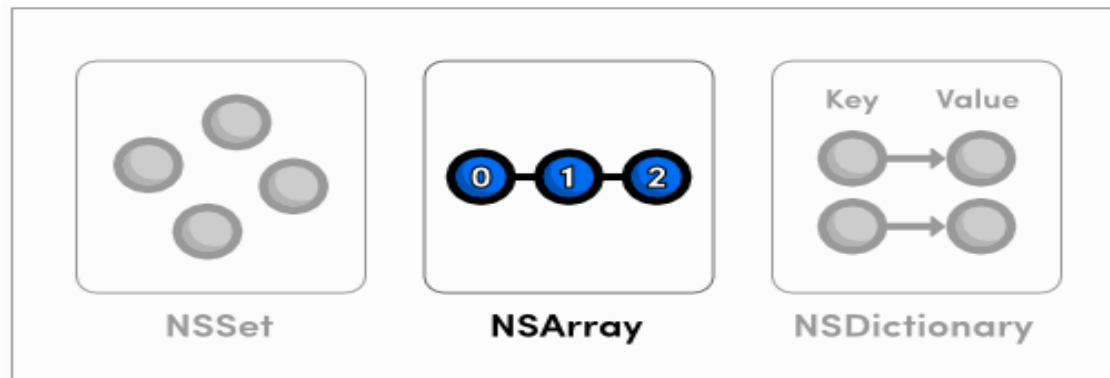
@end

Difference between Category & Extension

Category	Extension
Categories to define additional methods of an existing class—even one whose source code is unavailable to you	A class extension is similar to a category, but it can only be added to a class for which you have the source code
Category have category name	Extension dont have name
It helps to add some more functionality to existing class, but only functions	It helps to add some more functionality to existing class, but only properties and instance variables
It come with its own .h and .m file	It comes with .m file only
@interface MyClass (Category Name) // method declarations @end	@interface MyClass () // method declarations @end

9. Fast Enumeration-NSArray

- NSArray is **general-purpose** array type.
- It represents an **ordered collection of objects**.
- Like NSSet, **NSArray is immutable**, so you cannot dynamically add or remove items.
- Immutable arrays can be defined as literals using the **@[]** syntax.



Fast Enumeration-NSArray (con...)

- Eg.

```
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]);
```

Fast Enumeration-NSArray (con...)

- Fast-enumeration is the most **efficient way to iterate over an NSArray**, and its contents are guaranteed to appear in the correct order.
- Eg.

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

Fast Enumeration-NSArray (con...)

// 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]);
}
```

Fast Enumeration-NSArray (con...)

- There are several **advantages** to using fast enumeration:
 - The enumeration is considerably **more efficient** than, for The syntax is concise
 - **Enumeration is “safe”**—the enumerator has a mutation guard so that if you attempt to modify the collection during enumeration, an exception is raised

Fast Enumeration-NSDictionary

- The NSDictionary class represents an **unordered collection** of objects
- They associate each **value with a key**, which acts like a label for the value. This is useful for modeling relationships between **pairs of objects**
- NSDictionary is immutable, but the **NSMutableDictionary** data structure lets you dynamically add and remove entries as necessary.
- Immutable dictionaries can be defined using the literal **@{}** syntax.
- Factory methods
 - dictionaryWithObjectsAndKeys:
 - dictionaryWithObjects:forKeys:

Fast Enumeration-NSDictionary (con...)

- Eg.

// 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], };
```


Fast Enumeration-NSDictionary (con...)

Eg.

// 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];
```

Fast Enumeration-NSDictionary (con...)

- Eg.

// 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);
```

Fast Enumeration-NSDictionary

- 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.
- Eg.

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

```
for (id key in inventory) {  
    NSLog(@"There are %@ %@'s in stock",  
          inventory[key], key); }
```

10. Methods and Selectors

- A **selector** refers to the name used to select a method to execute for an object
- It is used to **identify a method**
 - **Compiler writes** each method name into a table
 - **Pairs the name with a unique identifier** that represents the method at runtime
 - The **runtime system** makes sure each identifier is unique
 - **No two selectors are the same**, and all methods with the same name have the same selector

11. Static Class

- A class is a **blue print for the members** like, static variable and static methods.
- A static variable are declared using the modifier **static**.
- Syntax

static <data_type> <variable_name>;

- Eg.

static int number;

Static Class (con...)

- For **static method** which is also known as class method, you can use the **+ sign** instead of the – sign when declaring the method.
- Syntax
+ (data_type)method_name;
- Eg.
+ (int)getNumber;

11.1 Dynamic Objects

- Dynamic binding is determining the **method to invoke at runtime instead of at compile time**.
Dynamic binding is also referred to as late binding.
- In Objective-C, all **methods are resolved dynamically at runtime**.
- The exact code executed is determined by both the method name (the selector) and the receiving object.
- Dynamic binding enables **polymorphism**.

Dynamic Objects (con...)

- Eg.
 - Consider a collection of objects including **Rectangle and Square**.
 - Each object has its **own implementation** of a `printArea` method.
 - The actual code that should be executed by the expression **[anObject printArea]** is determined at runtime.
 - In this, **printArea method** is dynamically selected in runtime.

Example

```
#import <Foundation/Foundation.h>
@interface Square:NSObject
{
    float area;
}
- (void)calculateAreaOfSide:(CGFloat)side;
- (void)printArea;
@end
@implementation Square
- (void)calculateAreaOfSide:(CGFloat)side
{
    area = side * side;
}
- (void)printArea
{
    NSLog(@"The area of square is %f",area);
}
@end
```

Example (con...)

```
@interface Rectangle:NSObject
{
    float area;
}
- (void)calculateAreaOfLength:(CGFloat)length andBreadth:(CGFloat)breadth;
- (void)printArea;
@end

@implementation Rectangle
- (void)calculateAreaOfLength:(CGFloat)length andBreadth:(CGFloat)breadth
{
    area = length * breadth;
}
- (void)printArea
{
    NSLog(@"The area of Rectangle is %f",area);
}
@end
```

Example (con...)

```
int main()
{
    Square *square = [[Square alloc]init];
    [square calculateAreaOfSide:10.0];
    Rectangle *rectangle = [[Rectangle alloc]init];
    [rectangle calculateAreaOfLength:10.0 andBreadth:5.0];
    NSArray *shapes = [[NSArray alloc]initWithObjects: square,
        rectangle,nil];
    id object1 = [shapes objectAtIndex:0];
    [object1 printArea];
    id object2 = [shapes objectAtIndex:1];
    [object2 printArea];
    return 0;
}
```

12. Exception Handling

- An exception is a special condition that **interrupts the normal flow** of program execution
- There are a **variety of reasons** why an exception may be generated, by hardware as well as software
- Exception handling is made available in Objective-C with foundation class **NSException**

Exception Handling (con...)

- Objective-C exception support **four compiler directives**:
 - **@try** - This block tries to execute a set of statements
 - **@catch** - This block tries to catch the exception in try block
 - **@throw** - throw exceptions if you find yourself in a situation that indicates a programming error, and want to stop the application from running

Exception Handling (con...)

- **@finally** - This block contains set of statements that always execute
- **Eg.** division by zero, underflow or overflow, calling undefined instructions

```
#import <Foundation/Foundation.h>
```

```
int main(){
```

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

```
NSMutableArray *array = [[NSMutableArray alloc] init];
```

```
@try {
```

```
NSString *string = [array objectAtIndex:10];
```

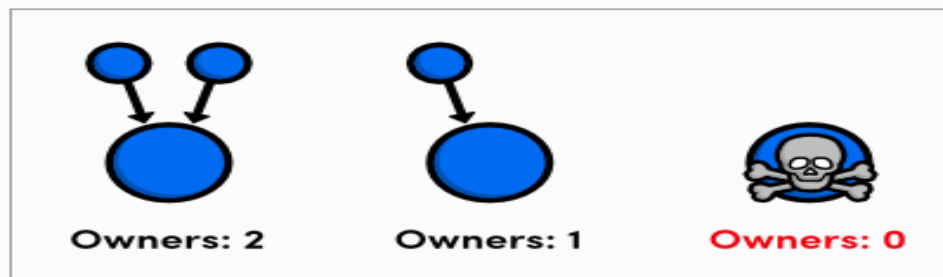
```
}
```

Exception Handling (con...)

```
@catch (NSException *exception) {  
    NSLog(@"%@ ",exception.name);  
    NSLog(@"Reason: %@ ",exception.reason);  
}  
  
@finally {  
    NSLog(@"@finaly Always Executes");  
}  
  
[pool drain];  
return 0;  
}
```

13. Memory Management

- Objects reside in memory, and especially on mobile devices this is a **scarce resource**
- To make sure that programs **don't take up any more space** than they need
- The goal of any memory management system is to reduce the memory footprint of a program by controlling the lifetime of all its objects. iOS and OS X applications accomplish this through **object ownership**, which makes sure objects exist as long as they have to, but no longer
- Many languages accomplish this through **garbage collection**, but Objective-C uses **object ownership**



SIT1402 - Mobile Application Development
Destroying an object with no owners

13.1 Manual Retain Release environment

- **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.

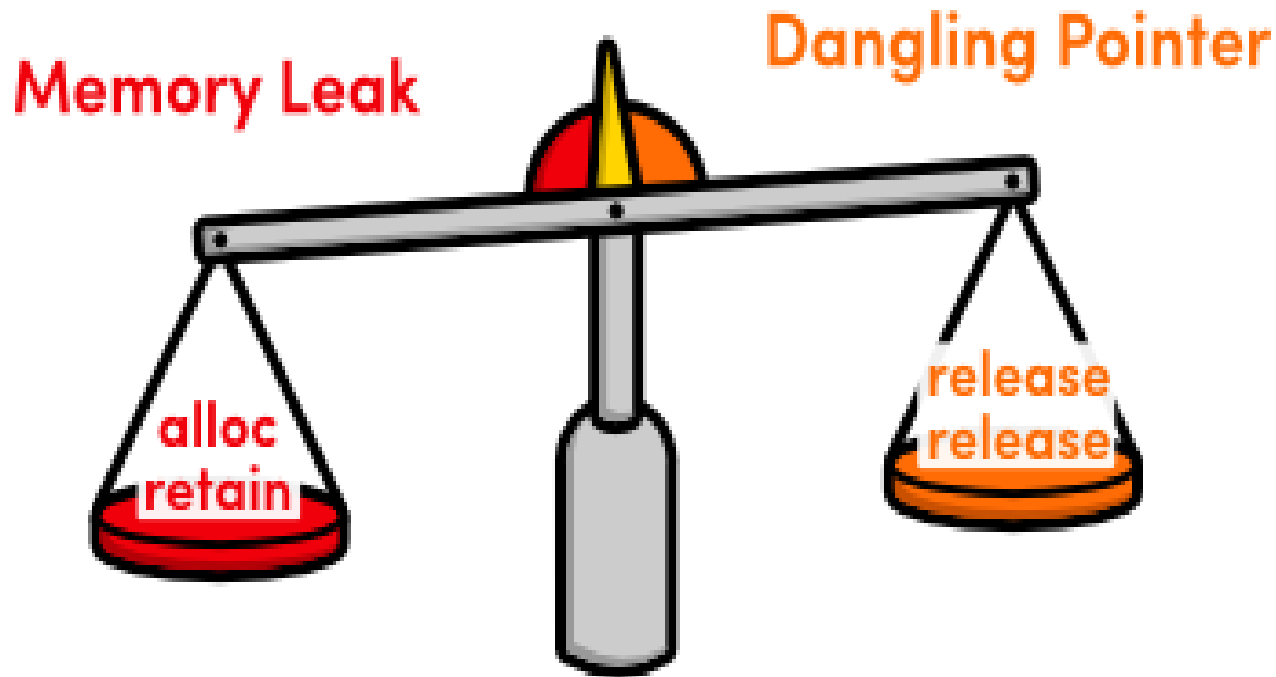
Manual Retain Release environment (con...)

- Manually controlling object ownership might seem like a **daunting task**, but it's actually very easy.
- All you have to do is **claim ownership** of any object you need and remember to **relinquish ownership** when you're done with it.
- When you **forget to balance** these calls, one of two things can happen.
 - If you forget to release an object, its underlying memory is never freed, resulting in a **memory leak**.

Manual Retain Release environment (con...)

- Small leaks won't have a visible effect on your program, but **if you eat up enough memory**, your program will eventually crash.
- On the other hand, if you try to release an object too many times, you'll have what's called a **dangling pointer**.
- When you try to access the dangling pointer, you'll be requesting an **invalid memory address**, and your program will most likely crash.

Manual Retain Release environment (con...)



13.2 The alloc Method

- Using the **alloc** method to create objects. But, it's not just allocating memory for the object, it's also **setting its reference count to 1**.
- Eg.

```
#import <Foundation/Foundation.h>

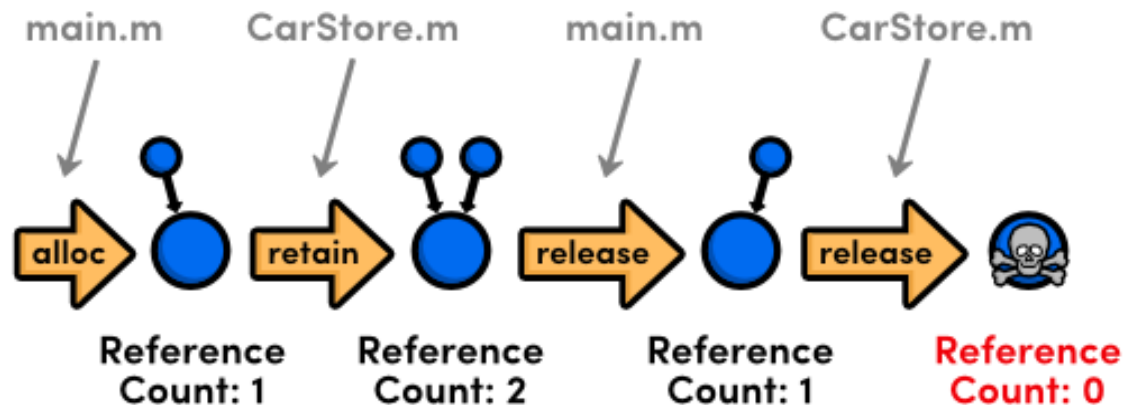
int main(int argc, const char * argv[]) {
    @autoreleasepool {
        NSMutableArray *inventory = [[NSMutableArray alloc] init];
        [inventory addObject:@"Honda Civic"];
        NSLog(@"%@", inventory);
    }
    return 0;}
```

13.3 The release Method

- The **release** method relinquishes ownership of an object by **decrementing its reference count**.
- So, we can get **rid** of our memory leak by adding the following line after the NSLog()
- Eg.
`[inventory release];`

13.4 The Retain Method

- The retain method **claims ownership** of an existing object.
- It's like telling the operating system, “**Hey! I need that object too, so don't get rid of it!**”.



The Retain Method (con...)

- Eg.

```
// CarStore.h
```

```
#import <Foundation/Foundation.h>
```

```
@interface CarStore : NSObject
```

```
    -(NSMutableArray *)inventory;
```

```
    -(void)setInventory:(NSMutableArray *)newInventory;
```

```
@end
```

```
// CarStore.m
```

```
    -(void)setInventory:(NSMutableArray *)newInventory {  
        _inventory = [newInventory retain];  
    }
```

```
}
```


13.5 The autorelease method

- The **autorelease** method relinquishes ownership of an object, but instead of destroying the object immediately, it defers the actual freeing of memory until later on in the program.
- This allows you to **release objects** when you are “supposed” to, while **still keeping them around for others to use**.
- Eg.

```
// CarStore.h
```

```
+ (CarStore *)carStore;
```

```
// CarStore.m
```

```
+ (CarStore *)carStore {
```

```
    CarStore *newStore = [[CarStore alloc] init];
```

```
    return [newStore autorelease];
```

```
}
```

13.6 The dealloc Method

- An object's **dealloc** method is the opposite of its init method.
- It's called right before the object is destroyed, giving you a chance to **clean up any internal objects**.
- This method is called **automatically by the runtime**—you should never try to call dealloc yourself.
- Eg.

// CarStore.m

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

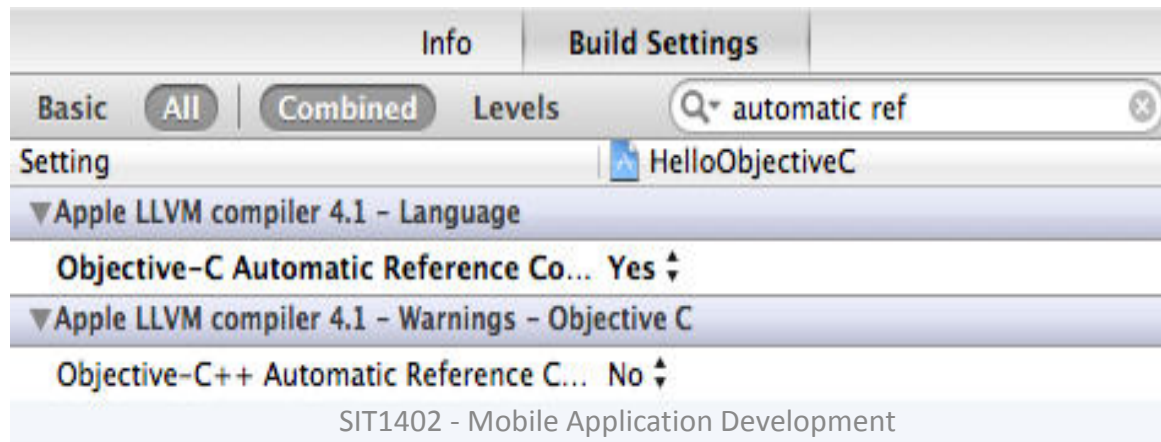
Memory Management (con...)

- Automatic Reference Counting works the exact same way as MRR, but it **automatically inserts the appropriate memory-management methods for you.**
- This is a big deal for Objective-C developers, as it lets them focus entirely on ***what*** their application needs to do rather than ***how*** it does it.
- **ARC takes the human error out of memory management** with virtually no downside, so the only reason *not* to use it is when you're interfacing with a legacy code base.
- The rest of this module explains the major changes between MRR and ARC

Memory Management (con...)

- Enabling ARC

- First, let's go ahead and **turn ARC back on** in the project's *Build Settings* tab.
- Change the *Automatic Reference Counting* compiler option to **Yes**.
- Again, this is the default for all **Xcode templates**, and it's what you should be using for all of your projects

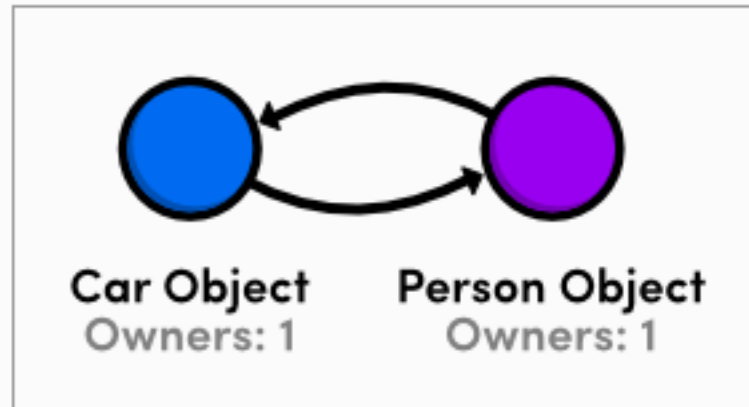


Memory Management (con...)

- **Automatic Reference Counting**, the compiler manages all of your object ownership automatically
- We **never to worry** about how the memory management system actually works
- To understand the **various attributes of @property**, since they tell the compiler what kind of relationship objects should have
 - Strong attribute
 - Weak attribute
 - Copy attribute

Memory Management (con...)

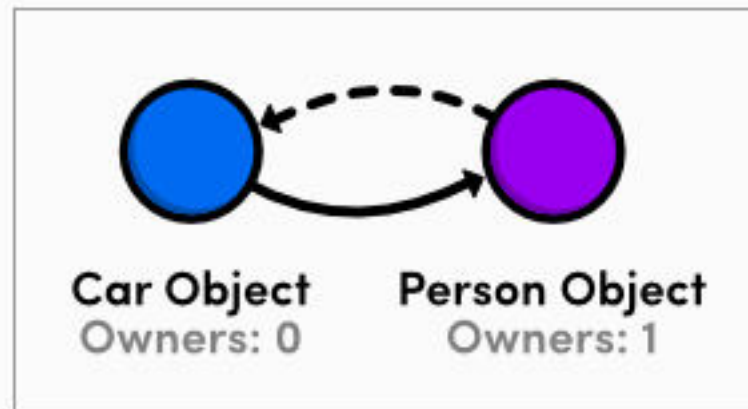
- **The strong Attribute**
 - It creates an **owning relationship** to whatever object is assigned to the property
 - It makes sure the **value exists** as long as it's assigned to the property



A retain cycle between the Car and Person classes

Memory Management (con...)

- The weak Attribute
 - The weak attribute creates a **non-owning relationship**
 - Possible to maintain a cyclical relationship **without creating a retain cycle**
 - Two objects should **never** have strong references to each other



A weak reference from the Person class to Car

Memory Management (con...)

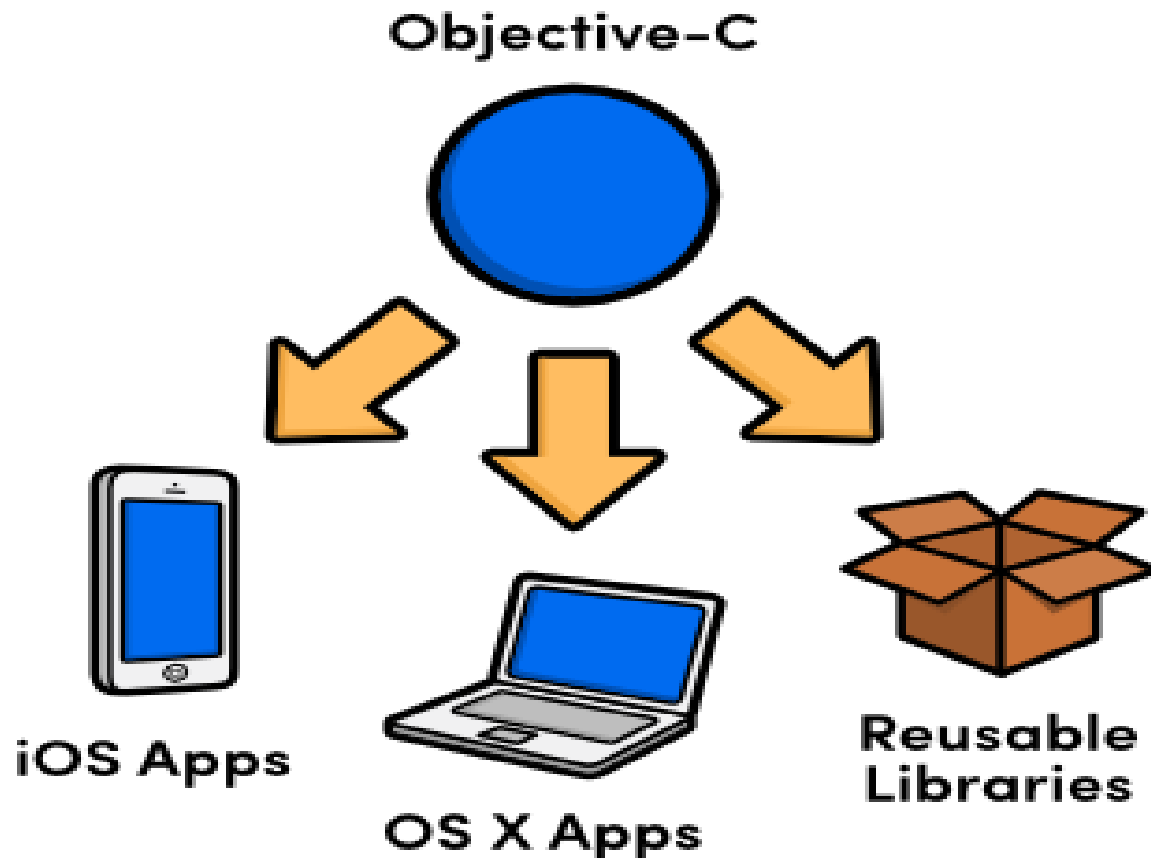
- The copy Attribute

- 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
- Properties that represent values are good candidates for copying
- Eg. @property (nonatomic, copy) NSString *model;

14. Required Tools

- Objective-C is the native programming language for **Apple's iOS and OS X operating systems**.
- It's a compiled, general-purpose language capable of building everything **from command line utilities to animated GUIs to domain-specific libraries**.
- It also **provides many tools** for maintaining large, scalable frameworks.

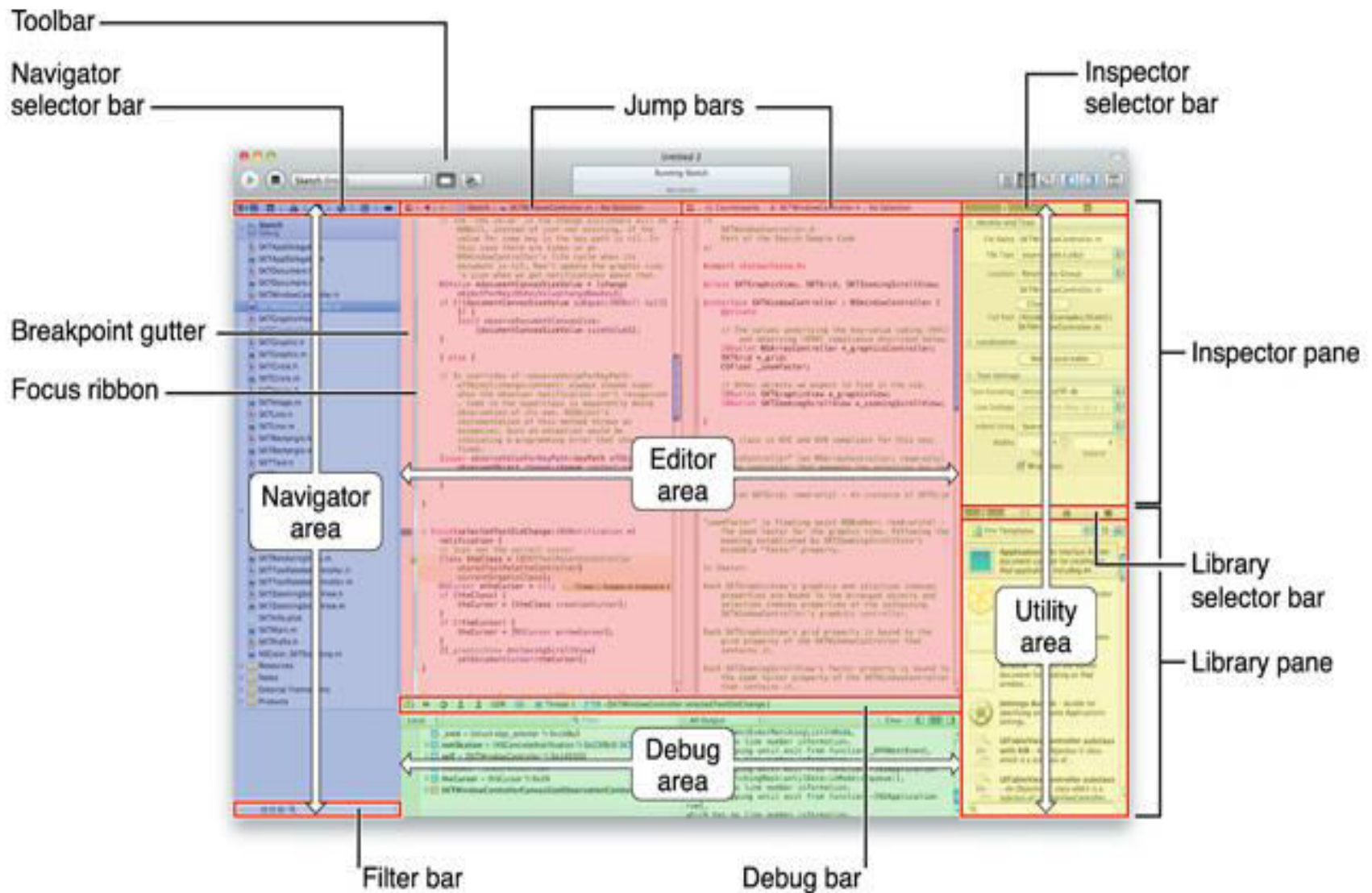
Types of programs written in Objective-C



14.1 Xcode

- Xcode is the Integrated Development Environment (IDE) designed for developing **iOS and Mac OS** apps
- The Xcode IDE includes **editors** used to design and implement your apps
- Xcode can **show you mistakes** in both syntax and logic, and even suggests fixes as you type
- Finally, **Build and run** your apps

Xcode (con...)



Components Of Xcode

- **Xcode IDE** : IDE that enables you to manage, edit, and debug your projects
- **Dashcode** : IDE that enables you to develop web-based iPhone and iPad applications and Dashboard widgets
- **iOS Simulator** : Provides a software simulator to simulate an iPhone or an iPad on your Mac
- **Interface Builder** : Visual editor for designing user interfaces for your iPhone and iPad applications
- **Instruments** : Analysis tool to help you both optimize your application and monitor for memory leaks in real time

14.1.1 Creating an Application

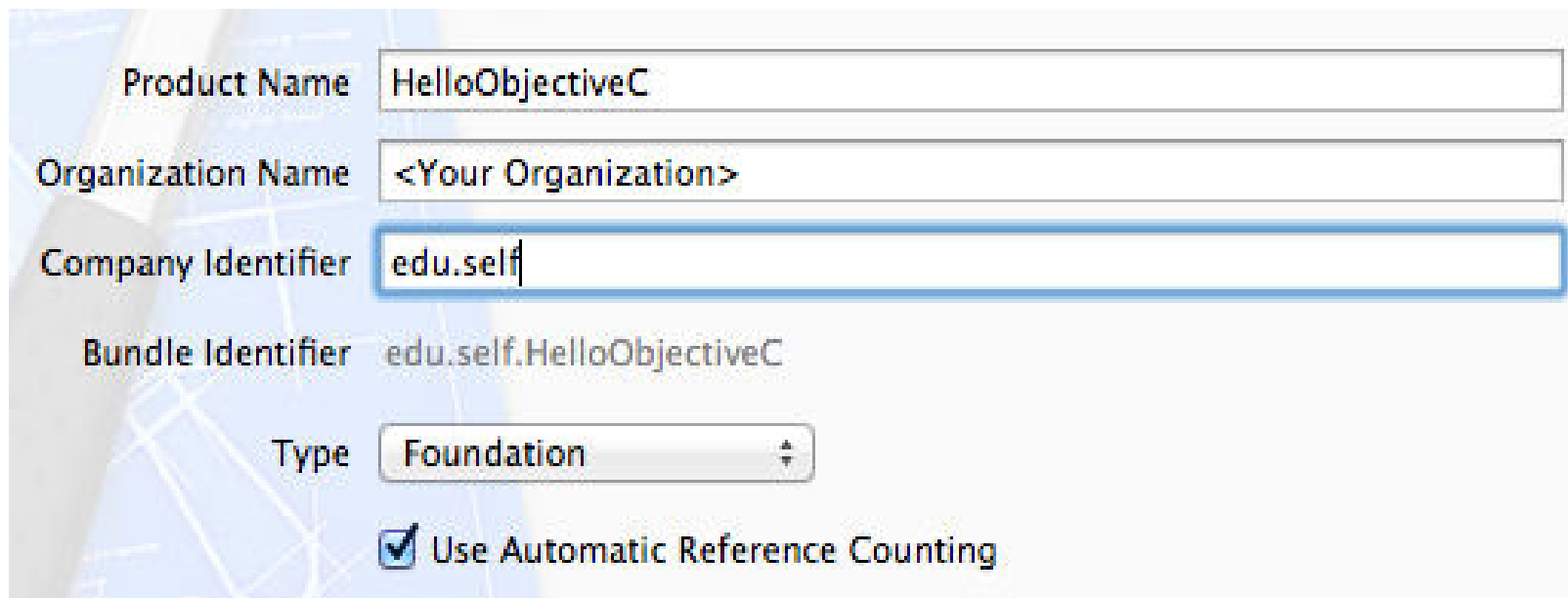
- Xcode provides **several templates** for various types of iOS and OS X applications. All of them can be found by navigating to File > New > Project... or using the Cmd+Shift+N shortcut.
- This will open a dialog window asking you to **select a template**:



Creating a command line application

14.1.2 Configuring a command line application

- This **opens another dialog** asking you to configure the project:



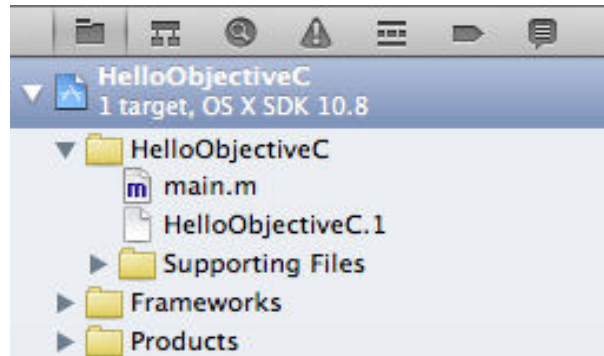
The screenshot shows a configuration dialog for a new project. The fields are as follows:

Product Name	HelloObjectiveC
Organization Name	<Your Organization>
Company Identifier	edu.self
Bundle Identifier	edu.self.HelloObjectiveC
Type	Foundation
<input checked="" type="checkbox"/> Use Automatic Reference Counting	

Configuring a command...(con...)

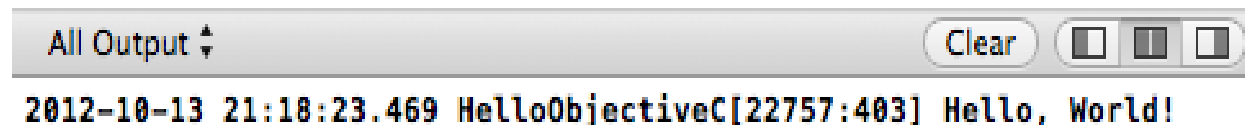
- You can use whatever you like for the **Product Name and Organization Name** fields.
- Finally, the Use **Automatic Reference Counting** checkbox should always be selected for new projects.
- Clicking Next prompts you for a **file path** to store the project (save it anywhere you like), and you should now have a brand new Xcode project to play with.
- In the left-hand column of the Xcode IDE, you'll find a file called **main.m** (along with some other files and folders). At the moment, this file contains the entirety of your application.
- Note that the **.m extension** is used for Objective-C source files.

Configuring a command...(con...)



main.m in the **Project Navigator**

- To compile the project, click the **Run button** in the upper-left corner of the IDE or use the Cmd+R shortcut.
- This should display Hello, World! in the **Output Panel** located at the bottom of the IDE:



14.1.3 The main() Function

```
#import <Foundation/Foundation.h>
int main(int argc, const char * argv[])
{
    @autoreleasepool
    {
        // insert code here...
        NSLog(@"Hello, World!");
    }
    return 0;
}
```

14.2 iOS Simulator

- The **iOS Simulator app**, available within Xcode, presents the iPhone, iPad, or Apple Watch user interface in a window on your Mac computer.
- You **interact with iOS Simulator** by using the keyboard and the mouse to emulate taps, device rotation, and other user actions.
- There are **two different ways to access iOS Simulator** through Xcode.
 - The first way is to **run your app in iOS Simulator**.
 - The second way is to **launch iOS Simulator without running an app**.

14.2.1 Running Your App in iOS Simulator

- When testing an app in iOS Simulator, it is easiest to **launch and run your app in iOS Simulator directly** from your Xcode project.
- To run your app in iOS Simulator, **choose an iOS simulator**—for example, iPhone 6 or iPad Air—from the Xcode scheme pop-up menu and click Run.
- Xcode builds your project and then launches the most recent version of your app **running in iOS Simulator on your Mac screen**

Running Your App...(con...)



14.3 Instruments

- Instruments is a **powerful and flexible performance-analysis and testing tool** that's part of the Xcode tool set.
- It's designed to help you **profile your OS X and iOS apps, processes, and devices** in order to better understand and optimize their behavior and performance.
- **Incorporating Instruments** into your workflow from the beginning of the app development process can save you time later by **helping you find issues early in the development cycle.**

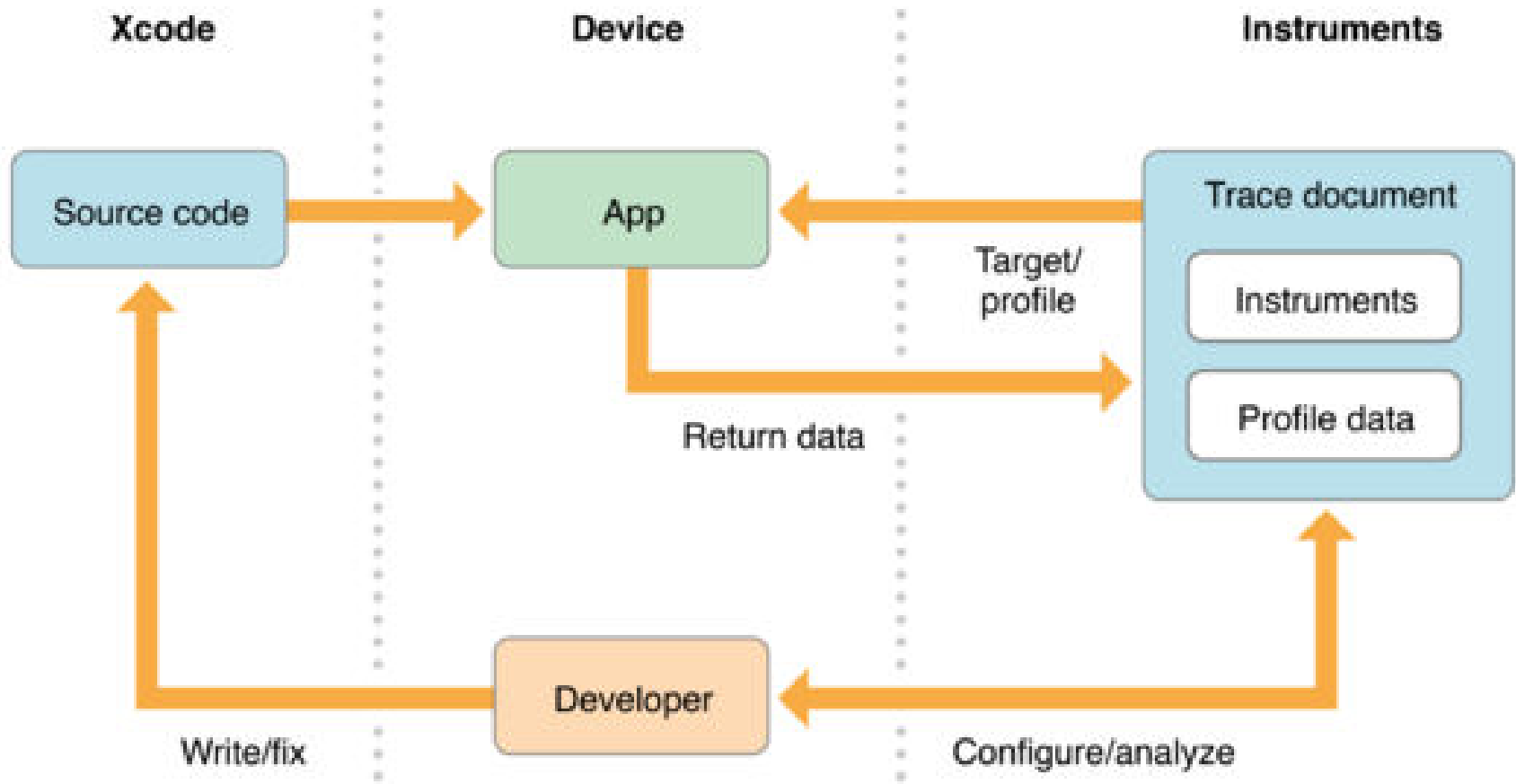
Instruments (con...)

- In Instruments, you use specialized tools, known as *instruments*, to trace different aspects of your apps, processes, and devices over time.
- Instruments collects data as it profiles, and presents the results to you in detail for analysis.
- By using Instruments effectively, you can:
 - Examine the behavior of one or more apps or processes
 - Examine device-specific features, such as Wi-Fi and Bluetooth
 - Perform profiling in a simulator or on a physical device

Instruments (con...)

- Create **custom DTrace instruments** to analyze aspects of system and app behavior
- **Track down problems** in your source code
- Conduct **performance analysis** on your app
- **Find memory problems in your app**, such as leaks, abandoned memory, and zombies
- **Identify ways to optimize** your app for greater power efficiency
- Perform general **system-level troubleshooting**
- **Automate testing of your iOS app** by running custom scripts to perform a sequence of user actions and replaying them to reliably reproduce those events and collect data over multiple runs
- **Save** instrument configurations as templates

14.3.1 The Instruments Workflow



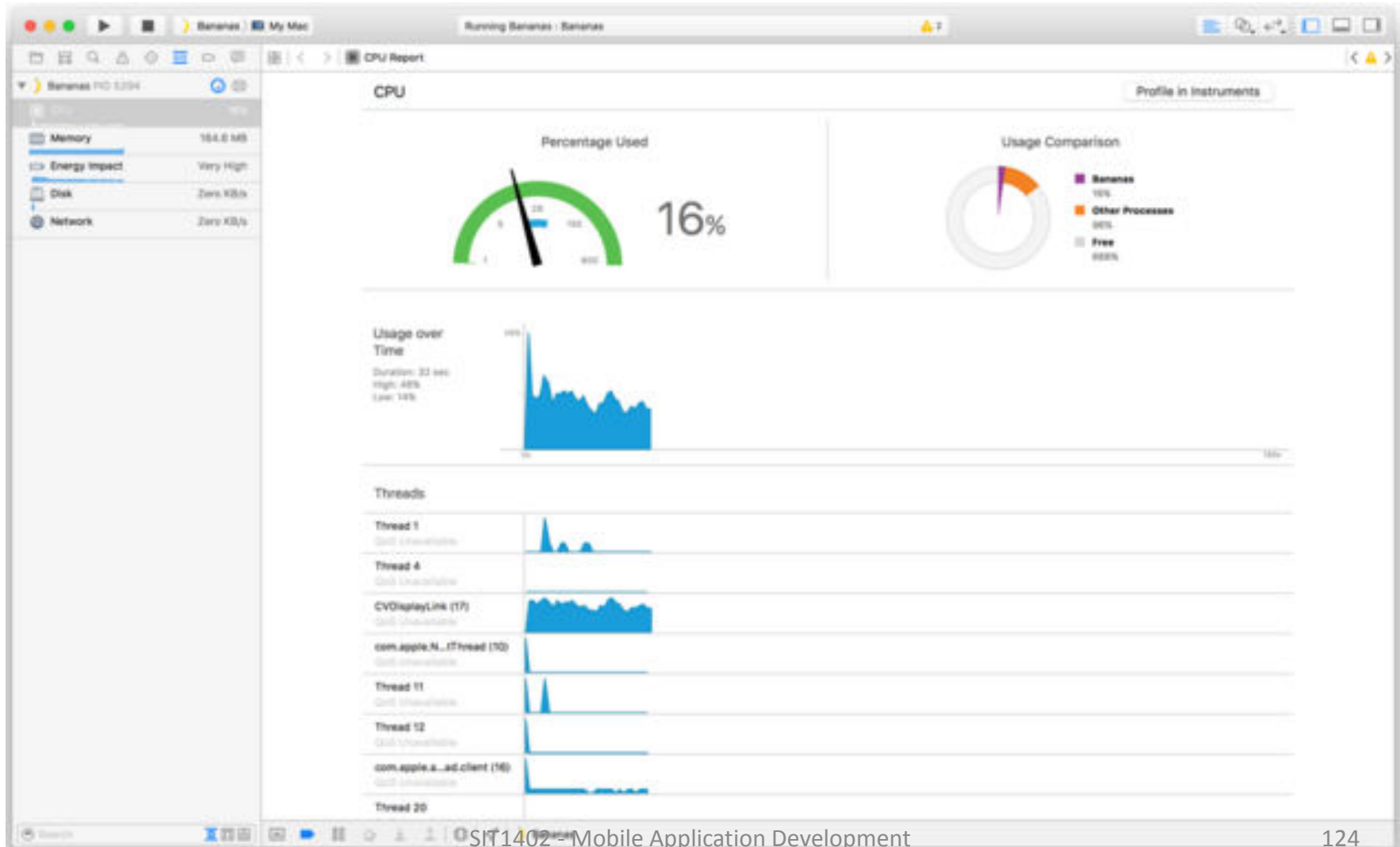
The Instruments Workflow (con...)

- It can be used to gather all kinds of useful information about your app, and help you diagnose and resolve problems.
- At a high level, it consists of the following main phases:
 - Set up a trace document containing the desired instruments and settings.
 - Target a device and an app to profile.
 - Profile the app.
 - Analyze the data captured during profiling.
 - Fix any problems in your source code.

14.3.2 Know When to Use Instruments

- While testing your app with Xcode, **consult the debug navigator gauges** (see Figure) before diving into Instruments.
- These **gauges provide high-level information** about your app's CPU, memory, energy usage, and more.
- Often, they **provide all the information you need** to improve performance and resolve common problems quickly.
- Use Instruments when you need to **perform more detailed analysis**

14.3.3 The CPU debugging gauge in Xcode



14.4 ARC

- Automatic Reference Counting (ARC) to track and manage your app's memory usage.
- Every time you create a new instance of a class, ARC allocates a chunk of memory to store information about that instance.
- This memory holds information about the type of the instance, together with the values of any stored properties associated with that instance.
- Additionally, when an instance is no longer needed, ARC frees up the memory used by that instance so that the memory can be used for other purposes instead.

ARC (con...)

- This ensures that class instances **do not take up space in memory** when they are no longer needed.
- **ARC were to deallocate an instance** that was still in use, it would no longer be possible to access that instance's properties, or call that instance's methods.
- Indeed, if you tried to access the instance, your app would most likely **crash**.
- **ARC tracks how many properties, constants, and variables are currently referring** to each class instance.

ARC (con...)

- ARC will not deallocate an instance as long as at least **one active reference** to that instance still exists.
- To make this possible, whenever you assign a class instance to a property, constant, or variable, that property, constant, or variable makes a **strong reference to the instance**.
- The reference is called a “**strong**” reference because **it keeps a firm hold on that instance**, and does not allow it to be deallocated for as long as that strong reference remains.

14.5 Framework

- A framework is a collection of resources; it collects a static library and its header files into a single structure that Xcode can easily incorporate into your projects.
- The Foundation framework defines a base layer of Objective-C classes.
- In addition to providing a set of useful primitive object classes, it introduces several paradigms that define functionality not covered by the Objective-C language.

Framework (con...)

- The **Foundation framework** is designed with these goals in mind:
 - Provide a small set of **basic utility classes**
 - Make **software development easier** by introducing consistent conventions for things such as deallocation
 - **Support** Unicode strings, object persistence, and object distribution
 - Provide a **level of OS independence** to enhance portability

Framework (con...)

- The framework was developed by **NeXTStep**, which was acquired by Apple and these foundation classes became part of Mac OS X and iOS.
- Since it was developed by NeXTStep, it has class prefix of "**NS**".
- We have used Foundation Framework in all our sample programs. It is almost a must to use **Foundation Framework**.

Framework (con...)

- Generally, we use something like **#import** **<Foundation/NSString.h>** to import a Objective-C class, but in order avoid importing too many classes, it's all imported in **#import** **<Foundation/Foundation.h>**.
- **NSObject** is the base class of all objects including the foundation kit classes. It provides the methods for memory management.
- It also **provides basic interface to the runtime system and ability to behave as Objective-C objects**. It doesn't have any base class and is the root for all classes.

Framework (con...)

- Foundation Classes based on functionality

Loop Type	Description
Data storage	NSArray , NSDictionary , and NSSet provide storage for Objective-C objects of any class.
Text and strings	NSCharacterSet represents various groupings of characters that are used by the NSString and NSScanner classes. The NSString classes represent text strings and provide methods for searching, combining, and comparing strings. An NSScanner object is used to scan numbers and words from an NSString object.

Framework (con...)

Dates and times	The NSDate , NSTimeZone , and NSCalendar classes store times and dates and represent calendrical information. They offer methods for calculating date and time differences. Together with NSLocale , they provide methods for displaying dates and times in many formats and for adjusting times and dates based on location in the world.
Exception handling	Exception handling is used to handle unexpected situations and it's offered in Objective-C with NSException .
File handling	File handling is done with the help of class NSFileManager .
URL loading system	A set of classes and protocols that provide access to common Internet protocols.

Practices

- To study various tools involved to develop an iOS app.
- Write a structure Objective-C program.
- Write a Objective-C program for finding given number is prime or not.
- Write a Objective-C program to reverse a given number.
- Explain various conditional branching and looping statements in Objective-C.
- Mention the exception handling mechanism in Objective-C
- Explain about memory management in Objective-C