Q4. Classes and Objects

- The main purpose of objective-C is to add object orientation to the C programming Language
- Classes are the central feature of objective-C, they support object-oriented programming, and are often called as user-defined types.
- The data and methods within a class are called members of the class.

Characteristics of Objective-C →

- The class is defined in two different sections namely @interface and @implementation
- Almost everything is in the form of objects
- Objects recieve messages, and are often referred to as recievers.
- Objects contain instance variables
- Objects and instance variables have scope
- Classes hide an object's implementation

Class Definitions →

- A class definition starts with the keyword @interface, followed by the interface (class) name.
- The class body is enclosed by a pair of curly braces.
- In objective-C, all classes are derived from the base class called NSObject.
  It is the superclass of all Objective-C classes
  It provides basic methods like memory allocation and initialization

example →

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

Allocating and initializing objects →

- A class provides the blueprints for objects, so basically objects are created from a class.

- We declare objects with the same sort of declaration that we use to declare variables of basic type

example →

```
Box  box1 = [[Box alloc] init] ; // create box1 object of
                                    the type Box
Box  box2 = [[Box alloc] init]; // create box2 object of
                                    the type Box
```

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

@implementation Box @synthesize height;
- (id) init
{
```

```
    self = [super init];
    length = 1.0;
    breadth = 1.0;
    return self;
}
-(double) volume
{
    return length* breadth * height;
}
@end

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;
}
```

Function →

- A function is a group of statements that together perform a task.

- A function declaration tells a compiler about the function's name, return type and parameters.

- The function definition provides the actual body of a function.
The function is called as a method

Method Declaration →

syntax →

```
-(return_type) function_name : (argument_Type 1)
    arguementName1 joiningArgument2 : (arguementType2)
    arguementName2 joining ArguementN: (arguementTypeN)
    arguementNameN ;
```

eg.

```
-(int) max : (int) num1 and Num2 : (int) num2 ;
```

Method Definition →

Syntax →

```
-(return_type) function_name : (argumentType1)
    arguementName1 joiningArgument2 : (arguementType2)
    arguementName2 joining ArguementN: (arguementTypeN)
    arguement NameN
{
    body of the function
}
```

example program →

```
#import <Foundation/Foundation.h>
@interface SampleClass : NSObject

//method declaration
-(int) max: (int) num1 andNum2: (int) num2;

@end

@implementation SampleClass

//method definition
-(int) max: (int) num1 andNum2: (int) num2
{
    int result;
    if (num1 > num2)
    result = num1;
    else
    result = num2;
    return result;
}
@end

int main()
{
    int a= 100;
    int b= 200;
    int ret;
    SampleClass *sample= [[SampleClass alloc]init];
    ret = [sample max: a andNum2: b];
    NSLog (@"Max val is: %d \n", ret);
    return 0;
}
```

Q3. Data types and Decision making

★ Classification of Data Types →

- Basic Types - They are arithmetic types and consist of the two types : integer types and floating point types.

- Void Type - The type specifier void indicates that no valve is available

- Derived Types - They include pointer types, array types, structure types , function types , union types.

- Enumerated types - They are again arithmetic types, and they are used to define variables that can only be assigned to certain discrete integer values throughout the program.

★ 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

- Floating point Types - float - 4 byte
  double - 8 byte
  long double - 10 byte

- Void Type - function returns as void
  function arguments as void

* 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 it's size and layout of the variable's memory (range of values that can be stored within that memory, and set of operations that can be applied to the variable)

- Basic variable types include → char, int, float, double, void.

eg.

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

int main()
{
// variable definition
  int a, b;
  int c;
```

```
float f;
// variable 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;
}
```

★ Constants →

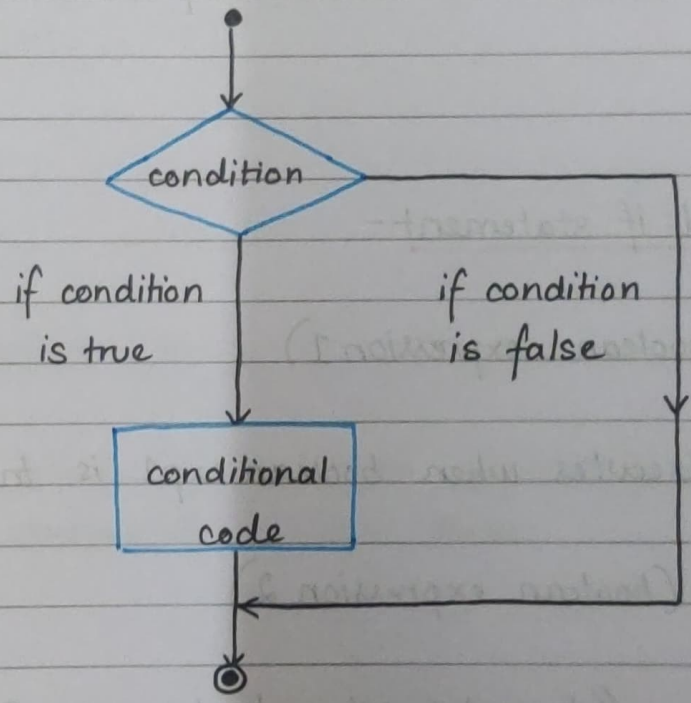
- Constants refer to fixed values that the program may not alter during it's execution
Constants can be of any of the basic data types, like integer constant, floating constant, character constant, enumeration constant, string literal etc.

★ Operators →

Operators include →
- → Arithmetic operators
- → Logical operators
- → Relational operators
- → Bitwise operators
- → Assignment operators
- → Miscellaneous operators
  - sizeof()
  - & operator
  - * operator
  - ?: operator

✯ Decision Making - Branching →

- Decision making structures require a programmers to specify one or more conditions to be evaluated and tested by the program, along with the statement/ 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.



✯ Types of decision making statements →

- ~~if-else-if~~ if statement -

```
if (boolean_expression)
{
   // statements will execute if the boolean exp is true
}
```

- **if else statement -**

```
if (boolean_expression)
{
   // statements will execute if boolean exp is true
}
else
{
   // statements will execute of boolean exp is false
}
```

- **Nested if statement -**

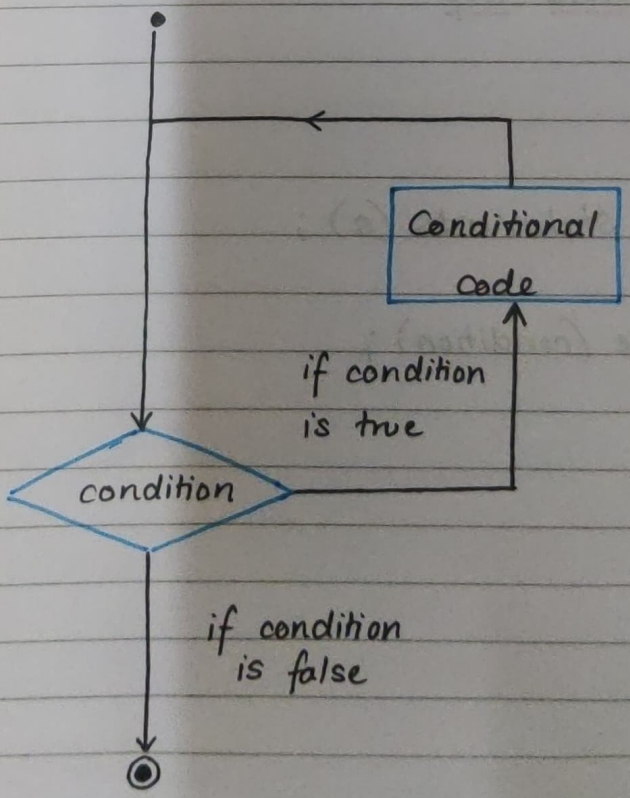
```
if (boolean_expression 1)
{
   // Executes when boolean exp1 is true

    if (boolean expression 2)
    {
        // Executes when boolean exp2 is true
    }

}
```

- **Switch statement -**

```
switch (expression)
{
    case constant_expression :
       statements (s) ;
       break ;
      //you can have any such no. of case statements
      default :
      // default is optional
       statements (s) ;
       break ;

}
```

✱ Decision Making - Looping →

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



★ Types of looping statements →

- **While loop -**

```
while (condition)
{
    statements (s);
}
```

- **For loop -**

```
for (init ; condition ; increment)
{
    statements (s);
}
```

- **Do while loop -**

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

Q2. 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
- Exceptions can be generated by hardware as well as software
- Exception handling is made available in Objective-C with foundation class NSException.
- Objective-C supports four compiler directives for exception handling →

@ try → This block tries to execute a set of statements

@ catch → This block tries to catch the exception in the try block

@ throw → throw exception if you find yourself in a situation that indicates a programming error, and you want to stop the application from running

@finally → This block contains set of statements that always execute.

- Examples of situations that can generate error include →
- division by zero
- underflow
- overflow
- calling undefined instruction.
- Example program for exception handling →

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

int main ()
{
    NSAutoreleasePool *pool = [[NSAutoreleasePool
                                alloc] init];

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

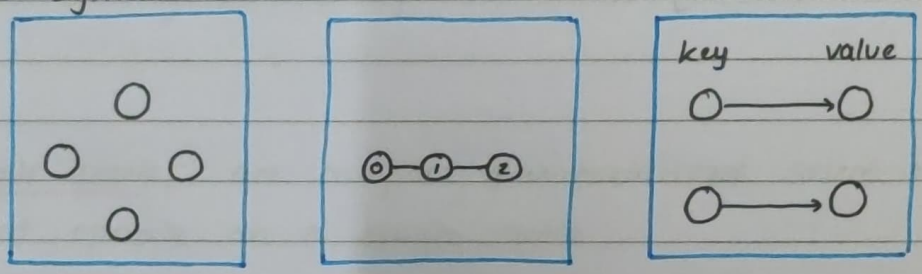
```
@try
{
    NSString *string = [array objectAtIndex : 10];
}

@catch (NSException *exception)
{
    NSLog (@"%@", exception.name);

    NSLog (@"Reason: %@", exception.reason);

}

@finally
{
    NSLog (@"@finally always Executes");
}

[pool drain];

return 0;

}
```

Unit - 3

Q1. NS array and NS dictionary

NSArray →

- NSArray is a 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



NSSet NSArray NSDictionary

eg.

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

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

Output → First german make : Mercedes-Benz

Example of NsArrayWithObjects →

```
NSArray *ukMakes = [NSArrayWithObjects :
    @"Lotus", @"Jaguar", @"Bentley", nil];

NSLog (@"First uk make: %@", [ukMakes
        objectAtIndex : 0]);
```

Output → First uk Make : Lotus

Fast Enumeration - NSArray →

- Fast - enumeration is the most efficient way to iterate over an NSArray
- The contents are guaranteed to appear in the correct order

```
eg. NSArray *germanMakes = @[@"Merceder - Benz",
    @"BMW", @"Porsche", @"Volkswagen",
    @"Audi"];

// With FastEnumeration
for (NSString *item in germanMakes)
{
    NSLog (@"%@", item);
}
// With Traditional for loop
for (int i=0 ; i< [germanMakes count] ; i++)
{
    NSLog (@"%@", germanMakes[i]);
}
```

- There are several advantages to Fast Enumeration →

  → It is considerably more efficient than traditional for

  → Syntax is more concise

  → Enumeration is "safe", as it has a mutation guard.
  If you attempt to modify the collection during enumeration, exception will be raised

NSDictionary →

- It represents an unordered collection of objects
- They associate each value with a key, which acts as a label for the value.
- NSDictionary is immutable, but the NSMutableDictionary data structure allows you to dynamically add and remove entries as necessary.
- NSDictionary can be defined using the literal syntax @{}

eg. // Literal syntax

```
NSDictionary * inventory = @{@"A" : [NSNumber
    numberWithInt : 1], @"B" : [NSNumber
    numberWithInt : 2], @"C" : [NSNumber
    numberWithInt : 3], @"D" : [NSNumber
    numberWithInt : 4]};
```

// Values and keys as arguements

```
NSDictionary * inventory = [NSDictionary dictionaryWithObjectsAndKeys:
    [NSNumber numberWithInt : 1], @"A",
    [NSNumber numberWithInt : 2], @"B",
    [NSNumber numberWithInt : 3], @"C",
    [NSNumber numberWithInt : 4], @"D", nil];
```

// Values and keys as arrays

```
NSArray * letters = @[@"A", @"B", @"C", @"D"];
NSArray * numbers = @[[NSNumber numberWithInt : 1],
    [NSNumber numberWithInt : 2], [NSNumber
    numberWithInt : 3], [NSNumber numberWithInt : 4]];
inventory = [NSDictionary dictionaryWithObjects : letters
                          forKeys : numbers];

NSLog (@"%@", inventory);
```

Fast Enumeration →

- Fast-enumeration is the most efficient way to enumerate a dictionary, it loops through the keys (not the values)

- NSDictionary also defines a count method which returns the no. of entries in the collection

eg. for (id key in inventory)
{
NSLog (@"%@ : %@", inventory[key], key);
}