CSIT242 Mobile Application Development

LECTURE 3 - JAVA & SWIFT SYNTAX AND MENUS, ALERTS, NOTIFICATIONS

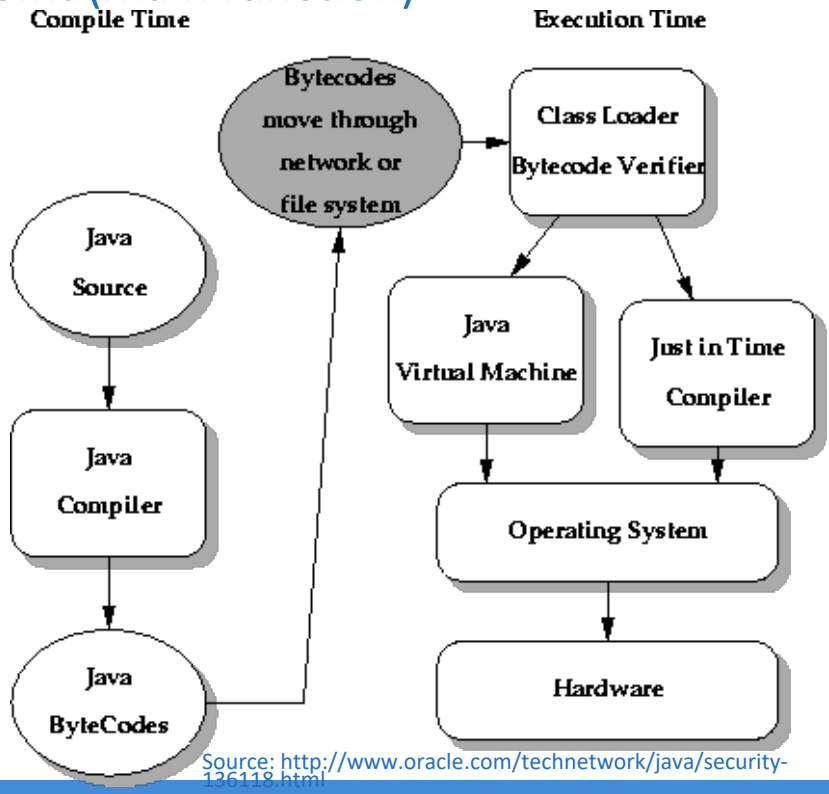
Outline

- Java & Swift Syntax
  - DVM, ART
  - Java
  - Swift Syntax
- Menus, Alerts, Notifications
  - Android – Styles & theme; AppBar; Menus; Dialogs; Toasts; Snackbar; Notifications; Fragments.
  - 2. iOS – Navigation Bars; Search Bars; Status Bar; Tab Bars; Toolbars; Action Sheets; Activity Views; Alerts; Popovers; Notifications.

Java & Swift Syntax

Java

- Java is Object Orientated Language (syntax similar to C and C++)
  - Compiles to a intermediate representation called Java bytecode
  - Java Virtual Machine (JVM) finds the desired entry point (main function)
  - The JVM performs following operation:
    - Loads the code
    - Verifies the code
    - Executes the code
    - Provides runtime environment
  - Just in time compiler interprets the bytecode in sections and compiles it on the fly



Source: http://www.oracle.com/technetwork/java/security-136118.html

For the Java Platform, Java Bytecode is executed by the Java Virtual Machine. The compiler converts source code written in the Java Programming Language into bytecode for the Java Virtual Machine.

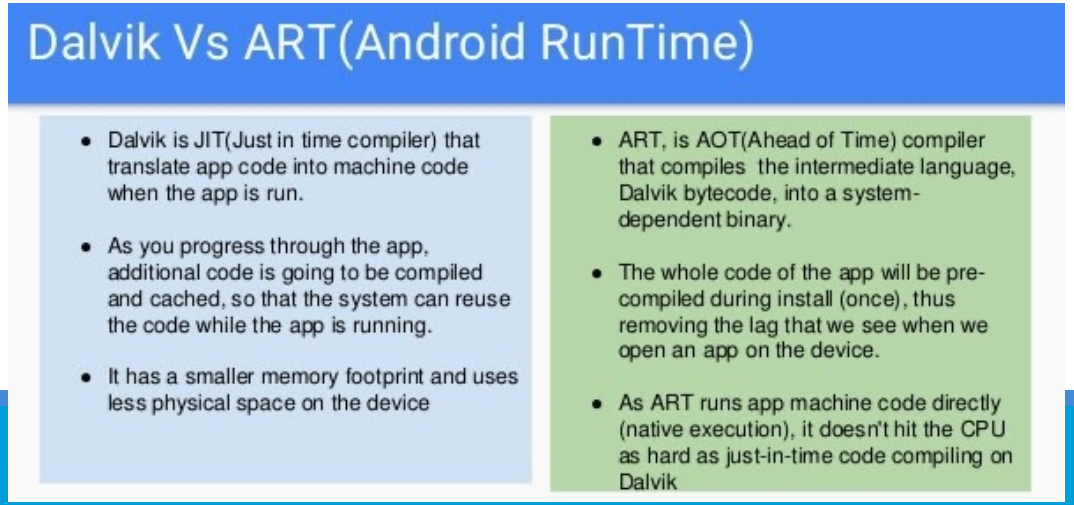
Java for Android

- For Android, the compiler converts source code written in the Java Programming Language into bytecode referred as DEX (Dalvik Executable) format
- Pre Android 4.4
  - The Dalvik Virtual Machine (DVM) is an android virtual machine optimized for mobile devices
    - It optimizes the virtual machine for memory, battery life and performance
    - Optimised for running multiple JVM processes simultaneously
    - Standard class libraries shared among all app processes, reducing per-process memory footprint
    - Allows for inter process calls (IPC) based on Android
  - Every time you launch an app, you spin an instance of the DVM

Android does not have a JVM. The JVM and DVM work in entirely different ways.
JVM is a stack-based system, DVM is a register based system.

Java for Android

- After Android 4.4
- ART (Android RunTime) is the next version of Dalvik
  - Development and debugging improvements:
    - Support for sampling profiler
    - Support for more debugging features
    - Improved diagnostic detail in exceptions and crash reports
- ART has two main features compared to Dalvik:
  - Ahead-of-Time (AOT) compilation, which improves speed (particularly startup time) and reduces the pressure on CPU (no JIT – Just-in-time compiling)
  - Improved Garbage Collection (GC)



Pre Android 4.4 - Dalvik Virtual Machine.
After Android 4.4 - Android Runtime (ART).

Java for Android

- Benefits of ART
  - Reduces startup time of applications as native code is directly executed
  - Improves battery performance as power utilized to interpret byte codes line by line is saved
  - There isn’t any on the fly compilation and native code storage - less RAM is required for application to run
- Drawbacks of ART
  - As dex bytecodes are converted to native machine code on installation itself, installation takes more time
  - As the native machine code generated on installation is stored in internal storage, more internal storage is required

Java

- Types
  - Primitive types: int, long, float
  - Object types: Object, String, Integer
- This
  - Used to pass a reference of itself or to identify class instance variables and functions

```
public void doSomething(){
        this.variable = 4;
        this.setVariable(4);
        SomeOtherClass.function(this);
}
```

Java

- Classes
  - A class is an abstraction defined by a programmer. Classes can be interpret as blueprints, or prototypes for objects
- Object
  - Objects are major software building blocks
  - Object are instantiated from the classes
  - All of Java's objects subclass from Object

```
public class Person extends Object{
    String firstName;
    private static final int age = 1;

    public void sayHello(){ …
    }
}
```

- Package
  - Allows a developer to group classes and interfaces together
  - Class members declared as protected are only accessible by members of the same package

Swift

- Swift is a powerful and intuitive programming language for macOS, iOS, watchOS and tvOS
- Swift is the result of the latest research on programming languages, combined with decades of experience building Apple platforms
  - Swift code is safe by design
  - Named parameters are expressed in a clean syntax that makes APIs in Swift even easier to read and maintain
  - Inferred types make code cleaner and less prone to mistakes, while modules eliminate headers and provide namespaces
  - Memory is managed automatically, without need to type semi-colons
- Swift has many other features that make the code more expressive:
  - Closures unified with function pointers
  - Tuples and multiple return values
  - Generics
  - Fast and concise iteration over a range or collection
  - Structs that support methods, extensions, and protocols
  - Functional programming patterns, e.g., map and filter
  - Native error handling

Swift

- Swift standard library
  - Fundamental data types (Int, Double, String)
  - Common data structures (Array, Dictionary, Set)
  - Common functions print(_:), sort(), abs(_:)
  - Many different protocols

Swift

- Value containers can be declared using
  - ***let*** for a constant (once assigned, value cannot be changed)
  - ***var*** for a variable (value can be changed)
- Swift variables are “ValueTypes”
- Class variables are “ReferenceTypes”

```
let nCourses : Int = 24
var markPercentage : Double = 70.5
------------------
let nCourses = 24
var markPercentage = 70.5
-------------
let nCourses = 24
var addNCourses = nCourses
addNCourses += 1
// nCourses is 24
// addNCourses is 25
```

Type safe programming language - once the data type of a variable has been identified, that variable cannot subsequently be used to store data of any other type.

Type inference allows the compiler to figure out the type!

Swift

- When a constant is declared without a type annotation it must be assigned a value at the point of declaration:

```
let courseName = “CSIT242-Mobile Apps Development”
```

- If a type annotation is used when the constant is declared, however, the value can be assigned later in the code. For example:

```
let courseName : String
..
courseName = "CSIT242-Mobile Apps Development"
```

Swift

- Strings
  - Strings can be constructed using combinations of strings, variables, constants, expressions, …
  - Swift has a straight forward String builder syntax
  - There is no implicit conversion to another type

```
let sentencePrefix = “Amount of students in the class is “
var amountOfStudents = 40
let sentence = sentencePrefix + String(amountOfStudents)
```

```
var students = 40
let sentence = “Amount of students in the class is \(students)“
```

escape sequences:

\n - New line	*\\ - Backslash*
\r - Carriage return	*\" - Double quote (used when placing a double quote into a string declaration)*
\t - Horizontal tab	*\' - Single quote (used when placing a single quote into a string declaration)*

Swift

- Swift equivalent to cout << or System.out.println() is ***print***

```
let stringVariable = “This subject is mobile programming”
print(stringVariable)
```

Swift

- Optional type
  - New concept (not included in C++ or Java)
  - Optional type provides a safe and consistent approach to handling situations where a variable or constant may not have any value assigned to it
  - You must explicitly declare in Swift if it is possible for an object to be “nothing”

```
//declares an optional Int variable named index
var index: Int?
```

  - The variable index can now
    - either have an integer value assigned to it, or
    - have nothing assigned to it
  - Behind the scenes, and as far as the compiler and runtime are concerned, an optional with no value assigned to it actually has a value of nil

Swift

- If an optional has a value assigned to it, that value is said to be “wrapped” within the optional.
- The value wrapped in an optional may be accessed using a concept referred to as forced unwrapping.
  - This means that the underlying value is extracted from the optional data type, by placing an exclamation mark (!) after the optional name.

```
var index: Int?
index = 3

print(“Value is “ + index) //error - Value of optional type ‘Int?’
not unwrapped. Did you mean to use ? or !

print(“Value is “ + index!) //the value of index has been unwrapped
//If index is nil - this will crash at runtime!!!
//‘!’ indicated that it is unsafe
```

Swift

- Optional keep the code safe and need to unwrap a variable to use it
  - best way is to use optional binding

```
var index : Int?
index = 3

if let ind = index {
        print (“value is \(ind)”)
}
```

  - The example performs two tasks:
    - the statement ascertains whether or not the designated optional contains a value,
    - in the event that the optional has a value, that value is assigned to the declared constant or variable and the code within the body of the statement is executed.

Swift

- It is also possible to declare an optional as being implicitly unwrapped
  - In this case the underlying value can be accessed without having to perform forced unwrapping or optional binding
- An optional is declared as being implicitly unwrapped by replacing the question mark (?) with an exclamation mark (!) in the declaration

```
var index : Int!


if index != nil {
        print (“value is \(index)”)
} else {
    print("index does not contain a value")
}
```

Only optional types are able to have no value or a value of nil assigned to them. Therefore, it is not possible to assign a nil value to a non-optional variable or constant.

var myInt = nil // Invalid code
var myString: String = nil // Invalid Code
let myConstant = nil // Invalid code

Swift

- For situations where the compiler is unable to identify the specific type of a value a concept referred to as type casting can be used

```
let myValue = record.object(forKey: "comment") as! String
```

  - Upcasting is performed using the as keyword and is also referred to as guaranteed conversion since the compiler can tell from the code that the cast will be successful

```
let myButton: UIButton = UIButton()
let myControl = myButton as UIControl
```

  - Downcasting, on the other hand, occurs when a conversion is made from one class to another where there is no guarantee that the cast can be made safely or that an invalid casting attempt will be caught by the compiler

```
let myScrollView: UIScrollView = UIScrollView()
let myTextView = myScrollView as! UITextView
```

Swift

- A safer approach to downcasting is to perform an optional binding using as?. If the conversion is performed successfully, an optional value of the specified type is returned, otherwise the optional value will be nil:

```
if let classB = classA as? UITextView {
    print("Type cast to UITextView succeeded")
} else {
    print("Type cast to UITextView failed")
}
```

- It is also possible to type check a value using the is keyword

```
if myobject is MyClass {
    // myobject is an instance of MyClass
}
```

Swift

- Operators (arithmetic & compound)

Operator	Description
- (unary)	Negates the value of a variable or expression
*	Multiplication
/	Division
+	Addition
-	Subtraction
%	Remainder/Modulo
x += y	Add x to y and place result in x
x -= y	Subtract y from x and place result in x
x *= y	Multiply x by y and place result in x
x /= y	Divide x by y and place result in x
x %= y	Perform Modulo on x and y and place result in x

Swift

- Operators (comparison & logical)

Operator	Description
x == y	Returns true if x is equal to y
x > y	Returns true if x is greater than y
x >= y	Returns true if x is greater than or equal to y
x < y	Returns true if x is less than y
x <= y	Returns true if x is less than or equal to y
x != y	Returns true if x is not equal to y
!	NOT - Inverts the current value of a Boolean variable
||	OR - Returns true if one of its two operands evaluates to true, otherwise it returns false.
&&	AND - Returns true only if both operands evaluate to be true

Swift

- Conditional statements - if

```
let student1 = 56
let student2 = 52
if student1 > student2 { //No () around condition required.
            print("student1 Wins!")
}
else if student2 > student1 { //Must have { } braces
            print("student2 Wins!")
}
else {
      print("It's a draw!")
}
```

Swift

- Ternary operator or conditional operator - provide a shortcut way of making decisions within code

```
let x = 10
let y = 20
print("Largest number is \(x > y ? x : y)")}
```

Swift

- Conditional statements - switch
  - No break statements - Values do not carry through cases
  - Case can be a result of an expression
  - Must cover all potential options

```
let car = “Mazda Mx5”
switch car {
            case “BMW”:
                let comment = “You have excellent taste in cars”
            case “Holden”, “Ford”:
                let comment = car + “ – good choice”
            case car.hasPrefix(“Mazda”):
                let comment = “Brilliant car, you should never change”
            default:
                let comment = “I don’t have an opinion about that car”
}
```

When we should use break? Why we use fall through?

Swift

- While & repeat statements

```
var n = 1

while n < 10 {
    n++
}
--------

repeat {
    n++
} while n < 10
```

Swift

- For statements

```
//for-in loop
let lecturers = [“Elena”, “Ben”, “Tom”];
for lecturer in lecturers {
        print(lecturer)
}

//what is the output?
let power = 10
var answer = 1
for _ in 1...power {
    answer *= 5
}
print(answer)
```

Swift

- Range operators
  - closed range operator - represents the range of numbers starting at x and ending at y where both x and y are included within the range

```
for index in 0...5 {
        print (“\(index) ”)          //prints 0 1 2 3 4 5
}
```

  - half-closed range operator - encompasses all the numbers from x up to, but not including, y

```
for index in 0..<5 {
        print (“\(index) ”)          //prints 0 1 2 3 4
}
```

Swift

- Range operators and switch

```
let temperature = 83
switch (temperature)
{
    case 0...49:
        print("Cold")
    case 50...79:
        print("Warm")
    case 80...110:
        print("Hot")
    default:
        print("Temperature out of range")
}
```

Swift

- break & continue

```
var j = 10

for i in 0 ..< 100
{
    j += i
    if j > 100 {
        break
    }
    print("j = \(j)")
}
```

```
var i = 1

while i < 20
{
    i += 1
    if (i % 2) != 0 {
        continue
    }
    print("i = \(i)")
}
```

Swift

- where
  - where statement may be used within a switch case match to add additional criteria required for a positive match

```
switch (temperature)
{
     case 0...49 where temperature % 2 == 0:
          print("Cold and even")
     case 50...79 where temperature % 2 == 0:
          print("Warm and even")
     default:
          print("Temperature out of range or odd")
}
```

  - where statement may be used to chain together multiple if lets and use the result

```
for index in 0…5 where index !=4 {
       print(“\(index)”)
}                                  //prints 0 1 2 3  5
```

Swift

- Guard statement
  - A guard statement contains a Boolean expression which must evaluate to true in order for the code located after the guard statement to be executed
  - The guard statement must include an else clause to be executed in the event that the expression evaluates to false
    - The code in the else clause must contain a statement to exit the current code flow

```
func multiplyByTen(value: Int?) {
    guard let number = value, number < 10 else {
        print("Number is too high")
        return
    }
    let result = number * 10
    print(result)
}
```

Unwrapped number variable is available to the code outside of the guard statement!

Swift

- Tuples - way to temporarily group together multiple values into a single entity.
  - The items stored in a tuple can be of any type and there are no restrictions requiring that those values all be of the same type

```
let error404 = (404, “Page not found”) //Type is (Int, String)
print(error404.1) //prints Page not found

//Explicitly declared tuple
let error404 = (statusCode:404, description:”Page not found”)
print(error404.description)

//Extract a tuple
let (statusCode, statusError) = error404
let (statusCode, _) = error404
```

Swift

- Arrays
  - When declaring an array, type is inferred
  - An empty array must have a type declared
  - Arrays contents can be of more than one type

```
var lecturers = [String]() //Empty String array
var lecturers: [String] = [“Elena”, “Ben”];
var lecturers = [“Elena”, “Ben”, 40];
```

```
var lecturers = [“Elena”, “Ben”];
lecturers.append(“Luke”)
lecturers += [“Tom”, “Sam”, “Chris”]
if lecturers.isEmpty {
    print(“Array is empty”)
} else {
    let itemCount = lecturers.count
    print(itemCount)
}
```

Swift

- Dictionaries are a collection of key:value pairs
  - Similar to other variables, type is inferred when creating a dictionary
  - An empty dictionary must have a type declared
  - Dictionary contents can be of more than one type

```
var airports: [String: String] = ["YYZ":"Toronto Pearson", "DUB":"Dublin"]
---

var airports = ["YYZ": "Toronto Pearson", "DUB": "Dublin"]
---

airports["LHR"] = "London" // Use a new key of the appropriate type as the
                           // subscript index, and assign a new value
---
for (airportCode, airportName) in airports {
    print("\(airportCode): \(airportName)")
}
```

Swift

- Function is a named block of code that can be called upon to perform a specific task
  - It can be provided data on which to perform the task and is capable of returning results to the code that called it.

```
//Function without parameter
func orderFood(){ … }
----

//Function with String parameter
func orderFood(item : String) { … }

-----
//Function with String parameter and returns a Double
func orderFood(item : String) -> Double { … }
```

- A method is essentially a function that is associated with a particular class, structure or enumeration

Swift

- Calling a function
  - Similar format to other languages

```
//calling function without parameter
orderFood()
----

//calling function with String parameter
orderFood(item:“Pizza”)

-----
//calling function with String parameter and returns a Double
let amount =  orderFood(item:“Pizza”)
```

Swift

- Functions with multiple parameters

```
//Function with same local and external parameters names
func sayHello(name: String, count: Int) {
    print("Hello \(name), you are number \(count)")}

//function call
sayHello(name:"Elena", count:1)
```

```
//Function with different local and external parameters names
func sayHello(uName name: String, uCount count: Int){
    print("Hello \(name), you are number \(count)")}

//function call
sayHello(uName:"Elena", uCount:1)
```

Swift

- Functions with multiple parameters

```
//Function with removed external parameters names
func sayHello(_ name: String, _ count: Int) {
    print("Hello \(name), you are number \(count)")}

//function call
sayHello("Elena", 1)
```

```
//Function with default parameters
func sayHello(_ name: String = "customer", count: Int){
    print("Hello \(name), you are number \(count)")}

//function call
sayHello(count: 1)
```

Swift

- Returning multiple results from a function

```
func minMax(array: [Int]) -> (min: Int, max: Int) {
    …
    return (currentMin, currentMax)
}
// function call
let bounds = minMax(array: [8, -6, 2, 109, 3, 71])
print("min is \(bounds.min) and max is \(bounds.max)")
```

- In order to make any changes made to a parameter persist after the function has returned, the parameter needs to be declared as *inout*

```
//Function with inout parameter
func changeValue(_ value: inout Int) -> Int{
    value+=value
    return (value)
}
//function call
Print("returned value is \(changeValue(&myValue))")
```

Swift

- Variable Numbers of Function Parameters
  - If the number of function's parameters are not known in advance, Swift offers possibility of using variadic parameters
  - Variadic parameters - declared by three periods (...) - indicate that the function accepts zero or more parameters of a specified data type

```
func displayNum(_ numbers: String...){
    for num in numbers {
        print(string)
    }
}
---
displayNum("one", "two", "three", "four")
```

Swift

- Functions as parameters and in scope

```
//assign a function to a constant (or variable)
func getPrice(_ item:String) -> Double {
    …
}

let myPriceFun = getPrice
let result =  myPriceFun (“Cheeseburger”)
```

```
//Functions can live inside of functions
func outputMessage(_ message: String) {

    func addGreeting() {
        print("Hello! \(message)")
    }
    addGreeting()
}
---
//function call
outputMessage("Jack")
```

Swift

- Functions as Parameters (example)

```
func inchesToFeet (_ inches: Float) -> Float {
    return inches * 0.0833333
}
func inchesToYards (_ inches: Float) -> Float {
    return inches * 0.0277778
}
let toFeet = inchesToFeet
let toYards = inchesToYards

func outputConversion(_ converterFunc: (Float) -> Float, value: Float) {
    let result = converterFunc(value)
    print("Result of conversion is \(result)")
}

outputConversion(toYards, value: 10) // Convert to Yards
outputConversion(toFeet, value: 10) // Convert to Inches
```

Swift

- A closure generally refers to the combination of a self-contained block of code and one or more variables that exist in the context surrounding that code block
- Three forms:
  - Global functions are closures that have a name and do not capture any values
  - Nested functions are closures that have a name and can capture values from their enclosing function
  - Closure expressions are unnamed closures written in a lightweight syntax that can capture values from their surrounding context
- Closure expressions are a way to write inline closures in a brief syntax. There are several syntax optimizations for writing closures in shorten form (by omission of the “in”, parameter types, parentheses, …)

Closure is a block of code that can be a assign to a variable!

Swift

- Closure - examples

```
//closure expression syntax
{ (parameters) -> returntype in
    statements
}
```

```
let multiply = {(_ val1:Int, _ val2:Int) -> Int in
    return val1*val2
}
---
let result = multiply(10, 20)
```

```
func fA() -> () -> Int{
    var count = 0
    func fB() -> Int{
        return count+10
    }
    return fB
}
---
let myClosure = fA()
let result = myClosure()
```

```
func fA() -> (Int) -> Int{
        var count = 0
        return { (_ x: Int) -> Int in
            return count+x
        }
}
---
let myClosure = fA()
let result = myClosure(5)
print(result)
//print(myClosure(5))
```

Swift

- Classes and objects

What is instance method?

What is type method?

What is initializer?

```
class BankAccount {
    var accountBalance: Float = 0
    var accountNumber: Int = 0

    init(number: Int, balance: Float)
    {
        accountNumber = number
        accountBalance = balance
    }
    deinit {
        // Perform any necessary clean up here
    }
    func displayBalance()
    {
        print("Number \(accountNumber)")
        print("Current balance is \(accountBalance)")
    }
}
---
//creating an instance of a class
var account1 = BankAccount(number: 12312312, balance: 400.54)
Account1.displaybalance()
```

Swift

- Getters and setters (stored and computed properties)

```
class BankAccount {
    var accountBalance: Float = 0
    var accountNumber: Int = 0
    let fees: Float = 25.0
    …
    var balanceLessFees: Float {
        get {
            return accountBalance - fees
        }
        set(newBalance)
        {
            accountBalance = newBalance - fees
        }
    }
    …
}
…
var balance1 = account1.balanceLessFees
account1.balanceLessFees = 12123.12
```

Swift

- using self

```
class MyClass {
    var myNumber = 10 // class property

    func addTen(myNumber: Int) {
        print(myNumber) // Output the function parameter value
        print(self.myNumber) // Output the class property value
    }
}
```

Swift

- Subclass
  - Single inheritance

```
class SavingsAccount: BankAccount {
    var interestRate: Float = 0.0

    func calculateInterest() -> Float
    {
        return interestRate * accountBalance
    }
}
```

Swift

- Overriding inherited methods

```
class SavingsAccount: BankAccount {
    var interestRate: Float

    init(number: Int, balance: Float, rate: Float)
    {
        interestRate = rate
        super.init(number: number, balance: balance)
    }   // init method of the superclass must be called after
                                        // the initialization task for the subclass
    func calculateInterest() -> Float
    {
        return interestRate * accountBalance
    }

    override func displayBalance()
    {
        print("Number \(accountNumber)")
        print("Current balance is \(accountBalance)")
        print("Prevailing interest rate is \(interestRate)")
    }   // first two lines can be eliminated by super.displayBalance()
}
```

Swift

- Extension can be used to add feature such as methods, initializers, computed properties and subscripts to an existing class without need to create and reference a subclass.

```
extension Double{
    var squared: Double{
        return self*self
    }
}
```

Swift

- Swift uses Automatic Reference Counting (ARC) to track and manage app's memory usage
  - ARC automatically frees up the memory used by class instances when those instances are no longer needed
- 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
- A weak reference is a reference that does not keep a strong hold on the instance it refers to, and so does not stop ARC from disposing of the referenced instance

Swift

- Remarks for the labs:
  - In Objective-C, a selector is a type that refers to the name of an Objective-C method
  - In Swift, Objective-C selectors are represented by the Selector structure, and are created using the #selector expression
    - Place the name of the method within the #selector expression: #selector(myMethod(…))
  - @objc - expose a method to Objective-C
    - Swift generates code that is only available to other Swift code, but for all of UIKit the interaction in runtime is with the Objective-C
    - @objc attribute instructs Swift to make those things available to Objective-C as well as Swift code

Menus, Alerts, Notifications

Android – Styles and Themes

- A style is a collection of properties that specify the look and format for a View or window
  - A style can specify properties such as height, padding, font colour, font size, background colour, and much more
- A style is defined in an XML resource that is separate from the XML that specifies the layout
- Styles in Android share a similar philosophy to cascading stylesheets in web design - they allow you to separate the design from the content
- A theme is a style applied to an entire Activity or application, rather than an individual View
  - When a style is applied as a theme, every View in the Activity or application will apply each style property that it supports
  - For example, you can apply the same style as a theme for an Activity and then all text inside that Activity will have same style

Android – Styles and Themes

```
<TextView
        android:id="@+id/myTextView"
        android:layout_width="215dp"
        android:layout_height="35dp"
        android:text="@string/textStr"
        android:textAlignment="center"
        style="@style/CustomText"
/>
```

Parent attribute is omitted

In style.xml file

```
----
<style name="CustomText">
        <item name="android:textSize">40sp</item>
        <item name="android:textColor">#00FF00</item>
 </style>
```

```
<manifest …>
  <application
       …
       android:theme = "@style/Theme.App"…>
       …
  </application>
  …
</manifest>
```

```
<manifest …>
  <activity
       …
       android:theme = "@style/Theme.App"…>
       …
  </activity>
  …
</manifest>
```

Please note: Styles and themes are declared in a style resource file in res/values/, usually named styles.xml. In newer versions of Android Studio the xml file is named themes.xml

Android – Styles and Themes

- Defining the Style:
  - Create `style.xml` file (if it is missing)
  - For each style add a `<style>` element to the file with a name that uniquely identifies the style (this attribute is required)
  - Add an `<item>` element for each property of that style, with a name that declares the style property and a value to go with it (this attribute is required)
    - The value for the `<item>` can be a keyword string, a hex color, a reference to another resource type, or other value depending on the style property
- There are two ways to set a style:
  - To an individual View, by adding the style attribute to a View element in the (XML) layout
  - To an entire Activity or application, by adding the `android:theme` attribute to the `<activity>` or `<application>` element in the Android manifest

Please note: Styles and themes are declared in a style resource file in res/values/, usually named styles.xml. In newer versions of Android Studio the xml file is named themes.xml

Android - Menus

- A menu is a set of options presented to the user
  - Menus are a common user interface component in many types of applications
- From Android 3.0 (API level 11), Android apps provide an app bar to present common user actions
- Three fundamental types of menus or action presentations :
  - **Options menu and action bar** - The options menu is the primary collection of menu items for an activity, where can be placed actions that have a global impact on the app (for example: “Search”, “Compose email”, "Settings”)
  - **Context menu and contextual action mode** - A context menu is a floating menu that appears when the user performs a long-click on an element; It provides actions that affect the selected content or context frame
  - **Popup menu** - A popup menu displays a list of items in a vertical list that's anchored to the view that invoked the menu; It is good for providing an overflow of actions that relate to specific content or to provide options for a second part of a command

Android - Menus

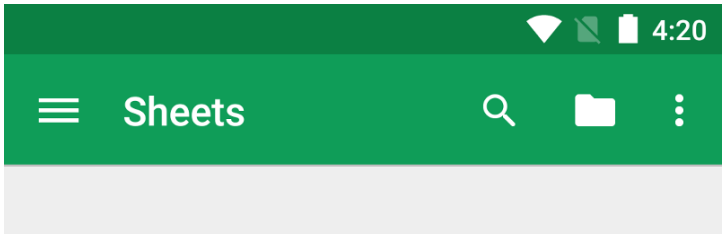
- Android provides a standard XML format to define menu items (for all menu types)
  - a menu and all its items need to be defined in an XML menu resource;
  - after, the menu resource needs to be inflated in the activity or fragment (load as a Menu object) using *MenuInflater.inflate()* method
- The *[menu].xml* file should be placed inside the project's `res/menu/` directory

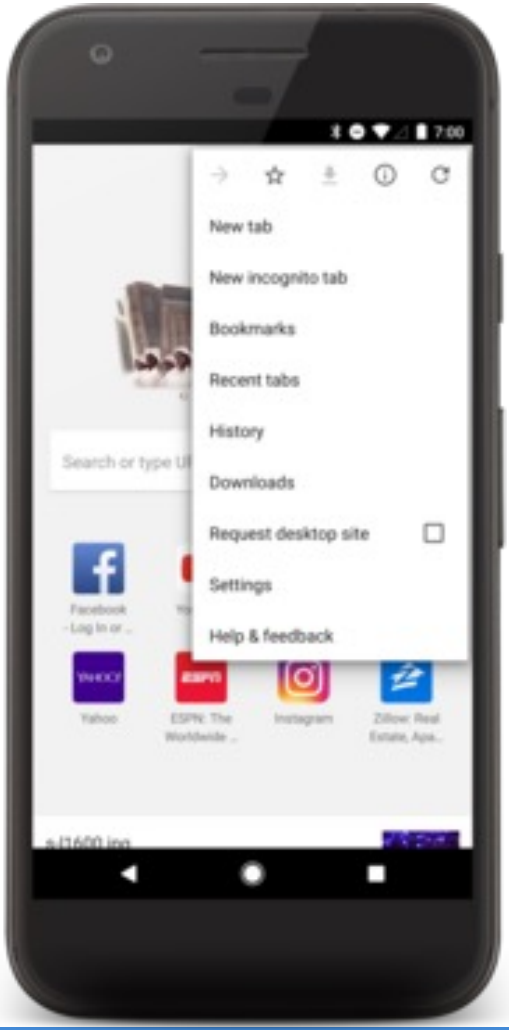
```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">
        <item android:id="@+id/item_1"
            android:icon="@drawable/ic_item"
            android:title="New Item"
            app:showAsAction="always" />
        <item android:id="@+id/help"
             android:icon="@drawable/ic_help"
            android:title="HELP"
             app:showAsAction= "ifRoom"/>
</menu>
```

This structure allows creating a submenus as well.

Android – Options menu

- The options menu includes actions and other options that are relevant to the current activity context
  - For Android 2.3.x (API level 10) or lower, when the user presses the Menu button the contents of the options menu appear at the top of the screen
  - For Android 3.0 (API level 11) and higher, items from the options menu are available in the app bar



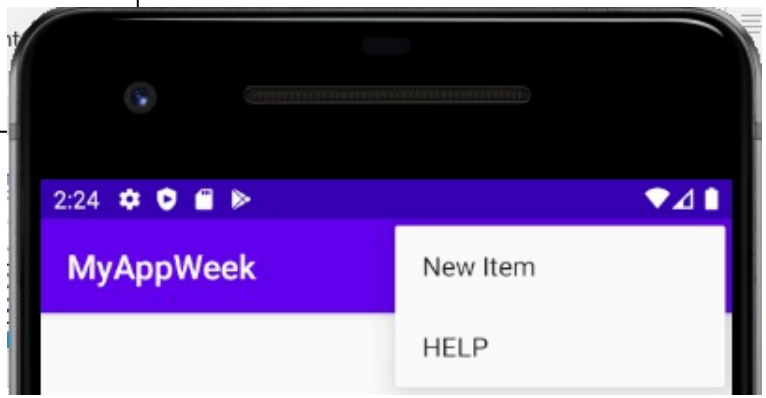


Android – Options menu

- To specify the options menu for an activity, `onCreateOptionsMenu()` method should be overridden
- When the user selects an item from the options menu (including action items in the app bar), the system calls activity's method `onOptionsItemSelected()`
  - This method passes the MenuItem selected. The item can be identified by calling `getItemId()` (returns the unique ID for the menu item)

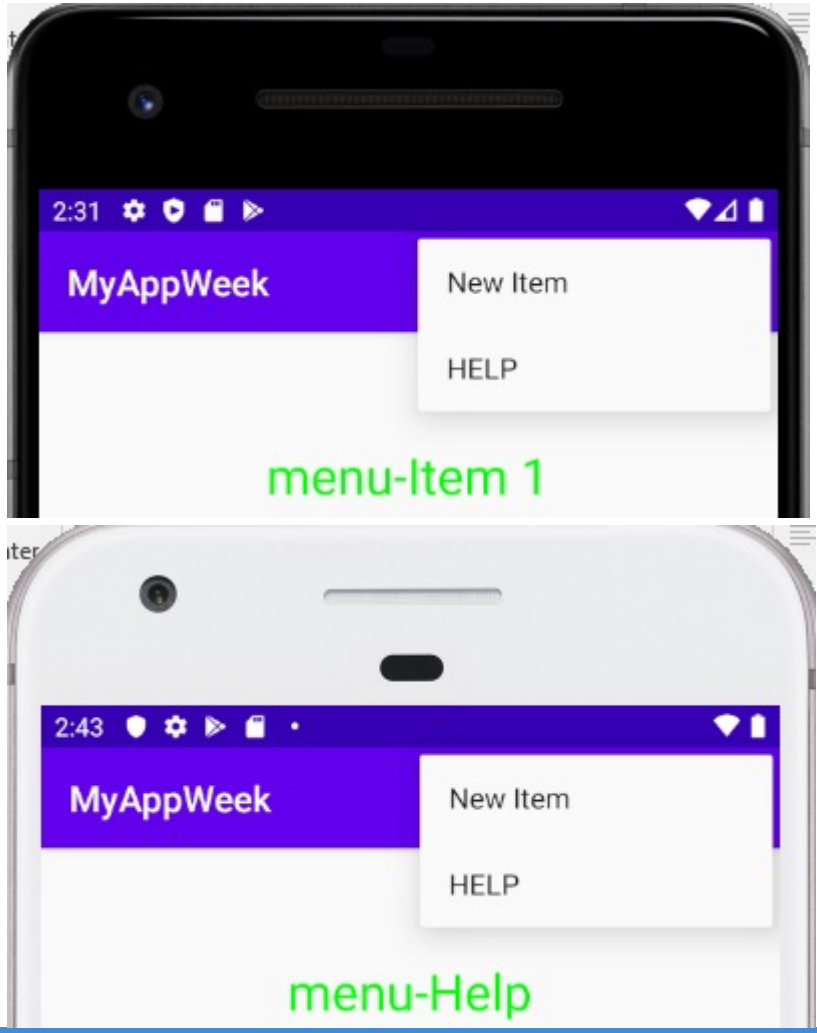
```
<?xml version="1.0" encoding="utf-8"?>
<menu  xmlns:android="http://schemas.android.com/apk/res/android">
        <item android:id="@+id/item_1"
              android:title="New item" />
        <item android:id="@+id/help"
            android:title="Help" />
</menu>
```

```
@Override
public boolean onCreateOptionsMenu(Menu mymenu) {
        MenuInflater inflater = getMenuInflater();
        inflater.inflate(R.menu.my_menu, mymenu);
        return true;
}
```



Android – Options menu

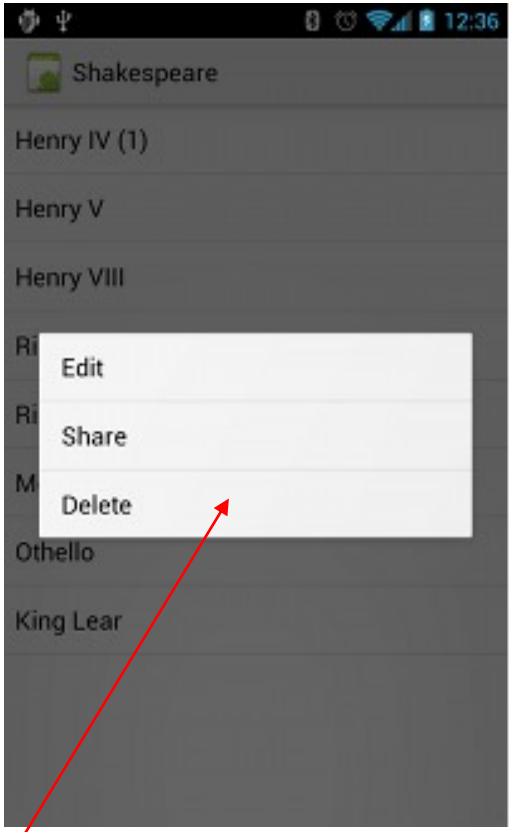
```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
        // Handle item selection
        switch (item.getItemId()) {
            case R.id.item_1:
                statusText.setText("menu-Item 1");
                return true;
                    // When you successfully handle
                    // a menu item, return true.
            case R.id.help:
                statusText.setText("menu-Help");
                return true;
            default:
                return super.onOptionsItemSelected(item);
                // If you don't handle the menu item,
                // you should call the superclass
                // implementation of onOptionsItemSelected()-
                // the default implementation returns false
        }
    }
```



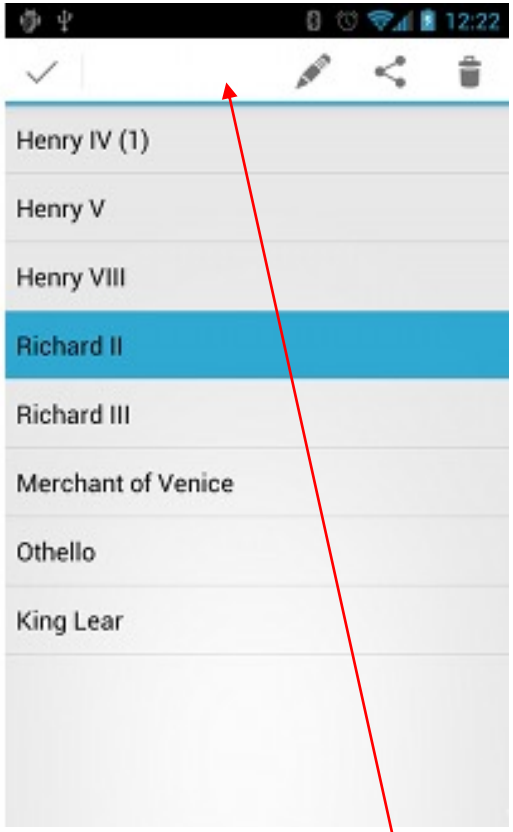
Android - Contextual Menus

- A contextual menu offers actions that affect a specific item or context frame in the UI
- Context menu can be provided for any view
  - Context menu are most often used for items in a ListView, GridView, or other view collections in which the user can perform direct actions on each item
- Two ways to provide contextual actions:
  - In a floating context menu - A menu appears as a floating list of menu items (similar to a dialog) when the user performs a long-click (press and hold) on a view that declares support for a context menu. Users can perform a contextual action on one item at a time
  - In the contextual action mode - This mode is a system implementation of ActionMode that displays a contextual action bar at the top of the screen with action items that affect the selected item(s); When this mode is active, users can perform an action on multiple items at once (if the app allows it)
- For this type of menu,
  - *View* to which the context menu should be associated needs to be registered by calling `registerForContextMenu()` and pass it the View
  - the methods `onCreateContextMenu(...)` and `onContextItemSelected(…)` should be overridden

Android - Contextual Menus



floating contextual menu

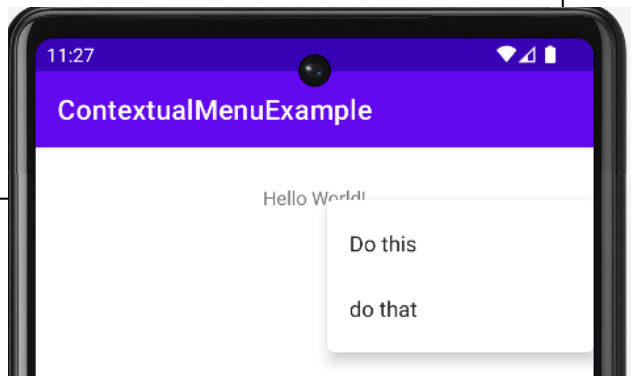


contextual action bar

Source: https://google-developer-training.github.io/android-developer-fundamentals-course-concepts-v2/unit-2-user-experience/lesson-4-user-interaction/4-3-c-menus-and-pickers/4-3-c-menus-and-pickers.html

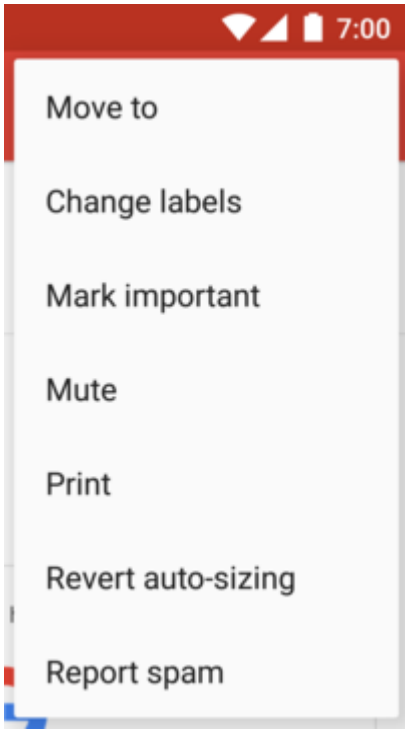
Android - Contextual Menus

```
@Override
    protected void onCreate(Bundle savedInstanceState) {
        . . .
        TextView textView = findViewById(R.id.textview);
        registerForContextMenu(textView);
        . . .
    }

    @Override
    public void onCreateContextMenu(ContextMenu menu, View v, ContextMenu.ContextMenuInfo menuInfo) {
        super.onCreateContextMenu(menu, v, menuInfo);
        MenuInflater inflater = getMenuInflater();
        inflater.inflate(R.menu.context_menu, menu);

    }

    @Override
    public boolean onContextItemSelected(@NonNull MenuItem item) {
        . . .
    }
```



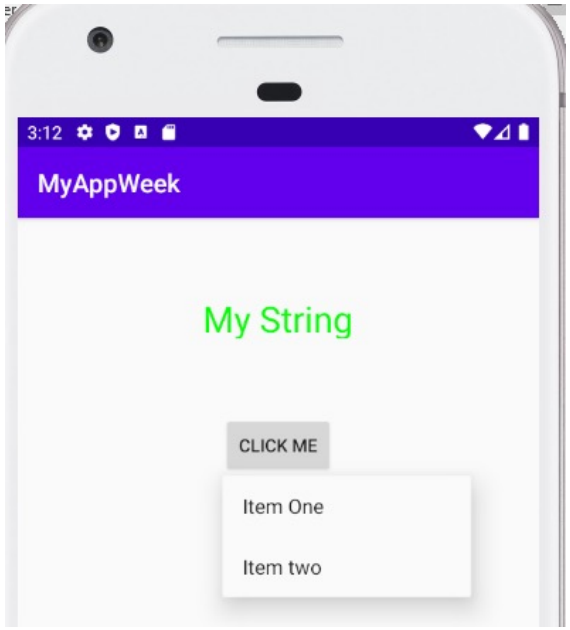
Android - Popup Menu

- A PopupMenu is a modal menu anchored to a View. It appears below the anchor view if there is room, or above the view otherwise
- A PopupMenu :
  - Provide an overflow-style menu for actions that relate to specific content (for example Gmail's email headers)
  - Provide a second part of a command sentence (such as a button marked "Add" that produces a popup menu with different "Add" options)
  - Provide a drop-down similar to Spinner that does not retain a persistent selection
- For this menu the class `PopupMenu` should be used



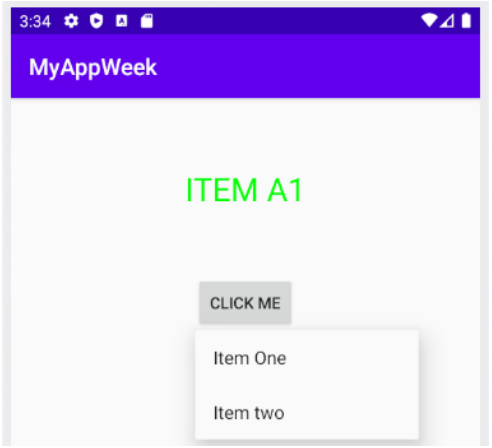
Android - Popup Menu

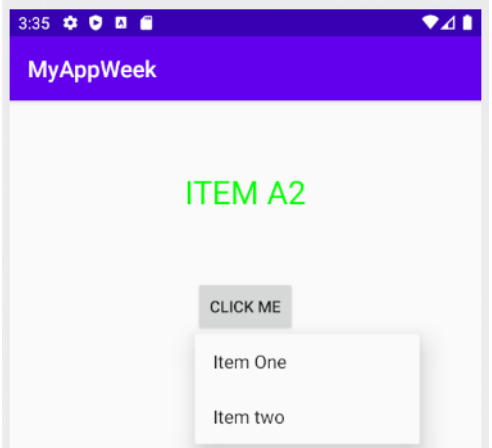
```
 button.setOnClickListener(
     new Button.OnClickListener() {
         public void onClick(View v) {
               showPopupMenu(v);
         }
});
---
public void showPopupMenu(View v){
        PopupMenu popup = new PopupMenu(this, v);
        MenuInflater inflater = popup.getMenuInflater();
        inflater.inflate(R.menu.popup_menu, popup.getMenu());
        popup.show();
}
```



Android - Popup Menu

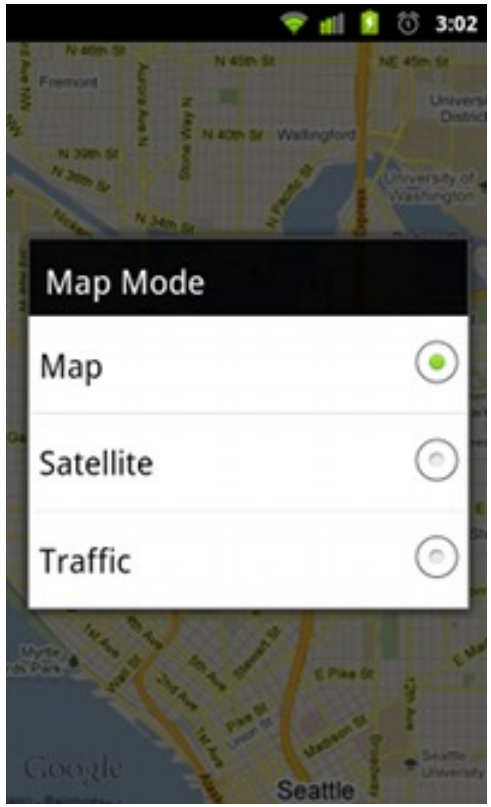
```
public void showPopupMenu(View v){

        PopupMenu popup = new PopupMenu(this, v);

        MenuInflater inflater = popup.getMenuInflater();
        inflater.inflate(R.menu.popup_menu, popup.getMenu());

        popup.setOnMenuItemClickListener(new PopupMenu.OnMenuItemClickListener() {
            public boolean onMenuItemClick(MenuItem item) {
                switch (item.getItemId()) {
                    case R.id.itemA1:
                        statusText.setText("ITEM A1");
                        return true;
                    case R.id.itemA2:
                        statusText.setText("ITEM A2");
                        return true;
                    default:
                        return false;
                }
            }
        });
        popup.show();
}
```





Android - Menu Groups

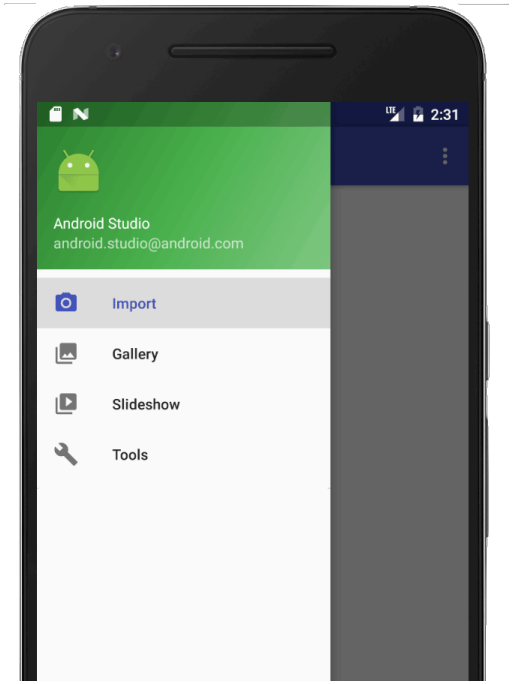
- A menu group is a collection of menu items that share certain traits, allowing:
  - Show or hide all items with `setGroupVisible()`
  - Enable or disable all items with `setGroupEnabled()`
  - Specify whether all items are checkable with `setGroupCheckable()`
- A group can be created by nesting `<item>` elements inside a `<group>` element in the menu resource or by specifying a group ID with the `add()` method




```
<?xml version="1.0" encoding="utf-8"?>
<menu … />
 …
    <!-- menu group -->
    <group android:id="@+id/group_1">
        <item android:id="@+id/item_1"
              android:title= "one" />
        <item android:id="@+id/item_2"
              android:title= "two" />
    </group>
</menu>
```

Android - Navigation Drawer

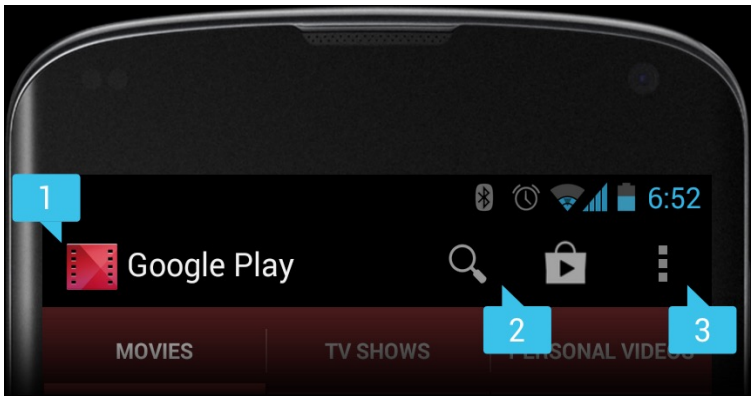
- The navigation drawer is a UI panel that shows the app's main navigation menu
- The drawer appears when the user touches the drawer icon in the app bar or when the user swipes a finger from the left edge of the screen
  - To add a navigation drawer, the layout should be declared with a `DrawerLayout` as the root view
  - Inside the `DrawerLayout`, layout for the main content for the UI (the primary layout when the drawer is hidden) should be added and another view that contains the contents of the navigation drawer

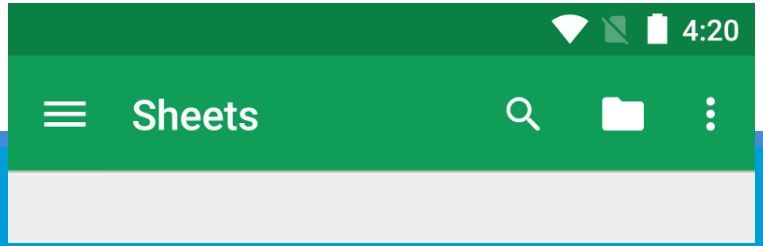


Source: https://developer.android.com/guide/navigation/navigation-ui

Android – Action Bar

- The action bar (also known as the app bar) is a window feature that identifies the user location, and provides user actions and navigation modes
- It is one of the most important design elements in the app's activities, because it provides a visual structure and interactive elements
- Using the action bar offers users a familiar interface across applications that the system gracefully adapts for different screen configurations
- An action bar can includes:
  - app icon (app identity)
  - important actions items
  - action overflow





Android – Action Bar

- The action bar provides several key functions:
  - Provides a dedicated space for giving your app an identity and indicating the user's location in the app
  - Makes important actions prominent and accessible in a predictable way (such as Search)
  - Supports consistent navigation and view switching within apps (with tabs or drop-down lists)
- Beginning with Android 3.0 (API level 11), all activities that use the default theme have an ActionBar as an app bar
- The native ActionBar behaves differently depending on what version of the Android system a device may be using

Android – Action Bar

- There are a lot of properties and methods available in the *ActionBar class*
  - Hiding the action bar at runtime can be done by calling `hide()`

```
ActionBar actionBar = getActionBar();
                    //getSupportActionBar();
actionBar.hide();
```

  - When the action bar hides, the system adjusts your layout to fill the screen space now available. Bringing the action bar back can be done by calling `show()`

Be certain you import the ActionBar class (and related APIs) from the appropriate package:
If supporting API levels lower than 11:

```
import android.support.v7.app.ActionBar;
```

If supporting API level 11 and higher:

```
import android.app.ActionBar;
```

If supporting API level 28 and higher:

```
import androidx.appcompat.app.ActionBar;
```

Android – Toolbar

- A Toolbar is a generalization of action bars for use within application layouts
- A Toolbar may contain a combination of different optional elements like navigation button, logo, title and subtitle, custom or menu view…
- A Toolbar may be placed at any arbitrary level of nesting within a view hierarchy
  - An application may choose to designate a Toolbar as the action bar for an Activity using the `setActionBar()/setSupportActionBar()` method
- The most recent app bar features have been added to the Toolbar class

Android – Toolbar

- Adding a Toolbar includes:
  - Adding the AndroidX Support Library (`androidx.appcompat.widget.Toolbar`) as a dependency
  - Make sure that activity extends the `AppCompatActivity` class
  - Remove the existing app bar (change the app style in e.g. `styles.xml` file)

```
<style name="AppTheme" parent="Theme.AppCompat.Light.NoActionBar">
```

  - Add a toolbar to the layout
    - The toolbar can be added in the layout file, or can be defined as a separate layout

```
<androidx.appcompat.widget.Toolbar
        android:id="@+id/toolbar"
  …
/>
```

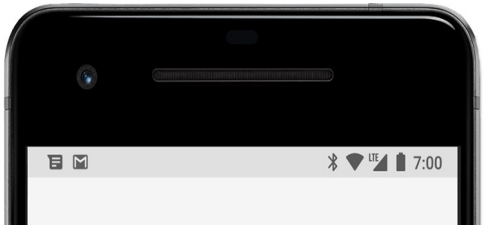
```
<?xml version="1.0" encoding="utf-8"?>
<androidx.appcompat.widget.Toolbar
xmlns:android="http://schemas.android.com/apk/res/android"
 . . . >
</androidx.appcompat.widget.Toolbar>
---
<include
 layout="@layout/toolbar_main"
 android:id="@+id/toolbar" />
```

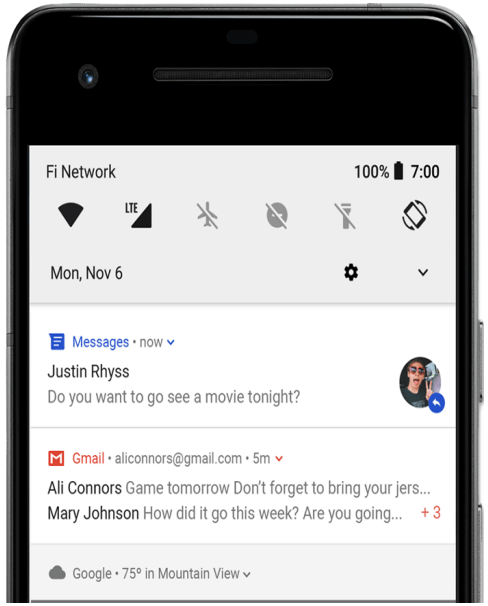
  - Update the activity to set the toolbar as the activity's app bar (in `onCreate(…)` method)

```
Toolbar toolbar = (Toolbar) findViewById(R.id.my_toolbar);
setSupportActionBar(toolbar);
```

Android – Notifications

- Notifications provide a way for an app to convey a message to the user when the app is either not running or is currently in the background
- Notifications can be categorized as:
  - **Local** notification - triggered by the app itself on the device on which it is running
  - **Remote** notifications - initiated by a remote server and delivered to the device for presentation to the user
- Notifications appear to users in different locations and formats:
  - icon in the status bar
  - detailed entry in the notification drawer (which can include actions such as a button to open the app that sent the notification)
  - badge on the app's icon
  - sound/icon on paired wearables
- Android 7 has introduced Direct Reply - the user can type in and submit a response to a notification from within the notification panel
- Android 8 (and newer) will display any pending notifications of the app when user long press the app's icon

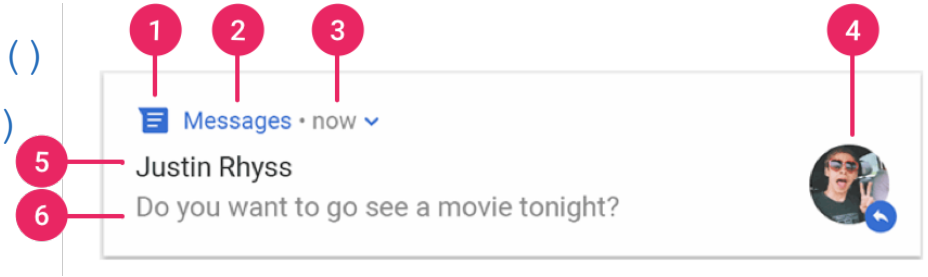




Source: https://developer.android.com/guide/topics/ui/notifiers/notifications

Android – Notifications

- A basic design of the notification includes:
  1. Small icon: required - set with `setSmallIcon()`
  2. App name: provided by the system
  3. Time stamp: provided by the system but can be overriden with `setWhen()` or hiden with `setShowWhen(false)`
  4. Large icon: optional - set with `setLargeIcon()`
  5. Title: optional - set with `setContentTitle()`
  6. Text: optional - set with `setContentText()`



- Expandable notification can include larger text area, image, in inbox style, a chat conversation, or media playback controls
- Beside using the Android notification templates, creating custom notification layout is also supported

Source: https://developer.android.com/guide/topics/ui/notifiers/notifications

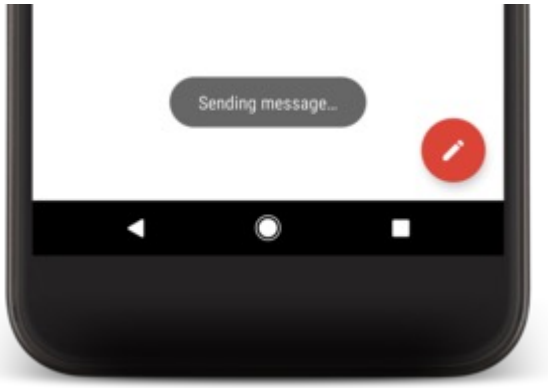
Android – Notifications

- Separate notifications can be grouped into notification group (available on Android 7.0 and higher).
- Starting in Android 8.0 (API level 26), all notifications must be assigned to a channel (or will not appear). This allows users to
  - disable specific notification channels for the app (instead of disabling all app's notifications)
  - control the importance, visual and auditory options for each channel (using Android system settings)
- Android uses the importance (urgent, high, medium, low) of a notification to determine how much the notification should interrupt the user (visually and audibly)
  - The higher the importance of a notification, the more interruptive the notification will be
- Importance of a notification is determined by the importance of the channel the notification was posted to (Android 8+)
  - On Android 7.1 (API level 25) and below, importance of each notification is determined by the notification's priority

Android – Toasts

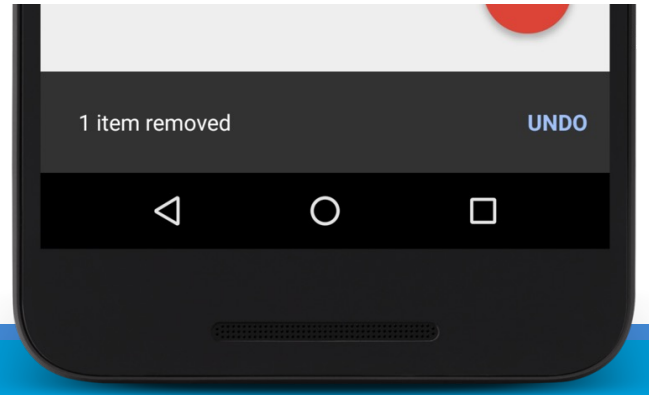
- A toast provides simple feedback about an operation in a small popup
  - It only fills the amount of space required for the message, leaving the current activity visible and interactive
- Toasts automatically disappear after a timeout
- Programmicaly can be changed the position of toast's appearance, or customize the toast's layout

```
Context context = getApplicationContext();
CharSequence text = "Sending message…";
int duration = Toast.LENGTH_SHORT; //Toast.LENGTH_LONG

Toast toast = Toast.makeText(context, text, duration);
toast.show();
```



Android – SnackBar

- The Snackbar component provides a way to present the user with information - lightweight feedback about an operation
  - It appears in the form of a panel at the bottom of the screen
- Snackbar instances contain a brief text message and an optional action button (that perform a task when tapped by the user)
- A Snackbar can
  - timeout automatically or
  - be removed manually by the user via a swiping action
- The app continues to function and respond to user interactions during the appearance of the Snackbar



Source: https://developer.android.com/training/snackbar/action

Android – SnackBar

- A Snackbar is normally used to indicate something that you want the user to see, but that's not critical for them to they see

```
Snackbar.make(view, message, length).show();
```

- It is common to set an Action with the Snackbar
  - This is not done using the normal `setOnClickListener()`, but using `setAction()`, whose second argument is the familiar `OnClickListener`

```
Snackbar mySnBar = Snackbar.make(view, “this is snackbar”, Snackbar.LENGTH_LONG);
mySnBar.setAction("Tap Me!", new MySnListener());
mySnBar.show();
---
public class MySnListener implements View.OnClickListener {
   @Override
   public void onClick(View v) {       statusText.setText("Snackbar");   }
}
```

Android – Dialogs

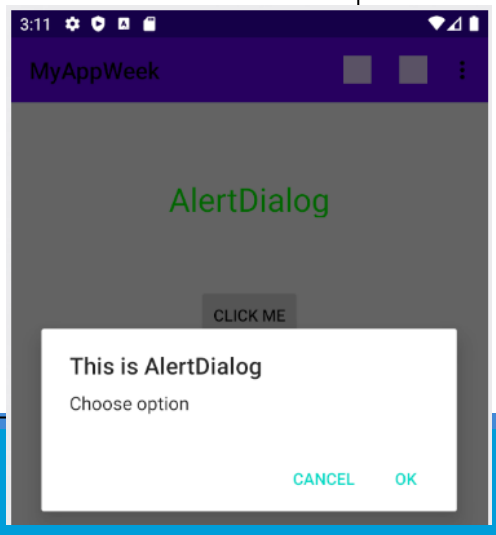
- A dialog is a small window that prompts the user to make a decision or enter additional information
  - A dialog does not fill the screen and is normally used for modal events that require users to take an action before proceeding
- The `AlertDialog` class allows building a variety of dialog designs
- There are three regions of an alert dialog:

1. Title (optional) - should be used only when the content area is occupied by a detailed message
2. Content area - display a message, a list, or other custom layout
3. Action buttons - no more than three action buttons in a dialog

- Other classes (with pre-defined UI) that allows user to select date or time are: `DatePickerDialog` and `TimePickerDialog`

Android – Simple Alert Dialog

```
// 1. Instantiate object of AlertDialog.Builder
        AlertDialog.Builder aBuilder = new AlertDialog.Builder(MainActivity.this);

// 2. Chain together various setter methods to set the dialog characteristics
        aBuilder.setTitle("This is AlertDialog");
        aBuilder.setMessage("Choose option");

// 3. Add the buttons
        aBuilder.setPositiveButton("OK", new DialogInterface.OnClickListener() {
            public void onClick(DialogInterface dialog, int id) {
                //executable code if user clicked OK
            }
        });
        aBuilder.setNegativeButton("Cancel", new DialogInterface.OnClickListener() {
            public void onClick(DialogInterface dialog, int id) {
                //executable code if user clicked CANCEL
            }
        });
// 4. Set other dialog properties (if any)

// 5. create and display AlertDialog
        AlertDialog dialog = aBuilder.create();
        dialog.show();
```

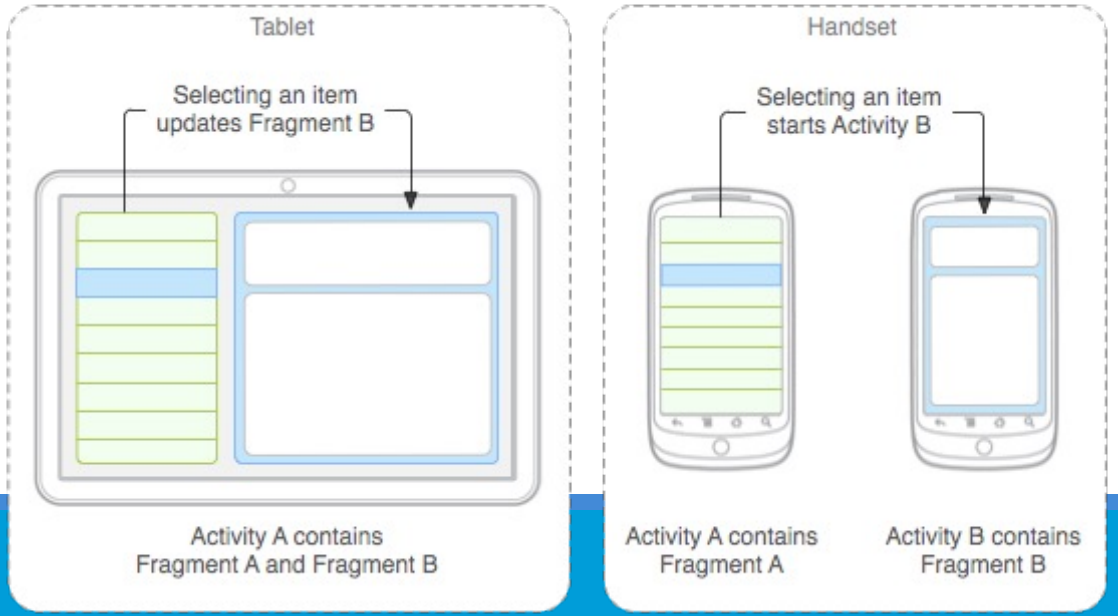


Android – Fragments

- A fragment is a self-contained, modular section of an application's user interface and corresponding behaviour that can be embedded within an activity
- A fragment is usually used as part of an activity's user interface and contributes its own layout to the activity
- Multiple fragments can be combined in a single activity to build a multi-pane UI and reuse a fragment in multiple activities
  - fragment is a modular section of an activity
  - has its own lifecycle
  - receives its own input events, and
  - can be added or removed while the activity is running (sort of like a "sub activity" that can be reused in different activities)
- Fragment's lifecycle is directly affected by the host activity's lifecycle
  - For example, when the activity is paused, so are all fragments in it, and when the activity is destroyed, so are all fragments

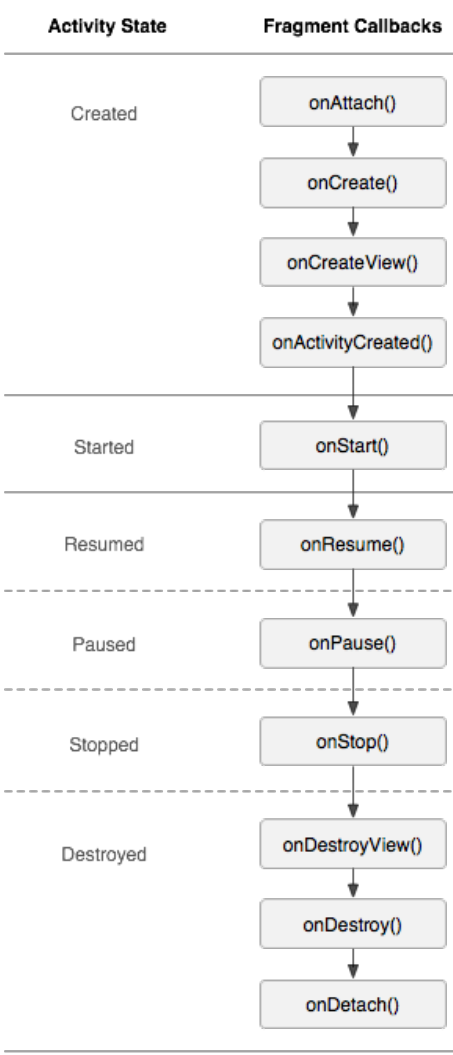
Android – Fragments

- Fragments are stored in the form of XML layout files and may be added to an activity by
  - placing appropriate `<fragment>` elements in the activity's layout file, or
  - directly through code within the activity's class implementation
- When a fragment is added as a part of activity layout, it lives in a `ViewGroup` inside the activity's view hierarchy and the fragment defines its own view layout
- To manage the fragments in the activity an instance of `FragmentManager` class needs to be used



Android – Fragments

- Managing the lifecycle of a fragment is a lot like managing the lifecycle of an activity - a fragment can exist in three states:
  - Resumed - The fragment is visible in the running activity
  - Paused - Another activity is in the foreground and has focus, but the activity in which this fragment lives is still visible
  - Stopped - The fragment is not visible; Either the host activity has been stopped or the fragment has been removed from the activity but added to the back stack
- The most significant difference in lifecycle between an activity and a fragment is how one is stored in its respective back stack
  - An activity is placed into a back stack of activities that is managed by the system when it is stopped
  - A fragment is placed into a back stack managed by the host activity only with explicit request for the instance to be saved (by calling `addToBackStack()` during a transaction that removes the fragment)



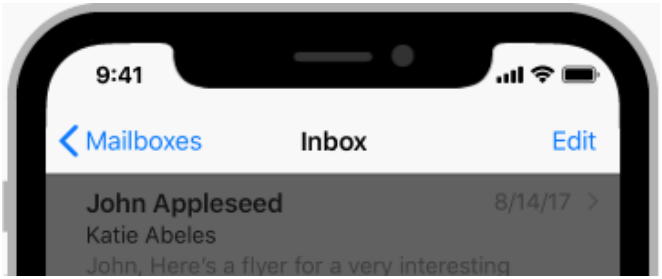
iOS – Status Bar

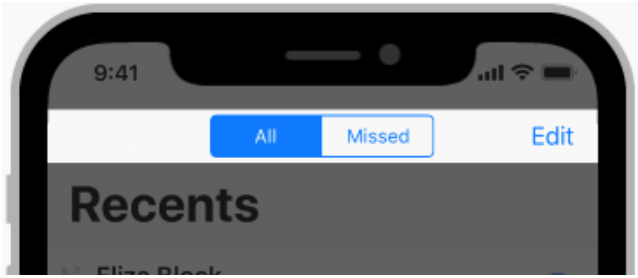
- The status bar appears along the upper edge of the screen and displays useful information about the device's current state, like the time, cellular carrier, network status, and battery level
- The actual information shown in the status bar varies depending on the device and system configuration

Source: https://developer.apple.com/design/human-interface-guidelines/ios/bars/status-bars/

iOS – Navigation Bars

- A navigation bar appears at the top of an app screen, below the status bar, and enables navigation through a series of hierarchical screens (`UINavigationBar`)
  - When a new screen is displayed, a back button (labelled with “Back” or the title of the previous screen) appears on the left side of the bar
  - The right side of a navigation bar can contains a control (Edit or a Done button) for managing the content within the active view
  - In most cases, a title helps people understand what they’re looking at

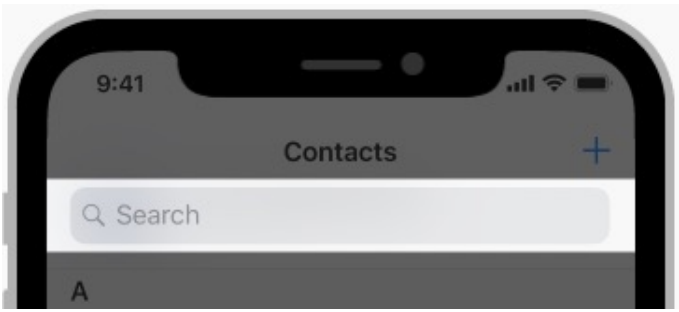


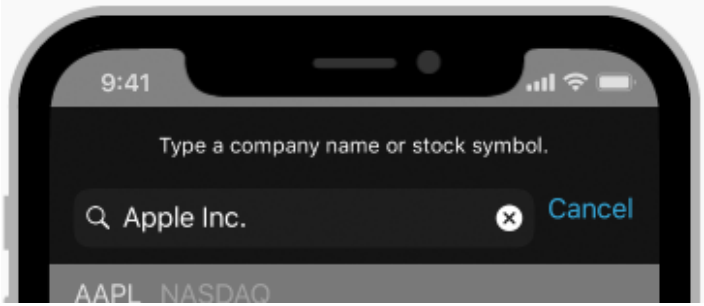


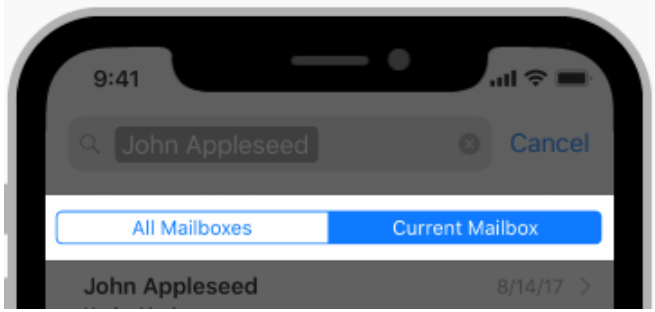
Source: https://developer.apple.com/design/human-interface-guidelines/ios/bars/navigation-bars/

iOS – Search Bars

- A search bar allows people to search through a large collection of values by typing text into a field (`UISearchController, UISearchBar`)
  - A search bar can be displayed alone, or in a navigation bar or content view
  - When displayed in a navigation bar, a search bar can be pinned to the navigation bar so it's always accessible, or it can be collapsed until the user swipes down to reveal it
  - A search bar can have Clear button, Cancel button, or provide hints and context (when appropriate)
- A scope bar can be added to a search bar to let people refine the scope of a search







Source: https://developer.apple.com/design/human-interface-guidelines/ios/bars/search-bars/

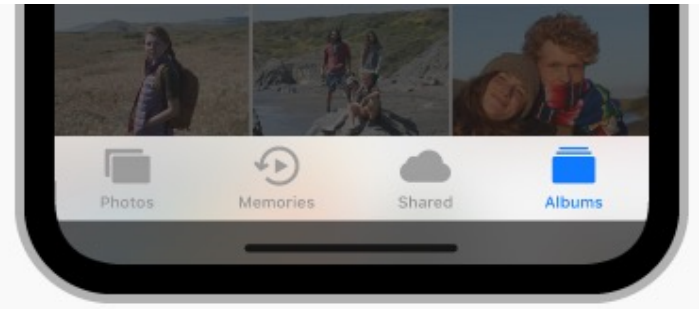
iOS – Tab Bar

- A Tab bar appears at the bottom of an app screen and provides the ability to quickly switch between different sections of an app (`UITabBar`)
- Tab bars are translucent, may have a background tint, maintain the same height in all screen orientations, and are hidden when a keyboard is displayed
- A Tab bar may contain any number of tabs, but the number of visible tabs varies based on the device size and orientation
  - If some tabs can't be displayed due to limited horizontal space, the final visible tab becomes a More tab, which reveals the additional tabs in a list on a separate screen

It's important to understand the difference between a tab bar and a toolbar, because both types of bars appear at the bottom of an app screen. A tab bar lets the user switch quickly between different sections of an app, such as the Alarm, Stopwatch, and Timer tabs in the Clock app. A toolbar contains buttons for performing actions related to the current context, like creating an item, deleting an item, adding an annotation, or taking a photo.

iOS – Tab Bar

- Tab bar can be used to organize information at the app level
  - Use a tab bar strictly for navigation
  - Tab bar buttons should not be used to perform actions
  - Avoid having too many tabs
  - Don’t remove or disable a tab when its function is unavailable



- In a multiview application, taps on the tab bar will refer to the tab bar controller, but taps anywhere else on the screen get processed by the controller corresponding to the content view currently displayed

Source: https://developer.apple.com/design/human-interface-guidelines/ios/bars/tab-bars/

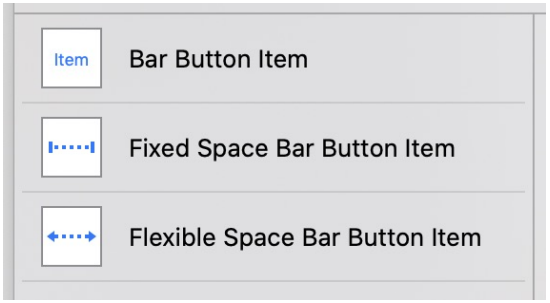
iOS – Toolbars

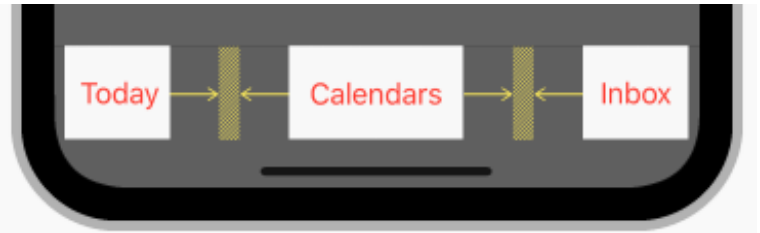
- A Toolbar appears at the bottom of an app screen and contains buttons for performing actions relevant to the current view or content within (`UIToolbar`)
- Toolbars are translucent, may have a background tint, and often hide when people are unlikely to need them
  - For example, in Safari, the toolbar hides when you begin scrolling the page since you are likely reading. You can show it again by tapping the bottom of the screen. Toolbars are also hidden when a keyboard is onscreen.

It's important to understand the difference between a toolbar and a tab bar, because both types of bars appear at the bottom of an app screen. A toolbar contains buttons for performing actions related to the current context, such as creating an item, deleting an item, adding an annotation, or taking a photo.
A tab bar lets the user switch quickly between different sections of an app, for example, the Alarm, Stopwatch, and Timer tabs in the Clock app.

iOS – Toolbars

- Provide relevant toolbar buttons
  - A toolbar should contain frequently used commands that make sense in the current context
  - Consider whether icons or text-titled buttons are right for your app
  - Give text-titled buttons enough room



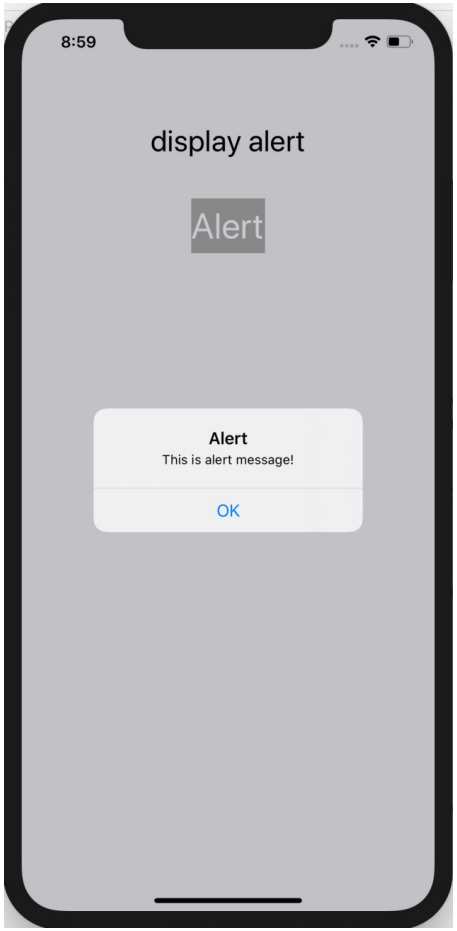


Source: https://developer.apple.com/design/human-interface-guidelines/ios/bars/toolbars/

iOS – Alerts

- Alerts convey important information related to the state of your app or the device, and often request feedback (`UIAlertController`)
- An alert consists of a title, an optional message, one or more buttons, and optional text fields for gathering input
  - Aside from these configurable elements, the visual appearance of an alert is static and can't be customized

```
let alert = UIAlertController(title: "Alert",
                              message: "This is alert message!",
                              preferredStyle: UIAlertController.Style.alert)
alert.addAction(UIAlertAction(title: "OK", style: .default,
                              handler: { (action) -> Void in
                                  self.textLabel.text="OK"}))
self.present(alert, animated: true, completion: nil)
```

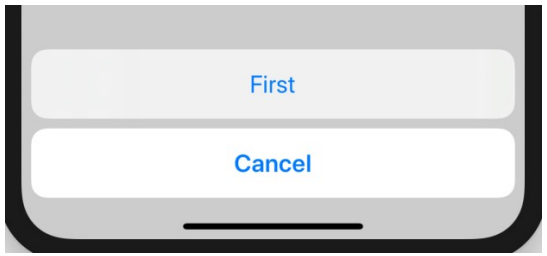


iOS – Action Sheets

- An action sheet is a specific style of alert that appears in response to a control or action, and presents a set of two or more choices related to the current context
- An action sheet can be used to let people initiate tasks, or to request confirmation before performing a potentially destructive operation
  - On smaller screens, an action sheet slides up from the bottom of the screen
  - On larger screens, an action sheet appears all at once as a popover

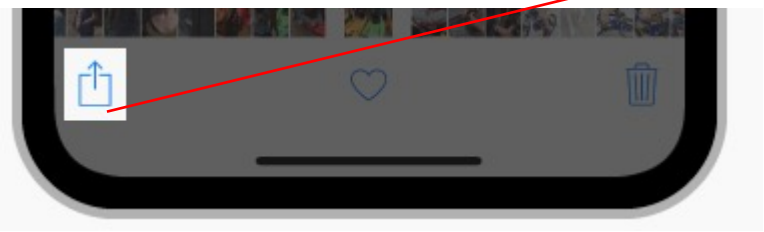
iOS – Action Sheets

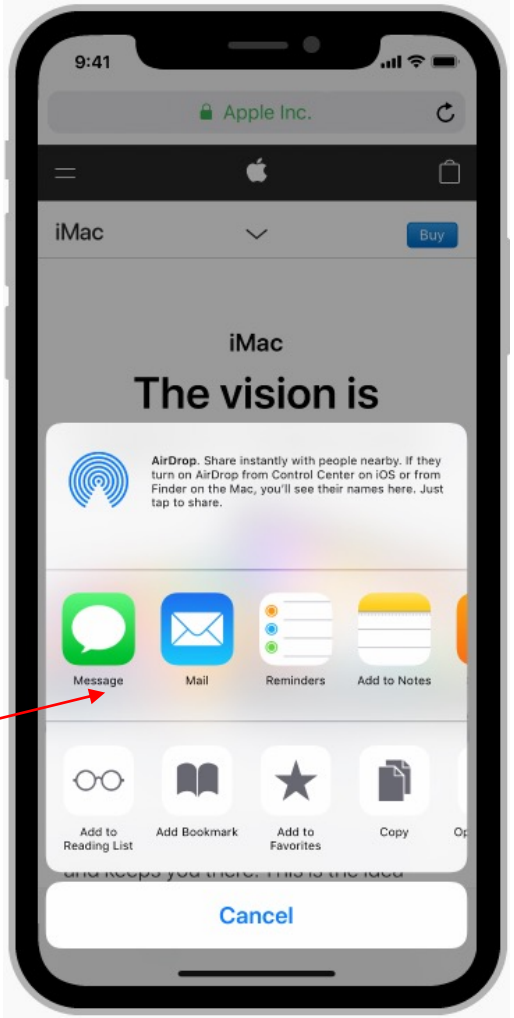
```
let alertController = UIAlertController(title: "Action Sheet",
                        message: "Allows you to Choose option", preferredStyle: .actionSheet)

//let alertController = UIAlertController ()

let firstB = UIAlertAction(title: "First", style: .default,
                           handler: { (action) -> Void in
                           self.textLabel.text="First option"})

let cancelB = UIAlertAction(title: "Cancel", style: .cancel,
                           handler: { (action) -> Void in
                           self.textLabel.text="Cancel button"})
 alertController.addAction(firstB)
 alertController.addAction(cancelB)

 self.present(alertController, animated: true, completion: nil)
```



iOS – Activity Views

- An activity is a task, such as Copy, Favorite, or Find, that's useful in the current context (`UIActivityViewContoller`, `UIActivity`)
  - Once initiated, an activity can perform a task immediately, or ask for more information before proceeding
  - Activities are managed by an activity view, which appears as a sheet or popover, depending on the device and orientation
- The system provides a number of built-in activities, including Print, Message, and AirPlay
  - These tasks always appear first in activity views and can't be reordered There is no need to create custom activities that perform these built-in tasks

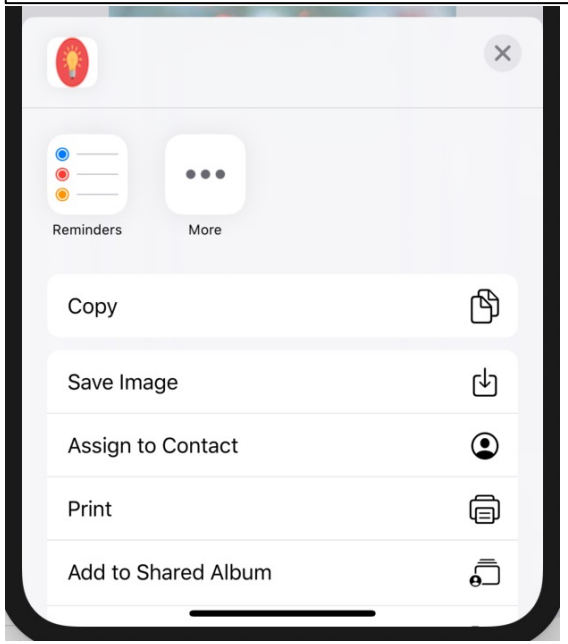




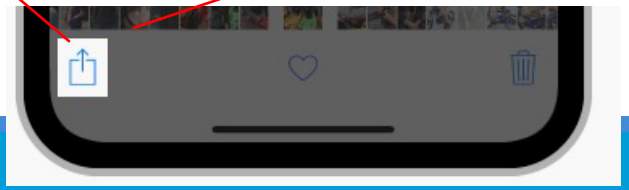
Source: https://developer.apple.com/design/human-interface-guidelines/ios/views/activity-views/

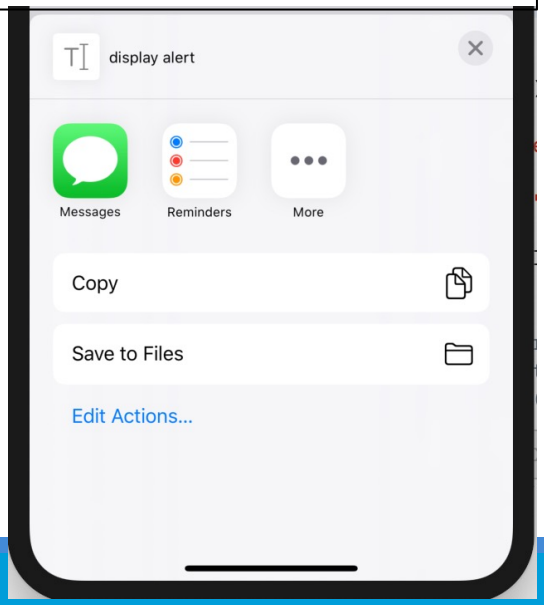
iOS – Activity Views

```
let vc = UIActivityViewController(activityItems: [imageView.image!],
                                  applicationActivities: [])
present(vc, animated: true)
```



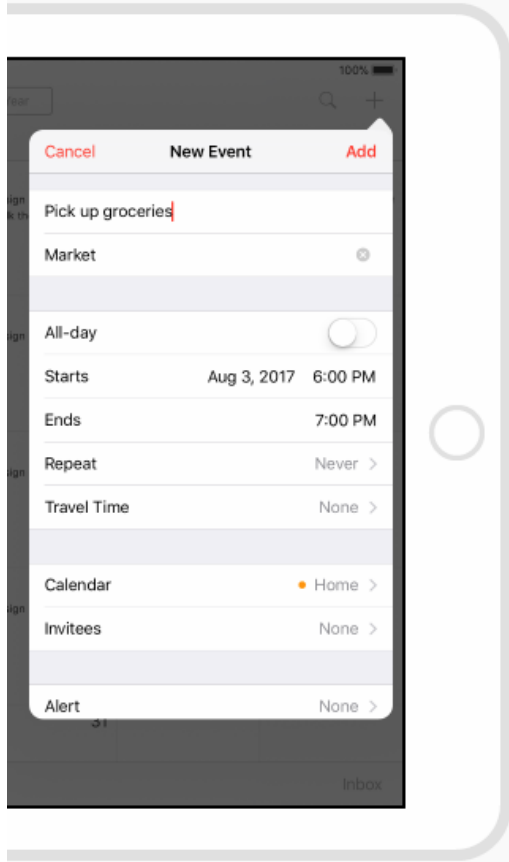

```
let vc = UIActivityViewController(activityItems: [textLabel.text!],
                                  applicationActivities: [])
present(vc, animated: true)
```





iOS – Popovers

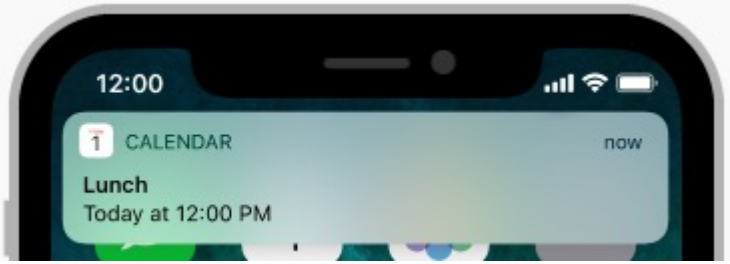
- A popover is a transient view that appears above other content onscreen when you tap a control or in an area (`UIPopoverPresentationController`)
- A popover includes an arrow pointing to the location from which it emerged
- Popovers can be nonmodal or modal
  - A nonmodal popover is dismissed by tapping another part of the screen or a button on the popover
  - A modal popover is dismissed by tapping a Cancel or other button on the popover
- Popovers are most appropriate on larger screens and can contain any variety of elements, including navigation bars, toolbars, tab bars, tables, collections, images, maps, and custom views



Source: https://developer.apple.com/ios/human-interface-guidelines/views/popovers/

iOS – Notifications

- Apps can use notifications to provide timely and important information anytime, whether the device is locked or in use
  - For example, notifications may occur when a message has arrived, an event is about to occur, new data is available, or the status of something has changed
- People see notifications on the lock screen, at the top of the screen while using the device, and in Notification Center, which is opened by swiping down from the top edge of the screen
- Each notification includes the app name, a small app icon, and a message.
  - Notifications may also be accompanied by a sound, and may cause a badge to appear or update on the corresponding app's icon



- Notifications can be local or remote
  - Local notifications originate and are delivered on the same device
  - Remote notifications, also called push notifications, come from a server

Source: https://developer.apple.com/design/human-interface-guidelines/ios/system-capabilities/notifications/

Resources

- iOS 17 App Development Essentials, Neil Smyth, Payload Media, Inc., 2023.
- iOS 13: Programming Fundamentals with Swift - Swift, XCode and Cocoa Basics, Matt Neuburg, O'Reilly Media Inc., 2020.
- iOS 15 Programming Fundamentals with Swift - Swift, XCode and Cocoa Basics, Matt Neuburg, O'Reilly Media Inc., 2021.
- https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html
- https://developer.apple.com/swift/
- https://source.android.com/devices/tech/dalvik

- You may try to Swift on Web Browser in the links below.
  - https://swiftfiddle.com/ - support swift 2.2 to 5.9
  - https://www.tutorialspoint.com/compile_swift_online.php

Resources

- Android Studio Dolphin Essentials - Java Edition, Neil Smyth, Payload Media Inc., 2019
- Programming iOS 14 - Dive Deep into Views, View Controllers, and Frameworks, Matt Neuburg, O'Reilly Media Inc., 2021.
- iOS 12 App Development Essentials, Neil Smyth, Payload Media, Inc., 2018.

- https://developer.android.com/guide/topics/ui/menus
- https://developer.android.com/training/appbar
- https://developer.android.com/guide/components/fragments.html
- https://developer.apple.com/ios/human-interface-guidelines/bars/navigation-bars/
- https://developer.apple.com/ios/human-interface-guidelines/views/action-sheets/
- https://developer.apple.com/ios/human-interface-guidelines/views/popovers/
- https://developer.apple.com/ios/human-interface-guidelines/system-capabilities/notifications/