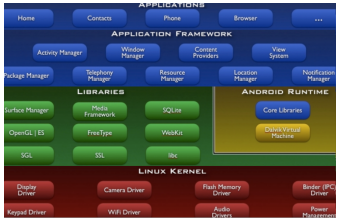
1 Android

1.1 Architecture



Linux Kernel: hardware abstraction layer (HAL), device drivers, memory / process management, networking

Libraries: C/C++ libraries. Interface through Java. Surface Manager. 2D and 3D Graphics. Media codecs, SQLite, Browser engine

Android Runtime: Android runtime (ART) and its predecessor Dalvik are the managed runtime used by apps and some system services. Executes Dalvik Code (translated from Java bytecode). Supports Ahead-of-time (AOT) compilation, garbage collections, profiling and debugging. Optimized for systems that are constrained in terms of memory and processor speed.

Application Framework: API interface, Activity manager (Manages the application life cycle).

Applications: Built-in and user applications. Can replace built-in applications.

1.1.1 Components

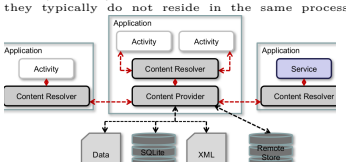
App is built of Components that interacts. Goal: Easy to reuse and replace. Components of other apps can be used (e.g. Gallery). Needs to be registered in the AndroidManifest (<activity android:name=".ActivityB" />) (else exception).

Activity User interface component typically corresponding to one screen. (Moving to next screen means change of Activity).

Service Runs in the background without user interface. Example: music player, network download, etc

Broadcast Receiver Component that receives and reacts to broadcast announcements (<- are Intents too). Many broadcasts originate in system code (E.g., announcements that the time zone has changed, that the battery is low. Incoming SMS) Receiver are implemented by extending BroadcastReceiver.

Content Provider recommended way to share data between Android applications (E.g. address book, photo gallery). Represented by URI and MIME type. Applications do not call content providers directly. They call ContentResolvers instead as they typically do not reside in the same process.



1.1.2 Transferring Program Control

Intents: (Passive object, Set of Strings). Used for transfering control or notify components (VIEW, CALL, PLAY, ...). Systems matches Intent with most suitable.

```
public void onClickSendBtn(final View btn){
  Intent intent = new Intent(this, Receiver.class);
  intent.putExtra("msg", "Hello World");
  startActivity(intent);
}
```

Android uses a requestId to **return results from a sub-activity**:

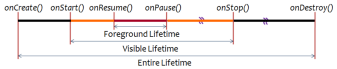
```
startActivityForResult(new Intent(this, A.class),
        i1);
startActivityForResult(new Intent(this, B.class),
        i2);

// in sub-activity A or B
setResult(resultCode, intent); finish();

// back in main activity
@Override
protected void onActivityResult(int requestCode,
        int resultCode, Intent
data) {
  switch (requestCode) {
    case i1: // result from call with i1
    if (resultCode == RESULT_OK) { /* ... */ }
    break;
    case i2: // result from call with i2
    if (resultCode == RESULT_OK) { /* ... */ }
    break;
  }
}
```

1.1.3 Activity Lifecycle

State of an activity is managed by the system.

System may: 1) move another activity into the foreground. 2) ask the activity to finish. 3) even simply kill its process.

State	Description
Running	An activity is in the **foreground** of the screen (at the top of the activity stack for the current task)
Paused	An activity has lost focus but is **still visible** to the user
Stopped	An activity is **completely obscured** by another activity. It still retains all state and member information



System notifies an activity of a state transition by calling methods: **onCreate:** first create or when activity was killed. **onStart:** just before activity becomes visible. **onRestart:** after activity has been stopped, to being started again. **onResume:** before activity starts interacting with user (input goes to activity). **onPause:** when about to resuming other activity (commit unsaved changes here! stop animiations and CPU consumings) **onStop:** when no longer visible to user (e.g. when destroyed or other activity resumed) **onDestroy:** before destroy, but there is no guarantee.

1.1.4 Configuration

Strings for localization, Images for different resolutions, Layouts for different devices, ...

Seperated from code with resource files. Stored in **res** directory and grouped by type: *drawable*, *layout*, *values*. For example *res/values-de/strings.xml*

```
<?xml version="1.0" encoding="utf-8"?>
  <resources>
      <string name="hello_world">Hello World!</string>
  </resources>
</xml>
```

Reference between resources with e.g. @string/hello„*orld*.

Accessing resources in Java code with **wrapper class called R**, that contains resource ids as static integers.

```
// Load a custom layout for the current screen
setContentView(R.layout.main_screen);
// Set the text on a TextView object.
TextView msgTextView = (TextView)findViewByID(R.id.
        msg);
msgTextView.setText(R.string.hello_message);
// Set the title from a resource
this.getWindow().setTitle(Resources.getText(R.
        string.main_ title));
// Load a background for the current screen from a
        resource
this.getWindow().setBackgroundDrawableResource(R.
        drawable.my_
background_image);
```

1.1.5 AndroidManifest

Properties of Application: Name / ID (package), Version of App, Technical User (sharedUserId), Required SDK, Required Privileges, Components

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.android.lifecycle"
    android:versionCode="1"
    android:versionName="1.0" >
    <uses-sdk
        android:minSdkVersion="14"
        android:targetSdkVersion="21" />
    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name" >
        <activity
            android:name=".ActivityA"
            android:launchMode="singleTask" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category
android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
</application>
</manifest>
```

To be available from the launcher it must include an intent filter listening for the MAIN action and the LAUNCHER category

1.1.6 Application Lifecycle

1.2 TODO SOME QUESTIONS OF BSKON EXAMS BECAUSE SAME TEACHER AND SAME CONTENT

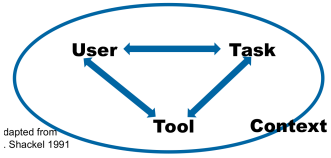
2 Usability

2.1 Secrets of simplicity

1. Remove features (get rid of things you never use) 2. Hide features (put some of the features where they won't get in the way) 3. Group features (easier to find) 4. Display features (on-screen menu) Adding more instructions can be less simple >< (close). Remove too much can make user feel out of control. Notebook L2 cache too complicated, too less information experts won't buy. Shade things ore make bigger to stand out more.

2.2 Usability

User, Task, Tool, Context: All need to be considered for good usability. (all connected and inside a circle - the context). All 4 can be real, simulated or ignored. Good user research documents observation of: representative set of users, doing a set of meaningful and representative tasks, using their current tools & strategies, in a meaningulf and representative context. **Finden von zukÃ¤ijnfigten Nutzern:** - we could not do proper user research, until system development was completed. + user needs, tasks, contexts, strategies and basic tools must be around. Goals are reached already today, just not easily. **Testen der Korrektheit von Anforderungen:** - we could not test our requirements, because the system was not yet completed. + Good tests are cheap, quick, relevant and valid. There is a standard for it: ISO 9241-11: effectiveness, efficiency, satisfaction. Quesenberry 5E Model : effective, easy to learn, efficient, error tolerant, engaging. If ease of use were the only requirement, we would all be riding tricicles

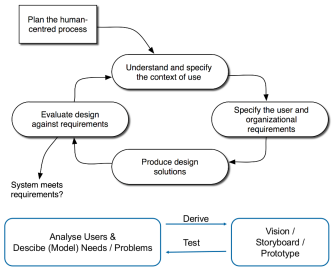


dapted from
. Shackel 1991

2.3 Product Criteria by Stone

Visability: first step to goal should be clear, **Affordance:** Control suggests how to use it. Result conforms to expectation generated by control. **Feedback:** Should be clear what happened or is happening. **Simplicity:** As simple as possible and task-focused. **Structure:** Content organized sensibly. **Consistency:** Similarity for predictability. **Tolerance:** Prevent errors, help recovery. **Accessibility:** Usable by all intended users, despite handicap, access device or environmental conditions.

2.4 User Centered Design Process



2.5 Usability vs User Experience

Usability: effective, efficient, learnable, error-preventing. **User Experience:** value & meaningful, pleasurable / impressive / memorable, end-toend experience, product & service experience → pre-use: anticipated use, search, unboxing. regular use: first success, usability. post-use: loyality, re-use design, upgrade, replace, recycle.

2.6 Garrett's Framework classify Usability

top(surface) = concrete, bottom = abstract (strategy). **surface:** visual design (color, fonts, design). **skeleton:** interface design, navigation design, information design (layout grid). **structure:** interaction design, information architecture (navigation, conceptual model). **scope:** functional specification, content requirements (features). **strategy:** user needs, site objectives (target-group, needs, "value", meaning). UCD Techniques: **Interviews, contextual inquiry** (strategy, scope | analyse design). **Scenarios, storyboards** (scope, structure | (analyse) design). **Wireframes, prototypes & testing** (structure, skeleton | (design) test).
Benutzerbefragung ist keine User Centered Design → People don't know what they want. You have to show it to them first. First rule of usability: Don't listen to users. Observe what they do not what they say. Customer → problem expert. Designer → solution expert.

2.7 Scenarios and Personas

story of the user solving a problem that arises out of logical needs of the situation. **Problem-Scenarios** show current (problematic) situations. **Future-Scenarios** show users with the same needs and in a similar context as in the problem-scenarios. They illustrate how new tools lead to better outcomes. Good Scenarios need good personas and good user research. (Garrett: User Segmentation + Selection) Scenario = Text or Storyboard. Elements = Problem description(User goal) & context. User (Persona). Trigger. Steps. Solution (maybe fail). **Good Scenarios:** should include first success, repeated success (triggered), virality. should have plausible needs, goals, context, trigger, persona. NOT CRUD questions with answers for locations. BUT First use scenario: Peter got a recommendation for the local experts app from a friend .... On the first launch asks permission, he agrees...AND Repeated Success / Triggered Scenario. Peter is in Chur, a place he doesn't know. It is dinner time... remembers the app.

2.8 Needs

Apps: I want to share something ("Check In / Status") → Social Media, Photo.. I am bored (I want to be entertained / distracted) → Games, News.. I want to be productive (repetitive now, micro tasking) → Sort E-Mail, Quick ppt edits. I want to find something here (urgent, local) → Map, Schedule, Restaurant-Finder (location-based services).

2.9 Usable in varying use context

User holing patterns should be respected. Reachability & touch target size: Users cognitive limitations should be respected: Users might be in very noisy (or very quiet) contexts. Users may be from varying age groups, with varying visual abilities, and in varying lighting situations (contrast, font size, colors). Users might be in constant mode of distraction (App needs to remind users of its existence, quick results even when users are distracted, interrupted or first time use or long since last use.) Users in hazardous (resource limited) situations. Users might show varying levels of involvement.

2.10 Core Future Scenarios Mobile

Scenario: First success: Why (how, when) was the app installed by the user? Why is the app used the first time (trigger, motivation to start / to go through all the required steps until success)? When is the first time(step) the user gets a recognizable reward/benefit from the use of the app?

Scenario: Repeated success: Why is the user starting the app again (trigger)? What are the repeated benefits? How does the app cater for experts without losing infrequent users?

Scenario: Virality Why (how) will the user tell others about the app or ge them involved?

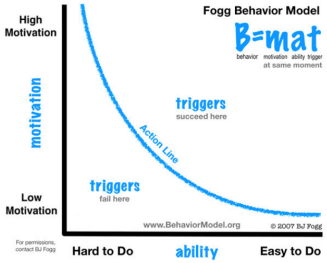
Phases of app use: I Attract (visual, desirable) II Delight (information / function, useful and usable) III Retain (repeated use, notifications) Goal → Use power of viral marketing. 26 percent of apps are used only once. Sport apps seems to be used the longest.

2.11 Mobile vs Desktop

Mobile: Small screen, input a few characters, slow (or no) network, photograph anything, used anywhere, location aware (mostly). **Desktop:** Large screen, type text, fast Internet, photograph user, used when seated, location unaware.) → Apps should make use of location information: Determine current context: (GPS, WiFi cell, beacons, ambient sound, image reco, sensors(gyro)). Provide info about: (Points of interest, direction, notes (location based notes, leave notes to others), location of friends).

2.12 Challenge for Apps

Increase motivation (psychology). Removing Friction (usability) Mountain. Increase motivation to climb over the mountain or make the mountain smaller. Apps must provide motivation, ability and trigger:



Masclow's Hierarchy of Needs: Physiological, Safety, Love/belonging, Esteem, Selfactualization.

2.13 Techniques of User Centered Design

ANALYSE - Stakeholder-Analysis, User Interview, Usability Test & Heuristic Review (current system), Competitive Analysis, Contextual Inquiry / Ethnographic Interview, Persona & Senario Modelling, Visioning & Storyboarding, Card Sorting, Wireframing (Heuristic Review, Hallway Testing), Usability Lab Test - DESIGN

	Themen	Analyze Collect Document	Design Plan, Rank	Test Evaluate Observe
Surface	Color Fonts Animation		Mood-Board	
Skeleton	Layout Grid Animation	Device Screen Sizes & Resolution Analysis	Page Grid	Usability Lab
Structure	Navigation Information Architecture	Card Sort	Site Map	Paper Prototype Test
Scope	Features	Problem scenarios	Future Scenarios	Expert Evaluation
Strategy	Target Group „Value"	Contextual Inquiry	Personas	Pilot Tests

2.14 Mobile Design Process

Start small (small set of features (1+2), focused target group). Ideation / Concept Development (parallel versions) → Identify user needs (hypothesis), validate user needs (Observation, Validate Problem and Future Scenarios). Select one or two concepts for refinement. Refine Concept(Develop "Paper" Mockup for Scenario → redesign, validate with walkthrough, test scenarios with mockup → redesign) apply platform guidelines → retest, Test detail interactions → animation). In parallel: remove technical risks. Implement and test scenarios (redesign if necessary).
For MSE App: Users, What to observe. How to observe. Hypothesis of needs. Why installed (trigger, motivation, ability)*. Possible first success scenario *. Possible reuse scenario *. Possible virality scenario *. How to demonstrate validity of scenarios

2.15 Design Concept

Good Concept-Design: Identifies strong situational needs. Identifies a core set of matching scenarios (including Personas)= Co-evolves tested wireframes, scenarios and needs. **Goal must be:** All features represented as screen flows (sequence of filled wireframes supporting a scenario). No untested wireframes (No out-of-scenario wireframes). No wireframes without scenario data. **Step towards goal:** 1) Create a reasonable empty wireframes collection. Create initial set of scenarios. Walk through wireframes. Iterate. 2) Create testable screen flows and test-task description (few at a time). Validate: Check with Cognitive Walkthrough: do enough pre tests. Plan 3-5 real tests. Iterate

2.16 Card Sort

Useful technique to determine navigation hierarchies and naming of menu item. **Open Card Sort:** Start with content cards. Let future users create groups and name them (5+ users). **Closed Card Sort:** Start with content cards AND GROUP LABELS. Let future users match content cards to group labels. IF YOU THINK YOU HAVE TO USE CARD SORT FOR APPS THEN IT POSSIBLY HAS TOO MANY FEATURES.

2.17 Screen Map

Lists all screens of an app, groupings and major navigation links. The screen map for horizontal tablets might differ from the one vertical tablets or for small screens. Horizontal tablet layouts often combine multiple views. They show descendant and lateral navigation (also maybe back and up). Show List, Grid, Carousel, simple buttons, dashboard, tabs, swipe etc. **Abstract Screen Map** Home, Photo List, Photo View. Story View etc. **Wireframe Screen Map** Show the screens and what happens if menu button is pressed etc.

2.18 Prototyping, tools and usability testing

Using just paper, can be faster and more efficient for testing. Tools can be used to make the same electronic for Interaction, Animation, Gestures, Design, Demoing, Documentation, Responsive Design (Marvel for example). **Usability testing challenges:** Defining good scenarios with plausible needs, goals, context, trigger. persona (user can log in is bad). Creating inexpensive and quickly the needed screen flows for testing (not collection of empty wireframes). Creating matching task descriptions that communicate needs... (not log in as user: test-user, pw 123). Inviting the right test persons (beware of friends and family). Making test persons understand that the system/concept is tested (pre- and post- questionnaire). Make test persons think aloud (let them read the description than they should continue with talking. Only controlled help).

2.19 Co-Developing Screen Flows & Test Tasks

Scenarios are the basics for creating screen flows and description of the test tasks. Test tasks specify: user context, need, goal and trigger. Do not specify: specific terms that should be used, specific steps that should be taken. Example triggered task: see Scenarios and Personas. Screen flows include the data that would be entered for an optimal task performance.

2.20 Testing Mistakes

1 Recruiting unsuitable participants. 2 Not testing early and often. 3 Following a test plan too rigid. 4 Not rehearsing the setup. 5 Using a one-way mirror. 6 Not meeting participants in reception. 7 Asking leading questions. 8 Undertaking two roles in testing session. 9 Not considering external influences. **Things that can go wrong** 1 Users don't show up. 2 Facilitators gets sick. 3 Internet goes down. 4 Awkward moments. 5 Distractions. 6 Users are quiet. 7 Software stops working. 8 Takes too long. 9 Forget to record the time. 10 Video didn't record.

2.21 Designing App Skeleton (Pages + Grid)

Difficult to know what screen size user will interact with the app. Goal should be achievable on all devices and orientations. Knowledge about orientation and device can help to optimize. Tablets are more used at home and older people. Holding patterns should be used to optimize visibility.

Touch targets should be at least $1cm^2$. Best is 0.9cm + 0.2cm padding. (more space needed inf used in stressful situations)

2.22 Mobile Design Pattern

Empty Datasets: You haven't liked any photos yet. Spingboard: Like like tic tac toe. List Menu, Tab Menu, Gallery. Primary Navigation (Transient) → Side Drawer, Popup Menu. Secondary Navigation → Page swiping (hor or vert). Tips: Make primary actions obvious: High-contrast button affordance. Segmented Control instead of Toggle Menu. ZIP instead City state zip. Inline validation: did you mean gmail.com Use Switch Slider Segmented Controls. Mobile first, don't port Desktop UI to mobile.

2.23 Design in Mind

Error message close to action. Keep in mind that 9 per cent of men have color vision deficiency. **Mistakes:** Too many steps to first success (create profile, tutorial). Touch areas too small. Non standard controls. Android users designing for iOS or vice versa. Web designers designing for mobile. Corporate Design and marketing knows it better...

2.24 Android Guideline

Has a back button and app stack (Back != up). Put back button in app is bad (if necessary provide up button), same with exit button. **Antipatterns:** Splash screen (better image placeholders), tutorial screen (better explain in time, context), Confirmation window (better provide undo), Menu button (outdated), Hiding status bar, sipe overlay quick actions, using non-android design. Don't mix actions and navigation in a single bar.

2.25 IOS Platform Guideline

The iOS HIG (Human Interface Guideline) is like material design but for ios (Overview, Interactions, Features, Visual Design, Graphics, UI Bars, UI Views, UI Controls, Extensions, Technologies, Resources, Related Guidelines). Consider putting a segmented control in a navigation bar at the top level of an app. Helps to flatten the information hierarchy, making it easier to find things. Be sure to choose accurate back button titles. The Floating Action Button is not that good, better right side of navigation bar or tool bar. iOS needs close buttons! iOS design: everything clickable no side menus, better no side menus in iOS. Google has side menu integrated (older than 40 not used to click on hamburger icon to get to menu). Tab-bar new at the bottom for both system. Modern take swift: Statically, strongly typed. Compiler can often infer types (type annotation can often be omitted). Compiles to native code. No main, semicolons required. print() is defined in the Standard library (implicitly imported). Only file which can contain top level code is main.swift (else top level declarations). Goal: safer, more flexible more fun more than Objective C (interoperability).Integer overflow traps. + Better chance to find overflow bugs. + Well defined behavior. - Requires run-time checks. Has types Int, Float, Double, String, Bool, Array<T> or [T], Set<T>, Dictionary<K,V> or [K:V]. All have value semantics. Some use coopy on write in order to be efficient. Are nominal types: can be extended (initializer (ctor) methods etc.)

2.26 Material Design

Principles: Material is the metaphor: Elevation of materials, what is above which element, how height. Bold, graphic, intentional: typography, grids, color, scale, space, create hierarchy, meaning, focus. Motion provides meaning: focus attention, giving feedback. **Components:** Bottom Navigation. **Patterns:** Empty States: image = neutral, purpose and potential like icon, positive tone, consistent with brand, should not look like it's an action. Permissions: simple, transparent and understandable. Should clarify why permission is needed. Runtime permissions = at the moment user needs to perform action. Denied permissions should provide feedback and options. Types of permissions: educate before asking, ask up front, ask in context, educate in context, provide an immediate benefit, only ask for relevant permissions. Scrolling: Use flexible space to accommodate images in the app bar with the desired aspect ratio.

2.27 Agile SW Development

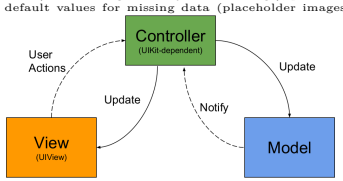
DESIGN (create mockups) → DEVELOP → COMPILE → TEST → REFACTOR
COMPILE DISTRIBUTION VERSION → TEST → RELEASE/PUBLISH

2.28 Costs

40k for a kart, 100k for a Skoda, 500k for a BMW, 1mio+ for a Rolls Royce. Switzerland iPhone Country (2/3 : 1/3) but worldwide android 80-90. **When go native:** If security is very important (SDKs NDK), Performance or resource optimization (battery, memory), Use newest technologies (APIs, wearables etc), When only one platform must be supported. Pixel perfect UIs. **When go cross:** Low budget, only basic requirements for UI, Web programming skills available but no native skills, prototyping or proof of concepts, Game engines, 3D visualization (unity). Mostly it's not as much faster to implement and not much less cost as expected. 60 per cent is not yet using swift.

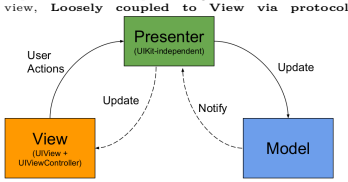
2.29 MVC - Model-View-Controller

Model = represents app's data, notifies the controller about changes in the data, takes care of things like persistence, model objects and networking. View(UIView) = represents the face of the app, notifies the controller about user-actions, reusable classes without domain-specific logic. Controller(UIKit-dependent) = mediates between model and view, implements domain-specific logic, updates model and view. Problems = Tight coupling between View and ViewController, Controller is hard to test because of UIKit dependency, MVC == Massive View Controller = Delegate / DataSource methods, Target-Action methods, ViewController Lifecycle methods, Layout code, Formatting of data (transforming data object into strings), providing default values for missing data (placeholder images)



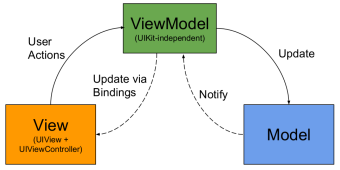
2.30 MVP - Model-View-Presenter

Model = represents app's data, notifies the controller about changes in the data, takes care of things like persistence, model objects and networking. View (UIView + UIViewController) = Represents the face of the app, Notifies the presenter about user-actions, **knows the presenter**. Presenter(UIKit-independent) = mediates between model and view, implements domain-specific logic, updates model and view, **Loosely coupled to View via protocol**.



2.31 MVVM - Model-View-ViewModel

Modle = represents the app's data, notifies the controller about changes in the data, takes care of things like persistence, model objects and networking. View (UIView + UIViewController) = represents the face of the app, **notifies ViewModel about user-actions and observes properties of View-Model, Knows the ViewModel**. ViewModel(UIKit-independent) = mediates between Model and View, Implements domain-specific logic, updates model and view (indirectly via bindings), **loosely coupled to view via bindings / Observer-pattern**.



2.32 Swift

Swift is statically typed (types known at compile time), strongly typed (there aren't a lot of implicit type coercions (pass int instead of double needs cast)), compiler can often infer types (type annotations can be omitted), uses automatic reference counting (ARC) for memory management. Compiles to native code (doesnt run in virtual machine), may rely on Objective-C runtime (not available on linux). No main() required. print() defined in standard library (implicitly imported). Only main.swift can contain top-level code (all others only top level declarations). Goals = safer, more flexible more fun than Objective C. Each significant change is described in a proposal (Markdown). idea mailing list - write proposal - request review - core team member who accept pull request becomes review manager - number assigned - anyone can review - core team decides if accepted rejected or deferred.

2.33 Numeric Types

Some of the types use copy-on write in order to be efficient. They all have value semantics. Are nominal types (can be extended). var x = 2, x += 2 Int, let y = 4.5 Double ,let z: Float = 4.5 Float, let (d1,d2) = (2,4.5) func f(_x: Double) ,f(x) cant convert Int to Double ,f(4) works integer literal

2.34 Strings

Are unicode-compilant, value semantics, different views for various unicode representations. var str = "Hello", str += " x!" (x = emoji), for c in str.characters print(c) = human readable characters, str.characters.count = 8, str.utf8.count = 11, str.utf16.count = 9, str.uppercased(), str.lowercased()

2.35 Frameworks

2.36 Structure of an App

2.37 TODO SOME QUESTIONS FROM UINT1 UINT2 WHICH ARE COVERED IN THIS CLASS

3 Swift and IOS

3.1 Arrays

have to be same type, value semantics, empty array [], [Int] = Array<Int>

```
let ints1 = [1, 2, 3, 4, 5] //Array<Int>
var ints2 = ints1 // mutable copy
ints2.append(6) // here copy
print(ints1)
let strs = Array(repeating: "Hi", count: 10)
for s in strs { ... }
for (i, s) in strs.enumerated() { ... }
ints2[0...<3] = [0, 0]
ints2[0...4] = []
```

3.2 Sets

Elements needs to conform Hashable protocol. Value semantics.

```
var letters: Set<Character> = []
for c in "it is a test".characters {
   letters.insert(c) }
if letters.contains(" ") { // compiler knows its
        char not str
   print(letters.count) }
```

3.3 Dictionaries

keys need to conform to Hashable protocol. value semantics, empty dictionary [:] [TypeK:TypeV] = Dictionary<TypeK, TypeV>

```
let population = ["Switzerland" : 8_000_000,
                  "Germany" : 80_000_000]
for (country, count) in population {
   print("\(country): \(count) people") }
print(population["Germany"])
print(population["Italy"]) // nil
population["France"] = 66_000_000 //new
for k in population.keys { }
for v in population.values { }
```

3.4 Tuples

Tuples, function types, any, anyobjects cant be extended ! multiple values into single compound value, can have different types, no single-element tuples Type(Int) = type int. Expression ("hello") = type String not (String). Empty tuple () is a valid type. Has a single value, same as Void

```
let john = (33, "John") // (Int, String)
print("\(john,1) is \(john.0).")
let dora1 = (age: 26, name: "Dora1")
var dora2 = dora1
dora2.name = "Dora2"
dora2.age += 1
print(\(dora2.name) is \(dora2.age))
```

3.5 Function Types ** buggy

```
func f1() {}  // (()) -> () **
func f2(_x: Int) -> Int { return x } // (Int) ->
        Int
func f3(_x: Int, _y: Int) {} // ((Int, Int)) -> ()
        **
func f4(_x: (Int, Int)) {} // ((Int, Int)) -> Int
```

3.6 Any vs AnyObject

any: existential type without requirements, build into compiler, all types are implicit subtypes of it

```
func f(_x: Any) {}
class C {}
let c = C()
f(c)
f(2)
f((0.5, "test"))
f([true, false, true])
f(f)
```

They all work. If AnyObject instead of Any it has to be a class. Only f(c) works. (class requirement) Never: uninhabited type in stl (doesnt have any value) public enum Never, means that function can not return, examples fatalError() exit(), they can be used in else-clause of guard statement.

3.7 Type Inference

uses bi-directional type inference (not like C++, Java, Objective C), scope limited to single statement let x, x = 10 is not possible! (has to be x:Int). Sometimes doesnt work as expected or takes a bit longer to compile.

```
let d = 5.5
let f: Float = 5.5
func id<T>(_x: T) -> T { return x }
func(g) -> Int { return 42 }
func g() -> String { return "Test" }
let x = id(g()) //error ambiguous
let i:Int = id(g()) // 42
let s:String = id(g()) // "Test"
```

3.8 Force Unwrapping

```
var optInt: Int?  //nil = Optional<Int>
optInt = 42  // Optional<Int>
print(optInt!) //42 if nil = error
```

3.9 Optional Binding

Creates a new variable from optional but only if not nil. Can be used in condition (if while guard) true if not nil.

```
if let text = readLine(),
  let number = Int(text) {
  print("Number = \(number)")
} else { print("No number") }
```

3.10 Optional Chaining

```
var text = readLine()?.uppercased() // () nil ->
        nil
print(type(of: text)) //Optional<String> res =
        Optional
text?.append("test")  //text nil -> not called
```

3.11 Nil Coalescing Operator

```
let text = readLine() ?? ""
let number = Int(text) ?? -1 // res non optional
```

3.12 If Statement

```
let arr ? [1, 2, 3]
let opt: Int? = 42
if !arr.isEmpty, let opt = opt {
   // array is not empty, optional not nil
} else { empty or optional nil }
```

3.13 Switch Statement // doesnt fall through cases

```
let peopleCount = 42
switch peopleCount {
   case 0:
      print("no people")
   case 1:
      print("one person")
   case 2...10:
      print("a few people")
   default:
      print("lots of people") }
```

3.14 For-In Statement

```
let numbers = [4, 8, 15, 16, 23, 42]
for n in numbers {
   print(n) }
for (i,n) in numbers.enumerated(){ //tuple
   print("numbers[\(i))] = \(n)") }
for n in numbers where n % 2 == 0 {
   print(n) }
```

3.15 While Statement

```
while let line = readLine() {
   print(line) }
```

3.16 Repeat-While Statement

```
repeat{
   if let pw = readLine() {
      if pw == "secret" {
         break  // successful }
   } else {
      break }
} while true
```

3.17 Guard & Defer Statement

```
import Foundation
func readFile(at path: String) -> String? {
   guard let file = FileHandle(
     forReadingAtPath: path) else {
      return nil } // file path not exist
   defer { file.closeFile() } // closed at end of f
   let data = file.readDataToEndOfFile()
   guard let content = String(data: data,
     encoding: utf8) else {
      file.closeFile()
      return nil }
   return content }
if let content = readFile(at: "/path/file.txt") {
   print(content) }
```

3.18 Error Handling

```
enum FileError: Error {
   case notFound
   case unknownEncoding }
... readFile(...) throws -> String {
        guard ... else { throw FileError.notFound }
        ...
        " { throw FileError.unknownEncoding }
        ... }
do {
        let content = try readFile... }
catch FileError.notFound { print("error nf")}
catch FileError.unknownEncoding ...
// instead of do try catch throw
// 1. let content = try? readFile(...)  nil
// 2, let content = try! readFile(...)  fatal error
```

3.19 Stored Properties

```
var a: Int   // cant print now
a = 8  // ok
var b = "Hello" //String infered by compiler
var c1 = 2, c2 = 4.5
var (d1,d2) = (2, 4.5) // useful for return
var x: Int = 0 {
   willSet { //called before change }
   didSet { //called after change } }
```

3.20 Computed Properties

```
import Foundation
var v = {6.0, 8.0}
var vlen: Double{
   return sqrt(v.0 * v.0 + v.1 * v.1) }
var radius = 5.0
var area: Double{
   get { return radius * radius * Double.pi }
   set { radius = sqrt(newValue / Double.pi) } }
```

3.21 Lazy Properties

```
class File {
...
   lazy var content: String? = {
      return try? String(contentsOfFile: self.path
        ,...)
   } }
let file = File(path: "as.txt") //content not
print(file.content) // file is read, accessed 1st
print(file.content) // not read again
```

3.22 Functions Parameter Names

functions can be overloaded, generic, are reference types, first-class types = can be passed to other functions, can return other functions, declarations can be nested. Parameters have internal (person, hometown) and external name (person, from).

```
func greet(person: String, from hometown: String) {
   print("Hello, \(person) from \(hometown)!") }
func square(_ n: Int) -> Int {
   return n * n; } //no external name, inernal n
greet(person: "Tim", from: "BR")
print(square(5))
```

3.23 Higher-Order Function

```
let numbers = [1, 2, 3, 4, 5]
func multiplyByTwo(n: Int) -> Int {
   return 2 * n }
print(numbers.map(multiplyByTwo))
func makeMultiplier(factor: Int) -> (Int) -> Int {
      func multiplier(n: Int) -> Int {
         return factor * n }
      return multiplier }
let multiplyByThree = makeMultiplier(factor: 3)
print(numbers.map(multiplyByThree))
```

3.24 Generic Functions

```
func _min<T: Comparable> (_ x: T, _ y: T) -> T {
   return y < x ? y : x }
func sum<T: Sequence>(_ numbers: T) -> Int where T.
      iterator.Element == Int {
   return numbers.reduce(0,+) }
```

3.25 Inout Parameter

when the function is called, the value of the argument is copied. in the body of the function the copy is modified, when the function returns the copy's value is assigned to the original argument.

```
func _swap<T>(_ x: inout T, _y: inout T) {
   (x,y) = (y,x) }
_swap(&i1, &i2)
```

3.26 print

```
func print(_ items: Any..., separator: String = " "
      , terminator: String = "\n")
//variadic parameter, because the parameter
      separator and terminator have an external
      name we can omit eithr one or both of them.
```

3.27 Closures (anonymous functions)

```
let numbers = [1, 2, 3, 4, 5]
//full closure syntax
let squaredNumbes = numbers.map({ (n: Int) -> Int
      in return n * n })
//infer parameter type and return type
.. = numbers.map( { n in return n * n})
//use implicit parameter names ($0, $0) and
      implicit return
.. = numbers.map({ $0 * $1})
//use trailing closure syntax
.. = numbers.map {$0 * $0}
// by default captured by ref
let closure1 = { print(x) } //x change = change
// by value
let closure2 = { [y] in print(y) } // y change =
      same
```

3.28 Classes

are reference types, support single inheritance, can adopt zero or more protocols, can be generic, initializers and deinitializer. If all properties of a type have a default value, a default initializer is implicitly generated. For structs, a member-wise initializer is generated.

```
class Person {
   var name: String
   init(name: String) {
      self.name =  name } }
let p1 = Person(name: "Tim")
p1 = Person(name: "Tom") // error
p1.name = "Tom" // ok
var p2 = Person(name: "Steve")
p2 = p1
```

3.29 Initializers

```
init() { self.name = "<unknown>" }
init?(name: String) { // failable initializer
   guard !name.isEmpty else { return nil }
   self.name = name; }
```

3.30 Casting Operators

```
class Animal {}  //downcasting needs !
class Cat: Animal {}
class Dog: Animal {}
let cat1 = Cat()  // stat cat, dyn cat
let cat2: Animal = Cat() // stat an, dyn cat
let x1 = cat1 as Animal // stat an, dyn cat
let x2 = cat2 as! Cat // stat cat, dyn cat
let x3 = cat2 as! Dog // runtime error!
if let x4 = cat2 as? Dog {..} // better
var a: Animal = Dog() //stat an, dyn dog
if (a is Dog) {..}
a = Cat()  //stat Animal, dyn Cat
switch a { case is Cat: ...}
```

3.31 Subscript

```
class Matrix {
   ...
   var grid: [Double]
   init(rows: Int, cols: Int) {
      self.rows = rows
      self.cols = cols
      grid = Array(repeating: 0.0, count: rows *
        cols) }
   subscript(row: Int, col: Int) -> Double {
      get { return grid[(row * cols) + col] }
      set { grid[(row * cols) * col] = newValue} }
        }
let m = Matrix(rows: 5, cols: 5)
m[3, 3] = 10
print(m[3,3])
```

3.32 Strong vs Weak References

uses ARC, it's a form of garbage collection but different from Java's Mark and Sweep. Benefits: Deterministic destruction, better for real time applications where you dont want garbage collection pauses. Drawbacks: there can be strong reference cycle = memory leaks. How it works: reference count for each class instance. New reference points to an instance = increment. Reference goes out of scope = decrement. When counter is 0 = deallocate. (only for reference types such as class but not struct!)

```
class ClassA {
   var b: ClassB?
   //weak var b: ClassB? // must be class type,
        optional, variable not left-constant, is
        nill when deallocated, no increment!
   deinit {print("ClassB")} }
class ClassB {
   var a: ClassA?
   //weak var a: ClassA?
   deinit {print("ClassA")} }
func f() {
   let a = ClassA(), b = ClassB()
   a.b = b // +1 but +0 if weak ref
   b.a = a  } // +1 if out of scope still 1 = leak
```

3.33 Access Control

TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO

3.34 structs

value types, dont support inheritance, can adopt 0 or more protocols, can be generic, initializers but no deinitializers. Int, Double,. Bool, String, Array<T> are implemented with structs.

```
struct Person {
   var name: String }
let p1 = Person(name: "Tim")
p1 = Person(name: "Tom") //error
p1.name = "Tom" //error
var p2 = p1 // mutable copy of p1
p2.name = "Tom" // ok
```

3.35 Copy-on Write Example

in objective C many types immutable and mutable variant. Are all reference types. inherit from their immutable counter part. swift prefers value types and uses copy on write to only make deep copies when needed.

```
import Foundation // objective C class
struct MyData {
   var data = Box(NSMutableData()) // Buffer
   var dataForWriting: NSMutableData {
      mutating get { // non mutable by default
        if isKnownUniquelyReference(&data) {
          return data.value }
        data = Box(data.value.mutableCopy() as!
        NSMutableData)
        return data.value }}
   mutating func append(_ bytes: [UInt8]) { //
        makes copy if needed
      dataForWriting.append(bytes, length: bytes.
        count) }  }
class Box<T> { // isKnownUniq... only works with
        swift
   let value: T  // needs helper class
   init(_ value: T) {
      self.value = value } }
var data = MyData()
var copy = data // shallow copy
for _ in 0..<10 { // only 1. it deep copy
   data.append([0x0b,0xad,0xf0,0x0d])}
```

3.36 Enums

```
public enum Optional<Wrapped> {
   case none
   case some(Wrapped) }
---
import Foundation
enum Result<T> {
   case success(T)
   casse error(String) }
func fetch(_ urlString: String) -> Result<String> {
   guard let url = URL(string: urlString) else {
        return .error("invalid")}
   guard let html = try? String(contentsOf: url,
        encoding .utf8) else {
        return .error("connection error") }
   return .success(html) }
let result = fetch("http://example.com)
switch result {
   case .success(let html):
      print(html)
   case .error(let message):
      print(message)
}
```

3.37 Operators

Most are defined in STL but assignment operators. Can overload existing op for own types. Can add new. pre-post-infix. Postfix > Prefix > Infix. Precedence groups: Multiplication (*,&,%) > Addition (+,& +,|,hoch) > Casting(as,as?,is) > Comparison > LogicalConjunction > LogicalDisjunction (||) = Default > Ternary (?:) > Assignment.

3.38 Overloading an existing prefix / infix operators

```
struct Vec2D {
   var x: Int
   var y: Int }
prefix func -(v: Vec2D) -> Vec2D {
   return Vec2D(x: -v.x, y: -v.y) }
let v1 = Vec2D(x: 1, y: 2)
let v2 = Vec2D(x: 4, y: 2)
print(-v1) // -1, -2
//func +(lhs: Vec2D, rhs: Vec2D) -> Vec2D {
//   return Vec2D(x: lhs.x + rhs.x, y: lhs.y + rhs.
        y) }
static func +(lhs: Vec2D, rhs: Vec2D) -> Vec2D { //
        more performant, typechecker only needs to
        look in here
   return Vec2D(x: lhs.x + rhs.x, y: lhs.y + rhs.y)
        }
print(v1 + v2)
```

3.39 Adding a new prefix / postfix / infix Operator

```
postfix operator ++
prefix operator ++
prefix func ++(x: inout Int) -> Int {
x += 1
return x}
postfix func ++(x: inout Int) -> Int {
let oldx = x
x += 1
return oldx }
---
infix operator ** // Default Presedence
func **(lhs: Int, rhs: Int) -> Int {
   return Array(repeating: lhs, count: rhs).reduce
        (1,*) }
print (10 ** 3 ** 2) // left or right first? add ()
---
infix operator **: MultiplicationPrecedence
func **(lhs: Int, rhs: Int) -> Int {
return Array(repeating: lhs, count: rhs).reduce
        (1,*) }
```

3.40 Protocols like interface in java (struct, enum, class)

```
// can require properties, methods, initializers,
        subsripts or associated types
// comparable and hashable inherit from equatable
public protocol CustomStringConvertible{
   var description: String{ get } } //requirement
---
struct Person: CustomStringConvertible {
   var name: String
   var age: Int
   var description: String {
      return "\(name) (\(age)) yrs old)"  } }
let p = Person(name: "Wait", age: 50)
print(p)  // Walt (50 years old)
---
public protocol Equatable {
   static func ==(lhs: Self, rhs: Self) -> Bool }
public func !=<T: Equatable<lhs: T, rhs: T) -> Bool
        {
   return !(lhs == rhs) }
---
struct Point: Equatable {  //  != is for free
   var x: Int
   var y: Int
   static func ==(lhs: Point, rhs: Point) -> Bool {
      return lhs.x == rhs.x && lhs.y == rhs.y } }
---
public protocol ExpressibleByArrayLiterl {
   assisiotedtype Element
   init(arrayLiteral elements: Element...) }
---
struct MyCollection<T>: ExpressibleByArrayLiteral {
   let elements: [T]
   init(arrayLiteral elements T...) {
      self.elements = elements } }
let mc: MyCollection<Int> = [1, 2, 3]
```

3.41 Extensions

add new computed property, initializer, method or subscript to existing type (class, struct, enum or protocol). also used to group related methods (e.g. methods required by the same protocol). Also works for stl types.

```
extenstion Int {
   func times(_ action: () -> ()) {
      for _ in 0..<self {
         action() } } }
5000.times {
   print("Please hold the line.") }
---
extension Sequence where Iterator.Element == Int {
   func average() -> Double {
      var sum = 0, count = 0
      for n in self {
         sum += n
         count += 1 }
      return Double(sum) / Double(count) } }
let range = 1...6 // or array [1,2,3]
print(range.average())
```

3.42 Protocol Extension

classes have many drawbacks: implicit sharing because of reference semantics, inheritance leads to high coupling between related classes. benefits of protocol oriented programming: works with value types (structs, enums) and ref types. less coupling, static type relationship. first step for a new abstraction should always be a protocol.

```
protocol Human {
var firse: String { get }
var last: String { get }
var age: Int { get } }
extension Human {
var fullName: String { return first + " " + last }
func isAdult() -> Bool { return age >= 18 } }
struct Person: Human {
var first: String
var last: String
var age: Int }
```

3.43 Sequence

may be destructive, infinite. All sequences = map(), reduce(), filter(), reversed(). With equatable elements: contains(), starts(with:). With Comparable: max(), min(), lexicographicallyPrecedes(). Collection = sequence whose elements can be traversed multiple times, nondestructively and accessed by indexed subscript. (inherits from sequence, must be finite). BidirectionalCollection = supports backward and forward traversal (inherits from collection). RandomAccessCollection = efficient random-access index traversal (inherits from bidirectional).

```
public protocol Sequence {
   associatedtype Iterator : IteratorProtocol
   func makeIterator() -> Iterator }
public protocol IteratorProtocol {
   associatedtype Element
   mutating func next() -> Element? }
---
struct FibonacciSequence: Sequence {
   let count: Int
   func makeIterator() -> FibonacciIterator {
      return FibonacciIterator(self) } }
struct FibonacciIterator: IteratorProtocol {
   var previous = 0, current = 1, remaining: Int
   init(_ sequence: FibonacciSequence) { self.
        remaining = sequence.count }
   mutating func next() -> Int? {
      guard remaining > 0 else { return nil }
      defer {
         (previous, current) = (current, previous +
         current)
         remaining -= 1 }
      return current } }
let numbers = FibonacciSequence(count: 10)
for n in numbers { print(n) }
//print(//numbers.reversed() // contains(13)
print(numbers.filter { $0 % 2 == 0 } )
```

3.44 Mutating Method

Explanation: In struct types, we need to tell the compiler, which methods are mutating the state of the instance.In the example below, the method inc() increments the stored property count and is therefore clearly altering the state of the Counter instance. Thus, it has

to be marked with the 'mutating' modifier.If we would create a new Counter instance with the let keyword, we could not call the inc() method.This makes sense, because let means that the instance should be immutable and inc() is a mutating method. can not be called for instances of this struct that are declared with let. Same concept as C++ const. Property setters are implicitly mutating.

```
struct Counter {
private(set) var count: Int

mutating func inc() {
count += 1 } }

var counter = Counter(count: 0)
print(counter.count)
counter.inc()
counter.inc()
print(counter.count)
```

3.45 AutoClosure

We expect that the logical conjunction operator has the same short-circuiting behaviour as in other languages. In other words, when the first operand evaluates to false, the second operand is not evaluated, because it's already clear that the result of the entire expression will be false. The way this is implemented in Swift is with a closure that has an autoclosure attribute. This way, the second operand is automatically wrapped inside a closure which will only be called, when lhs is true: infix operator &&&: LogicalConjunctionPrecedence

```
func &&&(lhs: Bool, rhs: @autoclosure () -> Bool)
        -> Bool {
if lhs {
return rhs() }
return false }
func f() -> Bool {
print("f() is called")
return true }
print(true &&& f())     // f() is called; result is
        true
print(false &&& f())    // f() is not called;
        result is false
```

3.46 Application Delegate

@UIApplicationMain attribute creates entry point to your app and a run loop that delivers input events to your app.

```
import UIKit
@UIApplicationMain
class AppDelegate: UIResponder,
        UIApplicationDelegate  {
var window: UIWindow?
func application(_ application: UIApplication,
didFinishLaunchingWithOptions launchOptions: [
        UIApplicationLaunchOptionsKey : Any]? = nil
        ) -> Bool {
window = UIWindow(frame: UIScreen.main.bounds)
window?.rootViewController = ViewController()
window?.makeKeyAndVisible()
return true } }
```

3.47 Configuring the Navigation Bar

```
class ViewControlle: UIViewController {
override func viewDidLoad() {
super.viewDidLoad()
title = "Hello, world" // implicitly sets
        naviationItem.title
let rithItem = UIBarButtonItem(barButtonSystemItem:
        .play, target: self, action: #selector(
        play))
navigationItem.rightBarButtonItem = rightItem }
func play() { print("play something")} }
```

3.48 Preparing a segue

```
override func prepare(for segue: UIStoryboardSegue,
         sender: Any?) {
switch segue.identifier! {
case "ShowAddShowTableViewController":
let nc = segue.destination as!
        UINavigationController
let tvc = nc.topViewController as!
        AddShowTableViewController
tvc.coreDataStack = coreDataStack
case "ShowEpisodes":
let tvc = segue.destination as!
        EpisodeViewController
guard let indexPath = tableView.
        indexPathForSelectedRow else {return}
tvc.show = fetchedResultsController.object(at:
        indexPath)
default:
fatalError() } }
```

3.49 TODO EXAMPLE FOR UI (AUTOLAYOUT)-> FROM EXERCISES (IT WILL BE IN THE EXAM)

3.50 Common Views and Controls

```
override func viewDidLoad() {
super.viewDidLoad()
let label = UILabel()
let button = UIButton(type: .custom)
let file = UITextField()
let image = UIImage(named: "kitten")
let iv = UIImageView(image: image)
view.addSubView(iv) // or label button...
label.text = "Hello, World"
label.font = UIFont(name: "Chalkduster", size: 40)
label.textColor = UIColor.orange
button.setTitle("Do Something", for: .normal)
button.setTitleColor(UIColor.purple, for: .
        highlighted)
button.addTarget(self, action: #selector(
        doSomething), for: .touchUpInside)
field.borderStyle = .roundedRect
field.placeholder = "Username"
field.addTarget(self, action: #selector(doSomething
        ), for: .editingChanged)
//translateAutoresizingMaskIntoContraints false,
        leftAnchor.constraint, right.}
fund doSomething(sender: UITextField) { //empty for
         Button
if let text = sender.text { print(test) } }
```

3.51 Outlet and Actions

```
import UIKit
class ViewController: UIViewController {
@IBOutlet weak var nameLabel: UILabel!
override func viewDidLoad() {
super.viewDidLoad()
nameLabel.text = "Tom"} }
@IBAction func buttonPressed(_ sender: AnyObject){
..} // attibute ignored by compiler, par could also
        be UIButton
```

3.52 TableViews

```
//example 1 without sections
class ViewController: UITableViewController {
let months = ["January", "February", ...]
override func tableView(_ tableView: UITableView,
        numberOfRowsInSection section: Int) -> Int{
return months.count}
override func tableView(_ tableView: UITableView,
        cellForRowAt indexPath: indexPath) ->
        UITableViewCell {
let cell = tableView.dequeueReusableCell(
        withIdentifier: "CellIdentifier", for:
        indexPath)
cell.textLabel?.text = months[indexPath.row]
cell.accessoryType = .disclosureIndicator
return cell } }
override func tableView(_ tableView: UITableView,
        didSelectRowAt indexPath: IndexPath) {
tableView.deselectRow(at: indexPath, animated: true
        )
print("selected"\(months[indexPath.row])")} }
//example 2 with sections class not written again
let seasons = [Season(name: "Spring", months: ["Mar
        ","Apr","May"]),...]
override func numberOfSections(in tableView:
        UITableView) -> Int { return seasons.count}
override func tableView(_ tableView: UITableView,
        numberOfRowsInSection section: Int) -> Int
        {
return seasons[section].months.count}
override func tableView(_ tableView: UITableView,
        titleForHeaderInSection section: Int) ->
        String? {
return seasons[section].name}
override func tableView(_ tableView: UITableView,
        cellForRowAt indexPath: IndexPath) ->
        UITableViewCell {
let cell = tableView.dequeueReusableCell(
        withIdentifier: "CellIdentifier", for:
        indexPath)
cell.textLabel?.text = season[indexPath.section].
        months[indexPath.row]
return cell}
```

3.53 MVC

```
class GreetingViewController: UIViewController {
var person: Person!
@IBOutlet weak var greetingLabel: UILabel!
override func viewDidLoad() {
super.viewDidLoad()
greetingLabel.text = "Tap the button" }
@IBAction func didTapButton(_ sender: Any) {
greetingLabel.text = "Hello " + person.firstName} }
```

3.54 MVP

```
protocol GreetingView: class {
func setGreeting(_ greeting: String) }
class GreetingViewController: UIViewController,
        GreetingView {
var presenter: GreetingPresenter!
@IBOutlet weak var greetingLabel: UILabel!
override func viewDidLoad() {
super.viewDidLoad()
presenter.initializeUI() }
@IBAction func didTapButton(_ sender: Any) {
        presenter.showGreeting() }
func setGreeting(_ greeting: String) {
        greetingLabel.text = greeting } }

class GreetingPresenter {
weak var view: GreetingView?
let person: Person
init(view: GreetingView, person: Person) {
self.view = view
self.person = person }
func initializeUI() {
view?.setGreeting("Tap the button") }
func showGreeting() {
let greeting = "Hello" + " " + person.firstName + "
        " + person.lastName
view?.setGreeting(greeting) } }

class GreetingMVPTests: XCTestCase {
class MockGreetingView: GreetingView {
var greeting: String!
func setGreeting(_ greeting: String) {
self.greeting = greeting } }
func testShowGreeting() {
let view = MockGreetingView()
let presenter = GreetingPresenter(view: view,
person: Person(firstName:"First", lastName:"Last"))
presenter.showGreeting()
XCTAssertEqual("Hello First Last", view.greeting) }
// more tests... }
```

3.55 MVVM

```
import RxSwift
class GreetingViewModel: NSObject {
let person: Person
let greetingText = Variable<String>("")
init(person: Person) {
self.person = person }
func initializeUI() {
greetingText.value = "Tap the button" }
func showGreeting() {
greetingText.value = "Hello " + person.firstName +
        " " + person.lastName } }

import UIKit
import RxSwift
import RxCocoa
class GreetingViewController: UIViewController {
var vm: GreetingViewModel!
let disposeBag = DisposeBag() // removes observer
        when view controller is deinitialized
@IBOutlet weak var greetingLabel: UILabel!
@IBOutlet weak var button: UIButton!
override func viewDidLoad() {
super.viewDidLoad()
vm.initializeUI()
button.addTarget(vm, action: #selector(vm.
        showGreeting), for: .touchUpInside)
vm.greetingText.asObservable().bindTo(greetingLabel
        .rx.text)
.addDisposableTo(disposeBag) } }

class GreetingMVVMTests: XCTestCase {
func testInitializeUI() {
let vm = GreetingViewModel(person: Person(firstName
        : "First", lastName: "Last"))
vm.initializeUI()
XCTAssertEqual("Tap the button", vm.greetingText.
        value) }
func testShowGreeting() {
let vm = GreetingViewModel(person: Person(firstName
        : "First", lastName: "Last"))
vm.showGreeting()
XCTAssertEqual("Hello First Last", vm.greetingText.
        value)} }
```

3.56 Contacts

```
//AppDelegate.swift
import UIKit

@UIApplicationMain
class AppDelegate: UIResponder,
        UIApplicationDelegate {

var window: UIWindow?
}
//PeopleViewController.swift
import UIKit

class PeopleViewController: UITableViewController {
let people = [Person(name: "Anna", birthday: "
        01.05.1955", phone: "012 345 67 89", email:
         "anna@example.com"),
Person(name: "Jenny", birthday: "17.09.2001", phone
        : "012 345 67 89", email: "jenny@example.
        com"),
Person(name: "John", birthday: "10.07.1975", phone:
         "012 345 67 89", email: "john@example.com"
        ),
Person(name: "Tom", birthday: "05.10.1988", phone:
        "012 345 67 89", email: "tom@example.com"),
Person(name: "Walter", birthday: "24.12.1969",
        phone: "012 345 67 89", email: "
        walter@example.com")]

override func tableView(_ tableView: UITableView,
        numberOfRowsInSection section: Int) -> Int
        {
return people.count
}

override func tableView(_ tableView: UITableView,
        cellForRowAt indexPath: IndexPath) ->
        UITableViewCell {
let cell = tableView.dequeueReusableCell(
        withIdentifier: "ContactCell", for:
        indexPath)
let person = people[indexPath.row]
cell.textLabel?.text = person.name
cell.accessoryType = .disclosureIndicator
return cell
}

override func prepare(for segue: UIStoryboardSegue,
         sender: Any?) {
switch segue.identifier! {
case "ShowPerson":
let personViewController = segue.destination as!
        PersonViewController
personViewController.person = people[tableView.
        indexPathForSelectedRow!.row]
default:
fatalError()
}
}
}
//Person.swift
import Foundation

struct Person {
let name: String
let birthday: String
let phone: String
let email: String
}
//PersonViewController
import UIKit

class PersonViewController: UIViewController {
var person: Person!

@IBOutlet weak var nameLabel: UILabel!
@IBOutlet weak var birthdayLabel: UILabel!
@IBOutlet weak var phoneLabel: UILabel!
@IBOutlet weak var emailLabel: UILabel!

override func viewDidLoad() {
super.viewDidLoad()

title = person.name
nameLabel.text = person.name
birthdayLabel.text = person.birthday
phoneLabel.text = person.phone
emailLabel.text = person.email
}
}
```

3.57 REST Countries

```
//APPDelegate.swift
import UIKit

@UIApplicationMain
class AppDelegate: UIResponder,
        UIApplicationDelegate {
var window: UIWindow?
}
//APIClient.swift
import Foundation

enum Result<T> {
case success(T)
case error(String)
}

final class APIClient {
let session: URLSession

init() {
let configuration = URLSessionConfiguration.default
configuration.httpAdditionalHeaders = ["Accept": "
        application/json"]
configuration.requestCachePolicy = .
        reloadIgnoringLocalCacheData
session = URLSession(configuration: configuration)
}

func getCountries(callback: @escaping (Result<[
        Country]>) -> Void) {
let url = URL(string: "https://restcountries.eu/
        rest/v1/all")!
session.dataTask(with: url) { (data, response,
        error) in
let result = self.getResult(data: data, response:
        response, error: error)

OperationQueue.main.addOperation {
callback(result)
}
}.resume()
}

func getResult(data: Data?, response: URLResponse?,
         error: Error?) -> Result<[Country]> {
guard error == nil else {
return .error(error!.localizedDescription)
}

guard let response = response as? HTTPURLResponse,
200..<300 ~= response.statusCode,
let data = data else {
return .error("Server Error")
}

guard let json = try? JSONSerialization.jsonObject(
        with: data),
let countries = parseCountries(json) else {
return .error("Invalid data")
}

return .success(countries)
}

func parseCountries(_ json: Any) -> [Country]? {
guard let arrayOfJsonDicts = json as? [[String: Any
        ]] else { return nil }
return arrayOfJsonDicts.flatMap { Country(json: $0)
         }
}
}
// CountriesViewController.swift
import UIKit

class CountriesViewController:
        UITableViewController {
var countries: [Country] = []

override func viewDidLoad() {
super.viewDidLoad()

let client = APIClient()
client.getCountries { result in
switch result {
case .success(let countries):
self.countries = countries
self.tableView.reloadData()
case .error(let message):
let alertController = UIAlertController(title: "
        Error", message: message, preferredStyle: .
        alert)
alertController.addAction(UIAlertAction(title: "OK"
        , style: .default))
self.present(alertController, animated: true)
}
}
}

override func tableView(_ tableView: UITableView,
        numberOfRowsInSection section: Int) -> Int
        {
return countries.count
}

override func tableView(_ tableView: UITableView,
        cellForRowAt indexPath: IndexPath) ->
        UITableViewCell {
let cell = tableView.dequeueReusableCell(
        withIdentifier: "CountryCell", for:
        indexPath) as! CountryCell
let country = countries[indexPath.row]
cell.countryLabel.text = country.name

let capital = country.capital.isEmpty ? "N/A" :
        country.capital
cell.capitalLabel.text = "Capital: \(capital)"

let formatter = NumberFormatter()
formatter.groupingSeparator = "'"
formatter.usesGroupingSeparator = true
formatter.groupingSize = 3
let population = country.population == 0 ? "N/A" :
        formatter.string(from: country.population
        as NSNumber)!
cell.populationLabel.text = "Population: \(
        population)"

return cell
}
}
// Country.swift
struct Country {
let name: String
let capital: String
let population: Int

init?(json: [String: Any]) {
guard let name = json["name"] as? String,
let capital = json["capital"] as? String,
let population = json["population"] as? Int else {
return nil
}

self.name = name
self.capital = capital
self.population = population
}
}
// CountryCell.swift
import UIKit

class CountryCell: UITableViewCell {
@IBOutlet weak var countryLabel: UILabel!
@IBOutlet weak var capitalLabel: UILabel!
@IBOutlet weak var populationLabel: UILabel!
}
```

3.58 Auto Layout

```
import ÁÃÑ ãÃÑ UIKit
class ãÃÑ ãÃÑ ViewController ãÃÑ : ãÃÑ
        UIViewController ãÃÑ {
let ãÃÑ ãÃÑ container ãÃÑ = ãÃÑ UIView ãÃÑ ()
let ãÃÑ ãÃÑ label ãÃÑ = ãÃÑ UILabel ãÃÑ ()
let ãÃÑ ãÃÑ textField ãÃÑ = ãÃÑ UITextField ãÃÑ ()
let ãÃÑ ãÃÑ button ãÃÑ = ãÃÑ UIButton ãÃÑ ()
override ãÃÑ ãÃÑ func ãÃÑ ãÃÑ viewDidLoad ãÃÑ () {
super ãÃÑ .viewDidLoad()
view.addSubview(container)
container.backgroundColor = ãÃÑ UIColor ãÃÑ .orange
container.translatesAutoresizingMaskIntoConstraints
        = ãÃÑ false
container.leftAnchor.constraint(equalTo: view.
        leftAnchor, constant: ãÃÑ 20 ãÃÑ ).isActive
        = ãÃÑ true
container.centerXAnchor.constraint(equalTo: view.
        centerXAnchor).isActive = ãÃÑ true
container.heightAnchor.constraint(equalToConstant:
        ãÃÑ 100.0 ãÃÑ ).isActive = ãÃÑ true
container.centerYAnchor.constraint(equalTo: view.
        centerYAnchor).isActive = ãÃÑ true
view.addSubview(label)
label.text = ãÃÑ "Login-Form"
label.translatesAutoresizingMaskIntoConstraints =
        ãÃÑ false
label.leftAnchor.constraint(equalTo: container.
        leftAnchor, constant: ãÃÑ 5 ãÃÑ ).isActive
        = ãÃÑ true
label.bottomAnchor.constraint(equalTo: container.
        topAnchor, constant: ãÃÑ - ãÃÑ 5 ãÃÑ ).
        isActive = ãÃÑ true
container.addSubview(textField)
textField.placeholder = ãÃÑ "Enter Password"
textField.borderStyle = .roundedRect
textField.translatesAutoresizingMaskIntoConstraints
        = ãÃÑ false
textField.widthAnchor.constraint(equalTo: container
        .widthAnchor, multiplier: ãÃÑ 0.5 ãÃÑ ).
        isActive = ãÃÑ true
textField.centerXAnchor.constraint(equalTo:
        container.centerXAnchor).isActive = ãÃÑ
        true
textField.topAnchor.constraint(equalTo: container.
        topAnchor, constant: ãÃÑ 20 ãÃÑ ).isActive
        = ãÃÑ true
}
container.addSubview(button)
button.setTitle( ãÃÑ "Login" ãÃÑ , ãÃÑ for ãÃÑ : .
        normal)
button.setTitleColor(.blue, ãÃÑ for ãÃÑ : .normal)
button.translatesAutoresizingMaskIntoConstraints =
        ãÃÑ false
button.centerXAnchor.constraint(equalTo: container.
        centerXAnchor).isActive = ãÃÑ true
button.topAnchor.constraint(equalTo: textField.
        bottomAnchor, constant: ãÃÑ 10 ãÃÑ ).
        isActive = ãÃÑ true
}
```