Mobile Application Development

Notes Repack

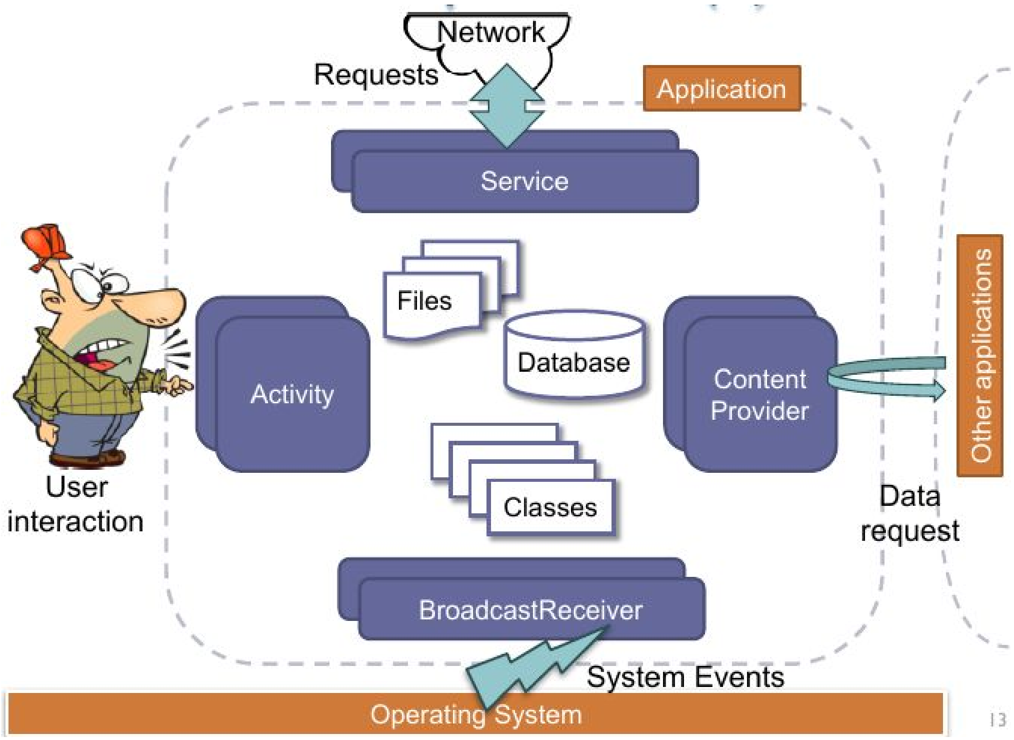
by N.M.Scaraffia and L.Gabriele

Contents

1 Android main features

+ designed to be robust and secure
|→ for each app a process, a virtual machine instance ~ ART - Dalvik ~, a user
|→ no difference in system app and normal app
|→ Android apps **does not have a single entry point as normal programs**

|→ Activity	→ user interaction
|→ Service	→ long tasks in background ~ network requests
|→ Content Provider	→ manages app data and other applications interactions
|→ BroadcastListener	→ it waits for messages; system events



2 Manifest file

an XML file used as "Contract" between the app and the execution environment containing permissions and app configuration

- if some apps try to use a system component without requesting the permission the app will crash
- it is read at installation time
- All the components exported by an application are listed here (intent reactions) together with zero or more intent-filters

3 Application process startup

During app startup for each application an *Application* object is instantiated disregarding to the application type (Activity, Service, etc...) and it will be in execution till the end.

<u>Startup calls:</u>

1. **[user tap]** Information about the chosen activity is packed in an Intent, which is then delivered to a system process named Zygote, which takes care of creating all other processes via forks.
2. **[zygote fork ~ like init]** Zygote has a pre-warmed VM, which makes the startup time faster, because only application specific classes need to be instantiated, while general purpose ones have already been loaded by the parent process.
3. **[message loop]** The forked process starts a message loop and by using the intent locates and loads the proper apk and manifest file.
4. **[Application's OnCreate()]** The main thread of the forked process instantiates an Application object and invokes the OnCreate() method.
5. **[Activity's OnCreate()]** The main Activity is then instantiated, its OnCreate() method is invoked and the application starts running.

4 Intents

An intent is an abstract description of an action to be performed and it defines an <u>action</u>, a related set of <u>data</u> and a <u>category</u> (addition informations on intent management)

+ *<u>Why Intents as interprocess communication?</u>*

Because since android has been designed to be secure all the standard Linux/UNIX interprocess communication systems like pipes, dbus, etc.. have been removed

+ *<u>Intent dispatch</u>*

When an intent is delivered, Android tries to match it against all filters, in order to detect which component should be activated.

<u>DEF:</u> Asynchronous messaging mechanism used by the operating system to associate process requests with the appropriate activities and/or services

Intents can be of two kinds:

- Implicit: the intent doesn't have a precise target, but it will match all applications that registered as listeners to the parameters of the intent.
- Explicit: the intent targets a specific Activity by its class name. It is possible to add a Bundle to transfer data between the activities.

5 ADB

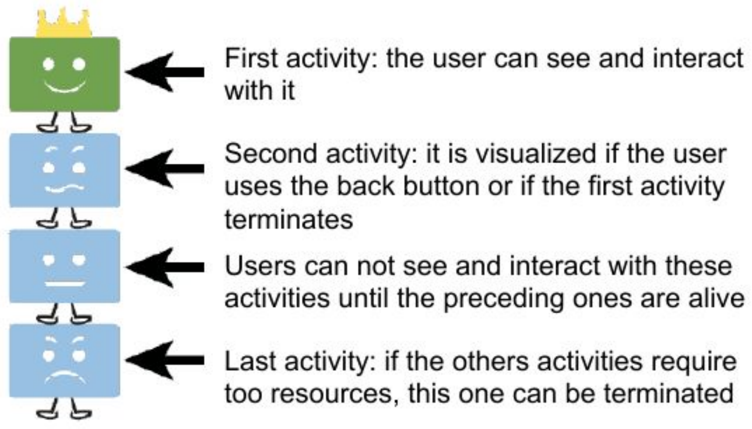
Android Debug Bridge, consists of three components:

- A client, which runs on the development machine and is invoked by the adb command.
- A server, which runs as a background process on the development machine and manages communication between the client and the adb daemon running on an emulator or device.
- A daemon, which runs as a background process on each emulator or device instance.

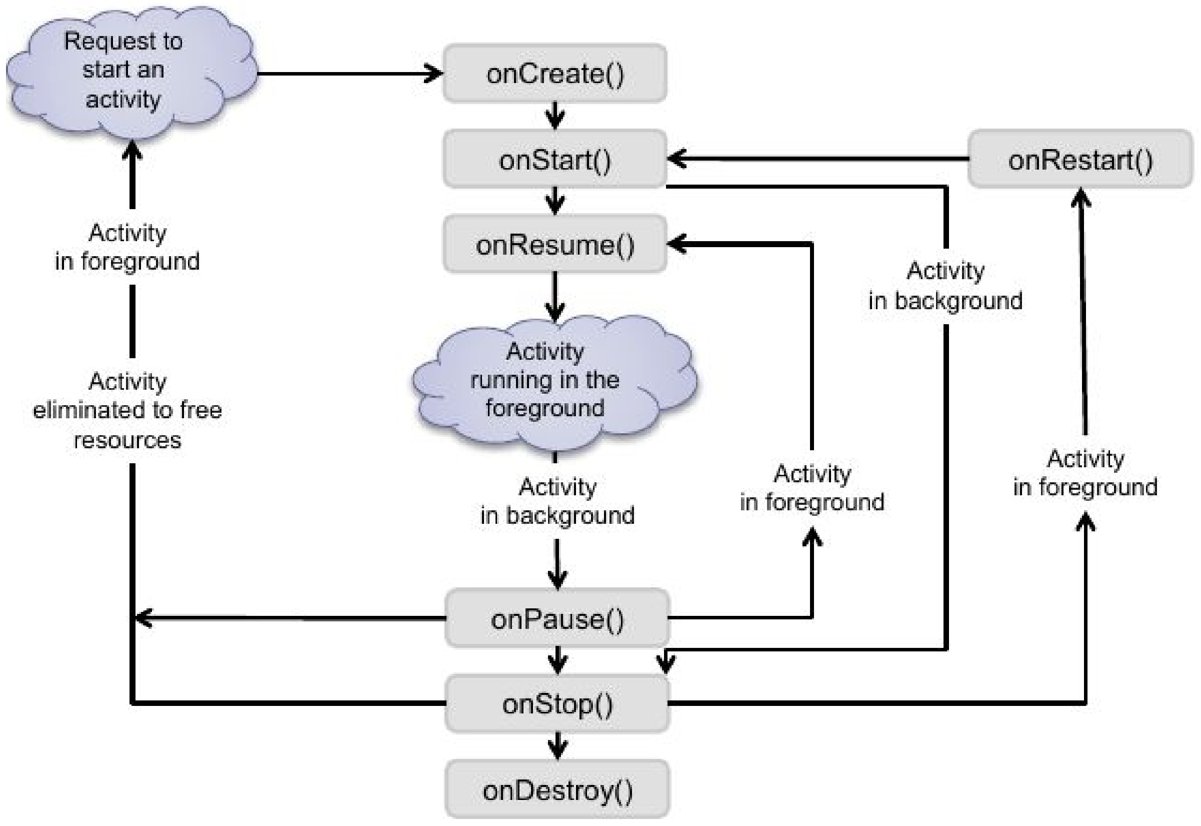
6 Activity

An activity manages user interaction:

- Acquires the required resources (requires the use of sensors, loads contents, ...)
- Builds and configures a graphical user interface
- Reacts to the events triggered by user interactions with widgets to implement the desired behavior
- Manages the notifications regarding its own life cycle, storing data collected by the interface and releasing resources when the operating system asks so
- *An activity plays the role of the controller in the MVC paradigm (in iOS this is made by UIViewController)*


First activity: the user can see and interact with it
Second activity: it is visualized if the user uses the back button or if the first activity terminates
Users can not see and interact with these activities until the preceding ones are alive
Last activity: if the others activities require too resources, this one can be terminated

6.1 Lifecycle



● onCreate - onDestroy	~ exists but invisible
● onStart - onStop	~ visible but the user can't interact
● onResume - onPause	~ visible and the user can interact

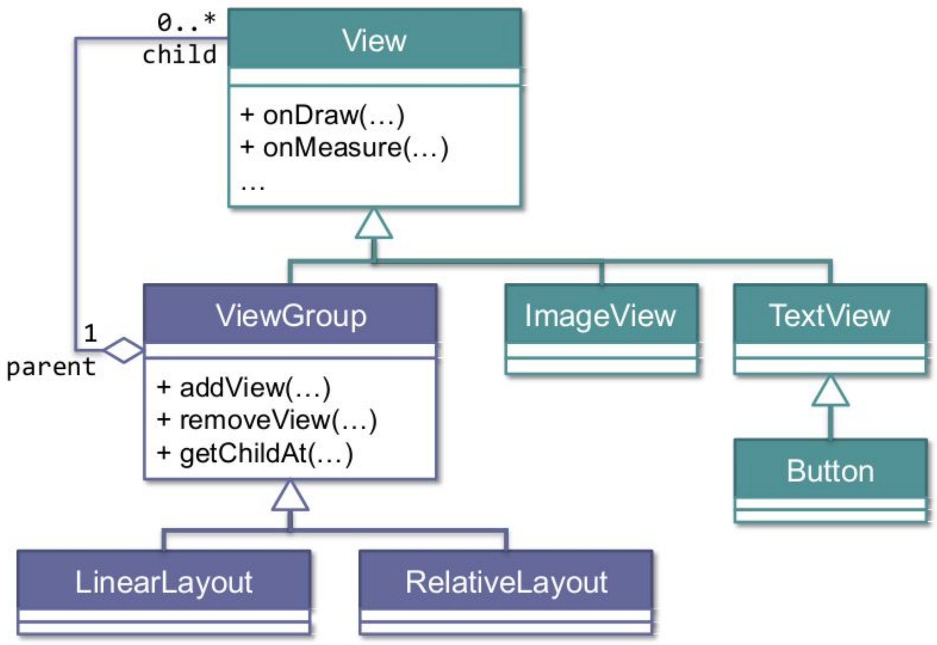
6.1.1 OnResume
In this method, it is possible to start animations, videos, sounds, ... , and to acquire temporary resources

6.1.2 OnPause
The onPause() method is called when the activity has to be moved to the second position of the activity stack: all the operations interacting with the user must be stopped, like sounds, videos and animations and data operations committed → it must be fast

6.1.3 OnStop
Called when the application is no longer visible to the user

6.1.4 OnDestroy
- The activity has been terminated and will be removed from memory
- The invocation may be the result of an explicit request of the application (calling finish())
- Or of the need to free resources

7 The composite pattern
Android, like most other GUI frameworks, uses the composite pattern to model a hierarchy of visual elements. There are two kind of elements that can appear in a tree:
- Elementary views, which are the "leaves" of the tree.
- Container views, which can contain elementary views or other containers.



To create a GUI it is possible to use an XML based approach, defining all elements in a XML file, or a programmatically approach, defining everything via code. Each method has pros and cons and usually a hybrid approach is used, with a base written in XML and customizations done via code.

8 Saving the state of an activity

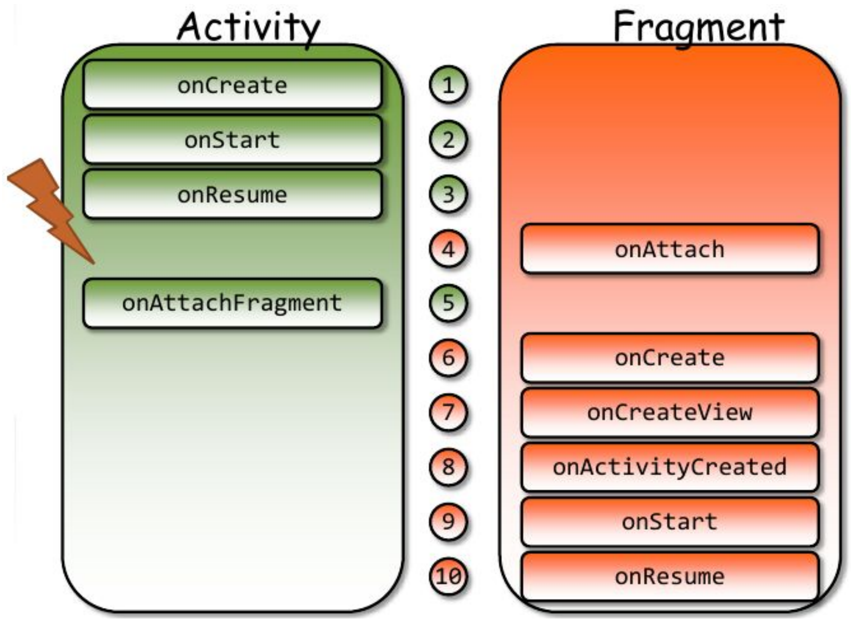
As we said, the operating system may decide to kill an application if on low memory. Luckily, Android provides a method that is called before this happens: OnSaveInstanceState(Bundle b), where it is possible to store primitive data or data implementing the Serializable or Parcelable interface.
To restore the values, it is possible to access the Bundle in many places, such as OnCreate() and OnRestoreInstanceState().

9 Context and resources

Activities, services and applications have a common root class: Context. This class provides the functionality needed to access resources and application-specific classes and to interact with the operating system. Each resource is identified by a unique number, auto-generated at compile time and stored in the R class. It is possible to access a resource using the getResources() method.

10 Fragments

A Fragment is an object that, conceptually, stands between Activity and View
> As an Activity, it has a complex life cycle, managed by several callbacks
> Encapsulates a hierarchy of views and, as a View, can be directly inserted into a xml layout
> In the absence of special actions, the fragment will follow the life cycle of the activity
> It will be destroyed when the activity will be destroyed
There exists an old compatibility version v4 (< 2.3) and an integrated one



11 Custom Views

11.1 Space negotiation

The space negotiation process involves two phases:

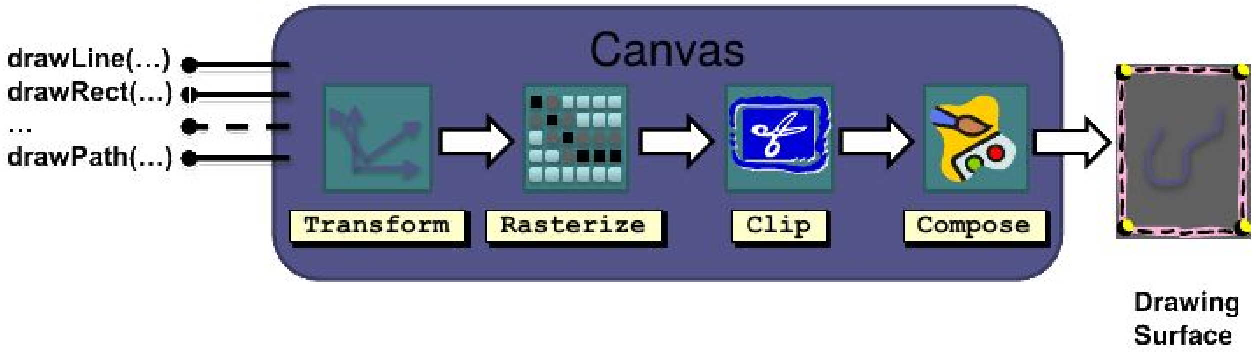
- Measuring all child views → involves all views
  - *measure()* method called by all views from the root; each view proposes a size and the container makes the computation for its each own size;
  - the *measure()* call involves *OnMeasure(x,y)* where x and y are the dimensions proposed by the container;
  - after it the child proposes the needed space by *setMeasuredDimensions(...)*; *OnMeasure()* without *setMeasuredDimensions()* generates an exception;
  - *OnMeasure()* may be called many times to negotiate the space when children do not agree to the proposed dimensions;
- Calculating their sizes and positions → only the containers
  - *OnLayout()* is called on the root of the view tree the will compute boundaries for each child and call *layout()* for each of them

11.2 Drawing phase

- *draw()* on the root >>> draws the background
  - *OnDraw()* for each single View >> contains the drawing logic
- *dispatchDraw()* called recursively on all children views
- *onDrawScrollbars()* only on root

11.2.1 OnDraw()

It gets a Canvas to use to draw bidimensional graphics; internally it performs the functions in the pipeline below:



The Canvas class provides a set of methods to perform geometric transformations like rotate, scale, skew, translate that can set a new new projection matrix or be concatenated to the current one.
Since OnDraw() is frequently called during animations it is better to not perform all the computations within it and avoid creating new objects here (reusing existing objects it is a good approach)

11.2.1.1 Path class

- *Abstract vector representation of a geometric shape* (rectangle, bezier curves, etc...)
- All rendering operations within a Canvas begins by creating a temporary object of type Path: this object is then manipulated on the basis of the attributes of the Paint associated with the draw operations and forms the basis of the subsequent processing operations

11.2.1.2 Paint class

Controls the graphics primitives rendering, providing the necessary parameters for the rasterization and composition done within the pipeline (color, style, fonts, effects, etc...)

11.2.1.3 Composition management

The result of the pipeline is combined with the drawing surface through a composition rule, like XOR between values considered opaque (setXfermode() method of the object Paint)

11.2.1.4 Drawable

- Abstract class that models a generic, not interactive, graphical content
- It does not receive any direct notification regarding user-related actions or system-related events (positioning, visibility, ...)
- It can have different states (driven by the application code) corresponding to different visualizations
- It encapsulates the concept of "level" to express a numerical quantity in a graphical way

12 Touch events

multitouch available since v2.2

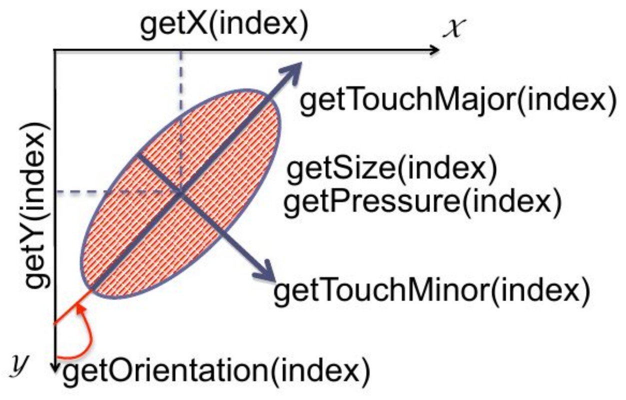
onTouchEvent(...) callback to locate the movements in a view

OR

GestureDetector

ScaleGestureDetector

Each touch is modeled as an ellipse labeled with a unique ID



12.1 Delivery process

Since views are organized hierarchically in the view tree, Android performs a depth-first search to determine if and by whom a given event should be processed;

>>> *The event is first notified to the current activity that forwards it to the root of the view tree*

The tree is first visited downward… >>> dispatchTouchEvent(...)

Then it is visited upward >>> Calling onTouchEvent(...)

If onTouchEvent(...) returns true, the component is selected as the handler of the event *and the upward visit is terminated*

IF NO TRUE > sent to activity

IF TRUE > sub-views NO notifications

13 Animations

- **Frame-by-frame** based on a set of Drawable resources, which will be shown one after the other - provided by the AnimationDrawable class
- **Tweening** perform a series of simple transformations (position, size, rotation, ...) on the contents of a view - contained in the android.view.animation package

13.1 Frame-by-frame

- Each image is displayed for a split of second, then you move on to the next
- can be created by XML or programmatically

13.2 Tweening

Two types:

- **Animations of the layout** connected to any view, they perform elementary operations on transparency or on the transformation matrix (Scale-, Rotate-) of a given view; They notify an observer of the progress of the animation
- **Animated effects on components** Affect the area of the canvas of the component - must be invoked manually

Two classes:

- **animation class**: older, with limits: Animation operates only at View level: everything about the logic of the interaction is not affected (e.g. If a button is moved, for example, the sensitive area remains the original one)
- **animator class:** introduced since v3.0 - changes the value of a property of an object, interpolating between a set of given values
    - it has an interpolator
    - several choreographies

14 Android threading

The threads created at start time are:

- Main app thread
- **GC** - low priority garbage collector
- **JDWP** - memory inspection and debugging
- **Compiler** - just in time compilation
- **ReferenceQueueD, FinalizerDeamon, FinalizerWatchd** - object finalization and memory cleanup
- **Binder_x** - intent manager and other IPC requests
- **Signal catcher** - QUIT USR1 USR2 receiver
- **GL updater**
- **hwUITask** - Handle incoming messages from UI

14.1 Main thread

Contains a message queue that uses to dispatch messages and signalling events to the various components. Responsible for:

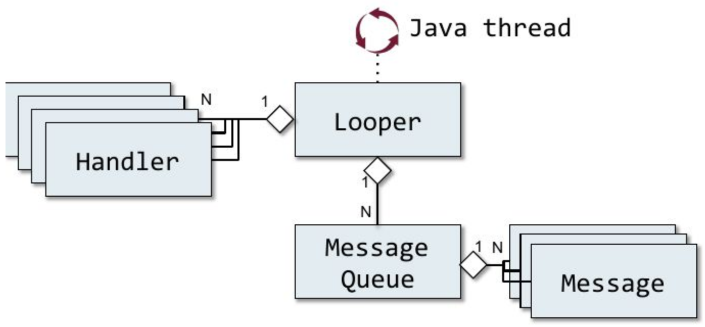
- Instantiating the Application and Activity objects
- Notifying them the events related to their life cycle
- Sending drawing requests to the views
- Delivering to the suitable listeners the events related to user interaction

The access to the queue is synchronized and all the operations must be quick

14.2 General threading

- Threads are not aware of the application lifecycle
- they cannot access to the view tree

14.3 Looper



Class that manages a message queue associated with a single thread

- The queue is not directly accessible to the programmer, but it is created through the Looper.prepare() static method
- The thread that invokes this method becomes the owner of the queue
- When the Looper.loop() static method is called, an infinite loop starts, waiting for messages from other threads

- Looper threads naturally lend themselves to implement the producer/consumer pattern

14.4 MessageQueue

Unbounded linked list of messages to be processed on the consumer thread

>> Every looper (and thread) has at most one MessageQueue { >> *Looper.myQueue();* }

14.5 Handler

The handler mechanism can be used to asynchronously communicate with a thread

A looper thread can be associated with many different handlers, but they all insert messages in the same queue; when the message will be picked up, it will be processed by the handler via which it was inserted

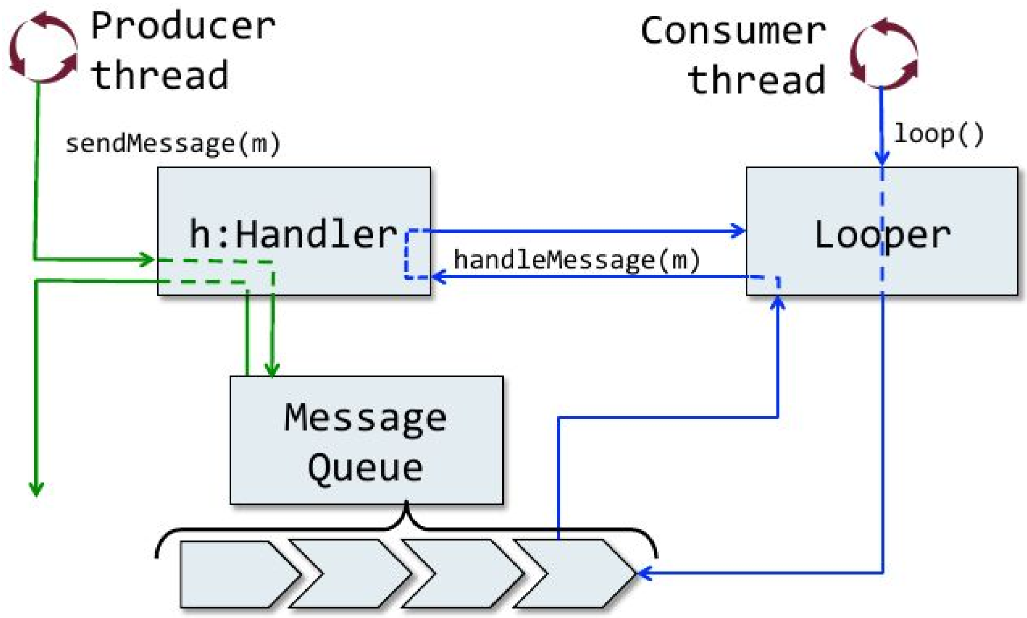
14.6 Message

Models an event described by an integer value, a pair of (integers) parameters and an eventual object to be delivered to the recipient

- The use of integer values makes efficient the use and reuse of objects of this type
- Given the allocation and releasing cost of new items, the operating system maintains a queue of messages that are not in use

14.7 Producer-consumer pattern

- A producer thread asynchronously generates pieces of information, storing them in Message objects
- Messages are inserted in the MessageQueue of the consumer via Handler sendMessage() method
- Consumer thread extracts one message at a time and dispatches it to the proper handler that process it



14.8 Sending messages to the main thread

- **activity.runOnUiThread(Runnable r)**
  - When this method is executed, if the current thread is not the main one, a new message that encapsulates a Runnable object is inserted into the queue
  - When the message will be processed, the main thread will invokes the run() method of its parameter, and the contained code will be executed
  - It requires the secondary thread to hold a reference to the current ongoing activity
- **view.postRunnable(Runnable r)**
  - Inserts the Runnable object in the message queue
  - It can be called from any thread as long as the view is displayed inside a window
- **view.postDelayed(Runnable r, long l)**
  - Similarly to the previous case, it inserts the object into the queue, but it will not be processed before an interval equal to "l" milliseconds
  - In this case, too, the view must be connected to a window and be visible
- **queue messages via** [Using instances of the Handler class, created in the main thread]
  - sendMessage(...)
  - sendMessageDelayed(...)
  - sendMessageAtTime(...)
  - sendMessageAtFrontOfQueue(...)
- **forward a Runnable object via** [Using instances of the Handler class, created in the main thread]
  - post(...)
  - postDelayed(...)
  - postAtTime(...)
  - postAtFrontOfQueue(...)

14.9 HandlerThread

Helper class that builds a secondary thread that incorporates a Looper and a MessageQueue

14.10 AsyncTask

Abstract and parametric class that allows you to define in a simple way activities to be performed in a secondary thread and offering at the same time a set of methods that will be invoked from the main thread in the appropriate time.
To invoke an AsyncTask it is necessary to subclass it and create a new instance on which the execute() method is called. This is a “one shot” operation, meaning that to start again the same task it is necessary to create a new instance of it.
The AsyncTask subclass must specify three data types:

1. **Params:** the type of the parameter to be passed to the secondary thread
2. **Progress:** the data type that will be communicated to the main thread to indicate the progress of the activity

3. **Result:** the type of the result

An AsyncTask should at least implement the doInBackground() method, which contains the code that will be run in the secondary thread. Other methods such as OnPostExecute() are instead called in the main thread and allow to access Views.

It is possible to ask for the interruption of the secondary thread by invoking the cancel() method on the task from the main thread. This gives no guarantee that the task will be stopped, but it just sets a flag in the AsyncTask. It is duty of the programmer to check this flag through the isCancelled() method and exit the thread accordingly.

Great care must be taken to stop running tasks when the activity is destroyed, because otherwise the secondary threads will keep running and when trying to update Views, null pointer exceptions will be raised, because the old Views from the old activity are no longer present.

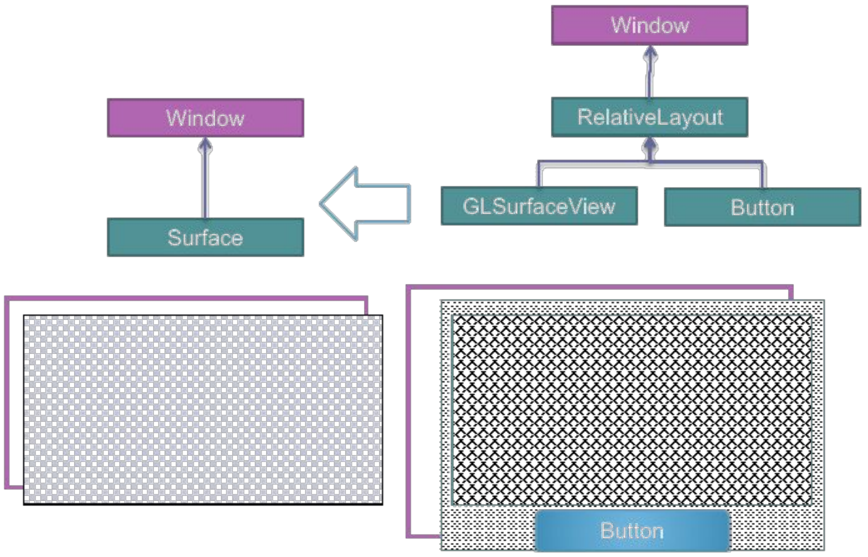
Another version of the execute() method, executeOnExecutor(), allows to specify the expected level of concurrency of the task: THREAD_POOL_EXECUTOR to execute each thread on a separate core or SERIAL_EXECUTOR to execute threads serially.

15 OpenGL ES

Android offers a complete graphics engine which pipeline is accessible via the Canvas class, but it is fully executed by the CPU; so to use device GPU hw acceleration OpenGL APIs are required.

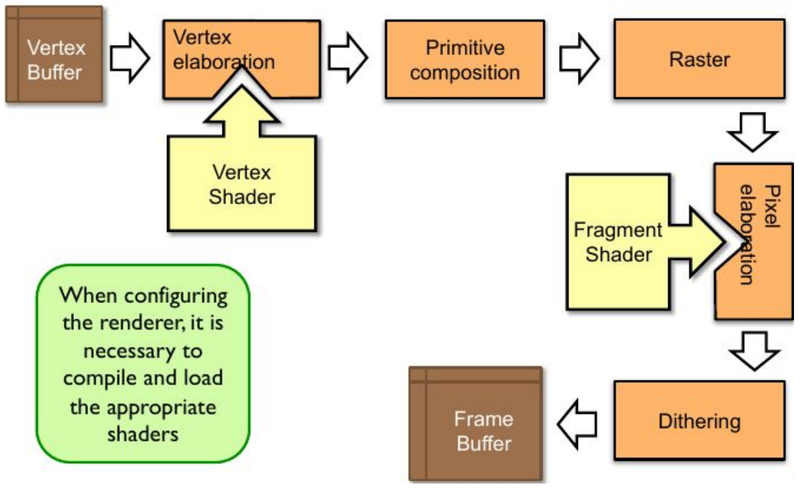
15.1 Visual hierarchy

To use the GPU-based rendering, it is necessary to create an appropriate surface (*GLSurfaceView*) ~ *window*



The *GLSurfaceView* class internally manages the creation of a secondary thread responsible for displaying the content; the *Renderer* methods are continuously called in the context of this thread.
The *onDrawFrame(...)* method is responsible to display the graphic content (many primitives are supported, except quadrilaterals).

15.2 OpenGL ES 2.0 pipeline



Unlike what happens in earlier versions of OpenGL, vertex informations (position, color coordinates, etc...) meaning is not predetermined > It is necessary to program the pipeline to indicate how such information is handled.

15.3 Shaders

Each shader is a program written in a special language (GLSL) that runs inside the GPU

- Receives some information as input
- It produces other information as output
- The GPU can run many instances of the shader at the same time

OpenGL ES 2.0 supports two types of shaders:

- **Vertex shaders** – transform the properties of each vertex (expressed w.r.t. an arbitrary reference system) in properties expressed with respect to the image plane
- **Fragment shaders** – calculate the properties of each pixel enclosed by the specific primitive which is bounded by a given group of vertices

15.3.1 Shader structure

- block of code + entry point (main())
- they can access special input and output variables as well as to a set of predefined constants to be able to interact with the other parts of the pipeline
- additional attributes can be passed

15.3.2 Vertex shader

It transforms vertex data from space of the model to that of the final image

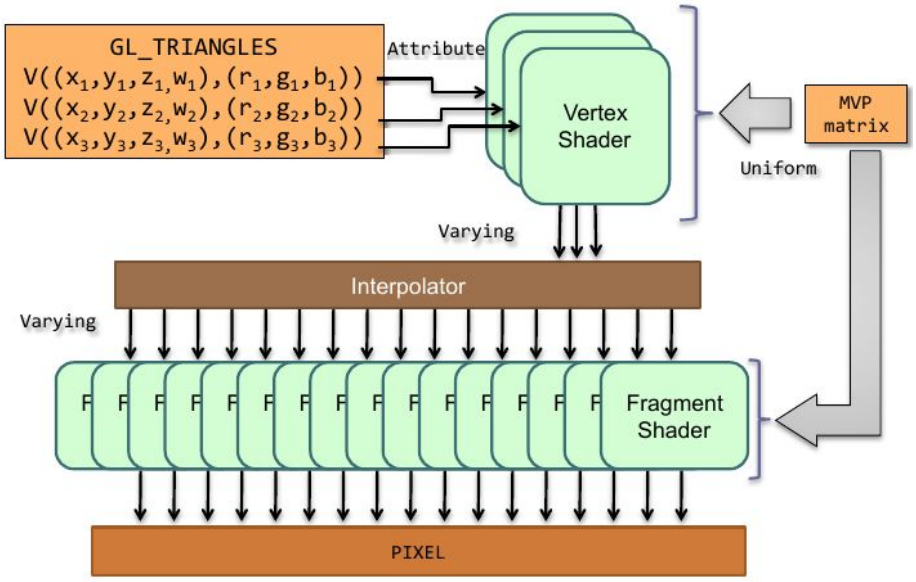
- It must set the predefined variable *gl_Position* with the coordinates of the point corresponding to the current vertex in the image space
- It receives information on the vertex via global variables of type "*attribute*"
- It receives context information via global variables of type "*uniform*" (read-only)
- It can produce more information to be placed in global variables of type "*varying*"

```
uniform     mat4  mvp_matrix; // projection  matrix
attribute   vec4 a_vertex;    // vertex position
attribute   vec3 a_color;     // vertex colour
varying     vec4 v_color;     // fragment colour

void main(void)
{
      v_color.xyz = a_color.xyz;   //colour to be interpolated
      v_color.w  = 1.0;            //completely opaque

      gl_Position = mvp_matrix * a_vertex;     //position to be interpolated
}
```

From primitives to pixels



15.3.3 Fragment shader

It assigns a color to a given pixel of a graphics primitive
Input variable: *gl_FragCoord*
Output variable: *glFragColor*

- It can use any algorithm to assign the output value
- It can use all the "*varying*" information made available by the vertex shader and the uniform passed to the pipeline

```
precision   mediump float;    // inizialize data precision
varying     vec4 v_color;     // color assigned to the fragment

void main(void)
{
      gl_FragColor = v_color; // assigned color, no other transform
}
```

The shaders are normally stored as a raw resource, in the form of text files and **compiled at runtime.**

15.3.4 Matrices

Vertex transformations are performed through matrix multiplication (4x4), which for performance reasons is stored as an array of 16 floating point numbers.
Usually, it is convenient to express the transformation matrix as a product of three separate matrixes: Model, View and Projection.

- **Model:** this matrix allows to rotate / translate / scale a shape to be placed on the scene

- **View:** this matrix defines the point of view by which the scene is observed
- **Projection:** this matrix defines how it is possible to move from the coordinates of the normalized space to those of the image. It can be orthogonal or perspective.

15.3.5 Textures

Images may be mapped on the faces of the triangles that form a solid to increase the degree of realism; they must be power of 2 (1024x1024 ~ 2048x2048).
A fragment shader refers to a texture through a *uniform* variable of type sampler2D.

15.3.6 Display thread

The events related to the interaction with the user are notified in the main thread; often these have implications on what needs to be visualized from the secondary thread

16 jPCT Framework

Free framework supporting 3D interactive visualizations for both Java SE and Android

- It relies on a GLSurfaceView and its corresponding Renderer to which scene creation and management is delegated.
- **FrameBuffer** - contains the representation of the scene
- **World** - contains the objects to be displayed, lights and camera, etc...; represent the "global reference system" where all objects are placed
    - geometry, visual appearance and collision strategies for objects is stored in instances of the *Object3D* class
    - *Object3D* (simple hierarchy), have their own vertex mesh, a texture and, possibly, a custom shader to control their rendering
    - the world class also contain a collection of Polyline objects
        - not considered for collisions
        - drawn after *Object3D*
    - the world class may contain several lights
    - the world class contains a single *Camera* object

16.1 Collisions

Several methods are available in class World to control collisions between a "material point" and objects that are part of the scene; *Object3D* should have a collision policy detection different from COLLISION_CHECK_NONE.
Collisions between the Camera and other Object3D inside the scene are supported too

16.2 Shaders

jPCT-AE comes with a set of default shaders that will be used if nothing else is specified:

- Ambient light
- Color transparency
- Fogging
- Up to 4 level texturing
- Up to 8 point lights

17 Networking

Java.net package: contains the classes that model and manage network connections at low level >>> usually not used because too complex
URLConnection: model high-level connections, providing support http, https, ftp, file, jar
NOTE: by default it does not manage coockie, but a CoockieManager can be associated
ApacheHttpClient libraries: offer richer and simpler functionalities; deprecated since andr. v22

17.1 Volley Library

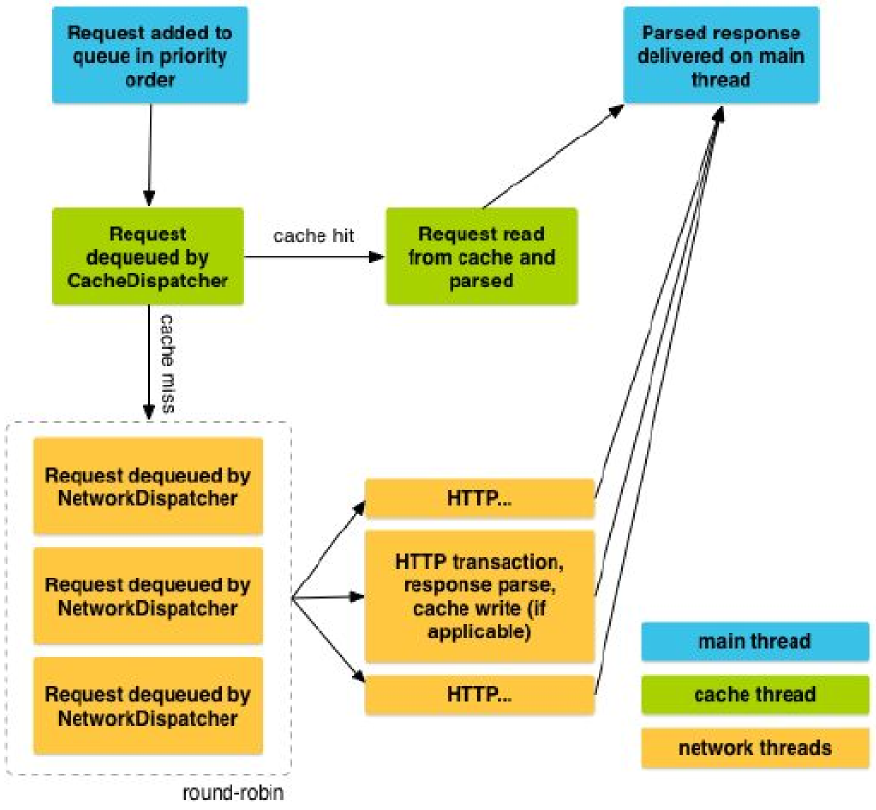
Main features

- Request scheduling via RequestQueue
- Multiple concurrent connections
- Transparent caching
- Cancellation API
- Automatic thread management
- Ease of customization

17.1.1 RequestQueue class

It provides **a way to collect requests and dispatch them to background worker threads**
Requires a *network transport* and a *cache implementation*.

Request Lifecycle


Request added to queue in priority order
Parsed response delivered on main thread
Request dequeued by CacheDispatcher
cache hit
Request read from cache and parsed
cache miss
Request dequeued by NetworkDispatcher
Request dequeued by NetworkDispatcher
Request dequeued by NetworkDispatcher
HTTP...
HTTP transaction, response parse, cache write (if applicable)
HTTP...
round-robin
main thread
cache thread
network threads

Summarizing: get a request, checks whether the response is already present in the cache or not; if no it delivers the request to a network thread in charge of performing the HTTP request and dispatching it to the initial requester.

17.1.2 Cancelling requests

Any Volley request can be cancelled, invoking the corresponding method of the Request object

- An attempt to stop processing the request is made:
  - in any case, none of the request handlers will ever be invoked
- Requests can be tagged with a unique string chosen by the programmer (e.g., the Activity class fully qualified name)
  - *RequestQueue* offers method *cancelAll(tag)* that removes all requests labelled with the given tag
  - This may be conveniently invoked in the Activity *onStop()* method to prevent further processing of the pending responses

17.1.3 WebView

Allows to show web contents on an activity like a browser, but with limitations:

- does not provide buttons to reload the page;
- no title;

and extra features:

- allows listening for events;
- allows requesting JavaScript dialog;
- internal JS can communicate with the outer Java environment and vice versa;

18 Multimedia

Nothing interesting to say; the only particular note is that these kind of resources (camera) must be released as soon as possible, not later than *onPause().*

19 Persistent data

Various options:

- **Preferences:** lightweight mechanism where simple key/value pairs can be stored;
- **Files**: store text/binary data; they can also be used to access content in the removable storage;
- **SQLite DBs:** provide a way to manage structured relational data;

19.1 Cursors

Objects that allow you to scan the list of results of a query

The life cycle of the cursor is related to that of the activity

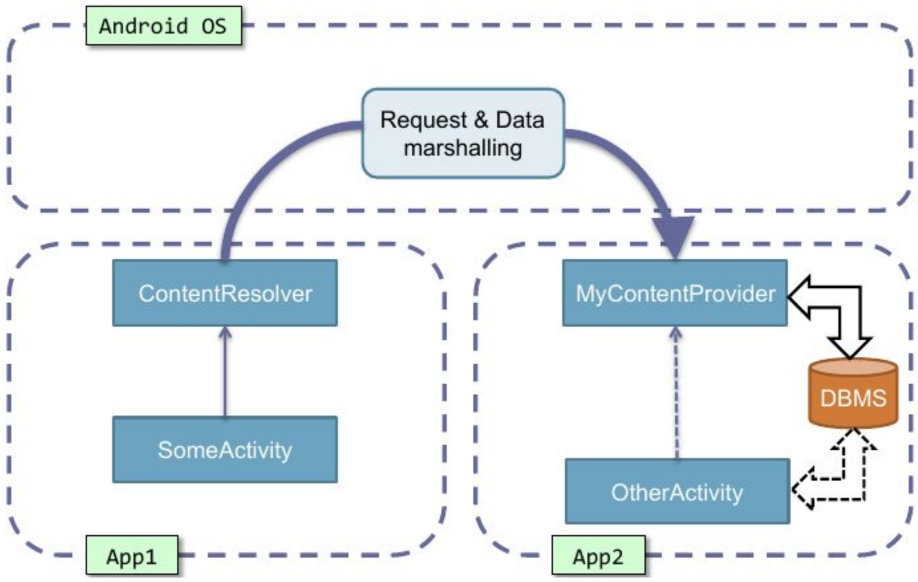
- When the application is paused or closed, the cursor must be deactivated by calling the *deactivate()* method
- When the application is restarted and you want the cursor to be updated, call the *requery()* method

- When the cursor is no longer needed, close it, via the *close()* method

19.2 Exposing data to other apps

<u>SQLite dbs are private to each application</u>

- If contained data is to be exposed to other apps, a **content provider** must be built
- **Content providers** are the standard interface that connects data in one process with code running in another process
    - The remote process access a *ContentResolver* as a proxy for the data exposed by the content provider



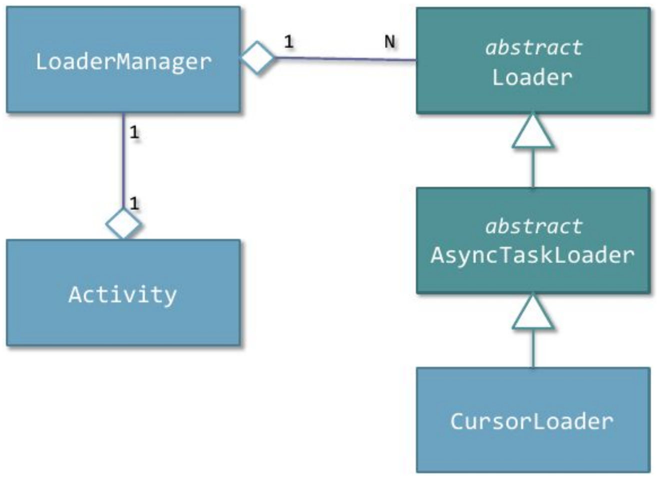
In order to be accessible from external process, content providers must be declared in the manifest file.
Each content provider should implement other five methods (CRUD) (besides OnCreate()) related to the specific operation requested:

- They should all be thread-safe, since several concurrent requests may arrive at the same time
- If the content provider does not intend to expose the corresponding operation, the method implementation can be minimal

19.3 Loaders and LoaderManager

Loaders are a general purpose mechanism to asynchronously load data in an activity or fragment.

- They monitor the source of their data and deliver new results when the content changes
- They support automatic reconnection to the data source when a configuration change causes them to be recreated



19.3.1 LoaderManager

Each Activity (and Fragment) has a single *LoaderManager*, **responsible for managing one or more *Loader* instances and used to manage long-running operations** that might span the lifecycle of the *Activity*.
To interact with the *LoaderManager*, an instance of the *LoaderCallbacks* is to be provided: *Loader* lifecycle events are reported to it (*onLoaderCreate*, *onLoadFinished*, *onLoaderReset*) and it is responsible to actually implement the operations that cause the *View* to be updated.

19.3.2 CursorLoader

A subclass of *AsyncTaskLoader* that queries a *ContentResolver* and returns a *Cursor*.

20 Services

A service is an application component that performs operations in background

- It does not provide any user interface: a component can connect to a service and interact with it, even if it belongs to another process
- Two types of services:
  - **started**
  - **bound**

*<u>A service is "**started**" when an application component, such as an activity, starts it by calling the startService(...) method</u>*

- Once started, a service can run indefinitely in background, even if the component that started it has been destroyed;
- Generally, a "started" service runs a single operation and does not return any result to the calling component (for example, a file can be downloaded from the network);
- When an operation ends, the service can stop itself;

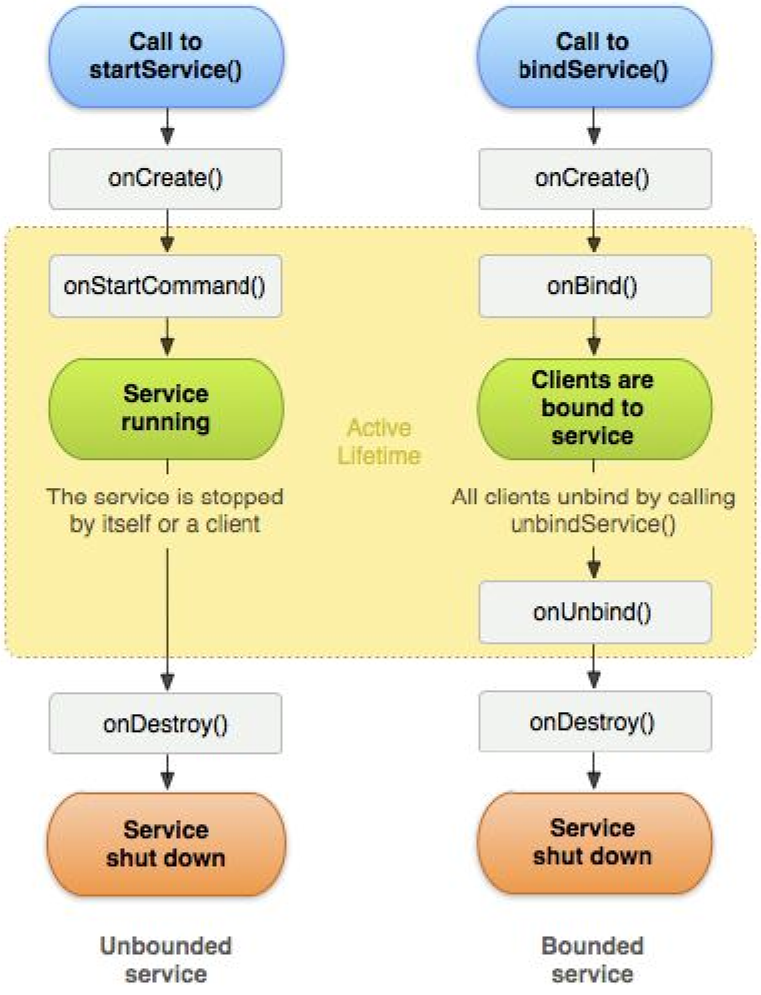
*<u>A service is called "**bound**" when an application component is associated to it by calling the bindService(...) method</u>*

- It offers a client-server interface
- It allows the components to interact with the service, sending requests, obtaining results, running inter process communication (IPC)
- Its duration is related to those of the component to which is bound
- Several components can be associated to the service at the same time, but when all the associated component are disassociated from it, the service is destroyed

20.1 Abstract class Service

It is required to override some methods:

- **onStartCommand(...)**: the system calls this method when another component, such as an activity, invokes the service, via the startService(...) method; once this method is executed, the service is started and can run in background indefinitely; to stop it, it is required to call stopSelf(...) or stopService(...), passing the suitable intent
- **onBind(...)**: the system calls this method when an other component wants to associate to the service, by calling bindService(...);
  It is needed to provide an interface that allow the client to communicate with the service, returning a class that implements the *IBinder* interface; if this interface is not implemented, binding is not allowed;
- **onCreate(...)**: the system calls this method when the service object has just been created; call happens before either onStartCommand(...) or onBind(...); if the service has already been started, this method is not called;
- **onDestroy(...):** The system calls this method when the service is no more used and it is to be destroyed; services should implement this method in order to release resources Thread, registered listener, receiver, …; it is the last call received by the service



20.2 Started services

They can be created using *Service* or *IntentService*

>>> *Service* is the base class for all services

- It is better to create a thread to which delegate service operations, because by default the service uses the application main thread
- *IntentService* is a subclass of *Service*; it uses a thread to manage all requests to start the service, one at a time; it requires to override *onHandleIntent(...)*

20.2.1 IntentService

It automatically creates a thread handling Intents requested by ***onStartCommand()***, and this thread creates a queue that buffers intents before handling them one by one using *onHandleIntent(...).* This will automatically stop as soon as all requests have been managed; by default *onStartCommand()* sends the intent to the queue.

If *onCreate(...), onStartCommand(...), onDestroy(...)* methods are implemented, they should call the parent implementation, using *super.onCreate(...), super.onStartCommand(...), etc...*

In this way, the IntentService class can manage the worker thread life cycle The *onBind(...)* method does not have to call the parent class implementation.

20.3 Bound services

A bound service operates as a server in a client-server interface
It allows activities (or other components) to bind to itself in order to send requests and receive responses, possibly across process boundaries;
A bound service implements the onBind() method returning an object implementing the IBinder interface; this is a sort of “remote control” that let the client request operations to the service.

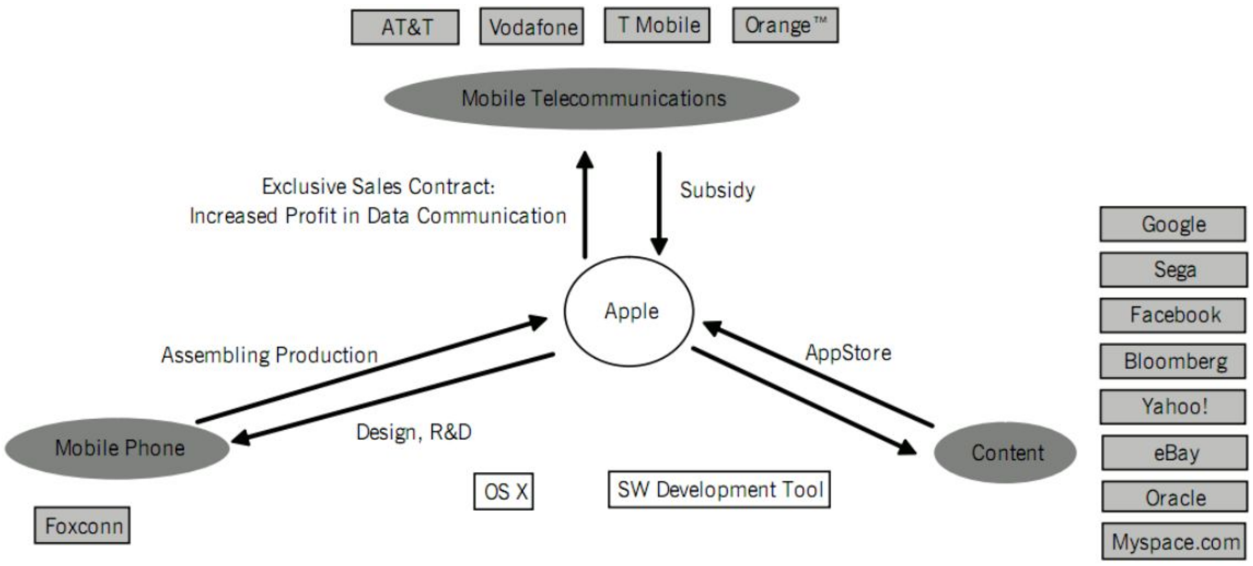
onBind()
This method is responsible to create and return an IBinder object
>> the OS guarantees that this method is invoked only once, the first time that a client component invokes bindService(...)
>> The returned object is cached by the OS, and returned to all other clients invoking bindService, as long as the service is running
>> When the last client disconnects from the IBinder object, the service is destroyed.

20.3.1 Connecting to a bound service

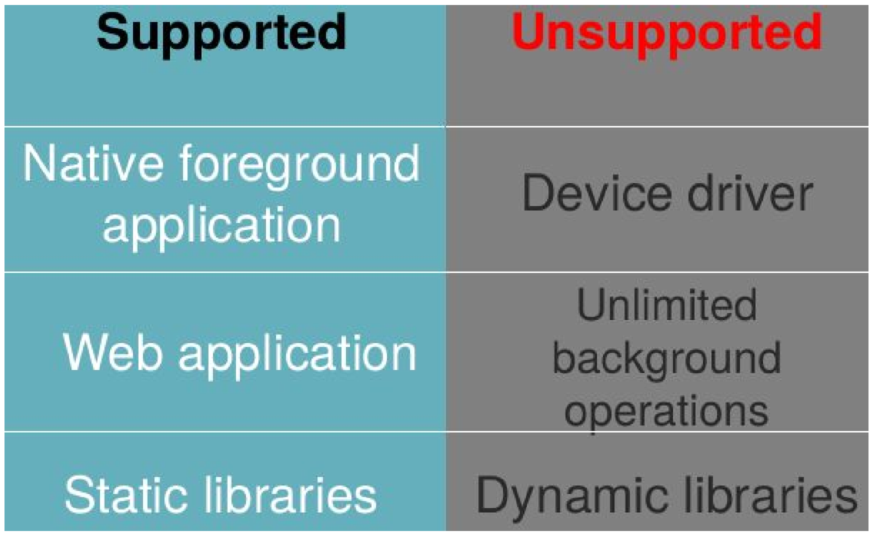
A component can connect to a bound service by invoking the ctx.bindService(...) method providing a *ServiceConnection* implementation that get notified of the binding lifecycle;
when binding is active, the returned object can be used to transparently access the service functionalities.

21 iPhone introduction

21.1 iPhone Business model



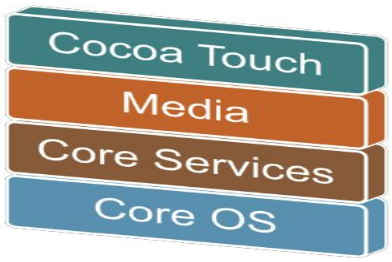
21.2 SDK Limitations



Supported	Unsupported
Native foreground application	Device driver
Web application	Unlimited background operations
Static libraries	Dynamic libraries

21.3 iOS

21.3.1 Layers



Cocoa touch = UIKit + Foundation Framework = basic tools and infrastructure to implement graphical, event-driven applications in iOS

Media Layer = graphics and media technologies (OpenGL ES, Core Audio, Video technologies,...) in iOS that are geared toward creating most advanced multimedia experience on a mobile device

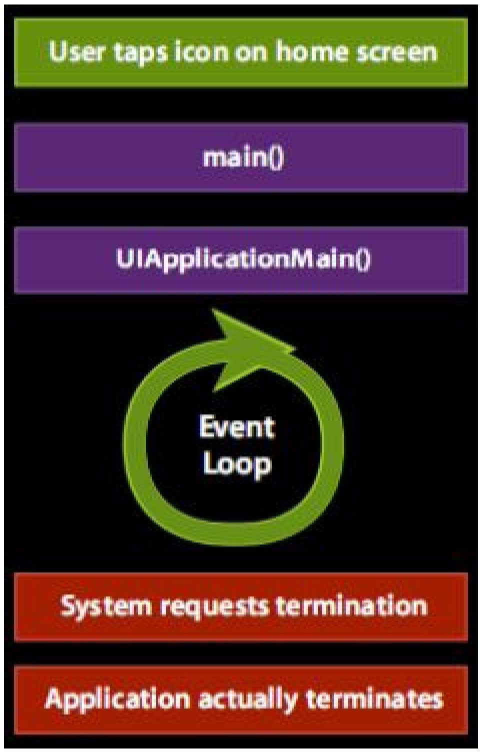
The **Core Services** layer provides the fundamental system services that all applications use such as

- Core location
- CFNetwork
- Address book
- SQLite
- Security

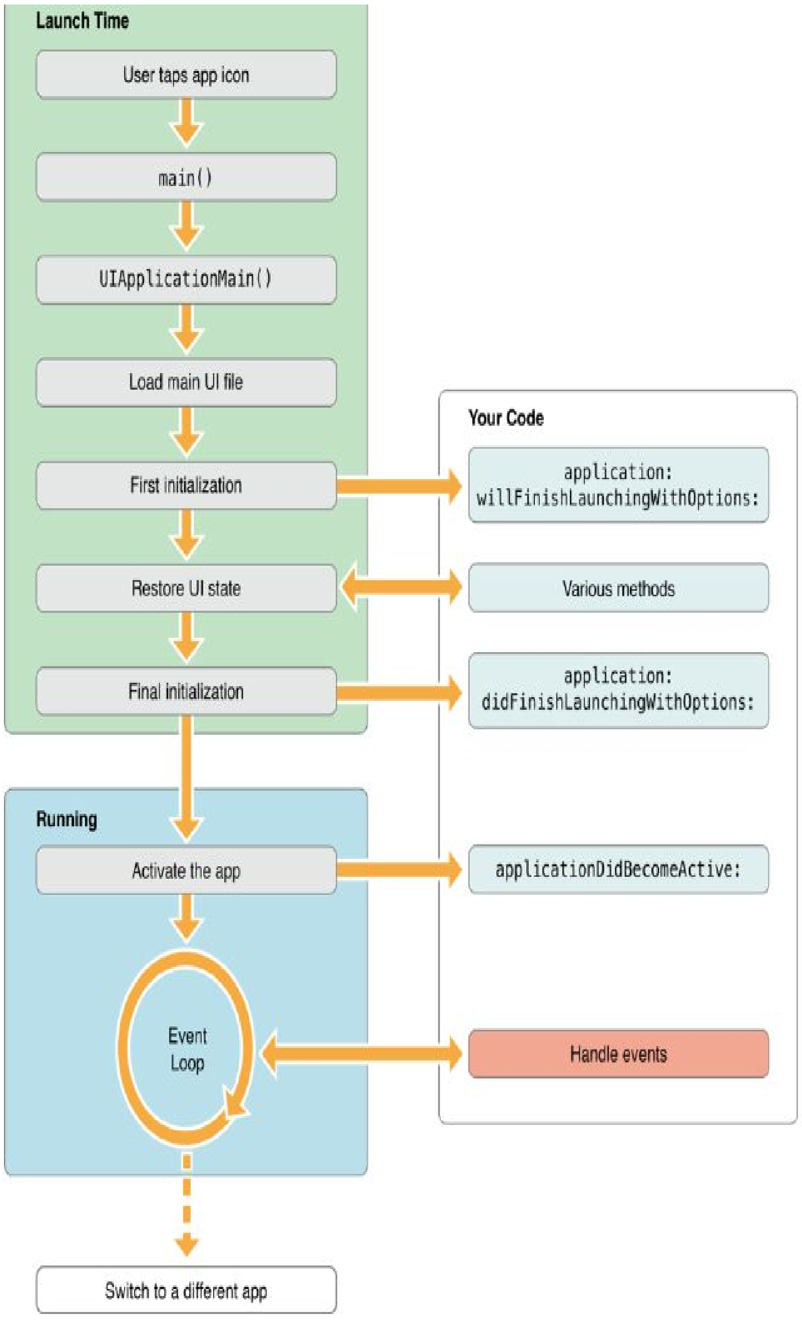
The **Core OS** layer encompasses the kernel environment, drivers, and basic interfaces of the OS

- Threading (POSIX threads)
- Networking (BSD sockets)
- File-system access
- Standard I/O
- Bonjour and DNS services
- Memory allocation

21.3.2 App Lifecycle

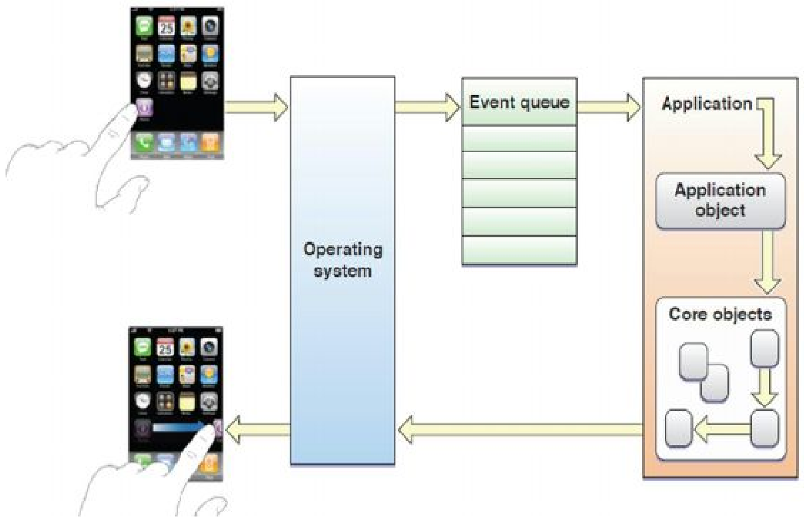

User taps icon on home screen
main()
UIApplicationMain()
Event Loop
System requests termination
Application actually terminates

J



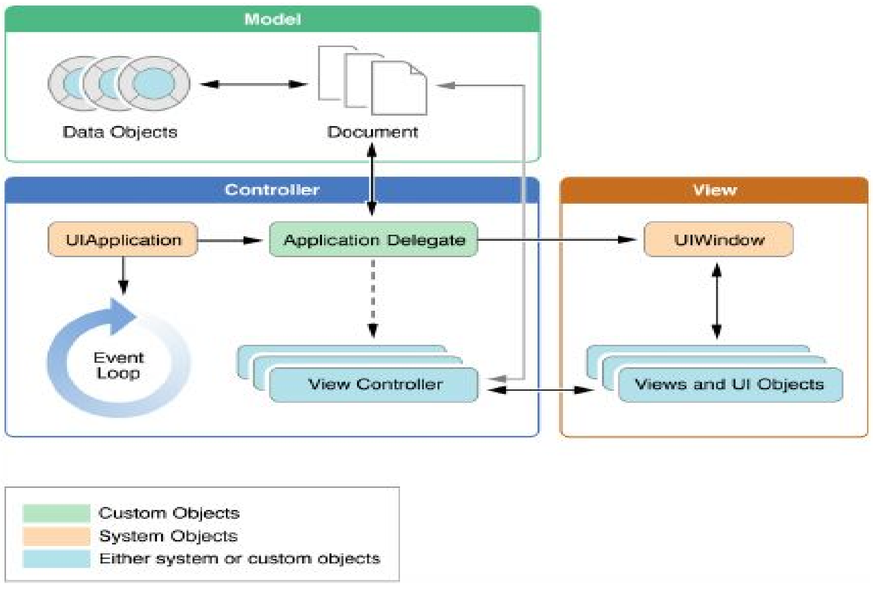
Main()
Just like any other main functions (C, C++, Java, ...); it creates the **top-level autorelease pool** and starts application with UIApplicationMain
UIApplicationMain()
Creates instance of *UIApplication* that is responsible for actually launching your application

21.3.3 The event handling cycle



1. System receives an event
2. System sends event on to your application instance
3. Application instance then forwards the event to the **First Responder**, who starts sending the event up the chain. This is called the **responder chain**
4. Take-away: Event handling is complicated, and all you really need to know is that you can intercept these events to use them (like *touchesBegan:withEvent*)

21.3.4 Application architecture



UIApplication

- It manages the app event loop and coordinates other high-level app behaviors
- Delegates custom logic management to the app delegate

AppDelegate

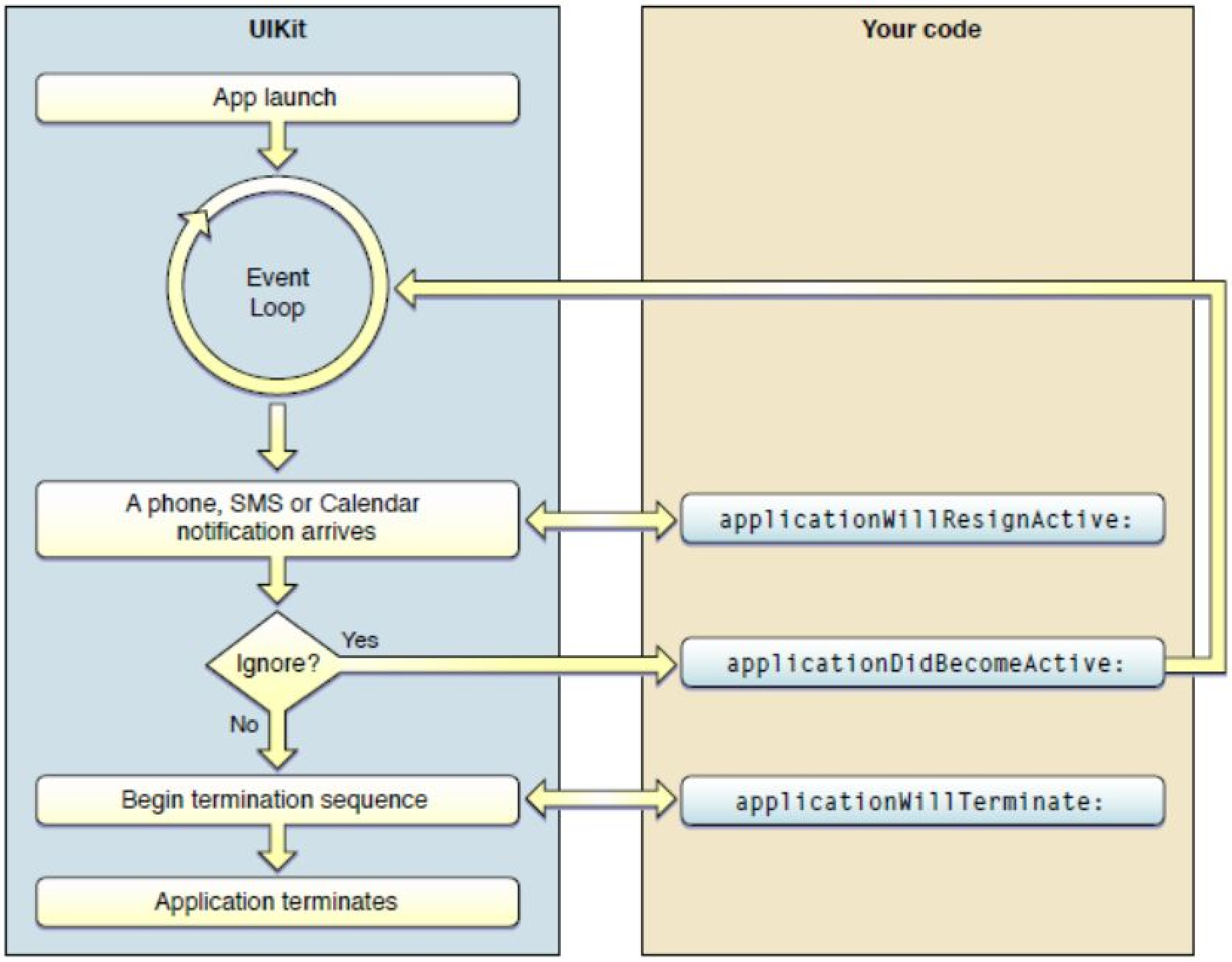
- Custom object created at app launch time
- It handles state transitions within the app
- Should extend the *UIResponder* system class

Data model objects

- Store information relevant to the app
- iOS makes no assumption about their structure

View Controllers

- Manage the presentation of the app content on screen
- Each view controller manages a single view hierarchy and make them visible on the screen
- Should extend the *UIViewController* class


UIKit
Your code
App launch
Event Loop
A phone, SMS or Calendar notification arrives
applicationWillResignActive:
Yes
Ignore?
applicationDidBecomeActive:
No
Begin termination sequence
applicationWillTerminate:
Application terminates

22 Objective C

- Dynamic, untyped language
- Single inheritance
- Method dispatch done at runtime

MetaClass
A class that is used to create class objects is called metaclass

Class type
It is defined as `typedef struct objc_class *Class;`

Objects

- Each object is a block of memory allocated inside the heap
- Objects contain a reference to their own class as first instance member
- Type **"id"** represent a reference to a generic object => **void***

Class = *interface* + *implementation* = *.h + *.m

```
@interface DotView: NSView
{
      NSPoint center;
      NSColor *color;
      float radius;
}
-(id) initWithFrame: (NSRect)frame;
-(void) setX:(int)x andY:(int)y;
+ (void)initialize;

@end
```

DotView is a new class inheriting from *NSView* with 3 attributes, 2 instance methods (object methods) and 1 class method (~ static method)

```
@implementation DotView
- (id)initWithFrame:(NSRect)frame
{
      // Implementation here
}
- (void)setX: (int)x andY: (int)y
{
      // More implementation
}
```

```
+ (void)initialize
{
      // Static method; No instance variables!
}
@end
```

@public, @private, @protected are used like in C++
self = like **this** in Java
super = like **super** in Java
e.g. `[self setStringValue:@"Hello!"];`

22.1 Constructing Objects

What in Java is simply done in one unique step using the constructor, in Objective-C it is made in two steps: **allocation + variable initialization**, using static methods
`MyClass *obj= [[MyClass alloc] init];`
Memory can be released sending the **release** message (= calling the release method)

Message	Reference counter
alloc	+1
release	-1
retain	+1 (without allocating anything, used to indicate a new pointer)

The **dealloc** method can be implemented to behave like C++ destructor, but it should never be directly invoked => it is automatically called by **release**

22.2 Autorelease pool

If a method returns a freshly created object, the responsibility of its deallocation is moved to the message sender

```
method{
      return [s autorelease];
}
```

An *NSAutoreleasePool* object contains a list of objects that have received an *autorelease* message; when the pool is drained it sends a *release* message to each of those objects and so sending autorelease extends the lifetime of the receiver until the pool itself is drained
<u>*If the object is to be kept for a longer time, it must be retained*</u>

- When a pool is created, it is pushed onto a stack

- When an object receives an autorelease message, the pool on top of the stack will store a reference to that object

AppKit creates an autorelease pool at the beginning of every cycle of the event loop, and drains it at the end, releasing any autoreleased objects generated while processing an event

22.3 Automatic Reference Counting (ARC)

Automatic Reference Counting (ARC) is a compiler feature that provides automatic memory management of Objective-C objects, inserting the appropriate memory management method calls at compile-time.

Rules:

- no explicit invocation to *dealloc, retain, release, retainCount* or *autorelease;*
- *dealloc* must be overridden if it is needed to manage resources other than releasing instance variables;
- it is forbidden releasing instance variables: ARC takes care of it;
- *CFRetain*, *CFRelease*, and other related functions with Core Foundation-style objects are still available;
- *NSAllocateObject* or *NSDeallocateObject* cannot be used: objects are created using *alloc* and the runtime will take care of their management;
- object pointers in C structures cannot be used → classes must be used instead
- there is no casual casting between **`id`** and **`void*`** → special casts must be used to tell the compiler about object lifetime
- *NSAutoreleasePool* objects cannot be used → ARC provides *@autoreleasepool* blocks instead
- memory zones cannot be used, also because there is no need of *NSZone* anymore

New qualifiers added by ARC:

- **`__strong`** is the default. <u>An object remains "alive" as long as there is a strong pointer to it</u>
- **`__weak`** specifies a reference that <u>does not keep the referenced object alive. A weak reference is set to nil when there are no strong references to the object</u>
- **`__unsafe_unretained`** specifies a reference that does not keep the referenced object alive and is not set to nil when there are no strong references to the object
- **`__autoreleasing`** is used to denote arguments that are passed by reference `(id *)` and are autoreleased on return

Note about strong/weak references:
A strong reference (which it is used in most cases) means that you want to "own" the object you are referencing with this property/variable. The compiler will take care that any object that you assign to this property will not be destroyed as long as you (or any other object) points to it with a strong reference. Only once you set the property to `nil` will the object get destroyed (unless one or more other objects also hold a strong reference to it).

In contrast, with a weak reference you signify that you don't want to have control over the object's lifetime. The object you are referencing weakly only lives on because at least one other object holds a strong reference to it. Once that is no longer the case, the object gets destroyed and your weak property will automatically get set to nil. The most frequent use cases of weak references in iOS are:

1. delegate properties, which are often referenced weakly to avoid retain cycles, and
2. subviews/controls of a view controller's main view because those views are already strongly held by the main view.

atomic vs. nonatomic refers to the thread safety of the getter and setter methods that the compiler synthesizes for the property. atomic (the default) tells the compiler to make the accessor methods thread-safe (by adding a lock before an ivar is accessed) and nonatomic does the opposite. The advantage of nonatomic is slightly higher performance. On iOS, Apple uses nonatomic for almost all their properties so the general advice is for you to do the same.

22.3.1 Properties (= java get/set)

The **@property** directive provides a simple way to declare and implement an object's accessor methods allowing a more natural syntax for interacting with the corresponding values.
In the `@interface` section the **@property** directive declares a property while in the `@implementation` section the **@sintesize** directive asks the compiler to generate corresponding setter and getter methods

```
@interface DotView : NSView {
      NSPoint center;
      NSColor *color;
      float radius;
}

@property (copy) NSPoint center;
@property (copy) NSColor *color;
@property float radius;

@end
/****************** IMPLEMENTATION **********************************/
@implementation DotView
-(void)dealloc {
      [color release];
      [super dealloc];
}

@synthesize center;
@synthesize color;
@synthesize radius;
@end
```

Properties are accessed using the standard dot notation

Attributes

Copy	In the setter, the old object is released, and a copy of the parameter is made and stored in the name instance variable
Assign	In the setter, the old reference is replaced with the parameter - Useful for simple types - Beware of dangling pointers!
Retain	In the setter, the old manager object is released and the manager instance variable is set to a retained parameter
Getter	We tell the compiler to change the name of the getter to xxxxx instead of the conventional name yyyyyy

Manual synthesis: setters may need to send a **`mutableCopy`** message to its parameter to be able to manipulate it later

22.4 Protocols (= java's Interface)

A protocol is a list of method declarations grouped in `@protocol/@end` directive pair
A class may adopt a protocol, providing the corresponding method implementations.

```
@interface NSData : NSObject <NSCopying, NSMutableCopying> { . . . }
@end
```

The class NSData inherits from NSObject and *adopts* (in java jargon "implements") NSCopying and NSMutableCopying.
A protocol may contain **`@optional`** and **`@required`** methods; to test if they are implemented it **`@selector(`**`method_name`**`)`** can be used.

22.5 NSObject

It is both a class and a protocol and its methods can be divided in four groups:

Class access and testing	`(Class) class;`
	`(Class) superclass;`
	`(BOOL) isKindOfClass: (Class)c;`
	`(BOOL) isMemberOfClass: (Class)c;`
	`(BOOL) respondsToSelector: (SEL)s;`

	(**BOOL**) conformsToProtocol:
	(**Protocol** *) p;
	(**BOOL**) isProxy;

Memory management	(**id**) retain;
	(**void**) release;
	(**id**) autorelease;
	(**NSUInteger**) retainCount;
	(**NSZone** *) zone;

Object access and comparison	(**BOOL**) isEqual: (id)other;
	(**NSUInteger**) hash;
	(**id**) self;
	(**NSString** *) description;

Sending messages	(**id**) performSelector: (SEL)s;
	(**id**) performSelector: (SEL)s withObject: (**id**)o;
	(**id**) performSelector: (SEL)s withObject: (**id**)o1
	withObject: (**id**)o2;

22.6 Object lifecycle

```
+(id)alloc
-(id)init
+(id)new
-(void)dealloc
-(void)finalize
...
```

- **alloc** returns a block of memory large enough to contain a new instance of the receiving class
- **init** is implemented by subclasses to initialize a new object after its memory has been allocated
- **new** is almost equivalent to `[[Class alloc] init];`

22.6.1 Object ownership

You only own objects you created or explicitly retained using a method whose name begins with "**alloc**" or "**new**" or contains "**copy**"; many classes provide methods of the form **+className...** that can be used to obtain a new instance of the class: but they are not owned.
The use of the ARC feature simplifies the task of managing object ownership but it still requires understanding the lower level mechanisms.

Within a given block of code the following equation must be respected to avoid memory leakage
`n*copy + m*alloc = p*release + q*autorelease`
where n,m,p and q are the number of times that those methods are used

`description` method = java .toString() method

22.6.2 NSArray

IMMUTABLE array of objects → they can neither be added nor removed, `nil` is not allowed

- Its instances maintain strong references to their contents
- When an array is initialized with a list of objects, every object in the list receives a retain message

To have a mutable array use **`NSMutableArray`** that extends `NSArray` adding all the other features present for example in a java `ArrayList`

22.6.3 NSDictionary

IMMUTABLE key/value association; they can neither be added or remove, `nil` is not allowed
`NSMutableDictionary` is used to have mutable objects
Note: Keys stored inside the dictionary are copies of the original ones; keys must conform to the NSCopying protocol; moreover, in order to prevent memory overhead keys having large storage requirements should be avoided.

23 iOS

All iOS applications are built using the *UIKit* framework and they have the same core architecture, based on the Model/View/Controller paradigm:

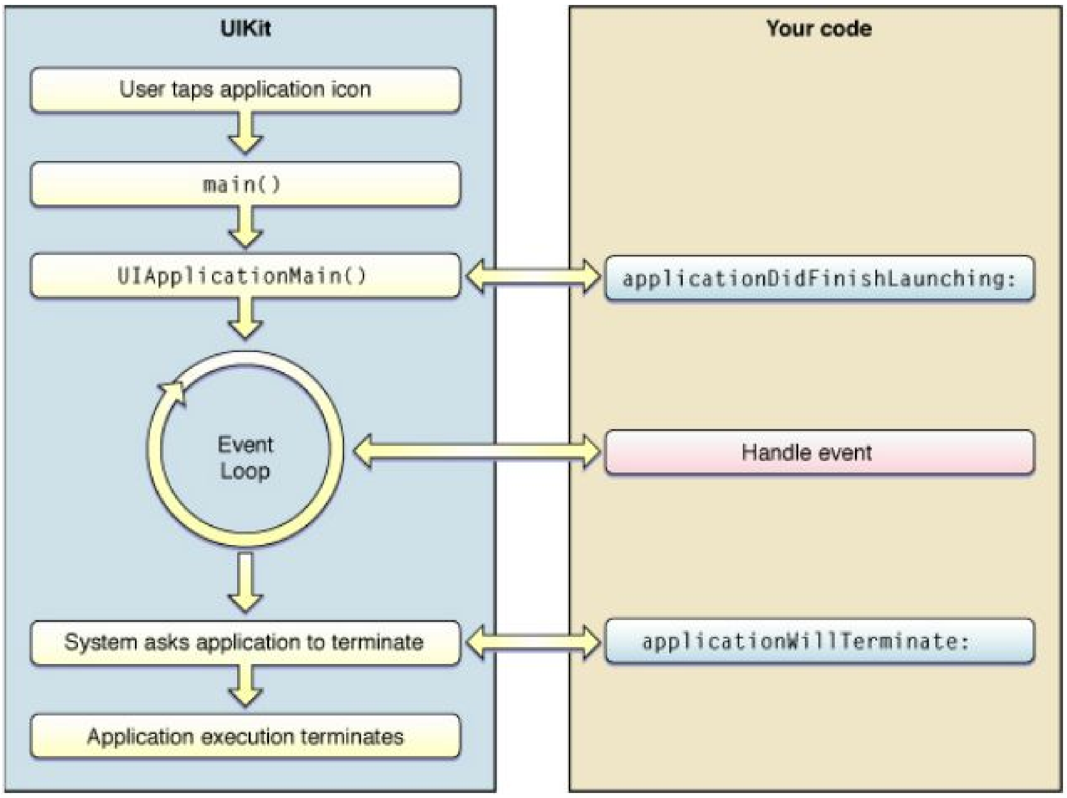
- **Views** are represented by **`UIView`** objects
- **Controllers** extends to the **`UIViewController`** class
- No restriction is put on **models**

At startup:
`main()` → creates an AutoReleasePool and invokes **`UIMainApplication()`**
`UIMainApplication()` → creates a singleton instance of the `UIApplication` object and to start the event handling loop

23.1 UIApplication class

It provides a centralized point of control and coordination for running applications and exists in any application providing facilities to manage the main events and features (getting application windows, controlling and handling events, opening an URL resource …)

Whenever a relevant event occurs, the `UIApplication` instance notifies its delegate, which conforms to the **`UIApplicationDelegate`** protocol where all methods are optional;
if the delegate does not implement the corresponding method, the event is discarded.


UIKit
Your code
User taps application icon
main()
UIApplicationMain()
applicationDidFinishLaunching:
Event Loop
Handle event
System asks application to terminate
applicationWillTerminate:
Application execution terminates

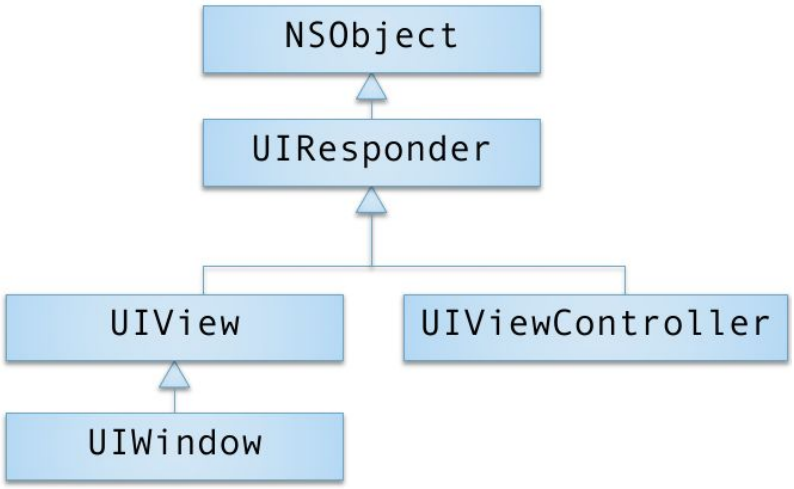
23.2 Window

Every application has a window

◦ Instance of the **`UIWindow`** class

◦ It represent a portion of the screen where display can take place

◦ It is the root of the display tree



23.3 UIViewController

An object that manages a full screen view providing basic support for managing modal views and rotating views in response to device orientation changes; its subclasses provide additional behavior for managing complex hierarchies of view controllers and views

23.4 UIView

A view defines a rectangular area on the screen

- Views draw content in their area Views manage a list of subviews
- Views define their own resizing behaviors in relation to their parent view
- They can manually change the size and position of their subviews

Being a subclass of **`UIResponder`**, a view receives touch events and participate in the responder chain; `UIView` is the base class of a densely populated hierarchy

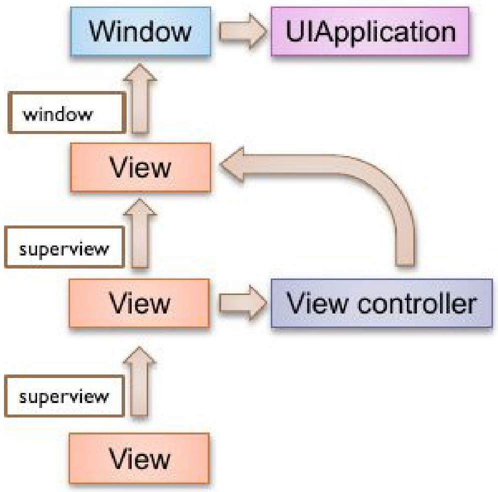
23.5 UIResponder

UIResponder manages touch events; each of them is described by an UITouch object

23.5.1 Event dispatching

An event occurs in a view:

- If the view object does not handle it, it is forwarded to the view controller
- Or to the superview
- Or to the window
- Or, eventually to the application



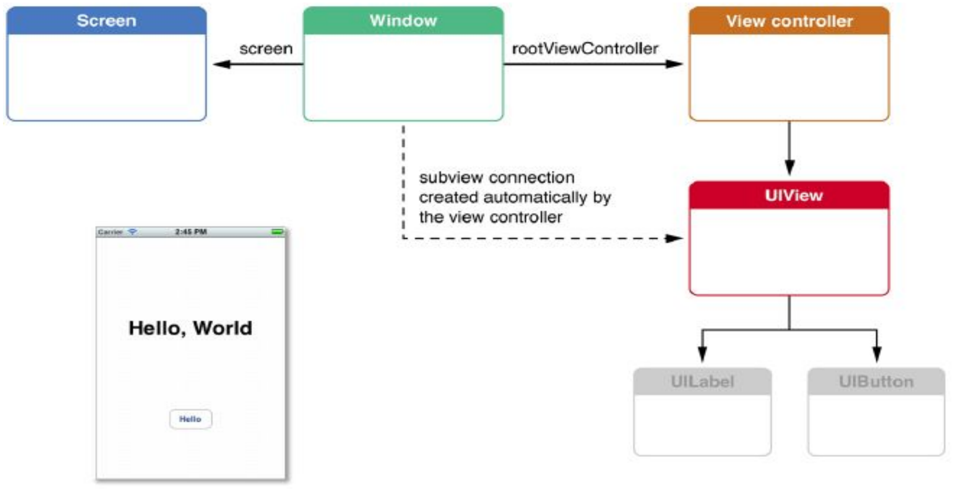
Widget properties are declared in the view controller using the **`IBOutlet`** keyword to be referenced by the visual designer and then it is responsibility of the **`loadView`** method to set these properties to the corresponding objects.

24 iOS ViewController

A view controller acts as a bridge between application data and their visual representation
◦ They represent the major building block of the application framework in iOS
◦ **They may be viewed as the iOS counterpart of Android activities**

View controllers derive from the **`UIViewController`** class that specifies methods for handling life-cycle notifications and manage their resources.
Each view controller owns a visual hierarchy accessible via the "view" property

Whenever an app is launched, its **`AppDelegate`** is notified via the `application:didFinishLaunching` method: its responsibility is to allocate a screen-wide window and assign it a root view controller

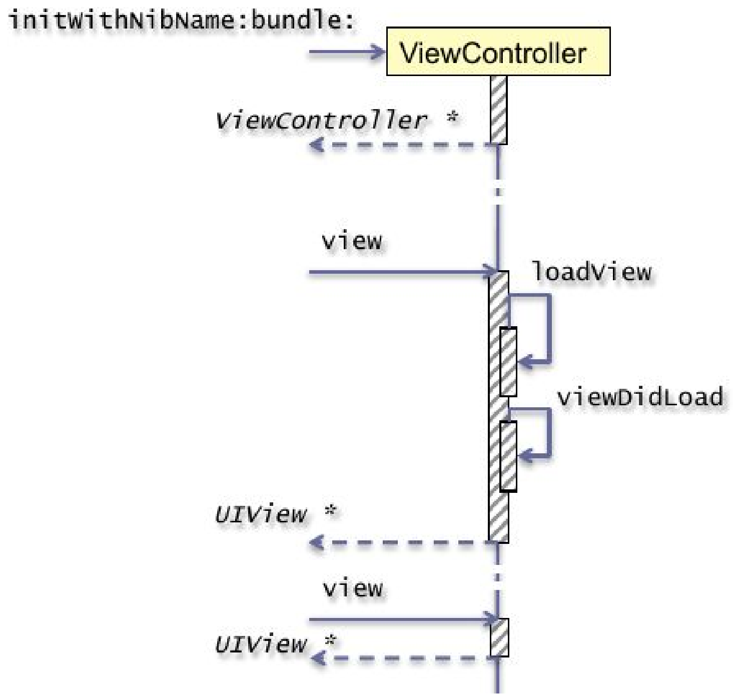


24.1 UIViewController

Owns a view hierarchy, referenced by its view property
◦ Exposes a set of methods that receive notifications about its own life cycle
Has the responsibility for:

- creating, resizing and laying out its view
- adjusting the contents of the view
- acting on behalf of the views, when the user interacts with them

24.1.1 Lifecycle



25 UX - User Experience

<u>*It is the experience that a person lives when he/she interacts with a product in the real world*</u>

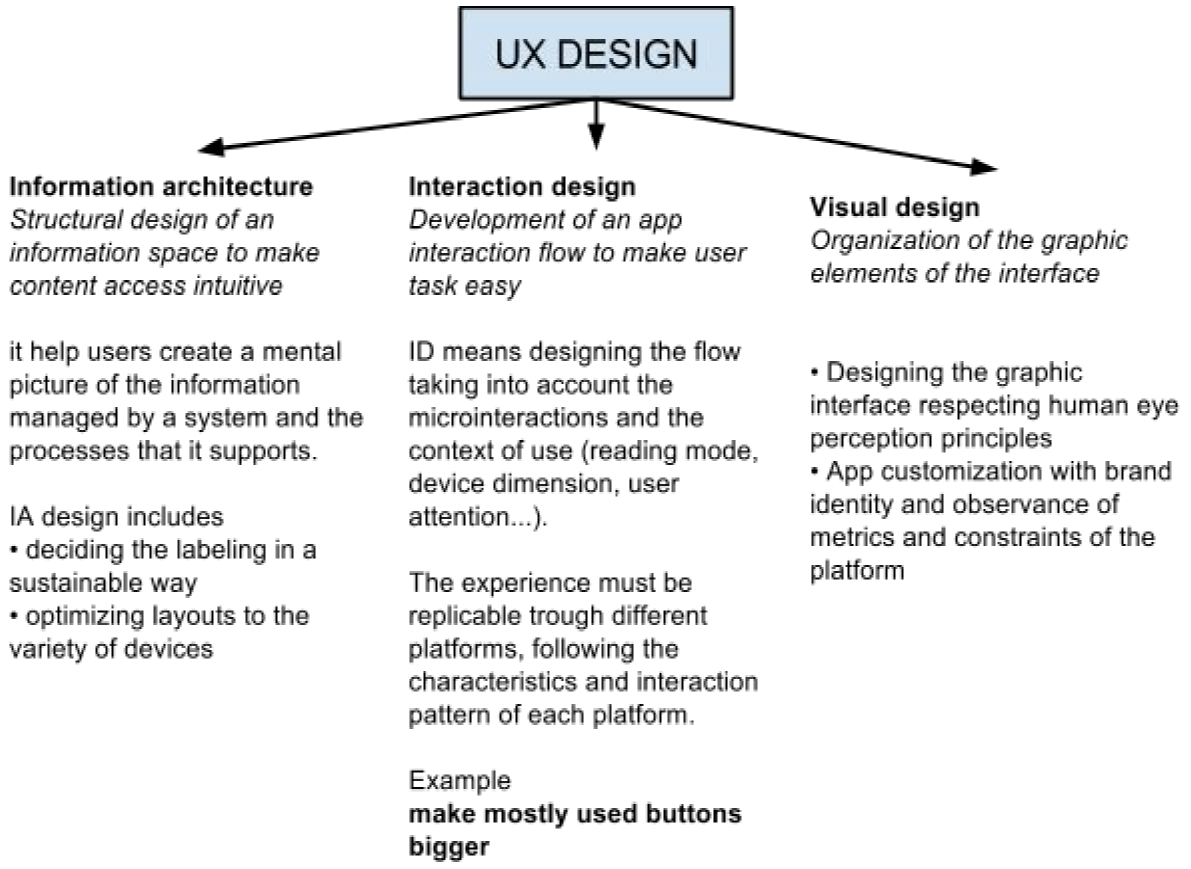
- A product/service is **working** when it can be perceived inside users' minds through interactions, behaviors and emotions
- If a product **doesn't work** as a person thinks it should do, the person will be frustrated even if he/she will be able to achieve the task

KISS (keep it simple stupid)

25.1 UX Design

Realizing an efficient and pleasurable UX is based on ***user centered design (UCD)*** *concept*
Opposite to **Technology-driven design**
<u>>>> user interests are the key points of the development, not the technology</u>

1. Understand mental processes of the target of the product (visual thinking, feeling)
2. Understand the needs and unspoken aspirations which the product intends to resolve (design challenge, concept definition)
3. Consider a variety of possible solutions and switching from research to real solutions (synthesis, brainstorming, prototyping)
4. Avoid imposing a wrong solution to the mass, analysing the reactions of people (Test and feedback)



Information architecture
Structural design of an information space to make content access intuitive

it help users create a mental picture of the information managed by a system and the processes that it supports.

IA design includes
• deciding the labeling in a sustainable way
• optimizing layouts to the variety of devices

Interaction design
Development of an app interaction flow to make user task easy

ID means designing the flow taking into account the microinteractions and the context of use (reading mode, device dimension, user attention...).

The experience must be replicable trough different platforms, following the characteristics and interaction pattern of each platform.

Example
make mostly used buttons bigger

Visual design
Organization of the graphic elements of the interface

• Designing the graphic interface respecting human eye perception principles
• App customization with brand identity and observance of metrics and constraints of the platform

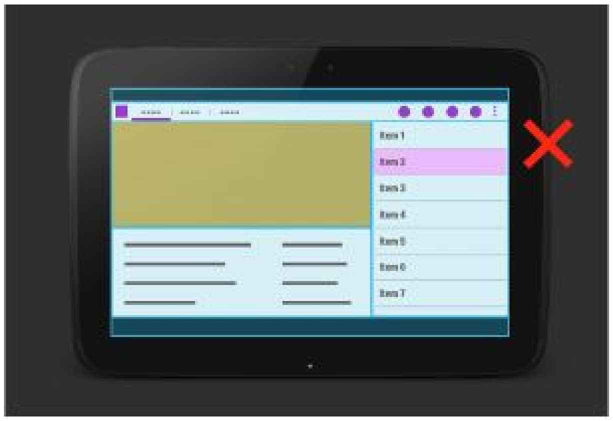
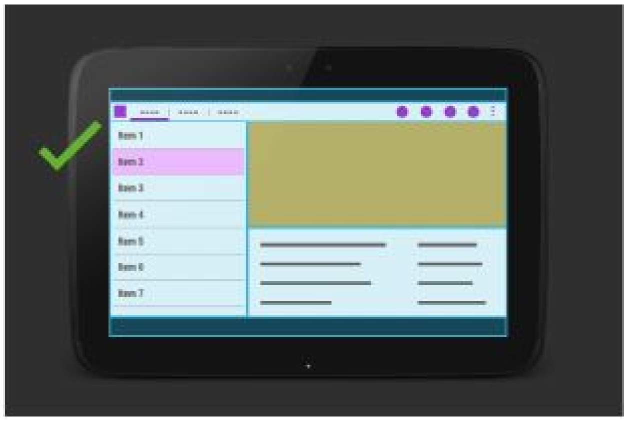
26 Real world interpretation

Every time we interact with the real world to achieve a goal, some processes take place

- **Perception**
- **Memory**
- **Interpretation**

26.1 Perception and experience

People create mental models observing a specific disposition again and again

26.2 Memory

Working memory

◦ It is used for less than a minute

Long term memory

◦ It keeps the information available for long periods (hours, days, months, years ...)

The working memory becomes long term memory in two ways:

◦ Repetition

◦ Link with something you already know

27 Visual design

Take a brief look at this slides set; there are just a set of picture showing bad and good designs for each kind of layout and component

28 Exam questions

- canvas pipeline
- iOS controller view (lifecycle)
- canvas frame
- activity
- fragment
- services lifecycle
- synchronization constructs
- MVC pattern
- ARC
- resources and openGL shaders
- difference between weak and strong allocation in iOS
- adapters