

**تکلیف ارائه کتبی**

**داده‎های هوشیار در سیستم اندروید - Live Data in Android**

**نام استاد: استاد پیمان کبیری**

**دانشجو: محمد امین قسوری**

**شماره دانشجویی: ۹۷۵۲۱۴۳۲**

**خرداد ۱۴۰۰**

**به نام خدا**

سپاس‎،  
از استاد حائری برای کمک‎هایشان در راستای یادگیری برنامه‎نویسی برای سیستم‎عامل اندروید و همچنین دکتر کبیری برای راهنمایی‎هایشان در گزارش نویسی، نهایت سپاس و تشکر را دارم.

فهرست مطالب

[مقدمه 6](#_Toc74714077)

[الگوی طراحی مشاهده‎گر 6](#_Toc74714078)

[پیاده‎سازی این الگوی طراحی 7](#_Toc74714079)

[داده‎های هوشیار 8](#_Toc74714080)

[تفاوت کلاس داده‎ی هوشیار و تفاوت با مشاهده‎شونده عادی 8](#_Toc74714081)

[نحوه ساخت یک داده هوشیار 9](#_Toc74714082)

[بررسی کلاس‎های داده‎های هوشیار 9](#_Toc74714083)

[کلاس LiveData [3] 9](#_Toc74714084)

[کلاس MutableLiveData [4] 9](#_Toc74714085)

[کلاس MediatorLiveData [5] 9](#_Toc74714086)

[فرق تنظیم یک مقدار با فرستادن یک مقدار 9](#_Toc74714087)

[استفاده از داده‏های هوشیار در معماری MVVM 10](#_Toc74714088)

[تعریف در لایه ‏ذخیره‎کننده 10](#_Toc74714089)

[تعریف در لایه View Model 10](#_Toc74714090)

[لایه رابط کاربری یا View 11](#_Toc74714091)

[منابع 12](#_Toc74714092)

**فهرست نگاره‎ها**

[شکل 1 الگوی طراحی مشاهده‎گر 7](#_Toc74713482)

[شکل 2 نمودار کلاس ها 7](#_Toc74713483)

[شکل 3 روند طراحی در MVVM 10](#_Toc74713484)

**چکیده**

بحث داده‎های هوشیار[[1]](#footnote-1) در اندروید، بحثی بسیار داغی است که با استفاده از آن‎ها در معماری MVVM[[2]](#footnote-2) می‎‎توانیم به ساختار برنامه‎ی خود کمک کنیم. یکی از امتیازهای استفاده از این داده‎های هوشیار در مقابل کلاس‎های درون‎ساخته‎شده‎ی[[3]](#footnote-3) جاوا مانند کلاس Observable این است که داده‎های هوشیار با چرخه‎ی حیات اجزای اندروید[[4]](#footnote-4) بسیار همروند[[5]](#footnote-5) شده است. این موضوع باعث می شود تا از واقعه‎ی نشت حافظه[[6]](#footnote-6) که بحث مهمی در طراحی نرم‎افزارهای گوشی است جلوگیری شود. داده‎های هوشیار، ایده‎ای شبیه به الگوی طراحی[[7]](#footnote-7) مشاهده‎گر[[8]](#footnote-8) را پیاده سازی کرده است و شامل کلاس‎هایی مانند LiveData، MutableLiveData، MediatorLiveData و ... است. همچنین با استفاده از این ابزار، توسعه‎دهنده برای به‎روزرسانی وضعیت برنامه خود دردسر کمتری خواهد داشت و دیگر لازم ندارد که به صورت دستی هر تغییری را به‎روز‎رسانی کند. از دیگر امتیاز‎های داده‎های هوشیار، بهبود هم سرعت شدن توسعه نرم‎افزار با ایده‎های تیم محصول خواهد بود و همچنین مشکل‎یابی و بهبود‎ مشکل را برای ما راحت‎تر از قبل خواهد کرد.

# مقدمه

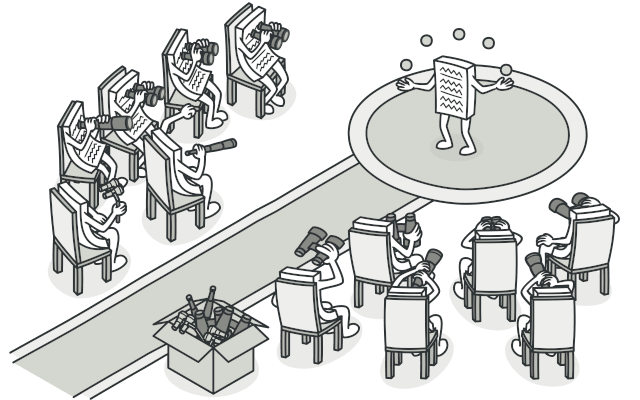
در طی چندین سال اخیر، به توصیه‎ی شرکت گوگل، توسعه‎دهنده‎های اندروید به استفاده از معماری جدیدی برای پیاده سازی برنامه های خود پرداخته‎اند. خود چارچوب[[9]](#footnote-9) اندروید شباهت زیادی به MVC[[10]](#footnote-10) دارد. معماری که در چندین سال اخیر به توصیه مستندات گوگل خیلی مورد توجه قرار گرفته است، معماری MVVM است. در کل توسعه‎دهنده‎های اندروید بیشتر درگیر چگونگی معماری و طراحی خود در سیستم‎های نرم‎افزاری شده اند. با استفاده از ابزارها و کتابخانه‎هایی که در چارچوب اندروید وجود دارند می‎توان نرم‎افزارهای با کیفیت‎تری را تولید نمود. داده‎های هوشیار، یکی از مواردی است که می‎توانیم در جهت بهبود برنامه‎ی خود استفاده کنیم. همچنین داده‎های هوشیار با معماری MVVM و کتابخانه View Model اندروید هم‎روند شده است.[1]

در ابتدا به بررسی کلاس‎های داده‎های هوشیار بپردازیم. داده‎های هوشیار به گونه‌ای الگوی طراحی مشاهده‎گر را در خود پیاده‎سازی نموده‎اند. ابتدا به الگوی طراحی مشاهده‎گر می‎پردازیم و به طور خلاصه این موضوع را بررسی می‎کنیم. در ادامه به عنوان نمونه سه کلاس از کلاس‎های داده‎های هوشیار در اندروید را بررسی ‎می‎کنیم. همچنین در ادامه به نحوه پیاده سازی آن نیز اشاره‎ای خواهیم داشت. در نهایت هم به تفاوت‎های آن با کلاس‎های درون‎ساخته زبان جاوا و نحوه‎ی صحیح استفاده از آن در معماری MVVM می‎پردازیم.

# الگوی طراحی مشاهده‎گر

برای آنکه بتوانیم به موضوع داده‎های هوشیار یا پویا در چارچوب اندروید بپردازیم، لازم است ابتدا با مفهوم الگوی طراحی مشاهده‎گر آشنا باشیم. الگوی مشاهده‎گر شامل دو قسمت می باشد، مشاهده‎گر[[11]](#footnote-11) و مشاهده‎شونده[[12]](#footnote-12). مشاهده‎شونده یکسری متغیرها دارد که وضعیت آن کلاس مشاهده‎شونده را مشخص می‎نماید. مشاهده‎گرها به گونه‎ای وابسته به آن متغیرها و وضعیت‎های مشاهده‎شونده هستند. برای همین نیاز داریم با به وجود آمدن تغییری در وضعیت آن‎ها باخبر شده و وضعیت برنامه خود را به روز رسانی کنیم.

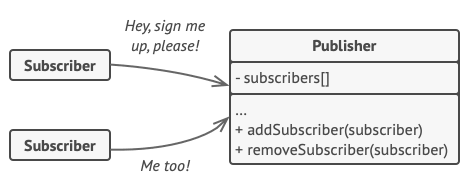
به عنوان مثال، یک کلاس را به عنوان بیانگر وضعیت آب‏ و هوایی در نظر بگیرید. این کلاس یک متغیر به نام isCloudly دارد که نشان دهنده آن است که هوا ابری است یا خیر. کلاسهای دیگری ممکن است وجود داشته باشند که با توجه به وضعیت این متغیر در این کلاس رفتارهای مختلفی داشته باشند. فرض کنید اگر هوا ابری باشد یک بازی فوتبال کنسل خواهد شد یا اگر هوا ابری نباشد سرعت بیشینه یک ماشین برای کلاس ماشین بیشتر خواهد بود. برای همین همه این کلاس‌ها که وابسته به این متغیر مشاهده شونده هستند یک مشاهده گر هستند و باید این کلاس را اصطلاحا مشاهده کنند.[[13]](#footnote-13)



شکل 1 الگوی طراحی مشاهده‎گر[2]

## پیاده‎سازی این الگوی طراحی

برای اینکه بتوانیم همچنین الگوی طراحی را پیاده‎سازی نمایید، نیاز است که لیستی از مشاهده‎گرها در کلاس مشاهده‎شونده نگه‎داری بشود. به عملیات افزودن یک مشاهده‎گر به مشاهده شونده، ثبت[[14]](#footnote-14) گفته می‌شود و به عملیات حذف یک کلاس یا مشاهده‎گر از لیست مشاهده‎شونده، حذف‎[[15]](#footnote-15) گفته می شود. تا زمانی که رفتار یک کلاس وابسته به وضعیت یک کلاس مشاهده‎شونده باشد، این کلاس باید در لیست مشاهده‎شونده قرار داشته باشد و در آن ثبت شود. زمانی که رفتار این کلاس مستقل از مشاهده‎شونده می‎شود، باید آن را از این لیست حذف نمایید.[2]



شکل 2 نمودار کلاس ها[2]

با داشتن این لیست از مشاهده‎گرها می‌توانیم به راحتی آنها را از تغییری که در متغیر کلاس مشاهده‎شونده ایجاد می‌شود، با‎خبر سازیم. برای این کار لازم است که تغییراتی که در جهت وضعیت کلاس مشاهده شونده است با استفاده از یک تابع تنظیم کننده[[16]](#footnote-16) صورت گیرد. همانطور که در کد زیر می بینید ابتدا وضعیت کلاس مشاهده‎شونده را تغییر می‎دهیم و سپس به ازای همه مشاهده‎گرها در لیست مشاهده‎شونده یک بار تابع onChange یا تغییر را صدا خواهیم زد. همچنین مقدار جدید برای آن متغییر را در آرگومان انتخابه قرار می دهیم کلاس های مشاهده گر با گرفتن این آرگومان می‎توانند تغییرات و رفتارهایی که وابسته به این وضعیت بوده را کنترل کنند و در صورت نیاز به‎روز‎رسانی انجام دهند.

این کد نمونه ای از پیاده‎سازی الگوری مشاهده‎گر در جاوا است.[2]

**package** refactoring\_guru.observer.example.publisher;

**import** refactoring\_guru.observer.example.listeners.EventListener;

**import** java.io.File;

**import** java.util.ArrayList;

**import** java.util.HashMap;

**import** java.util.List;

**import** java.util.Map;

**public** **class** **EventManager** {

Map<**String**, List<EventListener>> listeners = **new** HashMap<>();

**public** EventManager(**String**... operations) {

**for** (**String** operation : operations) {

**this**.listeners.put(operation, **new** ArrayList<>());

}

}

**public** **void** subscribe(**String** eventType, EventListener listener) {

List<EventListener> users = listeners.get(eventType);

users.add(listener);

}

**public** **void** unsubscribe(**String** eventType, EventListener listener) {

List<EventListener> users = listeners.get(eventType);

users.remove(listener);

}

**public** **void** notify(**String** eventType, File file) {

List<EventListener> users = listeners.get(eventType);

**for** (EventListener listener : users) {

listener.update(eventType, file);

}

}

}

# داده‎های هوشیار

حال که با الگوی طراحی مشاهده‎گر آشنا شدیم می‌توانیم به موضوع داده‎های هوشیار در اندروید بپردازید. استفاده از داده‎های هوشیار در زمان هایی که ما از معماری MVVM استفاده می کنیم می‎تواند مفید باشد. در معماری MVVM ما با هر تغییر در سمت مدل هایمان نیاز است که رابط گرافیکی را به طور کلی آپدیت کنیم. این داده‎های هوشیار در اینجا به ما بسیار کمک می کند. به‎گونه‌ای که هرنوع تغییر وضعیتی که نیاز به به‎روزرسانی رابط کاربری[[17]](#footnote-17) داشته باشد را می توانیم مدیریت کنیم. با استفاده از این داده‎های پویا منتظر تغییرات وضعیت بمانیم و به محض آن که تغییری در وضعیت این متغیرها به وجود آمد به‎روزرسانی‎های لازم را برای رابطه گرافیکی انجام دهیم.

مبحث داده‎های هوشیار در چهار چوب اندروید در سال‎های اخیر بسیار مورد استفاده قرار گرفته است به‎گونه ای که ابزار های مدیریت پایگاه داده[[18]](#footnote-18) برای مدیریت داده‎های کاربر بر بروی گوشی، مانند کتابخانه Room که توسط گوگل ساخته شده است، از این داده‎های هوشیار برای پیاده سازی پایگاه داده استفاده کرده‎اند و بصورت همروند عمل میکنند. قبل از صحبت در مورد نحوه استفاده از داده‎های پویا در MVVM لازم است که به تفاوت آن‎ها با مشاهده‎شونده اشاره کنیم و همچنین انواع اصلی آن را با هم بررسی کنیم.

## تفاوت کلاس داده‎ی هوشیار و تفاوت با مشاهده‎شونده عادی

در کل یک کلاس به نام LiveData وجود دارد که تقریبا می توان گفت همان کلاس مشاهده‎شونده ما است. فرق اصلی داده‎های پویا با کلاس مشاهده‎شونده درون‎ساخته‎شده در زبان جاوا این مسئله هست که در LiveData لیست مشاهده‎شونده به صورت مدیریت شده قرار دارد و حضور مشاهده‎گرها در این لیست با وضعیت زنده بودن اجزا اندروید تکامل یافته و همروند شده است. به‎گونه‎ای که با پایان یافتن عمر یکی از اجزای اندروید مانند یک صفحه[[19]](#footnote-19) تمام مشاهده‎گرهای وابسته به آن چرخه حیات از بین خواهند رفت[1] و حذف خواهند شد. این عملیات نمونه‌ای از مزایای کلاس داده هوشیار بر کلاس مشاهده شونده در زبان جاوا است چرا که این موضوع باعث می‎شود ما به مشکل نشت مموری نخوریم. همچنین این روند به ما کمک می‌کند تا دیگر لازم نباشد به صورت دستی بخواهیم رابط کاربری گرافیکی را برای برنامه خود کنترل نمایید.

## نحوه ساخت یک داده هوشیار

کلاس‎های داده‎های هوشیار از مفهوم چند نوعی[[20]](#footnote-20) استفاده می‎کنند بصورتی که ما نوع متغییر و وضعیت را در هنگام تعریف این نوع مشخص میکنیم.

**public** **MutableLiveData<String> currentName = new MutableLiveData();**

## بررسی کلاس‎های داده‎های هوشیار

### کلاس LiveData [3]

ما در اینجا می‎خواهیم به بررسی سه نمونه از کلاس‎های داده پویا در اندروید بپردازیم. اولین و پایه‎ای‎ترین کلاس که به عبارتی پدر کلاس‎های دیگر هم است. کلاس LiveData که دقیقا همین الگوی طراحی مشاهده‎گر را در خود پیاده‎سازی کرده است. نکته‎ای که در مورد این کلاس وجود دارد این است که متغیرهایی و شی‎هایی که از این جنس ساخته می شوند قابل تغییر نخواهند بود، یعنی نمیتوان وضعیت آن‎ها را تغییر داد. پس تابعی در آنها موجود نیست که با استفاده از آن‌ها بتوان وضعیت مشاهده‎شونده را تغییر داد. در اینجا شاید سوال به وجود بیاید که چرا باید همچین کلاسی را داشته باشیم. جلوتر در زمانی که از استفاده داده پویا در لایه View Model صحبت می کنیم به فلسفه وجود این کلاس پی میبریم.

### کلاس MutableLiveData [4]

کلاس دومی که در اینجا قصد بر بررسی آن را داریم، کلاس داده هوشیار تغییرپذیر یا MutableLiveData است. همان‎گونه که از اسم این کلاس مشخص است برخلاف پدرش که همان LiveData است، می‎توان با استفاده از توابع تنظیم‎کننده‎ای مقدار آن را اصطلاحا تنظیم کرد[[21]](#footnote-21) یا فرستاد[[22]](#footnote-22). پس مقدار وضعیت و متغیر درون این داده پویا را تغییر می‎توان داد. بعد از این تغییر همه مشاهده‎کنندگان موجود در لیست این مشاهده‎شونده از مقدار جدید این متغیر در سرتاسر برنامه باخبر خواهند شد و می‌توانند رفتار خود را به‎روزرسانی نمایند.

### کلاس MediatorLiveData [5]

کلاس دیگری که بررسی نکرده‌ایم کلاس MediatorLiveData است که خود فرزند MutableLiveData هست. حال فرقی که اینکه باکلاس پدر خود دارد این است که می تواند تعدادی منبع[[23]](#footnote-23) را در خود جای دهد. این منابع از جنس داده پویا هستند یعنی این کلاس خود یک داده پویا است که شامل منابعی از داده پویا خواهد بود. حال با تغییر وضعیت در هرکدام از آن منابع نیاز است که تغییری در متغیر این مشاهده‎شونده داده بشود. در حقیقت این کلاس هم مشاهده‎شونده است هم مشاهده‎گر. این منابع مانند مشاهده‎گرها می‎توانند در این کلاس ثبت یا حذف بشنود.

## فرق تنظیم یک مقدار با فرستادن یک مقدار

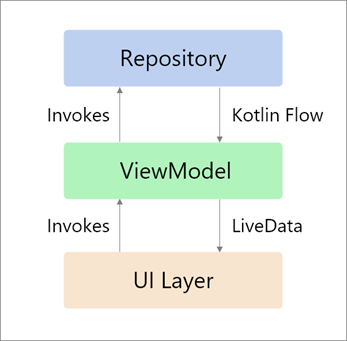
ما برای تغییر وضعیت متغیر درون این داده پویا نیاز داریم که از دو تابع setValue یا postValue استفاده کنیم. فرق این دو متد در بحث موازی‎بودن یا متوالی‎بودن آنها است. اگر از تابع setValue استفاده کنیم به محض اجرای آن خط از کد مقدار جدید در متغیر قرار داده شده و تمام مشاهده‎گرها با مقدار جدید آن وضعیت به روز رسانی خواهد شد. این در حالی است که اگر تابع postValue را استفاده کنیم، داده به صورت موازی تغییر پیدا می‌کند و این بدین معنی است که ممکن است چندین لحظه بعد از اجرای آن خط از کد تازه وضعیت متغیر عوض بشود. برای همین به‎روزرسانی مشاهده‎گر‎ها و اجرای خط های بعدی کد به‎گونه‌ای به صورت موازی و هم روند صورت می‌گیرد. پس باید توجه مهمی به این موضوع داشته باشیم که این اجرا شدن کد ها به صورتی است که ادامه اجرای کد دچار مشکلی نشود. بطور خلاصه تابع تنظیم کننده این تغییر وضعیت را در نخ‎واره اصلی[[24]](#footnote-24) انجام می‎دهد در حالی که عملیات فرستادن در نخ‎واره‎های غیر اصلی‎[[25]](#footnote-25) انجام می‎گیرد.

نکته بسیار مهمی که در اینجا وجود دارد این است که در هنگام تغییر وضعیت اگر در نخ‎واره اصلی نباشیم به هیچ وجه نمی‎توانیم از تابع تنظیم کننده استفاده‎ کنیم. در صورت استفاده از این تابع در نخ‎واره ای به جز نخ‎واره اصلی با پیامی مبنا بر ممنوعیت استفاده از این تابع در زمان اجرای برنامه[[26]](#footnote-26) مواجه خواهیم شد و برنامه ما نابود[[27]](#footnote-27) می‎شود.[1]

# استفاده از داده‏های هوشیار در معماری MVVM

## تعریف در لایه ‏ذخیره‎کننده

استفاده درست از مفهوم داده‎های پویا در معماری MVVM بدین شکل است که در لایه ذخیره‎سازی[[28]](#footnote-28) اطلاعات، متغیرهایی از نوع داده پویا داریم. لازم به بیان این موضوع دیده می‎شود که لایه ذخیره‎سازی بطوری مسئولیت نگهداری وضعیت برنامه را به عهده دارد. حال این لایه می‎تواند تابعی از رفتار کاربر باشد یا می تواند بیانگر وضعیت پایگاه داده ما و یا پاسخی که از سمت ابر[[29]](#footnote-29) می آید باشد. این داده‎های هوشیار باید از نوع MutableLiveData باشند.



شکل 3 روند طراحی در MVVM[1]

## تعریف در لایه View Model

در قسمت دوم، یعنی View Model ما باید متغیرهایی را برای کلاس View Model تعریف کنیم. این متغییر‎ها از LiveData هستند و مقادیر آن‎ها برابر متغیر متناظر آن‎ها در لایه ذخیره‎سازی است. حال سوالی که می‌تواند به وجود بیاید این است که ما در لایه‎ی ذخیره‌سازی از نوع MutableLiveData استفاده کردیم ولی در لایه View Model آنها را برابر متغیرهایی از نوع LiveData بودند قرار می دهیم. چگونه مشکلی به وجود نمی آید؟ بحث چندریختی[[30]](#footnote-30) یکی از اجزاء برنامه نویسی شی گرا[[31]](#footnote-31) است. همان‎گونه که می‌دانید یک متغیر از نوع پدش می‎تواند برابر یک متغیر از نوع بچه‎های خود قرار بگیرد و این دقیقاً یعنی همان چندریختی. پس مشکلی به وجود نخواهد آمد. تنها تأثیری که این کار ما خواهد داشت این است که کدهای نوشته شده در لایه View بدون اجازه ما قادر به تغییر وضعیت این داده های پویا نخواهند داشت مگر با استفاده از توابعی که در این View Model تعریف می کنید و در آن ها دوباره با استفاده از توابع که در ذخیره‎شونده موجود است به تغییر وضعیت آن داده پویا می پردازیم. البته لازم به ذکر است که راه‎های دیگری برای پیاده‎سازی وجود دارند که بستگی به شرایط دارند ولی ساختار اصلی به همین شکل است. همان گونه که دیدید کلاس لایو دیتا در View Model باعث شد که ما با وجود اینکه داده‎های پویایی از وضعیت کنونی برنامه داریم ولی نتوانیم در عین حال مقدار آن را تغییر دهیم. یعنی به‌گونه‌ای این رابطه یک طرفه است و تنها قادر به استفاده از آخرین وضعیت دقیق آن متغیر در مشاهده‎شونده هستیم و تمام روابط ما با لایه بعدی کنترل شده است. همه متدهایی که تغییری بوجود می‎آورند را باید خودمان تعریف کنیم.

## لایه رابط کاربری یا View

تنها کاری که لازم است انجام بدهیم این است که از تابع observe() که در کلاس LiveData پیاده‌سازی شده است استفاده کنیم که به عنوان ورودی این تابع نیاز است که صاحب چرخه[[32]](#footnote-32) جز اندرویدی که در آن هستیم را قرار بدهیم. همچنین به عنوان آرگومان دوم یک تابع صدا زننده[[33]](#footnote-33) تعریف می شود که مسئول رفتار کلاس مشاهده‎گر است و همیشه به محض ایجاد تغییر در وضعیت متغیر مشاهده شونده، از تغییر و مقدار جدید آن برای متغیر باخبر می‌شود و می‌تواند وضعیت برنامه را مانند رابط گرافیکی کاربر را به‎روزرسانی نماید.

کد زیرنمونه استفاده از موارد بالا است.[1]

**// Create the observer which updates the UI.**

**final Observer<String> nameObserver = new Observer<String>() {**

**@Override**

**public void onChanged(@Nullable final String newName) {**

**// Update the UI, in this case, a TextView.**

**nameTextView.setText(newName);**

**}**

**};**

**// Observe the LiveData, passing in this activity as the LifecycleOwner and the observer.**

**model.getCurrentName().observe(this, nameObserver);**

# منابع

[1] D. A. Document. "Live Data." <https://developer.android.com/topic/libraries/architecture/livedata> (accessed.

[2] Refactoring.guru. "Observer Design Pattern." <https://refactoring.guru/design-patterns/observer> (accessed.

[3] D. A. Document. "Live Data Class." <https://developer.android.com/reference/android/arch/lifecycle/LiveData> (accessed.

[4] D. A. Document. "Mutable Live Data." <https://developer.android.com/reference/android/arch/lifecycle/MutableLiveData> (accessed.

[5] D. A. Document. "Mediator Live Data." <https://developer.android.com/reference/android/arch/lifecycle/MediatorLiveData> (accessed.

1. Live Data [↑](#footnote-ref-1)
2. Model View View-Model [↑](#footnote-ref-2)
3. Built-in [↑](#footnote-ref-3)
4. Android life cycles [↑](#footnote-ref-4)
5. Integrated [↑](#footnote-ref-5)
6. Memory lick [↑](#footnote-ref-6)
7. Design pattern [↑](#footnote-ref-7)
8. Observer [↑](#footnote-ref-8)
9. Framework [↑](#footnote-ref-9)
10. Model View Controller [↑](#footnote-ref-10)
11. Observer [↑](#footnote-ref-11)
12. Observable [↑](#footnote-ref-12)
13. Observe [↑](#footnote-ref-13)
14. Register/Subscribe [↑](#footnote-ref-14)
15. Unregister/Unsubscribe [↑](#footnote-ref-15)
16. Setter [↑](#footnote-ref-16)
17. User Interface [↑](#footnote-ref-17)
18. Database Management Systems Tools [↑](#footnote-ref-18)
19. Activity [↑](#footnote-ref-19)
20. Generics [↑](#footnote-ref-20)
21. Set [↑](#footnote-ref-21)
22. Post [↑](#footnote-ref-22)
23. Source [↑](#footnote-ref-23)
24. Main Thread / UI Thread [↑](#footnote-ref-24)
25. Background Threads [↑](#footnote-ref-25)
26. Runtime [↑](#footnote-ref-26)
27. Crash [↑](#footnote-ref-27)
28. Repository [↑](#footnote-ref-28)
29. Cloud [↑](#footnote-ref-29)
30. Polymorphism [↑](#footnote-ref-30)
31. Object Oriented Programming [↑](#footnote-ref-31)
32. Life Cycle Owner [↑](#footnote-ref-32)
33. Callback Function [↑](#footnote-ref-33)