Retrofit

You can use replacement blocks and query parameters to adjust the URL. A replacement block is added to the relative URL with `{}`. With the help of the `@Path` annotation on the method parameter, the value of that parameter is bound to the specific replacement block.

```
@GET("users/{name}/commits")
Call<List<Commit>> getCommitsByName(@Path("name") String name);
```

Query parameters are added with the `@Query` annotation on a method parameter. They are automatically added at the end of the URL.

```
@GET("users")
Call<User> getUserById(@Query("id") Integer id);
```

The `@Body` annotation on a method parameter tells Retrofit to use the object as the request body for the call.

```
@POST("users")
Call<User> postUser(@Body User user)
```

@Url annotation

@Url annotation. With the help of this annotation, we can provide the URL for this request. This allows us to change the URL for each request dynamically. We need this for the *comments_url* field of the GithubIssue class.

The `@Path` annotation binds the parameter value to the corresponding variable (curly brackets) in the request URL. This is needed to specify the owner and the repository name for which the issues should be requested

```
    @GET("/repos/{owner}/{repo}/issues")
    Single<List<GithubIssue>> getIssues(@Path("owner") String owner, @Path("repo") String
repository);

    @POST
    Single<ResponseBody> postComment(@Url String url, @Body GithubIssue issue);
```

Interceptor

Interceptor this is class that can costomise every rquest that we make by retrofit

```
class AuthInterceptor(private val apiKey: String) : Interceptor {

    override fun intercept(chain: Interceptor.Chain): Response {
        val request = chain.request().newBuilder()
            // Добавляем auth header в запрос
            .addHeader("api-key", apiKey)
            .build()

        return chain.proceed(request)
    }
}
```

```
OkHttpClient.Builder()
    .addInterceptor(AuthInterceptor(API_KEY))
    .build()
```

and add OkHttP to our Retrofit

```
Retrofit retrofit = new Retrofit.Builder()
    .baseUrl("https://api.example.com")
    .client(okHttpClient)
    .build();
```

ViewModel

The `ViewModel` class is designed to store and manage the UI-related data. In this app, each `ViewModel` is associated with one fragment.

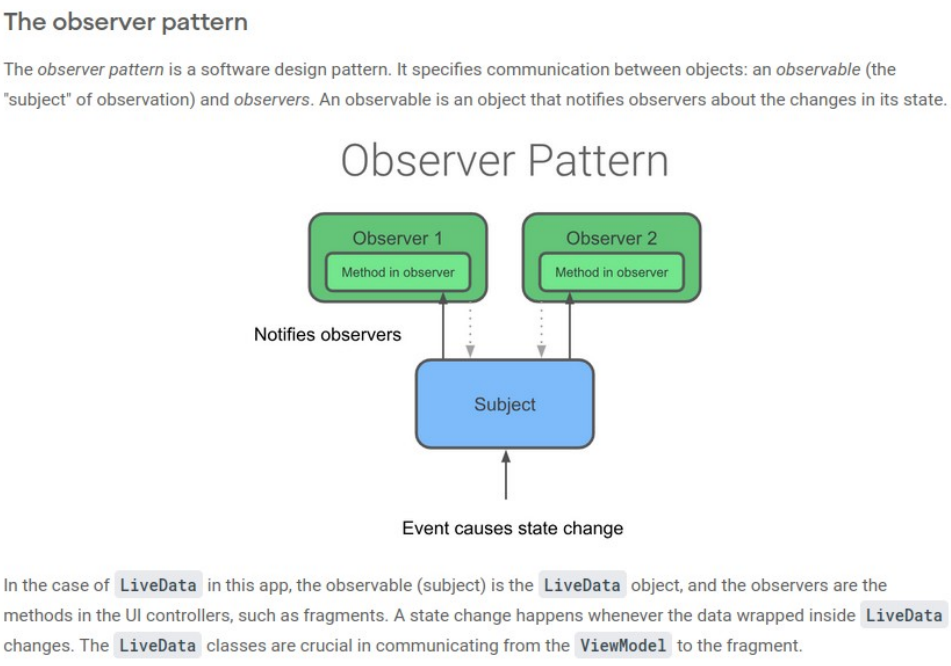
To create view model class we have to use ViewModelProvider and then the esepmlar of class ViewModel will not recreate every time when activity will cahnge oriebtation of screen.

To put bata inti ViewModel constructor we have to use View model Factory

```
class ScoreViewModelFactory(private val finalScore: Int) : ViewModelProvider.Factory {
   override fun <T : ViewModel?> create(modelClass: Class<T>): T {
      if (modelClass.isAssignableFrom(ScoreViewModel::class.java)) {
         return ScoreViewModel(finalScore) as T
      }
      throw IllegalArgumentException("Unknown ViewModel class")

   }}
```

LifeData



In the case of `LiveData` in this app, the observable (subject) is the `LiveData` object, and the observers are the methods in the UI controllers, such as fragments. A state change happens whenever the data wrapped inside `LiveData` changes. The `LiveData` classes are crucial in communicating from the `ViewModel` to the fragment.

data binding

App arcitacture wiithout databinding inside VM

ViewModel passed into the data binding

It would be simpler if the views in the layout communicated directly with the data in the `ViewModel` objects, without relying on UI controllers as intermediaries.


```
<data>

    <variable
        name="gameViewModel"
        type="com.example.android.guesstheword.screens.game.GameViewModel" />
</data>
```

Aftrer this inside constructor of Fragment (**IMPORTANT**)

In this step, you replace the click listeners in the `GameFragment` with listener bindings in the `game_fragment.xml` file.

1. In `game_fragment.xml`, add the `onClick` attribute to the `skip_button`. Define a binding expression and call the `onSkip()` method in the `GameViewModel`. This binding expression is called a *listener binding*.

```
<Button
   android:id="@+id/skip_button"
   ...
   android:onClick="@{() -> gameViewModel.onSkip()}"
   ... />
```

Life data width Data binding

Set it to the `LiveData` object, `word` from the `GameViewModel`, using the binding variable, `gameViewModel`.

```
<TextView
   android:id="@+id/word_text"
   ...
   android:text="@{gameViewModel.word}"
   ... />
```

Notice that you don't have to use `word.value`. Instead, you can use the actual `LiveData` object. The `LiveData` object displays the current value of the `word`. If the value of `word` is null, the `LiveData` object displays an empty string.

2. In the `GameFragment`, in `onCreateView()`, after initializing the `gameViewModel`, set the fragment view as the lifecycle owner of the `binding` variable. This defines the scope of the `LiveData` object above, allowing the object to automatically update the views in the layout, `game_fragment.xml`.

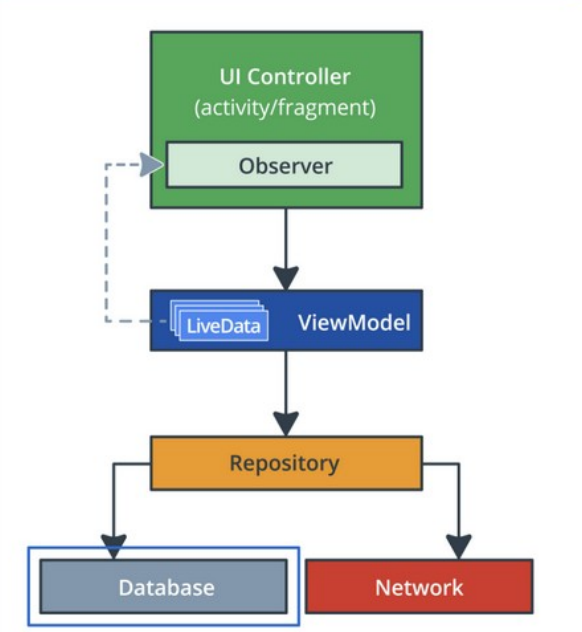
```
binding.gameViewModel = ...
// Specify the fragment view as the lifecycle owner of the binding.
// This is used so that the binding can observe LiveData updates
binding.lifecycleOwner = viewLifecycleOwner
```

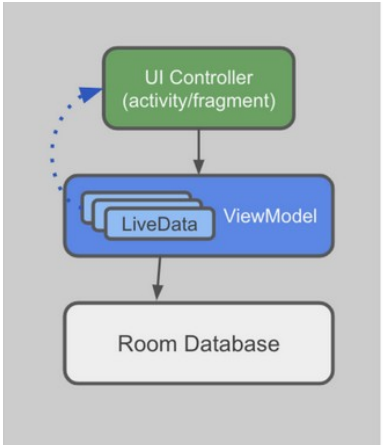
Transformations Lifedata

The `Transformations.map()` method provides a way to perform data manipulations on the source `LiveData` and return a result `LiveData` object. These transformations aren't calculated unless an observer is observing the returned `LiveData` object.

Room

Three main part		
Entities (Table)	DAO	Dtabace (include all Entities)





entity *represents an object or a concept, along with its properties, to store in the database. In our application code, we need an entity class that defines a table*

query *is a request for data or information from a database table or combination of tables, or a request to perform an action on the data*

DAO

Comonly used DAO annotation

@Insert

@Update

@Query("")

DATABACE

`@Volatile` - *The value of a volatile variable will never be cached, and all writes and reads will be done to and from the main memory. This helps make sure the value of* `INSTANCE` *is always up-to-date and the same to all execution threads. It means that changes made by one thread to* `INSTANCE` *are visible to all other threads immediately, and you don't get a situation where, say, two threads each update the same entity in a cache, which would create a problem.*

To INIT DATABACE :

```
companion object {

    @Volatile
    private var INSTANCE: SleepDatabase? = null

    fun getInstance(context: Context): SleepDatabase {
        synchronized(this) {
            var instance = INSTANCE

            if (instance == null) {
                instance = Room.databaseBuilder(
                    context.applicationContext,
                    SleepDatabase::class.java,
                    "sleep_history_database"
                )
                    .fallbackToDestructiveMigration()
```

```
                .build()
            INSTANCE = instance
        }
        return instance
    }
  }
}
```

Dagger 2

Dagger 2 is dependency injection framework. It is based on the Java Specification Request (JSR) 330. It uses code generation and is based on annotations. The generated code is very relatively easy to read and debug.

Dagger supports all three types of injection: *Constructor*, *field* and *method* injection.

You should use *constructor injection* the most because it allows you to set all the dependencies for an object when you create it. This is the type of injection you used for the `NewsDetailPresenterImpl` class.

Sometimes this isn't possible because you don't have direct control over the creation of the instance of a class. This is the case of classes like `Activity` and `Fragment` whose lifecycle is the responsibility of the Android environment. In this case, you use the *field injection*, which injects dependencies into a class field, like

Dagger 2 uses the following annotations:

- `@Module` and `@Provides`: define classes and methods which provide dependencies
- `@Inject`: request dependencies. Can be used on a constructor, a field, or a method
- `@Component`: enable selected modules and used for performing dependency injection

Dagger 2 uses generated code to access the fields and not reflection. Therefore it is not allowed to use private fields for field injection.

2.6. Special treatment of fields in Dagger

Dagger 2 does not inject fields automatically. It can also not inject private fields. If you want to use field injection you have to define a method in your `@Component` interface which takes the instance into which you want to inject as parameter.

```
package com.vogella.java.dagger2.component;

import javax.inject.Singleton;

import com.vogella.java.dagger2.BackendService;
import com.vogella.java.dagger2.modules.BackEndServiceModule;
import com.vogella.java.dagger2.modules.UserModule;

import dagger.Component;

@Singleton
@Component(modules = { UserModule.class, BackEndServiceModule.class })
public interface MyComponent {

    // provide the dependency for dependent components
    // (not needed for single-component setups)
    BackendService provideBackendService();

    // allow to inject into our Main class
    // method name not important
    void inject(Main main);
}
```

3.5. Define components

Components define from which modules (or other components) dependencies are provided. Dagger 2 uses this interface to generate the accessor class which provides the methods defined in the interface.

First of all we have to create instance of Daggercomponent

```
private Main() {
    component = DaggerMyComponent.builder().build();
    component.inject(this);
}
```

We have to initialize dagger inside class where we inject fields

```
@Override
public void onCreate() {
    super.onCreate();
    DaggerMyApplicationComponent.create().inject(this);
}
```

When `@Inject` is annotated on a class constructor, it's telling Dagger how to provide instances of that class. When it's annotated on a class field, it's telling Dagger that it needs to populate the field with an instance of that type.

@Binds

- Дает даггеру понять какую реализацию подставлять как зависимость на место интерфейса.
- Используется с абстрактным методом.
- Даггер не вызывает и не имплементирует биндинг метод.
- Реализация - класс в параметре.

- Интерфейс - возвращаемый тип.

Use `@Binds` to tell Dagger which implementation it needs to use when providing an interface.

@BindsInstance

We use this annotation to pass Context of application bt Component Factory

```
@Component(modules = [StorageModule::class])
interface AppComponent {

    // Factory to create instances of the AppComponent
    @Component.Factory
    interface Factory {
        // With @BindsInstance, the Context passed in will be available in the graph
        fun create(@BindsInstance context: Context): AppComponent
    }

    fun inject(activity: RegistrationActivity)
}
```

`@BindsInstance` tells Dagger that it needs to add that instance in the graph and whenever `Context` is required, provide that instance.

Subcomponents

Subcomponents are components that inherit and extend the object graph of a parent component. Thus, all objects provided in the parent component will be provided in the subcomponent too. In this way, an object from a subcomponent can depend on an object provided by the parent component.

```
import dagger.Subcomponent

// Definition of a Dagger subcomponent
@Subcomponent
interface RegistrationComponent {

}
```

And we have to declare subcomponent inside app component

```kotlin
@Singleton
@Component(modules = [StorageModule::class])
interface AppComponent {

    @Component.Factory
    interface Factory {
        fun create(@BindsInstance context: Context): AppComponent
    }

    // Expose RegistrationComponent factory from the graph
    fun registrationComponent(): RegistrationComponent.Factory

    fun inject(activity: MainActivity)
}
```

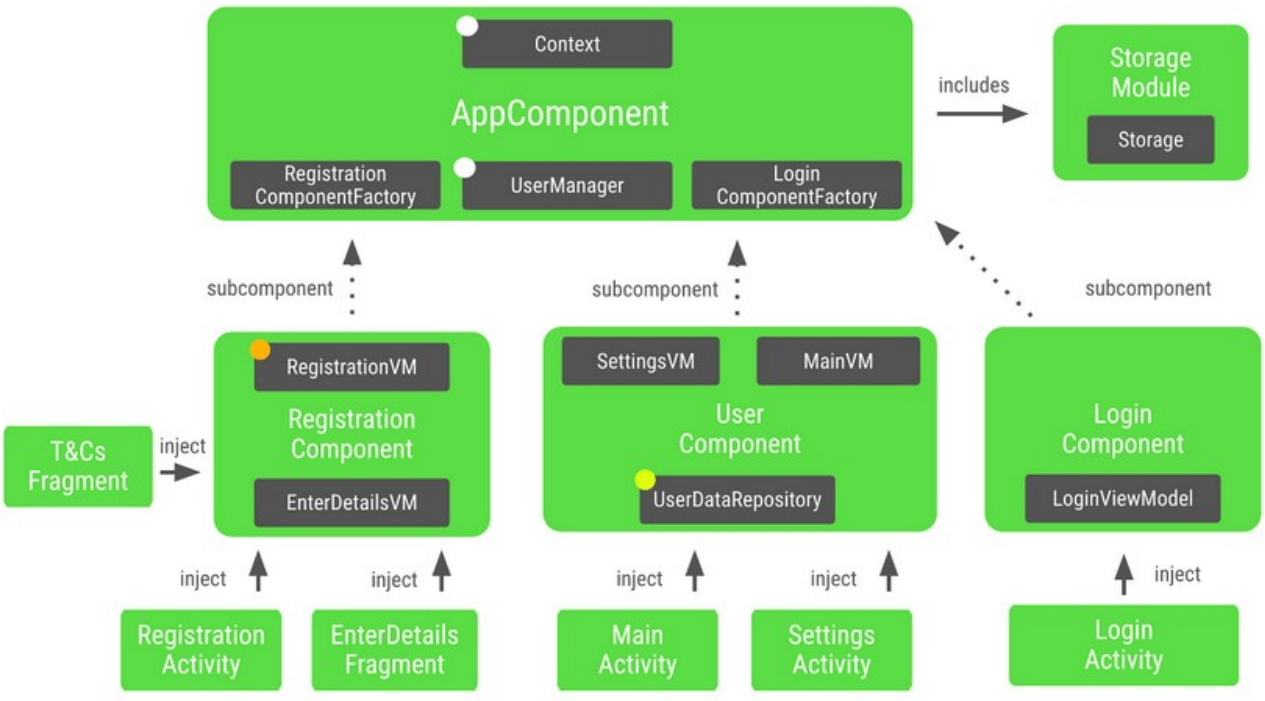
Then we have to add subcomponent to module and add this module to our AppComponent

```kotlin
// This module tells AppComponent which are its subcomponents
@Module(subcomponents = [RegistrationComponent::class])
class AppSubcomponents
```

This new module also needs to be included in the `AppComponent`:

AppComponent.kt

```kotlin
@Singleton
@Component(modules = [StorageModule::class, AppSubcomponents::class])
interface AppComponent { ... }
```



Kotlin fundamential

Classes are final by default; to make a class inheritable, mark it as open.

```
open class Shape

class Rectangle(var height: Double, var length: Double): Shape() {
    var perimeter = (height + length) * 2
}
```

Coroutines

In Kotlin, coroutines are the way to handle long-running tasks elegantly and efficiently instead of callbacks. Kotlin coroutines let you convert callback-based code to sequential code. Code written sequentially is typically easier to read and maintain.

Coroutines use suspend functions to make asynchronous code sequential.

The keyword `suspend` is Kotlin's way of marking a function, or function type, as being available to coroutines. When a coroutine calls a function marked with `suspend`, instead of blocking until the function returns like a normal function call, the coroutine suspends execution until the result is ready. Then the coroutine resumes where it left off, with the result.

While the coroutine is suspended and waiting for a result, it unblocks the thread that it's running on. That way, other functions or coroutines can run.

The `suspend` keyword doesn't specify the thread that the code runs on. A suspend function may run on a background thread, or on the main thread.

`CoroutineScope` :

`ViewModelScope`: `A ViewModelScope is defined for each ViewModel in your app. Any coroutine launched in this scope is automatically canceled if the ViewModel is cleared. In this codelab you will use ViewModelScope to initiate the database operations.`

Room and Dispatcher

When using the Room library to perform a database operation, Room uses a `Dispatchers.IO` to perform the database operations in a background thread. You don't have to explicitly specify any `Dispatchers`. Room does this for you.

Spanned

Type `Spanned`, which is an HTML-formatted string. This is very convenient because Android's TextView has the ability to render basic HTML.

return@label

In Kotlin, the `return@label` syntax specifies the function from which this statement returns, among several nested functions.

```
fun onStopTracking() {
   viewModelScope.launch {
       val oldNight = tonight.value ?: return@launch
       oldNight.endTimeMilli = System.currentTimeMillis()
       update(oldNight)
   }
}
```

in operator

```
when {
    "orange" in items -> println("juicy")
    "apple" in items -> println("apple is fine too")
}
```

is operator

```
fun getStringLength(obj: Any): Int? {
    if (obj is String) {
        // `obj` is automatically cast to `String` in this branch
        return obj.length
    }

    // `obj` is still of type `Any` outside of the type-checked branch
    return null
}
```

Lifecycle

`Lifecycle` is a class that holds the information about the lifecycle state of a component (like an activity or a fragment) and allows other objects to observe this state

LifecycleOwner

`LifecycleOwner` is a single method interface that denotes that the class has a `Lifecycle`. It has one method, `getLifecycle()`, which must be implemented by the class. If you're trying to manage the lifecycle of a whole application process instead, see `ProcessLifecycleOwner`.

LiveData

`LiveData` is an observable data holder class. Unlike a regular observable, LiveData is lifecycle-aware, meaning it respects the lifecycle of other app components, such as activities, fragments, or services. This awareness ensures LiveData only updates app component observers that are in an active lifecycle state.

Save UI states

To bridge the gap between user expectation and system behavior, use a combination of `ViewModel` objects, the `onSaveInstanceState()` method, and/or local storage to persist the UI state across such application and activity instance transitions. Deciding how to combine these options depends on the complexity of your UI data, use cases for your app, and consideration of speed of retrieval versus memory usage.

- Local persistence: Stores all data you don't want to lose if you open and close the activity.
    - Example: A collection of song objects, which could include audio files and metadata.
- `ViewModel`: Stores in memory all the data needed to display the associated UI Controller.
    - Example: The song objects of the most recent search and the most recent search query.
- `onSaveInstanceState()`: Stores a small amount of data needed to easily reload activity state if the system stops and then recreates the UI Controller. Instead of storing complex objects here, persist the complex objects in local storage and store a unique ID for these objects in `onSaveInstanceState()`

Saved State module for ViewModel

Kotlin Java

```kotlin
class SavedStateViewModel(private val state: SavedStateHandle) : ViewModel() { ... }
```

You can then retrieve an instance of your `ViewModel` without any additional configuration. The default `ViewModel` factory provides the appropriate `SavedStateHandle` to your `ViewModel`.

Kotlin Java

```kotlin
class MainFragment : Fragment() {
    val vm: SavedStateViewModel by viewModels()

    ...
}
```

Paging Library

Paging Library содержит инструменты для постраничной подгрузки данных. Т.е. когда данные подгружаются не все сразу, а по мере прокрутки списка.

PagedListAdapter - это RecyclerView.Adapter, заточенный под чтение данных из PagedList.

```java
class EmployeeAdapter extends PagedListAdapter<Employee, EmployeeViewHolder> {

    protected EmployeeAdapter(DiffUtil.ItemCallback<Employee> diffUtilCallback) {
        super(diffUtilCallback);
    }

    @NonNull
    @Override
    public EmployeeViewHolder onCreateViewHolder(@NonNull ViewGroup parent, int viewType) {
        View view = LayoutInflater.from(parent.getContext()).inflate(R.layout.employee, parent, false
        EmployeeViewHolder holder = new EmployeeViewHolder(view);
        return holder;
    }

    @Override
    public void onBindViewHolder(@NonNull EmployeeViewHolder holder, int position) {
        holder.bind(getItem(position));
    }

}
```

PagedList

PagedList - это обертка над List. Он тоже содержит данные и умеет отдавать их методом get(position). Но при этом он проверяет, насколько запрашиваемый элемент близок к концу имеющихся у него данных и при необходимости подгружает себе новые данные с помощью DataSource.

```
PagedList<Employee> pagedList = new PagedList.Builder<>(dataSource, config)
        .setBackgroundThreadExecutor(Executors.newSingleThreadExecutor())
        .setMainThreadExecutor(new MainThreadExecutor())
        .build();
```

DataSource

DataSource - это посредник между PagedList и Storage. Он должен расширять extends **PositionalDataSource<Класс который поучаем из стореджа>** и должен переопределять функции :

```
@Override
public void loadInitial(@NonNull LoadInitialParams params, @NonNull
LoadInitialCallback<Employee> callback)

@Override
public void loadRange(@NonNull LoadRangeParams params, @NonNull
LoadRangeCallback<Employee> callback)
```

1) loadInitial - первоначальная загрузка данных.
Когда мы создаем PagedList, он сразу запрашивает порцию данных у DataSource. Делает он это методом loadInitial. В качестве параметров он передает нам:
requestedStartPosition - с какой позиции подгружать
requestedLoadSize - размер порции

Используя эти параметры, мы запрашиваем данные у Storage. Полученный результат передаем в callback.onResult

2) loadRange - подгрузка новой порции данных

Когда мы прокручиваем список, PagedList подгружает новые данные. Для этого он вызывает метод loadRange. В качестве параметров он передает нам позицию, с которой надо подгружать данные, и размер порции.

Используя эти параметры, мы запрашиваем данные у Storage. Полученный результат передаем в callback.onResult

WorkManager

Необходимо сделать наследование класса Worker и переопределить метод doWork()

```
public class MyWorker extends Worker {

    static final String TAG = "workmng";

    @NonNull
    @Override
    public WorkerResult doWork() {
        Log.d(TAG, "doWork: start");

        try {
            TimeUnit.SECONDS.sleep(10);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        Log.d(TAG, "doWork: end");

        return WorkerResult.SUCCESS;
    }
}
```

После этого необходим MyWorker обернуть в [WorkRequest]

```
OneTimeWorkRequest myWorkRequest = new OneTimeWorkRequest.Builder(MyWorker.class).build();
```

OneTimeWorkRequest – выполняеться один раз и всё

PeriodicWorckRequest – выполняеться переодично

```
val saveRequest =
   PeriodicWorkRequestBuilder<SaveImageToFileWorker>(1, TimeUnit.HOURS)
    // Additional configuration
           .build()
```

Запустим воркменеджер

```
WorkManager.getInstance().enqueue(myWorkRequest);
```

WorkManager позволяет нам задать критерии запуска задачи

Критерий :

Constraints constraints = new Constraints.Builder()

.setRequiresCharging(true)

.build();

Добавление критерия :

```
OneTimeWorkRequest myWorkRequest = new OneTimeWorkRequest.Builder(MyWorker.class)
       .setConstraints(constraints)
           .build();
```

Критерии :

setRequiresBatteryNotLow - Критерий: уровень батареи не ниже критического.

SetRequiredNetworkType - Критерий: наличие интернет

Запуск задач паралельно в разных потоках

WorkManager.getInstance().enqueue(myWorkRequest1, myWorkRequest2, myWorkRequest3);

Последывательное выполнение задач

```
WorkManager.getInstance()
.beginWith(myWorkRequest1)
.then(myWorkRequest2)
.then(myWorkRequest3)
.enqueue();
```

Unique work

Мы можем сделать последовательность задач уникальной. Для этого начинаем последовательность методом beginUniqueWork.

```
WorkManager.getInstance()
.beginUniqueWork("work123", ExistingWorkPolicy.REPLACE, myWorkRequest1)
.then(myWorkRequest3)
.then(myWorkRequest5)
.enqueue();
```

В качестве режима мы указали **REPLACE**. Это означает, что, если последовательность с таким именем уже находится в работе, то еще один запуск приведет к тому, что текущая выполняемая последовательность будет остановлена, а новая запущена.

Режим **KEEP** оставит в работе текущую выполняемую последовательность. А новая будет проигнорирована.

Передача данных в Воркменеджер :

Сначала рассмотрим как передать в задачу входные данные:

```
Data myData = new Data.Builder()
       .putString("keyA", "value1")
       .putInt("keyB", 1)
       .build();

OneTimeWorkRequest myWorkRequest1 = new OneTimeWorkRequest.Builder(MyWorker1.class)
       .setInputData(myData)
       .build();
```

Данные помещаем в объект Data с помощью его билдера. Далее этот объект передаем в метод setInputData билдера WorkRequest.

Когда задача будет запущена, то внутри ее (в MyWorker1.java) мы можем получить эти входные данные так:

```
String valueA = getInputData().getString("keyA", "");
int valueB = getInputData().getInt("keyB", 0);
```

Получение данных из Воркменеджера :

Когда задача обработана, нужно вернуть результат обработки. Входящие - это inputData, а исходящие - outputData:

```
class MainWorker : Worker() {
    override fun doWork(): WorkerResult {
        Log.d("MainWorker", inputData.getString("time",""))
        outputData = Data.Builder()
            .putString("name", "SouthernBox")
            .build()
        return WorkerResult.SUCCESS
    }
}
Скопировать код
```

Каждый WorkRequest будет иметь идентификатор, с помощью которого можно получить WorkStatus соответствующе задачи, и он предоставляется в форме LiveData:

```
WorkManager.getInstance()
        .getStatusById(request.id)
        .observe(this, Observer { workStatus ->
            if (workStatus != null && workStatus.state.isFinished) {
                Log.d("MainActivity", workStatus.outputData.getString("name", ""))
            }
        })
Скопировать код
```

NOTIFICATION

Before you can deliver the notification on Android 8.0 and higher, you must register your app's notification channel with the system by passing an instance of `NotificationChannel` to `createNotificationChannel()`. So the following code is blocked by a condition on the `SDK_INT` version:

```
private fun createNotificationChannel() {
    // Create the NotificationChannel, but only on API 26+ because
    // the NotificationChannel class is new and not in the support library
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
        val name = getString(R.string.channel_name)
        val descriptionText = getString(R.string.channel_description)
        val importance = NotificationManager.IMPORTANCE_DEFAULT
        val channel = NotificationChannel(CHANNEL_ID, name, importance).apply {
            description = descriptionText
        }
        // Register the channel with the system
        val notificationManager: NotificationManager =
            getSystemService(Context.NOTIFICATION_SERVICE) as NotificationManager
        notificationManager.createNotificationChannel(channel)
    }
}
```

After this you can create notofocation by bilder

```
var builder = NotificationCompat.Builder(this, CHANNEL_ID)
        .setSmallIcon(R.drawable.notification_icon)
        .setContentTitle(textTitle)
        .setContentText(textContent)
        .setPriority(NotificationCompat.PRIORITY_DEFAULT)
```

Show the notification

To make the notification appear, call `NotificationManagerCompat.notify()`, passing it a unique ID for the notification and the result of `NotificationCompat.Builder.build()`. For example:

Kotlin Java

```
with(NotificationManagerCompat.from(this)) {
    // notificationId is a unique int for each notification that you must define
    notify(notificationId, builder.build())
}
```

Animations

1)Property Animation

```
ObjectAnimator.ofFloat(textView, "translationX", 100f).apply {
    duration = 1000
    start()
}
```

Animation listeners

You can listen for important events during an animation's duration with the listeners described below.

- `Animator.AnimatorListener`
  - `onAnimationStart()` - Called when the animation starts.
  - `onAnimationEnd()` - Called when the animation ends.
  - `onAnimationRepeat()` - Called when the animation repeats itself

Specify keyframes

A `Keyframe` object consists of a time/value pair that lets you define a specific state at a specific
Each keyframe can also have its own interpolator to control the behavior of the animation in the i
previous keyframe's time and the time of this keyframe.

To instantiate a `Keyframe` object, you must use one of the factory methods, `ofInt()`, `ofFloat`
to obtain the appropriate type of `Keyframe`. You then call the `ofKeyframe()` factory method to
`PropertyValuesHolder` object. Once you have the object, you can obtain an animator by passir
`PropertyValuesHolder` object and the object to animate. The following code snippet demonst

Kotlin Java

```
val kf0 = Keyframe.ofFloat(0f, 0f)
val kf1 = Keyframe.ofFloat(.5f, 360f)
val kf2 = Keyframe.ofFloat(1f, 0f)
val pvhRotation = PropertyValuesHolder.ofKeyframe("rotation", kf0, kf1, kf2)
ObjectAnimator.ofPropertyValuesHolder(target, pvhRotation).apply {
    duration = 5000
}
```

Разница между object animator и properety animator

One ObjectAnimator

Kotlin Java

```
val pvhX = PropertyValuesHolder.ofFloat("x", 50f)
val pvhY = PropertyValuesHolder.ofFloat("y", 100f)
ObjectAnimator.ofPropertyValuesHolder(myView, pvhX, pvhY).start()
```

ViewPropertyAnimator

Kotlin Java

```
myView.animate().x(50f).y(100f)
```

AnimationDrawable

One way to animate Drawables is to load a series of Drawable resources one after another to create an animation. This is a traditional animation in the sense that it is created with a sequence of different images, played in order, like a roll of film. The AnimationDrawable class is the basis for Drawable animations.

```
<animation-list xmlns:android="http://schemas.android.com/apk/res/android"
    android:oneshot="true">
    <item android:drawable="@drawable/rocket_thrust1" android:duration="200" />
    <item android:drawable="@drawable/rocket_thrust2" android:duration="200" />
    <item android:drawable="@drawable/rocket_thrust3" android:duration="200" />
</animation-list>
```

```
private lateinit var rocketAnimation: AnimationDrawable

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.main)

    val rocketImage = findViewById<ImageView>(R.id.rocket_image).apply {
        setBackgroundResource(R.drawable.rocket_thrust)
        rocketAnimation = background as AnimationDrawable
    }

    rocketImage.setOnClickListener({ rocketAnimation.start() })
}
```

Animating by Pass

```
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.LOLLIPOP) {
    val path = Path().apply {
        arcTo(0f, 0f, 1000f, 1000f, 270f, -180f, true)
    }
    val animator = ObjectAnimator.ofFloat(view, View.X, View.Y, path).apply {
        duration = 2000
        start()
    }
} else {
    // Create animator without using curved path
}
```

`Transition` animation

We create two scenes from layout where Ids is

Then we can create Xml width describing of transition

Class	Tag	Attributes	Effect
`AutoTransition`	<autoTransition/>	-	Default transition. Fade out, move and resize, and fade in views, in that order.

```
var fadeTransition: Transition =
    TransitionInflater.from(this)
                      .inflateTransition(R.transition.fade_transition)
```

Create a transition instance in your code

This technique is useful for creating transition objects dynamically if you modify the user interface in your code, and to create simple built-in transition instances with few or no parameters.

To create an instance of a built-in transition, invoke one of the public constructors in the subclasses of the `Transition` class. For example, the following code snippet creates an instance of the `Fade` transition:

Kotlin Java

```
var fadeTransition: Transition = Fade()
```

android:transitionName=""

ViewPager

`ViewPager` objects have built-in swipe gestures to transition through pages, and they display screen slide animations by default, so you don't need to create your own animation. `ViewPager` uses `PagerAdapter` objects as a supply for new pages to display, so the `PagerAdapter` will use the fragment class that you created earlier.

- Sets the content view to be the layout with the `ViewPager`.
- Creates a class that extends the `FragmentStatePagerAdapter` abstract class and implements the `getItem()` method to supply instances of `ScreenSlidePageFragment` as new pages. The pager adapter also requires that you implement the `getCount()` method, which returns the amount of pages the adapter will create (five in the example).
- Hooks up the `PagerAdapter` to the `ViewPager`.

Questions

Question 1

How do you indicate that a class represents an entity to store in a `Room` database?

- Make the class extend `DatabaseEntity`.
- Annotate the class with `@Entity`.
- Annotate the class with `@Database`.
- Make the class extend `RoomEntity` and also annotate the class with `@Room`.

Question 2

The DAO (data access object) is an interface that `Room` uses to map Kotlin functions to database queries.

How do you indicate that an interface represents a DAO for a `Room` database?

- Make the interface extend `RoomDAO`.
- Make the interface extend `EntityDao`, then implement the `DaoConnection()` method.
- Annotate the interface with `@Dao`.
- Annotate the interface with `@RoomConnection`.

Question 3

Which of the following statements are true about the `Room` database? Choose all that apply.

- You can define tables for a `Room` database as annotated data classes.
- If you return `LiveData` from a query, `Room` will keep the `LiveData` updated for you if the `LiveData` changes.
- Each `Room` database must have one, and only one, DAO.
- To identify a class as a `Room` database, make it a subclass of `RoomDatabase` and annotate it with `@Database`.

Question 4

Which of the following annotations can you use in your `@Dao` interface? Choose all that apply.

- `@Get`
- `@Update`
- `@Insert`
- `@Query`

Question 5

How can you verify that your database is working? Select all that apply.

- Write instrumented tests.
- Continue writing and running the app until it displays the data.
- Replace the calls to the methods in the DAO interface by calls to equivalent methods in the `Entity` class.
- Run the `verifyDatabase()` function provided by the `Room` library.

Question 1

Which of the following is not a benefit of using coroutines?:

- They are non-blocking
- They run asynchronously.
- They can be run on a thread other than the main thread.
- They always make app runs faster.
- They can use exceptions.
- They can be written and read as linear code.

Question 2

Which of the following is not true for suspend functions.?

- An ordinary function annotated with the `suspend` keyword.
- A function that can be called inside coroutines.
- While a suspend function is running, the calling thread is suspended.
- Suspend functions must always run in the background.

Question 3

Which of the following statements is NOT true?

- When execution is blocked, no other work can be executed on the blocked thread.
- When execution is suspended, the thread can do other work while waiting for the offloaded work to complete.
- Suspending is more efficient, because threads may not be waiting, doing nothing.
- Whether blocked or suspended, execution is still waiting for the result of the coroutine before continuing.

Question 1

One way to enable your app to trigger navigation from one fragment to the next is to use a `LiveData` value to indicate whether or not to trigger navigation.

What are the steps for using a `LiveData` value, called `gotoBlueFragment`, to trigger navigation from the red fragment to the blue fragment? Select all that apply:

- In the `ViewModel`, define the `LiveData` value `gotoBlueFragment`.
- In the `RedFragment`, observe the `gotoBlueFragment` value. Implement the `observe{}` code to navigate to `BlueFragment` when appropriate, and then reset the value of `gotoBlueFragment` to indicate that navigation is complete.
- Make sure your code sets the `gotoBlueFragment` variable to the value that triggers navigation whenever the app needs to go from `RedFragment` to `BlueFragment`.
- Make sure your code defines an `onClick` handler for the `View` that the user clicks to navigate to `BlueFragment`, where the `onClick` handler observes the `goToBlueFragment` value.

Question 2

You can change whether a `Button` is enabled (clickable) or not by using `LiveData`. How would you ensure that your app changes the `UpdateNumber` button so that:

- The button is enabled if `myNumber` has a value greater than 5.
- The button is not enabled if `myNumber` is equal to or less than 5.

Assume that the layout that contains the `UpdateNumber` button includes the `<data>` variable for the `NumbersViewModel` as shown here:

```
<data>
   <variable
       name="NumbersViewModel"
       type="com.example.android.numbersapp.NumbersViewModel" />
</data>
```

Assume that the ID of the button in the layout file is the following:

`android:id="@+id/update_number_button"`

What else do you need to do? Select all that apply.

- In the `NumbersViewModel` class, define a `LiveData` variable, `myNumber`, that represents the number. Also define a variable whose value is set by calling `Transform.map()` on the `myNumber` variable, which returns a boolean indicating whether or not the number is greater than 5.

Specifically, in the `ViewModel`, add the following code:

```
val myNumber: LiveData<Int>
val enableUpdateNumberButton = Transformations.map(myNumber) {
   myNumber > 5
}
```

- In the XML layout, set the `android:enabled` attribute of the `update_number_button` button to `NumberViewModel.enableUpdateNumbersButton`.

```
android:enabled="@{NumbersViewModel.enableUpdateNumberButton}"
```

- In the `Fragment` that uses the `NumbersViewModel` class, add an observer to the `enabled` attribute of the button.

Specifically, in the `Fragment`, add the following code:

```
// Observer for the enabled attribute
viewModel.enabled.observe(this, Observer<Boolean> { isEnabled ->
   myNumber > 5
})
```