[Урок 1. Lifecycle](https://startandroid.ru/ru/uroki/vse-uroki-spiskom/27-course/architecture-components/524-urok-1.html" \o "Урок 1. Lifecycle)

public class MyServer implements LifecycleObserver {

   @OnLifecycleEvent(Lifecycle.Event.ON\_START)

   public void connect() {

       // ...

   }

   @OnLifecycleEvent(Lifecycle.Event.ON\_STOP)

   public void disconnect() {

       // ...

   }

}

@Override

protected void onCreate(Bundle savedInstanceState) {

   super.onCreate(savedInstanceState);

   setContentView(R.layout.activity\_main);

   // ...

   getLifecycle().addObserver(myServer);

}

|  |  |
| --- | --- |
| 1  2  3  4 | @OnLifecycleEvent(ON\_ANY)  void onAny(LifecycleOwner source, Lifecycle.Event event) {      // ...  } |

# [Урок 2. LiveData](https://startandroid.ru/ru/uroki/vse-uroki-spiskom/27-course/architecture-components/525-urok-2-livedata.html)

private MutableLiveData<String> liveData = new MutableLiveData<>();

LiveData<String> getData() {

   return liveData;

}

# [Урок 3. LiveData. Дополнительные возможности](https://startandroid.ru/ru/uroki/vse-uroki-spiskom/27-course/architecture-components/526-urok-3-livedata.html)

LiveData<String> liveData = ...;

LiveData<Integer> liveDataInt = Transformations.map(liveData, new Function<String, Integer>() {

   @Override

   public Integer apply(String input) {

       return Integer.parseInt(input);

   }

});

private LiveData<User> getUser(long id) {

   // ...

}

LiveData<Long> liveDataId = ...;

LiveData<User> liveDataUser = Transformations.switchMap(liveDataId, new Function<Long, LiveData<User>>() {

   @Override

   public LiveData<User> apply(Long id) {

       return getUser(id);

   }

});

## **MediatorLiveData**

MutableLiveData<String> liveData1 = new MutableLiveData<>();

MutableLiveData<String> liveData2 = new MutableLiveData<>();

MediatorLiveData<String> mediatorLiveData = new MediatorLiveData<>();

mediatorLiveData.addSource(liveData1, new Observer<String>() {

   @Override

   public void onChanged(@Nullable String s) {

       mediatorLiveData.setValue(s);

   }

});

mediatorLiveData.addSource(liveData2, new Observer<String>() {

   @Override

   public void onChanged(@Nullable String s) {

       mediatorLiveData.setValue(s);

   }

});

# [Урок 4. ViewModel](https://startandroid.ru/ru/uroki/vse-uroki-spiskom/27-course/architecture-components/527-urok-4-viewmodel.html)

Модель жива, пока Activity не закроется окончательно.

 public class MyViewModel extends ViewModel {

   // ...

   MutableLiveData<String> data;

   public LiveData<String> getData() {

       if (data == null) {

           data = new MutableLiveData<>();

           loadData();

       }

       return data;

   }

   private void loadData() {

       dataRepository.loadData(new Callback<String>() {

           @Override

           public void onLoad(String s) {

               data.postValue(s);

           }

       });

   }

}

Когда Activity окончательно закрывается, провайдер удаляет ViewModel, предварительно вызвав его метод [onCleared](https://developer.android.com/reference/android/arch/lifecycle/ViewModel.html" \l "onCleared()" \t "_blank)

Context

Не стоит передавать Activity в модель в качестве Context. Это может привести к утечкам памяти.

Если вам в модели понадобился объект Context, то вы можете наследовать не ViewModel, а [AndroidViewModel](https://developer.android.com/reference/android/arch/lifecycle/AndroidViewModel.html" \t "_blank).

## **Передача данных между фрагментами**

ViewModel может быть использована для передачи данных между фрагментами, которые находятся в одном Activity. В документации есть отличный пример кода:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public class SharedViewModel extends ViewModel {      private final MutableLiveData<Item> selected = new MutableLiveData<Item>();        public void select(Item item) {          selected.setValue(item);      }        public LiveData<Item> getSelected() {          return selected;      }  } |

SharedViewModel - модель с двумя методами: один позволяет поместить данные в LiveData, другой - позволяет получить этот LiveData. Соответственно, если два фрагмента будут иметь доступ к этой модели, то один сможет помещать данные в его LiveData, а другой - подпишется и будет получать эти данные. Таким образом два фрагмента будут обмениваться данными ничего не зная друг о друге.

Чтобы два фрагмента могли работать с одной и той же моделью, они могут использовать общее Activity. Код получения модели в фрагментах выглядит так:

|  |  |
| --- | --- |
| 1 | SharedViewModel model =  ViewModelProviders.of(getActivity()).get(SharedViewModel.class); |

Для обоих фрагментов getActivity вернет одно и то же Activity. Метод ViewModelProviders.of вернет провайдера этого Activity. Далее методом get получаем модель.

## **onSavedInstanceState**

ViewModel - здесь удобно держать все данные, которые нужны вам для формирования экрана. Они будут жить при поворотах экрана, но умрут, когда приложение будет убито системой.

onSavedInstanceState - здесь нужно хранить тот минимум данных, который понадобится вам для восстановления состояния экрана и данных в ViewModel после экстренного закрытия Activity системой. Это может быть поисковый запрос, ID и т.п.

## **RxJava**

У LiveData есть одно большое преимущество - он учитывает состояние Activity. Т.е. он не будет слать данные, если Activity свернуто. И он отпишет от себя Activity, которое закрывается.

А вот Flowable этого не умеет. Если в модели есть Flowable, и Activity подпишется на него, то этот Flowable будет держать Activity, пока оно само явно не отпишется (или пока Flowable не завершится).

# [Урок 5. Room. Основы](https://startandroid.ru/ru/uroki/vse-uroki-spiskom/27-course/architecture-components/529-urok-5-room-osnovy.html)

Room имеет три основных компонента: Entity, Dao и Database. Рассмотрим их на небольшом примере, в котором будем создавать базу данных для хранения данных по сотрудникам (англ. - employee).

При работе с Room нам необходимо будет писать SQL запросы. Если вы не знакомы с ними, то имеет смысл прочесть хотя бы основы.

## **Entity** в SQLite не важен регистр в именах таблиц,

## **Dao**

## **Database**

@Database(entities = {Employee.class}, version = 1)

public abstract class AppDatabase extends RoomDatabase {

   public abstract EmployeeDao employeeDao();

} + синглетон

В параметрах аннотации Database указываем, какие Entity будут использоваться, и версию базы. Для каждого Entity класса из списка entities будет создана таблица.

В Database классе необходимо описать абстрактные методы для получения Dao объектов, которые вам понадобятся.

companion object {

@Volatile private var instance: FactDatabase? = null

fun getInstance(context: Context): FactDatabase =  
 instance ?: *synchronized*(this) **{**   
 instance ?: buildDatabase(context).*also* **{** instance = **it }  
 }**  
 private fun buildDatabase(context: Context): FactDatabase =  
 Room.databaseBuilder(context, FactDatabase::class.*java*, *FACT\_TODO\_DATABASE\_NAME*).build()  
}

}

## **Практика**

AppDatabase db = App.getInstance().getDatabase();

EmployeeDao employeeDao = db.employeeDao();

## **UI поток**

Но эти операции должны выполняться не в UI потоке. Иначе мы получим Exception.

Повторюсь, операции по работе с базой данных - синхронные, и должны выполняться не в UI потоке.

В случае с Query операциями мы можем сделать их асинхронными используя [LiveData](https://startandroid.ru/ru/uroki/vse-uroki-spiskom/27-course/architecture-components/525-urok-2-livedata.html) или [RxJava](https://startandroid.ru/ru/uroki/vse-uroki-spiskom/19-course/rxjava/435-urok-1.html). Об этом еще поговорим в следующих уроках.

В этом случае вы не будете получать Exception при работе в UI потоке. Но вы должны понимать, что это плохая практика, и может добавить ощутимых тормозов вашему приложению.

## **Переход на Room**

Если вы надумали с SQLite мигрировать на Room, то вот пара полезных ссылок по этой теме:

<https://medium.com/google-developers/incrementally-migrate-from-sqlite-to-room-66c2f655b377>

<https://medium.com/@price.yvonne.86/quick-and-easy-migration-to-room-d40dbb142b51>

# [Урок 6. Room. Entity](https://startandroid.ru/ru/uroki/vse-uroki-spiskom/27-course/architecture-components/530-urok-6-room-entity.html)

# В этом уроке более подробно рассмотрим возможности Entity. Как задать имя таблицы. Как задать имя или тип поля. Как создать составной или внешний ключ. Как создать индекс. Как использовать вложенные объекты.

## **Имя таблицы**

# Entity класс используется для создания таблицы. По умолчанию в качестве имени таблицы используется имя этого класса. Но мы можем указать свое имя, используя параметр tableName.

## **Имя поля**

# По умолчанию в качестве имени полей в таблице используются имена полей Entity класса. Но мы можем указать свое имя, используя параметр name в аннотации [ColumnInfo](https://developer.android.com/reference/android/arch/persistence/room/ColumnInfo.html" \t "_blank).

## **Тип поля**

# По умолчанию Room определяет тип данных для поля в таблице по типу данных поля в Entity классе. Но мы можем явно указать свой тип.

   @ColumnInfo(typeAffinity = TEXT)

   public int salary;

## **Модификаторы доступа**

# Но есть возможность использовать private поля. Для этого надо добавить set/get методы

## **Первичный ключ**

# Каждый Entity класс должен содержать хотя бы одно такое поле.

@PrimaryKey(autoGenerate = true)

public long id;

# База сама найдет ближайшее свободное значение и использует его.

# Чтобы создать составной ключ, используйте параметр primaryKeys.

@Entity(primaryKeys = {"key1", "key2"})

public class Item {

   public long key1;

   public long key2;

   // ...

}

## **Внешний ключ**

Внешние ключи позволяют связывать таблицы между собой. Если вы еще не знакомы с ними, то можете почитать о них в инете.

@Entity(foreignKeys = @ForeignKey(entity = Employee.class, parentColumns = "id", childColumns = "employee\_id"))

public class Car {

   @PrimaryKey(autoGenerate = true)

   public long id;

   public String model;

   public int year;

   @ColumnInfo(name = "employee\_id")

   public long employeeId;

}

В поле employee\_id будет храниться id сотрудника, к которому прикреплена эта машина.

Еще один параметр аннотации ForeignKey - это deferred, имеющий по умолчанию значение false. Если задать этому параметру значение true, то внешний ключ станет отложенным.

## **Индекс**

@Entity(indices = {

               @Index("salary"),

               @Index(value = {"first\_name", "last\_name"})

           }

       )

public class Employee {

   @PrimaryKey(autoGenerate = true)

   public long id;

   @ColumnInfo(name = "first\_name")

   public String firstName;

   @ColumnInfo(name = "last\_name")

   public String lastName;

   public int salary;

}

# Индекс дает возможность установить для его полей проверку на уникальность. Это делается параметром unique = true.

# Индекс для одного поля также может быть настроен через параметр index аннотации ColumnInfo

# @ColumnInfo(index = true)

# А как использовать????? Где сортировка ???

## **Вложенные объекты**

public class Address {

   public String city;

   public String street;

   public int number;

}

@Entity()

public class Employee {

   @PrimaryKey(autoGenerate = true)

   public long id;

   public String name;

   public int salary;

   @Embedded

   public Address address;

}

Embedded подскажет Room, что надо просто взять поля из Address и считать их полями таблицы Employee.

Т.е. в базе будет создана таблица Employee с полями id, name, salary, city, street, number.

Мы создаем вложенный объект Address, но Room разберется, и запишет все в таблицу, как плоскую структуру.

Embedded объекты могут включать в себя другие Embedded объекты.

Если у вас получается так, что совпадают имена каких-то полей в основном объекте и в Embedded объекте, то используйте префикс для Embedded объекта.

|  |  |
| --- | --- |
| 1  2 | @Embedded(prefix = "address")  public Address address; |

## **Ignore**

Аннотация [Ignore](https://developer.android.com/reference/android/arch/persistence/room/Ignore.html" \t "_blank) позволяет подсказать Room, что это поле не должно записываться в базу или читаться из нее.

@Entity

public class Employee {

   @PrimaryKey

   public long id;

   public String name;

   public int salary;

   @Ignore

   public Bitmap avatar;

}

# Нам не нужно хранить Bitmap в базе, поэтому добавляем Ignore к этому полю.

# [Урок 7. Room. Insert, Update, Delete, Transaction](https://startandroid.ru/ru/uroki/vse-uroki-spiskom/27-course/architecture-components/531-urok-7-room-insert-update-delete-transaction.html)

## **Insert**

### **Вставка нескольких объектов**

### **Получение id**

# Для этого надо описать метод так, чтобы он возвращал long.

@Insert(onConflict = OnConflictStrategy.REPLACE)

void insert(Employee employee);

# Также есть режим IGNORE. В этом режиме будет оставлена старая запись и операция вставки не будет выполнена.

## **Update**

# Так же, как и с Insert мы можем использовать коллекции и varargs, чтобы обновлять несколько объектов сразу.

## **Delete**

@Delete

void delete(Employee employee);

# Delete ищет в бд запись по ключу.

# Мы можем получить количество удаленных записей. Для этого необходимо описать метод так, чтобы он возвращал int.

## **Транзакции**

# Аннотация [@Transaction](https://developer.android.com/reference/android/arch/persistence/room/Transaction.html) позволяет выполнять несколько методов в рамках одной транзакции.

@Dao

public abstract class EmployeeCarDao {

   @Insert

   public abstract void insertEmployee(Employee employee);

   @Insert

   public abstract void insertCar(Car car);

   @Transaction

   public void insertCarAndEmployee(Car car, Employee employee) {

       insertCar(car);

       insertEmployee(employee);

   }

}

Обратите внимание, что в этом случае Dao - не интерфейс, а абстрактный класс.

# [Урок 8. Room. Query](https://startandroid.ru/ru/uroki/vse-uroki-spiskom/27-course/architecture-components/532-urok-8-room-query.html)

# В этом уроке поговорим подробнее про Query. В каком виде мы можем получать данные: List, массив, Cursor, LiveData. Как передавать параметры. Как получать только некоторые поля. Как с помощью Query выполнять update и delete запросы в Room.

## **List, массив, Cursor**

## **LiveData**

# Получаем LiveData и подписываемся на него.

# Использование LiveData имеет огромное преимущество перед использование списка или массива. Подписавшись на LiveData, вы будете получать свежие данные при их изменении в базе. Т.е. при добавлении новых, удалении старых или обновлении текущих данных в таблице employee, Room снова выполнит ваш Query запрос, и вы получите в onChanged методе актуальные данные с учетом последних изменений. Вам больше не надо самим запрашивать эти данные каждый раз. И все это будет приходить вам в UI поток.

## **Передача параметров**

Например, запрос данных по id

|  |  |
| --- | --- |
| 1  2 | @Query("SELECT \* FROM employee WHERE id = :employeeId")  Employee getById(long employeeId); |

Перед параметром employeeId в запросе должно стоять двоеточие. Room возьмет значение этого параметра из метода и подставит его в запрос.

Поиск сотрудников по списку id.

@Query("SELECT \* FROM employee WHERE id IN (:idList)")

List<Employee> getByIdList(List<Long> idList);

## **Subsets**

# Часто при запросе данных нам нужно получить из таблицы не все поля, а только некоторые. Такие запросы быстрее и легче, чем тянуть все поля.

# В этом случае мы можем использовать отдельный объект.

# Обратите внимание, что он не Entity. Это обычный класс. С помощью ColumnInfo мы настраиваем имена полей, чтобы они совпадали с полями таблицы.

public class Name {

   @ColumnInfo(name = "first\_name")

   public String firstName;

   @ColumnInfo(name = "last\_name")

   public String lastName;

}

# Используем этот класс в методе запроса:

@Query("SELECT first\_name, last\_name FROM employee")

List<Name> getNames();

# Вы также можете в этих не Entity классах использовать вложенные классы с аннотацией @Embedded. Подробно об этой аннотации мы говорили в [Уроке 6](https://startandroid.ru/ru/uroki/vse-uroki-spiskom/27-course/architecture-components/530-urok-6-room-entity.html).

## **insert, update и delete запросы**

# Обновление зарплат у сотрудников по списку id.

@Query("UPDATE employee SET salary = :newSalary WHERE id IN (:idList)")

int updateSalaryByIdList(List<Long> idList, int newSalary);

# Вызов метода будет выглядеть так:

# int updatedCount = db.employeeDao().updateSalaryByIdList(Arrays.asList(1L, 3L, 4L), 10000);

# Удаление сотрудников по списку id

@Query("DELETE from employee WHERE id IN (:idList)")

int deleteByIdList(List<Long> idList);

# [Урок 9. Room. RxJava](https://startandroid.ru/ru/uroki/vse-uroki-spiskom/27-course/architecture-components/533-urok-9-room-rxjava.html)

# В этом уроке рассмотрим возможность совместного использования RxJava и Room. Как получать данные в Flowable, Single и Maybe.

## **Flowable**

@Query("SELECT \* FROM employee")

Flowable<List<Employee>> getAll();

## **Single**

@Query("SELECT \* FROM employee WHERE id = :id")

Single<Employee> getById(long id);

## **Maybe**

@Query("SELECT \* FROM employee WHERE id = :id")

Maybe<Employee> getById(long id);

## **В каком случае что лучше использовать?**

Flowable подходит, если вы запрашиваете данные и далее планируете автоматически получать их обновления.

Single и Maybe подходят для однократного получения данных. Разница между ними в том, что Single логичнее использовать, если запись должна быть в базе. Если ее нет, вам придет ошибка. А Maybe допускает, что записи может и не быть.

# [Урок 10. Room. Запрос из нескольких таблиц. Relation](https://startandroid.ru/ru/uroki/vse-uroki-spiskom/27-course/architecture-components/534-urok-10-room-zapros-iz-neskolkih-tablic-relation.html)

Для примера будем использовать две таблицы: сотрудники и отделы. Каждый сотрудник прикреплен к какому-либо отделу.

Entity объект для отделов:

@Entity

public class Department {

   @PrimaryKey

   public int id;

   public String name;

}

Entity объект для сотрудников:

@Entity

public class Employee {

   @PrimaryKey

   public long id;

   public String name;

   public int salary;

   @ColumnInfo(name = "department\_id")

   public int departmentId;

}

# В поле departmentId хранится id отдела, к которому прикреплен сотрудник.

# Мы хотим получить список работников, в котором будет следующая информация: имя работника, его зарплата, наименование его отдела. Для этого нам надо будет написать запрос, который вытащит данные из двух таблиц.

Dao

public interface EmployeeDao {

   @Query("SELECT employee.name, employee.salary, department.name AS department\_name " +

       "FROM employee, department " +

       "WHERE department.id == employee.department\_id")

   public List<EmployeeDepartment> getEmployeeWithDepartment();

   // ...

   }

Т.к. поле name есть в обоих таблицах, то для отдела переименовываем его в department\_name

Обратите внимание на тип объектов, который мы будем получать от этого метода. Это EmployeeDepartment. Нам нужно создать этот объект, и указать в нем все поля, которые мы ожидаем получить от запроса.

public class EmployeeDepartment {

   public String name;

   public int salary;

   @ColumnInfo(name = "department\_name")

   public String departmentName;

}

Это не Entity объект, а обычный класс. Поля этого класса должны совпадать с полями результата, который вернет запрос. Room конвертирует результаты запроса в список этих объектов, и мы получим то, что хотели.

## **Relation**

# Аннотация [Relation](https://developer.android.com/reference/android/arch/persistence/room/Relation.html" \t "_blank) также позволяет делать запросы из нескольких таблиц, но структура результата будет немного другой. И нам самим не придется писать сложные запросы. Room все сделает за нас.

# Давайте представим, что нам надо получить список отделов. И к каждому отделу должен прилагаться список сотрудников.

Структура для этих данных будет выглядеть так:

public class DepartmentWithEmployees {

   public int id;

   public String name;

   @Relation(parentColumn = "id", entityColumn = "department\_id")

   public List<Employee> employees;

}

Это не Entity, а обычный класс. В полях id и name будут данные отдела.

В employees будет список сотрудников этого отдела. Для этого мы помечаем список аннотацией Relation, и Room сам заполнит его для нас. Давайте разбираться, как именно Room поймет, что он должен поместить в этот список. Откуда он будет брать данные и по какому условию?

Тип данных списка - это Employee. Это Entity объект, для него в базе данных создана таблица. Из этой таблицы Room и будет читать данные по сотрудникам. В параметрах parentColumn и entityColumn указываем названия полей, которые участвуют в условии выборки данных. В результате, Room будет искать сотрудников, у которых entityColumn (т.е. department\_id) равен parentColumn (т.е. id) отдела. Все найденные сотрудники окажутся в employees.

По требованиям Room, тип employees должен быть List или Set.

Осталось описать метод в Dao:

@Dao

public interface DepartmentDao {

   @Query("SELECT id, name from department")

   List<DepartmentWithEmployees> getDepartmentsWithEmployees();

   // ...

}

Это простой запрос, который вытащит необходимые данные по отделу. А запрос по сотрудникам для каждого отдела сделает за нас Room.

В классе DepartmentWithEmployees мы используем поля id и name для данных по отделу. Но класс Department имеет точно такую же структуру - id и name. Поэтому мы в DepartmentWithEmployees можем заменить эти поля на одно поле с типом Department и аннотацией Embedded:

public class DepartmentWithEmployees {

   @Embedded

   public Department department;

   @Relation(parentColumn = "id", entityColumn = "department\_id")

   public List<Employee> employees;

}

Предположим, что нам нужны не все данные по сотрудникам, а только некоторые поля. Например, name и salary. Создаем под них класс:

public class EmployeeNameAndSalary {

   public String name;

   public int salary;

}

И используем его, как тип в Relation-списке

public class DepartmentWithEmployees {

   public int id;

   public String name;

   @Relation(parentColumn = "id", entityColumn = "department\_id", entity = Employee.class)

   public List<EmployeeNameAndSalary> employees;

}

А чтобы Room знал, откуда брать данные по сотрудникам, указываем Entity класс Employee в параметре entity.

Relation может быть вложенным. Т.е. в нашем примере класс EmployeeNameAndSalary также может содержать в себе Relation, который будет для каждого сотрудника собирать, например, список техники, записанной на него.

Relation не может быть использован в Entity классах, только в обычных. Relation поле не может задаваться через конструктор. Оно должно быть public или иметь public set-метод.

## **Relation + Transaction**

При использовании Relation, Room выполняет несколько запросов, чтобы собрать все данные. Имеет смысл выполнять все эти запросы в одной транзакции, чтобы получить корректные данные. Для этого можно использовать аннотацию Transaction

@Transaction

@Query("SELECT id, name from department")

List<DepartmentWithEmployees> getDepartmentsWithEmployees();

# [Урок 11. Room. Type converter](https://startandroid.ru/ru/uroki/vse-uroki-spiskom/27-course/architecture-components/539-urok-11-room-type-converter.html)

Иногда ваши Entity объекты могут содержать поля, которые не являются примитивами, и не могут быть сохранены в БД.

В качестве примера рассмотрим класс работника. У него вполне может быть поле, в котором мы хотим перечислить его хобби. Используем для этого поле hobbies с типом List<String>

Ок, давайте создадим конвертер. Он должен уметь конвертировать List<String> в какой-нибудь простой тип, который может быть сохранен в базу, например, String. Также конвертер должен уметь конвертировать в обратную сторону, т.е. из String в List<String>, чтобы Room мог прочесть данные из базы в поле Entity объекта.

public class HobbiesConverter {

   @TypeConverter

   public String fromHobbies(List<String> hobbies) {

       return hobbies.stream().collect(Collectors.joining(","));

   }

   @TypeConverter

   public List<String> toHobbies(String data) {

       return Arrays.asList(data.split(","));

   }

}

Первый метод преобразует List<String> в String. Второй - наоборот. Оба метода помечаем аннотацией TypeConverter.

Осталось указать этот конвертер для поля hobbies. Это делается аннотацией TypeConverters с указанием класса конвертера.

@Entity()

public class Employee {

   @PrimaryKey

   public long id;

   public String name;

   public int salary;

   @TypeConverters({HobbiesConverter.class})

   public List<String> hobbies;

}

Ну и самое глобальное решение - прописать конвертер для Database

|  |  |
| --- | --- |
| 1  2  3  4  5 | @Database(entities = {Employee.class}, version = 1)  @TypeConverters({DateConverter.class})  public abstract class AppDatabase extends RoomDatabase {     public abstract EmployeeDao employeeDao();  } |

# [Урок 12. Room. Миграция версий базы данных](https://startandroid.ru/ru/uroki/vse-uroki-spiskom/27-course/architecture-components/540-urok-12-migracija-versij-bazy-dannyh.html)

ибо, как вариант, мы можем использовать fallbackToDestructiveMigration в билдере создания базы:

|  |  |
| --- | --- |
| 1  2  3 | database = Room.databaseBuilder(this, AppDatabase.class, "database")         .fallbackToDestructiveMigration()         .build(); |

В этом случае миграцию можно не настраивать. Но если при запуске приложения Room увидит, что необходима миграция, то он просто пересоздаст базу в соответствии с новой структурой Entity классов и все данные пропадут. Так себе вариант, особенно с точки зрения пользователя. Он потеряет все свои данные.

# [Урок 13. Room. Тестирование](https://startandroid.ru/ru/uroki/vse-uroki-spiskom/27-course/architecture-components/541-urok-13-room-testirovanie.html)

## **Dao**

## **Миграция**

## **Схемы**

# [Урок 18. Android Data Binding. Основы](https://startandroid.ru/ru/courses/architecture-components/27-course/architecture-components/551-urok-18-data-binding-osnovy.html)

Сразу хочу заметить, что если мы теперь в коде будем изменять объект Employee, то данные на экране меняться не будут. Они считались один раз и далее не отслеживаются (при такой реализации).

Чтобы экран получил новые данные, надо снова передать биндингу измененный объект Employee:

|  |  |
| --- | --- |
| 1 | binding.setEmployee(employee); |

Или можно вызывать метод invalidateAll:

|  |  |
| --- | --- |
| 1 | binding.invalidateAll(); |

# [Урок 19. Android Data Binding. Код в layout. Доступ к View](https://startandroid.ru/ru/courses/architecture-components/27-course/architecture-components/552-urok-19-android-data-binding-vozmozhnosti.html)

 android:text="@{String.valueOf(employee.salary)}"

 android:text='@{employee.name + ", " + employee.address}'

<import type="android.view.View"/>

<import type="android.text.TextUtils"/>

android:text="@{employee.address}"

android:visibility="@{TextUtils.isEmpty(employee.address) ? View.GONE : View.VISIBLE}"

Использование resources значения: strings, dimens и пр.

|  |  |
| --- | --- |
| 1  2  3  4 | <TextView     android:layout\_width="wrap\_content"     android:layout\_height="wrap\_content"     android:text="@{TextUtils.isEmpty(employee.address) ? @string/empty\_address : employee.address}"/> |

Если адрес пустой, то показываем заглушку из strings.

|  |
| --- |
| android:text='@{employee.name + "(" + department.name + ")"}' /> |

В одном TextView показываем данные из двух переменных.

 android:text='@{employee.name + (department == null ? "" : " (" + department.name + ")

[") }'](mailto:\") }'public List<String> hobbies;  android:text=\"@{employee.hobbies[0)

[public List<String> hobbies;  android:text="@{employee.hobbies[0](mailto:\") }'public List<String> hobbies;  android:text=\"@{employee.hobbies[0)]}"

Если нам необходимо использовать список, как отдельную переменную в layout, то variable будет выглядеть так:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1  2  3 | <variable     name="hobbies"     type="java.util.List&lt;String&gt;" />  Map коллекции описываются аналогично:   |  |  | | --- | --- | | 1  2  3  4  5 | <import type="java.util.Map"/>    <variable     name="map"     type="Map&lt;String, String&gt;"/> |   Получение значения по ключу:   |  |  | | --- | --- | | 1  2  3  4 | <TextView     android:layout\_width="wrap\_content"     android:layout\_height="wrap\_content"     android:text="@{map[`key`]}" /> | |

# [Урок 20. Android Data Binding. Обработка событий](https://startandroid.ru/ru/courses/architecture-components/27-course/architecture-components/553-urok-20-android-data-binding-obrabotka-sobytiy.html)

Создаем свой класс обработчик:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public class MyHandler  {       public void onDelete(View view) {         // ...     }    } |

Он не обязательно должен наследовать OnClickListener. Но его метод должен быть public и иметь те же параметры, что и метод OnClickListener.onClick(View view), т.е. должен быть один параметр типа View. Имя метода может быть любым.

Прописываем этот обработчик, как variable в layout.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | <data>     <variable         name="handler"         type="ru.startandroid.application.MyHandler" />       ...    </data> |

В onClick кнопки [ссылаемся на его метод](https://vertex-academy.com/tutorials/ru/java-8-ssylki-na-metody/) onDelete:

|  |  |
| --- | --- |
| 1  2  3  4  5 | <Button     android:layout\_width="wrap\_content"     android:layout\_height="wrap\_content"     android:text="@string/delete"     android:onClick="@{handler::onDelete}"/> |

Осталось создать объект MyHandler и передать его в биндинг:

|  |  |
| --- | --- |
| 1  2 | MyHandler myHandler = new MyHandler();  binding.setHandler(myHandler); |

 public void onDelete(Employee employee) {

android:onClick="@{(view) -> handler.onDelete(employee)}"

 android:onClick="@{() -> handler.onDelete(employee)}"

Таким образом биндинг поймет, что его View нам не нужно, и не будет его передавать. Имейте ввиду, что если в стандартном обработчике несколько параметров, то вы можете указать либо все параметры, либо ни одного.

В обработчике создаем метод,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public class MyHandler  {       public void onEnabled(Employee employee, boolean enabled) {         // ...     }    } |

Будем получать объект Employee и состояние чекбокса.

В onCheckedChanged пишем вызов метода нашего обработчика.

|  |  |
| --- | --- |
| 1  2  3  4  5 | <CheckBox     android:layout\_width="wrap\_content"     android:layout\_height="wrap\_content"     android:text="@string/enabled"     android:onCheckedChanged="@{(view, checked) -> handler.onEnabled(employee, checked)}" /> |

В лямбде указываем параметры, которые пришли бы нам в стандартном обработчике OnCheckedChangeListener.onCheckedChanged(CompoundButton compoundButton, boolean checked).

Параметр view нам не понадобится, а вот checked передаем в метод вместе с employee.

Теперь по нажатию на чекбокс, биндинг будет вызывать метод onEnabled и передавать туда Employee объект и состояние чекбокса.

 android:onCheckedChanged="@{(view, checked) -> checked ? handler.onEnabled(employee) : handler.onDisabled(employee)}"

android:onCheckedChanged="@{(view, checked) -> checked ? handler.onEnabled(employee) : void}"

В биндиге по умолчанию есть переменная context, которую вы всегда можете использовать, если есть необходимость.

|  |  |
| --- | --- |
| 1  2  3  4  5 | <CheckBox     android:layout\_width="wrap\_content"     android:layout\_height="wrap\_content"     android:text="@string/enabled"     android:onCheckedChanged="@{(view, checked) -> handler.onEnabled(employee, checked, context)}"/> |

Значение переменной context получено вызовом метода getContext у корневого View вашего layout.

В этих примерах я создавал отдельный объект обработчика, но разумеется вы можете создавать интерфейсы. прописывать их в качестве variable в layout и использовать хоть само Activity в качестве реализации.

# [Урок 21. Android Data Binding. Observable поля. Двусторонний биндинг.](https://startandroid.ru/ru/courses/architecture-components/27-course/architecture-components/554-urok-21-android-data-binding-observable.html)

## **Observable поля**

## **BaseObservable**

public class Employee extends BaseObservable {

@Bindable

   public String getName() {

       return name;

   }

   public void setName(String name) {

       this.name = name;

       notifyPropertyChanged(BR.name);

   }

   @Bindable

   public int getSalary() {

       return salary;

   }

   public void setSalary(int salary) {

       this.salary = salary;

       notifyPropertyChanged(BR.salary);

   }

Поле id я оставил обычным. А поля name и salary будут отслеживаться биндингом. Для этого надо пометить get-методы аннотацией @Bindable, а в set-методах вызывать notifyPropertyChanged метод, который и будет уведомлять биндинг об изменениях значения поля.

В layout все будет как обычно.

## **Двусторонний биндинг**

Кстати, если после передачи в биндинг вы нигде не храните у себя объект employee, то вы всегда можете получить его обратно методом binding.getEmployee().

Рассмотрим на примере пары полей в Employee:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public class Employee {       public String name = "";     public boolean enabled = true;    } |

Поле name и статус enabled. Настроим биндинг этих полей в EditText и CheckBox.

При этом сделаем так, чтобы биндинг работал в обе стороны. Для этого надо в строке биндинга добавить символ = между @ и {...}

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | <EditText     android:layout\_width="wrap\_content"     android:layout\_height="wrap\_content"     android:text="@={employee.name}"/>    <CheckBox     android:text="enabled"     android:layout\_width="wrap\_content"     android:layout\_height="wrap\_content"     android:checked="@={employee.enabled}"/> |

# [Урок 22. Android Data Binding. Adapter. Conversion.](https://startandroid.ru/ru/courses/architecture-components/27-course/architecture-components/555-urok-22-android-data-binding-include-viewstub-holders.html)

В этом уроке разберем, как можно расширить возможности биндинга с помощью Binding Adapter и Binding Conversion.

## **Binding Adapter**

 <ImageView

           android:id="@+id/image"

           android:layout\_width="200dp"

           android:layout\_height="200dp"

           android:scaleType="centerInside"

           app:url="@{employee.avatarUrl}"/>

@BindingAdapter({"app:url"})

public static void loadImage(ImageView view, String url) {

   Picasso.get().load(url).into(view);

}

<ImageView

   android:id="@+id/image"

   android:layout\_width="200dp"

   android:layout\_height="200dp"

   android:scaleType="centerInside"

   app:url="@{employee.avatarUrl}"

   app:errorImage="@{@drawable/ic\_error}"/>

@BindingAdapter({"app:url", "app:errorImage"})

public static void loadImage(ImageView view, String url, Drawable errorImage) {

   Picasso.get().load(url).error(errorImage).into(view);

}

## **Conversion**

public class Employee {

   public List<Hobby> hobbies;

}

<TextView

   android:layout\_width="wrap\_content"

   android:layout\_height="wrap\_content"

   android:text="@{employee.hobbies}"/>

@BindingConversion

public static String convertHobbiesToString(List<Hobby> hobbies) {

   StringBuilder sb = new StringBuilder();

   for (Hobby hobby: hobbies) {

       if (sb.length() > 0) sb.append(", ");

       sb.append(hobby.name);

   }

   return sb.toString();

}

# [Урок 23. Android Data Binding. Использование с include, ViewStub и RecyclerView.](https://startandroid.ru/ru/courses/architecture-components/27-course/architecture-components/556-urok-23-android-data-binding.html)

## **include**

С помощью include можно использовать layout файл внутри другого layout файла. При этом мы можем использовать биндинг без каких-либо дополнительных усилий.

<include layout="@layout/employee\_details"

           app:employee="@{employee}"/>

<?xml version="1.0" encoding="utf-8"?>

<layout xmlns:android="http://schemas.android.com/apk/res/android">

   <data>

       <variable

           name="employee"

           type="ru.startandroid.application.data.Employee" />

   </data>

Он получит переменную employee из внешнего layout и сможет использовать ее значения.

## **ViewStub**

iewStub - это механизм для ленивого создания View. Т.е. если у вас в layout есть View, которые отображаются в исключительных случаях, то нет смысла создавать их при каждом отображении экрана. В этом случае лучше использовать ViewStub, который сможет создать эти View при необходимости.

<ViewStub

           android:id="@+id/employeeAddressStub"

           android:layout\_width="wrap\_content"

           android:layout\_height="wrap\_content"

           android:layout="@layout/employee\_address"/>

<?xml version="1.0" encoding="utf-8"?>

<layout xmlns:android="http://schemas.android.com/apk/res/android">

   <data>

       <variable

           name="employee"

           type="ru.startandroid.applicationtest003.data.Employee" />

   </data>

binding.employeeAddressStub.setOnInflateListener(new ViewStub.OnInflateListener() {

   @Override

   public void onInflate(ViewStub viewStub, View view) {

       EmployeeAddressBinding binding = DataBindingUtil.bind(view);

       binding.setEmployee(employee);

   }

});

## **RecyclerView**