Version 0.2

Android's Architecture Components

Mark L. Murphy

CommonsWare

Android's Architecture Components

by Mark L. Murphy

Android's Architecture Components
by Mark L. Murphy

Printing History:
August 2017: Version 0.2

Table of Contents

Headings formatted in ***bold-italic*** have changed since the last version.

Preface

Thanks!

Thanks for your interest in Android app development, the world's most popular operating system! And, thanks for your interest in the Android Architecture Components, released by Google in 2017 to help address common "big-ticket" problems in Android app development.

And, most of all, thanks for your interest in this book! I sincerely hope you find it useful!

(OTOH, if you find it completely useless… um, don't tell anyone, OK?)

How the Book Is Structured

We start off with a look at Room, an object/relational mapping (ORM) library. This makes it a bit easier to integrate your app with SQLite, the built-in relational database engine in Android.

We then move into the lifecycle components. These components help you deal with objects that have lifecycles, particularly activities and services. The `LiveData` class in particular gives you a lightweight "reactive" way of consuming data while still honoring things like configuration changes and the typical activity/fragment destroy-and-recreate cycle. We will also peek at `ViewModel`, the Architecture Components' way of helping you maintain state across configuration changes.

In future editions of this book, we will then explore more advanced topics related to the Architecture Components, such as how these components tie into things like data binding, RxJava/RxAndroid, `ContentProvider`, and more.

Prerequisites

This book is targeted at:

- People who have read the core chapters of the companion volume, *The Busy Coder's Guide to Android Development*, or
- Intermediate Android app developers — those with some experience but not necessarily "experts" in the field

About the Updates

This book will be updated a few times per year, to reflect new advances with the Architecture Components.

If you obtained this book through the Warescription, you will be able to download updates as they become available, for the duration of your subscription period.

If you obtained this book through other channels… um, well, it's still a really nice book!

Each release has notations to show what is new or changed compared with the immediately preceding release:

- The Table of Contents shows sections with changes in bold-italic font
- Those sections have changebars on the right to denote specific paragraphs that are new or modified

And, there is the "What's New" section, just below this paragraph.

What's New in Version 0.2?

This update:

- Adds a new chapter on M:N relations in Room, showing how to set up and use join entities
- Adds a new chapter on transformations with `LiveData`, to map data from one type to another as part of observing the stream of events
- Fixes various errata and makes other improvements

Warescription

If you purchased the Warescription, read on! If you obtained this book from other channels, feel free to jump ahead.

The Warescription entitles you, for the duration of your subscription, to digital editions of this book and its updates, in PDF, EPUB, and Kindle (MOBI/KF8) formats. You also have access to a version of the book as its own Android APK file, complete with high-speed full-text searching. You also have access to other titles that CommonsWare publishes during that subscription period, such as the aforementioned *The Busy Coder's Guide to Android Development*.

Each subscriber gets personalized editions of all editions of each title. That way, your books are never out of date for long, and you can take advantage of new material as it is made available.

However, you can only download the books while you have an active Warescription. There is a grace period after your Warescription ends: you can still download the book until the next book update comes out after your Warescription ends. After that, you can no longer download the book. Hence, **please download your updates as they come out**. You can find out when new releases of this book are available via:

1. The CommonsBlog
2. The CommonsWare Twitter feed
3. The Warescription newsletter, which you can subscribe to off of your Warescription page
4. Just check back on the Warescription site every month or two

Subscribers also have access to other benefits, including:

- "Office hours" — online chats to help you get answers to your Android application development questions. You will find a calendar for these on your Warescription page.
- A Stack Overflow "bump" service, to get additional attention for a question that you have posted there that does not have an adequate answer.
- A discussion board for asking arbitrary questions about Android app development

Book Bug Bounty

Find a problem in the book? Let CommonsWare know!

Be the first to report a unique concrete problem in the current digital edition, and CommonsWare will extend your Warescription by six months as a bounty for helping CommonsWare deliver a better product.

By "concrete" problem, we mean things like:

1. Typographical errors
2. Sample applications that do not work as advertised, in the environment described in the book
3. Factual errors that cannot be open to interpretation

By "unique", we mean ones not yet reported. Be sure to check the book's errata page, though, to see if your issue has already been reported. One coupon is given per email containing valid bug reports.

We appreciate hearing about "softer" issues as well, such as:

1. Places where you think we are in error, but where we feel our interpretation is reasonable
2. Places where you think we could add sample applications, or expand upon the existing material
3. Samples that do not work due to "shifting sands" of the underlying environment (e.g., changed APIs with new releases of an SDK)

However, those "softer" issues do not qualify for the formal bounty program.

Questions about the bug bounty, or problems you wish to report for bounty consideration, should be sent to bounty@commonsware.com.

Source Code and Its License

The source code samples shown in this book are available for download from the book's GitHub repository. All of the Android projects are licensed under the Apache 2.0 License, in case you have the desire to reuse any of it.

Copying source code directly from the book, in the PDF editions, works best with Adobe Reader, though it may also work with other PDF viewers. Some PDF viewers, for reasons that remain unclear, foul up copying the source code to the clipboard when it is selected.

Creative Commons and the Four-to-Free (42F) Guarantee

Each CommonsWare book edition will be available for use under the Creative Commons Attribution-Noncommercial-ShareAlike 3.0 license as of the fourth anniversary of its publication date, or when 4,000 copies of the edition have been sold, whichever comes first. That means that, once four years have elapsed (perhaps sooner!), you can use this prose for non-commercial purposes. That is our Four-to-Free Guarantee to our readers and the broader community. For the purposes of this guarantee, new Warescriptions and renewals will be counted as sales of this edition, starting from the time the edition is published.

This edition of this book will be available under the aforementioned Creative Commons license on *1 August 2021*. Of course, watch the CommonsWare Web site, as this edition might be relicensed sooner based on sales.

For more details on the Creative Commons Attribution-Noncommercial-ShareAlike 3.0 license, visit the Creative Commons Web site

Note that future editions of this book will become free on later dates, each four years from the publication of that edition or based on sales of that specific edition. Releasing one edition under the Creative Commons license does not automatically release *all* editions under that license.

Acknowledgments

Room

Room Basics

First, let's spend some time working with Room.

Google describes Room as providing "an abstraction layer over SQLite to allow fluent database access while harnessing the full power of SQLite."

In other words, Room aims to make your use of SQLite easier, through a lightweight annotation-based implementation of an object-relational mapping (ORM) engine.

NOTE: The material in this chapter — and in all the chapters of this book edition — is based on the `alpha3` release of Room and the rest of the Android Architecture Components. Since this is a preview release, there may be changes in newer versions that affect you.

Wrenching Relations Into Objects

If you have ever worked with a relational database — like SQLite — from an object-oriented language — like Java — undoubtedly you have encountered the "object-relational impedance mismatch". That is a very fancy way of saying "gosh, it's a pain getting stuff into and out of the database".

In object-oriented programming, we are used to objects holding references to other objects, forming some sort of object graph. However, traditional SQL-style relational databases work off of tables of primitive data, using foreign keys and join tables to express relationships. Figuring out how to get our Java classes to map to relational tables is aggravating, and it usually results in a lot of boilerplate code.

Traditional Android development uses `SQLiteDatabase` for interacting with SQLite. That, in turn, uses `Cursor` objects to represent the results of queries and

`ContentValues` objects to represent data to be inserted or updated. While `Cursor` and `ContentValues` are objects, they are fairly generic, much in the way that a `HashMap` or `ArrayList` is generic. In particular, neither `Cursor` nor `ContentValues` has any of our business logic. We have to somehow either wrap that around those objects or convert between those objects and some of ours.

That latter approach is what object-relational mapping engines, or ORMs, take. A typical ORM works off of Java code and either generates a suitable database structure or works with you to identify how the Java classes should map to some existing table structure (e.g., a legacy one that you are stuck with). The ORM usually generates some code for you, and supplies a library, which in combination hide much of the database details from you.

The quintessential Java ORM is Hibernate. However, Hibernate was developed with server-side Java in mind and is not well-suited for slim platfoms like Android devices. However, a vast roster of Android ORMs have been created over the years to try to fill that gap. Some of the more popular ones have been:

- DBFlow
- greenDAO
- OrmLite
- Sugar ORM

Room also helps with the object-relational impedance mismatch. It is not as deep of an ORM as some of the others, as you will be dealing with SQL a fair bit. However, Room has one huge advantage: it is from Google, and therefore it will be deemed “official” in the eyes of many developers and middle managers.

While this book is focused on the Architecture Components — and Room is part of those — you may wish to explore other ORMs if you are interested in using Java objects but saving the data in SQLite. Room is likely to become popular, but it is far from the only option.

Room Requirements

To use Room, you need two dependencies in your module’s `build.gradle` file:

1. The runtime library version, using the standard `compile` directive
2. An annotation processor, using the `annotationProcessor` directive

```
compile "android.arch.persistence.room:runtime:1.0.0-alpha8"
annotationProcessor "android.arch.persistence.room:compiler:1.0.0-alpha8"
```

(from Trips/RoomBasics/app/build.gradle)

Note that `alpha3` of Room has a `minSdkVersion` requirement of API Level 15 or higher. If you attempt to build with a lower `minSdkVersion`, you will get a build error. If you try to override Room's `minSdkVersion` using manifest merger elements, while the project will build, expect Room to crash horribly.

Room Furnishings

Roughly speaking, your use of Room is divided into three sets of classes:

1. Entities, which are POJOs that model the data you are transferring into and out of the database
2. The data access object (DAO), that provides the description of the Java API that you want for working with certain entities
3. The database, which ties together all of the entities and DAOs for a single SQLite database

If you have used Square's Retrofit, some of this will seem familiar:

- The DAO is roughly analogous to your Retrofit `interface` on which you declare your Web service API
- Your entities are the POJOs that you are expecting Gson (or whatever) to create based on the Web service response

In this chapter, we will look at the `Trips/RoomBasics` sample project. This app is the first of a linked series of apps that we will examine in this book, as we build a travel itinerary manager. It will track your upcoming trips in a database and allow you to add, edit, and remove trips. Right now, though, we are settling for being able to see some *very* rudimentary trips get into and out of a database.

Entities

In many ORM systems, the entity (or that system's equivalent) is a POJO that you happen to want to store in the database. It usually represents some part of your overall domain model, so a payroll system might have entities representing departments, employees, and paychecks.

With Room, a better description of entities is that they are POJOs representing:

- the data that you want to store into the database, and
- a typical unit of a result set that you are trying to retrieve from the database

That difference may sound academic. It starts to come into play a bit more when we start thinking about relations.

However, it also more closely matches the way Retrofit maps to Web services. With Retrofit, we are not describing the contents of the Web service's database. Rather, we are describing how we want to work with defined Web service endpoints. Those endpoints have a particular set of content that we can work with, courtesy of whoever developed the Web service. We are simply mapping those to methods and POJOs, both for input and output. Room is somewhere in between a Retrofit-style "we just take what the Web service gives us" approach and a full ORM-style "we control everything about the database" approach.

Tactically, an entity is a Java class marked with the `@Entity` annotation. For example, here is a `Trip` class that serves as a Room entity:

```
package com.commonsware.android.room;

import android.arch.persistence.room.Entity;
import android.arch.persistence.room.Ignore;
import android.arch.persistence.room.PrimaryKey;
import java.util.UUID;

@Entity(tableName = "trips")
class Trip {
  @PrimaryKey
  public final String id;

  public final String title;
  public final int duration;

  @Ignore
  Trip(String title, int duration) {
    this(UUID.randomUUID().toString(), title, duration);
  }

  Trip(String id, String title, int duration) {
    this.id=id;
    this.title=title;
    this.duration=duration;
  }
  @Override
  public String toString() {
    return(title);
  }
}
```

(from Trips/RoomBasics/app/src/main/java/com/commonsware/android/room/Trip.java)

There is no particular superclass required for entities, and the expectation is that often they will be simple POJOs, as we see here.

Sometimes, your fields will be marked with annotations describing their roles. In this example, the `id` field has the `@PrimaryKey` annotation, telling Room that this is the unique identifier for this entity. Room will use that to know how to update and delete `Trip` objects by their primary key values.

Similarly, sometimes your methods will be marked with annotations. In this case, `Trip` has two constructors: one that generates the `id` from a `UUID`, and one that takes the `id` as a constructor parameter. Room needs to know which constructor(s) are eligible for its use; you mark the other constructors with the `@Ignore` annotation.

For Room to work with a field, it needs to be `public` or have JavaBean-style getter and setter methods, so Room can access them. If the fields are `final`, as they are on `Trip`, Room will try to find a constructor to use to populate the fields, as `final` fields will lack setters.

We will explore entities in greater detail in an upcoming chapter.

DAO

"Data access object" (or DAO for short) is a fancy way of saying "the API into the data". The idea is that you have a DAO that provides methods for the database operations that you need: queries, inserts, updates, deletes, whatever.

In Room, the DAO is identified by the `@Dao` annotation, applied to either an `abstract` class or an `interface`. The actual concrete implementation will be code-generated for you by the Room annotation processor.

The primary role of the `@Dao`-annotated `abstract` class or `interface` is to have one or more methods, with their own Room annotations, identifying what you want to do with the database and your entities. This serves the same role as the methods annotated `@GET` or `@POST` in Retrofit.

The sample app has a `TripStore` that is our DAO:

```
package com.commonsware.android.room;

import android.arch.persistence.room.Dao;
import android.arch.persistence.room.Delete;
import android.arch.persistence.room.Insert;
import android.arch.persistence.room.OnConflictStrategy;
import android.arch.persistence.room.Query;
import android.arch.persistence.room.Update;
import java.util.List;

@Dao
interface TripStore {
  @Query("SELECT * FROM trips ORDER BY title")
  List<Trip> selectAll();

  @Query("SELECT * FROM trips WHERE id=:id")
  Trip findById(String id);

  @Insert
  void insert(Trip... trips);

  @Update
  void update(Trip... trips);

  @Delete
  void delete(Trip... trips);
}
```

(from Trips/RoomBasics/app/src/main/java/com/commonsware/android/room/TripStore.java)

Besides the `@Dao` annotation on the `TripStore` interface, we have five methods, each with their own annotations. Your four main annotations for these methods are `@Query`, `@Insert`, `@Update`, and `@Delete`, which map to the corresponding database operations.

Two `TripStore` methods — `selectAll()` and `findById()` — have the `@Query` annotation. Principally, `@Query` will be used for SQL `SELECT` statements, where you put the actual SQL in the annotation itself. To a large extent, any valid SQLite query can be used here. However, instead of using `?` as placeholders for arguments, as we would in traditional SQLite, you use `:`-prefixed method parameter names. So, in findById(), we have a String parameter named id, so we can use :id in the SQL statement wherever we might have used ? to indicate the value to bind in.

The remaining three methods use the @Insert, @Update, and @Delete annotations, mapped to methods of the same name. Here, the methods take a varargs of Trip, meaning that we can insert, update, or delete as many Trip objects as we want (passing in zero Trip objects works, though that would be rather odd).

If you want custom code on your DAO, beyond the code-generated implementations of your Room-annotated methods, use an abstract class and mark all the Room-annotated methods as abstract. If, on the other hand, all you need on the DAO are the Room-annotated methods, you can use an interface and skip all the abstract keywords, as we did with Trip.

We will explore the DAO in greater detail in an upcoming chapter.

Database

In addition to entities and DAOs, you will have at least one @Database-annotated abstract class, extending a RoomDatabase base class. This class knits together the database file, the entities, and the DAOs.

In the sample project, we have a TripDatabase serving this role:

```java
package com.commonsware.android.room;

import android.arch.persistence.room.Database;
import android.arch.persistence.room.Room;
import android.arch.persistence.room.RoomDatabase;
import android.content.Context;

@Database(entities={Trip.class}, version=1)
abstract class TripDatabase extends RoomDatabase {
  abstract TripStore tripStore();

  private static final String DB_NAME="trips.db";
  private static volatile TripDatabase INSTANCE=null;

  synchronized static TripDatabase get(Context ctxt) {
    if (INSTANCE==null) {
      INSTANCE=create(ctxt, false);
    }

    return(INSTANCE);
  }

  static TripDatabase create(Context ctxt, boolean memoryOnly) {
    RoomDatabase.Builder<TripDatabase> b;

    if (memoryOnly) {
      b=Room.inMemoryDatabaseBuilder(ctxt.getApplicationContext(),
        TripDatabase.class);
    }
    else {
      b=Room.databaseBuilder(ctxt.getApplicationContext(), TripDatabase.class,
        DB_NAME);
    }

    return(b.build());
  }
}
```

(from Trips/RoomBasics/app/src/main/java/com/commonsware/android/room/TripDatabase.java)

The `@Database` annotation configures the code generation process, including:

- Identifying all of the entity classes that you care about in the `entities` collection
- Identifying the schema version of the database (as you see with `SQLiteOpenHelper` in conventional Android SQLite development)

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

(from Trips/RoomBasics/app/src/main/java/com/commonsware/android/room/TripDatabase.java)

Here, we are saying that we have just one entity class (`Trip`), and that this is schema version 1.

You also need `abstract` methods for each DAO class that return an instance of that class:

```
  abstract TripStore tripStore();
```

(from Trips/RoomBasics/app/src/main/java/com/commonsware/android/room/TripDatabase.java)

In this app, we have but one DAO (`TripStore`), so we have an `abstract` method to return an instance of `TripStore`.

Extending `RoomDatabase`, having the `@Database` annotation, and having the `abstract` method(s) for your DAOs are the requirements. Anything beyond that is up to you, and some apps may elect to have nothing more here.

In our case, we have a bit more logic.

Get a Room

In this example, the database is a singleton. `TripDatabase` has a `static` getter method, cunningly named `get()`, that creates our singleton. `get()`, in turn, calls a `create()` method that is responsible for creating our `TripDatabase`:

```
static TripDatabase create(Context ctxt, boolean memoryOnly) {
  RoomDatabase.Builder<TripDatabase> b;

  if (memoryOnly) {
    b=Room.inMemoryDatabaseBuilder(ctxt.getApplicationContext(),
      TripDatabase.class);
  }
  else {
    b=Room.databaseBuilder(ctxt.getApplicationContext(), TripDatabase.class,
      DB_NAME);
  }

  return(b.build());
}
```

(from Trips/RoomBasics/app/src/main/java/com/commonsware/android/room/TripDatabase.java)

To create a `TripDatabase`, we use a `RoomDatabase.Builder`, which we get by calling one of two methods on the `Room` class:

- `databaseBuilder()` is what you will normally use
- `inMemoryDatabaseBuilder()` does what the method name suggests: it creates an in-memory SQLite database, useful for instrumentation tests where you do not necessarily need to persist the data for a user

Both of those methods take a `Context` and the Java `Class` object for the desired `RoomDatabase` subclass. `databaseBuilder()` also takes the filename of the SQLite database to use, much as `SQLiteOpenHelper` does in traditional Android SQLite development.

While there are some configuration methods that can be called on the `RoomDatabase.Builder`, we skip those here, simply calling `build()` to build the `TripDatabase`. The result is that when we call `get()`, we get a singleton lazy-initialized `TripDatabase`.

From there, we can:

- Call `tripStore()` on the `TripDatabase` to retrieve the `TripStore` DAO
- Call methods on the `TripStore` to query, insert, update, or delete `Trip` objects

We will see how to do that in the next chapter, where we look at how to write instrumentation tests for our Room-generated database code.

Be Careful with Table Name Prefixes

In SQLite, as with other databases, sometimes you need to use table name prefixes as part of implementing a query. This is particularly true when using `JOIN` syntax, as now 2+ tables are referenced in the query, and sometimes they have duplicate column names. We prefix the column name with the table name (e.g., `table_name.column`) to disambiguate the references.

Room, in general, does not mind this… but there is at least one bug where Room crashes at compile time when encountering a table name prefix. Hopefully, this will get resolved soon.

Testing Room

Once you have a `RoomDatabase` and its associated DAO(s) and entities set up, you should start testing it.

The good news is that testing Room is not dramatically different than is testing anything else in Android. Room has a few characteristics that make it a bit easier than some things to test, as it turns out.

Writing Instrumentation Tests

On the whole, writing instrumentation tests for Room — where the tests run on an Android device or emulator — is unremarkable. You get an instance of your `RoomDatabase` subclass and exercise it from there.

So, for example, here is an instrumentation test case class to exercise the `TripDatabase` from the preceding chapter:

```java
package com.commonsware.android.room;

import android.support.test.InstrumentationRegistry;
import android.support.test.runner.AndroidJUnit4;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import java.util.List;
import static junit.framework.Assert.assertNotNull;
import static junit.framework.Assert.assertTrue;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotEquals;
```

```
@RunWith(AndroidJUnit4.class)
public class TripTests {
  TripDatabase db;
  TripStore store;

  @Before
  public void setUp() {
    db=TripDatabase.create(InstrumentationRegistry.getTargetContext(), true);
    store=db.tripStore();
  }

  @After
  public void tearDown() {
    db.close();
  }

  @Test
  public void basics() {
    assertEquals(0, store.selectAll().size());

    final Trip first=new Trip("Foo", 2880);

    assertNotNull(first.id);
    assertNotEquals(0, first.id.length());
    store.insert(first);

    assertTrip(store, first);

    final Trip updated=new Trip(first.id, "Foo!!!", 1440);

    store.update(updated);
    assertTrip(store, updated);

    store.delete(updated);
    assertEquals(0, store.selectAll().size());
  }

  private void assertTrip(TripStore store, Trip trip) {
    List<Trip> results=store.selectAll();

    assertNotNull(results);
    assertEquals(1, results.size());
    assertTrue(areIdentical(trip, results.get(0)));

    Trip result=store.findById(trip.id);

    assertNotNull(result);
    assertTrue(areIdentical(trip, result));
  }

  private boolean areIdentical(Trip one, Trip two) {
    return(one.id.equals(two.id) &&
      one.title.equals(two.title) &&
      one.duration==two.duration);
  }
}
```

(from Trips/RoomBasics/app/src/androidTest/java/com/commonsware/android/room/TripTests.java)

Here, we:

- Create an empty database
- Get the DAO (`TripStore`)
- Confirm that there are no trips in the database
- Create a `Trip` object and `insert()` it into the database, then confirm that the database was properly inserted
- Create a new `Trip` object with the same ID as the first, `update()` the database using it, then confirm that the database was properly inserted
- Delete the `Trip` object, then confirm that the database has no trips once again

Using In-Memory Databases

When testing a database, though, one of the challenges is in making those tests hermetic, or self-contained. One test method should not depend upon another test method, and one test method should not affect the results of another test method accidentally. This means that we want to start with a known starting point before each test, and we have to consider how to do that.

One approach — the one taken in the above `TripTests` class — is to use an in-memory database. The static `create()` method on `TripDatabase`, if you pass `true` for the second parameter, creates a `TripDatabase` backed by memory, not disk:

```
  static TripDatabase create(Context ctxt, boolean memoryOnly) {
    RoomDatabase.Builder<TripDatabase> b;

    if (memoryOnly) {
      b=Room.inMemoryDatabaseBuilder(ctxt.getApplicationContext(),
        TripDatabase.class);
    }
    else {
      b=Room.databaseBuilder(ctxt.getApplicationContext(), TripDatabase.class,
        DB_NAME);
    }

    return(b.build());
  }
```

(from Trips/RoomBasics/app/src/main/java/com/commonsware/android/room/TripDatabase.java)

There are two key advantages for using an in-memory database for instrumentation testing:

1. It is intrinsically self-contained. Once the `TripDatabase` is closed, its memory is released, and if separate tests use separate `TripDatabase` instances, one will not affect the other.
2. Reading and writing to and from memory is much faster than is reading and writing to and from disk, so the tests run much faster.

On the other hand, this means that the instrumentation tests are useless for performance testing, as (presumably) your production app will actually store its database on disk. You could use Gradle command-line switches, custom build types and `buildConfigField`, or other means to decide when tests are run whether they should use memory or disk.

Importing Starter Data

The one downside to having an empty starter database, such as a fresh in-memory database, is that you have no data. Eventually, you need some data to test.

That could come from test code, such as what `TripTests` does. In many cases, this is a necessary part of testing, to confirm that all of your DAO methods work as expected.

Alternatives include:

- Loading the data from some neutral format (e.g., JSON) via some utility method
- Packaging one or more starter database as assets in the instrumentation tests (e.g., `src/androidTest/assets/`), then using `ATTACH DATABASE ...` and `INSERT INTO ... SELECT FROM ...` SQLite code to copy from the starter database to the database to be used in testing

Writing Unit Tests via Mocks

Let's look again at the TripStore DAO:

```java
package com.commonsware.android.room;

import android.arch.persistence.room.Dao;
import android.arch.persistence.room.Delete;
import android.arch.persistence.room.Insert;
import android.arch.persistence.room.OnConflictStrategy;
import android.arch.persistence.room.Query;
import android.arch.persistence.room.Update;
import java.util.List;

@Dao
interface TripStore {
  @Query("SELECT * FROM trips ORDER BY title")
  List<Trip> selectAll();

  @Query("SELECT * FROM trips WHERE id=:id")
  Trip findById(String id);

  @Insert
  void insert(Trip... trips);

  @Update
  void update(Trip... trips);

  @Delete
  void delete(Trip... trips);
}
```

(from Trips/RoomBasics/app/src/main/java/com/commonsware/android/room/TripStore.java)

This is a pure interface. More importantly, other than annotations, its API is purely domain-specific. Everything revolves around our Trip entity and other business logic (e.g., String values as identifiers).

Room DAOs are designed to be mocked, using a mocking library like Mockito, so that you can write unit tests (tests that run on your development machine or CI server) in addition to — or perhaps instead of — instrumentation tests.

The primary advantage of unit tests is execution speed, as they do not have to be run on Android devices or emulators. On the other hand, setting up mocks can be tedious.

The RoomBasics project not only has the instrumentation tests shown earlier in this chapter, but an equivalent unit test in test/, embodied in a TripUnitTests class:

```
package com.commonsware.android.room;

import org.junit.Before;
import org.junit.Test;
import org.mockito.Matchers;
import org.mockito.Mockito;
import org.mockito.invocation.InvocationOnMock;
import org.mockito.stubbing.Answer;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.HashMap;
import java.util.List;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotEquals;
import static org.junit.Assert.assertNotNull;
import static org.junit.Assert.assertTrue;
import static org.mockito.Matchers.any;
import static org.mockito.Mockito.doAnswer;

public class TripUnitTests {
  private TripStore store;

  @Before
  public void setUp() {
    store=Mockito.mock(TripStore.class);

    final HashMap<String, Trip> trips=new HashMap<>();

    doAnswer(new Answer() {
      @Override
      public Object answer(InvocationOnMock invocation) throws Throwable {
        ArrayList<Trip> result=new ArrayList<>(trips.values());

        Collections.sort(result, new Comparator<Trip>() {
          @Override
          public int compare(Trip left, Trip right) {
            return(left.title.compareTo(right.title));
          }
        });

        return(result);
      }
    }).when(store).selectAll();
```

```
doAnswer(new Answer() {
  @Override
  public Object answer(InvocationOnMock invocation) throws Throwable {
    String id=(String)invocation.getArguments()[0];

    return(trips.get(id));
  }
}).when(store).findById(any(String.class));

doAnswer(new Answer() {
  @Override
  public Object answer(InvocationOnMock invocation) throws Throwable {
    for (Object rawTrip : invocation.getArguments()) {
      Trip trip=(Trip)rawTrip;

      trips.put(trip.id, trip);
    }

    return(null);
  }
}).when(store).insert(Matchers.<Trip>anyVararg());

doAnswer(new Answer() {
  @Override
  public Object answer(InvocationOnMock invocation) throws Throwable {
    for (Object rawTrip : invocation.getArguments()) {
      Trip trip=(Trip)rawTrip;

      trips.put(trip.id, trip);
    }

    return(null);
  }
}).when(store).update(Matchers.<Trip>anyVararg());

doAnswer(new Answer() {
  @Override
  public Object answer(InvocationOnMock invocation) throws Throwable {
    for (Object rawTrip : invocation.getArguments()) {
      Trip trip=(Trip)rawTrip;

      trips.remove(trip.id);
    }

    return(null);
  }
}).when(store).delete(Matchers.<Trip>anyVararg());
```

```
  }

  @Test
  public void basics() {
    assertEquals(0, store.selectAll().size());

    final Trip first=new Trip("Foo", 2880);

    assertNotNull(first.id);
    assertNotEquals(0, first.id.length());
    store.insert(first);

    assertTrip(store, first);

    final Trip updated=new Trip(first.id, "Foo!!!", 1440);

    store.update(updated);
    assertTrip(store, updated);

    store.delete(updated);
    assertEquals(0, store.selectAll().size());
  }

  private void assertTrip(TripStore store, Trip trip) {
    List<Trip> results=store.selectAll();

    assertNotNull(results);
    assertEquals(1, results.size());
    assertTrue(areIdentical(trip, results.get(0)));

    Trip result=store.findById(trip.id);

    assertNotNull(result);
    assertTrue(areIdentical(trip, result));
  }

  private boolean areIdentical(Trip one, Trip two) {
    return(one.id.equals(two.id) &&
      one.title.equals(two.title) &&
      one.duration==two.duration);
  }
}
```

(from Trips/RoomBasics/app/src/test/java/com/commonsware/android/room/TripUnitTests.java)

The `basics()` test method, and its supporting utility methods, are the same as in the instrumentation tests. What differs is where the `TripStore` comes from. In the instrumentation tests, we created an in-memory `TripDatabase` and retrieved a `TripStore` from it. In the unit tests, we use Mockito to create a mock `TripStore` (via `Mockito.mock(TripStore.class)`), then teach the mock how to respond to its methods. In this case, we mock a database with a simple `HashMap`, with a roster of the trips, keyed by their ID values. Each of the `doAnswer()` calls mocks a specific method on the `TripStore`, down to the details of having `selectAll()` return the trips ordered by title.

Whether this is worth the effort is for you to decide. For many projects, instrumentation tests will suffice. For larger projects, where the speed difference between unit tests and instrumentation tests is substantial, it might be worth the engineering time to create the mocks. While mocking is also useful for scenarios that are difficult to reproduce, it is unlikely that your DAO will have any of those scenarios.

The Dao of Entities

Two chapters ago, we went through the basic steps for setting up Room:

- Create and annotate your entity classes
- Create, annotate, and define operator methods on your DAO(s)
- Create a subclass of `RoomDatabase` to tie the entities and DAO(s) together
- Create an instance of that `RoomDatabase` at some likely point in time, while you are safely on a background thread
- Use the `RoomDatabase` instance to retrieve your DAO and from there work with your entities

However, we only scratched the surface of what can be configured on entities and DAOs. In this chapter — and the subsequent chapters on custom types and relations — we will explore the rest of the configuration for entities and DAOs.

Many of the code snippets shown in this chapter come from the `General/RoomDao` sample project. This contains a library module (`stuff`) with entity and DAO code along with instrumentation tests for bits of that code.

Configuring Entities

The only absolute requirements for a Room entity class is that it be annotated with the `@Entity` annotation and have a field identified as the primary key, typically by way of a `@PrimaryKey` annotation. Anything above and beyond that is optional.

However, there is a fair bit that is "above and beyond that". Some — though probably not all — of these features will be of interest in larger apps.

Primary Keys

If you have a single field that is the primary key for your entity, using the `@PrimaryKey` annotation is simple and helps you clearly identify that primary key at a later point.

However, you do have some other options.

Auto-Generated Primary Keys

In SQLite, if you have an `INTEGER` column identified as the `PRIMARY KEY`, you can optionally have SQLite assign unique values for that column, by way of the `AUTOINCREMENT` keyword.

In Room, if you have an `int` or `Integer` field that is your `@PrimaryKey`, you can optionally apply `AUTOINCREMENT` to the corresponding column by adding `autoGenerate=true` to the annotation:

```
@Entity
public class Constant {
  @PrimaryKey(autoGenerate=true)
  public int id;
  String title;
  double value;

  @Override
  public String toString() {
    return(title);
  }
}
```

By default, `autoGenerate` is `false`. Setting that property to `true` gives you `AUTOINCREMENT` in the generated `CREATE TABLE` statement:

```
CREATE TABLE IF NOT EXISTS constants (id INTEGER PRIMARY KEY AUTOINCREMENT, title
TEXT, value REAL NOT NULL)
```

However, this starts to get complicated in the app. You do not know your primary key until you insert the entity into a database. That presents “trickle-down” complications — for example, you cannot make the primary key field `final`, as then you cannot create an instance of an entity that is not yet in the database. While you can try to work around this (e.g., default the `id` to `-1`), then you have to keep checking to see whether you have a valid identifier.

Most of the samples in this book will use a `UUID` instead. While these take up much more room than a simple `int`, they can be uniquely generated outside of the database. For your production apps, you will need to decide if the headaches surrounding database-generated identifiers are worth their benefits.

Also, notice that the `value` column has `NOT NULL` applied to it. Room's rule is that primitive fields (`int`, `double`, etc.) will be `NOT NULL`, while their object equivalents (`Integer`, `Double`, etc.) will allow null values.

Composite Primary Keys

In some cases, you may have a composite primary key, made up of two or more columns in the database. This is particularly true if you are trying to design your entities around an existing database structure, one that used a composite primary key for one of its tables (for whatever reason).

If, logically, those are all part of a single object, you could combine them into a single field, as we will see in the next chapter. However, it may be that they should be individual fields in your entity, but they happen to combine to create the primary key. In that case, you can skip the `@PrimaryKey` annotation and use the `primaryKeys` property of the `@Entity`.

One scenario for this is data versioning, where we are tracking changes to data over time, the way a version control system tracks changes to source code and other files over time. There are several ways of implementing data versioning. One approach has all versions of the same entity in the same table, with a version code attached to the "natural" primary key to identify a specific version of that content. In that case, you could have something like:

```
@Entity(primaryKeys={"id", "versionCode"})
class VersionedThingy {
  public final String id;
  public final int versionCode;

  VersionedThingy(String id, int versionCode) {
    this.id=id;
    this.versionCode=versionCode;
  }
}
```

Room will then use the `PRIMARY KEY` keyword in the `CREATE TABLE` statement to set up the composite primary key:

```
CREATE TABLE IF NOT EXISTS VersionedThingy (id TEXT, versionCode INTEGER NOT NULL,
PRIMARY KEY(id, versionCode))
```

Adding Indexes

Your primary key is indexed automatically by SQLite. However, you may wish to set up other indexes for other columns or collections of columns, to speed up queries. To do that, use the `indices` property on `@Entity`. This property takes a list of `@Index` annotations, each of which declares an index.

For example, as part of a `Customer` entity, you might have an address, which might contain a `postalCode`. You might be querying directly on `postalCode` as part of a search form, and so having an index on that would be useful. To do that, add the appropriate `@Index` to `indices`:

```
@Entity(indices={@Index("postalCode")})
class Customer {
  @PrimaryKey
  public final String id;

  public final String postalCode;
  public final String displayName;

  Customer(String id, String postalCode, String displayName) {
    this.id=id;
    this.postalCode=postalCode;
    this.displayName=displayName;
  }
}
```

Room will add the requested index:

```
CREATE INDEX index_Customer_postalCode ON Customer (postalCode)
```

If you have a composite index, consisting of two or more fields, `@Index` takes a comma-delimited list of column names and will generate the composite index.

If the index should also enforce uniqueness — only one entity can have the indexed value — add `unique=true` to the `@Index`. This requires you to assign the column(s) for the index to the `value` property, due to the way Java annotations work:

```
@Entity(indices={@Index(value="postalCode", unique=true)})
class Customer {
  @PrimaryKey
  public final String id;

  public final String postalCode;
  public final String displayName;

  Customer(String id, String postalCode, String displayName) {
    this.id=id;
    this.postalCode=postalCode;
    this.displayName=displayName;
  }
}
```

This causes Room to add the `UNIQUE` keyword to the `CREATE INDEX` statement:

```
CREATE UNIQUE INDEX index_Customer_postalCode ON Customer (postalCode)
```

Ignoring Fields

If there are fields in the entity class that should not be persisted, annotate them with `@Ignore`:

```
@Entity(primaryKeys={"id", "versionCode"})
class VersionedThingy {
  public final String id;
  public final int versionCode;

  @Ignore
  private String something;

  VersionedThingy(String id, int versionCode) {
    this.id=id;
    this.versionCode=versionCode;
  }
}
```

That annotation is required. For example, this does not work:

```
@Entity(primaryKeys={"id", "versionCode"})
class VersionedThingy {
  public final String id;
  public final int versionCode;

  private String something;

  VersionedThingy(String id, int versionCode) {
    this.id=id;
    this.versionCode=versionCode;
  }
}
```

You might think that since the field is private and has no setter, that Room would ignore it automatically. Room, instead, generates a build error, as it cannot tell if you want to ignore that field or if you simply forgot to add it properly.

Similarly, `transient` is insufficient:

```
@Entity(primaryKeys={"id", "versionCode"})
class VersionedThingy {
  public final String id;
  public final int versionCode;

  public transient String something;

  VersionedThingy(String id, int versionCode) {
    this.id=id;
    this.versionCode=versionCode;
  }
}
```

Since this is a `public` field, Room will try persisting it, even though you have the transient keyword in the Java class. You still need to add `@Ignore` to it.

Note that you can also `@Ignore` constructors. This may be required to clear up Room build errors, if the code generator cannot determine what constructor to use:

```
@Entity(primaryKeys={"id", "versionCode"})
class VersionedThingy {
  public final String id;
  public final int versionCode;

  @Ignore
  private String something;

  @Ignore
  VersionedThingy() {
    this(UUID.randomUUID().toString(), 1);
  }

  VersionedThingy(String id, int versionCode) {
    this.id=id;
    this.versionCode=versionCode;
  }
}
```

NOT NULL Fields

As noted earlier, primitive fields get converted into `NOT NULL` columns in the table, while object fields allow null values.

If you want an object field to be `NOT NULL`, apply the `@NonNull` annotation:

```
@Entity(indices={@Index("postalCode")})
class Customer {
  @PrimaryKey
  public final String id;

  @NonNull
  public final String postalCode;
  public final String displayName;

  Customer(String id, String postalCode, String displayName) {
    this.id=id;
    this.postalCode=postalCode;
    this.displayName=displayName;
  }
}
```

This will make the associated column have `NOT NULL` applied to it.

Custom Table and Column Names

By default, Room will generate names for your tables and columns based off of the entity class names and field names. In general, it does a respectable job of this, and so you may just leave them alone. However, you may find that you need to control these names, particularly if you are trying to match an existing database schema (e.g., you are migrating an existing Android app to use Room instead of using SQLite directly). And for table names in particular, setting your own name can simplify some of the SQL that you have to write for `@Query`-annotated methods.

To control the table name, use the `tableName` property on the `@Entity` attribute, and give it a valid SQLite table name. For example, while in Java we might want to call the class `VersionedThingy`, we might prefer the table to just be `thingy`:

```
@Entity(tableName="thingy", primaryKeys={"id", "versionCode"})
class VersionedThingy {
  public final String id;
  public final int versionCode;

  @Ignore
  private String something;

  @Ignore
  VersionedThingy() {
    this(UUID.randomUUID().toString(), 1);
  }

  VersionedThingy(String id, int versionCode) {
    this.id=id;
    this.versionCode=versionCode;
  }
}
```

To rename a column, add the `@ColumnInfo` annotation to the field, with a `name` property that provides your desired name for the column:

```
@Entity(tableName="thingy", primaryKeys={"id", "versionCode"})
class VersionedThingy {
  public final String id;

  @ColumnInfo(name="version_code")
  public final int versionCode;

  @Ignore
  private String something;

  @Ignore
  VersionedThingy() {
    this(UUID.randomUUID().toString(), 1);
  }

  VersionedThingy(String id, int versionCode) {
    this.id=id;
    this.versionCode=versionCode;
  }
}
```

Here, we changed the `versionCode` field's column to `version_code`, along with specifying the table name.

However, this fails. The values in the `primaryKeys` property are the *column names*, not the field names. Since we renamed the column, we need to update `primaryKeys` to match:

```
package com.commonsware.android.room.dao;

import android.arch.persistence.room.ColumnInfo;
import android.arch.persistence.room.Entity;
import android.arch.persistence.room.Ignore;
import android.arch.persistence.room.PrimaryKey;
import java.util.UUID;

@Entity(tableName="thingy", primaryKeys={"id", "version_code"})
class VersionedThingy {
  public final String id;

  @ColumnInfo(name="version_code")
  public final int versionCode;

  @Ignore
  private String something;

  @Ignore
  VersionedThingy() {
    this(UUID.randomUUID().toString(), 1);
  }

  VersionedThingy(String id, int versionCode) {
    this.id=id;
    this.versionCode=versionCode;
  }
}
```

(from General/RoomDao/stuff/src/main/java/com/commonsware/android/room/dao/VersionedThingy.java)

DAOs and Queries

One popular thing to do with a database is to get data out of it. For that, we add `@Query` methods on our DAO.

Those do not have to be especially complicated, as we saw with the `TripStore`:

```
package com.commonsware.android.room;

import android.arch.persistence.room.Dao;
import android.arch.persistence.room.Delete;
import android.arch.persistence.room.Insert;
import android.arch.persistence.room.OnConflictStrategy;
import android.arch.persistence.room.Query;
import android.arch.persistence.room.Update;
import java.util.List;

@Dao
interface TripStore {
  @Query("SELECT * FROM trips ORDER BY title")
  List<Trip> selectAll();

  @Query("SELECT * FROM trips WHERE id=:id")
  Trip findById(String id);

  @Insert
  void insert(Trip... trips);

  @Update
  void update(Trip... trips);

  @Delete
  void delete(Trip... trips);
}
```

(from Trips/RoomBasics/app/src/main/java/com/commonsware/android/room/TripStore.java)

However, SQL queries with SQLite can get remarkably complicated. Room tries to support a lot of the standard SQL syntax, but Room adds its own complexity, in terms of trying to decipher how to interpret your `@Query` method's arguments and return type.

Adding Parameters

As we saw with `findById()` on `TripStore`, you can map method arguments to query parameters by using `:` syntax. Put `:` before the argument name and its value will be injected into the query:

```
  @Query("SELECT * FROM thingy WHERE id=:id AND version_code=:versionCode")
  VersionedThingy findById(String id, int versionCode);
```

(from General/RoomDao/stuff/src/main/java/com/commonsware/android/room/dao/StuffStore.java)

Bear in mind that the rest of the SQL statement is based on the *table*, not the *entity*. Table names and column names will either be the code-generated names or your overridden names (via `tableName` and `@ColumnInfo`).

WHERE Clause

Principally, your method arguments will be injected into your `WHERE` clause, such as in the above examples.

Note that Room has special support for `IN` in a `WHERE` clause. So, while this works for a single `postalCode`:

```
@Query("SELECT * FROM Customer WHERE postalCode IN (:postalCodes)")
List<Customer> findByPostalCodes(String postalCodes);
```

…you can also do:

```
@Query("SELECT * FROM Customer WHERE postalCode IN (:postalCodes)")
List<Customer> findByPostalCodes(List<String> postalCodes);
```

…or even:

```
@Query("SELECT * FROM Customer WHERE postalCode IN (:postalCodes)")
List<Customer> findByPostalCodes(String... postalCodes);
```

Room will convert the collection argument into a comma-delimited list for use with the SQL query.

Other Clauses

If SQLite allows `?` placeholders, Room should allow method arguments to be used instead.

So, for example, you can parameterize a `LIMIT` clause:

```
  @Query("SELECT * FROM Customer WHERE postalCode IN (:postalCodes) LIMIT :max")
  List<Customer> findByPostalCodes(int max, String... postalCodes);
```

(from General/RoomDao/stuff/src/main/java/com/commonsware/android/room/dao/StuffStore.java)

Here, because Java needs the varargs to be the last parameter, we need to have `max` first.

What You Can Return

We have seen that a `@Query` can return a single entity (e.g., `findById()` returning a single `Trip`) or a collection of entity (e.g., `selectAll()` returning a `List` of `Trip` entities).

While those are simple, Room offers a fair bit more flexibility than that. In particular, not only does Room support reactive return values, but we can return objects that are not actually entities.

Specific Return Types

In addition to returning single objects or collections of objects, a Room `@Query` can return a good old-fashioned `Cursor`. This is particularly useful if you are migrating legacy code that uses `CursorAdapter` or other `Cursor`-specific classes. Similarly, if you are looking to expose part of a Room-defined database via a `ContentProvider`, it may be more convenient for you to get your results in the form of a `Cursor`, so that you can just return that from the provider's `query()` method.

Beyond that, a `@Query` method can return:

- A `Flowable` or `Publisher` from RxJava2, a popular framework for reactive programming
- A `LiveData` object

We will explore what a `LiveData` object is later in this book.

Breadth of Results

For small entities, like `Trip`, usually we will retrieve all columns in the query. However, the real rule is: the core return object of the `@Query` method must be something that Room knows how to fill in from the columns that you request.

For wider tables with many columns, this is important. For example, perhaps for a `RecyclerView`, you only need a couple of columns, but for all entities in the table. In that case, it might be nice to only retrieve those specific columns. You have two ways to do that:

1. Have your `@Entity` support only a subset of columns, allowing the rest to be `null` or otherwise tracking the fact that we only retrieved a subset of columns from the table
2. Return something other than the entity that you have associated with this table

If you look at your `@Dao`-annotated interface, you will notice that while methods might refer to entities, its annotations do not. That is because the DAO is somewhat independent of the entities. The entities describe the table, but the DAO is not limited to using those entities. So long as the DAO can fulfill the contract stipulated by the SQL, the method arguments, and the method return type, Room is perfectly content.

So, for example, suppose that `Customer` not only tracks an `id` and a `postalCode`, but also has *many* other fields, including a `displayName`:

```java
package com.commonsware.android.room.dao;

import android.arch.persistence.room.Entity;
import android.arch.persistence.room.Ignore;
import android.arch.persistence.room.Index;
import android.arch.persistence.room.PrimaryKey;
import java.util.UUID;

@Entity(indices={@Index(value="postalCode", unique=true)})
class Customer {
  @PrimaryKey
  public final String id;

  public final String postalCode;
  public final String displayName;

  @Ignore
  Customer(String postalCode, String displayName) {
    this(UUID.randomUUID().toString(), postalCode, displayName);
  }

  Customer(String id, String postalCode, String displayName) {
    this.id=id;
    this.postalCode=postalCode;
    this.displayName=displayName;
  }
}
```

(from General/RoomDao/stuff/src/main/java/com/commonsware/android/room/dao/Customer.java)

Perhaps to show a list of customers, we need the `displayName` (to show in the list) and the `id` (to know which specific customer this is). But we do not need the `postalCode` or the rest of the fields in the `Customer` class.

We can still return a `Customer`:

```
@Query("SELECT id, displayName FROM Customer WHERE postalCode IN (:postalCodes) LIMIT
:max")
List<Customer> findByPostalCodes(List<String> postalCodes, int max);
```

The code that Room generates will simply fill in `null` for the `postalCode`, since that was not one of the returned columns. However, then it is not obvious whether a given instance of `Customer` is completely filled in from data in the table (and it is genuinely missing its `postalCode`) or whether this is a partially-populated `Customer` object.

However, we could also define a dedicated `CustomerDisplayTuple` class:

```
package com.commonsware.android.room.dao;

public class CustomerDisplayTuple {
  public final String id;
  public final String displayName;

  public CustomerDisplayTuple(String id, String displayName) {
    this.id=id;
    this.displayName=displayName;
  }
}
```

(from General/RoomDao/stuff/src/main/java/com/commonsware/android/room/dao/CustomerDisplayTuple.java)

Then, we can return a `List` of `CustomerDisplayTuple` from our DAO:

```
  @Query("SELECT id, displayName FROM Customer WHERE postalCode IN (:postalCodes) LIMIT :max")
  List<CustomerDisplayTuple> loadDisplayTuplesByPostalCodes(int max, String... postalCodes);
```

(from General/RoomDao/stuff/src/main/java/com/commonsware/android/room/dao/StuffStore.java)

This way, we get our subset of data, and we know by class whether we have the full `Customer` or just the subset for display purposes.

Note that `@ColumnInfo` annotations can be used on any class, not just entities. In particular, if you use `@ColumnInfo` on a field in an entity, you will need the same

`@ColumnInfo` on any "tuple"-style classes that represent subsets of data that include that same field.

Aggregate Functions

A `@Query` can also return an `int`, for simple aggregate functions:

```
  @Query("SELECT COUNT(*) FROM Customer")
  int getCustomerCount();
```

(from General/RoomDao/stuff/src/main/java/com/commonsware/android/room/dao/StuffStore.java)

If you wish to compute several aggregate functions, create a "tuple"-style class to hold the values:

```
package com.commonsware.android.room.dao;

public class CustomerStats {
  public final int count;
  public final String max;

  public CustomerStats(int count, String max) {
    this.count=count;
    this.max=max;
  }
}
```

(from General/RoomDao/stuff/src/main/java/com/commonsware/android/room/dao/CustomerStats.java)

…and use `AS` to name the aggregate function "columns" to match the tuple:

```
  @Query("SELECT COUNT(*) AS count, MAX(postalCode) AS max FROM Customer")
  CustomerStats getCustomerStats();
```

(from General/RoomDao/stuff/src/main/java/com/commonsware/android/room/dao/StuffStore.java)

Dynamic Queries

Sometimes, you do not know the query at compile time.

One scenario for this is when you want to expose a Room-managed database via a `ContentProvider` to third-party apps. You could document that you support a limited set of options in your provider's `query()` method, ones that you can map to `@Query` methods on your DAO. Alternatively, you could generate a SQL statement using SQLiteQueryBuilder that supports what your table offers, but then you need to somehow execute that statement and get a Cursor back.

For that, RoomDatabase has a query() method that is analogous to rawQuery() on a SQLiteDatabase. Pass it the SQL statement and an Object array of position parameters, and RoomDatabase will give you a Cursor back.

Other DAO Operations

To get data out of a database, generally it is useful to put data into it. We have seen basic @Insert, @Update, and @Delete DAO methods on TripStore:

```
package com.commonsware.android.room;

import android.arch.persistence.room.Dao;
import android.arch.persistence.room.Delete;
import android.arch.persistence.room.Insert;
import android.arch.persistence.room.OnConflictStrategy;
import android.arch.persistence.room.Query;
import android.arch.persistence.room.Update;
import java.util.List;

@Dao
interface TripStore {
  @Query("SELECT * FROM trips ORDER BY title")
  List<Trip> selectAll();

  @Query("SELECT * FROM trips WHERE id=:id")
  Trip findById(String id);

  @Insert
  void insert(Trip... trips);

  @Update
  void update(Trip... trips);

  @Delete
  void delete(Trip... trips);
}
```

(from Trips/RoomBasics/app/src/main/java/com/commonsware/android/room/TripStore.java)

Generally speaking, these scenarios are simpler than @Query. The @Insert, @Update, and @Delete set up simple methods for inserting, updating, or deleting entities passed to their methods… and that is pretty much it. However, there are a few additional considerations that we should explore.

Parameters

`@Insert`, `@Update`, and `@Delete` work with entities. `TripStore` uses varargs, so we can pass zero, one, or several `Trip` objects, though passing zero objects would be a waste of time.

However, in addition to varargs, you can have these methods accept:

- A single entity
- Individual entities as separate parameters (`void insert(Trip trip1, Trip trip2)`)
- A `List` of entities

Return Values

Frequently, you just have these methods return `void`.

However:

- For `@Update` and `@Delete`, you can have them return an `int`, which will be the number of rows affected by the update or delete operation
- For an `@Insert` method accepting a single entity, you can have it return a `long` which will be the `ROWID` of the entity (and, if you are using an auto-increment `int` as your primary key, this will also be that key)
- For an `@Insert` method accepting multiple entities, you can have it return an array of `long` objects or a `List` of `Long` objects, being the corresponding `ROWID` values for those inserted entities

Conflict Resolution

`@Insert` and `@Update` support an optional `onConflict` property. This maps to SQLite's `ON CONFLICT` clause and indicates what should happen if there is either a uniqueness violation (e.g., duplicate primary keys) or a `NOT NULL` violation when the insert or update should occur.

The value of `onConflict` is an `OnConflictStrategy` value:

Value	Meaning
`OnConflictStrategy.ABORT`	Cancel this statement but preserve prior results in the transaction and keeps the transaction alive
`OnConflictStrategy.FAIL`	Like `ABORT`, but accepts prior changes by this specific statement (e.g., if we fail on the 50th row to be updated, keep the changes to the preceding 49)
`OnConflictStrategy.IGNORE`	Like `FAIL`, but continues processing this statement (e.g., if we fail on the 50th row out of 100, keep the changes to the other 99)
`OnConflictStrategy.REPLACE`	For uniqueness violations, deletes other rows that would cause the violation before executing this statement
`OnConflictStrategy.ROLLBACK`	Rolls back the current transaction

The default strategy for `@Insert` and `@Update` is `ABORT`. You might want to consider changing that to be `ROLLBACK`, particularly if you start using transactions:

```
@Insert(onConflict=OnConflictStrategy.ROLLBACK)
void insert(Trip... trips);
```

Other Operations

The primary problem with `@Insert`, `@Update`, and `@Delete` is that they need entities. In part, that is so the DAO method knows what table to work against.

For anything else, use `@Query`. `@Query` does not only work with operations that return result sets, but *any* SQL that you wish to execute.

So, for example, you could have:

```
@Query("DELETE FROM Customer")
void nukeCustomersFromOrbit();
```

...or:

```
  @Query("DELETE FROM Customer WHERE id IN (:ids)")
  int nukeCertainCustomersFromOrbit(String... ids);
```

(from General/RoomDao/stuff/src/main/java/com/commonsware/android/room/dao/StuffStore.java)

...or `INSERT INTO ... SELECT FROM ...` syntax, or pretty much any other combination that cannot be supported directly by `@Insert`, `@Update`, and `@Delete` annotations directly.

Consider `@Insert`, `@Update`, and `@Delete` to be "convenience annotations" for entity-based operations, where `@Query` is the backbone for your DAO methods.

Transactions and Room

Many times, we need to wrap a number of SQL statements into a transaction. `RoomDatabase` offers the same `beginTransaction()`, `endTransaction()`, and `setTransactionSuccessful()` methods that you see on `SQLiteDatabase`, and so you use the same basic algorithm:

```
roomDb.beginTransaction();

try {
  // bunch of DAO operations here
  roomDb.setTransactionSuccessful();
}
finally {
  roomDb.endTransaction();
}
```

Threads and Room

`@Insert`, `@Update`, and `@Delete`-annotated methods are synchronous, performing their work on the current thread. Hence, they should only be called from a background thread.

`@Query` methods that return entities, `int`, tuples, etc. directly also are synchronous. However, `@Query` methods that return an RxJava type (e.g., `Flowable`) or a `LiveData` are *not* synchronous. Instead, the real work will be performed on a background thread.

As noted earlier, we will explore what this "`LiveData`" is later in the book. For now, take it on faith that it is another piece of the Android Architecture Components, one that offers an alternative to RxJava for reactive programming

Room and Custom Types

So far, all of our fields have been basic primitives (`int`, `float`, etc.) or `String`. There is a good reason for that: those are all that Room understands "out of the box". Everything else requires some amount of assistance on our part.

Sometimes, a field in an entity will be related to another entity. Those are relations, and we will consider those in the next chapter.

However, sometimes our preferred Java entity implementation does not map directly to primitives and `String` types. For example:

- What do we do with a Java `Date` or `Calendar` object? Do we want to store that as a milliseconds-since-the-Unix-epoch value as a Java `long`? Do we want to store a string representation in a standard format, for easier readability (at the cost of disk space and other issues)?
- What do we do with a `Location` object? Here, we have two pieces: a latitude and a longitude. Do we have two columns that combine into one field? Do we convert the `Location` to and from a `String` representation?
- What do we do with collections of strings, such as lists of tags?
- What do we do with enums?

And so on.

In this chapter, we will explore two approaches for handling these things without creating another entity class: type converters and embedded types.

Type Converters

Type converters are a pair of methods, annotated with `@TypeConverter`, that map the type for a single database column to a type for a Java field. So, for example, we can:

- Map a `Date` field to a `Long`, which can go in a SQLite `INTEGER` column
- Map a `Location` field to a `String`, which can go in a SQLite `TEXT` column
- Map a collection of `String` values to a single `String` (e.g., comma-separated values), which can go in a SQLite `TEXT` column
- And so forth

However, type converters offer only a 1:1 conversion: a single Java field to and from a single SQLite column. If you have a single Java field that should map to several SQLite columns, the `@Embedded` approach can handle that, as we will see later in this chapter.

Setting Up a Type Converter

First, define a Java class somewhere. The name, package, superclass, etc. do not matter.

Next, for each type to be converted, create two `public static` methods that convert from one type to the other. So for example, you would have one `public static` method that takes a `Date` and returns a `Long` (e.g., returning the milliseconds-since-the-Unix-epoch value), and a counterpart method that takes a `Long` and returns a `Date`. If the converter method is passed `null`, the proper result is `null`. Otherwise, the conversion is whatever you want, so long as the "round trip" works, so that the output of one converter method, supplied as input to the other converter method, returns the original value.

Then, each of those methods get the `@TypeConverter` annotation. The method names do not matter, so pick a convention that works for you.

Finally, you add a `@TypeConverters` annotation, listing this and any other type converter classes, to… something. What the "something" is controls the scope of where that type converter can be used.

The simple solution is to add `@TypeConverters` to the `RoomDatabase`, which means that anything associated with that database can use those type converters. However, sometimes, you may have situations where you want different conversions between the same pair of types, for whatever reason. In that case, you can put the `@TypeConverters` annotations on narrower scopes:

`@TypeConverters` Location	Affected Areas
Entity class	all fields in the entity
Entity field	that one field in the entity
DAO class	all methods in the DAO
DAO method	that one method in the DAO, for all parameters
DAO method parameter	that one parameter on that one method
POJO	all fields on the POJO

The `General/RoomTypes` sample project illustrates the use of type converters. As with the `RoomDao` project from the preceding chapter, this project contains a single library module with an associated instrumentation test case. In fact, it is a clone of the `RoomDao` project, just with some type converters.

Example: Dates and Times

A typical way of storing a date/time value in a database is to use the number of milliseconds since the Unix epoch (i.e., the number of milliseconds since midnight, 1 January 1970). `Date` has a `getTime()` method that returns this value.

So, the project has a `TypeTransmogrifiers` class that contains two methods, each annotated with `@TypeConverter`, for converting `Date` to and from a `Long`:

```
@TypeConverter
public static Long fromDate(Date date) {
  if (date==null) {
    return(null);
  }

  return(date.getTime());
}

@TypeConverter
public static Date toDate(Long millisSinceEpoch) {
  if (millisSinceEpoch==null) {
    return(null);
  }

  return(new Date(millisSinceEpoch));
}
```

(from General/RoomTypes/stuff/src/main/java/com/commonsware/android/room/dao/TypeTransmogrifier.java)

`StuffDatabase` then has the `@TypeConverters` annotation, listing `TypeTransmogrifier` as the one class that has type conversion methods:

```
package com.commonsware.android.room.dao;

import android.arch.persistence.room.Database;
import android.arch.persistence.room.Room;
import android.arch.persistence.room.RoomDatabase;
import android.arch.persistence.room.TypeConverters;
import android.content.Context;

@Database(
  entities={Customer.class, VersionedThingy.class},
  version=1
)
@TypeConverters({TypeTransmogrifier.class})
abstract class StuffDatabase extends RoomDatabase {
  abstract StuffStore stuffStore();

  private static final String DB_NAME="stuff.db";
  private static volatile StuffDatabase INSTANCE=null;

  synchronized static StuffDatabase get(Context ctxt) {
    if (INSTANCE==null) {
      INSTANCE=create(ctxt, false);
    }

    return(INSTANCE);
  }

  static StuffDatabase create(Context ctxt, boolean memoryOnly) {
    RoomDatabase.Builder<StuffDatabase> b;

    if (memoryOnly) {
      b=Room.inMemoryDatabaseBuilder(ctxt.getApplicationContext(),
        StuffDatabase.class);
    }
    else {
      b=Room.databaseBuilder(ctxt.getApplicationContext(), StuffDatabase.class,
        DB_NAME);
    }

    return(b.build());
  }
}
```

(from General/RoomTypes/stuff/src/main/java/com/commonsware/android/room/dao/StuffDatabase.java)

Now, classes like `Customer` can use `Date` fields, which will be stored in `INTEGER` columns in the database.

```
CREATE TABLE IF NOT EXISTS Customer (id TEXT, postalCode TEXT, displayName TEXT,
creationDate INTEGER, PRIMARY KEY(`id`))
```

Example: Locations

A `Location` object contains a latitude, longitude, and perhaps other values (e.g., altitude). If we only care about the latitude and longitude, we could save those in the database in a single `TEXT` column, so long as we can determine a good format to use for that string. If we use `Locale.US` formatting for the latitude and longitude, so that the decimal place is denoted by a `.`, we could use a two-element comma-separated values list for the string.

That is what these two type converter methods on `TypeTransmogrifiers` do:

```
@TypeConverter
public static String fromLocation(Location location) {
  if (location==null) {
    return(null);
  }

  return(String.format(Locale.US, "%f,%f", location.getLatitude(),
    location.getLongitude()));
}

@TypeConverter
public static Location toLocation(String latlon) {
  if (latlon==null) {
    return(null);
  }

  String[] pieces=latlon.split(",");
  Location result=new Location("");

  result.setLatitude(Double.parseDouble(pieces[0]));
  result.setLongitude(Double.parseDouble(pieces[1]));

  return(result);
}
```

(from General/RoomTypes/stuff/src/main/java/com/commonsware/android/room/dao/TypeTransmogrifier.java)

Since TypeTransmogrifiers is registered on the StuffDatabase, a Customer could have a Location field, which would be mapped to a TEXT column in the database:

```
CREATE TABLE IF NOT EXISTS Customer (id TEXT, postalCode TEXT, displayName TEXT,
creationDate INTEGER, officeLocation TEXT, PRIMARY KEY(`id`))
```

However, the downside of using this approach is that we cannot readily search based on location. If your location data is not a searchable field, and it merely needs to be available when you load your entities from the database, using a type converter like this is fine. Later in this chapter, we will see another approach (@Embedded) that allows us to store the latitude and longitude as separate columns while still mapping them to a single POJO in Java.

Example: Simple Collections

TEXT and BLOB columns are very flexible. So long as you can marshal your data into a String or byte array, you can save that data in TEXT and BLOB columns. As with the comma-separated values approach in the preceding section, though, columns used this way are poor for searching.

So, suppose that you have a Set of String values that you want to store, perhaps representing tags to associate with an entity. One approach is to have a separate Tag entity and set up a relation. This is the best approach in many cases. But, perhaps you do not want to do that for some reason.

You can use a type converter, but you need to decide how to represent your data in a column. If you are certain that the tags will not contain some specific character (e.g., a comma), you can use the delimited-list approach demonstrated with locations in the preceding section. If you need more flexibility than that, you can always use JSON encoding, as these type converters do:

```
@TypeConverter
public static String fromStringSet(Set<String> strings) {
  if (strings==null) {
    return(null);
  }

  StringWriter result=new StringWriter();
  JsonWriter json=new JsonWriter(result);

  try {
    json.beginArray();
```

```
      for (String s : strings) {
        json.value(s);
      }

      json.endArray();
      json.close();
    }
    catch (IOException e) {
      Log.e(TAG, "Exception creating JSON", e);
    }

    return(result.toString());
  }

  @TypeConverter
  public static Set<String> toStringSet(String strings) {
    if (strings==null) {
      return(null);
    }

    StringReader reader=new StringReader(strings);
    JsonReader json=new JsonReader(reader);
    HashSet<String> result=new HashSet<>();

    try {
      json.beginArray();

      while (json.hasNext()) {
        result.add(json.nextString());
      }

      json.endArray();
    }
    catch (IOException e) {
      Log.e(TAG, "Exception parsing JSON", e);
    }

    return(result);
  }
```

(from General/RoomTypes/stuff/src/main/java/com/commonsware/android/room/dao/TypeTransmogrifier.java)

Here, we use the `JsonReader` and `JsonWriter` classes that have been part of Android since API Level 11. Alternatively, you could use a third-party JSON library (e.g., Gson).

Note that type converter methods cannot throw checked exceptions, as the Room code generator does not wrap type converter calls in a `try/catch` block. Here, the `IOExceptions` should never happen, since we are working with strings, not files or other types of streams. In other cases, though, you may need to wrap the checked exception in some form of `RuntimeException` and throw that, to trigger your app's unhandled-exception logic, as it is unlikely that you can recover from within a type converter method.

But, given these type conversion methods, we can now use a `Set` of `String` values in `Customer`:

```java
package com.commonsware.android.room.dao;

import android.arch.persistence.room.Entity;
import android.arch.persistence.room.Ignore;
import android.arch.persistence.room.Index;
import android.arch.persistence.room.PrimaryKey;
import android.location.Location;
import java.util.Date;
import java.util.HashSet;
import java.util.Set;
import java.util.UUID;

@Entity(indices={@Index(value="postalCode", unique=true)})
class Customer {
  @PrimaryKey
  public final String id;

  public final String postalCode;
  public final String displayName;
  public final Date creationDate;
  public final Location officeLocation;
  public final Set<String> tags;

  @Ignore
  Customer(String postalCode, String displayName, Location officeLocation,
           Set<String> tags) {
    this(UUID.randomUUID().toString(), postalCode, displayName, new Date(),
      officeLocation, tags);
  }

  Customer(String id, String postalCode, String displayName, Date creationDate,
           Location officeLocation, Set<String> tags) {
    this.id=id;
    this.postalCode=postalCode;
    this.displayName=displayName;
    this.creationDate=creationDate;
    this.officeLocation=officeLocation;
    this.tags=tags;
  }
}
```

(from General/RoomTypes/stuff/src/main/java/com/commonsware/android/room/dao/Customer.java)

…where the `tags` will be stored in a `TEXT` column:

```
CREATE TABLE IF NOT EXISTS Customer (id TEXT, postalCode TEXT, displayName TEXT,
creationDate INTEGER, officeLocation TEXT, tags TEXT, PRIMARY KEY(`id`))
```

Embedded Types

With type converters, we are teaching Room how to deal with custom types, but we are limited to mapping from one field to one column. That field might be complex, but it still goes into one column in the table.

What happens, though, when we have multiple columns that should combine to create a single field?

In that case, we can use the `@Embedded` annotation on some POJO, then use that POJO as a type in an entity.

Example: Locations

For example, as was noted earlier in this chapter, cramming a location into a single `TEXT` field works, but we cannot readily query on the resulting field. If we want to query for locations near some location in the database, it would be much more convenient to have the latitude and longitude stored as individual `REAL` columns. But, using type converters, we cannot map two columns to one field.

With `@Embedded`, we can, as we can see in the `General/RoomEmbedded` sample project. This is a clone of the `RoomTypes` project from earlier in this chapter, where we have changed `Customer` to have the `officeLocation` be represented by a `LocationColumns` POJO:

```
package com.commonsware.android.room.dao;

import android.arch.persistence.room.Embedded;
import android.arch.persistence.room.Entity;
import android.arch.persistence.room.Ignore;
import android.arch.persistence.room.Index;
import android.arch.persistence.room.PrimaryKey;
import android.location.Location;
import java.util.Date;
import java.util.HashSet;
import java.util.Set;
import java.util.UUID;

@Entity(indices={@Index(value="postalCode", unique=true)})
class Customer {
  @PrimaryKey
  public final String id;

  public final String postalCode;
  public final String displayName;
  public final Date creationDate;

  @Embedded
  public final LocationColumns officeLocation;

  public final Set<String> tags;

  @Ignore
  Customer(String postalCode, String displayName, LocationColumns officeLocation,
           Set<String> tags) {
    this(UUID.randomUUID().toString(), postalCode, displayName, new Date(),
      officeLocation, tags);
  }

  Customer(String id, String postalCode, String displayName, Date creationDate,
           LocationColumns officeLocation, Set<String> tags) {
    this.id=id;
    this.postalCode=postalCode;
    this.displayName=displayName;
    this.creationDate=creationDate;
    this.officeLocation=officeLocation;
    this.tags=tags;
  }
}
```

(from General/RoomEmbedded/stuff/src/main/java/com/commonsware/android/room/dao/Customer.java)

The `@Embedded` annotation tells Room to combine the columns from the annotated type into the table for this entity. In this case, `LocationColumns` has two fields, for latitude and longitude:

```
package com.commonsware.android.room.dao;

public class LocationColumns {
  public final double latitude;
  public final double longitude;

  public LocationColumns(double latitude, double longitude) {
    this.latitude=latitude;
    this.longitude=longitude;
  }
}
```

(from General/RoomEmbedded/stuff/src/main/java/com/commonsware/android/room/dao/LocationColumns.java)

`LocationColumns` itself is a POJO, not an entity, though you can use `@ColumnInfo` annotations if needed to rename the columns associated with the POJO's fields.

Now, Room will use individual `REAL` columns for our latitude and longitude:

```
CREATE TABLE IF NOT EXISTS Customer (id TEXT, postalCode TEXT, displayName TEXT,
creationDate INTEGER, tags TEXT, latitude REAL, longitude REAL, PRIMARY KEY(id))
```

…and we can query on those:

```
  @Query("SELECT * FROM Customer WHERE ABS(latitude-:lat)<.000001 AND ABS(longitude-:lon)<.000001")
  List<Customer> findCustomersAt(double lat, double lon);
```

(from General/RoomEmbedded/stuff/src/main/java/com/commonsware/android/room/dao/StuffStore.java)

Simple vs. Prefixed

What happens if we need *two* locations, though? Perhaps we need `officeLocation` and `affiliateLocation`, or something like that.

By default, Room generates column names based on the `@Embedded` POJO's field names, perhaps modified by `@ColumnInfo` annotations on the POJO. In this case, though, if we have two `LocationColumns` fields in the `Customer` entity, we would wind up with two `latitude` and two `longitude` columns, which neither Room nor SQLite will support.

To address this, the `@Embedded` annotation accepts an optional `prefix` property:

```
@Embedded(prefix = "office_")
public final LocationColumns officeLocation;
```

The columns for that POJO will have the `prefix` added:

```
CREATE TABLE IF NOT EXISTS Customer (id TEXT, postalCode TEXT, displayName TEXT,
creationDate INTEGER, tags TEXT, office_latitude REAL, office_longitude REAL, PRIMARY
KEY(id))
```

Hence, having two `LocationColumns` simply means that one or both need to use distinct `prefix` values.

However, bear in mind that this changes the column names, so you will also need to adjust any `@Query` method that references those names, so that you use the appropriate prefix.

Updating the Trip Sample

Back in the chapter on Room basics, we started in on an app to track upcoming travel. The `Trips/RoomConverters` sample project extends that app with four new fields on `Trip`:

- `priority`, representing how important the trip is to the user
- `startTime`, indicating when the trip is to begin
- `creationTime`, indicating when the `Trip` was first created… somewhere
- `updateTime`, indicating when the `Trip` was last changed… somewhere

Those latter two are largely ignored for the moment, though they will become more important later in the book.

The latter three are all `Date` fields, and so we need to have some code to support getting them into and out of our table. So, this project has a `TypeTransmogrifier` class, akin to the ones seen above, but right now only with the `Date` converters:

```
package com.commonsware.android.room;

import android.arch.persistence.room.TypeConverter;
import java.util.Date;

public class TypeTransmogrifier {
  @TypeConverter
  public static Long fromDate(Date date) {
    if (date==null) {
      return(null);
    }

    return(date.getTime());
  }

  @TypeConverter
  public static Date toDate(Long millisSinceEpoch) {
    if (millisSinceEpoch==null) {
      return(null);
    }
    return(new Date(millisSinceEpoch));
  }
}
```

(from Trips/RoomConverters/app/src/main/java/com/commonsware/android/room/TypeTransmogrifier.java)

`priority`, though, is an `enum`, as there is a list of valid values:

```
package com.commonsware.android.room;

import android.arch.persistence.room.TypeConverter;

enum Priority {
  LOW(0), MEDIUM(1), HIGH(2), OMG(3);

  private final int level;

  @TypeConverter
  public static Priority fromLevel(Integer level) {
    for (Priority p : values()) {
      if (p.level==level) {
        return(p);
      }
    }

    return(null);
  }

  @TypeConverter
  public static Integer fromPriority(Priority p) {
    return(p.level);
  }

  Priority(int level) {
    this.level=level;
  }
}
```

(from Trips/RoomConverters/app/src/main/java/com/commonsware/android/room/Priority.java)

Here, we implement the `@TypeConverter` methods right on `Priority`, as there is little value in having them elsewhere. Note that the `enum` assigns explicit numeric values to the priorities (`level`). That way, we are in control over the mapping between `Priority` values and their representation in the database.

Rather than apply these type converters on the `TripDatabase` (though we could), we instead apply them on the `Trip` model:

```
package com.commonsware.android.room;

import android.arch.persistence.room.Entity;
import android.arch.persistence.room.Ignore;
import android.arch.persistence.room.PrimaryKey;
import android.arch.persistence.room.TypeConverters;
import java.util.Date;
import java.util.UUID;

@Entity(tableName = "trips")
@TypeConverters({TypeTransmogrifier.class})
class Trip {
  @PrimaryKey
  public final String id;

  public final String title;
  public final int duration;

  @TypeConverters({Priority.class})
  public final Priority priority;

  public final Date startTime;
  public final Date creationTime;
  public final Date updateTime;

  @Ignore
  Trip(String title, int duration, Priority priority, Date startTime) {
    this(UUID.randomUUID().toString(), title, duration, priority, startTime,
      null, null);
  }

  Trip(String id, String title, int duration, Priority priority,
       Date startTime, Date creationTime, Date updateTime) {
    this.id=id;
    this.title=title;
    this.duration=duration;
    this.priority=priority;
    this.startTime=startTime;
    this.creationTime=creationTime;
    this.updateTime=updateTime;
  }

  @Override
  public String toString() {
    return(title);
  }
}
```

(from Trips/RoomConverters/app/src/main/java/com/commonsware/android/room/Trip.java)

The `Priority` type converters are applied specific to the `priority` field, as this specific conversion is only needed here. The `TypeTransmogrifier` is registered on the `Trip` class, as there are multiple `Date` fields.

Room and Relations

SQLite is a relational database. At some point, Room should support relations. Right?

Right?!?

Well, actually, the story is a bit more complicated than that. Yes, Room supports entities being related to other content in other tables. Room does *not* support entities being directly related to other entities, though.

And if that sounds strange, there is "a method to the madness".

In this chapter, we will explore how you implement relational structures with Room and why Room has the restrictions that it does.

The Classic ORM Approach

Java ORMs have long supported entities having relations to other entities, though not every ORM uses the "entity" term.

One Android ORM that does is greenDAO. It allows you to use annotations to indicate relations, such as:

```
@Entity
public class Thingy {
  @Id private Long id;

  private long otherThingyId;

  @ToOne(joinProperty="otherThingyId")
  private OtherThingy otherThingy;

  // other good stuff here
}

@Entity
public class OtherThingy {
  @ID private Long id;
}
```

These annotations result in `getOtherThingy()` and `setOtherThingy()` methods to be synthetically added to `Thingy` (or, more accurately, to a hidden subclass of `Thingy`, but for the purposes of this section, we will ignore that). Which `OtherThingy` our `Thingy` relates to is tied to that `otherThingyId` field, which is stored as a column in the table. When you call `getOtherThingy()`, greenDAO will query the database to load in the `OtherThingy` instance, assuming that it has not been cached already.

That is where the threading problem creeps in.

A History of Threading Mistakes

In Android app development, we are constantly having to fight to keep disk I/O off of the main application thread. Every millisecond that our code executes on the main application thread is a millisecond that the main application thread is not updating our UI. Disk I/O — such as queries on complex structures – can easily take dozens or hundreds of milliseconds, particularly on older or low-end devices. As a result, we freeze our UI while that disk I/O is occurring, possibly resulting in visual “jank” for the user. Our objective is to move as much disk I/O as possible off the main application thread.

The problem is that the nice encapsulation that we get from object-oriented programming also encapsulates knowledge of whether disk I/O will be done when we call a particular method.

Classic use of `SQLiteDatabase` encounters this with the `rawQuery()`/`query()` family of methods. They return a `Cursor`. You might think — reasonably – that those methods execute the SQL query that you request. In truth, they do not. All they do is create a `SQLiteCursor` instance that holds onto the query and the `SQLiteDatabase`. Later, when you call a method that requires the actual query result (e.g., `getCount()`, to get the number of returned rows), *then* the query is executed against the database. As a result, all the work that you do to call `rawQuery()` or `query()` on a background thread gets wasted if you do not *also* do something to force the query to be executed on that same background thread. Otherwise, you may wind up with the query being executed on the main application thread, with impacts on the UI.

greenDAO relations can work the same way. If you retrieve your `Thingy` on a background thread, then call `getOtherThingy()` on the main application thread, depending on what else has all occurred, `getOtherThingy()` might need to perform a database query… which you do not want on the main application thread.

The Room Approach

Room behaves a bit like other annotation-based Android ORMs, but when it comes to relations, Room departs from norms, in an effort to reduce the likelihood of threading problems.

No Direct Entity References

Unlike the greenDAO example above, with Room, a `Thingy` cannot have a field for an `OtherThingy` that Room is expected to manage. You could have a field for an `OtherThingy` marked as `@Ignore`, but then you are on your own for dealing with that field.

The implication of an entity referencing another entity directly is that developers would expect that when Room retrieves the outer entity, that Room either will automatically retrieve the inner entity or will retrieve it lazily later on. The former approach avoids threading issues but runs the risk of loading more data than is necessary. The latter approach runs the risk of trying to do disk I/O on the main application thread.

Foreign Keys

This does not mean that you cannot have foreign keys. Room fully supports foreign key relationships, by way of a `@ForeignKey` annotation. This sets up the foreign keys in the appropriate tables… but that’s about it. Room does very little else with these keys.

Cascades on Updates and Deletes

Part of what you can place on a `@ForeignKey` annotation are `onUpdate` and `onDelete` properties. These indicate what actions should be taken on this entity when the parent of the foreign key relationship is updated or deleted. There are five possibilities, denoted by `ForeignKey` constants:

Constant Name	If the Parent Is Updated or Deleted…
`NO_ACTION`	…do nothing
`CASCADE`	…update or delete the child
`RESTRICT`	…fail the parent's update or delete operation, unless there are no children
`SET_NULL`	…set the foreign key value to `null`
`SET_DEFAULT`	…set the foreign key value to the column(s) default value

`NO_ACTION` is the default, though `CASCADE` will be a popular choice.

Cascades on… Retrievals?

You cannot have an entity automatically retrieve related objects via a `@Query`.

You *can* have an arbitrary POJO automatically retrieve related objects via a `@Query`, by means of a `@Relation` annotation.

This seeming inconsistency will be explored later in this chapter.

Plans for Trips

Let's explore how `@ForeignKey` works by adding some more entities to the trip-tracking app, as seen in the `Trips/RoomRelations` sample project.

The app itself does not make use of these new changes in its fledgling UI — we will address that much later in the book. This sample also drops off the mock-database unit tests. For now, the focus is on adding the necessary Room bits and updating the instrumentation tests.

The Domain Model

In the beginning, we had just the `Trip` entity. However, a trip is made up of lots of pieces, so in this sample, we add two more: flights and lodgings. Not surprisingly, these come in the form of `Flight` and `Lodging` entity classes. A `Trip` can have zero or more related `Flight` instances and zero or more related `Lodging` instances.

However, many of the pieces of data that we need to track for these things – title, duration, start time, etc. — are in common. So, we will pull those things into an `abstract` base class named `Plan`, from which `Trip`, `Flight`, and `Lodging` will all inherit.

The New Entities

As a result, `Plan` itself is pretty much what `Trip` used to be:

```
package com.commonsware.android.room;

import android.arch.persistence.room.Ignore;
import android.arch.persistence.room.PrimaryKey;
import android.arch.persistence.room.TypeConverters;
import java.util.Date;
import java.util.UUID;

abstract class Plan {
  @PrimaryKey
  public final String id;

  public final String title;
  public final int duration;

  @TypeConverters({Priority.class})
  public final Priority priority;

  public final Date startTime;
  public final Date creationTime;
  public final Date updateTime;

  @Ignore
  Plan(String title, int duration, Priority priority, Date startTime) {
    this(UUID.randomUUID().toString(), title, duration, priority, startTime,
      null, null);
  }

  Plan(String id, String title, int duration, Priority priority,
       Date startTime, Date creationTime, Date updateTime) {
    this.id=id;
    this.title=title;
    this.duration=duration;
    this.priority=priority;
    this.startTime=startTime;
    this.creationTime=creationTime;
    this.updateTime=updateTime;
  }

  @Override
  public String toString() {
    return(title);
  }
}
```

(from Trips/RoomRelations/app/src/main/java/com/commonsware/android/room/Plan.java)

Note that while we have the `Priority` `TypeConverter` registered for the `Priority` field, we do not have the `TypeTransmogrifier` registered on the `Plan` class, the way we had it for `Trip`. That is due to a limitation in Room, whereby class-level `@TypeConverters` annotations are not inherited, though field-level ones are.

Instead, the `TypeTransmogrifier` `@TypeConverters` annotation appears on our rump `Trip` class:

```
package com.commonsware.android.room;

import android.arch.persistence.room.Entity;
import android.arch.persistence.room.Ignore;
import android.arch.persistence.room.PrimaryKey;
import android.arch.persistence.room.TypeConverters;
import java.util.Date;
import java.util.UUID;

@Entity(tableName = "trips")
@TypeConverters({TypeTransmogrifier.class})
class Trip extends Plan {
  @Ignore
  Trip(String title, int duration, Priority priority, Date startTime) {
    super(title, duration, priority, startTime);
  }

  Trip(String id, String title, int duration,
       Priority priority, Date startTime, Date creationTime,
       Date updateTime) {
    super(id, title, duration, priority, startTime, creationTime, updateTime);
  }
}
```

(from Trips/RoomRelations/app/src/main/java/com/commonsware/android/room/Trip.java)

The relations that we are setting up from `Trip` to `Flight` and `Lodging` are 1:N relations. As such, the parent (`Trip`) does not need any foreign keys. Those are held by the children of the relation… such as `Lodging`:

```
package com.commonsware.android.room;

import android.arch.persistence.room.Entity;
import android.arch.persistence.room.ForeignKey;
import android.arch.persistence.room.Ignore;
import android.arch.persistence.room.Index;
import android.arch.persistence.room.TypeConverters;
import java.util.Date;
import static android.arch.persistence.room.ForeignKey.CASCADE;

@Entity(
  tableName="lodgings",
  foreignKeys=@ForeignKey(
    entity=Trip.class,
    parentColumns="id",
    childColumns="tripId",
    onDelete=CASCADE),
  indices=@Index("tripId"))
@TypeConverters({TypeTransmogrifier.class})
class Lodging extends Plan {
  public final String address;
  public final String tripId;

  @Ignore
  Lodging(String title, int duration, Priority priority, Date startTime,
          String address, String tripId) {
    super(title, duration, priority, startTime);
    this.address=address;
    this.tripId=tripId;
  }

  Lodging(String id, String title, int duration,
          Priority priority, Date startTime, Date creationTime,
          Date updateTime, String address, String tripId) {
    super(id, title, duration, priority, startTime, creationTime, updateTime);
    this.address=address;
    this.tripId=tripId;
  }
}
```

(from [Trips/RoomRelations/app/src/main/java/com/commonsware/android/room/Lodging.java]())

Here, `Lodging` also extends from `Plan`, adding two fields, one to track the address of the hotel (or whatever) and the `tripId` of the `Trip` that contains this `Lodging`. That `tripId` field is then referenced in the `@ForeignKey` annotation,which:

- Sets up the relation as being with `Trip` (`entity=Trip.class`)
- Ties the `id` column on `Trip` (`parentColumns="id"`) to the `tripId` on `Lodging` (`childColumns="tripId"`)
- Indicates that if the `Trip` is deleted, associated `Lodging` instances should also be deleted (`onDelete=CASCADE`)

`Lodging` also sets up an index on `tripId` (`indices=@Index("tripId")`). Querying on `tripId` will be fairly common, as we look up the `Lodging` instances associated with a given `Trip`. Hence, typically you will want to set up an index on your foreign keys. Room will even warn you about this, if you examine the Gradle Console output from a build.

`Flight` works similarly:

```
package com.commonsware.android.room;

import android.arch.persistence.room.Entity;
import android.arch.persistence.room.ForeignKey;
import android.arch.persistence.room.Ignore;
import android.arch.persistence.room.Index;
import android.arch.persistence.room.TypeConverters;
import java.util.Date;
import static android.arch.persistence.room.ForeignKey.CASCADE;

@Entity(
  tableName="flights",
  foreignKeys=@ForeignKey(
    entity=Trip.class,
    parentColumns="id",
    childColumns="tripId",
    onDelete=CASCADE),
  indices=@Index("tripId"))
@TypeConverters({TypeTransmogrifier.class})
class Flight extends Plan {
  public final String departingAirport;
  public final String arrivingAirport;
  public final String airlineCode;
  public final String flightNumber;
  public final String seatNumber;
  public final String tripId;

  @Ignore
  Flight(String title, int duration, Priority priority, Date startTime,
         String departingAirport, String arrivingAirport, String airlineCode,
         String flightNumber, String seatNumber, String tripId) {
    super(title, duration, priority, startTime);
    this.departingAirport=departingAirport;
    this.arrivingAirport=arrivingAirport;
    this.airlineCode=airlineCode;
    this.flightNumber=flightNumber;
    this.seatNumber=seatNumber;
    this.tripId=tripId;
  }

  Flight(String id, String title, int duration,
         Priority priority, Date startTime, Date creationTime,
         Date updateTime, String departingAirport, String arrivingAirport,
         String airlineCode, String flightNumber, String seatNumber,
         String tripId) {
    super(id, title, duration, priority, startTime, creationTime, updateTime);
    this.departingAirport=departingAirport;
    this.arrivingAirport=arrivingAirport;
    this.airlineCode=airlineCode;
    this.flightNumber=flightNumber;
    this.seatNumber=seatNumber;
    this.tripId=tripId;
  }
}
```

(from Trips/RoomRelations/app/src/main/java/com/commonsware/android/room/Flight.java)

The Updated DAO and Database

Since we added new entities, `TripDatabase` needs to know about them, via the `entities` property on the `@Database` annotation:

```
package com.commonsware.android.room;

import android.arch.persistence.room.Database;
import android.arch.persistence.room.Room;
import android.arch.persistence.room.RoomDatabase;
import android.content.Context;
```

```
@Database(
  entities={Trip.class, Lodging.class, Flight.class},
  version=2
)
abstract class TripDatabase extends RoomDatabase {
  abstract TripStore tripStore();

  private static final String DB_NAME="trips.db";
  private static volatile TripDatabase INSTANCE=null;

  synchronized static TripDatabase get(Context ctxt) {
    if (INSTANCE==null) {
      INSTANCE=create(ctxt, false);
    }

    return(INSTANCE);
  }

  static TripDatabase create(Context ctxt, boolean memoryOnly) {
    RoomDatabase.Builder<TripDatabase> b;

    if (memoryOnly) {
      b=Room.inMemoryDatabaseBuilder(ctxt.getApplicationContext(),
        TripDatabase.class);
    }
    else {
      b=Room.databaseBuilder(ctxt.getApplicationContext(), TripDatabase.class,
        DB_NAME);
    }

    return(b.build());
  }
}
```

(from Trips/RoomRelations/app/src/main/java/com/commonsware/android/room/TripDatabase.java)

Note that now we are still on `version=2`. Ideally, this sort of change would involve updating an existing database in-place, so as not to disturb any existing data. Room calls these "migrations", and they are covered in an upcoming chapter.

`TripStore`, our DAO, now needs methods for `Lodging` and `Flight` as well:

```
package com.commonsware.android.room;

import android.arch.persistence.room.Dao;
import android.arch.persistence.room.Delete;
import android.arch.persistence.room.Insert;
import android.arch.persistence.room.OnConflictStrategy;
import android.arch.persistence.room.Query;
import android.arch.persistence.room.Update;
import java.util.List;

@Dao
interface TripStore {
  /*
    Trip
   */

  @Query("SELECT * FROM trips ORDER BY title")
  List<Trip> selectAllTrips();

  @Query("SELECT * FROM trips WHERE id=:id")
  Trip findTripById(String id);

  @Insert
  void insert(Trip... trips);

  @Update
  void update(Trip... trips);

  @Delete
  void delete(Trip... trips);

  /*
    Lodging
   */

  @Query("SELECT * FROM lodgings WHERE tripId=:tripId")
  List<Lodging> findLodgingsForTrip(String tripId);

  @Insert
  void insert(Lodging... lodgings);

  @Update
  void update(Lodging... lodgings);

  @Delete
  void delete(Lodging... lodgings);

  /*
    Flight
   */

  @Query("SELECT * FROM flights WHERE tripId=:tripId")
  List<Flight> findFlightsForTrip(String tripId);
  @Insert
  void insert(Flight... flights);

  @Update
  void update(Flight... flights);

  @Delete
  void delete(Flight... flights);
}
```

(from Trips/RoomRelations/app/src/main/java/com/commonsware/android/room/TripStore.java)

The `Lodging` and `Flight` `@Query` methods retrieve only those for a particular `Trip`, based on the ID. There is nothing stopping us from having other `@Query` methods (e.g., searching across all `Lodging`, regardless of `Trip`), but these will suffice for now.

We could elect to have separate DAO classes for each entity, or have nested `@Dao`-annotated classes inside the entity for these sorts of methods. In those cases, `TripDatabase` would have to be augmented with additional `abstract` methods to return instances of those classes, mirroring the existing `tripStore()` method.

Self-Referential Relations for Tree Structures

With care, you can use Room for self-referential relations: an entity having a foreign key back to itself. This is most commonly seen in tree structures:

- Categories having sub-categories
- Folders having folders and items
- And so on

The `General/RoomTree` sample project demonstrates the first of those examples: a `Category` entity that has an optional parent `Category`:

```
package com.commonsware.android.room.dao;

import android.arch.persistence.room.Entity;
import android.arch.persistence.room.ForeignKey;
import android.arch.persistence.room.Ignore;
import android.arch.persistence.room.Index;
import android.arch.persistence.room.PrimaryKey;
import java.util.UUID;
import static android.arch.persistence.room.ForeignKey.CASCADE;
```

```
@Entity(
  tableName="categories",
  foreignKeys=@ForeignKey(
    entity=Category.class,
    parentColumns="id",
    childColumns="parentId",
    onDelete=CASCADE),
  indices=@Index(value="parentId"))
public class Category {
  @PrimaryKey
  public final String id;
  public final String title;
  public final String parentId;

  @Ignore
  public Category(String title) {
    this(title, null);
  }

  @Ignore
  public Category(String title, String parentId) {
    this(UUID.randomUUID().toString(), title, parentId);
  }

  public Category(String id, String title, String parentId) {
    this.id=id;
    this.title=title;
    this.parentId=parentId;
  }
}
```

(from General/RoomTree/stuff/src/main/java/com/commonsware/android/room/dao/Category.java)

Here, `Category` has a `@ForeignKey` that points back to `Category` as the `entity`, with a `parentId` column holding the `id` of the parent `Category`. `onDelete` is set to `CASCADE`, so that when a parent `Category` is deleted, its children are deleted as well.

Now we can have DAO methods that work with the `Category` tree:

```
  @Query("SELECT * FROM categories")
  List<Category> selectAllCategories();

  @Query("SELECT * FROM categories WHERE parentId IS NULL")
  Category findRootCategory();

  @Query("SELECT * FROM categories WHERE parentId=:parentId")
  List<Category> findChildCategories(String parentId);
  @Insert
  void insert(Category... categories);

  @Delete
  void delete(Category... categories);
```

(from General/RoomTree/stuff/src/main/java/com/commonsware/android/room/dao/StuffStore.java)

Using @Relation

If you have a POJO class — one that does not *directly* have the `@Entity` annotation — you can use `@Relation` to automatically retrieve entities related to… something in the POJO.

For example, in other Android ORMs, one might expect that `Category` would have methods, fields, or something to get at the parent `Category` (where there is one) or the child `Category` instances (where there are some). However, that is not supported by Room and `@Entity`, but it is supported by separate POJO classes.

To that end, we can set up a `CategoryTuple`:

```
package com.commonsware.android.room.dao;

import android.arch.persistence.room.Relation;
import java.util.List;

public class CategoryTuple {
  public final String id;
  public final String title;
  public final String parentId;

  public CategoryTuple(String id, String title, String parentId) {
    this.id=id;
    this.title=title;
    this.parentId=parentId;
  }

  @Relation(parentColumn="id", entityColumn="parentId")
  public List<Category> children;

  @Relation(parentColumn="parentId", entityColumn="id")
  public List<Category> parents;
}
```

(from General/RoomTree/stuff/src/main/java/com/commonsware/android/room/dao/CategoryTuple.java)

Here, we have two @Relation annotations. These go on *fields*, not methods, and they indicate fields that Room should fill in when a @Query returns instances of this POJO. The field type needs to be a List or Set of the related *entity*, not the POJO. Hence, children and parents are lists of Category instances, not CategoryTuple.

The two required properties on @Relation are parentColumn and entityColumn. entityColumn is the name of a column in the entity's table; parentColumn is the name of a *field* in the POJO representing the parent entity. In this case, the entity for both is Category, as we are working with a self-referential relation. In the generated code, Room is going to run a query that finds all objects whose entityColumn has the value pulled from this POJO's parentColumn field. More specifically:

- For the children field, Room will query the categories table to return all rows where the parentId column equals the id of this CategoryTuple
- For the parent field, Room will query the categories table to return all rows where the id column equals the parentId of this CategoryTuple

For a 1:N relation, Room's restriction on @Relation data types (must be List or Set) means that both the 1 side and the N side get represented by collection fields… even though one should only ever have at most one element.

If there are no matching entities (e.g., no parent for the root Category, no children for a leaf Category), the resulting field is either null or an empty collection.

But now, our DAO methods will not only set up the POJOs but all entities that are called for by the @Relation fields:

```
@Query("SELECT * FROM categories WHERE parentId IS NULL")
CategoryTuple findRootCategoryTuple();

@Query("SELECT * FROM categories WHERE parentId=:parentId")
List<CategoryTuple> findChildCategoryTuples(String parentId);
```

(from General/RoomTree/stuff/src/main/java/com/commonsware/android/room/dao/StuffStore.java)

However, this involved a lot of copying. CategoryTuple has the same fields as Category. It would not *have* to have all of those fields, of course, as a POJO need not have fields for all columns in the table. But, still, it seems to be a bit wasteful.

Another related approach is to create a "POJO" *subclass* of the entity… such as this CategoryShadow:

```java
package com.commonsware.android.room.dao;

import android.arch.persistence.room.Relation;
import java.util.List;

public class CategoryShadow extends Category {
  public CategoryShadow(String id, String title, String parentId) {
    super(id, title, parentId);
  }

  @Relation(parentColumn="id", entityColumn="parentId")
  public List<Category> children;
}
```

(from General/RoomTree/stuff/src/main/java/com/commonsware/android/room/dao/CategoryShadow.java)

Even though `CategoryShadow` inherits from `Category`, and even though `Category` is an entity, Room treats `CategoryShadow` as a POJO, and we can have `@Relation` fields, such as the `children` one shown. If you need most or all of the fields from the entity, this subclass approach involves less code duplication than does the standalone-POJO approach.

Representing No Relation

While much of this book will use `UUID` values for primary keys, plenty of other Room examples will use `int`, particularly with `autoGenerate` set to `true`, to have SQLite generate the keys.

However, this does not work well if those keys will be used as foreign key values, in cases where there may be no value for the relation.

For example, `Category` uses `String` for its `id` (created from a `UUID`), and we represented a root category by means of having `null` for its `parentId` value. That works because `String` fields can be `null`.

If, however, we used `int`, we have no way of representing the no-relation scenario. You cannot assign `null` to an `int` field in Java.

Hence, if you want to support the no-relation scenario, your foreign key field needs to allow for `null` values. If you want to use auto-generated SQLite identifiers, use `Integer`, not `int`.

Room and Migrations

When you first ship your app, you think your database schema is beautiful, a true work of art.

Then, you wake up the next morning and realize that you need to make changes to that schema.

During initial development — and for silly little book examples — you just go in and make changes to your entities, and Room will rebuild your database for you. However, it does so by dropping all of your existing tables, taking all the data with it. In development, that may not be so bad. In *production*… well, let's just say that users get a little irritated when you lose their data.

And that's where migrations come into play.

What's a Migration?

With traditional Android SQLite development, we typically use `SQLiteOpenHelper`. This utility class manages a `SQLiteDatabase` for us and addresses two key problems:

1. What happens when our app first runs on a device — or after the user has cleared our app's data — and we have no database at all?
2. What happens when we need to modify the database schema from what it was to some new structure?

`SQLiteOpenHelper` would do that by calling `onCreate()` and `onUpgrade()` callbacks, where we could implement the logic to create the tables and adjust them as the schemas change.

While `onCreate()` worked reasonably well, `onUpgrade()` rapidly grew out of control. Long-lived apps might have dozens of different schemas, evolving over time. Because users are not forced to take on app updates, our apps need to be able to transition from any prior schema to the latest-and-greatest one. This meant that `onUpgrade()` would need to identify exactly what bits of code are needed to migrate the database from the old to the new version, and this could get unwieldy.

Room addresses this somewhat through the `Migration` class. You create subclasses of `Migration` — typically as anonymous inner classes — that handle the conversion from some older schema to a newer one. You pass a bunch of `Migration` instances to Room, representing different pair-wise schema upgrade paths. Room then determines which one(s) need to be used at any point in time, to update the schema from whatever it was to whatever it needs to be.

When Do We Migrate?

On our `RoomDatabase` subclass, we have a `@Database` annotation. One of the properties is `version`. This works like the version code that we would pass into the `SQLiteOpenHelper` constructor. It is a monotonically increasing integer, with higher numbers indicating newer schemas. The `version` in the code represents the schema version that this code is expecting.

Once your app ships, any time you change your schema — mostly in the form of modifying entity classes — you need to increment that `version` and create a `Migration` that knows how to convert from the prior version to this new one.

Note that there is no requirement that you increment the `version` by 1, though that is a common convention. If using a date-based format like `YYYMMDD` (e.g., `20170627`) makes your life easier, you are welcome to do so.

But First, a Word About the Support Database Classes

So far, this book has portrayed Room as being an ORM-style bridge between your code and SQLite.

Technically, that is not accurate.

Part of what we get with Room is a series of classes and interfaces in the `android.support.persistence.db` package. These classes come from a separate artifact (`android.arch.persistence.room:support-db`) and represent an abstraction for SQLite-style database access.

We *also* get implementations of that abstraction, in the form of the "framework" classes (from `android.arch.persistence.room:support-db-impl`). Those classes use the Android standard SQLite environment. Room's artifacts pull in these support artifacts by default, and when we use `RoomDatabase.Builder` to set up our `RoomDatabase`, we are using those "framework" classes for the database access.

There are two reasons why this is important.

First, database migrations are largely outside of Room itself. Room is expecting the database to be set up with the appropriate schema. While a `RoomDatabase.Builder` can *use* `Migration` objects to migrate the database schema, Room itself is not yet ready at this point. We wind up using a `SupportSQLiteDatabase` class for modifying the schema, where this class is from that abstraction library. So, while most of Room hides you from most of SQLite-related Java code, migrations are one area where this stuff becomes more visible.

Second, just because Room uses the device implementation of SQLite by default does not mean that *you* have to use it. One of the methods on `RoomDatabase.Builder` is `openHelperFactory()`, where you supply a `SupportSQLiteOpenHelper.Factory` to use for working with the database. That, in turn, can pull in another whole set of implementations of the database abstraction. For example, you can use this approach to have Room interoperate with SQLCipher for Android, an encrypted edition of SQLite. A later chapter will explore such a library.

…And a Word About Exporting Schemas

One of the side-effects of using Room is that you do not write your own schema for the database. Room generates it, based on your entity definitions. During the ordinary course of programming, this is perfectly fine and saves you time and effort.

However, when it comes to migrations, now we have a problem. We cannot create code to migrate from an old to a new schema without knowing what those schemas are. And while schema information is baked into some code generated by Room's annotation processor, that is only for the current version of your entity classes (and, hence, your current schema), not for any historical ones.

Fortunately, Room offers something that helps a bit: exported schemas. You can teach Room's annotation processor to not only generate Java code but also generate a JSON document describing the schema. Moreover, it will do that for each schema version, saving them to version-specific JSON files. If you hold onto these files — for example, if you save them in version control – you will have a history of your schema and can use that information to write your migrations.

However, the real reason for those exported schemas is to help with testing your migrations. As a result, the JSON format is not designed for developers to read.

To set this up, in the `defaultConfig` closure of your module's `build.gradle` file, you can add the following `javaCompileOptions` closure:

```
    javaCompileOptions {
      annotationProcessorOptions {
        arguments = ["room.schemaLocation": "$projectDir/schemas".toString()]
      }
    }
```

(from Trips/RoomMigrations/app/build.gradle)

This teaches Room to save your schemas in a `schemas/` directory off of the module root directory. In principle, you could store them elsewhere by choosing a different value for the `room.schemaLocation` argument.

The next time you (re-)build your project, that directory will be created. Subdirectories with the fully-qualified class names of your `RoomDatabase` classes will go inside there, and inside each of those will be a JSON file named after your schema version (e.g., `1.json`):

```
{
  "formatVersion": 1,
  "database": {
    "version": 1,
    "identityHash": "d46bfccddeca286f2948a702a4938d56",
    "entities": [
      {
        "tableName": "trips",
        "createSql": "CREATE TABLE IF NOT EXISTS `${TABLE_NAME}` (`id` TEXT, `title` TEXT, `duration`
INTEGER NOT NULL, `priority` INTEGER, `startTime` INTEGER, `creationTime` INTEGER, `updateTime` INTEGER,
PRIMARY KEY(`id`))",
        "fields": [
          {
            "fieldPath": "id",
            "columnName": "id",
            "affinity": "TEXT"
          },
          {
            "fieldPath": "title",
            "columnName": "title",
            "affinity": "TEXT"
          },
          {
            "fieldPath": "duration",
            "columnName": "duration",
            "affinity": "INTEGER"
          },
          {
            "fieldPath": "priority",
            "columnName": "priority",
            "affinity": "INTEGER"
          },
          {
            "fieldPath": "startTime",
            "columnName": "startTime",
            "affinity": "INTEGER"
          },
          {
            "fieldPath": "creationTime",
            "columnName": "creationTime",
            "affinity": "INTEGER"
          },
          {
            "fieldPath": "updateTime",
            "columnName": "updateTime",
            "affinity": "INTEGER"
          }
        ],
        "primaryKey": {
          "columnNames": [
            "id"
          ],
          "autoGenerate": false
        },
        "indices": [],
        "foreignKeys": []
      }
    ],
    "setupQueries": [
      "CREATE TABLE IF NOT EXISTS room_master_table (id INTEGER PRIMARY KEY,identity_hash TEXT)",
      "INSERT OR REPLACE INTO room_master_table (id,identity_hash) VALUES(42,
\"d46bfccddeca286f2948a702a4938d56\")"
    ]
  }
}
```

(from Trips/RoomMigrations/app/schemas/com.commonsware.android.room.TripDatabase/1.json)

The JSON properties that will matter to you will be the `createSql` ones. There are ones that create your tables and others that create your indexes.

Writing Migrations

A `Migration` itself has only one required method: `migrate()`. You are given a `SupportSQLiteDatabase`, which resembles a `SQLiteDatabase` and allows you to execute SQL statements to modify the schema as needed.

The `Migration` constructor takes two parameters: the old schema version number and the new schema version number. Hence, the recommended pattern is to use anonymous inner classes, where you can provide the `migrate()` method to use for migrating the schema between that particular pair of schema versions.

To determine what needs to be done, you need to examine that schema JSON and determine what is different between the old and the new. Someday, we may get some tools to help with this. For now, you are largely stuck "eyeballing" the SQL. You can then craft the `ALTER TABLE` or other statements necessary to change the schema, much as you might have done in `onUpgrade()` of a `SQLiteOpenHelper`.

For example, the [`Trips/RoomMigrations`](Trips/RoomMigrations) sample project has a `FROM_1_TO_2` migration:

```
  static final Migration FROM_1_TO_2=new Migration(1,2) {
    @Override
    public void migrate(SupportSQLiteDatabase db) {
      db.execSQL("CREATE TABLE IF NOT EXISTS `lodgings` (`id` TEXT, `title` TEXT, `duration` INTEGER NOT NULL, `priority` INTEGER, `startTime` INTEGER, `creationTime` INTEGER, `updateTime` INTEGER, `address` TEXT, `tripId` TEXT, PRIMARY KEY(`id`), FOREIGN KEY(`tripId`) REFERENCES `trips`(`id`) ON UPDATE NO ACTION ON DELETE CASCADE )");
      db.execSQL("CREATE  INDEX `index_lodgings_tripId` ON `lodgings` (`tripId`)");
      db.execSQL("CREATE TABLE IF NOT EXISTS `flights` (`id` TEXT, `title` TEXT, `duration` INTEGER NOT NULL, `priority` INTEGER, `startTime` INTEGER, `creationTime` INTEGER, `updateTime` INTEGER, `departingAirport` TEXT, `arrivingAirport` TEXT, `airlineCode` TEXT, `flightNumber` TEXT, `seatNumber` TEXT, `tripId` TEXT, PRIMARY KEY(`id`), FOREIGN KEY(`tripId`) REFERENCES `trips`(`id`) ON UPDATE NO ACTION ON DELETE CASCADE )");
      db.execSQL("CREATE  INDEX `index_flights_tripId` ON `flights` (`tripId`)");
    }
  };
```

(from Trips/RoomMigrations/app/src/main/java/com/commonsware/android/room/Migrations.java)

Here, we create two tables and two indexes in `migrate()`. The SQL is mostly copied from the `2.json` file, representing the schema for version 2:

```
{
  "formatVersion": 1,
  "database": {
    "version": 2,
    "identityHash": "4ebc3fa474b72f9567fafb4658d7c9ce",
    "entities": [
      {
        "tableName": "trips",
        "createSql": "CREATE TABLE IF NOT EXISTS `${TABLE_NAME}` (`id` TEXT, `title` TEXT, `duration`
INTEGER NOT NULL, `priority` INTEGER, `startTime` INTEGER, `creationTime` INTEGER, `updateTime` INTEGER,
PRIMARY KEY(`id`))",
        "fields": [
          {
            "fieldPath": "id",
            "columnName": "id",
            "affinity": "TEXT",
            "notNull": false
          },
          {
            "fieldPath": "title",
            "columnName": "title",
            "affinity": "TEXT",
            "notNull": false
          },
          {
            "fieldPath": "duration",
            "columnName": "duration",
            "affinity": "INTEGER",
            "notNull": true
          },
          {
            "fieldPath": "priority",
            "columnName": "priority",
            "affinity": "INTEGER",
            "notNull": false
          },
          {
            "fieldPath": "startTime",
            "columnName": "startTime",
            "affinity": "INTEGER",
            "notNull": false
          },
          {
            "fieldPath": "creationTime",
            "columnName": "creationTime",
            "affinity": "INTEGER",
            "notNull": false
          },
          {
            "fieldPath": "updateTime",
            "columnName": "updateTime",
            "affinity": "INTEGER",
            "notNull": false
          }
        ],
        "primaryKey": {
          "columnNames": [
            "id"
          ],
          "autoGenerate": false
        },
        "indices": [],
        "foreignKeys": []
      },
      {
        "tableName": "lodgings",
        "createSql": "CREATE TABLE IF NOT EXISTS `${TABLE_NAME}` (`id` TEXT, `title` TEXT, `duration`
INTEGER NOT NULL, `priority` INTEGER, `startTime` INTEGER, `creationTime` INTEGER, `updateTime` INTEGER,
`address` TEXT, `tripId` TEXT, PRIMARY KEY(`id`), FOREIGN KEY(`tripId`) REFERENCES `trips`(`id`) ON UPDATE
NO ACTION ON DELETE CASCADE )",
        "fields": [
          {
            "fieldPath": "id",
            "columnName": "id",
            "affinity": "TEXT",
            "notNull": false
          },
          {
            "fieldPath": "title",
            "columnName": "title",
            "affinity": "TEXT",
            "notNull": false
          },
          {
            "fieldPath": "duration",
            "columnName": "duration",
            "affinity": "INTEGER",
            "notNull": true
          },
          {
            "fieldPath": "priority",
            "columnName": "priority",
            "affinity": "INTEGER",
            "notNull": false
          },
          {
            "fieldPath": "startTime",
            "columnName": "startTime",
            "affinity": "INTEGER",
            "notNull": false
          },
          {
            "fieldPath": "creationTime",
            "columnName": "creationTime",
            "affinity": "INTEGER",
            "notNull": false
          },
          {
            "fieldPath": "updateTime",
            "columnName": "updateTime",
            "affinity": "INTEGER",
            "notNull": false
          },
          {
            "fieldPath": "address",
            "columnName": "address",
            "affinity": "TEXT",
            "notNull": false
          },
          {
            "fieldPath": "tripId",
            "columnName": "tripId",
            "affinity": "TEXT",
            "notNull": false
          }
        ],
        "primaryKey": {
          "columnNames": [
            "id"
          ],
          "autoGenerate": false
        },
        "indices": [
          {
            "name": "index_lodgings_tripId",
            "unique": false,
            "columnNames": [
              "tripId"
            ],
            "createSql": "CREATE  INDEX `index_lodgings_tripId` ON `${TABLE_NAME}` (`tripId`)"
          }
        ],
        "foreignKeys": [
          {
            "table": "trips",
            "onDelete": "CASCADE",
            "onUpdate": "NO ACTION",
            "columns": [
              "tripId"
            ],
            "referencedColumns": [
              "id"
            ]
          }
        ]
      },
      {
        "tableName": "flights",
        "createSql": "CREATE TABLE IF NOT EXISTS `${TABLE_NAME}` (`id` TEXT, `title` TEXT, `duration`
INTEGER NOT NULL, `priority` INTEGER, `startTime` INTEGER, `creationTime` INTEGER, `updateTime` INTEGER,
`departingAirport` TEXT, `arrivingAirport` TEXT, `airlineCode` TEXT, `flightNumber` TEXT, `seatNumber`
TEXT, `tripId` TEXT, PRIMARY KEY(`id`), FOREIGN KEY(`tripId`) REFERENCES `trips`(`id`) ON UPDATE NO ACTION
ON DELETE CASCADE )",
        "fields": [
          {
            "fieldPath": "id",
            "columnName": "id",
            "affinity": "TEXT",
            "notNull": false
          },
          {
            "fieldPath": "title",
            "columnName": "title",
            "affinity": "TEXT",
            "notNull": false
          },
          {
            "fieldPath": "duration",
            "columnName": "duration",
            "affinity": "INTEGER",
            "notNull": true
          },
          {
            "fieldPath": "priority",
            "columnName": "priority",
            "affinity": "INTEGER",
            "notNull": false
          },
          {
            "fieldPath": "startTime",
            "columnName": "startTime",
            "affinity": "INTEGER",
            "notNull": false
          },
          {
            "fieldPath": "creationTime",
            "columnName": "creationTime",
            "affinity": "INTEGER",
            "notNull": false
          },
          {
            "fieldPath": "updateTime",
            "columnName": "updateTime",
            "affinity": "INTEGER",
            "notNull": false
          },
          {
            "fieldPath": "departingAirport",
            "columnName": "departingAirport",
            "affinity": "TEXT",
            "notNull": false
          },
          {
            "fieldPath": "arrivingAirport",
            "columnName": "arrivingAirport",
            "affinity": "TEXT",
            "notNull": false
          },
          {
            "fieldPath": "airlineCode",
            "columnName": "airlineCode",
            "affinity": "TEXT",
            "notNull": false
          },
          {
            "fieldPath": "flightNumber",
            "columnName": "flightNumber",
            "affinity": "TEXT",
            "notNull": false
          },
          {
            "fieldPath": "seatNumber",
            "columnName": "seatNumber",
            "affinity": "TEXT",
            "notNull": false
          },
          {
            "fieldPath": "tripId",
            "columnName": "tripId",
            "affinity": "TEXT",
            "notNull": false
          }
        ],
        "primaryKey": {
          "columnNames": [
            "id"
          ],
          "autoGenerate": false
        },
        "indices": [
          {
            "name": "index_flights_tripId",
            "unique": false,
            "columnNames": [
              "tripId"
            ],
            "createSql": "CREATE  INDEX `index_flights_tripId` ON `${TABLE_NAME}` (`tripId`)"
          }
        ],
        "foreignKeys": [
          {
            "table": "trips",
            "onDelete": "CASCADE",
            "onUpdate": "NO ACTION",
            "columns": [
              "tripId"
            ],
            "referencedColumns": [
              "id"
            ]
          }
        ]
      }
    ],
    "setupQueries": [
      "CREATE TABLE IF NOT EXISTS room_master_table (id INTEGER PRIMARY KEY,identity_hash TEXT)",
      "INSERT OR REPLACE INTO room_master_table (id,identity_hash) VALUES(42,
\"4ebc3fa474b72f9567fafb4658d7c9ce\")"
    ]
  }
}
```

(from Trips/RoomMigrations/app/schemas/com.commonsware.android.room.TripDatabase/2.json)

In the JSON, the `createSql` properties have the table name as a template-style macro (`${TABLE_NAME}`), which you will need to replace with the actual table name. The backticks are supported in SQLite as they are in MySQL, and since they cause no harm here, usually it is simpler just to leave them in there.

Employing Migrations

Simply creating a `Migration` as a `static` field somewhere is necessary but not sufficient to have Room know about performing the migration. Instead, you need to use the `addMigrations()` method on `RoomDatabase.Builder` to teach Room about your `Migration` objects. `addMigrations()` accepts a varargs, and so you can pass in one or several `Migration` objects as needed.

```
package com.commonsware.android.room;

import android.arch.persistence.room.Database;
import android.arch.persistence.room.Room;
import android.arch.persistence.room.RoomDatabase;
import android.content.Context;

@Database(
  entities={Trip.class, Lodging.class, Flight.class},
  version=2
)
abstract class TripDatabase extends RoomDatabase {
  abstract TripStore tripStore();

  private static final String DB_NAME="trips.db";
  private static volatile TripDatabase INSTANCE=null;

  synchronized static TripDatabase get(Context ctxt) {
    if (INSTANCE==null) {
      INSTANCE=create(ctxt, false);
    }

    return(INSTANCE);
  }

  static TripDatabase create(Context ctxt, boolean memoryOnly) {
    return(create(ctxt, DB_NAME, memoryOnly));
  }

  static TripDatabase create(Context ctxt, String name, boolean memoryOnly) {
    RoomDatabase.Builder<TripDatabase> b;

    if (memoryOnly) {
      b=Room.inMemoryDatabaseBuilder(ctxt.getApplicationContext(),
        TripDatabase.class);
    }
    else {
      b=Room.databaseBuilder(ctxt.getApplicationContext(), TripDatabase.class,
        name);
    }

    return(b.addMigrations(Migrations.FROM_1_TO_2).build());
  }
}
```

(from Trips/RoomMigrations/app/src/main/java/com/commonsware/android/room/TripDatabase.java)

Here, we teach the `RoomDatabase.Builder` about the `FROM_1_TO_2` `Migration`. In this sample project, the migrations are implemented in a separate `Migrations` class, though you are welcome to have them directly in your `RoomDatabase` class or wherever makes sense for you.

How Room Applies Migrations

When you create your `RoomDatabase` instance via the `Migration`-enhanced `Builder`, Room will use `SQLiteOpenHelper` semantics to see if the schema version in the existing database is older than the schema version that you declared in your `@Database` annotation. If it is, Room will try to find a suitable `Migration` to use, falling back to dropping all of your tables and rebuilding them from scratch, as happens during ordinary development.

Much of the time, the schema will jump from one version to the next. If you are using a simple numbering scheme starting at 1, the schema will then move to 2, then 3, then 4, and so on, for a given device. Hence, your primary `Migration` objects will be ones that implement these incremental migrations.

However, it may be that for some device you need to skip a schema version, such as moving from version 1 to version 3. Room is smart enough to find a chain of `Migration` objects to use, and so if you have `Migration` objects for each incremental schema change, Room can handle any combination of changes. For example, to go from 1 to 3, Room might first use your `(1,2)` migration, then the `(2,3)` migration.

Sometimes, though, this can lead to unnecessary work. Suppose in schema version 2, you created a bunch of new tables and stuff… then reverted those changes in schema version 3. By using the incremental migrations, Room will create those tables and then turn around and drop them right away.

However, all else being equal, Room will try to use the shortest possible chain. Hence, you can create additional `Migration` objects where appropriate to streamline particular upgrades. You could create a `(1,3)` migration that bypasses the obsolete schema version 2, for example. This is optional but may prove useful from time to time.

Testing Migrations

It would be nice if your migrations worked. Users, in particular, appreciate working code… or, perhaps more correctly, get rather angry with non-working code.

Hence, you might want to test the migrations.

This gets a bit tricky, though. The code-generated Room classes are expecting the latest-and-greatest schema version, so you cannot use your DAO for testing older schemas. Besides, `RoomDatabase.Builder` wants to set up your database with that latest-and-greatest schema automatically.

Fortunately, Room ships with some testing code to help you test your schemas in isolation… though you bypass most of Room to do that.

Adding the Artifact

This testing code is in a separate `android.arch.persistence.room:testing` artifact, one that you can add via `androidTestCompile` to put in your instrumentation tests but leave out of your production code:

```
dependencies {
  compile "com.android.support:recyclerview-v7:26.0.0"
  compile "android.arch.persistence.room:runtime:1.0.0-alpha8"
  annotationProcessor "android.arch.persistence.room:compiler:1.0.0-alpha8"
  androidTestCompile "com.android.support:support-annotations:26.0.0"
  androidTestCompile 'com.android.support.test:rules:0.5'
  androidTestCompile "android.arch.persistence.room:testing:1.0.0-alpha8"
}
```

(from Trips/RoomMigrations/app/build.gradle)

Adding the Schemas

Remember those exported schemas? While we used them for helping us write the migrations, their primary use is for this testing support code.

By default, those schemas are stored outside of anything that goes into your app. After all, you do not need those JSON files cluttering up your production builds. However, this also means that those schemas are not available to your test code, by default.

However, we can fix that, by adding those schemas to the `assets/` used in the `androidTest` source set, by having this closure in your `android` closure of your module's `build.gradle` file:

```
sourceSets {
  androidTest.assets.srcDirs += files("$projectDir/schemas".toString())
}
```

(from Trips/RoomMigrations/app/build.gradle)

Here, `"$projectDir/schemas".toString()` is the same value that we used for the `room.schemaLocation` annotation processor argument. This snippet tells Gradle to include the contents of that `schemas/` directory as part of our `assets/`.

The result is that our instrumentation test APK will have those directories named after our `RoomDatabase` classes (e.g., `com.commonsware.android.room.TripDatabase/`) in the root of `assets/`. If you have code that uses `assets/`, make sure that you are taking steps to ignore these extra directories.

Creating and Using a MigrationTestHelper

The testing support comes in the form of a `MigrationTestHelper` that you can employ in your instrumentation tests.

Adding the Rule

`MigrationTestHelper` is a JUnit4 rule, which you add to your test case class via the `@Rule` annotation:

```
@Rule
public MigrationTestHelper helper;
```

(from Trips/RoomMigrations/app/src/androidTest/java/com/commonsware/android/room/MigrationTests.java)

Setting Up the Helper

You then need to create an instance of the `MigrationTestHelper`, such as in a `@Before`-annotated method:

```
@Before
public void setUp() {
  helper=new MigrationTestHelper(InstrumentationRegistry.getInstrumentation(),
    TripDatabase.class.getCanonicalName());
}
```

(from Trips/RoomMigrations/app/src/androidTest/java/com/commonsware/android/room/MigrationTests.java)

`MigrationTestHelper` takes two parameters, both of which are a bit unusual.

First, it takes an `Instrumentation` object. We use those in our test code, but it is rare that we pass them as a parameter. You get your `Instrumentation` usually by calling `getInstrumentation()` on the `InstrumentationRegistry`.

Then, it takes what appears to be the fully-qualified class name of the `RoomDatabase` whose migrations we wish to test. Technically speaking, this is actually the relative path, inside of `assets/`, where the schema JSON files are for this particular `RoomDatabase`. Given the above configuration, each database's schemas are put into a directory named after the fully-qualified class name of the `RoomDatabase`, which is why this works. However, if you change the configuration to put the schemas somewhere else in `assets/`, you would need to modify this parameter to match.

Creating a Database for a Schema Version

There are two main methods on `MigrationTestHelper` that we will use in testing. One is `createDatabase()`. This creates the database, as a specific database file, for a specific schema version… including any of our historical ones found in those schema JSON files. Here, we ask the helper to create a database named `DB_NAME` for schema version 1:

```
    SupportSQLiteDatabase db=helper.createDatabase(DB_NAME, 1);
```

(from Trips/RoomMigrations/app/src/androidTest/java/com/commonsware/android/room/MigrationTests.java)

That `SupportSQLiteDatabase` object has an API reminiscent of a trimmed-down `SQLiteDatabase`. `query()` replaces `rawQuery()` and is used for executing arbitrary SQL `SELECT` statements. We also have `execSQL()`, `insert()`, `update()`, and `delete()`, for other operations.

As part of testing a migration, you will need to add some sample data to the database, using whatever schema you asked to be used, so that you can confirm that the migration worked as expected and did not wreck the existing data. This code will not be very Room-ish, but more like classic SQLite Android programming:

```
    SupportSQLiteDatabase db=helper.createDatabase(DB_NAME, 1);

    db.execSQL("INSERT INTO trips (title, duration) VALUES (NULL, 0)");

    final Cursor firstResults=db.query("SELECT COUNT(*) FROM trips");

    assertEquals(1, firstResults.getCount());
    firstResults.moveToFirst();
    assertEquals(1, firstResults.getInt(0));

    firstResults.close();
    db.close();
```

(from Trips/RoomMigrations/app/src/androidTest/java/com/commonsware/android/room/MigrationTests.java)

Testing a Migration

The other method of note on `MigrationTestHelper` is `runMigrationsAndValidate()`. After you have set up a database in its starting conditions via `createDatabase()` and CRUD operations, `runMigrationsAndValidate()` will migrate that database from its original schema version to the one that you specify:

```
    db=helper.runMigrationsAndValidate(DB_NAME, 2, true,
      Migrations.FROM_1_TO_2);
```

(from Trips/RoomMigrations/app/src/androidTest/java/com/commonsware/android/room/MigrationTests.java)

You need to supply the same database name (`DB_NAME`), a higher schema version (`2`), and the specific `Migration` that you want to use (`Migration.FROM_1_TO_2`).

Not only does this method perform the migration, but it validates the resulting schema against what the entities have set up for that schema version, based on the schema JSON files. If there is something wrong — your migration forgot a newly-added column, for example — your test will fail with an assertion violation. The `true` parameter shown above determines whether this schema validation will checked for un-dropped tables. `true` means that if you have unnecessary tables in the database, the test fails; `false` means that unnecessary tables are fine and will be ignored.

However, all `MigrationTestHelper` can do is confirm that you set up the new schema properly. It cannot determine whether the data is any good after the migration. That you would need to test yourself. In many cases, there is little to test, particularly if you are just setting up empty tables as we are doing in this migration. However, if you had a complex table change, perhaps requiring a temp table and statements like `INSERT INTO ... SELECT FROM ...`, you could write test code that confirms the data is OK. However, you cannot use the Room DAO for this either; instead, you will use the `SupportSQLiteDatabase` and work with the tables “the old-fashioned way”, using `query()` and `Cursor` and such.

Migrating Under Protest

In the `1.0.0-alpha8` edition of Room, subtle changes in your object model have unexpected impacts on your database storage.

In particular, changing the order of fields as they appear in your entity forces you to have a do-nothing migration if you want to keep your data. This appears to be because:

- Room creates columns in the table in the same order as the fields appear in the entity, so changing the field order changes the column order
- Room's logic for detecting if a migration is needed does not take into account the fact that the order of fields in a table does not matter

With luck, this will be addressed in a future update to Room.

Similarly, pay close attention to the release notes for Room. Updating your Room implementation may require you to implement or modify a migration. For example, upgrading from `1.0.0-alpha3` to `1.0.0-alpha8` changed the nature of columns generated from `int` fields — these are now `NOT NULL`, where formerly they allowed null values. However, since this changes the schema, you will now need to take this account in the migration, altering those columns to be `NOT NULL`. With luck, once `1.0.0` ships in final form, breaking schema changes will be relegated to major version releases.

Securing Your Room

Room, by default, works with the device's stock copy of SQLite. This is fine, as far as it goes. However, from a security standpoint, SQLite stores its data unencrypted. Many apps should be considering encrypting their data "at rest", when it is stored in a database, to protect their users.

Fortunately, as noted in the last chapter, Room supports a pluggable SQLite implementation, and so we can plug in a SQLite edition that supports encryption, such as SQLCipher for Android. This chapter will outline how to do this.

Meet the Players

There are two pieces to the encrypted-database puzzle: a SQLite implementation with encryption capability, and the "glue code" that allows Room to work with that SQLite implementation.

SQLCipher for Android

Since SQLite is public domain, it is easy for people to grab the source code and hack on it. SQLite also offers an extension system, making it relatively easy for developers to add functionality with a minimal number of changes to SQLite's core code. As a result, a few encryption options for SQLite have been published.

One of these is SQLCipher, whose development is overseen by Zetitec. This offers transparent AES-256 encryption of everything in the database: data, schema, etc.

With the help of the Guardian Project, Zetitec released SQLCipher for Android. This combines a pre-compiled version of SQLite with Java classes that mimic an old edition of Android's native SQLite classes (e.g., `SQLiteOpenHelper`). SQLCipher for

Android is open source, and if you can live with the increase in app size due to the native binaries, it is an effective solution.

However, it knows nothing about Room.

CWAC-SafeRoom

To fill that gap, the author of this book has released CWAC-SafeRoom. This is an implementation of Room's pluggable database API to bridge between Room and SQLCipher for Android. Using SQLCipher for Android then becomes mostly a matter of a single method call on the `RoomDatabase.Builder` to use the CWAC-SafeRoom code — everything else works as normal.

That being said, at the time of this writing, the latest release of Room is `1.0.0-alpha3`, and CWAC-SafeRoom is `0.0.1`. These are early days for both libraries, and so changes may occur either at the Room or the CWAC-SafeRoom level.

Using CWAC-SafeRoom

Fortunately, using CWAC-SafeRoom is fairly straightforward, at least in terms of the Java code.

The fact that SQLCipher for Android makes use of native libraries will make your APK substantially larger, though using ABI filters and splits can help manage that. However, those concerns would be the same for any use of SQLCipher for Android and are not unique to CWAC-SafeRoom.

Adding the Dependency

As with all the CWAC libraries, you get CWAC-SafeRoom from the CWAC artifact repository:

```
repositories {
    maven {
        url "https://s3.amazonaws.com/repo.commonsware.com"
    }
}
```

(or use `http://repo.commonsware.com` if you cannot use SSL for your builds, for some scary reason)

Then, it is merely a matter of adding a dependency on the `com.commonsware.cwac:saferoom` artifact. At the time of this writing, the artifact is only available as a `0.0.1` release:

```
compile 'com.commonsware.cwac:saferoom:0.0.1'
```

Using CWAC-SafeRoom

When you use Room, you use `Room.databaseBuilder()` or `Room.inMemoryDatabaseBuilder()` to get a `RoomDatabase.Builder`. After configuring that object, you call `build()` to get an instance of your custom subclass of `RoomDatabase`, whichever one that you supplied as a Java class object to the `Room.databaseBuilder()` or `Room.inMemoryDatabaseBuilder()` method.

To use SafeRoom, on the `RoomDatabase.Builder`, before calling `build()`:

- Create an instance of `com.commonsware.cwac.saferoom.SafeHelperFactory`, passing in the passphrase to use
- Pass that `SafeHelperFactory` to the `RoomDatabase.Builder` via the `openHelperFactory()` method

```
// EditText passphraseField;
SafeHelperFactory factory=SafeHelperFactory.fromUser(passphraseField.getText());

StuffDatabase db=Room.databaseBuilder(ctxt, StuffDatabase.class, DB_NAME)
  .openHelperFactory(factory)
  .build();
```

Passphrase Management

A cardinal rule of passphrases in Java is: do not hold them in `String` objects. You have no means of clearing those from memory, as a `String` is an immutable value.

The `SafeHelperFactory` constructor takes a `char` array for the passphrase. If you are getting the passphrase from the user via an `EditText` widget, use the `fromUser()` factory method instead, supplying the `Editable` that you get from `getText()` on the `EditText`.

SafeRoom will zero out the `char` array once the database is opened. If you use `fromUser()`, SafeRoom will also clear the contents of the `Editable`.

More to Come!

More material on CWAC-SafeRoom and the use of SQLCipher for Android will be added to this book in the future, as Room, CWAC-SafeRoom, and this book all mature.

Lifecycle Components and ViewModels

Lifecycles and Owners

Programmers, in any environment, often encounter one or more topics that inspire the five stages of grief. It might be related to threads, to security, to UI implementation (e.g., how to deal with resizeable windows).

Android developers experience this sort of grief on all those topics.

Another one that triggers this sort of grief is the concept of lifecycles. On the surface, the concept seems unremarkable: objects are in use for a time and then become discarded, and along the way we receive callbacks regarding their state. However, dealing with the ramifications of those lifecycles — such as handling configuration changes, like screen rotation — vex even seasoned Android developers.

Part of the Architecture Components is a series of classes designed to help you deal with lifecycles in a more consistent fashion.

A Tale of Terminology

The Architecture Components have very specific definitions for certain terms, and these definitions affect the classes that we wind up using.

Lifecycle

A lifecycle is a series of states that an object can be in. Hence, a trivial lifecycle simply has "alive" and "dead" or similar states.

The eponymous `Lifecycle` class, however, models a *specific* lifecycle, that of activities and fragments.

Lifecycle Owner

A lifecycle owner is simply something that goes through a lifecycle. If the lifecycle is the state, the lifecycle owner is what has the trigger events for navigating through the state machine.

A `LifecycleOwner` is a Java interface, with a `getLifecycle()` method, that returns the `Lifecycle` for a given owner. As we will see, various classes already implement `LifecycleOwner`, and adding it to something else is not especially difficult.

Lifecycle Observers

A lifecycle observer is something that is notified about the change in state of some lifecycle. It finds out about those trigger events and the movement of the lifecycle from state to state.

A `LifecycleObserver` is another Java interface, one that mostly serves as a marker, with no required methods. However, a `LifecycleObserver` can have one or more methods marked with an `@OnLifecycleEvent` annotation, and those methods will be called when the `Lifecycle` enters a certain state.

Adding the Lifecycle Components

As with Room, the lifecycle-related libraries are housed in Google's Maven repository, and you need to teach Gradle where that is. The convention is to add the repository URL in the `allprojects` closure in the project's root `build.gradle` file:

```
allprojects {
    repositories {
        jcenter()
        maven { url 'https://maven.google.com' }
    }
}
```

(from General/Lifecycle/build.gradle)

Then, you need a runtime dependency and an annotation processor, once again akin to how Room is set up:

```
dependencies {
    compile 'com.android.support:recyclerview-v7:26.0.0'
    compile 'android.arch.lifecycle:runtime:1.0.0-alpha8'
    annotationProcessor 'android.arch.lifecycle:compiler:1.0.0-alpha8'
}
```

(from General/Lifecycle/app/build.gradle)

Getting a Lifecycle

Everything dealing with `Lifecycle` comes down to a `LifecycleOwner`. You have several possibilities of where to get one of those.

…From a LifecycleActivity or LifecycleFragment

The pre-release versions of the lifecycle artifacts include `LifecycleActivity` and `LifecycleFragment` classes. These extend `FragmentActivity` and `Fragment` from the Android Support Library, respectively. Hence, if you were already using those base classes, you can swap in `LifecycleActivity` and `LifecycleFragment` and be set up with access to `Lifecycle` instances.

However, there are two problems:

1. Google has indicated that these classes will be deprecated when the Architecture Components ship a 1.0.0 final release
2. Most likely, you are not using `FragmentActivity` directly

…From an AppCompatActivity

Perhaps you are using the `appcompat-v7` artifact. In that case, you are inheriting from `AppCompatActivity` instead of `FragmentActivity` or `Activity`.

The good news is that sometime after the Architecture Components ship a 1.0.0 final release, there should be an update to `appcompat-v7` that makes `AppCompatActivity` a `LifecycleOwner`.

However, as of the time of this writing, that has not happened yet.

What you can do in the meantime is create your own `AppCompatLifecycleActivity`:

```
public class AppCompatLifecycleActivity extends AppCompatActivity
  implements LifecycleRegistryOwner {
  private LifecycleRegistry registry=new LifecycleRegistry(this);

  @Override
  public LifecycleRegistry getLifecycle() {
    return(registry);
  }
}
```

`Lifecycle` itself is an `abstract` class. The concrete implementation of it is `LifecycleRegistry`. Normally, you do not need to worry about this detail, as most of your code will just work with the `Lifecycle` class. However, here, we need a concrete implementation, and typically you will use `LifecycleRegistry` for that.

`LifecycleRegistryOwner` extends `LifecycleOwner`. The lifecycle artifact knows to look for activities and fragments that implement `LifecycleRegistryOwner` and knows to forward callbacks like `onCreate()` and `onPause()` to the `Lifecycle`.

Now, you can use the combination of `AppCompatLifecycleActivity` and `LifecycleFragment` until such time as `appcompat-v7` is more formally integrated with the Architecture Components.

Note, though, that this may not work, as is covered in the next section.

…From an Activity or Fragment

Perhaps you are using the classic `Activity` and `Fragment` classes. Those will never directly implement `LifecycleOwner`, as framework classes cannot depend upon libraries.

In theory, you will need to have your own `Activity` and `Fragment` base classes that implement `LifecycleOwner`, akin to the `AppCompatLifecycleActivity` shown above:

```
public class SimpleLifecycleActivity extends Activity
  implements LifecycleRegistryOwner {
  private LifecycleRegistry registry=new LifecycleRegistry(this);

  @Override
  public LifecycleRegistry getLifecycle() {
    return(registry);
  }
}
```

According to the documentation, using `LifecycleRegistryOwner` on an activity or fragment will cause the standard Android lifecycle events to be forwarded to the `Lifecycle` automatically.

Unfortunately, this does not work, due to a bug. This means that we need to handle this in a more complex fashion, outlined in the next section.

…From Anything Else

In principle, you could have other objects that are themselves tied into the activity and fragment lifecycle. After all, the backport of fragments in the Android Support Library are just that sort of "other objects". It so happens that Google takes care of managing that backport. However, you might find other objects that, for whatever reason, are similar in concept to the fragments backport and therefore should be *suppliers* of lifecycle events.

And, as noted earlier, due to bugs, we have to treat regular activities and fragments as "other objects".

In that case, you can implement `LifecycleOwner` on those classes. However, you will *also* need to call `handleLifecycleEvent()` method on the `LifecycleRegistry` at appropriate points.

For example, here is a `SimpleLifecycleActivity` that handles the standard activity lifecycle events, forwarding them to the `LifecycleRegistry`:

```
package com.commonsware.android.lifecycle;

import android.app.Activity;
import android.arch.lifecycle.Lifecycle;
import android.arch.lifecycle.LifecycleOwner;
import android.arch.lifecycle.LifecycleRegistry;
import android.os.Bundle;
import android.support.annotation.Nullable;

public class SimpleLifecycleActivity extends Activity
  implements LifecycleOwner {
  private LifecycleRegistry registry=new LifecycleRegistry(this);

  @Override
  public Lifecycle getLifecycle() {
    return(registry);
  }

  @Override
  protected void onCreate(@Nullable Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    registry.handleLifecycleEvent(Lifecycle.Event.ON_CREATE);
  }

  @Override
  protected void onStart() {
    super.onStart();

    registry.handleLifecycleEvent(Lifecycle.Event.ON_START);
  }

  @Override
  protected void onResume() {
    super.onResume();

    registry.handleLifecycleEvent(Lifecycle.Event.ON_RESUME);
  }

  @Override
  protected void onPause() {
    super.onPause();

    registry.handleLifecycleEvent(Lifecycle.Event.ON_PAUSE);
  }

  @Override
  protected void onStop() {
    super.onStop();

    registry.handleLifecycleEvent(Lifecycle.Event.ON_STOP);
  }

  @Override
  protected void onDestroy() {
    super.onDestroy();

    registry.handleLifecycleEvent(Lifecycle.Event.ON_DESTROY);
  }
}
```

(from General/Lifecycle/app/src/main/java/com/commonsware/android/lifecycle/SimpleLifecycleActivity.java)

Observing a Lifecycle

To observe the events associated with a `Lifecycle`, you create a Java class that implements `LifecycleObserver`. As noted above, `LifecycleObserver` is purely a marker interface — there are no specific methods to override. Instead, you annotate methods with `@OnLifecycleEvent`, and they will be called when the identified event occurs.

So, for example, here is an observer that passes all events to a `RecyclerView.Adapter` named `EventLogAdapter`:

```
static class LObserver implements LifecycleObserver {
  private final EventLogAdapter adapter;

  LObserver(EventLogAdapter adapter) {
    this.adapter=adapter;
  }

  @OnLifecycleEvent(Lifecycle.Event.ON_CREATE)
  void created() {
    adapter.add("ON_CREATE");
  }

  @OnLifecycleEvent(Lifecycle.Event.ON_START)
  void started() {
    adapter.add("ON_START");
  }

  @OnLifecycleEvent(Lifecycle.Event.ON_RESUME)
  void resumed() {
    adapter.add("ON_RESUME");
  }

  @OnLifecycleEvent(Lifecycle.Event.ON_PAUSE)
  void paused() {
    adapter.add("ON_PAUSE");
  }

  @OnLifecycleEvent(Lifecycle.Event.ON_STOP)
  void stopped() {
    adapter.add("ON_STOP");
  }

  @OnLifecycleEvent(Lifecycle.Event.ON_DESTROY)
  void destroyed() {
    adapter.add("ON_DESTROY");
  }
}
```

(from General/Lifecycle/app/src/main/java/com/commonsware/android/lifecycle/MainActivity.java)

Note:

- There is also a `Lifecycle.Event.ON_ANY` event that you can request; this triggers your method to be called for any lifecycle event… though you have no way of knowing what event it was
- A single method can only have one `@OnLifecycleEvent` annotation, and that annotation accepts only a single `Lifecycle.Event` value (not a list)

Then, you can register the observer, and it will start being called for the various events:

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.main);

  setTitle(getString(R.string.title, hashCode()));

  RecyclerView rv=findViewById(R.id.transcript);

  adapter=new EventLogAdapter(getLastNonConfigurationInstance());
  rv.setAdapter(adapter);

  getLifecycle().addObserver(new LObserver(adapter));
}
```

(from General/Lifecycle/app/src/main/java/com/commonsware/android/lifecycle/MainActivity.java)

All of this code is from the `General/Lifecycle` sample project, which shows you the events in a `RecyclerView` as they come in. The `MainActivity` handles configuration changes via `onRetainNonConfigurationInstance()`, so you can see the lifecycle events across a configuration change. Through an overflow menu item, you can kick off another instance of `MainActivity`, then press BACK to see the flow of lifecycle events as the original instance comes and goes from the foreground.

So, What's the Point of This?

On the surface, this all seems fairly silly. One could just as easily override the lifecycle methods on `MainActivity` and log directly to the `RecyclerView`, bypassing all this `Lifecycle` and `LifecycleOwner` stuff.

Most developers will not be creating their own `LifecycleObserver` classes, though anyone can, as the sample app demonstrates. Instead, developers will tend to use observers created by others. Most notable among those is `LiveData` from the Architecture Components, and the subject of the next chapter.

LiveData

`Lifecycle`, `LifecycleOwner`, and related classes mostly exist to provide the foundation for `LiveData`. `LiveData` is the next generation of various Android asynchronous solutions, such as `AsyncTask` and the `Loader` framework. `LiveData`, in particular, is modeled somewhat after RxJava, a popular reactive programming library.

All of this is to set up ways for you to be able to observe changes to data without having to worry as much about activity and fragment lifecycles… though, as it turns out, you cannot escape them entirely.

Observables Are the New Black

The observer pattern in software design has been around for decades. Yet, it has caught fire in the past few years, repackaged as "reactive programming". Reactive programming visualizes an app as a set of streams of data changes, whether from the user (e.g., UI widget interactions), from a server (e.g., updates to data from a sync operation), or from something else (e.g., GPS fixes). Developers set up observers to respond ("react") to these data changes and apply updates to the UI.

The centerpiece for reactive programming in Android is RxJava, typically combined with RxAndroid. RxJava provides the basic framework for observing streams of data changes, with RxAndroid primarily providing ways to route results of observations to the main application thread. This book is not going to go into details of how you use RxJava/RxAndroid in general — for that, see *The Busy Coder's Guide to Android Development* or other books.

One problem with RxJava, though, is that "it is difficult to get your head wrapped around it". Reactive programming works great in platforms that implemented reactive programming from the outset. Reactive programming is more difficult to bolt onto an existing platform, both from a technical standpoint and from a documentation standpoint. RxJava is the sort of technology that is easy to illustrate in "hello, world"-level examples but gets difficult to explain for more practical scenarios. In part, that is because RxJava is extremely flexible, and with great flexibility comes great need for great documentation… which RxJava historically lacked.

`LiveData` is designed to be a much lighter-weight approach to reactive programming, designed to do one thing (deliver asynchronous data changes regardless of lifecycle events) and do it reasonably well.

Yet More Terminology

First, let's review some new and exciting terms that we need to understand in order to use `LiveData`.

LiveData

`LiveData` itself is a source of data, both for a point in time and (via an observer) for changes to that data over time. Something will create and hand you a `LiveData` object, where the work to get that data and update it over time is handled by some background thread coming from the `LiveData` supplier.

Observer

In principle, you can call `getValue()` on a `LiveData` to get the current value for whatever stream of data the `LiveData` is tracking. In practice, this will not be especially common.

Instead, you will register an `Observer` with the `LiveData`, usually via an `observe()` method. Your `Observer` will be called with `onChanged()` when:

- You start observing and there is already data in the `LiveData`, and
- When the `LiveData` finds out about a change in the data

Your `onChanged()` method is given the data (a `Location`, a `SensorEvent`, a Room entity, whatever) on the main application thread, with an eye towards you using it to update the UI by one means or another.

Active State

If a `LiveData` was instantiated in a forest, and nobody was there to observe data changes, does the `LiveData` really exist?

The answer is: yes, but it hopefully is not consuming any resources.

A `LiveData` implementation will be called with `onActive()` when it receives its first active observer. Here, "active" means that, if the observer is tied to a `LifecycleOwner`, the lifecycle is in the started or resumed state. Conversely, the `LiveData` will be called with `onInactive()` once it no longer has any active observers, either because all observers have been unregistered or none of them are active, as their lifecycles are all paused, stopped, or destroyed.

The idea is that a `LiveData` would only start consuming significant system resources — such as requesting GPS fixes — when there are active observers, releasing those resources when there are no more active observers. This works in many cases, though there are some that will require more finesse. For example, given that the GPS radio takes some time before it starts generating GPS fixes, a `LiveData` for GPS might want to wait some amount of time after losing its last active observer before releasing the GPS radio, in case a new observer pops up quickly, to avoid delays in getting those GPS fixes.

Implementing LiveData

With that as background, let's see `LiveData` in action. The `General/LiveSensor` sample project implements `LiveData` for sensor readings coming from a `SensorManager`. We can use this to track the accelerometer, ambient light, and so on.

However, the technique shown here can be used for lots of different system-level data sources, such as:

- Other system services (e.g., `LocationManager`, `ClipboardManager`)
- System broadcasts, for cases where you want to dynamically register for the broadcast via `registerReceiver()`
- Local broadcasts, using `LocalBroadcastManager`
- Content changes in providers, via a `ContentObserver`

Dependencies

To use `Lifecycle` and `LifecycleOwner`, you needed two dependencies: the lifecycle `runtime` library and its `compiler` annotation processor.

For some reason, `LiveData` is in a third dependency, called `extensions`:

```
dependencies {
    compile 'com.android.support:recyclerview-v7:26.0.0'
    compile 'android.arch.lifecycle:runtime:1.0.0-alpha8'
    compile 'android.arch.lifecycle:extensions:1.0.0-alpha8'
    annotationProcessor 'android.arch.lifecycle:compiler:1.0.0-alpha8'
}
```

(from General/LiveSensor/app/build.gradle)

State Transitions

We have a `SensorLiveData` class that extends the `LiveData` base class, offering to support a custom `Event static` nested class:

```
package com.commonsware.android.livedata;

import android.arch.lifecycle.LiveData;
import android.content.Context;
import android.hardware.Sensor;
import android.hardware.SensorEvent;
import android.hardware.SensorEventListener;
import android.hardware.SensorManager;
import java.util.Date;

class SensorLiveData extends LiveData<SensorLiveData.Event> {
  final private SensorManager sensorManager;
  private final Sensor sensor;
  private final int delay;

  SensorLiveData(Context ctxt, int sensorType, int delay) {
    sensorManager=
      (SensorManager)ctxt.getApplicationContext()
        .getSystemService(Context.SENSOR_SERVICE);
    this.sensor=sensorManager.getDefaultSensor(sensorType);
    this.delay=delay;

    if (this.sensor==null) {
      throw new IllegalStateException("Cannot obtain the requested sensor");
    }
  }

  @Override
  protected void onActive() {
    super.onActive();

    sensorManager.registerListener(listener, sensor, delay);
  }

  @Override
  protected void onInactive() {
    sensorManager.unregisterListener(listener);

    super.onInactive();
  }

  final private SensorEventListener listener=new SensorEventListener() {
    @Override
    public void onSensorChanged(SensorEvent event) {
      setValue(new Event(event));
    }

    @Override
    public void onAccuracyChanged(Sensor sensor, int accuracy) {
      // unused
    }
  };

  class Event {
    final Date date=new Date();
    final float[] values;

    Event(SensorEvent event) {
      values=new float[event.values.length];

      System.arraycopy(event.values, 0, values, 0, event.values.length);
    }
  }
}
```

(from General/LiveSensor/app/src/main/java/com/commonsware/android/livedata/SensorLiveData.java)

In the constructor, we hold onto configuration details, such as the particular sensor to monitor and how frequently we should ask for updates. We also obtain an instance of the `SensorManager` system service and try to find the actual requested `Sensor`, throwing a runtime exception if there is no matching sensor on this device.

However, we do not register for sensor events in the constructor. Until we have 1+ active observers, we do not need those events, and monitoring sensor events drains the battery. So, we postpone registering for events until `onActive()`, unregistering in the corresponding `onInactive()` callback.

Updating the Observers

The `SensorEventListener` that we use, in its `onSensorChanged()` method, creates a new instance of our `Event`, grabbing data from the `SensorEvent`. We use our own `Event` class for two reasons:

1. `SensorEvent` objects get recycled, and so it is not safe to hold onto one of those after the end of `onSensorChanged()`, so we copy the sensor results `float` values into our own object
2. While a `SensorEvent` has a timestamp, it is a pain to use, and this is a casual book sample, so we just track our own `Date` for simplicity

That `Event` is passed to `setValue()` on the `LiveData`, which in turn will pass the result to observers. Note that `setValue()` needs to be called on the main application thread — we will see how to handle events originating on background threads later in this chapter.

Retaining the LiveData

So, we have a `LiveData` for sensor readings. We can have an activity that displays those readings, by having it create a `SensorLiveData` instance and registering to observe those events. But now we run into a problem… what do we do with the `SensorLiveData` object after that?

One possibility is that we just hold onto it in a field, mostly to ensure that nothing gets garbage-collected that would interrupt the sensor readings. If we undergo a configuration change, we just create a new `SensorLiveData` objects and a fresh observer. While this is not completely ridiculous for this particular scenario, it is bad for cases where setting up the `LiveData` is expensive.

The idea behind `LiveData` is that it is the unique source of the specific data for the entire app. In other words, if we had several activities and fragments that all needed a particular sensor reading, we should set up a single `SensorLiveData` for all of them. That suggests using a singleton, and we will see how to do that later in this chapter. And, in truth, this is going to be the most common answer. However, it does raise some bits of complexity — in the case of `SensorLiveData`, there are many possible sensors, and a few possible delay periods, and so we would need a fairly sophisticated manager object to reuse or lazily create the appropriate `SensorLiveData` for a given client.

In this sample app, we take a middle-ground approach, and use `onRetainNonConfigurationInstance()` inside the activity that is going to use the sensor readings. Since the UI is going to be a `RecyclerView` of readings, we also need to hold onto past readings, so we do not lose them when we undergo the configuration change.

So, we have a `State` `static` nested class that holds onto the `SensorLiveData` and outstanding readings:

```
private static class State {
  final ArrayList<SensorLiveData.Event> events=new ArrayList<>();
  SensorLiveData sensorLiveData;
}
```

(from General/LiveSensor/app/src/main/java/com/commonsware/android/livedata/MainActivity.java)

In `onCreate()`, we set up that `State` if we do not already have one, storing it in a `state` field. This includes setting up the `SensorLiveData`, in this case for the ambient light sensor:

```
private State state;

@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.main);

  RecyclerView rv=findViewById(R.id.transcript);

  state=(State)getLastNonConfigurationInstance();

  if (state==null) {
    state=new State();
    state.sensorLiveData=
      new SensorLiveData(this, Sensor.TYPE_LIGHT,
        SensorManager.SENSOR_DELAY_UI);
  }

  adapter=new EventLogAdapter();
  rv.setAdapter(adapter);
```

```
    state.sensorLiveData.observe(this, new Observer<SensorLiveData.Event>() {
      @Override
      public void onChanged(@Nullable SensorLiveData.Event event) {
        adapter.add(event);
      }
    });
  }
```

(from General/LiveSensor/app/src/main/java/com/commonsware/android/livedata/MainActivity.java)

We also register our `Observer`, which will be called with `onChanged()` with a new `Event` as sensor readings come in. Our `EventLogAdapter` knows how to `add()` that to the list of historical readings and update the `RecyclerView`.

However, the `LiveData` will automatically deliver the last-received reading to our observer when we attach a fresh observer after a configuration change. That could result in `onChanged()` being given the same `Event` object as before, one that we already put into the `ArrayList`. So, the `EventLogAdapter` `add()` method checks that first, before actually adding it:

```
    void add(SensorLiveData.Event what) {
      if (!state.events.contains(what)) {
        state.events.add(what);
        notifyItemInserted(getItemCount());
      }
    }
```

(from General/LiveSensor/app/src/main/java/com/commonsware/android/livedata/MainActivity.java)

And we override `onRetainNonConfigurationInstance()` to return the `State` instance, so `onCreate()` can retrieve it after a configuration change:

```
  @Override
  public Object onRetainNonConfigurationInstance() {
    return(state);
  }
```

(from General/LiveSensor/app/src/main/java/com/commonsware/android/livedata/MainActivity.java)

Other LiveData Examples

Let's take a look at a few more examples of using `LiveData`, to explore other facets of how this can be used.

Event Bus

`LocalBroadcastManager` implements an in-process event bus, where events are delivered to you on the main application thread, and where "events" are `Intent` objects.

You can accomplish the same thing, with greater flexibility, by means of a `LiveData` object, as can be seen in the `General/LiveBus` sample project.

This sample app is derived from one shown in *The Busy Coder's Guide to Android Development*, where we have `AlarmManager` triggering a service. In principle, that service should do some work, which we are skipping here because we are lazy. However, the fake work is something that the user might care about, and so we want to let the UI layer know about the event if we happen to be in the foreground. Otherwise, we want to raise a `Notification`. In *The Busy Coder's Guide to Android Development*, implementations of this sample are available for a few event buses, including `LocalBroadcastManager` and greenrobot's EventBus.

Here, though, we will use a `MutableLiveData` singleton:

```
static final MutableLiveData<Intent> BUS=new MutableLiveData<>();
private static int NOTIFY_ID=1337;
private Random rng=new Random();

public ScheduledService() {
  super("ScheduledService");
}

@Override
protected void doWakefulWork(Intent intent) {
  Intent event=new Intent(EventLogFragment.ACTION_EVENT);
  long now=Calendar.getInstance().getTimeInMillis();
  int random=rng.nextInt();

  event.putExtra(EventLogFragment.EXTRA_RANDOM, random);
  event.putExtra(EventLogFragment.EXTRA_TIME, now);

  if (BUS.hasActiveObservers()) {
    BUS.postValue(event);
  }
  else {
    NotificationCompat.Builder b=new NotificationCompat.Builder(this);
    Intent ui=new Intent(this, EventDemoActivity.class);
```

```
      b.setAutoCancel(true).setDefaults(Notification.DEFAULT_SOUND)
       .setContentTitle(getString(R.string.notif_title))
       .setContentText(Integer.toHexString(random))
       .setSmallIcon(android.R.drawable.stat_notify_more)
       .setTicker(getString(R.string.notif_title))
       .setContentIntent(PendingIntent.getActivity(this, 0, ui, 0));

      NotificationManager mgr=
          (NotificationManager)getSystemService(NOTIFICATION_SERVICE);

      mgr.notify(NOTIFY_ID, b.build());
    }
  }
}
```

(from General/LiveBus/app/src/main/java/com/commonsware/android/livedata/bus/ScheduledService.java)

MutableLiveData is a subclass of LiveData, with one key feature: it offers a postValue() method that works like setValue() but can be called from a background thread. Here, our events are in the form of Intent objects, the way they would be for LocalBroadcastManager. However, you could create your own custom event objects if you prefer, and typically that would be a better idea. In this case, the sample is demonstrating a quick-and-dirty change from LocalBroadcastManager, so we are keeping the event objects the same to reduce the number of code changes.

The service, as part of its work, asks the BUS whether there are any active observers, by means of hasActiveObservers(). If hasActiveObservers() returns true, we use postValue() to post the event onto our BUS. Otherwise, we raise a Notification, as our UI is not in the foreground.

(note: this service extends WakefulIntentService, and so the method is doWakefulWork() instead of the onHandleIntent() that you might typically use with an IntentService).

Our UI is in the form of a ListFragment. However, ListFragment itself is not tied to a Lifecycle. So, we have a LifecycleListFragment that provides this capability:

```
package com.commonsware.android.livedata.bus;

import android.app.ListFragment;
import android.arch.lifecycle.Lifecycle;
import android.arch.lifecycle.LifecycleOwner;
import android.arch.lifecycle.LifecycleRegistry;
import android.os.Bundle;
import android.support.annotation.Nullable;
```

```
public class LifecycleListFragment extends ListFragment
  implements LifecycleOwner {
  private LifecycleRegistry registry=new LifecycleRegistry(this);

  @Override
  public Lifecycle getLifecycle() {
    return(registry);
  }

  @Override
  public void onCreate(@Nullable Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    registry.handleLifecycleEvent(Lifecycle.Event.ON_CREATE);
  }

  @Override
  public void onStart() {
    super.onStart();

    registry.handleLifecycleEvent(Lifecycle.Event.ON_START);
  }

  @Override
  public void onResume() {
    super.onResume();

    registry.handleLifecycleEvent(Lifecycle.Event.ON_RESUME);
  }

  @Override
  public void onPause() {
    super.onPause();

    registry.handleLifecycleEvent(Lifecycle.Event.ON_PAUSE);
  }

  @Override
  public void onStop() {
    super.onStop();

    registry.handleLifecycleEvent(Lifecycle.Event.ON_STOP);
  }

  @Override
  public void onDestroy() {
    super.onDestroy();
    registry.handleLifecycleEvent(Lifecycle.Event.ON_DESTROY);
  }
}
```

(from General/LiveBus/app/src/main/java/com/commonsware/android/livedata/bus/LifecycleListFragment.java)

That allows our `EventLogFragment` to register an observer on the `BUS`, adding the events to its `ArrayAdapter`:

```
    ScheduledService.BUS.observe(this, new Observer<Intent>() {
      @Override
      public void onChanged(@Nullable Intent intent) {
        adapter.add(intent);
      }
    });
```

(from General/LiveBus/app/src/main/java/com/commonsware/android/livedata/bus/EventLogFragment.java)

Unlike `LocalBroadcastManager`, this approach performs no `Intent` filtering. However, unlike `LocalBroadcastManager`, we can have as many `MutableLiveData` objects as needed. So, you can create custom buses for different event channels, instead of using action strings as you might with `LocalBroadcastManager`.

Room

Having DAO methods in Room return a `LiveData` is simply a matter of setting them up that way:

```
  @Query("SELECT * FROM Customer WHERE postalCode IN (:postalCodes) LIMIT :max")
  LiveData<List<Customer>> findByPostalCodes(int max, String... postalCodes);
```

(from General/LiveRoom/stuff/src/main/java/com/commonsware/android/room/dao/StuffStore.java)

Now, `findByPostalCodes()` will return a `LiveData`. Moreover, it will do so *immediately* when called, with the actual query being performed on a Room-supplied background thread. You can arrange to register an observer to find out when the results are ready. And, by using the same `LiveData` instance after a configuration change, you can get the last-loaded results without having to perform another round of disk I/O.

However, Room has an additional feature: if you make changes to the database through your DAO, Room will deliver fresh results to any registered observer of your `LiveData`. So, for example:

- You register an observer on a LiveData, returned by a Room @Query, that represents a list of your entities
- Shortly thereafter, you get the list of entities as they exist in the database at present, for you to fill into your RecyclerView (or whatever)
- Later on, as part of processing a request from the user, you invoke an @Insert method on your DAO to add a new entity to the database
- Your registered observer gets the updated list of entities as they exist in the database, for you to fill into your RecyclerView (or whatever)
- And so on

In effect, Room attempts to give you ContentObserver capabilities, for your own database, tied directly into the LiveData system.

Note, though, that these changes are tied in large part to your use of the DAO. For example, if you want to insert 100 entities, you could:

- Call a single @Insert method that takes a List of those entities, in which case you will get a single update from the LiveData
- Call a one-entity @Insert method 100 times, in which case you will get 100 updates from the LiveData

Doing things in batch form generally will be more efficient, both from a disk I/O standpoint and a LiveData-updating standpoint. On the other hand, this means that a LiveData update might represent several changes, and that may require additional smarts to handle properly in terms of updating the UI (e.g., use DiffUtil to efficiently update a RecyclerView).

We will see using LiveData with Room in the next chapter.

Testing LiveData

LiveData works asynchronously, and so your tests have to deal with this. There are various patterns for handling asynchronous tests. One is to use a CountDownLatch.

The General/LiveRoom sample project has the revised findByPostalCodes() method shown above, and so we need to modify the instrumentation tests to match.

The DaoTests class now has two additional fields:

1. A `CountDownLatch` named `responseLatch`
2. A `List` of `Customer` objects, named `customers`

In `setUp()`, we initialize the `CountDownLatch`, set to track one event:

```
@Before
public void setUp() {
  db=StuffDatabase.create(InstrumentationRegistry.getTargetContext(), true);
  store=db.stuffStore();
  responseLatch=new CountDownLatch(1);
}
```

(from General/LiveRoom/stuff/src/androidTest/java/com/commonsware/android/room/dao/DaoTests.java)

In the test, we can retrieve the `LiveData`, register an observer, and have the observer save the results in `customers` and `countDown()` the `CountDownLatch`:

```
    final LiveData<List<Customer>>
      liveResult=store.findByPostalCodes(10, firstCustomer.postalCode);

    liveResult.observeForever(new Observer<List<Customer>>() {
      @Override
      public void onChanged(@Nullable List<Customer> customers) {
        DaoTests.this.customers=customers;
        responseLatch.countDown();
      }
    });

    responseLatch.await();

    assertEquals(1, customers.size());
```

(from General/LiveRoom/stuff/src/androidTest/java/com/commonsware/android/room/dao/DaoTests.java)

Instrumentation test methods run on background threads, and `onChanged()` is called on the main application thread. So, we block the test thread via `await()`, to wait on the disk I/O to complete. At that point, we have our `List` of `Customer` objects for assertions.

ViewModel

Many Android apps are trivial. The smaller the app, the less likely it is that you need much in the way of a true GUI architecture. Slapping together whatever you want wherever you want it most likely will suffice. Your average soundboard, flashlight, front-facing-camera “mirror”, and similar apps just do what they do, and their developers do not need to worry about the alphabet soup of MVC, MVP, MVVM, MVI, and so on.

If you are reading this book, you may have an app in mind that is not so trivial.

The more complex the app, the more likely it is that you are going to want to think more seriously about the GUI architecture. The Architecture Components contribution to this is the `ViewModel`, which we will explore in this chapter.

ViewModels, As Originally Envisioned

Microsoft devised the model-view-viewmodel (MVVM) GUI architecture in 2005, and it has remained generally murky ever since. This is not terribly surprising, as many of the “alphabet soup” GUI architectures have malleable definitions which developers can twist and tweak to match what it is that they want to write.

Roughly speaking, in this GUI architecture, the “view model” represents a collection of data and other state, necessary to render a view, derived from the underlying models. The view model would be responsible for things like data formatting (e.g., converting the model’s `long` Unix epoch time into something that the user will be able to read). The view updates the view model, which in turn updates the model at the appropriate time.

Ideally, the view model knows nothing much about the view, but rather just exposes data and operations that the view needs.

The Architecture Components ships with a `ViewModel` class. This class does almost nothing. This will be an important point, as what little we get from `ViewModel` can be implemented in other ways without significant difficulty. But, for now, consider `ViewModel` to be a place to hold the data necessary to represent your views. For example, a `ViewModel` might hold a list of objects, obtained from Room, that are used to populate a `RecyclerView`.

ViewModel Versus…

The objective of `ViewModel`, in particular, is to be able to survive past configuration changes.

Of course, we have been dealing with configuration changes for years, before the Architecture Components were a glimmer in any Google engineer's eye.

So, when would we use a `ViewModel`, and when would we use other techniques?

…Saved Instance State

Saved instance state — what you put into the `Bundle` supplied to `onSaveInstanceState()` – survives process termination. A `ViewModel` does not. So while both can help deal with configuration changes, only saved instance state can help with the process termination scenario:

- User is in your app, in an activity
- User navigates to something else (e.g., presses HOME, switches to another task via the overview screen)
- A few minutes later, Android terminates your process to free up system RAM
- A few minutes after that — but within 30 minutes of the user navigating away – the user returns to your task
- Android recreates the activity atop your task's back stack as part of forking a fresh process for you, and Android hands you your saved instance state `Bundle` back

However, the saved instance state `Bundle` has size limits (should be well under 1MB) and type limits (only objects that can go into a `Parcel`).

As a result:

- Use the ViewModel for holding onto data in your process necessary to be able to rapidly repopulate the UI after a configuration change
- Use the saved instance state Bundle to hold identifiers and other data that will help you rebuild the UI after process termination, even if you wind up having to re-read from disk or the network as part of that work

…Retained Objects

In the end, the ViewModelProviders system supplied by the Architecture Components is an oddly-written wrapper around retained fragments. As a result, there is nothing that you can do with a ViewModel that you could not do using retained objects, whether those are retained fragments or using onRetainNonConfigurationInstance(). We will see examples of this later in this chapter.

Mommy, Where Do ViewModels Come From?

You might think that you create a ViewModel via whatever constructor you set up for it. And, if you are going to manage a ViewModel yourself — via the retained object pattern described above — then this is perfectly fine.

The Architecture Components expect you to get a ViewModel instance by using ViewModelProvider. A ViewModelProvider instance is tied to either:

- A FragmentActivity (or a subclass, like AppCompatActivity), or
- A Fragment, from the fragments backport

If you do not have one of those, you cannot use ViewModelProvider.

If you *do* have one of those, call the static of() method on the ViewModelProviders class (note the plural) to get a ViewModelProvider (note the singular) tied to your FragmentActivity or Fragment. This ViewModelProvider is tied to the *logical* instance of this activity or fragment, regardless of configuration changes. So, if the activity is destroyed and recreated as part of a configuration change, you will get the same ViewModelProvider instance in the new activity as you had in the old one.

Then, to get a `ViewModel`, call `get()` on the `ViewModelProvider`, passing in the Java class object for your subclass of `ViewModel` (e.g., `MyViewModel.class`). If there already is an instance of this `ViewModel` tied to this `ViewModelProvider`, you get that instance. Otherwise, a fresh instance will be created for you, from the zero-argument constructor. If using the zero-argument constructor is not what you want, you can:

- Create an implementation of the `ViewModelProvider.Factory` interface, implementing the `create()` method to create an instance of your `ViewModel` by whatever constructor you want
- Associate an instance of your `ViewModelProvider.Factory` with the `ViewModelProvider` by supplying it as a second parameter to the `of()` method on `ViewModelProviders`

So, in the typical case, you wind up with code like this:

```
    TripRosterViewModel vm=
      ViewModelProviders.of(this).get(TripRosterViewModel.class);
```

(from Trips/ViewModels/app/src/main/java/com/commonsware/android/room/TripsFragment.java)

Here, `this` inherits from the `Fragment` backport, and we are retrieving a `TripRosterViewModel` to use in that fragment.

We will see this code snippet again in the next section.

ViewModels, Google's Way

So, let's take a look at the `Trips/ViewModels` sample project. This adds a `ViewModel` to our app showing a roster of upcoming trips. More specifically, we will use `ViewModelProvider`, the way Google envisioned it.

Earlier editions of this sample used Android's native `Activity` and `Fragment` classes. Those do not work with `ViewModelProviders`. So, in this sample, `MainActivity` has been revised to extend from `FragmentActivity` and `RecyclerViewFragment` has been revised to extend from `LifecycleFragment`. Using `LifecyleFragment` allows us to use `LiveData` for retrieving our trips from Room. Otherwise, we could just use the backport `Fragment` class, as `ViewModelProvider` has nothing to do with the lifecycle classes.

Defining a ViewModel

The idea is that a `ViewModel` should hold the data necessary to render the UI. In our case, that is simply a roster of `Trip` objects, pulled in from Room.

For `ViewModelProvider` to work, the class must be `public`, even though your IDE might suggest otherwise. So, our `TripRosterViewModel` is `public`:

```java
package com.commonsware.android.room;

import android.app.Application;
import android.arch.lifecycle.AndroidViewModel;
import android.arch.lifecycle.LiveData;
import java.util.List;

public class TripRosterViewModel extends AndroidViewModel {
  final LiveData<List<Trip>> allTrips;

  public TripRosterViewModel(Application app) {
    super(app);

    allTrips=TripDatabase.get(app).tripStore().selectAllTrips();
  }
}
```

(from Trips/ViewModels/app/src/main/java/com/commonsware/android/room/TripRosterViewModel.java)

Note that `TripRosterViewModel` extends from `AndroidViewModel`. `AndroidViewModel` itself extends `ViewModel`. The only difference between the two is the constructor: `ViewModel` has a zero-argument constructor, while `AndroidViewModel` has a one-argument constructor, supplying the `Application` instance. In our case, we need the `Application` instance to `get()` our `TripDatabase` (as Room needs a `Context` for this).

`TripRosterViewModel`, in its constructor, sets up an `allTrips` field that is a `LiveData` of our roster of `Trip` objects. Since this is `LiveData`, the actual work will not be done until we ask it to, by registering an observer to use the results.

Getting a ViewModel

Our `TripsFragment` needs access to the `TripRosterViewModel`, in order to be able to get to the `allTrips` data and request the roster of `Trip` objects.

However, now we have a decision to make: is the TripRosterViewModel tied to the fragment or to the activity?

Since a fragment can get to its hosting activity via getActivity(), a fragment can choose either scope:

- Pass this into of() to get the ViewModelProvider tied to the fragment, or
- Pass getActivity() into of() to get the ViewModelProvider tied to the activity

Either is perfectly legitimate. Frequently, it will boil down to who needs the data. Data that is only needed by a single fragment should be owned by a ViewModel tied to that fragment. Data needed by multiple fragments, or by a fragment and the activity, or just by the activity, should be owned by a ViewModel tied to the activity. A fragment can also elect to do both, using two ViewModel instances, one for its own data and one that it gets via the activity.

In this case, the only UI is the TripsFragment, so we can say that the TripRosterViewModel is owned by the fragment and retrieve it as part of our onViewCreated() work:

```
    TripRosterViewModel vm=
      ViewModelProviders.of(this).get(TripRosterViewModel.class);
```

(from Trips/ViewModels/app/src/main/java/com/commonsware/android/room/TripsFragment.java)

The first time we run through these lines, we will get a fresh TripRosterViewModel instance. If we undergo a configuration change, when this fragment is recreated, the new fragment instance will get the same TripRosterViewModel as before.

Using the ViewModel

Given our TripRosterViewModel, our TripsFragment can now get at the roster of Trip objects, by registering an Observer:

```
    vm.allTrips.observe(this, new Observer<List<Trip>>() {
      @Override
      public void onChanged(@Nullable List<Trip> trips) {
        setAdapter(new TripsAdapter(trips, getActivity().getLayoutInflater()));

        if (trips==null || trips.size()==0) {
          final TripStore store=TripDatabase.get(getActivity()).tripStore();

          new Thread() {
            @Override
            public void run() {
              store.insert(new Trip("Vacation!", 10080, Priority.MEDIUM, new Date()),
                new Trip("Business Trip", 4320, Priority.OMG, new Date()));
            }
          }.start();
        }
      }
    });
```

(from Trips/ViewModels/app/src/main/java/com/commonsware/android/room/TripsFragment.java)

A typical app would just have the `setAdapter()` call, to pass the `Trip` roster over to the `TripsAdapter`, to show the roster in the `RecyclerView`. In this case, we want to lazy-create some trips, as otherwise we will have no data. So, if we have no trips, we insert some in a background thread.

However, there are two issues with that approach. One is the possible race condition, where the user rotates the screen while the background thread is going on, and so we fork a second thread. Since this code is not the sort of thing you would do in a production app, what we have here will suffice for now.

But, if you run the app, you will see that our data shows up in the `RecyclerView`, even after a fresh run of the app, when we did not have any data. Yet, our `Thread` is not doing anything to refresh the UI. So, the second issue is: how is this working?

The answer is that Room is monitoring our DAO for changes and is automatically updating the `LiveData` to reflect those changes, as was mentioned in the chapter on `LiveData`.

Getting Rid of the ViewModel

Ideally, you should not have to do anything to explicitly "get rid of" a `ViewModel`. If you are using `LiveData`, it is lifecycle-aware, and so it should clean up itself when the activity or fragment is destroyed. If you have anything else in the `ViewModel` that needs cleanup when the activity or fragment is destroyed, use the lifecycle classes or `LiveData` for that.

ViewModels as Simple POJOs

The primary limitation of `ViewModelProviders` is that it is inextricably tied to `FragmentActivity` and the backport of `Fragment`. If you are using those classes, or things inheriting from them (e.g., `AppCompatActivity`), great! If not, you will need to pursue alternatives.

One alternative is to ignore `ViewModel` entirely, and implement a view model yourself as a POJO, as we will explore in the `Trips/ViewModelPOJO` sample project. This is a clone of the previous sample, except that we are using a POJO and `onRetainNonConfigurationInstance()` rather than `ViewModel` and `ViewModelProviders`.

Since we are not using `ViewModelProviders`, `MainActivity` inherits from `Activity` and `RecyclerViewFragment` inherits from the framework's implementation of `Fragment`.

Defining a ViewModel

The `ViewModel` class itself, from the Architecture Components, mostly serves as a marker. It adds very little logic. So, modifying `TripRosterViewModel` to be a POJO simply involves removing `AndroidViewModel`:

```
package com.commonsware.android.room;

import android.app.Application;
import android.arch.lifecycle.LiveData;
import java.util.List;

class TripRosterViewModel {
  final LiveData<List<Trip>> allTrips;

  TripRosterViewModel(Application app) {
    allTrips=TripDatabase.get(app).tripStore().selectAllTrips();
  }
}
```

(from Trips/ViewModelPOJO/app/src/main/java/com/commonsware/android/room/TripRosterViewModel.java)

And, in this case, since we will create our `TripRosterViewModel` conventionally via its constructor, it can be package-private, rather than `public`.

Getting a ViewModel

One downside to the POJO approach is that the simple way of using it as a view model limits your scope to activities. Fragments do not have a trivial `onRetainNonConfigurationInstance()/getLastNonConfigurationInstance()` implementation the way activities do. It is certainly possible to do something to retain a per-fragment view model across configuration changes, but it requires more work (e.g., retained fragments).

So, here, we move the `TripRosterViewModel` management into `MainActivity`:

```
package com.commonsware.android.room;

import android.app.Activity;
import android.os.Bundle;
import android.support.annotation.Nullable;

public class MainActivity extends Activity {
  private TripRosterViewModel viewModel;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    viewModel=(TripRosterViewModel)getLastNonConfigurationInstance();

    if (viewModel==null) {
      viewModel=new TripRosterViewModel(getApplication());
    }

    if (getFragmentManager().findFragmentById(android.R.id.content)==null) {
      getFragmentManager().beginTransaction()
        .add(android.R.id.content,
          new TripsFragment()).commit();
    }
  }

  @Override
  public Object onRetainNonConfigurationInstance() {
    return(getViewModel());
  }

  TripRosterViewModel getViewModel() {
    return(viewModel);
  }
}
```

(from Trips/ViewModelPOJO/app/src/main/java/com/commonsware/android/room/MainActivity.java)

`onCreate()` retrieves the `TripRosterViewModel`, creating a new instance if we do not have one. `onRetainNonConfigurationInstance()` returns that `TripRosterViewModel`, so we retain it across configuration changes. And we expose the `TripRosterViewModel` to the `TripsFragment` via a `getViewModel()` method.

Now, `TripsFragment` can get the `TripRosterViewModel` by a simple call on the hosting activity:

```
    TripRosterViewModel vm=((MainActivity)getActivity()).getViewModel();
```

(from Trips/ViewModelPOJO/app/src/main/java/com/commonsware/android/room/TripsFragment.java)

Nothing else needs to change:

- We `observe()` the `allTrips` `LiveData` as before
- We still do not need to worry about cleaning up the `LiveData` when the activity is destroyed and the `TripRosterViewModel` is no longer needed

Other Lifecycle Owners

Activities and fragments are not the only things with lifecycles. The Architecture Components also support other forms of lifecycle owner:

- Services, and
- What the documentation will refer to as "the process"

ProcessLifecycleOwner

With a name like `ProcessLifecycleOwner`, you might think that this modeled the lifecycle of a process. Then, you quickly realize that this makes little sense, as the only "lifecycle" that a process goes through is creation and termination, and we cannot get control in the latter event.

Insted, `ProcessLifecycleOwner` might better be named `ForegroundLifecycleOwner`. Whereas `LifecycleActivity` models the lifecycle of an individual activity, `ProcessLifecycleOwner` models the lifecycle of all activities combined:

- `ON_CREATE` is triggered when the process starts up
- `ON_START` and `ON_RESUME` are triggered when an activity goes through those lifecycle events, and no other activity had been started recently
- `ON_PAUSE` and `ON_STOP` are triggered, after a delay, when an activity goes through those lifecycle events, if another activity is not started and resumed by this time
- `ON_DESTROY` is never triggered

The delay period is 700ms (as of `1.0.0-alpha3`), so as long as another activity is started and resumed after a prior activity was paused and stopped within 700ms, the *process* has not undergone a lifecycle change, even though those individual activities did.

So, imagine a single-activity app:

- `ON_CREATE` happens right away
- `ON_START` and `ON_RESUME` happen shortly thereafter, assuming that the process is starting because an activity is being displayed
- The user rotates the screen, causing the activity to be destroyed and recreated
- `ON_PAUSE` and `ON_STOP` *do not occur*, because a new activity was started and resumed before the `ProcessLifecycleOwner` delay period elapsed
- `ON_START` and `ON_RESUME` *do not occur*, because we did not move through the paused and stopped lifecycle states, even though the new activity instance did
- The user presses HOME, BACK, or otherwise leaves this activity for another task
- `ON_PAUSE` and `ON_STOP` happen after the delay period, since no activity from this process went through `ON_START` and `ON_RESUME` during that time

Note that this comes at a cost: the `extensions` artifact automatically adds a `<provider>` element to your manifest, one that initializes the `ProcessLifecycleOwner`… even if your app does not use `ProcessLifecycleOwner`. This is simply so `ProcessLifecycleOwner` code can be invoked as soon as your process is started.

The `General/ProcessLifecycle` sample project has a `LifecycleApplication` that registers itself as an observer of the singleton instance of `ProcessLifecycleOwner` and dumps all the events to LogCat:

```java
package com.commonsware.android.recyclerview.videolist;

import android.app.Application;
import android.arch.lifecycle.Lifecycle;
import android.arch.lifecycle.LifecycleObserver;
import android.arch.lifecycle.OnLifecycleEvent;
import android.arch.lifecycle.ProcessLifecycleOwner;
import android.util.Log;

public class LifecycleApplication extends Application
  implements LifecycleObserver {
  @Override
  public void onCreate() {
    super.onCreate();

    ProcessLifecycleOwner.get().getLifecycle().addObserver(this);
  }

  @OnLifecycleEvent(Lifecycle.Event.ON_CREATE)
  public void created() {
    Log.d(getClass().getSimpleName(), "ON_CREATE");
  }

  @OnLifecycleEvent(Lifecycle.Event.ON_START)
  public void started() {
    Log.d(getClass().getSimpleName(), "ON_START");
  }

  @OnLifecycleEvent(Lifecycle.Event.ON_RESUME)
  public void resumed() {
    Log.d(getClass().getSimpleName(), "ON_RESUME");
  }

  @OnLifecycleEvent(Lifecycle.Event.ON_PAUSE)
  public void paused() {
    Log.d(getClass().getSimpleName(), "ON_PAUSE");
  }

  @OnLifecycleEvent(Lifecycle.Event.ON_STOP)
  public void stopped() {
    Log.d(getClass().getSimpleName(), "ON_STOP");
  }

  @OnLifecycleEvent(Lifecycle.Event.ON_DESTROY)
  public void destroyed() {
    Log.d(getClass().getSimpleName(), "ON_DESTROY");
  }
}
```

(from General/ProcessLifecycle/app/src/main/java/com/commonsware/android/recyclerview/videolist/LifecycleApplication.java)

That `LifecycleApplication` is then registered in the manifest via `android:name` on `<application>`:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest package="com.commonsware.android.recyclerview.videolist"
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:versionCode="1"
  android:versionName="1.0">

  <supports-screens
    android:anyDensity="true"
    android:largeScreens="true"
    android:normalScreens="true"
    android:smallScreens="false" />

  <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />

  <application
    android:allowBackup="false"
    android:name=".LifecycleApplication"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/Theme.Apptheme">
    <activity android:name=".MainActivity">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />

        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
    <activity
      android:name=".VideoPlayerActivity"
      android:configChanges="screenSize|smallestScreenSize|screenLayout|orientation"
      android:launchMode="singleTask"
      android:supportsPictureInPicture="true"
      android:theme="@style/Theme.Apptheme.NoActionBar" />

    <receiver android:name=".RemoteActionReceiver" />

  </application>

</manifest>
```

(from General/ProcessLifecycle/app/src/main/AndroidManifest.xml)

The app itself is a clone of one from *The Busy Coder's Guide to Android Development*. It consists of two activities. One shows a list of all videos indexed by the `MediaStore`. The other plays back a selected video using a `VideoView`. And, on Android 8.0+ devices, the video player activity will have a FAB that switches that activity into picture-in-picture mode.

(NOTE: to run this sample, your test device will need 1+ videos)

If you run it, you will see the `ON_CREATE`, `ON_START`, and `ON_RESUME` events logged in rapid succession. And, if you do not press that enticing FAB, and just use the video player in normal mode, `ON_PAUSE` and `ON_STOP` get invoked at normal times, such as when the user navigates to some other task (e.g., presses HOME).

The FAB, though, changes things, as it moves the video player to a floating picture-in-picture (PiP) window.

If you tap the FAB, and do not touch anything else for a bit, you will see `ON_PAUSE`, then `ON_RESUME`, get logged. This is because:

- The PiP window never has the foreground from an input standpoint, and so its activity is paused, but not stopped (as it is still visible)
- The underlying activity is started and resumed, though with a few seconds' delay, for inexplicable reasons

Similarly, if you tap the PiP window, to bring up the controls, you will see `ON_PAUSE` logged, as the list-of-videos activity is paused (it no longer has the foreground input) but the PiP window is *not* resumed (the input is handled by the system UI, not the activity). After a few moments of inactivity, that PiP window will return to its regular state, and `ON_RESUME` will be logged.

Playing around with the PiP further (e.g., closing it via the X in the corner) allows you to see how PiP mode ties into activity lifecycles.

LifecycleService

If you have a class that extends `Service`, you can replace it with `LifecycleService` and get a service that is a `LifecycleOwner`. Four of the six lifecycle events are honored:

This Lifecycle Event…	Is Triggered When…
`ON_CREATE`	the service is created
`ON_START`	when the service is first started or bound to
`ON_RESUME`	**unused**
`ON_PAUSE`	**unused**
`ON_STOP`	when the service is destroyed
`ON_DESTROY`	also when the service is destroyed

Of note, `LifecycleService` does not attempt to model binding/unbinding as a lifecycle (e.g., calling `ON_STOP` when the service is unbound and has no more active bindings).

However, most services do not directly inherit from `Service`. Instead, they extend `IntentService` or `JobService` or any one of dozens of other specialized service implementations. Few, if any, of those will extend `LifecycleService`, as most of them come from the core framework, which cannot depend on libraries like the Architecture Components.

Wait… Where Is LifecycleProvider and LifecycleReceiver?

A `ContentProvider` has no real "lifecycle". It is called with `onCreate()` when the process starts up… and that's about it. Similarly, a `BroadcastReceiver` is called with `onReceive()`… and that's about it.

Intermediate Topics

M:N Relations in Room

For 1:1 relations, one entity has a foreign key back to the other entity.

For 1:N relations, one entity has a foreign key back to the other entity. In other words, 1:1 is simply 1:N for a specific small value of N.

In SQL, implementing M:N relations requires a join table of some form, where the join table has foreign keys back to the entities being related. Room, using SQL at its core, does not change this. And since Room does not model relations, but only foreign keys, to create an M:N relation, you have to create a "join entity" that winds up creating the associated join table.

In this chapter, we will take a look at how that is accomplished. Along the way, we will also look at other Room tidbits, such as how to use static classes as entities.

Implementing a Join Entity

The `General/RoomMN` sample project demonstrates an M:N relation. From earlier chapters, we have a `Customer` entity and a `Category` entity. Previously, those were unrelated. Now, let's implement an M:N relation between them, so a `Customer` can be a member of zero or more categories, and a `Category` can have zero or more customers.

Note that we are retaining the tree structure for `Category` used previously. For the purposes of this chapter, we are ignoring that, considering a customer to belong to a category only via a direct relationship. So for example, if customer `Foo` belongs to category `Child`, which has a parent category `Parent`, `Foo` is not a member of `Parent`. The tree structure simply organizes categories, without impacting customers.

Static Entity Classes

Much of the time, your entity classes will be standard, top-level Java classes. Sometimes, though, you might have some utility class that you would rather have as a static class, nested inside something else. For example, in the case of a join entity, perhaps you might want to tuck it inside of one of the entities being joined, just to reduce the clutter of your namespace.

Fortunately, this works, albeit with a wrinkle.

In the sample project, the `Customer` class — which itself is an entity — has a `static` class named `CategoryJoin` that will serve as the join entity:

```java
package com.commonsware.android.room.dao;

import android.arch.persistence.room.Embedded;
import android.arch.persistence.room.Entity;
import android.arch.persistence.room.ForeignKey;
import android.arch.persistence.room.Ignore;
import android.arch.persistence.room.Index;
import android.arch.persistence.room.PrimaryKey;
import java.util.Date;
import java.util.Set;
import java.util.UUID;
import static android.arch.persistence.room.ForeignKey.CASCADE;

@Entity(indices={@Index(value="postalCode", unique=true)})
class Customer {
  @PrimaryKey
  public final String id;

  public final String postalCode;
  public final String displayName;
  public final Date creationDate;

  @Embedded
  public final LocationColumns officeLocation;

  public final Set<String> tags;

  @Ignore
  Customer(String postalCode, String displayName, LocationColumns officeLocation,
           Set<String> tags) {
    this(UUID.randomUUID().toString(), postalCode, displayName, new Date(),
      officeLocation, tags);
  }

  Customer(String id, String postalCode, String displayName, Date creationDate,
           LocationColumns officeLocation, Set<String> tags) {
    this.id=id;
    this.postalCode=postalCode;
    this.displayName=displayName;
    this.creationDate=creationDate;
    this.officeLocation=officeLocation;
    this.tags=tags;
  }

  @Entity(
    tableName="customer_category_join",
    primaryKeys={"categoryId", "customerId"},
    foreignKeys={
      @ForeignKey(
        entity=Category.class,
        parentColumns="id",
        childColumns="categoryId",
        onDelete=CASCADE),
      @ForeignKey(
        entity=Customer.class,
        parentColumns="id",
        childColumns="customerId",
        onDelete=CASCADE)},
    indices={
      @Index(value="categoryId"),
      @Index(value="customerId")
    }
  )
  public static class CategoryJoin {
    public final String categoryId;
    public final String customerId;

    public CategoryJoin(String categoryId, String customerId) {
      this.categoryId=categoryId;
      this.customerId=customerId;
    }
  }
}
```

(from General/RoomMN/stuff/src/main/java/com/commonsware/android/room/dao/Customer.java)

Room is perfectly content to work with this class, so long as you also register it with your `RoomDatabase` via its `@Database` annotation:

```
@Database(
  entities={Customer.class, Category.class, Customer.CategoryJoin.class},
  version=1
)
```

(from General/RoomMN/stuff/src/main/java/com/commonsware/android/room/dao/StuffDatabase.java)

However, note that this is a `static` class. Room will not be able to work with a non-`static` nested class, as only instances of the outer class can create instances of the nested class.

Also, note that the default table name is based on the plain class name. In this case, the default table name is `CategoryJoin`. The outer class name (`Customer`) is not added into the table name. Normally, this will not be a problem, and you might be renaming the table anyway. However, where you can get tripped up is if you decided to have two (or more) classes with the same name, such as having `CategoryJoin` inside both `Customer` and some other entity. Then, you would wind up with two entity classes both trying to define the same table name by default, and Room will not like that very much.

Foreign Keys and Indices

Let's take a closer look at the `@Entity` annotation on `Customer.CategoryJoin`:

```
@Entity(
  tableName="customer_category_join",
  primaryKeys={"categoryId", "customerId"},
  foreignKeys={
    @ForeignKey(
      entity=Category.class,
      parentColumns="id",
      childColumns="categoryId",
      onDelete=CASCADE),
    @ForeignKey(
      entity=Customer.class,
      parentColumns="id",
      childColumns="customerId",
      onDelete=CASCADE)},
  indices={
    @Index(value="categoryId"),
    @Index(value="customerId")
  }
)
```

(from General/RoomMN/stuff/src/main/java/com/commonsware/android/room/dao/Customer.java)

Here, we declare four properties.

`tableName` renames the table to something that is more unique to this situation, incorporating both "customer" and "category" in the name. That way, if we do wind up with `CategoryJoin` elsewhere, we can avoid table name collisions.

`primaryKeys` is used, instead of `@PrimaryKey`, because we need a composite key. The uniqueness is determined by the combination of the IDs of the `Customer` and `Category`, held in `customerId` and `categoryId` columns, respectively.

A join entity will need foreign keys back to both entities that it is joining. So, here, we have two `@ForeignKey` annotations for the `foreignKeys` property, connecting to both `Customer` and `Category` by their respective IDs. We also use onDelete=CASCADE, so if the parent entity (Customer or Category) is deleted, we also delete all join entities associated with that parent.

And, since Room does not automatically add indices for foreign key columns, we add them ourselves, for each parent entity ID individually, so we can rapidly find all of the join entity instances for a given Customer or Category.

Implementing DAO Methods

In addition to setting up the join entity, we need to leverage it in our DAO. Otherwise, the join entity is pointless.

Adding and Removing Relations

In many ORMs, where relations are directly implemented on model objects, you connect objects by direct manipulation. In our case, a Customer might have addCategory() and removeCategory() methods, and Category might have addCustomer() and removeCustomer().

Since Room models foreign keys, not relations, that's not how we connect a Customer and a Category. Instead, we do it much the same way as you would with plain SQL: @insert and @Delete Customer.CategoryJoin instances representing a particular customer-category connection.

And, to that end, we have suitable DAO methods for this:

```
@Insert
void insert(Customer.CategoryJoin... joins);

@Delete
void delete(Customer.CategoryJoin... joins);
```

(from General/RoomMN/stuff/src/main/java/com/commonsware/android/room/dao/StuffStore.java)

And, to connect a specific Customer instance to a specific Category instance, we set up the Customer.CategoryJoin instance and insert() it:

```
    tags.add("scuplture");
    tags.add("bronze");
    tags.add("slow-pay");

    final LocationColumns loc=new LocationColumns(40.7047282, -74.0148544);

    final Customer firstCustomer=new Customer("10001", "Fearless Girl", loc, tags);
    tags.remove("slow-pay");
    tags.add("large");

    final Customer secondCustomer=new Customer("10002", "Charging Bull", loc, tags);

    store.insert(firstCustomer, secondCustomer);

    final Category root=new Category("Root!");
    final Category child=new Category("Child!", root.id);

    store.insert(root, child);

    final Customer.CategoryJoin join=
      new Customer.CategoryJoin(root.id, secondCustomer.id);

    store.insert(join);
```

(from General/RoomMN/stuff/src/androidTest/java/com/commonsware/android/room/dao/DaoTests.java)

Fetching Via the Join

If an ORM offers `addCategory()` and `removeCustomer()` methods, presumably that ORM also offers `getCategories()` on `Customer` and `getCustomers()` on `Category`, to identify the members of a relation with a specific entity.

Again, Room does not work that way.

Instead, we crack open our SQL syntax reference and craft an `INNER JOIN` ourselves, to use in a `@Query` method:

```
  @Query("SELECT categories.* FROM categories\n"+
    "INNER JOIN customer_category_join ON categories.id=customer_category_join.categoryId\n"+
    "WHERE customer_category_join.customerId=:customerId")
  List<Category> categoriesForCustomer(String customerId);

  @Query("SELECT Customer.* FROM Customer\n"+
    "INNER JOIN customer_category_join ON Customer.id=customer_category_join.customerId\n"+
    "WHERE customer_category_join.categoryId=:categoryId")
  List<Customer> customersForCategory(String categoryId);
```

(from General/RoomMN/stuff/src/main/java/com/commonsware/android/room/dao/StuffStore.java)

Here we have methods that return the members of a specific relation, so we can find the categories for a `Customer` or the customers for a `Category`. And the DAO methods return sensible data types. But, it is our job to set up the SQL.

So, in the case of `categoriesForCustomer()`, our SQL:

- Retrieves all columns from the `categories` table…

- …where we `JOIN` with `customer_category_join` based on the IDs…
- …and find all those where the join entity points to a specific customer ID

Where's That Good Ol' Object Feel?

By this point, some of you may be wanting to dismiss Room outright, as being too thin of a wrapper around the SQL. Certainly, Room has, um, room for improvement.

However, a lot of the pain may come from what you are thinking that entities represent. Many developers, particularly those using ORMs in other environments, will think of entities as being model objects.

That's not the best approach with Room.

Instead, consider entities to be more akin to data transfer objects (DTOs). They are a means of getting data from point (SQLite) to point (your application code), and not much more.

For example, pretend that the SQLite database was on a server somewhere, and you wrapped it in a Web service which you accessed from your Android app via Retrofit or some similar library. Developers are used to thinking of the POJOs that you might get back from a REST call to be DTOs, objects that model the Web service response, not necessarily modeling any business logic within the app.

Room is much the same. The entities are DTOs from the relational data store to your app, but may or may not line up with how you would want to represent that data in memory as "real" model objects. So, just as you sometimes convert the Retrofit response object graph into something more useful, you sometimes convert the Room response POJOs into something more useful.

Consider the DAO and the entities to be a low-level API, much as you might consider Retrofit or other REST access layers. If you need a richer object representation of your data, wrap the DAO and entities in some sort of repository object, one that knows more about your app's needs and can perform the conversions as needed. That repository can also handle details like transactions, to keep your business logic clean from any details about how the data storage is accomplished. The ultimate goal would be to replace one repository implementation (e.g., using Room) with another (e.g., using Realm or Couchbase

Mobile or some non-SQL solution), without having to change anything related to the business logic itself.

LiveData Transformations

Sometimes, the data that you want is not the data that you get:

- You want to capitalize those names before showing them in a list
- You want to restrict the results to some subset of what you are receiving
- You do not need the data, but rather some calculation made upon batches of the data, grouped by some key
- And so on

The `LiveData` system has some limited support for "transformations", which help you adapt an existing `LiveData` into one that changes the data to better suit your needs. You can also create your own transformations, if desired. In this chapter, we will explore all of this.

The Bucket Brigade

`LiveData` is designed to be a simplified form of a reactive framework like RxJava.

Anyone who has looked at RxJava code knows that it has a tendency towards long chains of calls, to configure a stream of data, and sometimes to modify that stream along the way.

For example, you will find code like:

```
Observable<String> observable=Observable
  .create(new WordSource(getActivity()))
  .subscribeOn(Schedulers.io())
  .map(s -> (s.toUpperCase()))
  .observeOn(AndroidSchedulers.mainThread())
  .doOnComplete(() -> {
    Toast.makeText(getActivity(), R.string.done, Toast.LENGTH_SHORT)
      .show();
  });
```

Here, we:

- Request a roster of words (`create(new WordSource(getActivity()))`)
- Ask to retrieve that roster on a background thread, as it involves disk I/O (`subscribeOn(Schedulers.io())`)
- Convert the words to uppercase (`map(s -> (s.toUpperCase()))`)
- Ask to get the results on the main application thread, so we can update our UI with these words (`observeOn(AndroidSchedulers.mainThread())`)
- Show a `Toast` when we are done processing the words (`doOnComplete() ...`)

(if you are interested in learning more about RxJava, see [*The Busy Coder's Guide to Android Development*])(https://commonsware.com/Android)

In particular, `map()` is a transformation "operator", in Rx terms. `map()` takes an object from our stream of data (in this case, a word) and transforms it into something else, which flows downstream to the subsequent chained calls. In this case, `map()` transforms a `String` into a `String`, where the "transformation" is converting the input `String` to uppercase to use as the output `String`.

RxJava has a dozens of such operators. In contrast, `LiveData` has two, and we will implement a third ourselves to see how that is accomplished.

Mapping Data to Data

Both RxJava and `LiveData` offer a `map()` transformation. As seen in the preceding section, a `map()` converts an item of data from the stream (e.g., a `String`) to some other item of data to flow downstream (e.g., an uppercase `String`).

However, whereas `map()` is a method on RxJava's `Observable` and related classes, with `LiveData`, the transformations are held in a separate `Transformations` class.

For example, suppose we have a DAO method like:

```
@Query("SELECT * FROM Customer")
LiveData<List<Customer>> allCustomers();
```

Here, we are expecting a stream of results. However, Room only supports returning one item in the stream: a list of `Customer` objects.

Suppose, though, we do not need the `Customer` objects, but instead need their IDs. The simplest and most performant solution would be to have a different DAO method:

```
@Query("SELECT id FROM Customer")
LiveData<List<String>> allCustomerIds();
```

However, that does not use `Transformations`, and so it is boring. Plus, not every possible transformation is simply cutting a POJO down to a single field from that POJO.

The `Transformations` equivalent would be something like this:

```
LiveData<List<String>> liveCustomerIds=
  Transformations.map(store.allCustomers(),
    new Function<List<Customer>, List<String>>() {
      @Override
      public List<String> apply(List<Customer> customers) {
        ArrayList<String> result=new ArrayList<>();

        for (Customer customer : customers) {
          result.add(customer.id);
        }

        return(result);
      }
    });
```

`map()` takes two parameters: a `LiveData` of the stream to manipulate, and a `Function` that converts items from that stream from one data type to another.

Here is where Room's insistence on a single-object response becomes a pain. If this were a stream of `Customer` objects, our `Function` could just get the `id` from the `Customer` and return it. But we do not have a stream of `Customer` objects — we have a stream of a list of `Customer` objects. That means we need to return a list of customer IDs, requiring allocating a new `ArrayList` and iterating over each `Customer` to add its `id` to that list.

This would be a little bit cleaner with Java 8 lambda expressions. The proof of this is left as an exercise for the reader, until sometime after Android Studio 3.0 ships, at which time this book will be updated to use lambda expressions.

Mapping Data to… LiveData?

So now we have a list of `Customer` IDs. Suppose that we now want to retrieve the categories associated with all of the `Customer` entities. That requires another database request via our DAO:

```
@Query("SELECT categories.* FROM categories\n"+
  "INNER JOIN customer_category_join ON
categories.id=customer_category_join.categoryId\n"+
  "WHERE customer_category_join.customerId IN (:customerIds)")
LiveData<List<Category>> categoriesForCustomers(List<String> customerIds);
```

And if we are on the main application thread — as is typical when working with `LiveData` results — we need the DAO to return another `LiveData`.

In principle, you could use `map()` for this. However, for this scenario, there is the oddly-named `switchMap()`. This is analogous to the equally-oddly-named `flatMap()` of RxJava and says that the objects being created via the mapping are themselves `LiveData`. This help the `LiveData` system keep everything in sync, particularly across lifecycle events.

So, given the `liveCustomerIds` from the preceding section, we can get the categories via:

```
final LiveData<List<Category>> liveCategories=
  Transformations.switchMap(liveCustomerIds,
    new Function<List<String>, LiveData<List<Category>>>() {
      @Override
      public LiveData<List<Category>> apply(List<String> customerIds) {
        return(store.categoriesForCustomers(customerIds));
      }
    });
```

And, if we arrange to `observe()` that `liveCategories` object, we will be called with `onChanged()` when the list of `Category` objects is available, after the initial database I/O to get the customers, then the secondary database I/O to get the categories for those customers.

Unfortunately, this does not work, due to a bug in Room.

Writing a Transformation

Another RxJava transformation operator is `filter()`. This takes a stream of objects and a function that tests each object and returns `true` for the ones to be sent downstream. The ones that test out to `false` are dropped. Hence, the stream becomes filtered by whatever rule is encoded in that function.

`Transformations` does not have a `filter()` method, but we can write one, to see what a transformation method looks like.

Earlier in the book, we had the `LiveSensor` sample, where we had a `LiveData` reporting sensor events, specifically the light level. The `General/LiveFilter` sample project is a clone of that project, one that introduces a filter, to only report those readings that fall between 40 and 60 lux.

To that end, we have a `LiveTransmogrifiers` class that serves as a home for our transformation methods:

```
package com.commonsware.android.livedata;

import android.arch.lifecycle.LiveData;
import android.arch.lifecycle.MediatorLiveData;
import android.arch.lifecycle.Observer;
import android.support.annotation.MainThread;
import android.support.annotation.NonNull;
import android.support.annotation.Nullable;

class LiveTransmogrifiers {
  interface Confirmer<T> {
    boolean test(T thingy);
  }

  @MainThread
  static <X> LiveData<X> filter(@NonNull LiveData<X> source,
                                @NonNull final Confirmer<X> confirmer) {
    final MediatorLiveData<X> result=new MediatorLiveData<>();

    result.addSource(source, new Observer<X>() {
      @Override
      public void onChanged(@Nullable X x) {
        if (confirmer.test(x)) {
          result.setValue(x);
        }
      }
    });

    return(result);
  }
}
```

(from General/LiveFilter/app/src/main/java/com/commonsware/android/livedata/LiveTransmogrifiers.java)

The RxJava `filter()` operator uses a `Predicate` as the function for testing an object to determine if it should be passed or not. Unfortunately, `Predicate` is part of the Java 8 classes added in Android 7.0, and so it is unavailable for older devices. So, we have a `Confirmer` interface that fills that role. The `test()` method on a `Confirmer` needs to return `true` for objects that should pass the filter, `false` otherwise.

The `filter()` method on `LiveTransmogrifiers` takes a `LiveData` of some type and a `Confirmer` of that type. It then uses a `MediatorLiveData`, which is a `LiveData` object that can chain onto an existing `LiveData` and expose the `onChanged()` method for outside parties to use. In this case, our `onChanged()` method uses the `Confirmer` to see if the new object passes the `test()`, and if it does, we call `setValue()` on the `MediatorLiveData` to have that object flow along to anything that observes that `MediatorLiveData`. `filter()` then returns that `MediatorLiveData`. The net effect is as if `filter()` wraps the original `LiveData` in another `LiveData` that applies our filtering rule.

We can now use `filter()` to limit the readings that we get from the sensor:

```
    final LiveData<SensorLiveData.Event> filtered=
      LiveTransmogrifiers.filter(state.sensorLiveData,
      new LiveTransmogrifiers.Confirmer<SensorLiveData.Event>() {
        @Override
        public boolean test(SensorLiveData.Event event) {
          return(event.values[0]>40 && event.values[0]<60);
        }
      });

    filtered.observe(this, new Observer<SensorLiveData.Event>() {
      @Override
      public void onChanged(@Nullable SensorLiveData.Event event) {
        adapter.add(event);
      }
    });
```

(from General/LiveFilter/app/src/main/java/com/commonsware/android/livedata/MainActivity.java)

We pass our original SensorLiveData to filter(), along with a Confirmer that sees if the light level is between 40 and 60. Then, we observe the results of the filter() call and only add those objects — not every reading from the SensorLiveData — to the EventLogAdapter.

The net result, if you compare and contrast the output of this sample with the original, is that while the original reports everything, this new sample only reports a subset of the data.

Do We Really Want This?

LiveData was not set up to have a vast library of transformations, the way that RxJava has its vast library of operators. map() and switchMap() are almost afterthoughts. And while Google may not add many more transformations to the Transformations class, undoubtedly somebody will create a library with implementations of filter() and a handful of other RxJava-style operators.

However, those libraries will be limited, because LiveData itself is not as rich a framework as is RxJava. There is no notion in LiveData of propagating errors, or indicating that a stream is completed. Some RxJava operators will be difficult or impossible to implement as a result.

And this is by design.

LiveData is designed to be simple and lifecycle aware. That's it. If your needs transcend what LiveData can handle well, consider migrating to RxJava. Conversely, if LiveData handles everything that you need, you can skip RxJava's complexity.