*In San Francisco? Join us for a* ***drink with the entire Realm team + roadmap sneak peek*** *on Tuesday March 24.*

(http://www.meetup.com/realm-sf/events/220589206/)

*Realm is a mobile database: a replacement for SQLite & Core Data. This is our documentation.*

---

Documentation available in the following languages — want to help us with another language? Email Arwa (mailto:aj@realm.io)!

English (/docs/java/0.80.0/)

Español (/es/docs/java/0.78.0/)

Français (/fr/docs/java/0.72.0/)

日本語 (/jp/docs/java/0.72.0/)

한국어 (/kr/docs/java/0.79.0/)

# Getting started

## Prerequisites

- We do not support Java outside of Android at the moment.
- Android Studio >= 0.8.6 — to use Realm from Eclipse, see below.
- A recent version of the Android SDK.
- JDK version >=7.
- We support all Android versions since API Level 9 (Android 2.3 Gingerbread & above).

## Installation

Android Studio Eclipse

You can either use Maven or manually add a Jar to your project.

### Maven

1. Make sure your project uses *jcenter* as a dependency repository (default on latest version of the Android Gradle plugin)
2. Add `compile 'io.realm:realm-android:0.80.0'` to the dependencies of your project

3. In the Android Studio menu: Tools->Android->Sync Project with Gradle Files

Jar

1. Download (http://static.realm.io/downloads/java/realm-java-0.80.0.zip) the release package and unzip.
2. Create a new project with Android Studio
3. Copy the `realm-VERSION.jar` folder into `app/libs`
4. In the Android Studio menu: Tools->Android->Sync Project with Gradle Files

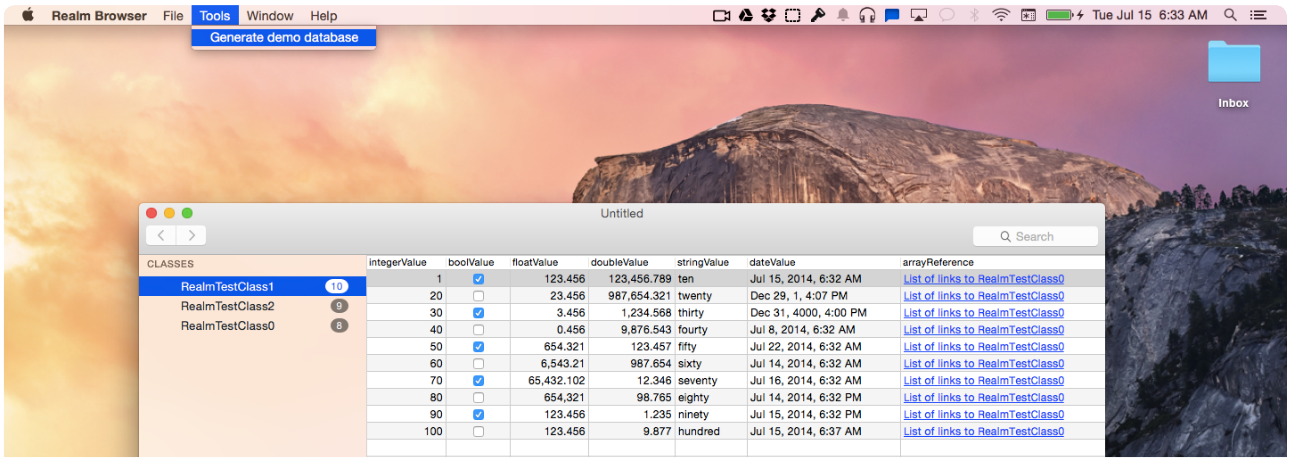
## ProGuard

Realm generates a proxy class for each RealmObject (api/io/realm/RealmObject.html) at compile time. To ensure that these classes can be found after running an obfuscation and static analysis tool like ProGuard (http://developer.android.com/tools/help/proguard.html) add the configuration below to your ProGuard configuration file.

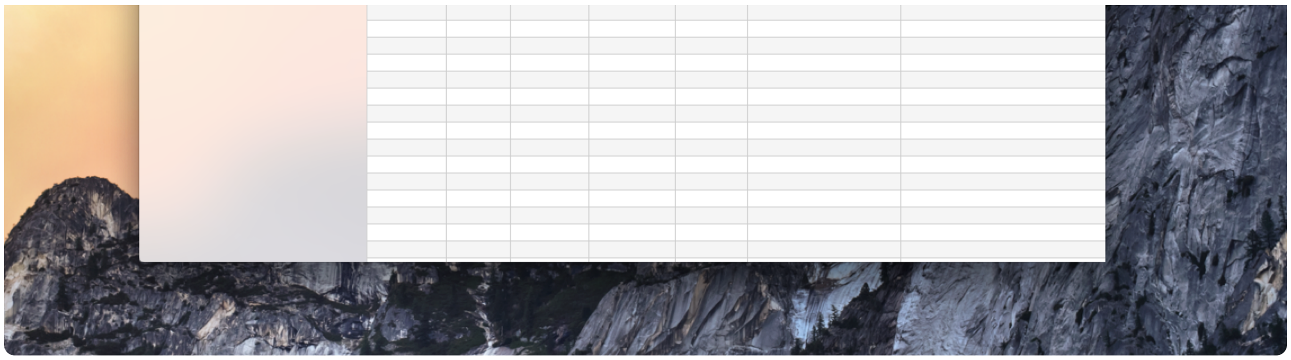
```
-keepnames public class * extends io.realm.RealmObject
-keep class io.realm.** { *; }
-dontwarn javax.**
-dontwarn io.realm.**
```

## Realm Browser

Only available on Mac OS X at the moment! We are working on Windows & Linux versions.



| integerValue | boolValue | floatValue | doubleValue | stringValue | dateValue | arrayReference |
|---|---|---|---|---|---|---|
| 1 | ✓ | 123.456 | 123,456.789 | ten | Jul 15, 2014, 6:32 AM | List of links to RealmTestClass0 |
| 20 | ☐ | 23.456 | 987,654.321 | twenty | Dec 29, 1, 4:07 PM | List of links to RealmTestClass0 |
| 30 | ✓ | 3.456 | 1,234.568 | thirty | Dec 31, 4000, 4:00 PM | List of links to RealmTestClass0 |
| 40 | ☐ | 0.456 | 9,876.543 | fourty | Jul 8, 2014, 6:32 AM | List of links to RealmTestClass0 |
| 50 | ✓ | 654.321 | 123.457 | fifty | Jul 22, 2014, 6:32 AM | List of links to RealmTestClass0 |
| 60 | ☐ | 6,543.21 | 987.654 | sixty | Jul 14, 2014, 6:32 AM | List of links to RealmTestClass0 |
| 70 | ✓ | 65,432.102 | 12.346 | seventy | Jul 16, 2014, 6:32 AM | List of links to RealmTestClass0 |
| 80 | ☐ | 654,321 | 98.765 | eighty | Jul 14, 2014, 6:32 PM | List of links to RealmTestClass0 |
| 90 | ✓ | 123.456 | 1.235 | ninety | Jul 15, 2014, 6:32 PM | List of links to RealmTestClass0 |
| 100 | ☐ | 123.456 | 9.877 | hundred | Jul 15, 2014, 6:37 AM | List of links to RealmTestClass0 |



We also provide a standalone app to read and edit .realm databases.

You can find it in our **Cocoa release zip (http://static.realm.io/downloads/cocoa/realm-cocoa-0.91.1.zip)** under `browser/`.

You can generate a test database with dummy data using the menu item **Tools > Generate demo database**.

## API Reference

You can consult our **full API reference (api/)** for all classes, methods & more.

## Examples

The root folder contains a few examples to get you started. You just `Import Project` in Android Studio and hit `run`.

The `RealmIntroExample` in the root folder contains simple examples of how you use the current API. Checkout the source code, as you will only see little output in the Log.

The `RealmGridViewExample` is a trivial app that shows how to use Realm as the backing store for a GridView. It also shows how you could populate the database with JSON.

The `RealmThreadExample` is a simple app that shows how to use Realm in a multithreaded environment.

The `RealmAdapterExample` shows how to use the RealmBaseAdapter (api/io/realm /RealmBaseAdapter.html) to bind RealmResults (api/io/realm/RealmResults.html) to Android lists in a convenient way.

The `RealmJsonExample` illustrates how to use the new Realm JSON facilities.

The `RealmEncryptionExample` shows you how to work with encrypted Realms.

## Getting Help

- Attend our monthly Online Office Hours (http://j.mp/realm-office-hours) to ask questions or show us your app.
- Reproducible Bugs & Feature Requests should be filed directly against our Github repo (https://github.com/realm/realm-java/issues).
- Discussions & Support: realm-java@googlegroups.com (https://groups.google.com/d/forum/realm-java).
- StackOverflow: look for previous questions under the tag #realm (https://stackoverflow.com/questions/tagged/realm?sort=newest) — or open a new one (http://stackoverflow.com/questions/ask?tags=realm).
- Sign up for our Community Newsletter (http://eepurl.com/VEKCn) to get regular tips, learn about other use-cases and get alerted of blogposts and tutorials about Realm.

# Models

Realm data models are defined by implementing something very similar to a traditional Java Bean (http://en.wikipedia.org/wiki/Java_Bean). Simply extend our RealmObject (api/io/realm/RealmObject.html) class and let the Realm annotations processor generate proxy classes.

```
public class User extends RealmObject {

    @PrimaryKey
    private String          name;
    private int             age;

    @Ignore
    private int             sessionId;

    // Standard getters & setters generated by your IDE…
    public String getName() { return name; }
    public void   setName(String name) { this.name = name; }
    public int    getAge() { return age; }
    public void   setAge(int age) { this.age = age; }
    public int    getSessionId() { return sessionId; }
    public void   setSessionId(int dontPersist) { this.sessionId = sessionId; }
}
```

Be aware that the getters and setters will be overridden by the generated proxy class used in the back by RealmObjects (api/io/realm/RealmObject.html), so any custom logic you add to the getters & setters will not actually be executed.

## Field types

Realm supports the following field types: `boolean`, `short`, `int`, `long`, `float`, `double`, `String`, `Date` and `byte[]`. The integer types `short`, `int`, and `long` are all mapped to the same type (`long` actually) within the realm. Moreover, subclasses of `RealmObject` and `RealmList<? extends RealmObject>` are supported to model relationships.

## Ignoring properties

The annotation `@Ignore` implies that a field should not be persisted to disk. Ignored fields are useful if your input contains more fields than your model, and you don't wish to have many special cases for handling these unused data fields.

## Search index

The annotation `@Index` will add a search index to the field. This will make inserts slower and the data file larger but queries will be faster. So it's recommended to only add index when queries need to be faster. Only string fields can currently be indexed (other types will be supported in future release), and it is not possible to remove a search index.

## Primary keys

To promote a field to primary key, you use the annotation `@PrimaryKey`, and the field type has to be either string or integer (`short`, `int` or `long`). It is not possible to use multiple fields (compound key) as a primary key. Using a string field as a primary key implies that the field is indexed (the annotation `@PrimaryKey` implicitely sets the annotation `@Index`).

When Realm objects are created, all fields are set to default values. In order to avoid conflicts with another object with the same primary key, it is suggested to create a standalone object, set the values of the fields, and then copy the object to Realm, using the `copyToRealm()` method.

Using primary keys makes it possible to use the `createOrUpdate()` method, which will look for an existing object with this primary key, an update it if it finds one; if none are found, it will create a new object instead.

Using primary keys have an effect on the performance. Creating and updating object will be a little slower while querying is expected to be a bit faster. It is hard to give numbers as the changes in performance depend on the size of your dataset.

# Writes

Read operations are implicit which means that objects can be accessed and queried at any time. All write operations (adding, modifying, and removing objects) must be wrapped in write transactions. A write transaction can either be committed or cancelled. During the commit, all changes will be written to disk, and the commit will only succeed if all changes can be persisted. By cancelling a write transaction, all changes will be discarded. Using write transactions, your data will always be in a consistent state.

Write transactions are also used to ensure thread safety:

```
// Obtain a Realm instance
Realm realm = Realm.getInstance(this);

realm.beginTransaction();

//... add or update objects here ...

realm.commitTransaction();
```

While working with your RealmObjects (api/io/realm/RealmObject.html) inside a write transaction, you might end up in a situation where you wish to discard the change. Instead of committing it, and then reverting it, you can simply cancel the write transaction:

```
realm.beginTransaction();
User user = realm.createObject(User.class);

//  ...

realm.cancelTransaction();
```

Please note that writes block each other, and will block the thread they are made on if other writes are in progress. This can cause ANR errors if you are doing writes from the UI thread while also doing writes from a background thread. To avoid this, first create objects in memory first outside a transaction, and only perform a simple **Realm.copyToRealm() (api/io/realm/Realm.html#copyToRealm(E))** inside the transaction, which will keep blocking times to a minimum.

Thanks to Realm's MVCC architecture, reads are not blocked while a write transaction is open! This means that unless you need to make simultaneous writes from many threads at once, you can favor larger write transactions that do more work over many fine-grained write transactions. When you commit a write transaction to a Realm, all other instances of that Realm will be notified, and the read implicit transactions will refresh your Realm objects automatically.

Please note read & write access in Realm is **ACID (http://en.wikipedia.org/wiki/ACID)**.

## Creating objects

Because RealmObjects (api/io/realm/RealmObject.html) are strongly tied to a Realm, they should be instantiated through the Realm directly:

```
realm.beginTransaction();
User user = realm.createObject(User.class); // Create a new object
user.setName("John");
user.setEmail("john@corporation.com");
realm.commitTransaction();
```

Alternatively you can create an instance of an object first and add it later using realm.copyToRealm() (api/io/realm/Realm.html#copyToRealm(E)). Realm supports as many custom constructors as you like as long as one of them is a public no arguments constructor (http://docs.oracle.com/javase/tutorial/java/javaOO/constructors.html).

```
User user = new User("John");
user.setEmail("john@corporation.com");

// Copy the object to Realm. Any further changes must happen on realmUser
realm.beginTransaction();
User realmUser = realm.copyToRealm(user);
realm.commitTransaction();
```

When using `realm.copyToRealm()` it is important to remember that only the returned object is managed by Realm, so any futher changes to the original object will not be persisted.

## Transaction blocks

Instead of manually keeping track of `realm.beginTransaction()`, `realm.commitTransaction()`, and `realm.cancelTransaction()` you can use the realm.executeTransaction() (api/io/realm/Realm.html#executeTransaction(io.realm.Realm.Transaction)) method, which will automatically handle begin/commit, and cancel if an error happens.

```
realm.executeTransaction(new Realm.Transaction() {
        @Override
        public void execute(Realm realm) {
                User user = realm.createObject(User.class);
                user.setName("John");
                user.setEmail("john@corporation.com");
        }
});
```

# Queries

All fetches (including queries) are lazy in Realm, and the data is never copied.

Realm’s query engine uses a Fluent interface (https://en.wikipedia.org/wiki/Fluent_interface) to construct multi-clause queries.

To find all users named John or Peter you would write:

```
// Build the query looking at all users:
RealmQuery<User> query = realm.where(User.class);

// Add query conditions:
query.equalTo("name", "John");
query.or().equalTo("name", "Peter");

// Execute the query:
RealmResults<User> result1 = query.findAll();

// Or alternatively do the same all at once (the "Fluent interface"):
RealmResults<User> result2 = realm.where(User.class)
                                  .equalTo("name", "John")
                                  .or()
                                  .equalTo("name", "Peter")
                                  .findAll();
```

This gives you a new instance of the class ` RealmResults (api/io/realm/RealmResults.html), containing the users with the name John or Peter. Objects are not copied - you get a list of references to the matching objects, and you work directly with the original objects that matches your query. The RealmResults (api/io/realm/RealmResults.html) inherits from Java’s AbstractList, and behaves in similar ways. For example, RealmResults (api/io/realm/RealmResults.html) are ordered, and you can access the individual objects through an index.

When a query does not have any matches, the returned RealmResults (api/io/realm /RealmResults.html) object will not be `null`, but the `size()` (api/io/realm /RealmResults.html#size()) method will return 0.

If you wish modify or delete any of the objects in a RealmResults (api/io/realm/RealmResults.html), you must do so in a write transaction.

## Retrieving objects by type

The most basic method for retrieving objects from a Realm is `realm.allObjects()` (api/io/realm/Realm.html#allObjects), which returns a RealmResults (api/io/realm/RealmResults.html) of all instances of the model class being queried.

There are specialized versions of `allObjects()` which offer sorting functionality i.e., you can specify sorting order per field. See `realm.allObjectsSorted()` (api/io/realm/Realm.html#allObjectsorted) for details.

## Conditions

The following hopefully self explanatory conditions are supported:

- `between`, `greaterThan()`, `lessThan()`, `greaterThanOrEqualTo()` & `lessThanOrEqualTo()`
- `equalTo()` & `notEqualTo()`
- `contains()`, `beginsWith()` & `endsWith()`

Not all conditions are applicable for all data types. Please consult the API reference for RealmQuery (api/io/realm/RealmQuery.html) for details.

## Modifiers

String conditions can ignore case for characters A-Z and a-z by using the `CASE_INSENSITIVE` modifier.

## Logical Operators

Each condition is implicitly logical-and together. Logical-or must be applied explicitly with `or()`.

You can also also group conditions with “parentheses” to specify order of evaluation: `beginGroup()` is your “left parenthesis” and `endGroup()` your “right parenthesis”:

```
RealmResults<User> r = realm.where(User.class)
                            .greaterThan("age", 10)  //implicit AND
                            .beginGroup()
                                .equalTo("name", "Peter")
                                .or()
                                .contains("name", "Jo")
                            .endGroup()
                            .findAll();
```

Futhermore, it is possible to negate a condition with `not()`. The `not()` operator can be used with `beginGroup()`/`endGroup()` to negate subconditions only.

## Sorting

Once you have done your query, you can sort the results like this:

```
RealmResults<User> result = realm.where(User.class).findAll();
result.sort("age"); // Sort ascending
result.sort("age", RealmResults.SORT_ORDER_DESCENDING);
```

## Chaining Queries

Since results are never copied and computed on request, you can very efficiently chain queries to gradually filter your data:

```
RealmResults<User> teenagers = realm.where(User.class).between("age", 13, 20).findAll();
RealmResults<User> firstJohn = teenagers.where().equalTo("name", "John").findFirst();
```

## Aggregation

A RealmResults (api/io/realm/RealmResults.html) also has various aggregation methods:

```
long   sum     = result.sum("age").longValue();
long   min     = result.min("age").longValue();
long   max     = result.max("age").longValue();
double average = result.average("age");

long   matches = result.size();
```

## Iterations

To iterate through all objects in a RealmResults (api/io/realm/RealmResults.html) you can take advantage of `Iterable`:

```
for (User u : result) {
    // ... do something with the object ...
}
```

or use a traditional `for` loop:

```
for (int i = 0; i < result.size(); i++) {
    User u = result.get(i);
    // ... do something with the object ...
}
```

## Deletion

You can delete the results of a query from the Realm:

```
// All changes to data must happen in a transaction
realm.beginTransaction();

// remove single match
result.remove(0);
result.removeLast();

// remove a single object
Dog dog = result.get(5);
dog.removeFromRealm();

// Delete all matches
result.clear();

realm.commitTransaction()
```

# Realms

Realms are our equivalent of a database: they contain different kinds of objects, and map to one file on disk.

## The Default Realm

You may have noticed so far that we have always initialized our realm variable by calling `Realm.getInstance(Context context)`. This static constructor will return a Realm instance for your thread, that maps to a file called `default.realm` located in `Context.getFilesDir()`.

The file is located at the root of the writable directory for your application. As Realm uses internal storage for the default Realm, your app does not require any read or write permissions. In most cases, you can find the file in the folder `/data/data/files/`.

It is always possible to obtain the absolute path of a realm by using the `realm.getPath()` method.

It is important to note that Realm instances are thread singletons, meaning that the static constructor will return the same instance for every thread.

## Other Realms

It's sometimes useful to have multiple realms, persisted at different locations, for example if you have different data groupings, different databases per feature, or you need to package some read-only files with your app, separate from the database your users will be editing.

```
Realm realm = Realm.getInstance(this, "allmymovies.realm");
```

## Using a Realm across Threads

The only rule to using Realm across threads is to remember that **Realm, RealmObject or RealmResults instances cannot be passed across threads**. When you want to access the same data from a different thread, you should simply obtain a new Realm instance (i.e. `Realm.getInstance(Context context)` or its cousins) and get your objects through a query. The objects will map to the same data on disk, and will be readable & writeable from any thread!

## Closing Realm instances

`Realm` implements `Closeable` in order to take care of native memory deallocation and file descriptors so it is important to remember to close your Realm instances when you are done with them.

`Realm` instances are reference counted, which means that if you call `getInstance()` twice in a thread, you will also have to call `close()` twice as well. This allows you to implement `Runnable` classes without having to worry in what thread they will be executed: simply start it with a `getInstance()` and end it with a `close()` and you are good to go!

For the UI thread the easiest way is to execute `realm.close()` in the `onDestroy()` method.

For `AsyncTask` this is a good pattern (as it is for any `Closeable`):

```
protected Long doInBackground(Context... contexts) {
    Realm = null;
    try {
        realm = Realm.getInstance(contexts[0]);

        // ... Use the Realm instance
    } finally {
        if (realm != null) {
            realm.close();
        }
    }
}
```

If you need to create another `Looper` thread other than the UI one you can use this pattern:

```
public class MyThread extends Thread {
    private final Context;

    public MyThread(Context context) {
        this.context = context;
    }

    public void run() {
        Looper.prepare();
        Realm realm = null;
        try {
            realm = Realm.getInstance(context);

            //... Setup the handlers using the Realm instance
            Lopper.loop();
        } finally {
            if (realm != null) {
                realm.close();
            }
        }
    }
}
```

If you have the luck to work on an app with `minSdkVersion >= 19` then you can use try-with-resources:

```
try (Realm realm = Realm.getInstance(context)) {
        // No need to close the Realm instance manually
}
```

## Auto-Refresh

If a Realm instance has been obtained from a thread that is associated with a Looper (http://developer.android.com/reference/android/os/Looper.html) (the UI thread is by default) then the Realm instance comes with an auto-refresh feature. This means that the Realm instance will be automatically updated to the latest version on every occurrence of the event loop. This is a handy feature that allows you to keep your UI constantly updated with the latest content with very little effort.

If you get a Realm instance from a thread that does not have a `Looper` attached, then such instance will not auto-update unless the `refresh()` method is called. It is important to note that having to hold on to an old version of your data is expensive in terms of memory and disk space and the cost increases with the number of versions between the one being retained and the latest. This is why it is important to close the Realm instance as soon as you are done with it in the thread.

If you want to be sure wether your Realm instance has auto-refresh activated or not you can use the `isAutoRefresh()` method.

# Relationships

Any two RealmObjects (api/io/realm/RealmObject.html) can be linked together.

```
public class Email extends RealmObject {
    private String address;
    private boolean active;
    // ... setters and getters left out
}

public class Contact extends RealmObject {
    private String name;
    private Email email;
    // ... setters and getters left out
}
```

Relationships are generally cheap in Realm. This means that following a link is not expensive in terms of speed, and the internal presentation of relationships is highly efficient in terms of memory consumption.

## Many-to-One

Simply declare a property with the type of one of you RealmObject (api/io/realm /RealmObject.html) subclasses:

```
public class Contact extends RealmObject {
    private Email email;
    // Other fields…
}
```

Each contact (instance of `Contact`) have either 0 or 1 email (instance of `Email`). In Realm, nothing prevent you from using the same email object in multiple contacts, and the model above can be a many-to-one relationship but often used to model one-to-one relationships.

## Many-to-Many

You can establish a relationship to 0, 1 or more objects from a single object via a RealmList (api/io/realm/RealmList.html) field declaration

```
public class Contact extends RealmObject {
    private RealmList<Email> emails;
    // Other fields…
}
```

RealmList (api/io/realm/RealmList.html) are basically containers of RealmObjects (api/io/realm/RealmObject.html), that behave very much like a regular Java `List`. There is no limitation in Realm to using the same object twice (or more) in different RealmList (api/io/realm/RealmList.html), and you can use this to model both one-to-many, and many-to-many relationsships.

You can add standard getters & setters to access the data in the link.

```
realm.beginTransaction();
Contact contact = realm.createObject(Contact.class);
contact.setName("John Doe");

Email email1 = realm.createObject(Email.class);
email1.setAddress("john@example.com");
email1.setActive(true);
contact.getEmails().add(email1);

Email email2 = realm.createObject(Email.class);
email2.setNumber("jd@example.com");
email2.setActive(false);
contact.getEmails().add(email2);

realm.commitTransaction();
```

It is possible to declare recursive relationships which can be useful when modelling certain types of data.

```
public class Person extends RealmObject {
    private String name;
    private RealmList<Person> friends;
    // Other fields…
}
```

Use recursive relationships with care as Realm does not currently have cycle detection and you can easily end in infinite loops.

## Link queries

It is possible to query links or relationships. Consider the model above. If you wish to find all contacts with an active email address, you can do:

```
RealmResults<Contact> contacts = realm.where(Contact.class).equalTo("email.active", true).f
```

First of all, notice that the field name in the `equalsTo` condition contains the path through the relationships (separated by period `.`).

The query above should be read "give me all contacts with at least one active email address". It is important to understand that the result will contain the `Email` objects which do not fulfill the condition since they are part of the `Contact` objects.

# JSON

It is possible to add RealmObjects (api/io/realm/RealmObject.html) represented as JSON directly to Realm whether they are represented as a String, a **JSONObject (http://developer.android.com/reference/org/json/JSONObject.html)** or a **InputStream (http://developer.android.com/reference/java/io/InputStream.html)**. Realm will ignore any properties in the JSON not defined by the RealmObject (api/io/realm/RealmObject.html). Single objects are added through **Realm.createObjectFromJson() (api/io/realm/Realm.html#createObjectFromJson-java.lang.Class-java.lang.String-)** while lists of objects are added using **Realm.createAllFromJson() (api/io/realm/Realm.html#createAllFromJson-java.lang.Class-**

java.lang.String-).

```
// Insert from a string
realm.beginTransaction();
realm.createObjectFromJson(City.class, "{ city: \"Copenhagen\", id: 1 }");
realm.commitTransaction();

// Insert multiple items using a InputStream
InputStream is = new FileInputStream(new File("path_to_file"));
realm.beginTransaction();
try {
    realm.createAllFromJson(City.class, is);
    realm.commitTransaction();
} catch (IOException e) {
    realm.cancelTransaction();
}
```

# Notifications

If you have a backgroud thread adding data to a Realm, your UI or other threads can get notified of changes in a realm by adding a listener, which is executed when the Realm is changed (by another thread or process):

```
realm.addChangeListener(new RealmChangeListener() {
    @Override
    public void onChange() {
        // ... do something with the updates (UI, etc.) ...
    }
});
```

You can easily close all listeners when needed:

```
realm.removeAllChangeListeners();
```

## Migrations

Migrations are a **work in progress**. The feature is fully functional, but the current interface is cumbersome, and will be rewritten soon.

When working with any database, it is likely your data models (i.e. your database schema) will change over time. Since data models in Realm are defined as standard Objects, changing them is as easy as changing the interface of the corresponding RealmObject (api/io/realm/RealmObject.html) subclass.

Just changing your code to the new definition will work fine, if you have no data stored on disk under the old database schema. But if you do, there will be a mismatch between what Realm sees defined in code & the data Realm sees on disk, so an exception will be thrown.

We provide built-in methods so you can upgrade your schema on disk, and the data you stored for previous versions of the schema. See our **migrationSample app (https://github.com/realm/realm-java/tree/master/examples/migrationExample)** for details.

If there is no file on disk when Realm launches, no migration is needed, and Realm will just create a new .realm file & schema based on the latest models defined in your code. This means that if you are in the middle of development and changing your schema very often, *and you are OK with losing all your data*, you can delete your .realm file on disk (and the entire dataset it contained!) instead of having to write a migration. This can be helpful when tinkering with models early in the development cycle of your app.

## Encryption

Please take note of the **Export Compliance** section of our LICENSE, as it places restrictions against the usage of Realm if you are located in countries with an export restriction or embargo from the United States.

The Realm file can be stored encrypted on disk by passing a 256-bit encryption key to `Realm.create()`:

```
byte[] key = new byte[32];
new SecureRandom().nextBytes(key);
Realm realm = Realm.create(this, key);

// ... use the Realm as normal ...
```

This ensures that all data persisted to disk is transparently encrypted and decrypted with standard AES-256 encryption. The same encryption key must be supplied each time a Realm instance for the file is created.

See examples/encryptionExample (https://github.com/realm/realm-java/tree/master/examples/encryptionExample) for a complete example of how to securely store keys between runs in the Android KeyStore so that other applications cannot read them.

Using Encryption requires building Realm from source.

1. Follow the normal build instructions (https://github.com/realm/realm-java#building-realm), but before running `./gradlew assemble`, add the line `encryption=true` to `local.properties`.
2. After building Realm, replace the copy of `realm-<VERSION>.aar` in your project with the one found at `realm/build/outputs/aar/realm-<VERSION>.aar`.

## Adapter

Realm provides an abstract utility class to help binding data coming from RealmResults (api/io/realm/RealmResults.html) to UI widgets. The RealmBaseAdapter (api/io/realm/RealmBaseAdapter.html) class will take care of all required wiring if you implement the `getView()` method:

```
public class MyAdapter extends RealmBaseAdapter<TimeStamp> implements ListAdapter {

    private static class MyViewHolder {
        TextView timeStamp;
    }

    public MyAdapter(Context context, int resId,
                     RealmResults<TimeStamp> realmResults,
                     boolean automaticUpdate) {
        super(context, realmResults, automaticUpdate);
    }


    @Override
    public View getView(int position, View convertView, ViewGroup parent) {
        ViewHolder viewHolder;
        if (convertView == null) {
            convertView = inflater.inflate(android.R.layout.simple_list_item_1,
                                           parent, false);
            viewHolder = new ViewHolder();
            viewHolder.timestamp = (TextView) convertView.findViewById(android.R.id.text1);
            convertView.setTag(viewHolder);
        } else {
            viewHolder = (ViewHolder) convertView.getTag();
        }

        TimeStamp item = realmResults.get(position);
        viewHolder.timestamp.setText(item.getTimeStamp());
        return convertView;
    }

    public RealmResults<TimeStamp> getRealmResults() {
        return realmResults;
    }
}
```

## Other libraries

This section describes how you can integrate Realm with other commonly used libraries for Android.

## GSON

GSON (https://code.google.com/p/google-gson/) is a library created by Google for deserializing and serializing JSON. When using Realm with GSON 2.3.1 (latest version), you have to specify an ExclusionStrategy (http://google-gson.googlecode.com/svn/trunk/gson/docs/javadocs/com/google/gson/ExclusionStrategy.html).

```
Gson gson = new GsonBuilder()
        .setExclusionStrategies(new ExclusionStrategy() {
            @Override
            public boolean shouldSkipField(FieldAttributes f) {
                return f.getDeclaringClass().equals(RealmObject.class);
            }

            @Override
            public boolean shouldSkipClass(Class<?> clazz) {
                return false;
            }
        })
        .create();

String json = "{ name : 'John', email : 'john@corporation.com' }";
User user = gson.fromJson(json, User.class);
```

You can also see an example of how GSON can work with Realm in our GridViewExample (https://github.com/realm/realm-java/blob/master/examples/gridViewExample/src/main/java/io/realm/examples/realmgridview/GridViewExampleActivity.java).

## Otto

Otto (http://square.github.io/otto/) is an event bus from Square (https://github.com/square). Otto works with Realm out of the box, but there are some timing and thread issues to be aware of.

Otto normally receives an event on the same thread it was sent. This means that it is possible to add a RealmObject (api/io/realm/RealmObject.html) to an event and read its data in the receiver method without any problems. However if you are using the hack described here (https://github.com/square/otto/issues/38) to always post on the main thread, you cannot have RealmObjects (api/io/realm/RealmObject.html) as part of your event data unless you always send events from the UI thread also. This is normally done to be able to manipulate UI elements in response to an event which is only possible from the main thread.

When a RealmObject (api/io/realm/RealmObject.html) is changed in one thread, Realm schedules a refresh of Realm data on other threads using a Handler (http://developer.android.com/reference/android/os/Handler.html). `Otto.post(event)` on the other hand, dispatch events immediately. So if you post an event to another thread to notify about changes to Realm data, you will have to manually call `realm.refresh()` to get the latest data.

```
@Subscribe
public void handleEvent(OttoEvent event) {
    realm.refresh();
    // Continue working with Realm data loaded in this thread
}
```

## Retrofit

Retrofit (http://square.github.io/retrofit/) is a library from Square (https://github.com/square) that makes it easy to work with a REST API in a typesafe manner.

As Retrofit uses GSON internally, it also needs a properly configured GsonConverter if you want to deserialize network JSON data to RealmObjects (api/io/realm/RealmObject.html).

```
Gson gson = new GsonBuilder()
        .setExclusionStrategies(new ExclusionStrategy() {
            @Override
            public boolean shouldSkipField(FieldAttributes f) {
                return f.getDeclaringClass().equals(RealmObject.class);
            }

            @Override
            public boolean shouldSkipClass(Class<?> clazz) {
                return false;
            }
        })
        .create();

// Configure Retrofit to use the proper GSON converter
RestAdapter restAdapter = new RestAdapter.Builder()
    .setEndpoint("https://api.github.com")
    .setConverter(new GsonConverter(gson))
    .build();

GitHubService service = restAdapter.create(GitHubService.class);
```

Retrofit does not automatically add objects to Realm, instead you must manually add them using the `realm.copyToRealm()` method.

```
GitHubService service = restAdapter.create(GitHubService.class);
List<Repo> repos = service.listRepos("octocat");

// Copy elements from Retrofit to Realm to persist them.
realm.beginTransaction();
List<Repo> realmRepos = realm.copyToRealm(repos);
realm.commitTransaction();
```

### Robolectric

Robolectric (http://robolectric.org) is a library that allows you to run JUnit tests directly in the JVM instead of in a phone or emulator. Currently Robolectrics does not support native libraries like those that are bundled with Realm. This means that for now it is not possible to test Realm using Robolectric.

You can follow the feature request here: https://github.com/robolectric/robolectric/issues/1389 (https://github.com/robolectric/robolectric/issues/1389)

## Next Steps

Take a look at our examples (https://github.com/realm/realm-java/tree/master/examples) to see Realm used in practice in an app.

Happy hacking! You can always talk to a live human developer on realm-java (https://groups.google.com/d/forum/realm-java).

## Current limitations

Realm is currently in beta and we are continuously adding features and fixing issues while working towards a 1.0 release. Until then, we've compiled a list of our most commonly hit limitations.

Please refer to our GitHub issues (https://github.com/realm/realm-java/issues) for a more comprehensive list of known issues.

## General

Realm aims to strike a balance between flexibility and performance. In order to accomplish this goal, realistic limits are imposed on various aspects of storing information in a Realm. For example:

1. The upper limit of class names is 57 characters. Realm for Android prepend `class_` to all names, and the browser will show it as part of the name.
2. The length of field names has a upper limit of 63 character.
3. The dates are truncated with a precision of one second. In order to maintain compability between 32 bits and 64 bits devices, it is not possible to store dates before 1900-12-13 and after 2038-01-19.
4. Nested transactions are not supported, and an exception is throw if it is detected.
5. Strings and byte arrays (`byte[]`) cannot be larger than 16 MB.
6. Case insensitive string matches in queries are only supported for character sets in 'Latin Basic', 'Latin Supplement', 'Latin Extended A', 'Latin Extended B' (UTF-8 range 0-591).

## Sorting

Sorting is currently limited to character sets in 'Latin Basic', 'Latin Supplement', 'Latin Extended A', 'Latin Extended B' (UTF-8 range 0-591). For other charsets, sorting will not change the RealmResults (api/io/realm/RealmResults.html) object.

## Threads

Although Realm files can be accessed by multiple threads concurrently, you cannot hand over Realms, Realm objects, queries, and results between threads. Moreover, asynchronous queries are currently not supported. The thread example (https://github.com/realm/realm-java/tree/master/examples/threadExample) shows how to use Realm in a multithreading environment.

## null values

It is not possible to store `null` as a value for primitive data types (booleans, integers, floating-point numbers, dates, and strings). Supporting `null` is under active development but until fully supported, we suggest that you add an extra boolean field to capture if a field is null or not.

## Migration

The migration support has not matured yet, and you have to rely on a number of internal classes. We plan to have easy-to-use migration support in the future but currently the best option is to look at the migration example (https://github.com/realm/realm-java/tree/master/examples/migrationExample).

## Realm files cannot be accessed by concurrent processes

Although Realm files can be accessed by multiple threads concurrently, they can only be accessed by a single process at a time. Different processes should either copy Realm files or create their own. Multi-process support is coming soon.

# FAQ

How can find and view the content of my Realm file(s)?
This SO question (http://stackoverflow.com/questions/28478987/how-to-view-my-realm-file-in-the-realm-browser) describes where to find your Realm file. You can then view the content with our Realm Browser.

How big is the Realm library?
Once your app is built for release and split for distribution, Realm should only add about 800KB to your APK in most cases. The releases we distribute are significantly larger because they include support for more architectures (ARM7, ARMv7, ARM64, x86, MIPS). The APK file contains all supported architectures but the Android installer will only install native code for the device's architecture. As a consequence, the installed app is smaller than the size of the APK file.

Should I use Realm in production applications?

Realm has been used in production in commercial products since 2012.

You should expect our Java APIs to break as we evolve the product from community feedback — and you should expect more features & bugfixes to come along as well.

### Do I have to pay to use Realm?

No, Realm for Android is entirely free to use, even in commercial projects.

### How do you guys plan on making money?

We're actually already generating revenue selling enterprise products and services around our technology. If you need more than what is currently in our releases or in **realm-java (http://github.com/realm/realm-java)**, we're always happy to chat **by email (info@realm.io)**. Otherwise, we are committed to developing **realm-java (http://github.com/realm/realm-java)** in the open, and to keep it free and open-source under the Apache 2.0 license.

### I see references to "tightdb" or a "core" in the code, what is that?

TightDB is the old name of our core C++ storage engine. The core is not currently open-source but we do plan on open-sourcing it, also under the Apache 2.0 license, once we've had a chance to clean it, rename it, and finalize major features inside of it. In the meantime, its binary releases are made available under the Realm Core (TightDB) Binary **License (https://github.com/realm/realm-java/blob/master/LICENSE)**.

### What is the difference between a normal Java object and a Realm object?

The main difference is that a normal Java object contains its own data while a Realm object doesn't contain data but get or set the properties directly in the database.

This has a two implications: Realm objects are generally more lightweight than normal Java objects and Realm objects automatically always have the newest data where Java object needs to be updated manually.

### Why do model classes need to extend RealmObject?

The reason for this is so we can add Realm specific functionality to your model classes. Currently only *removeFromRealm()*, but others might show up. It also allows us to use generics in our API's, making it easier to read and use.

### What are the *RealmProxy classes about?

The RealmProxy classes are our way of making sure that the Realm object doesn't contain any data itself, but instead access the data directly in the database.

For every model class in your project, the Realm annotation processor will generate a corresponding RealmProxy class. This class extends your model class and is what is returned when you call *Realm.createObject()*, but from the point of view of the IDE you won't notice any difference.

### Why do I need to have getters and setters for all my fields?

This is a result of how our proxy classes works.

To ensure that all field accesses use the database instead of a local copy of the data, all field accessors are overriden by the proxy classes. This is only possible in Java with private fields with getters and setters.

This also has the implication that you need to use the getters and setters in all private methods as well as when implementing interfaces like Comparable.

It is not an ideal situation and we would love to get rid of this requirement. Some solutions like AspectJ or Javassist might make it possible but we are still investigating the possibilities.

A consequence of proxy classes overriding setters and getters is that you cannot create customised versions of setters and getters. A workaround is to use an ignored field (the `@Ignore` annotation), and use its setter and getter to add your customisation to. A model class found look like:

```java
package io.realm.entities;

import io.realm.RealmObject;
import io.realm.annotations.Ignore;

public class StringOnly extends RealmObject {

    private String name;

    @Ignore
    private String kingName;

    // custom setter
    public void setKingName(String kingName) { setName("King " + kingName); }

    // custom getter
    public String getKingName() { return getName(); }

    // setter and getter for 'name'
}
```

You can then use the `setKingName()` as a custom setter instead of `setName()`. Please note that the custom setter uses `setName()` and does not assign directly to the field.

### Why can I not create a RealmList myself?

While a RealmList (api/io/realm/RealmList.html) does look like an ordinary List, it is not. It is our way of representing relationships and thus needs to be managed by Realm. This is the reason for the protected constructor, and why it is created for you when calling *Realm.createObject()*.

We do have plans however to add functionality to allow easy mapping of standard Lists to RealmLists (api/io/realm/RealmList.html).

### Why can I not create instances of RealmObjects myself?

This is due to the same reasons as for RealmList (api/io/realm/RealmList.html). The RealmObject (api/io/realm/RealmObject.html) is a proxy for the data in the database, and needs to be managed by Realm to work correctly.

Just like for RealmLists (api/io/realm/RealmList.html) we plan to add functionality to convert your own initialized model objects into proper Realm objects.

### Why do I need to use transactions when writing Realm objects?

Transactions (http://en.wikipedia.org/wiki/Database_transaction) are needed to ensure multiple fields are updated as one atomic operation. It allows you to define the scope of the updates that must be either fully completed or not completed at all (in case of errors or controlled rollback). By specifying the scope of the transaction you can control how frequent (or fast) your updates are persisted (i.e. insert multiple objects in one operation).

When doing inserts in a normal SQL based database like SQLite you are inserting multiple fields at once. This is automatically wrapped in a transaction, but is normally not visible to the user. In Realm these transactions are always explicit.

### What to do about out-of-memory exceptions?

Realm for Android is built upon an embedded storage engine. The storage engine does not allocate memory on the JVM heap but in ‘native’ memory, and your app can run out of memory. In such situations, Realm will throw an `io.realm.internal.OutOfMemoryError` exception. It is important not to ignore this exception. If your app does, you might leave your Realm file in a corrupted state.

### Large Realm file size

You should expect a Realm database to take less space on disk than an equivalent SQLite database.

In order to give you a consistent view of your data, Realm operates on multiple versions of a realm. If you read some data from a realm and then block the thread on a long-running operation while writing to the realm on other threads, the version is never updated and Realm has to hold on to intermediate versions of the data which you may not actually need, resulting in the file size growing steadily. (This extra space would eventually be reused by future writes, or could be compacted — for example by calling `compactRealmFile` (api/io/realm/Realm.html#compactRealmFile).)

**What does the exception 'Annotation processor may not have been executed.' mean?**
During compilation of your app, the model classes are processed and proxy classes are generated. If this annotation process fails, the proxy classes or their methods cannot be found. When your app begins running, it Realm will throw exception with message 'Annotation processor may not have been executed.'. You may see this in case you use Java 6 as it doesn't support inheriting annotations. You then have to add `@RealmClass` before your models. Otherwise it can often be solved by removing/cleaning all generated or intermediate files and rebuild.

---

@realm (https://github.com/realm) @realm (http://twitter.com/realm)

#realm (https://stackoverflow.com/questions/tagged/realm?sort=newest)

news (http://eepurl.com/VEKCn) help (https://attendee.gotowebinar.com/rt/1182038037080364033)

A Y Combinator (http://ycombinator.com/) company