

ObjectBox Java

Java API

This is the ObjectBox documentation for our Java API. We strive to provide you with the easiest and fastest solution to store and retrieve data. Your feedback on ObjectBox and this documentation is very welcome. Use the "Was this page helpful?" smiley at the end of each page or send us your comments to [contact\[at\]objectbox.io](mailto:contact[at]objectbox.io) - thank you! :)

→ [Getting started](#)

[/getting-started](#)

ObjectBox Changelog

V2.6.0-RC - 2020/04/28

Note: this is a release candidate. Only some minor details may change before 2.6.0.

- **@DefaultValue("") annotation for properties:** If used, a null value returned from the database is changed to the given default value (only empty string at this time). This is useful if a new property is added to an entity that should be not-null, but there are existing entities in the database that will return null for the new property. Note: naming is not final, e.g. it may change to e.g. **@AbsentValue("")** . [GH#157](#)
- **RxJava 3 support library:** available under the new artifact `objectbox-rxjava3` . It includes Kotlin extension functions to more easily obtain Rx types, e.g. use `query.observable()` to get an `Observable` . [GH#83](#)
- Fix error handling if ObjectBox can't create a Java entity (the proper exception is now thrown).
- Support setting an alias after combining conditions using `and()` or `or()` . [GH#83](#)
- Turn on incremental annotation processing by default. [GH#620](#)
- Add documentation that string property conditions ignore case by default. Point to using case-sensitive conditions for high-performance look-ups, e.g. [when using string UIDs](#).
- Repository Artifacts are signed once again.

V3.0.0-alpha2 - 2020/03/24

Note: this is a preview release. Future releases may add, change or remove APIs.

- Add Kotlin infix extension functions for creating conditions using the [new Query API](#). See [the documentation](#) for examples.
- The old Query API now also supports setting an alias after combining conditions using `and()` or `or()` . [GH#834](#)
- Add documentation that string property conditions ignore case by default. Point to using case-sensitive conditions for high-performance look-ups, e.g. [when using string UIDs](#).
- Java's `String[]` and Kotlin's `Array<String>` are now a supported database type. A converter is no longer necessary to store these types. Using the `arrayProperty.equal("item")` condition, it is possible to query for entities where "item" is equal to one of the array items.
- Support `@Unsigned` to indicate that values of an integer property (e.g. `Integer` and `Long` in Java) should be treated as unsigned when doing queries or creating indexes. See the Javadoc of the annotation for more details.
- Add [new library to support RxJava 3](#), `objectbox-rxjava3` . In addition `objectbox-kotlin` adds extension functions to more easily obtain Rx types, e.g. use `query.observable()` to get an `Observable` . [GH#839](#)

To use this release change the version of `objectbox-gradle-plugin` to `3.0.0-alpha2` . The plugin now properly adds the preview version of `objectbox-java` to your dependencies.

```
1  buildscript {
2      dependencies {
3          classpath "io.objectbox:objectbox-gradle-plugin:3.0.0-alpha2"
4      }
5  }
6
7  dependencies {
8      // Artifacts with native code remain at 2.5.1.
9      implementation "io.objectbox:objectbox-android:2.5.1"
10 }
```

The `objectbox-android` , `objectbox-linux` , `objectbox-macos` and `objectbox-windows` artifacts shipping native code remain at version 2.5.1 as there have been no changes. If you explicitly include them, make sure to specify their version as `2.5.1` .

V3.0.0-alpha1 - 2020/03/09

Note: this is a preview release. Future releases may add, change or remove APIs.

- A new [Query API](#) provides support for nested AND and OR conditions. See [the documentation](#) for examples and notable changes. [GH#201](#)
- Subscriptions now publish results in serial instead of in parallel (using a single thread vs. multiple threads per publisher). Publishing in parallel could previously lead to outdated results getting delivered after the latest results. As a side-effect transformers now run in serial instead of in parallel as well (on the same single thread per publisher). [GH#793](#)
- Turn on incremental annotation processing by default. [GH#620](#)

To use this release change the version of `objectbox-gradle-plugin` to `3.0.0-alpha1` and add a dependency on `objectbox-java` version `3.0.0-alpha1` .

```
1  buildscript {
2      dependencies {
3          classpath "io.objectbox:objectbox-gradle-plugin:3.0.0-alpha1"
4      }
5  }
6
7  dependencies {
8      implementation "io.objectbox:objectbox-java:3.0.0-alpha1"
9      // Artifacts with native code remain at 2.5.1.
10     implementation "io.objectbox:objectbox-android:2.5.1"
11 }
```

The `objectbox-android` , `objectbox-linux` , `objectbox-macos` and `objectbox-windows` artifacts shipping native code remain at version 2.5.1 as there have been no changes. However, if your project explicitly depends on them they will pull in

version 2.5.1 of `objectbox-java` . Make sure to add an explicit dependency on of `objectbox-java` version `3.0.0-alpha1` as mentioned above.

V2.5.1 - 2020/02/10

- Support Android Gradle Plugin 3.6.0. [GH#817](#)
- Support for incremental annotation processing. [GH#620](#) It is off by default. To turn it on set `objectbox.incremental` to `true` in `build.gradle` :

```
1  android {  
2      defaultConfig {  
3          javaCompileOptions {  
4              annotationProcessorOptions {  
5                  arguments = [ "objectbox.incremental" : "true" ]  
6              }  
7          }  
8      }  
9  }
```

V2.5.0 - 2019/12/12

Important bug fix - please update asap if you are using N:M relations!

- Fixed corner case for N:M ToMany (not the backlinks for ToOne) returning wrong results

Improvements and New Features

- Property queries compute sums and averages more precisely (improved algorithms and wider internal types)
- Query adds "describe" methods to obtain useful debugging information
- New method `removeAllObjects()` in `BoxStore` to clear the database of all data

V2.4.1 - 2019/10/29

- More helpful error messages if annotations can not be combined.
- Improved documentation on various annotations.

V2.4.0 - 2019/10/15

Upgrade Notes

- Android: the AAR libraries ship Java 8 bytecode. Your app will not build unless you upgrade `com.android.tools.build:gradle` to 3.2.1 or later.
- Android: the [ObjectBox LiveData](#) and [Paging integration](#) migrated from Android Support Libraries to Jetpack (AndroidX) Libraries. If you are using them the library will not work unless you make the following changes in your app:
 - Upgrade `com.android.tools.build:gradle` to 3.2.1 or later.
 - Upgrade `compileSdkVersion` to 28 or later.
 - Update your app to use Jetpack (AndroidX); follow the instructions in [Migrating to AndroidX](#).
- Note: this version requires backwards-incompatible changes to the generated MyObjectBox file. Make sure to rebuild your project before running your app so the MyObjectBox file is re-generated.

Improvements & Fixes

V2.4.0 - 2019/10/15

- Class transformation works correctly if absolute path contains special characters. [GH#135](#)

V2.4.0-RC - Release Candidate 2019/10/03

- Box: add `getRelationEntities` , `getRelationBacklinkEntities` , `getRelationIds` and `getRelationBacklinkIds` to directly access relations without going through `ToMany`.
- Box: add `putBatched` to put entities using a separate transaction for each batch.
- `Box.removeByKeyes()` is now deprecated; use `removeByIds()` instead.
- Query: fixed performance regressions introduced in version 2.3 on 32 bit devices in combination with ordered results
- Fixed removing a relation and the related entity class. [GH#490](#)
- Resolved issue to enable query conditions on the target ID property of a `ToOne` relation. [GH#537](#)
- `Box.getAll` always returns a mutable list. [GH#685](#)

- Do not overwrite existing objectbox-java or objectbox-kotlin dependency. [GH#693](#)
- Resolved a corner case build time crash when parsing package elements. [GH#698](#)
- When trying to find an appropriate get-method for a property, also check if the return type matches the property type. [GH#720](#)
- Explicitly display an error if two entities with the same name are detected. [GH#744](#)
- The code in MyObjectBox is split up by entity to make it less likely to run into the Java method size limit when using many @Entity classes. [GH#750](#)
- Query: improved performance for ordered results with a limit. [GH#769](#)
- Query: throw if a filter is used incorrectly with count or remove. [GH#771](#)
- Documentation and internal improvements.

V2.3.4 - 2019/03/19

- Avoid UnsatisfiedLinkError on Android devices that are not identifying as Android correctly
- Fix displaying large objects in Object Browser 32 bit
- Kotlin properties starting with "is" of any type are detected
- Add `objectbox-kotlin` to dependencies if `kotlin-android` plugin is applied (previously only for `kotlin` plugin)
- @BaseEntity classes can be generic

V2.3.3 - 2019/02/14

- Fixed a bug introduced by V2.3.2 affecting older Android versions 4.3 and below

V2.3.2 - 2019/02/04

- Potential work around for UnsatisfiedLinkError probably caused by installation errors mostly in alternative app markets
- Support for Android Gradle Plugin 3.3.0: resolves deprecated API usage warnings.

V2.3.1 - 2019/01/08

- Fixed a corner case for Box.getAll() after removeAll() to return a stale object if no objects are stored

V2.3 - 2018/12/30

Improvements & Fixes

- Query improvements: findIds and LazyList also consider the order; offset and limit for findIds
- Improved 32 bit support: Windows 32 version officially deployed, fixed a corner case crash
- Property queries for a boolean property now allow sum()
- Added Box.isEmpty()
- Supporting older Linux distributions (now starting at e.g. Ubuntu 16.04 instead of 18.04)
- Fix for a corner case with Box.count() when using a maximum
- Minor improvements to the ObjectBox code generator
- Android: set extractNativeLibs to false to avoid issues with extracting the native library

V2.2 - 2018/09/27

Improvements & Fixes

- *Fix:* the unique check for string properties had false positives resulting in UniqueViolationException. This occurs only in combination with IndexType.HASH (the default) when hashes actually collide. We advise to update immediately to the newest version if you are using hashed indexes.
- The release of new [ObjectBox C API](#) made us change name of the JNI library for better distinction. This should not affect you unless you depended on that (internal) name.
- Improved compatibility with class transformers like Jacoco
- Fixed query links for M:N backlinks
- Improved error messages for the build tools
- The Object Browser AAR now includes the required Android permissions

V2.1 - 2018/08/16

Minor Improvements & Fixes

- Entity counts are now cached for better performance

- Deprecated aggregate function were removed (deprecation in 1.4 with introduction of PropertyQuery)
- Object browser hot fix: the hashed indexes introduced in 2.0 broke the object browser
- Object browser fixes: filters with long ints, improved performance in the schema view
- NPE fix in ToOne
- Added a specific NonUniqueResultException if a query did not return an expected unique result

V2.0 - 2018/07/25

New Features/Improvements

- Links and relation completeness and other features already announced in the 2.0 beta
- Unique constraint for properties via [@Unique annotation](#)
- Hash index: for strings the new default index is hash-based, which is more space efficient
- Support for char type (16 bit)
- [RX lib](#) deployed in JCenter
- Rework of Query APIs: type safe properties (property now knows its owning entity)
- Allow query conditions of links using properties (without parameter alias)
- Query performance improvements when using order
- [Property based count](#): query for non-null or unique occurrences of entity properties (non-null and unique)
- Additional query conditions for strings: "greater than", "less than", "in"
- Added query conditions for byte arrays
- Set query parameters for "in" condition (int[] and long[])

V2.0 beta – 2018/06/26

New Features/Improvements

- Query across relation bounds [using links](#) (aka "join"): queries just got much more powerful. For example, query for orders that have a customer with an address on "Sesame Street". Or all persons, who have a grand parent called "Alice".
- [Backlinks for to-many relations](#): now ObjectBox is "relation complete" with a bi-directional many-to-many relation.

- Query performance improvements: getting min/max values of indexed properties in constant time
- Android: added [Paging library support](#) (architecture components)
- [Kotlin extensions](#): more Kotlin fun with ObjectBox KTX
- [Query parameters aliases](#): helps setting query parameters in complex scenarios (e.g. for properties of linked entities)
- Improved query parameter verification
- Many internal improvements to keep us going fast in the future

V1.5 and earlier

Check the [release history](#) for older releases

Getting started



Note: We focus on Android on this page. You can [use ObjectBox in a plain Java project](#) as well.



<https://www.youtube.com/watch?v=flmAeYY-u9I>

Adding ObjectBox to your Android Project

ObjectBox is available from the jcenter repository. To add ObjectBox to your Android project, open the root `build.gradle` file of your project (not the ones for your app or module) and add a global variable for the version and the ObjectBox Gradle plugin:

```
1  buildscript {
2      ext.objectboxVersion = '2.5.1'
3      repositories {
4          jcenter()
5      }
6      dependencies {
7          // Android Gradle Plugin 3.2.1 or later supported.
8          classpath 'com.android.tools.build:gradle:3.5.4'
9          classpath "io.objectbox:objectbox-gradle-plugin:$objectboxVersion"
10     }
11 }
```

Open the `build.gradle` file for your app or module and, after the `com.android.application` plugin, apply the `io.objectbox` plugin:

Java

```
1 apply plugin: 'com.android.application'
2 apply plugin: 'io.objectbox' // Apply last.
```

Kotlin

```
1 apply plugin: 'com.android.application'
2 apply plugin: 'kotlin-android'
3 apply plugin: 'kotlin-kapt' // Required for annotation processing.
4 apply plugin: 'io.objectbox' // Apply last.
```



If you encounter any problems in this or later steps, check out the [FAQ](#) and [Troubleshooting](#) pages.

Then do "Sync Project with Gradle Files" so the Gradle plugin automatically adds the required ObjectBox libraries and code generation tasks.

Optional: Advanced Setup

The ObjectBox plugin uses reasonable defaults and detects most configurations automatically. However, if needed you can configure the model file path, the MyObjectBox package, enable debug mode and more [using advanced setup options](#).

Entity Classes

Next, define your model by annotating at least one class with `@Entity` and `@Id`. A simple entity representing a user could look like this:

Java

```
1 // User.java
2 @Entity
3 public class User {
4     @Id public long id;
5     public String name;
6 }
```

Kotlin


```
1 // User.kt
2 @Entity
3 data class User(
4     @Id var id: Long = 0,
5     var name: String? = null
6 )
```

⚠ When using a data class, **add default values for all properties**. This will ensure your data class will have a constructor that can be called by ObjectBox. (Technically this is only required if using custom or transient properties or relations, but it's a good idea to do it always.)

Entities must have one `@Id` property of type `long` (or `Long` in Kotlin). If you need to use other types, like a String ID, [see the @Id annotation docs](#). Also, it must have **not-private visibility** (or a not-private getter and setter method).

For a deeper explanation and a look at all other available annotations (e.g. for relations and indexes) check the [Entity Annotations](#) page.


Now **build your project** to generate required classes, for example using **Build > Make Project** in Android Studio.

 Note: If you make significant changes to your entities, e.g. by moving them or modifying annotations, make sure to **rebuild** the project so generated ObjectBox code is updated.

Model file

Among other files ObjectBox generates a JSON **model file**, by default to `app/objectbox-models/default.json`. This JSON file will change every time you change your entities or we make some internal changes to ObjectBox.

Keep this JSON file, commit the changes to version control!

 In Android Studio you might have to switch the Project view from Android to Project to see the `default.json` model file.

This file keeps track of unique IDs assigned to your entities and properties. This ensures that an older version of your database can be smoothly upgraded if your entities or properties change. It also enables to keep data [when renaming entities or properties](#) or to [resolve conflicts](#) when two of your developers make changes at the same time.

Core Classes

The following core classes are the essential interface to ObjectBox:

MyObjectBox: Generated based on your entity classes, MyObjectBox supplies a [builder](#) to set up a BoxStore for your app.

BoxStore: The entry point for using ObjectBox. BoxStore is your direct interface to the database and manages Boxes.

Box: A box persists and queries for entities. For each entity, there is a Box (supplied by BoxStore).

Core Initialization

The BoxStore for your app is initialized using the builder returned by the generated MyObjectBox class, for example in a small helper class like this:

Java

```
1 public class ObjectBox {
2     private static BoxStore boxStore;
3
4     public static void init(Context context) {
5         boxStore = MyObjectBox.builder()
6             .androidContext(context.getApplicationContext())
7             .build();
8     }
9
10    public static BoxStore get() { return boxStore; }
11 }
```

Kotlin

```
1 object ObjectBox {
2     lateinit var boxStore: BoxStore
3     private set
4
5     fun init(context: Context) {
6         boxStore = MyObjectBox.builder()
7             .androidContext(context.applicationContext)
8             .build()
9     }
10 }
```



You might receive crash reports due to `UnsatisfiedLinkError` or `LinkageError` on the build call. See [App Bundle, split APKs and Multidex](#) for solutions.

The best time to initialize ObjectBox is when your app starts. We suggest to do it in the `onCreate` method of your [Application class](#):

Java

```
1 public class ExampleApp extends Application {
2     @Override
3     public void onCreate() {
4         super.onCreate();
5         ObjectBox.init(this);
6     }
7 }
```


Kotlin

```
1 class ExampleApp : Application() {  
2     override fun onCreate() {  
3         super.onCreate()  
4         ObjectBox.init(this)  
5     }  
6 }
```

Now you can easily get a hold of BoxStore throughout your app (usually in fragments, activities) and access the specific Box that you need:

Java

```
Box<User> userBox = ObjectBox.get().boxFor(User.class);
```

Kotlin

```
val userBox: Box<User> = ObjectBox.boxStore.boxFor()
```

Here, `User` is an ObjectBox entity. And now that we have its Box, we can start storing and retrieving it from the database.

Basic Box operations

The `Box` class is likely the class you interact with most. As seen previously, you get Box instances via `BoxStore.boxFor()`. A Box instance gives you access to objects of a

particular type. For example, if you have `User` and `Order` entities, you need two `Box` objects to interact with each:

Java

```
1 Box<User> userBox = boxStore.boxFor(User.class);
2 Box<Order> orderBox = boxStore.boxFor(Order.class);
```

Kotlin

```
1 val userBox: Box<User> = ObjectBox.boxStore.boxFor()
2 val orderBox: Box<Order> = ObjectBox.boxStore.boxFor()
```

These are some of the operations offered by the `Box` class:

- **put:** Inserts a new or updates an existing object with the same ID. When inserting and `put` returns, an ID will be assigned to the just inserted object (this will be explained below). `put` also supports putting multiple objects, which is more efficient.
- **get and getAll:** Given an object's ID reads it back from its box. To get all objects in the box use `getAll`.
- **remove and removeAll:** Remove a previously put object from its box (deletes it). `remove` also supports removing multiple objects, which is more efficient. `removeAll` removes (deletes) all objects in a box.
- **count:** Returns the number of objects stored in this box.
- **query:** Starts building a query to return objects from the box that match certain conditions. See [queries](#) for details.

For a complete list of methods available in the `Box` class, check [its JavaDoc](#).

Object IDs

By default **IDs for new objects are assigned by ObjectBox**. When a new object is put, it will be assigned the next highest available ID:

Java

```
1 User user = new User();
2 // user.id == 0
3 box.put(user);
4 // user.id != 0
5 long id = user.id;
```

Kotlin

```
1 val user = User()
2 // user.id == 0
3 box.put(user)
4 // user.id != 0
5 val id = user.id
```

For example, if there is an object with ID 1 and another with ID 100 in a box, the next new object that is put will be assigned ID 101.

If you try to assign a new ID yourself and put the object, ObjectBox will throw an error.



If you **need to assign IDs by yourself** have a look at [how to switch to self-assigned IDs](#) and what side-effects apply.

Reserved Object IDs

Object IDs **can not be**:

- `0` (zero) or `null` (if using `java.lang.Long`) As said above, when putting an object with ID zero it will be assigned an unused ID (not zero).
- `0xFFFFFFFFFFFFFFFF` (-1 in Java) Reserved for internal use.

For an advanced explanation see the page on [Object IDs](#).

Transactions

While ObjectBox offers powerful transactions, it is sufficient for many apps to consider just some basics guidelines about transactions:

- A `put` runs an implicit transaction.
- Prefer `put` bulk overloads for lists (like `put(entities)`) when possible.
- For a high number of DB interactions in loops, consider explicit transactions, such as using `runInTx()` .

For more details check the separate [transaction documentation](#).

Have an app with greenDAO? DaoCompat is for you!

DaoCompat is a compatibility layer that gives you a greenDAO like API for ObjectBox. It makes switching from greenDAO to ObjectBox simple. Have a look at [the documentation](#) and [the example](#). [Contact us](#) if you have any questions!

Next steps

- Check out the [ObjectBox example projects on GitHub](#).
- Learn about [Queries](#) and [Relations](#).

Tutorial: Demo Project

What is ObjectBox? It's a mobile database that makes object persistence simple and super fast.

This tutorial will walk you through a simple note-taking app explaining how to do basic operations with ObjectBox. To just integrate ObjectBox into your project, look for the [Getting Started](#) page.

It's a good idea to clone the [example project repository](#) from GitHub right now:

```
git clone https://github.com/objectbox/objectbox-examples.git
```

This allows you to run the code and explore it in its entirety. The example project discussed here is in the `android-app` folder (or `android-app-kotlin` for the Kotlin version). It is a simple Android app for taking notes where you can add new notes by typing in some text and delete notes by clicking on an existing note.

The Note entity and Box class

To begin let's jump right into the code: in the `src` folder you will find the entity class for a note, `Note`. It is persisted to the database and contains all data that is part of a note, like an id, note text and the creation date.

Java

Note.java

```
1 @Entity
2 public class Note {
3
4     @Id
5     long id;
6
7     String text;
8     String comment;
9     Date date;
10
11     ...
12 }
```

Kotlin

Note.kt

```
1 @Entity
2 data class Note(
3     @Id var id: Long = 0,
4     var text: String? = null,
5     var comment: String? = null,
6     var date: Date? = null
7 )
```

In general, an ObjectBox entity is an annotated class persisted in the database with its properties. In order to extend our note or to create new entities, you simply modify or create new plain Java classes and annotate them with `@Entity` and `@Id`.

→ [Entity Annotations](#)

/entity-annotations

Go ahead and build the project, for example by using **Build > Make project** in Android Studio. This triggers ObjectBox to generate some classes, like `MyObjectBox.java`, and some other classes used by ObjectBox internally.

Inserting notes

To see how new notes are added to the database, take a look at the `NoteActivity` class. First of all a `Box` object for the `Note` class is prepared, which is done in `onCreate()` :

Java

`NoteActivity.java`

```
1  @Override
2  public void onCreate(Bundle savedInstanceState) {
3      ...
4      notesBox = ObjectBox.get().boxFor(Note.class);
5      ...
6  }
```

Kotlin

`NoteActivity.kt`

```
1  public override fun onCreate(savedInstanceState: Bundle?) {
2      ...
3      notesBox = ObjectBox.boxStore.boxFor()
4      ...
5  }
```



Note: In the example project, `ObjectBox` is the name of a helper class to set up and keep a reference to `BoxStore`.

When the user clicks the ADD button the method `addNote()` is called. There, a new `Note` object is created and put into the database using the `Box` reference:

Java

```
NoteActivity.java

1  private void addNote() {
2      ...
3      Note note = new Note();
4      note.setText(noteText);
5      note.setComment(comment);
6      note.setDate(new Date());
7      notesBox.put(note);
8      Log.d(App.TAG, "Inserted new note, ID: " + note.getId());
9      ...
10 }
```

Kotlin

```
NoteActivity.kt

1  private fun addNote() {
2      ...
3      val note = Note(text = noteText, comment = comment, date = Date())
4      notesBox.put(note)
5      Log.d(App.TAG, "Inserted new note, ID: " + note.id)
6      ...
7  }
```

Note that the ID is left at 0 when creating the note. In this case ObjectBox assigns an ID during `put()` .

Removing/deleting notes

When the user taps a note it should be deleted. To remove (or delete) a note from its box use `remove()` or one of its overloads. See `noteClickListener` :

Java

NoteActivity.java

```
1  OnItemClickListener noteClickListener = new OnItemClickListener() {
2      @Override
3      public void onItemClick(AdapterView<?> parent, View view, int po
4          Note note = notesAdapter.getItem(position);
5          notesBox.remove(note);
6          Log.d(App.TAG, "Deleted note, ID: " + note.getId());
7      ...
8  }
9  };
```

Kotlin

NoteActivity.kt

```
1  private val noteClickListener = OnItemClickListener { _, _, position
2      notesAdapter.getItem(position)?.also {
3          notesBox.remove(it)
4          Log.d(App.TAG, "Deleted note, ID: " + it.id)
5      }
6      ...
7  }
```

Querying notes

To query and display notes with a list adapter a `Query` instance is built once in `onCreate()` :

Java

NoteActivity.java

```
1  @Override
2  public void onCreate(Bundle savedInstanceState) {
3      ...
4      // Query all notes, sorted a-z by their text.
5      notesQuery = notesBox.query().order(Note_.text).build();
6      ...
7  }
```

Kotlin

NoteActivity.kt

```
1  public override fun onCreate(savedInstanceState: Bundle?) {
2      ...
3      // Query all notes, sorted a-z by their text.
4      notesQuery = notesBox.query {
5          order(Note_.text)
6      }
7      ...
8  }
```

And then executed each time any notes change:

Java

NoteActivity.java

```
1 private void updateNotes() {  
2     List<Note> notes = notesQuery.find();  
3     notesAdapter.setNotes(notes);  
4 }
```

Kotlin

NoteActivity.kt

```
1 private fun updateNotes() {  
2     val notes = notesQuery.find()  
3     notesAdapter.setNotes(notes)  
4 }
```

In addition to an order, you can add various conditions, like equality or less/greater than, when building a query.

[→ Queries](#)

/queries

Updating notes and more

What is not shown in the example, but is just as easy is how to update an existing (== the ID is not 0) note. Just modify any of its properties and call `put()` again with the changed object:

Java

```
1 note.setText("This note has changed.");  
2 notesBox.put(note);
```

Kotlin

```
1 note.text = "This note has changed."  
2 notesBox.put(note)
```

There are **additional methods to put, find, query, count or remove** entities. Check out the methods of [the Box class](#) to learn more.

→ [Getting started](#)

[/getting-started](#)


Setting up the database

Now that you saw ObjectBox in action, how did we get that [BoxStore](#) instance? Typically you should set up a BoxStore once for the whole app. This example uses a [helper class as recommended in the Getting Started guide](#).

→ [Getting started](#)

[/getting-started](#)

Remember: ObjectBox is a NoSQL database on its own and thus NOT based on SQL or SQLite. That's why you do not need to set up "CREATE TABLE" statements during initialization.

 Note: it is perfectly fine to never close the database. That's even recommended for most apps.

Entity Annotations

ObjectBox - Database Persistence with Entity Annotations

ObjectBox is a database that persists objects. For a clear distinction, we sometimes call those persistable objects **entities**.

To let ObjectBox know which classes are entities you annotate them with `@Entity`, for example:

Java

```
1 @Entity
2 public class User {
3
4     @Id
5     private long id;
6
7     private String name;
8
9     // Not persisted:
10    @Transient
11    private int tempUsageCount;
12
13    // TODO: getters and setters.
14 }
```

```
1 @Entity
2 data class User(
3     @Id var id: Long = 0,
4     var name: String? = null,
5     // Not persisted:
6     @Transient var tempUsageCount: Int = 0
7 )
```

The `@Entity` annotation identifies the class `User` as a persistable entity. This will trigger ObjectBox to generate persistence code tailored for this class.



Note:

- It's often good practice to model entities as “**dumb**” **data classes (POJOs)** with just properties.
- For Kotlin: When using a data class, **add default values for all properties**. This will ensure your data class will have a constructor that can be called by ObjectBox. (Technically this is only required if using custom or transient properties or relations, but it's a good idea to do it always.)

Object IDs: @Id

In ObjectBox, **entities must have one `@Id` property of type `long`** (or `Long` in Kotlin) with **not-private visibility** (or not-private getter and setter method) to efficiently get or reference objects. You can use the nullable type `java.lang.Long`, but we do not recommend it.

Java

```
1 @Entity
2 public class User {
3
4     @Id public long id;
5
6     ...
7 }
```

Kotlin

```
1 @Entity
2 data class User(
3     @Id var id: Long = 0,
4     ...
5 )
```

If you **need to use another type for IDs** (such as a string UID given by a server), model them as regular properties and use [queries](#) to look up objects by your application specific ID. Also make sure to index the property, and if it's a string use a case-sensitive condition, to speed up look-ups.

Java

```
1 @Entity
2 class StringIdEntity {
3     @Id public long id;
4     @Index public String uid;
5 }
6
7 StringIdEntity entity = box.query()
8     .equal(StringIdEntity_.uid, uid, StringOrder.CASE_SENSITIVE)
9     .build().findUnique()
```

Kotlin

```
1 @Entity
2 data class StringIdEntity(
3     @Id var id: Long = 0,
4     @Index var uid: String? = null
5 )
6
7 val entity = box.query()
8     .equal(StringIdEntity_.uid, uid, StringOrder.CASE_SENSITIVE)
9     .build().findUnique()
```

ID properties are unique and indexed by default.

When you put a new object you do not assign an ID. By default **IDs for new objects are assigned by ObjectBox**. See the page on [Object IDs](#) for details.



If you **need to assign IDs by yourself** have a look at [how to switch to self-assigned IDs](#) and what side-effects apply.

Make entity data accessible

ObjectBox needs to access the data of your entity's properties (e.g. in generated Cursor classes). You have two options:

1. Give your property fields at least “package private” (not “private”) visibility. In Kotlin, you can use `@JvmField`.
2. Provide standard getters (your IDE can generate them easily).

To improve performance when ObjectBox constructs your entities, you might also want to provide an all-properties constructor (for Kotlin data classes, make sure all properties have a default value instead). For example:

Java

```
1  @Entity
2  public class User {
3
4      @Id private long id;
5
6      private String name;
7
8      // Not persisted:
9      @Transient private int tempUsageCount;
10
11     public User() { /* Default constructor */ }
12
13     public User(id, name) {
14         this.id = id;
15         this.name = name;
16     }
17
18     // Getters and setters for properties...
19 }
```

Kotlin

```
1 // Ensure all properties have default values:
2 @Entity
3 data class User(
4     @Id var id: Long = 0,
5     var name: String? = null,
6     // Not persisted:
7     @Transient var tempUsageCount: Int = 0
8 )
```

Basic annotations for entity properties

Java

```
1 ...
2 @NameInDb("username")
3 private String name;
4
5 @Transient
6 private int tempUsageCount;
7 ...
```

```
1  ...
2  @NameInDb("username")
3  var name: String? = null,
4
5  @Transient
6  var tempUsageCount: Int = 0,
7  ...
```

@NameInDb lets you define a name on the database level for a property. This allows you to rename the field (or property in Kotlin) without affecting the property name on the database level.



Note:

- To rename properties and even entities you should [use @Uid annotations](#) instead.
- @NameInDb only works with inline constants to specify a column name.

@Transient (or alternatively the `transient` modifier) marks properties that should not be persisted, like the temporary counter above. **static** properties will also not be persisted.

Property Indexes with @Index

Annotate a property with **@Index** to create a database index for the corresponding database column. This can improve performance when querying for that property.

Java

```
1 ...  
2 @Index  
3 private String name;  
4 ...
```

Kotlin


```
1 ...  
2 @Index  
3 var name: String? = null,  
4 ...
```



@Index is currently not supported for `byte[]` , `float` and `double` in Java or the equivalent `ByteArray` , `Float` and `Double` in Kotlin.

An index stores additional information in the database to make look-ups faster. As an analogy we could look at Java-like programming languages where you store objects in a list. For example you could store persons using a `List<Person>` . Now, you want to search for all persons with a specific name so you would iterate through the list and check for the name property of each object. This is an $O(N)$ operation and thus does not scale well with an increasing number of objects. To make this more scalable you can introduce a second data structure `Map<String, Person>` with the name as a key. This will give you a constant lookup time ($O(1)$). The downside of this is that it needs more resources (here: RAM) and slows down add/remove operations on the list a bit. These principles can be transferred to database indexes, just that the primary resource consumed is disk space.

Index types (String)

 Since 2.0.0

ObjectBox 2.0 introduced index types. Before, every index used the property **value** for all look-ups. Now, ObjectBox can also use a **hash** to build an index. Because `String` properties are typically taking more space than scalar values, ObjectBox switched the default index type to hash for strings.

You can instruct ObjectBox to use a value-based index for a `String` property by specifying the index `type` :

Java

```
1 ...  
2 @Index(type = IndexType.VALUE)  
3 private String name;  
4 ...
```


Kotlin


```
1 ...  
2 @Index(type = IndexType.VALUE)  
3 var name: String? = null,  
4 ...
```

Keep in mind that for `String` , depending on the length of your values, a value-based index may require more storage space than the default hash-based index.


ObjectBox supports these index types:

- **Not specified or DEFAULT** Uses best index based on property type (HASH for `String` , VALUE for others).
- **VALUE** Uses property values to build index. For `String`, this may require more storage than a hash-based index.
- **HASH** Uses 32-bit hash of property values to build index. Occasional collisions may occur which should not have any performance impact in practice. Usually a better choice than HASH64, as it requires less storage.
- **HASH64** Uses long hash of property values to build the index. Requires more storage than HASH and thus should not be the first choice in most cases.

 **When migrating data from pre-2.0 ObjectBox versions**, the default index type for strings has changed from value to hash. With a plain `@Index` annotation this will update the indexes automatically: the old value-based indexes will be deleted and the new hash-based indexes will be build. A side effect of this is that the database file might grow in the process. If you want to prevent this, you stick to the "old" index type using `@Index(type = IndexType.VALUE)` .

 **Limits of hash-based indexes:** Hashes work great for equality checks, but not for "**starts with**" type conditions. If you frequently use those, you should use value-based indexes instead.

Unique constraints

 Since 2.0.0

Annotate a property with `@Unique` to enforce that values are unique before an entity is put:

Java

```
1 ...
2 @Unique
3 private String name;
4 ...
```

Kotlin

```
1 ...
2 @Unique
3 var name: String? = null,
4 ...
```

A `put()` operation will abort and throw a `UniqueViolationException` if the unique constraint is violated:

Java

```
1 try {
2     box.put(new User("Sam Flynn"));
3 } catch (UniqueViolationException e) {
4     // A User with that name already exists.
5 }
```



```
1 try {  
2     box.put(User("Sam Flynn"))  
3 } catch (e: UniqueViolationException) {  
4     // A User with that name already exists.  
5 }
```



Unique constraints are based on an index. You can further configure the index by adding an **@Index** annotation.

Relations

Creating to-one and to-many relations between objects is possible as well, see the [Relations](#) documentation for details.

Triggering generation

Once your entity schema is in place, you can trigger the code generation process by compiling your project. For example using **Build > Make project** in Android Studio.

If you encounter errors after changing your entity classes, try to rebuild (clean, then build) your project to make sure old generated classes are cleaned.

Android

Android Local Unit Tests


Android Local Unit Tests

ObjectBox supports local unit tests. This gives you the full ObjectBox functionality for running super fast test directly on your development machine.

On [Android, unit tests](#) can either run on an Android device (or emulator), so called instrumented tests, or they can run on your local development machine. Running local unit tests is typically much faster.

To learn how local unit tests for Android work in general have a look at the Android developers documentation on [Building Local Unit Tests](#). Read along to learn how to use ObjectBox in your local unit tests.

Set Up Your Testing Environment


 **The setup step is only required for ObjectBox 1.4 or older** (or if you want to manually add the dependency). In newer versions the ObjectBox plugin automatically adds the native ObjectBox library required for your current operating system.

Add the native ObjectBox library to your existing test dependencies in your app's

`build.gradle` file:

```
1 dependencies {
2     // Required -- JUnit 4 framework
3     testImplementation 'junit:junit:4.12'
4     // Optional -- manually add native ObjectBox library to override auto-
5     testImplementation "io.objectbox:objectbox-linux:$objectboxVersion"
6     testImplementation "io.objectbox:objectbox-macos:$objectboxVersion"
7     testImplementation "io.objectbox:objectbox-windows:$objectboxVersion"
```

Local unit tests are currently only supported on 64-bit operating systems.

 Note: on Windows you might have to install the [Microsoft Visual C++ 2015 Redistributable \(x64\)](#) packages to use the native library.

Create a Local Unit Test Class

You create your local unit test class as usual under `module-name/src/test/java/`. To use `ObjectBox` in your test methods you need to build a `BoxStore` instance using the generated `MyObjectBox` class of your project. You can use the [directory\(File\) method](#) on the `BoxStore` builder to ensure the test database is stored in a specific folder on your machine. To start with a clean database for each test you can delete the existing database using [BoxStore.deleteAllFiles\(File\)](#).


The following example shows how you could implement a local unit test class that uses `ObjectBox`:

```
1 public class NoteTest {
2
3     private static final File TEST_DIRECTORY = new File("objectbox-example
4     private BoxStore store;
5
6     @Before
7     public void setUp() throws Exception {
8         // delete database files before each test to start with a clean da
9         BoxStore.deleteAllFiles(TEST_DIRECTORY);
10        store = MyObjectBox.builder()
11            // add directory flag to change where ObjectBox puts its d
12            .directory(TEST_DIRECTORY)
13            // optional: add debug flags for more detailed ObjectBox l
14            .debugFlags(DebugFlags.LOG_QUERIES | DebugFlags.LOG_QUERY_
15            .build();
16    }
```

```

17
18     @After
19     public void tearDown() throws Exception {
20         if (store != null) {
21             store.close();
22             store = null;
23         }
24         BoxStore.deleteAllFiles(TEST_DIRECTORY);
25     }
26
27     @Test
28     public void exampleTest() {
29         // get a box and use ObjectBox as usual
30         Box<Note> noteBox = store.boxFor(Note.class);
31         assertEquals(...);
32     }
33
34 }

```

 Note: To help diagnose issues you can enable log output for ObjectBox actions, such as queries, by specifying one or more [debug flags](#) when building BoxStore.

Base class for tests

It's usually a good idea to extract the setup and tear down methods into a base class for your tests. E.g.:

```

1  public abstract class AbstractObjectBoxTest {
2
3      protected static final File TEST_DIRECTORY = new File("objectbox-examp
4      protected BoxStore store;
5
6      @Before
7      public void setUp() throws Exception {
8          // delete database files before each test to start with a clean da
9          BoxStore.deleteAllFiles(TEST_DIRECTORY);
10         store = MyObjectBox.builder()

```

```

11         // add directory flag to change where ObjectBox puts its d
12         .directory(TEST_DIRECTORY)
13         // optional: add debug flags for more detailed ObjectBox l
14         .debugFlags(DebugFlags.LOG_QUERIES | DebugFlags.LOG_QUERY_
15         .build();
16     }
17
18     @After
19     public void tearDown() throws Exception {
20         if (store != null) {
21             store.close();
22             store = null;
23         }
24         BoxStore.deleteAllFiles(TEST_DIRECTORY);
25     }
26 }

```

Testing Entities with Relations



Only required for ObjectBox 1.4.4 or older.


To test entities that have [relations](#), like ToOne or ToMany properties, on the local JVM you must initialize them and add a transient BoxStore field.

See the [documentation about "initialization magic"](#) for an example and what to look out for.

Background: the "initialization magic" is normally done by the ObjectBox plugin using the Android plugin Transform API which allows to modify byte-code. But transforms do not run for local unit tests (they do for instrumented tests).

LiveData (Arch. Comp.)

ObjectBox - LiveData with Android Architecture Components

 Since 1.2.0. Have a look at [the example project on GitHub](#).

As an alternative to ObjectBox' [data observers and reactive queries](#), you can opt for the [LiveData](#) approach supplied by Android Architecture Components. ObjectBox comes with `ObjectBoxLiveData`, a class that can be used inside your [ViewModel](#) classes.

A simple ViewModel implementation for our note example app includes the special `ObjectBoxLiveData` that is constructed using a regular [ObjectBox query](#):

```
1 public class NoteViewModel extends ViewModel {
2
3     private ObjectBoxLiveData<Note> noteLiveData;
4
5     public ObjectBoxLiveData<Note> getNoteLiveData(Box<Note> notesBox) {
6         if (noteLiveData == null) {
7             // query all notes, sorted a-z by their text
8             noteLiveData = new ObjectBoxLiveData<>(notesBox.query().order(
9         )
10        return noteLiveData;
11    }
12 }
```

Note that we did choose to pass the box to `getNoteLiveData()`. Instead you could use `AndroidViewModel`, which provides access to the `Application` context, and then call `((App)getApplication()).getBoxStore().boxFor()` inside the ViewModel. However, the first approach has the advantage that our ViewModel has no reference to Android classes. This makes it easier to unit test.


Now, when creating the activity or fragment we get the ViewModel, access its LiveData and finally register to observe changes:

```
1 NoteViewModel model = ViewModelProviders.of(this).get(NoteViewModel.class)
2 model.getNoteLiveData(notesBox).observe(this, new Observer<List<Note>>() {
3     @Override
4     public void onChanged(@Nullable List<Note> notes) {
5         notesAdapter.setNotes(notes);
6     }
7 });
```


The ObjectBoxLiveData will now subscribe to the query and notify observers when the results of the query change, if there is at least one observer. In this example the activity is notified if a note is added or removed. If all observers are destroyed, the LiveData will cancel the subscription to the query.

If you have used [ObjectBox observers](#) in the past this might sound familiar. Well, because it is! ObjectBoxLiveData just wraps a DataObserver on the query you give to it.

Paging (Arch. Comp.)

 Since 2.0.0

ObjectBox supports integration with the [Paging library](#) that is part of Google's [Android Architecture Components](#). To that end, the [ObjectBox Android library](#) (`objectbox-android`) provides the `ObjectBoxDataSource` class. It is an implementation of the Paging library's `PositionalDataSource`.

 Note: the following assumes that you have already [added and set up the Paging library](#) in your project.

Using ObjectBoxDataSource

Within your `ViewModel`, similar to [creating a LiveData directly](#), you first [build your ObjectBox query](#). But then, you construct an `ObjectBoxDataSource` factory with it instead. This factory is then passed to a `LivePagedListBuilder` to build the actual `LiveData`.

Here is an example of a `ViewModel` class doing just that:

```
1 public class NotePagedViewModel extends ViewModel {
2
3     private LiveData<PagedList<Note>> noteLiveDataPaged;
4
5     public LiveData<PagedList<Note>> getNoteLiveDataPaged(Box<Note> notesB
6         if (noteLiveDataPaged == null) {
7             // query all notes, sorted a-z by their text
8             Query<Note> query = notesBox.query().order(Note_.text).build()
9             // build LiveData
10            noteLiveDataPaged = new LivePagedListBuilder<>(
11                new ObjectBoxDataSource.Factory<>(query),
12                20 /* page size */
13            ).build();
14        }
```

```
15         return noteLiveDataPaged;
16     }
17 }
```

Note that the `LiveData` holds your entity class, here `Note`, wrapped inside a `PagedList`. You observe the `LiveData` as usual in your activity or fragment, then submit the `PagedList` on changes to your `PagedListAdapter` of the Paging library.

We will not duplicate how this works here, see the [Paging library documentation](#) for details about this.

Next steps

- Have a look at the [ObjectBox Architecture Components example code](#).
- Check out [ObjectBox support for LiveData](#).
- Learn how to [build queries](#).


App Bundle, split APKs and Multidex

Your app might observe crashes due to `UnsatisfiedLinkError` or `LinkageError` (since ObjectBox 2.3.4) on some devices. This has mainly two reasons: If your app uses the [App Bundle](#) format, the legacy split APK feature or [Multidex](#) the native library can't be found. Or if your minimum SDK is below API 23 (Marshmallow), there are known bugs in Android's native library loading code.

Let us know if you have more info on this in [GitHub issue 605](#).

App Bundle and split APKs

When using an App Bundle or split APKs Google Play only delivers the split APKs required for each user's device configuration, including its architecture (ABI). If users bypass Google Play to install your app ("sideloading") they might not install all of the required split APKs. If the split APK containing the ObjectBox native library required for the device ABI is missing, your app will crash with `LinkageError` when building BoxStore.

 **Update:** With the Play Core library Google now provides an official, easy [solution to detect if an app is incorrectly installed](#) and direct the user to fix the problem (e.g. to reinstall the app from Google Play). **We recommend using the Play Core library instead of the below workarounds.** See [how we updated our example app](#) to use the Play Core detection.

If you do not want to use the Play Core library, there are two alternatives:

Alternative: turn off splitting by ABI

The simplest solution is to always include native libraries for all supported ABIs. However, this will increase the download size of your app for all users.

```

1  android {
2      bundle {
3          abi {
4              // This property is set to true by default.
5              enableSplit = false
6          }
7      }
8  }

```

Source: [Android Developers](#)

Alternative: Catch exception and inform users

You can guard the `MyObjectBox` build call and for example display an activity with an info message (e.g. direct users to reinstall the app from Google Play, send you an error report, ...):

```

1  // guard the build call and set some flag (here setting the boxStore field
2  try {
3      boxStore = MyObjectBox.builder()
4          .androidContext(context.getApplicationContext())
5          .build();
6  } catch (LinkageError e) {
7      boxStore = null;
8      Log.e(App.TAG, "Failed to load ObjectBox: " + e.getMessage());
9  }
10
11
12 // then for example in the main activity check the flag in onCreate and
13 // direct to an info/error message without the app crashing:
14 if (ObjectBox.get() == null) {
15     startActivity(new Intent(this, ErrorActivity.class));
16     finish();
17     return;
18 }

```

As an example see [how we added this to our Android app example](#).

Buggy devices

On some devices and if your minimum SDK is below API 23 (Marshmallow), loading the native library may fail due to bugs. To counter this ObjectBox includes support for the ReLinker tool which will try to extract the native library manually if loading it normally fails.

To set this up add [ReLinker](#) to your dependencies:

```
implementation 'com.getkeepsafe.relinker:relinker:1.3.1'
```

❗ ObjectBox is calling ReLinker via reflection. If you are using ProGuard or Multidex, make sure to add keep rules so that ReLinker code is not stripped from the final app or is not in the primary dex file.

For ProGuard add this line:


```
-keep class com.getkeepsafe.relinker.** { *; }
```

For Multidex add a multiDexKeepProguard file to your build file:

```
1  android {
2      buildTypes {
3          release {
4              multiDexKeepProguard file('multidex-config.pro')
5          }
6      }
7  }
```

And in the multidex-config.pro file add the same rule as above:

```
-keep class com.getkeepsafe.relinker.** { *; }
```

 **Multidex supports two file formats to keep files.** We are using the ProGuard format (multiDexKeepProguard property). You can also use the multiDexKeepFile property, but make sure to adapt the rule above to that format.

Enable ReLinker debug log

To enable debug logs for ReLinker you can pass a custom `ReLinkerInstance` when building `BoxStore` :

```
1 boxStore = MyObjectBox.builder()
2   .androidContext(App.this)
3   .androidReLinker(ReLinker.log(new ReLinker.Logger() {
4       @Override
5       public void log(String message) { Log.d(TAG, message); }
6   }))
7   .build();
```

Java Desktop Apps

ObjectBox – Embedded Database for Java Desktop Apps

ObjectBox does not only [work with Android projects](#), but also for plain Java (JVM) desktop apps running on **Windows, Linux, and macOS**. Just like on Android, ObjectBox stands for a super simple API and high performance. It's designed for objects and outperforms other database and ORM solutions. Because it is an embedded database, ObjectBox runs in your apps' process and needs no maintenance. Read on to learn how to create a Java project using ObjectBox. We believe it's fairly easy. Please let us know your thoughts on it.

Desktop Project Setup

Because ObjectBox comes with a Gradle plugin, you use Gradle as a build system.



There is an experimental Maven plugin available. See the [Java Maven example](#).

To get set up, in your project's `build.gradle` file, you apply the ObjectBox Gradle plugin. This must be done **after** applying the Java plugin.

This is how a Gradle build file using ObjectBox typically looks like:

Gradle 5.2 or newer

```
1  buildscript {
2      ext.objectboxVersion = '2.5.1'
3      repositories {
4          jcenter()
5      }
6      dependencies {
7          classpath "io.objectbox:objectbox-gradle-plugin:$objectboxVe
8      }
9  }
10
11 repositories {
12     jcenter()
13 }
14
15 apply plugin: 'java'
16 apply plugin: 'io.objectbox'
```



If you use IntelliJ IDEA, make sure to use 2019.1 or newer as it has improved Gradle support, like delegating build actions to Gradle.


In Gradle 5.1 or older annotation processor support is incomplete, so adding an annotation processor plugin (e.g. [gradle-apt-plugin](#)) is recommended:

```
1  buildscript {
2      ext.objectboxVersion = '2.5.1'
3      repositories {
4          jcenter()
5          maven { url "https://plugins.gradle.org/m2/" }
6      }
7      dependencies {
8          classpath "net.ltgt.gradle:gradle-apt-plugin:0.20"
9          classpath "io.objectbox:objectbox-gradle-plugin:$objectboxVersion"
10     }
11 }
12
13 repositories {
14     jcenter()
15 }
16
17 apply plugin: 'java'
18 // Annotation processor plugin for better Gradle + IntelliJ support.
19 apply plugin: 'net.ltgt.apt-idea'
20 apply plugin: 'io.objectbox'
```

This set up is the main difference between the Android and plain Java desktop setup. Other than this, **ObjectBox works the same across platforms.**

Native Libraries

Under the hood, ObjectBox is an object database running mostly in native code written in C/C++ for optimal performance (there's no way to make this work as fast in plain Java). Thus, ObjectBox will load a native library: a ".dll" on Windows, a ".so" on Linux, and a ".dylib" on macOS. By default, the ObjectBox Gradle plugin adds a dependency to the native library matching your system. This means that your app is already set up to run on your system.

 Note: on Windows you might have to install the [Microsoft Visual C++ 2015 Redistributable \(x64\)](#) packages to use the ObjectBox DLL.

Note that ObjectBox binaries are build for 64 bit systems for best performance. Talk to us if you require 32 bit support.

Add Libraries for distribution

While the default build configuration ensures that your app runs on your development system, you may want to support all major platforms (Windows, Linux, macOS) when you **distribute** your app. For this, just add all platform dependencies to your project's

`build.gradle` file like this:

```
1 dependencies {
2     // Optional: include all native libraries for distribution
3     implementation "io.objectbox:objectbox-linux:$objectboxVersion"
4     implementation "io.objectbox:objectbox-macos:$objectboxVersion"
5     implementation "io.objectbox:objectbox-windows:$objectboxVersion"
6 }
```

For reference, the following dependencies to the ObjectBox Java API and annotation processor are added **by default** by the ObjectBox Gradle plugin. Usually, there's **no** need to set them up manually:

```
1 dependencies {
2     // Added automatically by the plugin - just for reference.
3     implementation "io.objectbox:objectbox-java:$objectboxVersion"
4     annotationProcessor "io.objectbox:objectbox-processor:$objectboxVersion"
5 }
```

Optional: Change the Model File Path

By default, the ObjectBox model file is stored in

`module-name/objectbox-models/default.json` . You can change the file path and name by passing the `objectbox.modelPath` argument to the ObjectBox annotation processor.

In your project's `build.gradle` file after the java plugin, add the necessary compiler argument:

```
1 tasks.withType(JavaCompile) {
2     options.compilerArgs += [ "-Aobjectbox.modelPath=$projectDir/schemas/o
3 }
```

Optional: Change the MyObjectBox package

1.5 or newer

By default the MyObjectBox class is generated in the same or a parent package of your entity classes. You can define a specific package by passing the `objectbox.myObjectBoxPackage` argument to the ObjectBox annotation processor.

In your project's `build.gradle` file after the java plugin, add the necessary compiler argument:

```
1 tasks.withType(JavaCompile) {
2     options.compilerArgs += [ "-Aobjectbox.modelPath=$projectDir/schemas/o
3 }
```

Optional: Enable Debug Mode

You can enable debug output for the plugin and for the annotation processor if you encounter issues while setting up your project.

Just add the necessary options in your project's `build.gradle` file after the java and `io.objectbox` plugin

```
1 // enable debug output for plugin
2 objectbox {
3     debug true
4 }
5 // enable debug output for annotation processor
6 tasks.withType(JavaCompile) {
7     options.compilerArgs += [ "-Aobjectbox.debug=true" ]
8 }
```

Add Entity Classes

You must create at least one class annotated with `@Entity` to use ObjectBox. If you don't know yet how to do this, learn here [how to create and annotate entity classes](#).

The following code snippet shows how a simple entity class might look like:

```
1 @Entity
2 public class Note {
3
4     @Id
5     public long id;
6
7     public String text;
8     public String comment;
9     public Date date;
10
11     public Note(long id, String text, String comment, Date date) {
12         this.id = id;
13         this.text = text;
14         this.comment = comment;
15         this.date = date;
16     }
17
18     public Note() {
19     }
20 }
```

Build BoxStore

To get [a Box for saving your entities](#) you need to build a BoxStore first. After creating your entity classes and building your project, e.g. by running `gradlew build`, the class `MyObjectBox` will be generated. Use `MyObjectBox.builder()` to build your BoxStore.

This code snippet demonstrates a simple program that builds a BoxStore and saves a Note entity object:

```
1 public static void main(String[] args) {
2     BoxStore store = MyObjectBox.builder().name("objectbox-notes-db").build();
3     Box<Note> box = store.boxFor(Note.class);
4
5     String text = args.length > 0 ? String.join(" ", args) : "No text given";
6     box.put(new Note(text));
7
8     System.out.println(box.count() + " notes in ObjectBox database:");
9     for (Note note : box.getAll()) {
10         System.out.println(note);
11     }
12     store.close();
13 }
```

You can use the `name(String)` method of the builder to change the directory name your database files are stored in. See the [BoxStoreBuilder](#) documentation for more configuration options.

Building Unit Tests

In your project's `build.gradle` file, add the JUnit testing framework to your test dependencies:

```
1 dependencies {
2     testCompile 'junit:junit:4.12'
3 }
```

You create your unit test classes as usual under `module-name/src/test/java/`. To use `ObjectBox` in your test methods you need to build a `BoxStore` instance using the generated `MyObjectBox` class of your project. You can use the [directory\(File\) method](#) on the `BoxStore` builder to ensure the test database is stored in a specific folder on your machine. To start with a clean database for each test you can delete the existing database using [BoxStore.deleteAllFiles\(File\)](#).

The following example shows how you could implement a unit test class that uses `ObjectBox`:

```
1 public class NoteTest {
2     private File boxStoreDir;
3     private BoxStore store;
4
5     @Before
6     public void setUp() throws IOException {
7         // store the database in the systems temporary files folder
8         File tempFile = File.createTempFile("object-store-test", "");
9         // ensure file does not exist so builder creates a directory instead
10        tempFile.delete();
11        boxStoreDir = tempFile;
12        store = MyObjectBox.builder()
13            // add directory flag to change where ObjectBox puts its database
14            .directory(boxStoreDir)
15            // optional: add debug flags for more detailed ObjectBox logging
16            .debugFlags(DebugFlags.LOG_QUERIES | DebugFlags.LOG_QUERY_RESULTS)
17            .build();
18    }
19
20    @After
21    public void tearDown() throws Exception {
22        if (store != null) {
23            store.close();
24            store.deleteAllFiles();
25        }
26    }
27
28    @Test
29    public void testPutAndGet() {
30        Box<Note> box = store.boxFor(Note.class);
31        assertEquals(...);
32    }
33 }
```

To help diagnose issues you can enable log output for ObjectBox actions, such as queries, by specifying one or more [debug flags](#) when building BoxStore.

Kotlin Support

ObjectBox and Kotlin

ObjectBox comes with full Kotlin support for Android. This allows entities to be modeled in Kotlin classes (regular and data classes). With Kotlin support you can build faster apps even faster.



This page assumes that you have added ObjectBox to your project and that you are familiar with basic functionality. The Getting Started page will help you out if you are not. This page discusses additional capabilities for Kotlin only.



[Getting started](#)

[/getting-started](#)

Kotlin Entities

ObjectBox supports regular and data classes for entities. However, **@Id properties must be var** (not val) because ObjectBox assigns the ID after putting a new entity. They also should be of non-null type `Long` with the special value of zero for marking entities as new.

See the Getting Started or Entity Annotations guide for examples.



[Getting started](#)

[/getting-started](#)



[Entity Annotations](#)

[/entity-annotations](#)


Defining Relations in Kotlin Entities

When [defining relations](#) in Kotlin, keep in mind that **relation properties must be `var`**. Otherwise they can not be initialized as described in the [relations docs](#). To avoid null checks use a lateinit modifier. When using a data class this requires the relation property to be moved to the body.


See the Relations page for examples.

→ [Relations](#)

[/relations](#)

 Two data classes that have the same property values (excluding those defined in the class body) [are equal and have the same hash code](#). Keep this in mind when working with ToMany which uses a HashMap to keep track of changes. E.g. adding the same data class multiple times has no effect, it is treated as the same entity.

Using the provided extension functions

 Since 2.0.0

To simplify your code, you might want to use the Kotlin extension functions provided by ObjectBox. The library containing them is added automatically if the Gradle plugin detects a Kotlin project.

To add it manually, modify the dependencies section in your app's `build.gradle` file:

```
1 dependencies {
```

```
2     implementation "io.objectbox:objectbox-kotlin:$objectboxVersion"
3 }
```

Now have a look at what is possible with the extensions compared to standard Kotlin idioms:

Get a box:

```
1 // Regular:
2 val box = store.boxFor(DataClassEntity::class.java)
3
4 // With extension:
5 val box: Box<DataClassEntity> = store.boxFor()
```

Build a query:

```
1 // Regular:
2 val query = box.query().run {
3     equal(property, value)
4     order(property)
5     build()
6 }
7
8 // With extension:
9 val query = box.query {
10     equal(property, value)
11     order(property)
12 }
```

Use the in filter of a query:

```
1 // Regular:
2 val query = box.query().`in`(property, array).build()
3
4 // With extension:
5 val query = box.query().inValues(property, array).build()
```

Modify a [ToMany](#):

```
1 // Regular:
2 toMany.apply {
3     reset()
4     add(entity)
5     removeById(id)
6     applyChangesToDb()
7 }
8
9 // With extension:
10 toMany.applyChangesToDb(resetFirst = true) { // default is false
11     add(entity)
12     removeById(id)
13 }
```

Something missing? [Let us know](#) what other extension functions you want us to add.

Next Steps

- Check out the [Kotlin example on GitHub](#).
- Continue with [Getting Started](#).

Queries

ObjectBox queries return persisted objects that match user defined criteria. With ObjectBox DB you use the QueryBuilder class to specify criteria and create Query objects. The Query class will actually run the query and return matching objects.

QueryBuilder

The `QueryBuilder<T>` class lets you build custom queries for your entities. Create an instance via `Box.query()` .

`QueryBuilder` offers several methods to define query conditions for properties of an entity. To specify a property ObjectBox does not use Strings but meta information "underscore" classes (like `User_`) that are generated during build time. The meta information classes have a static field for each property (like `User_.firstName`). This allows to reference properties safely with compile time checks to prevent runtime errors, for example because of typos.

Here are two examples using a simple and a more complicated query:


Simple condition example: Query for all users with the first name “Joe”:

```
List<User> joes = userBox.query().equal(User_.firstName, "Joe").build().find()
```

Multiple conditions example: Get users with the first name “Joe” that are born later than 1970 and whose last name starts with “O”.

```
1 QueryBuilder<User> builder = userBox.query();
2 builder.equal(User_.firstName, "Joe")
3     .greater(User_.yearOfBirth, 1970)
4     .startsWith(User_.lastName, "O");
5 List<User> youngJoes = builder.build().find();
```

New Query API

 The new Query API is in preview. Share your [feedback in the GitHub issue](#).

ObjectBox `3.0.0-alpha1+` introduces a new query API that accepts complex nested conditions, e.g. the equivalent of `(A or B) and (C or D)` .

To build a query with the new API use `Box.query(condition)` and supply a `condition` built using entity `Property` methods, like `property.equal(value)` . All properties of an entity can be accessed using its underscore class. For example, for an entity `User` a `property` could be `User_.firstName` , a `condition` using it could be `User_.firstName.equal("Joe")` .

Simple condition example: Query for all users with the first name “Joe”.

```
1 List<User> joes = userBox.query(User_.firstName.equal("Joe"))
2     .build()
3     .find();
```

Multiple conditions example: Get users with the first name “Joe” that are born later than 1970 and whose last name starts with “O”.

Java

```
1 List<User> youngJoes = userBox.query(  
2     User_.firstName.equal("Joe")  
3         .and(User_.yearOfBirth.greater(1970))  
4         .and(User_.lastName.startsWith("O"))  
5     .build()  
6     .find();
```

Kotlin

```
1 val youngJoes = userBox.query(  
2     User_.firstName equal "Joe"  
3         and (User_.yearOfBirth greater 1970)  
4         and (User_.lastName startsWith "O")  
5     .build()  
6     .find();
```

Conditions are combined with `QueryCondition.and(condition)` or `QueryCondition.or(condition)`. AND has precedence over OR.

To nest conditions pass a single or combined `condition`.

Java

```
1 // equal AND (less OR oneOf)  
2 Query<User> query = box.query(  
3     User_.firstName.equal("Joe")  
4         .and(User_.age.less(12)  
5             .or(User_.stamp.oneOf(new long[]{1012})))  
6     .order(User_.age)  
7     .build();
```

```
1 // equal AND (less OR oneOf)
2 val query = box.query(
3     User_.firstName equal "Joe"
4     and (User_.age less 12
5         or (User_.stamp oneOf longArrayOf(1012)))
6     .order(User_.age)
7     .build()
```

Other notable changes

- `property.oneOf(array)` replaces the `in(property, array) (inValues(property, array) for Kotlin)` condition.
- Use Kotlin infix functions to write `condition and condition` and `condition or condition`.
- Use `condition.alias(aliasName)` to set an alias for a `condition` that can later be used with the `Query setParameter` methods.

Notable conditions

In addition to expected conditions like `equal()`, `notEqual()`, `greater()` and `less()` there are also conditions like:

- `isNull()` and `notNull()`,
- `between()` to filter for values that are between the given two,
- `in()` and `notIn()` to filter for values that match any in the given array,
- `startsWith()`, `endsWith()` and `contains()` for extended String filtering.

In addition, there is `and()` and `or()` to build more complex combinations of conditions.

For an overview of all available criteria, please refer to the [QueryBuilder](#) class and its method documentations.

Ordering results

In addition to specifying conditions you can order the returned results using the `order()` method:

```
1 userBox.query().equal(User_.firstName, "Joe")
2   .order(User_.lastName) // in ascending order, ignoring case
3   .find();
```

You can also pass flags to `order()` to sort in descending order, to sort case sensitive or to specially treat null values. For example to sort the above results in descending order and case sensitive instead:

```
1 userBox.query().equal(User_.firstName, "Joe")
2   .order(User_.lastName, QueryBuilder.DESENDING | QueryBuilder.CASE_SENSITIVE)
3   .find();
```

Order conditions can also be chained. Check the [method documentation](#) for details.

Query

[Queries](#) are created (and not yet executed) by calling `build()` on the `QueryBuilder`.

```
Query<User> query = builder.build();
```

Finding objects

There are a couple of find methods to retrieve objects matching the query:


```
1 // return all entities matching the query
2 List<User> joes = query.find();
3
4 // return only the first result or null if none
5 User joe = query.findFirst();
6
7 // return the only result or null if none, throw if more than one result
8 User joe = query.findUnique();
```

To return all entities matching the query simply call `find()` .

To only return the first result, use `findFirst()` .

If you expect a unique result call `findUnique()` instead. It will give you a single result or null, if no matching entity was found and throw an exception if there was more than one result.

Reusing Queries and Parameters

If you frequently run a `Query` you should cache the `Query` object and re-use it. To make a `Query` more reusable you can change the values, or query parameters, of each condition you added even after the `Query` is built. Let's see how.

Assume we want to find a list of `User` with specific `firstName` values. First, we build a regular `Query` with an `equal()` condition for `firstName` . Because we have to pass an initial parameter value to `equal()` but plan to override it before running the `Query` later, we just pass an empty string:

```
1 // build query
2 Query<User> query = userBox.query().equal(User_.firstName, "").build();
```

Now at some later point we want to actually run the `Query` . To set a value for the `firstName` parameter we call `setParameter()` on the `Query` and pass the `firstName` property and the new parameter value:

```

1 // change firstName parameter to "Joe", get results
2 List<User> joes = query.setParameter(User_.firstName, "Joe").find();
3 ...
4 // change firstName parameter to "Jake", get results
5 List<User> jakes = query.setParameter(User_.firstName, "Jake").find();

```

So you might already be wondering, what happens if you have more than one condition using `firstName` ? For this purpose you can **assign each parameter an alias** by calling `parameterAlias()` right after specifying the condition:

```

1 // assign alias "name" to the equal() query parameter
2 Query<User> query = userBox.query()
3     .equal(User_.firstName, "").parameterAlias("name");

```

Then, when setting a new parameter value pass the alias instead of the property:

```

1 // change parameter with alias "name" to "Joe", get results
2 List<User> joes = query.setParameter("name", "Joe").find();

```

Limit, Offset, and Pagination

Sometimes you only need a subset of a query, for example the first 10 elements to display in your user interface. This is especially helpful (and resourceful) when you have a high number of entities and you cannot limit the result using query conditions only. The built `Query<T>` has a `.find(long offset, long limit)` method with offset and limit arguments:

```

1 Query<User> query = userBox.query().equal(UserProperties.FirstName, "Joe")
2 List<User> joes = query.find(/* offset by */ 10, /* limit to */ 5 /* result

```

offset: The first `offset` results are skipped.

`limit:` At most `limit` results of this query are returned.

Lazy loading results

To avoid loading query results right away, Query offers `findLazy()` and `findLazyCached()` which return a `LazyList` of the query results.

`LazyList` is a thread-safe, unmodifiable list that reads entities lazily only once they are accessed. Depending on the find method called, the lazy list will be cached or not. Cached lazy lists store the previously accessed objects to avoid loading entities more than once. Some features of the list are limited to cached lists (e.g. features that require the entire list). See the [LazyList class documentation](#) for more details.

Removing Objects

To remove all objects matching a query, call `query.remove()` .

PropertyQuery

If you only want to return the values of a certain property and not a list of full objects you can use a `PropertyQuery`. After building a query simply call `property(Property)` to define the property followed by the appropriate find method.

For example, instead of getting all `User` s, to just get their email addresses:

```
1 String[] emails = userBox.query().build()
2   .property(User_.email)
3   .findStrings();
```

In general there is always a find method to just return the value of the first result, like `findString()` , or the values of all results, like `findStrings()` .



Note: the returned array of property values is **not in any particular order**, even if you did specify an order when building the query.

Handling null values

By default null values are not returned. However, you can specify a replacement value to return if a property is null:

```
1 // returns 'unknown' if email is null
2 String[] emails = userBox.query().build()
3     .property(User_.email)
4     .nullValue("unknown")
5     .findStrings();
```

Distinct and unique results

The property query can also only return distinct values:

```
1 // returns 'joe'
2 String[] names = userBox.query().build()
3     .property(User_.firstName)
4     .distinct()
5     .findStrings();
```

By default the case of strings is ignored. However, the distinct call can be overloaded to enable case sensitivity:

```
1 // returns 'Joe', 'joe', 'JOE'
2 String[] names = userBox.query().build()
3     .property(User_.firstName)
4     .distinct(StringOrder.CASE_SENSITIVE)
5     .findStrings();
```

If only a single value is expected to be returned the query can be configured to throw if that is not the case:

```
1 // throws if not exactly one name
2 String[] names = userBox.query().build().equal(User_.isAdmin, true)
3   .property(User_.firstName)
4   .unique()
5   .findStrings();
```


The distinct and unique flags can be combined.

Aggregating values

Property queries ([JavaDoc](#)) also offer aggregate functions to directly calculate the minimum, maximum, average, sum and count of all found values:

- `min()` / `minDouble()` : Finds the minimum value for the given property over all objects matching the query.
- `max()` / `maxDouble()` : Finds the maximum value.
- `sum()` / `sumDouble()` : Calculates the sum of all values. *Note: the non-double version detects overflows and throws an exception in that case.*
- `avg()` : Calculates the average (always a double) of all values.
- `count()` : returns the number of results. This is faster than finding and getting the length of the result array. Can be combined with `distinct()` to count only the number of distinct values. *Since 2.0.0.*

Add query conditions for related entities (links)

 Since 2.0.0

After creating a relation between entities, you might want to add a query condition for a property that only exists in the related entity. In SQL this is solved using JOINS. But as ObjectBox is not a SQL database we built something very similar: links. Let's see how this works using an example.

Assume there is a `Person` that can be associated with multiple `Address` entities:

Java

```
1 @Entity
2 public class Person {
3     @Id long id;
4     String name;
5     ToMany<Address> addresses;
6 }
7 @Entity
8 public class Address {
9     @Id long id;
10    String street;
11    String zip;
12 }
```

Kotlin

```
1 @Entity
2 class Person {
3     @Id var id: Long = 0
4     var name: String? = null
5     lateinit var addresses: ToMany<Address>
6 }
7 @Entity
8 class Address {
9     @Id var id: Long = 0
10    var street: String? = null
11    var zip: String? = null
12 }
```

To get a `Person` with a certain name that also lives on a specific street, we need to query the associated `Address` entities of a `Person`. To do this, use the `link(RelationInfo)` method of the query builder to tell that the `addresses` relation should be queried. Then add a condition for `Address`:

Java

```
1 // get all Person objects named "Elmo"...
2 QueryBuilder<Person> builder = personBox
3     .query().equal(Person_.name, "Elmo");
4 // ...which have an address on "Sesame Street"
5 builder.link(Person_.addresses).equal(Address_.street, "Sesame Street");
6 List<Person> elmosOnSesameStreet = builder.build().find();
```

Kotlin

```
1 // get all Person objects named "Elmo"...
2 val builder = personBox.query().equal(Person_.name, "Elmo")
3 // ...which have an address on "Sesame Street"
4 builder.link(Person_.addresses).equal(Address_.street, "Sesame Street")
5 val elmosOnSesameStreet = builder.build().find()
```

What if we want to get a list of `Address` instead of `Person`? If you know ObjectBox relations well, you would probably add a `@Backlink` relation to `Address` and build your query using it with `link(RelationInfo)` as shown above:

Java

```
1 @Entity
2 public class Address {
3     // ...
4     @Backlink(to = "addresses")
5     ToMany<Person> persons;
6 }
7
8 // get all Address objects with street "Sesame Street"...
9 QueryBuilder<Address> builder = addressBox
10     .query().equal(Address_.street, "Sesame Street");
11 // ...which are linked from a Person named "Elmo"
12 builder.link(Address_.persons).equal(Person_.name, "Elmo");
13 List<Address> sesameStreetsWithElmo = builder.build().find();
```

Kotlin

```
1 @Entity
2 class Address {
3     // ...
4     @Backlink(to = "addresses")
5     lateinit var persons: ToMany<Person>
6 }
7
8 // get all Address objects with street "Sesame Street"...
9 val builder = addressBox.query().equal(Address_.street, "Sesame Stre
10 // ...which are linked from a Person named "Elmo"
11 builder.link(Address_.persons).equal(Person_.name, "Elmo")
12 val sesameStreetsWithElmo = builder.build().find()
```

But actually you do not have to modify the `Address` entity (you still can if you need the `@Backlink` elsewhere). Instead we can use the `backlink(RelationInfo)` method to create a backlink to the `addresses` relation from `Person` just for that query:

Java

```
1 // get all Address objects with street "Sesame Street"...
2 QueryBuilder<Address> builder = addressBox.query()
3   .equal(Address_.street, "Sesame Street");
4 // ...which are linked from a Person named "Elmo"
5 builder.backlink(Person_.addresses).equal(Person_.name, "Elmo");
6 List<Address> sesameStreetsWithElmo = builder.build().find();
```

Kotlin

```
1 // get all Address objects with street "Sesame Street"...
2 val builder = addressBox.query().equal(Address_.street, "Sesame Stre
3 // ...which are linked from a Person named "Elmo"
4 builder.backlink(Person_.addresses).equal(Person_.name, "Elmo")
5 val sesameStreetsWithElmo = builder.build().find()
```

Eager Loading of Relations

By default [relations](#) are loaded lazily: when you first access a `ToOne` or `ToMany` property it will perform a database lookup to get its data. On each subsequent access it will use a cached version of that data.

Java

```
1 List<Customer> customers = customerBox.query().build().find();
2 // Customer has a ToMany called orders.
3 // First access: this will cause a database lookup.
4 Order order = customers.get(0).orders.get(0);
```

Kotlin

```
1 val customers = customerBox.query().build().find()
2 // Customer has a ToMany called orders
3 val order = customers[0].orders[0] // first access: this will cause
```

While this initial lookup is fast, you might want to prefetch `ToOne` or `ToMany` values before the query results are returned. To do this call the `QueryBuilder.eager` method when building your query and pass the `RelationInfo` objects associated with the `ToOne` and `ToMany` properties to prefetch:

Java

```
1 List<Customer> customers = customerBox.query()
2     .eager(Customer_.orders) // Customer has a ToMany called orders.
3     .build()
4     .find();
5 // First access: this will cause a database lookup.
6 Order order = customers.get(0).orders.get(0);
```

Kotlin

```
1 val customers = customerBox.query()
2     .eager(Customer_.orders) // Customer has a ToMany called orders
3     .build()
4     .find()
5 customers[0].orders[0] // first access: this will NOT cause a databa
```

Eager loading only works one level deep. If you have **nested relations** and you want to prefetch relations of all children, you can instead add a query filter as described below. Use

it to simply access all relation properties, which triggers them to lookup their values as described above.

Query filters

Query filters come into play when you are looking for objects that need to match complex conditions, which cannot be fully expressed with the `QueryBuilder` class. Filters are written in Java and thus can express any complexity. Needless to say that database conditions can be matched more efficiently than Java based filters. Thus you will get best results when you use both together:

1. Narrow down results using standard database conditions to a reasonable number (use `QueryBuilder` to get “candidates”)
2. Now filter those candidates using the [QueryFilter](#) Java interface to identify final results

A `QueryFilter` implementation looks at one candidate object at a time and returns true if the candidate is a result or false if not.

Example:

```
1 // Reduce object count to reasonable value.
2 songBox.query().equal(Song_.bandId, bandId)
3     // Filter is performed on candidate objects.
4     .filter((song) -> song.starCount * 2 > song.downloads);
```

Notes on performance: 1) `ObjectBox` creates objects very fast. 2) The virtual machine is tuned to garbage collect short lived objects. Notes 1) and 2) combined makes a case for filtering because `ObjectBox` creates candidate objects of which some are not used and thus get garbage collected quickly after their creation.

Query filters and ToMany relation

The `ToMany` class offers additional methods that can be convenient in query filters:

- hasA: returns true if one of elements matches the given QueryFilter
- hasAll: returns true if all of elements match the given QueryFilter
- getById: return the element with the given ID (value of the property with the @Id annotation)

Data Observers & Rx

ObjectBox - Data Observers and Reactive Extensions

ObjectBox makes it easy for your app to react to data changes by providing:

- data observers,
- reactive extensions,
- and an optional library to work with RxJava.

This makes setting up data flows easy while taking care of threading details.

Reactive Observers: A first Example

Let's start with an example to demonstrate what you can do with reactive data observers:

```
1 Query<Task> query = taskBox.query().equal(Task_.complete, false).build();
2 query.subscribe(subscriptions)
3     .on(AndroidScheduler.mainThread())
4     .observer(data -> updateUi(data));
```

The first line creates a regular query to get `Task` objects where `task.complete == false`. The second line connects an observer to the query. This is what happens:

- the query is executed in the background
- once the query finishes the observer gets the result data
- whenever changes are made to `Task` objects in the future, the query will be executed again
- once updated query results are in, they are propagated to the observer
- the observer is called on Android's main thread

So it's just two lines of code – but a lot of stuff happening behind the scenes.

What we did not cover yet is the parameter of the call `subscribe(subscriptions)` : this is used to track all the subscriptions so they can be canceled with one call once your app wants to stop observing.


Now, let's dive into the details.

Data Observers Basics

When objects change, ObjectBox notifies subscribed data observers. They can either subscribe to changes of certain object types (via `BoxStore`) or to query results. To create a data observer you need to implement the generic `io.objectbox.reactive.DataObserver` interface:

```
1 public interface DataObserver<T> {  
2     void onData(T data);  
3 }
```

This observer will be called by ObjectBox when necessary: typically shortly after subscribing and when data changes.

 Note: `onData()` is called asynchronously and decoupled from the thread causing the data change (like the thread that committed a transaction).


Observing General Changes

`BoxStore` allows a `DataObserver` to subscribe to object types. Let's say we have a to-do list app where `Task` objects get added. To get notified when `Task` objects are added in another place in our app we can do the following:

```
1 DataObserver<Class<Task>> taskobserver = new DataObserver<Class<Task>>() {
```

```
2     @Override public void onData(Class<Note> data) {  
3         // do something  
4     }  
5 };  
6 boxStore.subscribe(Task.class).observer(taskObserver);
```

Here `onData()` is not called with anything useful as data. If you need more than being notified, like to get a list of `Task` objects following the above example, read on to learn how to observe queries.

 Note: there is also `subscribe()` which takes no arguments. It subscribes the observer to receive changes for all available object classes.

Observing Queries

ObjectBox let's you **build queries** to find the objects matching certain criteria. Queries are an essential part of ObjectBox: whenever you need a specific set of data, you will probably use a query.

Combining queries and observers results in a convenient and powerful tool: query observers will automatically deliver fresh results whenever changes are made to entities in a box. Let's say you display a list of to-do tasks in your app. You can use a `DataObserver` to get all tasks that are not yet completed and pass them to a method `updateUi()` (note that we are using lambda syntax here):

```
1 Query<Task> query = taskBox.query().equal(Task_.completed, false).build();  
2 subscription = query.subscribe().observer(data -> updateUi(data));
```

So when is our observer lambda called? Immediately when an observer is subscribed, the query will be run in a separate thread. Once the query result is available, it will be passed to the observer. This is the first call to the observer.

Now let's say a task gets changed and stored in `ObjectBox`. It doesn't matter where and how; it might be the user who marked a task as completed, or some backend thread putting additional tasks during synchronization with a server. In any case, the query will notify all observers with (potentially) updated query results.

Note that this pattern can greatly simplify your code: there is a single place where your data comes in to update your user interface. There is no separate initialization code, no wiring of events, no re-running queries, etc.

See the [subscribe\(\)-method](#) documentation for more details.

Canceling Subscriptions

When you call `observer()`, it returns a subscription object implementing the `io.objectbox.reactive.DataSubscription` interface:

```
1 public interface DataSubscription {
2     void cancel();
3     boolean isCanceled();
4 }
```

So, if you plan to unsubscribe your `DataObserver` later, it is a good idea to hold on to the `DataSubscription`. Call `cancel()` on it to let `ObjectBox` know that the observer should not be notified anymore:


```
1 DataSubscription subscription = boxStore.subscribe().observer(myObserver);
2
3 // At some later point:
4 subscription.cancel();
```

Because you often have more than one query subscription, we generally recommend using `DataSubscriptionList` instead of keeping track of potentially multiple `DataSubscription` objects. The basic pattern goes like this:

```

1 private DataSubscriptionList subscriptions = new DataSubscriptionList();
2
3 protected void onStart() {
4     super.onStart();
5     Query<X> query = box.query()... .build();
6     query.subscribe(subscriptions)... .observe(...);
7 }
8
9 protected void onStop() {
10    super.onStop();
11    subscriptions.cancel();
12 }

```

 Note: On Android, you would typically create the subscription in one of the `onCreate()/onStart()/onResume()` lifecycle methods and cancel it in its counterpart `onDestroy()/onStop()/onPause()` .

Observers and Transactions

Observer notifications occur after a transaction is committed. For some scenarios it is especially important to know transaction bounds. If you call `box.put()` or `remove()` individually, an implicit transaction is started and committed. For example, this code fragment would trigger data observers on `User.class` twice:

```

1 box.put(friendUser);
2 box.put(myUser);

```

There are several ways to combine several operations into one transaction, for example using one of the `runInTx()` or `callInTx()` methods in the `BoxStore` class. For our simple example, we can simply use an overload of `put()` accepting multiple objects:

```
box.put(friendUser, myUser);
```

This results in a single transaction and thus in a single `DataObserver` notification.

Reactive Extensions

In the first part you saw how data observers can help you keep your app state up to date. But there is more: ObjectBox comes with simple and convenient reactive extensions for typical tasks. While most of these are inspired by RxJava, they are not actually based on RxJava. ObjectBox brings its own features because not all developers are familiar with RxJava (for the RxJava ObjectBox library see below). We do not want to impose the complexity (Rx is almost like a new language to learn) and size of RxJava (~10k methods) on everyone. So, let's keep it simple and neat for now.

Thread Scheduling

On Android, UI updates must occur on the main thread only. Luckily, ObjectBox allows to switch the observer from a background thread over to the main thread. Let's take a look on a revised version of the to-do task example from above:

```
1 Query<Task> query = taskBox.query().equal(Task_.complete, false).build();
2 query.subscribe().on(AndroidScheduler.mainThread()).observer(data -> update)
```

Where is the difference? The additional `on()` call is all that is needed to tell where we want our observer to be called. `AndroidScheduler.mainThread()` is a built-in scheduler implementation. Alternatively, you can create an `AndroidScheduler` using a custom `Looper`, or build a fully custom scheduler by implementing the `io.objectbox.reactive.Scheduler` interface.

Transforming Data

Maybe you want to transform the data before you hand it over to an observer. Let's say, you want to keep track of the count of all stored objects for each type. The `BoxStore`

subscription gives you the classes of the objects, and this example shows you how to transform them into actual object counts:

```
1 boxStore.subscribe()  
2   .transform(clazz -> return boxStore.boxFor(clazz).count())  
3   .observer(count -> updateCount(count));
```

Note that the transform operation takes a `Class` object and returns a `Long` number. Thus the `DataObserver` receives the object count as a `Long` parameter in `onData()`.

While the lambda syntax is nice and brief, let's look at the

`io.objectbox.reactive.Transformer` interface for clarification of what the `transform()` method expects as a parameter:

```
1 public interface DataTransformer<FROM, TO> {  
2     TO transform(FROM source) throws Exception;  
3 }
```

Some additional notes on transformers:

- Transforms are not required to actually “transform” any data. Technically, it is fine to return the same data that is received and just do some processing with (or without) it.
- Transformers are always executed asynchronously. It is fine to perform long lasting operations.

ErrorObserver

Maybe you noticed that a transformer may throw any type of exception. Also, a `DataObserver` might throw a `RuntimeException`. In both cases, you can provide an `ErrorObserver` to be notified about an exception that occurred. The `io.objectbox.reactive.ErrorObserver` is straight-forward:

```
1 public interface ErrorObserver {  
2     void onError(Throwable th);  
3 }
```

To specify your `ErrorObserver` , simply call the `onError()` method after `subscribe()` .

Single Notifications vs. Only-Changes

When you subscribe to a query, the `DataObserver` gets both of the following by default:

- Initial query results (right after subscribing)
- Updated query results (underlying data was changed)

Sometimes you may be interested in only one of those. This is what the methods `single()` and `onlyChanges()` are for (call them after `subscribe()`). Single subscriptions are special in the way that they are cancelled automatically once the observer is notified. You can still cancel them manually to ensure no call to the observer is made at a certain point.

Weak References

Sometimes it may be nice to have a weak reference to a data observer. Note that for the sake of a deterministic flow, it is advisable to cancel subscriptions explicitly whenever possible. If that does not scare you off, use `weak()` after `subscribe()` .

Threading overview

To summarize threading as discussed earlier:

- Query execution runs on a background thread (exclusive for this task)
 - `DataTransformer` runs on a background thread (exclusive for this task)
 - `DataObserver` and `ErrorObserver` run on a background thread unless a scheduler is specified via the `on()` method.
-

ObjectBox RxJava Extension Library

By design, there are zero dependencies on any Rx libraries in the core of ObjectBox. As you saw before ObjectBox gives you simple means to transform data, asynchronous processing, thread scheduling, and one time (single) notifications. However, you still might want to integrate with the mighty RxJava 2 (we have no plans to support RxJava 1). For this purpose we created the ObjectBox RxJava extension library:

```
implementation "io.objectbox:objectbox-rxjava:$objectboxVersion"
```

It provides the classes `RxQuery` and `RxBoxStore`. Both offer static methods to subscribe using RxJava means.

For general object changes, you can use `RxBoxStore` to create an `Observable`.

`RxQuery` allows to subscribe to query objects using:

- `Flowable`
- `Observable`
- `Single`

Example usage:

```
1 Query query = box.query().build();
2 RxQuery.observable(query).subscribe(this);
```

The extension library is open-source and available [GitHub](#).

Relations

ObjectBox - Relations

Prefer to dive right into code? Check the relation project in the [example repo](#).

Objects may reference other objects, for example using a simple reference or a list of objects. In database terms, we call those references **relations**. The object defining the relation we call the **source** object, the referenced object we call **target** object. So the relation has a direction.

If there is one target object, we call the relation **to-one**. And if there can be multiple target objects, we call it **to-many**. Relations are lazily initialized: the actual target objects are fetched from the database when they are first accessed. Once the target objects are fetched, they are cached for further accesses.

To-One Relations



To-One Relations

You define a to-one relation using the `ToOne` class, a smart proxy to the target object. It gets and caches the target object transparently. For example, an order is typically made by one customer. Thus, we could model the `Order` class to have a to-one relation to the `Customer` like this:

Java

```
1 // Customer.java
2 @Entity
3 public class Customer {
4
5     @Id public long id;
6
7 }
8
9 // Order.java
10 @Entity
11 public class Order {
12
13     @Id public long id;
14
15     public ToOne<Customer> customer;
16
17 }
```

Kotlin

```
1 @Entity
2 data class Customer(
3     @Id var id: Long = 0
4 )
5
6 @Entity
7 data class Order(
8     @Id var id: Long = 0
9 ) {
10     lateinit var customer: ToOne<Customer>
11 }
```

Now let's add a customer and some orders. **To set** the related customer object, call `setTarget()` (or assign `target` in Kotlin) on the `ToOne` instance **and put** the order object:

Java

```
1 Customer customer = new Customer();
2 Order order = new Order();
3 order.customer.setTarget(customer);
4 // Puts order and customer:
5 long orderId = boxStore.boxFor(Order.class).put(order);
```

Kotlin

```
1 val customer = Customer()
2 val order = Order()
3 order.customer.target = customer
4 // Puts order and customer:
5 val orderId = boxStore.boxFor(Order::class.java).put(order)
```

If the customer object does not yet exist in the database, the ToOne will put it. If it already exists, the ToOne will only create the relation (but not put it). **See further below for details about [updating relations](#).**



Note: if your related entity uses self-assigned IDs with `@Id(assignable = true)` it will not be inserted. See below about [updating ToOne](#) for details.

To **get** the customer of an order call `getTarget()` (or access `target` in Kotlin) on the ToOne instance:

Java

```
1 Order order = boxStore.boxFor(Order.class).get(orderId);
2 Customer customer = order.customer.getTarget();
```

Kotlin

```
1 val order = boxStore.boxFor(Order::class.java)[orderId]
2 val customer = order.customer.target
```

This will do a database call on the first access (lazy loading). It uses lookup by ID, which is very fast in ObjectBox. If you only need the ID instead of the whole target object, call `getTargetId()` instead. It can be more efficient because it does not touch the database at all.

We can also **remove** the relationship to a customer:

Java

```
1 order.customer.setTarget(null);
2 boxStore.boxFor(Order.class).put(order);
```

Kotlin

```
1 order.customer.target = null
2 boxStore.boxFor(Order::class.java).put(order)
```

Note that this does not remove the customer from the database, it just dissolves the relationship.

How ToOne works behind the scenes

If you look at your model in `objectbox-models/default.json` you can see, a ToOne property is not actually stored. Instead the ID of the target object is saved in a virtual property named like the ToOne property appended with *Id*.

Expose the ToOne target ID property

You can directly access the target ID property by defining a `long` (or `Long` in Kotlin) property in your entity class with the expected name:

Java

```
1 @Entity
2 public class Order {
3     @Id public long id;
4
5     public long customerId; // ToOne target ID property
6     public ToOne<Customer> customer;
7 }
```

Kotlin

```
1 @Entity
2 data class Order(
3     @Id var id: Long = 0,
4     var customerId: Long = 0
5 ) {
6     lateinit var customer: ToOne<Customer>
7 }
```

You can change the name of the expected target ID property by adding the `@TargetIdProperty(String)` annotation to a `ToOne`.

Initialization Magic

Did you notice that the `ToOne` field `customer` was never initialized in the code example above? Why can the code still use `customer` without any `NullPointerException`? Because the field actually is initialized – the initialization code just is not visible in your sources.

The ObjectBox Gradle plugin will transform your entity class (**only supported for plain Java and Android projects**) to do the proper initialization in constructors before your code is executed. Thus, even in your constructor code, you can just assume `ToOne` and `ToMany / List` properties have been initialized and are ready for you to use:

```
1  @Entity
2  public class Example {
3
4      ToOne<Order> order;
5      ToMany<Order> orders;
6
7      transient BoxStore __boxStore; // <-- Transform adds this field
8
9      public Example() {
10         // Transform inits ToOne and ToMany in constructors
11         // not calling another constructor
12         this.order = new ToOne<>(this, Example_.order);
13         this.orders = new ToMany<>(this, Example_.orders);
14     }
15
16     public Example(String value) {
17         this();
18         // Calls another constructor, so Transform does not add init
19     }
20
21 }
```



If your setup does not support transformations, like non-Android Kotlin code, add the above modifications yourself. You also will have to call `box.attach(entity)` before modifying ToOne or ToMany properties.

Improve Performance

To improve performance when ObjectBox constructs your entities, you should **provide an all-properties constructor**.

For a ToOne you have to add an id parameter, typically named like the ToOne field appended with `Id`. Check your `objectbox-models/default.json` file to find the correct name.

An example:

```
1  @Entity
2  public class Order {
3
4      @Id public long id;
5
6      public ToOne<Customer> customer;
7
8      public Order() { /* default constructor */ }
9
10     public Order(long id, long customerId /* virtual ToOne id property */)
11         this.id = id;
12         this.customer.setTargetId(customerId);
13     }
14
15 }
```

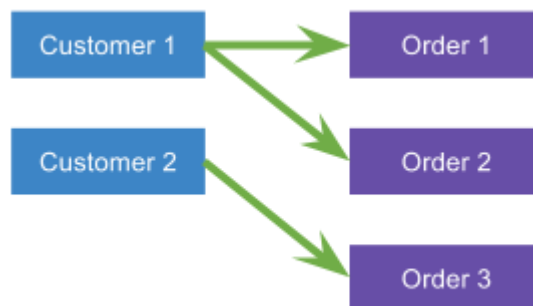
To-Many Relations

To define a to-many relation, you can use a property of type `List` or the `ToMany` class. As the `ToOne` class, the `ToMany` class helps you to keep track of changes and to apply them to the database. If you do not need or want that, use type `List` and take care of applying database changes yourself.

Note that **to-many relations are resolved lazily** on first access, and then **cached** in the source entity inside the `ToMany` object. So subsequent calls to any method, like `size()` of the `ToMany`, do not query the database, even if the relation was changed elsewhere. To get the latest data fetch the source entity again or call `reset()` on the `ToMany`.

There is a slight difference if you require a one-to-many (1:N) or many-to-many (N:M) relation. A 1:N relation is like the example above where a customer can have multiple orders, but an order is only associated with a single customer. An example for an N:M relation are students and teachers: students can have classes by several teachers but a teacher can also instruct several students.

One-to-Many (1:N)



One-to-Many (1:N)

To define a one-to-many relation, you need to annotate your relation property with `@Backlink`. It links back to a to-one relation in the target object. Using the customer and orders example, we can modify the customer class to have a to-many relation to the customers orders:

```
1 // Customer.java
2 @Entity
3 public class Customer {
4
5     @Id public long id;
6
7     // 'to' is optional if only one relation matches.
8     @Backlink(to = "customer")
9     public ToMany<Order> orders;
10
11 }
12
13 // Order.java
14 @Entity
15 public class Order {
16
17     @Id public long id;
18
19     public ToOne<Customer> customer;
20
21 }
```

Kotlin

```
1 @Entity
2 data class Customer(
3     @Id var id: Long = 0
4 ) {
5     // 'to' is optional if only one relation matches.
6     @Backlink(to = "customer")
7     lateinit var orders: ToMany<Order>
8 }
9
10 @Entity
11 data class Order(
12     @Id var id: Long = 0
13 ) {
14     lateinit var customer: ToOne<Customer>
15 }
```

The `@Backlink` annotation tells `ObjectBox` which `ToOne` relation to use to populate the list of orders. If there would be multiple to-one relations using `Customer` inside the `Order` class, you would need to explicitly specify the name like `@Backlink(to = "customer")`.

Let's **add** some orders together with a new customer. The `ToMany` implements the Java `List` interface, so we can simply add orders to it:

Java

```
1 Customer customer = new Customer();
2 customer.orders.add(new Order("Order 1"));
3 customer.orders.add(new Order("Order 2"));
4 // Puts customer and orders:
5 long customerId = boxStore.boxFor(Customer.class).put(customer);
```

```
1 val customer = Customer()
2 customer.orders.add(Order(text = "Order 1"))
3 customer.orders.add(Order(text = "Order 2"))
4 // Puts customer and orders:
5 val customerId = boxStore.boxFor(Customer::class.java).put(customer)
```

i **For Kotlin:** two data classes that have the same property values (excluding those defined in the class body) [are equal and have the same hash code](#). Keep this in mind when working with ToMany which uses a HashMap to keep track of changes. E.g. adding the same data class multiple times has no effect, it is treated as the same entity.

If the order entities do not yet exist in the database, the ToMany will put them. If they already exist, the ToMany will only create the relation (but not put them). See further below for details about [updating relations](#).

i **Note:** if your entities use self-assigned IDs with `@Id(assignable = true)` the above will not work. See below about [updating ToMany](#) for details.

We can easily **get** the orders of a customer back by accessing the list of orders:

Java

```
1 Customer customer = boxStore.boxFor(Customer.class).get(customerId);
2 for (Order order : customer.orders) {
3     // Do something with each order.
4 }
```

Kotlin

```
1 val customer = boxStore.boxFor(Customer::class.java).get(customerId)
2 for (order in customer.orders) {
3     // Do something with each order.
4 }
```

Removing orders from the relation works as expected:

Java

```
1 // Remove the relation to an order:
2 Order order = customer.orders.remove(0);
3 boxStore.boxFor(Customer.class).put(customer);
4 // Optional: also remove the order entity from its box:
5 // boxStore.boxFor(Order.class).remove(order);
```

Kotlin

```
1 // Remove the relation to an order:
2 val order = customer.orders.removeAt(0)
3 boxStore.boxFor(Customer::class.java).put(customer)
4 // Optional: also remove the order entity from its box:
5 // boxStore.boxFor(Order::class.java).remove(order)
```

Many-to-Many (N:M)



Many-to-Many (N:M)

To define a many-to-many relation you simply add a property using the `ToMany` class. Assuming a students and teachers example, this is how a simple student class that has a to-many relation to teachers can look like:

Java

```
1 // Teacher.java
2 @Entity
3 public class Teacher{
4
5     @Id public long id;
6
7 }
8
9 // Student.java
10 @Entity
11 public class Student{
12
13     @Id public long id;
14
15     public ToMany<Teacher> teachers;
16
17 }
```

Kotlin

```
1 @Entity
2 data class Teacher(
3     @Id var id: Long = 0
4 )
5
6 @Entity
7 data class Student(
8     @Id var id: Long = 0
9 ) {
10     lateinit var teachers: ToMany<Teacher>
11 }
```

Adding the teachers of a student works exactly like with a list:

Java

```
1 Teacher teacher1 = new Teacher();
2 Teacher teacher2 = new Teacher();
3
4 Student student1 = new Student();
5 student1.teachers.add(teacher1);
6 student1.teachers.add(teacher2);
7
8 Student student2 = new Student();
9 student2.teachers.add(teacher2);
10
11 // Puts students and teachers:
12 boxStore.boxFor(Student.class).put(student1, student2);
```

Kotlin

```
1 val teacher1 = Teacher()
2 val teacher2 = Teacher()
3
4 val student1 = Student()
5 student1.teachers.add(teacher1)
6 student1.teachers.add(teacher2)
7
8 val student2 = Student()
9 student2.teachers.add(teacher2)
10
11 // Puts students and teachers:
12 boxStore.boxFor(Student::class.java).put(student1, student2)
```

If the teacher entities do not yet exist in the database, the ToMany will also put them. If they already exist, the ToMany will only create the relation (but not put them). See further below for details about [updating relations](#).



Note: if your entities use self-assigned IDs with `@Id(assignable = true)` the above will not work. See below about [updating ToMany](#) for details.

To **get** the teachers of a student we just access the list:

Java

```
1 Student student = boxStore.boxFor(Student.class).get(studentId);
2 for (Teacher teacher : student.teachers) {
3     // Do something with each teacher.
4 }
```

Kotlin


```
1 val student = boxStore.boxFor(Student::class.java).get(studentId)
2 for (teacher in student.teachers) {
3     // Do something with each teacher.
4 }
```

And if a student drops out of a class, we can **remove** a teacher from the relation:


Java

```
1 student.teachers.remove(0);
2 // Simply put the student again:
3 // boxStore.boxFor(Student.class).put(student);
4 // Or more efficient than using put:
5 student.teachers.applyChangesToDb();
```

```
1 student.teachers.removeAt(0)
2 // Simply put the student again:
3 // boxStore.boxFor(Student::class.java).put(student)
4 // Or more efficient than using put:
5 student.teachers.applyChangesToDb()
```

 Note: instead of using `put()` you can also use `applyChangesToDb()` of the `ToMany` to persist changes that affect it only.

Access Many-To-Many in the reverse direction

 Since 2.0.0

Following the above example, you might want an easy way to find out what students a teacher has. Instead of having to perform a query, you can just add a to-many relation to the teacher and annotate it with the `@BackLink` annotation:

```
1 // Teacher.java
2 @Entity
3 public class Teacher{
4
5     @Id public long id;
6
7     // Backed by the to-many relation in Student:
8     @Backlink(to = "teachers")
9     public ToMany<Student> students;
10
11 }
12
13 // Student.java
14 @Entity
15 public class Student{
16
17     @Id public long id;
18
19     public ToMany<Teacher> teachers;
20
21 }
```

```
1 @Entity
2 data class Teacher(
3     @Id var id: Long = 0
4 ) {
5     // Backed by the to-many relation in Student:
6     @Backlink(to = "teachers")
7     lateinit var students: ToMany<Student>
8 }
9
10 @Entity
11 data class Student(
12     @Id var id: Long = 0
13 ) {
14     lateinit var teachers: ToMany<Teacher>
15 }
```

Updating Relations

The ToOne and ToMany classes assist you to persist the relation state. They keep track of changes and apply them to the database once you **put the entity owning the relation**. ObjectBox supports relation updates for new (not yet persisted; ID == 0) and existing (persisted before; ID != 0) entities.

For convenience, ToOne and ToMany will put related entities if they do **not yet exist** (ID == 0). If they already **exist** (their ID != 0, or you are using `@Id(assignable = true)`), only the relation will be created or destroyed. In that case, to put changes to the properties of related entities use their specific Box instead:

```
1 // update a related entity using its box
2 Order orderToUpdate = customer.orders.get(0);
3 orderToUpdate.text = "Revised description";
4 // DOES NOT WORK
5 // boxStore.boxFor(Customer.class).put(customer);
6 // WORKS
```




```
7 boxStore.boxFor(Order.class).put(orderToUpdate);
```

Updating ToOne


The ToOne class offers the following methods to update the relation:

- `setTarget(entity)` makes the given entity (new or existing) the new relation target; pass `null` to clear the relation
- `setTargetId(entityId)` sets the relation to the given ID of an existing target entity; pass `0` (zero) to clear the relation
- `setAndPutTarget(entity)` makes the given entity (new or existing) the new relation target and puts the enclosing entity and if needed the target entity.

```
1 order.customer.setTarget(customer); // or order.customer.setCustomerId(customerId);
2 orderBox.put(order);
```

 **Note:** if your entity was not put yet before calling `setAndPutTarget()` you need to attach its box first:

```
1 Order order = new Order(); // new entity
2 orderBox.attach(order); // need to attach box first
3 order.customer.setAndPutTarget(customer);
```

 **Note:** if you are using self-assigned IDs with `@Id(assignable = true)` a new target entity will not be stored when storing the parent. Read on for details:

If your target entity uses self-assigned IDs, you have to store it before updating the ToOne relation:


```
1 customer.id = 12; // self-assigned id
2 customerBox.put(customer); // need to put customer first
3 order.customer.setTarget(customer); // or order.customer.setCustomerId(customer.id);
4 orderBox.put(order);
```

This is because **ObjectBox** only puts related entities with an ID of 0. See the [documentation about IDs](#) for background information.

Updating ToMany

The ToMany class implements the `java.lang.List` interface while adding change tracking for entities. If you add entities to an ToMany object, those are scheduled to be put in the database. Similarly, if you remove entities from the ToMany object, those are also scheduled to be put. Note that removing entities from the List does not actually remove the entity from the database; just the relation is cleared. Do not forget to put the owning entity to apply changes tracked by ToMany objects to the database.

```
1 customer.orders.add(order1);
2 customer.orders.remove(order2);
3 customerBox.put(customer);
```

 **Note:** if you are using self-assigned IDs with `@Id(assignable = true)` the above will not work. Read on for details:

If your parent entity uses `@Id(assignable = true)` you need to attach its box before modifying its ToManys:

```
1 customer.id = 12; // self-assigned id
2 customerBox.attach(customer); // need to attach box first
3 customer.orders.add(order);
4 customerBox.put(customer);
```

If your related entity, like `order` above, is using self-assigned IDs you need to put the related entities yourself before adding them to a relation:

```
1 order.id = 42; // self-assigned id
2 orderBox.put(order); // need to put order first
3 customer.orders.add(order);
4 customerBox.put(customer); // put customer, add relation to order
```

In this case, when putting the parent entity only the relation is updated. This is because **ObjectBox only puts related entities with an ID of 0**. See the [documentation about IDs](#) for background information.

Example: Modelling Tree Relations

You can model a tree relation with a to-one and a to-many relation pointing to itself:

Java

```
1 @Entity
2 public class TreeNode {
3     @Id long id;
4
5     ToOne<TreeNode> parent;
6
7     @BackLink
8     ToMany<TreeNode> children;
9 }
```

Kotlin

```
1 @Entity
2 data class TreeNode(
3     @Id var id: Long = 0
4 ) {
5     lateinit var parent: ToOne<TreeNode>
6
7     @Backlink
8     lateinit var children: ToMany<TreeNode>
9 }
```

The generated entity lets you navigate its parent and children:

Java

```
1 TreeNode parent = entity.parent.getTarget();
2 List<TreeNode> children = entity.children;
```

Kotlin

```
1 val parent: TreeNode = entity.parent.target
2 val children: List<TreeNode> = entity.children
```

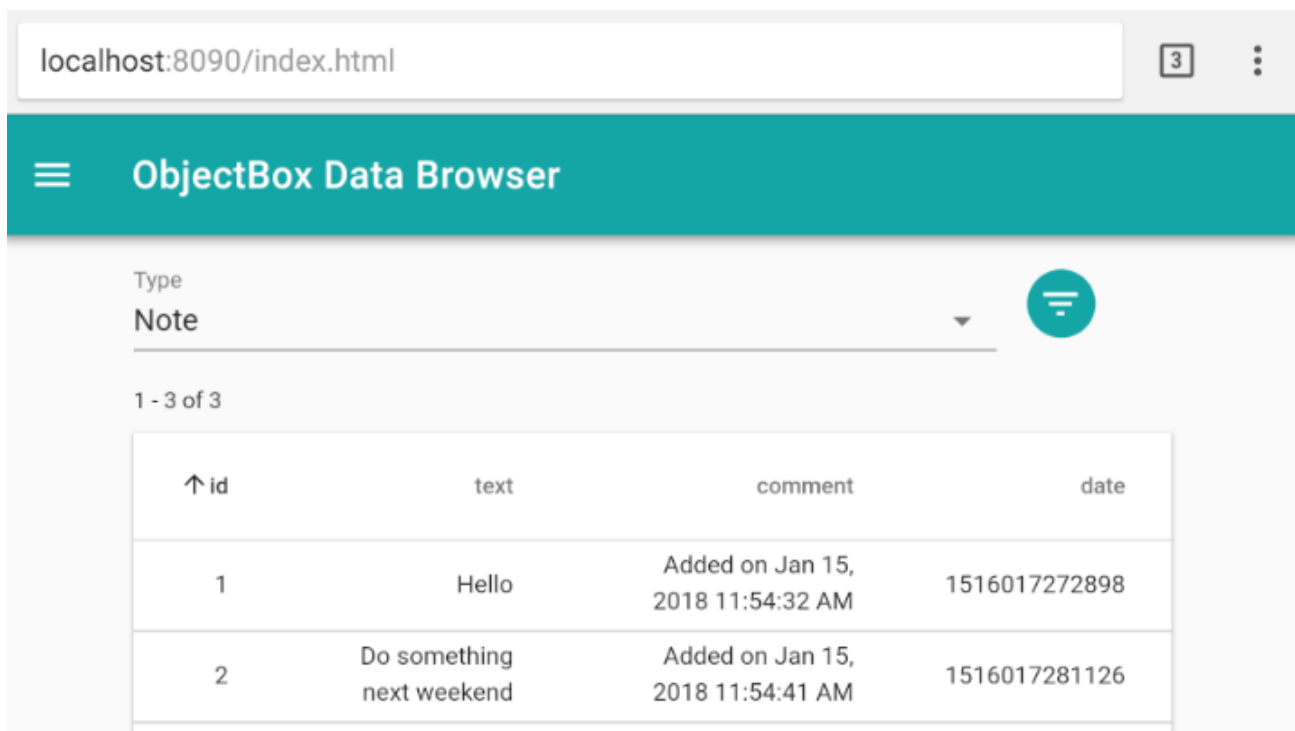
Data Browser

ObjectBox - Data Browser

The ObjectBox data browser (object browser) allows you to

- view the entities and schema of your database inside a regular web browser,
- and download entities in JSON format.

The object browser runs directly on your device or on your development machine. Behind the scenes this works by bundling a simple HTTP browser into ObjectBox when building your app. If triggered, it will then provide a basic web interface to the data and schema in your box store.



ObjectBox - Data Brower

Setup

We strongly recommend to use the object browser **only for debug builds**.

Add the `objectbox-android-objectbrowser` library to your debug configuration, and the `objectbox-android` library to your release configuration. Make sure to also **apply the “io.objectbox” plugin after the dependencies block:**

```
1 dependencies {
2     debugImplementation "io.objectbox:objectbox-android-objectbrowser:$obj
3     releaseImplementation "io.objectbox:objectbox-android:$objectboxVersio
4 }
5
6 // apply the plugin after the dependencies block
7 apply plugin: 'io.objectbox'
```

Otherwise the build will fail with a duplicate files error (like `Duplicate files copied in APK lib/armeabi-v7a/libobjectbox.so`) because the `ObjectBox` plugin will add the `objectbox-android` library again.

The `objectbox-android-objectbrowser` artifact adds required permissions to your `AndroidManifest.xml` (since version 2.2.0). The permissions added are:

```
1 <!-- Required to provide the web interface -->
2 <uses-permission android:name="android.permission.INTERNET" />
3 <!-- Required to run keep-alive service when targeting API 28 or higher -->
4 <uses-permission android:name="android.permission.FOREGROUND_SERVICE"/>
```

If you only use the browser library for debug builds as recommended above, they will not be added to your release build.

After creating your `BoxStore` , start the object browser using an `AndroidObjectBrowser` instance. Typically in the `onCreate()` method of your `Application` class:

```
1 boxStore = MyObjectBox.builder().androidContext(this).build();
2 if (BuildConfig.DEBUG) {
3     boolean started = new AndroidObjectBrowser(boxStore).start(this);
4     Log.i("ObjectBrowser", "Started: " + started);
5 }
```

Browse data on your device

When the app starts an object browser notification should appear. Tapping it will launch a service to keep the app alive and opens the data browser interface in the web browser on the device.

To stop the service keeping your app alive, tap the 'Stop' button in the notification.

Browse data on your dev machine

To open the browser website on your development machine check the Logcat output when launching the app. It will print the port and the ADB command needed to forward the port to your machine:

```
1 I/ObjectBrowser: ObjectBrowser started: http://localhost:8090/index.html
2 I/ObjectBrowser: Command to forward ObjectBrowser to connected host: adb f
```

If available, port 8090 is used by default. So in most cases just run this command on your dev machine:

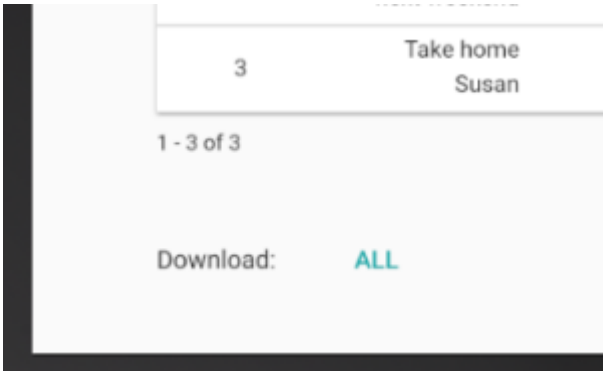
```
adb forward tcp:8090 tcp:8090
```

Once the port is forwarded you can open a browser and go to

```
http://localhost:8090/index.html .
```

Download entities

When viewing entities tap the download button at the very bottom. This will download entities formatted as JSON.



Download entities

Transactions

ObjectBox - Transactions

ObjectBox is a fully transactional database satisfying [ACID](#) properties. A transaction can group several operations into a single unit of work that either executes completely or not at all. If you are looking for a more detailed introduction to transactions in general, please consult other resources like Wikipedia on [database transactions](#). For ObjectBox transactions continue reading:

You may not notice it, but almost all interactions with ObjectBox involve transactions. For example, if you call `put` a write transaction is used. Also if you `get` an object or query for objects, a read transaction is used. All of this is done under the hood and transparent to you. It may be fine to completely ignore transactions altogether in your app without running into any problems. With more complex apps however, it's usually worth learning transaction basics to make your app more consistent and efficient.

Explicit transactions

We learned that all ObjectBox operations run in implicit transactions – unless an explicit transaction is in progress. In the latter case, multiple operations share the (explicit) transaction. In other words, with explicit transactions you control the transaction boundary. Doing so can greatly improve efficiency and consistency in your app.

The class [BoxStore](#) offers the following methods to perform explicit transactions:

- **runInTx:** Runs the given runnable inside a transaction.
- **runInReadTx:** Runs the given runnable inside a read(-only) transaction. Unlike write transactions, multiple read transactions can run at the same time.
- **runInTxAsync:** Runs the given Runnable as a transaction in a separate thread. Once the transaction completes the given callback is called (callback may be null).
- **callInTx:** Like runInTx(Runnable), but allows returning a value and throwing an exception.

The advantage of explicit transactions over the bulk put operations is that you can perform any number of operations and use objects of multiple boxes. In addition, you get a consistent (transactional) view on your data while the transaction is in progress.

Example for a write transaction:

```
1 boxStore.runInTx(() -> {
2     for(User user: allUsers) {
3         if(modify(user)) box.put(user);
4         else box.remove(user);
5     }
6 });
```

Transaction costs

Understanding transactions is essential to master database performance. If you just remember one sentence on this topic, it should be this one: a write transaction has its price.

Committing a transaction involves syncing data to the physical storage, which is a relatively expensive operation for databases. Only when the file system confirms that all data has been stored in a durable manner (not just memory cached), the transaction can be considered successful. This file sync required by a transaction may take a couple of milliseconds. Keep this in mind and try to group several operations (e.g. `put` calls) in one transaction.

Consider this example:

```
1 for(User user: allUsers) {
2     modify(user); // modifies properties of given user
3     box.put(user);
4 }
```

Do you see what's wrong with that code? There is an implicit transaction for each user which is very inefficient, especially for a high number of objects. It is much more efficient to use one of the put overloads to store all users at once:

```
1 for(User user: allUsers) {  
2     modify(user); // modifies properties of given user  
3 }  
4 box.put(allUsers);
```

Much better! If you have 1,000 users, the latter example uses a single transaction to store all users. The first code example uses 1,000 (!) implicit transactions, causing a massive slow down.

Read Transactions

In ObjectBox, read transactions are cheap. In contrast to write transactions, there is no commit and thus no expensive sync to the file system. Operations like `get` , `count` , and queries run inside an implicit read transaction if they are not called when already inside an explicit transaction (read or write). Note that it is illegal to `put` when inside a read transaction: an exception will be thrown.

While read transactions are much cheaper than write transactions, there is still some overhead to starting a read transaction. Thus, for a high number of reads (e.g. hundreds, in a loop), you can improve performance by grouping those reads in a single read transaction (see explicit transactions below).

Multiversion concurrency

ObjectBox gives developers [Multiversion concurrency control \(MVCC\)](#) semantics. This allows multiple concurrent readers (read transactions) which can execute immediately without blocking or waiting. This is guaranteed by storing multiple versions of (committed)

data. Even if a write transaction is in progress, a read transaction can read the last consistent state immediately. Write transactions are executed sequentially to ensure a consistent state. Thus, it is advised to keep write transactions short to avoid blocking other pending write transactions. For example, it is usually a bad idea to do networking or complex calculations while inside a write transaction. Instead, do any expensive operation and prepare objects before entering a write transaction.

Note that you do not have to worry about making write transactions sequential yourself. If multiple threads want to write at the same time (e.g. via `put` or `runInTx`), one of the threads will be selected to go first, while the other threads have to wait. It works just like a lock or `synchronized` in Java.

Locking inside a Write Transaction

Avoid locking (e.g. via `synchronized` or `java.util.concurrent.locks`) when inside a write transaction when possible. Because write transactions run exclusively, they effectively acquire a write lock internally. As with all locks, you need to pay close attention when multiple locks are involved. Always obtain locks in the same order to avoid deadlocks. If you acquire a lock “X” inside a transaction, you must ensure that your code does not start another write transaction while having the lock “X”.

Advanced

Advanced Setup

ObjectBox - Advanced Setup

First, [add and apply the ObjectBox plugin](#) to your project.

To then change the default behavior of the ObjectBox plugin and processor read on for advanced setup options.

Manually Add Libraries

If the ObjectBox plugin does not automatically add the required library and annotation processor to your dependencies, you can add them manually. In your app's `build.gradle` file, add the objectbox-android library and the objectbox-processor annotation processor.

For Android projects using Java:

```
1 dependencies {
2     // all below should be added automatically by the plugin
3     compile "io.objectbox:objectbox-android:$objectboxVersion"
4     annotationProcessor "io.objectbox:objectbox-processor:$objectboxVersion"
5 }
```

For Android projects using Kotlin:

```
1 dependencies {
2     // all below should be added automatically by the plugin
3     compile "io.objectbox:objectbox-android:$objectboxVersion"
4     kapt "io.objectbox:objectbox-processor:$objectboxVersion"
5     // some useful Kotlin extension functions
6     compile "io.objectbox:objectbox-kotlin:$objectboxVersion"
7 }
```

Change the Model File Path

By default the ObjectBox model file is stored in

`module-name/objectbox-models/default.json` . You can change the file path and name by passing the `objectbox.modelPath` argument to the ObjectBox annotation processor.

In your app's `build.gradle` file, add the necessary processor option.


For Android projects using Java:

```
1  android {  
2      defaultConfig {  
3          javaCompileOptions {  
4              annotationProcessorOptions {  
5                  arguments = [ "objectbox.modelPath" : "$projectDir/schemas/objectbox.json"  
6              }  
7          }  
8      }  
9  }
```

For Android projects using Kotlin:

```
1  kapt {  
2      arguments {  
3          arg("objectbox.modelPath", "$projectDir/schemas/objectbox.json")  
4      }  
5  }
```

Change the MyObjectBox package

 Since 1.5.0

By default the `MyObjectBox` class is generated in the same or a parent package of your entity classes. You can define a specific package by passing the `objectbox.myObjectBoxPackage` argument to the `ObjectBox` annotation processor.

In your app's `build.gradle` file, add the necessary processor option.

For Android projects using Java:

```
1  android {
2      defaultConfig {
3          javaCompileOptions {
4              annotationProcessorOptions {
5                  arguments = [ "objectbox.myObjectBoxPackage" : "com.examp
6              }
7          }
8      }
9  }
```

For Android projects using Kotlin:

```
1  kapt {
2      arguments {
3          arg("objectbox.myObjectBoxPackage", "com.example.custom")
4      }
5  }
```

Enable Debug Mode

You can enable debug output for the annotation processor if you encounter issues while setting up your project and entity classes.

In your app's `build.gradle` file, add the necessary options and then run Gradle with the `-info` option to see the debug output.

For Android projects using Java:

```
1  android {  
2      defaultConfig {  
3          javaCompileOptions {  
4              annotationProcessorOptions {  
5                  arguments = [ 'objectbox.debug' : 'true' ]  
6              }  
7          }  
8      }  
9  }
```

For Android projects using Kotlin:

```
1  kapt {  
2      arguments {  
3          arg("objectbox.debug", true)  
4      }  
5  }
```

Enable DaoCompat mode

ObjectBox can help you migrate from greenDAO by generating classes with a greenDAO-like API.

See the [DaoCompat documentation](#) on how to enable and use this feature.

Next steps

Tips when using Kotlin

→ Kotlin Support

/kotlin-support

Object IDs

ObjectBox - Object IDs

Objects must have an ID property of type `long` . You are free to use the wrapper type `java.lang.Long` , but we advise against it in most cases. `long` IDs are enforced to make ObjectBox very efficient internally.

If your application requires other ID types (such as a string UID given by a server), you can model them as standard properties and use [queries](#) to look up entities by your application specific ID.

Object ID: new vs. persisted entities

When you create new entity objects (on the language level), they are not persisted yet and their ID is (zero). Once an entity is put (persisted), ObjectBox will assign an ID to the entity. You can access the ID property right after the call to `put()` .

Those are also applied the other way round: ObjectBox uses the **ID as a state indication** whether an entity is new (zero) or already persisted (non-zero). This is used internally, e.g. for relations which heavily rely on IDs.

Special Object IDs

Object IDs may be any `long` value, with two exceptions:

- **0 (zero):** Objects with an ID of zero (and `null` if the ID is of type `Long`) are considered new (not persisted before). Putting such an object will always insert a new object and assign an unused ID to it.
 - **0xFFFFFFFFFFFFFFFF (-1 in Java):** This value is reserved for internal use by ObjectBox and may not be used by the app.
-

Object ID assignment (default)

By default object IDs are assigned by ObjectBox. For each new object, ObjectBox will assign an unused ID that is above the current highest ID value used in a box. For example, if there are two objects with ID 1 and ID 100 in a box the next object that is put will be assigned ID 101.

By default, only ObjectBox may assign IDs. If you try to put an object with an ID greater than the currently highest ID, ObjectBox will throw an error.

Self-assigned Object IDs

If you **need to assign IDs by yourself** you can change the ID annotation to:

```
1 @Id(assignable = true)
2 long id;
```

This will allow putting an entity with any valid ID. You can still set the ID to zero to let ObjectBox auto assign a new ID.



Warning: self-assigned IDs break automatic state detection (new vs. persisted entity based on the ID). Therefore, you should **put entities with self-assigned IDs immediately and may have to attach the box manually**, especially when working with relations.

For details see the documentation about [updating relations](#).

String ID alias (future work)

Check [this issue](#) on Github for status.

Custom Types

ObjectBox - Supported Types

With ObjectBox you can store pretty much any type (class), given that it can be converted to any of the built-in types.

ObjectBox can store the following built-in types without a converter:

Java

```
1  boolean, Boolean
2  int, Integer
3  short, Short
4  long, Long
5  float, Float
6  double, Double
7  byte, Byte
8  char, Character
9  byte[]
10 String
11 Date // Time with millisecond precision.
12
13 // As of 3.0.0-alpha2 the following work out of the box:
14 String[]
15 @Type(DateNano) long, Long // Time with nanosecond precision.
```

```
1 Boolean, Boolean?
2 Int, Int?
3 Short, Short?
4 Long, Long?
5 Float, Float?
6 Double, Double?
7 Byte, Byte?
8 Char, Char?
9 ByteArray
10 String, String?
11 Date, Date? // Time with millisecond precision.
12
13 // As of 3.0.0-alpha2 the following work out of the box:
14 Array<String>
15 @Type(DateNano) Long, Long? // Time with nanosecond precision.
```

To store any other type, configure a converter like shown below.

Convert annotation and property converter

To add support for a custom type, you can map properties to one of the built-in types using a **@Convert** annotation. You also need to provide a `PropertyConverter` implementation.

For example, you could define a color in your entity using a custom `Color` class and map it to an `Integer`. Or you can map the popular `org.joda.time.DateTime` from Joda Time to a `Long`.

Here is an example mapping an `enum` to an `Integer`:

```
1 @Entity
2 public class User {
3     @Id
4     private Long id;
```

```

5
6     @Convert(converter = RoleConverter.class, dbType = Integer.class)
7     private Role role;
8
9     public enum Role {
10         DEFAULT(0), AUTHOR(1), ADMIN(2);
11
12         final int id;
13
14         Role(int id) {
15             this.id = id;
16         }
17     }
18
19     public static class RoleConverter implements PropertyConverter<Role, Integer> {
20         @Override
21         public Role convertToEntityProperty(Integer databaseValue) {
22             if (databaseValue == null) {
23                 return null;
24             }
25             for (Role role : Role.values()) {
26                 if (role.id == databaseValue) {
27                     return role;
28                 }
29             }
30             return Role.DEFAULT;
31         }
32
33         @Override
34         public Integer convertToDatabaseValue(Role entityProperty) {
35             return entityProperty == null ? null : entityProperty.id;
36         }
37     }
38 }

```

Things to look out for


If you define your custom type or converter **inside your entity class**, they have to be **static**.

Don't forget to **handle null values** correctly – usually you should return null if the input is null.

Database types in the sense of the converter are the primitive Java types offered by ObjectBox, as mentioned in the beginning. It is recommended to **use a primitive type** that

is **easily convertible** (int, long, byte array, String, ...).

You must **not interact with the database** (such as using `Box` or `BoxStore`) inside the converter. The converter methods are called within a transaction, so for example getting or putting entities to a box will fail.

 Note: For optimal performance, **ObjectBox will use a single converter instance** for all conversions. Make sure the converter does not have any other constructor besides the parameter-less default constructor. Also, make it thread safe, because it might be called concurrently on multiple entities.

List/Array types

You can use a converter with List types. For example, you could convert a List of Strings to a JSON array resulting in a single string for the database. At the moment it is not possible to use an array with converters (you can track this [feature request](#)).

How to convert Enums correctly

Enums are popular with data objects like entities. When persisting enums, there are a couple of best practices:

- **Do not persist the enum's ordinal or name:** Both are unstable, and can easily change the next time you edit your enum definitions.
 - **Use stable ids:** Define a custom property (integer or string) in your enum that is guaranteed to be stable. Use this for your persistence mapping.
 - **Prepare for the unknown:** Define an UNKNOWN enum value. It can serve to handle null or unknown values. This will allow you to handle cases like an old enum value getting removed without crashing your app.
-

Custom types in queries

`QueryBuilder` is unaware of custom types. You have to **use the primitive DB type for queries**.

So for the Role example above you would get users with the role of admin with the query condition `.equal(UserProperties.Role, 2)`.

Entity Inheritance

Objectbox - Entity inheritance

ObjectBox allows entity inheritance to share persisted properties in super classes. While ObjectBox always allowed entities to extend a non-entity base class, ObjectBox 1.4+ also allows extending entities. In addition to the `@Entity` annotation, we introduced a `@BaseEntity` annotation for base classes, which can be used instead of `@Entity`.

There three types of base classes, which are defined via annotations:

- **No annotation:** The base class and its properties are not considered for persistence.
- **@BaseEntity:** Properties are considered for persistence in sub classes, but the base class itself cannot be persisted.
- **@Entity:** Properties are considered for persistence in sub classes, and the base class itself is a normally persisted entity.

For example:

```
1 // base class:
2 @BaseEntity
3 public abstract class Base {
4
5     @Id long id;
6     String baseString;
7
8     public Base() {
9     }
10
11     public Base(long id, String baseString) {
12         this.id = id;
13         this.baseString = baseString;
14     }
15 }
16
17 // sub class:
18 @Entity
19 public class Sub extends Base {
20
21     String subString;
22 }
```

```

23     public Sub() {
24     }
25
26     public Sub(long id, String baseString, String subString) {
27         super(id, baseString);
28         this.subString = subString;
29     }
30 }

```

The model for Sub, Sub_, will now include all properties: `id` , `baseString` and `subString` .

It is also possible to inherit properties from another entity:

```

1  // entities inherit properties from entities
2  @Entity
3  public class SubSub extends Sub {
4
5      String subSubString;
6
7      public SubSub() {
8      }
9
10     public SubSub(long id, String baseString, String subString, String sub
11         super(id, baseString, subString);
12         this.subSubString = subSubString;
13     }
14 }

```

Notes on usage

- It is possible to have classes in the inheritance chain that are not annotated with `@BaseEntity`. Their properties will be ignored and will not become part of the entity model.

- It is not generally recommend to have a base entity class consisting of an ID property only. E.g. Java imposes an additional overhead to construct objects with a sub class.
 - Depending on your use case using interfaces may be more straightforward.
-

Restrictions

- Superclasses annotated with **@BaseEntity** **can not be part of a library**.
- There are **no polymorphic queries** (e.g. you cannot query for a base class and expect results from sub classes).
- Currently any superclass, whether it is an @Entity or @BaseEntity, **can not have any relations** (like a ToOne or ToMany property).

```
1 // THIS WILL NOT WORK
2 @BaseEntity
3 public abstract class Base {
4     @Id long id;
5     ToOne<OtherEntity> other;
6     ToMany<OtherEntity> others;
7 }
```

Data Model Updates

ObjectBox - Data Model Updates

ObjectBox manages its data model (schema) mostly automatically. The data model is defined by the entity classes you define. When you **add or remove** entities or properties of your entities, **ObjectBox takes care** of those changes without any further action from you.

For other changes like **renaming or changing the type**, ObjectBox needs **extra information** to make things unambiguous. This works using unique IDs (UIDs) and an `@uid` annotation, as we will see below.

UIDs

ObjectBox keeps track of entities and properties by assigning them unique IDs (UIDs). All those UIDs are stored in a file “objectbox-models/default.json”, which you should add to your version control system (e.g. git). If you are interested, we have [in-depth documentation on UIDs and concepts](#). But let's continue with how to rename entities or properties.

In short: To make UID-related changes, put an `@uid` annotation on the entity or property and build the project to get further instructions. Repeat for each entity or property to change.

Renaming Entities and Properties

So why do we need that UID annotation? If you simply rename an entity class, ObjectBox only sees that the old entity is gone and a new entity is available. This can be interpreted in two ways:

- The old entity is removed and a new entity should be added, the old data is discarded. This is the **default behavior** of ObjectBox.

- The entity was renamed, the old data should be re-used.

So to tell ObjectBox to do a rename instead of discarding your old entity and data, you need to make sure it knows that this is the same entity and not a new one. You do that by attaching the internal UID to the entity.

The same is true for properties.

Now let's walk through how to do that. The process works the same if you want to rename a property:

How-to and Example

Step 1: Add an empty `@Uid` annotation to the entity/property you want to rename:

```
1 @Entity
2 @Uid
3 public class MyName { ... }
```

Step 2: Build the project. The build will fail with an error message that gives you the current UID of the entity/property:

```
1 error: [ObjectBox] UID operations for entity "MyName":
2   [Rename] apply the current UID using @Uid(6645479796472661392L) -
3   [Change/reset] apply a new UID using @Uid(4385203238808477712L)
```

Step 3: Apply the UID from the [Rename] section of the error message to your entity/property:


```
1 @Entity
2 @Uid(6645479796472661392L)
3 public class MyName { ... }
```

Step 4: The last thing to do is the actual rename on the language level (Java, Kotlin, etc.):


```
1 @Entity
2 @Uid(6645479796472661392L)
3 public class MyNewName { ... }
```

Step 5: Build the project again, it should now succeed. You can now use your renamed entity/property as expected and all existing data will still be there.

Repeat the steps above to rename another entity or property.

 Note: Instead of the above you can also find the UID of the entity/property in the ObjectBox default.json file yourself and add it together with the @Uid annotation before renaming your entity/property. This can be **faster when renaming multiple properties**.

Changing Property Types

 ObjectBox does not support migrating existing property data to a new type. You will have to take care of this yourself, e.g. by keeping the old property and adding some migration logic.

There are two solutions to changing the type of a property:

- Add a new property with a different name (this only works if the property has no @Uid annotation already):


```
1 // old:
2 String year;
3 // new:
4 int yearInt;
```

- Set a new UID for the property so ObjectBox treats it as a new property. Let's walk through how to do that:

How-to and Example

Step 1: Add the `@Uid` annotation to the property where you want to change the type:

```
1 @Uid
2 String year;
```

Step 2: Build the project. The build will fail with an error message that gives you a newly created UID value:

```
1 error: [ObjectBox] UID operations for property "MyEntity.year":
2   [Rename] apply the current UID using @Uid(6707341922395832766L) -
3   [Change/reset] apply a new UID using @Uid(9204131405652381067L)
```

Step 3: Apply the UID from the [Change/reset] section to your property:

```
1 @Uid(9204131405652381067L)
2 int year;
```

Step 4: Build the project again, it should now succeed. You can now use the property in your entity as if it were a new one.

Repeat the steps above to change the type of another property.

Meta Model, IDs, and UUIDs

Unlike relational databases like SQLite, ObjectBox does not require you to create a database schema. That does not mean ObjectBox is schema-less. For efficiency reasons, ObjectBox manages a **meta model** of the data stored. This meta model is actually ObjectBox's equivalent of a schema. It includes known object types including all properties, indexes, etc. A key difference to relational schemas is that ObjectBox tries to manage its meta model automatically. In some cases it needs your help. That's why we will look at some details.

IDs

In the ObjectBox meta model, everything has an ID and a UUID. IDs are used internally in ObjectBox to reference entities, properties, and indexes. For example, you have an entity "User" with the properties "id" and "name". In the meta model the entity (type) could have the ID 42, and the properties the IDs 1 and 2. Property IDs must only be unique within their entity.



Note: do not confuse object IDs with meta model IDs: object IDs are the values of the @Id property (see Object IDs in [basics](#)). In contrast, all objects are instances of the entity type associated with a single meta model ID.

ObjectBox assigns meta model IDs sequentially (1, 2, 3, 4, ...) and keeps track of the last used ID to prevent ID collisions.

UUIDs

As a rule of thumb, for each meta model ID there's a corresponding UUID. They complement IDs and are often used in combination (e.g. in the JSON file). While IDs are

assigned sequentially, UUIDs are a random long value. The job of UUIDs is detecting and resolving concurrent modifications of the meta model.

A UUID is unique across entities, properties, indexes, etc. Thus unlike IDs, a UUID already used for an entity may not be used for a property. As a precaution to avoid side effects, ObjectBox keeps track of “retired” UUIDs to ensure previously used but now abandoned UUIDs are not used for new artifacts.

JSON for consistent IDs

ObjectBox stores a part of its meta model in a JSON file. This file should be available to every developer and thus checked into a source version control system (e.g. git). The main purpose of this JSON file is to ensure consistent IDs and UUIDs in the meta model across devices.

This JSON file is stored in the file `objectbox-models/default.json`. For example, look at the file from the [ObjectBox example project](#):

```
1  {
2    "_note1": "KEEP THIS FILE! Check it into a version control system (VCS)",
3    "_note2": "ObjectBox manages crucial IDs for your object model. See docs",
4    "_note3": "If you have VCS merge conflicts, you must resolve them accord",
5    "entities": [
6      {
7        "id": "1:6645479796472661392",
8        "lastPropertyId": "4:1260602348787983453",
9        "name": "Note",
10       "properties": [
11         {
12           "id": "1:9211738071025439652",
13           "name": "id"
14         },
15         {
16           "id": "2:8804670454579230281",
17           "name": "text"
18         },
19         {
20           "id": "3:6707341922395832766",
21           "name": "comment"
22         },
23       ]
24     }
25   ]
26 }
```

```
23     {
24         "id": "4:1260602348787983453",
25         "name": "date"
26     }
27 ]
28 }
29 ],
30 "lastEntityId": "1:6645479796472661392",
31 "lastIndexId": "0:0",
32 "lastSequenceId": "0:0",
33 "modelVersion": 2,
34 "retiredEntityUids": [],
35 "retiredIndexUids": [],
36 "retiredPropertyUids": [],
37 "version": 1
38 }
```

As you can see, the “id” attributes combine the ID and UID using a colon. This protects against faulty merges. When applying the meta model to the database, ObjectBox will check for consistent IDs and UIDs.

Meta Model Synchronization

At build time, ObjectBox gathers meta model information from the entities (@Entity classes) and the JSON file. The complete meta model information is written into the generated class `MyObjectBox`.

Then, at runtime, the meta model assembled in `MyObjectBox` is synchronized with the meta model inside the ObjectBox database (file). UIDs are the primary keys to synchronize the meta model with the database. The synchronization involves a couple of consistency checks that may fail when you try to apply illegal meta data.

Stable Renames using UIDs

At some point you may want to rename an entity class or just a property. Without further information, ObjectBox will remove the old entity/property and add a new one with the new

name. This is actually a valid scenario by itself: removing one property and adding another. To tell ObjectBox it should do a rename instead, you need to supply the property's previous UID.

Add an `@uid` annotation without any value to the entity or property you want to rename and trigger a project build. The build will fail with a message containing the UID you need to apply to the `@uid` annotation.

Also check out this [how-to guide](#) for hands-on information on renaming and resetting.

Resolving Meta Model Conflicts

In the section on UIDs, we already hinted at the possibility of meta model conflicts. This is caused by developers changing the meta model concurrently, typically by adding entities or properties. The knowledge acquired in the previous paragraphs helps us to resolve the conflicts.

The Nuke Option

During initial development, it may be an option to just delete the meta model and all databases. This will cause a fresh start for the meta model, e.g. all UIDs will be regenerated. Follow these steps:

- Delete the JSON file (objectbox-models/default.json)
- Build the project to generate a new JSON file from scratch
- Commit the recreated JSON file to your VCS (e.g. git)
- Delete all previously created ObjectBox databases (e.g. for Android, delete the app's data or uninstall the app)

While this is a simple approach, it has its obvious disadvantages and is completely useless once an app has been published.

Pitfall: If the database file does not seem to get deleted, check for the presence of `android:allowBackup="true"` in your Android manifest. Backups may resurrect old files and thus should be off for this approach.

Manual conflict resolution

Usually, it is preferred to edit the JSON file to resolve conflicts and fix the meta model. This involves the following steps:

- **Ensure IDs are unique:** in the JSON file the id attribute has values in the format “ID:UID”. If you have duplicate IDs after a VCS merge, you should assign a new ID (keep the UID part!) to one of the two. Typically, the new ID would be “last used ID + 1”.
- **Update last ID values:** for entities, update the attribute “lastEntityId”; for properties, update the attribute “lastPropertyId” of the enclosing entity
- **Check for other ID references:** do a text search for the UID and check if the ID part is correct for all UID occurrences

To illustrate this with an example, let's assume the last assigned entity ID was 41. Thus the next entity ID will be 42. Now, the developers Alice and Bob add a new entity without knowing of each other. Alice adds a new entity “Ant” which is assigned the entity ID 42. At the same time, Bob adds the entity “Bear” which is also assigned the ID 42. After both developers committed their code, the ID 42 does not unique identify an entity type (“Ant” or “Bear”?). Furthermore, in Alice’s ObjectBox the entity ID 42 is already wired to “Ant” while Bob’s ObjectBox maps 42 to “Bear”. UIDs make this situation resolvable. Let’s say the UID is 12345 for “Ant” and 9876 for “Bear”. Now, when Bob pulls Alice’s changes, he is able to resolve the conflict. He manually assigns the entity ID 43 to “Bear” and updates the lastEntityId attribute accordingly to “43:9876” (ID:UID). After Bob commits his changes, both developers are able to continue with their ObjectBox files.

FAQ

Does ObjectBox support Kotlin? RxJava?

ObjectBox comes with full [Kotlin support](#) including data classes. And yes, it supports [RxJava and reactive queries without RxJava](#).

Does ObjectBox support object relations?

Yes. ObjectBox comes with strong relation support and offers features like “eager loading” for optimal performance.

Does ObjectBox support multi-module projects? Can entities be spread across modules?

The ObjectBox plugin only looks for entities in the current module, it does not search library modules. However, you can have a separate database (`MyObjectBox` file) for each module. Just make sure to pass different database names when building your BoxStore.

Is ObjectBox a “zero copy” database? Are properties fetched lazily?

It depends. Internally and in [the C API](#), ObjectBox does zero-copy reads. Java objects require a single copy only. However, copying data is only a minor factor in overall performance. In ObjectBox, objects are POJOs (plain objects), and all properties will be properly initialized. Thus, there is no run time penalty for accessing properties and values do not change in unexpected ways when the database updates.

Are there any threading constrictions?

No. The objects you get from ObjectBox are POJOs (plain objects). You are safe to pass them around in threads.

On which platforms does ObjectBox run?

ObjectBox supports **Android 4.0.3** (API level or minimum **SDK 15**) and above and works on most devices (armeabi-v7a, arm64-v8a, x86 and x86_64). It works with Java and Kotlin projects.

ObjectBox also runs on Linux (64 bit), Windows (64 bit), macOS and iOS with support for [Kotlin](#), [Java](#), [Go](#), [C](#), [Swift](#) and [Python](#).

Can I use ObjectBox on the desktop/server?

Yes, you can ObjectBox on the desktop/server side. Contact us for details if you are interested in running ObjectBox in client/server mode or containerized!

Can I use ObjectBox on smart IoT devices?


Generally speaking: Yes. You can run the ObjectBox database on any IoT device that runs Linux. We also offer Go and C APIs.

How do I rename object properties or classes?

If you only do a rename on the language level, ObjectBox will by default remove the old and add a new entity/property. [To do a rename, you must specify the UID](#).

How much does ObjectBox add to my APK size?

The Google Play **download size** increases by around 2.0 MB (checked for ObjectBox 2.5.0) as a native library for each [supported architecture](#) is packaged. If you [build multiple APKs](#) split by ABI or use [Android App Bundle](#) it only increases around 0.5 MB.

 Tip: Open your APK or AAB in Android Studio and have a look at the lib folder to see the raw file size and download size added.

The **raw file (APK or AAB) size** increases around 5.3 MB. This is because ObjectBox adds `extractNativeLibs="false"` to your `AndroidManifest.xml` [as recommended by Google](#). This turns off compression. However, this allows Google Play to optimally compress the APK before downloading it to each device (see download size above) and reduces the size of your app updates (on Android 6.0 or newer). [Read this Android developers post for details](#). It also avoids issues that might occur when extracting the libraries.


If you rather have a smaller APK instead of smaller app downloads and updates (e.g. when distributing in other stores) you can override the flag in your `AndroidManifest.xml` :

```
1 <application
2     ...
3     // not recommended, increases app update size
4     android:extractNativeLibs="true"
5     tools:replace="android:extractNativeLibs"
6     ...
7 </applicaton>
```

More importantly, ObjectBox adds little to the APK method count since it's mostly written in native code.

Can I ship my app with a pre-built database?

Yes. ObjectBox stores all data in a single database file. Thus, you just need to prepare a database file and copy it to the correct location on the first start of your app (before you touch ObjectBox's API).


 There is an **experimental** `initialDbFile()` method when building `BoxStore`. Let us know if this is useful!

The database file is called `data.mdb` and is typically located in a subdirectory called `objectbox` (or any name you passed to `BoxStoreBuilder`). On Android, the DB file is located inside the app's files directory inside `objectbox/objectbox/`. Or `objectbox/<yourname>` if you assigned the custom name `<yourname>` using `BoxStoreBuilder`.

How to reclaim disk space used by ObjectBox?

To reclaim disk space, `close()` the `BoxStore` and delete the database files using `BoxStore.deleteAllFiles(objectBoxDirectory)`. To avoid having to close `BoxStore` delete files before building it, e.g. during app start-up.

```
1 // If BoxStore is in use, close it first.
2 store.close();
3
4 BoxStore.deleteAllFiles(new File(BoxStoreBuilder.DEFAULT_NAME));
5
6 // TODO Build a new BoxStore instance.
```

 `BoxStore.removeAllObjects()` does not reclaim disk space. It keeps the allocated disk space so it returns fast and to avoid the performance hit of having to allocate the same disk space when data is put again.

Answers to other questions

→ Troubleshooting

/troubleshooting

Questions not related to Java, Kotlin or Android are answered in the [general ObjectBox FAQ](#).

If you believe to have found a bug or missing feature, please create an issue.

<https://github.com/objectbox/objectbox-java/issues>

If you have a usage question regarding ObjectBox, please post on Stack Overflow.

<https://stackoverflow.com/questions/tagged/objectbox>

Troubleshooting

Unresolved reference: MyObjectBox (class not found, etc.)

Make sure you do not have any other errors in your build. There have been problems reported in relation to “android-apt”, so remove it from your build if possible. For Kotlin, explicitly apply the kapt plugin (`apply plugin: 'kotlin-kapt'`) before the ObjectBox plugin.

Merge conflict or DbException after concurrent data model modifications

If your team makes concurrent modifications to the data model (e.g. adding/removing entities or properties) it may clash with your changes. Read the [meta model docs](#) on how to resolve the conflicts.

DbException after switching git branch (no concurrent data model modifications)

You might get an exception like

`DbException DB's last entity ID 4 is higher than 3 from model` after switching back from a branch and running your app. This is expected if that branch makes incompatible changes to entities. You need to uninstall or clear all data of your app before running it again.

Here is why: Let's say you have created a new entity on a new branch and ran your app on a device. This will upgrade the ObjectBox database on that device to support the new entity (ObjectBox keeps track of entity types internally by storing some meta info inside the database). Now, when you return to your previous branch and try to run the app ObjectBox

will fail with an error message similar to the one above. This is because the entity types passed by your code no longer match those stored inside the database. In other words, the database can not be downgraded to the previous version. Thus, it is advised to clear the database before running your app after switching branches. Check the [meta model docs](#) for more details.

Couldn't find "libobjectbox.so"

This can have various reasons. In general **check your ABI filter setup or add one** in your Gradle build file.

If your app explicitly **ships code for "armeabi"**: For Android, ObjectBox comes with binaries for "armeabi-v7a" and "arm64-v8a" ABIs. We consider "armeabi" to be outdated and thus do not support it. Check if you have a Gradle config like `abiFilters "armeabi"`, which is causing the problem (e.g. remove it or change it to "armeabi-v7a").

If your app uses **split APKs or App Bundle**: some users might have sideloaded your APK that includes the library for a platform that is incompatible with the one of their device. See [App Bundle, split APKs and Multidex](#) for workarounds.

Version conflict with 'com.google.code.findbugs:jsr305'

If you are doing Android instrumentation (especially with Espresso), you may get a warning like this:

```
Error:Conflict with dependency 'com.google.code.findbugs:jsr305' in project ':app'. Resolved versions for app (3.0.2) and test app (2.0.1) differ. See http://g.co/androidstudio/app-test-app-conflict for details.
```

You can easily resolve the version conflict by adding this Gradle dependency:

```
androidTestCompile 'com.google.code.findbugs:jsr305:3.0.2'
```

[Background info.](#)

Incompatible property type

Check the [data model migration guide](#) if you get an exception like:

```
io.objectbox.exception.DbException: Property [...] is not compatible to its  
previous definition. Check its type.
```

or

```
Cannot change the following flags for Property
```

Help with other issues

→ [FAQ](#)

[/faq](#)

If you believe to have found a bug or missing feature, please create an issue.

<https://github.com/objectbox/objectbox-java/issues>

If you have a usage question regarding ObjectBox, please post on Stack Overflow.

<https://stackoverflow.com/questions/tagged/objectbox>

Release History

V2.0 and later: check the [changelog on the homepage](#)

V1.5.0 – 2018/04/17

New Features/Improvements

- Full support for [Android local tests](#): use full ObjectBox features in local tests
- New count method optimized for a given maximum count
- Gradle option to [define the package for MyObjectBox](#) explicitly
- Query condition startsWith now uses index if available for better performance

Fixes

- Fixed some static methods in BoxStore to ensure that the native lib is loaded
- Internal optimizations for 64 bit devices
- Some fixes for entities in the default package
- Entity can be named `Property` , no longer conflicts with ObjectBox Property class
- Property queries for strings crashed on some Android devices if there were more than 512 results
- Object Browser uses less threads
- Object Browser now displays negative int/long values correctly
- Changes to relations object in constructors were overwritten when constructors delegated to other constructors

V1.4.4 – 2018/03/08

New Features/Improvements

- Supply an initial database file using BoxStoreBuilder
- Gradle plugin detects [plain Java projects](#) and configures dependencies
- Improved Box.removeAll() performance for entities that have indexes or relations

Fixes

- Fixed converting from arrays in entities
- Fixed @NameInDb
- Fixed Gradle “androidTestCompile is obsolete” warning

V1.4.3 – 2018/03/01

New Features

- macOS support: with Linux, Windows, and macOS, ObjectBox now supports all major desktop/server platforms. Use it for local unit tests or standalone Java applications.

Fixes

- Fixed BoxStore.close being stuck in rare scenarios
- Fixed an issue with char properties in entities

V1.4.2 – 2018/02/25

Note: This release requires the [Android Gradle Plugin 3.0.0](#) or higher.

Improvements

- JCenter: we’ve moved the ObjectBox artifacts to the JCenter repository. This simplifies set up and improves accessibility (e.g. JCenter is not blocked from China).
- Instant App support (only with Android Gradle Plugin 3.0.0 or higher)

V1.4.1 – 2018/01/23

Improvements

- Added DbExceptionListener as a central place to listen for DB related exceptions
- Minor improvements for ToMany and generated Cursor classes

Fixes

- ToMany: fixed handling of duplicate entries (e.g. fixes swap and reverse operations)
- ToMany: fixed removal of non-persisted element for standalone relations

V1.4.0 – 2018/01/11

New Features

- Property queries that return individual properties only (including distinct values, unique, null values, primitive result arrays or scalars)
- Entity inheritance (non-polymorphic)
- 50% size reduction of native libraries

V1.3.4 – 2017/12/07

Improvements

- ToOne now implements equals() and hashCode() based on the targetId property
- Android ABI x86_64 was added to the aar

Fixes

- ID verification does not complain about “resurrected” objects that were loaded, removed, and put again
- Fixed setting Query parameters for Date type
- Fixes for ObjectBox browser

V1.3.3 (1.3.x) – 2017/12/04

Please update to the latest version. We made important changes and fixes under the hood to make ObjectBox perform better, generally, and especially in concurrent scenarios. In addition, 1.3.x comes with several improvements for developers.

Improvements

- [Flag for query parameter logging](#)
- Object browser lets you download all entities as JSON
- Object browser efficiency improvements: introduced streamed processing to reduce memory consumption and increase performance for large data sets
- Improved transaction logging, e.g. numbered transactions and waiting times for write transactions

- Closing the store (e.g. for tests, an app should just leave it open) will wait for any ongoing write transaction to finish
- Two additional [overloads for static BoxStore.deleteAllFiles\(\)](#)
- Added automatic retries for read transactions; also configurable for queries

Fixes

- Fixes for concurrent setups (multi threaded, in live apps with up to 100 threads); internally we improved our testing automation and CI infrastructure significantly
- Fix for sumDouble throwing an exception
- Fixed ProGuard rule for ToOne

V1.2.1 – 2017/11/10

Improvements

- Improved debug logging for transactions and queries: enable this using BoxStoreBuilder.debugFlags(...) with values from the [DebugFlags](#) class
- Improved package selection for MyObjectBox if you use entities in multiple packages (please check if you need to adjust your imports after the update)
- ObjectBox Browser's UI is more compact and thus better usable on mobile devices

Fixes

- Fix for ObjectBoxLiveData firing twice

V1.2.0 – 2017/10/31

Compatibility note: We removed some Box.find methods, which were all tagged as @Temporary. Only the Property based ones remain (for now, also @Temporary).

New Features

- [ObjectBoxLiveData](#): Implements LiveData from Android Architecture Components
- Object ID based methods for [ToMany](#): getById, indexOfId, removeById
- More robust Android app directory detection that works around Android bugs

- Using the new official FlatBuffers Maven dependency (FlatBuffer is not anymore embedded in the artifact)
- UI improvements for ObjectBox browser
- Other minor improvements

Fixes

- Fixed query order by float and double
- Fixed an missing import if to-many relations referenced a entity in another package
- Other minor fixes

V1.1.0 – 2017/10/03

New Features

- Object Browser to view DB contents (Android)
- Plain Java support to run ObjectBox on Windows and Linux
- Added ToMany.hasA()/hasAll() to simplify query filters in Java
- Sort query result via Comparator
- Improved error messages on build errors
- Internal clean up, dropping legacy plugin

Fixes

- Annotation processor detects boolean getters starting with “is”
- Fixed a NPE with eager and findFirst() when there is no result

V1.0.1 – 2017/09/10

First bug fix release for [ObjectBox 1.0](#).

New Features

- ToMany allows setting a Comparator to order the List (experimental)

Fixes

- Fix UID assignment process: use @Uid without value to see options (pin UID, reset/change)
- Fix relation code generation for entities in different packages
- Fix Kotlin extension functions in transformed (library) project
- Fix ToOne access if field is inaccessible (e.g. in Kotlin data classes if they are part of constructor – lateinit were OK)

V1.0.0 – 2017/09/04

ObjectBox is out of beta! See our [announcement blog post](#) for details.

New Features

- Eager loading of relations via query builder
- Java filters as query post-processing
- Minor improvements like a new callInReadTx method and making Query.forEach breakable

Fixes

- Fixed two corner cases with queries

V0.9.15 (beta) 2017/08/21 Hotfixes

Fixes

- Fixed: Android flavors in caused the model file (default.json) to be written into the wrong folder (inside the build folder) causing the build to fail
- Fixed: failed builds if entity constructor parameters are of specific types

V0.9.14 (beta) 2017/08/14 Standalone relations, new build tools

For upgrade notes, please check the [announcement post](#).

New Features

- No more in-place code generation: Java source code is all yours now. This is based on the new build tool chain introduced in 0.9.13. Thus Kotlin and Java share the same build system. The old Java-based plugin is still available (plugin ID “io.objectbox.legacy”) in this version.
- “Standalone” to-many relations (without backing to-one properties/relations)
- Gradle plugin tries to automatically add runtime dependencies (also (k)apt, but this does not always work!?)
- Improved error reporting

Fixes

- Fixed the issue causing a “Illegal state: Tx destroyed/inactive, writeable cursor still available” error log

V0.9.13 (beta) 2017/07/12 Kotlin Support

New Features

- Kotlin support (based on a new annotation processor)
- Started “object-kotlin”, a sub-project for Kotlin extensions (tiny yet, let us know your ideas!)
- BoxStoreBuilder: added maxReaders configuration
- Get multiple entities by their IDs via Box methods (see get/getMap(Iterable))
- ToOne and ToMany are now serializable (which does not imply serializing is a good idea)
- ObjectBox may now opt to construct entities using the no-args constructor if the all-args constructor is absent
- Prevents opening two BoxStores for the same DB file, which may have side effects that are hard to debug
- Various minor and internal improvements and fixes

Fixes

- Fixed ToOne without an explicit target ID property
- Fixed type check of properties to allow ToMany (instead of List)
- Fixed @Convert in combination with List

- Fixed a race condition with cursor deletion when Java's finalizer kicked in potentially resulting in a SIGSEGV
- Fixed a leak with potentially occurring with indexes

V0.9.12 (beta) 2017/05/08 ToMany class

- **Update 2017/05/19:** We just released 0.9.12.1 for the Gradle plugin (only), which fixes two problems with parsing of to-many relations.
- Added the new list type ToMany which represents a to-many relation. A ToMany object will be automatically assigned to List types in the entity, eliminating a lot of generated code in the entity.
- ToMany comes with change tracking: all changes (add/remove) are automatically applied to the DB when its hosting entity is persisted via put(). Thus, the list content is synced to the DB, e.g. their relationship status is updated and new entities are put.
- Streamlined annotations (breaking API changes):
@Generated(hash = 123) becomes @Generated(123),
@property was removed,
@NameInDb replaces attributes in @Entity and the former @Property,
Backlinking to-many relations require @Backlink (only),
@Relation is now only used for to-one relations (and is subject to change in the next version)

V0.9.11 (beta) 2017/04/25: Various improvements

- Smarter to-one relations: if you put a new object that also has a new to-one relation object, the latter will also be put automatically.
- Getters and setters for properties are now only generated if no direct field access is possible
- JSR-305 annotations (@Nullable and others) to help the IDE find problems in your code
- @Uid(-1) will reassign IDs to simplify some migrations (docs will follow soon)
- No more getter for ToOne objects in favor of direct field access
- Quite a few internal improvements (evolved EntityInfo meta info object, etc.)

V0.9.10 (beta) 2017/04/10: Bug Fixes and minor improvements

New features and improvements

- Breaking API: Replaced “uid” attribute of @Entity and @Property with @Uid annotation
- An empty @Uid will retrieve the current UID automatically
- Some minor efficiency improvements for read transactions
- Better DB resources clean up for internal thread pool

Bug fixes

- Better compatibility with Android Gradle plugin
- Fixes for multithreaded reads of relation and index data
- Fixed compilation error in generated sources for Entities without non-ID properties

V0.9.9 (beta) 2017/03/07: Bug Fixes

New features

- Query.forEach() to iterate efficiently over query result objects

Bug fixes

- Various bug fixes

V0.9.8 (beta) 2017/02/22: Going Reactive

New features

- Data observers with reactive extensions for transformations, thread scheduling, etc.
- Optional RxJava 2 library
- OR conditions for QueryBuilder allow more powerful queries

Bug fixes

- Fixed: Changing the order of an entity's properties could cause errors in some cases
- Fixed: Querying using relation IDs

V0.9.7 (beta) 2017/02/10

New features

- LazyList returned by Query: another query option to defer getting actual objects until actually accessing them. This enables memory efficient iterations over large results. Also minimizes the time for a query to return. Note: LazyList cannot be combined with order specifications just yet.
- QueryBuilder and Query now support Date and boolean types directly
- QueryBuilder supports now a notIn operator
- put() now uses entity fields directly unless they are private (can be more efficient than calling getters)

Breaking internal changes

At this early point in the beta we decided to break backward compatibility. This allowed us to make important improvements without worrying about rather complex migrations of previous versions. We believe this was a special situation and future versions will likely be backward compatible although we cannot make promises. If you intend to publish an app with ObjectBox it's a good idea to contact us before.

- The internal data format was optimized to store data more compact. Previous database files are not compatible and should be deleted.
- We improved some details how IDs are used in the meta model. This affects the model file, which is stored in your project directory (objectbox-models/default.json). Files created by previous versions should be deleted.

V0.9.6 (first public beta release) 2017/01/24

See [ObjectBox Announcement](#)