D

Eclipse IDE Tips and Tricks

The Eclipse IDE is the most popular development environment for Android developers. In this appendix, we provide a number of helpful tips and tricks for using Eclipse to develop Android applications quickly and effectively.

Organizing Your Eclipse Workspace

In this section, we provide a number of tips and tricks to help you organize your Eclipse workspace for optimum Android development.

Integrating with Source Control Services

Eclipse has the ability to integrate with many source control packages using add-ons or plug-ins. This allows Eclipse to manage checking out a file—making it writable—when you first start to edit a file, checking a file in, updating a file, showing a file's status, and a number of other tasks, depending on the support of the add-on.

Tip

Common source control add-ons are available for CVS, Subversion, Perforce, git, Mercurial, and many other packages.

Generally speaking, not all files are suitable for source control. For Android projects, any file with the bin and gen directories shouldn't be in source control. To exclude these generically within Eclipse, go to Preferences, Team, Ignored Resources. You can add file suffixes such as *.apk, *.ap_, and *.dex by clicking the Add Pattern button and adding one at a time. Conveniently, this applies to all integrated source control systems.

Repositioning Tabs Within Perspectives

Eclipse provides some pretty decent layouts with the default perspectives. However, not every one works the same way. We feel that some of the perspectives have poor default layouts for Android development and could use some improvement.

Tip

Experiment to find a tab layout that works well for you. Each perspective has its own layout, too, and the perspectives can be task oriented.

For instance, the Properties tab is usually found on the bottom of a perspective. For code, this works fine because this tab is only a few lines high. But for resource editing in Android, this doesn't work so well. Luckily, in Eclipse, this is easy to fix: Simply drag the tab by left-clicking and holding on the tab (the title) itself and dragging it to a new location, such as the vertical section on the right side of the Eclipse window. This provides the much-needed vertical space to see the dozens of properties often found there.

Tip

If you mess up a perspective or just want to start fresh, you can reset it by choosing Window, Reset Perspective.

Maximizing Windows

Sometimes, you might find that the editor window is just too small, especially with all the extra little metadata windows and tabs surrounding it. Try this: Double-click the tab of the source file that you want to edit. Boom! It's now nearly the full Eclipse window size! Just double-click to return it to normal.

Minimizing Windows

You can minimize entire sections, too. For instance, if you don't need the section at the bottom that usually has the console or the one to the left that usually has the file explorer view, you can use the minimize button in each section's upper-right corner. Use the button that looks like two little windows to restore it.

Viewing Windows Side by Side

Ever wish you could see two source files at once? Well, you can! Simply grab the tab for a source file and drag it either to the edge of the editor area or to the bottom. You then see a dark outline, showing where the file will be docked—either side-by-side with another file or above or below another file. This creates a parallel editor area where you can drag other file tabs, as well. You can repeat this multiple times to show 3, 4, or more files at once.

Viewing Two Sections of the Same File

Ever wish you could see two places at once in the same source file? You can! Right-click the tab for the file in question and choose New Editor. A second editor tab for the same file comes up. With the previous tip, you can now have two different views of the same file.

Closing Unwanted Tabs

Ever feel like you get far too many tabs open for files you're no longer editing? I do! There are a number of solutions to this problem. First, you can right-click a file tab and choose Close Others to close all other open files. You can quickly close specific tabs by

middle-clicking on each tab. (This even works on a Mac with a mouse that can middle click, such as one with a scroll wheel.)

Keeping Windows Under Control

Finally, you can use the Eclipse setting that limits the number of open file editors:

1. Open Eclipse's Preferences dialog.
2. Expand General, choose Editors, and check Close Editors Automatically.
3. Edit the value in Number of Opened Editors Before Closing.

Eight seems to be a good number to use for the Number of Opened Editors Before Closing option to keep the clutter down but to have enough editors open to still get work done and have reference code open. Note also that if you check Open New Editor under When All Editors Are Dirty or Pinned, more files will be open if you're actively editing more than the number chosen. Thus, this setting doesn't affect productivity when you're editing a large number of files all at once but can keep things clean during most normal tasks.

Creating Custom Log Filters

Every Android log statement includes a tag. You can use these tags with filters defined in LogCat. To add a new filter, click the green plus sign button in the LogCat pane. Name the filter—perhaps using the tag name—and fill in the tag you want to use. Now there is another tab in LogCat that shows messages that contain this tag. In addition, you can create filters that display items by severity level.

Android convention has largely settled on creating tags based on the name of the class. You see this frequently in the code provided with this book. Note that we create a constant in each class with the same variable name to simplify each logging call. Here's an example:

```
public static final String DEBUG_TAG = "MyClassName";
```

This convention isn't a requirement, though. You could organize tags around specific tasks that span many activities or could use any other logical organization that works for your needs.

Writing Code in Java

In this section, we provide a number of tips and tricks to help you implement the code for your Android applications.

Using Auto-Complete

Auto-complete is a great feature that speeds up code entry. If this feature hasn't appeared for you yet or has gone away, you can bring it up by pressing Ctrl+spacebar. Auto-complete not only saves time in typing but can be used to jog your memory about methods—or to help you find a new method. You can scroll through all the methods of a class and even see the Javadocs associated with them. You can easily find static methods by using the class name or the instance variable name. You follow the class or variable name with a dot (and maybe Ctrl+spacebar) and then scroll through all the names. Then you can start typing the first part of a name to filter the results.

Formatting Code

Eclipse has a built-in mechanism for formatting Java code. Formatting code with a tool is useful for keeping the style consistent, applying a new style to old code, or matching styles with a different client or target (such as a book or an article).

To quickly format a small block of code, select the code and press Ctrl+Shift+F in Windows (or Command+Shift+F on a Mac). The code is formatted to the current settings. If no code is selected, the entire file is formatted. Occasionally, you need to select more code—such as an entire method—to get the indentation levels and brace matching correct.

The Eclipse formatting settings are found in the Properties pane under Java Code Style, Formatter. You can configure these settings on a per-project or workspace-wide basis. You can apply and modify dozens of rules to suit your own style.

Creating New Classes

You can quickly create a new class and corresponding source file by right-clicking the package to create it and choosing New, Class. Then you enter the class name, pick a superclass and interfaces, and choose whether to create default comments and method stubs for the superclass for constructors or abstract methods.

Creating New Methods

Along the same lines as creating new classes, you can quickly create method stubs by right-clicking a class or within a class in the editor and choosing Source, Override/Implement Methods. Then you choose the methods for which you're creating stubs, where to create the stubs, and whether to generate default comment blocks.

Organizing Imports

When referencing a class in your code for the first time, you can hover over the newly used class name and choose Import "*Classname*" (*package name*) to have Eclipse quickly add the proper import statement.

In addition, the Organize Imports command (Ctrl+Shift+O in Windows or Cmd+Shift+O on a Mac) causes Eclipse to automatically organize your imports. Eclipse removes unused imports and adds new ones for packages used but not already imported.

If there is any ambiguity in the name of a class during automatic import, such as with the Android Log class, Eclipse prompts you with the package to import. Finally, you can configure Eclipse to automatically organize the imports each time you save a file. This can be set for the entire workspace or for an individual project.

Configuring this for an individual project gives you better flexibility when you're working on multiple projects and don't want to make changes to some code, even if the changes are an improvement. To configure this, perform the following steps:

1. Right-click the project and choose Properties.
2. Expand Java Editor and choose Save Actions.
3. Check Enable Project Specific Settings, Perform the Selected Actions on Save, and Organize Imports.

Renaming Almost Anything

Eclipse's Rename tool is quite powerful. You can use it to rename variables, methods, class names, and more. Most often, you can simply right-click the item you want to rename and then choose Refactor, Rename. Alternatively, after selecting the item, you can press Ctrl+Alt+R in Windows (or Cmd+Alt+R on a Mac) to begin the renaming process. If you are renaming a top-level class in a file, the filename has to be changed as well. Eclipse usually handles the source control changes required to do this, if the file is being tracked by source control. If Eclipse can determine that the item is in reference to the identically named item being renamed, all instances of the name are renamed as well. Occasionally, this even means comments are updated with the new name. Quite handy!

Refactoring Code

Do you find yourself writing a whole bunch of repeating sections of code that look, for instance, like the following?

```
TextView nameCol = new TextView(this);
nameCol.setTextColor(getResources().getColor(R.color.title_color));
nameCol.setTextSize(getResources().
getDimension(R.dimen.help_text_size));
nameCol.setText(scoreUserName);
table.addView(nameCol);
```

This code sets text color, text size, and text. If you've written two or more blocks that look like this, your code could benefit from refactoring. Eclipse provides two useful tools—Extract Local Variable and Extract Method—to speed this task and make it almost trivial.

Using the Extract Local Variable Tool

Follow these steps to use the Extract Local Variable tool:

1. Select the expression `getResources().getColor(R.color.title_color)`.
2. Right-click and choose Refactor, Extract Local Variable (or press Ctrl+Alt+L).
3. In the dialog that appears, enter a name for the variable and leave the Replace All Occurrences check box selected. Then click OK and watch the magic happen.
4. Repeat steps 1–3 for the text size.

The result should now look like this:

```
int textColor = getResources().getColor(R.color.title_color);
float textSize = getResources().getDimension(R.dimen.help_text_size);
TextView nameCol = new TextView(this);
nameCol.setTextColor(textColor);
nameCol.setTextSize(textSize);
nameCol.setText(scoreUserName);
table.addView(nameCol);
```

All repeated sections of the last five lines also have this change made. How convenient is that?

Using the Extract Method Tool

Now you're ready for the second tool. Follow these steps to use the Extract Method tool:

1. Select all five lines of the first block of code.
2. Right-click and choose Refactor, Extract Method (or choose Ctrl+Alt+M).
3. Name the method and edit the variable names anything you want. (Move them up or down, too, if desired.) Then click OK and watch the magic happen.

By default, the new method is below your current one. If the other blocks of code are actually identical, meaning the statements of the other blocks must be in the exact same order, the types are all the same, and so on, they will also be replaced with calls to this new method! You can see this in the count of additional occurrences shown in the dialog for the Extract Method tool. If that count doesn't match what you expect, check that the code follows exactly the same pattern. Now you have code that looks like the following:

```
addTextToRowWithValues(newRow, scoreUserName, textColor, textSize);
```

It is easier to work with this code than with the original code, and it was created with almost no typing! If you had ten instances before refactoring, you've saved a lot of time by using a useful Eclipse feature.

Reorganizing Code

Sometimes, formatting code isn't enough to make it clean and readable. Over the course of developing a complex activity, you might end up with a number of embedded classes and methods strewn about the file. A quick Eclipse trick comes to the rescue: With the file in question open, make sure the outline view is also visible.

Simply click and drag methods and classes around in the outline view to place them in a suitable logical order. Do you have a method that is only called from a certain class but available to all? Just drag it in to that class. This works with almost anything listed in the outline, including classes, methods, and variables.

Providing Javadoc-Style Documentation

Regular code comments are useful (when done right). Comments in Javadoc style appear in code completion dialogs and other places, thus making them even more useful. To quickly add a Javadoc comment to a method or class, simply press Ctrl+Shift+J in Windows (or Cmd+Alt+J on a Mac). Alternatively, you can choose Source, Generate Element Comment to prefill certain fields in the Javadoc, such as parameter names and author, thus speeding the creation of this style of comment.

Resolving Mysterious Build Errors

Occasionally, you might find that Eclipse is finding build errors where there were none just moments before. In such a situation, you can try a couple quick Eclipse tricks.

First, try refreshing the project: Simply right-click the project and choose Refresh or press F5. If this doesn't work, try deleting the R.java file, which you can find under the gen directory under the name of the particular package being compiled. (Don't worry: This file is created during every compile.) If the Compile Automatically option is enabled, the file is re-created. Otherwise, you need to compile the project again.

Finally, you can try cleaning the project. To do this, choose Project, Clean and choose the projects you want to clean. Eclipse removes all temporary files and then rebuilds the project(s). If the project was an NDK project, don't forget to recompile the native code.

Note

Send us your own tips or tricks for Android development in Eclipse! You can email them to androidwirelessdev+awad2e@gmail.com.

This page intentionally left blank

E

The SQLite Quick-Start Guide

The Android System allows individual applications to have private SQLite databases in which to store their application data. This Quick-Start Guide is not a complete documentation of the SQLite commands. Instead, it is designed to get you up and running with common tasks. The first part of this appendix introduces the features of the `sqlite3` command-line tool. We then provide an in-depth database example using many common SQLite commands. See the online SQLite documentation (www.sqlite.org) for a complete list of features, functionality, and limitations of SQLite.

Exploring Common Tasks with SQLite

SQLite is a lightweight and compact, yet powerful, embedded relational database engine available as public domain. It is fast and has a small footprint, making it perfect for phone system use. Instead of the heavyweight server-based databases such as Oracle and Microsoft SQL Server, each SQLite database is within a self-contained single file on disk.

Android applications store their private databases (SQLite or otherwise) under a special application directory:

```
/data/data/<application package name>/databases/<databasename>
```

For example, the database for the `PetTracker` application provided in this book is found at

```
/data/data/com.androidbook.PetTracker/databases/pet_tracker.db
```

The database file format is standard and can be moved across platforms. You can use the Dalvik Debug Monitor Service (DDMS) File Explorer to pull the database file and inspect it with third-party tools, if you like.

Tip

Application-specific SQLite databases are private files accessible only from within that application. To expose application data to other applications, the application must become a content provider. Content providers are covered in Chapter 11, "Sharing Data Between Applications with Content Providers."

Using the `sqlite3` Command-Line Interface

In addition to programmatic access to create and use SQLite databases from within your applications, which we discuss in Chapter 10, "Using Android Data and Storage APIs," you can also interact with the database using the familiar command-line `sqlite3` tool, which is accessible via the Android Debug Bridge (ADB) remote shell.

The command-line interface for SQLite, called `sqlite3`, is exposed using the ADB tool, which we cover in Appendix C, "The Android Debug Bridge Quick-Start Guide."

Launching the ADB Shell

You must launch the ADB shell interface on the emulator or device (if it is rooted) to use the `sqlite3` commands. If only one Android device (or emulator) is running, you can connect by simply typing

```
c:\>adb shell
```

If you want to connect to a specific instance of the emulator, you can connect by typing

```
adb –s <serialNumber> shell
```

For example, to connect to the emulator at port 5554, you would use the following command:

```
adb –s emulator-5554 shell
```

For more information on how to determine the serial number of an emulator or device instance, please see Appendix C.

Connecting to a SQLite Database

Now you can connect to the Android application database of your choice by name. For example, to connect to the database we created with the PetTracker application, we would connect like this:

```
c:\>adb -e shell
# sqlite3 /data/data/com.androidbook.PetTracker/databases/pet_tracker.db
SQLite version 3.6.22
Enter ".help" for instructions
sqlite>
```

Now we have the `sqlite3` command prompt, where we can issue commands. You can exit the interface at any time by typing

```
sqlite>.quit
```

or

```
sqlite>.exit
```

Commands for interacting with the `sqlite3` program start with a dot (.) to differentiate them from SQL commands you can execute directly from the command line. This syntax might be different from other programs you are familiar with (for example, `mysql` commands).

> **Warning**
>
> Most Android devices don't allow running the `sqlite3` command as emulators do. Rooted devices do allow this command.

Exploring Your Database

You can use the `sqlite3` commands to explore what your database looks like and interact with it. You can

- List available databases
- List available tables
- View all the indices on a given table
- Show the database schema

Listing Available Databases

You can list the names and file locations attached to this database instance. Generally, you have your main database and a temp database, which contains temp tables. You can list this information by typing

```
sqlite> .databases
seq name file
--- ---- -----------------------------------------------
0   main /data/data/com.androidbook.PetTracker/databases/...
1   temp
sqlite>
```

Listing Available Tables

You can list the tables in the database you connect to by typing

```
sqlite> .tables
android_metadata table_pets       table_pettypes
sqlite>
```

Listing Indices of a Table

You can list the indices of a given table by typing

```
sqlite>.indices table_pets
```

Listing the Database Schema of a Table

You can list the schema of a given table by typing

```
sqlite>.schema table_pets
CREATE TABLE table_pets (_id INTEGER PRIMARY KEY
AUTOINCREMENT,pet_name TEXT,pet_type_id INTEGER);
sqlite>
```

Listing the Database Schema of a Database

You can list the schemas for the entire database by typing

```
sqlite>.schema
CREATE TABLE android_metadata (locale TEXT);
CREATE TABLE table_pets (_id INTEGER PRIMARY KEY
AUTOINCREMENT,pet_name TEXT,pet_type_id INTEGER);
CREATE TABLE table_pettypes (_id INTEGER PRIMARY KEY
AUTOINCREMENT,pet_type TEXT);
sqlite>
```

Importing and Exporting the Database and Its Data

You can use the `sqlite3` commands to import and export database data and the schema and interact with it. You can

- Send command output to a file instead of to STDOUT (the screen)
- Dump the database contents as a SQL script (so you can re-create it later)
- Execute SQL scripts from files
- Import data into the database from a file

> **Note**
>
> The file paths are on the Android device, not your computer. You need to find a directory on the Android device in which you have permission to read and write files. For example, `/data/local/tmp/` is a shared directory.

Sending Output to a File

Often, you want the sqlite3 command results to pipe to a file instead of to the screen. To do this, you can just type the `output` command followed by the file path to which the results should be written on the Android system. For example

```
sqlite>.output /data/local/tmp/dump.sql
```

Dumping Database Contents

You can create a SQL script to create tables and their values by using the `dump` command. The `dump` command creates a transaction, which includes calls to CREATE TABLE and INSERT to populate the database with data. This command can take an optional table name or dump the whole database.

Tip

The `dump` command is a great way to do a full archival backup of your database.

For example, the following commands pipe the dump output for the `table_pets` table to a file, and then sets the output mode back to the console:

```
sqlite>.output /data/local/tmp/dump.sql
sqlite>.dump table_pets
sqlite>.output stdout
```

You can then use DDMS and the File Explorer to pull the SQL file off the Android file system. The resulting `dump.sql` file looks like this:

```
BEGIN TRANSACTION;
CREATE TABLE table_pets (
_id INTEGER PRIMARY KEY AUTOINCREMENT,
pet_name TEXT,
pet_type_id INTEGER);

INSERT INTO "table_pets" VALUES(1,'Rover',9);
INSERT INTO "table_pets" VALUES(2,'Garfield',8);
COMMIT;
```

Executing SQL Scripts from Files

You can create SQL script files and run them through the console. These scripts must be on the Android file system. For example, let's put a SQL script called `myselect.sql` in the `/data/local/tmp/` directory of the Android file system. The file has two lines:

```
SELECT * FROM table_pettypes;
SELECT * FROM table_pets;
```

We can then run this SQL script by typing

```
sqlite>.read /data/local/tmp/myselect.sql
```

You see the query results on the command line.

Importing Data

You can import formatted data using the import and separator commands. Files such as CSV use commas for delimiters, but other data formats might use spaces or tabs. You specify the delimiter using the separator command. You specify the file to import using the import command.

For example, put a CSV script called `some_data.csv` in the `/data/local/tmp/` directory of the Android file system. The file has four lines. It is a comma-delimited file of pet type IDs and pet type names:

```
18,frog
19,turkey
20,piglet
21,great white shark
```

You can then import this data into the `table_pettypes` table, which has two columns: an `_id` column and a pet type name. To import this data, type the following command:

```
sqlite>.separator ,
sqlite>.import /data/local/tmp/some_data.csv table_pettypes
```

Now, if you query the table, you see it has four new rows.

Executing SQL Commands on the Command Line

You can also execute raw SQL commands on the command line. Simply type the SQL command, making sure it ends with a semicolon (;). If you use queries, you might want to change the output mode to column so that query results are easier to read (in columns) and the headers (column names) are printed. For example

```
sqlite> .mode column
sqlite> .header on
sqlite> select * from table_pettypes WHERE _id < 11;
_id        pet_type
---------- ----------
8          bunny
9          fish
10         dog
sqlite>
```

You're not limited to queries, either. You can execute any SQL command you see in a SQL script on the command line if you like.

Tip

We've found it helpful to use the `sqlite3` command line to test SQL queries if our Android SQL queries with `QueryBuilder` are not behaving. This is especially true of more complicated queries.

You can also control the width of each column (so text fields don't truncate) using the `width` command. For example, the following command prints query results with the first column 5 characters wide (often an ID column), followed by a second column 50 characters wide (text column).

```
sqlite> .width 5 50
```

Warning

SQLite keeps the database schema in a special table called `sqlite_master`. You should consider this table read-only. SQLite stores temporary tables in a special table called `sqlite_temp_master`, which is also a temporary table.

Using Other `sqlite3` Commands

A complete list of `sqlite3` commands is available by typing

```
sqlite> .help
```

Understanding SQLite Limitations

SQLite is powerful, but it has several important limitations compared to traditional SQL Server implementations, such as the following:

- SQLite is not a substitute for a high-powered, server-driven database.
- Being file-based, the database is meant to be accessed in a serial, not a concurrent, manner. Think "single user"—the Android application. It has some concurrency features, but they are limited.
- Access control is maintained by file permissions, not database user permissions.
- Referential integrity is not maintained. For example, FOREIGN KEY constraints are parsed (for example, in CREATE TABLE) but not enforced automatically. However, using TRIGGER functions can enforce them.
- ALTER TABLE support is limited. You can use only RENAME TABLE and ADD COLUMN. You may not drop or alter columns or perform any other such operations. This can make database upgrades a bit tricky.
- TRIGGER support is limited. You cannot use `FOR EACH STATEMENT` or `INSTEAD OF`. You cannot create recursive triggers.
- You cannot nest TRANSACTION operations.
- VIEWs are read-only.
- You cannot use RIGHT OUTER JOINs or FULL OUTER JOINs.
- SQLite does not support STORED PROCEDUREs or auditing.
- The built-in FUNCTIONs of the SQL language are limited.
- See the SQLite documentation for limitations on the maximum database size, table size, and row size. The Omitted SQL page is very helpful (http://www.sqlite.org/omitted.html), as is the Unsupported SQL Wiki (http://www.sqlite.org/cvstrac/wiki?p=UnsupportedSql).

Learning by Example: A Student Grade Database

Let's work through a student "Grades" database to show standard SQL commands to create and work with a database. Although you can create this database using the `sqlite3` command line, we suggest using the Android application to create the empty Grades database, so that it is created in a standard "Android" way.

The setup: The purpose of the database is to keep track of each student's test results for a specific class. In this example, each student's grade is calculated from their performance on

- Four quizzes (each weighted as 10% of overall grade)
- One midterm (weighted as 25% of overall grade)
- One final (weighted as 35% of overall grade)

All tests are graded on a scale of 0–100.

Designing the Student Grade Database Schema

The Grades database has three tables: Students, Tests, and TestResults.

The Students table contains student information. The Tests table contains information about each test and how much it counts toward the student's overall grade. Finally, all students' test results are stored in the TestResults table.

Setting Column Datatypes

`sqlite3` has support for the following common datatypes for columns:

- INTEGER (signed integers)
- REAL (floating point values)
- TEXT (UTF-8 or UTF-16 string; encoded using database encoding)
- BLOB (data chunk)

Tip

Do not store files such as images in the database. Instead, store images as files in the application file directory and store the filename or URI path in the database.

Creating Simple Tables with AUTOINCREMENT

First, let's create the Students table. We want a student id to reference each student. We can make this the primary key and set its AUTOINCREMENT attribute. We also want the first and last name of each student, and we require these fields (no nulls). Here's our SQL statement:

```
CREATE TABLE Students (
id INTEGER PRIMARY KEY AUTOINCREMENT,
fname TEXT NOT NULL,
lname TEXT NOT NULL );
```

For the Tests table, we want a test id to reference each test or quiz, much like the Students table. We also want a friendly name for each test and a weight value for how much each test counts for the student's final grade (as a percentage). Here's our SQL statement:

```
CREATE TABLE Tests (
id INTEGER PRIMARY KEY AUTOINCREMENT,
testname TEXT,
weight REAL DEFAULT .10 CHECK (weight<=1));
```

Inserting Data into Tables

Before we move on, let's look at several examples of how to add data to these tables. To add a record to the Students table, you need to specify the column names and the values in order. For example

```
INSERT into Students
(fname, lname)
VALUES
('Harry', 'Potter');
```

Now, we're going to add a few more records to this table for Ron and Hermione. At the same time, we need to add a bunch of records to the Tests table. First, we add the Midterm, which counts for 25 percent of the grade:

```
INSERT into Tests
(testname, weight)
VALUES
('Midterm', .25);
```

Then we add a couple quizzes, which use the default weight of 10 percent:

```
INSERT into Tests (testname) VALUES ('Quiz 1');
```

Finally, we add a Final test worth 35 percent of the total grade.

Querying Tables for Results with SELECT

How do we know the data we've added is in the table? Well, that's easy. We simply query for all rows in a table using a SELECT:

```
SELECT * FROM Tests;
```

This returns all records in the Tests table:

```
id    testname        weight
----- --------------- ------
1     Midterm         0.25
2     Quiz 1          0.1
3     Quiz 2          0.1
4     Quiz 3          0.1
5     Quiz 4          0.1
6     Final           0.35
```

Now, ideally, we want the weights to add up to 1.0. Let's check using the SUM aggregate function to sum all the weight values in the table:

```
SELECT SUM(weight) FROM Tests;
```

This returns the sum of all weight values in the Tests table:

```
SUM(weight)
-----------
1.0
```

We can also create our own columns and alias them. For example, we can create a column alias called `fullname` that is a calculated column: It's the student's first and last names concatenated using the || concatenation.

```
SELECT fname||' '|| lname AS fullname, id FROM Students;
```

This gives us the following results:

```
fullname         id
------------     --
Harry Potter      1
Ron Weasley       2
Hermione Granger  3
```

Using Foreign Keys and Composite Primary Keys

Now that we have our students and tests all set up, let's create the TestResults table. This is a more complicated table. It's a list of student-test pairings, along with the score.

The TestResults table pairs up student IDs from the Students table with test IDs from the Tests table. Columns, which link to other tables in this way, are often called foreign keys. We want unique student-test pairings, so we create a composite primary key from the student and test foreign keys. Finally, we enforce that the scores are whole numbers between 0 and 100. No extra credit or retaking tests in this class!

```
CREATE TABLE TestResults (
studentid INTEGER REFERENCES Students(id),
testid INTEGER REFERENCES Tests(id),
score INTEGER CHECK (score<=100 AND score>=0),
PRIMARY KEY (studentid, testid));
```

Tip

SQLite does not enforce foreign key constraints, but you can set them up anyway and enforce the constraints by creating triggers. For an example of using triggers to enforce foreign key constraints in SQL, check out the FullDatabase project provided on the book website for Chapter 10.

Now it's time to insert some data into this table. Let's say Harry Potter received an 82 percent on the Midterm:

```
INSERT into TestResults
(studentid, testid, score)
VALUES
(1,1,82);
```

Now let's input the rest of the student's scores. Harry is a good student. Ron is not a good student, and Hermione aces every test (of course). When they're all added, we can

list them. We can do a SELECT * to get all columns, or we can specify the columns we want explicitly like this:

```
SELECT studentid, testid, score FROM TestResults;
```

Here are the results from this query:

```
studentid  testid     score
---------- ---------- -----
1          1          82
1          2          88
1          3          78
1          4          90
1          5          85
1          6          94
2          1          10
2          2          90
2          3          50
2          4          55
2          5          45
2          6          65
3          6          100
3          5          100
3          4          100
3          3          100
3          2          100
3          1          100
```

Altering and Updating Data in Tables

Ron's not a good student, and yet he received a 90 percent on Quiz #1. This is suspicious, so as the teacher, we check the actual paper test to see if we made a recording mistake. He actually earned 60 percent. Now we need to update the table to reflect the correct score:

```
UPDATE TestResults
SET score=60
WHERE studentid=2 AND testid=2;
```

You can delete rows from a table using the DELETE function. For example, to delete the record we just updated:

```
DELETE FROM TestResults WHERE studentid=2 AND testid=2;
```

You can delete all rows in a table by not specifying the WHERE clause:

```
DELETE FROM TestResults;
```

Querying Multiple Tables Using JOIN

Now that we have all our data in our database, it is time to use it. The preceding listing was not easy for a human to read. It would be much nicer to see a listing with the names of the students and names of the tests instead of their IDs.

Combining data is often handled by performing a JOIN with multiple table sources; there are different kinds of JOINS. When you work with multiple tables, you need to specify which table a column belongs to (especially with all these different id columns). You can refer to columns by their column name or by their table name, then a dot (.), and then the column name.

Let's relist the grades again, only this time, include the name of the test and the name of the student. Also, we limit our results only to the score for the Final (`test id 6`):

```
SELECT
Students.fname||' '|| Students.lname AS StudentName,
Tests.testname,
TestResults.score
FROM TestResults
JOIN Students
       ON (TestResults.studentid=Students.id)
JOIN Tests
       ON (TestResults.testid=Tests.id)
WHERE testid=6;
```

which gives us the following results (you could leave off the WHERE to get all tests):

```
StudentName        testname       score
------------------ -------------- -----
Harry Potter       Final          94
Ron Weasley        Final          65
Hermione Granger   Final          100
```

Using Calculated Columns

Hermione always likes to know where she stands. When she comes to ask what her final grade is likely to be, we can perform a single query to show all her results and calculate the weighted scores of all her results:

```
SELECT
Students.fname||' '|| Students.lname AS StudentName,
Tests.testname,
Tests.weight,
TestResults.score,
(Tests.weight*TestResults.score) AS WeightedScore
FROM TestResults
JOIN Students
       ON (TestResults.studentid=Students.id)
```

```
JOIN Tests
       ON (TestResults.testid=Tests.id)
WHERE studentid=3;
```

This gives us predictable results:

```
StudentName        testname weight score WeightedScore
----------------   -------- ------ ----- ------------
Hermione Granger   Midterm  0.25   100        25.0
Hermione Granger   Quiz 1   0.1    100        10.0
Hermione Granger   Quiz 2   0.1    100        10.0
Hermione Granger   Quiz 3   0.1    100        10.0
Hermione Granger   Quiz 4   0.1    100        10.0
Hermione Granger   Final    0.35   100        35.0
```

We can just add up the Weighted Scores and be done, but we can also do it via the query:

```
SELECT
Students.fname||' '|| Students.lname AS StudentName,
SUM((Tests.weight*TestResults.score)) AS TotalWeightedScore
FROM TestResults
JOIN Students
       ON (TestResults.studentid=Students.id)
JOIN Tests
       ON (TestResults.testid=Tests.id)
WHERE studentid=3;
```

Here we get a nice consolidated listing:

```
StudentName      TotalWeightedScore
---------------- ------------------
Hermione Granger 100.0
```

If we wanted to get all our students' grades, we need to use the GROUP BY clause. Also, let's order them so the best students are at the top of the list:

```
SELECT
Students.fname||' '|| Students.lname AS StudentName,
SUM((Tests.weight*TestResults.score)) AS TotalWeightedScore
FROM TestResults
JOIN Students
       ON (TestResults.studentid=Students.id)
JOIN Tests
       ON (TestResults.testid=Tests.id)
GROUP BY TestResults.studentid
ORDER BY TotalWeightedScore DESC;
```

This makes our job as teacher almost too easy, but at least we're saving trees by using a digital grade book.

```
StudentName               TotalWeightedScore
------------------------- ------------------
Hermione Granger          100.0
Harry Potter              87.5
Ron Weasley               46.25
```

Using Subqueries for Calculated Columns

You can also include queries within other queries. For example, you can list each Student and a count of how many tests they "passed," in which passing is getting a score higher than 60, as in the following:

```
SELECT
Students.fname||' '|| Students.lname AS StudentName,
Students.id AS StudentID,
(SELECT COUNT(*)
FROM TestResults
WHERE TestResults.studentid=Students.id
AND TestResults.score>60)
AS TestsPassed
FROM Students;
```

Again, we see that Ron needs a tutor:

```
StudentName       StudentID TestsPassed
-----------       --------- ----------
Harry Potter      1         6
Ron Weasley       2         1
Hermione Granger  3         6
```

Deleting Tables

You can always delete tables using the DROP TABLE command. For example, to delete the TestResults table, use the following SQL command:

```
DROP TABLE TestResults;
```