CHAPTER 3

From Sketch to Play Store

In this chapter, we will go through all the steps involved in the creation of a Processing for Android project, from sketching and debugging to exporting the project as a signed app ready for upload to the Google Play Store. We will use the vine drawing sketch from the previous chapter as the project to upload to the store.

Sketching and Debugging

In the first two chapters, we talked about "code sketching," where immediate visual output and quick iteration are key techniques to develop projects with Processing. Another important technique that we did not mention yet is the identification and resolution of errors or "bugs" in the code, a process called debugging.

Debugging can take us as much time as writing the code itself. What makes debugging challenging is that some bugs are the result of faulty logic or incorrect calculations, and because there are no typos or any other syntactical errors in the code, Processing can still run the sketch. Unfortunately, there is no foolproof technique to eliminate all bugs in a program, but Processing provides some utilities to help us with debugging.

Getting Information from the Console

The simplest way to debug a program is printing the values of variables and messages along various points of the execution flow of the program. Processing's API includes text-printing functions, `print()` and `println()`, which output to the console area in the PDE. The only difference between these two functions is that `println()` adds a new line break at the end while `print()` does not. Listing 3-1 shows a sketch using println() to indicate the occurrence of an event (a mouse press in this case) and the value of built-in variable.

Listing 3-1. Using println() in a sketch to show information on the console

```
void setup() {
  fullScreen();
}

void draw() {
  println("frame #", frameCount);
}

void mousePressed() {
  println("Press event");
}
```


A. Colubri, *Processing for Android*, https://doi.org/10.1007/978-1-4842-9585-4_3

Processing's console shows anything that is printed with these functions, but also warning or error messages indicating a problem in the execution of the sketch (Figure 3-1).

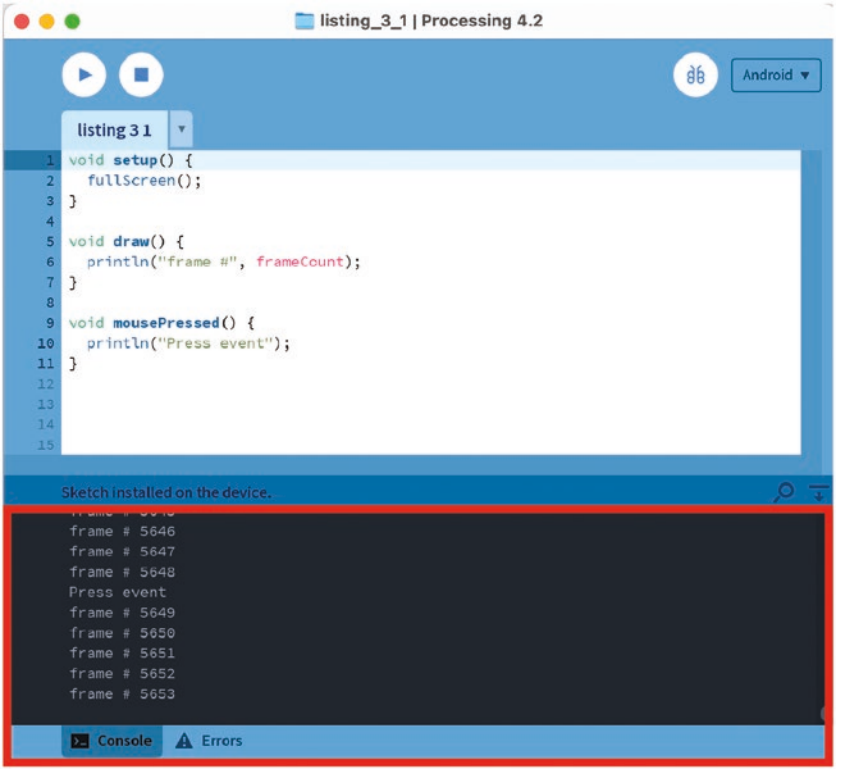

Figure 3-1. *Console area in the PDE, outlined with red*

The main problem of printing messages to the console for debugging is that it requires adding these additional function calls for each variable we want to keep track of. Once we are done with debugging, we need to remove or comment out all these calls, which can become inconvenient for large sketches.

■ **Note** Comments in Processing work the same way as in Java: we can comment out a single line of code using two consecutive forward slashes, "//", and an entire block of text with "/*" at the beginning of the block and "*/" at the end. We can also use the Comment/Uncomment option under the "Edit" menu in the PDE.

Getting more information with logcat

We can obtain a lot of useful information from the Processing console, but sometimes, this may not be enough to find out what is wrong with our sketch. The Android SDK includes several command-line tools that can help us with debugging. The most important SDK tool is adb (Android Debug Bridge), which makes possible the communication between the computer we are using for development and the device or emulator. In fact, Processing uses adb under the hood to query what devices are available and to install the sketch on the device or emulator when running it from the PDE.

We can also use adb manually, for example, to get more detailed debug messages. To do this, we need to open a terminal console, and once in it, we would need to change to the directory where the Android SDK is installed. In case the SDK was automatically installed by Processing, it should be located inside the sketchbook folder, the modes/AndroidMode subfolder. Within that folder, the SDK tools are found in sdk/platform-tools. Once there, we can run the adb tool with the logcat option, which prints out the log with all the messages. For instance, the following is the sequence of commands we would need on Mac to run logcat:

```
$ cd ~/Documents/Processing/android/sdk/platform-tools
$ ./adb logcat
```

By default, logcat prints all messages generated by the Android device or emulator, not only from the sketch we are debugging but also from all processes that are currently running, so we might get too many messages. The print messages from Processing can be displayed if using logcat with the –I option. Logcat has additional options to only show error messages (-E) or warnings (-W). The full list of options is available on the Google's developer site (https://developer.android.com/tools/logcat).

Using the Integrated Debugger

The Android mode offers an "integrated debugger" tool that makes it easy to keep track of the internal state of a running sketch. We turn the debugger on by pressing the button with the butterfly icon on the left of the menu bar, next to the mode selector, or selecting the "Enable Debugger" in the Debug menu. Once enabled, we can access many additional options in the PDE to use when the sketch is running. For example, we can add "checkpoints" to any line in the code of our sketch. A checkpoint signals where the execution of the sketch should stop to allow us to inspect the value of all the variables in the sketch, both user defined and built-in.

We can create a new checkpoint by double-clicking on the line number in the left margin of the code editor. A diamond sign will indicate that the line has flagged with a checkpoint. When we run a sketch containing one or more checkpoints, Processing will stop execution when it reaches each checkpoint, at which moment we can inspect the value of the variables using the variable inspector window (Figure 3-2). We resume execution by pressing the continue button on the toolbar. We can also step line by line by pressing the Step button and see how each variable changes its value after each line.

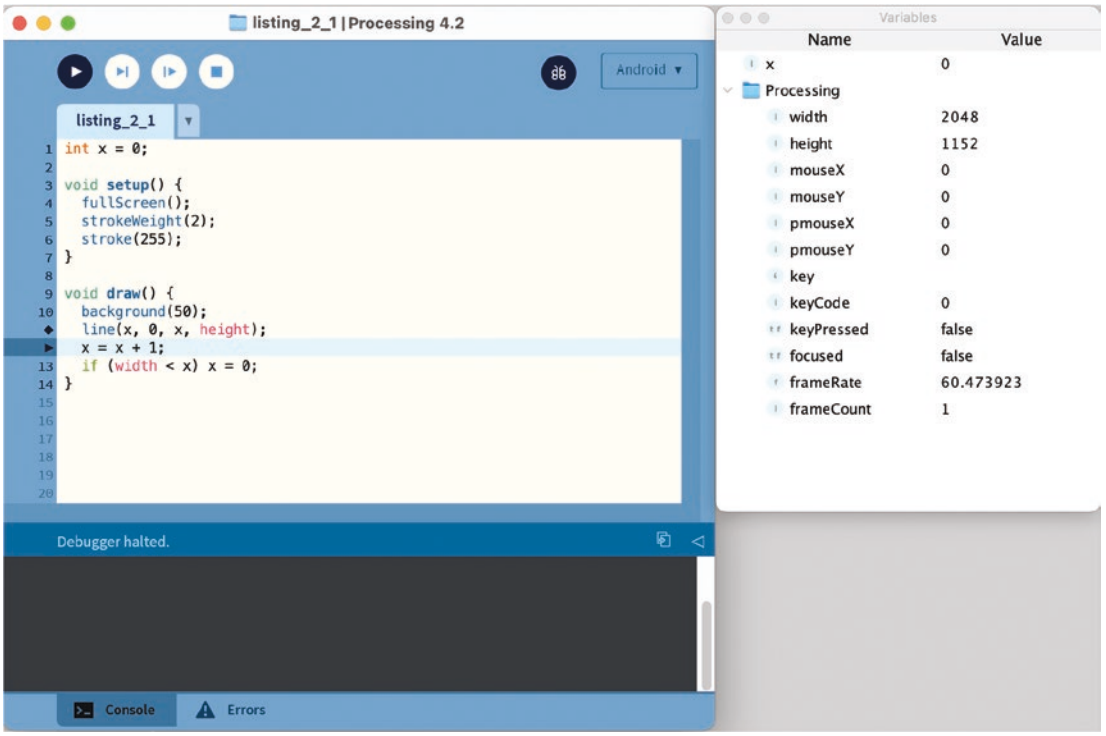

Figure 3-2. *Debugging session with the integrated debugger in the Android mode*

All this functionality in the integrated debugger could help us identify bugs in the code without the need of adding print instructions, although fixing a tricky bug is always challenging and can take a long time even with the debugger. At the end, it comes down to carefully understanding the logic of the code in the sketch and its possible consequences and edge cases, based on information we get from the debugger or print instructions. In this way, we can narrow down the portion of the code where the bug is likely to be.

Reporting Processing Bugs

Sometimes, an unexpected or erroneous behavior in a Processing sketch may be due not to a bug in our code, but in Processing itself! If you have strong suspicion that you have found a Processing bug, you can report it in the GitHub page of the project. If it is a bug affecting the Android mode, please open a new issue in the processing-android repository at `https://github.com/processing/processing-android/issues` and include as much information as possible to reproduce the bug and to help the developers replicate the bug and eventually fix it.

Preparing a Sketch for Release

After debugging a sketch in the PDE, we may want to bundle it for public release through Google Play Store. When working from the PDE, Processing creates a debug app bundle that can only be installed on our own device or emulator for testing purposes. Creating an app suitable for general distribution from our sketch requires some additional steps and considerations to make sure it can be uploaded to the Play Store.

Adjusting for Device's DPI

A first step to prepare our sketch for public release is to check that it can be run on most of the Android devices in use. When writing and debugging our sketch, it is often the case that we work with one or only a few different devices, so it may be hard to anticipate issues on devices we do not have access to. A common situation is that the graphics might look either too big or too small when running our Processing sketches on different devices. The reason for this is that both resolution (number of pixels) and physical screen size can vary quite a lot across phones, tablets, and watches, and so graphic elements designed with one resolution in mind and viewed on a screen of a particular size will likely look wrong on another. Since Android is designed to support various combinations of screen sizes and resolutions, we need a way in Processing to adapt the visual design of our sketch so it looks as intended across different devices.

The ratio of the resolution to the screen size gives what is called the DPI (dots per inch, which, in the context of computer screens, is equivalent to pixels per inch or PPI). The DPI is the basic magnitude to compare across devices. It is important to keep in mind that higher DPI does not necessarily mean a higher resolution, since two different devices with the same resolution may have different screen sizes. For example, the Galaxy Nexus (4.65" diagonal) has a 720×1280 pixels resolution, while the Nexus 7 (7" diagonal) has an 800×1280 pixels resolution. The DPIs of these devices are 316 and 216, even though their resolutions are very similar.

Android classifies devices in density buckets according to the following six generalized densities (a specific device will fall in one of these categories depending on which one is closest to its actual DPI):

- ldpi (low): ~120dpi
- mdpi (medium): ~160dpi
- hdpi (high): ~240dpi
- xhdpi (extra-high): ~320dpi
- xxhdpi (extra-extra-high): ~480dpi
- xxxhdpi (extra-extra-extra-high): ~640dpi

The generalized density levels are important in Processing to generate the app icons, as we will see later in this chapter, but not so much when writing our code. To make sure that the visual elements in our sketch scale properly across different devices, there is another built-in constant from Android that Processing makes available through its API. This is the "display density," a number that represents how much bigger (or smaller) is the pixel in our device when compared with a reference 160 DPI screen (e.g., a 320×480, 3.5" screen). Thus, on a 160 DPI screen, this density value will be 1; on a 120 DPI screen, it would be .75; etc.

■ **Note** Google's API Guide on Multiple Screen Support gives detailed information about the density independence on Android: https://developer.android.com/guide/practices/screens_support.html.

The display density is available in Processing as the constant named `displayDensity`, which we can use from anywhere in our code. The simplest way of adjusting the output to the device's DPI is to multiply the size of all the graphical elements in the sketch by `displayDensity`, which is the approach in Listing 3-2. As we can see in Figure 3-3, the size of the circles drawn by the sketch is the same across devices with different DPIs. Also, this example uses the `map()` function to convert the index variables i and j, which go from 0 to maxi and maxj, to the coordinate values x and y, which should be in the range (0, width) and (0, height), respectively.

Listing 3-2. Using displayDensity to adjust our sketch to different screen sizes and resolutions

```
void setup() {
  fullScreen();
  noStroke();
}

void draw() {
  background(0);
  float r = 50 * displayDensity;
  int maxi = int(width/r);
  int maxj = int(height/r);
  for (int i = 0; i <= maxi; i++) {
    float x = map(i, 0, maxi, 0, width);
    for (int j = 0; j <= maxj; j++) {
      float y = map(j, 0, maxj, 0, height);
      ellipse(x, y, r, r);
    }
  }
}
```

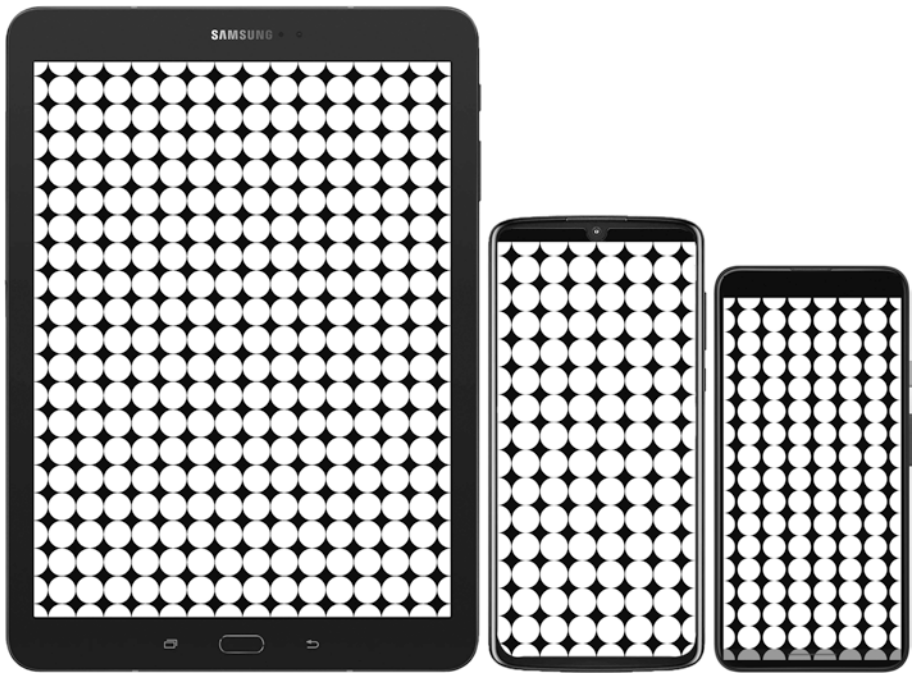

Figure 3-3. *From left to right, output of our sketch on a Samsung Galaxy Tab S3 (9.7", 20480×1536 px, 264 dpi), a Moto Z4 (6.4", 2340×1080 px, 403 dpi), and a Pixel 4a (5.81", 2340×1080 px, 433 dpi)*

We can return now to our vine drawing sketch from the previous chapter and add `displayDensity` in the parts of the code where we need to scale the graphics. More specifically, any variable or value that represents the size of shapes and the position of vertices on the screen, or is related to handling mouse or touch pointer, should be multiplied by `displayDensity`. Listing 3-3 shows these changes applied to the original drawing sketch.

Listing 3-3. Adding displayDensity to the vine drawing sketch from Chapter 2

```
void setup() {
  fullScreen();
  strokeWeight(2 * displayDensity);
  stroke(121, 81, 49, 150);
  fill(255);
  background(255);
}

void draw() {
  if (mousePressed) {
    line(pmouseX, pmouseY, mouseX, mouseY);
    ellipse(mouseX, mouseY, 13 * displayDensity, 13 * displayDensity);
    if (30 * displayDensity < dist(pmouseX, pmouseY, mouseX, mouseY)) {
      drawLeaves();
    }
  }
}

void drawLeaves() {
  int numLeaves = int(random(2, 5));
  for (int i = 0; i < numLeaves; i++) {
    float leafAngle = random(0, TWO_PI);
    float leafLength = random(20, 100) * displayDensity;
    pushMatrix();
      translate(mouseX, mouseY);
      rotate(leafAngle);
      line(0, 0, leafLength, 0);
      translate(leafLength, 0);
      pushStyle();
        noStroke();
        fill(random(170, 180), random(200, 230), random(80, 90), 190);
        float r = random(20, 50) * displayDensity;
        beginShape();
        int numSides = int(random(4, 8));
        for (float angle = 0; angle <= TWO_PI; angle += TWO_PI/numSides) {
          float x = r * cos(angle);
          float y = r * sin(angle);
          vertex(x, y);
        }
        endShape();
      popStyle();
    popMatrix();
  }
}
```

Using the Emulator

We briefly discussed the emulator in the first chapter. Even when we have our own device, the emulator could be useful, because it allows us to test hardware configurations that we do not have access to. Processing creates a default Android Virtual Device (AVD) to run in the emulator, using the Pixel 6 settings (1080×2400 px, 411 dpi). We can create other AVDs with different properties to test our sketches on, using the command-line tool avdmanager, included in the Android SDK. We need to keep in mind that the emulator will likely run slower than an actual device, especially if creating high-resolution AVDs or with other high-end capabilities.

Since avdmanager is a command-line tool, we first need to open a terminal console and change to the tools directory where avdmanager and the emulator launcher are located inside the SDK folder. The sequence of steps to create a new AVD using the device definition for a Pixel C tablet, and then launching this AVD with the emulator, is as follows:

```
$ cd ~/Documents/Processing/android/sdk
$  cmdline-tools/latest/bin/avdmanager create avd -n processing-tablet -k "system-
images;android-33;google_apis;x86_64" -d "pixel_c" --skin "pixel_c" -p ~/Documents/
Processing/android/avd/processing-tablet
$ emulator/emulator -avd processing-tablet -gpu auto -port 5566
```

In the line running the avdmanager command, we provided four arguments:

- -n processing-tablet: The name of the AVD, which could be any name we wish to use.
- -k “system-images;android-33;google_apis;x86_64”: The SDK package to use for the AVD; to find out which SDK packages are available in our SDK, we need to look at the system-images subfolder inside the SDK folder.
- -d “pixel_c”: A device definition containing the hardware parameters of the device we want to emulate. We can list all the available device definitions by running the command “./avdmanager list devices”.
- --skin “pixel_c”: The name of the skin containing the images that the emulator will use to draw the frame of the device. This is optional; if a skin name is not provided, then the emulator window will not have a frame. The skin files need to be included in the SDK; when the Android mode downloads the default SDK, it also retrieves some skins from Google’s servers and places them inside the skins subfolder inside the SDK folder.
- -p ~/Documents/Processing/android/avd/processing-tablet: The folder where we will store this AVD; in this case, we are using “android/avd/processing-tablet” inside the sketchbook folder, since this is the default location the mode uses for the default AVDs.

After we created a new AVD with the avdmanager, we can manually edit the configuration file that contains all the parameters of the AVD, which, in the case of this example, will be stored in “~/Documents/Processing/android/avd/ processing-tablet/config.ini”. We can change the skin name and the path by modifying the values of the parameters skin.name and skin.path, respectively. If we remove the skin path altogether, we can set the AVD to any arbitrary resolution by entering a value of the form “wxh” as the skin.name, as shown in Figure 3-4.

Figure 3-4. *Adding a skin resolution to the AVD's config.ini file*

In general, the default device definitions together with a matching skin should be enough to have a working AVD to test our sketches on. The emulator command to launch the AVD includes the following arguments:

- -avd processing-tablet: The name of the AVD we want to launch.
- -gpu auto: Enables the emulator to use hardware acceleration on the computer to render the AVD's screen faster if it is available. Otherwise, it will use a slower software renderer.
- -port 5566: Sets the TCP port number to connect the console and adb with the emulator.

To use our new AVD in place of Processing's default, we should launch it manually as we did in this example, and then Processing will install our sketches in this AVD instead of the default AVD. However, we need to make sure to use the right port parameter, because Processing will only be able to communicate with phone emulators running on port 5566 and watch emulators on port 5576.

■ **Note** Google's Android developer site includes pages on avdmanager (https://developer.android.com/tools/avdmanager) and running the emulator from the command line (https://developer.android.com/studio/run/emulator-commandline.html) where we can find more information about these tools.

Setting Icons and Bundle Name

Android apps require icons of various sizes to be displayed at different pixel densities in the app launcher menu. Processing uses a set of default, generic icons when running a sketch from the PDE, but these icons should not be used for a public release.

To add our own icons to the project, we need to create the following files: icon-36, icon-48, icon-72, icon-96, icon-144, and icon-192 in .PNG format, for the ldpi (36×36), mdpi (48×48), hdpi (72×27), xhdpi (96×96), xxhdpi (144×144), and xxxhdpi (192×192) resolutions. Once we have these files, we must place them in the sketch's folder before exporting the signed bundle.

For the vine drawing app from the previous chapter, we will use the set of icons shown in Figure 3-5.

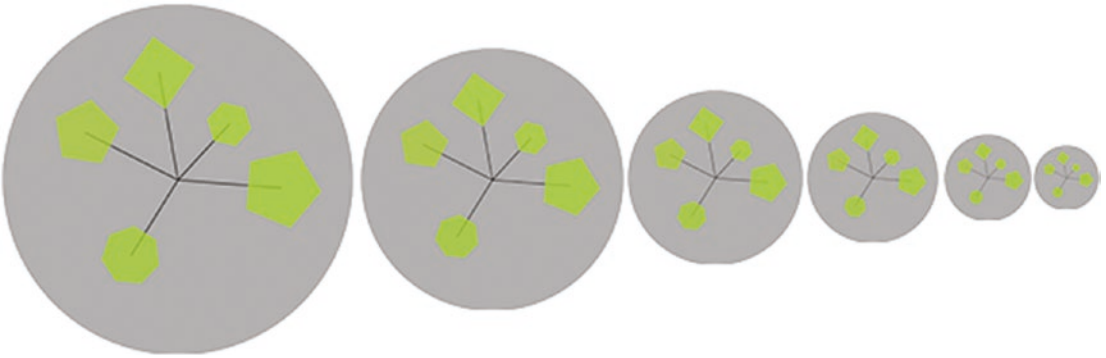

Figure 3-5. *Set of icons for the vine drawing app*

Google has published a set of guidelines and resources for icon creation, according to their material UI style, available at https://m2.material.io/design/platform-guidance/android-icons.html.

Setting Package Name and Version

Apps in the Google App Store are uniquely identified by a package name, which is a string of text that looks something like "com.example.helloworld". This package name follows the Java package naming convention, where the app name ("helloworld") is last, preceded by the website of the company or person developing the app in reverse order ("com.example").

Processing constructs this package name automatically by prepending a base name to the sketch name. The default base name is "processing.test", and we can change it by editing the manifest.xml file that Processing generates in the sketch folder after we run it for the first time from the PDE (either on a device or in the emulator). We can also set the version code and version name. For example, in the following manifest file generated by Processing, the base package name is "com.example", the version code is 10, and the version name is 0.5.4:

```
<?xml version="1.0" encoding="UTF-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
          android:versionCode="10" android:versionName="0.5.4"
          package="com.example">
    <application android:icon="@drawable/icon"
                 android:label="Vines Draw">
        <activity android:name=".MainActivity"
                  android:theme=
```

```
                "@style/Theme.AppCompat.Light.NoActionBar.FullScreen">
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
                <category android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
        </activity>
    </application>
</manifest>
```

If we save our sketch as "HelloWorld", then the full package name will be "com.package.helloworld" (Processing will set all the letters in the name to lowercase). Note that the package name of our app must be unique since there cannot be two apps on the Google Play Store with the same package name. Also, we should set the application name using the android:label attribute in the application tag. Android will use this label as the visible title of the app in the launcher and other parts of the UI.

Exporting As Signed Bundle

The Android mode simplifies the publishing of our sketch by signing and aligning the app so we can upload it to the Google Play Developer Console without any extra additional steps. All we need to do is to select the "Export Signed Bundle" option under the "File" menu (Figure 3-6).

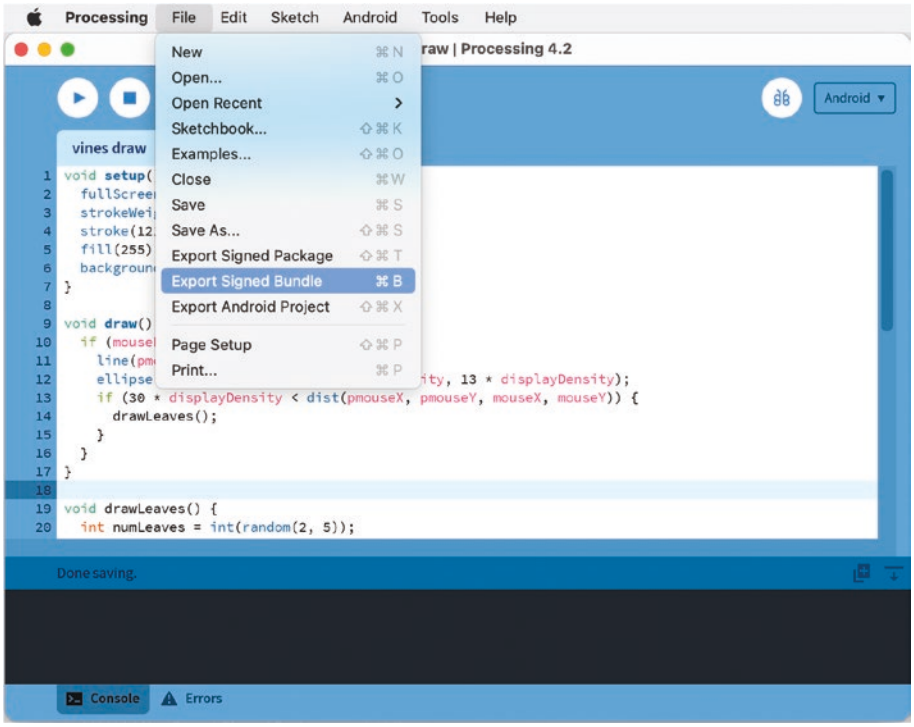

Figure 3-6. *The "Export Signed Bundle" option in the PDE's "File" menu*

After selecting this option, Processing will ask us to create a new keystore to store the upload key to sign the app bundle. The keystore requires a password and additional information about the keystore issuer (name, organization, city, state, country), although those are optional. The keystore manager window that allows us to enter all this information is displayed in Figure 3-7.

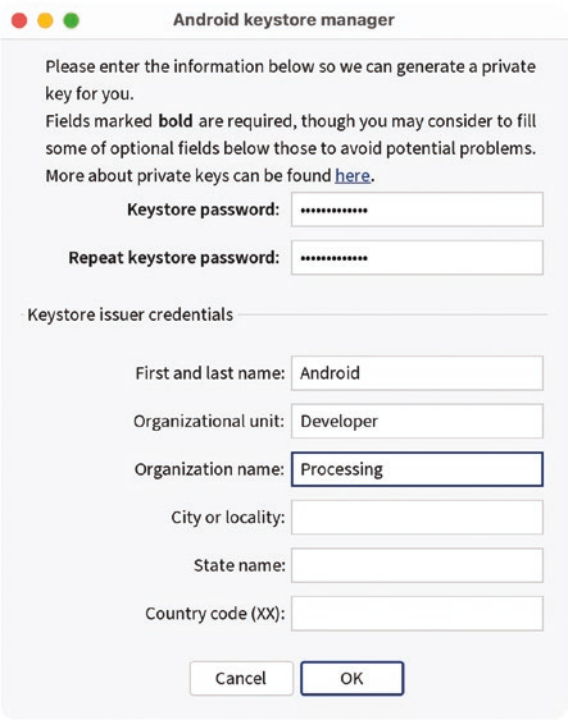

Figure 3-7. *Entering the information needed to create a keystore in Processing*

Remember this password for the upload key, as you will have to use it every time you export a new signed bundle. If you lose access to the upload key, or it gets compromised, you can still create a new one and then contact Google support to reset the key. This is explained in this article from Play Console Help: https://support.google.com/googleplay/android-developer/answer/9842756.

The signed and aligned bundle will be saved in the build subfolder inside the sketch's folder, under the name [Sketch name in lowercase]_release.aab. Once we have this file, we can follow the instructions from Google to complete the app publishing process: https://play.google.com/console/about/guides/releasewithconfidence/.

If we follow all these steps with our vine drawing sketch, we should be able to generate a signed and aligned bundle ready to upload to the Play Store. We can also install it manually on our device using the adb tool from the command line:

```
$ cd ~/Documents/Processing/android/sdk/platform-tools
$ ./adb install ~/Documents/Processing/vines_draw/buildBundle/vides_draw_release.aab
```

If we install the final drawing app bundle either manually or through the Play Store, we should see it in the app launcher with the icon we created for it (Figure 3-8).

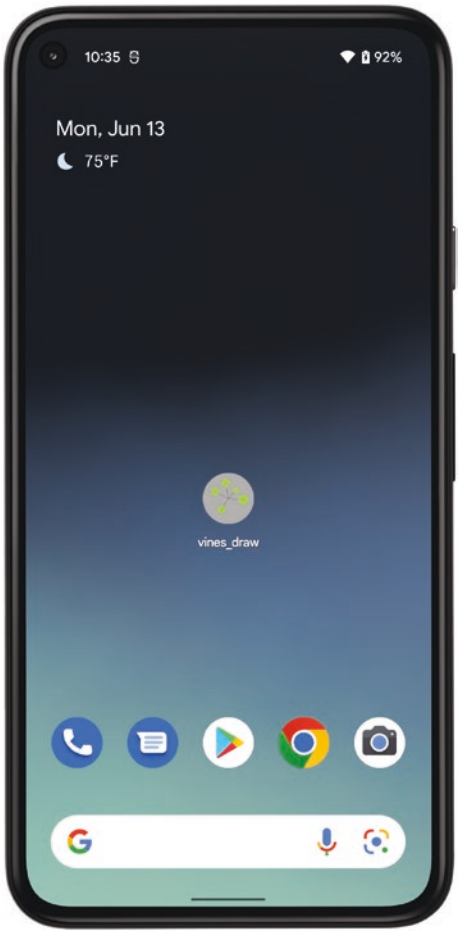

Figure 3-8. *The vine drawing app installed on our device*

Summary

This final chapter in the first part of the book covered several additional important topics, ranging from debugging our code using Processing's console, the integrated debugger, or the logcat option in adb; scaling the output of our sketches according to the device's DPI; and finally exporting our sketch as signed bundle to upload to the Play Store. With these tools, we are ready to share our creations with all the Android users around the world!