Real-Time IoT Imaging with Deep Neural Networks

Using Java on the Raspberry Pi 4

Nicolas Modrzyk

Apress®

Real-Time IoT Imaging with Deep Neural Networks

Using Java on the Raspberry Pi 4

Nicolas Modrzyk

Apress®

Real-Time IoT Imaging with Deep Neural Networks

Nicolas Modrzyk
Tokyo, Tokyo, Japan

ISBN-13 (pbk): 978-1-4842-5721-0 ISBN-13 (electronic): 978-1-4842-5722-7
https://doi.org/10.1007/978-1-4842-5722-7

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Nikhil Karkal
Development Editor: Rita Fernando
Coordinating Editor: Divya Modi

Cover designed by eStudioCalamar

Cover image designed by Pixabay

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-5721-0. For more detailed information, please visit www.apress.com/source-code.

Printed on acid-free paper

This book is dedicated to three very special cats: Marcel, Otto, and Mofu.

They probably will never read this book…but maybe they will sleep on it.

Table of Contents

About the Author

Nicolas Modrzyk has more than 15 years of IT experience in Asia, Europe, and the United States. He is currently the CTO of an international consulting company in Tokyo, Japan. An author of four other published books, he mostly focuses on the Clojure language and expressive code. When not bringing new ideas to customers, he spends time with his two fantastic daughters, Mei and Manon, and plays live music internationally.

About the Technical Reviewer

David Thevenin is a software engineer in a securities brokerage firm. After having defended his PhD in computer science, he decided to do research and development in private work. He is interested in new technologies that improve access to services in mobile environments and that reduce the gap between the user and the computer. He specializes in mobile devices, web technologies, graphic toolkit design, and natural language processing.

Acknowledgments

Thank you to Mei and Manon, for your love, energy, and smiles.

Thank you to my grandparents, because you are the source of life and the profound inspiration for this book.

Thank you to all my family—parents, brothers, sisters, faraway cousins, aunts, and uncles; even though I'm so far away, you always give invaluable love, care, and support.

Thank you, Parrain; I have one special line for you, because I'm trying to live by your high moral standards, and it is f*** hard.

Thank you, Diyya; I could feel you looking over my shoulder to see whether I was working on this or not. Wait, are you still looking?

Thank you, Nikhil, for kick-starting this.

Thank you, David, for your patience through all those "Nico, ca marche pas ton truc."

Thank you, Rita, for your support, understanding, and letting me keep all those carefully chosen quotes.

Thank you, Lemons, because you musically squeeze so well.

Thank you, soccer friends, because I would be fat and lazy if you were not there.

Thank you, Karabiner team, because you make me realize everyone is so different, but everyone has an important part to play to win the game.

Thank you to friends around the world—the ones still here but also the ones who parted. Each encounter is an inspiration.

Finally, thank you, Yoko, for spitting wine on my brand new costume and still managing to make it so beautifully romantic.

Introduction

My grandfather, of Polish origins, used to cook, make, build, plant, grow, and create everything with his own two hands. He also loved to share with adults and children alike how he was doing all these creative tasks. As a child, I was always impressed by his "If I don't have it, I will try to make it myself" attitude. He put in the time and the effort to make something original all the time.

One of the cutest memories I have is of us walking in the garden in the cold winter. We saw some birds looking for tiny pieces of food. We tried to give them French bread and grains, but they would not, or maybe could not, get to them because of the freezing winter cold. So, he proposed we build a wooden house for the birds so they could come and eat the grains while being protected from the winds. And so we did. We spent the day grabbing dry wood around the nearby river and cleaning, drying, sawing, polishing, and assembling it so it would eventually look like the house in Figure I-1.

Figure I-1. *Bird house*

This book is not actually about building bird houses, but it is about making things by yourself or, indeed, coming up with ideas and shaping them into something useful for other people (instead of birds).

With the Internet of Things (IoT), you can use an existing development kit to empower you to create something in no time. In fact, big tech companies have provided so many of these development kits that it's hard to choose one and get started.

Those kits, while easy to use, create a dependency on the whole kit environment, such that working without one is hard. Moving away from them is almost impossible unless you redesign and rebuild everything you created from scratch.

Speaking of "scratch," a few years ago, Arduino came out. It's a small Italian-made embedded board that you can easily connect to sensors, rotors, motors, and wireless networks. You can use Arduino to easily write the custom logic needed to interact with all the development kits previously mentioned (Figure I-2).

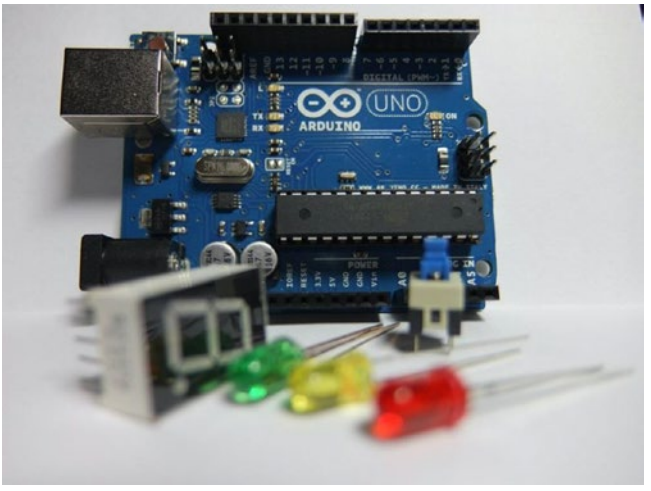

Figure I-2. *Arduino*

Arduino can be programmed in C, Scratch4Arduino, or S4A (http://s4a.cat), which is a modified version of the Scratch graphical environment that compiles code for Arduino and is shown in Figure I-3.

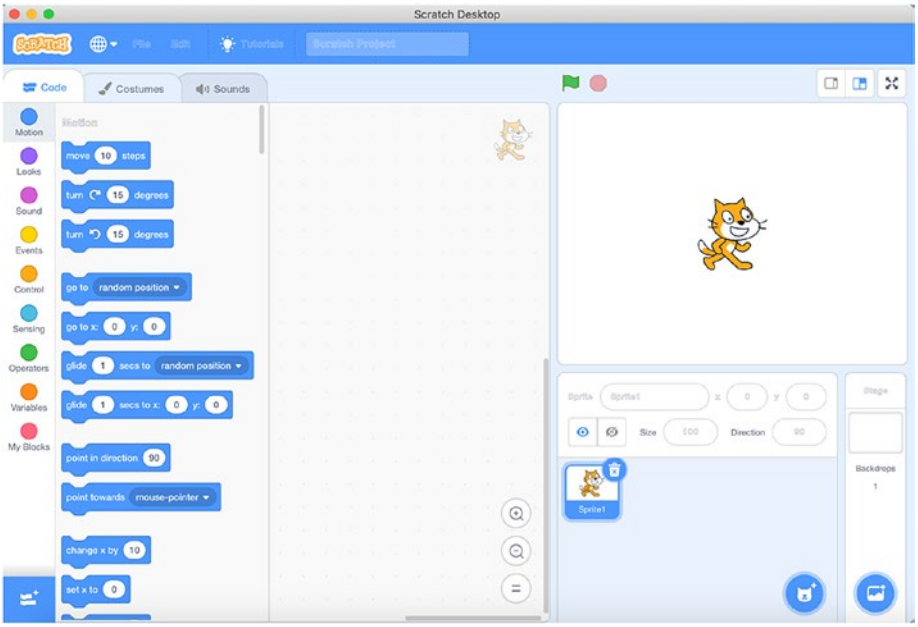

Figure I-3. *Scratch for Arduino*

In the world of dedicated hobbyists, Arduino was a breath of fresh air. It created an open source, free, unified ecosystem of libraries and connectors to plug things together. Users of the platform exchanged and competed for ideas in newly created labs within universities or renovated factory buildings.

I remember many years ago going to my friend's open source, open hardware lab in Shanghai where they were moving robots around and re-creating arcade game rooms with old-school games.

While that was fun for real geeks, others may have felt they were left out a bit. The learning curve was just a bit high, and the power of Arduino made it so that running anything but C code was not practical.

Then, from England came the Raspberry Pi, a full-blown computer that could be held in your hands. At around $40 USD, it was an expensive purchase for students, but it provided some massively capable hardware. Hobbyists bought tons of those little devices. The Raspberry Pi 2 was quite a big jump in specifications compared to the Arduino. The biggest difference for the end user, aside from the connectors, was that you could finally run a full-blown operating system and get some standardization between the different components. Raspberry 3, especially model B, had 1Gb of memory, meaning you could run Java applications nicely with good performance. The underlying ARM-based processor was still quite limited in real-time computation, though.

Fast-forward to 2019: the Raspberry 4, a small computer with the same size and tag price, is a beast. In fact, Japan, not an EU member yet, bought a license in September 2019 to officially make the device usable within its borders. The Japan government went through the effort mostly because the Wi-Fi and Bluetooth modules embedded in the device allow for military-level spying and should be monitored carefully. Just kidding. But some extra checks are still officially required on those small devices.

So, what's new with version 4? The main difference is power, both CPU and memory-wise; it also has various video-rendering upgrades. But isn't that what every salesperson will tell you to get you to buy their latest and greatest version?

Well, yes and no. While in previous versions you could run Java processes and enjoy quite a bit of speed, the Raspberry Pi 4 is fast enough and powerful enough that you can run real-time object detection on real-time video streams. That is a game-changer and the very reason for this book: it is now possible to have some video fun by simply using the Java wrappers for OpenCV.

OpenCV (https://opencv.org/), as you may know or not know, is one of the most used open source imaging libraries. Its license, Berkley Software Distribution (BSD), is permissive enough that you can deploy OpenCV in source or binary form anywhere. OpenCV is also widely available for the Android operating system, where similar Java bindings are also heavily in use. The main supported code for OpenCV is C, and there is support for every other programming language to access OpenCV by simply creating bindings around the OpenCV native binaries, so in theory the small overhead introduced by the Java wrapping layer should not cause additional problems of speed.

So, now we have a rather cheap device that can be coded using the Java language and therefore connect to most of the available libraries and still run real-time processing on videos.

In this book, we will perform some basic OpenCV analysis first, in Java, and then we will move on to analyzing real-time videos. The goal of this book is to get you started working on small devices and connecting the different pieces, such as the code, the language, the connectivity setup, together. Then we'll move on to the fun things.

One other novelty in this book is that instead of accessing the Raspberry Pi physically and running on top of it, we will go through a Secure Shell (SSH) connection to do our coding.

What does that mean, and why would we do that?

This means we want the editor where we write the Java code to be running somewhere other than on the Raspberry Pi itself. We want to do this to keep as many Raspberry Pi resources as possible doing the image and video processing for the device while running only the minimum amount of software.

At the same time, we want to have the ease of not recompiling everything, like is often necessary in Java, while having everything seem to be executing directly from our editor. This part is being made possible by the excellent work done by Microsoft in the Visual Studio Code Editor and its latest plug-in, the Remote SSH plugin (Figure I-4).

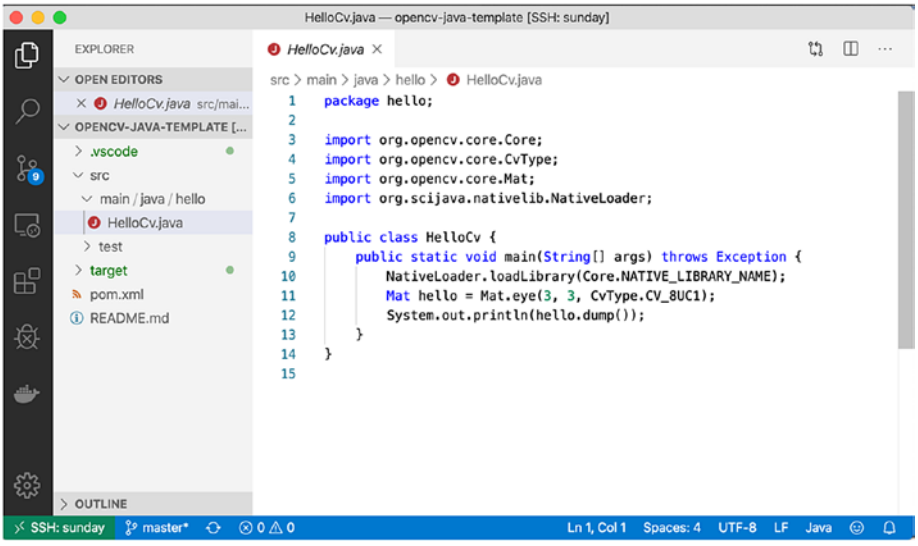

Figure I-4. *Visual Studio Code Remote SSH plugin*

Finally, we will look into connecting to home automation software, especially via voice.

Rhasspy (https://rhasspy.readthedocs.io) will be our open source software of choice. You will be using your newly acquired video streams knowledge to plug in voice commands and object detection. You eventually will be able to start automating your home by recognizing when your cat enters the kitchen, turning on the light for the kids when they wake up in the morning, or automatically starting the vacuuming robot when you leave the house.

This book is divided broadly into five chapters, with progressive challenges.

Chapter 1 focuses on getting the pieces working on a standard computer, writing the Java code to run basic image processing with OpenCV, and understanding how the different pieces of software fit together.

Chapter 2 takes you from processing images to analyzing video streams, while running feature-based and network-based object detection on pictures and videos.

Chapter 3 shows how to run code on the Raspberry Pi and will thus focus on doing the bare minimum needed to get the setup functional.

Chapter 4 helps you run on-board applications, focusing mostly on object detection in real-time video streams, both recorded and live.

Finally, Chapter 5 expands the Raspberry Pi setup to a home automation application, where we plug in voice commands to video streaming analyses.

CHAPTER 1

Getting Started

One of the goals of this book is to get you ready to perform real-time IoT imaging quickly, avoiding a lengthy installation process. Being ready quickly doesn't mean we are going to take any shortcuts, it means we will get the tooling part out of the way so we can focus on the creation process.

In this chapter, you'll run your first example.

Visual Studio Code Primer

You can't knock on opportunity's door and not be ready.

—Bruno Mars

The playground setup introduced in this book is fairly standard for people who are used to writing code. It also has a little bit of a "new kid on the block" feeling—only the cool kids use it. The *stack*, or the environment, we will use consists of the following:

- Visual Studio Code, a small but pluggable text editor
- The Java Development Kit and its runtime, so we can write Java code and run it
- A Java plugin to Visual Studio Code so that the editor understands the Java code and runs it

Basically, that's all.

N. Modrzyk, *Real-Time IoT Imaging with Deep Neural Networks*,
https://doi.org/10.1007/978-1-4842-5722-7_1

It does not really matter whether you're using Windows, Mac, or Linux. All the different pieces of XX are made to run anywhere. To install Visual Studio Code, you need to head to https://code.visualstudio.com/.

Click the download button on the top-right side of the page. After you click the button, you'll be presented with options for the different packages, one for each computer platform, as shown in Figure 1-1.

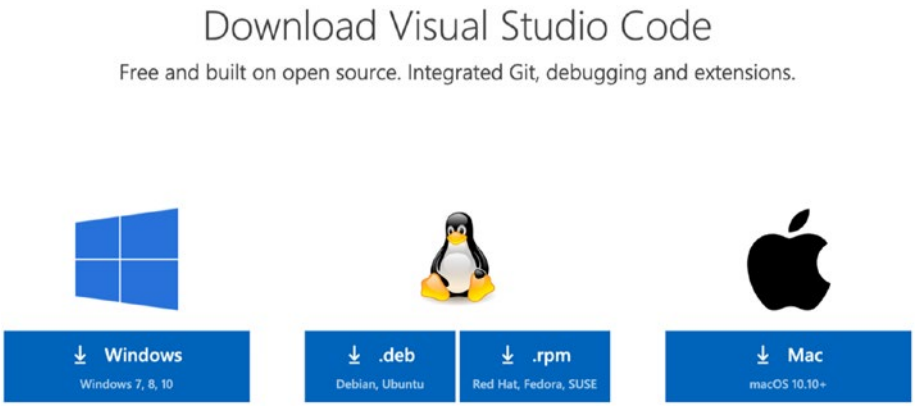

Figure 1-1. *Choosing your download*

At the time of writing, the current version is 1.40, but newer versions would only be better.

Running the installer and opening Visual Studio Code for the first time gives you a screen similar to the one in Figure 1-2.

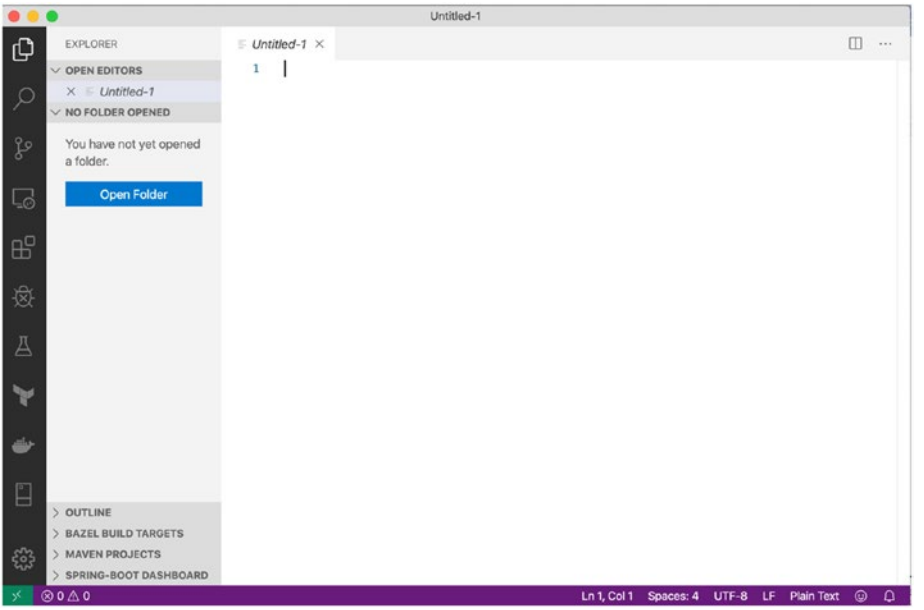

Figure 1-2. *Visual Studio Code*

The second step is to install Java, if it is not already installed. You can either head to the OpenJDK web site (`https://jdk.java.net/`) and download a zip file or go to the Oracle web site and download a ready-to-use installer for your machine (see Figure 1-3 and Figure 1-4). Specifically, you can go to `https://www.oracle.com/technetwork/java/javase/downloads/index.html`.

Figure 1-3. *Oracle Java download page*

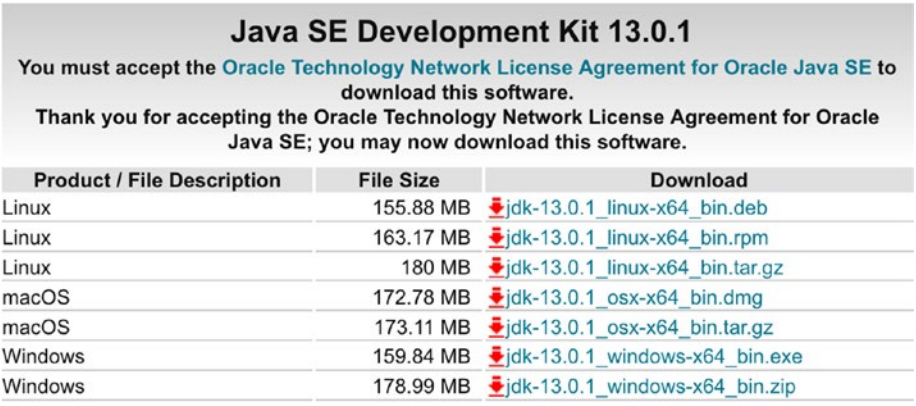

Figure 1-4. *Java download link*

Let the installer run to the end, and do not stop it even if it tries to open another installer while running. Java, being a development kit, does not come with a fancy application to check that it has been installed properly, so after the Java installation is finished, a quick way to check that things are in place is to open a terminal in Visual Studio Code and check the Java version.

You can press Ctrl+Shift+@ to open a terminal inside Visual Studio Code, or you can open it from the menu, as shown in Figure 1-5.

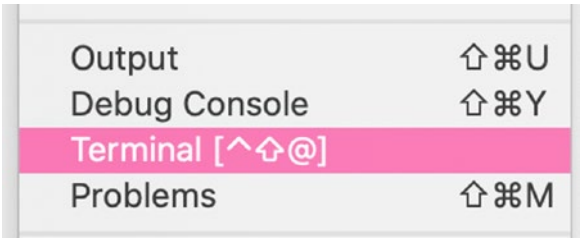

Figure 1-5. *Opening a terminal window from within Visual Studio Code*

This will pop up a small tab, usually at the bottom of the editor, as shown in Figure 1-6.

Figure 1-6. *The Terminal tab*

Then inside the terminal, type the following command:

```
java -version
```

Figure 1-7 shows the expected version output if you have installed the OpenJDK version.

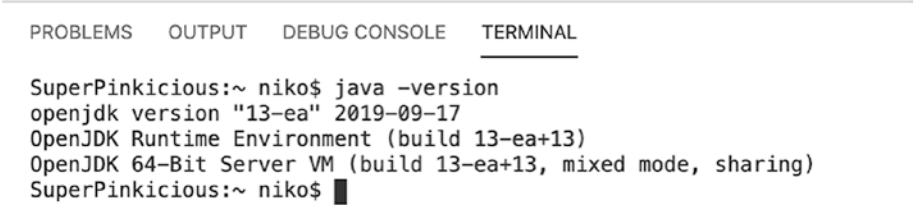

Figure 1-7. *Java version*

Note that the Java version itself is not important; any version between 8 and 14 is expected to work. Heck, let's be optimistic—things should also work for the foreseeable future.

We are almost there with all the setup, so bear with me for a few more seconds. In Visual Studio Code, everything you write will be handled as simple text. Basically, if you do not tell a computer what to do with text, it will not do anything.

Let's tell Visual Studio Code to understand our Java files by installing the Java plugin.

The left sidebar of the editor has a set of rather cute icons, and if you click the one at the bottom of Figure 1-8, you can access the Visual Studio Code Marketplace.

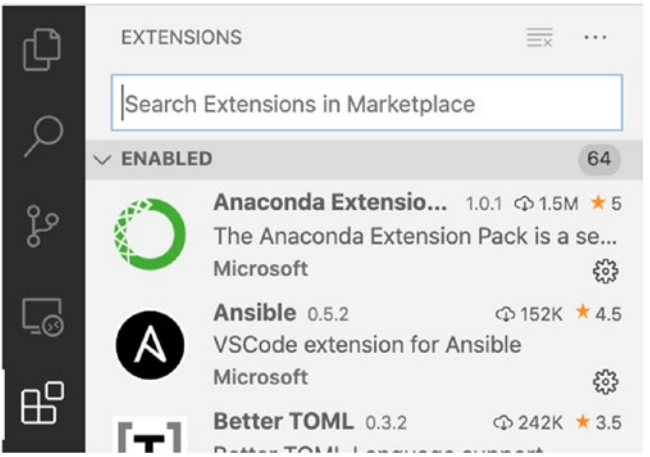

Figure 1-8. *Visual Studio Code Marketplace*

From here, you can install plug-ins in the editor to extend it in many different ways. We are going to make heavy use of plugins in this book, and build a setup which you can reuse and enhance for other situations.

For now, we want to install Java support from within the editor, which is done by searching for the Java extension using the search bar (see Figure 1-9).

Figure 1-9. *In search of the perfect Java plugin*

Selecting the plugin on the left gives an extensive description on a tab on the right, as shown in Figure 1-10.

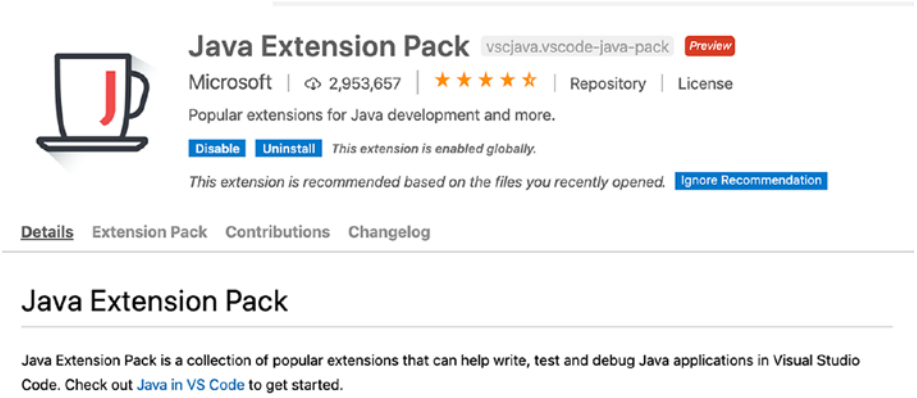

Figure 1-10. *Description of the Java plugin*

The Java extension is actually a collection of plugins, and there is a full entry and description available in the Visual Studio Code documentation, available here:

`https://code.visualstudio.com/docs/java/extensions`

Basically, the following plugins will be installed when you install the Java Extension Pack:

- Language Support for Java by Red Hat:
  https://marketplace.visualstudio.com/items?itemName=redhat.java
- Debugger for Java:
  https://marketplace.visualstudio.com/items?itemName=vscjava.vscode-java-debug
- Java Test Runner:
  https://marketplace.visualstudio.com/items?itemName=vscjava.vscode-java-test
- Maven for Java:
  https://marketplace.visualstudio.com/items?itemName=vscjava.vscode-maven
- Java Dependency Viewer:
  https://marketplace.visualstudio.com/items?itemName=vscjava.vscode-java-dependency
- Visual Studio IntelliCode:
  https://marketplace.visualstudio.com/items?itemName=VisualStudioExptTeam.vscodeintellicode

These plugins will allow you to perform all the tasks that are useful when writing code, such as autocompletion, code inspection, debugging, and more.

At this stage, after reloading the editor, you are essentially done with the installation steps. The fun development environment is ready; it hasn't cost you a penny, and almost no personal data has been leaked.

Running Your First Java Application

So, with the editor open, let's create a file, write some Java code in it, and see how to run the code directly from within the editor.

Running code in Java usually implies a compilation step, where the Java text file is converted to a form that the computer, and here the Java runtime, can execute.

In our setup, all those steps are being handled in the background by Visual Studio Code.

Our first example will just output a greeting, as is usually the case with any well-behaved computer program.

First, let's create a file named `First.java` and drop in some very basic Java code. The code will output some basic greeting text; this will also help us check that the setup is fully working.

Let's write the content in Listing 1-1 in the Java file.

Listing 1-1. Your First Java Program

```
public class First {
    public static void main(String[] args) {
        System.out.println("Hello Java");
    }
}
```

After entering this code (or just opening the sample provided), you will notice a few dynamic refreshes happening in the editor itself. The editor will highlight the code for you and associate the different Java features to your typing, as shown in Figure 1-11.

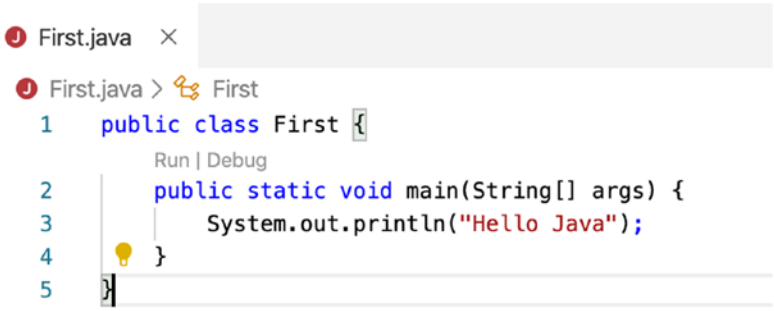

Figure 1-11. *Editor magic*

Autocompletion of the code is included for free and can be triggered by using the Tab key, as shown in Figure 1-12.

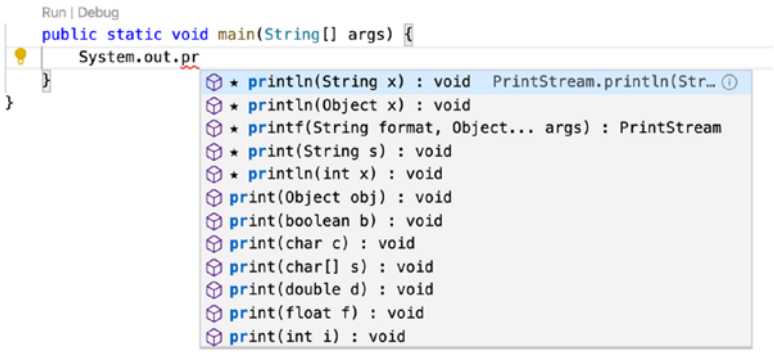

Figure 1-12. *Autocompletion*

You can also access the Java documentation when available, as shown in Figure 1-13, by using the keyboard combination Ctrl+spacebar.

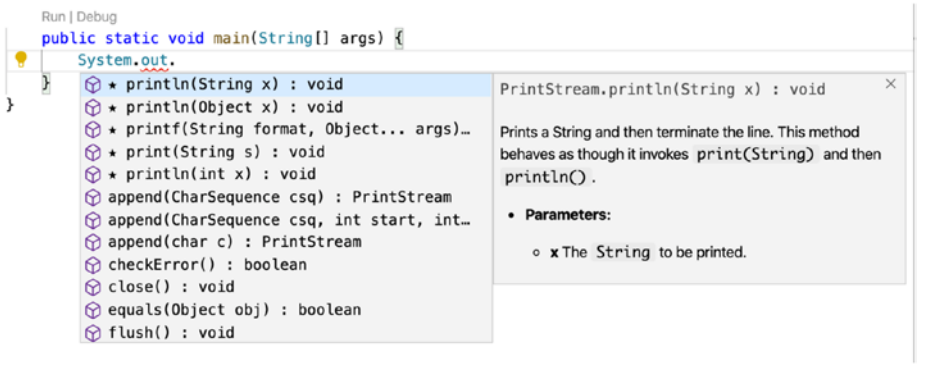

Figure 1-13. *Inline documentation*

You have also probably already noticed the Run and Debug links at the top of the main Java method, as shown in Figure 1-14. You can click these links or trigger them with keyboard shortcuts. Press F5 for debugging and press Ctrl+F5 to run the code.

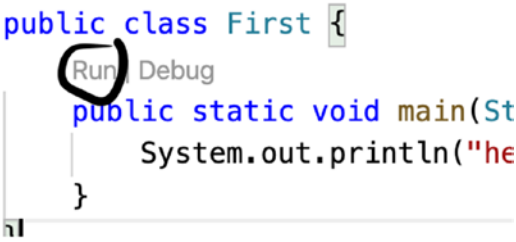

Figure 1-14. *Run and Debug links*

Clicking the Run button will, as you would expect, trigger the execution of the code inside the editor and display some text in the Terminal tab we opened earlier (Figure 1-15).

Figure 1-15. *Running some Java code for real*

Importing Core Java Packages

You will need to have external classes loaded eventually, and this can be done for you using the editor's import function. This seems trivial at this stage, but when using OpenCV in Java, some of the packages are hard to figure out, especially when a lot of the code samples do not bother writing the imports for conciseness.

First, let's write a piece of code that displays the current time. The easiest way to do this is to simply use a `Date` object, as shown in Listing 1-2.

Listing 1-2. Before Imports

```
public class First {
    public static void main(String[] args) {
        Date d = new Date();
        System.out.println(String.format("hello java. it is
        %s", d));
    }
}
```

In the print version of this book, everything looks OK, but in Visual Studio the editor rightfully highlights the parts of the code that cannot be compiled properly. In Figure 1-16, notice how the code cannot be understood by the compiler and is being underlined.

Figure 1-16. *Missing Date class import*

To import the class, you can click the small light bulb next to the `Date` word that is underlined in Red (Figure 1-17).

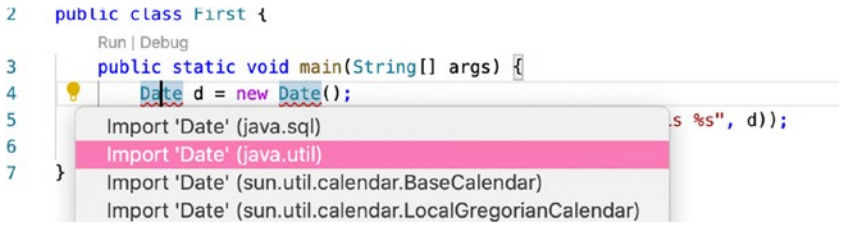

Figure 1-17. *Importing using the light bulb*

Alternatively, you can trigger the Visual Studio command menu by pressing Ctrl+Shift+P or Command+Shift+P and start typing **Organize Imports** in the bar (see Figure 1-18).

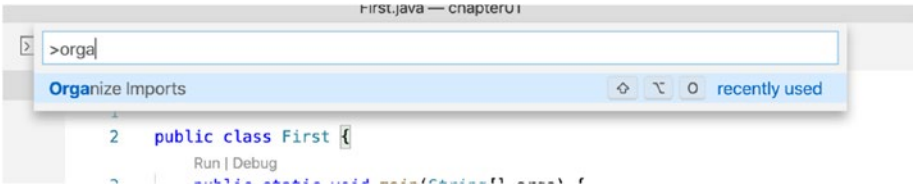

Figure 1-18. *Organizing imports*

You can select the exact Java class that needs to be imported in the current file, which here is `java.util.Date` (see Figure 1-19).

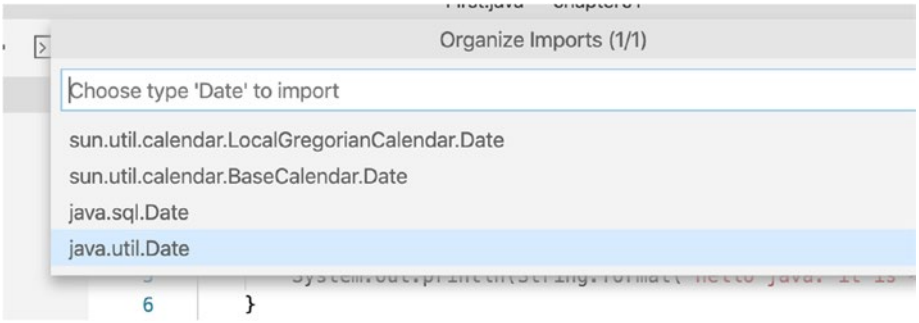

Figure 1-19. *Selecting the proper class*

Now that the code is fixed, it is ready to be executed again. Let's click the same Run link or use the F5 keyboard shortcut to start the execution.

The execution output is once again shown in the terminal, as shown in Figure 1-20.

Figure 1-20. *Wow. That code was written and executed in November 2019*

Debugging Lesson

This is your book, so you can skip this section on debugging and come back to it later if you want. I decided to add this section on debugging now, when we're working with some simple code, because getting this part done up front allows you to master the OpenCV code better and go through each OpenCV computation step in a nice and easy-to-understand way.

If you decided to keep reading this part, then great! We are going to spend just a little bit of time looking at how to debug Java code from within Visual Studio Code.

As you have seen, executing the code was quite simple from within the editor. In Java and with this setup, we also get a free debugger. You even get free debuggers in browsers nowadays when running JavaScript code from web pages.

Why is a debugger that important? A debugger allows you to stop the execution at any point in the code and then locally inspect it and act from the position of the stopped code.

That means you can start executing the code but also ask the runtime to stop wherever you want. Also, you do not have to choose where to stop code execution before starting it. Let's say you do video processing in real time, and at some stage, you would like to know why the code analyzing the camera feed is not finding objects as you would expect it to do. In that case, you can stop the code execution right in the frame capture loop and either dump the picture or rerun the analysis step-by-step so you can find out if the model is wrong or if the picture is still in an unexpected size or color mode.

Java itself does not have a really useful read-eval-print-loop (REPL). A REPL allows you to execute code line by line, and it works quite well with my favorite language, Clojure, and even with Kotlin and of course Python. Java does not make it easy to run and add things line by line, which can be almost forgiven now that debugging has been made so easy from Visual Studio Code.

Enough talking, let's see how to use the debugger to perform basic debugging tasks:

- Add a breakpoint
- Execute the code step-by-step
- Watch a variable
- Change a variable value

Add a Breakpoint

A breakpoint is, as its name implies, a point in time to take a break. It is shown as a red dot in the editor while moving the mouse on the left side of the line numbers (Figure 1-21).

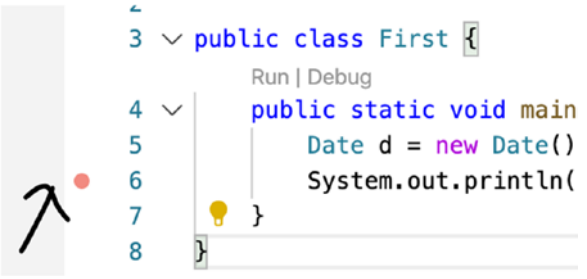

Figure 1-21. *Breakpoint*

To create a breakpoint, simply click the line number on which you want to stop the execution. Note that the breakpoint will stop the code before any of the code on that line is executed, so adding a breakpoint on line 5 will not show the `Date` object when the execution stops, but adding a dot on line 6 will, as shown in Figure 1-21.

Execute the Code Step-by-Step

Clicking the Debug link will start the execution in Debug mode and will stop it at the first breakpoint, the one you just added on line 6, and the editor will display a slightly different layout, as shown in Figure 1-22.

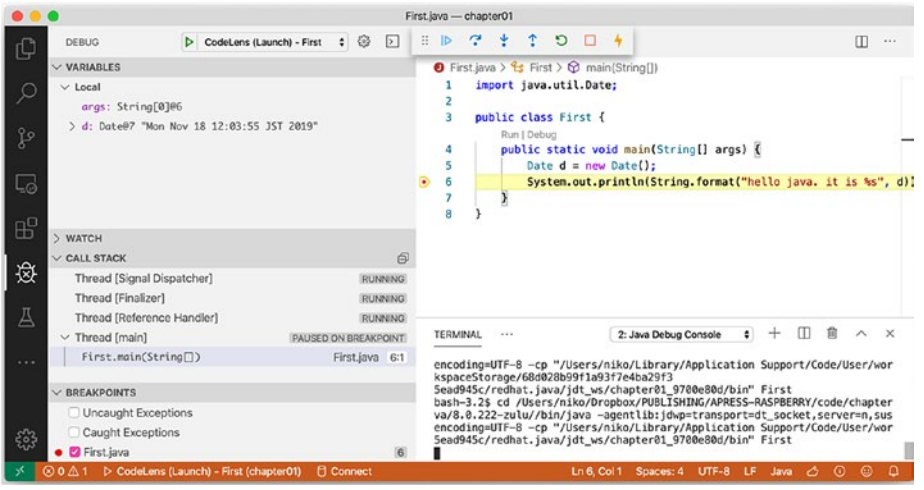

Figure 1-22. *Taking a break*

On the left side, you can see on the small Call Stack tab where the code execution has been stopped; here it's on line 6 in the class `First` in the main method (Figure 1-23).

Figure 1-23. *Where am I?*

You can also see the value of the variable `d`, which is a `Date` object, and you can also see the input parameters given to the program if there are any; here you see an empty array (Figure 1-24).

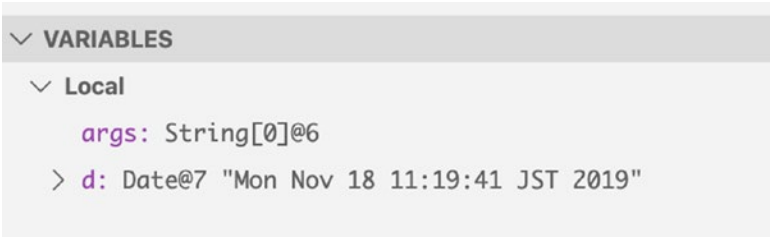

Figure 1-24. *Variables*

You can of course play with the different arrows and show the extended content, recursively, and fields of each variable one by one, as shown in Figure 1-25.

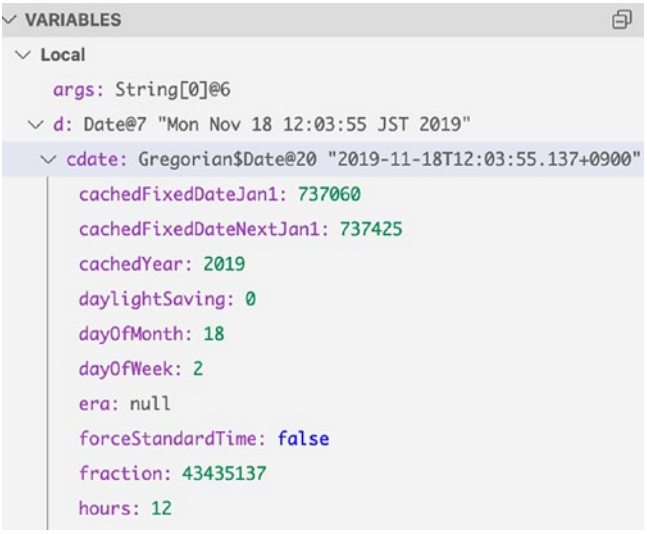

Figure 1-25. *Expanded variables*

In Figure 1-25 you can see for yourself that the `Date` object has a field named `cdate`, and you can see recursively all the fields of that field.

Resume Execution

To resume or finish the execution from the breakpoint, you have a few options that are available in the runtime bar located at the top of the editor while doing a debugging session (Figure 1-26).

Figure 1-26. *Execution commands*

From left to right you can perform actions explained in Table 1-1.

Table 1-1. *Debugging Actions*

Icon	Action	Comments
	Resume	This will tell the execution to go to either the end of the program or the next breakpoint without stopping.
	Step Over	This executes all the code on the current line, goes to the next line of code, and then stops again.
	Step Inside	This goes inside each block of execution, so in the previous example, on the first stop, the execution will go inside the format function and then stop.
	Step outside	This goes to the outer part of the code, so if you were currently in the format function, then this goes back to the original `main()` function.
	Restart	This stops all the debugging and restarts from scratch.
	Stop	This stops the code execution.
	Hot Code Replace	This updates the code in the runtime, with code from the editor. Real-time coding!

The action you will use most of the time is Step Over, which just executes the code line by line. This gives you a good overview of the variables that have been changed and the newly assigned variables and their values. See Figure 1-27.

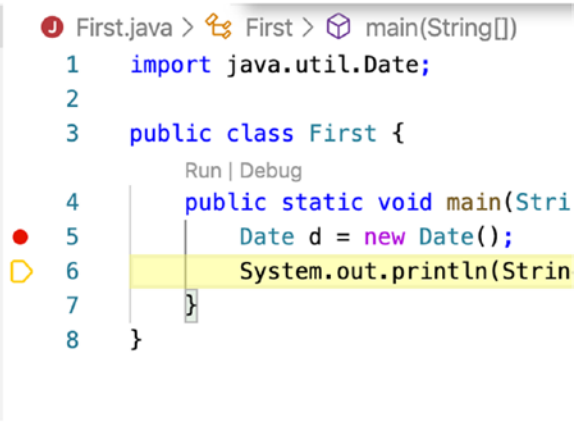

Figure 1-27. *Next line*

Going to the next line and then to the next one will eventually finish the program execution, ideally with the proper message printed in the terminal window.

Watch an Expression

You may have noticed the Watch tab on the left side of the editor and may already have wondered what it is for.

On the top-left Variables tab, you can see the direct values of objects and fields, but on the Watch tab you can add function calls on all the available objects, in what is called a *watch expression*. Watch expressions can be defined before the variable or the object is instantiated.

In object vision, and while doing direct rendering, you can use a watch expression to produce some side effects, create functions to, for example, turn the input picture from colored to black and white, or quickly create

a contour-only version of an image. You can, of course, do all this in real time, but in the coming chapters we will be working with low processing power and memory, so briefly pausing the code execution and using a one-off call to a watch expression is definitely better to keep the extra memory and computation to a minimum.

In this first chapter, we will add simple computations on the watch expression—one with no side effect directly returning a value based on the input parameters and one with a side effect printing to a log file.

We'll start from the code sample in Listing 1-3, which is simply looping over a value that is incremented in a `while` loop. We do not want to burn up the computer while doing this, so we'll add a short sleep call inside the loop to give some time to the execution loop.

Listing 1-3. The Incredible Loop Over an Integer

```
public class Second {
    public static void main(final String[] args) throws
    Exception {
        int i = 0;
        while (true) {
            i++;
            Thread.sleep(500);
        }
    }
}
```

We'll add the expression `i+2` in the watch expression and start the debugging session without actually setting a breakpoint yet.

After waiting some time, let's click to add a breakpoint on line 6 and see the editor stop as if we had set the breakpoint before starting the execution.

The Visual Studio Code layout will look like Figure 1-28.

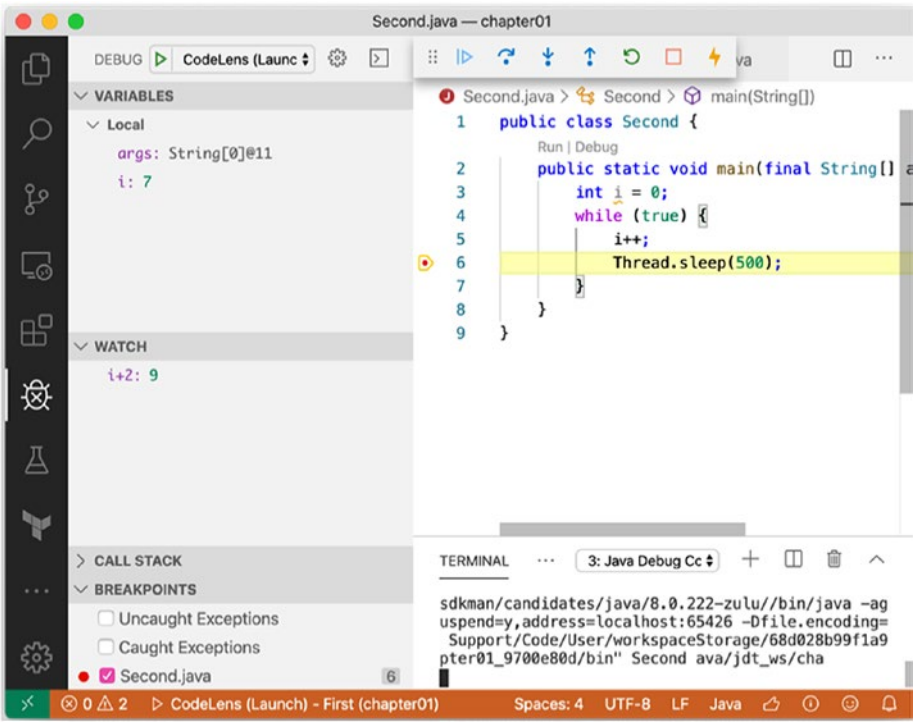

Figure 1-28. *The incredible loop in Debug mode*

By clicking and creating a breakpoint, you have just stopped the execution, and the debug layout appears. As you can see, the variable `i` already has a value of 7, and the watch expression of `i+2` is showing the correct value of 9.

Nice.

Now for some exercises that you can try on your own. Try implementing two functions and use them inside two watch expressions.

- One function will just do a basic computation on the variable `i` and display it.
- Another function will not return anything but append a value to an external file.

This should take around 5–10 minutes of your time. Good luck! Once you're finished, you can read and compare your code with Listing 1-4.

Listing 1-4. Watched Expressions with Side Effects, and No Side Effect

```
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.Writer;

public class SecondFinal {

    static int myfunction1(final int i) {
        return 3 + i + 2;
    }

    final static String filePath = "my.log";

    static void myfunction2(final int i) {
        try (Writer writer = new BufferedWriter(new
        FileWriter(filePath))) {
            writer.write(String.format(">> %d\n", i));
        } catch (Exception e) {

        }
    }

    public static void main(final String[] args) throws
    Exception {
        int i = 0;
        while (true) {
            i++;
            Thread.sleep(500);
        }
    }
}
```

The different watch expressions are showing as expected on the left side in the debugging layout of Visual Studio Code (Figure 1-29).

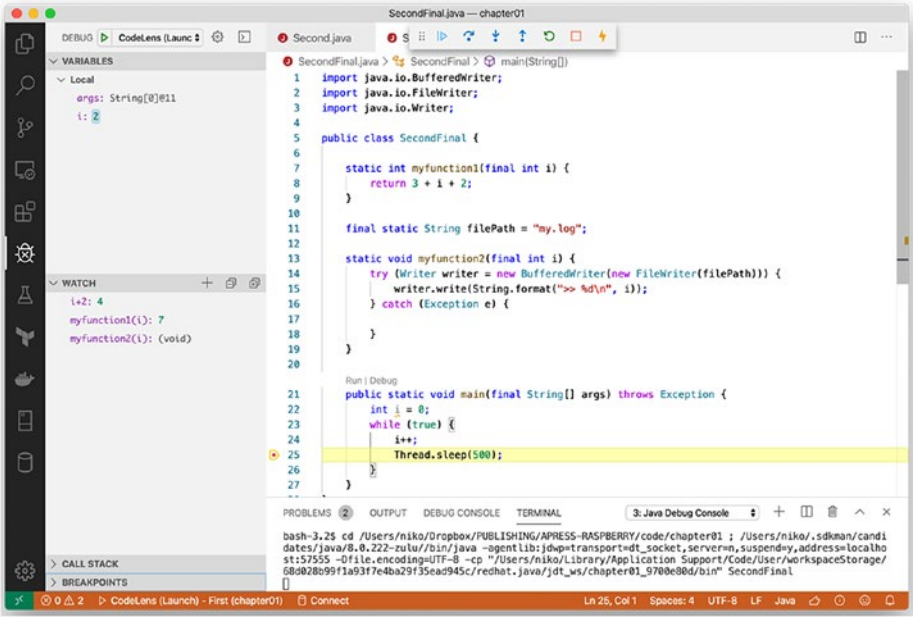

Figure 1-29. *Watch expression*

That wasn't so hard, was it? Now the good news is that you can reuse those techniques when working with the imaging library OpenCV. For example, you can do the following:

- Output the width and height, as well as the number of color channels of an image
- Output the Hough lines of an image in an external file
- Output the number of detected faces in an image

Remember, you can add watch expressions after starting the code execution in Debug mode, so add them any time you need them.

Change a Variable Value

One last thing to know about debugging in Visual Studio Code before being done is that you can change the value of a variable in real time, straight from the Variables tab.

Now that is some superpower!

You can do this by double-clicking the variable value you would like to change, as shown in Figure 1-30.

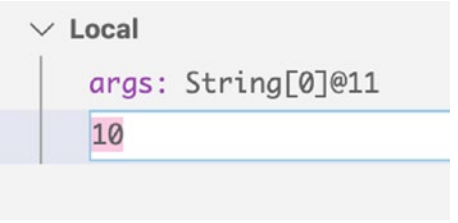

Figure 1-30. *Updating a variable value*

Observe how the related watch expressions are being recomputed for you in Figure 1-31.

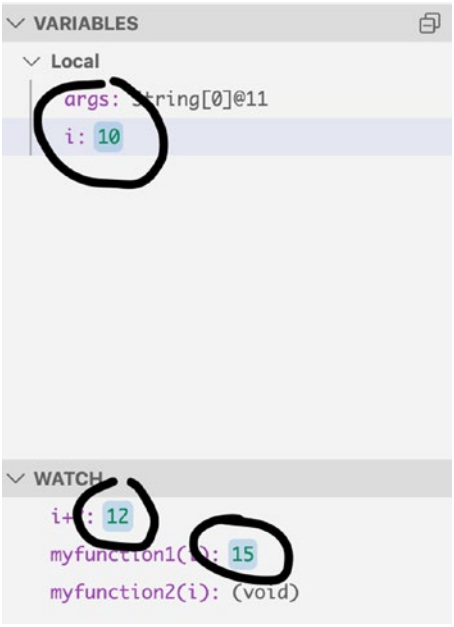

Figure 1-31. *You can see the related watch values being updated*

Sweet. In the context of imaging and neural networks, you can directly update alpha, gamma, contrast, etc., directly and have the recomputed images or detection happen in real time.

Wrapping Things Up

We spent a bit of time learning about debugging, so picture how much can be done by reusing the techniques you have just seen in the context of imaging and object detection.

For example, Figure 1-32 shows how a well-designed debugging layout can be used as a full user interface to update values for a sepia conversion.

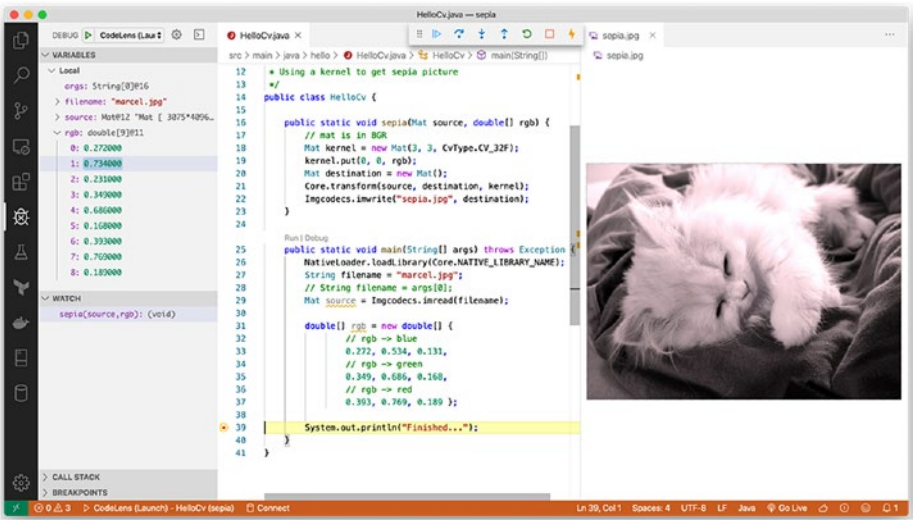

Figure 1-32. *Marcel the cat in sepia*

In this layout, you can see the following:

- The array of values used to recompute the sepia has been extracted in the main code as a double array.
- While debugging, the `rgb` array and its values are available on the Variables tab and can be updated at will.
- The `sepia` function here is used only as a debugging function, not in the main code.
- That `sepia` function outputs a picture in a file named `sepia.jpg`.

- A watch expression is added, making a simple call to the `sepia` function defined earlier.
- The `sepia.jpg` picture is opened within Visual Studio Code, and any update of values in the `rgb` array will output a new picture in real time.

Now that you have an idea of how things can be worked on from within Visual Code Studio, let's move on to some OpenCV fun.

CHAPTER 2

Object Detection in Video Streams

Most of the available how-to guides for working with OpenCV in Java require you to have an insane amount of knowledge before getting started. The good news for you is that with what you have learned up to now with this book, you can get started with OpenCV in Java in seconds.

Going Sepia: OpenCV Java Primer

> *Choosing sepia is all to do with trying to make the image look romantic and idealistic. It's sort of a soft version of propaganda.*
>
> —Martin Paar[1]

In this section, you will be introduced to some basic OpenCV concepts. You'll learn how to add the files needed for working with OpenCV in Java, and you'll work on a few simple OpenCV applications, like smoothing, blurring images, or indeed turning them into sepia images.

[1] https://www.theguardian.com/artanddesign/2009/apr/15/madonna-photo-malawi-adoption

N. Modrzyk, *Real-Time IoT Imaging with Deep Neural Networks*,
https://doi.org/10.1007/978-1-4842-5722-7_2

A Few Files to Make Things Easier…

Visual Studio Code is clever, but to understand the code related to the OpenCV library, it needs some instructions. Those instructions are included in a project's metadata file, which specifies what library and what version to include to run the code.

A template for OpenCV/Java has been prepared for you, so to get started, you can simply clone the project template found here:

```
git clone git@github.com:hellonico/opencv-java-template.git
```

Or you can use the zip file, found here:

```
https://github.com/hellonico/opencv-java-template/archive/
master.zip
```

This will give you the minimum set of files required, as shown here:

```
.
├── pom.xml
└── src
    ├── main
    │   └── java
    │       └── hello
    │           └── HelloCv.java
    └── test
        └── java
            └── hello
                └── AppTest.java
```

There are seven directories and three files. Since this setup can be autogenerated, let's focus on the template content, listed here:

- `HelloCv.java`, which contains the main Java code
- `AppTest.java`, which contains the bare minimum Java test file

- `pom.xml`, which is a project descriptor that Visual Studio Code can use to pull in external Java dependencies

You can open that top folder from the project template from within Visual Studio Code, which gives you a familiar view, as shown in Figure 2-1.

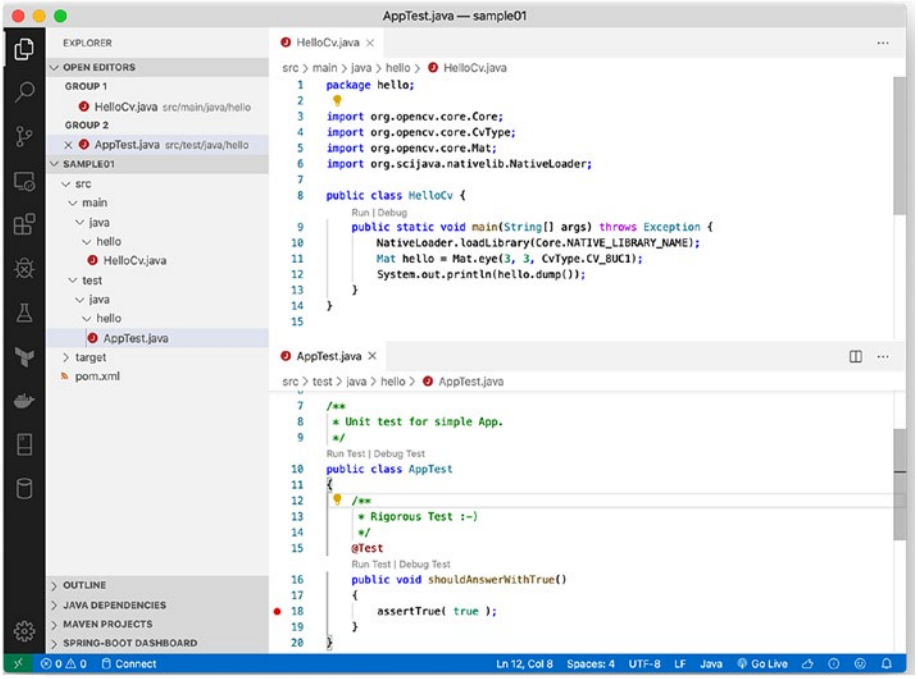

Figure 2-1. *Project layout*

Figure 2-1 shows the two Java files, so you can see the content of each; in addition, the project layout has been expanded so all the files are listed on the left.

This view should look familiar to you from the previous chapter, and you can immediately try running the `HelloCv.java` code using the Run or Debug link at the top of the main function.

The output in the terminal should look like the following code, which is an OpenCV `Mat` object. To put it simply, it's a matrix of size 3×3, with 1 on the top-left, bottom-right diagonal.

```
[  1,   0,   0;
   0,   1,   0;
   0,   0,   1]
```

Listing 2-1 probably looks familiar to users of OpenCV.

Listing 2-1. First OpenCV Mat

```
package hello;

import org.opencv.core.Core;
import org.opencv.core.CvType;
import org.opencv.core.Mat;
import org.scijava.nativelib.NativeLoader;

public class HelloCv {
    public static void main(String[] args) throws Exception {
        NativeLoader.loadLibrary(Core.NATIVE_LIBRARY_NAME);
        Mat hello = Mat.eye(3, 3, CvType.CV_8UC1);
        System.out.println(hello.dump());
    }
}
```

Let's go through the code line by line to understand what is happening behind the scenes.

First, we use the `NativeLoader` class to load a native library, as shown here:

```
NativeLoader.loadLibrary(Core.NATIVE_LIBRARY_NAME);
```

This step is required because OpenCV is not a Java library; it is a binary compiled especially for your environment.

Usually in Java, you load a native library with `System.loadLibrary`, but that requires these two things:

- You have a library compiled for your machine.
- The library is placed somewhere where the Java runtime on your computer can find it.

In this book, we will rely on some packaging magic, where the library is downloaded and loaded automatically for you and where, along the way, the library is placed in a location that `NativeLoader` handles for you. So, there's nothing to do here except to add that one-liner to the start of each of your OpenCV programs; it's best to put it at the top of the main function.

Now, let's move on to the second line of the program, as shown here:

```
Mat hello = Mat.eye(3, 3, CvType.CV_8UC1);
```

That second line creates a `Mat` object. A `Mat` object is, as explained a few seconds ago, a matrix. All image manipulation, all video handling, and all the networking are done using that `Mat` object. It wouldn't be too much of a stretch to say that the main thing OpenCV is programmatically doing is proposing a great programming interface to work with optimized matrices, in other words, this `Mat` object.

In Visual Studio Code, you can already access operations directly using the autocompletion feature, as shown in Figure 2-2.

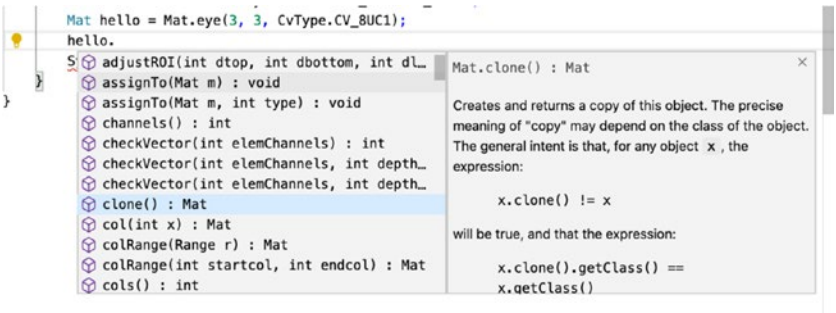

Figure 2-2. *Inline documentation for the OpenCV Mat object*

Within the one-liner, you create a 3×3 matrix, and the internal type of each element in the matrix is of type `CV_8UC1`. You can think of `8U` as 8 bits unsigned, and you can think of `C1` as channel 1, which basically means one integer per cell of the matrix.

The naming scheme for types is as follows:

```
CV_<bit-depth>{U|S|F}C<number_of_channels>
```

Understanding the types used in `Mat` is important, so be sure to review the most common types shown in Table 2-1.

Table 2-1. *OpenCV Types for OpenCV Mat Object*

Name	Type	Bytes	Signed	Range
CV_8U	Integer	1	No	0 to 255
CV_8S	Integer	1	Yes	−128 to 127
CV_16S	Integer	2	Yes	−32768 to 32767
CV_16U	Integer	2	No	0 to 65535
CV_32S	Integer	4	Yes	−2147483648 to 2147483647
CV_16F	Float	2	Yes	-6.10×10^{-5} to 6.55×10^{4}
CV_32F	Float	4	Yes	-1.17×10^{-38} to 3.40×10^{38}

The number of channels in a `Mat` is also important because each pixel in an image can be described by a combination of multiple values. For example, in RGB (the most common channel combination in images), there are three integer values between 0 and 256 per pixel. One value is for red, one is for green, and one is for blue.

See for yourself in Listing 2-2, where we show a 50×50 blue `Mat`.

Listing 2-2. Some Blue…

```
public class HelloCv2 {
    public static void main(String[] args) throws Exception {
        NativeLoader.loadLibrary(Core.NATIVE_LIBRARY_NAME);
        Mat hello = Mat.eye(50, 50, CvType.CV_8UC3);
        hello.setTo(new Scalar(190, 119, 0));
        HighGui.imshow("rgb", hello);
        HighGui.waitKey();
        System.exit(0);
    }
}
```

Executing the previous code will give you this 50×50 `Mat`, where each pixel is made of three channels, meaning three values, but note that in OpenCV, RGB values are reversed by design (why, oh, why?), so it is actually BGR. The value for blue comes first in the code.

So, before the value of all the pixels, set using the function `setTo`, is B:190, G:119, Red:0, which gives the nice ocean sea blue shown in Figure 2-3.

Figure 2-3. *Sea blue*

Operations on `Mat` objects are usually done using the `org.opencv.core.Core` class. For example, adding two `Mat` objects is done using the `Core.add` function, two input objects, two `Mat` objects, and a receiver for the result of the addition, which is another `Mat`.

To understand this, we can use a 1×1 `Mat` object. You can see that when adding two `Mat` objects together, you get a resulting `Mat` where each cell value is the result of the addition of the same location in the first `Mat` and the second `Mat`. Finally, the result is stored in `dest Mat`, as shown in Listing 2-3.

Listing 2-3. Adding Two Mat Objects Together

```
Mat hello = Mat.eye(1, 1, CvType.CV_8UC3);
hello.setTo(new Scalar(190, 119, 0));

Mat hello2 = Mat.eye(1, 1, CvType.CV_8UC3);
hello2.setTo(new Scalar(0, 0, 100));

Mat dest = new Mat();
Core.add(hello, hello2, dest);

System.out.println(dest.dump());
```

The result is shown in the following 1×1 `dest Mat`. Don't get confused; this is only one pixel with three values, one per channel.

```
[190, 119, 100]
```

Now, we'll let you practice a little bit and perform the same operation on a 50×50 `Mat` and show the result in a window again with HighGui.

The result should look like Figure 2-4.

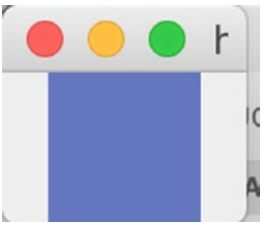

Figure 2-4. *Adding two colored Mats*

OpenCV Primer 2: Loading, Resizing, and Adding Pictures

You've seen how to add super small `Mat` objects together, but let's see how things work when adding two pictures, specifically, two big `Mat` objects, together.

My sister has a beautiful cat, named Marcel, and she very nicely agreed to provide a few pictures of Marcel to use for this book. Figure 2-5 shows Marcel taking a nap.

Figure 2-5. *Marcel at work*

I've never seen Marcel at the beach, but I would like to have a shot of him near the ocean.

In OpenCV, we can achieve this by taking Marcel's picture and adding it to a picture of the beach (Figure 2-6).

Figure 2-6. *The beach where Marcel is heading to*

Adding these two pictures together properly is going to take just a bit of work the first time, but it will be useful to understand how OpenCV works.

Simple Addition

Loading a picture is done using the `imread` function from the `Imgcodec` class. Calling `imread` with a path returns a `Mat` object, the same kind of object you have been working with so far. Adding the two `Mats` (the Marcel `Mat` and the beach `Mat`) is as simple as using the same `Core.add` function as used earlier, as shown in Listing 2-4.

Listing 2-4. First Try at Adding Two Mats

```
Mat marcel = Imgcodecs.imread("marcel.jpg");
Mat beach = Imgcodecs.imread("beach.jpeg");
Mat dest = new Mat();
Core.add(marcel, beach, dest);
```

However, running the code for the first time returns an obscure error message, as shown here:

```
Exception in thread "main" CvException [org.opencv.core.
CvException: cv::Exception: OpenCV(4.1.1) /Users/niko/
origami-land/opencv-build/opencv/modules/core/src/arithm.
cpp:663: error: (-209:Sizes of input arguments do not match)
The operation is neither 'array op array' (where arrays have
the same size and the same number of channels), nor 'array op
scalar', nor 'scalar op array' in function 'arithm_op']
        at org.opencv.core.Core.add_2(Native Method)
        at org.opencv.core.Core.add(Core.java:1926)
        at hello.HelloCv4.simplyAdd(HelloCv4.java:16)
        at hello.HelloCv4.main(HelloCv4.java:41)
```

Don't get scared; let's quickly check with the debugger what is happening and add a breakpoint at the proper location; see the `Mat` objects on the Variables tab, as shown in Figure 2-7.

Figure 2-7. *Debugging Mats*

Before adding a breakpoint, we can see that the Marcel `Mat` is indeed 2304×1728, while the beach `Mat` is smaller at 333×500, so we definitely need to resize the second `Mat` to match that of Marcel. If we do not perform

this resizing step, OpenCV does not know how to compute the result of the `add` function and gives the error message shown earlier.

Our second try produces the code in Listing 2-5, where we use the `resize` function of class `Imgproc` to change the beach `Mat` to be the same size as the Marcel `Mat`.

Listing 2-5. Resizing

```
Mat marcel = Imgcodecs.imread("marcel.jpg");
Mat beach = Imgcodecs.imread("beach.jpeg");
Mat dest = new Mat();
Imgproc.resize(beach, dest, marcel.size());
Core.add(marcel, dest, dest);
Imgcodecs.imwrite("marcelOnTheBeach.jpg", dest);
```

Running this code, the output image looks similar to Figure 2-8.

Figure 2-8. *White, white, very white Marcel at the beach*

Hmm. It's certainly better because the program runs to the end without error, but something is not quite right. The output picture gives an impression of being over-exposed, and looks way too bright. And indeed, if you look, most of the output image pixels have maximum RGB values of 255,255,255, which is the RGB value for white.

We should do an addition in a way that keeps the feeling of each `Mat` but still not goes over the maximum 255,255,255 value.

Weighted Addition

Preserving meaningful values in `Mat` objects is certainly something that OpenCV can do. The `Core` class comes with a weighted version of the `add` function that is conveniently named `addWeighted`. What `addWeighted` does is multiply values of each `Mat` object, each by a different scaling factor. It's even possible to adjust the result value with a parameter called `gamma`.

The function `addWeighted` takes no less than six parameters; let's review them one by one.

- The input `image1`
- `alpha`, the factor to apply to `image1`'s pixel values
- The input `image2`
- `beta`, the factor to apply to `image2`'s pixel values
- `gamma`, the value to add to the sum
- The destination `Mat`

So, to compute each pixel of the result `Mat` object, `addWeighted` does the following:

```
image1 x alpha + image2 x beta + gamma = dest
```

Replacing `Core.add` with `Core.addWeighted` and using some meaningful parameter values, we now get the code shown in Listing 2-6.

Listing 2-6. Marcel Goes to the Beach

```
Mat marcel = Imgcodecs.imread("marcel.jpg");
Mat beach = Imgcodecs.imread("beach.jpeg");
Mat dest = new Mat();
Imgproc.resize(beach, dest, marcel.size());
Core.addWeighted(marcel, 0.8, dest, 0.2, 0.5, dest);
```

The output of the program execution gives something much more useful, as shown in Figure 2-9.

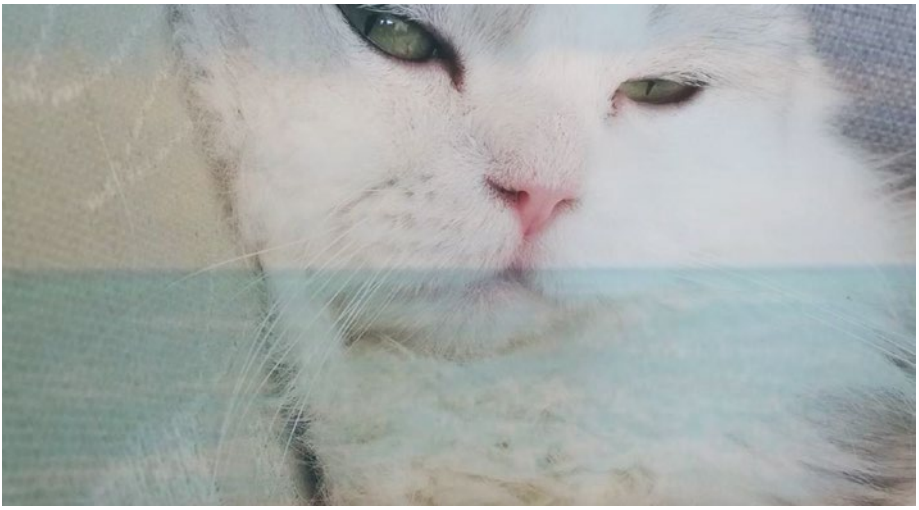

Figure 2-9. *addWeighted. Marcel can finally relax at the beach*

Back to Sepia

At this stage, you know enough about `Mat` computation in OpenCV that we can go back to the sepia example presented a few pages earlier.

In the sepia sample, we were creating a kernel, another `Mat` object, to use with the `Core.transform` function.

`Core.transform` applies a transformation to the input image where

- The number of channels of each pixel in the output equals the number of rows of the kernel.
- The number of columns in the kernel must be equal to the number of channels of the input, or +1.
- The output value of each pixel is a matrix transformation, and the matrix transformation of every element of the array `src` stores the results in `dst`, where `dst[ I ] = m x src[ I ]`.

See Table 2-2 for examples of how this works.

Table 2-2. *Core.transform Samples*

Source	Kernel	Output	Computation
[2 3]	[5]	[10 15]	10 = 2 × 5 15 = 3 × 5
[2 3]	[5 1]	[11 16]	11 = 2 × 5 + 1 16 = 3 × 5 + 1
[2 3]	[5 10]	[(10, 20) (15, 30)]	(10 = 2 × 5 20 = 2 × 10) (15 = 3 × 5 30 = 3 × 10)
[2]	[1 2 3 4]	[(4 10)]	4 = 2 × 1 + 2 10 = 3 × 2 + 4
[2 3]	[1 2 3 4]	[(4 10) (5 13)]	4 = 2 × 1 + 2 10 = 2 × 3 + 4 5 = 3 × 1 + 2 13 = 3 × 3 + 4

(*continued*)

Table 2-2. (*continued*)

Source	Kernel	Output	Computation
[2]	[1 2 3]	[(2 4 6)]	2 = 2 × 1 4 = 2 × 2 6 = 2 × 3
[2]	[1 2 3 4 5 6]	[(4 10 16)]	4 = 2 × 2 + 1 10 = 2 × 3 + 4 16 = 2 × 5 + 6
[(190 119 10)]	[0.5 0.2 0.3]	[122]	122 = 190 × 0.5 + 119 × 0.2 + 10 × 0.3

The OpenCV `Core.transform` function can be used in many situations and is also at the root of turning a color image into a sepia one.

Let's see whether Marcel can be turned into sepia. We apply a straightforward transform with a 3×3 kernel, where each value is computed as was just explained.

The simplest and most famous sepia transform uses a kernel with the following values:

```
[      0.272 0.534 0.131
       0.349 0.686 0.168
       0.393 0.769 0.189]
```

So, for each resulting pixel, the value for blue is as follows:

```
0.272 x Source Blue + 0.534 x Source Green + 0.131 x Source Red
```

The target value for green is as follows:

```
0.349 x Source Blue + 0.686 x Source Green + 0.168 x Source Red
```

Finally, the value for red is as follows:

```
0.393 x Source Blue + 0.769 x Source Green + 0.189 x Source Red.
```

As, you can see, the values for red are largely pinned down, with each multipliers around 0.1, and green has the most impact on the pixel values for the resulting `Mat` objects.

The input image of Marcel this time is shown in Figure 2-10.

Figure 2-10. *Marcel does not know he's going to turn into sepia soon*

To turn Marcel into sepia, first we read a picture using `imread`, and then we apply the sepia kernel, as shown in Listing 2-7.

Listing 2-7. Sepia Marcel

```
Mat marcel = Imgcodecs.imread("marcel.jpg");
Mat sepiaKernel = new Mat(3, 3, CvType.CV_32F);
sepiaKernel.put(0, 0,
        // bgr -> blue
        0.272, 0.534, 0.131,
        // bgr -> green
```

```
        0.349, 0.686, 0.168,
        // bgr -> red
        0.393, 0.769, 0.189);

Mat destination = new Mat();
Core.transform(marcel, destination, sepiaKernel);
Imgcodecs.imwrite("sepia.jpg", destination);
```

As you can see, the BGR output of each pixel is computed from the value of each channel of the same pixel in the input.

After we run the code, Visual Studio Code outputs the picture in a file named `sepia.jpg`, as in Figure 2-11.

Figure 2-11. *Marcel turned into sepia*

That was a bit of yellowish sepia. What if we need more red? Go ahead and try it.

Increasing the red means increasing the values of the R channel, which is the third row of the kernel matrix.

Upping the value of `kernel[3,3]` from 0.189 to 0.589 gives more power to the red in the input to the red in the output. So, with the following kernel, Marcel turns into something redder, as shown in Figure 2-12:

```
// bgr -> blue
0.272, 0.534, 0.131,
// bgr -> green
0.349, 0.686, 0.168,
// bgr -> red
0.393, 0.769, 0.589
```

Figure 2-12. *Red sepia version of Marcel*

You can play around and try a few other values for the kernel so that the sepia turns greener or bluer…and of course try it with your own cat if you have one. But, can you beat the cuteness of Marcel?

Finding Marcel: Detecting Objects Primer

Let's talk about how to detect objects using Marcel the cat.

Finding Cat Faces in Pictures Using a Classifier

Before processing speeds got faster and neural networks were making the front pages of all the IT magazines and books, OpenCV implemented a way to use classifiers to detect objects within pictures.

Classifiers are trained with only a few pictures, using a method that feeds the classifier with training pictures and the features you want the classifier to detect during the detection phase.

The three main types of classifiers in OpenCV, named depending on the type of features they are extracting from the input images during the training phase, are as follows:

- Haar features
- Hog features
- Local binary pattern (LBP) features

The OpenCV documentation on cascade classifiers (`https://docs.opencv.org/4.1.1/db/d28/tutorial_cascade_classifier.html`) is full of extended details when you want to get some more background information. For now, the goal here is not to repeat this freely available documentation.

What Is a Feature?

Features are key points extracted from a set of digital pictures used for training, something that can be reused for matching on totally new digital inputs.

For example, an ORB classifier, nicely explained in "Object recognition with ORB and its Implementation on FPGA" (http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.405.9932&rep=rep1&type=pdf) is a very fast binary descriptor based on BRIEF. The other famous feature-based algorithms are Scale Invariant Feature Transform (SIFT) and Speed Up Robust Features (SURF); all these are looking to match features for a set of known images (what we are looking for) on new inputs.

Listing 2-8 shows a super-brief example of how feature extraction is done. Here we are using an ORB detector to find the key points of a picture.

Listing 2-8. ORB Feature Extraction

```
Mat src = Imgcodecs.imread("marcel2.jpg", Imgcodecs.IMREAD_
GRAYSCALE);

ORB detector = ORB.create();
MatOfKeyPoint keypoints = new MatOfKeyPoint();
detector.detect(src, keypoints);

Mat target = src.clone();
target.setTo(new Scalar(255, 255, 255));

Features2d.drawKeypoints(target, keypoints, target);
Imgcodecs.imwrite("orb.png", target);
```

Basically, the features are extracted into a set of points. We don't do that usually, but drawing the key points here gives something like Figure 2-13.

Figure 2-13. *Extracting ORB features on Marcel*

Cascade classifiers are called that because they internally have a set of different classifiers, each of them getting a deeper, more detailed chance of a match at a cost of speed. So, the first classifier will be very fast and get a positive or a negative, and if positive, it will pass the task on to the next classifier for some more advanced processing, and so on.

Haar-based classifiers, as proposed by Paul Viola and Michael Jones, are based on analyzing four main features, as shown in Figure 2-14.

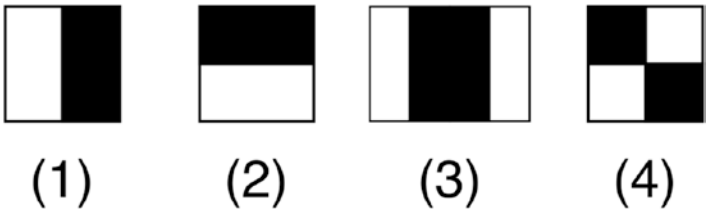

Figure 2-14. *Haar feature types*

Since the features are easily computed, the number of images required for training Haar-based object detection is quite low. Most importantly, up until recently, with low CPU speeds on embedded systems, those classifiers had the advantage of being very fast.

Where in the World Is Marcel?

So, enough talking and reading about research papers. In short, a Haar-based cascade classifier is good at finding features of faces, of humans or animals, and can even focus on eyes, noses, and smiles, as well as on features of a full body.

The classifiers can also be used to count the number of moving heads in video streams and find out whether anyone is stealing things from the fridge after bedtime.

OpenCV makes it a no-brainer to use a cascade classifier and get some instant gratification. The way to put cascade classifiers into action is as follows:

1. Load the classifier from an XML definition file containing values describing the features to look for.
2. Access a `Mat` object called `detectMultiScale` directly on the input `Mat` object, and a `Mat` object called `MatOfRect`, which is a specific OpenCV object designed to handle lists of rectangles nicely.
3. Once the previous call is finished, `MatOfRect` is filled with a number of rectangles, each of them describing a zone of the input image, where a positive has been found.
4. Do some artsy drawing on the original picture to highlight what was found by the classifier.
5. Save the output.

The Java code for this is actually rather simple and just barely longer than the equivalent Python code. See Listing 2-9.

Listing 2-9. Calling a Cascade Classifier on an Image

```
String classifier
    = "haarcascade_frontalcatface.xml";

CascadeClassifier cascadeFaceClassifier
    = new CascadeClassifier(classifier);

Mat cat = Imgcodecs.imread("marcel.jpg");

MatOfRect faces = new MatOfRect();
cascadeFaceClassifier.detectMultiScale(cat, faces);

for (Rect rect : faces.toArray()) {
     Imgproc.putText(
       cat,
       "Chat",
      new Point(rect.x, rect.y - 5),
      Imgproc.FONT_HERSHEY_PLAIN, 10,
      new Scalar(255, 0, 0), 5);

     Imgproc.rectangle(
       cat,
       new Point(rect.x, rect.y),
       new Point(rect.x + rect.width, rect.y + rect.height),
      new Scalar(255, 0, 0),
      5);
}

Imgcodecs.imwrite("marcel_blunt_haar.jpg", cat);
```

After running the code in Listing 2-9, you will realize quickly what the problem is with the slightly naïve approach. The classifier is finding a lot of extra positives, as shown in Figure 2-15.

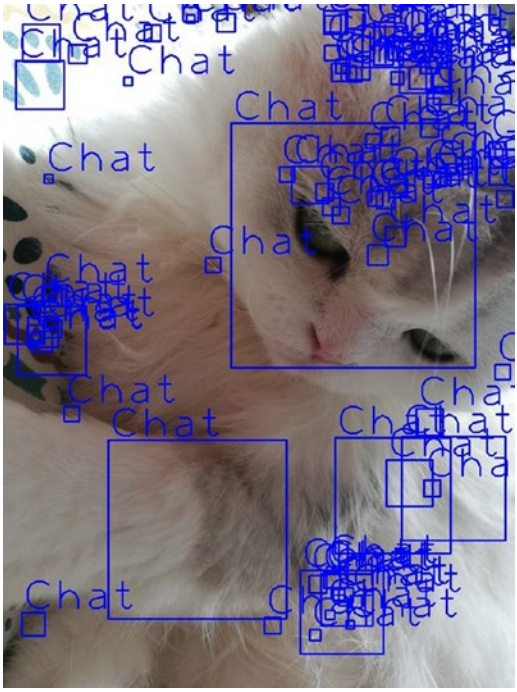

Figure 2-15. *Too many Marcels…*

There are two techniques to reduce the number of detected objects.

- Filter the rectangles based on their sizes in the loop on rectangles. While this is often used thanks to its simplicity, this adds a chance to focus on false positives.
- Pass in extra parameters to `detectMultiScale`, specifying, among other things, a certain number of required neighbors to get a positive or indeed a minimum size for the returned rectangles.

The full version has all the parameters shown in Table 2-3.

Table 2-3. *detectMultiScale Parameters*

Parameter	Description
`image`	Matrix of the type `CV_8U` containing an image where objects are detected.
`objects`	Vector of rectangles where each rectangle contains the detected object; the rectangles may be partially outside the original image.
`scaleFactor`	Parameter specifying how much the image size is reduced at each image scale.
`minNeighbors`	Parameter specifying how many neighbors each potential candidate rectangle should have to be retained.
`flags`	Not used for a new cascade.
`minSize`	Minimum possible object size. Objects smaller than this value are ignored.
`maxSize`	Maximum possible object size. Objects larger than this value are ignored if the `maxSize == minSize` model is evaluated on a single scale.

Based on the information from Table 2-3, let's apply some sensible parameters to `detectMutiScale`, as shown here:

- `scalefactor=2`
- `minNeighbors=3`
- `flags=-1` (ignored)
- `size=300x300`

Building on the code in Listing 2-9, let's replace the line containing `detectMultiScale` with this updated one:

```
cascadeFaceClassifier.detectMultiScale(cat, faces, 2, 3, -1,
new Size(300, 300));
```

Applying the new parameters, and (why not?) running a debug session on the new code, gives you the layout shown in Figure 2-16.

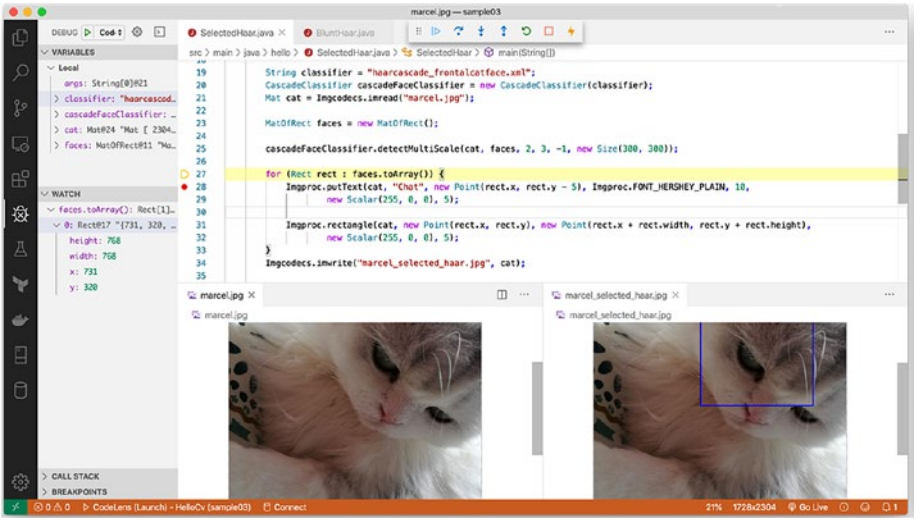

Figure 2-16. *Debugging session while looking for Marcel*

The output is already included in the layout, but you will notice the found rectangles are now limited to only one rectangle, and the output gives only one picture (and, yes, there can be only one Marcel).

Finding Cat Faces in Pictures Using the Yolo Neural Network

We have not really seen how to train cascade classifiers to recognize things we want them to recognize (because it's beyond the scope of this book). The thing is, most classifiers have a tendency to recognize some things better than others, for example, people more than cars, turtles, or signs.

Detection systems based on those classifiers apply the model to an image at multiple locations and scales. High-scoring regions of the image are considered detections.

The neural network Yolo uses a totally different approach. It applies a neural network to the full image. This network divides the image into regions and predicts bounding boxes and probabilities for each region. These bounding boxes are weighted by the predicted probabilities.

Yolo has proven fast in real-time object detection and is going to be our neural network of choice to run in real time on the Raspberry Pi in Chapter 3.

Later, you will see how to train a custom Yolo-based model to recognize new objects that you are looking for, but to bring this chapter to a nice and exciting end, let's quickly run one of the provided default Yolo networks, trained on the COCO image set, that can detect a large set of 80 objects, cats, bicycle, cars, etc., among other objects.

As for us, let's see whether Marcel is up to the task of being detected as a cat even through the eyes of a modern neural network.

The final sample introduces a few more OpenCV concepts around deep neural networks and is a great closure for this chapter.

You probably know already what a neural network is; it is modeled on how the connections in the brain work, simulating threshold-based neurons triggered by incoming electrical signals. If my highly neuron-dense brain were a drawing, it would look something like Figure 2-17.

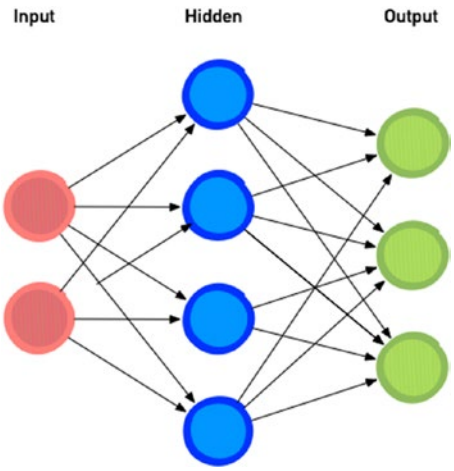

Figure 2-17. *My brain*

I actually do hope reality is slightly different and my brain is way more colorful.

What you see in Figure 2-17 at first sight is the configuration of the network. Figure 2-17 shows only one hidden layer in between the input and output layers, but standard deep neural networks have around 60 to 100 hidden layers and of course way more dots for inputs and outputs.

In many cases, the network maps the image, with a hard-coded size specific to that network, from one pixel to one dot. The output is actually slightly more complicated, containing among other things probabilities and names, or at least an index of names in a list.

During the training phase, each of the arrows in the diagram, or each neuron connection in the network, is slowly getting a weight, which is something that impacts the value of the next neuron, which is a circle, in the next hidden or output layer.

When running the code sample, we will need both a config file for the graphic representation of the network, with the number of hidden layers and output layers, and a file for the weights, which includes a number for each connection between the circles.

Let's move from theory to practice now with some Java coding. We want to give some file as the input to this Yolo-based network and recognize, you guessed it, our favorite cat.

The Yolo example is split into several steps, listed here:

1. Load the network as an OpenCV object using two files. As you have seen, this needs both a weight file and a config file.
2. In the loaded network, find out the unconnected layers, meaning the layers that do not have an output. Those will be output layers themselves. After running the network, we are interested in the values contained in those layers.
3. Convert the image we would like to detect objects from to a blob, something that the network can understand. This is done using the OpenCV function `blobFromImage`, which has many parameters but is quite easy to grasp.
4. Feed this beautiful blob into the loaded network, and ask OpenCV to run the network using the function `forward`. We also tell it the nodes we are interested in retrieving values from, which are the output layers we computed before.
5. Each output returned by the Yolo model is a set of the following:
    - Four values for the location (basically four values to describe the rectangle)
    - Values representing the confidence for each possible object that our network can recognize, in our case, 80

6. We move to a postprocess step where we extract and construct sets of boxes and confidences from the outputs returned by the network.

7. Yolo has a tendency to send many boxes for the same results; we use another OpenCV function, `Dnn.NMSBoxes`, that removes overlapping by keeping the box with the highest confidence score. This is called *nonmaximum suppression* (NMS).

8. We display all this on the picture using annotated rectangles and text, as is the case for many object detection samples.

See Listing 2-10 for the full code. Java, being verbose, results in quite a few lines, but this is not something you should be afraid of anymore. Right?

Listing 2-10. Running Yolo on Images

```
    static final String OUTFOLDER = "out/";
    static final Scalar BLACK = new Scalar(0, 0, 0);

    static {
        new File(OUTFOLDER).mkdirs();
    }
public static void main(String[] args) throws Exception {
    NativeLoader.loadLibrary(Core.NATIVE_LIBRARY_NAME);
    runDarknet(new String[] { "marcel.jpg", "marcel2.jpg",
    "chats.jpg", });
}

private static void runDarknet(String[] sourceImageFile) throws
IOException {
    // read the labels
    List<String> labels = Files.readAllLines(Paths.get("yolov3/
    coco.names"));
```

```
    // load the network from the config and weights files
    Net net = Dnn.readNetFromDarknet("yolov3/yolov3.cfg",
    "yolov3/yolov3.weights");
    // look up for the output layers
    // this is network configuration dependent
    List<String> layersNames = net.getLayerNames();
    List<String> outLayers = net.getUnconnectedOutLayers().
    toList().stream().map(i -> i - 1).map(layersNames::get)
            .collect(Collectors.toList());

    // run the inference for each input
    for (String image : sourceImageFile) {
        runInference(net, outLayers, labels, image);
    }
}

private static void runInference(Net net, List<String> layers,
List<String> labels, String filename) {
    final Size BLOB_SIZE = new Size(416, 416);
    final double IN_SCALE_FACTOR = 0.00392157;
    final int MAX_RESULTS = 20;

    // load the image, convert it to a blob, and
    // then feed it to the network
    Mat frame =
       Imgcodecs.imread(filename, Imgcodecs.IMREAD_COLOR);
    Mat blob = Dnn.blobFromImage(frame, IN_SCALE_FACTOR, BLOB_
    SIZE, new Scalar(0, 0, 0), false);
    net.setInput(blob);
    // glue code to receive the output of running the
    // network
```

```
    List<Mat> outputs = layers.stream().map(s -> {
        return new Mat();
    }).collect(Collectors.toList());

    // run the inference
    net.forward(outputs, layers);

    List<Integer> labelIDs = new ArrayList<>();
    List<Float> probabilities = new ArrayList<>();
    List<String> locations = new ArrayList<>();

    postprocess(filename, frame, labels, outputs, labelIDs,
    probabilities, locations, MAX_RESULTS);
}

private static void postprocess(String filename, Mat frame,
List<String> labels, List<Mat> outs,
        List<Integer> classIds, List<Float> confidences,
        List<String> locations, int nResults) {

    List<Rect> tmpLocations = new ArrayList<>();
    List<Integer> tmpClasses = new ArrayList<>();
    List<Float> tmpConfidences = new ArrayList<>();
    int w = frame.width();
    int h = frame.height();

    for (Mat out : outs) {
        final float[] data = new float[(int) out.total()];
        out.get(0, 0, data);

        int k = 0;
        for (int j = 0; j < out.height(); j++) {
            // Each row of data has 4 values for location,
            // followed by N confidence values
            // which correspond to the labels
```

```
            Mat scores = out.row(j).colRange(5, out.width());
            // Get the value and location of the maximum score
            Core.MinMaxLocResult result =
                   Core.minMaxLoc(scores);
            if (result.maxVal > 0) {
                float center_x = data[k + 0] * w;
                float center_y = data[k + 1] * h;
                float width = data[k + 2] * w;
                float height = data[k + 3] * h;
                float left = center_x - width / 2;
                float top = center_y - height / 2;

                tmpClasses.add((int) result.maxLoc.x);
                tmpConfidences.add((float) result.maxVal);
                tmpLocations.add(
                  new Rect(
                         (int) left,
                   (int) top,
                         (int) width,
                         (int) height));
            }
            k += out.width();
        }
    }
    annotateFrame(frame, labels, classIds, confidences,
    nResults, tmpLocations, tmpClasses, tmpConfidences);
    Imgcodecs.imwrite(OUTFOLDER + new File(filename).getName(),
    frame);
}

private static void annotateFrame(Mat frame, List<String>
labels, List<Integer> classIds, List<Float> confidences,
```

```
        int nResults, List<Rect> tmpLocations, List<Integer>
        tmpClasses, List<Float> tmpConfidences) {

    // Perform non maximum suppression to eliminate
    // redundant overlapping boxes with
    // lower confidences and sort by confidence
    // many overlapping results coming from yolo
    // so have to use it
    MatOfRect locMat = new MatOfRect();
    locMat.fromList(tmpLocations);

    MatOfFloat confidenceMat = new MatOfFloat();
    confidenceMat.fromList(tmpConfidences);

    MatOfInt indexMat = new MatOfInt();
    Dnn.NMSBoxes(locMat, confidenceMat, 0.1f, 0.1f, indexMat);

    // at this stage we only have the non-overlapping boxes,
    // with the highest confidence scores
    // so we draw them on the pictures.
    for (int i = 0; i < indexMat.total() && i < nResults; ++i) {
        int idx = (int) indexMat.get(i, 0)[0];
        classIds.add(tmpClasses.get(idx));
        confidences.add(tmpConfidences.get(idx));
        Rect box = tmpLocations.get(idx);
        String label = String.format("%s [%.0f%%]", labels.
        get(classIds.get(i)), 100 * tmpConfidences.get(idx));

        Imgproc.rectangle(frame, box, BLACK, 2);
        Imgproc.putText(frame, label, new Point(box.x, box.y),
        Imgproc.FONT_HERSHEY_PLAIN, 5.0, BLACK, 3);
    }
}
```

Running the example on images of Marcel actually brings really high confidence scores that are close to a perfect location, as shown in Figure 2-18.

Figure 2-18. *Yolo based detection confirms Marcel is a cute cat*

Running the inference on many cats also gives great results, but on the second run we are missing one of the cats because of its close position to another detected cat and because of our introduction of the postprocessing step, which removes overlapping matching boxes. See Figure 2-19.

Figure 2-19. *Many cats*

An exercise for you at this point is to change the parameters of the `Dnn.NMSBoxes` function call to see whether you can get the two boxes to show at the same time.

The problem with static images is that it is difficult to get the extra context that we have in real life. This is a shortcoming that goes away when dealing with sets of input images coming from a live video stream.

So, with Chapter 2 wrapped up, you can now do object detection on pictures. Chapter 3 will take it from here and use the Raspberry Pi to teach you about on-device detection.

CHAPTER 3

Vision on Raspberry Pi 4

> *Imagine a world in which your entire possession is one raspberry, and you give it to your friend.*
>
> —Gerda Weissmann Klein,
> *All But My Life* (Hill and Wang, 1995)

In Chapters 1 and 2, you saw how to get up and running with computer vision using Java on a regular desktop computer. The setup was a bit manual, but all the different building blocks were freely available, so it was just a matter of putting those blocks together.

In this chapter, you'll add one more block to the mix, the Raspberry Pi 4. The first goal of the chapter is to perform the same kinds of tasks that you did on a computer but run as much as possible on the IoT device, while typing code and controlling the code execution from your local desktop. In other words, you'll write on your computer but execute on the Raspberry.

For the code execution part, you need to have some specific hardware and cabling, so you should prepare yourself to go shopping. We'll provide a shopping list and then move on to the setup.

Specifically, we will go through creating the SD card and setting up the Raspberry Pi, but it is assumed that most of those steps are already properly covered in other books and blogs, so you can go and read those if you need more than the basics presented here.

N. Modrzyk, *Real-Time IoT Imaging with Deep Neural Networks*,
https://doi.org/10.1007/978-1-4842-5722-7_3

At the end of this chapter, you will have good grasp of the Raspberry, its speed, and its energy consumption while running the different vision operation and detection algorithms. You will also be ready to integrate this functionality on a larger scale into a voice-controlled home assistant of your design and thus expand your personal home or work office automation.

Bringing the Raspberry to Life

"Man," I cried, "how ignorant art thou in thy pride of wisdom!"

—Mary Shelley, *Frankenstein* (1818)

We're no Doctor Frankenstein, but to bring the Pi to life, we will have to work as if we are designing one: plugging in the cables, assembling, and turning on the power of the little device. Some shopping will be needed, and we have to gather the physical hardware before doing anything useful. Luckily, most of the things to buy are cheap and can be reused from other people's old computers or from your grandma's attic. She won't notice.

Once the shopping is done and the Raspberry Pi and all the essential parts are assembled, we will move to putting in place the same vision setup on Visual Studio Code as in Chapter 1 and learn how to program OpenCV in Java again.

Shopping

While the Raspberry Pi is a solid computer on its own, most of the time you will not be able to do anything with it without a few other extra pieces. In this chapter, you will need a few accessories.

Table 3-1 provides a shopping list, but you can decide what type and where to buy things.

Careful There are a wide range of cables and power adapters these days. Be sure to buy a power adapter that gets around 4.8A to 5A output if you want to do overclocking. Your chosen USB cable should also be able to carry the electrical current, but these days most of them do.

Table 3-1. *Shopping List*

Name	What	Approximate Price (USD)	Required?	Always Needed?	
Raspberry Pi 4 4GB	Main computer	$55	Yes	Yes	☐
USB-C to USB-A cable	Connect the computer to power	$6	Yes	Yes	☐
Power adapter with USB-A (at least 4.8A or even 5A for overclocking)	Electric power	$10	Yes	Yes	☐
HDMI cable	Connect to screen	$7	Yes	No	☐
SD card 128GB	Hard disk	$10	Yes	Yes	☐
ReSpeaker microphone	Recommended microphone for snips (Chapter 3)	$25	No	If bought, yes	☐
Display with HDMI port	Display for initial debugging	$25	Yes	No	☐
USB webcam	Record video	$20	Yes	Yes	☐
Keyboard and mouse		$20	Maybe	No	☐
Total		$180			

Once you have gathered all the parts (Figure 3-1), it's time to get the software running on the Pi.

Figure 3-1. *Raspberry Pi 4, power adapter, webcam, screen, etc*

Downloading the OS

The Raspberry Pi will refuse to even look at a beautiful cat like Marcel without an operating system. There are a few operating systems readily available on the Raspberry web site.

Here's a short list of the main ones:

- Ubuntu Mate
- Ubuntu Core
- Windows 10 IoT Core
- Noobs
- Raspbian (the official operating system)

Ubuntu Mate and Ubuntu Core are new and good alternatives, but we will focus on the official operating system, Raspbian, because most of the documentation available for the Raspberry is based on it. It's also quite easy to get started with it.

Raspbian is a Debian-based operating system, so most Linux aficionados will be at ease with it and its included aptitude package manager. Even though packages have to be compiled specifically for the ARM-based CPU of the Raspberry, the package list is quite up-to-date and mostly in sync with the main Debian line.

To be able to use the operating system as a bootable image that the Raspberry can use, we first need to download the image file onto a local computer and create a bootable disk specifically for the Raspberry Pi.

To download the very big OS file (don't try this while tethering on your phone), head to `https://www.raspberrypi.org/downloads/` and look for the Raspbian icon shown in Figure 3-2.

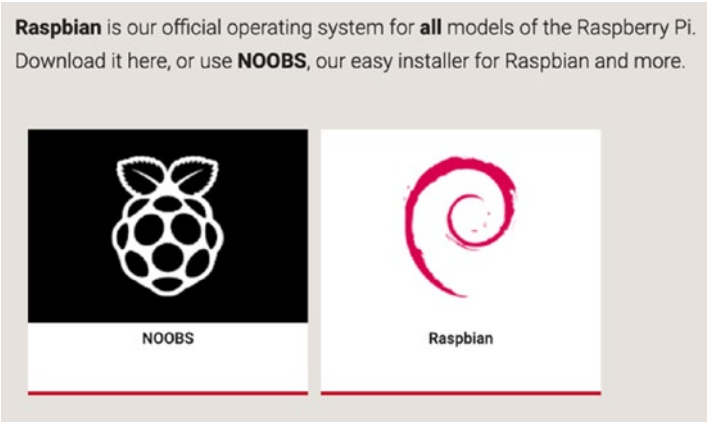

Figure 3-2. *Raspbian icon*

Click the icon. You will then land on the Raspbian-specific download page, as shown in Figure 3-3, where you can find the link to the zip file to retrieve.

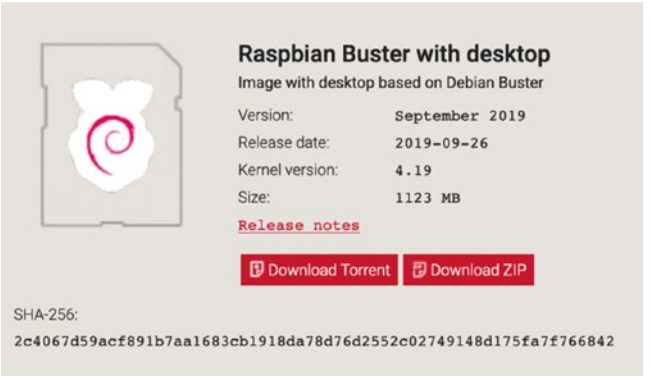

Figure 3-3. *Torrent or zip file for Raspbian*

We don't need a full desktop install and all the proposed software, but we will aim for simplify so "Raspbian Buster with desktop" will be sufficient for us now.

Creating the Bootable SD Card

With the OS file on your computer, you are ready to create the bootable device. As noted on the Raspberry web site, the easiest way to get the OS on the Pi is by using Etcher, which is software to create the bootable SD card from a zip file or an `.img` file.

Etcher is available from `https://www.balena.io/etcher/`, and it can run on any platform, such as macOS, Windows, and Linux. The screenshots in this book were taken on Windows.

The first step after starting the application is to select the downloaded zip file, as shown in Figure 3-4.

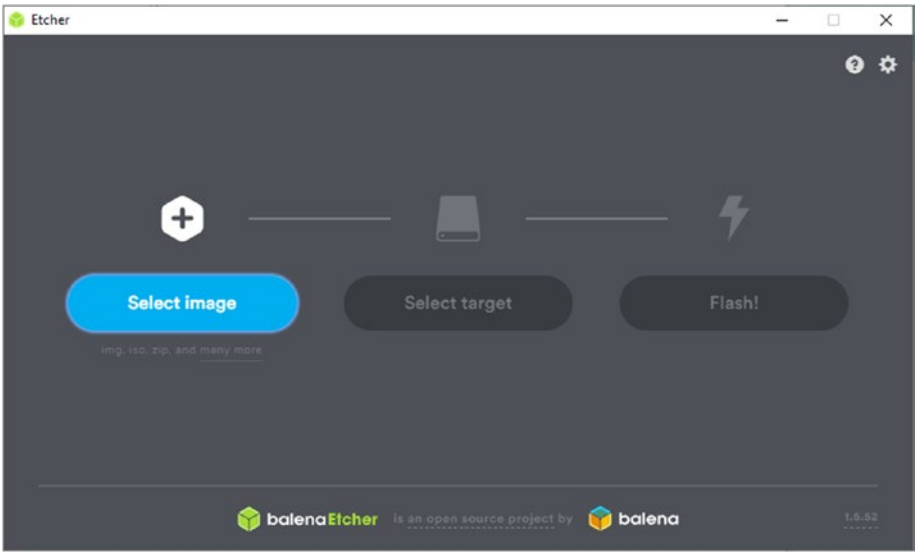

Figure 3-4. *Selecting the image file*

With the image file selected and the SD card inserted into an SD slot of your computer, let's select the SD card you want to write the OS onto, as shown in Figure 3-5.

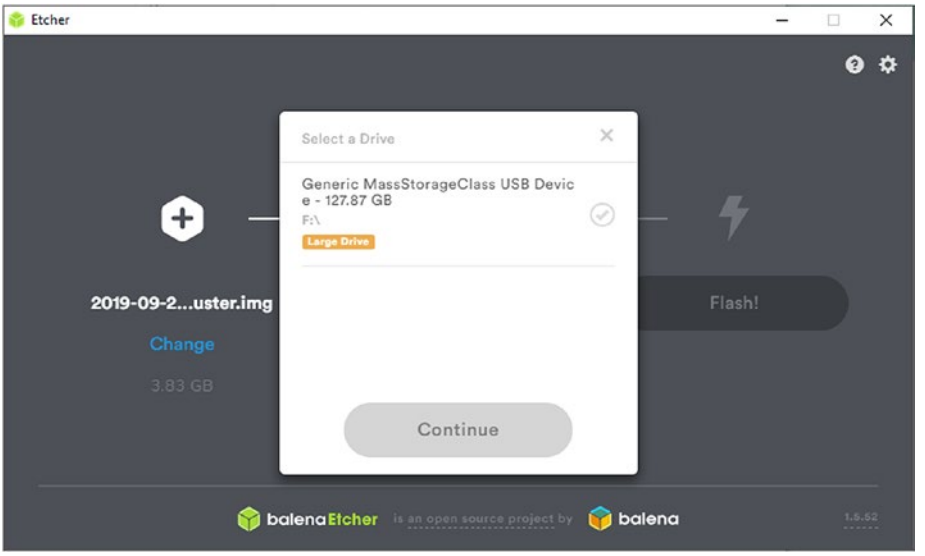

Figure 3-5. *Selecting the micro SD card*

The card used here was specifically chosen because it's so big, so you ignore the message shown about its size, as shown in Figure 3-6.

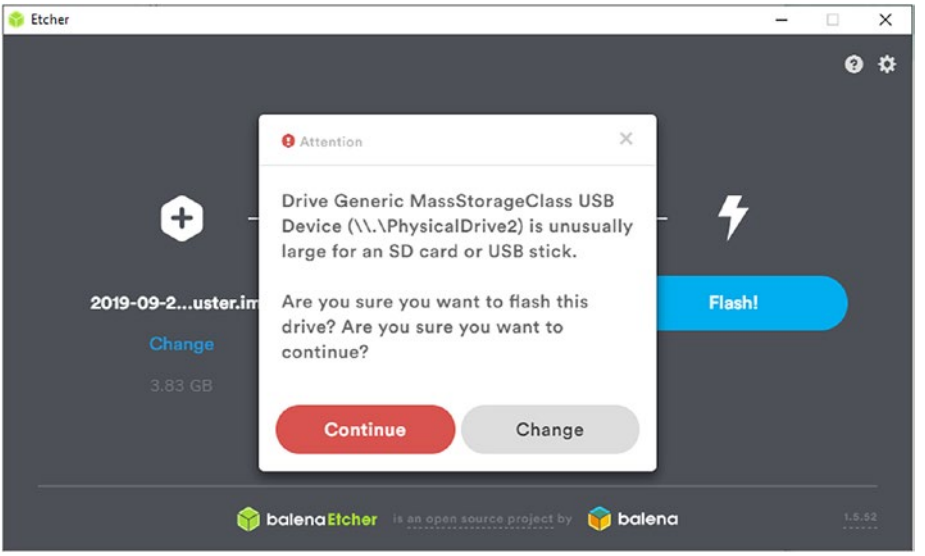

Figure 3-6. *The SD card is big*

Finally, let's go ahead and "flash" the card, in other words, write the files onto the card. The progress is shown and should take no more than two to three minutes, as shown in Figure 3-7.

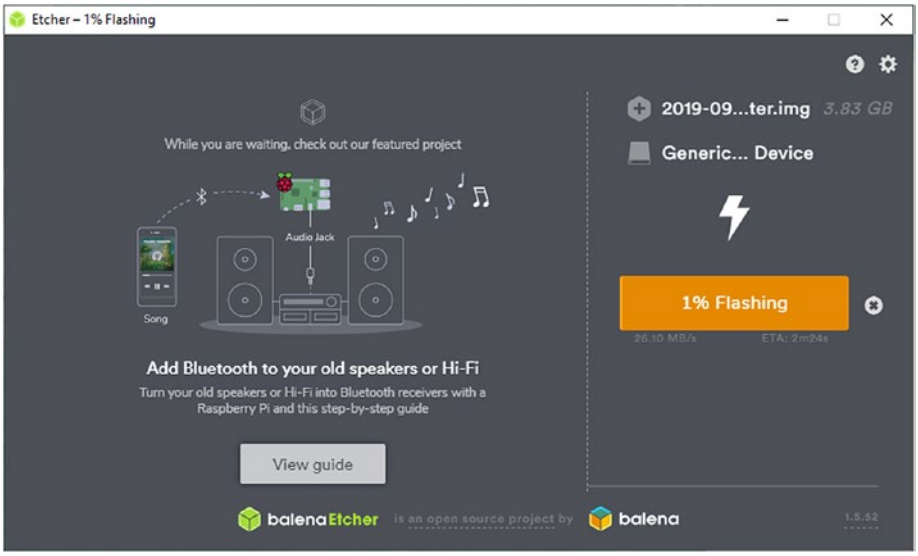

Figure 3-7. *Flashing*

In the last step, Etcher will check the integrity of the copied data, as shown in Figure 3-8.

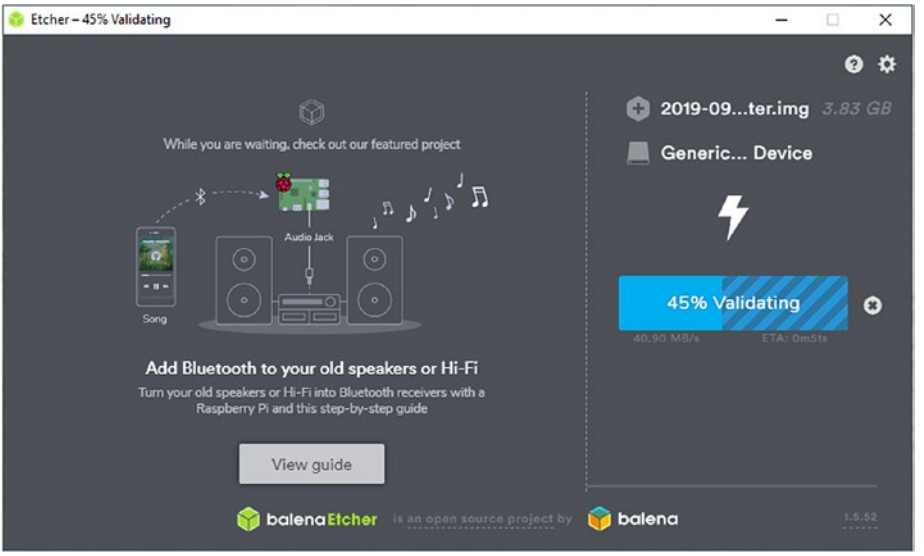

Figure 3-8. *Validating*

Once the flashing and the validation are done, Windows will try to use the drive as before, but the format of the SD card is now not readable without a specific driver, as shown in Figure 3-9. You can safely ignore the message and click Cancel.

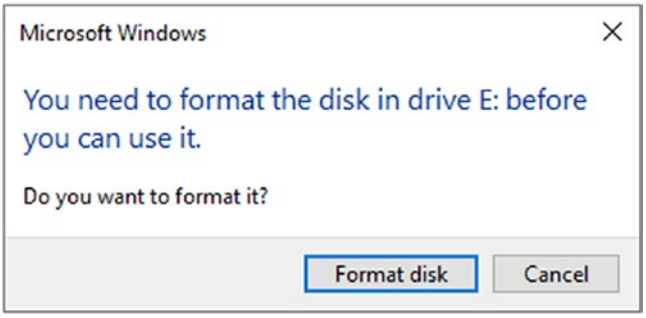

Figure 3-9. *Don't be scared…*

The card is here, so let's now plug in the cables of the Pi.

Connecting the Cables

> *Electricity can be dangerous. My nephew tried to stick a penny into a plug. Whoever said a penny doesn't go far didn't see him shoot across that floor. I told him he was grounded.*
>
> —Tim Allen

I have found plugging things together to be quite fun and relaxing and could actually be used in personal therapy for the tough world we live in.

Table 3-2 provides a nonexhaustive list of the connections that have to be made.

Table 3-2. *What to Plug Where Checklist*

From	To	Why	
Power AC	Power adapter	Get some power	☐
USB-A	Power adapter	Power to the Pi	☐
Webcam	Pi USB port	Get some video stream	☐
Mouse	Pi USB port	Use a pointer	☐
Keyboard	Pi USB port	Use a keyboard	☐
Screen HDMI	Pi Micro HDMI	Display	☐
USB-C	Raspberry Pi	Power to the Pi	☐

Note Make sure to plug either the power adapter or the USB-C to Raspberry adapter in last; otherwise, the power will turn the board on without anything to do.

Things usually are easier to understand with a picture, so make sure to refer to Figure 3-10 to see where things fit together on the actual board.

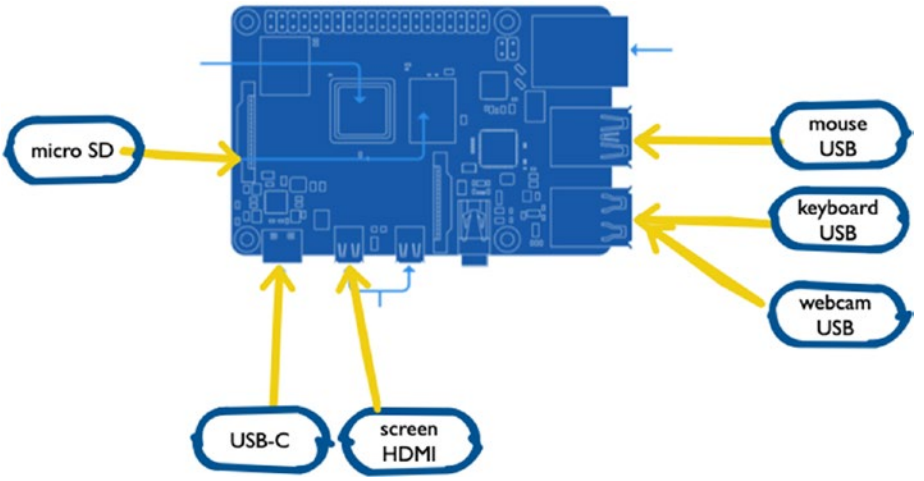

Figure 3-10. *Original blueprints of the Raspberry Pi 4 and what to connect to*

Finally, with the micro SD card inserted into the Raspberry Pi 4, you are ready to power things on (or have you already?).

First Boot

The first boot is quite fast. The boot process includes a step where the file system needs to be resized, which is now handled automatically by Raspbian on first boot. You should get a welcome screen like the one shown in Figure 3-11.

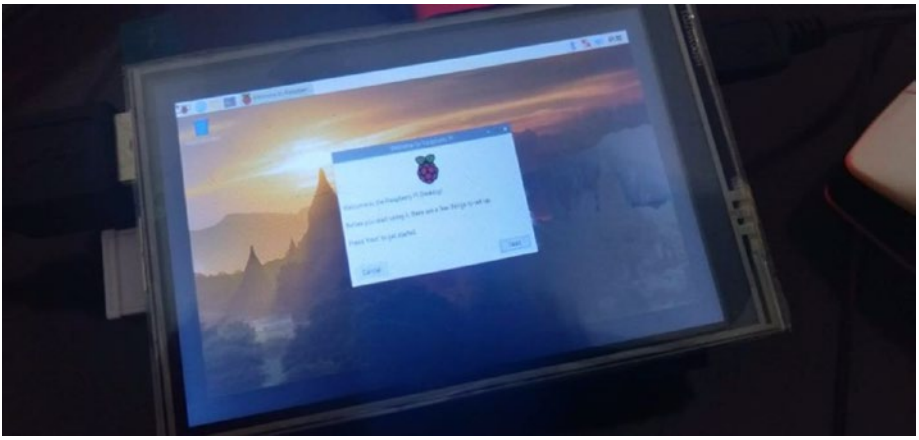

Figure 3-11. *First boot*

With the keyboard, mouse, and screen plugged in, you can do quite a bit, but for most of the tasks we are going to do in this chapter, we want to work over SSH, which requires a network connection.

You can plug an Ethernet cable into the Ethernet port of the Pi, but WiFi works quite well. On the top right, click the WiFi icon and enter your WiFi settings in the dialog box, as shown in Figure 3-12.

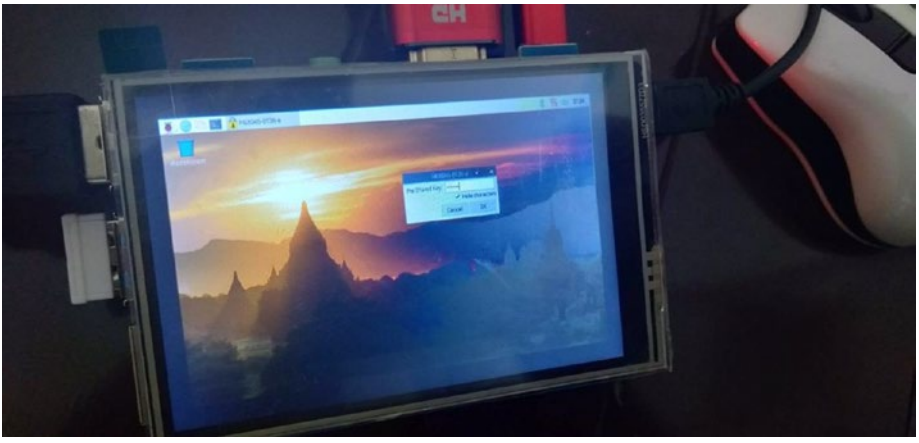

Figure 3-12. *Setting up the WiFi*

One last step is required before we can connect via SSH, which is to enable the Raspberry Pi–embedded OpenSSH server, which can be enabled via the following menu:

```
Preferences ➤ Raspberry Pi Configuration ➤ Interfaces
```

Click to enable the SSH server, as shown in Figure 3-13.

Figure 3-13. *Enabling the installed SSH server*

Now let's connect to the Raspberry via SSH.

Finding Your Raspberry Using nmap

As you probably know, to remotely connect to the Raspberry Pi, you need its IP address, which you can find from within the Raspberry Pi by opening a terminal and typing the following:

```
ip addr
```

This will display the IP address you need to use to connect to the Raspberry Pi.

If you're out of luck and without a keyboard, a screen, a mouse, or all of these, you can revert to some simple commando countermeasures and use `nmap` to find all the Raspberry Pis on the network (and all the computers and Chinese smartphones for that matter).

This is done by running a command similar to the following one from the main computer:

```
sudo nmap -O -p 22 192.168.1.*
```

Here, 192.168.1.∗ is the start of your own desktop IP address and where the Raspberry would be. Note that for this to work, we're assuming that your computer and the Raspberry are on the same network.

What does the command actually do? In short, `nmap` simply scans the local network for available machines and open ports.

The `-p` switch scans for a given port, and the `-O` switch says to gather as much information as possible about each found host.

Running the previous `nmap` command on the local network gives us something like Figure 3-14.

```
Nmap scan report for raspberrypi (192.168.1.17)
Host is up (0.064s latency).

PORT   STATE SERVICE
22/tcp open  ssh
MAC Address: DC:A6:32:0E:C3:A1 (Raspberry Pi Trading)
Warning: OSScan results may be unreliable because we could not find at least 1 open and 1 closed port
Device type: general purpose
Running: Linux 3.X|4.X
OS CPE: cpe:/o:linux:linux_kernel:3 cpe:/o:linux:linux_kernel:4
OS details: Linux 3.2 - 4.9
Network Distance: 1 hop
```

Figure 3-14. *Looking for your Raspberry Pi*

A Raspberry (hopefully yours!) has been found on the network. You can now SSH through it using the default username/password, which should still be `pi / raspberry`, with the following command:

```
ssh pi@192.16.1.17
```

and then get connected, as shown in Figure 3-15.

```
SuperPinkicious:opencv-java-template niko$ ssh pi@192.168.1.17
The authenticity of host '192.168.1.17 (192.168.1.17)' can't be established.
ECDSA key fingerprint is SHA256:oe1HUXgEJHiWeggUlrvfiogGCW1xsNL0EnVQCYKI2uE.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.1.17' (ECDSA) to the list of known hosts.
pi@192.168.1.17's password:
Linux raspberrypi 4.19.75-v7l+ #1270 SMP Tue Sep 24 18:51:41 BST 2019 armv7l

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Thu Sep 26 01:31:22 2019

SSH is enabled and the default password for the 'pi' user has not been changed.
This is a security risk - please login as the 'pi' user and type 'passwd' to set a new password.

pi@raspberrypi:~ $
```

Figure 3-15. *Starting an SSH connection*

We made one connection, but we're going to have to repeat those connection steps over and over, so let's save a few SSH settings for later by modifying a configuration file both locally and on the remote Raspberry.

Setting Up SSH Easily

Usually when connecting via SSH, you disable password authentication, especially the default authentication of the Raspberry Pi, and set up a key from your local machine to connect to the remote machine and then set up that remote machine to accept only one key, or a limited number of them, so as to prevent random security attacks.

This works like this:

1. Create your personal key pair to use with SSH. One private key stays on your computer, and one public key is added as a known entity on the remote machine. Until quantum computing comes in, that key pair is said to be secure, and it's close to impossible to guess one's private key by just knowing the public key. For more information on this, read Joshua Davies' article at https://commandlinefanatic.com/cgi-bin/showarticle.cgi?article=art054.

2. Register the public key that got generated on the remote device, here the Raspberry Pi. Go ahead and give the public key to as many people as you want; by default, this is the file ending with the `.pub` extension.

3. Add some configuration for SSH to use that generated private key when connecting to the Raspberry through a shortcut. That private key, though, is yours, so make sure you don't give it to anyone!

On Amazon Web Services (AWS), you usually get a `.pem` file to connect through a new running instance. That `.pem` file is automatically registered for you on the new VM, so only you can access that newly created VM, thus allowing a secure access to your in-the-cloud machines.

To create your key on your own local machine, you use `ssh-keygen` with the `-t` switch to specify the algorithm and the `-b` switch to specify the number of bits (nowadays, 4,096 is considered very secure).

```
% ssh-keygen -t rsa -b 4096
Generating public/private rsa key pair.
Enter file in which to save the key (~/.ssh/id_rsa):
```

Once the command has run successfully, two files are generated: an `id_rsa.pub` file and an `id_rsa` file. `pub` is your public key that you can give to anyone, and `id_rsa` is your private key and should be handled with the utmost care.

To tell the Raspberry Pi about your public key and recognize you, you add it as a one-liner to the `/home/pi/.ssh/authorized_keys` file, creating the file and the `.ssh` parent folder if needed.

Once this is done, on your computer, it's time to add details of the SSH connection settings to the `$HOME/.ssh/config` file, again creating the file if you don't have it.

```
Host pi4
      User pi
      IdentityFile ~/.ssh/id_rsa
      HostName 192.168.1.17
      ForwardX11 yes
      ForwardX11Trusted yes
```

Here is what the connection settings mean:

- `pi4` is the shortcut that we will use to connect to the Raspberry from now on.
- `User` is the default user on the Raspberry Pi and usually is `pi`.
- `IdentifyFile` is the path to the private key of your computer that will be used to authenticate you; this is the file you created a few moments ago.
- `HostName` is the IP address of the Raspberry. Use `nmap` again if you forgot it.
- `ForwardX11` and `ForwardX11Trusted` are required because we will redirect the video stream to the main computer in a few seconds.

With all these settings, you may be wondering if this was all worth it. Let's get a bit of instant gratification to prove that it was. Via an SSH connection, we will now open an application with a UI, for example the included Chromium browser.

You can switch the display of the windows on the Raspberry screen by changing the setting of the `DISPLAY` variable.

- To display the Raspberry Pi screen, use `export DISPLAY=:0`.
- To display the local computer screen, use `export DISPLAY=:10`.

Here's the list of commands:

```
ssh pi4
export DISPLAY=:10
chromium-browser
```

Figure 3-16 shows how Chromium is executing on the Pi but opening its window on the computer.

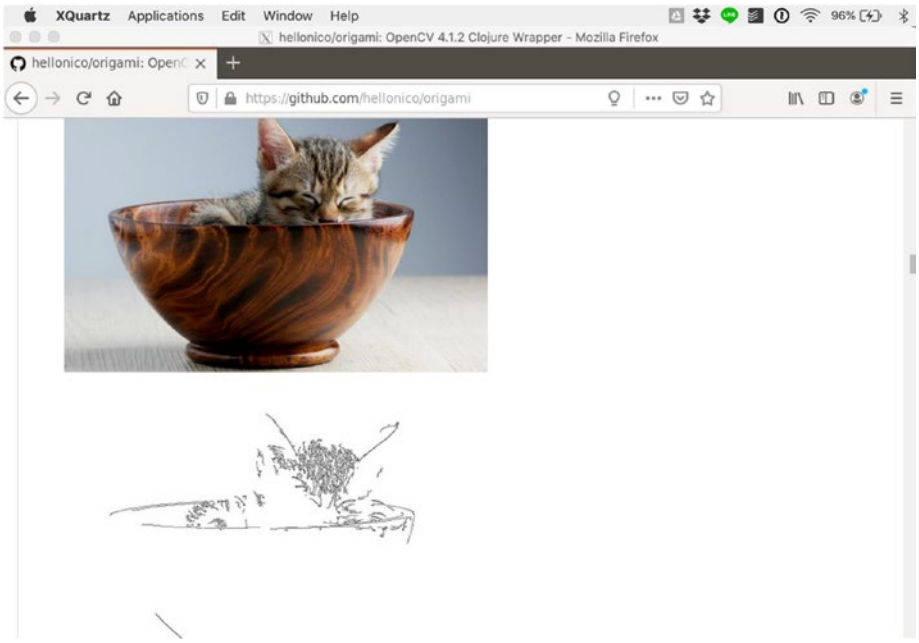

Figure 3-16. *Opening a remote application*

Setting Up Visual Code Studio for Remote Use

One of the main concepts of this book is that you will be able to write code on a machine with a big screen and then get it to run on a remote device, redirecting the visual output to the big screen.

Up to now, you have been writing code in Visual Studio Code and running it locally. From now on, you'll write code locally and run it remotely.

All those SSH setup efforts earlier in the chapter were made because Visual Studio Code has a plugin named Remote-SSH that, once you have the SSH setup done, does all the rest for you and fully integrates with the editor's ecosystem.

Let's first install the Remote-SSH plugin, as shown in Figure 3-17.

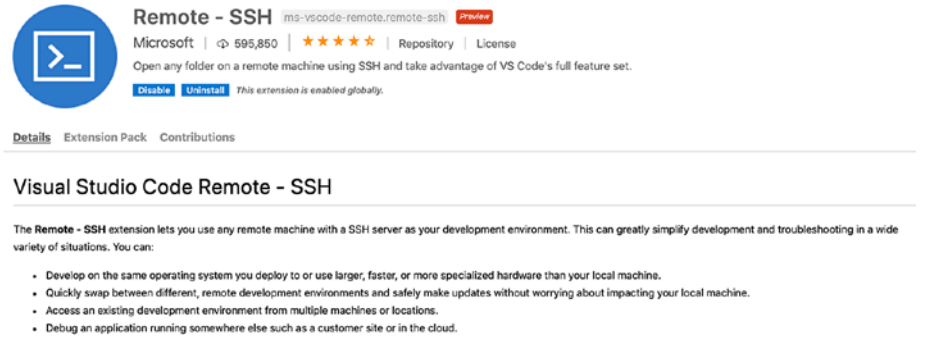

Figure 3-17. *Installing the Remote-SSH plugin*

Once the plugin is installed and after a quick refresh of the editor, you can start the Connect to Host command available from the Command launcher, as shown in Figure 3-18.

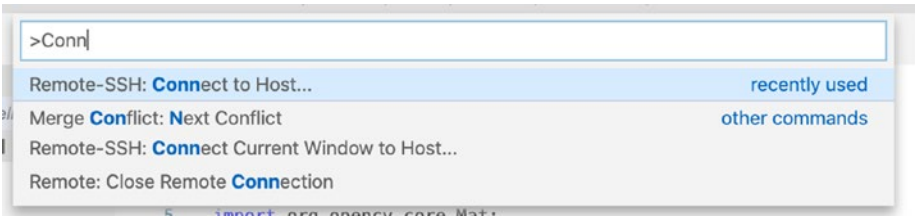

Figure 3-18. *Connecting to the host*

Caution The Remote-SSH plugin may not work properly if the current project has too many files. If that seems to be the case, make sure to open a new blank window in Visual Studio Code before opening a new Remote-SSH session.

Figure 3-19, Figure 3-20, and Figure 3-21 show you the steps necessary to get connected to your Raspberry Pi. Note that the `pi4` shortcut is taken from the one you just set up in the `$HOME/.ssh/config` file. This will be different if you used another shortcut.

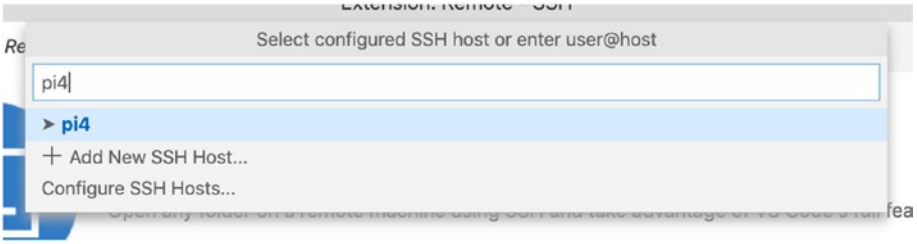

Figure 3-19. *Choosing pi4*

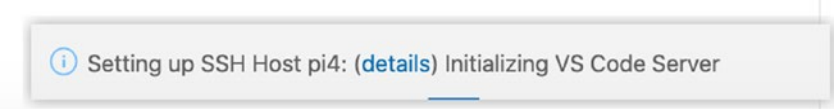

Figure 3-20. *Setting up the SSH host*

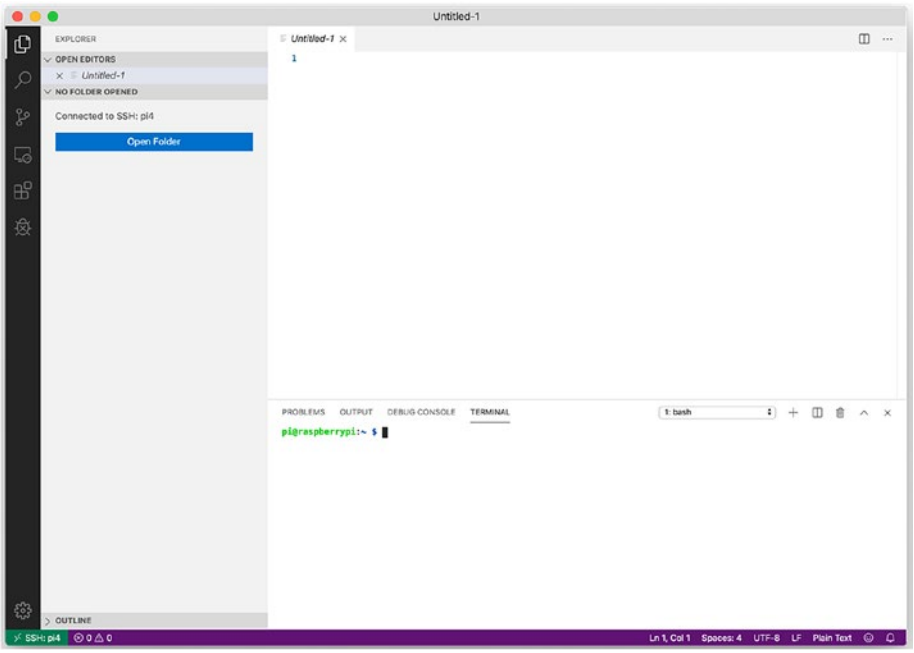

Figure 3-21. *Connected to the Raspberry Pi from Visual Studio Code*

At this stage, from inside Visual Studio Code, you have a terminal remotely connected to your Raspberry Pi. That means you can launch remote commands at will, but you can also unplug the keyboard and the mouse from your Raspberry Pi; the rest will pretty much be done from your local computer.

As a first check, we can run a command to check the IP address of the Pi, as shown here:

```
ip addr
```

The output will be shown on the Terminal tab of Visual Studio Code, as shown in Figure 3-22.

```
pi@raspberrypi:~ $ ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc mq state DOWN group default qlen 1000
    link/ether dc:a6:32:0e:c3:a0 brd ff:ff:ff:ff:ff:ff
3: wlan0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether dc:a6:32:0e:c3:a1 brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.17/24 brd 192.168.1.255 scope global noprefixroute wlan0
       valid_lft forever preferred_lft forever
    inet6 240d:1a:d6:f400:dea6:32ff:fe0e:c3a1/128 scope global dynamic noprefixroute
       valid_lft 10111sec preferred_lft 10111sec
    inet6 fe80::e170:d079:836e:dfef/64 scope link
       valid_lft forever preferred_lft forever
pi@raspberrypi:~ $
```

Figure 3-22. *Confirming the IP address on the Terminal tab*

Outputting the IP address is obviously not the main thing we can do with that terminal, so let's move on and install the required Java Development Kit now.

Setting Up the Java OpenJDK

All the code in this book is Java code, so we'll need to install the Java Development Kit and runtime on the Raspberry Pi.

As an optional dependency, you can also install Maven. Maven is not really required for most of the examples of the book, but this will stop Visual Studio Code from displaying annoying messages that it cannot find the (unnecessary) Maven executable.

So, on the Terminal tab of Visual Studio Code, let's get the software package installer, `apt`, to install Java for us. This is done via the small snippet shown here:

```
sudo apt update
sudo apt install openjdk-11-jdk
```

Optionally, also run the following command to install Maven:

```
sudo apt install maven
```

sudo at the start of each command line is used to get the required escalated privileges to install new software with apt.

The output of the previous commands should be similar to Figure 3-23.

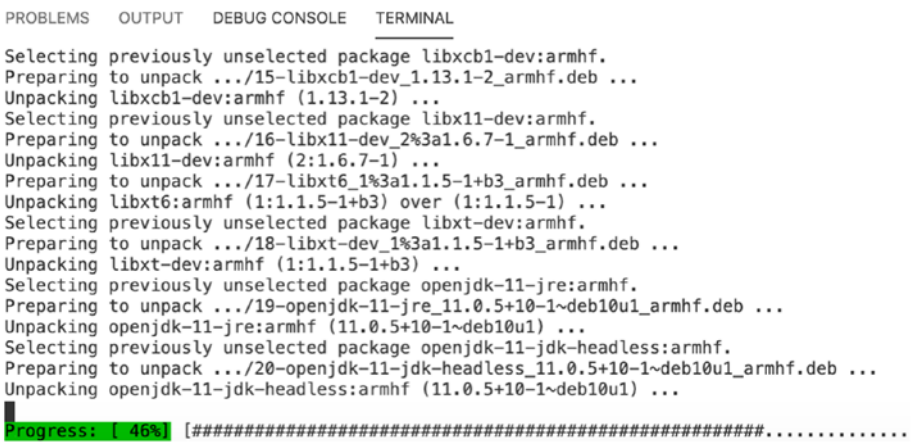

Figure 3-23. *Installing Java*

Alternative to Setting Up the Java SDK

The current selection of JDK on Raspberry Pi 4 running Buster (the latest Raspbian version) is quite impressive. Sometimes, however, especially on older versions of Raspbian, you could be stuck with an outdated JDK for quite some time.

It's worth noting that companies like Azul and Bellsoft have released up-to-date versions of the JDK specifically compiled for the ARM CPU, the processor running on the Raspberry Pi.

Visit these two sites for direct links to the JDK packages:

```
https://bell-sw.com/pages/java-11.0.5%20for%20Embedded/
https://www.azul.com/downloads/zulu-community/
```

The following short snippet will download and install the JDK 11 from Bellsoft:

```
sudo wget https://download.bell-sw.com/java/11.0.5+11/bellsoft-
jdk11.0.5+11-linux-arm32-vfp-hflt.deb
sudo dpkg -i bellsoft-jdk11.0.5+11-linux-arm32-vfp-hflt.deb
```

The available version is indeed Java 11, as shown in the following output:

```
pi@raspberrypi:~ $ java -version
openjdk version "11.0.5-BellSoft" 2019-10-15
OpenJDK Runtime Environment (build 11.0.5-BellSoft+11)
OpenJDK 32-Bit Server VM (build 11.0.5-BellSoft+11, mixed mode)
```

With the Java Development Kit now installed on your Raspberry Pi, let's move on to the meat of this chapter, running OpenCV code in Java and on the Pi.

Checking Out the OpenCV/Java Template

We'll start by checking out the OpenCV/Java project template directly on the Raspberry. There are a few ways to do this; the three main ones are as follows:

- Directly perform a Git clone of the template
- Download the zip file from the Git project template
- Generate the templated project using Maven

Let's quickly review these three one by one.

Performing a Git Clone

It is possible and convenient to simply check out code from Visual Studio Code straight to your Pi. This is done via the command palette and typing anything starting with `git`.

You will get a few options, one of them being Git: Clone, as shown in Figure 3-24.

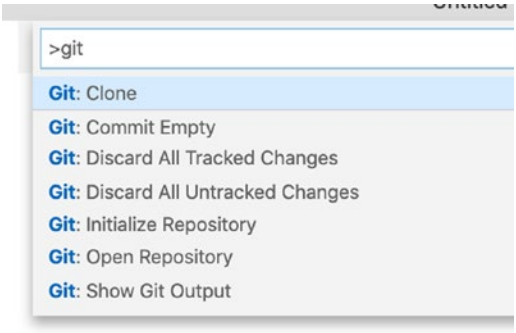

Figure 3-24. *Git:Clone option in Visual Studio Code*

From here, enter the location of the template repository as follows, as shown in Figure 3-25:

```
https://github.com/hellonico/opencv-java-template.git
```

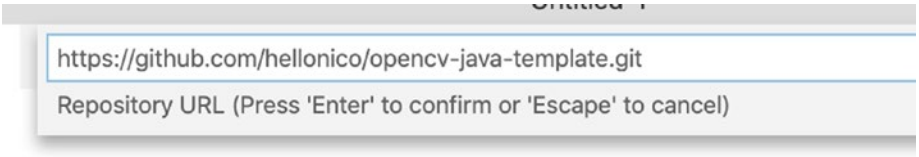

Figure 3-25. *Entering the template URL*

Finally, confirm by clicking Open in the dialog box (Figure 3-26).

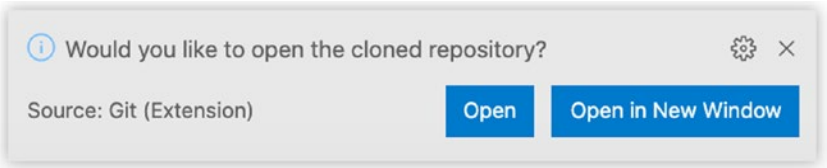

Figure 3-26. *Of course, open the cloned repository*

The project is now checked out on the Raspberry and ready to be executed.

Downloading the Zip File

Alternatively, you can download the zip file from the project template. This can be done using the `wget` command in the terminal in case you do not want to use Git at all.

```
wget https://github.com/hellonico/opencv-java-template/archive/
master.zip
unzip master.zip
```

Figure 3-27 shows the Terminal tab of Visual Code Studio.

```
pi@raspberrypi:~ $ wget https://github.com/hellonico/opencv-java-template/archive/master.zip
--2019-12-08 04:14:51--  https://github.com/hellonico/opencv-java-template/archive/master.zip
Resolving github.com (github.com)... 13.114.40.48
Connecting to github.com (github.com)|13.114.40.48|:443... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://codeload.github.com/hellonico/opencv-java-template/zip/master [following]
--2019-12-08 04:14:52--  https://codeload.github.com/hellonico/opencv-java-template/zip/master
Resolving codeload.github.com (codeload.github.com)... 13.112.159.149
Connecting to codeload.github.com (codeload.github.com)|13.112.159.149|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: unspecified [application/zip]
Saving to: 'master.zip'

master.zip                              [ <=>                                                ]   3.69K  --.-KB/s    in 0.001s

2019-12-08 04:14:52 (6.10 MB/s) - 'master.zip' saved [3775]
```

Figure 3-27. *Downloading the zip file from the Pi*

This has the advantage of not being reliant on Git but has the consequence that it is harder to share updates between different Raspberry Pis.

Using Maven

A third way to set up the project is the same technique that was used to originally create the Git template repository: using the build tool Maven. In the Java ecosystem, quite a few people are a bit afraid of going the Maven way, but for a set of given tasks, Maven is a pretty solid choice.

Maven, once installed, can create a project for you with all the required files through the Maven archetypes.

The following command, once copied and pasted onto the Terminal tab, will generate the project structure for you, as shown here:

```
mvn org.apache.maven.plugins:maven-archetype-plugin:2.4:generate \
    -DarchetypeArtifactId=maven-archetype \
    -DarchetypeGroupId=origami \
    -DarchetypeVersion=1.0 \
    -DarchetypeCatalog=https://repository.hellonico.info/
     repository/hellonico/ \
    -Dversion=1.0-SNAPSHOT \
    -DgroupId=hello \
    -DartifactId=opencv-java-template
```

The important parameters of the command have been highlighted in the previous snippet and are explained here:

- `archetypeVersion`: This is the version of the project structure, here 1.0. There may be a few updates soon, but don't expected too many. It's better to create your own.
- `version`: This is the version of your project; it usually starts at `1.0-SNAPSHOT`.
- `groupId`: This is the Maven group of your project, like `com.google`, etc.
- `artifactId`: This is the name of the project within the `groupId`. It must be unique.

Here we are using an older version of the archetype plugin; with this version, it is easier to retrieve artifacts from the custom Maven repository at `repository.hellonico.info`.

Later versions of the archetype plugin require a deeper knowledge of the Maven framework, which is out of the scope of this book.

Whichever technique you use, you can now open the remote folder. You should see something similar to the tree in Figure 3-28.

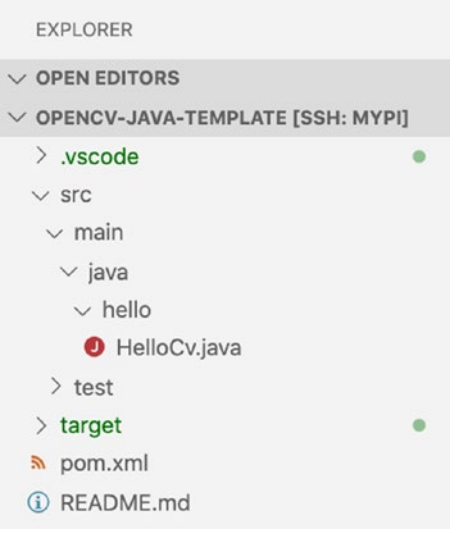

Figure 3-28. *Project files on the Raspberry Pi*

Again, the files are located on the Raspberry Pi, the Java code will be executed on the Raspberry Pi, but the editor is effectively running on your local computer.

Installing the Visual Code Java Extension Pack Remotely

If you remember from Chapter 1, we installed plugins to get Visual Studio Code to understand and run Java. We'll do the same here, but this time on the Raspberry Pi.

Figure 3-29 shows the Marketplace tab, where you can search for the same plugin that was installed before, the Java Extension Pack. Note in Figure 3-30 how Visual Studio Code nicely asks you if you want to install the plugin on the Raspberry.

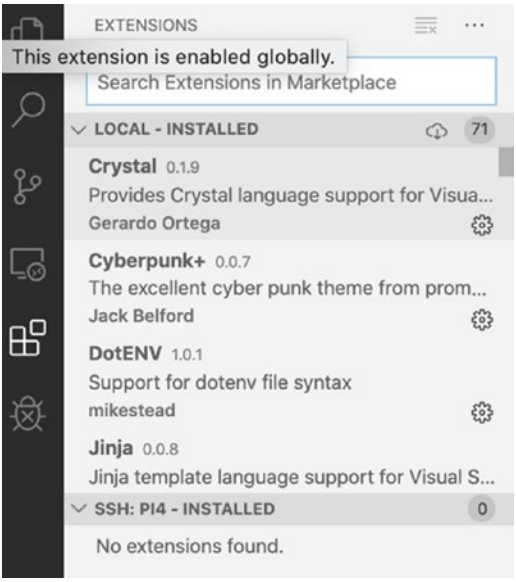

Figure 3-29. *"SSH. PI4" has no extension yet*

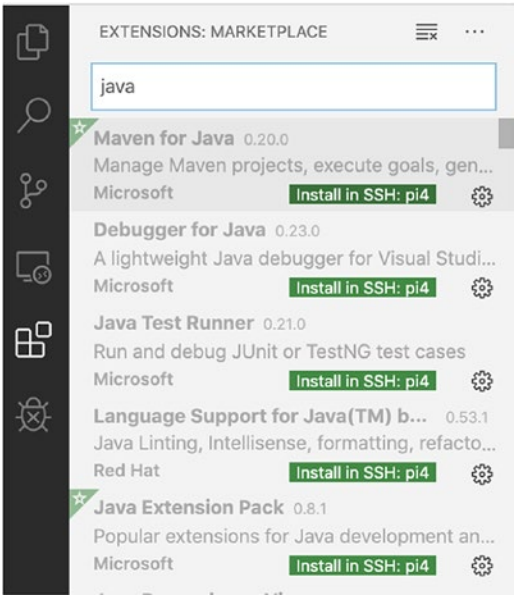

Figure 3-30. *Looking for the Java Extension Pack again*

Figure 3-31 and Figure 3-32 show the last two steps of installing the Visual Code Java plugin on the Raspberry Pi.

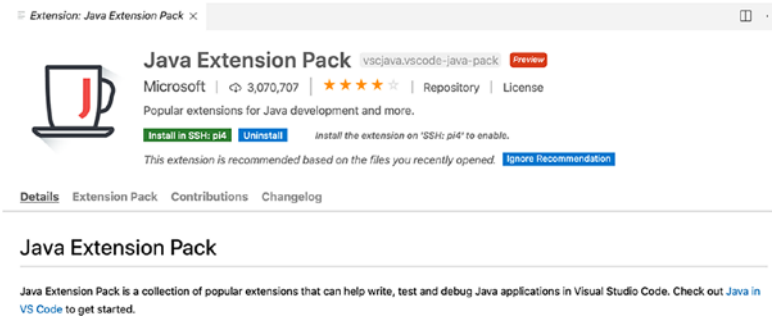

Figure 3-31. *Finding the Java Extension Pack again*

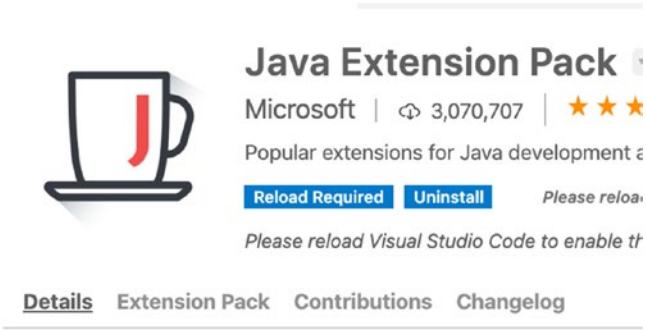

Figure 3-32. *Editor reload required again*

You should now have your editor showing the familiar Run and Debug links straight in the editor, as shown in Figure 3-33.

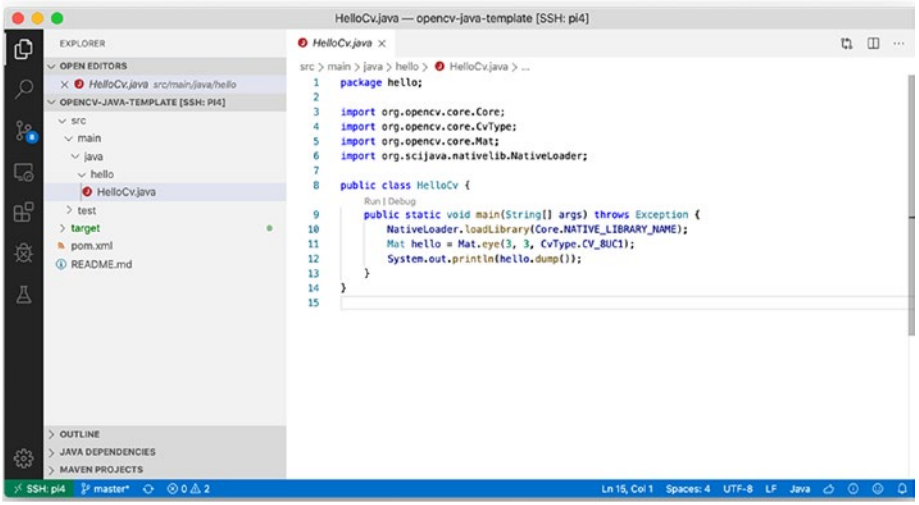

Figure 3-33. *Familiar Run and Debug buttons showing*

All the steps were quite long the first time around, but now you're ready for some good times. You are all set up for running some OpenCV/Java on the Raspberry Pi.

Running the First OpenCV Example

Without further ado, let's click the Run button and get the program to execute.

Clicking the Run button runs the same program as in Chapter 1, but the program is executed on the Raspberry Pi this time. The output shows the familiar 3×3 matrix, as in Figure 3-34.

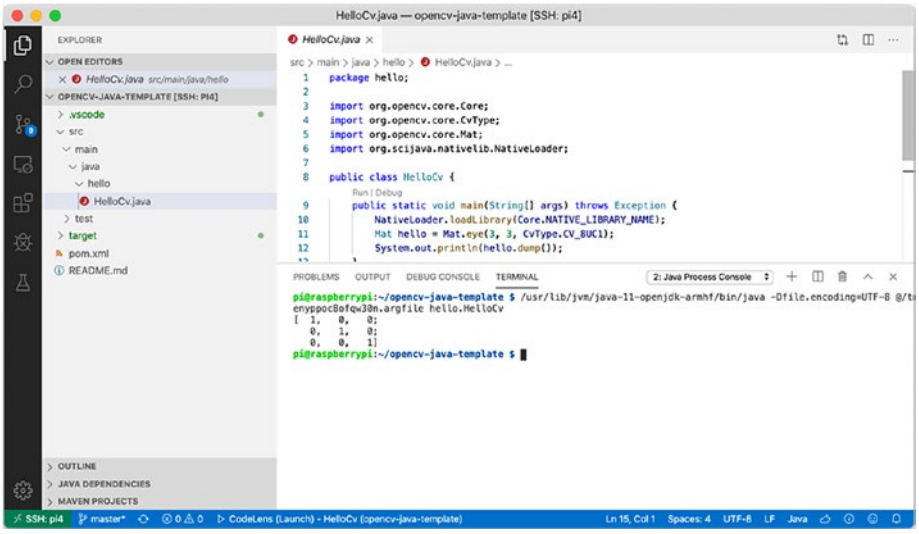

Figure 3-34. *The code has executed on the Raspberry Pi*

Note that various downloads are done in the background on the first run, so the command takes a bit of time. On the second run, things are quicker.

As a reminder, debugging works here in the same way as in Chapter 1. As an exercise, try adding a breakpoint to pause the program execution after running Debug and then add a watch expression on `System.getProperty()` to show the `java.vm.vendor` value. Since we are now running on the Raspberry Pi, the value should be `Raspbian`, as shown in Figure 3-35.

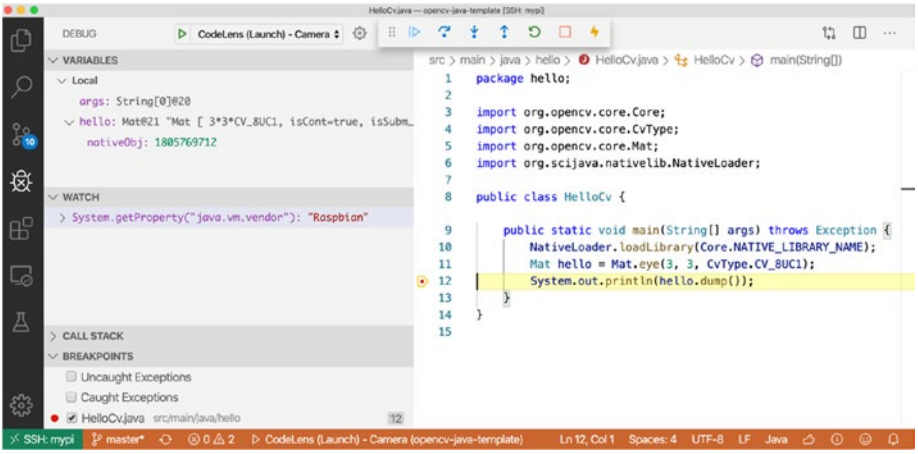

Figure 3-35. *Debugging and displaying java.vm.vendor*

Running on Linux or a VM with AWS Instead

So, what if you were actually not running on a Raspberry Pi but on a remote Linux server or a cloud virtual machine?

All the setup you have done up to now was actually very generic, so you could have been connecting to a standard box running Linux, provided you have SSH access to it.

For example, in Figure 3-36, you can see the same setup connecting to a remote Linux box and running my favorite desktop distribution, Manjaro Linux.

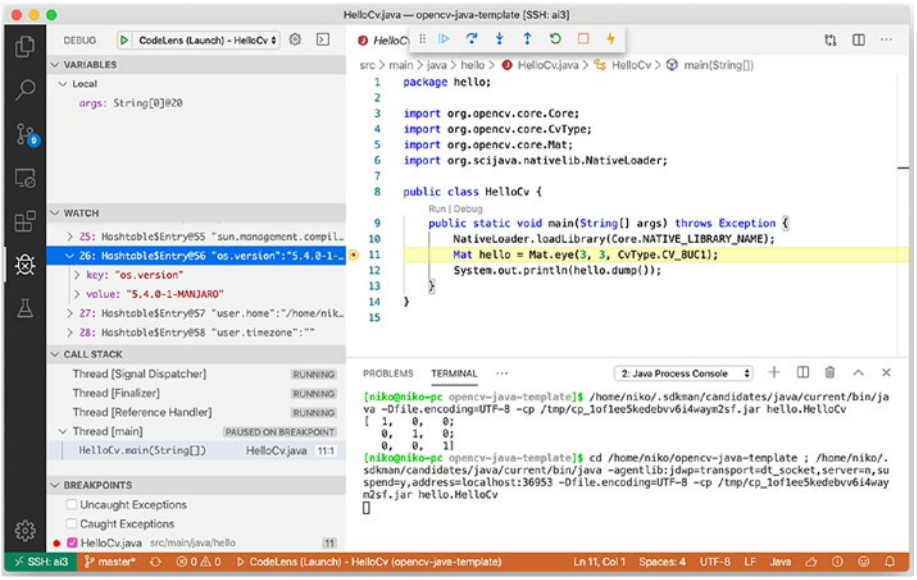

Figure 3-36. *Connecting remotely to a running Linux box*

To give you some more creative ideas, I am also using that same setup to connect to remote machines on Amazon Web Services.

Capturing a Video Live Stream

If your webcam is plugged in properly via the USB port, you are now ready to write some code to capture the video stream straight from your IoT device.

We use two building blocks here to get us started.

- OpenCV's `VideoCapture` Java class to get the hardware to the software (or more precisely, from the video stream to the OpenCV `Mat` objects). In this example, the `VideoCapture` object takes an `int` as the ID of the video capture device to use.

- The `ImShow` class, which is not part of the core OpenCV Java package. It's a small class similar to OpenCV's `HighGui` class (https://docs.opencv.org/master/javadoc/org/opencv/highgui/HighGui.html), but it makes it easier to just plug into video streams and show them on the screen.

We also use `init` OpenCV with a convenient method called `Origami.init()`, which searches for and loads the proper binary OpenCV library for us.

Listing 3-1 shows the full code for this first streaming example.

Listing 3-1. Accessing and Displaying the Video Stream

```
package hello;

import org.opencv.core.Mat;
import org.opencv.videoio.VideoCapture;

import origami.ImShow;
import origami.Origami;

public class Webcam {

    public static void main(String[] args) {
        Origami.init();

        VideoCapture cap = new VideoCapture(0);

        Mat matFrame = new Mat();
        ImShow ims = new ImShow("Camera", 800, 600);
        while (cap.grab()) {
            cap.retrieve(matFrame);
            ims.showImage(matFrame);
        }
```

```
        cap.release();
    }
}
```

The code is pretty standard OpenCV code. Running the code should give a picture like in Figure 3-37 (without the frame rate showing).

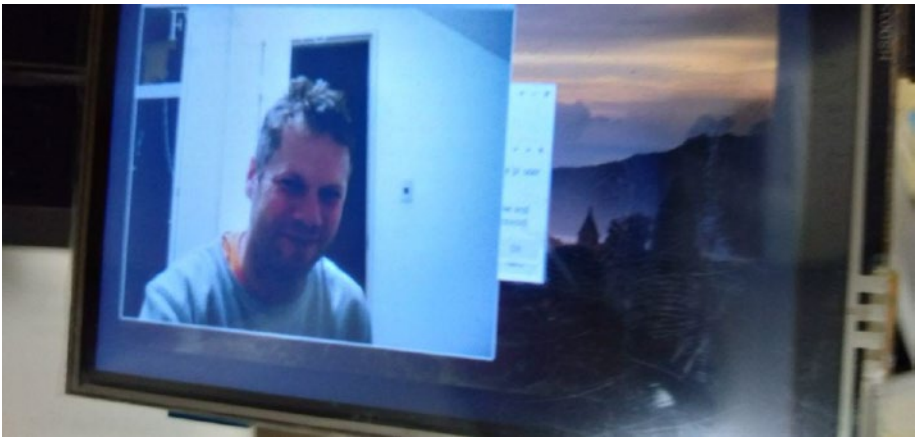

Figure 3-37. *Video stream without frame rate*

You'll notice the speed is very close to real-time. The code in Listing 3-2 can be added to the frame loop to show the frame rate, as in Figure 3-38.

Listing 3-2. Adding the Frame per Second on the Frame

```
long now = System.currentTimeMillis();
frame++;
Imgproc.putText(matFrame, "FPS " + (frame / (1 + (now - start) /
1000)), new Point(50, 50),
Imgproc.FONT_HERSHEY_COMPLEX, 2.0, new Scalar(255, 255, 255));
```

Figure 3-38. *Video stream with the frame rate*

Usually between 15 and 20 frames per second is an expected value for this exercise.

You may have run into problems while running the previous example. Sometimes the program dies with the stack trace shown in Listing 3-3.

Listing 3-3. No X11 DISPLAY Set

```
Exception in thread "main" java.awt.HeadlessException:
No X11 DISPLAY variable was set, but this program performed an
operation which requires it.
        at java.desktop/java.awt.GraphicsEnvironment.checkHeadl
        ess(GraphicsEnvironment.java:208)
        at java.desktop/java.awt.Window.<init>(Window.java:548)
        at java.desktop/java.awt.Frame.<init>(Frame.java:423)
        at java.desktop/java.awt.Frame.<init>(Frame.java:388)
        at java.desktop/javax.swing.JFrame.<init>(JFrame.
        java:180)
        at origami.ImShow.<init>(ImShow.java:52)
        at hello.Webcam.main(Webcam.java:22)
```

To run a graphic program via SSH, like you are doing right now, Java requires a system environment variable to be set. The name of this variable, as you can see, is `DISPLAY`. You can check its value on the Terminal tab with the following:

```
echo $DISPLAY
```

There are three main states you could be in depending on the value of this variable, as listed here:

- `<empty>`, which is not so good. You need to set the value yourself.
- `:0`, or `localhost:0`, which means any visual frame will be shown in the Raspberry screen. This is OK for debugging the setup, but it's not the one you usually want when working from your computer.
- `:10`, or `localhost:10`, which means the frames will be shown on your computer. On macOS, this is done via the XQuartz application, which runs a graphical environment compatible with the one from the Raspberry Pi. On Windows, the equivalent is Xming: https://sourceforge.net/projects/xming/.

If you set up the variable properly, then the stream will display on your computer screen, as in Figure 3-39.

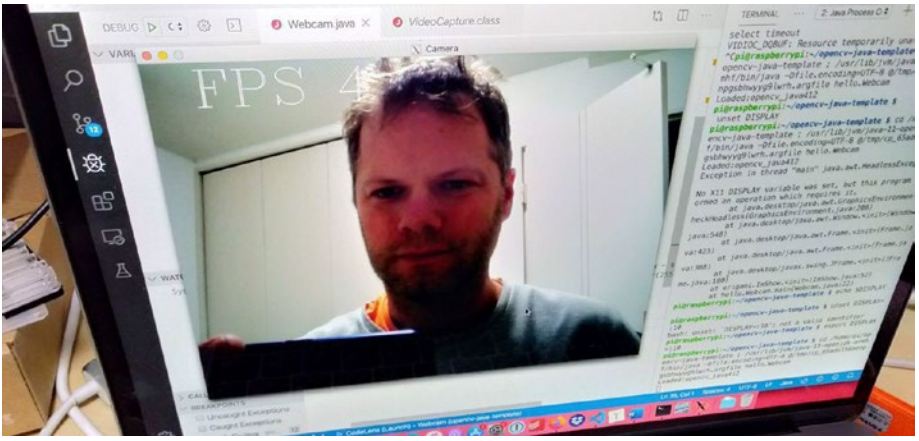

Figure 3-39. *Remote Raspberry Pi stream straight to your computer*

You know everything about setting up the Pi for remote development. Let's move on to some recipes to analyze the content of video streams.

Playing a Video

You may not always have access to a video stream and may have to revert to playing recorded footage. That works too with the `VideoCapture` class. You can give it the path to the video file you want to play, as shown here:

```
public class PlayVideo {

    public static void main(String[] args) {
        Origami.init();

        VideoCapture cap = new VideoCapture("marcel_1.mp4");

        Mat matFrame = new Mat();
        ImShow ims = new ImShow("Camera", 400, 300);
        long start = System.currentTimeMillis();
        long frame = 0;
```

```
        while (cap.grab()) {
            cap.retrieve(matFrame);
            long now = System.currentTimeMillis();
            frame++;
            Imgproc.putText(matFrame, "FPS " + (frame / (1 +
            (now - start) / 1000)), new Point(50, 50),
                    Imgproc.FONT_HERSHEY_COMPLEX, 2.0, new
                    Scalar(255, 255, 255));
            ims.showImage(matFrame);
        }
           cap.release();
    }
}
```

This code is the same as playing a stream taken from a webcam, except the file name is a parameter to the `VideoCapture` constructor.

Note that you can also find some free sample videos on the following web site:

`https://sample-videos.com/index.php`

Figure 3-40 shows some of Marcel's morning activities.

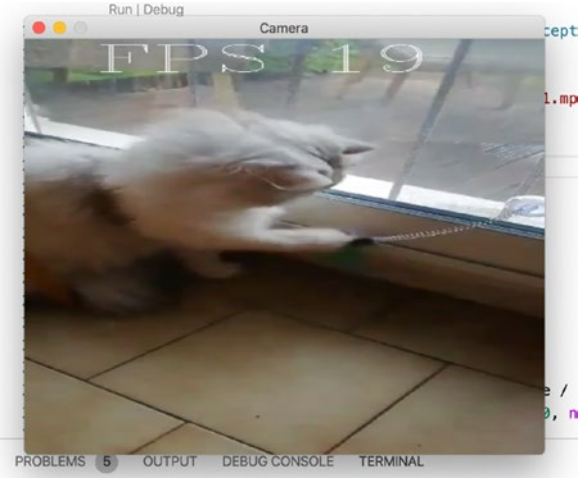

Figure 3-40. *Marcel in the morning*

If you try to enter an `http://` URL as a path to `VideoCapture`, you will see that things do not work as you probably expected. The remote video you wanted to play will not load.

A URL in the path is used to play streams from a network webcam, not static files. If you have one on hand, it's a good exercise to plug the web URL of the webcam directly into the `VideoCapture` object.

CHAPTER 4

Analyzing Video Streams on the Raspberry Pi

In this chapter, you'll learn how to analyze video streams using concepts taken from functional programming. Specifically, you'll use the `Filter` interface and combine it with a `Pipeline` object and then apply them to the video stream.

We will start with an overview of filters. Then we'll look at different basic, fun filters, and we'll gradually move on to object detection using different vision techniques. Finally, we'll talk about neural networks.

Overview of Applying Filters

In the Clojure language, you can apply a set of transformations directly to the `Mat` object of a video stream without using any extra boilerplate code. I seriously recommend taking a look at even the most basic of the origami examples, available in the README:

```
https://github.com/hellonico/origami/blob/master/README.
md#support-for-opencv-412-is-in
```


N. Modrzyk, *Real-Time IoT Imaging with Deep Neural Networks*,
https://doi.org/10.1007/978-1-4842-5722-7_4

Listing 4-1 shows how to load a picture, change it to gray, and apply a canny function all in one pipeline. This might even make you want to try the Clojure version of OpenCV.

Listing 4-1. Read, Turn to Gray, Canny Resize, and Save

```
(require
  '[opencv4.utils :as u]
  '[opencv4.core :refer :all])

(->
  (imread "doc/cat_in_bowl.jpeg")
  (cvt-color! COLOR_RGB2GRAY)
  (canny! 300.0 100.0 3 true)
  (bitwise-not!)
  (u/resize-by 0.5)
  (imwrite "doc/canny-cat.jpg"))
```

This is a Java book, though, so let's see how we can apply the same concepts in Java.

Here we introduce the concept of a pipeline of filters, where each filter performs one operation on the `Mat` object.

Here are a few examples of what filters can do:

- Turning a `Mat` object to gray
- Applying a Canny effect
- Looking for edges
- Pencil sketching
- Instagram filters, like sepia or vintage
- Doing background subtraction

- Detecting cat or people faces using Haar objection detection or color detection
- Running a neural network and identifying objects

The following are the two Java types that we'll introduce to implement these concepts:

- The `Filter` interface, which consists of only one function, `apply(Mat in)`, and returns a `Mat` object, just like in functional programming.
- The `Pipeline` class, itself a `Filter`, that takes a list of classes or already instantiated filters. When `apply` is called, it applies the classes one by one.

Listing 4-2 shows the (simple) `Filter` interface.

Listing 4-2. The Filter Interface

```
import org.opencv.core.Mat;

public interface Filter {
    public Mat apply(Mat in);
}
```

Listing 4-3 shows an example of implementing the `Pipeline` class, where you simply combine the different filters by calling them one by one.

Listing 4-3. Many Filters

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

import org.opencv.core.Mat;

public class Pipeline implements Filter {
```

```
    List<Filter> filters;

    public Pipeline(Class... __filters) {
        List<Class<Filter>> _filters = (List) Arrays.asList
        (__filters);
        this.filters = _filters.stream().map(i -> {
            try {
                return (Filter) Class.forName(i.getName()).
                newInstance();
            } catch (Exception e) {
                return null;
            }
        }).collect(Collectors.toList());
    }

    public Pipeline(Filter... __filters) {
        this.filters = (List) Arrays.asList(__filters);
    }

    @Override
    public Mat apply(Mat in) {
        Mat dst = in.clone();
        for (Filter f : filters) {
            dst = f.apply(dst);
        }
        return dst;
    }
}
```

Note here the usage of the deprecated version of the `newInstance` function directly in the class. This may not be calling the constructor you really want, but it works well enough for the examples in this book.

So far, `Filter` and `Pipeline` haven't done a lot, of course, so let's review some basic examples in the upcoming section.

Applying Basic Filters

In this section, we'll look at several examples of basic filters.

Gray Filter

The most obvious use of a filter is to turn a `Mat` object from color to gray. In OpenCV, this is done using the function `cvtColor` from the `Imgproc` class.

Omitting the package definition, we combine the webcam code from a few pages ago with the standard `cvtColor` wrapper in a class that implements the recently introduced `Filter` interface (Listing 4-4).

Listing 4-4. Gray Filter on the Stream

```
import org.opencv.core.Mat;
import org.opencv.imgproc.Imgproc;
import org.opencv.videoio.VideoCapture;

import origami.ImShow;
import origami.Origami;

public class WebcamWithFilters {

    public static void main(final String[] args) {
        Origami.init();

        final VideoCapture cap = new VideoCapture(0);
        final ImShow ims = new ImShow("Camera", 800, 600);
        final Mat buffer = new Mat();
        Filter gray = new Gray();
        while (cap.read(buffer)) {
```

```
            ims.showImage(gray.apply(buffer));
        }
        cap.release();
    }

}

class Gray implements Filter {

    public Mat apply(final Mat img) {
        final Mat mat1 = new Mat();
        Imgproc.cvtColor(img, mat1, Imgproc.COLOR_RGB2GRAY);
        return mat1;
    }

}
```

You can run this directly on the Pi, and if you stand in front of your webcam, you will get something like I did in Figure 4-1.

Figure 4-1. *Not only my hair, but the whole picture, is turning gray*

Edge Preserving Filter

In the same fashion, we can implement a wrapper around the `EdgePreserving` function of OpenCV's `Photo` class. This is used in many different applications to smooth and remove unwanted lines in pictures. For example, Listing 4-5 is really just a basic call of the function `edgePreservingFilter`.

Listing 4-5. EdgePreservering Class Re-implemented as a Filter

```
import org.opencv.photo.Photo;

class EdgePreserving implements Filter {
    public int flags = Photo.RECURS_FILTER;
    // int flags = NORMCONV_FILTER;
    public float sigma_s = 60;
    public float sigma_r = 0.4f;

    public Mat apply(Mat in) {
        Mat dst = new Mat();
        Photo.edgePreservingFilter(in, dst, flags, sigma_s,
        sigma_r);
        return dst;
    }
}
```

You can use this new filter by modifying the main method of `WebcamWithFilters`, as shown in Listing 4-6.

Listing 4-6. Modified Main Function to Use Edge Preserving Filter

```
Filter filter = new EdgePreserving();
while (cap.read(buffer)) {
      ims.showImage(filter.apply(buffer));
}
```

Now let's move on to executing the new code. Again, if you're facing your webcam, you should get something similar to what I look like in Figure 4-2.

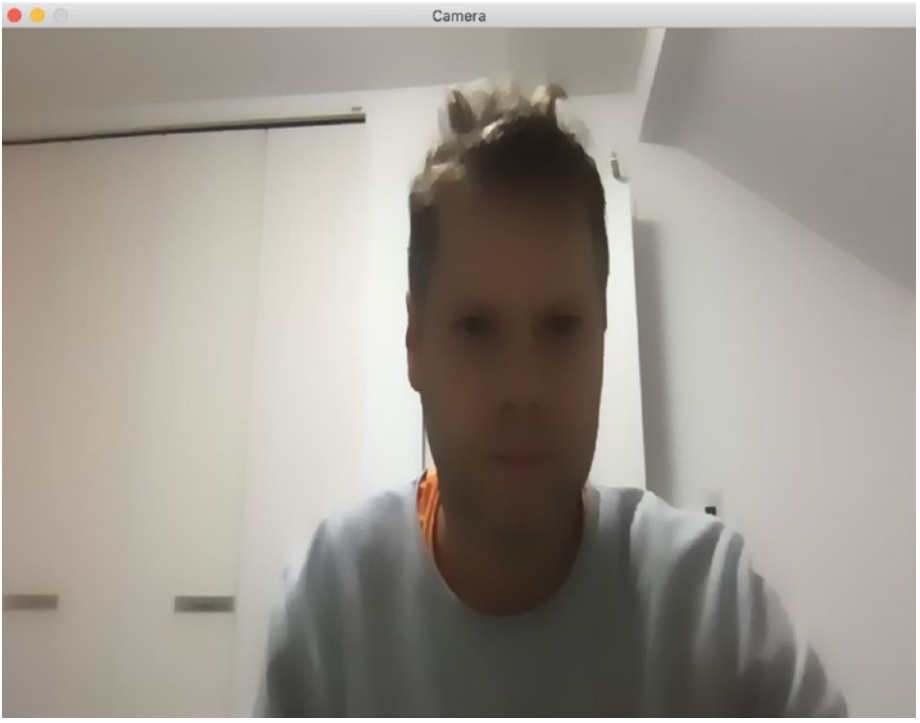

Figure 4-2. *Edge Preserving filter*

Canny

Another useful filter in the OpenCV world is to apply a Canny effect, which is a fast and effective way to find contours and shapes in a `Mat` object. A quick implementation of `Canny` as a filter is shown in Listing 4-7.

Listing 4-7. OpenCV's Canny in a Filter

```
class Canny implements Filter {
    public boolean inverted = true;
    public int threshold1 = 100;
    public int threshold2 = 200;
```

```
    @Override
    public Mat apply(Mat in) {
        Mat dst = new Mat();
        Imgproc.Canny(in, dst, threshold1, threshold2);
        if (inverted) {
            Core.bitwise_not(dst, dst, new Mat());
        }
        Imgproc.cvtColor(dst, dst, Imgproc.COLOR_GRAY2RGB);
        return dst;
    }
}
```

Figure 4-3 shows the result of applying the Canny filter to the main loop.

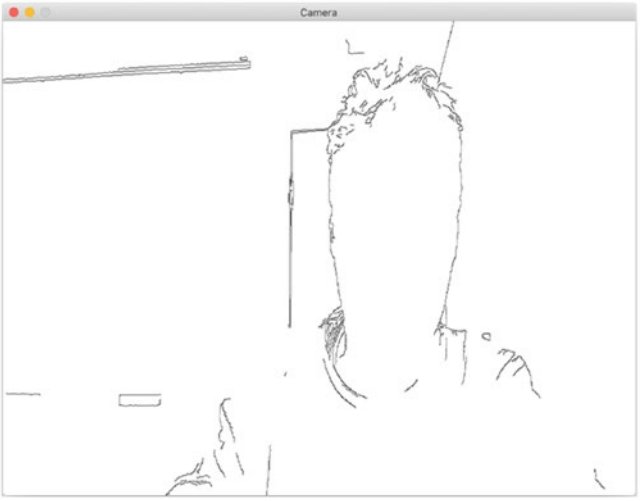

Figure 4-3. *Canny filter on webcam stream*

Debugging (Again)

Here we go again with some notes on debugging. If you add a breakpoint in the main capture loop of `WebcamWithFilters`, you'll get access to all the different fields of the filter. As shown in Figure 4-4, let's change the value of the inverted Boolean value to false.

Figure 4-4. *Updating the filter parameters in real time*

Then, let's remove the breakpoint and restart the code execution normally. Figure 4-5 shows how changing the value of the filter straightaway changed the running code and the color of the `Mat` object displaying on the screen.

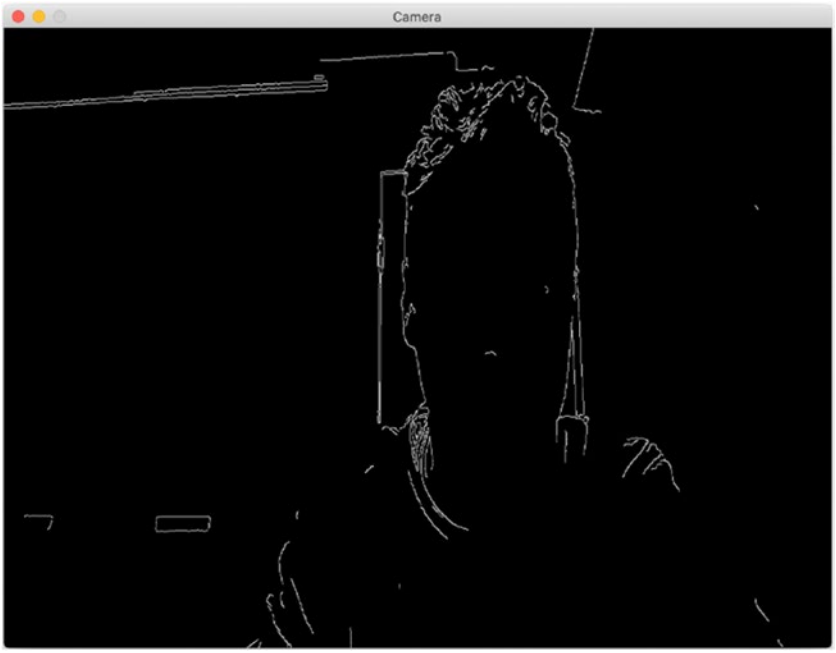

Figure 4-5. *Noninverted Canny filter*

When implementing your own filters, it's a good idea to keep the most influential variables as fields of the class so you're not flooded with values but still have access to the important ones.

Combining Filters

You probably realized in the previous sections that you will want to find out the performance of each filter.

Performance actually results from how many frames per seconds you can handle when reading from the video file or directly from the webcam device.

Here we're going to do the following:

- Display the frame rate directly on the image
- Combine a Gray filter with the Framerate filter using the `Pipeline` class we defined earlier in this chapter

We already have the code for the Gray filter, so we'll move directly to the code for displaying the frame rate per second (FPS).

I always thought I could access the value of the frame rate using OpenCV's set of properties on `VideoCapture`. Unfortunately, we're pretty much always stuck with a hard-coded value and not what is actually showing on the screen.

So, the implementation of FPS in Listing 4-8 is a small work-around that does some simple arithmetic based on how many frames have been displayed since the beginning of the filter's life.

Finally, it uses `putText` to apply the text directly onto the frame. It works well enough for simple use cases.

Listing 4-8. FPS Filter

```
class FPS implements Filter {

    long start = System.currentTimeMillis();
    int count = 0;

    Point org = new Point(50, 50);
    int fontFace = Imgproc.FONT_HERSHEY_PLAIN;
    double fontScale = 4.0;
    Scalar color = new Scalar(0, 0, 0);
    int thickness = 3;

    public Mat apply(Mat in) {
        count++;
        String text = "FPS: " + count / (1 + ((System.
        currentTimeMillis() - start) / 1000));
```

```
            Imgproc.putText(in, text, org, fontFace, fontScale,
            color, thickness);
            return in;
        }
}
```

Now let's get back to combining the FPS showing on the stream with the Gray filter.

You'll be happy to know that the only line you have to update in the `main()` function of `WebcamWithFilters` is the one where the filter is instantiated, as shown here:

```
Filter filter = new Pipeline(Gray.class, FPS.class);
```

When you run the sample again from your Raspberry Pi, you'll get something that looks like what's shown in Figure 4-6. Applying the two filters together, I usually get around 15 frames per second on the Raspberry Pi.

Figure 4-6. *Combined Gray filter with FPS*

Applying Instagram-like Filters

Enough serious work for now. Let's take a short break and have a bit of fun with some Instagram-like filters.

Color Map

Let's start this fun section by using OpenCV's `colormap` function from `ImgProc`. We move the parameter of the `colormap` to the constructor, as shown in Listing 4-9, so that we can update it via the debug screen.

Listing 4-9. Color Map

```
class Color implements Filter {

    int colormap = 0;

    public Color(int colormap) {
        this.colormap = colormap;
    }

    public Color() {
        this.colormap = Imgproc.COLORMAP_INFERNO;
    }

    public Mat apply(Mat img) {
        Mat threshed = new Mat();
        Imgproc.applyColorMap(img, threshed, colormap);
        return threshed;
    }
}
```

To instantiate the filter, we need to pass in the color map we want to use, so this is done directly in the constructor. Here we need to have the instantiated filter instead of just its class, so we use the second constructor of the `Pipeline` class, with the instantiated `Filter` objects passed to the constructor.

```
Filter filter = new Pipeline(new Color(Imgproc.COLORMAP_
INFERNO), new FPS());
```

When you execute this, you get something like Figure 4-7.

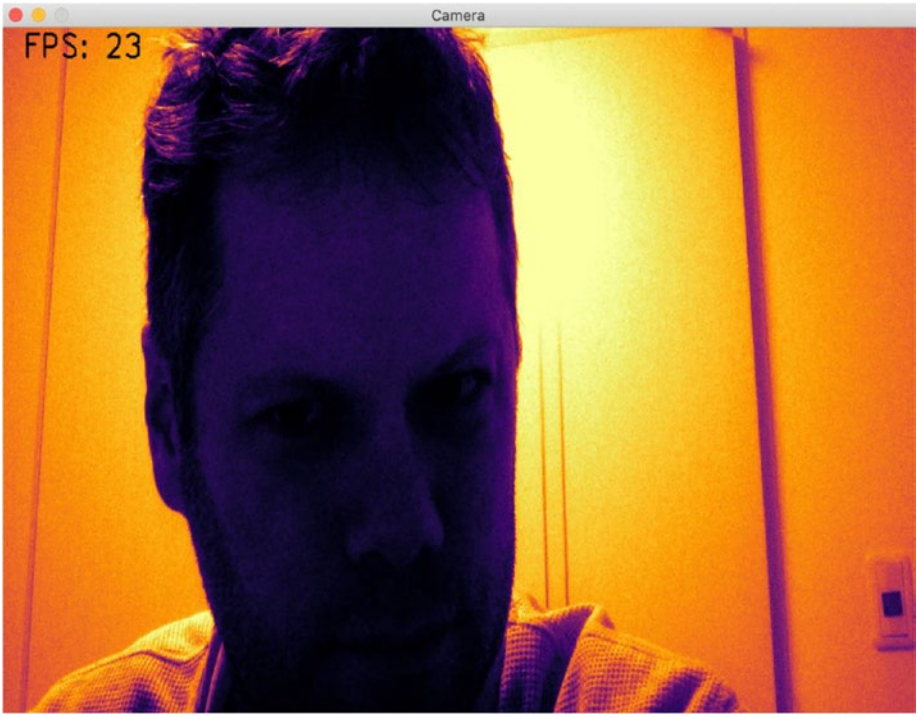

Figure 4-7. *Inferno*

Thresh

Thresh is another fun filter made by applying the `threshold` function of `Imgproc`. It applies a fixed-level threshold to each array element of the `Mat` object.

The original purpose of the Thresh filter was to segment elements of a picture, such as to remove the noise of a picture by removing unwanted elements. It is not usually used for Instagram filtering, but it does look nice and can give you some creative ideas.

Listing 4-10 shows how the Thresh filter can be implemented.

Listing 4-10. Applying Threshold

```
class Thresh implements Filter{

    int sensitivity = 100;
    int maxVal = 255;
    public Thresh() {

    }
    public Thresh(int _sensitivity) {
        this.sensitivity = _sensitivity;
    }

    public Mat apply(Mat img) {
        Mat threshed = new Mat();
        Imgproc.threshold(img, threshed, sensitivity, maxVal,
        Imgproc.THRESH_BINARY);
        return threshed;
    }
}
```

Figure 4-8 shows the result.

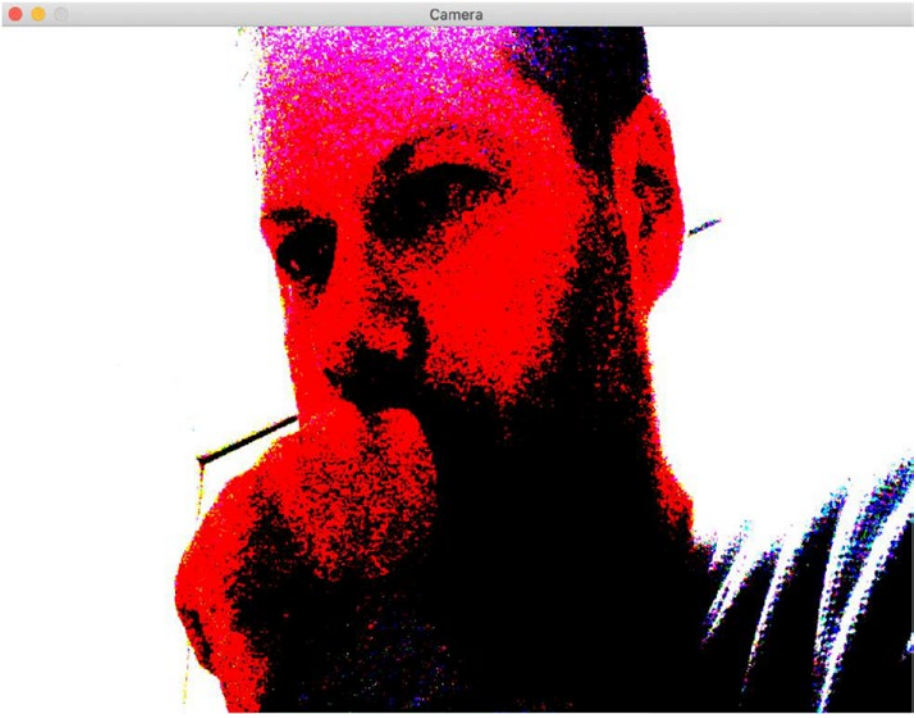

Figure 4-8. *Applying Thresh for a burning effect*

Sepia

Let's use the old (pun intended) Sepia effect again here, as implemented in Listing 4-11.

Listing 4-11. Sepia

```
class Sepia implements Filter {
    public Mat apply(Mat source) {
        Mat kernel = new Mat(3, 3, CvType.CV_32F);
```

```
            kernel.put(0, 0,
                    0.272, 0.534, 0.131,
                    0.349, 0.686, 0.168,
                    0.393, 0.769, 0.189);
            Mat destination = new Mat();
            Core.transform(source, destination, kernel);
            return destination;
        }
}
```

When used with the video stream, the Sepia effect gives you output that looks like Figure 4-9.

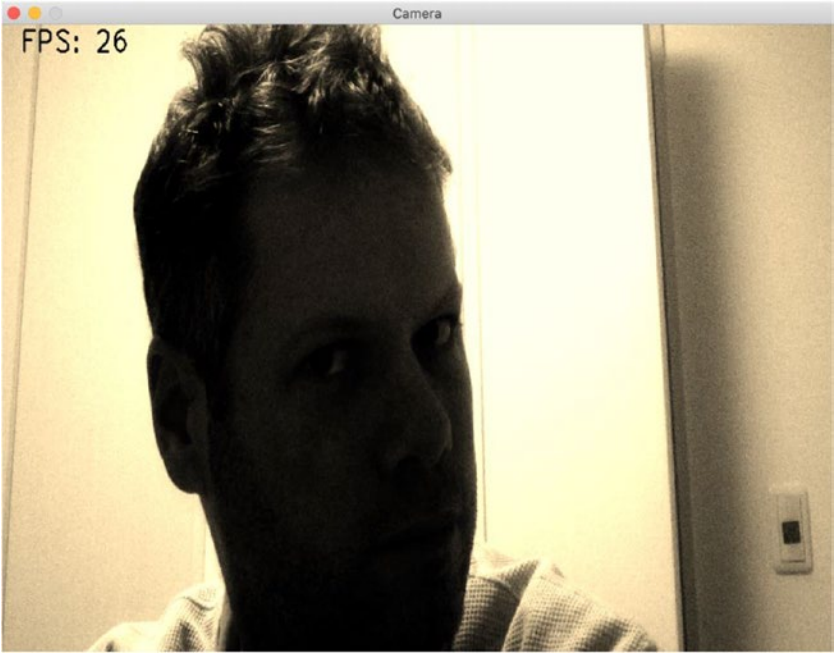

Figure 4-9. *Sepia effect*

Cartoon

This naïve implementation of a Cartoon effect takes the important feature-defining lines of the base picture and, after applying a smoothing effect and a blur effect, applies a threshold on each pixel value. Then it combines the result of those operations, as shown in Listing 4-12.

Listing 4-12. Cartoon Filter

```
class Cartoon implements Filter {

    public int d = 17;
    public int sigmaColor = d;
    public int sigmaSpace = 7;
    public int ksize = 7;

    public double maxValue = 255;
    public int blockSize = 19;
    public int C = 2;

    public Mat apply(Mat inputFrame) {
        Mat gray = new Mat();
        Mat co = new Mat();
        Mat m = new Mat();
        Mat mOutputFrame = new Mat();

        Imgproc.cvtColor(inputFrame, gray, Imgproc.COLOR_BGR2GRAY);
        Imgproc.bilateralFilter(gray, co, d, sigmaColor,
        sigmaSpace);
        Mat blurred = new Mat();
        Imgproc.blur(co, blurred, new Size(ksize, ksize));
        Imgproc.adaptiveThreshold(blurred, blurred, maxValue,
        Imgproc.ADAPTIVE_THRESH_MEAN_C, Imgproc.THRESH_BINARY,
                blockSize, C);
        Imgproc.cvtColor(blurred, m, Imgproc.COLOR_GRAY2BGR);
```

```
        Core.bitwise_and(inputFrame, m, mOutputFrame);
        return mOutputFrame;
    }
}
```

Applying the Cartoon filter on the video stream gives you an effect like in Figure 4-10.

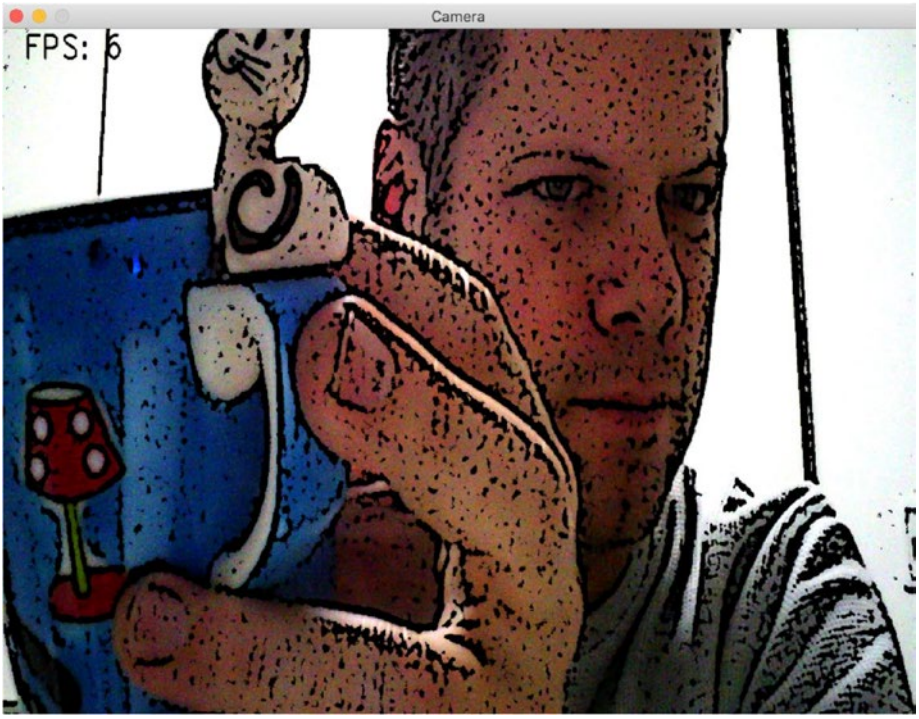

Figure 4-10. *Cartoon effect*

Pencil Effect

I love the Pencil effect, obtained by calling the `pencilSketch` method from the core OpenCV `Photo` class. Unfortunately, it is much too slow to apply in real time on the Raspberry Pi. It does give some pretty results with hardly any implementation effort, though. See for yourself in Listing 4-13.

Listing 4-13. Pencil Effect

```
class PencilSketch implements Filter {
    float sigma_s = 60;
    float sigma_r = 0.07f;
    float shade_factor = 0.05f;
    boolean gray = false;

    @Override
    public Mat apply(Mat in) {
        Mat dst = new Mat();
        Mat dst2 = new Mat();
        pencilSketch(in, dst, dst2, sigma_s, sigma_r,
        shade_factor);
        return gray ? dst : dst2;
    }
}
```

When this effect is applied, it gives you a result like the one shown in Figure 4-11.

Figure 4-11. *Pencil sketching*

Woo-hoo. That was quite a few effects ready to use and apply for leisure. There are a few others available in the origami repositories, and you can of course contribute your own, but for now, let's move on to serious object detection.

Performing Object Detection

Object detection is the concept of finding an object within a picture using different programming algorithms. It is a task that has long been done by human beings and that was quite hard for brain-less computers to do. But that has changed recently with advances in technology.

In this section of this chapter, we'll review different computer vision techniques to identify objects within an image, without any information about its content. Specifically, we'll review the following:

- Using a simple contours-drawing filter
- Detecting objects by colors
- Using Haar classifiers
- Using template matching
- Using a neural network like Yolo

The examples go in somewhat progressive order of difficulty, so it's best to try them in the order of this list.

Removing the Background

Removing the background is a technique you can use to remove unnecessary artifacts from a scene. The objects you are trying to find are probably not static and are likely to be moving across the set of pictures or video streams. To remove artefacts efficiently the algorithm needs to be able to differentiate two `Mat` objects and use some kind of short-term memory to differentiate moving things (in the foreground) from standard scene objects in the background.

In OpenCV, two easy-to-use `BackgroundSubtractor` classes are available for you to use. Their introduction and full explanation can be found on the following web site:

`https://docs.opencv.org/master/d1/dc5/tutorial_background_subtraction.html`

Basically, you give more and more frames to the background subtractor, which can then detect what is moving in the foreground and what is not.

Listing 4-14 is pretty easy to follow; just be careful not to mistake the `apply` function from the `subtractor` class with the one we have in the `Filter` interface.

Listing 4-14. BackgroundSubtractor Class

```
class BackgroundSubtractor implements Filter {
    boolean useMOG2 = true;
    BackgroundSubtractor backSub;
    double learningRate = 1.0;
    boolean showMask = true;

    public BackgroundSubtractor() {
        if (useMOG2) {
            backSub = Video.createBackgroundSubtractorMOG2();
        } else {
            backSub = Video.createBackgroundSubtractorKNN();
        }
    }

    @Override
    public Mat apply(Mat in) {
        Mat mask = new Mat();
        backSub.apply(in, mask);
        Mat result = new Mat();
        if (showMask) {
            Imgproc.cvtColor(mask, result, Imgproc.COLOR_GRAY2RGB);
            return result;
        } else {
            in.copyTo(result, mask);
            return result;
        }
    }
}
```

The filter is loaded by calling the constructor, and you should get a result similar to Figure 4-12.

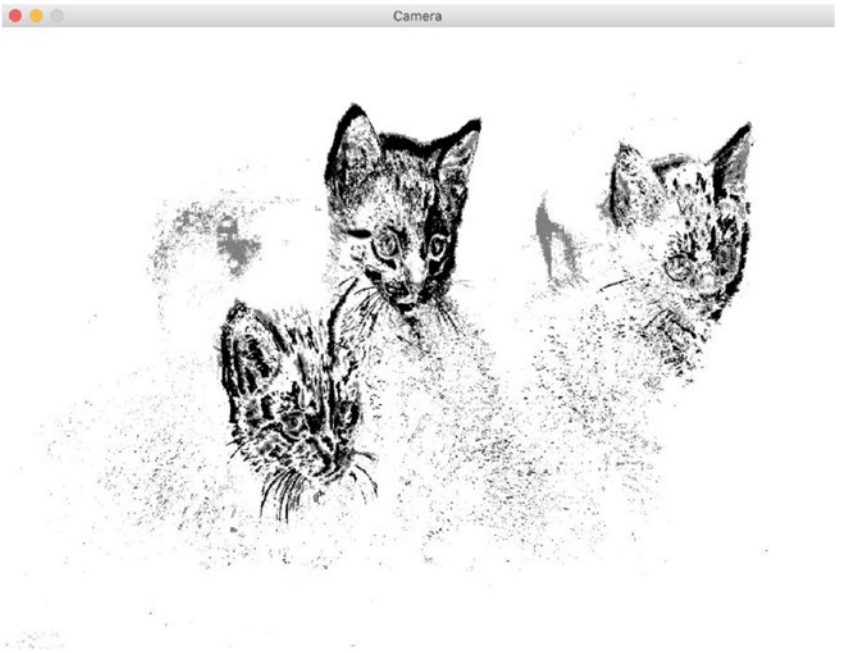

Figure 4-12. *Removing the background*

USE KNN BACKGROUND SUBTRACTOR

Once you have this filter running, try switching to the KNN-based `BackgroundSubtractor` and see the difference in speed (looking at the frame rate) and the accuracy of the results.

Detecting by Contours

The second most basic OpenCV feature is to be able to find the contours in an image. The Contours filter uses the `findContours` function from the `Imgproc` class.

`findContours` usually brings better results when you first do the following:

- Turn the input `Mat` object to gray
- Apply a Canny filter

Those two steps are added in Listing 4-15; then we draw the contours on a black `Mat` object, created with the `zeros` function.

Listing 4-15. Detecting Contours

```
class Contours implements Filter {
    private int threshold = 100;

    public Mat apply(Mat srcImage) {
        Mat cannyOutput = new Mat();
        Mat srcGray = new Mat();
        Imgproc.cvtColor(srcImage, srcGray, Imgproc.COLOR_
        BGR2GRAY);
        Imgproc.Canny(srcGray, cannyOutput, threshold,
        threshold * 2);

        List<MatOfPoint> contours = new ArrayList<>();
        Mat hierarchy = new Mat();
        Imgproc.findContours(cannyOutput, contours, hierarchy,
        Imgproc.RETR_TREE, Imgproc.CHAIN_APPROX_SIMPLE);

        Mat drawing = Mat.zeros(cannyOutput.size(), CvType.
        CV_8UC3);
```

```
        for (int i = 0; i < contours.size(); i++) {
            Scalar color = new Scalar(256, 150, 0);
            Imgproc.drawContours(drawing, contours, i, color,
            2, 8, hierarchy, 0, new Point());
        }

        return drawing;
    }

}
```

USE A PIPELINE FOR CONTOURS

The careful reader will notice that since the previous code is using the `Pipeline` class, it would actually be nicer if it were written like this:

```
Pipeline(new Canny(), new Gray(), new Contours())
```

with a Contours filter that just extracts contours. Try it!

Applying the Contours filter on a video of Marcel gives an artistic look (Figure 4-13).

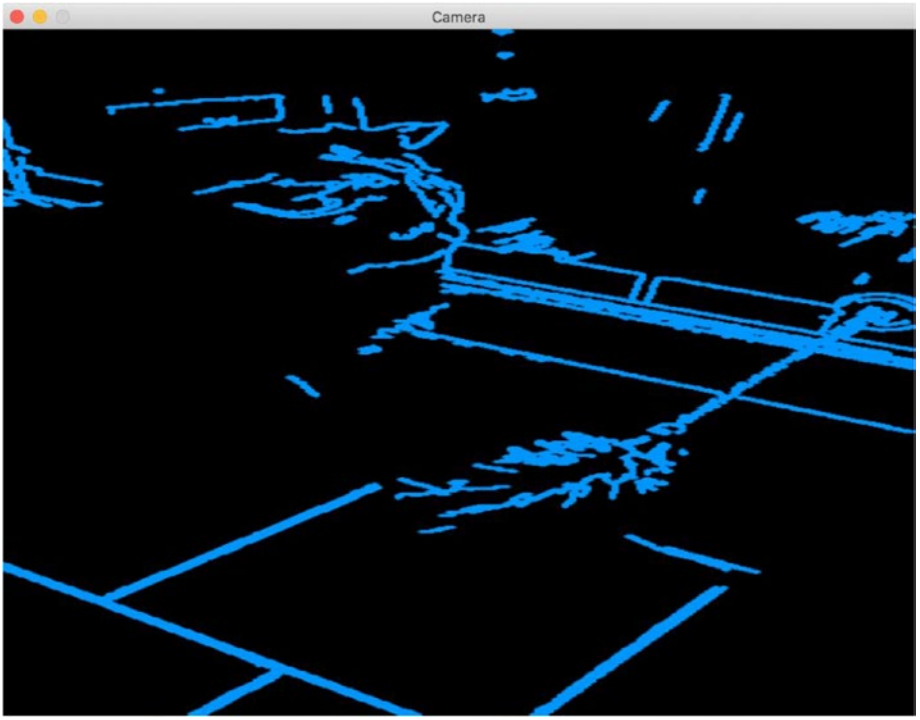

Figure 4-13. *Using OpenCV's detecting contours feature*

REMOVE THE FIRST CANNY FILTER AND COMPARE

Here's an exercise for you: try removing the two steps of converting to gray and applying the Canny filter and then compare the results to the original.

Detecting by Color

A picture, or a `Mat` object, in OpenCV is usually in a red/green/blue (RGB) color space (actually blue/green/red in OpenCV to be accurate). This is easy to understand if you think of it that each pixel is assigned a value for each channel. To see the possible values for those channels, you can review the following site:

`https://www.rapidtables.com/web/color/RGB_Color.html`

The problem with this color space is that the amount of luminosity and contrast are mingled with the amount of color itself.

When looking for specific colors in `Mat` objects, we switch to a color space named HSV (for hue, saturation, value). In this color space, the color directly translates to the Hue value.

The values for Hue are usually between 0 and 360, like the number of degrees in a cylinder. OpenCV has a slightly different scheme, with a range divided by 2 (so it takes less space in memory). Table 4-1 lists the Hue value ranges.

Listing 4-16 converts the color space and checks for Hue values in the range of the wanted color using `inRange`, with some added magic to draw the shapes properly using `findContours` again at the end of the example.

Table 4-1. *Hue Values in OpenCV*

Color	Hue Range
Red	0 to 30 *and* 150 to 180
Green	30 to 90
Blue	90 to 150

Listing 4-16. Detecting Red

```
class ColorDetector implements Filter {

    Scalar minColor, maxColor;

    public ColorDetector(Scalar minColor, Scalar maxColor) {
        this.minColor = minColor;
        this.maxColor = maxColor;
    }
```

```
@Override
public Mat apply(Mat input) {
    Mat array255 = new Mat(input.height(), input.width(),
    CvType.CV_8UC1);
    array255.setTo(new Scalar(255));
    Mat distance = new Mat(input.height(), input.width(),
    CvType.CV_8UC1);

    List<Mat> lhsv = new ArrayList<Mat>(3);
    Mat circles = new Mat();

    Mat hsv_image = new Mat();
    Mat thresholded = new Mat();
    Mat thresholded2 = new Mat();

    Imgproc.cvtColor(input, hsv_image, Imgproc.COLOR_BGR2HSV);
    Core.inRange(hsv_image, minColor, maxColor, thresholded);

    Imgproc.erode(thresholded, thresholded, Imgproc.
    getStructuringElement(Imgproc.MORPH_RECT, new Size(8, 8)));
    Imgproc.dilate(thresholded, thresholded, Imgproc.
    getStructuringElement(Imgproc.MORPH_RECT, new Size(8, 8)));

    Core.split(hsv_image, lhsv);
    Mat S = lhsv.get(1);
    Mat V = lhsv.get(2);
    Core.subtract(array255, S, S);
    Core.subtract(array255, V, V);
    S.convertTo(S, CvType.CV_32F);
    V.convertTo(V, CvType.CV_32F);
    Core.magnitude(S, V, distance);
```

```
        Core.inRange(distance, new Scalar(0.0),
        new Scalar(200.0), thresholded2);
        Core.bitwise_and(thresholded, thresholded2,
        thresholded);
        Imgproc.GaussianBlur(thresholded, thresholded,
        new Size(9, 9), 0, 0);
        List<MatOfPoint> contours = new ArrayList<MatOfPoint>();
        Imgproc.HoughCircles(thresholded, circles, Imgproc.
        CV_HOUGH_GRADIENT, 2, thresholded.height() / 8, 200,
        100, 0, 0);
        Imgproc.findContours(thresholded, contours,
        thresholded2, Imgproc.RETR_LIST, Imgproc.CHAIN_APPROX_
        SIMPLE);
        Imgproc.drawContours(input, contours, -2,
        new Scalar(10, 0, 0), 4);

        return input;
    }

}

class RedDetector extends ColorDetector {
    public RedDetector() {
        super(new Scalar(0, 100, 100), new Scalar(10, 255, 255));
    }
}
```

The result of applying this filter to a video of roses looks something like Figure 4-14.

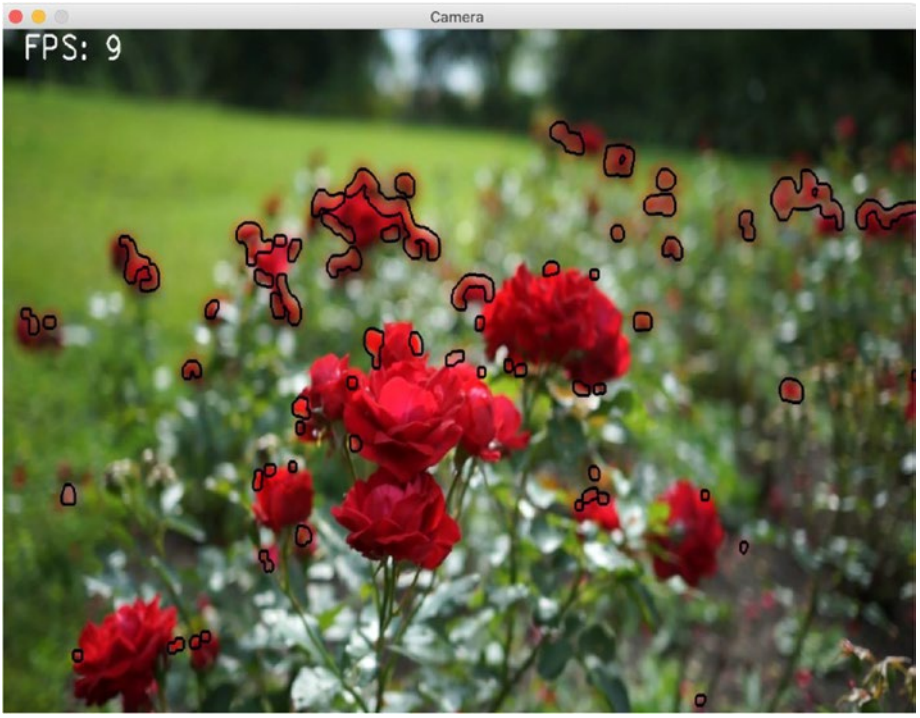

Figure 4-14. *Detecting red roses*

IMPLEMENT A DETECT BLUE FILTER

Looking at the values for Hue in Table 4-1, you can see it would not be so difficult to implement a filter that searches for blue colors. This is left as an exercise for you.

Detecting by Haar

As you saw in Chapter 1, you can use a Haar-based classifier to identify objects and/or people in a `Mat` object. The code is pretty much the same as you have seen, with an added emphasis on the number and size of shapes we are looking for.

Specifically, the following code shows how two sizes are used as parameters to specify the minimum size and the maximum size of the objects we are looking for.

```
classifier.detectMultiScale(input, faces, 1.1, 2, -1,
new Size(100, 100), new Size(500, 500));
```

So, Listing 4-17, with an extra main example function, shows how to use different XML files as parameters to the Haar classifier detection.

Listing 4-17. Haar Classifier–Based Detection

```
public class DetectWithHaar {

    public static void main(String[] args) {
        Origami.init();

        VideoCapture cap = new VideoCapture(0);

        Mat buffer = new Mat();
        ImShow ims = new ImShow("Camera", 800, 600);
        Filter filter = new Pipeline(new Haar("haarcascades/
        haarcascade_frontalface_default.xml"), new FPS());
        while (cap.grab()) {
            cap.retrieve(buffer);
            ims.showImage(filter.apply(buffer));
        }
        cap.release();
    }

}

class Haar implements Filter {

    private CascadeClassifier classifier;
    Scalar white = new Scalar(255, 255, 255);
```

```
    public Haar(String path) {
        classifier = new CascadeClassifier(path);
    }

    public Mat apply(Mat input) {
        MatOfRect faces = new MatOfRect();
        classifier.detectMultiScale(input, faces, 1.1, 2, -1,
        new Size(100, 100), new Size(500, 500));
        for (Rect rect : faces.toArray()) {
            Imgproc.putText(input, "Face", new Point(rect.x,
            rect.y - 5), 3, 5, white);
            Imgproc.rectangle(input, new Point(rect.x, rect.y),
            new Point(rect.x + rect.width, rect.y + rect.height),
                    white, 5);
        }
        return input;
    }
}
```

If you have a cat at home or are using a cat video from the examples, when you apply this to a webcam stream, you should get a result that looks like Figure 4-15.

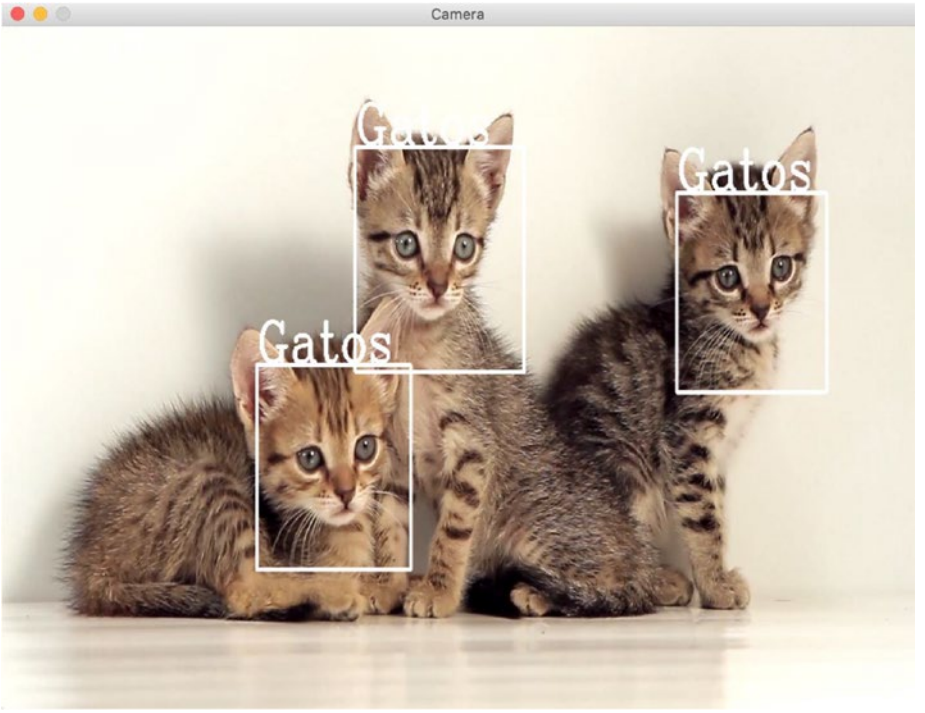

Figure 4-15. *Finding cats*

USING OTHER HAAR DEFINITIONS

There are other XML files for Haar cascades in the samples. Feel free to use one to detect people, eyes, or smiling faces as an exercise.

Transparent Overlay on Detection

When drawing the rectangles in the previous examples, you might have wondered whether it is possible to draw something other than rectangles on the detected shapes.

Listing 4-18 shows how to do this by loading a mask that will be overlaid at the location of the detected shape. This is pretty much what you use in your smartphone applications all the time.

Note that there is a trick here with the transparency layer in the `drawTransparency` function. The mask for the overlay is loaded using `IMREAD_UNCHANGED` as the loading flag; you have to use this or the transparency layer is lost.

Once you have the transparency layer, you then use it as a mask while copying the overlay so as to copy the exact pixels of the `Mat` object you want.

Listing 4-18. Adding an Overlay

```
class FunWithHaar implements Filter {

    CascadeClassifier classifier;
    Mat mask;
    Scalar white = new Scalar(255, 255, 255);

    public FunWithHaar(String path) {
        classifier = new CascadeClassifier(path);
        mask = Imgcodecs.imread("masquerade_mask.png",
        Imgcodecs.IMREAD_UNCHANGED);
    }

    void drawTransparency(Mat frame, Mat transp, int xPos, int
    yPos) {
        List<Mat> layers = new ArrayList<Mat>();
        Core.split(transp, layers);
        Mat mask = layers.remove(3);
        Core.merge(layers, transp);
        Mat submat = frame.submat(yPos, yPos + transp.rows(),
        xPos, xPos + transp.cols());
        transp.copyTo(submat, mask);
    }
```

```
    public Mat apply(Mat input) {
        MatOfRect faces = new MatOfRect();
        classifier.detectMultiScale(input, faces);
        Mat maskResized = new Mat();
        for (Rect rect : faces.toArray()) {
            Imgproc.resize(mask, maskResized, new Size
            (rect.width, rect.height));
            int adjusty = (int) (rect.y - rect.width * 0.2);
            try {
                drawTransparency(input, maskResized, rect.x,
                adjusty);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
        return input;
    }
}
```

Depending on the transparent `Mat` object used, you may need to adjust the location, but otherwise you should get something that looks like Figure 4-16. Finally you can add some Venice Carnival feeling to your video streams!

Figure 4-16. *Adding a mysterious mask as an overlay*

USE A BATMAN MASK

I actually tried and could not find a proper Batman mask to use as an overlay to the video stream. Maybe you can help me by sending me the code with an appropriate `Mat` overlay to use!

Detecting by Template Matching

Template matching with OpenCV is just plain simple. It's so simple that this should maybe have come earlier in the order of detection methods. Template matching means looking for a `Mat` within another `Mat`. OpenCV has a superpower function named `matchTemplate` that does this.

Listing 4-19 mostly revolves around using `matchTemplate`. Look for the usage of `Core.minMaxLoc` on the result received from `matchTemplate`. It is used to locate the index of the best score and will be used again when running neural networks.

Listing 4-19. Pattern Matching

```
class Template implements Filter {
    Mat template;

    public Template(String path) {
        this.template = Imgcodecs.imread(path);
    }

    @Override
    public Mat apply(Mat in) {
        Mat outputImage = new Mat();
        Imgproc.matchTemplate(in, template, outputImage,
        Imgproc.TM_CCOEFF);

        MinMaxLocResult mmr = Core.minMaxLoc(outputImage);
        Point matchLoc = mmr.maxLoc;

        Imgproc.rectangle(in, matchLoc, new Point(matchLoc.x +
        template.cols(), matchLoc.y + template.rows()),
                new Scalar(255, 255, 255), 3);

        return in;
    }

}
```

Now, let's find a box containing a ReSpeaker like the one shown in Figure 4-17 because we are going to need this speaker in the next chapter, and I just can't find it right now. Let's use OpenCV to find it for us.

Figure 4-17. *The template*

Detecting through OpenCV's template matching is surprisingly fast and accurate, as shown in Figure 4-18.

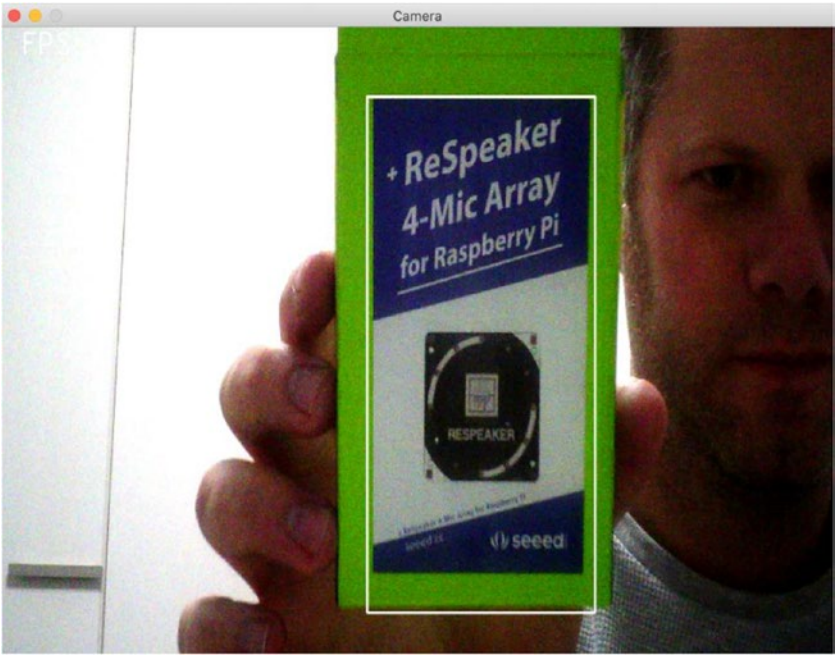

Figure 4-18. *Finding the box for the speaker*

It's a bit hard to see the frame rate in Figure 4-18, but it is actually around 10 to 15 frames per second on the Raspberry Pi 4.

Detecting by Yolo

This is the final detection method presented in this chapter. Let's say we want to apply a trained neural network to identify objects in a stream. After some testing on the hardware with little computing power, I got quite speedy results with Yolo/Darknet and the freely available Darknet networks trained on the Coco dataset.

The advantage of using neural networks on random inputs is that most of the trained networks are quite resilient and give good results, with 80 percent to 90 percent accuracy on close to real-time streams.

Training is the hardest part of using neural networks. In this book, we'll restrict ourselves to running detection code on the Raspberry Pi, not training. You can find the steps for how to organize your pictures for re-training networks on the Darknet/Yolo web site.

The sequence of steps to achieve object detection with Darknet in OpenCV are as follows:

1. Load a network from its configuration file and weight file.
2. Find the output layers/nodes of that network, because this is where the results will be. The output layers are the layers that are not connected to more output layers.
3. Transform a `Mat` object into a blob for the network. A blob is an image, or a set of images, tuned to match the format expected by the network in terms of size, channel orders, etc.
4. We then run the network, meaning we feed it the blob and retrieve the values for the layers marked as output layers.
5. For each line in the results, we actually get a confidence value for each of the expected possible recognizable features. In Coco, the network is trained to be able to recognize 80 different possible objects, such as people, bicycles, cars, etc.
6. We then use `MinMaxLocResult` again to get the index for the most probable recognized object, and if the value for that index is more than 0, we keep it.

7. The first four values in each result line are actually the four values describing a box where the detected object was found, so we extract those four values and keep the rectangle and the index for its label.

8. Before drawing all the boxes, we also usually make use of `NMSBoxes`, which removes overlapping boxes. Most of the time overlapping boxes are multiple versions of the same positive detection on the same object.

9. Finally, we draw the remaining rectangle and add the label for the recognized object.

Listing 4-20 shows the full code for the base `YoloDetector` implemented as a filter.

Listing 4-20. Neural Network–Based Detection

```
class YoloDetector implements Filter {
    final static Size sz = new Size(416, 416);
    List<String> outBlobNames;
    Net net;
    List<String> layers;
    List<String> labels;

    List<String> getOutputsNames(Net net) {
        List<String> layersNames = net.getLayerNames();
        return net.getUnconnectedOutLayers().toList().stream().
        map(i -> i - 1).map(layersNames::get)
                .collect(Collectors.toList());
    }

    public YoloDetector(String modelWeights, String
    modelConfiguration) {
```

```
    net = Dnn.readNetFromDarknet(modelConfiguration,
    modelWeights);
    layers = getOutputsNames(net);

    try {
        labels = Files.readAllLines(Paths.get(LABEL_FILE));
    } catch (Exception e) {
        throw new RuntimeException(e);
    }

}

@Override
public Mat apply(Mat in) {
    findShapes(in);
    return in;
}

final int IN_WIDTH = 416;
final int IN_HEIGHT = 416;
final double IN_SCALE_FACTOR = 0.00392157;
final int MAX_RESULTS = 20;
final boolean SWAP_RGB = true;
final String LABEL_FILE = "yolov3/coco.names";

void findShapes(Mat frame) {

    Mat blob = Dnn.blobFromImage(frame, IN_SCALE_FACTOR,
    new Size(IN_WIDTH, IN_HEIGHT), new Scalar(0, 0, 0),
             SWAP_RGB);
    net.setInput(blob);

    List<Mat> outputs = new ArrayList<>();
    for (int i = 0; i < layers.size(); i++)
        outputs.add(new Mat());
```

```
        net.forward(outputs, layers);
        postprocess(frame, outputs);
    }

    private void postprocess(Mat frame, List<Mat> outs) {

        List<Rect> tmpLocations = new ArrayList<>();
        List<Integer> tmpClasses = new ArrayList<>();
        List<Float> tmpConfidences = new ArrayList<>();

        int w = frame.width();
        int h = frame.height();

        for (Mat out : outs) {

            final float[] data = new float[(int) out.total()];
            out.get(0, 0, data);

            int k = 0;
            for (int j = 0; j < out.height(); j++) {

                Mat scores = out.row(j).colRange(5, out.width());
                Core.MinMaxLocResult result = Core.
                minMaxLoc(scores);
                if (result.maxVal > 0) {
                    float center_x = data[k + 0] * w;
                    float center_y = data[k + 1] * h;
                    float width = data[k + 2] * w;
                    float height = data[k + 3] * h;
                    float left = center_x - width / 2;
                    float top = center_y - height / 2;

                    tmpClasses.add((int) result.maxLoc.x);
                    tmpConfidences.add((float) result.maxVal);
                    tmpLocations.add(new Rect((int) left, (int)
                    top, (int) width, (int) height));
```

```
                }
                k += out.width();
            }
        }

        annotateFrame(frame, tmpLocations, tmpClasses,
        tmpConfidences);
    }

    private void annotateFrame(Mat frame, List<Rect>
    tmpLocations, List<Integer> tmpClasses,
            List<Float> tmpConfidences) {

        MatOfRect locMat = new MatOfRect();
        MatOfFloat confidenceMat = new MatOfFloat();
        MatOfInt indexMat = new MatOfInt();

        locMat.fromList(tmpLocations);
        confidenceMat.fromList(tmpConfidences);

        Dnn.NMSBoxes(locMat, confidenceMat, 0.1f, 0.1f, indexMat);

        for (int i = 0; i < indexMat.total() && i < MAX_
        RESULTS; ++i) {
            int idx = (int) indexMat.get(i, 0)[0];
            int labelId = tmpClasses.get(idx);
            Rect box = tmpLocations.get(idx);
            String label = labels.get(labelId);
            annotateOne(frame, box, label);
        }
    }

    private void annotateOne(Mat frame, Rect box, String label) {
        Imgproc.rectangle(frame, box, new Scalar(0, 0, 0), 2);
```

```
        Imgproc.putText(frame, label, new Point(box.x, box.y),
        Imgproc.FONT_HERSHEY_PLAIN, 4.0, new Scalar(0, 0, 0), 3);
    }

}
```

You can now run your own set of object detections and experiments using the different available networks. Listing 4-21 shows how to load each of the main Yolo-based networks.

Listing 4-21. Java Classes and Constructors for the Different Yolo Networks

```
class Yolov2 extends YoloDetector {

    public Yolov2() {
        super("yolov2/yolov2.weights", "yolov2/yolov2.cfg");
    }
}

class TinyYolov2 extends YoloDetector {

    public TinyYolov2() {
        super("yolov2-tiny/yolov2-tiny.weights", "yolov2-tiny/
        yolov2-tiny.cfg");
    }
}

class Yolov3 extends YoloDetector {

    public Yolov3() {
        super("yolov3/yolov3.weights", "yolov3/yolov3.cfg");
    }
}
```

```
class TinyYolov3 extends YoloDetector {

    public TinyYolov3() {
        super("yolov3-tiny/yolov3-tiny.weights", "yolov3-tiny/
        yolov3-tiny.cfg");
    }
}
```

Running on the Raspberry Pi, Yolo v3 can detect cars and people on a busy street of Lisbon (Figure 4-19) and more cats (Figure 4-20).

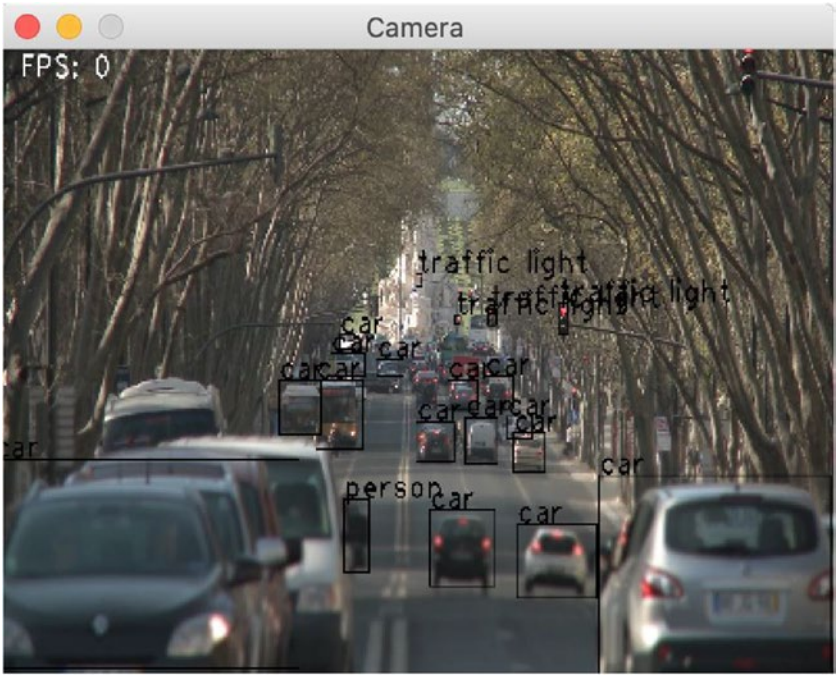

Figure 4-19. *Yolo v3 detecting cars and people*

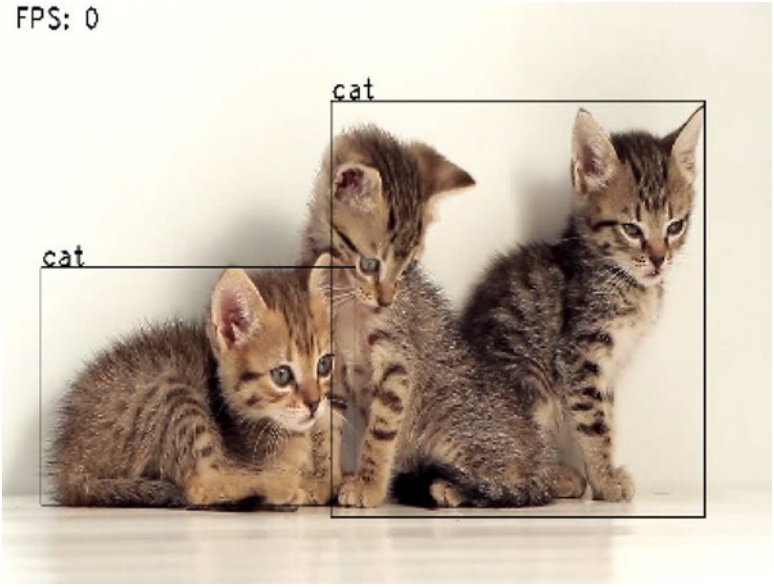

Figure 4-20. *Yolo v3 detecting cats*

As you can see, the frame rate was actually very low for standard Yolo v3.

When trying this experiment with Yolo v3 Tiny, you can actually get close to 5 to 6 frames per second, which is still slightly below real time, but it still gives results with very good accuracy. See Figure 4-21.

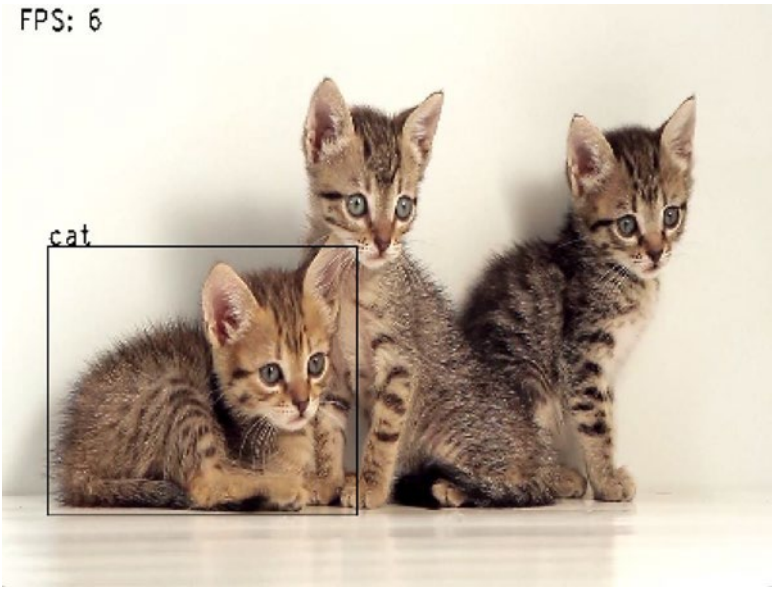

Figure 4-21. *TinyYoloV3 detecting cats*

> *You know my method. It is founded upon the observation of trifles.*
>
> —Arthur Conan Doyle,
> "The Boscombe Valley Mystery" (1891)

You're down to the last few lines of this chapter, and it has been a long ride where you saw most of the object detection concepts with OpenCV in Java on the Raspberry Pi.

Namely, you learned about the following:

- How to set up the Raspberry Pi for real-time object detection programming
- How to use filters and pipelines to perform image and real-time video processing

- How to implement some basic filters for `Mat`, used directly on real-time video streams from external devices and file-based videos
- How to add some fun with Instagram-like filters
- How to implement numerous object detection techniques using filters and pipelines
- How to run a neural network trained on the Coco dataset that can be used in real time on the Raspberry Pi

In the next chapter, you'll be introduced to Rhasspy, a voice recognition system, to connect the concepts presented in this chapter and apply them to home, office, or cat house automation.

CHAPTER 5

Vision and Home Automation

> *Home is a place you grow up wanting to leave and grow old wanting to get back to.*
>
> —John Ed Pearce

The first four chapters of this book showed you how to bring in video streams of all sorts and analyze them first on a computer and then on the small limited device that is the Raspberry Pi.

This last chapter closes the gap while opening up the possibilities by linking the video part to the voice part.

With the advent of voice platforms such as Apple Siri, Google Assistant, Amazon Alexa, and Microsoft Cortona, voice-assisted services are popping up everywhere. Humans are language creatures; we have to express ourselves. Drawing is also an option to interact with devices, like was done in the movie *Minority Report*, but to be honest, you need pretty big and expensive screens to do something like that. Voice works everywhere, and we don't need expensive tooling to use it. Using voice is intuitive and empowering.

I personally have many examples of how voice assistants could be used in everyday life. Start by entering a coffee shop and asking for a "double-shot cappuccino with cinnamon on top, on the front terrace, please", and then it is automatically made and delivered to you while you're sitting in the sun outside.

N. Modrzyk, *Real-Time IoT Imaging with Deep Neural Networks*,
https://doi.org/10.1007/978-1-4842-5722-7_5

Here's another simple example: I wish it were possible to go to an ATM and say, "Please withdraw $100 from my main account, in five notes of $20." The voice assistant recognizes your voice so you do not need to scroll through endless confirmation screens; you just get your bills. (I know, with all the virtual currencies nowadays, you don't really even need to go to the ATM anymore, do you?)

Those are some day-to-day examples, but you can also look to science-fiction movies to see voice-controlled spaceships as well as coffee machines. This is big.

So, why don't we use one of the existing software assistants for voice recognition, like Siri? Well, all of the big players basically control all of your user data and can access whatever data is going through their pipelines at any time. This can be annoying as an end user—just about as annoying when creating and engineering your own solutions. But, as much as possible, you want to be able to control where your data flows or, indeed, doesn't flow.

In this chapter, we thus introduce Rhasspy (`https://github.com/synesthesiam/rhasspy`).

Rhasspy was created for advanced users who want to have a voice interface to a home assistant but who value privacy and freedom. Rhasspy is free/open source and can do the following:

- Function completely disconnected from the Internet
- Work well with Home Assistant, Hass.io, and Node-RED

In this chapter, we'll do the following:

- We will install and learn how to interact with the MQTT protocol and Mosquitto, a broker that can be used to send recognized messages from Rhasspy.
- We will set up the Rhasspy interface to listen to intents and view those messages in the MQTT queue.

- We will integrate those messages with the content of the previous chapters and run real-time video analysis, updating the object detection parameters and using the voice commands received from Rhasspy.

Rhasspy Message Flow

Basically, Rhasspy continuously looks for "wake-up words." It wakes up when it's called by one of these words and records the next sentence and turns it into words. Then it analyzes the sentence against what it can recognize. Finally, if the sentence matches against known sentences, it returns a result with probabilities.

From an application point of view, the main thing you need to interact with when building a voice application is the messaging queue, which in most IoT cases is an MQTT server. MQTT was developed to focus on telemetry measures and thus had to be lightweight and as close to real time as possible. Mosquitto is an open source implementation of the MQTT protocol; it is lightweight and secure and was designed specifically to service IoT applications and small devices.

To do voice recognition, you start by creating Rhasspy commands, or *intents*, with a set of sentences. Each sentence is of course a set of words and can contain one or more variables. Variables are words that can be replaced by others and are assigned corresponding values once the voice detection has been performed by the voice engine.

In a simple weather service example, the following sentence:

> What is the weather *tomorrow*?

would be defined as follows:

> What is the weather <when:=datetime>?

Here, `when` is of type `datetime` and could be anything like *today*, *tomorrow*, *this morning*, or *the first Tuesday of next month*.

"What is the weather" is fixed within the context of that command and will never change. But the `<datetime>` location is a variable, so we give the engine hints on its type for better recognition.

Once an intent is recognized, Rhasspy posts a JSON message in a Mosquitto topic associated with the intent.

The example we will build in this chapter is to ask the application to highlight certain objects detected in the video stream.

For example, in the following sentence, where *cats* is the variable container, we specify which object to detect:

> Show me only *cats*!

In this example, when the intent has been recognized, the message sent will be a JSON-based message with the following main sections:

- Some message headers, notably the recognized input.
- Automatic speech recognition (ASR) tokens, or the confidence of each word of the sentence.
- The overall ASR confidence.
- The intent that was recognized and its confidence score.
- The value for each of the "slots," the variable defined within the sentence, and their associated confidence scores.
- Alternatives intents are also provided, although I found them to not be very useful in most contexts; therefore, looking for second best choice and trying to recover the flow is usually not worth it.

Listing 5-1 shows a sample of the JSON message.

Listing 5-1. JSON Message

```
 {
"sessionId": "9d355e0e-218b-4efa-bf36-9c8b13a7df42",
...
"input": "show me only cats",
"asrTokens": [
    [
        {
            "value": "show",
            "confidence": 1,
            "rangeStart": 0,
            "rangeEnd": 4,
            "time": {
                "start": 0,
                "end": 0.98999995
            }
        },
        ...
    ]
],
"asrConfidence": 0.9476952,
"intent": {
    "intentName": "hellonico:highlight",
    "confidenceScore": 1
},
"slots": [
    {
        "rawValue": "cats",
        "value": {
            "kind": "Custom",
            "value": "cats"
```

```
            },
            "alternatives": [],
            "range": {
                "start": 13,
                "end": 17
            },
            "entity": "string",
            "slotName": "object",
            "confidenceScore": 0.80663073
        }
    ],
    "alternatives": [
        {
            "intentName": "hellonico:hello",
            "confidenceScore": 0.28305322,
            "slots": []
        },
       ...
    ]
}
```

JSON messages from intents are sent to a given named message queue, one queue per intent, in the MQTT broker. The queue name used to send the intent message follows this pattern:

```
hermes/intent/<appname>:<intentname>
```

So, this is what it looks like in our example:

```
hermes/intent/hellonico:highlight
```

To summarize, Figure 5-1 shows a simple message flow.

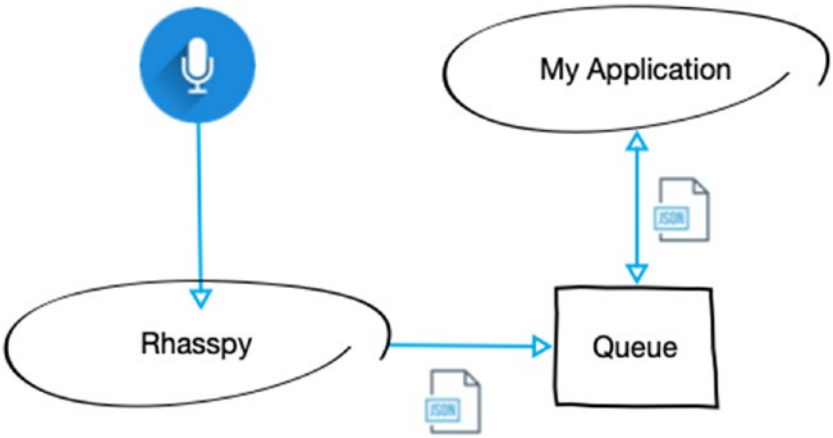

Figure 5-1. *Rhasspy message flow*

Since the main point of interaction for the system is the system queue, let's see first how we can install the queue broker and then interact with it.

MQTT Message Queues

To be able to use the message queue and receive messages from Rhasspy, we will install the message broker Mosquitto, one of the most widely used MQTT brokers. MQTT is a lightweight and energy-efficient protocol, with different levels of quality of service that can be set at the message level. Mosquitto is a very lightweight implementation of MQTT.

Installing Mosquitto

Mosquitto install instructions are available for every platform, and the download page has the links to the installers.

`https://mosquitto.org/download/`

On Windows, you should download the `.exe`-based installer, but other platforms have software available through the usual package managers, as shown here:

```
# on mac os
brew install mosquitto
# on debian/ubuntu
apt install mosquitto
# with snap
snap install mosquitto
```

Once Mosquitto is installed, you should check that the Mosquito service has started properly and is ready to relay messages. For example, on a Mac, use this:

```
$ brew services list mosquitto
```

```
Name          Status    User    Plist
mosquitto     started   niko    /Users/niko/Library/LaunchAgents/
                                homebrew.mxcl.mosquitto.plist
```

On the command line, you already have two commands available to you: one to publish messages on given topics and one to subscribe to messages on given topics.

Comparison of Other MQTT Brokers

For your reference, you can find a comparison of a few other MQTT brokers here: https://github.com/mqtt/mqtt.github.io/wiki/server-support.

RabbitMQ is a strong open source contender, but while clustering support is definitely robust, the setup is not quite easy.

MQTT Messages on the Command Line

Once Mosquitto is installed, on the command line it's rather easy to send messages about any topic to the broker using the `mosquitto_pub` command. For example, the following command sends the message "I am a MQTT message" on the topic `hello` to the broker located on host 0.0.0.0:

```
mosquitto_pub -h 0.0.0.0 -t hello -m "I am a MQTT message"
```

You, of course, have an equivalent subscriber command with `mosquitto_sub`, as follows:

```
$ mosquitto_sub -h 0.0.0.0 -t hello
I am a MQTT message
```

MESSAGE TO RASPBERRY

It's good practice to start the queue on the Raspberry Pi and read messages from the computer, or vice versa.

The main problem is simply in knowing the IP address or the hostname of the target machine, computer, or Raspberry, but sending messages is done in the same way using the `mosquitto_pub` and `mosquitto_sub` commands.

As an extra step, you could even run your Mosquitto MQTT broker in the cloud, for example, creating and integrating a queue from a service like https://www.cloudmqtt.com/.

Usually, my favorite graphical way to see messages is to use MQTT Explorer, a graphical client to MQTT, as shown in Figure 5-2.

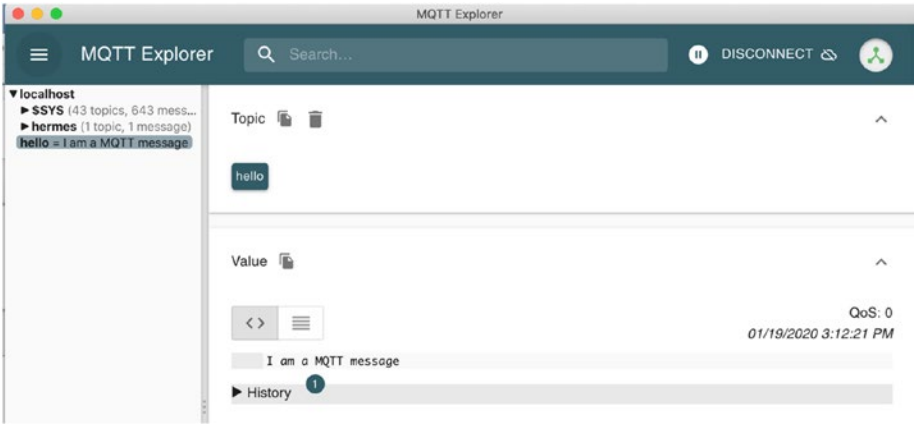

Figure 5-2. *MQTT Explorer, I am a MQTT message*

You can download MQTT Explorer from the following location:

`https://github.com/thomasnordquist/MQTT-Explorer/releases`

It should also be available through your favorite package manager.

Let's get back to sending the full equivalent of an intent message. You can use the command line to send a JSON file directly to the target intent queue, which as we have just seen is as follows: `hermes/intent/hellonico:highlight`.

So, using `mosquitto_pub`, this gives you the following:

```
mosquitto_pub -f onlycats.json -h 0.0.0.0 -t hermes/intent/
hellonico:highlight
```

where `onlycats.json` is the JSON file with the content we have just seen in Listing 5-1.

If you still have MQTT Explorer running, you will see the message in the corresponding queue, as shown in Figure 5-3.

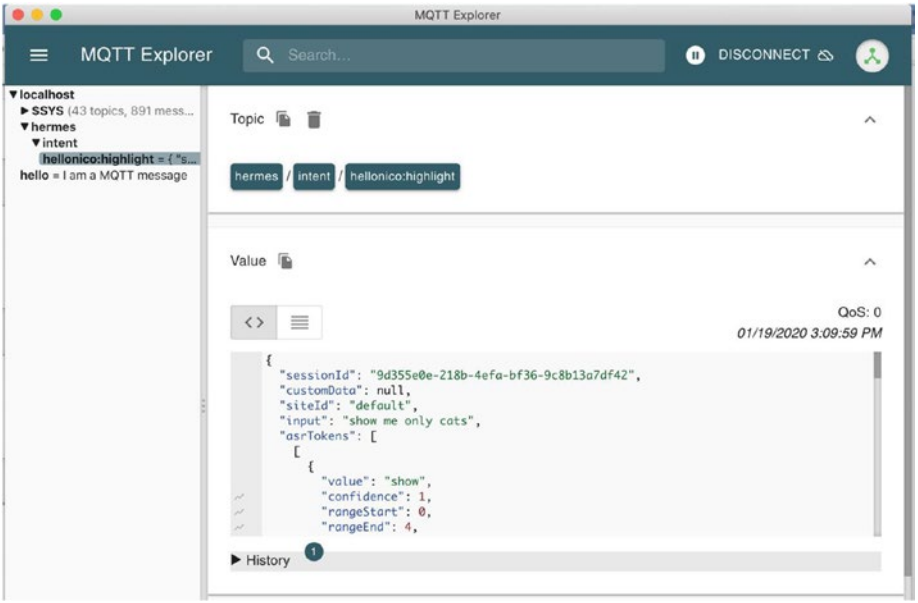

Figure 5-3. *Sending Rhasspy-like intent messages from the command line*

We have seen how to interact, publish, and subscribe to messages with MQTT from the command line. Let's do the same using Java now.

MQTT Messaging in Java

In this section, we will send and receive messages to and from our Java/Visual Studio Code setup. This will help you understand the message flow more easily.

Dependencies Setup

Either using the project from previous chapters or creating a new one based on the project template, we are going to add some Java libraries to interact with the MQTT messaging queue and to parse JSON content.

The dependencies section in the `pom.xml` file should look like Listing 5-2.

Listing 5-2. Java Dependencies

```
<dependencies>
<dependency>
  <groupId>org.eclipse.paho</groupId>
  <artifactId>org.eclipse.paho.client.mqttv3</artifactId>
  <version>1.1.0</version>
 </dependency>

 <dependency>
  <groupId>origami</groupId>
  <artifactId>origami</artifactId>
  <version>4.1.2-5</version>
 </dependency>

 <dependency>
  <groupId>org.json</groupId>
  <artifactId>json</artifactId>
  <version>20190722</version>
 </dependency>

 <dependency>
  <groupId>origami</groupId>
  <artifactId>filters</artifactId>
  <version>1.3</version>
 </dependency>
</dependencies>
```

Basically, we will make use of the following third-party libraries:

- `origami` and `origami-filters` for real-time video processing

- `org.json` for JSON content parsing
- `mqttv3` for interacting with the MQTT broker and handling messages

Sending a Basic MQTT Message

Since I'm writing part of this book on a plane flying over Russia, we will send a quick "hello" message to our fellow Russian spies using the MQTT protocol.

To do this, we will go through the following steps:

1. Create an MQTT client object, connecting to the host where the broker is running.
2. Use that object to connect to the broker using the connect method.
3. Create an MQTT message with a payload made of a string converted to bytes.
4. Publish the message.
5. Finally, cleanly disconnect.

Listing 5-3 shows the rather short code snippet.

Listing 5-3. Hello, Russia

```
package practice;

import org.eclipse.paho.client.mqttv3.MqttClient;
import org.eclipse.paho.client.mqttv3.MqttMessage;

public class MqttZero {
    public static void main(String... args) throws Exception {
        MqttClient client = new MqttClient("tcp://
        localhost:1883", MqttClient.generateClientId());
```

```
        client.connect();
        MqttMessage message = new MqttMessage();
        message.setPayload(new String("good morning Russia").
        getBytes());
        client.publish("hello", message);
        client.disconnect();
    }
}
```

To check that things are in place, verify that the message is showing in your running MQTT Explorer instance, as shown in Figure 5-4.

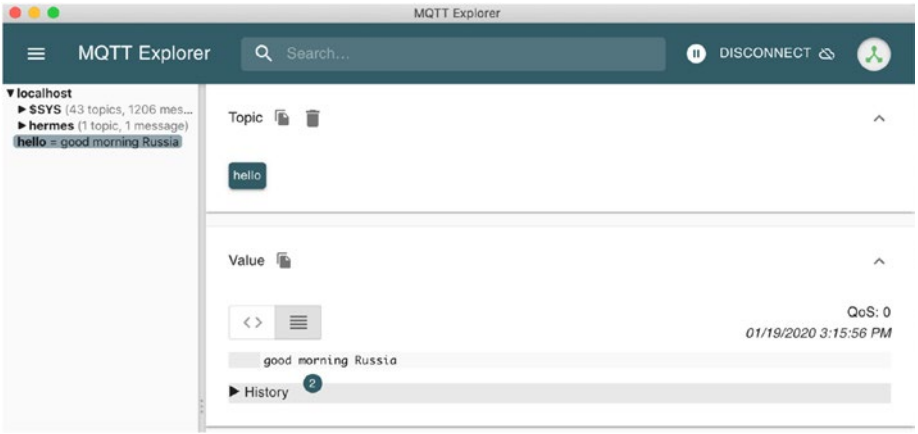

Figure 5-4. *MQTT Explorer in Russia*

Note that you should of course be listening to the `hello` topic first.

Simulating a Rhasspy Message

Sending the equivalent of a Rhasspy message is not going to be that much more difficult; we just read the content of the JSON file into a string and then send it like we just did with a simple string, as shown in Listing 5-4.

Listing 5-4. Intent message handling from Java

```
package practice;

import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.List;
import java.util.stream.Collectors;

import org.eclipse.paho.client.mqttv3.MqttClient;
import org.eclipse.paho.client.mqttv3.MqttMessage;

public class Client {
    public static void main(String... args) throws Exception {
        MqttClient client = new MqttClient("tcp://
        localhost:1883", MqttClient.generateClientId());
        client.connect();

        List<String> nekos = Files.readAllLines(Paths.
        get("onlycats.json"));
        String neko = nekos.stream().collect(Collectors.
        joining("\n"));
        MqttMessage message = new MqttMessage();
        message.setPayload(neko.getBytes());
        client.publish("hermes/intent/hellonico:highlight",
        message);
        client.disconnect();

    }
}
```

Our never-ending love of cats is again showing in Figure 5-5 in the MQTT Explorer window.

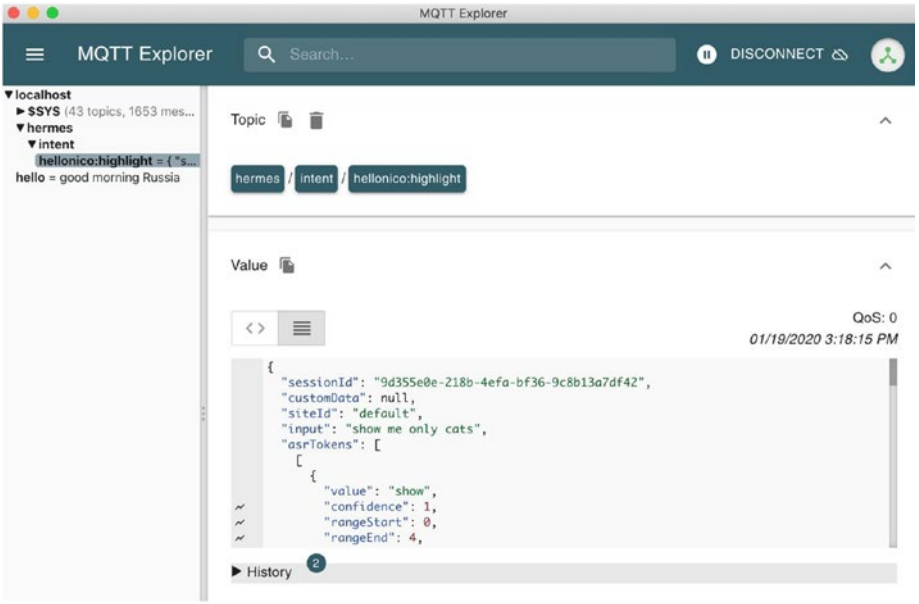

Figure 5-5. *JSON files showing in the MQTT Explorer window*

JSON Fun

Before being able to handle Rhasspy messages from the MQTT topic, we need to be able to do a bit of JSON parsing.

Obviously, there are many ways to do parsing in Java; we can parse manually (hint, don't do this) or use different libraries. Here we are going to use the `org.json` library because it's pretty fast at handling incoming MQTT messages.

Here are the steps for parsing a string to get the wanted value with the `org.json` library:

1. Create a `JSONObject` using the string version of the JSON message.

2. Use one of the following functions to navigate the JSON document:
   - get(),
   - getJSONObject()
   - getJSONArray()
3. Get the wanted value with `getInt`, `getBoolean`, `getString`, etc.

OVERENGINEERING

Most of the time, Java is seen as complicated because somewhere in the layers a complicated piece of middleware was introduced. I actually find the language pretty succinct these days and fast to work with given all the autocompletion and refactoring tooling.

Anyway, here we are just pulling out one value of the message, but you could also unmarshal the whole JSON message into a Java object...and also create an open source library for us. Let me know when it is ready to use!

That's it. Applied to the `somecats.json` file we have used before, if we want to retrieve the value that was recognized by Rhasspy, we will do something along the lines of Listing 5-5.

Listing 5-5. Parsing JSON Content

```
package practice;

import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.List;
import java.util.stream.Collectors;
```

```
import org.json.JSONObject;

public class JSONFun {

    public static String whatObject(String json) {
        JSONObject obj = new JSONObject(json);
        JSONObject slot = ((JSONObject) obj.
        getJSONArray("slots").get(0)).getJSONObject("value");
        String cats = slot.getString("value");
        return cats;
    }

    public static void main(String... args) throws Exception {
        List<String> nekos = Files.readAllLines(Paths.
        get("onlycats.json"));
        String json = nekos.stream().collect(Collectors.
        joining("\n"));
        System.out.println(whatObject(json));
    }
}
```

Listening to MQTT Basic Messages

Listening to messages, or *subscribing*, is done via the following steps:

1. Subscribe to a topic or parent topic.
2. Adda listener that implements the `IMqttMessageListener` interface.

The simplest code for subscribing is shown in Listing 5-6.

Listing 5-6. From Russia with Love

```
package practice;

import org.eclipse.paho.client.mqttv3.IMqttMessageListener;
import org.eclipse.paho.client.mqttv3.MqttClient;
import org.eclipse.paho.client.mqttv3.MqttMessage;

public class Sub0 {

    public static void main(String... args) throws Exception {

        MqttClient client = new MqttClient("tcp://
        localhost:1883", MqttClient.generateClientId());
        client.connect();
        client.subscribe("hello", new IMqttMessageListener() {
            @Override
            public void messageArrived(String topic,
            MqttMessage message) throws Exception {
                String hello = new String(message.getPayload(),
                "UTF-8");
                System.out.println(hello);
            }
        });
    }
}
```

Note that the message-handling part can of course be replaced with Java lambdas, thus making it even more readable, as in Listing 5-7.

Listing 5-7. Messages with Java Lambdas

```
MqttClient client = new MqttClient("tcp://localhost:1883",
MqttClient.generateClientId());
client.connect();
client.subscribe("hello", (topic, message) -> {
```

```
 String hello = new String(message.getPayload(), "UTF-8");
 System.out.println(hello);
});
```

If you're running everything from your Raspberry Pi, then the following Visual Studio Code setup will do everything properly and display the Russian message, as shown in Figure 5-6.

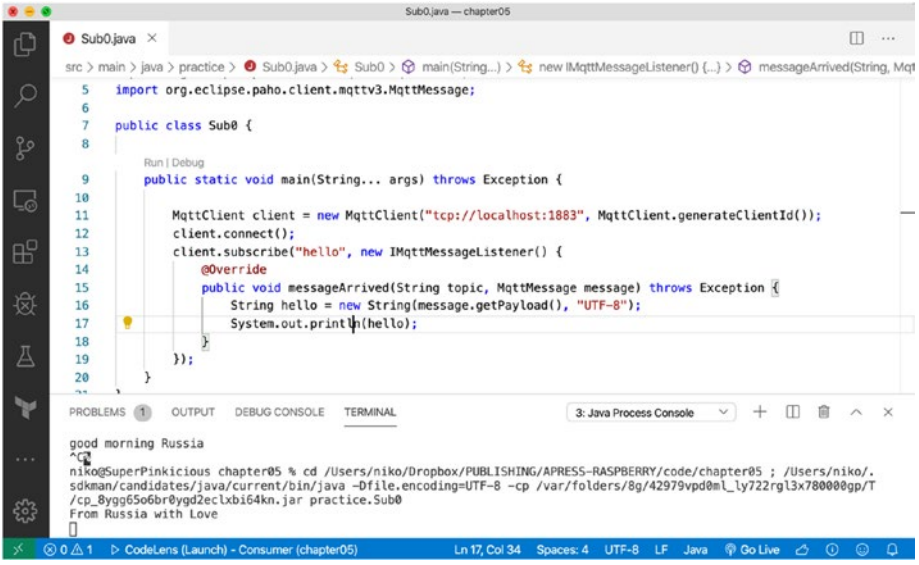

Figure 5-6. *From Russia with love*

SENDING A MESSAGE FROM JAVA TO THE RASPBERY PI

Obviously, the fun is in having a distributed system and handling messages coming from different places. Here, try to run the broker on the Raspberry Pi. After setting up the IP address of your Raspberry Pi in the sender listing, send a message from your computer to the Raspberry Pi in Java.

Or send a message in Java from the Raspberry Pi to a broker located remotely or, again, in the cloud.

Listening to MQTT JSON Messages

So, now it is the time to listen for and parse the content of the JSON message from within Java, which is mostly a combination of listening to basic messages and implementing the JSON fun you had in the previous pages. The code snippet is in Listing 5-8.

Listing 5-8. Cats Coming

```
package practice;

import org.eclipse.paho.client.mqttv3.IMqttMessageListener;
import org.eclipse.paho.client.mqttv3.MqttClient;
import org.eclipse.paho.client.mqttv3.MqttMessage;
import org.json.JSONObject;

public class Sub {

    public static void main(String... args) throws Exception {

        MqttClient client = new MqttClient("tcp://
        localhost:1883", MqttClient.generateClientId());
        client.connect();
        client.subscribe("hermes/intent/#", new
        IMqttMessageListener() {
            @Override
            public void messageArrived(String topic,
            MqttMessage message) throws Exception {
                String json = new String(message.getPayload(),
                "UTF-8");
                JSONObject obj = new JSONObject(json);
                JSONObject slot = ((JSONObject) obj.
                getJSONArray("slots").get(0)).
                getJSONObject("value");
```

```
                String cats = slot.getString("value");
                System.out.println(cats);
            }
        });
    }
}
```

Now that you know everything about keys and brokers, it is time to get and install Rhasspy.

Voice and Rhasspy Setup

The Rhasspy voice platform needs a set of services running to be able to deploy applications with intents. In this section, you will see how to install the platform first and then how to create applications with intents.

The easiest way to install Rhasspy is actually via Docker (https://www.docker.com/), which is a container engine on which you can run images. The Rhasspy maintainer has created a ready-to-use Docker image for Rhasspy, so we will take advantage of that.

Preparing the Speaker

You do need a working microphone setup before executing any of the voice commands.

We recommend ReSpeaker. If you have it, you can find the installation instructions here:

http://wiki.seeedstudio.com/ReSpeaker_4_Mic_Array_for_Raspberry_Pi/

Provided you have plugged in the speaker on the Pi, you can perform the necessary module installation with the following short shell script:

```
sudo apt-get update
sudo apt-get upgrade
git clone https://github.com/respeaker/seeed-voicecard.git
cd seeed-voicecard
sudo ./install.sh
reboot
```

Standard USB microphones should be plug-and-play, and I have been using conference room microphones with pretty good results.

So, if things are working, the microphone should show up in the list output from this `arecord` command:

```
arecord -L
```

You ReSpeaker/Raspberry Pi setup should look like the one in Figure 5-7.

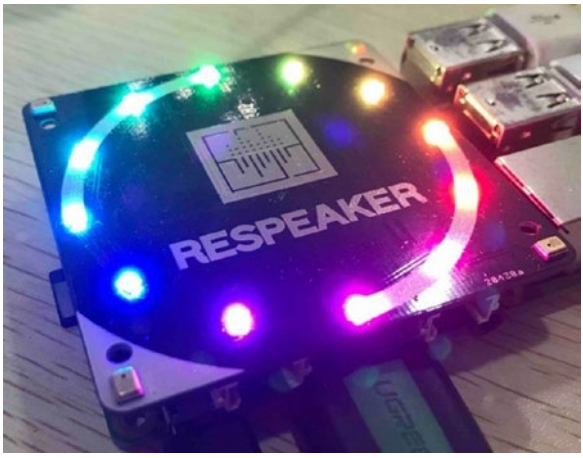

Figure 5-7. *ReSpeaker lights*

Installing Docker

The Docker web site has a recent entry on how to set up Docker for the Raspberry Pi; you can find it here:

```
https://www.docker.com/blog/happy-pi-day-docker-raspberry-pi/
```

All the steps are repeated in the following code:

```
#   a. Install the following prerequisites.
sudo apt-get install apt-transport-https ca-certificates
software-properties-common -y
#   b. Download and install Docker.
curl -fsSL get.docker.com -o get-docker.sh && sh get-docker.sh
#   c. Give the 'pi' user the ability to run Docker.
sudo usermod -aG docker pi
#   d. Import Docker CPG key.
sudo curl https://download.docker.com/linux/raspbian/gpg
#   e. Setup the Docker Repo.
sudo echo "deb https://download.docker.com/linux/raspbian/
stretch stable" >> /etc/apt/sources.list
#   f. Patch and update your Pi.
sudo apt-get update
sudo apt-get upgrade
#   g. Start the Docker service.
systemctl start docker.service
#   h. To verify that Docker is installed and running.
docker info
#    i. You should now some information in regards to
versioning, runtime,etc.
```

Installing Rhasspy with Docker

The whole Rhasspy suite can be started by running the prepared container, and this is done by executing the following command on the Raspberry Pi, provided you have installed Docker properly.

You use `docker run` to start the image itself; you can picture the image as being a small OS containing prepackaged software and configuration and thus being easy to deploy.

```
docker run -d -p 12101:12101 \
      --restart unless-stopped \
      -v "$HOME/.config/rhasspy/profiles:/profiles" \
      --device /dev/snd:/dev/snd \
      synesthesiam/rhasspy-server:latest \
      --user-profiles /profiles \
      --profile en
```

Here's a breakdown of the code:

- `docker`: This is the command-line executable that communicates with the Docker daemon and sends commands to start and stop the container.
- `-d`: This tells Docker to run the image as a daemon, not waiting for the execution of the image to finish and starting the container as a background process.
- `-p 12101:12101`: This does port mapping between Docker and the host; without this, open ports within the container cannot be viewed from the outside world. Here we tell Docker to map port 12101 of the container to port 12101 of the Raspberry Pi.

- `--restart unless-stopped`: This speaks for itself, meaning even if the container is dying because of an internal exception or error, the Docker daemon will automatically restart the image.
- `-v "$HOME/.config/rhasspy/profiles:/profiles"`: Here we want to "mount" a folder of the Raspberry Pi and make it available to the Docker container, and vice versa. This is kind of a shared-network folder. It will be used to store files downloaded by Rhasspy for us.
- `--device /dev/snd:/dev/snd`: This is pretty neat; it maps the raw data coming to the sound system of the Raspberry Pi and makes it available to the Docker container.
- `synesthesiam/rhasspy-server:latest`: This is the name of the image used to start the container. Think of this image as a verbatim copy of the file system used to start the container.
- The next two are parameters that the Rhasspy image understands:
  - `--user-profiles /profiles`: This is where to store the profiles, and this is the path of the folder we are sharing with the Raspberry Pi.
  - `--profile en`: This is the language to use.

To check that the container is started properly, let's verify its status with the `docker` command, as shown here:

```
docker ps --format "table {{.Status}}\t{{.Ports}}\t{{.Image}}
\t{{.ID}}"
```

This should show something similar to the following; make sure to especially check the status field, which we have put up front:

```
STATUS              PORTS
IMAGE                                        CONTAINER ID
Up 22 minutes       0.0.0.0:12101->12101/tcp
synesthesiam/rhasspy-server:latest    af7c56961d5c
```

The Rhasspy server and its console are ready, so let's go back to the main computer and access the Raspberry Pi via the browser-based IDE.

Starting the Rhasspy Console

To access the console, go to `http://192.168.1.104:12101/`, where 192.168.1.104 is the IP address of the Raspberry Pi.

You're then presented with the lively screen shown in Figure 5-8.

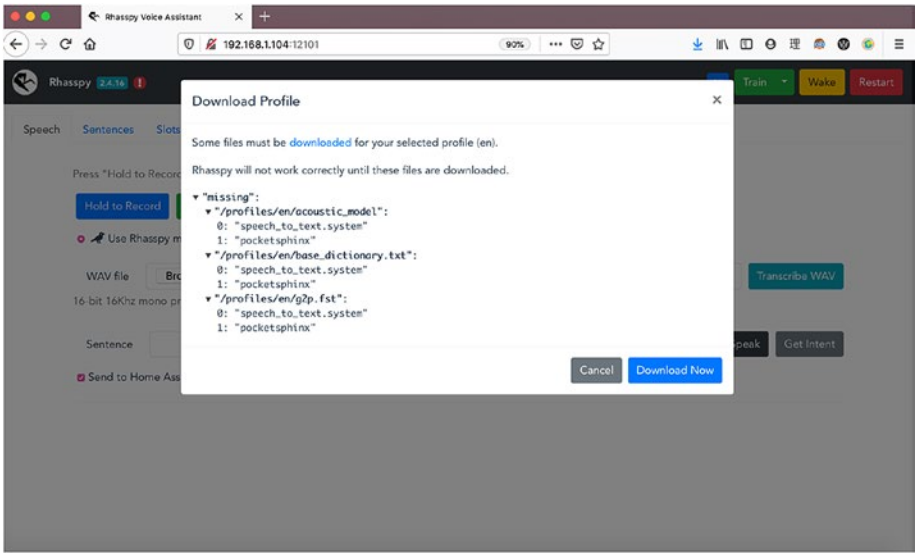

Figure 5-8. *The Rhasspy console*

The console makes it quite obvious for you on first start that it should go and fetch some remote files, so provided your Raspberry Pi is connected to the Internet, just click the Download Now button.

The operation takes a few minutes to complete, so don't lose your patience here. Wait for the dialog box shown in Figure 5-9.

Figure 5-9. *Download complete*

The console refreshes by itself, and we're now being greeted with Figure 5-10.

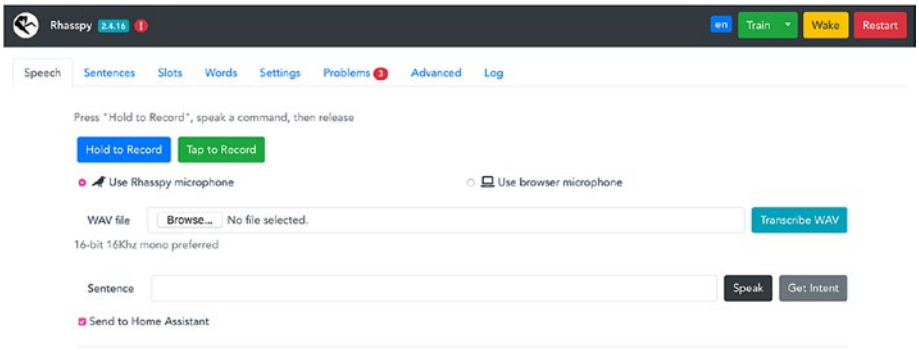

Figure 5-10. *Refreshed Rhasspy console*

But we seem to still have some markers telling us something is missing, as shown in Figure 5-11 and Figure 5-12.

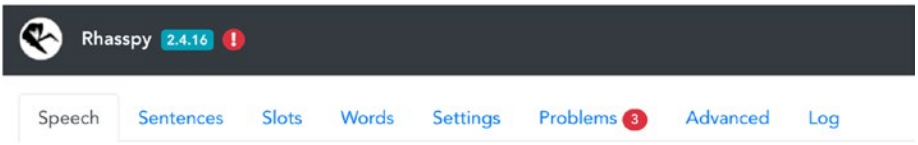

Figure 5-11. *Red markers*

Component	Problem	Description
FsticuffsRecognizer	Missing intent FST	Intent finite state transducer (FST) not found at profiles/en/intent.fst. Did you train your profile?
PocketsphinxDecoder	Missing dictionary	Dictionary not found at profiles/en/dictionary.txt. Did you train your profile?
PocketsphinxDecoder	Missing language model	Language model not found at profiles/en/language_model.txt. Did you train your profile?

Figure 5-12. *The setup problems are nicely explained*

Fortunately, and for now, clicking the top-right green Train button makes our console shiny and free of red markers, as in Figure 5-13. We are now ready to use the Rhasspy console.

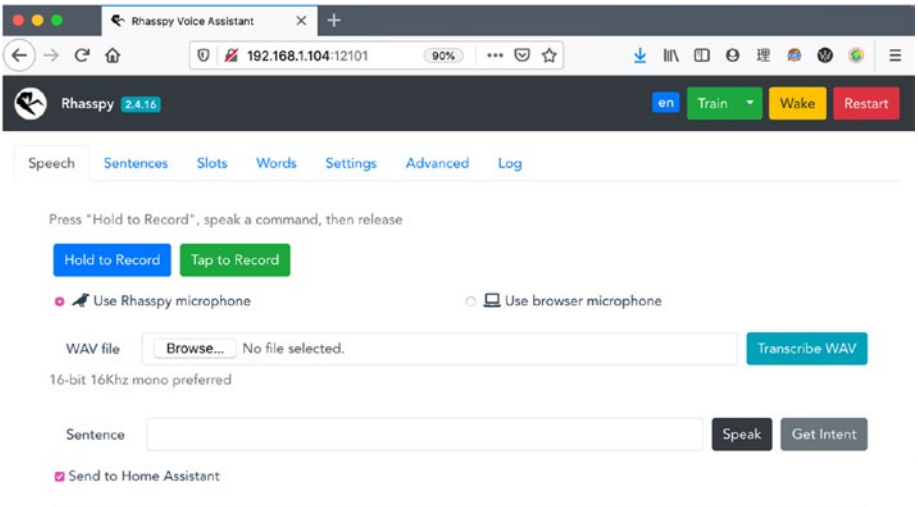

Figure 5-13. *The console is ready!*

The Rhasspy Console

Let's take a look at the Rhasspy menu in more detail. Figure 5-14 shows the different sections for configuring the home assistant.

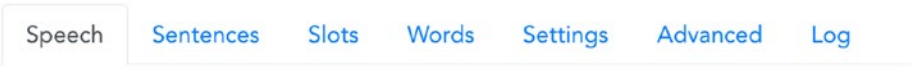

Figure 5-14. *More configuration options*

The usage of each tab is explained in the following list:

- **Speech:** This is where you can try your assistant.
- **Sentences:** This is where you define what your vocal assistant can detect and recognize.
- **Slots:** Sentences have variables in them. Each possible value for a variable can be defined in a line in the Sentences section, or you can create lists of possible values in the Slots section.
- **Words:** This is the section where you define how to pronounce words. This is especially useful for names, as the English engine is sometimes not so well prepared for French people's pronunciations!
- **Settings:** This is an all-encompassing screen where you can configure the way engines work and what to do with recognized intents.
- **Advanced:** This is the same as Settings except the configuration is exposed as a text file that can be edited directly in the browser.
- **Log:** This is where engine logs can be viewed in real time.

A few notes…

The browser IDE (to be correct, the processes inside the Docker container) will ask you to restart Rhasspy each time you change a configuration setting.

It will also ask you to train Rhasspy each time you make changes to either the Sentences or Words section.

There is no specific reason to not do this, unless you're in the middle of surgery and need to ask your assistant for immediate help.

We know how to navigate the console now, so let's make our vocal assistant recognize something for us.

First Voice Command

To create a voice command that generates an intent, we need to make some edits in the Sentences section. As just explained, the Sentences section is where we define the sentences our engine can recognize.

First Command, Full Sentence

For once, the grammar used to define intents is quite simple and is inserted via an `.ini` file (which comes from the good old Windows 95 days).

The pattern is as follows:

```
[IntentName]
Sentence1
```

So, if it is Monday morning and you are really in need of coffee, you could write something like this:

```
[Coffee]
I need coffee
```

Let's try it. Let's start by deleting the content of that `.init` file and adding just the previous coffee intent, so the Sentences section now looks like Figure 5-15.

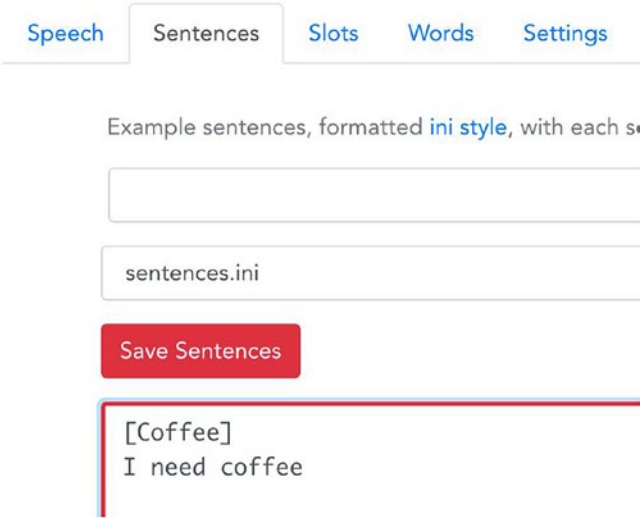

Figure 5-15. *I do really need coffee*

If you click the nicely visible Save Sentences button, you'll get the training reminder shown in Figure 5-16.

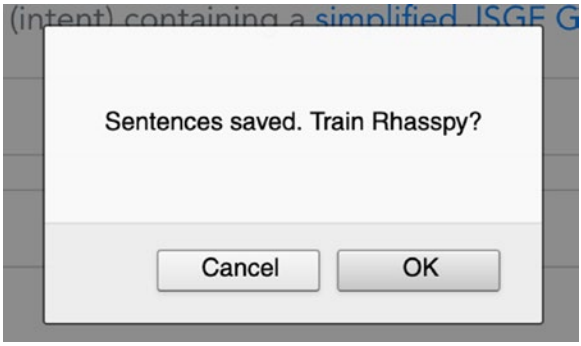

Figure 5-16. *First Rhasspy training*

Speech Section and Trying Your Intent

The Rhasspy engine is now trained, and you can try it by heading to the Speech section, shown again in Figure 5-17.

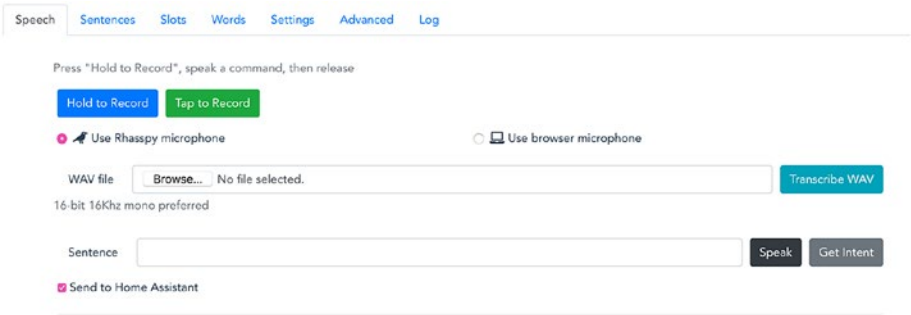

Figure 5-17. *Speech section*

The Speech section is where you can either speak or write a sentence and see whether and what intent is recognized.

Supposed we type “I need coffee” in the Sentence field; the engine should recognize and generate the “Coffee” response that we just defined, as in Figure 5-18.

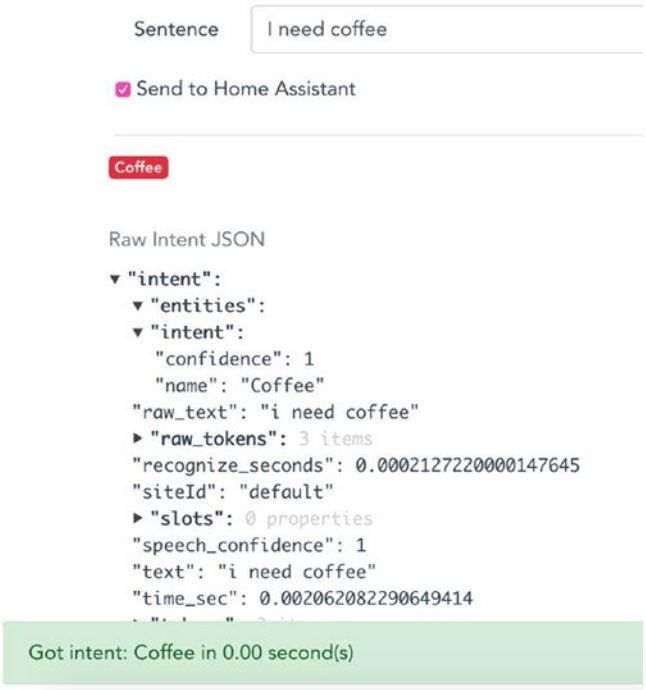

Figure 5-18. *I…need…coffee*

The voice command "Coffee" is recognized, and an associated generated JSON document is attached to it, with various inputs on confidence and raw values.

Since we're focusing on voice commands here, you can also just tap and hold the "Hold to record" button, say "I need coffee," and see the spoken sentence being turned into text first, then displayed in the Sentence field, and then analyzed as was just done.

This is probably the time to plug in the ReSpeaker or your USB microphone.

If you have installed eSpeak (http://espeak.sourceforge.net/), which is a text-to-speech command-line tool, you can also generate a WAV file and feed it to Rhasspy.

```
espeak "I need coffee" --stdout >> ineedcoffee.wav
```

Upload that `ineedcoffee.wav` file and click the Get Intent button.

Fine-Tuned Intents

It is of course possible to just pop in optional words, variables, and placeholders in the commands defined in the Sentences section. Let's improve our game here and add details to our intent so as to create better voice commands.

Optional Words

It is of course possible to make words optional in the sentence to make the commands more natural to users.

In the previous coffee example, you could have an optional urgency embedded into the intent, as in you may *really* need coffee at times. This is done by putting the optional words inside square brackets, as shown here:

```
[Coffee]
I [really] need coffee
```

After training, you see in Figure 5-19 and Figure 5-20 how the same Coffee intent is being properly recognized by Rhasspy.

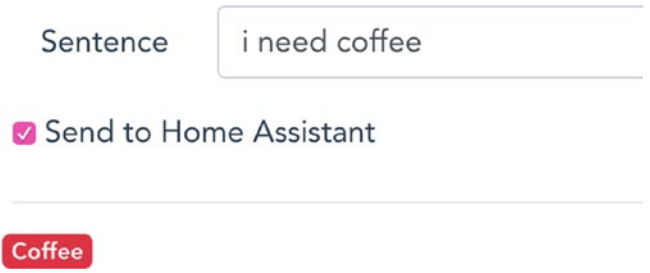

Figure 5-19. *Coffee intent without the optional word*

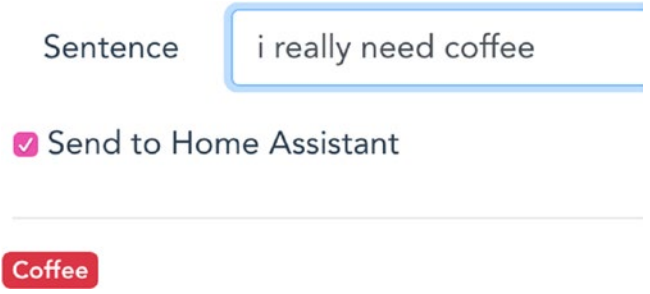

Figure 5-20. *Coffee intent with the optional word, which is really*

Adding Alternatives

But what about when it's a day where you feel really good about yourself and you don't really need that addicting black stuff to wake you up? Let's rewrite our coffee intent so it can recognize both when you need coffee and when you don't.

```
[Coffee]
I (need | don't need) coffee
```

Training and using this new intent, you can now tell your coffee assistant that you do not need coffee. Back on the Speech tab, you would get the intent shown in Figure 5-21.

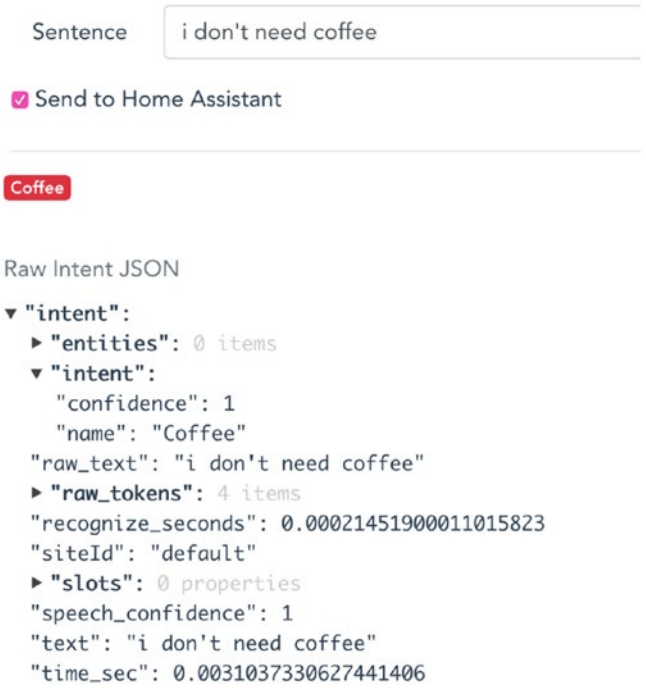

Figure 5-21. *I don't need coffee (although, to be honest, that's not true)*

You probably already noticed one problem with this generated intent. Yes, it's not very useful as is, because the same intent is generated whether we want coffee or not.

We could of course parse the received string or write two different intents, but what if we simply give a name to that list of possibilities? This is done with round brackets, as shown here:

```
[Coffee]
I ((need | don't need) {need}) coffee
```

Now, our Coffee intent will add a slot named "need" to the JSON document, and its value will be either "need" or "don't need."

Let's try again; see the result in Figure 5-22.

Figure 5-22. *Maybe I don't need coffee*

You should notice several things here. First, the summary of the intent is now showing more details, the intent is marked in red, and each slot is nicely shown in a list with its associated value (here, need = don't need).

Second, the full JSON document of the intent contains an `entities` section, and each entity has an entity name and associated value.

As a side note, the JSON document also contains a `slots` section, with a simple list of all the slots of the intent.

What's next?

Making Intents with Slots More Readable

You probably noticed that if we add one too many slots, then the whole sentence becomes messy quite quickly.

We can move the slot definition right below the intent name and assign possible values with =.

For example, the same coffee intent can be rewritten as shown here:

```
[Coffee]
need = ((need | don't need) {need})
I <need> coffee
```

Nothing has changed, and the intent behaves the same as before, but it is now vastly more readable.

That's great, but what's next?

Defining Reusable Slots

Yes, defining slots inline also brings extra complexity to our sentence definitions, so it is possible to move them out and define them on the Slots tab.

Slots are defined in a simple JSON file, and to move our coffee need to the Slots tab, you would write the JSON file shown here:

```
{
    "need": [
        "need",
        "don't need"
    ]
}
```

This shows up in the browser as in Figure 5-23.

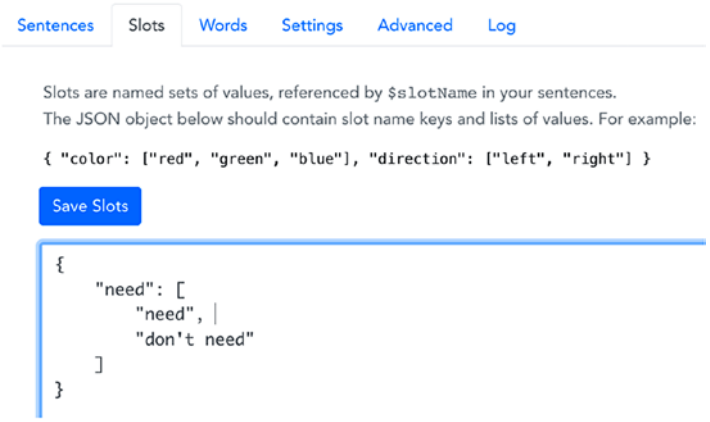

Figure 5-23. *Slots of coffee needs*

To use that in the sentence file, you would now have to write the intent, as shown here:

```
[Coffee]
myneed = $need {need}
I <myneed> coffee
```

Here is a breakdown of the code:

- `myneed`: This is the placeholder in the sentence, meaning its definition can be seen as being inlined at training time.
- `$need`: The dollar sign specifies that this is coming from the slots JSON file.
- `{need}`: This is the name of the slot within that intent, meaning you can reuse the same list in different intents, each time with a different slot name.

Great. Now we have highly configurable intents, with a complex list of options on hand.

Let's move to the section where the external system can make use of the generated JSON file when an intent is recognized.

Settings: Get That Intent in the Queue

The generated JSON file looks like it could be useful, but we want to have it external to Rhasspy and in the MQTT queue so that we can do more processing.

Let's head to the Settings section of the console, as shown in Figure 5-24.

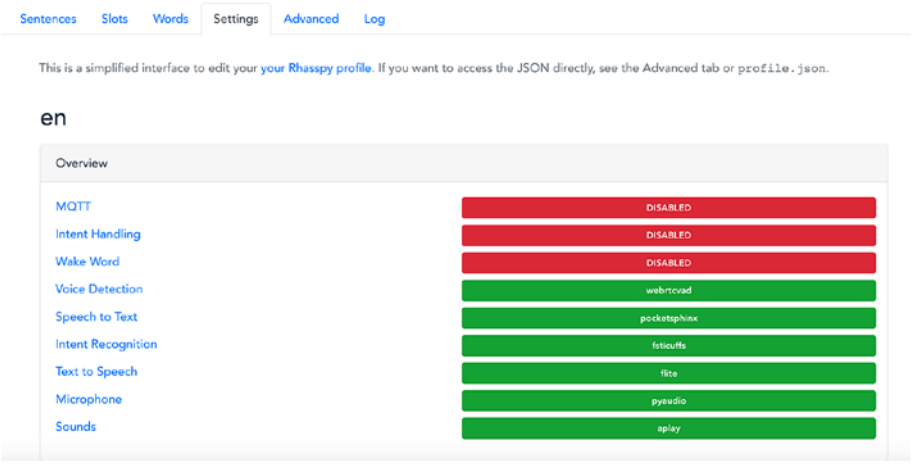

Figure 5-24. *Rhasspy Settings section*

Here we get an overview of how the system is configured. There are zillions of configurable options for Rhasspy, but we will focus on the following:

- **MQTT**: This sends the JSON to the MQTT queue.
- **Wake word**: The background process is always listening for possible "wake-up" keywords; this is your "Hey, Siri" or "Hey, Google" spy here.

Let's look at the MQTT interaction first. If you click the MQTT link, you'll be taken to the corresponding section, where you can enter the details of your running MQTT daemon, as shown in Figure 5-25.

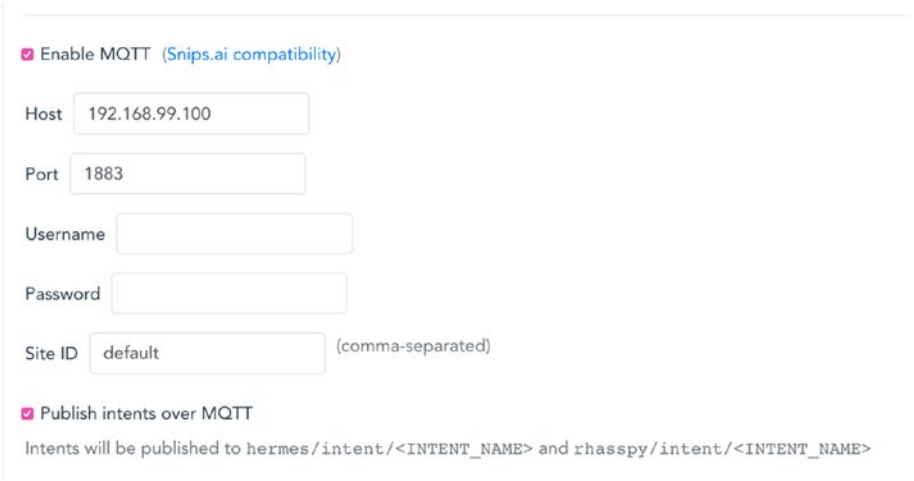

Figure 5-25. *MQTT settings*

Make sure to fill in the hostname or the IP address of your Raspberry Pi here. Restart Rhasspy when asked, and that's it. Now let's try to get some coffee on MQTT.

Back on the Speech tab, we can now trigger a new coffee intent, as in Figure 5-26.

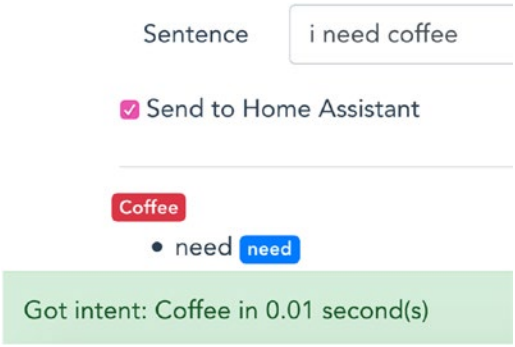

Figure 5-26. *One more coffee?*

Let's now take a look at the MQTT Explorer window of Figure 5-27.

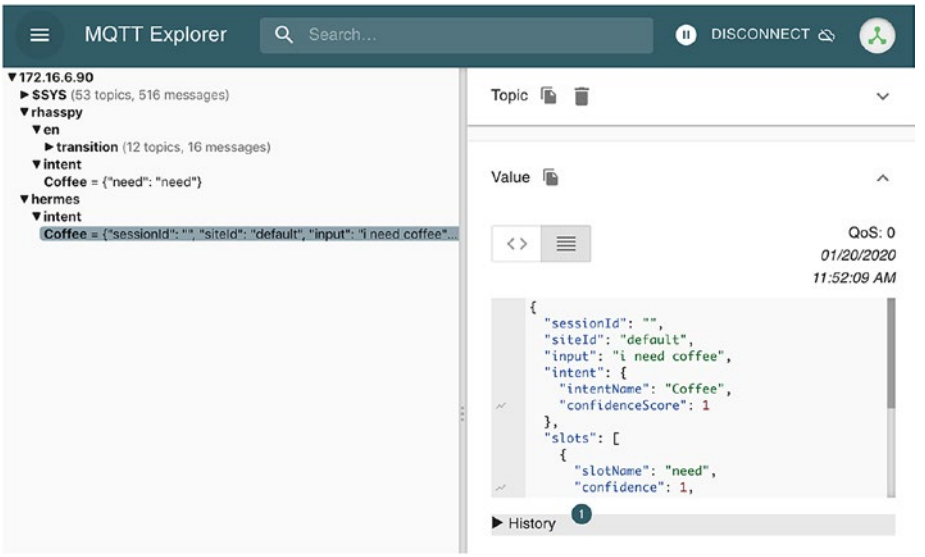

Figure 5-27. *Rhasspy and MQTT*

What do you see? We see a message being sent to the `hermes/intent/coffee` topic, which is something similar to what you saw in the previous sections of this chapter.

Yes, you can feel it too—this is going to work.

You can also see that there is a simple MQTT message sent to `Rhasspy/intent/coffee`, with just the slot's name and value in the message body. If you don't need to play with the confidence and other details, this is the best alternative. Working with this message is left as an exercise for you.

Hint This is easy.

You can get some instant gratification by trying a few messages specifying whether you want or don't want coffee; just make sure to not drink too much of it.

Settings: Wake-Up Word

It's nice to go back to speech every now and then, but it would be nice to just leave the Raspberry Pi by itself, and when saying a given word, or set of words, get Rhasspy to listen for intents.

This is done by using *wake-up words*, which are defined in the Settings section.

I have been having some luck with enabling the Snowboy kit (`https://snowboy.kitt.ai/`) in the configuration, as shown in Figure 5-28.

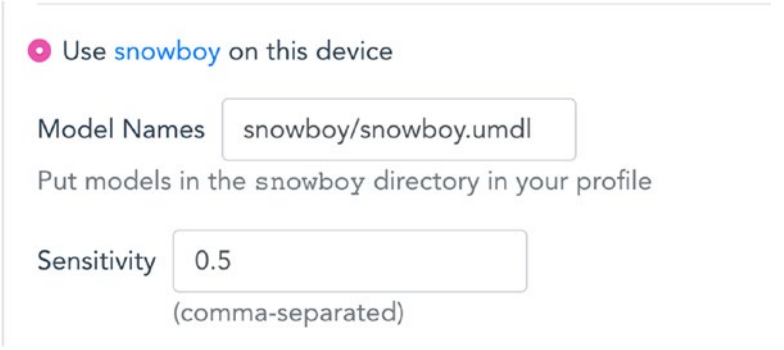

Figure 5-28. *Snowboy configuration*

There is a default UMDL file downloaded locally for you when you enable Snowboy; it will be at the following location on the Raspberry Pi:

```
/home/pi/.config/rhasspy/profiles/en/snowboy/snowboy.umdl
```

You can create your own files and keywords easily by recording your keyword and uploading the audio files to the Snowboy web site, as detailed here:

```
http://docs.kitt.ai/snowboy/
```

Now, when you say "Snowboy," Rhasspy will wake up, as if you were holding the Record button.

So, now, we just have to say the following:

- "Snowboy"
- "I need coffee"

We're getting there, aren't we?

Creating the Highlight Intent

Now, we have to create a small intent that we will use to tell our main object detection application which object we want to focus on.

The intent will be specified according to the following rules:

- **Name:** `highlight`
- **Command:** `show me only <something>`, where `<something>` is either `cat` or `person`
- **Name of the slot:** `only`

You should definitely try this on your own before reading ahead…

The resulting code for the intent to put in the sentence file is as follows:

```
[highlight]
object = (cats | person) {only}
show me only <object>
```

The best way to try is of course by saying the following:

- "Snowboy"
- "Show me only cats"

to have the intent JSON be published in the `hermes/intent/highlight` queue, with the content as shown in Figure 5-29.

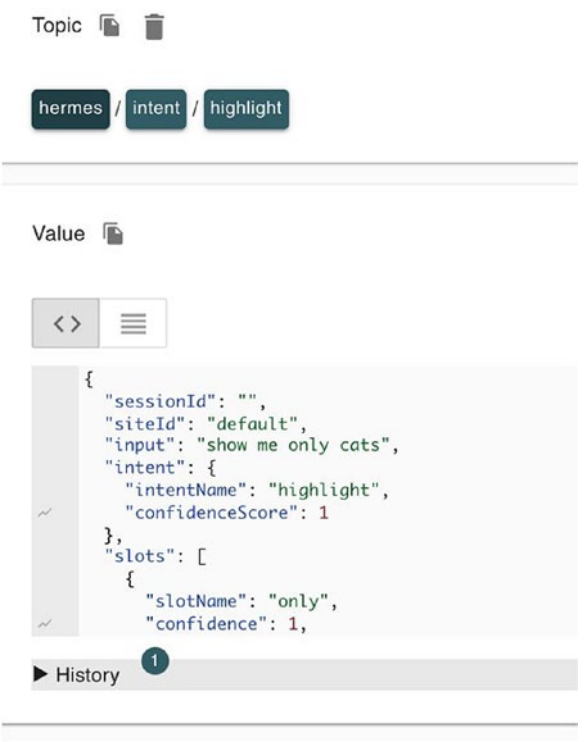

Figure 5-29. *Show me only cats, Snowboy*

Voilà, we're mostly done with the Rhasspy and voice command section, so we can now go back to detecting objects, cats, and people in real time.

Voice and Real-Time Object Detection

This is the part of the chapter where we bridge it all, meaning the video analyses from previous chapters and the voice recognition using Rhasspy.

First, let's get ready with a simple `origami` project setup, with messages being sent by Rhasspy.

Simple Setup: Origami + Voice

In this section, we will do the following:

1. Start a video streaming on the main camera in full-screen, in a new thread.
2. The video streaming is using the `origami` core filter `annotate`, where you can just add some text to the picture.
3. Connect to MQTT on the main thread.
4. On a new intent message, we update the `annotate` text.

The intent used here is the `highlight` intent that was defined earlier, with a slot to recognize the names taken from the object categories of the COCO dataset.

The rest of Listing 5-9 should be easy to follow.

Listing 5-9. Origami + Rhasspy

```
package practice;

import org.eclipse.paho.client.mqttv3.IMqttMessageListener;
import org.eclipse.paho.client.mqttv3.MqttClient;
import org.eclipse.paho.client.mqttv3.MqttMessage;
import org.json.JSONObject;
import origami.Camera;
import origami.Origami;
import origami.filters.Annotate;

public class Consumer {

    public static void main(String... args) throws Exception {
        Origami.init();
        Annotate myText = new Annotate();
```

```
        new Thread(() -> {
            new Camera().device(0).filter(myText).fullscreen().
            run();
        }).start();

        MqttClient client = new MqttClient("tcp://
        localhost:1883", MqttClient.generateClientId());
        client.connect();
        client.subscribe("hermes/intent/#", new
        IMqttMessageListener() {
            @Override
            public void messageArrived(String topic,
            MqttMessage message) throws Exception {
                String json = new String(message.getPayload(),
                "UTF-8");
                JSONObject obj = new JSONObject(json);
                JSONObject slot = ((JSONObject) obj.
                getJSONArray("slots").get(0)).
                getJSONObject("value");
                String cats = slot.getString("value");
                myText.setText(cats);
            }
        });
    }
}
```

If things are all good and you say “Show me only cats,” the annotate filter will update the upper-left text, as shown in Figure 5-30.

Figure 5-30. *Webcam streaming*

Now let's connect this setup with object recognition and with the real-time video streaming part.

Origami Real-Time Video Analysis Setup

From the previous chapter, you will remember that we were using an `origami` filter, called Yolo, to perform object detection. Just like the `annotate` filter, the Yolo filter is included in the `origami-filters` library, and in a very Java way, you just have to extend the base filter to highlight the detected objects.

Available in the `origami-filter` library, the Yolo filter allows you to download and retrieve networks on demand using a network specification.

So, for example, `networks.yolo:yolov3-tiny:1.0.0` will download `yolov3-tiny` and `cache` for further usage, depending on which device you're on. This is a feature of the `origami` framework.

For Yolo, the following network specs are available for training Yolo networks on the Coco data set:

- `networks.yolo:yolov3-tiny:1.0.0`
- `networks.yolo:yolov3:1.0.0`
- `networks.yolo:yolov2:1.0.0`
- `networks.yolo:yolov2-tiny:1.0.0`

Each one is either faster or more precise than the one before it.

Creating the Yolo Filter

In Listing 5-10, we prepare the groundwork for showing either the number of all the objects detected, using `annotateWithTotal`, or the display only, via the `only` switch. Note that the entry point is in the function `annotateAll`, where it is decided whether to display all the boxes or a subset of them.

Listing 5-10 shows the code for the MyYolo filter.

Listing 5-10. MyYolo Filter

```
package filters;

import org.opencv.core.Mat;
import org.opencv.core.Point;
import org.opencv.core.Rect;
import org.opencv.core.Scalar;
import org.opencv.imgproc.Imgproc;
import origami.filters.detect.Yolo;

import java.util.List;

public class MyYolo extends Yolo {
    Scalar color = new Scalar(110.0D, 220.0D, 0.0D);
    String only = null;
    boolean annotateWithTotal = false;

    public MyYolo(String spec) {
        super(spec);
    }

    public MyYolo annotateWithTotal() {
        this.annotateWithTotal = true;
        return this;
    }

    public MyYolo annotateAll() {
        this.annotateWithTotal = false;
        return this;
    }

    public MyYolo only(String only) {
        this.only = only;
        return this;
    }
```

```
    public MyYolo color(Scalar color) {
        this.color = color;
        return this;
    }

    @Override
    public void annotateAll(Mat frame, List<List> results) {
        if (only == null) {
            if (annotateWithTotal)
                annotateWithCount(frame, results.size());
            else
                super.annotateAll(frame, results);
        } else {
            if (!annotateWithTotal)
                results.stream().filter(result -> result.
                get(1).equals(only)).forEach(r -> {
                    annotateOne(frame, (Rect) r.get(0),
                    (String) r.get(1));
                });
            else
                annotateWithCount(frame, (int) results.
                stream().filter(result -> result.get(1).
                equals(only)).count());
        }
    }

    public void annotateWithCount(Mat frame, int count) {
        Imgproc.putText(frame, (only == null ? "ALL" : only) +
        " (" + count + ")", new Point(50, 500), 1, 4.0D,
        color, 3);
    }
```

```
    public void annotateOne(Mat frame, Rect box, String label) {
        if (only == null || only.equals(label)) {
            Imgproc.putText(frame, label, new Point(box.x,
            box.y), 1, 3.0D, color, 3);
            Imgproc.rectangle(frame, box, color, 2);
        }
    }
}
```

With the filter now ready, we can start doing analysis again.

Running the Video Analysis Alone

In this short section, we'll play a video by starting the camera on one thread and in another thread updating the selection of shown objects of the Yolo filter in real time. The second main thread directly calls the only function of the MyYolo filter, which we just defined in the previous section. Listing 5-11 shows the full code for this.

Listing 5-11. Running Yolo on a Video, Updating Parameters in Real Time

```
package practice;

import origami.Camera;
import origami.Filter;
import origami.Filters;
import origami.Origami;
import origami.filters.FPS;
import filters.MyYolo;

public class YoloAgain {
```

```
    public static void main(String[] args) {
        Origami.init();
        String video = "mei/597842788.852328.mp4";
//        Filter filter = new Filters(new MyYolo("networks.
          yolo:yolov3-tiny:1.0.0").only("car"), new FPS());
        MyYolo yolo = new MyYolo("networks.yolo:yolov3-
        tiny:1.0.0"); //.only("person");
        yolo.thresholds(0.4f, 1.0f);
        yolo.annotateWithTotal();

        Filter filter = new Filters(yolo, new FPS());
        new Thread(() -> {
            new Camera().device(video).filter(filter).run();
        }).start();

        new Thread(() -> {
            try {
                Thread.sleep(5000);
                System.out.println("only cat");
                yolo.only("cat");
                Thread.sleep(5000);
                System.out.println("only person");
                yolo.only("person");
            } catch (InterruptedException e) {
                //e.printStackTrace();
            }
        }).start();

    }
}
```

You can probably tell where we are heading now, linking the message received from Rhasspy to update the selection of the Yolo analysis.

Integrating with Voice

In this final section, we will highlight a subset of objects based on the input received from the Rhasspy intent.

To get this working, we have to do the following:

1. Start a video streaming on the main camera in full screen, in a new thread.
2. The video streaming is using a MyYolo filter loading the Yolo network.
3. Connect to MQTT on the main thread.
4. On a new message, we update the annotate text.
5. When a message arrives in the highlight queue, we do the following:
   - Parse the JSON object.
   - Retrieve the only value.
   - Update the selection of the MyYolo filter.

The final code snippet of this book shows how to put all of this together; see Listing 5-12.

Listing 5-12. Real-Time Detection, Integrating with the Rhasspy Queue

```
import filters.MyYolo;
import org.eclipse.paho.client.mqttv3.MqttClient;
import org.json.JSONObject;
import origami.Camera;
import origami.Origami;

import static java.nio.charset.StandardCharsets.*;
```

```
public class Chapter05 {

    public static String whatObject(String json) {
        JSONObject obj = new JSONObject(json);
        JSONObject slot = ((JSONObject) obj.
        getJSONArray("slots").get(0)).getJSONObject("value");
        String cats = slot.getString("value");
        return cats;
    }

    public static void main(String... args) throws Exception {
        Origami.init();
//        String video = "mei/597842788.852328.mp4";
        String video = "mei/597842788.989592.mp4";
        MyYolo yolo = new MyYolo("networks.yolo:yolov3-
        tiny:1.0.0");
        yolo.thresholds(0.2f, 1.0f);
//        yolo.only("cat");

        new Thread(() -> {
//            new Camera().device(0).filter(yolo).run();
            new Camera().device(video).filter(yolo).run();
        }).start();

        MqttClient client = new MqttClient("tcp://
        localhost:1883", MqttClient.generateClientId());
        client.connect();
        client.subscribe("hermes/intent/hellonico:highlight",
        (topic, message) -> {
```

```
            yolo.only(whatObject(new String(message.
            getPayload(), "UTF-8")));
        });

    }
}
```

In the different sample videos or directly in the video stream of the webcam of the Raspberry Pi, you can directly update the objects to be detected.

We could have look for cats in this final example, but in a timely manner my daughter Mei just sent me videos of her at school and out with friends at night. See her highlighted by our YOLO setup while she's in action in Figure 5-31 and Figure 5-32.

Figure 5-31. *Mei at school*

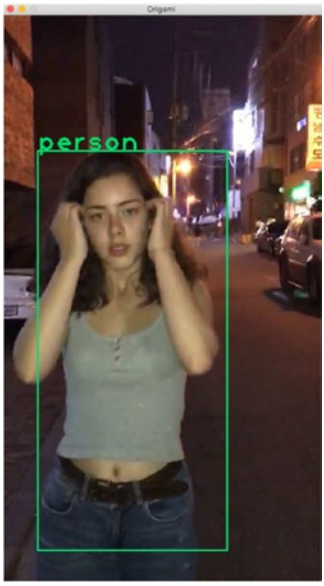

Figure 5-32. *Mei at night*

In this chapter, you learned about the following:

- The Rhasspy message flow
- How to interact from Java with MQTT for IoT messaging
- How to set up Rhasspy and intents
- How to connect messages sent by Rhasspy to our Java code
- How to update the real-time video analysis running on the Raspberry Pi by interpreting voice sent by Rhasspy with custom intents

This short book has come to an end, so now it's your turn to change the world. We've only scratched the surface of what is possible, but I really hope this gives you some ideas to try to implement.

Index

N. Modrzyk, *Real-Time IoT Imaging with Deep Neural Networks*,
https://doi.org/10.1007/978-1-4842-5722-7

F

G

H

I

J, K

L

M

N

O

P, Q

R

Y, Z