Chapter No. 1
"Introduction to the World of Leap Motion"
In this package, you will find:

The author’s biography
A preview chapter from the book, Chapter no.1 "Introduction to the World of Leap Motion"
A synopsis of the book’s content
Information on where to buy this book

About the Author

**Brandon Sanders** is an 18-year-old roboticist who spends much of his time designing, building, and programming new and innovative systems, including simulators, autonomous coffee makers, and robots for competition. At present, he attends Gilbert Finn Polytechnic (which is a homeschool) as he prepares for college. He is the founder and owner of Mechakana Systems, a website and company devoted to robotic systems and solutions.

As a home-educated student, he's had the unique opportunity to focus his efforts on the fields that interest him. This has made him successful as the team captain for the FIRST Robotics teams: #4982 Café Bot and #1444 the Lightning Lancers. He has also served as a scientific research assistant to the Chairman of the Washington University Physics Department, where he wrote software to aid in the calculation of equations of state for dense matter in neutron stars.

He has received numerous awards and accolades as a result of his involvement in various programs. Two of his most notable achievements are FIRST Robotics Competition Dean's List Award and FIRST Tech Challenge World Championship Inspire Award.

For More Information:

Acknowledgments

First and foremost, I would like to thank Dr. Woodie Flowers for not only being an inspiration to me and my peers but also for graciously taking out the time to write the foreword for this title.

In addition, I would also like to thank all my friends and family who helped double-check my work during the lengthy process of writing this title, including Dr. Anne Jensen-Urstad, Ethan Michaelicek, Jonas Kersulis, and my parents, Kim and Robert Sanders.

Finally, I wish to thank my editors and reviewers, Richard Harvey, Shaon Basu, Melita Lobo, Edwin Moses, Rudi Chen, Justin Kuzma, Lamtharn Hantrakul, and Maria Montenegro for their continued commitment throughout the duration of writing this title.

For More Information:
www.packtpub.com/application-development/mastering-leap-motion
Mastering Leap Motion

The Leap Motion Controller is a revolutionary system that blends the boundary between man and machine, or at least, between our hands and monitors. What if you wanted to literally grab a 3D model with your hands and manipulate it however you want? This is where the Leap Motion device and its prospective developers (you!) come in. After using the powerful Leap API and with some innovation and a lot of patience, you will be aware how this new device has the potential to revolutionize the way we work with our computers.

As you are no doubt already familiar with the Leap device itself, we'll keep the introduction brief. As you've probably gathered from the title, this book is devoted to mastering the process of designing, writing, and testing programs for the Leap Motion Controller. It will teach you how to develop with the Leap Motion device effectively while creating a polished user experience. Throughout the book, we'll cover a broad range of topics ranging from the API basics to user experience all the way to robotics integration. Yes, robotics integration. As my primary field of study (and subsequently, one of my forms of recreation) is robotics, I've integrated a bunch of different robots with the Leap device, ranging from simplistic quadrotor simulators all the way to competition robots that cost thousands of dollars. I'll be sharing a few of these different projects with you toward the end of the book!

Once you start getting a hang of things, we'll create both 2D and 3D applications to put all of this knowledge to work. Of course, we'll also cover the troubleshooting and debugging of programs. After all, nothing is worse than an app that's broken because of no obvious reason, right?

This book emphasizes user experience, which is the most important thing when using the Leap Motion Controller. You might not think about it at first, but something as simple as user fatigue is a big problem when programming for the Leap. You will need to make sure that your user can use the application, but at the same time, you need to make sure that their hands don't get tired from a series of repetitive gestures or motions. We will cover problems such as this one as we progress through this book.

In keeping with the natural user interface that the Controller offers, a good Leap application should be simple, intuitive, and easy to use. It's okay if the underlying software is complex, but the user interface and its controls should be obvious. For example, instead of using a fancy U-shaped gesture to perform a simple task like an undo operation, why not just have the user make a quick swipe to the left with one of their hands? A good developer should always be on the lookout for opportunities to simplify their interface without making it unusable!

For More Information:  
www.packtpub.com/application-development/mastering-leap-motion
What This Book Covers

Chapter 1, *Introduction to the World of Leap Motion*, shows you how to set up and test the Leap Motion device and a programming environment to use with it. Once everything is set up, we'll review the API briefly and finish the chapter off with a simple example program to make sure everything's working.

Chapter 2, *What the Leap Sees – Dealing with Fingers, Hands, Tools, and Gestures*, covers the software and hardware side of any Leap Motion application. This includes basic tracking data such as hands and fingers as well as more advanced features such as tools and gestures. We'll finish off with an overview of some of the limitations that you might run into when working with the API and the device.

Chapter 3, *What the User Sees – User Experience, Ergonomics, and Fatigue*, covers the user side of any Leap Motion application. This includes when and when not to make use of the Leap in an application, the importance of ergonomics, and the prevention of user fatigue.

Chapter 4, *Creating a 2D Painting Application*, walks you through the creation of a two-dimensional (or 2D) painting application for the Leap. We'll start out simple with the basic framework and graphical frontend and then move straight into rendering user input onto the screen.

Chapter 5, *Creating a 3D Application – a Crash Course in Unity 3D*, introduces you to a three-dimensional (or 3D) toolkit (Unity 3D) to prepare you for the next few chapters. We'll cover the basic installation and setup of the environment, which is followed by the creation of a blank template project for use in the next few chapters.

Chapter 6, *Creating a 3D Application – Integrating the Leap Motion Device with a 3D Toolkit*, walks you through the basic steps of integrating the Leap Motion device with an external 3D toolkit. You'll learn how to render hands, fingers, and buttons. We'll finish off by covering the detection of user input via the Leap. We will be using C# in this chapter instead of Java.

Chapter 7, *Creating a 3D Application – Controlling a Flying Entity*, guides you through the completion of our 3D application. We'll create a 3D entity, retrieve user input from the Leap, and then use that data to control the entity.

Chapter 8, *Troubleshooting, Debugging, and Optimization*, is devoted to the inevitable things that will arise during application development: bugs and problems and optimization. This chapter will go over a few different things you can use to fix common problems with your device or application, in addition to a few general best practices.

Chapter 9, *Going beyond the Leap Motion Controller*, covers a variety of subjects that go beyond the Leap Motion device itself. I'll talk about what you've learned so far, where the Leap Motion stands next to other emerging technologies, some concerns regarding the reliability and safety of the device in the industry, and even some ideas to control robots!

For More Information:
www.packtpub.com/application-development/mastering-leap-motion
In this chapter, we will walk through the setup and installation of the various tools that you need to begin working with the Leap Motion Controller. You will learn how to install the Leap Motion Controller, the Developers' SDK and, optionally, an Integrated Development Environment called Eclipse. Afterwards, we'll go through the Leap Motion API in detail, including the API structure, basic terminology, and anatomy of a program. At the end of this chapter, we'll create a simple program that you can use as a stepping stone to the more complex Leap-driven applications that appear later in this book.

Keep in mind that a majority of the code-related jargon used throughout this book is relative to the Java programming language.

Should you come across any issues during this chapter, refer to Chapter 8, Troubleshooting, Debugging, and Optimization, for troubleshooting tips and tricks.

This chapter is sprinkled with periodic Fun facts that offer high-level and entry-level factoids about scripting and programming for your reading pleasure.

For More Information:
www.packtpub.com/application-development/mastering-leap-motion
Introduction to the World of Leap Motion

Setting up the Leap Motion device
If you've already set up your Leap Motion device, you can safely skip this section. Otherwise, read on!

This section (and the rest of the book) will assume that you are on the Windows operating system.

Compared to the early days of Leap Motion development (when regedit.exe was far too common), setting up the device is quite easy now. All you need to do is follow these simple steps to get your device up and running:

1. Plug in your device. It may or may not begin installing the firmware; no worries if it doesn't.
2. Following this, head on over to https://www.leapmotion.com/setup and download the Leap Motion installer for your platform (Windows in our case).
3. Once the installer has been downloaded, go ahead and run it; this will get your device fully set up.

Once the installer has completed, go to your Start menu in Windows, type in "Leap Motion Visualizer" in the search bar (you can show the search bar in Windows 8/8.1 by pressing Windows Key + Q) and hit Enter.

For More Information:
www.packtpub.com/application-development/mastering-leap-motion
You're all set! You can try out your device by waving your hands in front of the Leap Motion device; wireframe representations of your fingers and hands will be rendered on the screen by the Leap Motion Visualizer, similar to the preceding screenshot.

If things did not go as anticipated, skip to Chapter 8, Troubleshooting, Debugging, and Optimization, to learn about troubleshooting and debugging the Leap Motion Controller to see what went wrong.

**Installing the Leap Motion Developers' SDK**

Now that we've got your device installed and ready to go, we need to download the Leap Motion Developers' Software Development Kit, or SDK.

The SDK contains a series of language-specific libraries, DLLs, and examples for any developer to freely use. Needless to say, it's very important that you have it installed if you're going to develop anything for Leap! So, without further ado, let's get the SDK installed with the following steps:

1. Head to [https://developer.leapmotion.com/](https://developer.leapmotion.com/) and sign in. If you do not have a developer's account, create one when prompted.
2. Following this, navigate to the Downloads tab at the top of the Leap Motion website and click on it.
3. You should then see a page that looks something like the one shown in this step. Leap Motion will attempt to autodetect your platform, presenting a screen that looks similar to the following one. If the information is correct, accept the terms and conditions (without reading them, naturally) and begin the download!

---

For More Information:  
Once the download completes, extract the contents of the downloaded .zip file (assuming you're using Windows) to a safe place. We'll be referring back to this folder quite a bit later on. With that, you're done!

Installing the Java JDK

Before we install Eclipse, the IDE you'll be using for the remainder of this book, we need to install an appropriate Java Development Kit, or JDK. Oracle (the company that manages Java) is constantly changing its site, so here is a generalized step-by-step process to get a JDK installed:

2. Look for a box that says JDK Download in big letters. Click on the Download button.
3. You will be taken to a page with a list of JDKs and download options. Scroll down until you see your platform listed; in the case of Windows, the platform will be named either Windows x86 (32-bit systems) or Windows x64 (64-bit systems).
4. Accept the license agreement, download the file, and run it to install the JDK.

You're done! Now we can move onto installing Eclipse.

Setting up your IDE

With the Leap Motion software and SDK installed and out of our way, we can move onto getting your Integrated Development Environment, or IDE, installed and set up!

When I'm developing applications for the Leap Motion device (usually to control robots), I prefer to use the Eclipse Integrated Development Environment.

The Eclipse IDE was originally created for Java, but it has since expanded into many other languages, including C/C++ and Python. Throughout this book, all of my examples and instructions will assume you're using the Eclipse IDE. While you're welcome to use any other IDE, or even a text editor, I highly suggest that you install Eclipse, as it will make this book easier to use!
So first off, you have to find the IDE. Head to https://www.eclipse.org/downloads/ and download Eclipse Standard Edition, as shown in the following screenshot:

Make absolutely sure that you choose a version that matches your Java JDK (32-bit Eclipse for a 32-bit JDK, 64-bit Eclipse for a 64-bit JDK, and so on), or else there will be issues!

Once the download has completed (which can take a while, since the IDE is usually more than 150 megabytes in size), extract the ZIP file to wherever you’d like Eclipse to be located. Since Eclipse doesn’t use an installer, everything you need is contained in the folder you have just extracted.

At this point, there’s nothing left to do with the setup; we can now move onto the Leap Motion device itself!

**Structure of the Leap Motion Application Programming Interface (API)**

The Leap Motion API, while containing many complex and advanced features, is relatively simple. At the time of writing this book, there are about 23 different classes within the API, each one serving a different task.

Many of the classes are utilities that you won’t directly instantiate; FingerList, Handlist, ToolList, Pointable, and so forth are some examples of these kinds of classes. On the other hand (no pun intended), we have classes that contain data about a specific hand, finger, or other object, such as the Finger, Hand, and Tool classes. In almost all cases, the API classes are intuitively named and relatively easy to remember. Now, why don’t we go over some of the more common classes and what they do?

For More Information:  
www.packtpub.com/application-development/mastering-leap-motion
The Vector class

The Vector class is a kind of utility class. It contains and manages a single set of x, y, and z coordinates. It also has a slew of built-in functions to make more common operations easier to perform. You’ll find yourself using this class all the time, whether you know it or not. Almost every Leap API class makes use of it! The following is a brief example of how we can use the Vector class to check a hand’s yaw (or simply put, to detect the rotation of your hand):

```java
Vector position = new Controller().frame().hands().get(0).direction();
System.out.println("Hand 0 Yaw: " + position.yaw());
```

Fun fact

When using 3D coordinates, we use three special words to refer to the rotation of an object. These words are Pitch, Yaw, and Roll, and they correspond to the forward and backward tilt (pitch), left and right rotation (yaw) and left and right tilt (roll) of an object.

This will first create a new Vector object, fill it with directional data for the first hand in Leap’s field of view and then output its approximate yaw (in degrees) to the console window. This illustrates just one of the many functions you can perform with the Vector class.

The Finger class

Next on the list is the Finger class. This class contains tracking data for a single finger that is, or was, within Leap’s field of view. Similar to the Vector class we just covered, this class contains a multitude of coordinates (which happen to be vectors themselves). However, it also includes a series of other things, including the hand that it belongs to, the current position of its tip, the direction its tip is facing, and more. Here is an example of how we can retrieve the position of a finger’s tip:

```java
Finger finger = new Controller().frame().fingers().frontmost();
System.out.println("Frontmost Finger data:" + "\nTip Position (X|Y|Z): " + finger.tipPosition().getX() + "|" + finger.tipPosition().getY() + "|" + finger.tipPosition().getZ());
```
The Hand class

Next in line after the Finger class is the Hand class! This class is just like the Finger class in respect to what it contains, with the addition of a few more items like the Sphere Radius and Palm Position of the hand. The Sphere Radius of the hand is the approximate radius of the biggest sphere that the hand can hold. Note that while it isn't effective for gauging the overall size of a hand, it can be useful to detect how spread apart a hand's fingers are. The Palm Position of the hand is relatively straightforward; this is the position of the hands palm in the x, y, and z coordinates. Here is an example of how we can retrieve the position of a hand's palm:

```java
Hand hand = new Controller().frame().hands(0);
System.out.println("First Hand data:"
                   + 
                   "\nPalm Position (X|Y|Z): "
                   + hand.palmPosition().getX() + "|" +
                   hand.palmPosition().getY() + "|" + hand.palmPosition().getZ());
```

The preceding example is almost identical to our previous one with the Finger class; it simply outputs the three vector coordinates of the oldest hand in view to the console window.

The Frame class

Now for the Frame class. Before we delve into the specifics of this class too much, it's important to understand that the Leap Motion Controller works using frames of information; that is, it's much like a video game or movie. Frames are processed as they are received from Leap, and as such, have no set refresh rate (or frame rate).

Each frame contains a complete set of tracking data for all of the hands, fingers, tools, gestures, and other things that were within Leap's field of view when the frame was taken. The following code counts the objects within Leap's field of view for a single frame:

```java
Frame frame = new Controller.frame();
System.out.println("Frame data:
                   + "\nHand count: " +
                   frame.hands().count() + "\nFinger count: " +
                   frame.fingers().count() + "\nTool count: " +
                   frame.tools().count() + "\nTimestamp: " + frame.timestamp());
```

The preceding example will retrieve the most recent frame from the Leap device and output the number of fingers, hands, and tools along with a timestamp to the console window.

The Leap Motion API is kind enough to cache the most recent sixty frames, allowing us to look at data from previous frames for whatever reason. To retrieve a specific frame, use the following syntax:

```java
Frame frame = new Controller.frame(frameNumber);
```

For More Information:
www.packtpub.com/application-development/mastering-leap-motion
Here, `frameNumber` is the number, 1-60, of the frame that you want to fetch. The oldest frame in memory is number 60, whereas the newest one is 1. To get the current frame, do not specify any frame number at all.

**The Controller class**

The Controller class at last! This class is your portal to the world of Leap motion, so to speak; you'll notice that we've used it in all of the previous mini-examples up until this point. The class itself is rather simple on the outside. The two methods that we'll be using most often are `frame` and `addlistener`. The `frame` method, as the name would suggest, returns the most recent frame received from Leap. The `addlistener` method, on the other hand, is used to register a custom `Listener` class with the device; this brings us to the next topic.

**The Listener class**

The `Listener` class, well, listens. To be specific, it's an event-driven class (or a callback, if you will) that is registered with the `Controller` class and responds to various things that are going on with Leap. It's a little hard to explain it with just text, so let's look at a practical example here, which is the `MyListener.java` file:

```java
public class MyListener extends Listener {
    public void onFrame(Controller controller) {
        Frame frame = controller.frame();

        if (!frame.hands().empty())
            System.out.println("First Hand data:
            \nPalm Position (X|Y|Z): "+ frame.hand(0).palmPosition().getX() + "|" + frame.hand(0).palmPosition().getY() + "|" + frame.hand(0).palmPosition().getZ());
    }
}
```

Then, in a separate file, `Main.java`, use this:

```java
public class Main {
    public static void main(String args[]) {
        Controller controller = new Controller();

        MyListener listener = new MyListener();
    }
}
```

For More Information:

```
www.packtpub.com/application-development/mastering-leap-motion
```
This was a bit longer than our previous examples, wasn't it? Let's break it down into chunks. First, let's look at the lines in the first class, MyListener:

```java
public class MyListener extends Listener
```

The first part of this class defines our new class, which will be extending the Listener class; this is what allows it to be compatible with the Leap API. Following this, let's take a look at the next line:

```java
public void onFrame(Controller controller)
```

The second part of this class defines an override for the Listener class' internal onFrame method so that we can respond whenever a new frame is received from the controller. Thusly, this function (once this class is registered, of course) will be called every time a new frame is generated by Leap. The next few lines from that point should be fairly familiar at this point; if they aren't, head back and review the other classes that we've covered so far!

Now, let's look at the lines of the Main class. The first few lines should be familiar (they simply initialize the controller), so I'm only going to focus on one particular line:

```java
controller.addListener(listener);
```

This line registers our instance of the MyListener class with the Leap API, allowing it to receive events from Leap. In this case, once registered, the onFrame method from our MyListener instance will be called every time a new frame is received from the controller.

Then, there's the last line:

```java
while (true) {} 
```

This line prevents our main class from exiting, allowing our Listener implementation to continue running and receiving events until we're ready to quit.

---

**Fun fact**

The Leap Motion Listener system runs inside its own thread at all times; this means that any listeners that you register will continue to receive updates, even if your application is single threaded and receives a blocking call or something similar.

---

For More Information:

Creating a simple framework program within the Eclipse IDE

Based on everything we've done this far, we can go ahead and create a basic framework program to build off. The end result will be an all but empty program that continuously outputs tracking data from a single hand within Leap's field of view. You can then use this template with the examples featured later on within this book. In addition, this section will help familiarize you with the Eclipse IDE if you haven't used it before.

So, without further ado, let's get started!

Setting up the project

First off, we need to create a new Eclipse Java project. This can be achieved by heading over to File | New | Java Project from within the IDE; you will then be greeted by a project creation wizard. Choose a name for your project (in this case, I called mine SimpleLeap) and then click on the Finish button.

Once your project is created, navigate to it in the Package Explorer window. The following screenshot illustrates what the package explorer should look like:
While my IDE layout will most likely drastically differ from yours, the preceding screenshot should give you a general idea of what the Package Explorer looks like (the giant orange arrow is pointing to it). It most conveniently displays all of your Java packages as a folder hierarchy, making navigating large and more complex projects a breeze.

For now, we should go ahead and create a new package to put our classes in. This can be achieved by right-clicking on the src folder for your project in Package Explorer and then going to New | Package tooltip. You can name it whatever you like; standard Java naming convention dictates that package names consist of your web address in the reverse order followed by the project name, so I named mine com.mechakana.tutorials.

Fun fact

Java developers use web addresses in the reverse order (like com.mechakana) as package names so that different developers can distribute packages with unique names. If I just named my package “tutorials”, it could potentially overwrite somebody else’s package named tutorials—if I use my own website and name it com.mechakana.tutorials, I can guarantee I’m the only person with that name.

Once our package is created, we can fill it with two empty files: SimpleLeap.java and LeapListener.java. To create a new class in Eclipse, right-click on our package (com.mechakana.tutorials in my case) and then go to New | Class. You should do this twice; once for the SimpleLeap class and again for the LeapListener class. Only specify a name for the class—keep all the other settings as their defaults.

Now, we need to configure the Leap libraries. While it would be super convenient to just start coding, it wouldn't do us a whole lot of good if we didn't have access to Leap, right? Fortunately, this is a relatively easy task that can be done in a few simple steps:

1. Navigate to your Leap Motion SDK directory that we created earlier and go to LeapSDK/lib/. Copy the LeapJava.jar file contained within to the root of our project in Eclipse. On Windows, you can simply drag the file from the Leap SDK folder into Eclipse and it will automatically be copied.

2. While we're still within LeapSDK/lib/, navigate one level deeper into the x86 folder. Copy the Leap.dll and LeapJava.dll files contained within to the root of our project in Eclipse.

For More Information:

www.packtpub.com/application-development/mastering-leap-motion
3. Our project should now look similar to the following screenshot when viewed from within the Eclipse IDE:

```
- SimpleLeap
  - src
    - com.mechakana.tutorials
      - LeapListener.java
      - SimpleLeap.java
  - JRE System Library [JavaSE-1.7]
    - Leap.dll
    - LeapJava.dll
    - LeapJava.jar
```

4. Now, with all of the items we need contained inside our project folder, hover over the SimpleLeap project in Package Explorer (inside Eclipse) and right-click on it. Navigate to Build Path | Configure Build Path in the tooltip that pops up.

5. Once inside, you will be presented with a window that contains a series of tabs, similar to the one pictured here:

For More Information:
www.packtpub.com/application-development/mastering-leap-motion
6. Navigate to the Libraries tab and hit the Add JARs... button. In the 
window that pops up, open our SimpleLeap project, navigate to the 
LeapJava.jar file and then double-click on OK.

With that, you should be done setting up the project for development! Aside from 
the creation of our two files, these steps are common to pretty much any Eclipse 
project that uses the Leap Motion device; be sure to remember them!

For more information about Eclipse, you can go to their official website at 

Let's write some code!

With our project set up and ready to go, let's start writing! First off, we need to 
create an implementation of the listener class. Go ahead and open up the 
LeapListener.java file, which we created earlier, and enter the following:

```java
package com.mechakana.tutorials;

import com.leapmotion.leap.*;

public class LeapListener extends Listener
{
    public void onFrame(Controller controller)
    {
        Frame frame = controller.frame();

        if (!frame.hands().isEmpty())
            System.out.println("First Hand data:" + 
            "{Palm Position (X|Y|Z): " + frame.hand(0).palmPosition().getX() + "|" + frame.hand(0).palmPosition().getY() + "|" + frame.hand(0).palmPosition().getZ());

        if (!frame.fingers().isEmpty())
            System.out.println("Frontmost Finger data:" + 
            "{Tip Position (X|Y|Z): " + frame.fingers().frontmost().tipPosition().getX() + "|" + frame.fingers().frontmost().tipPosition().getY() + "|" + frame.fingers().frontmost().tipPosition().getZ());
    }
}
```

Now, let's discuss what each major chunk of code does:

```java
import com.leapmotion.leap.*;
```

For More Information:
www.packtpub.com/application-development/mastering-leap-motion
This line imports the entire Leap API into the context of our file. Needless to say, without this statement, we cannot make use of Leap:

```java
public class LeapListener extends Listener
```

The preceding line defines our LeapListener class and states that it extends the Leap API's Listener class. This is what allows us to later register the class with a Controller object.

```java
public void onFrame(Controller controller)
```

This line overrides the Listener class' built-in onFrame method. Once our LeapListener class is registered with a Controller object, this method will be called every time a new frame is received from Leap.

The rest of the lines contained in this example code have been covered previously in this chapter, but basically, they collect tracking data from hands and fingers and output it to the console window.

Moving on, let's fill out the SimpleLeap class. Go ahead and open up SimpleLeap.java and fill it with the following code:

```java
package com.mechakana.tutorials;
import com.leapmotion.leap.*;

public class SimpleLeap()
{
    public static void main (String args[])
    {
        Controller controller = new Controller();

        LeapListener listener = new LeapListener();

        controller.addListener(listener);

        while (true) {} // This will cause the program to loop forever until it is closed
    }
}
```

This code should look pretty familiar to you; in fact, we've used it before in this chapter during the previous Listener example! Since there's nothing new to this code, I'll give you the rundown; it registers an instance of our LeapListener class with Leap so that it can receive callbacks. Once registered, it loops forever until the user closes the application.
Trying it out

With all of the hard stuff out of the way, we can finally test out our code! With Eclipse, it's quite easy for us to launch a console application; simply click on the green arrow at the top of your toolbar or press Ctrl + F11. If everything worked correctly, you'll see a console tab pop up somewhere within your workspace. Try waving your hands around in front of the Leap device, and you'll see an output similar to the following screenshot:

```
Frontmost Finger data:
Tip Position (X|Y|Z): 43.56153|188.03516|-10.34844
First Hand data:
Palm Position (X|Y|Z): 0.0|0.0|0.0

Frontmost Finger data:
Tip Position (X|Y|Z): 58.68755|194.34201|14.91054
First Hand data:
Palm Position (X|Y|Z): 0.0|0.0|0.0

Frontmost Finger data:
Tip Position (X|Y|Z): 77.96336|199.8300|42.57053
First Hand data:
Palm Position (X|Y|Z): 0.0|0.0|0.0

Frontmost Finger data:
Tip Position (X|Y|Z): 96.6447|202.90314|72.20538
First Hand data:
Palm Position (X|Y|Z): 0.0|0.0|0.0

Frontmost Finger data:
Tip Position (X|Y|Z): 116.53129|203.0597|103.52548
First Hand data:
Palm Position (X|Y|Z): 0.0|0.0|0.0

Frontmost Finger data:
Tip Position (X|Y|Z): 53.260206|138.96620|144.40932
First Hand data:
Palm Position (X|Y|Z): 0.0|0.0|0.0

Frontmost Finger data:
Tip Position (X|Y|Z): 160.90007|200.58589|188.61331
First Hand data:
Palm Position (X|Y|Z): 0.0|0.0|0.0

Frontmost Finger data:
Tip Position (X|Y|Z): 170.15788|190.04822|215.4569
First Hand data:
Palm Position (X|Y|Z): 0.0|0.0|0.0

Frontmost Finger data:
Tip Position (X|Y|Z): 151.57468|173.46835|269.0885
First Hand data:
Palm Position (X|Y|Z): 0.0|0.0|0.0
```

With that, you're ready to dive into the world of Leap Motion.

Let's get started!

For More Information:
www.packtpub.com/application-development/mastering-leap-motion
Looking forward – the Skeletal Tracking API

Recently, the Leap Motion development team has released a new version of the Leap SDK (v2.0), which contains an all new API for skeletal tracking (as in, you know, tracking the bones in your hand).

This new API brings not only a slew of new possibilities, but also an increase in the precision of current tracking and vision recognition functions. As the new API was still in the beta phase during this book with no stable release in sight (beta software in a book isn't fun for anyone!), we will not be making heavy usage of it in the projects that we work on.

With that said, as it is unlikely that the skeletal tracking API will change too much in the coming months and the ideas it brings to the table are relatively straightforward, I thought we might as well go over what it has to offer compared to the current API. Depending on your level of experience with the Leap Motion Controller up to this point, some of the things I talk about in the next few pages might or might not mean much to you—but I assure you, it's all relevant!

Different fingers? Not a problem

We'll be covering the primary disadvantages of Leap and how to correct them later in this book, but I'm going to go ahead and say it now: the new Skeletal Tracking API eliminates a lot of the issues present in the current API. It is, in simple terms, freaking awesome.

With the new API, individual fingers can now be differentiated from each other. This probably seems like something rather simple, but it can make a world of difference in many cases. Let's take a first-person shooter for example.

During normal gameplay, the player holds his or her primary hand out in front of the Leap device in the form of a gun (making a fist with only the index finger and thumb pointing out). Once the player has aimed at a target they want to shoot, they'll close their thumb on their hand as if firing the gun. This will cause the in-game weapon to fire in the direction aimed by their finger, ideally.

Now, if you can't tell the difference between the thumb and the pinky finger, or an index finger and a ring finger, this seemingly trivial programming task—calling a function when the user's thumb disappears from view or enters a specific threshold—suddenly becomes a beast of a programming task; this is where the new Skeletal Tracking API comes to the rescue!

For More Information:
www.packtpub.com/application-development/mastering-leap-motion
Earlier, we differentiated the fingers on a hand by trying various tricks like sorting the $x$ indices of each finger from left to right or right to left and then guessing and based on this data, which finger belonged to which "named" finger (such as a thumb, index, or pinky). We'll be covering this in the next few chapters. However, with the new API, all we have to do is check the `type` field on any valid finger object and the API will work its magic, providing you with an integer that represents the proper name for the finger that the object denotes.

The bare-bones snippet here illustrates, in two lines, how you'd query the finger type data using the new API:

```java
if (frame.hand(0).finger(0).type == 0)
    System.out.println("The first finger on the first hand is a thumb.");
```

The preceding example snippet will print out a notice if, and only if, the first finger on the first hand in a given frame is a thumb (finger 0 on hand 0, since the Leap Motion API is zero indexed).

**Handedness is no longer an issue**

Gone are the days of wondering if a given hand belonged to somebody's right arm or left arm. Using the magic of mathematics and other unknown formulae, relating to the structure of the skeleton of the average human hand, the shiny new API can now differentiate between the left and right hands.

Again, this is not as simple as it sounds. There are times when you want to figure out whether a user has their left or right hand out; why, I won't ask, but there are such times. Using the preskeletal tracking API, one of your best bets for figuring out which hand is which is detecting the side of a hand the farthest finger (the thumb) was on; if the thumb was on the left, it's a right hand, and vice versa. Of course, this is little better than a guess—what if the user's hand is upside down, for instance? You'd then have to take that into account, and so on, and so on...not fun.

Yet again, the new Skeletal Tracking API comes to the rescue with the `isLeft()` and `isRight()` functions, which call all of our `Hand` objects home in the new API! I certainly hope that the names of these functions are self-explanatory, but just in case, for the sake of clarification, I shall clarify. The `isLeft()` function for any `Hand` object will return a result of `true` if the object is a left hand and `false` otherwise. Likewise, the `isRight()` function for any `Hand` object will return a result of `true` if the object is a right hand and `false` otherwise. This is far superior to the previous method of guessing.

For More Information:

www.packtpub.com/application-development/mastering-leap-motion
Having confidence in tracking data

This is slightly less of a concern to the average developer, but it is a concern nonetheless. The confidence that your software has that a tracked hand is, indeed, a hand! The amount of confidence the Skeletal Tracking API has about the accuracy of a hand is based on how well the tracking data for a hand matches a predetermined model of an ideal hand, including posture and finger positions.

In the 2.0 Beta API, this confidence value is rated on a scale of 0.0 to 1.0, with 0.0 meaning it is extremely unlikely that a tracked hand is valid, while a value of 1.0 means it is extremely likely that a tracked hand is valid.

Since every Hand object has a confidence rating, all you have to do to get the confidence rating for a hand is call the confidence() member function of a Hand object and you will get a return value of 0.0 to 1.0, like so:

```java
System.out.println("First Hand Confidence Rating: " +
        String.valueOf(frame.hand(0).confidence()));
```

The preceding example snippet will print out the confidence rating of the first hand in a given frame to your console window.

Pinching and grabbing are now much easier

Gone are the days of trying to detect gestures or interpolating finger coordinates in an attempt to pick up pinches and grabs from the user. The new API introduces two new member functions for the Hand class that are specifically for detecting grabbing (closing of the fist) and pinching—grabStrength() and pinchStrength(), respectively.

The first function, grabStrength(), will return a value between 0.0 and 1.0 that represents how tightly a given hand is making a fist. A value at or near 0.0 means a hand is almost perfectly flat and not making a fist, while a value at or near 1.0 means a hand is making a tight fist. In other words, the more a hand curls its fingers inward into the form of a fist, the higher the returned value from grabStrength() will be. Similar to the other new methods we've covered already, usage of this one is relatively simple:

```java
System.out.println("First Hand Grab Strength: " +
        String.valueOf(frame.hand(0).grabStrength()));
```

The preceding example snippet will print out the approximate grab strength of the first hand in a given frame to your console window.
Moving on to the second function, `pinchStrength()` will return a (you guessed it) value between 0.0 and 1.0 that represents how close a hand’s thumb is to any other finger on the hand. This one is just a tad more confusing than the other functions; basically, a hand’s pinching strength is defined by a hand’s thumb being very close to any other finger on the hand—it doesn’t matter which finger, as long as it’s a finger.

A value at or near 0.0 means that a hand’s thumb isn’t very close to any other finger, and therefore the hand isn’t pinching very hard (or at all). On the other hand, a value at or near 1.0 means that a hand’s thumb is very close to other fingers and is pinching rather hard. Basically, the closer a hand’s thumb is to another finger, the higher the returned value from `pinchStrength()` will be. Just like `grabStrength()`, `pinchStrength()` is relatively simple to use:

```java
System.out.println("First Hand Pinch Strength: "+ String.valueOf(frame.hand(0).pinchStrength()));
```

The preceding example snippet will print out the approximate pinch strength of the first hand in a given frame to your console window.

**A new API class – Bones**

Of course, the Skeletal Tracking API update wouldn’t make much sense if it didn’t introduce a facility for dealing with, well, skeletons.

This is where the new `Bone` class comes in. Using it, you can glean all kinds of information from any valid `Finger` object, such as the length of the metacarpals, proximal phalanges, intermediate phalanges, and distal phalanges on a given finger. Don’t worry; I didn’t know what these names meant either until I read the shiny new API docs for developers. We won’t go too in depth about this new class since the API was (or still is) in the beta stage at the time of writing this, but let’s go ahead and touch on the basics.

In the new Skeleton Tracking API, every single `Finger` object contains an array of `Bone` objects—four to be precise. The exception is for thumbs; while they technically have four `Bone` objects, they only have three literal bones in real life. Therefore, the length of the metacarpal bone (the one closest to the hand, which thumbs don’t have) on a thumb will always be zero.

Every `Bone` object has a variety of useful and/or fun member functions, including:

- The standard `isValid()` and `isValid()` functions to check data integrity
- The `length()` function to check the length of the given bone
- The `prevJoint()` function that returns the `Vector` object for the point the given bone is anchored to
Introduction to the World of Leap Motion

- nextJoint() that returns the Vector object for the tip of the given bone
- type() for getting the proper name, or type, of the given bone

For what it’s worth, the names of the individual bones on a finger, in the order from the closest to the hand to the furthest from the hand, are: the metacarpal bone, proximal phalange bone, intermediate phalange bone, and the distal phalange bone.

That's it!

As you can see, the new Skeletal Tracking API is poised to revolutionize the way the Leap Motion software will be written by developers like you and me. Everything is easier, from simple things such as picking out named fingers to the more complex things such as iterating over the bones of different fingers to create more accurate 3D representations of our hands. Only time will tell what awesome and crazy applications will be written using this new API. Now... back to Mastering the Leap Motion Controller!

Summary

In this chapter, we covered all of the basics of the Leap Motion device. You installed and set up the Leap Motion device, its libraries, and an IDE to help you program for it. We then went over all of the different members of the Leap Motion API, including vectors, fingers, hands, frames, controllers, and listeners. Afterwards, you created a simple framework program with Eclipse to make sure everything was working correctly. We finished off with a lengthy look at the new Skeletal Tracking API and what it brings to the development table. Armed with this knowledge, you are now ready to tackle the next few chapters of this book.

In the next chapter, we'll begin diving into the world of Leap Motion. You'll be learning about how Leap interprets your hands, various ways you can detect user input, and some unavoidable limitations of the tracking software.

For More Information:
www.packtpub.com/application-development/mastering-leap-motion
Where to buy this book

You can buy Mastering Leap Motion from the Packt Publishing website: https://www.packtpub.com/application-development/mastering-leap-motion.

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.