1

Objects All the Way Down

WHAT'S IN THIS CHAPTER?

- A look at Kotlin syntax
- A brief history of Kotlin
- How Kotlin is like Java—and how it isn't
- Getting set up to code and run Kotlin
- Your first Kotlin program
- Why objects are cool (and why that matters)

KOTLIN: A NEW PROGRAMMING LANGUAGE

Kotlin, when you boil it all down, is just another programming language. If you're programming and writing code already, you'll pick up Kotlin quickly, because it has a lot in common with what you're already doing. That's even more the case if you're programming in an object-oriented language, and if you're coding in Java, well, Kotlin is going to feel very familiar, although different in some very nice ways.

If you're new to Kotlin, though, it's a great first language. It's very clear, it doesn't have lots of odd idioms (like, for example, Ruby or, god help us all, LISP), and it's well organized. You'll pick it up fast and find yourself comfortable quite quickly.

In fact, Kotlin is so straightforward that we're going to put aside a lot of explanation and history for now, and instead jump right into looking at some basic Kotlin code (check out Listing 1.1).

LISTING 1.1: A simple Kotlin program using classes and lists

```
data class User(val firstName: String, val lastName: String)

fun main() {
  val brian = User("Brian", "Truesby")
  val rose = User("Rose", "Bushnell")

  val attendees: MutableList<User> = mutableListOf(brian, rose)

  attendees.forEach {
    user -> println("$user is attending!")
  }
}
```

Take a minute or two to read through this code. Even if you've never looked at a line of Kotlin before, you can probably get a pretty good idea of what's going on. First, it defines a `User` class (and in fact, a special kind of class, a data class; more on that later). Then it defines a `main` function, which is pretty standard fare. Next up are two variables (or `val`s), each getting an instance of the `User` class defined earlier. Then a list is created called `attendees` and filled with the two users just created. Last up is a loop through the list, doing some printing for each.

If you ran this code, you'd get this rather unimpressive output:

```
User(firstName=Brian, lastName=Truesby) is attending!
User(firstName=Rose, lastName=Bushnell) is attending!
```

Obviously, parts of this probably look odd to you, whether you're brand new to writing code or an experienced Java pro. On top of that, it's likely you have no idea how to actually compile or run this code. That's OK, too. We'll get to all of that.

> **NOTE** *It bears repeating: You really don't need to understand the code in Listing 1.1. This book assumes you've programmed at least a little bit—and it's true that you'll likely understand Kotlin a bit faster if you have a Java background—but you will pick up everything you see in Listing 1.1 (and quite a bit more) just by continuing to read and working through the code samples. Just keep going, and you'll be programming in Kotlin in no time.*

For now, though, here's the point: Kotlin is really approachable, clean to read, and actually a pretty fun language to use. With that in mind, let's get some basics out of the way so you can get to writing code, not just looking at it.

WHAT IS KOTLIN?

Kotlin is an open-source programming language. It is, most notably, statically typed and object-oriented. Statically typed means that variables have types when you write your code and compile it, and those types are fixed. That also implies that Kotlin must be compiled, which is also true. Object-oriented means it has classes and inheritance, making it a familiar language for Java and C++ developers.

Kotlin was in fact created by a group of developers that worked on the JetBrains IDE, and it feels very much like a natural evolution of Java. It's been around in early form since 2011, but was officially released in 2016. That means it's new, which is good, but also means it's new, which at times can be bad. Kotlin is modern, can run inside a Java Virtual Machine (JVM), and can even be compiled to JavaScript—a cool feature we'll look at a little later.

It's also really important to note that Kotlin is a fantastic language for writing Android apps. In fact, many of its enhancements to Java reflect an Android usage. That said, even if you never intended to write a mobile app, you'll find Kotlin a welcome addition to your arsenal, and well suited for server-side programming.

What Does Kotlin Add to Java?

That's a good question that has a long answer. In fact, we'll spend most of this book answering that in various forms. But, for most, Kotlin adds or changes a few key features when compared to Java:

> **NOTE** *If you're new to Kotlin or not coming from a Java background, feel free to skip right on to the next section.*

- Kotlin ditches `NullPointerException` (and nullable variables altogether) in almost all situations.
- Kotlin supports extending functions without having to entirely override a parent class.
- Kotlin doesn't support checked exceptions (you may not find this to be an advancement, so fair warning).
- Kotlin adds components of functional programming, such as extensive lambda support and lazy evaluation.
- Kotlin defines data classes that let you skip writing basic getters and setters.

There's certainly a lot more to this list, but you can quickly see that Kotlin isn't just a slightly different version of Java. It seeks to be different and better, and in many ways, it very much is exactly that.

KOTLIN IS OBJECT-ORIENTED

At this point, most books and tutorials would have you put together a simple "Hello, World" program. That's all well and good, but the assumption here is that you want to get moving, and get moving quickly. For that reason, the logical place to begin with Kotlin is by creating an object.

An object is simple a programmatic representation of a thing. In the best case, that thing is a real-world object, like a car or a person or a product. For example, you could create an object to model a person like this:

```
class Person {

  /* This class literally does nothing! */

}
```

That's it. You can now create a new variable of type `Person` like this:

```
fun main() {
    val jennifer = Person()
}
```

If you put all this together into a single code listing, you'll have Listing 1.2.

LISTING 1.2: A very useless object in Kotlin (and a main function to use it)

```
class Person {

  /* This class literally does nothing! */

}

fun main() {
    val jennifer = Person()
}
```

Now, this is pretty lame code, honestly. It doesn't do anything, but it is object-oriented. Before we can improve it, though, you need to be able to run this good-for-almost-nothing code yourself.

INTERLUDE: SET UP YOUR KOTLIN ENVIRONMENT

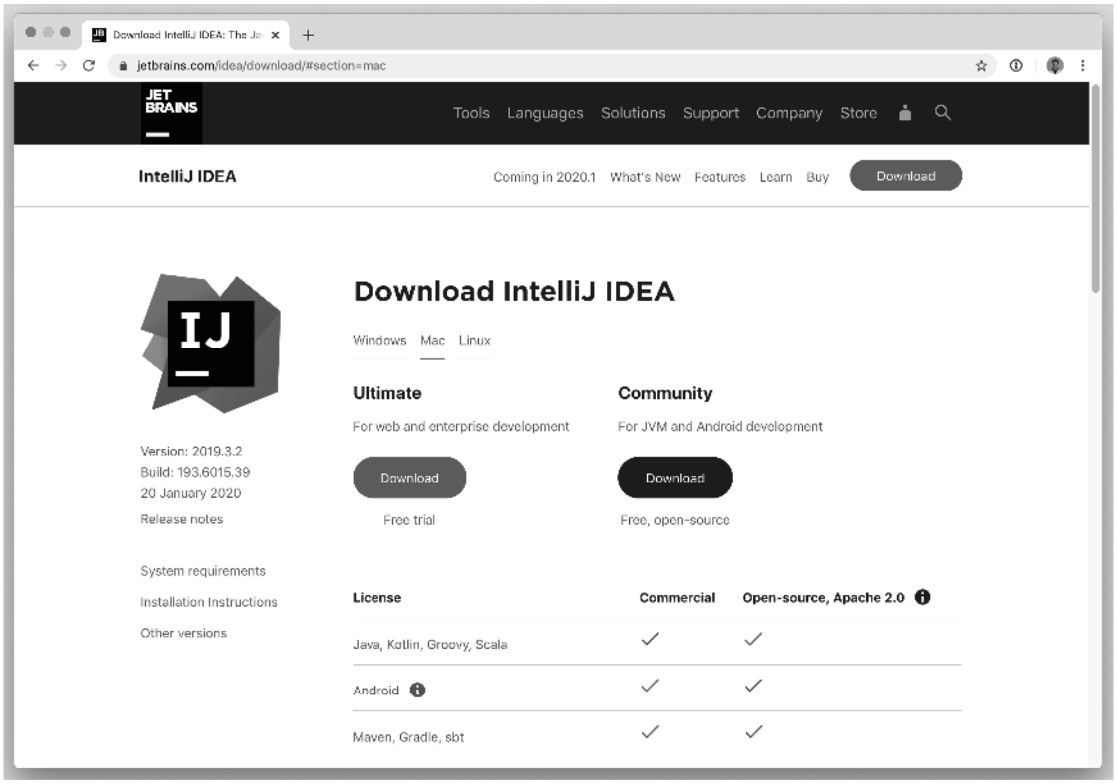
Getting a Kotlin program to run is relatively easy, and if you're an old Java hand, it's actually *really* easy. You'll need to install a Java Virtual Machine and then a Java Development Kit (JDK). Then you'll want one of the numerous IDEs that support Kotlin. Let's take a blazing-fast gallop through that process.

Install Kotlin (and an IDE)

One of the easiest IDEs to use with Kotlin is IntelliJ IDEA, and starting with version 15, IntelliJ comes bundled with Kotlin. Plus, since IntelliJ is actually from JetBrains, you're getting an IDE built by the same folks who came up with Kotlin itself.

Install IntelliJ

You can download IntelliJ from `www.jetbrains.com/idea/download`. This page (shown in Figure 1.1) will then redirect you to the appropriate platform (Mac OS X for most of this book's examples). Download the free Community version to get started without any cost. Once the (rather large) download completes, install it (see Figure 1.2), and you'll get a Java Runtime Environment (JRE) and the JDK as part of installation.



Download IntelliJ IDEA

Windows Mac Linux

Ultimate
For web and enterprise development
Download
Free trial

Community
For JVM and Android development
Download
Free, open-source

Version: 2019.3.2
Build: 193.6015.39
20 January 2020
Release notes

System requirements
Installation Instructions
Other versions

License	Commercial	Open-source, Apache 2.0
Java, Kotlin, Groovy, Scala	✓	✓
Android	✓	✓
Maven, Gradle, sbt	✓	✓

FIGURE 1.1: Download IntelliJ from the JetBrains download page.

NOTE *IntelliJ is not the only IDE that works with Kotlin, and the list is actually growing pretty quickly. Other notable options are Android Studio (*`developer.android.com/studio/preview/index.html`*) and Eclipse (*`www.eclipse.org/downloads`*). Eclipse in particular is immensely popular, but IntelliJ is still a great choice as it shares the JetBeans heritage with Kotlin.*

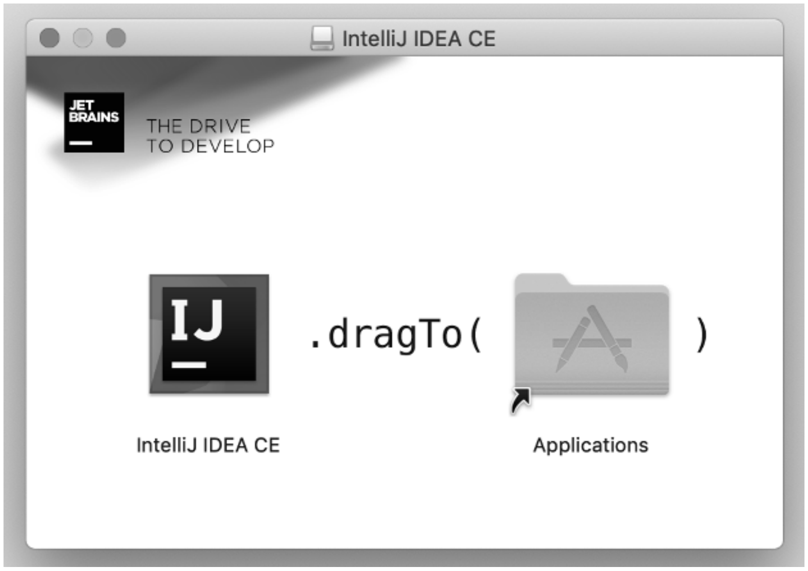


FIGURE 1.2: IntelliJ comes prepackaged with a system-specific installation process.

> **NOTE** *The "installation process" for IntelliJ on Mac OS X is pretty simple: just drag the package (presented as an icon) into your Applications folder. You'll then need to go to that folder and launch IntelliJ or drag the icon into your Dock, which is what I've done.*
>
> *For Windows, you download the executable and run it. You can then create a shortcut on your desktop if you like.*
>
> *In both cases, you can use the JetBrains Toolbox (which comes with the JetBrains Kotlin package) to keep your installation current and add updates when they're available.*

You'll be given a pretty large number of options to get your IDE set up. For IntelliJ, you'll pick a UI theme (either is fine), a Launcher Script (I'd suggest you accept the default and let it create the script), the default plugins, and a set of featured plugins. You can click through these quickly, and then your IDE will restart. You'll see a welcome screen similar to Figure 1.3, and you should select Create New Project.

> **WARNING** *You may need advanced permissions to install the Launcher Script that IntelliJ creates if you accepted the default location on Mac OS X.*

Be sure you select the Kotlin/JVM option when creating the project, as shown in Figure 1.4.

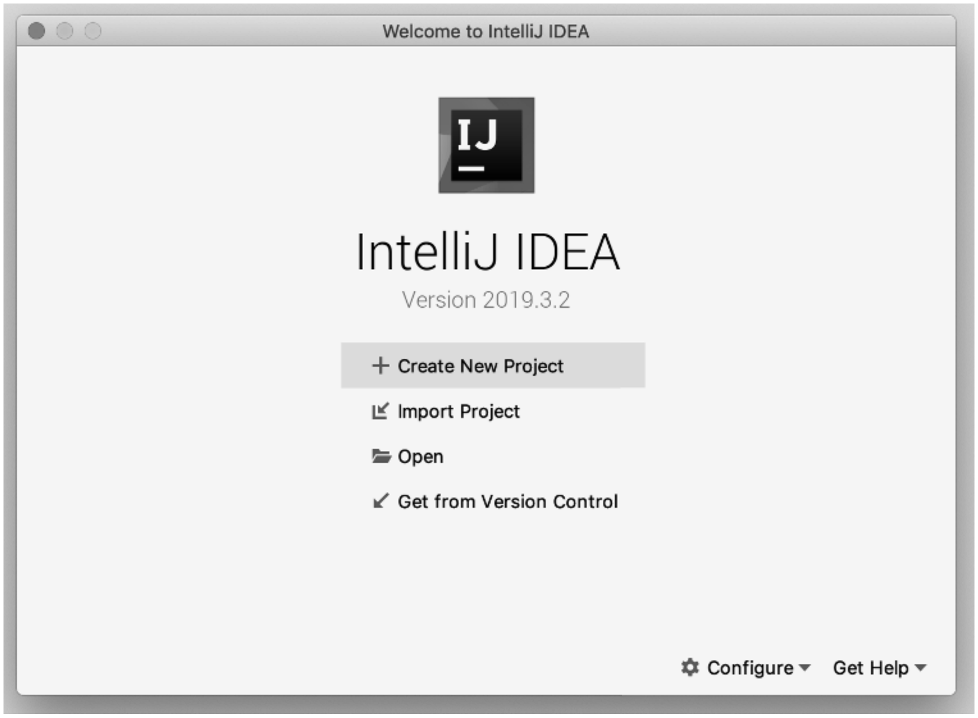
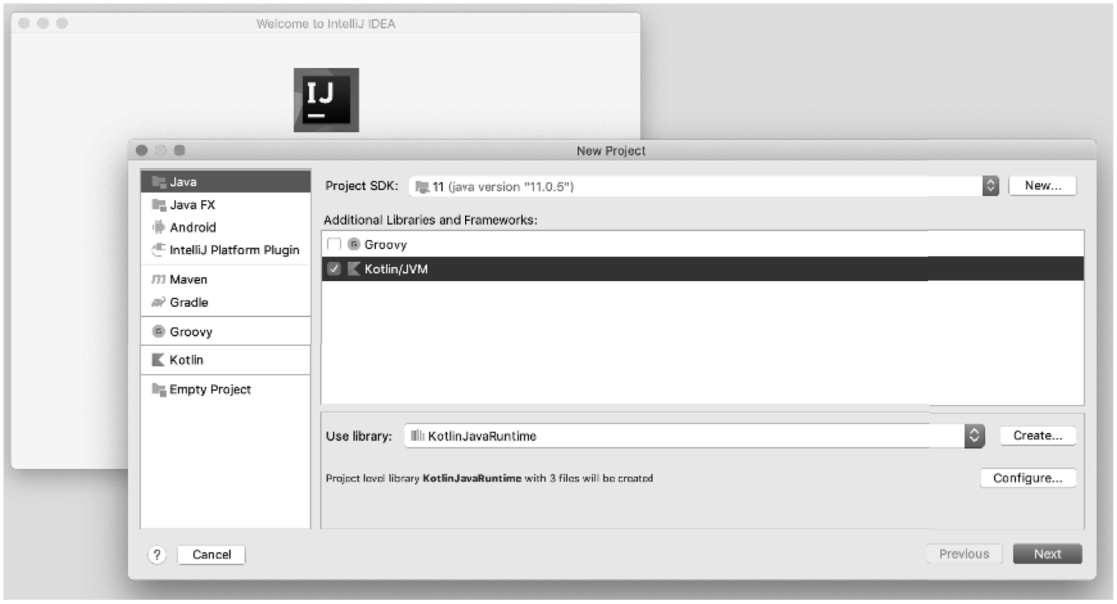


FIGURE 1.3: You'll generally be either creating a project from scratch or importing one from a code repository, like GitHub.



Project SDK: 11 (java version "11.0.5")

Additional Libraries and Frameworks:

Use library: KotlinJavaRuntime

Project level library **KotlinJavaRuntime** with 3 files will be created

FIGURE 1.4: IntelliJ makes getting going in Kotlin simple and prompts you on creating a new project to include Kotlin libraries.

Create Your Kotlin Program

Once your project is up and running, create a new Kotlin file. Find the `src/` folder in the left navigation pane, right-click that folder, and select New ➢ Kotlin File/Class (see Figure 1.5). You can enter the code from Listing 1.2, and it should look nice and pretty, as shown in Figure 1.6 (thanks IntelliJ!).

NOTE *Your IDE may not be configured exactly like mine. If you don't see the* `src/` *folder, you may need to click Project on the left side of your IDE to display the various folders, and possibly click again on the name of the project.*

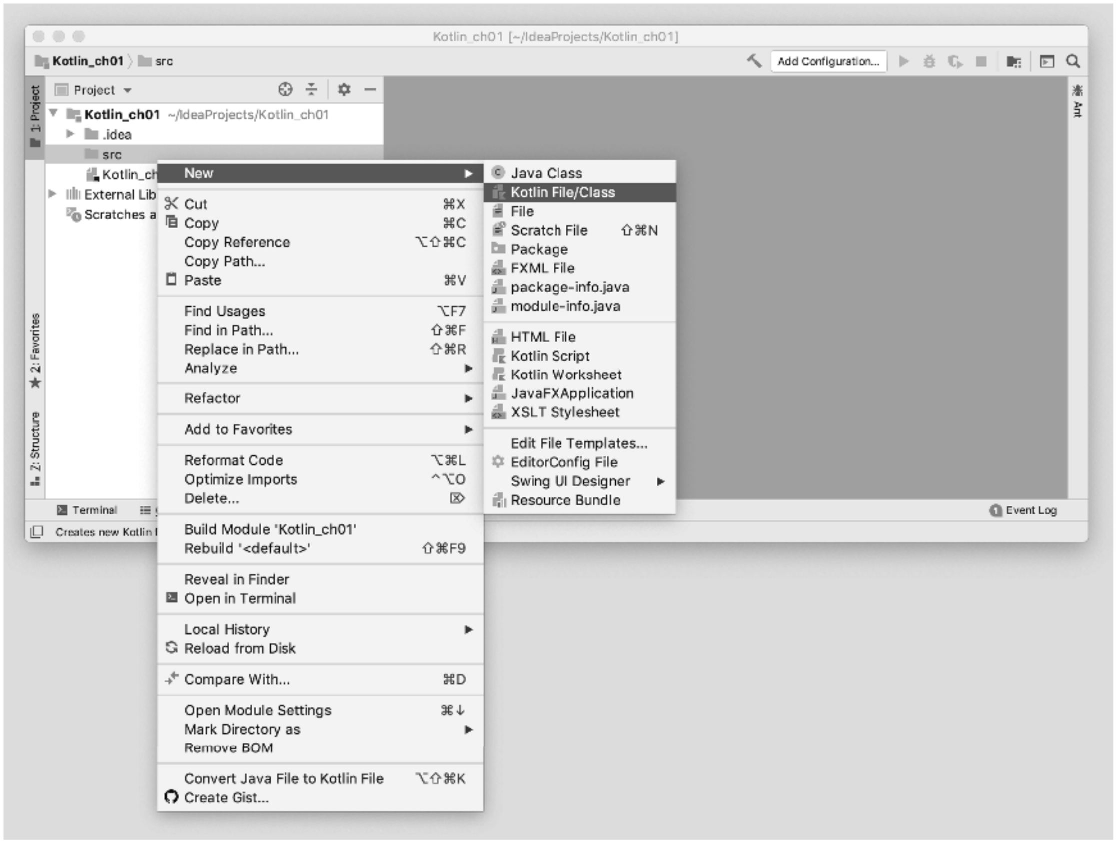
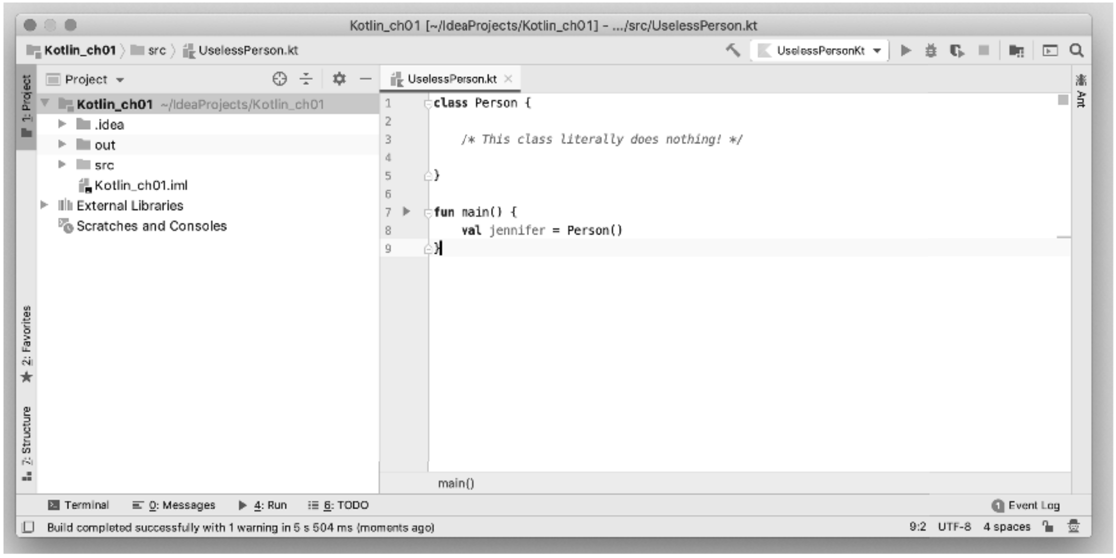


FIGURE 1.5: Kotlin code should go in the src/ folder.

NOTE *From this point forward, code will typically not be shown in an IDE. That way, you can use the IDE of your choice (or the command line), because you should get the same results across IDEs.*



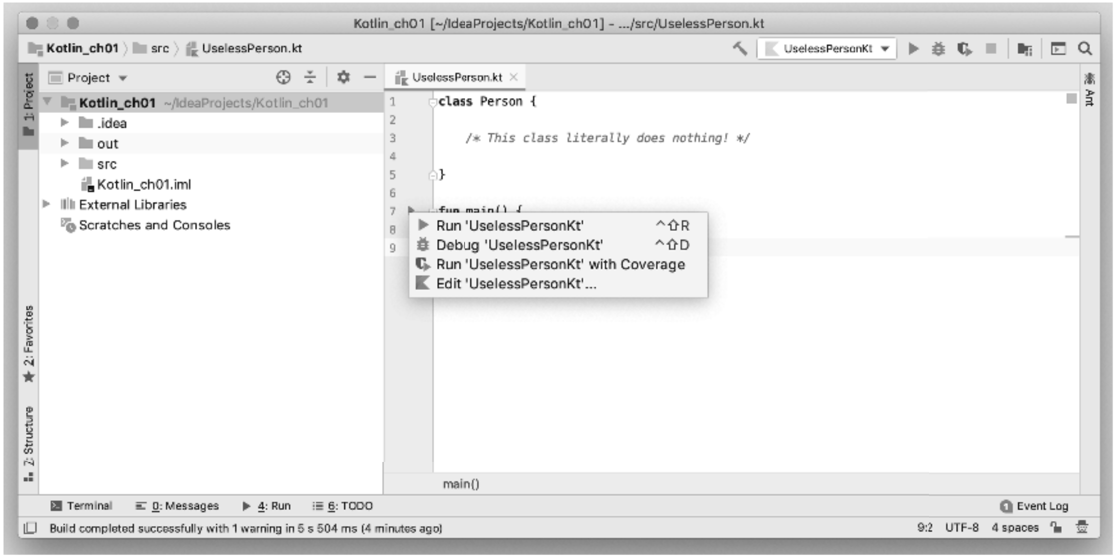
```
class Person {

    /* This class literally does nothing! */

}

fun main() {
    val jennifer = Person()
}
```

```
Build completed successfully with 1 warning in 5 s 504 ms (moments ago)
```

FIGURE 1.6: IntelliJ automatically formats code and adds sensible syntax highlighting.

Compile and Run Your Kotlin Program

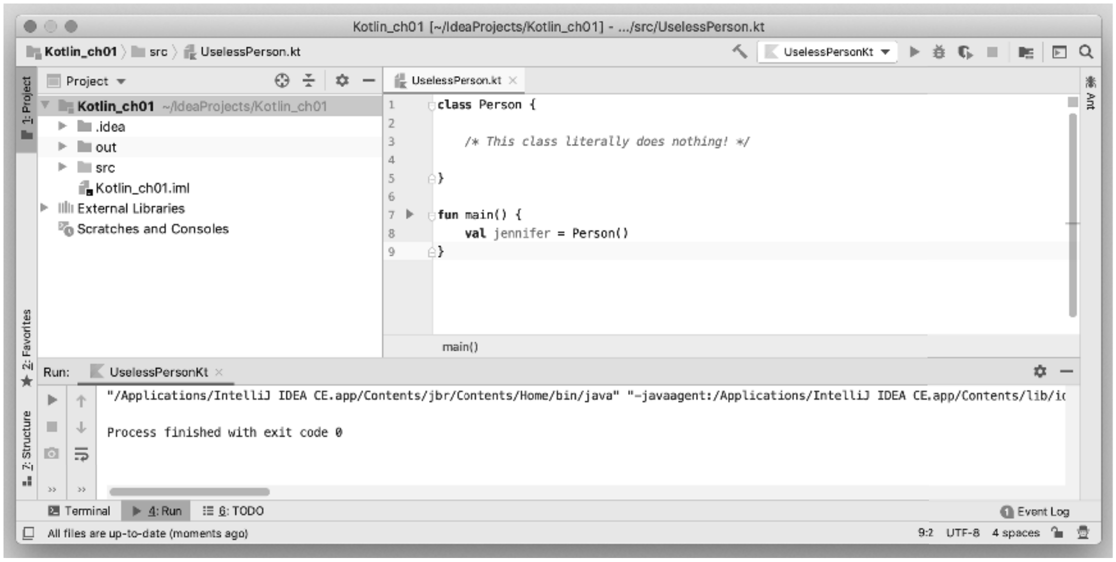
All that's left now is to compile and run the program yourself. This is easy, because IntelliJ gives you a convenient little green arrow to click when you have a Kotlin file with a `main()` function defined. Just hover over the arrow and click (see Figure 1.7). You can then select Run and your filename (I named mine "UselessPerson"). Your program will be compiled and run, with the output shown in a new pane at the bottom of the IDE (see Figure 1.8).



```
class Person {

    /* This class literally does nothing! */

}

fun main() {
```

```
Build completed successfully with 1 warning in 5 s 504 ms (4 minutes ago)
```

FIGURE 1.7: You can click the green Run button and select the first option to build and run your code.


Kotlin_ch01 [~/IdeaProjects/Kotlin_ch01] - .../src/UselessPerson.kt
Kotlin_ch01 › src › UselessPerson.kt
UselessPersonKt
Project
Kotlin_ch01 ~/IdeaProjects/Kotlin_ch01
.idea
out
src
Kotlin_ch01.iml
External Libraries
Scratches and Consoles
UselessPerson.kt
main()
Run: UselessPersonKt
Terminal
4: Run
6: TODO
Event Log
All files are up-to-date (moments ago)
9:2 UTF-8 4 spaces

```
class Person {

    /* This class literally does nothing! */

}

fun main() {
    val jennifer = Person()
}
```

```
"/Applications/IntelliJ IDEA CE.app/Contents/jbr/Contents/Home/bin/java" "-javaagent:/Applications/IntelliJ IDEA CE.app/Contents/lib/ic

Process finished with exit code 0
```

FIGURE 1.8: The empty output of your program (which will soon be non-empty) displays in its own window.

In this case, you shouldn't get any errors, but there's not any output either. We'll fix that shortly.

Fix Any Errors as They Appear

One last note before getting back to improving that useless `Person` class. IntelliJ and all other IDEs are great at giving you visual indicators when there is a problem with your code. For example, Figure 1.9 shows IntelliJ once it's tried to compile the same program with an error. In this case, the open and close parentheses are missing from line 8. You'll see an orange indicator in the code editor and an error indicating line 8 (and column 20) in the output window.

You can then easily fix the error and rebuild.

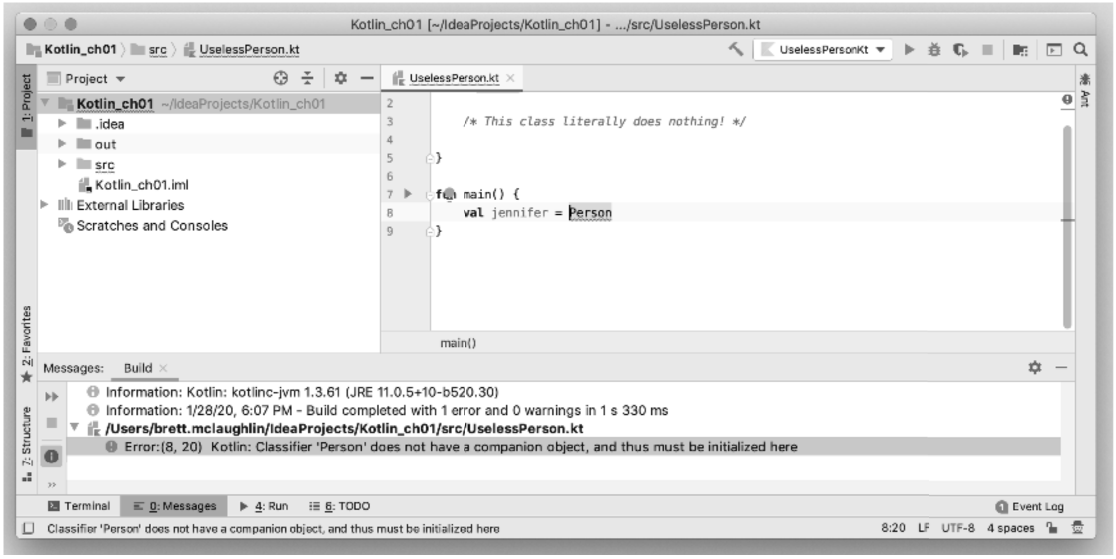
Install Kotlin (and Use the Command Line)

For power users, there's a tendency to want to use the command line for nearly everything. Kotlin is no exception. Because it's "mostly Java" in the sense that it runs using a JVM and JDK, you can get pretty far without a lot of work.

Command-Line Kotlin on Windows

For Windows users, you'll first need a JDK. You can download one from the Oracle Java download page at `www.oracle.com/technetwork/java/javase/downloads`. That download has version-specific instructions that are easy to follow.

Once you have Java, you need to get the latest Kotlin release from GitHub. You can find that at `github.com/JetBrains/kotlin/releases/latest` (that link will redirect you to the latest release). Download the release and follow the instructions and you'll be good to go.



```
    /* This class literally does nothing! */

}

fun main() {
    val jennifer = Person
}
```

```
Information: Kotlin: kotlinc-jvm 1.3.61 (JRE 11.0.5+10-b520.30)
Information: 1/28/20, 6:07 PM - Build completed with 1 error and 0 warnings in 1 s 330 ms
/Users/brett.mclaughlin/IdeaProjects/Kotlin_ch01/src/UselessPerson.kt
    Error:(8, 20)  Kotlin: Classifier 'Person' does not have a companion object, and thus must be initialized here
```

```
Classifier 'Person' does not have a companion object, and thus must be initialized here
```

FIGURE 1.9: Good IDEs help you quickly find and fix errors.

NOTE *These instructions are intentionally a bit sparse. If you're using the command line already, you probably don't need a lot of hand holding. For almost everyone else, though, using an IDE really is the best approach. As a bonus, you can also use IntelliJ as a proxy for the compiler, so you may just want to save the time it would take you to mess with the command line and put it into coding Kotlin!*

Command-Line Kotlin on Mac OS X

The easiest path to getting Kotlin working on Mac OS X is to use one of the package managers popular on Macs: either Homebrew (`brew.sh`) or MacPorts (`www.macports.org`). Both of these make getting Kotlin up and running trivial.

For MacPorts, just run the following command:

```
brett $ sudo port install kotlin
```

This requires elevated permissions, but after it runs, you'll be all set.

For Homebrew, first do an update:

```
brett $ brew update
```

Next up, install Kotlin:

```
brett $ brew install kotlin
```

Command-Line Kotlin on UNIX-Based Systems

If you're not on Mac OS X but still have a Unix flavor of operating system, you can use SDKMAN! (`sdkman.io`) for installing Kotlin.

> **NOTE** *To be accurate, Mac OS X is a Unix-based operating system, so you can use the SDKMAN! instructions for Macs instead of Homebrew or MacPorts.*

First, get SDKMAN!:

```
brett $ curl -s https://get.sdkman.io | bash
```

When you're finished, you'll need to open a new terminal or shell window or source the modified file as indicated at the end of the installation process.

Now, install Kotlin:

```
brett $ sdk install kotlin
```

Verify Your Command-Line Installation

However you've chosen to install Kotlin, when you're finished, you should be able to validate your installation with this command:

```
brett $ kotlinc
```

> **WARNING** *At this point, you may get prompted to install a Java runtime. This should fire up your system to handle this, and you can accept the prompts without a lot of worry. Find the JDK or JRE for your system, download it, and run it. Then come back and try out* `kotlinc` *again.*

If you have your system appropriately configured with Java, you should get back something like this:

```
brett $ kotlinc

Java HotSpot(TM) 64-Bit Server VM warning: Options -Xverify:none and -noverify were
deprecated in JDK 13 and will likely be removed in a future release.

Welcome to Kotlin version 1.3.61 (JRE 13.0.2+8)

Type :help for help, :quit for quit

>>>
```

This is the Kotlin REPL (Read-Eval-Print Loop), a tool for quickly evaluating Kotlin statements. We'll look at this in more detail later, but for now, exit the REPL by typing **`:quit`**.

You can also verify your version of Kotlin with the following command:

```
brett $ kotlin -version

Kotlin version 1.3.61-release-180 (JRE 13.0.2+8)
```

At this point, you're ready to roll!

CREATING USEFUL OBJECTS

With a working Kotlin environment, it's time to go make that `Person` class from Listing 1.2 something less vacuous. As mentioned earlier, objects should model real-world objects. Specifically, if an object represents a "thing" in the world (or in formal circles you'll sometimes hear that objects are "nouns"), then it should have the properties or attributes of that thing, too.

A person's most fundamental property is their name, specifically a first name and last name (or surname, if you like). These can be represented as properties of the object. Additionally, these are required properties; you really don't want to create a person without a first and last name.

For required properties, it's best to require those properties when creating a new instance of an object. An *instance* is just a specific version of the object; so you might have multiple instances of the `Person` class, each one representing a different actual person.

WARNING *As you may already be figuring out, there's a lot of technical vocabulary associated with classes. Unfortunately, it will get worse before it gets better: instances and instantiation and constructors and more. Don't get too worried about catching everything right away; just keep going, and you'll find that the vocabulary becomes second nature faster than you think. In Chapter 3, you'll dive in deeper, and in fact, you'll keep revisiting and growing your class knowledge throughout the entire book.*

Pass In Values to an Object Using Its Constructor

An object in Kotlin can have one or more constructors. A *constructor* does just what it sounds like: it constructs the object. More specifically, it's a special method that runs when an object is created. It can also take in property values—like that required first and last name.

You put the constructor right after the class definition, like this:

```
class Person constructor(firstName: String, lastName: String) {
```

In this case, the constructor takes in two properties: `firstName` and `lastName`, both `String` types. Listing 1.3 shows the entire program in context, along with creating the `Person` instance by passing in the values for `firstName` and `lastName`.

WARNING *You'll sometimes hear properties or property values called parameters. That's not wrong; a parameter is usually something passed to something else; in this case, something (a first name and last name) passed to something else (a constructor). But once they're assigned to the object instance, they're no longer parameters. At that point, they are properties (or more accurately, property values) of the object. So it's just easier to call that a property value from the start.*

See? The terminology is confusing. Again, though, it will come with time. Just keep going.

LISTING 1.3: A less useless object in Kotlin and its constructor

```
class Person constructor(firstName: String, lastName: String) {

    /* This class still doesn't do much! */

}

fun main() {
    val brian = Person("Brian", "Truesby")
}
```

Now the class takes in a few useful properties. But, as most developers know, there's a tendency to condense things in code. There's a general favoring of typing less, rather than typing more. (Note that this rarely applies to book authors!) So Listing 1.3 can be condensed; you can just drop the word `constructor` and things work the same way. Listing 1.4 shows this minor condensation (and arguable improvement).

LISTING 1.4: Cutting out the constructor keyword

```
class Person(firstName: String, lastName: String) {

    /* This class still doesn't do much! */

}

fun main() {
    val brian = Person("Brian", "Truesby")
}
```

Print an Object with toString()

This is definitely getting a little better. But the output is still empty, and the class is still basically useless. However, Kotlin gives you some things for free: most notably for now, every class automatically gets a `toString()` method. You can run this method by creating an instance of the class (which you've already done) and then calling that method, like this:

```
val brian = Person("Brian", "Truesby")

println(brian.toString())
```

Make this change to your `main` function. Create a new `Person` (give it any name you want), and then print the object instance using `println` and passing into `println` the result of `toString()`.

> **NOTE** *You may be wondering where in the world that* `toString()` *method came from. (If not, that's OK, too.) It does seem to sort of magically appear. But it's not magical it all. It's actually inherited. Inheritance is closely related to objects, and something we'll talk about in a lot more detail in both Chapter 3 and Chapter 5.*

Terminology Update: Functions and Methods

A few specifics on terminology again. A *function* is a piece of code that runs. `main` is an example of a function. A *method* is, basically, a function that's attached to an object. Put another way, a method is also a piece of code, but it's not "stranded" and self-standing, like a function is. A method is defined on an object and can be run against a specific object instance.

> **NOTE** *In much of the official Kotlin documentation, there is not a clear distinction drawn between a function and a method. However, I've chosen to draw this distinction because it's important in general object-oriented programming, and if you work or have familiarity with any other object-based language, you'll run across these terms. But you should realize that in "proper" Kotlin, all methods are functions.*

That last sentence is important, so you may want to read it again. It's important because it *means* that a method can interact with the object instance. For example, a method might want to use the object instance's property values, like, say, a first name and last name. And with that in mind, back to your code!

Print an Object (and Do It with Shorthand)

You can run the `println` function at any time, and you just pass it something to print. So you could say:

```
println("How are you?")
```

and you'd just get that output in your results window. You can also have it print the result from a method, like `toString()`, which is what you did earlier. But there's another shortcut. If you pass in something to `println()` that has a `toString()` method, that method is automatically run. So you can actually trim this code:

```
println(brian.toString())
```

down to just this:

```
println(brian)
```

In the latter case, Kotlin sees an object passed to `println()` and automatically runs `brian.toString()` and passes the result on for printing. In either case, you'll get output that looks something like this:

```
Person@7c30a502
```

That's not very useful, is it? It's essentially an identifier for your specific instance of `Person` that is useful to Kotlin internals and the JVM, but not much else. Let's fix that.

Override the toString() Method

One of the cool things about a class method is that you can write code and define what that method does. We haven't done that yet, but it's coming soon. In the meantime, though, what we have here is slightly different: a method that we *didn't* write code for, and that *doesn't* do what we want.

In this case, you can do something called *overriding* a method. This just means replacing the code of the method with your own code. That's exactly what we want to do here.

First, you need to tell Kotlin that you're overriding a method by using the `override` keyword. Then you use another keyword, `fun`, and then the name of the method to override, like this:

```
override fun toString()
```

NOTE *Earlier, you learned the difference between a function and a method. And* `toString()` *is definitely a method, in this case on* `Person`*. So why are you using the* `fun` *keyword? That looks an awful lot like "function," and that's, in fact, what it stands for.*

The official answer is that Kotlin essentially sees a method as a function attached to an object. And it was easier to not use a different keyword for a standalone function and an actual method.

But, if that bugs you, you're in good company. It bugs me, too! Still, for the purposes of Kotlin, you define both functions and methods with `fun`*.*

But `toString()` adds a new wrinkle: it returns a value. It returns a `String` to print. And you need to tell Kotlin that this method returns something. You do that with a colon after the parentheses and then the return type, which in this case is a `String`:

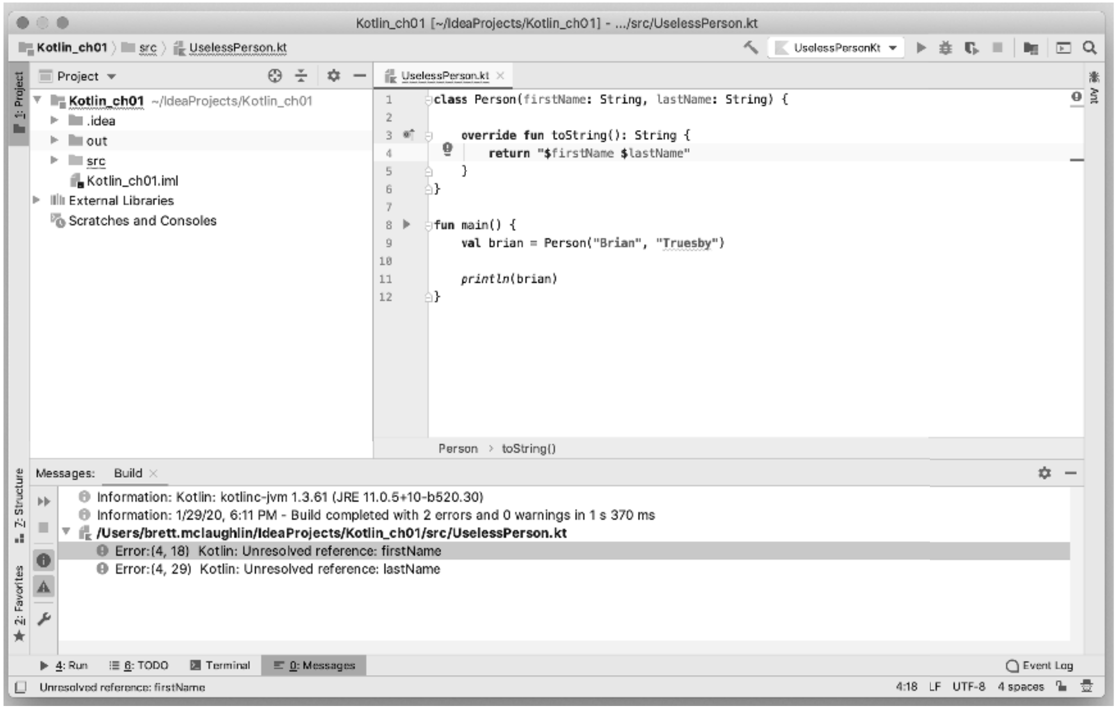
```
override fun toString(): String
```

Now you can write code for the method, between curly braces, like this:

```
class Person(firstName: String, lastName: String) {

    override fun toString(): String {
        return "$firstName $lastName"
    }
}
```

This looks good, and you've probably already figured out that putting a dollar sign (`$`) before a variable name lets you access that variable. So this takes the `firstName` and `lastName` variables passed into the `Person` constructor and prints them, right?

Well, not exactly. If you run this code, you'll actually get the errors shown in Figure 1.10.



```
class Person(firstName: String, lastName: String) {

    override fun toString(): String {
        return "$firstName $lastName"
    }
}

fun main() {
    val brian = Person("Brian", "Truesby")

    println(brian)
}
```

```
Information: Kotlin: kotlinc-jvm 1.3.61 (JRE 11.0.5+10-b520.30)
Information: 1/29/20, 6:11 PM - Build completed with 2 errors and 0 warnings in 1 s 370 ms
/Users/brett.mclaughlin/IdeaProjects/Kotlin_ch01/src/UselessPerson.kt
    Error:(4, 18)  Kotlin: Unresolved reference: firstName
    Error:(4, 29)  Kotlin: Unresolved reference: lastName
```

Unresolved reference: firstName

FIGURE 1.10: Why doesn't this override of toString() work?

What gives here? Well, it turns out to be a little tricky.

All Data Is Not a Property Value

You have a constructor and it takes in two pieces of data: `firstName` and `lastName`. That's controlled by the constructor declaration:

```
class Person(firstName: String, lastName: String) {
```

But here's what is tricky: just accepting those values does not actually turn them into property values. That's why you get the error in Figure 1.10; your `Person` object accepted a first and last name, but then promptly ignored them. They aren't available to be used in your `toString()` overridden method.

You need to use the `val` keyword in front of each piece of data to turn that data into property values. Here's the change you need to make:

```
class Person(val firstName: String, val lastName: String) {
```

Specifically, by using `val` (or `var`, which we'll talk about shortly), you've created variables, and assigned them to the `Person` instance being created. That then allows those variables (or properties, to be even more precise) to be accessed, like in your `toString()` method.

Make these changes (see Listing 1.5 to make sure you're caught up) and then compile and run your program.

LISTING 1.5: Converting data to actual properties

```
class Person(val firstName: String, val lastName: String) {

    override fun toString(): String {
        return "$firstName $lastName"
    }
}

fun main() {
    val brian = Person("Brian", "Truesby")

    println(brian)
}
```

You should get a single line of output:

```
Brian Truesby
```

Obviously, the name will be different if you passed in different values for first and last name, but the result is the same, and it's a big deal. You've now:

- Created a new object
- Defined a constructor for the object
- Accepted two pieces of data in that constructor and stored them as properties associated with the object instance
- Overridden a method and made it useful
- Written a `main` function
- Instantiated your custom object and passed in values
- Used the object to print itself out, using your overridden method

Not too bad for getting started! There's just one more detail to work through before closing shop on your first foray into Kotlin.

INITIALIZE AN OBJECT AND CHANGE A VARIABLE

Suppose you want to play around a bit with your `Person` class. Try this out: update your code to match Listing 1.6 (some of this may be confusing, but you can probably figure out most of what's going on).

LISTING 1.6: Creating a new property for a Person

```
class Person(val firstName: String, val lastName: String) {
    val fullName: String

    // Set the full name when creating an instance
    init {
        fullName = "$firstName $lastName"
    }

    override fun toString(): String {
        return fullName
    }
}

fun main() {
    // Create a new person
    val brian = Person("Brian", "Truesby")

    // Create another person
    val rose = Person("Rose", "Bushnell")

    println(brian)
}
```

You'll see a number of new things here, but none are too surprising. First, a new variable is declared inside the `Person` object: `fullName`. This is something you've already done in your `main` function. This time, though, because you're doing it inside the `Person` object, it automatically becomes part of each `Person` instance.

Another small change is the addition of a new `Person` instance in `main`; this time it's a variable named `rose`.

Then, there's a new keyword: `init`. That bears further discussion.

Initialize a Class with a Block

In most programming languages, Java included, a constructor takes in values (as it does in your `Person` class) and perhaps does some basic logic. Kotlin does this a bit differently; it introduces the idea of an initializer block. It's this block—identified conveniently with the `init` keyword—where you put code that should run every time an object is created.

This is a bit different than you may be used to: data comes in through a constructor, but it's separated from the initialization code, which is in the initializer block.

In this case, the initializer block uses the new `fullName` variable and sets it using the first and last name properties passed in through the class constructor:

```
// Set the full name when creating an instance
init {
    fullName = "$firstName $lastName"
}
```

Then this new variable is used in toString():

```
        override fun toString(): String {
    return fullName
}
```

WARNING *As much new material as this chapter introduces, you may have just run across the most important thing that you may learn, in the long-term sense. By changing* `toString()` *to use* `fullName`*, rather than also using the* `firstName` *and* `lastName` *variables directly, you are implementing a principle called DRY: Don't Repeat Yourself.*

In this case, you're not repeating the combination of first name and last name, which was done already in the initializer block. You assign that combination to a variable, and then forever more, you should use that variable instead of what it actually references. More on this later, but take note here: this is a big deal!

Kotlin Auto-Generates Getters and Setters

At this point, things are going well. Part of that is all you've added, but another big help is that Kotlin is doing a lot behind the scenes. It's running code for you automatically (like that initializer block) and letting you override methods.

It's doing something else, too: it's auto-generating some extra methods on your class. Because you made `firstName` and `lastName` property values (with that `val` keyword), and you defined a `fullName` property, Kotlin created getters and setters for all of those properties.

Terminology Update: Getters, Setters, Mutators, Accessors

A getter is a method that allows you to get a value. For instance, you can add this into your `main` function, and it will not only work, but print out just the first name of the `brian Person` instance:

```
// Create a new person
val brian = Person("Brian", "Truesby")
println(brian.firstName)
```

This works because you have a getter on `Person` for `firstName`. You can do the same with `fullName` and `lastName`, too. This getter is, more formally, an *accessor*. It provides access to a value, in this case a property of `Person`. And it's "free" because Kotlin creates this accessor for you.

Kotlin also gives you a setter, or (again, more formally) a *mutator*. A mutator lets you mutate a value, which just means to change it. So you can add this into your program:

```
// Create a new person
val brian = Person("Brian", "Truesby")
println(brian.firstName)

// Create another person
val rose = Person("Rose", "Bushnell")
rose.lastName = "Bushnell-Truesby"
```

Just as you can get data through an accessor, you can update data through mutators.

WARNING *For the most part, I'll be calling getters accessors, and calling setters mutators. That's not as common as "getter" or "setter," but as a good friend and editor of mine once told me, a setter is a hairy and somewhat fluffy dog; a mutator lets you update class data. The difference—and his colorful explanation—has stuck with me for 20 years.*

Now, if you've gone ahead and compiled this code, you've run into yet another odd error, and that's the last thing to fix before moving on from this initial foray into objects.

Constants Can't Change (Sort of)

Here's the code causing the problem:

```
// Create another person
val rose = Person("Rose", "Bushnell")
rose.lastName = "Bushnell-Truesby"
```

If you try to run this code, you'll get an error like this:

```
Error: Kotlin: Val cannot be reassigned
```

One of the things that is fairly unique about Kotlin is its strong stance on variables. Specifically, Kotlin allows you to not just declare the type of a variable, but also whether that variable is a *mutable* variable, or a *constant* variable.

NOTE *The terminology here is a bit confusing, so take your time. Just as with methods being declared with the* `fun` *keyword, the idea of a constant variable takes a little getting used to.*

When you declare a variable in Kotlin, you can use the keyword `val`, as you've already done:

```
val brian = Person("Brian", "Truesby")
```

But you can also use the keyword `var`, something you *haven't* done yet. That would look like this:

```
var brian = Person("Brian", "Truesby")
```

First, in both cases, you end up with a variable; `val` does not stand for value, for example, but is simply another way to declare a variable, alongside `var`. When you use `val`, you are creating a constant variable. In Kotlin, a constant variable can be assigned a value once, and only once. That variable is then constant and can never be changed.

You created the `lastName` variable in `Person` with this line:

```
class Person(val firstName: String, val lastName: String) {
```

That defines `lastName` (and `firstName`) as a constant variable. Once it's passed in and assigned when the `Person` instance is created, it can't be changed. That makes this statement illegal:

```
rose.lastName = "Bushnell-Truesby"
```

To clear up the odd error from earlier, what you need instead is for `lastName` to be a *mutable* variable; you need it to be changeable after initial assignment.

> **NOTE** *Not to beat a hairy and somewhat fluffy dog to death, but here is another reason to use mutator over setter; a mutator allows you to mutate a mutable variable. This aligns the terminology much more cleanly than using "setter."*

So change your `Person` constructor to use `var` instead of `val`. This indicates that `firstName` and `lastName` can be changed:

```
class Person(var firstName: String, var lastName: String) {
```

Now you should be able to compile the program again, without error. In fact, once you've done that, make a few other tweaks. You want to end up with your code looking like Listing 1.7.

LISTING 1.7: Using mutable variables

```
class Person(var firstName: String, var lastName: String) {
    var fullName: String

    // Set the full name when creating an instance
    init {
        fullName = "$firstName $lastName"
    }

    override fun toString(): String {
        return fullName
    }
}
```

```
fun main() {
    // Create a new person
    val brian = Person("Brian", "Truesby")
    println(brian)

    // Create another person
    val rose = Person("Rose", "Bushnell")
    println(rose)

    // Change Rose's last name
    rose.lastName = "Bushnell-Truesby"
    println(rose)
}
```

Here, `fullName` has been made mutable, and there's a little more printing in `main`. It should compile and run without error now.

But wait! Did you see your output? There's a problem! Here's what you probably get when you run your code:

```
Brian Truesby
Rose Bushnell
Rose Bushnell
```

Despite what Meat Loaf had to say about two out of three not being bad, this is not great. Why is Rose's name in the last instance not printing with her new last name?

Well, to solve that, it's going to take another chapter, and looking a lot more closely at how Kotlin handles data, types, and more of that automatically running code.