CHAPTER 4

Objects and Object-Oriented Programming: OOP Primer

Now that you have had an overview of JSON in Chapter 2 and the JSON Schema Core and its components in Chapter 3, this chapter examines what defines object-oriented programming (OOP), including the concepts behind it, advantages of using OOP, terms used to describe OOP, and the difference between hard objects used in Java and soft objects used in JavaScript. Remember that JSON is used with JavaScript, Java, C++, and a great many other OOP languages, but it is best integrated with JavaScript and also integrates seamlessly with ECMAScript-262.

First you look at OOP concepts via Java, which is used on every OS as well as in the popular Android operating system. Java uses a more complex, hard object paradigm, because it requires classes and constructors to create an object. Then you examine OOP in JavaScript, which uses a soft object paradigm: objects can be created without using special classes and constructor methods. (By the way: *methods* are called *functions* in JavaScript.)

Object-Oriented Programming: Overview

The initial programming languages were *top down* or *linear* and either had line numbers, as in BASIC, or processed lines of code in the order that they appeared. As you learned in Chapter 2, the first OOP was Simula, and it introduced some very advanced concepts that made programming an order of magnitude more flexible, extensible, and powerful, all at the same time. These faculties also made OOP more difficult to learn and comprehend, which is why this book spends this chapter making sure you understand OOP and the two types of objects created in the two most popular programming languages today: Java and JavaScript.

This chapter starts with Java because chances are you use it for application development or mobile development; the chapter finishes with JavaScript and at that point also introduces the JSON object value and how to define a JSON object. JavaScript has fewer complexities than Java, so covering Java first gets those complexities out of the way. Then you can look at how JavaScript differs; it is less rigid regarding rules, which is why it is called a *soft object model* whereas Java is called a *hard object model*. As far as JSON goes, it is as widely used with Java as it is with JavaScript, so all of this chapter

W. Jackson, *JSON Quick Syntax Reference*, DOI 10.1007/978-1-4842-1863-1_4

is directly relevant to JSON no matter how you slice it. Java has extensive JSON class libraries, just as JavaScript has a plethora of JSON functions. Some of these, such as `JSON.parse()`, you saw in Chapter 2.

Java OOP Concepts: Hard Object Construction

Let's make sure you and I are on the same page by reviewing the core concepts and principles behind the Java programming language. This chapter gives you a primer or comprehensive overview of an entire OOP language. The Java JDK (and JRE) that you install in the appendixes of this book are the foundation for the JSON IDE and Java applications as well as for the NetBeans 8.1 IDE, which you saw in Chapter 1. You also learned the basics of how the IDE you are using to code JSON using Java or JavaScript (HTML5) applications functions as a code editor or application-testing tool.

Many of the core Java OOP constructs and principles covered in this chapter go back quite far in the Java programming language—most of them as far back as Java 1 (known as 1.02). The most widely used version of Java SE currently is Java 6 (1.6); Java EE uses Java 8. Android 4.4 and earlier use Java 6, until the advent of Android 5 and 6, which use Java 7 (1.7). This chapter also covers the features added in Java 8 (1.8), which is the most recent release, as well as the new features planned for Java 9 (1.9), which will be released in the fourth quarter of 2016. All these versions of Java are used on billions of devices, including Java 6, which is used in the 32-bit Android 2.x, 3.x, and 4.x OS, and applications; Java 7, used in the 64-bit Android 5.x and 6.x OS and applications; Java 8, used across all popular personal computer operating systems, such as Microsoft Windows, Apple, Open Solaris, and a plethora of popular Linux distributions (custom Linux OS versions) such as: SUSE, Ubuntu, Mint, Fedora, and Debian; and Java 9. This chapter covers the most foundational Java programming language concepts, techniques, principles, and structures that span these four major versions of Java that are currently in widespread use today on personal computers, iTV sets, and handheld devices, such as tablets and phones.

You start out with the easiest concepts and progress to the more difficult ones. The chapter begins at the highest level of Java—the Java API and its package—and progresses to the hands-on Java programming constructs contained in the Java packages, which are called Java *classes*. You learn about methods, as well as the variables or constants that classes contain, and what superclasses, subclasses, and nested classes are. Finally, you learn about Java `Objects` and how they form the foundation of OOP. You see what a constructor method is and how it creates a Java `Object` using a special kind of method that has the same name as the class in which it is contained.

Java Packages: Organizing a Java API Using Functional Classes

At the highest level of a programming platform—such as Google's 32-bit Android 4 or earlier, which uses Java SE 6; or 64-bit Android 5 or later, which uses Java SE 7; or the current Oracle Java SE platform, which was released as Java SE 8—there is a collection of packages that contain classes, interfaces, methods, and constants, which collectively form the *application programming interface (API)*. This collection of Java code can be used by application developers to create professional-level software across many OSs, platforms, and consumer electronics devices, such as desktops, laptops, netbooks, tablets, HD iTV sets, UHD iTV sets, eBook readers, and smartphones.

To install any given version of a Java API, you install the software development kit (SDK) as described in the book's appendixes. The Java SDK has a special name: the *Java development kit (JDK)*. If you're familiar with Android 5/6, which is actually Java 7 on top of Linux OS, you know that a new API level is released every time a few new features are added.

In addition to the API level defined by the SDK you install and use, the highest-level construct in the Java programming language is the *package*. You always use the `package` keyword to declare an application package at the top of your Java code. This needs to be the first line of code declared other than comments, which are not processed.

As you may have ascertained from the name, a Java package packages together all of your Java programming constructs. These include classes and methods (functions) that relate to the application: for example, a board game package contains all of your code, as well as the code you import to create, compile, and run the 3D board game. You take a look at the concept of importing and the Java `import` keyword next; they are closely related to the package concept.

A Java package is useful for organizing and containing your application code. But it is even more useful for organizing and containing the SDK's (API's) Java code that you use along with your own Java programming logic to create Java games or IoT applications.

You can use any of the classes that are part of the API packages you are developing with, by using the Java `import` keyword in conjunction with your package and the classes that you wish to use. This is called an *import statement*.

An import statement begins with `import`, followed by the package name and class reference path (full proper name); the statement needs to be terminated using a semicolon. For example, an import statement used to import the JavaFX `EventHandler` class from the `javafx.event` package should look like the following (also see Figure 4-1):

```
package invincibagel;              // custom invincibagel package for game
import javafx.event.EventHandler; // imports EventHandler into invincibagel
```

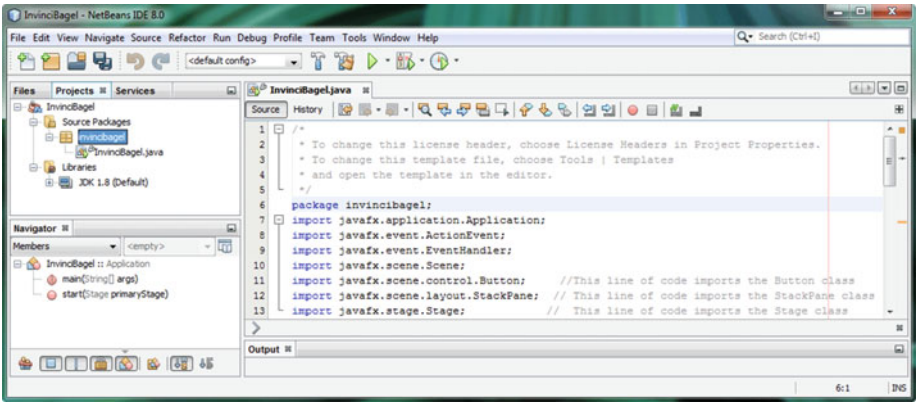

Figure 4-1. *The* `package` *and* `import` *keywords used in the Java code for the* `InvinciBagel` *game*

You can see the Java single-line comment `//` and multiline comment convention in this example, as well as in Figure 4-1. If you are interested in learning more about Java game programming, I took this example and screenshot from my book *Beginning Java 8 Game Programming* (Apress 2015), which covers the JavaFX new media engine.

An import statement informs the Java compiler that it needs to bring a specified external package into your custom package (in this example, import it into the `invincibagel` package), because you are using methods (and constants) from the class that is referenced using the `import` keyword, as well as specifying what package the class you are importing is stored in. If you use a class, method, or constants in your own Java class, such as the `BoardGame` class, and you haven't declared the class for use by using an import statement, the Java compiler will throw an error because it can't find the class it needs to use in your package.

Java Classes: OOP Modular Structures

The next-largest programming structure beneath the package level is the Java *class*; as you just saw, the import statement references both the package that contains the class and the class itself. Just as a package organizes all the related classes, a class organizes all of its related methods, data variables or data constants, and sometimes other nested classes as well (discussed in the next section). Classes let you make your code more modular, so it's not as structured (linear) as top-down programming languages. A Java class can be used to organize your Java code at the next logical level of functional organization, and therefore your classes contain Java code constructs that add specific functionality. These include methods, variables, constants, and nested classes, all of which are covered in this chapter.

Java classes can also be used to create Java hard objects, which are discussed after you learn about classes, nested classes, methods, and data fields. Java `Objects` are constructed using a Java class and have the same name as the Java class and the exact same name as the class's constructor method, which you also see later in this chapter.

You can preface the declaration with Java modifier keywords that declare the class as being public or private or with other designators regarding what the class can and will do and for whom. Java modifier keywords are always placed before the Java `class` keyword, using the following format:

```
<modifier keywords> class <your custom classname goes here>
```

One of the powerful features of Java classes is that they can be used to modularize Java code. Your core application features can be a part of a high-level class, which can be subclassed to create more specialized versions of that class. This is a core feature of OOP. Once a Java class has been used to create a subclass, it becomes the *superclass*. A class always subclasses another superclass using the Java `extends` keyword.

As you can see in Figure 4-2, you declare a class using a Java `public` access modifier, then the Java `class` keyword, and a name for the class (in this case, `InvinciBagel`), which then uses the `extends` keyword to become a JavaFX `Application`. The import statement for `Application` is shown in Figure 4-1, under the `package` statement, naming the project (custom) package `invincibagel`. Notice that I am using NetBeans 8, which can be used for both Java and HTML5 development, supporting JSON development across all environments.

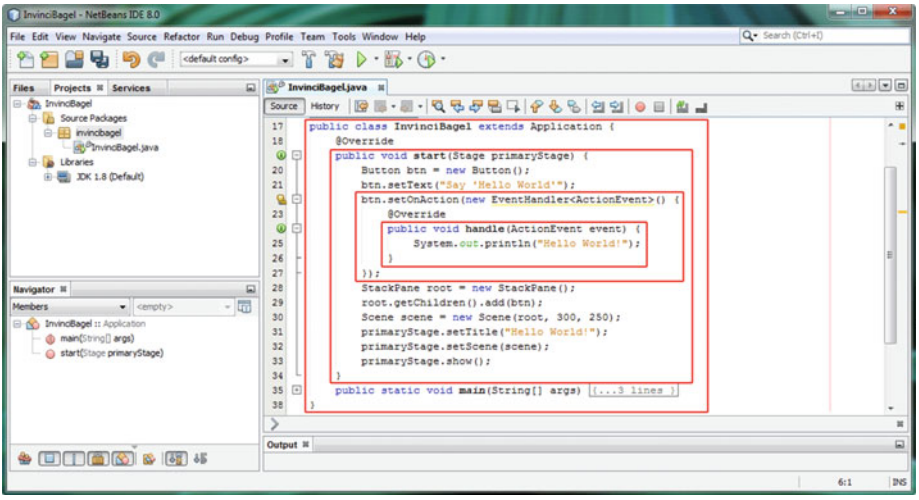

Figure 4-2. *Public class* `InvinciBagel` *extends* `Application`*, creating a JavaFX game application*

Using the Java `extends` keyword tells a Java compiler that you want the superclass's capabilities and functionality added (extended) to your class, which, once it uses `extends`, becomes a *subclass*. A subclass extends the core functionality that is provided by the superclass it is extending. To extend your class definition to include a superclass, you add to (or extend, no pun intended) your existing class declaration using the following format:

```
<modifier keywords> class <your custom classname> extends <superclass>
```

When you extend a superclass using your class, which becomes the subclass of that superclass, you can use all of that superclass's features (nested classes, inner classes, methods, constructors, variables, and constants) in your subclass. You can use this code without having to explicitly rewrite (recode) these Java constructs in the body of your class, which would be redundant (and disorganized), because your class extends this superclass, making it a part of itself.

The *body* of a class is coded in curly braces, shown in the outer red box in Figure 4-2, which follow your class declaration. In Figure 4-2, the `InvinciBagel` class extends the Application superclass from the JavaFX application package. Doing this gives `InvinciBagel` everything it needs to host, or run, the JavaFX application. The JavaFX `Application` class constructs this `Application` object so that it can use system memory and call an `.init()` method to initialize anything that may need initializing, and also call a `.start()` method, which you can see in Figure 4-2 in the second red box. A `.start()` method is where you put Java code statements that will ultimately be needed to fire up (start) any JavaFX application. When an end user finishes using your InvinciBagel Java application, the `Application` (object) created by this `Application` class, using the `.Application()` constructor method, will call its `.stop()` method and remove your application from system memory. This will free up system memory space for other applications used by your end users.

The chapter explores Java methods, constructors, and objects next, as you progress from the higher-level package and class constructs, to lower-level constructs.

Java Methods: Code Constructs Providing Core Logic Functions

In Java classes, you generally have *methods* and the *data fields* (variables or constants) that the methods use for data they operate on. Because you are going from outer structures to inner structures, or top-level structures to lower-level structures, this chapter covers methods next. Methods are sometimes called functions in other programming languages, such as JavaScript.

You can see an example of the `.start()` method in Figure 4-2; this method holds the programming logic that creates the basic Hello World application. The programming logic in this `.start()` method uses Java programming statements to create a stage and a scene, place a button on the screen in the `StackPane`, and define event-handling logic so that when the button is clicked, the bootstrap Java code writes the "Hello World" text to the NetBeans IDE output area.

Declaring a Method: Modifier, Return Type, and Method Name

A method declaration starts with an *access-control modifier* keyword: `public`, `protected`, `private`, or package private. Package private is designated by not using any access-control modifier keyword. In Figure 4-2, the `.start()` method is declared using the `public` access-control modifier.

After the access-control modifier, you declare the method's *return type*. This is the *type of data* that the method will return after it is called (invoked). Because this `.start()` method performs setup operations but doesn't return any specific type of value, it uses the `void` return type, which signifies that the method performs tasks but does not return any data to the calling entity. In this case, the calling entity is the JavaFX `Application` class, because the `.start()` method is one of the key methods; the others are the `.stop()` and `.init()` methods provided by the `Application` superclass that the `InvinciBagel` class extends. This class controls the *application lifecycle stages* for this JavaFX application.

After the return type, you supply your method's name, which, by convention (or programming rules) should start with a *lowercase letter* (or word, preferably a verb). Any subsequent (internal) words (nouns or adjectives) start with a capital letter.

For instance, the method to display the `SplashScreen` should logically be named `.displaySplashScreen()`. Because it does something but does not return a value, it is `void` and is therefore declared using this empty Java code structure:

```
public void displaySplashScreen() { Java code to display splashscreen here }
```

You may need to pass *parameters*, which are named data values that need to be passed in and operated on in the *body* of the method, which is the part in curly braces. These parameters go in the parentheses attached to the end of the method name. In Figure 4-2, the `.start()` method receives the `Stage` object named `primaryStage` as its parameter, using the following Java method declaration programming syntax:

```
public void start(Stage primaryStage) { code to start your Application }
```

You can provide as many parameters as you like, using data type /parameter name pairs, with each pair separated by a comma. Methods are not required to have any parameters. If a method has no parameters, the parentheses are empty (right next to each other). This is how method names are written in this book, so you know they are methods: dot notation before and parentheses characters after the method name, like this: `.start()`, `.stop()`, and so on.

The programming logic that defines this method is contained in the body of the method, which as you have already learned is in curly braces that define the beginning and the end of the method. The programming logic in a method includes variable declarations, programming logic statements, and iterative control structures (loops), among other things, all of which you use to create JavaFX applications.

Constructor Methods: Turning a Java Class into a Java Object

This section covers a specialized type of Java method: a *constructor method*. This can be used to create, or *construct*, Java `Objects`, as you see later in the chapter. A constructor method could be considered a hard object in contrast to a JSON soft object, because Java requires a constructor method to create a Java `Object`. `Objects` in Java are simply called *objects*; the hard versus soft distinction is in JavaScript, which has both types of object declarations, as you see soon.

Objects are the foundation of OOP, so it is important for you to have an understanding of constructor methods before you learn about the Java `Object` itself. Because this section covers methods, this is the most logical place to look at object constructors (as constructor methods are sometimes called by veteran Java developers).

Creating a Java Object: Invoking the Class Constructor Method

A Java class contains a constructor method that must use the exact same name as the class itself. This method can be used to create Java `Objects` using the class. constructor methods use the Java class that contains them as the blueprint to create an instance of that class in system memory, which creates the Java `Object`. This constructor method always returns a Java `Object` type and thus does not use any of the Java return types that other methods typically use (`void`, `String`, `float`, `int`, `byte`, and so on). A constructor method should always be invoked using the Java `new` keyword, because you are creating a new Java `Object`! If you do not create a constructor method, the Java compiler will auto-create one for each class.

You can see an example of this in the bootstrap JavaFX code in Figure 4-2, in lines 20, 28, and 30. The new `Button`, `StackPane`, and `Scene` objects are created, respectively, using the following object-declaration, object-naming, and object-creation Java code structure:

```
<class name> <object instance name> = new <constructor method name> ;
```

A Java `Object` is declared in this fashion, using the class name, the name of the object you are constructing, the Java `new` keyword, and that class's constructor method name (and parameters, if any) in a single Java statement terminated with a semicolon character because each Java `Object` is an *instance* of a Java class. For example, on line 20 of the Java

code in Figure 4-2, the portion to the left of the equals operator tells the Java language compiler that you want to create the `Button` object named `btn` using the JavaFX `Button` class as that object's blueprint. This declares the `Button` class (object type) and gives it a unique name.

The first part of creating the object is thus called the *object declaration*. The second part of creating a Java `Object` is called the *object instantiation*; it takes place on the right side of the equals operator and involves the object (class) constructor method along with the Java `new` keyword.

To instantiate the Java `Object`, you invoke, or use, the Java `new` keyword in conjunction with the object constructor method call. Because this takes place to the right of the equals operator, the result of the object instantiation is placed into the declared object, which is on the left side of the Java statement.

This completes the process of declaring (class name), naming (object name), creating (using the `new` keyword), configuring (using the constructor method), and loading (using the equals operator) your very own custom Java `Object`.

It's important to note that the declaration and instantiation parts of this process can be coded using *separate lines of Java code* as well. For instance, the `Button` object instantiation (Figure 4-2, line 20) could be coded as follows:

```
Button btn;          // Declare a Button object named btn
btn = new Button();  // Instantiate btn object using the Java new keyword
```

This is significant because coding object creation this way allows you to declare the object at the top of your class, where each method in the class that uses or accesses the objects can see it. In Java, objects or data fields are only visible in the Java programming construct (class or method) that they are declared in. So if methods are nested in a class, you should declare anything you want the methods to be able to access (see) at the top of the class construct and before the method constructs.

If you declare an object in a class, and therefore outside all the methods contained in the class, the methods in the class can then access (see or use) that object. Similarly, anything declared in a method is local to that method and is only visible to other members of that method, meaning all Java statements in that method's scope (what is within the `{...}` delimiters). In the current example, if you wanted to implement this separate object declaration in the class, outside the methods and object instantiation in the `.start()` method, the first few lines of the `InvinciBagel` class would look like the following:

```
public class InvinciBagel extends Application {
     Button btn;  // Declared outside of your start() method construct
     @Override
     public void start(Stage primaryStage) {
        btn = new Button();  // Instantiated in your start() method
        btn.setText("Say 'Hello World'");  // Object can now be utilized
        // other programming statements could continue in here
     }
}
```

When the object declaration and instantiation are split up this way, the `Button` object can be used or accessed by methods in the class other than `.start()`. In the previous code, other methods of the `InvinciBagel` class could call `.btn.setText()` without the Java compiler throwing any errors. The way the `Button` object is declared in Figure 4-2, only the `.start()` method can see the object, so only `.start()` can implement the method call; thus the `btn Button` object belongs solely to `.start()`, using the single-statement declare and instantiate approach.

Java Objects: Virtual Reality Using OOP with Java

Objects are the foundation of OOP languages—in this use case, Java. Everything in Java is based on the Java `Object` superclass. I like to call this the *master class*. The `Object` class is in the `java.lang` package, so an import statement for it references `java.lang.Object` (the full pathname to the Java `Object` class). All other Java classes are created (subclassed) using this class, because everything in Java is ultimately an `Object`.

Java `Objects` can be used to virtualize reality by allowing objects you see in everyday life. In Java applications, you can create objects using your imagination, to be realistically simulated. You do so by using data fields (variables and constants) and methods. These Java programming constructs make up your object's *characteristics* or attributes (constants), *states* (variables), and *behaviors* (methods). Java class constructs organize each object definition (constants, variables, and methods) to give birth to an instance of that object, using the constructor method for the class, which designs and defines the object construction.

Designing a Java Object: Constants, Variables, and Methods

One way to think about Java `Objects` is that like they are nouns; things (objects) that exist in and of themselves. The object behaviour is created using methods like verbs: things the nouns can do. As an example, let's consider that very popular object in everyone's life: a car. Let's define the `Car` object's attributes to see how an object can be defined using Java. Some characteristics, or attributes that do not change, held in *constants*, might be defined as follows:

- `Paint Color (Candy Apple Red, Metallic Blue, Silver, White, Black)`
- `Engine Fuel (gas, diesel, biodiesel, hydrogen, propane, electric)`
- `Drive Train (2 Wheel Drive or 4 Wheel Drive)`

Some *states*, which change, and which define the car in real time, held in *variables*, could be defined as follows. They hold the current values for direction, speed, and what gear you are in:

- `Direction (N, S, E, W)`
- `Speed (15 miles per hour)`
- `Current Gear (1, 2, 3, 4, 5, Reverse, Park)`

The following are some things the `Car` object should be able do—its *behaviors*, or how it functions. In JavaScript, these are called functions; in Java, however, they are called methods. Java `Object` behaviors might be defined using these methods:

- `Accelerate!`
- `Shift Gears`
- `Apply Brake`
- `Turn the wheels`
- `Turn on the stereo`
- `Use the headlights`
- `Use the turn signals`

You get the idea. Figure 4-3 shows a simple diagram of this Java `Object` structure. It includes the characteristics, or attributes, of the car that are central to defining the `Car` object, and the behaviors that can be used with the `Car` object. These attributes and behaviors define the car to the outside world.

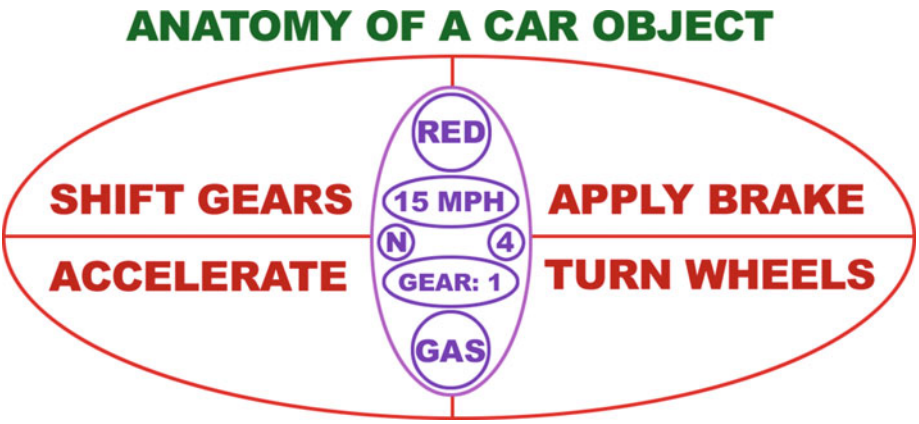

Figure 4-3. *The anatomy of a `Car` object, with methods encapsulating variables or constants*

Objects can be as complicated as you want them to be. Java `Objects` can also nest or contain other Java `Objects` in the object structure (object hierarchy). An object hierarchy is like a tree structure, with a main trunk, branches, and sub-branches as you move up (or down) the tree, very similar to JavaFX or 3D software scene graph hierarchies.

A perfect example of a hierarchy that you use every day is the multilevel directory (file folder) structure on your computer's hard disk drive (see Figure 1-1). Directories or folders on a hard drive can contain other directories or folders, which can in turn contain yet other directories and folders. This allows complex organizational hierarchies to be created—and an object is similar in its hierarchical organizational capabilities.

You'll notice that, in real life, objects can be made up of other objects. For example, a car engine object is made up of hundreds of discrete objects contained in subcomponents (like the carburetor) that all function together to make the engine object work as a whole.

This same construction of more complicated objects out of simpler objects should be mirrored using OOP languages: complex hierarchies of objects can contain other objects so the structure is well organized and logically defined. Many of these objects are created using preexisting, or previously developed, Java or JavaScript code. This is one of the objectives of object-oriented, modular programming practices.

As a good exercise, you should practice identifying different complex objects in the room around you and then break their definition, or description, down into variable states and constant characteristics, as well as behaviors, or things that these objects can do. Practice creating hypothetical object and sub-object hierarchies. This is a great exercise, because this is how you eventually need to start thinking in order to become more successful in your professional OOP endeavors using Java, JavaScript, or the JavaFX engine in the Java 8 programming language framework.

Encoding Objects: Turning an Object Design into Java Code

To illustrate how to define an object in Java, let's construct a basic class for the `Car` example. To create a `Car` class, you use the Java keyword `class`, followed by the custom name for the new class you are coding, and then curly brackets, which contain a Java class definition. The first things that you usually put in the class (in the curly {} brackets) are the data fields (variables). The variables hold the states, or characteristics, of this `Car` object. In this case, six data fields define the car's current gear, current speed, current direction, fuel type, color, and drive-train (two-wheel drive or four-wheel drive), as specified earlier in Figure 4-3.

With all six variables in place, the `Car` class (object blueprint) definition initially looks something just like this:

```
class Car {
    int speed = 15;
    int gear = 1;
    int drivetrain = 4;
    String direction = "N";
    String color = "Red";
    String fuel = "Gas";
}
```

Remember that because you're specifying the starting values using the equals sign for all the variables, these object properties (variables) all contain these default data values. These initial data values are set in the system memory as the `Car` object's default values at construction.

The next part of the Java class definition file contains the object methods. Java methods should define how your `Car` object functions—that is, how it operates on the variables you defined at the beginning of the class. Remember, these hold the `Car` object's current state of operation. Calling these methods invokes the variable state change; methods can also return data values to the entity that calls or invokes the method. Return values may include data values that have been successfully changed, or the result of an equation.

For instance, there should be a method to allow users to shift gears by setting the `Car` object's `gear` variable or attribute to a different value. This method should be declared `void`, because it performs a function but does not return any data values. In the `Car` class and `Car` object definition example, you have four methods, as defined in Figure 4-3.

The `.shiftGears()` method sets the `Car` object's `gear` attribute to the `newGear` value passed in to the `.shiftGears()` method. You should allow an integer to be passed in to this method to allow for user errors, just as in the real world a user might accidentally shift from first into fourth gear:

```
void shiftGears (int newGear) {
     gear = newGear;
}
```

The `.accelerateSpeed()` method takes your object's `speed` state variable and adds an acceleration factor to it. It then sets the result of this addition operation back into the original `speed` variable, so that the object's speed state now contains the new (accelerated) speed value:

```
void accelerateSpeed (int acceleration) {
     speed = speed + acceleration;
}
```

The `.applyBrake()` method takes the object's `speed` state variable and subtracts `brakingFactor` from it. It then sets the result of this subtraction back into the original `speed` variable, so the object's speed state now contains the updated (decelerated) braking value:

```
void applyBrake (int brakingFactor) {
     speed = speed - brakingFactor;
}
```

The `.turnWheels()` method is straightforward, much like the `.shiftGears()` method, except that it uses a `String` value of `N`, `S`, `E`, or `W` to control the direction in which the car turns. When `.turnWheels("W")` is used, the `Car` object turns to the left; and when `.turnWheels("E")` is used, the car turns to the right—given, of course, that the `Car` object is currently heading north, which, according to its default direction setting, it is:

```
void turnWheels (String newDirection) {
     direction = newDirection;
}
```

The methods that make a `Car` object function go in the class after the variable declarations, as follows:

```
class Car {
    int speed = 15;
    int gear = 1;
    int drivetrain = 4;
```

```
    String direction = "N";
    String color = "Red";
    String fuel = "Gas";

    void shiftGears (int newGear) {
        gear = newGear;
    }

    void accelerateSpeed (int acceleration) {
        speed = speed + acceleration;
    }

    void applyBrake (int brakingFactor) {
        speed = speed - brakingFactor;
    }

    void turnWheels (String newDirection) {
        direction = newDirection;
    }
}
```

Next, let's take a look at how to add the `.Car()` constructor method into this class.

Constructing Objects: Coding Your Constructor Method

This `Car` class lets you define a `Car` object even if you don't specifically include a `.Car()` constructor method, discussed in this section. This is why a collection of variable settings becomes the `Car` object's defaults. It is best to code your own constructor method, however, so that you take total control over your object creation and don't have to preinitialize the variables to one value or another. The first thing to do is make your variable declarations undefined, removing the equals sign and initial data values, as shown in this modified `Car` class:

```
class Car {
    String name;
    int speed;
    int gear;
    int drivetrain;
    String direction;
    String color;
    String fuel;

    public Car (String carName) {
        name = carName;
        speed = 15;
        gear = 1;
        drivetrain = 4;
```

```
        direction = "N";
        color = "Red";
        fuel = "Gas";
    }
}
```

The `.Car()` constructor method sets the default data values as a part of the construction and configuration of a `Car` object. As you can see, you add a `String` variable to hold the `Car` object's name, with a default `name` parameter set to the text data value `carName`.

Java constructor methods differ from regular Java methods in a number of distinct ways. First, they do not use any data return type, such as `void` or `int`, because they are used to create Java `Objects` rather than to perform functions. They do not return nothing (`void` keyword) or a number (`int` or `float` keyword), but rather return an object of type `java.lang.object`. Note that every class that needs to create an object features a constructor with the same name as the class itself, so a constructor is one method type whose name can and should always start with a capital letter. If you do not code a constructor, your Java compiler will create one for you.

Another difference between constructor methods and standard Java methods is that constructors need to use a `public`, `private`, or `protected` access-control modifier and cannot use any non-access-control modifiers. Therefore, be sure not to declare your constructor as static, final, abstract, or synchronized.

Creating Objects: Object Instantiation Using the new Keyword

The syntax for constructing an instance of a Java `Object` is similar to declaring a variable. It also uses the Java `new` keyword and the constructor method, using this format:

```
Car myCarObject = new Car();
```

To access the `Car` object's properties (variables) or characteristics (constants), you can use *dot notation*, which is used to chain, or reference, Java constructs to each other. For strict Java programming, you would follow the OOP principle of *encapsulation* and use getter `.getProperty()` and setter `.setProperty()` methods that must be called to access the object property in a more controlled fashion. I am covering the similarities to JavaScript here: dot notation can be used in both Java and JavaScript to access object properties directly without any encapsulation enforced.

Once a Java `Car` object has been declared, named, and instantiated, you can then reference its properties. This is done, for example, using the following Java `Object` format:

```
objectName.propertyName;
```

So to access the `Car` object name, you use the following Java code construct:

```
myCarObject.name
```

To invoke your Car object methods using this myCarObject Car object also requires the use of dot notation. For example, you can use the following Java construct:

```
objectName.methodName(parameter list variable);
```

So, to shift a Car object into third gear, if this Car object instance is named myCarObject, you use the following Java programming statement:

```
myCarObject.shiftGears(3);
```

This calls or invokes the .shiftGears() method of the myCarObject Car object and passes the gear parameter, which contains the integer value of 3. This is placed into the newGear variable, which is used by the .shiftGears() method's internal code to change the gear attribute of the myCarObject Car object instance to third gear. If you think about it, how Java works is very logical, and pretty darned cool as well.

Extending an Object Structure: The OOP Concept of Inheritance

There is also support in Java for developing different types of enhanced classes, and therefore enhanced (more complex or detailed) objects. This is done using a technique called *inheritance*. Inheritance lets you create more specialized classes that contain uniquely defined objects using your original (foundational) object. For instance, the Car class can be subclassed using the original Car superclass. Once you subclass any class, it becomes a superclass. Ultimately, there can be only one superclass, at the very top of the class chain, but there can be an unlimited number of subclasses. All of these subclasses inherit the methods and data fields from their superclass. The ultimate example of this in Java is the java.lang.Object superclass (I sometimes call this the master class), which is used to create all other classes in Java. Every class in Java, because it has a constructor, is also an object! Mind-boggling, to say the least—but once you wrap your head around it, OOP is both logical and powerful.

As an example of inheritance using the Car class, you can subclass an SUV class, using the Car class as the superclass. This is done by using the Java extends keyword, which extends a Car class definition in order to create an SUV class definition. The SUV class then defines *only those additional attributes* (constants), *states* (variables), and *behaviors* (methods) that apply to the SUV type of Car object. The SUV object additionally extends all the attributes (constants), states (variables), and behaviors (methods) that apply to all types of Car objects as defined by the Car superclass.

This is the functionality that the Java extends keyword provides for this subclassing (or inheritance) operation, and this is one of the more important and useful features of *code modularization* in OOP languages. You can see the modularization visually in Figure 4-4, which adds Car features for each of the subclasses (at the top, in orange).

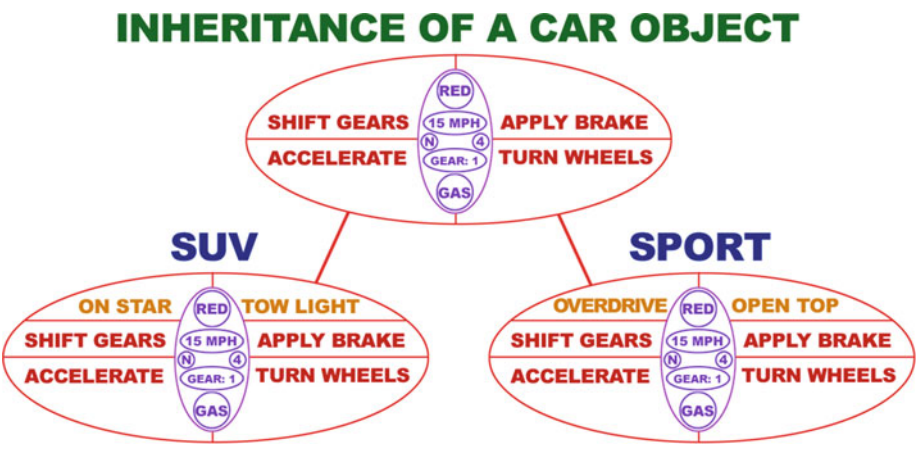

Figure 4-4. *OOP inheritance of the* `Car` *object allows you to create an SUV object or* `Sport` *object*

As an example, the SUV `Car` object subclass can have additional `.onStarCall()` and `.turnTowLightOn()` methods defined in addition to inheriting the usual operational methods that allow a `Car` object to shift gears, accelerate, apply the brakes, and turn the wheels.

Similarly, you can generate a second subclass called the `Sport` class, which creates `Sport Car` objects. These may, for example, include an `.activateOverdrive()` method to provide faster gearing, and maybe an `.openTop()` method to put down the convertible roof. To create the subclass using the superclass, you extend the subclass from the superclass by using a Java `extends` keyword in your class declaration. The Java subclass construct thus looks like the following Java SUV class construct, which uses the Java `super` keyword to generate a new `applyBrake()` method programming structure that makes the brakes twice as effective:

```
class SUV extends Car {
    void applyBrake (int brakingFactor) {
        super.applyBrake(brakingFactor);
        speed = speed - brakingFactor;
    }
}
```

This extends the SUV object to have access to (essentially, to contain) all the data fields and methods that the `Car` object (class) features. You can focus on just the new, or different, data fields and methods, which relate to differentiating the SUV object from the regular or master `Car` object superclass definition.

To refer to one of your superclass's methods from within the subclass you are coding, you can use the Java `super` keyword. For example, in the new SUV class, you may want to use the `Car` superclass's `.applyBrake()` method and then apply some additional functionality to the brake that is specific to the SUV. You call the `Car` object's `.applyBrake()` method by using `super.applyBrake(brakingFactor);` in the Java code for the SUV `Car` object.

The Java code shown previously adds functionality to the Car object's .applyBrake() method, in the SUV object's .applyBrake() method, by using the super keyword to access the Car object's .applyBrake() method; it then adds logic to make brakingFactor apply twice. This serves to give the SUV object twice the braking power of a standard car, which an SUV needs in order to stop its far greater mass.

Next, let's take a closer look at JavaScript's OOP approach. JavaScript is even more closely aligned with JSON than the Java programming languages are.

JavaScript OOP Concepts: Hard and Soft Objects

JavaScript can create objects in a couple of different ways. There are hard (constructed) objects and or soft (literal notation) objects, which is what you define using JSON. This section looks at the difference between them and how you access JavaScript objects using each of these two object-encoding approaches. This provides a parallel—at least, using the hard-object approach—to what you looked at in the previous Java Object sections. (I capitalize Object in Java usage because it is a proper class name, from the java.lang.Object master class.) You will find, possibly due to the popularity of Java, that JavaScript allows you to do things in a very similar fashion to Java, using *constructor functions*. Let's look at the hard-object encoding approach first and then look at the soft-object approach, which is more compatible with JSON structures. Finally, you see the JavaScript Object Notation (JSON) description format to finish this chapter.

JavaScript Hard Objects: Using a Constructor Function

Just as you saw with Java Objects and constructor methods, you can also create a constructor function in JavaScript to create an object. This section doesn't look at this in much detail, because you already saw this approach using the Java 9 OOP language, and because the JSON model is tailor-made for using the JavaScript soft-object approach (discussed next). A Car object constructor function in JavaScript looks like the following code, if you follow the object diagram shown in Figure 4-3 and the definition created in the previous section:

```
function carObject() {
     this.name = 'carName';
     this.speed = 15;
     this.gear = 1;
     this.drivetrain = 4;
     this.direction = 'North';
     this.color = 'Red';
     this.fuel = 'Gas';
     this.ApplyBrake = function() { Brake Application Code in Here };
     this.shiftGears = function() { Gear Shifting Code in Here };
     this.TurnWheels = function() { Wheel Turning Code in Here };
     this.accelerate = function() { Acceleration Code in Here };
};
```

JavaScript literal notation defines an object as a variable, using the `var` keyword, instead of as a constructor function. Let's take a look at that slightly different approach next.

JavaScript Soft Objects: Using Literal Notation to Define a Variable

The second way to define an object in JavaScript is as a *variable* using *literal notation*. A `Car` object variable declaration in JavaScript using literal notation looks like the following data construct, if you again follow the `Car` object diagram in Figure 4-3:

```
var carObject = {
     name : 'myCarsName',
     speed : 15,
     gear : 1,
     drivetrain : 4,
     direction : 'North',
     color : 'Red',
     fuel : 'Gas',
     ApplyBrake : function() { Brake Application Code in Here },
     shiftGears : function() { Gear Shifting Code in Here },
     TurnWheels : function() { Wheel Turning Code in Here },
     accelerate : function() { Acceleration Code in Here }
};
```

Let's look at the differences between these two object-definition approaches.

Differences Between a Constructor Function and Literal Notation

As you can see, the primary difference in the declaration of the constructor function is that it uses the `this` keyword. The `this` keyword is used in both Java and JavaScript and allows an object to reference itself. The literal notation does not use the `this` keyword. Secondarily, a constructor function uses the equals operator (`=`) to assign values to the object's properties and functions, whereas the literal notation object definition uses the colon assignment operator (`:`).

A constructor function can use (optional) *semicolons* (`;`) at the end of each assignment statement for both properties and functions. On the other hand, an object defined as a variable via literal notation is supposed to use *commas* (`,`) after each assignment statement if there is more than one, which there is in the majority of JSON applications.

As you can see, other than using different keywords (JavaScript uses function, and Java uses method), the OOP languages JavaScript and Java define object construction logic in a similar fashion. These also access object properties and functions in a similar format, using dot notation. Let's take a look at this next.

Accessing JavaScript Objects: Using Dot Notation

To access an object property, you use the object name, then a dot (period), and then the property name, like this: `ObjectName.PropertyName`. If you are using a constructor function (or method in Java), you first instantiate the object using the OOP `new` keyword, like this:

```
var myCarObject = new carObject;     (First Construct Your Object Instance)
myCarObject.name                     (this will return a String: myCarName)
myCarObject.accelerate()   (executes code inside the accelerate() function)
```

If you create an object using literal notation, you do not have to construct the object first, so you can access the JavaScript object without having to first construct an instance of it. The following code example accesses a property and a function:

```
carObject.name         (this will return the String myCarName)
carObject.accelerate() (executes the code inside the accelerate() function)
```

Now you're ready to see the differences when you define objects using JSON.

Defining Soft Objects: Using JavaScript Object Notation (JSON)

As you can see from the `json.org` object definition diagram reproduced in Figure 4-5, the JSON object definition uses the soft-object literal notation approach but does not support the function definitions. The JSON object data definition is contained in curly braces ({}) and uses a colon (`:`) to separate the key (object attribute name) from the value (object attribute data value) and a comma (`,`) to separate key-value pairs (object attributes and values) from each other.

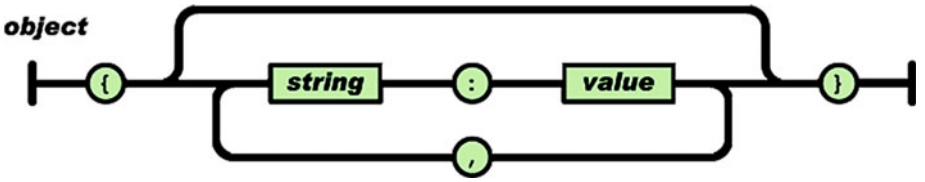

Figure 4-5. *The JSON object-definition diagram (from the `json.org` web site)*

This is what the `Car` object's attributes (data definition) look like using JSON:

```
{ "name":       "carName",
  "speed":      15,
  "gear":       1,
  "drivetrain": 4,
  "direction":  "North",
  "color":      "Red",
  "fuel":       "Gas"    }
```

There are ways to get JavaScript functions into a JSON object by using `String` data and parsing that into functions as part of your JSON parsing and evaluation process. If you really need this functionality (no pun intended), this limitation can be worked around. JSON is primarily designed to construct data object definitions that contain object parameters and their corresponding data values, so functions are generally externalized with JSON application development design approaches.

Summary

This chapter looked at two of the most popular object OOP languages in the world today: Oracle Java and ECMAScript-262 JavaScript. You saw how OOP languages allow you to logically stratify, define, and modularize your apps, and looked at how objects are constructed and referenced using Java, JavaScript, and JSON. In the next chapter, you get into arrays of data and how these are handled in JSON.