CHAPTER 1

What Is JSON?

Before we look at JSON from a low-level point of view, let's take a look at JSON from about 6,000 feet. From the mountain summit, we can see JSON flitting about in the world, carrying data in its lightweight format. If we look through our binoculars at JSON, we will see data among many curly bracket characters (`{}`). However, if we step back, and watch *how* it's being used, we will ultimately see that it is a data interchange format.

JSON Is a Data Interchange Format

A data interchange format is a text format used to exchange data between platforms. Another data interchange format you may already have heard of is XML. The world needs data interchange formats, like XML and JSON, to exchange data between very different systems.

Imagine for a moment a world comprised of hundreds of tiny, isolated islands among a vast ocean. Each island has its own unique language and customs. The islands all have seafaring merchants that travel long distances between the islands. Outside trade is an integral part of all the island economies and contributes to a high standard of living for the islanders. If it weren't for the highly trained carrier seagulls, this would not be possible.

The carrier seagulls move from island to island, carrying a paper report of data on which goods are in the highest demand. This way, merchants find out where they should move to next, and which goods they should acquire before embarking on their long voyages across the oceans. This important data allows all the islands to prosper without the threat of shortages.

Keep in mind, each island speaks a different language. If the data were passed around in several languages, each island would need to invest in researchers to learn all the

world's languages and employ a team of translators. This would be expensive and time consuming. This is an intelligent world, however, so the islands all agreed on a single language with a standard format for communicating their trade data. Each island employs just a single translator that understands the one data format of the trade reports brought by the carrier seagulls.

The real world of technology is much like the imagined island world example. There is a vast ocean, full of islands that have different languages, customs, and architecture. The ability for these unique systems to communicate is integral to many businesses and organizations. If each of these systems needed a translator for all the many ways other systems structure their data, then communications would consume an unreasonable amount of time and resources. Instead, the systems agree on a single format for data and employ a single translator.

JSON is a data interchange format that many systems have agreed on using for communicating data. You may hear it referred to as a "data exchange format," or simply a "data format." In this book, I will refer to JSON as a data interchange format because the definition of "interchange" reminds us that the data format is intended for two or more entities exchanging it.

Many, but not all systems have agreed on JSON for communicating data. There are data interchange formats, such as Extensible Markup Language (XML), that were around before JSON was even thought about. The real world is not quite as simple as the island example. Many systems have and still use other formats, such as XML, or more tabular, delimited formats such as comma-separated values (CSV). The decision by each island in the real world for which data format to accept for communication often has to do with how the data format relates to the customs, language, and architecture of the island.

In the island world example, each of the hundreds of islands had its own language. The data in the paper report that the carrier seagulls carried was in an agreed upon format that was independent of language. This way, a single translator of the trade reports data could be employed by each island. The same is true of JSON, except the data is carried across networks in zeros and ones instead of by seagulls. The translator isn't a human, it is a parser employed by the system consuming the data so it can be read within the system it is entering.

JSON Is Programming Language Independent

JSON stands for JavaScript Object Notation. The name of this data interchange format may mislead people into thinking they will need to learn JavaScript to understand and use JSON. There would be some value in learning JavaScript before learning JSON, as it was born out of a subset of JavaScript, but if you will not be using JavaScript anytime soon it would be unnecessary. You may remain dedicated to the

language or languages of your own island, for the spirit of a data interchange format is to be independent of language.

JSON is based on JavaScript object literals. A detailed explanation into the "how" of this is better suited for our discussion on syntax (Chapter 2) and data types (Chapter 3). For this chapter, the "why" is important. If a data interchange format is meant to be language independent, then it may seem contradictory to have a data format that is not only derived from a single language, but advertises it in its name: JavaScript Object Notation. Why, then?

If we return to the island example, imagine for a moment what the meeting to select the data format would have been like. When the representatives from each of the hundreds of islands arrived at this meeting, and looked to create a single data format, the first thing they would want to find is common ground.

The languages of each island may have been unique, but there were things the islanders found they had in common. Most of the languages were spoken primarily with the human voice and included a written form of the language represented by characters of some sort. Additionally, facial expressions and hand movements were also present. There were a few troublesome islands where the people communicated by other means, such as hitting sticks together or winking, but the majority of the islands found common ground with their written and spoken forms of language.

In the real world, there are hundreds of programming languages. Some are more popular and commonly used than others, but the language landscape is diverse. When college students major in computer science in preparation for a career in programming, they do not study all the programming languages. Students usually begin with one language, and the language itself is not so important as learning the universally accepted programming concepts. Once students gain an understanding of these concepts, they can more easily learn other programming languages through their ability to recognize the common features and functionalities.

If we set aside the word "JavaScript" from the name "JavaScript Object Notation," we would be left with "Object Notation." In fact, let's forget JavaScript all together. We could then say we are using an object notation data interchange format. "Object" is a common programming concept, in particular to object-oriented programming (OOP). Most computer science students studying programming will learn the concept of objects.

Without diving into an explanation of objects, let's settle our attention on the word "Notation." *Notation* implies a system of characters for representing data such as numbers or elements. With or without an understanding of objects in programming, it is not a stretch to see the value of having a notation to describe something that is common across programming languages.

Returning again to the island example, the islanders themselves found a notation that represented a common tie among the majority of languages. Most of the islanders had a similar way of representing numbers with tallies, and it was agreed they could understand a series of symbols for representing real-world objects such as wheat or fabric. Even the island that communicated by winking found this format acceptable.

Despite the agreement among the vast majority of islands, there were still a few islands, such as the island that communicated by hitting sticks together, that did not find the format understandable. A good data interchange format covers the majority, but there are usually outliers. When we talk about this coverage, a term often thrown around is *portability*. Portability, or the compatibility in transferring information between platforms and systems, is the very goal of a data interchange format.

Circling back to notation, the notation of JSON may originate from JavaScript, but the notation itself is the important part. Not only is JSON language independent, it represents data in a way that speaks to common elements of many programming languages. With the way that data is represented, such as numbers and words, even the programming languages that aren't object-oriented can find this format acceptable.

Key Terms and Concepts

This chapter covered the following key terms:

JSON
: JavaScript Object Notation

Notation
: A system of characters for representing data such as numbers or elements

Data interchange format
: Text used to exchange data between platforms or systems

Portability
: Transferring information between platforms in a way that is compatible with both systems

We also discussed these key concepts:

- JSON is a data interchange format
- JSON is programming language independent (JavaScript is not required to use it)
- JSON is based on the object literal notation of JavaScript (emphasis on the word "notation")
- JSON represents data in a way that is friendly to universal programming concepts

CHAPTER 2

JSON Syntax

JSON Is Based on JavaScript Object Literals

In the English language, the word "literal" is an adjective used to imply that what is being said is exact, not a metaphor. When your friend says "She showed up out of nowhere and I literally dropped my sandwich," he is stating that the dropping of the sandwich is not a metaphor.

In programming, the word "literal" is a noun. A *literal* is a value that is represented literally with data. It is written precisely as it is meant to be interpreted. If you aren't familiar with programming concepts, then this might seem strange. Let's take a quick look at literals.

Do you carry cash in your wallet, or a debit card? When I stop off at the sandwich shop and hand the cashier a five dollar bill for my sandwich, I physically watch my five dollars leave my wallet. When I swipe my debit card to pay for a sandwich, I know I have five dollars less in my back account, even though I didn't see it happen.

In programming, we often use variables to represent values. For example, I might use a *variable* I call *x* in an expression like:

```
x = 5
```

Then, later on, I might want to add five more to *x*:

```
x = x + 5
```

At this point, we know the value of *x* is 10, but we don't *see* 10. In this example, *x* was the variable, and 5 was a literal. In our sandwich shop example, we could say that the five dollars cash was a literal, and the debit card was a variable. When we *see* the actual value, it is a literal value.

In the "$x = 5$" example, 5 is a number literal. A number is a data type. Other types of data are strings (made up of characters), boolean (true or false), null (nothing), collections of values, and objects. Representing a number value in a way that we can see is simple, and we use a number character. Representing a boolean value is also simple and we can use true/false or 0/1. If you are familiar with the concept of objects, you will understand that representing an object is no easy or simple matter. If you aren't familiar with the concept of objects, that is OK too.

In programming, the concept of an object is similar to how you would describe a real-world object, such as your shoes. You could describe your shoes with attributes or properties such as color, style, brand, and the type of insole. Some of these attribute values could be a number, such as shoe size, and others could be a boolean (true/false) such as "has laces." Example 2-1shows an example.

Example 2-1. Using JSON to describe the shoes I'm wearing right now

```
{
    "brand": "Crocs",
    "color": "pink",
    "size": 9,
    "hasLaces": false
}
```

Don't be concerned just yet about the syntax in the shoe example; we will arrive there later in this chapter. The main point of the shoe example is that you (even a human) can literally read the attributes of my shoe. The data type for my JSON shoe example is object. We can see the literal value of the object which exposes the properties or attributes in a way which we can see (and read). These attributes or properties of the shoe object are represented as name-value pairs.

JSON is based on JavaScript object literals. The key phrase here is "based on." In JavaScript (and most programming languages with objects), the object can include a function. So not only could I represent the properties of my shoe with the JavaScript object, but I could create a function called "walk."

However, data interchange is about data, so JSON does not include the functions of JavaScript object literals. The way that JSON is based on JavaScript object literals is purely in the syntactic representation of the object literal and its properties. This representation of properties is achieved with name-value pairs.

Name-Value Pairs

The concept of name-value pairs is widespread in computing. They are called by other names as well: key-value pairs, attribute-value pairs, and field-value pairs. In this book, we will refer to them as name-value pairs.

If you are familiar with the concept of name-value pairs, JSON will seem natural to you. If you aren't familiar with name-value pairs, that's OK too. Let's take a quick look at name-value pairs.

In a name-value pair, you first declare the name. For example, "animal." Now, pair implies two things: a name and a value. So let's give our name, "animal," a value. To simplify this concept for this chapter, let's use a string value. With name-value pairs in JSON, the value can also be a number, a boolean, null, an array, or an object. We will go more in depth with the other value data types beyond string in Chapter 3. So, for our name-value pair, with the name "animal," we will use the string value, "cat":

```
"animal" : "cat"
```

"animal" is the name and "cat" is the value. There are many ways that we could choose to delimit, or separate, the name and the value. If I were to provide you with a directory of a company's employees with their job titles, I'd probably hand you a list that looks something like this:

- Bob Barker, Chief Executive Officer
- Janet Jackson, Chief Operations Officer
- Mr. Ed, Chief Financial Officer

For my employee directory, I used commas to separate my job titles (names) and employee names (values). I also placed the value on the left and the name on the right.

JSON uses the colon character (:) to separate the names and values. The name is always on the left and the value is always on the right. Let's take a look at a few more:

```
"animal" : "horse"

"animal" : "dog"
```

Simple, right? A name, and a value, and you have a name-value pair.

Proper JSON Syntax

Now let's take a look at proper JSON syntax. The name, which in our example is "animal," is always surrounded in double quotes. The name in the double quotes can be any valid string. So, you could have a name that looks like this, and it would be perfectly valid JSON:

```
"My animal": "cat"
```

You can even place an apostrophe in the name:

```
"Lindsay's animal": "cat"
```

Now that you know that this is valid JSON, I'm going to tell you why you shouldn't do this. The name-value pairs used in JSON are a friendly data structure to many systems. Having a space or special character (other than a–z, 0–9) in the name, would not be taking *portability* into consideration. In Chapter 1, we defined this key term as "transferring information between platforms in a way that is compatible with both systems." We can do things in our JSON data that decrease portability; therefore, we say it is important to avoid spaces or special characters for *maximum portability*.

The name in the name-value pair of your JSON, if it is to be loaded in memory by a system as an object, will become a "property" or "attribute." A property or attribute in some systems can include an underscore character (_) or numbers, but in most cases it is considered good form to stick to the characters of the alphabet, A–Z or a–z. So, if I wanted to include multiple words in my name, I would format like so:

```
"lindsaysAnimal": "cat"
```

or

```
"myAnimal": "cat"
```

The "cat" value in the example has double quotes. Unlike the name in the name-value pair, the value does not always have double quotes. If our value is a string data type, we must have double quotes. In JSON, the remaining data types are number, boolean, array, object, and null. These will not be surrounded in double quotes. The format of these will be covered in Chapter 3.

JSON stands for JavaScript Object Notation. So, the only thing we are missing is the syntax that makes it an object. We need curly brackets surrounding our name value pair to make it an object. So, one before...

```
{ "animal" : "cat" }
```

...and one after. When you are formatting your JSON, picture a knighting ceremony where the master of the ceremony dubs the new knight on the shoulders with a sword. You are the master of the ceremony, and you must dub your JSON as an object on each side with a curly bracket. "I dub thee, sir JSON." The ceremony would not be complete without a tap on each shoulder.

In JSON, multiple name-value pairs are separated by a comma. So, to extend the animal/cat example, let's add a color:

```
{ "animal" : "cat", "color" : "orange" }
```

Another way to look at JSON syntax would be through the eyes of the machine that is reading it. Unlike humans, machines are very rigidly rule- and instruction-oriented creatures. When you use any of the following characters outside of a string value (not surrounded in quotes), you are providing an instruction on how your data is to be read:

- { (left curly bracket) says “begin object”
- } (right curly bracket) says “end object”
- [(left square bracket) says “begin array”
-] (right square bracket) says “end array”
- : (colon) says “separating a name and a value in a name-value pair”
- , (comma) says “separating a name-value pair in an object” or “separating a value in an array”; can also be read as “here comes another one”

If you forget to say “end object” with a right curly bracket, then your object will not be recognized as an object. If you place a comma at the end of your list of name-value pairs, you are giving the instruction “here comes another one” and then not providing it. Therefore, it is important to be correct in your syntax.

A Story: The Double Quotes of JSON

One day I was peering over a student’s shoulder, looking at his computer screen. This guy was showing me some JSON that he was about to validate.

Example 2-2. The “JSON” that would not validate

```
{
    title : "This is my title.",
    body : "This is the body."
}
```

Upon validation he received a parsing error and became frustrated. He said “Look, there’s nothing wrong with it!”

I pointed out to him that he was missing quotes around “title” and “body.” He said “But I’ve seen JSON formatted both ways, with and without quotes around the names.” “Ah,” I said. “When you saw it without quotes around the names, that was not JSON. It was a JavaScript object.”

This confusion is understandable. JSON is based on JavaScript object literals, so it looks much the same. A JavaScript object literal does not need quotes around the name of the name-value pair. In JSON, it is absolutely required.

Another point of confusion can be the usage of single quotes instead of double quotes. In JavaScript, an object may have single quotes for syntax instead of double quotes.

Example 2-3. This is not valid JSON

```
{
    'title': 'This is my title.',
    'body': 'This is the body.'
}
```

In JSON, only double quotes are used, and they are absolutely required around the name of the name-value pair.

Example 2-4. Valid JSON

```
{
    "title": "This is my title.",
    "body": "This is the body."
}
```

Syntax Validation

Unlike machines, as a human using a keyboard, creating an error is as simple as a missed keystroke. It's amazing, really, that we don't produce more errors than we do. Validating JSON is an important part of working with JSON.

Your integrated development environment (IDE) might have built-in validation for your JSON. If your IDE supports plug-ins and add-ons, you might find a validation tool there if it is not already integrated. If you don't use an IDE, or have no idea what I'm talking about, that's OK too.

There are many online tools for formatting and validating JSON. A quick jaunt on your search engine for "JSON validation" will give you several results. Here are a few worth mentioning:

JSON Formatter & Validator
: A formatting tool with options, and a beautiful UI that highlights errors. The processed JSON displays in a window that doubles as a tree/node style visualization tool and a window to copy/paste your formatted code from.

JSON Editor Online
: An all-in-one validation, formatting, and visualization tool for JSON. An error indicator is displayed on the line of the error. Upon validation, helpful parsing error information is displayed. The visualization tool displays your JSON in a tree/node format.

JSONLint
: A no-bells-and-whistles validation tool for JSON. Simply copy, paste, and click "validate." It also kindly formats your JSON.

These are tools for *syntax validation*. Later, in Chapter 4, we'll discuss another type of validation called conformity validation. Syntax validation concerns the form of JSON itself, whereas conformity validation concerns a unique data structure. For Example 2-5, syntax validation would be concerned that our JSON is correct (surrounded in curly brackets, dividing our name-value pairs with commas). Conformity validation would be concerned that our data included a name, breed, and age. Addi-

tionally the conformity validation would be concerned that the value of age is a number, and the value of name is a string.

Example 2-5. Validation example

```
{
    "name": "Fluffy",
    "breed": "Siamese",
    "age": 2
}
```

JSON as a Document

You might find that in your future experiences with JSON, you are only ever creating it in code and passing it around in an unseen world that can only be inspected by developer tools. However, as a data interchange format, JSON can be its own document and live in a filesystem. The file extension for JSON is easy to remember: *.json*.

So, if I were to save my animal/cat JSON to a file and store it on my computer, it would look something like this: *C:\animals.json*.

The JSON MediaType

Oftentimes when you are passing data to someone else, you need to tell them ahead of time what type it is. You may hear this called an Internet media type, a content type, or a MIME type. This type is formatted as "type/subtype." One type that you may have already heard of is "text/html."

The MIME type for JSON is application/json.

The Internet Assigned Numbers Authority (IANA) maintains a comprehensive list of media types.

Key Terms and Concepts

This chapter covered the following key terms:

Literal
: A value that is written precisely as it is meant to be interpreted

Variable
: A value that can be changed and is represented by an identifier, such as *x*

Maximum portability (in data interchange)
: Transcending the base portability of the data format by ensuring the data itself will be compatible across systems or platforms

Name-Value Pair
: A name-value pair (or key-value pair) is a property or attribute with a name, and a corresponding value

Syntax Validation
: Validation concerned with the form of JSON

Conformity Validation
: Validation concerned with the unique data structure

We also discussed these key concepts:

- JSON is based on the syntactic representation of the properties of JavaScript object literals. This *does not* include the functions of JavaScript object literals.
- In the JSON name-value pair, the name is always surrounded by double quotes.
- In the JSON name-value pair, the value can be a string, number, boolean, null, object, or array.
- The list of name-value pairs in JSON is surrounded by curly brackets.
- In JSON, multiple name-value pairs are separated by a comma.
- JSON files use the *.json* extension.
- The JSON media type is application/json.

CHAPTER 3

JSON Data Types

If you've already learned a programming language or two, you likely have an understanding of data types. If not, that's OK too. Let's take a quick look.

Quick Look at Data Types

Imagine what would happen if you hand a little boy that knows nothing about tools a hammer, and you don't tell him what it's for. Property and bodily damage would likely occur. If this child is well behaved and coordinated, we could give this child a set of instructions for using the hammer. Instead of running around damaging things, the child would only ever use it for hammering nails and removing them (It's a well-behaved child, remember). Additionally, when you say to the child "Will you pass me the hammer, please?", he doesn't hand you the screwdriver. Knowing what something is ahead of time and how to use it is as useful in computing as it is in the real world.

In computing, we most often need to know what type of data we are dealing with because we can do different things with different types of data. I can multiply a number by another number, but I can't multiply a word by another number. If I have a list of words, I can sort them alphabetically. I can't sort the number 5 alphabetically. So, in programming, when a method (or function) says "Will you pass me the number, please?", if we know what a number is, we won't make the mistake of passing it the word "ketchup."

In computer science, there is a set of data types referred to as primitive data types. The word "primitive" evokes imagery of Stone Age cavemen sitting around a fire grunting and sharpening sticks. It's not that these data types are unrefined like the cavemen; rather, it's that they are some of the first, most basic types of data. Like

modern man and cavemen, some of the more modern and progressive data types in existence have their roots in these primitive data types:

- Numbers (e.g., 5 or 5.09)
  - Integer
  - Floating-point number
  - Fixed-point number
- Characters and strings (e.g., "a" or "A" or "apple")
- Booleans (i.e., true or false)

In different programming languages, the types of data that are "set in stone" are often referred to as the primitive data types, or the built-in types, of that language. This means that the definition of the type and what can be done with it is unchangeable. The programming language isn't going to allow you to redefine what it means to add two numbers together. These primitive data types vary from language to language, and will often include additions to the preceding list, such as byte or a reference (or pointer, or handler).

Beyond the primitive data types, there are other data types that are used in most programming languages. These are often referred to as composite data types, because they are fusion, or compound of the primitive data types. Composite data types, like a sandcastle, have a structure to them. If we took apart the sandcastle we could see that the structure was built with sand, sticks, and water. If we took apart the data structure of a composite data type, we would find that it was built with our primitive data types.

One example of a composite data type that is commonly used in programming languages is an enumeration data type. Earlier, I mentioned the sorting of a list alphabetically. A list of words could be represented with different data types in different programming languages (e.g., a list, or an array). If we took apart this data structure, we would find that it is made up of the primitive data types of characters or strings. The enumeration data type is a data structure that you can enumerate. I can mention each thing in the structure one by one and I can also count how many there are. See Example 3-1.

Example 3-1. "Let me enumerate the fine qualities of your personality" could be represented in programming as an array literal

```
[
    "witty",
    "charming",
    "brave",
    "bold"
]
```

You don't need to understand the array literal data structure in this example to see that we can mention each of these "fine qualities" one by one, and also establish that there are four of them.

Another composite data type is the object data type. In Chapter 2, we explored the object data type, because JavaScript Object Notation is based on the object literal notation of JavaScript.

Example 3-2. In the previous chapter, I used JSON to describe the shoes I was wearing

```
{
    "brand": "Crocs",
    "color": "pink",
    "size": 9,
    "hasLaces": false
}
```

This object literal allows us to see that the object data type here is made up of name-value pairs. If we were to deconstruct this data structure, we can see that it is made up of the primitive data types: string, number, and boolean. Our names (brand, color, size, hasLaces) in our name-value pairs are all string data types. The values "Crocs" and "pink" are both of the string data type. The value "9" is of the number data type. The value "false" is of the boolean data type.

The JSON Data Types

Though programming languages may vary when it comes to composite types, and even a little when it comes to additional primitive types, most share those primitive types I mentioned earlier in the chapter:

- Numbers (e.g., 5 or 5.09)
  - Integer
  - Floating-point number
  - Fixed-point number
- Characters and strings (e.g., "a" or "A" or "apple")
- Booleans (i.e., true or false)

The object data type is a data structure that is common to some of the more popular programming languages, such as Java and C#, but not all. With JSON being based on object literal notation and the object data type, you'd think this would be problematic for a data interchange format. After all, the goal of a data format is communicating between two different systems and common ground should be expressed in that format. Remember that the data structure that is the composite data type object can be deconstructed into the primitive types. Even to programming languages that do not

have the object data type, once the object data structure is deconstructed into those native types, it is quite friendly.

The JSON data types are:

- Object
- String
- Number
- Boolean
- Null
- Array

The JSON Object Data Type

The JSON object data type is simple. JSON, at its root, is an object. It is a list of name-value pairs surrounded in curly braces. When you create a name-value pair within your JSON that is also an object, your JSON will begin to look nested. In Example 3-3, this is illustrated with a person described with nested objects.

Example 3-3. Nested objects

```
{
    "person": {
        "name": "Lindsay Bassett",
        "heightInInches": 66,
        "head": {
            "hair": {
                "color": "light blond",
                "length": "short",
                "style": "A-line"
            },
            "eyes": "green"
        }
    }
}
```

The top-level name-value pair here is person, with the value of an object. This object has three name-value pairs: name, heightInInches, and head. The "name" name-value pair has a string value of "Lindsay Bassett." The "heightInInches" name-value pair has a number value. The "head" name-value pair has an object value. The "hair" name-value pair has an object value as well, with three string data typed name-value pairs: color, length, and style. The "head" object also has an "eyes" name-value pair with a value of "green."

The JSON String Data Type

We briefly explored the JSON string data type earlier in this book with the animal/cat example:

```
{ "animal" : "cat" }
```

The value "cat" has a string data type. In the real world, unless this data is for a pet shop, the string values in the data won't be quite as simple. Even a pet shop might have more to say in its data than a single word like "cat." Perhaps the shop wants to pass along the details of its latest promotion:

> Today at Bob's Best Pets you can get a free 8 oz. sample bag of Bill's Kibble with your purchase of a puppy. Just say "Bob's the best!" at checkout.

The JSON string can be comprised of any of the Unicode characters, and all the characters in that promotional text are valid. A string value must always be surrounded in *double* quotes.

In "A Story: The Double Quotes of JSON" on page 9 in Chapter 2, I mentioned that single quotes around a string value are not valid (Example 3-4).

Example 3-4. This is not valid JSON

```
{
    'title': 'This is my title.',
    'body': 'This is the body.'
}
```

This can be confusing, especially if you've seen JavaScript object literals in the past that use single quotes. In JavaScript, we are allowed to use single quotes or double quotes interchangeably. However, it is important to remember that JSON is not a JavaScript object literal; it is only *based on* JavaScript object literals. In JSON, only double quotes are allowed for surrounding a string value.

Also mentioned in the previous chapter was how JSON is read by a parser. In the eyes of a parser, when a value begins with a double quote ("), it expects a string of text that will be ended by another double quote. This poses a problem if the string of text contains double quotes in it.

For example, suppose we're running a pet shop promotion where customers need to say "Bob's the best!" at checkout to receive a free bag of kibble. If we use the code in Example 3-5, we would run into a problem, because we can't simply surround the promotional data in double quotes.

Example 3-5. This code won't work

```
{
    "promo": "Say "Bob's the best!" at checkout for free 8oz bag of kibble."
}
```

There are quotes inside of the value, and the parser is going to read that first quote character in front of "Bob" in the promotional text as the end of the string. Then, when the parser finds the remainder of the text just hanging out there and not belonging to a name value pair, it will produce an error. To deal with this, we must escape our quote inside of any string value by preceding it with a back slash character (\), as shown in Example 3-6.

Example 3-6. Using a backslash character to escape quotes inside of strings fixes the problem

```
{
    "promo": "Say \"Bob's the best!\" at checkout for free 8oz bag of kibble."
}
```

This backslash character will tell the parser that the quote is not the end of the string. Once the parser actually loads the string into memory, any backslash character that precedes a quote character will be removed and the text will come out on the other side as intended.

Quotes are not the only thing that need escaping when it comes to the JSON string. Because the backslash character is used to escape other characters, we must also escape the backslash. For example, the JSON shown in Example 3-7, which is meant to communicate the location of my Program Files directory, will produce an error. To fix this problem, we must escape the backslash character by adding another backslash character, as shown in Example 3-8.

Example 3-7. The backslash used in this code will throw an error

```
{
    "location": "C:\Program Files"
}
```

Example 3-8. The backslash character must be escaped with another backslash character

```
{
    "location": "C:\\Program Files"
}
```

In addition to the double quote and backslash characters, you must escape the following characters:

- \/ (forward slash)
- \b (backspace)
- \f (form feed)
- \t (tab)
- \n (new line)
- \r (carriage return)
- \u followed by hexadecimal characters (e.g., the smiley emoticon \u263A)

The JSON shown in Example 3-9 will produce a parser error because the tab and new line characters must be escaped. Example 3-10 shows how to fix the problem.

Example 3-9. The tab and new line characters used in this JSON will cause an error

```
{
    "story": "\t Once upon a time, in a far away land \n there lived a princess."
}
```

Example 3-10. JSON with tab and new line characters escaped

```
{
    "story": "\\t Once upon a time, in a far away land \\n there lived a princess."
}
```

The JSON Number Data Type

Numbers are a common piece of information to pass around in data. Inventory, money, latitude/longitude, and Earth's mass are all data that can be represented as numbers; see Example 3-11.

Example 3-11. Representing numbers in JSON

```
{
    "widgetInventory": 289,
    "sadSavingsAccount": 22.59,
    "seattleLatitude": 47.606209,
    "seattleLongitude": -122.332071,
    "earthsMass": 5.97219e+24
}
```

A number in JSON can be an integer, decimal, negative number, or an exponent.

My widget inventory is 289. Inventory is typically represented by an integer (or whole number). I don't typically sell a half of something, so my inventory number will never include a decimal point.

My sad savings account contains $22.59. Though some programming languages do have a data type for money, we typically represent money in JSON as a decimal number and omit the $.

If you take a look at the latitude and longitude for the City of Seattle, you will see they are both decimal numbers, and the longitude is a negative decimal number. Negative numbers are represented with the standard minus sign character preceding the number.

Additionally, I'm representing the very large number of Earth's mass in kg using E Notation. E Notation is particularly great for scientific data and is a supported number.

The JSON Boolean Data Type

In the English language, two of the simplest answers we have for questions are "yes" or "no." If you ask your friend the question "Would you like toast with your eggs?", he will answer "yes" or "no."

In computer programming, the boolean data type is simple. It is either true or false. If you ask your computer "Would you like toast with your eggs?", it will answer "true" or "false."

In some programming languages, the literal value for true can be 1, and 0 for false. Sometimes the literal value characters use casing—for example, True or TRUE, and false or FALSE. In JSON, the literal value for the boolean data type is always all lowercase: true or false. Any other casing will produce an error. In Example 3-12, booleans are used to communicate data about my preferences for breakfast and lunch.

Example 3-12. Preferences

```
{
    "toastWithBreakfast": false,
    "breadWithLunch": true
}
```

The JSON Null Data Type

When we have nothing of something, you might think it appropriate to say there is zero of that something. Right now, I own zero watches. The thing is, zero is a number. This implies we were counting in the first place.

What if there was a standard format of describing someone's wrist in JSON, which included some attributes? See Examples 3-13 and 3-14.

Example 3-13. Next-door neighbor Bob's might look like this

```
{
    "freckleCount": 0,
    "hairy": true,
    "watchColor": "blue"
}
```

Example 3-14. Mine would look like this

```
{
    "freckleCount": 1,
    "hairy": false,
    "watchColor": null
}
```

I don't have a watch color because I'm not wearing a watch. In programming, null is a way of saying zero, zilch, and none without having to use a number. The value for watch color cannot be defined, therefore it is null.

Null should not, however, be confused with "undefined," which you might run across in JavaScript. Undefined is not a JSON data type, but in JavaScript, undefined is what you get when you try to access an object or a variable that does not exist at all. In JavaScript, undefined has a relationship with an object or variable with both its declared name and value not existing, and null has a relationship only with a value of an object or a variable. Null is a value that means "no value." In JSON, null must always be all lowercase characters.

The JSON Array Data Type

Now let's explore the array data type. If you aren't familiar with arrays, that's OK. Let's take a quick look at what an array is.

Imagine a container that holds a dozen eggs. The container has 12 available compartments for eggs. When I first bought the eggs, there were 12. This would be an array of size 12, containing 12 eggs. See Example 3-15.

Example 3-15. This is an array of strings (for the sake of simplicity, I will use the string "egg" for each of the eggs in the compartments)

```
{
    "eggCarton": [
        "egg",
        "egg",
        "egg",
        "egg",
        "egg",
        "egg",
```

```
        "egg",
        "egg",
        "egg",
        "egg",
        "egg",
        "egg"
    ]
}
```

Notice that I have a name-value pair. The name is "eggCarton" and the value is an array. The array is always surrounded in square brackets ([]). Inside the array, we have a list, and each list item is separated by a comma. This might look similar to how we format our name-value pairs, but the key difference is that it is a list of only values. These values can be any of the valid JSON data types (string, number, object, boolean, array, and null).

Now suppose I take out two eggs to make myself eggs over easy with toast for breakfast. My egg carton still has 12 compartments, but two eggs are missing. See Example 3-16.

Example 3-16. I've removed two eggs from the carton to make breakfast

```
{
    "eggCarton": [
        "egg",
        null,
        "egg",
        "egg",
        "egg",
        "egg",
        "egg",
        "egg",
        "egg",
        "egg",
        null,
        "egg"
    ]
}
```

As you can see, I removed the eggs from two specific compartments. Those compartments became empty because the eggs no longer exist. We represent this with null.

An array has an index for each of the "compartments." We begin with 0, so the first compartment on the last has an index of 0, the second has an index of 1, and so on. The last compartment will have an index of 11. So, I removed the eggs at the indices of 1 and 10. This is a valid array.

If I put the number 5 in the empty compartment at index 10, in most programming languages this would be an invalid array. See Example 3-17.

Example 3-17. This would not be valid in most programming languages

```
{
    "eggCarton": [
        "egg",
        null,
        "egg",
        "egg",
        "egg",
        "egg",
        "egg",
        "egg",
        "egg",
        "egg",
        5,
        "egg"
    ]
}
```

I said "most programming languages," but in JSON, the mixing and matching of data types is valid. I will tell you why, and then I will tell you why you shouldn't do this in your JSON.

In JavaScript, you define a variable. For instance, in Example 3-18, we have a variable named "something" and we assign it the number 5 for its value.

Example 3-18. Defining a variable in JavaScript

```
var something = 5;
```

On the very next line, we could change that variable to have a string value (Example 3-19).

Example 3-19. Changing that variable to have a string value

```
something = "bob";
```

And we could even change it to have a value of an object (Example 3-20).

Example 3-20. Changing that variable to have a value of an object

```
something = { person: "bob" };
```

My "var something" (variable) can be a number, string, array, null, or object. In most programming languages, variables aren't allowed to be so shifty. Normally you would declare something, as either an int, a string, or an object. So, when you declared your variable called "something," you will say int something = 5. You would say string something = "bob" or you would say Person something = new Person("bob"). So in

most programming languages, when you declare an array, you are declaring ahead of time what data type has to be in each of your containers, and you can't just change it around after the fact.

JSON is a data interchange format. If you hand your JSON array over to someone that is not going to be using your JSON with JavaScript, your array will cause an error when it is being parsed.

For example, suppose you attend a convention where merchants are selling collections of rocks. You have a collection of rocks to sell. A guy shows up and wants to buy your collection of 50 rocks, and he takes them, but when he gets home he finds the rock collection does not contain 50 rocks, but it contains, 20 rocks, 20 sticks, and 10 pieces of gum (one of which has already been chewed).

Let's take a closer look at some examples of arrays of each data type. In JSON, the array can be any of the supported data types. So, we can have an array of strings, an array of numbers, an array of booleans, an array of objects, or an array of arrays. An array of arrays is called a multidimensional array. Let's take a look at some examples.

Suppose we have a roster with the names of students who signed up for a course. This can be represented using an array of strings (Example 3-21).

Example 3-21. Using an array of strings to represent a roster of students

```
{
    "students": [
        "Jane Thomas",
        "Bob Roberts",
        "Robert Bobert",
        "Thomas Janerson"
    ]
}
```

After the students have taken a test, we can use an array of numbers to represent their scores (Example 3-22).

Example 3-22. Using an array of numbers to represent test scores

```
{
    "scores": [
        93.5,
        66.7,
        87.6,
        92
    ]
}
```

If we need to create an answer key for a true/false test, we could use an array of booleans (Example 3-23).

Example 3-23. Using an array of booleans to represent the answers of a true/false test

```
{
    "answers": [
        true,
        false,
        false,
        true,
        false,
        true,
        true
    ]
}
```

An array of objects could be used to represent the entire test, including questions and answers (Example 3-24).

Example 3-24. Using an array of objects to represent the questions and answers of a test

```
{
    "test": [
        {
            "question": "The sky is blue",
            "answer": true
        },
        {
            "question": "The earth is flat.",
            "answer": false
        },
        {
            "question": "A cat is a dog.",
            "answer": false
        }
    ]
}
```

To represent the scores from three different tests, an array of arrays, or a multidimensional array, could be used (Example 3-25).

Example 3-25. Using an array of arrays to represent the scores from three different tests

```
{
    "tests": [
        [
            true,
            false,
```

```
            false,
            false
        ],
        [
            true,
            true,
            true,
            true,
            false
        ],
        [
            true,
            false,
            true
        ]
    ]
}
```

Key Terms and Concepts

This chapter covered the following key terms:

JSON string data type
: A string value, like “my string” surrounded in double quotes

JSON boolean data type
: A true or false value

JSON number data type
: A number value, like 42, that can be a positive or negative integer, decimal, or exponent

JSON null data type
: A null value represents an empty value

JSON array data type
: An array is a collection or list of values, and the values can have a data type of string, number, boolean, object or array; the values in an array are surrounded by square brackets ([]) and delimited by a comma

JSON object data type
: The object data type is a set of name-value pairs delimited by a comma and surrounded by curly brackets ({})

We also discussed these key concepts:

- The boolean data type value of true or false in JSON is always all lowercase characters (i.e., true, not True or TRUE).

- The null data type value in JSON is always all lowercase characters (i.e., null, not NULL or Null).
- One key difference between an object and an array is that an object is a list or collection of name-value pairs and an array is a list or collection of values.
- Another key difference between an array and an object is that an array's values *should* all have the same data type.