Quick answers to common problems

JavaScript JSON Cookbook

Over 80 recipes to make the most of JSON in your desktop, server, web, and mobile applications

Ray Rischpater
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 "Reading and Writing JSON on the Client"
- A synopsis of the book’s content
- More information on JavaScript JSON Cookbook
About the Author

Ray Rischpater is an engineer and author with over 20 years of experience in writing about and developing for mobile computing platforms.

During this time, he participated in the development of Internet technologies and custom applications for Java ME, Qualcomm BREW, Apple iPhone, Google Android, Palm OS, Newton, and Magic Cap, as well as several proprietary platforms. Currently, he's employed as a software development manager at Microsoft in Sunnyvale, where he works on mapping and data visualization.

When he is not writing about software development, he enjoys hiking and photography with his family and friends in and around the San Lorenzo Valley in Central California. When he can, he also provides public service through amateur radio as the licensed Amateur Extra station KF6GPE.


Ray holds a bachelor's degree in pure mathematics from the University of California, Santa Cruz, and is a member of the IEEE, ACM, and ARRL.
Preface

JavaScript Object Notation (JSON) has rapidly become the lingua franca for structured document exchange on the Web, outpacing XML in many domains. Three of the reasons for this are obvious: it plays well with JavaScript, it's simple, and it just works. However, there are other reasons for its success as well. As you'll see in the pages of this book, it's supported by a wide variety of languages and libraries, making it easy to use in all kinds of scenarios.

In this book, I provide recipes for common uses of JSON. You're welcome to read the book from cover to cover, seeing all the ways JSON can be used in building web and standalone applications. However, it's organized as a cookbook, so that you can quickly go to the chapter or recipe that addresses a particular problem you might want to solve with JSON now. I recommend skimming this preface to see what's where, taking a quick look at Chapter 1, Reading and Writing JSON on the Client, or Chapter 2, Reading and Writing JSON on the Server, depending on your interest, and then jumping right to the recipes that interest you the most.

What this book covers

Chapter 1, Reading and Writing JSON on the Client, gives you recipes to read and write JSON in a number of client environments, including JavaScript, C#, C++, Java, Perl, and Python.

Chapter 2, Reading and Writing JSON on the Server, goes the other way, looking at JSON on typical server-side languages such as Clojure, C#, Node.js, PHP, and Ruby. Of course, you can write client-side applications with these languages, as well, just as you can write a server in C# or Java. So the division of recipes between these chapters is somewhat arbitrary; pick a language and dive in!

Chapter 3, Using JSON in Simple AJAX Applications, shows you how to apply JSON for data exchange with today's browsers.

Chapter 4, Using JSON in AJAX Applications with jQuery and AngularJS, discusses how to use JSON with two popular web frameworks, jQuery and Angular.
Chapter 5, *Using JSON with MongoDB*, shows you how MongoDB, a popular NoSQL database, uses JSON for its stored document format and gives you recipes to use MongoDB as a REST service in your web applications.

Chapter 6, *Using JSON with CouchDB*, shows you how CouchDB, another popular NoSQL database, uses JSON and how you can use CouchDB as a standalone REST service in your web applications.

Chapter 7, *Using JSON in a Type-safe Manner*, looks at how you can adapt the type-free nature of JSON with the type safety provided by languages such as C#, Java, and TypeScript to reduce programming errors in your application.

Chapter 8, *Using JSON for Binary Data Transfer*, shows you how, even though JSON is a text-based document format, you can still use it to move binary data around if you have to do so.

Chapter 9, *Querying JSON with JSONPath and LINQ*, has recipes on how you can write queries against JSON documents to obtain just the slice of data you're looking for. This is especially powerful when combined with the recipes from Chapters 5, *Using JSON with MongoDB*, and Chapter 6, *Using JSON with CouchDB*.

Chapter 10, *JSON on Mobile Platforms*, shows you recipes for using JSON in mobile applications that use Android, iOS, and Qt.
In this chapter, we will cover the following recipes:

- Reading and writing JSON in JavaScript
- Reading and writing JSON in C++
- Reading and writing JSON in C#
- Reading and writing JSON in Java
- Reading and writing JSON in Perl
- Reading and writing JSON in Python

In addition to reading and writing JSON in Python, we will begin by showing you a brief review of JSON formatting to help set the stage for what follows in this book.

**Introduction**

**JSON** stands for JavaScript Object Notation. It's an open standard to represent data as attributes with values. Originally derived from the JavaScript syntax (hence its name) for use in web applications as an alternative to the more verbose and structured Extensible Markup Language (XML), it is now used for data serialization and transport in many standalone and web applications.

JSON provides an ideal means to encapsulate data between the client and server. In this first chapter, you will learn how to work with JSON in languages specified at the beginning of this chapter.
These languages are often used for client-side development, which is what we will focus on here. We'll look more at server-side languages in Chapter 2, Reading and Writing JSON on the Server.

Let's take a look at some JSON returned by the web API, available at http://www.aprs.fi, and modified a bit by me to make the example clear (later, in Chapter 4, Using JSON in AJAX Applications with jQuery and AngularJS, you'll learn how to fetch this data yourself using a web browser and JavaScript):

```json
{
    "command": "get",
    "result": "ok",
    "what": "loc",
    "found": 2,
    "entries": [
    {
        "class": "a",
        "name": "KF6GPE",
        "type": "l",
        "time": "1399371514",
        "lasttime": "1418597513",
        "lat": 37.17667,
        "lng": -122.14650,
        "symbol": "/-",
        "srccall": "KF6GPE",
    },
    {
        "class": "a",
        "name": "KF6GPE-7",
        "type": "l",
        "time": "1418591475",
        "lasttime": "1418591475",
        "lat": 37.17633,
        "lng": -122.14583,
        "symbol": "\\K",
        "srccall": "KF6GPE-7",
    }
    ]
}
```

**Downloading the example code**

You can download the example code files from your account at http://www.packtpub.com for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.
There are a few things to notice about this example:

- The data is organized into attributes and values, each separated by a colon. (Note that a JSON document can also be a single value, such as a string, float, integer, or Boolean value.)
- Attributes appear as character strings enclosed by double quotes on the left-hand side of a colon.
- Values are on the right side of the colon and can be the following:
  - Character strings (enclosed in double quotes) such as `KF6GPE`
  - Numbers (either integers or floating point) such as `2` or `37.17667`
  - Arrays (comma-delimited values contained in square brackets), such as the value for `entries`
  - Whole objects consisting of more attributes and values, such as the two-array values in the `entries` value
  - Alternatively (although this example doesn't show it), the Boolean values `true` and `false`
- Note that many other kinds of values, such as date/time pairs or individual characters are not supported by JSON.
- Although it's not entirely clear from this example, whitespace is insignificant. There's no need to have each pair on its own line, for example, and the indentation is completely arbitrary.

The attribute-name-attribute-value property of JSON, along with the ability to nest values and represent arrays, gives JSON a lot of flexibility. You can represent a lot of common objects using JSON, including most objects that don't have a lot of binary data (For ideas on how to represent binary data using JavaScript and JSON, see Chapter 8, Using JSON for Binary Data Transfer). This includes primitive values (self-documenting because each value is accompanied by an attribute), flat objects with simple values including maps, and arrays of simple or complex objects.

The self-documenting nature of JSON makes it an ideal choice for data transport as you develop new objects, despite its lack of support for comments as you might find in XML. Its plaintext nature makes it amenable to compression over the wire using popular compression schemes such as `gzip` (available inside most web servers and web clients), and its format is easier for humans to read than the more verbose XML.

Note that JSON documents are inherently trees, and thus, do not have support for cyclical data structures, such as graphs, where a node points to another node in the same data structure.

If you create such a data structure using the native representation in the programming language you're using and try to convert that to JSON, you'll get an error.
Reading and writing JSON in JavaScript

JSON originated as a means to carry data between web servers and JavaScript, so let's begin with a simple code snippet that reads and writes JSON in JavaScript in a web browser. We'll show the entirety of a web application using AJAX and JSON in Chapter 4, *Using JSON in AJAX Applications with jQuery and AngularJS*; what follows is how to obtain a JavaScript object from JSON and how to create a JSON string from a JavaScript object.

Getting ready

You'll need a way to edit the JavaScript and run it in your browser. In this example, and nearly all examples in this book, we'll use Google Chrome for this. You can download Google Chrome at https://www.google.com/chrome/browser. Once you install Google Chrome, you'll want to activate the JavaScript console by clicking on the Customize and control Doodle Chrome icon on the right-hand side, which looks like this:

![Browser window with JavaScript console](image1)

Then, you'll want to go to More Tools | JavaScript console. You should see a JavaScript console on the side of the web page, like this:

![JavaScript console](image2)

---

Reading and Writing JSON on the Client
If you prefer key commands, you can also use Ctrl + Shift + J on Windows and Linux, or control + option + J on a Macintosh.

From here, you can enter JavaScript on the lower right-hand corner and press Enter (return on a Mac OS X system) to evaluate the JavaScript.

### How to do it...

Modern web browsers, such as Chrome, define a JSON object in the JavaScript runtime that can convert the string data containing JSON to JavaScript objects, and convert a JavaScript object to JSON. Here's a simple example:

```javascript
> var json = '{"call":"KF6GPE","type":1,"time":1399371514,"lasttime":1418597513,"lat":37.17667,"lng":-
-122.14650,"result": "ok" }';
<- '{ "call": "KF6GPE", "type": 1, "time": "1399371514", "lasttime": "1418597513", "lat": 37.17667, "lng": -122.14650, "result": "ok" }'
> var object = JSON.parse(json);
<- Object {call:"KF6GPE",type:1,time:1399371514,
lasttime:1418597513,lat:37.17667, lng:-122.14650,result: "ok"}
> object.result
<- "ok"
> var newJson = JSON.stringify(object);
<- '{ "call": "KF6GPE", "type": 1, "time": "1399371514", "lasttime": "1418597513", "lat": 37.17667, "lng": -122.14650, "result": "ok" }'
```

In this and subsequent JavaScript examples, the text you type in the JavaScript console is preceded by a > symbol, while what the JavaScript console prints is anything beginning with <- symbol.

### How it works...

Chrome and other modern web browsers define the JSON object, which has methods to convert between strings containing JSON and JavaScript objects.

In the previous example, we begin by setting the value of the json variable to a simple JSON expression consisting of one attribute result with the value ok. The JavaScript interpreter returns the resulting value of the variable json.
Reading and Writing JSON on the Client

The next line uses the JSON method parse to convert the JSON string referenced by json into a JavaScript object:

```javascript
> var object = JSON.parse(json);
<- Object { call:"KF6GPE", type:"l", time:"1399371514",
    lasttime:"1418597513", lat:37.17667, lng:-122.14650, result: "ok"}
```

You can then access any of the values in the object, just as you would any other JavaScript object; it is, after all, just an object:

```javascript
> object.result;
<- "ok"
```

Finally, if you need to convert an object to JSON, you can do that with the JSON method stringify:

```javascript
> var newJson = JSON.stringify(object);
<- "{ "call":"KF6GPE","type":"l","time":"1399371514",
    "lasttime":"1418597513","lat": 37.17667,"lng": -122.14650,
    "result" : "ok" }"
```

There's more...

You should know two things about these methods. First of all, parse will throw an exception if the JSON you pass is malformed, or isn't JSON at all:

```javascript
> JSON.parse('{"result" = "ok" }')
<- VM465:2 Uncaught SyntaxError: Unexpected token =
```

The errors aren't very helpful but better than nothing if you're debugging JSON sent by a less-than-fully compliant and debugged JSON encoder.

Second, very old web browsers may not have a JSON object with these methods. In that case, you can use the JavaScript function eval after wrapping the JSON in parenthesis, like this:

```javascript
> eval('('+json+')')
<- Object {result: "ok"}
```

The eval function evaluates the string you pass as JavaScript, and the JSON notation is really just a subset of JavaScript. However, you should avoid using eval whenever you can for a few reasons. First, it's often slower than the methods provided by the JSON object. Second, it's not safe; your string might contain malicious JavaScript that can crash or otherwise subvert your JavaScript application, which is not a threat you should take lightly. Use the JSON object whenever it's available. Third, you can use the parse and stringify methods to handle simple values, such as Booleans, numbers, and strings; you're not limited to the key-value pairs in the previous example. If all I wanted to do was pass a Boolean (such as "the transaction succeeded"), I might just write the following:

```javascript
> var booleanValue = false;
<- false
```
var jsonSuccess = 'true';
<- "true"
> var flag = JSON.parse(jsonSuccess);

Finally, it's worth pointing out that both the parse and stringify methods to JSON take an optional replacer function, which is invoked on every key and value in the object being serialized or deserialized. You can use this function to perform on-the-fly data conversions as the JSON is being parsed; for example, you can use it to convert between the string representation of a date and the number of seconds since midnight at the start of the epoch, or to correct the capitalization of strings. I could use a replacer function for either side of the transformation, as shown in the following code, to make the call field lowercase:

> var object = JSON.parse(json, function(k, v) {
  if ( k == 'call') return v.toLowerCase();
});
<- Object { call:"kf6gpe", type:"l", time:"1399371514",
  lasttime:"1418597513", lat:37.17667, lng:-122.14650, result: "ok"}

You can also return undefined to remove an item from the results; to omit the type field from the JSON I generate, I can execute the following:

> var newJson = JSON.stringify(object, function (k, v) {
  if k == 'type') return undefined;
});
<- "{"call":"KF6GPE","time":"1399371514","lasttime":
  "1418597513","lat": 37.17667,"lng": -122.14650, "result" : "ok"
 }"

Reading and writing JSON in C++

C++ is a language that long-predates JSON, but is still relevant for many projects. There's no native support for JSON in C++ but there are a number of libraries that provide support for working with JSON. Perhaps the most widely used is JsonCpp, available from GitHub at https://github.com/open-source-parsers/jsoncpp. It's licensed under the MIT license or public domain if you so desire, so there are virtually no limitations on its use.

Getting ready

To use JsonCpp, you need to first go to the website and download the zip file with the entire library. Once you do so, you need to integrate it with your application's source code.

How you integrate it with your application's source code differs from platform to platform, but the general process is this:

1. Create an amalgamated source and header for the library using the instructions on the website. To do this, you'll need to have JsonCpp downloaded and Python 2.6 or later installed. From the top level directory of JsonCpp, run python amalgamate.py.
Reading and Writing JSON on the Client

2. Include the include file `dist/json/json.h` in any file where you want to use the JsonCpp library.

3. Include the source file `dist/jsoncpp.cpp` in your project's make file or build system.

Once you do this, you should have access to the JsonCpp interface in any file that includes the `json/json.h` header.

How to do it...

Here's a simple C++ application that uses JsonCpp to convert between `std::string` containing some simple JSON and a JSON object:

```cpp
#include <string>
#include <iostream>
#include "json/json.h"

using namespace std;

int main(int argc, _TCHAR* argv[]) {
  Json::Reader reader;
  Json::Value root;

  string json = "{"call": "KF6GPE","type":"l","time": "1399371514","lasttime": "1418597513","lat": 37.17667,"lng": -122.14650,"result": "ok"}");

  bool parseSuccess = reader.parse(json, root, false);

  if (parseSuccess) {
    const Json::Value resultValue = root["result"];
    cout << "Result is " << resultValue.asString() << "\n";
  }

  Json::StyledWriter styledWriter;
  Json::FastWriter fastWriter;
  Json::Value newValue;
  newValue["result"] = "ok";

  cout << styledWriter.write(newValue) << "\n";
  cout << fastWriter.write(newValue) << "\n";

  return 0;
}
```
How it works...

This example begins by including the necessary includes, including `json/json.h`, which defines the interface to JsonCpp. We explicitly reference the `std` namespace for brevity, although don't do so for the `Json` namespace, in which JsonCpp defines all of its interfaces.

The JsonCpp implementation defines `Json::Reader` and `Json::Writer`, specifying the interfaces to JSON readers and writers, respectively. In practice, the `Json::Reader` interface is also the implementation of a JSON class that can read JSON, returning its values as `Json::Value`. The `Json::Writer` variable just defines an interface; you'll want to use a subclass of it such as `Json::FastWriter` or `Json::StyledWriter` to create JSON from `Json::Value` objects.

The previous listing begins by defining `Json::Reader` and `Json::Value`; we'll use the reader to read the JSON we define on the next line and store its value in the `Json::Value` variable `root`. (Presumably your C++ application would get its JSON from another source, such as a web service or local file.)

Parsing JSON is as simple as calling the reader's `parse` function, passing the JSON and `Json::Value` into which it will write the JSON values. It returns a Boolean, which will be `true` if the JSON parsing succeeds.

The `Json::Value` class represents the JSON object as a tree; individual values are referenced by the attribute name in the original JSON, and the values are the values of those keys, accessible through methods such as `asString`, which returns the value of the object as a native C++ type. These methods of `Json::Value` includes the following:

- `asString`, which returns `std::string`
- `asInt`, which returns `Int`
- `asUInt`, which returns `UInt`
- `asInt64`, which returns `Int64`
- `asFloat`, which returns `float`
- `asDouble`, which returns `double`
- `asBool`, which returns `bool`

In addition, the class provides `operator []`, letting you access array elements.

You can also query a `Json::Value` object to determine its type using one of these methods:

- `isNull`, which returns `true` if the value is null
- `isBool`, which returns `true` if the value is `bool`
- `isInt`, which returns `true` if the value is `Int`
- `isUInt`, which returns `true` if the value is `UInt`
isIntegral, which returns true if the value is an integer
- isDouble, which returns true if the value is double
- isNumeric, which returns true if the value is numeric
- isString, which returns true if the value is a string
- isArray, which returns true if the value is an array
- isObject, which returns true if the value is another JSON object (which you can decompose using another Json::Value value)

At any rate, our code uses asString to fetch the std::string value encoded as the result attribute, and writes it to the console.

The code then defines Json::StyledWriter and Json::FastWriter to create some pretty-printed JSON and unformatted JSON in strings, as well as a single Json::Value object to contain our new JSON. Assigning content to the JSON value is simple because it overrides the operator[] and operator[]= methods with the appropriate implementations to convert standard C++ types to JSON objects. So, the following line of code creates a single JSON attribute/value pair with the attribute set to result, and the value set to ok (although this code doesn't show it, you can create trees of JSON attribute-value pairs by assigning JSON objects to other JSON objects):

newValue["result"] = "ok";

We first useStyledWriter and then FastWriter to encode the JSON value in newValue, writing each string to the console.

Of course, you can also pass single values to JsonCpp; there's no reason why you can't execute the following code if all you wanted to do was pass a double-precision number:

Json::Reader reader;
Json::Value piValue;

string json = "3.1415";
bool parseSuccess = reader.parse(json, piValue, false);
double pi = piValue.asDouble();

See also

For the documentation for JsonCpp, you can install doxygen from http://www.stack.nl/~dimitri/doxygen/ and run it over the doc folder of the main JsonCpp distribution.

There are other JSON conversion implementations for C++, too. For a complete list, see the list at http://json.org/.
Reading and writing JSON in C#

C# is a common client-side language for rich applications as well as for writing the client implementation of web services running on ASP.NET. The .NET library includes JSON serialization and deserialization in the System.Web.Extensions assembly.

Getting ready

This example uses the built-in JSON serializer and deserializer in the System.Web.Extensions assembly, one of the many .NET libraries that are available. If you've installed a recent version of Visual Studio (see https://www.visualstudio.com/en-us/downloads/visual-studio-2015-downloads-vs.aspx), it should be available. All you need to do to use this assembly is include it in the assemblies your application references in Visual Studio by right-clicking the References item in your project, choosing Add Reference, and scrolling down to System.Web.Extensions in the Framework Assemblies list.

How to do it...

Here's a simple application that deserializes some JSON, as a dictionary of attribute-object pairs:

```csharp
using System;
using System.Collections.Generic;
namespace JSONExample
{
    public class SimpleResult
    {
        public string result;
    }
    class Program
    {
        static void Main(string[] args)
        {
            JavaScriptSerializer serializer =
            string json = @"{ "call":"KF6GPE","type":"l","time":"1399371514","lasttime":"1418597513","lat":37.17667,"lng":-122.14650,"result":"ok" }";
```
Reading and Writing JSON on the Client

dynamic result = serializer.DeserializeObject(json);
foreach (KeyValuePair<string, object> entry in result)
{
    var key = entry.Key;
    var value = entry.Value as string;
    Console.WriteLine(String.Format("{0} : {1}", key, value));
}
Console.WriteLine(serializer.Serialize(result));

var anotherResult = new SimpleResult { result="ok" };
Console.WriteLine(serializer.Serialize(anotherResult));

How it works...

The System.Web.Extensions assembly provides the JavaScriptSerializer class in the System.Web.Script.Serialization namespace. This code begins by defining a simple class, SimpleResult, which we'll encode as JSON in our example.

The Main method first defines a JavaScriptSerializer instance, and then string containing our JSON. Parsing the JSON is as easy as calling the JavaScriptSerializer instance's DeserializerObject method, which returns an object whose type is determined at run-time based on the JSON you pass.

You can also use DeserializerObject to parse JSON in a type-safe manner, and then the type of the returned object matches the type you pass to the method. I'll show you how to do this in Chapter 7, Using JSON in a Type-safe Manner.

DeserializerObject returns a Dictionary of key-value pairs; the keys are the attributes in the JSON, and the values are objects representing the values of those attributes. In our example, we simply walk the keys and values in the dictionary, printing each. Because we know the type of the value in the JSON, we can simply cast it to the appropriate type (string, in this case) using the C# as keyword; if it wasn't string, we'd receive the value null.

You can use as or the type inference of C# to determine the type of unknown objects in your JSON, making it easy to parse JSON for which you lack strict semantics.
The JavaScriptSerializer class also includes a Serialize method; you can either pass it as a dictionary of attribute-value pairs, as we do with our deserialized result, or you can pass it as an instance of a C# class. If you pass it as a class, it'll attempt to serialize the class by introspecting the class fields and values.

**There's more...**

The JSON implementation that Microsoft provides is adequate for many purposes, but not necessarily the best for your application. Other developers have implemented better ones that typically use the same interface as the Microsoft implementation. One good choice is Newtonsoft's Json.NET, which you can get at [http://json.codeplex.com/](http://json.codeplex.com/) or from NuGet in Visual Studio. It supports a wider variety of .NET platforms (including Windows Phone), LINQ queries, XPath-like queries against the JSON, and is faster than the Microsoft implementation. Using it is similar to using the Microsoft implementation: install the package from the Web or NuGet, add a reference of the assembly to your application, and then use the JsonSerializer class in the Newtonsoft.Json namespace. It defines the same SerializeObject and DeserializeObject methods that the Microsoft implementation does, making switching to this library easy. James Newton-King, the author of Json.NET, makes it available under the MIT license.

As with other languages, you can also carry primitive types through the deserialization and serialization process. For example, after evaluating the following code, the resulting dynamic variable `piResult` will contain a floating-point number, 3.14:

```csharp
    string piJson = "3.14";
    dynamic piResult = serializer.DeserializeObject(piJson);
```

**See also**

As I previously hinted, you can do this in a type-safe manner; we'll discuss more of this in Chapter 7, *Using JSON in a Type-safe Manner*. You'll do this using the generic method `DeserializeObject<>`, passing a type variable of the type you want to deserialize into.

**Reading and writing JSON in Java**

Java, like C++, predates JSON. Oracle is presently working on adding JSON support to Java, but in the meantime, several implementations providing JSON support are available on the Web. Similar to the C++ implementation you saw previously in this chapter, you can convert between JSON and Java using a third-party library; in this case, packaged as a Java archive (JAR) file, whose implementation typically represents JSON objects as a tree of named objects.

Perhaps the best Java implementation of JSON parsing is Gson, available from Google at [http://code.google.com/p/google-gson/](http://code.google.com/p/google-gson/) licensed under the Apache License 2.0.
Getting ready

First, you'll need to get Gson; you can do this by doing a read-only checkout of the repository using SVN over HTTP with SVN by using the following command:

```bash
svn checkout http://google-gson.googlecode.com/svn/trunk/google-gson-read-only
```

Of course, this assumes that you have a Java development kit (http://www.oracle.com/technetwork/java/javase/downloads/index.html) and SVN (TortoiseSVN is a good client for Windows available at http://tortoisesvn.net/downloads.html) installed. Many Java IDEs include support for SVN.

Once you check out the code, follow the instructions that come with it to build the Gson JAR file, and add the JAR file to your project.

How to do it...

To begin, you need to create a `com.google.gson.Gson` object. This class defines the interface you'll use to convert between JSON and Java:

```java
Gson gson = new com.google.gson.Gson();
String json = "{"call": "KF6GPE", "type": "l", "time": "1399371514", "lasttime": "1418597513", "lat": 37.17667,
"lng": -122.14650,"result":"ok"}";
com.google.gson.JsonObject result = gson.fromJson(json,
JsonElement.class).getAsJsonObject();
```

The `JsonObject` class defines the top-level object for containing a JSON object; you use its get and add methods to get and set attributes, like this:

```java
JsonElement result = result.get("result").getAsString();
```

The Gson library uses the `JsonElement` class to encapsulate a single JSON value; it has the following methods that let you get the value contained in `JsonElement` as a plain Java type:

- `getAsBoolean`, which returns the value as `Boolean`
- `getAsByte`, which returns the value as `byte`
- `getAsCharacter`, which returns the value as `char`
- `getAsDouble`, which returns the value as `double`
- `getAsFloat`, which returns the value as `float`
- `getAsInt`, which returns the value as `int`
- `getAsJsonArray`, which returns the value as `JsonArray`
- `getAsJsonObject`, which returns the value as `JsonObject`
getAsLong, which returns the value as long
- getAsShort, which returns the value as short
- getAsString, which returns the value as String

You can also learn about the type in JsonElement using one of the following methods:

- isJsonArray, which returns true if the element is an array of objects
- isJsonNull, which returns true if the element is null
- isJsonObject, which returns true if the element is a composite object (another JSON tree) instead of a single type
- isJsonPrimitive, which returns true if the element is a primitive type, such as a number or string

There's more...

You can also convert instances of your classes directly to JSON, writing something like this:

```java
public class SimpleResult {
    public String result;
}

// Elsewhere in your code...
Gson gson = new com.google.gson.Gson();
SimpleResult result = new SimpleResult;
result.result = "ok";
String json = gson.toJson(result);
```

This defines a class SimpleResult, which we use to create a single instance, and then use the Gson object instance to convert to a string containing the JSON using the Gson method toJson.

Finally, because JsonElement encapsulates a single value, you can also handle simple values expressed in JSON, like this:

```java
Gson gson = new com.google.gson.Gson();
String piJson = "3.14";
double result = gson.fromJson(piJson,
JsonElement.class).getAsDouble();
```

This converts the primitive value 3.14 in JSON to a Java double.
See also

Like the C# example, you can convert directly from JSON to a plain old Java object (POJO) in a type-safe manner. You'll see how to do this in Chapter 7, Using JSON in a Type-safe Manner.

There are other JSON conversion implementations for Java, too. For a complete list, see the list at http://json.org/.

Reading and writing JSON in Perl

Perl predates JSON, although there's a good implementation of JSON conversion available from CPAN, the Comprehensive Perl Archive Network.

How to do it...

To begin with, download the JSON module from CPAN and install it. Typically, you'll download the file, unpack it, and then run the following code on a system that already has Perl and make configured:

```
perl Makefile.PL
make
make install
```

Here's a simple example:

```
use JSON;
use Data::Dumper;
my $json = '{ "call":"KF6GPE", "type":1, "time":1399371514,
            "lasttime":1418597513, "lat": 37.17667, "lng": -122.14650,
            "result" : "ok" }';
my %result = decode_json($json);
print Dumper(result);
print encode_json(%result);
```

Let's look at the interface the JSON module provides.

How it works...

The CPAN module defines the decode_json and encode_json methods to decode and encode JSON respectively. These methods interconvert between Perl objects, such as literal values and associative arrays, and strings containing JSON respectively.

The code begins by importing the JSON and Data::Dumper modules. Next, it defines a single string, $json, which contains the JSON we want to parse.
With the JSON in $json, we define $result to be the associative array containing the objects defined in the JSON, and dump the values in the hash on the next line.

Finally, we re-encode the hash as JSON and print the results to the terminal.

**See also**

For more information and to download the JSON CPAN module, visit [https://metacpan.org/pod/JSON](https://metacpan.org/pod/JSON).

### Reading and writing JSON in Python

Python has had native support for JSON since Python 2.6 through the json module. Using the module is as simple as using the import statement to import the module and then accessing the encoder and decoder through the json object that it defines.

**Getting ready**

Simply enter the following in your source code to be able to reference the JSON facility:

```python
import json
```

**How to do it...**

Here’s a simple example from the Python interpreter:

```python
>>> import json
>>> json = '{ "call":"KF6GPE","type":"l","time":"1399371514",
"lasttime":"1418597513","lat": 37.17667,"lng": -122.14650,
"result": "ok" }
>>> result = json.loads(json)
>>> result['result']
'ok'
>>> print json.dumps(result)
{"call":"KF6GPE","type":"l","time":"1399371514",
"lasttime":"1418597513","lat": 37.17667,"lng": -122.14650,
"result":"ok"}
```

```python
>>> print json.dumps(result,
```
Reading and Writing JSON on the Client

{
  "call":"KF6GPE",
  "type":"l",
  "time":"1399371514",
  "lasttime":"1418597513",
  "lat": 37.17667,
  "lng": -122.14650,
  "result": "ok"
}

Let's look at `loads` and `dumps` further.

How it works...

Python has great support for associative arrays through its object hierarchy. The `json` module offers a `json` object with `loads` and `dumps` method that convert from JSON in text strings to associative arrays, and from associative arrays to JSON in text strings. If you're familiar with the Python `marshal` and `pickle` modules, the interface is similar; you use the `loads` method to obtain a Python object from its JSON representation and the `dumps` method to convert an object into its JSON equivalent.

The previous listing does just this. It defines a variable `j` that contains our JSON, and then obtains a Python object `result` using `json.loads`. Fields in the JSON are accessible as named objects in the resulting Python object. (Note that we can't call our JSON string `json` because it would shadow the definition of the interface to the module.)

To convert to JSON, we use the `json.dumps` method. By default, `dumps` creates a compact machine-readable version of JSON with minimum whitespace; this is best used for over-the-wire transmissions or for storage in a file. When you're debugging your JSON, it helps to pretty-print it with indentation and some whitespace around separators; you can do this using the optional `indent` and `separators` arguments. The `indent` argument indicates the number of spaces that each successive nested object should be indented in the string, and `separators` indicates the separators between each object and between each attribute and value.

See also

For more documentation on the `json` module, see the Python documentation at https://docs.python.org/2/library/json.html.
Where to buy this book

You can buy JavaScript JSON Cookbook from the Packt Publishing website.

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

Click here for ordering and shipping details.