CHAPTER 2

Getting Started

In this chapter, you'll learn how to create your first Google map project, plot some markers, and add a bit of interactivity. Because JavaScript plays such a central role in controlling the maps, you'll also start to pick up a few essentials about that language along the way.

In this chapter, you'll see how to do the following:

- Get off the ground with a basic map and a Google Maps API key.
- Separate the map application's JavaScript functions, data, and XHTML.
- Unload finished maps to help browsers free their memory.
- Create map markers and respond to clicks on them with an information pop-up.

The First Map

In this section, you'll obtain a Google Maps API key, and then begin experimenting with it by retrieving Google's starter map.

Keying Up

Before you start a Google Maps web application, you need sign up for a Google Maps API key. To obtain your key, you must accept the Google Maps API Terms of Use, which stipulate, among other things, that you must not steal Google's imagery, obscure the Google logo, or hold Google responsible for its software. Additionally, you're prevented from creating maps that invade privacy or facilitate illegal activities.

Google issues as many keys as you need, but separate domains *must* apply for a separate key, as each one is valid for only a specific domain and subdirectory within that domain. For your first key, you'll want to give Google the root directory of your domain or the space in which you're working. This will allow you to create your project in any subdirectory within your domain. Visit `http://www.google.com/apis/maps/signup.html` (Figure 2-1) and submit the form to get your key. Throughout this book, nearly all of the examples will require you to include this key in the JavaScript `<script>` element for the Google Maps API, as we're about to demonstrate in Listing 2-1.

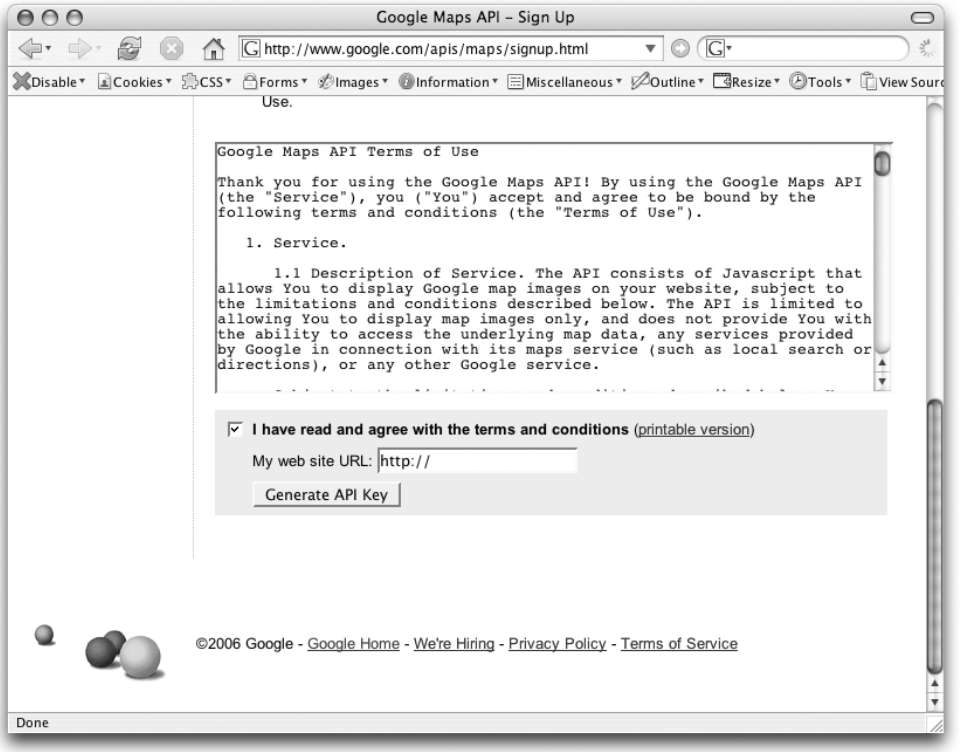

Figure 2-1. *Signing up for an API key. Check the box, and then enter the URL of your webspace.*

■ **Note** Why a key? Google has its reasons, which may or may not include seeing what projects are where, which are the most popular, and which may be violating the terms of service. Google is not the only one that makes you authenticate to use an API. Del.icio.us, Amazon, and others all provide services with APIs that require you to first obtain a key.

When you sign up to receive your key, Google will also provide you with a very basic "starter map" to help familiarize you with the fundamental concepts required to integrate a map into your website. We'll begin by dissecting and working with this starter code so you can gain a basic understanding of what's happening.

If you start off using Google's sample, your key is already embedded in the JavaScript. Alternatively, you can—as with all listings—grab the source code from the book's website at `http://googlemapsbook.com` and insert your own key by hand.

Either way, save the code to a file called `index.php`. Your key is that long string of characters following `key=`. (Our key, in the case of this book's website, is `ABQIAAAA33EjxkLYsh9SEveh_MphphQP1y➥R2bHJW2Brl_bW_lOKXsyt8cxTKO5Zz-UKoJ6IepTlZRxN8nfTRgw`).

Examining the Sample Map

Once you have the file in Listing 2-1 uploaded to your webspace, check it out in a browser. And ta-da, a map in action!

Listing 2-1. *The Google Maps API Starter Code*

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
    <head>
        <meta http-equiv="content-type" content="text/html; charset=utf-8"/>
        <title>Google Maps JavaScript API Example</title>
        <script src="http://maps.google.com/maps?file=api&amp;v=2&amp;key=ABQIAAAA➥
33EjxkLYsh9SEveh_MphphQP1yR2bHJW2Brl_bW_lOKXsyt8cxTKO5Zz-UKoJ6Ie➥
pTlZRxN8nfTRgw" type="text/javascript"></script>
        <script type="text/javascript">

        //<![CDATA[

        function load() {
            if (GBrowserIsCompatible()) {
                var map = new GMap2(document.getElementById("map"));
                map.setCenter(new GLatLng(37.4419, -122.1419), 13);
            }
        }

        //]]>
        </script>
    </head>

    <body onload="load()" onunload="GUnload()">
        <div id="map" style="width: 500px; height: 300px"></div>
    </body>
</html>
```

In Listing 2-1, the container holding the map is a standard XHTML web page. A lot of the listing here is just boilerplate—standard initialization instructions for the browser. However, there are three important elements to consider.

First, the head of the document contains a critical `script` element. Its `src` attribute points to the location of the API on Google's server, and your key is passed as a parameter:

```
<script src="http://maps.google.com/maps?file=api&amp;v=2&amp;key=YOUR_KEY_HERE"➥
type="text/javascript"></script>
```

Second, the `body` section of the document contains a `div` called `map`:

```
<div id="map" style="width: 500px; height: 300px"></div>
```

Although it appears empty, this is the element in which the map will sit. Currently, a `style` attribute gives it a fixed size; however, it could just as easily be set it to a dynamic size, such as `width: 50%`.

Finally, back in the `head`, there's a `script` element containing a short JavaScript, which is triggered by the document body's `onload` event. It's this code that communicates with Google's API and actually sets up the map.

```
function load() {
    if (GBrowserIsCompatible()) {
        var map = new GMap2(document.getElementById("map"));
        map.setCenter(new GLatLng(37.4419, -122.1419), 13);
    }
}
```

The first line is an `if` statement, which checks that the user's browser is supported by Google Maps. Following that is a statement that creates a `GMap2` object, which is one of several important objects provided by the API. The `GMap2` object is told to hook onto the map `div`, and then it gets assigned to a variable called `map`.

■ **Note** Keen readers will note that we've already encountered another of Google's special API objects: `GLatLng`. `GLatLng`, as you can probably imagine, is a pretty important class, that we're going to see a lot more of.

After you have your `GMap2` object in a `map` variable, you can use it to call any of the `GMap2` methods. The very next line, for example, calls the `setCenter()` method to center and zoom the map on Palo Alto, California. Throughout the book, we'll be introducing various methods of each of the API objects, but if you need a quick reference while developing your web applications, you can use Appendix B of this book or view the Google Maps API reference (`http://www.google.com/apis/maps/documentation/`) directly online.

Specifying a New Location

A map centered on Palo Alto is interesting, but it's not exactly groundbreaking. As a first attempt to customize this map, you're going to specify a new location for it to center on.

For this example, we've chosen the Golden Gate Bridge in San Francisco, California (Figure 2-2). It's a large landmark and is visible in the satellite imagery provided on Google Maps (`http://maps.google.com`). You can choose any starting point you like, but if you search for "Golden Gate Bridge" in Google Maps, move the view slightly, and then click Link to This Page, you'll get a URL in your location bar that looks something like this:

```
http://maps.google.com/maps?f=q&ll=37.818361,-122.478032&spn=0.029969,0.05579
```

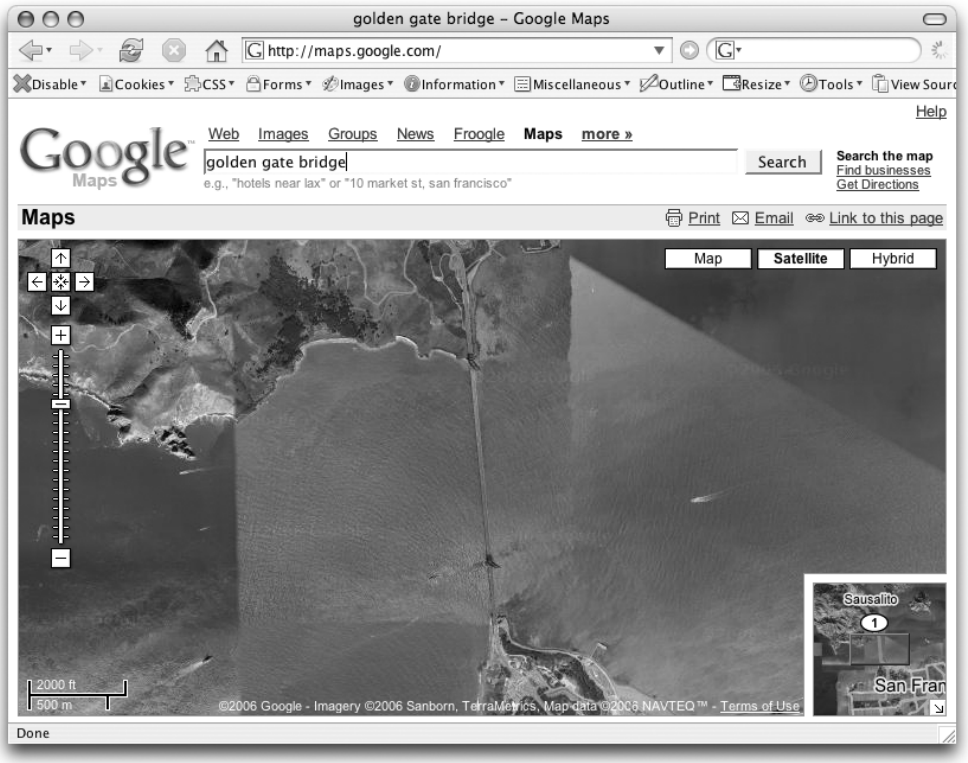

Figure 2-2. *The Golden Gate Bridge satellite imagery from Google Maps*

■ **Caution** If you use Google Maps to search for landmarks, the Link to This Page URL won't immediately contain the latitude and longitude variable but instead have a parameter containing the search terms. To also include the latitude and longitude, you need to adjust the zoom level or move the map so that the link is no longer to the default search position.

It's clear that the URL contains three parameters, separated by ampersands:

```
f = q
ll = 37.818361, -122.478032
spn = 0.029969, 0.05579
```

The `ll` parameter is the important one you'll use to center your map. Its value contains the latitude and longitude of the center of the map in question. For the Golden Gate Bridge, the coordinates are 37.82N and 122.48W.

■ **Note** *Latitude* is the number of degrees north or south of the equator, and ranges from –90 (South Pole) to 90 (North Pole). *Longitude* is the number of degrees east or west of the prime meridian at Greenwich, in England, and ranges from –180 (westward) to 180 (eastward). There are several different ways you can record latitude and longitude information. Google uses decimal notation, where a positive or negative number indicates the compass direction. The process of turning a street address into a latitude and longitude is called *geocoding*, and is covered in more detail in Chapter 4.

You can now take the latitude and longitude values from the URL and use them to recenter your own map to the new location. Fortunately, it's a simple matter of plugging the values directly into the `GLatLng` constructor.

Separating Code from Content

To further improve the cleanliness and readability of your code, you may want to consider separating the JavaScript into a different file. Just as Cascading Style Sheets (CSS) should not be mixed in with HTML, it's best practice to also keep JavaScript separated.

The advantages of this approach become clear as your project increases in size. With large and complicated Google Maps web applications, you could end up with hundreds of lines of JavaScript mixed in with your XHTML. Separating these out not only increases loading speeds, as the browser can cache the JavaScript independently of the XHTML, but their removal also helps prevent the messy and unreadable code that results from mixing XHTML with other programming languages. Your eyes and your text editor will love you if they don't have to deal with mixed XHTML and JavaScript at the same time.

In this case, you'll actually take it one step further and *also* separate the marker data file from the map functions file. This will allow you to easily convert the static data file to a dynamically generated file in later chapters, without the need to touch any of the processing JavaScript.

To accommodate these changes, we've separated the web application's JavaScript functions, data, and XHTML, putting them in separate files called `index.php` for the XHTML portion of the page, `map_functions.js` for the behavioral JavaScript code, and `map_data.php` for the data to plot on the map. Listing 2-2 shows the revised version of the `index.php` file.

Listing 2-2. *Extrapolated index.php File*

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"➥
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <script src="http://maps.google.com/maps?file=api&amp;v=2&amp;key=➥
ABQIAAAAfAb2RNhzPafOW1mtifapBRI9caN7296ZHDcvjSpGbL7PxwkwBS➥
ZidcfOwy4q2EZpjEJx3rc4Lt5Kg" type="text/javascript"></script>
    <script src="map_data.php" type="text/javascript"></script>
    <script src="map_functions.js" type="text/javascript"></script>
</head>
```

```
<body>
     <div id="map" style="width: 500px; height: 300px"></div>
</body>
</html>
```

Listing 2-2 is the same basic HTML document as before, except that now there are two extra `script` elements inside the `head`. Rather than referencing the external API, these reference local—on the server—JavaScript files called `map_data.php` and `map_functions.js`. For now, you'll leave the `map_data.php` file empty, but it will be used later in the chapter when we demonstrate how to map an existing list of markers. The important thing to note here is that it must be referenced first, before the `map_functions.js` file, so that the data is "available" to the code in the `map_functions.js` file. Listing 2-3 shows the revised `map_functions.js` file.

Listing 2-3. *Extrapolated map_functions.js File*

```
var centerLatitude = 37.818361;
var centerLongitude = -122.478032;
var startZoom = 13;

var map;

function init()
{
    if (GBrowserIsCompatible()) {
        map = new GMap2(document.getElementById("map"));
        var location = new GLatLng(centerLatitude, centerLongitude);
        map.setCenter(location, startZoom);
    }
}

window.onload = init;
```

Although the behavior is almost identical, the JavaScript code in Listing 2-3 has two important changes:

- The starting center point for latitude, longitude, and start zoom level of the map are stored in `var` variables at the top of the script, so it will be more straightforward to change the initial center point the next time. You won't need to hunt down a `setCenter()` call that's buried somewhere within the code.
- The initialization JavaScript has been moved out of the `body` of the XHTML and into the `map_functions.js` file. Rather than embedding the JavaScript in the body of the XHTML, you can attach a function to the `window.onload` event. Once the page has loaded, this function will be called and the map will be initialized.

For the rest of the examples in this chapter, the `index.php` file will remain exactly as it is in Listing 2-2, and you will need to add code only to the `map_functions.js` and `map_data.php` files to introduce the new features to your map.

■ **Caution** It's important to see the difference between `init` and `init()`. When you add the parentheses after the function name, it means "execute it." Without the parentheses, it means "give me a reference to it." When you assign a function to an event handler such as `document.onload`, you want to be very careful that you don't include the parentheses. Otherwise, all you've assigned to the handler is the function's return value, probably a `null`.

Cleaning Up

One more important thing to do with your map is to be sure to correctly unload it. The extremely dynamic nature of JavaScript's variables means that correctly reclaiming memory (called *garbage collection*) can be a tricky process. As a result, some browsers do it better than others.

Firefox and Safari both seem to struggle with this, but the worst culprit is Internet Explorer. Even up to version 6, simply *closing* a web page is not enough to free all the memory associated with its JavaScript objects. An extended period of surfing JavaScript-heavy sites such as Google Maps could slowly consume all system memory until Internet Explorer is manually closed and restarted.

Fortunately, JavaScript objects can be manually destroyed by setting them equal to `null`. The Google Maps API now has a special function that will destroy most of the API's objects, which helps keep browsers happy. The function is `GUnload()`, and to take advantage of it is a simple matter of hooking it onto the `body.onunload` event, as in Listing 2-4.

Listing 2-4. *Calling GUnload() in map_functions.js*

```
var centerLatitude = 37.818361;
var centerLongitude = -122.478032;
var startZoom = 13;

var map;

function init() {
    if (GBrowserIsCompatible()) {
        map = new GMap2(document.getElementById("map"));
        var location = new GLatLng(centerLatitude, centerLongitude);
        map.setCenter(location, startZoom);
    }
}

window.onload = init;
window.onunload = GUnload;
```

There's no obvious reward for doing this, but it's an excellent practice to follow. As your projects become more and more complex, they will eat up available memory at an increasing rate. On the day that browsers are perfect, this approach will become a hack of yesterday. But for now, it's a quiet way to improve the experience for all your visitors.

Basic Interaction

Centering the map is all well and good, but what else can you do to make this map more exciting? You can add some user interaction.

Using Map Control Widgets

The Google Maps API provides five standard controls that you can easily add to any map:

- `GLargeMapControl`, the large pan and zoom control, which is used on `maps.google.com`
- `GSmallMapControl`, the mini pan and zoom control, which is appropriate for smaller maps
- `GScaleControl`, the control that shows the metric and imperial scale of the map's current center
- `GSmallZoomControl`, the two-button zoom control used in driving-direction pop-ups
- `GMapTypeControl`, which lets the visitor toggle between Map, Satellite, and Hybrid types

---

■ **Tip** If you're interested in making your own custom controls, you can do so by extending the `GControl` class and implementing its various functions. We may discuss this on the `googlemapsbook.com` blog, so be sure to check it out.

In all cases, it's a matter of instantiating the control object, and then adding it to the map with the `GMap2` object's `addControl()` method. For example, here's how to add the small map control, which you can see as part of the next example in Listing 2-5:

```
map.addControl(new GSmallMapControl());
```

You use an identical process to add all the controls: simply pass in a new instance of the control's class.

■ **Note** What does *instantiating* mean? In object-oriented programming, a class is like a blueprint for a type of entity that can be created in memory. When you put `new` in front of a class name, JavaScript takes the blueprint and actually creates a usable copy (an *instance*) of the object. There's only one `GLatLng` class, but you can instantiate as many `GLatLng` *objects* as you need.

Creating Markers

The Google Maps API makes an important distinction between *creating* a marker, or pin, and *adding the marker to a map*. In fact, the map object has a general `addOverlay()` method, used for both the markers and the white information bubbles.

In order to plot a marker (Figure 2-3), you need the following series of objects:

- A `GLatLng` object stores the latitude and longitude of the location of the marker.
- An optional `GIcon` object stores the image that visually represents the marker on the map.
- A `GMarker` object is the marker itself.
- A `GMap2` object has the marker plotted on it, using the `addOverlay()` method.

Figure 2-3. *Marker plotted in the middle of the Golden Gate Bridge map*

Does it seem like overkill? It's less scary than it sounds. An updated `map_functions.js` is presented in Listing 2-5, with the new lines marked in bold.

Listing 2-5. *Plotting a Marker*

```
var centerLatitude = 37.818361;
var centerLongitude = -122.478032;
var startZoom = 13;

var map;

function init()
{
    if (GBrowserIsCompatible()) {
        map = new GMap2(document.getElementById("map"));
        map.addControl(new GSmallMapControl());
        var location = new GLatLng(centerLatitude, centerLongitude);
        map.setCenter(location, startZoom);
```

```
        var marker = new GMarker(location)
        map.addOverlay(marker);
    }
}

window.onload = init;
window.onunload = GUnload;
```

■ **Caution** If you try to add overlays to a map before setting the center, it will cause the API to give unpredictable results. Be careful to `setCenter()` your `GMap2` object before adding any overlays to it, even if it's just to a hard-coded dummy location that you intend to change again right away.

See what happened? We assigned the new `GLatLng` object to a variable, and then we were able to use it twice: first to center the map, and then a second time to create the marker.

The exciting part isn't creating one marker; it's creating many markers. But before we come to that, we must quickly look at the Google Maps facility for showing information bubbles.

WHITHER THOU, GICON?

You can see that we didn't actually use a `GIcon` object anywhere in Listing 2-5. If we had one defined, it would be possible to make the marker take on a different appearance, like so:

```
var marker = new GMarker(my_GLatLng, my_GIcon);
```

However, when the icon isn't specified, the API assumes the red inverted teardrop as a default. There is a more detailed discussion of how to use the `GIcon` object in Chapter 3.

Opening Info Windows

It's time to make your map respond to the user! For instance, clicking a marker could reveal additional information about its location (Figure 2-4). The API provides an excellent method for achieving this result: the info window. To know when to open the info window, however, you'll need to listen for a click event on the marker you plotted.

Figure 2-4. *An info window open over the Golden Gate Bridge*

Detecting Marker Clicks

JavaScript is primarily an event-driven language. The `init()` function that you've been using since Listing 2-3 is hooked onto the `window.onload` event. Although the browser provides many events such as these, the API gives you a convenient way of hooking up code to various events related to user interaction with the map.

For example, if you had a `GMarker` object on the map called `marker`, you could detect marker clicks like so:

```
function handleMarkerClick() {
    alert("You clicked the marker!");
}

GEvent.addListener(marker, 'click', handleMarkerClick);
```

It's workable, but it will be a major problem once you have a lot of markers. Fortunately, the dynamic nature of JavaScript yields a terrific shortcut here. You can actually just pass the function *itself* directly to `addListener()` as a parameter:

```
GEvent.addListener(marker, 'click',
    function() {
        alert("You clicked the marker!");
    }
);
```

Opening the Info Window

Chapter 3 will discuss the info window in more detail. The method we'll demonstrate here is `openInfoWindowHtml()`. Although you can open info windows over arbitrary locations on the

map, here you'll open them above markers only, so the code can take advantage of a shortcut method built into the `GMarker` object:

```
marker.openInfoWindowHtml(description);
```

Of course, the whole point is to open the info window only when the marker is clicked, so you'll need to combine this code with the `addListener()` function:

```
GEvent.addListener(marker, 'click',
    function() {
        marker.openInfoWindowHtml(description);
    }
);
```

Finally, you'll wrap up all the code for generating a pin, an event, and an info window into a single function, called `addMarker()`, in Listing 2-6.

Listing 2-6. *Creating a Marker with an Info Window*

```
var centerLatitude = 37.818361;
var centerLongitude = -122.478032;
var description = 'Golden Gate Bridge';

var startZoom = 13;
var map;

function addMarker(latitude, longitude, description) {
    var marker = new GMarker(new GLatLng(latitude, longitude));

    GEvent.addListener(marker, 'click',
        function() {
            marker.openInfoWindowHtml(description);
        }
    );

    map.addOverlay(marker);
}

function init() {
    if (GBrowserIsCompatible()) {
        map = new GMap2(document.getElementById("map"));
        map.addControl(new GSmallMapControl());
        map.setCenter(new GLatLng(centerLatitude, centerLongitude), startZoom);

        addMarker(centerLatitude, centerLongitude, description);
    }
}

window.onload = init;
window.onunload = GUnload;
```

This is a nice clean function that does everything you need for plotting a pin with a clickable information bubble. Now you're perfectly set up for plotting a whole bunch of markers on your map.

A List of Points

In Listing 2-3, we introduced the variables `centerLongitude` and `centerLatitude`. Global variables like these are fine for a single centering point, but what you *probably* want to do is store a whole series of values and map a bunch of markers all at once. Specifically, you want a list of latitude and longitude pairs representing the points of the markers you'll plot.

Using Arrays and Objects

To store the list of points, you can combine the power of JavaScript's `array` and `object` constructs. An *array* stores a list of numbered entities. An *object* stores a list of keyed entities, similar to how a dictionary matches words to definitions. Compare these two lines:

```
var myArray = ['John', 'Sue', 'James', 'Edward'];
var myObject = {'John': 19, 'Sue': 21, 'James': 24, 'Edward': 18};
```

To access elements of the array, you must use their numeric indices. So, `myArray[0]` is equal to `'John'`, and `myArray[3]` is equal to `'Edward'`.

The object, however, is slightly more interesting. In the object, the names *themselves* are the indices, and the numbers are the values. To look up how old Sue is, all you do is check the value of `myObject['Sue']`.

■ **Note** For accessing members of an object, JavaScript allows both `myObject['Sue']` and the alternative notation `myObject.Sue`. The second is usually more convenient, but the first is important if the value of the index you want to access is stored in *another* variable, for example, `myObject[someName]`.

For each marker you plot, you want an object that looks like this:

```
var myMarker = {
        'latitude': 37.818361,
        'longitude': -122.478032,
        'name': 'Golden Gate Bridge'
};
```

Having the data organized this way is useful because the related information is grouped as "children" of a common parent object. The variables are no longer just `latitude` and `longitude`—now they are `myMarker.latitude` and `myMarker.longitude`.

Most likely, for your application you'll want more than one marker on the map. To proceed from one to many, it's just a matter of having an *array* of these objects:

```
var myMarkers = [Marker1, Marker2, Marker3, Marker4];
```

Then you can cycle through the array, accessing the members of each object and plotting a marker for each entity.

When the nesting is combined into one step (Figure 2-5), it becomes a surprisingly elegant data structure, as in Listing 2-7.

Listing 2-7. *A JavaScript Data Structure for a List of Locations*

```
var markers = [
    {
        'latitude': 37.818361,
        'longitude': -122.478032,
        'name': 'Golden Gate Bridge'
    },
    {
        'latitude': 40.6897,
        'longitude': -74.0446,
        'name': 'Statue of Liberty'
    },
    {
        'latitude': 38.889166,
        'longitude': -77.035307,
        'name': 'Washington Monument'
    }
];
```

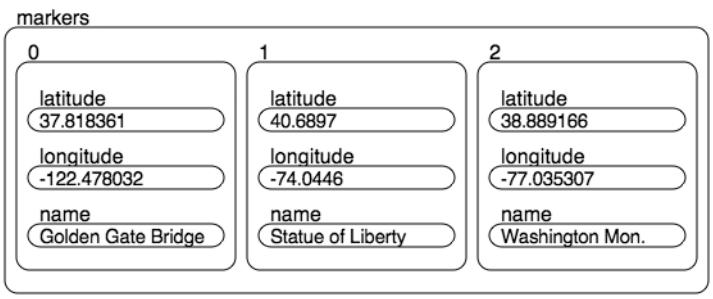

Figure 2-5. *A series of objects stored inside an array*

As you'll see in the next section, JavaScript provides some terrific methods for working with data in this type of format.

■ **Note** In this book, you'll see primarily MySQL used for storing data permanently. Some people however, have proposed the exact format in Figure 2-5 as an alternative to XML, calling it JSON, for JavaScript Object Notation. While there are some advantages, JSON's plethora of punctuation can be intimidating to a less technical person. You can find more information on JSON at `http://json.org`. We'll still be using a lot of JSON-like structures for communicating data from the server to the browser.

Iterating

JavaScript, like many languages, provides a `for` loop—a way of repeating a block of code *for* so many iterations, using a counter. One way of cycling through your list of points would be a loop such as this:

```
for (id = 0; id < markers.length; id++) {
        // create a marker at markers[id].latitude, markers[id].longitude
}
```

However, JavaScript also provides a much classier way of setting this up. It's called a `for in` loop. Watch for the difference:

```
for (id in markers) {
        // create a marker at markers[id].latitude, markers[id].longitude
}
```

Wow. It automatically gives you back every index that exists in an array or object, without needing to increment anything manually, or ever test boundaries. Clearly, you'll want to use a `for in` loop to cycle over the array of points.

Until now, the `map_data.php` file has been empty and you've been dealing mainly with the `map_functions.js` file. To show a list of markers, you need to include the list, so this is where `map_data.php` comes in. For this chapter, you're not going to actually use any PHP, but the intention is that you can populate that file from database queries or some other data store. We've named the file with the PHP extension so you can reuse the same base code in later chapters without the need to edit everything and start over. For now, pretend the PHP file is like any other normal JavaScript file and create your list of markers there. As an example, populate your `map_data.php` file with the structure from Listing 2-7.

To get that structure plotted, it's just a matter of wrapping the marker-creation code in a `for in` loop, as shown in Listing 2-8.

Listing 2-8. *map_functions.js Modified to Use the Markers from map_data.php*

```
var map;
var centerLatitude = -95.0446;
var centerLongitude = 40.6897;
var startZoom = 3;

function addMarker(longitude, latitude, description) {
    var marker = new GMarker(new GLatLng(latitude, longitude));

    GEvent.addListener(marker, 'click',
        function() {
           marker.openInfoWindowHtml(description);
        }
    );

    map.addOverlay(marker);
}
```

```
function init() {
    if (GBrowserIsCompatible()) {
        map = new GMap2(document.getElementById("map"));
        map.addControl(new GSmallMapControl());
        map.setCenter(new GLatLng(centerLatitude, centerLongitude), startZoom);

        for(id in markers) {
           addMarker(markers[id].latitude, markers[id].longitude, markers[id].name);
        }
    }
}

window.onload = init;
window.onunload = GUnload;
```

Nothing here should be much of a surprise. You can see that the `addMarker()` function is called for each of the markers, so you have three markers and three different info windows.

Summary

With this chapter complete, you've made an incredible amount of progress! You've looked at several good programming practices, seen how to plot multiple markers, and popped up the info window. And all of this is in a tidy, reusable package.

So what will you do with it? Plot your favorite restaurants? Mark where you parked the car? Show the locations of your business? Maybe mark your band's upcoming gigs?

The possibilities are endless, but it's really just the beginning. In the next chapter, you'll be expanding on what you learned here by creating your map data dynamically and learning the key to building a real community: accepting user-submitted information. After that, the weird and wonderful science of geocoding—turning street addresses into latitudes and longitudes—will follow, along with a variety of tips and tricks you can use to add flavor to your web applications.