Public beta 1.0

This is a public beta of a chapter from my upcoming book about Google Maps API v3.

The main reason that I am making it public is that I think the book will benefit from getting your feedback. So please! Let me know what you think about it. Is it unclear? Have I missed something or is it spot on? What you think is important and will add value to the book.

Send your feedback to gabriel@svennerberg.com or post it as a comment on http://www.svennerberg.com/?p=2333. On that web page you can also sign up to get updates about the progress of the book.

I hope you will enjoy the read!

Gabriel Svennerberg

Chapter 7 - X marks the spot

The most common use of maps on the Internet is to visualize the geographic position of something. The Google Maps Marker is the perfect tool for doing this.

A marker is basically a small image that is positioned on a map somewhere. Its most frequent incarnation is the familiar red drop-shaped marker that is the default marker in Google Maps.

A simple marker

If you want to put a marker on your map with the default look it's easily achieved with only a few lines of code.

The marker object is conveniently available in the `google.maps.Marker` namespace. It only takes one parameter: an object literal. The object literal can contain properties that we will see later. The parameter is called *options* and with it we can define many attributes, like how the marker should look and behave. For now we'll settle on the only 2 required properties: `position` and `map`.

`Map` is a reference to the map where we want to put our marker

`Position` defines the coordinates where the marker will be placed. It takes an argument in the form of a `google.maps.LatLng` object (check chapter 1 if you don't remember it).

```
var marker = new google.maps.Marker({
  position: new google.maps.LatLng(57.8,14.0),
  map: map
});
```

This little snippet of code will put a marker on the map. It has the look of the default red Google Maps marker and you can't do anything with it, but it dutifully marks a spot on the map as seen in figure 4.

Figure 1 - A simple marker

Adding a tooltip

The first thing we might want to do is to add a tooltip to the marker. A tooltip is a yellow box with some text in it that appears when you hover the cursor over an object. To add a tooltip to a marker you use the property `title`. It's as simple as adding the title property to the object literal with the marker options.

```
var marker = new google.maps.Marker({
  position: new google.maps.LatLng(57.8,14.0),
  map: map,
  title: 'Click me'
});
```

Figure 2 - Marker with a tooltip

Changing the icon

If you're not satisfied with the default icon you can easily switch it to a custom one. The easiest way to do this is to change the `icon` property of marker to an URL of a suitable image.

Google hosts a number of images that you are free to use. If you go to http://gmaps-samples.googlecode.com/svn/trunk/markers/blue/blank.png with your web browser you will find this image. And that's the one we're going to use for this example.

```
var marker = new google.maps.Marker({
  position: new google.maps.LatLng(40.761137,-73.97674),
  map: map,
  title: 'Click me',
  icon: 'http://gmaps-samples.googlecode.com/svn/trunk/
    markers/blue/blank.png'
});
```

Doing this will change the look of the marker, as shown in figure 3.

Figure 3 - A simple custom icon

In this example I've been using an icon that resides at Google's servers and that's OK since Google is also the one providing the API. Generally though you should not link images that reside at servers belonging to others, but rather keep them to your own server and feed them from there. The reason is that it's just plain wrong to steal other people's bandwidth without permission.

Icons supplied by Google

Google has a collection of standard icons that you can use on your map. Most of them use a similar URL that looks like this:

```
http://gmaps-samples.googlecode.com/svn/trunk/markers/color/markerx.png
```

Where `color` is one of the following values:

- blue
- green
- orange
- pink
- red

And `x` is a number between 1 and 99.

If you want a marker with no number, use the filename `blank.png`.

This is an easy way to construct a custom icon but if you look at it closely you will notice that it doesn't have a shadow. Further on in the book I will describe how to make a more complex icon with a shadow, custom shape and a defined clickable area. We will

also take a look at some really clever ways to deal with scenarios where you need lots of different marker icons.

Adding an InfoWindow

Often when marking places on a map you will want to show additional information related to that place. The Google Maps API offers a perfect tool for this and that's the `InfoWindow`. It looks like a speech bubble and typically appears over a marker when you click on it.

Figure 4 - An InfoWindow

A simple Info Window

Just like the marker object, the `InfoWindow` object resides in the `google.maps.InfoWindow` namespace and only takes one argument, and yes, you probably guessed it, an object literal containing *options*.

Like the marker option the `InfoWindow` option has several properties but the most important one is `content`. This property controls what will show inside of the `InfoWindow`. It can be plain text, HTML or a reference to a HTML-node. For now we will stick with plain text but do note that we can use full HTML if we like. That also means that we can include, images and video, and style it any way we want.

```
var infowindow=new google.maps.InfoWindow({
  content:'Hello world'
});
```

Tip – Controlling the size of the InfoWindow

To control the size of the `InfoWindow` you can add an HTML element with a defined width to it. This way you can control its size in your CSS.

Now we've created an `InfoWindow` object that will contain the text "Hello world" but it currently doesn't exist on the map. What we want to do is to connect it with the marker so that when we click on the marker the `InfoWindow` appears. To do this we need to attach a click-event to the Marker.

A word or two about Events

Every time you interact with something in a map or on a web page an *event* is triggered. Like for example when you click on a link, a *click-event* is triggered or when you press a button on the keyboard, a *keypress-event* is triggered.

These are all active events that are triggered by the user. But there are also other types of events: passive events, events that happen in the background. We've already looked at the *load-event* of the window object. It's triggered when the web page in a browser window has finished loading. Another example is the *focus-event* which triggers when an object gets focus. In the Google Maps API there are lots of these passive events, for example *tilesloaded*, *bounds_changed* and *center_changed* that are all events of the Maps object.

Listen for the events

What these events have in common is that we can catch them in our code and do stuff when they are triggered. To do this we need to add *listeners*. A listener is connected to an object and a certain event. It just sits quietly and waits for the event to happen. When it does it pops into action and runs some code. In the Google Maps API the `addListener()` method resides in the google.maps.events namespace and takes 3 arguments:

- The object it's attached to
- The event it should listen for
- A function that is executed when the event is triggered

Definition:
```
addListener(instance:Object, eventName:string, handler:Function)
```

Adding a click event to the marker

To add a click event to our marker we need to write this:

```
google.maps.event.addListener(marker,'click',function(){
  infowindow.open(map,marker);
});
```

This code will attach a click-event to the marker that will trigger the `open()` method of the `InfoWindow` object and pass the map- and the marker-object to it. The map object is needed for the `InfoWindow` so that it knows which map to attach itself to (in case you have more than one). The marker object is needed for the `InfoWindow` to know where to position itself. Typically the tip of the stem of the speech bubble will point at the marker.

Now we have all of the components in place and if we try this code out, the map will initially display our marker. When we click the marker the `InfoWindow` will display.

Figure 5 - An Info Window associated with the Marker

More markers

Now you know how to put a single marker on the map. You also have some rudimentary knowledge of how to tweak the marker a little bit and how to attach an InfoWindow to it. But what if you want to put more markers on the map? You could of course add them one by one, but eventually it's that's going to add up to a whole lot of code. A smarter thing to do is to use *arrays* and *loops*.

JavaScript Arrays

A JavaScript array is basically a list of stuff. It can contain whatever you want to put in it and each item will have its own index number.

There are two ways of creating an array in JavaScript. The first one is to call the constructor of the Array object:

```
var myArray = new Array();
```

The other way is to create an Array Literal.

```
var myArray = [];
```

These two do exactly the same thing but the last one is the preferred method these days and it's also the method that I will stick to throughout this book.

With the array literal method we can easily instantly fill the array with different things, like for example a list of fruit.

```
var myArray = ['apple', 'orange', 'banana'];
```

Each of the items in the array list gets an individual index number. The first item gets number 0, the second item gets number 1 and so on. So to retrieve an item from an array we simply use its index number.

```
myArray[0] // returns apple
myArray[1] // returns orange
```

Another way of adding items to an array is with the arrays native method `push`. What `push` does is to take the passed value and add it to the end of the array. So to create the same array as used above with this technique would look like this:

```
// First we create the array object
var myArray = [];

//Then we start adding items to it
myArray.push('apple');
myArray.push('orange');
myArray.push('banana');
```

This will produce exactly the same array as previously. This method is handy when you don't have all the values upfront and you need to add values as you go along.

Arrays also have a native `length` property that returns the number of items that it contains. In our case `length` will return 3 since myArray contains 3 items.

```
myArray.length        // returns 3
```

Knowing this we can loop through the array to extract its items.

Introducing loops

There are two kinds of loops in JavaScript. There are ones that execute a specified number of times called *for-loops*, and there are ones that executes while a specific condition is true called *while-loops*.

Loops are good for executing the same code several times. This is very handy when you for example, want to put lots of markers on a map. We then want to run the same code over and over but insert different data each time. For this task a for-loop is perfect.

The basic construction of a for-loop looks like this:

```
for (var i = 0; i< 3; i += 1) {
  document.writeln(i + ',');
}
```

This loop will produce the following result:

0, 1, 2,

Here's an explanation of what this code really does:

1. The first statement (var i = 0) defines the variable i and assign it the value 0. This will only be done before the first iteration.
2. With each iteration, the loop will check the second statement (i < 3) and see if it's true. If it's true it will run one more time and then check it again. This will go on until it eventually returns false, then the loop will stop.
3. The third statement will add 1 to i each iteration. So eventually i will be 3 and when it does the second statement will no longer be true and the loop will stop.

Adding US cities to the map

In this example we're going to put a few us cities on the map. First we're going to store the coordinates for the cities in an array. Then we're going to loop through that array to put each one of those on a map.

First we have to create an array containing the coordinates. Since we can store any object we like in an array I'm going to store a `google.maps.LatLng` object containing the coordinates for each city in each spot in the array. This we way we don't have to worry about that when we create the markers.

```
// Creating an array which will contain the coordinates
// for New York, San Fransisco and Seattle
var places=[];

// Adding a LatLng object for each city
places.push(new google.maps.LatLng(40.756054,-73.986951));
places.push(new google.maps.LatLng(37.775206,-122.419209));
places.push(new google.maps.LatLng(47.620973,-122.347276));
```

We now have an array containing all the data we need to put markers on the map. The next step is to loop through the array to extract this data.

```
// Looping through the places array

for (var i = 0;i < places.length; i += 1) {
  // Creating a new marker
  var marker = new google.maps.Marker({
    position: places[i],
    map: map,
    title: 'Place no ' + i
  });
}
```

This code loops through our array and each iteration it creates a new marker. Notice when we set the value for the property `position` we call the array by its index number `places[i]`. Also notice that we set a tooltip for each marker with the property `title`. It will get the text *"Place no"* followed by the current number of the iteration. The marker in the first iteration will get the tooltip *"Place no 0"*, the marker in the second iteration will get the tooltip *"Place no 1"*, and so on.

Figure 6 - Displaying more markers at the same time

Adding InfoWindows

Now we want to add InfoWindows to the marker so that when we click on them, an InfoWindow pops up. This is done by adding the following code inside of our loop, just beneath the code that creates the marker.

```
google.maps.event.addListener(marker, 'click', function() {
  var infowindow = new google.maps.InfoWindow({
    content:'Place no ' + i;
  });
  infowindow.open(map, marker);
});
```

What happens here is that a *click-event* is attached to the marker so that when you click it, a new InfoWindow with the content "Place no " and the number of the current iteration is created. The last line of the code opens up the InfoWindow.

Problem

When we run this code we will immediately spot a problem. No matter which marker we click on, the InfoWindow will open up for the marker that was created last. This happens due to a phenomena called closure and is caused by the fact that the JavaScript language has function scope.

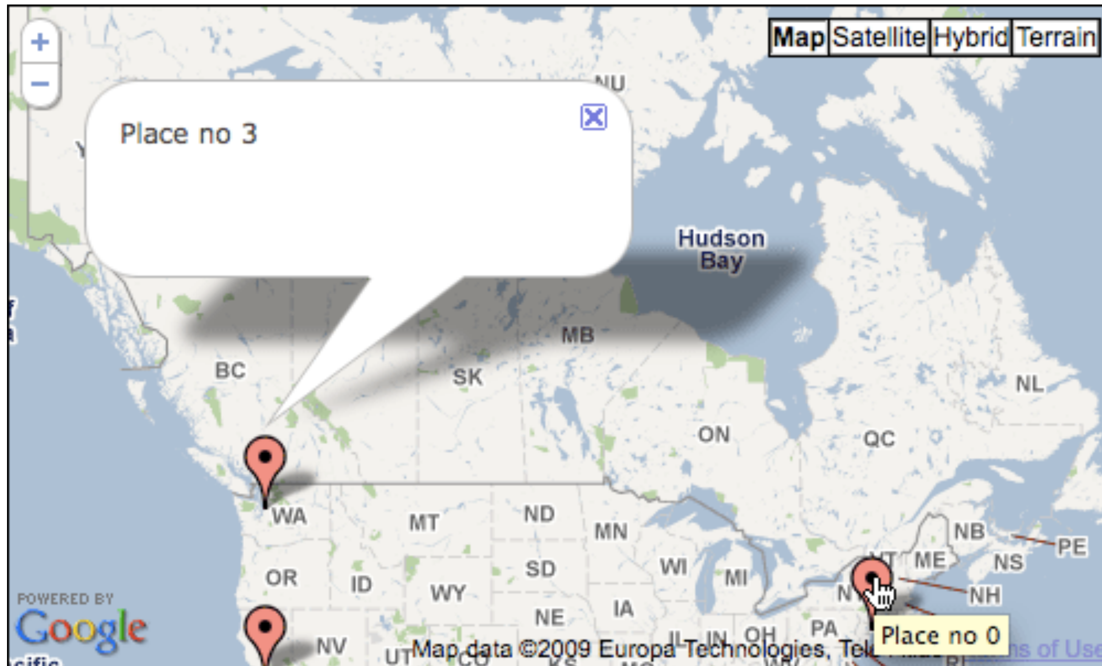

Figure 7 - Even though place no 0 is clicked place no 2 is opened with the text "Place no 3".

What that means is essentially that variables are passed to the code that creates the InfoWindow rather than the values of the variables. Since the variable marker, after the code has run, contains the last marker created it will only apply to that. Also notice that the text in the InfoWindow shows Place no 3 even though that Place no 2. That's because 3 is the last value assigned to i before the loop exits.

To fix this we need to pass the values of the variables instead of the variables themselves. This can be done by putting the code that creates the marker and the InfoWindow inside a function.

Function scope and closure

Most programming languages with C syntax have block scope, that is that stuff inside of a block with curly brackets {} have their own scope. Variables defined inside the block are not available for code outside the block. However, this is **not** the case with JavaScript. This can be quite confusing since JavaScript's syntax suggest that it does. Instead JavaScript got something called *Function scope*. That is; variables defined inside of a function, are not available to code outside that function.

In JavaScript it's possible to nest functions. Actually that's one of the most important features that makes the language so expressive. Anyway, a variable defined inside a function is available to all other functions nested inside of that function.

```
function foo() {
  var a = 1, b = 2;

  function bar() {
    alert(a); // Will display 1
     var b = 10;
  }
  alert(b); // Will display 2
}
```

Breaking out the code into a function

Ok, so now that we understand how scope works in JavaScript, we can put that knowledge to good use. To solve the problem with the variable being used rather than its value, we can take the part of the code that causes the problem and put it inside a function.

```
function addMarker(latlng,no) {
  var marker = new google.maps.Marker({
    position: latlng,
    map: map,
    title: 'Place no ' + no
  });

  google.maps.event.addListener(marker, 'click', function() {
     var infowindow = new google.maps.InfoWindow({
       content:'Place no '+no
    });
    infowindow.open(map, marker);
  });
}
```

As you probably can see, the code is almost identical. The only difference is that its been put inside of the function `addMarker()` and the `i` variable is changed to `no`.

Now in our loop we call our new function. Since arguments passed to a function is always *by value* in Javascript we no longer suffer from our previous problem.

```
for (var i = 0; i < places.length; i += 1) {
  addMarker(places[i], i);
}
```

This takes care of the closure problem! So now when we click on the markers, the correct InfoWindow is displayed.

Dealing with several windows

In Google Maps API 2 only one InfoWindow could be displayed at a time. The default behavior was that every time you opened an InfoWindow, all other InfoWindows would

close. This is not the case in version 3 of the API where you instead can open an infinite number of InfoWindows. In some situations that's great but most of the time you will probably only want to have one InfoWindow open at a time. An easy way to fix that is to simply have one InfoWindow that you reuse over and over again.

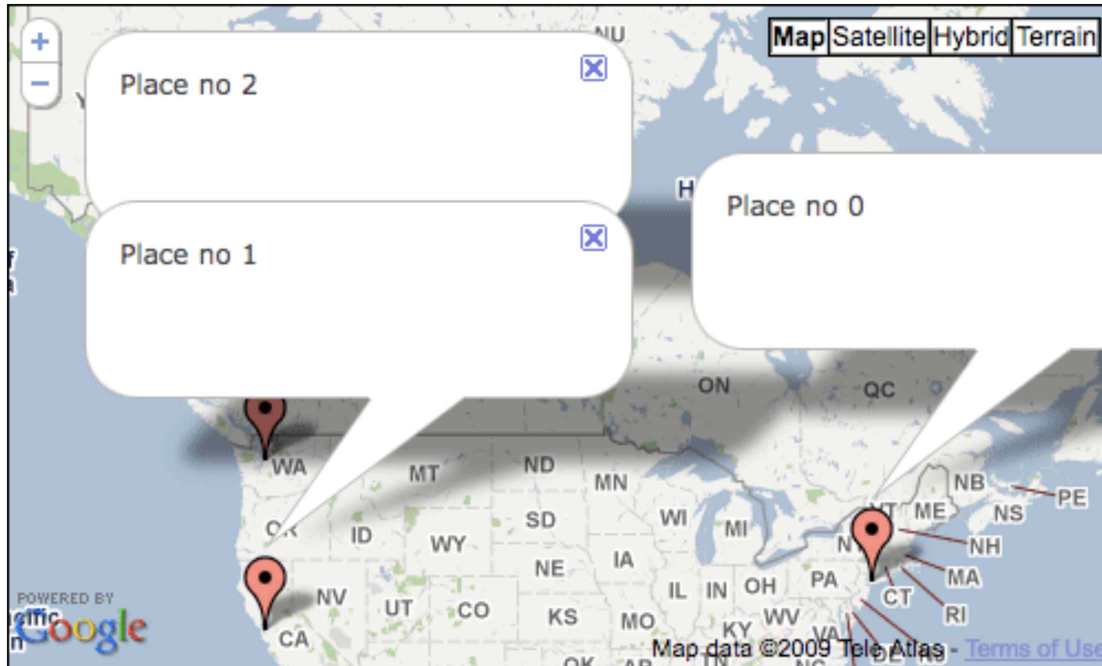

Figure 8 - Several InfoWindows

To do this we first need to declare a global variable that will hold the InfoWindow object. This will be our reusable object. Be sure to create this outside of the function so that it's readily available. (If we would declare it inside the function it would be recreated each time the function was called, due to the scope chain)

Next we need to add a check to see if our variable already contains an InfoWindow object. If it does we just use it, if it doesn't we need to create it.

Ok, so lets do it:

```
// Declare infowindow as a global variable
var infowindow;
```

Next rewrite the code inside the function that adds the InfoWindow to the marker so that it looks like this.

```
// Add click event to the marker
google.maps.event.addListener(marker, 'click', function() {
  // Check to see if the infowindow already exists and is not null
  if (!infowindow) {
    // if the infowindow doesn't exist,create an
    // empty InfoWindow object
    infowindow = new google.maps.InfoWindow();
```

```
  }
  // Set the content of the InfoWindow
  infowindow.set_content('Place no ' + no);
  // Tie the InfoWindow to the marker
  infowindow.open(map,marker);
});
```

What happens here is that instead of creating a new InfoWindow every time the user clicks a marker, we just move around the existing one and changes the content of it. It's easy to check if the variable `infowindow` is carrying an object with an if-statement. An empty variable will return undefined which is false. If it on the other hand carries an object it will return the object which is truth.

Automatically adjust the viewport to fit all markers

Sometimes you know beforehand which markers are going to be added to the map and can easily adjust the position and zoom level of the map to fit all the markers inside the viewport. But more than often you're dealing with dynamic data and don't know exactly where the markers are going to be positioned. You could of course have the map zoomed out so far out that you're certain that all the marker will fit, but a better solution is to have the map automatically adjust to the markers added. There's when the `LatLngBounds` object will come in handy.

Introducing the LatLngBounds object

A bounding box is a rectangle defining an area. Its corners consist of geographical coordinates and everything that's inside it is within its *bounds*. It can be used to calculate the viewport of the map but it's also useful for calculating if an object is in an certain area.

The bonding box in Google Maps is represented by the `google.maps.LatLngBounds` object. It takes two optional arguments, which are the south-west and the north-east corners of the rectangle. Those arguments are of the type `LatLng`.

To manually create a `LagLngBounds` object to fit our markers we have to first determine the coordinates for its corners and then create it.

```
var bounds = new google.maps.LatLngBounds(
  new google.maps.LatLng(37.775206,-122.419209),
  new google.maps.LatLng(47.620973,-73.986951)
);
```

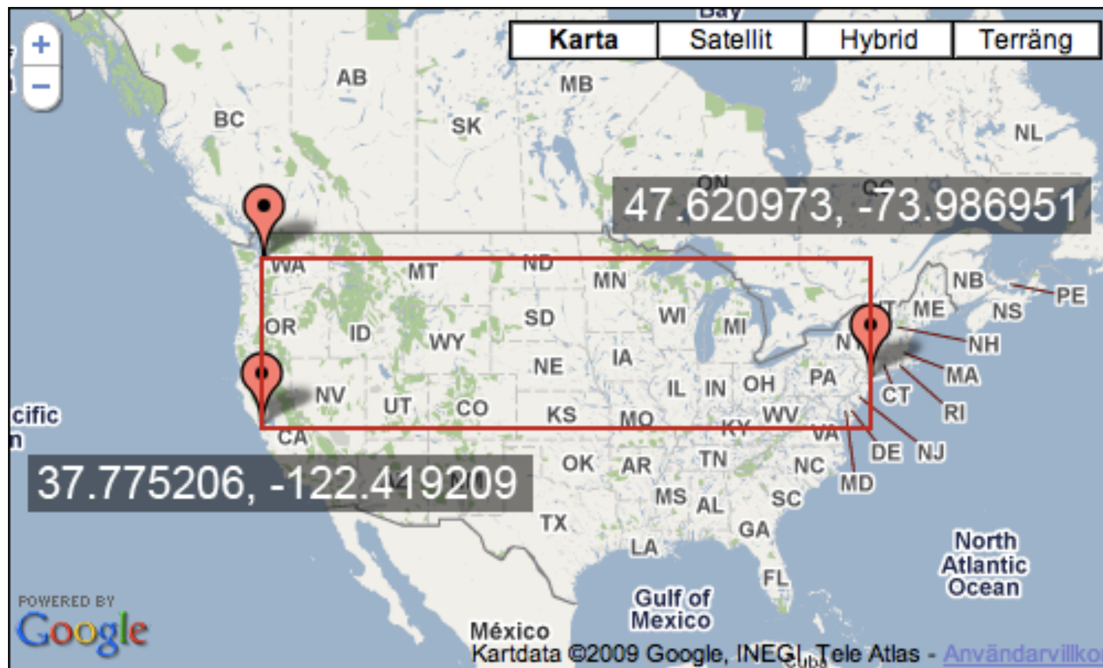

Figure 9 - A bounding box is made up of the south-west and north-east corners of the rectangle fitting all of the markers inside it.

Let the API do the heavy lifting

To extend our example to **automatically** adjust the viewport to fit the markers inside it, we need to add a `LatLngBounds` object to it.

First of all we create an empty `LatLangBounds` object:

```
// Creating a LatLngBounds object
var bounds = new google.maps.LatLngBounds();
```

Then we need to extend the bounds with each marker added to the map. This will automatically give us a bounding box of the correct dimensions.

```
// Looping through the places array
for (var i = 0; i < places.length; i += 1) {
  // Extending the bounds object with the LatLng of each marker
  bounds.extend(places[i]);
}
```

Lastly when we've iterated through all the markers, we're going to adjust the map using the `fitBounds()` method of the map object. It takes a `LatLngBounds` object as its argument and then use it to determine the correct position and zoom level of the map.

```
map.fitBounds(bounds);
```

Tada! We now have a map that perfectly fits all the markers inside the viewport. If we were to add additional cities to the map, they would automatically be taken in account when calculating the viewport.

> **Note:** Google Maps automatically adds some extra padding around the bounding box so that none of the markers appears exactly at its edge.

Now if we add Rio de Janeiro in Brazil to our array of cities and run the map, we will see that it automatically adjusts to the new bounding box.

Figure 10 - Now that Rio de Janeiro is added to the map, we can see that it adjusts the viewport to fit it as well

The complete code

We've done quite a lot in this chapter and kept adding more and more functionality to our map. Here's the complete code.

```
(function() {

  // Decleration av some variables that will be available
  // for all of the code
  var map, infowindow;

  window.onload = function() {

    // Reference to the HTML-element that will contain the map
    var mapDiv = document.getElementById('map');
```

```
    // Create an object literal containing the properties
    // we want to pass to the map
     var options = {
       zoom: 9,
       center: new google.maps.LatLng(37.09,-95.71),
       mapTypeId: google.maps.MapTypeId.ROADMAP
    };

    // Call the constructor, thereby initializing the map
    map = new google.maps.Map(mapDiv,options);

    // Create an array which will contain the coordinates
    // for New York, San Fransisco and Seattle
     var places = [];

    // Adding a LatLng object for each city
    places.push(new google.maps.LatLng(40.756054,-73.986951));
    places.push(new google.maps.LatLng(37.775206,-122.419209));
    places.push(new google.maps.LatLng(47.620973,-122.347276));
    places.push(new google.maps.LatLng(-22.933377,-43.184365));

    // Creating a LatLngBounds object
     var bounds = new google.maps.LatLngBounds();

    // Loop through the places array
    for (var i = 0; i < places.length; i += 1) {
      // Add marker to the map
      addMarker(places[i], i);

      // Extending the bounds object with the LatLng of each marker
      bounds.extend(places[i]);
    }

    // Adjusting the map to new bounding box
    map.fitBounds(bounds)

  }

  // This function adds a marker to the map
  // with an attached InfoWindow
  function addMarker(latlng,no) {
     var marker = new google.maps.Marker({
       position: latlng,
       map: map,
       title: 'Place no ' + no
    });
```

```
    // Add click event to the marker
    google.maps.event.addListener(marker, 'click', function() {
      // Check to see if the infowindow already exists
       if (!infowindow) {
        // if the infowindow doesn't exist, we create an empty
        // InfoWindow object
        infowindow = new google.maps.InfoWindow();
      }
      // Set the content of the InfoWindow
      infowindow.set_content('Place no ' + no);
      // Tie the InfoWindow to the marker
      infowindow.open(map,marker);
    });

  }

})();
```

Summary

In this chapter we've examined markers and what we can do with them. We've also looked at some basic usage of InfoWindows. Some of the things you've learned are:

- How to put a marker on the map
- Changing the marker icon
- Associating an InfoWindow with a marker
- How to attach events to objects
- Putting several markers on the map
- How to automatically adjust the viewport to fit the markers

With this knowledge you will be able to cope with most of the challenges of designing maps with a reasonable amount of markers. However, when the amount of markers start to add up to hundreds or maybe even thousands you will run into problems. But don't worry, in chapter x we will look at different strategies for dealing with massive amounts of markers.